

## 2장. 객체 지향 개념의 원리

### 개 요

- “객체 지향”이라는 용어가 사용되는 모든 곳에서 통용되는 보편적 개념을 설명
- 객체 지향에 관한 학문적인 정확한 정의는 없음
  - 예) 인간이란 무엇인가?
  - 객체 지향이 갖는 특성들을 통해 이해 가능
    - 주요 특징 : 자료 추상화, 상속, 다형 개념, 동적 바인딩
    - 기타 특징 : 다중 상속, 추상 클래스, 예외 처리, 인터페이스, 직렬화 등

## 2.1.1 “객체 지향” 용어에 대하여

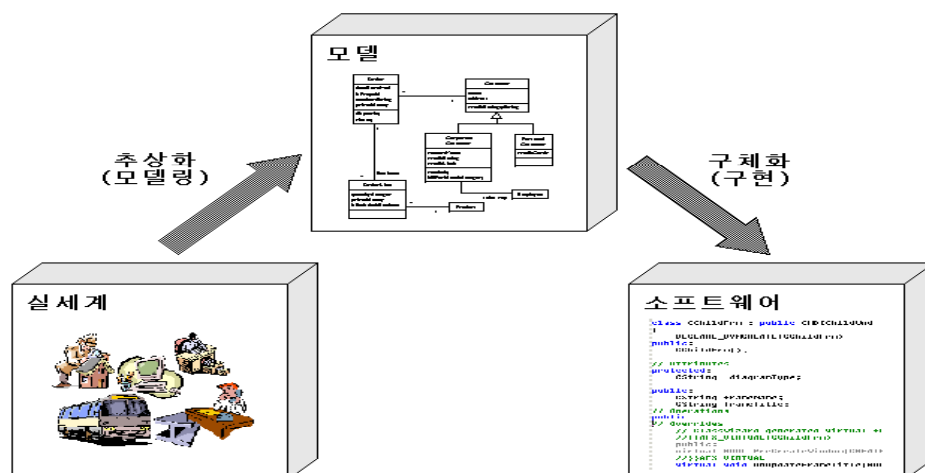
- ❑ 실세계를 해석하여 소프트웨어를 개발하고자 할 때 관점의 주된 대상을 객체 위주로 이해함
- ❑ 객체의 구성: 자료구조 + 연산
- ❑ 객체와 클래스의 구분(문맥에 따른 해석필요)

cf. 구조적 기법

- 프로세스 지향적(process oriented)
- 복잡성 야기, 신뢰성 있는 시스템 구현 어려움
- 재 사용될 수 있는 모듈로 나누기 어려움

## 2.1.2 “추상화”의 의미

- ❑ 소프트웨어 개발 작업은 “simulation”을 의미
  - 추상화 : 실 세계의 상황을 간결하고 명확하게 모델링 (시스템 분석 과정에 해당)
  - 구체화 : 추상적 모델을 프로그램으로 변환



## 추상화의 목적

1. 어떻게 하면 실제 시스템을 **간결하게** 표현할 수 있는가 ?
  2. 어떻게 하면 실제 시스템의 정보를 **모두 표현**하는가 ?
- 위 두 목적은 서로 상충됨
- 간결성과 상세화 중에서 어느 쪽에 비중을 둘 것인가 ?

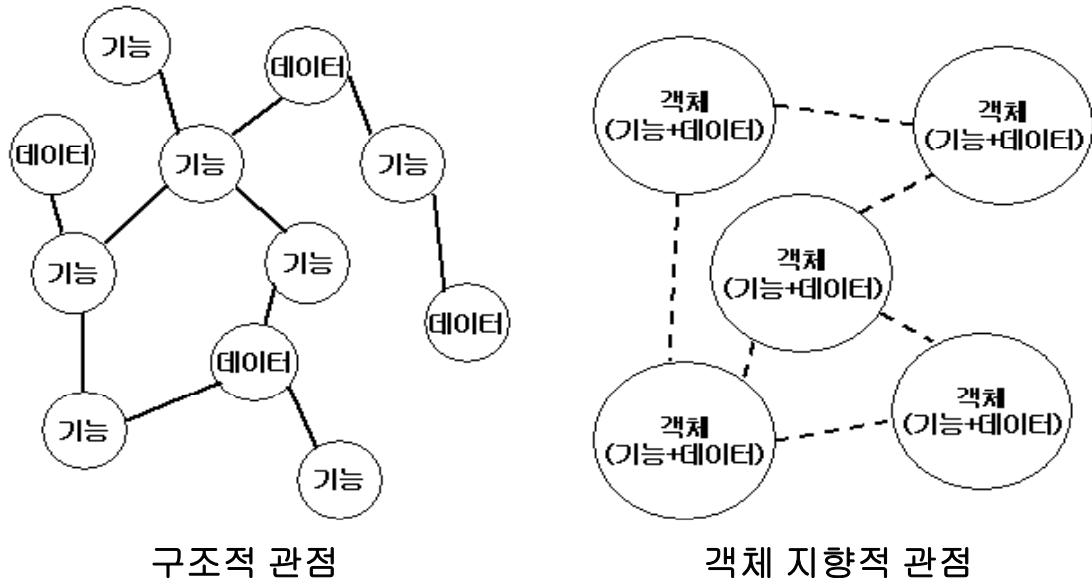


## 추상화를 이루기 위한 전제 조건

1. 모델이 통용될 수 있는 “사회”를 기반으로 한다.
  - 앞 그림의 예제는 저자의 가정에서만 통용됨
2. 모델을 표현할 수 있는 표기법이 미리 정의되어 있어야 한다.
  - 음악가들은 **악보**를 통하여
  - 건설업 종사자들은 **건축 설계도**를 통하여
  - 수학자들은 **미분 방정식**을 통하여
  - UML

## 2.1.3 시스템을 해석하는 관점

### □ 구조적 관점과 객체 지향 관점의 차이



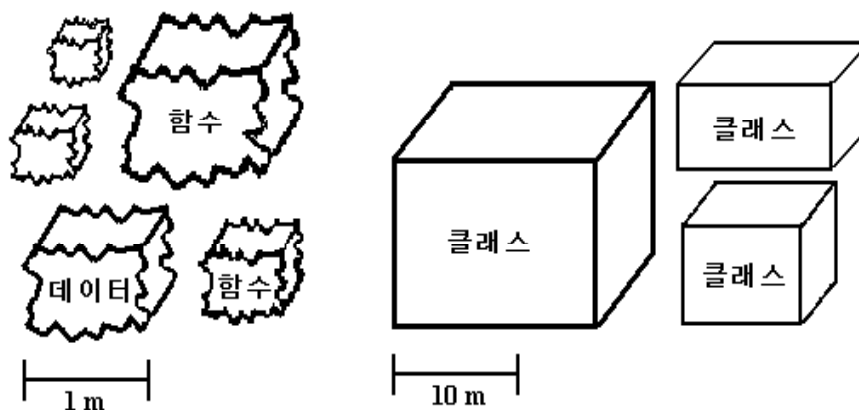
### □ 시스템이란

- 공동의 목표를 추구하는 사물(데이터, 함수, 객체 등)들의 연결된 그래프(connected graph)이다.
- 구조적 관점
  - 시스템이 데이터와 함수(기능)로 구성됨
    - 강한 연결 관계(실선으로 표기)
  - 데이터와 함수의 상호작용에 중요성 부여
  - 시스템이 커질수록 알고리즘의 복잡도를 제어하기 힘들어짐
- 객체 지향적 관점
  - 시스템을 구성하는 모든 것이 객체
  - 객체들 간에 느슨한 연결(점선으로 표기) 존재
    - 시스템의 이해가 단순
    - 시스템 변경이 용이
    - 자료 추상화, 동적 바인딩 특성과 관련됨

## □ 무엇이 시스템의 복잡도를 결정하는가?

- 시스템을 구성하는 **구성원의 수**
- 구성원을 연결하는 그래프의 **링크 수**
  - 구성원의 수가  $n$ 일 때, 링크의 수는  $O(n^2)$ 에 비례
- 구조적 관점에서 시스템을 구성하는 구성원(데이터, 함수)의 크기가 객체 지향적 관점의 구성원(객체)보다 상대적으로 작다.
  - 구조적 관점의 시스템이 더 복잡하게 된다는 의미

- 시스템을 구성하는 각 **노드의 인터페이스** 또한 시스템의 복잡도를 결정하는 중요한 요소이다.
- 예) 1m 벽돌로 건물 짓는 것과 10m 벽돌로 건물 짓는 것을 비교
  - 구조적 기법 : 벽돌 크기 작고, 인터페이스 복잡
  - 객체 지향 기법 : 벽돌 크기 크고, 인터페이스 단순



## 2.1.4 “객체”와 “클래스”

### □ “객체”와 “클래스”의 구별

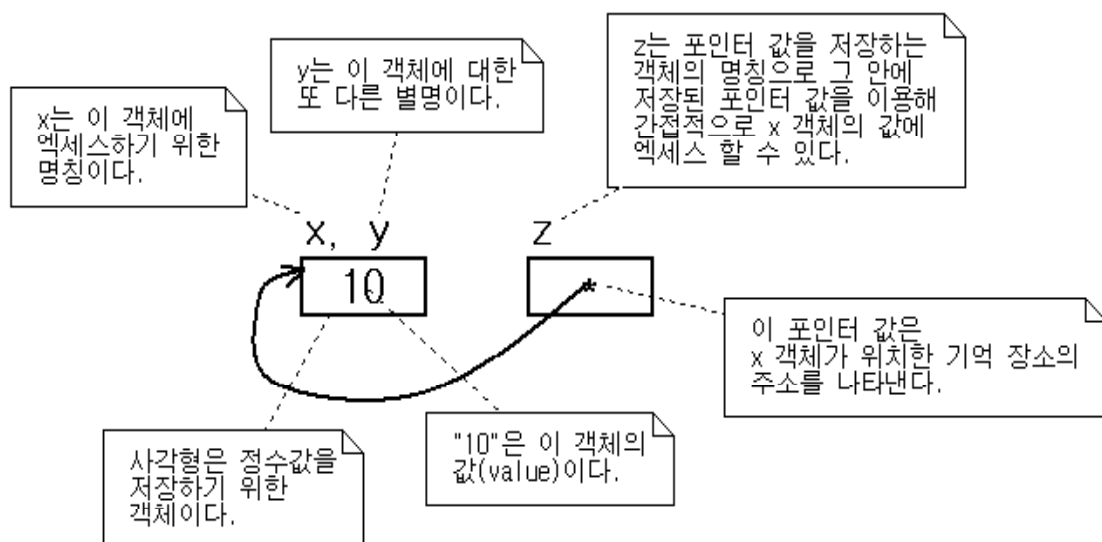
- 클래스는 객체의 집합이다.
- 때때로 객체와 클래스라는 용어의 구별이 의미가 없는 경우도 있다. (객체가 좀 더 포괄적인 의미를 갖기 때문)

### □ “객체”와 “객체 식별자”의 구별

- “객체 그 자체”와 “객체를 가리키는 명칭”을 확실히 구분할 수 있어야 함
- 객체는 “식별자”를 통해 접근 가능함
  - 특정한 객체를 access하기 위한 수단
  - Identifier, pointer, reference

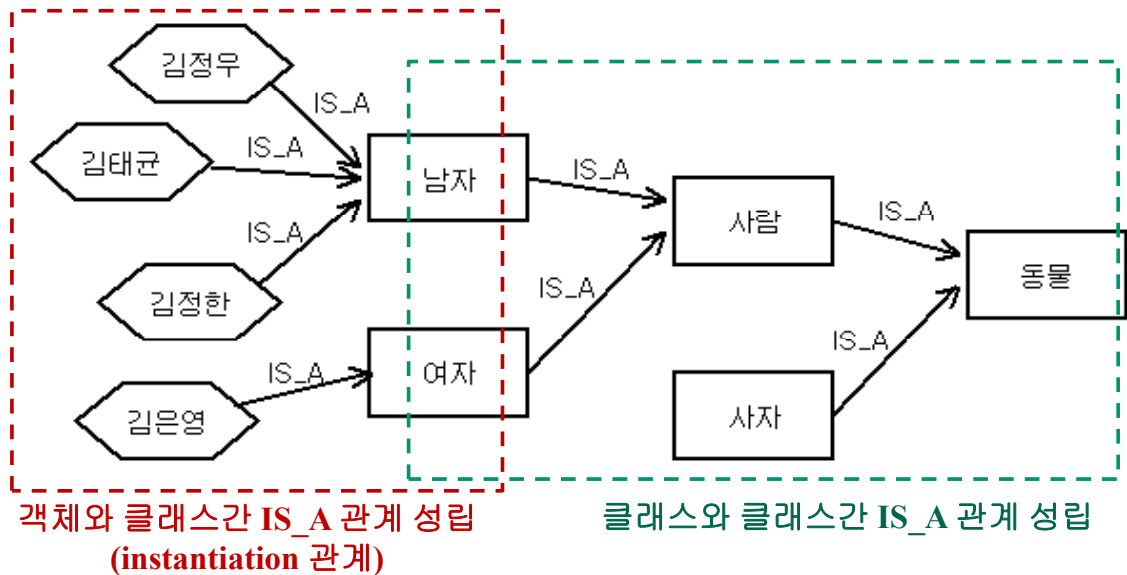
### □ 객체에 대한 접근 수단의 예

```
int x;  
int &y = x;  
int *z = &x;
```



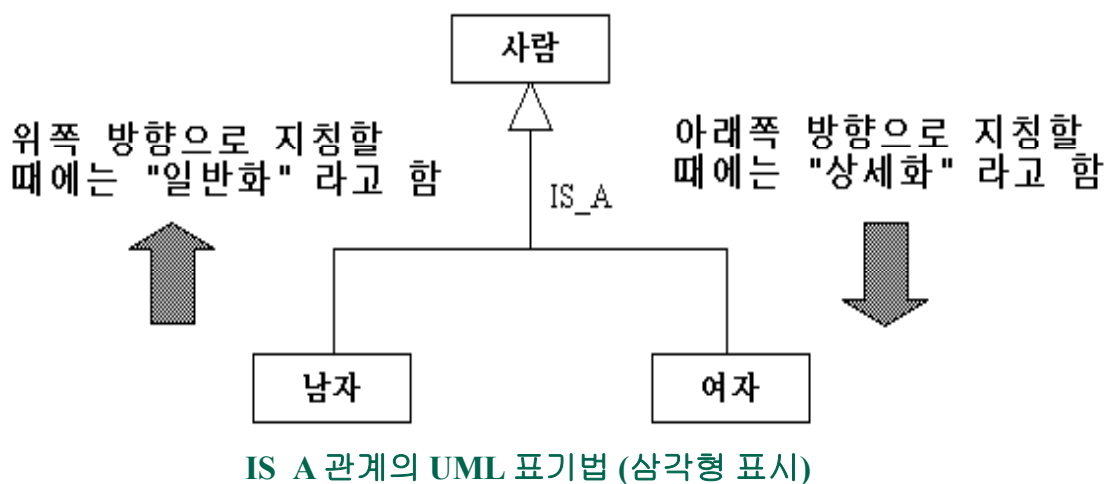
## ❑ 객체와 클래스의 관계

- Instantiation 관계
- IS\_A 관계 (cf. 상속 관계)
  - 예) 홍길동은 남자이다 = Hong is a man
  - IS\_A 관계는 **방향성**이 있음 (화살표로 표시)
- 클래스와 객체를 혼동하는 이유 : 아래 그림



## ❑ 클래스와 클래스의 관계

- Association, Relationship 관계
- 실제 모델링 시에는 클래스만을 이용
- 상속 관계(Inheritance)를 방향성에 따라 일반화 (Generalization), 상세화 (Specialization)로 구분
- OR 관계 (= IS\_A 관계)
  - 예) 사람의 종류에는 남자 OR 여자가 있다.



## 2.1.5 “같은 것”과 “다른 것”에 대한 고찰

- 객체 지향 분석을 위해서는 “같은 것”과 “다른 것”을 구별할 줄 아는 능력 필요
  - “객체”들의 구분을 통해서는 새로운 클래스를 유도
  - “클래스”들의 구분을 통해서는 클래스들 간의 상속 관계 유도
- 예) “비행기와 달이 같은가 혹은 다른가 ?”
  - 보는 관점에 따라 답이 다를 수 있음
- 객체나 클래스를 구분 짓는 기준은 상대적
- 응용 분야에 따라 다르게 모델링 되어야 함
  - 예) 모양을 기준으로 객체를 구별하는 경우, “달과 동전이 같다.”, “달과 비행기는 다르다.”

## 2.1.6 공급자와 소비자

- 공급자 (supplier)
  - 어떠한 클래스를 제작하여 다른 사람이 사용할 수 있도록 제공하는 사람
- 소비자 (consumer, = 사용자)
  - 제공된 클래스의 서비스를 이용하는 사람

예) 프로젝트에 참여하고 있는 사람

- 한편으로는 “소비자”의 입장에, 다른 한편으로는 “공급자”의 입장에 있게 됨

예) 다른 사람이 만들어 놓은 “Linked List” 클래스를 이용해 다른 사람이 사용할 수 있는 “Stack” 클래스를 작성



## □ 명세와 구현의 구별이 필요한 이유는 ?

- 정보 은닉(information hiding, data encapsulation)을 달성하기 위함
  - 소프트웨어 부품의 상세한 내용을 외부 소비자가 모르도록 하는 원리
- 시스템의 복잡도를 제어하기 위한 수단임

## □ 명세(Specification): 소비자 쪽의 편리성

- Public 문장에 의해 외부에 공개되는 메소드 이름
- 소비자 입장에서 최소의 노력으로 이해할 수 있게
- 공개된 서비스(클래스)의 사용 목적과 사용 방법

## □ 구현(Implementation): 공급자 쪽의 독립성

- 클래스 내부의 자료구조와 메소드의 본체
- 소비자에게는 숨겨져 있어도 되는 부분

예) “아스피린” 약

- 소비자: 효능, 용법, 공급자: 성분, 제조, 질량 등

## 2.1.7 객체의 부속품

### □ 분할 정복(divide and conquer)

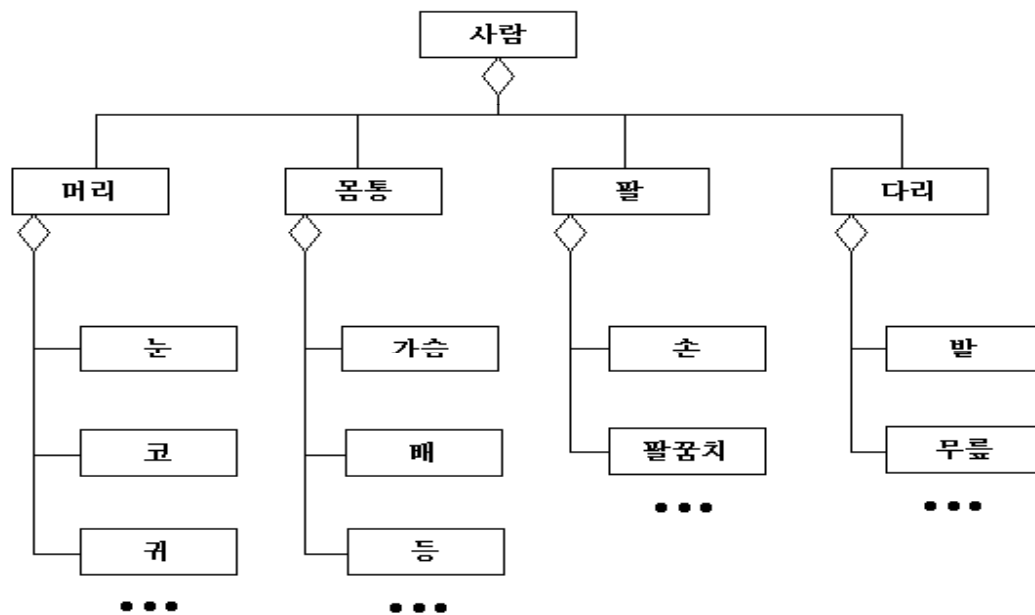
- 어떠한 객체를 해석할 때 그 객체의 구성 요소들을 통해 이해
- 주 객체를 객체의 부품들로 나누어 이해하고 개별 부품을 설계한 후에, 이들을 모아 주 객체를 설계하는 방식

### • 주 객체와 부속 객체는 HAS\_A (= PART\_OF) 관계를 가짐

- 자동차는 엔진을 구성 요소로 갖는다.
- A car has an engine.
- An engine is a part of a car.

- 집합화(aggregation) 관계라고도 함

- 집합화도 일반화와 같이 계층적 트리로 표현 가능
- AND 관계 (= HAS\_A)
  - 사람은 머리 AND 몸통 AND 팔 AND 다리로 구성된다.



HAS\_A 관계의 UML 표기법 (마름모 표시)

## 2.2 자료추상화

- 객체 지향 기법의 기본이 되는 원리
- 객체 정의 시 데이터와 데이터에 적용 가능한 연산을 함께 정의하는 방식
- 외부와 독립적으로 다루어질 수 있는 소프트웨어 부품을 구현하기 위함
  - 외부의 요구 변화에 영향을 덜 받음
- 자료추상화의 주 목적은 소프트웨어를 “명세부”와 “구현부”로 분리하여 정보를 은닉하는 것
  - 소프트웨어 생산자와 소비자의 관계가 분명해짐

## □ Data Abstraction

- 데이터 + 연산 → 클래스(타입)
- Information hiding (명세부/구현부 분리)

## □ 자료 추상화를 이용하여 S/W 객체를 기술하기 위한 요소들

### 1. Data definition (OOP에 적용)

- 컴퓨터 내부에서 표현하고자 하는 자료 구조 정의
- 공급자가 관심있는 구현의 대상

### 2. Operator definition (OOP에 적용)

- S/W 객체가 제공하는 함수 명칭과 시그너처
- 객체 소비자에게 공개되는 명세부

### 3. Axiom (아직은 OOP에 적용되고 있지 못함)

- 함수를 만들어야 하는 필요성 또는 요구사항 명시

### 4. Exception Handling (OOP에 적용)

- 함수 실행 시 발생 가능한 에러의 종류 나열

## □ STACK 정의를 자료추상화로 기술한 예(코드 2.1)

```
1: structure STACK(object)
2:   declare CREATE() -> stack
3:   ADD(object,stack) -> stack /* push */
4:   DELETE(stack) -> stack /* pop */
5:   TOP(stack) -> object
6:   ISEMTS(stack) -> boolean;
7:   for all S in stack, i in object let
8:     ISEMTS(CREATE) ::= true
9:     ISEMTS(ADD(i,S)) ::= false
10:    DELETE(CREATE) ::= error
11:    DELETE(ADD(i,S)) ::= S
12:    TOP(CREATE) ::= error
13:    TOP(ADD(i,S)) ::= i
14:   end
15: end STACK
16: CREATE(stack) ::= var stack: array[1..N] of objects;
17:    top: 0..N;
18: ISEMTS(stack) ::= if top = 0 then true
19:    else false;
20: TOP(stack) ::= if top = 0 then error
21:    else stack[top];
```

정의구역(domain)      치역(range)

“명세부”  
스택이 제공하는  
연산자 나열

“공리”  
스택이 만족해야  
할 조건 명시

스택 내부  
자료구조

연산자  
상세 구현사항  
& 예외 조건 명시

- 자료 추상화를 실제로 구현하는 프로그래밍 관점에 4가지 요소를 모두 수용할 수는 없다.
  - C++
    - 2.0 : 자료 정의, 연산자 정의
    - 3.0 : 예외 처리
  - C
    - 헤더 파일에 관련된 데이터와 함수를 함께 선언하려고 노력
    - 자료 추상화를 염두에 둔 결과
- 클래스 (Class)
  - 객체 지향 언어에서 자료 추상화를 지원하는 구조
  - 서비스 공급자와 소비자 입장을 구분할 수 있도록, “명세”와 “구현”을 분리하여 정의
  - 기능상으로 분명하며 독립적으로 분리된 클래스는 쉽게 재사용 가능

- 코드 2.1의 스택 명세로부터 구현한 예
  - 코드 2.2 자바 프로그램 : Hello.java

```

1: class StackEmptyException extends Exception {
2:     StackEmptyException() {
3:         super("Stack is empty.");
4:     }
5: }
6: class STACK {
7:     public STACK() { // (코드 2.1)의 라인 2 에 매치 됨
8:         top = -1;
9:         stack = new Object[N];
10:    }
11:    public void ADD(Object obj) { // (코드 2.1)의 라인 3 에 매치 됨
12:        top++;
13:        stack[top] = obj;
14:    }
15:    public Object DELETE() { // (코드 2.1)의 라인 4 에 매치 됨
16:        Object obj = stack[top];
17:        top--;
18:        return obj;
19:    }

```

   : 명세부 <    : 구현부

```

20: public Object TOP() throws StackEmptyException { // (코드 2.1)의 라인 5 에 매치 됨
21:     if (top == -1)
22:         throw new StackEmptyException(); // (코드 2.1)의 라인 20 에 매치 됨
23:     return stack[top]; // (코드 2.1)의 라인 21 에 매치 됨
24: }
25: public boolean ISEMTS() { // (코드 2.1)의 라인 6 에 매치 됨
26:     if (top == -1) return true; // (코드 2.1)의 라인 18 에 매치 됨
27:     return false; // (코드 2.1)의 라인 19 에 매치 됨
28: }
29: private int N = 100; // 구현시 필요한 상수값 정의
30: private Object stack[]; // (코드 2.1)의 라인 16 에 매치 됨
31: private int top; // (코드 2.1)의 라인 17 에 매치 됨
32: }
33: public class Hello {
34:     public static void main(String[] args) {
35:         try {
36:             STACK sampleStack = new STACK();
37:             sampleStack.ADD("hello");
38:             sampleStack.ADD("world");
39:             System.out.println(sampleStack.DELETE());
40:             System.out.println(sampleStack.DELETE());
41:             sampleStack.TOP();
42:         } catch (StackEmptyException exception) {
43:             System.out.println(exception.toString());
44:             System.out.println("This process will be terminated immediately.");
45:             java.lang.System.exit(-1);
46:         }
47:     }
48: }

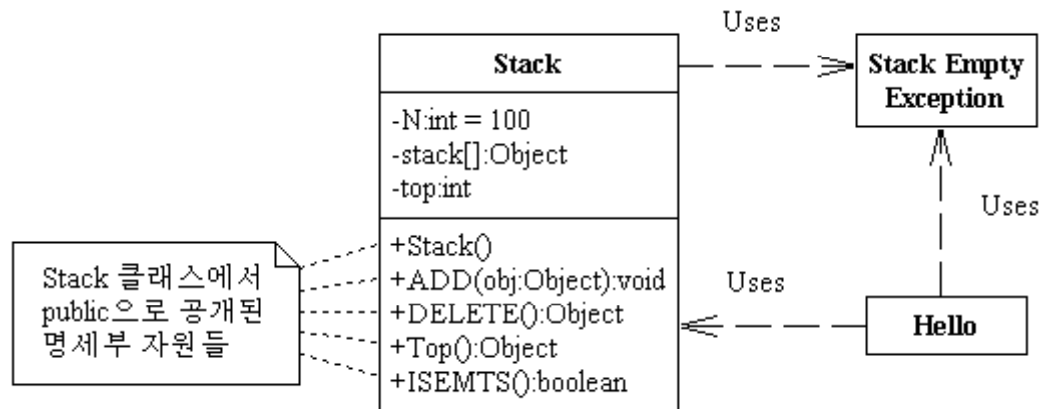
```

## □ (코드 2.2) 이해 시 필요한 점검 사항

- 자료 추상화를 이용한 명세가 구현에 어떠한 영향을 끼치는가 ?
- 코드 상에서 명세부와 구현부의 구분은 ?
  - 공개된 함수의 시그니처(사용법)와 목적
  - 그 외의 것들
- Axiom의 역할은 무엇인가 ?
  - 요구 사항 및 테스트
- 사용자의 부담을 덜어주는 요인은 무엇인가 ?
  - 명세부와 구현부의 부피 차이

## □ (코드 2.2) 클래스들간의 의존관계

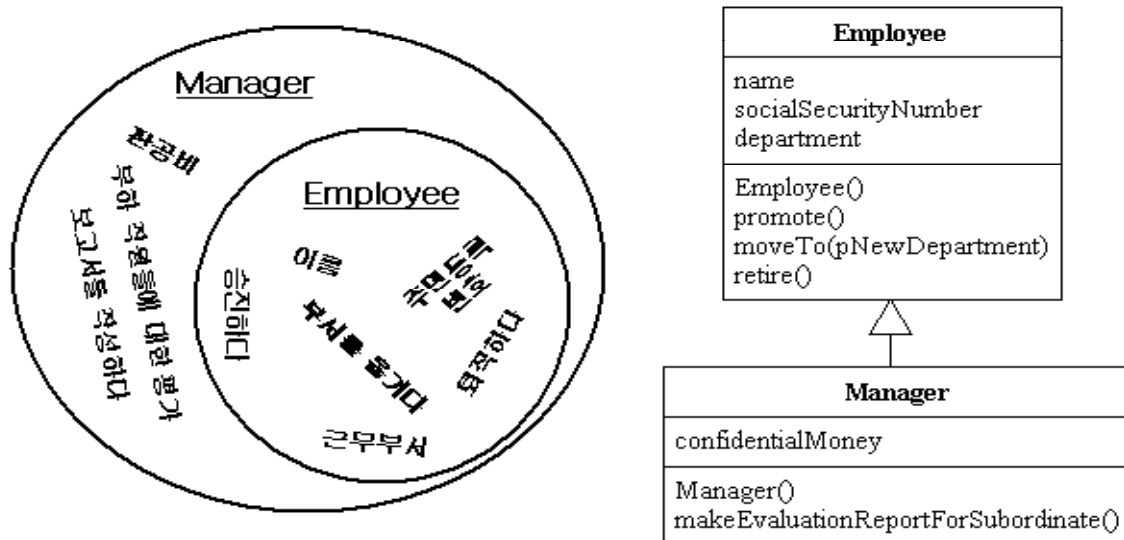
- 의존관계 (= 사용관계, 종속관계, USES관계)
  - UML에서 점선으로 표시
  - Caller:Callee, User:Supplier의 관계



## 2.3 상속

- 상속 (Inheritance)
  - 공통적인 속성과 연산을 공유할 수 있게 하는 기능
  - 기존에 만들어진 클래스의 속성과 연산을 재사용하기 위한 수단
- 하위 클래스에서 이루어지는 작업
  - 속성의 경우에는 추가 작업을 주로 함
  - 연산의 경우 추가 및 기존 연산의 수정 작업을 함
- 상위 클래스 = 부모 클래스 = 기반 클래스
- 하위 클래스 = 자식 클래스 = 파생 클래스

- IS\_A 관계의 확인이 반드시 필요함
- 라이브러리로 제공되는 상속 트리의 중요성:
  - 이해의 용이성



## □ 상속의 중요성

- 상속 트리가 시스템의 골격을 형성
- 프로젝트 수행 초기에 잘못 분석된 상속트리를 중간에 변경하는 경우
  - 엄청난 비용과 노력 부담해야
  - 유연성을 살리지 못함
- 단순히 한 두 가지의 자료 구조와 연산이 추가된다고 새로운 파생 클래스를 만드는 것은 위험
- 실 객체가 갖는 본질적 특성을 이해하고, 클래스간의 IS\_A 관계를 인식하며, 차후 변경 및 추가의 가능성을 고려해야 함

## 2.4 다형 개념

- 다형 개념 (Polymorphism)
  - Poly (다수, 많음) + Morphism (형태)
  - “같으면서도 다른 것” 혹은 “다르면서도 같은 것”을 다루기 위한 개념
  - OOP 적용 시에 두 가지의 경우로 나눌 수 있음
    - (1) Polymorphic Container (= Polymorphic List) 경우
    - (2) Operator Overloading 경우
  - 상속과도 밀접한 연관이 있으며 공통의 서비스를 잘 이해해야 함

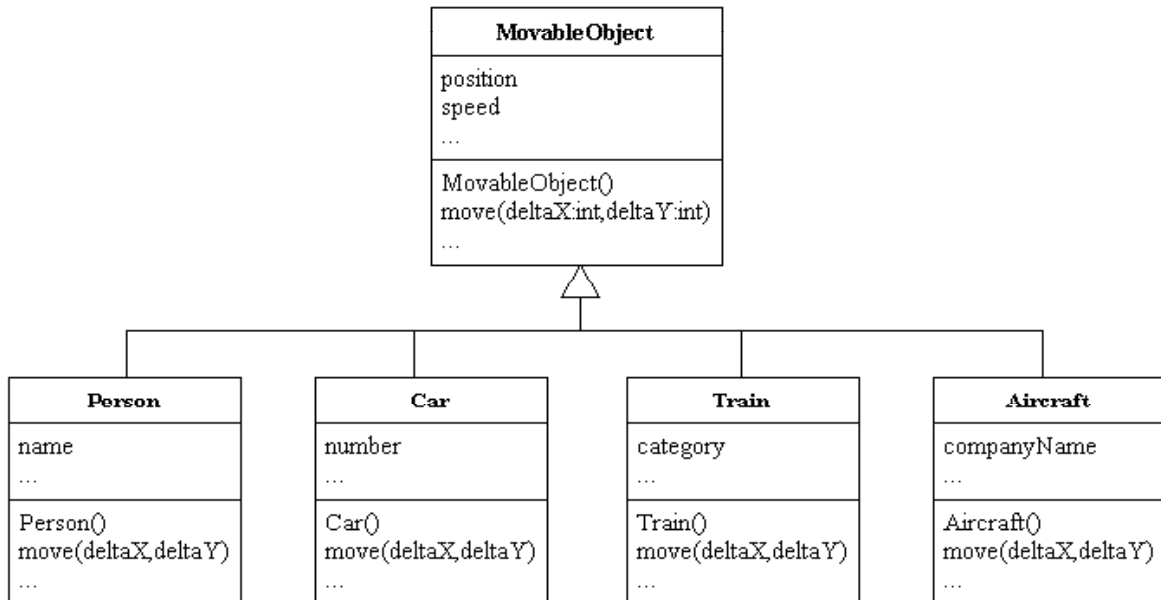
### 1. Polymorphic Container (= List)

- 동질적인 객체 리스트의 예
  - cf) studentList = {(“성규환”,100,80,90), (“이충실”,30,50,60), (“홍필희”,70,60,80), ...}
- 이질적인 객체 리스트의 예
  - **someList = { “사람 객체 김태균”, “자동차 객체 부산 30 더 6875”, “사람 객체 김은영”, “비행기 객체 KAL 700 편”, “기차 객체 새마을 123 편”, “비행기 객체 ASIANA 300 편” }**
- 주의) someList의 내용이 스트링이 아님
  - 논리적인 실 객체를 의미함
- someList에 속하는 객체들이 같은 것인가 다른 것인가 ?
- 이질적인 객체를 포함하는 리스트를 작성하여 프로그램의 유연성을 달성하고자 함
  - 모두 “움직일 수 있다”는 공통점 가짐



## ❖ 상속과의 관계

- 객체들은 개별 클래스의 관점에서는 다르지만 ...
- 상위 클래스의 관점에서는 같다.
- 그 이유는 **공통의 서비스**를 제공하기 때문이다.



## 2. Operator Overloading (연산자 중복)

- Operator Overloading 시조:

$1 + 2$

$1.0 + 2.0$

"1.0" + "2.0"

- **Person** 클래스의 **move()** 연산과 **Car** 클래스의 **move()** 연산은 같은 것인가 다른 것인가 ?
  - 서비스(연산자 이름)는 같고 메소드는 다르다.
  - 함수 중복 (method overloading)의 경우
- 이 경우도 상속과 관련이 있음
- 다형 개념은 연산자의 호출에 따른 메소드의 선택이 **실행 시에** 이루어지는 동적 바인딩 기능에 의해 효용성이 입증 됨

## 2.5 동적 바인딩

- 객체 지향 기법을 사용하는 중요한 이유
  - 사용자의 요구 변화에 적절히 대응하는 시스템 구현
  - 공급자가 제공하는 서비스의 추가나 변경이 발생했을 경우, 사용자가 영향을 받지 않도록 하는 방법 필요
  - 동적 바인딩
- 함수 바인딩
  - 함수 호출과 호출된 함수를 연결하는 메커니즘

- Fortran 프로그램에서 호출문의 예
  - 서브루틴 SORT()의 본체가 어디엔가 있다고 가정

```
...  
CALL SORT(DATA)  
...
```

- 컴파일 작업과 링크 작업이 끝난 후의 가상 어셈블리 코드

```
...  
현재 상태를 스택에 저장하고 actual parameter를 함수에 넘겨주기 위한 코  
드 부분  
branch 12345  
...
```

- 함수 호출을 위한 컴파일 및 링크 작업의 가장 중요한 결  
과는 호출될 함수의 논리적 주소 할당과 이의 연결
  - 12345 는 SORT() 함수가 존재하는 논리적인 번지수
  - 주 프로그램의 호출문은 어셈블리어에서는 분기문으로 변환됨
- 이때 분기해야 될 주소를 결정하는 것이 바인딩 작업의 임  
무임

## □ 정적 바인딩 (Static Bining)

- 함수 호출을 위한 바인딩이 컴파일, 또는 링크 또는 프로그램 로드 시까지는 결정되는 기법

## □ 동적 바인딩 (Dynamic Binding)

- 함수 호출을 위한 바인딩이 실행 시에 결정됨
- 장점 1: “엄청난” 유연성 제공
- 장점 2: 시스템의 느슨한 연결을 가능하게 함
- 장점 3: 많은 양의 조건문이 사라짐
- 단점 1: 실행 속도 저하
- 단점 2: static analysis를 이용한 프로그램의 정확성 검증이 어려움

### ❖ 동적 바인딩을 이용한 프로그래밍 스타일 (완전한 C++ 아님)

```
1: MovableObjectList someList;
2: ...
3: // someList에 객체들을 삽입하는 부분
4: someList.addTail(new Person("김 태균",MALE,100,200));
5: someList.addTail(new Car("부산 30 더 6875","비스토",110,210));
6: someList.addTail(new Person("김 은영",FEMALE,90,190));
7: someList.addTail(new Aircraft("KAL",500,115,200,0));
8: someList.addTail(new Train("새마을 123",10,100,100));
9: someList.addTail(new Aircraft("ASIANA 300",300,110,200,0));
10: ...
11: POSITION pos = someList.getHeadPosition(pos);
12: while (pos != NULL) {
13:     MovableObject *pMovingObject = someList.getNext(pos);
14:     pMovingObject->move( $\Delta x$ , $\Delta y$ );
15: }
16: ...
```

## ❖ 정적 바인딩을 이용한 프로그래밍 스타일

### ■ (코드 2.4)의 (a)

- (코드 2.3)과 같은 작업을 하기 위한 프로그래밍 스타일

### ■ (코드 2.4)의 (b)

- 구조적 언어에서 새로운 클래스가 추가될 때 해야 할 작업

### ■ 시스템의 유지보수 시에 엄청난 양의 조건문을 추가해야 함

```
1: struct MovableObjectList someList;
2: union MovableObject *ptr;
3: ...
4: /* someList에 객체를 삽입하는 부분 */
5: ptr = (union MovableObject *) malloc (sizeof(union MovableObject));
6: ptr->tag = PERSON;      /* 구조적 언어에서는 tag 필드의 사용이 필수적 이다 */
7: ptr->name = strdup("김 태균");
8: ptr->sex = MALE;
9: ptr->x = 100;
10: ptr->y = 200;
11: insertListItem(&someList, ptr);
12: /* 5 - 11 라인의 코딩 스타일을 반복적으로 적용하여 다른 객체를 someList에 삽입함 */
13: ...
14: while (someList의 끝에 도달하기 전까지) {
15:     ptr = getNextItem(&someList);
16:     switch (ptr->tag) {
17:         case PERSON :    movePerson(ptr,Δx,Δy);    break;
18:         case CAR :       moveCar(ptr,Δx,Δy);       break;
19:         case AIRCRAFT :  moveAircraft(ptr,Δx,Δy);   break;
20:         case TRAIN :     moveTrain(ptr,Δx,Δy);      break;
21:     }
22: }
```

(코드 2.4) (a)

(코드 2.4) (b)

```
13: ...
14: while (someList의 끝에 도달하기 전까지) {
15:     ptr = getNextItem(&someList);
16:     switch (ptr->tag) {
17:         case PERSON : movePerson(ptr, Δx, Δy); break;
18:         case CAR : moveCar(ptr, Δx, Δy); break;
19:         case AIRCRAFT : moveAircraft(ptr, Δx, Δy); break;
20:         case TRAIN : moveTrain(ptr, Δx, Δy); break;
21:         case BOAT : moveBoat(ptr, Δx, Δy); break;
22:     }
23: }
23: ...
```

## □ 동적 바인딩과 관련해서 ...

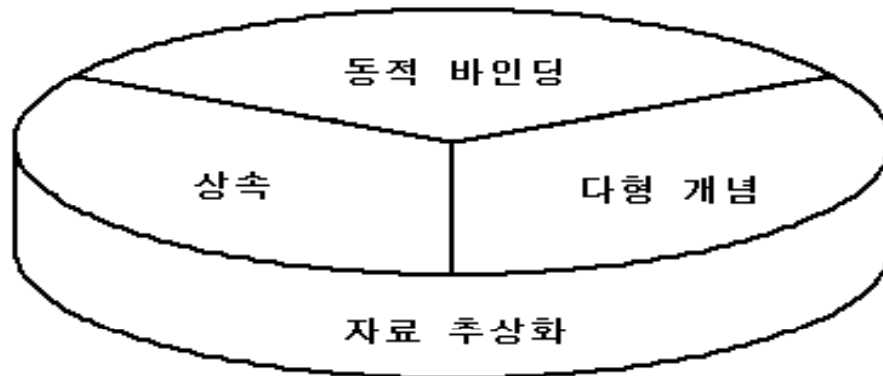
- C++ 경우에는 가상 함수, 포인터의 이용 필요
  - 정적 바인딩과 동적 바인딩을 선택적 적용 가능
- Polymorphic Container와 관련이 깊음
- 마찬가지로 상속과도 관련이 깊음 (상위 클래스의 역할이 막중함)

## □ 메시지 보내기 (message sending)

- 동적 바인딩을 염두에 둔 용어
- “함수 호출”과 물리적인 관점에서는 동일
- 논리적으로는 공급자 객체에 대해 서비스를 요구한다는 의미에서는 다름
- 예) `pMovingObject->move(Δx, Δy);`
  - `pMovingObject`에게  $\Delta x, \Delta y$  만큼 이동하라는 요구 사항을 보내니까 해당 객체가 알아서 처리해라

## □ 객체 지향 삼총사

- 지금까지 다룬 아래의 내용들이 객체 지향성을 나타내는 핵심적인 속성들
- 서로 독립적인 속성이 아니라 서로 맞물려 있는 속성
- 자료추상화 도입 이유
  - 사용자와 공급자의 구별, 정보 은닉의 구현, 명세부와 구현부의 구분을 통한 복잡성 제어 등을 이루기 위함
- 상속, 다형 개념, 동적 바인딩은 서로 분리할 수 없는 삼위일체 요소



- 객체 지향의 장점인 재사용성 향상, 유지 보수의 용이성 달성, 시스템의 복잡성 제어를 위해선 앞의 모든 개념을 모두 적용시켜야 함

[참고] C++로 구현한 시스템에서 해당 개념들이 제대로 적용되었는지 확인하는 방법

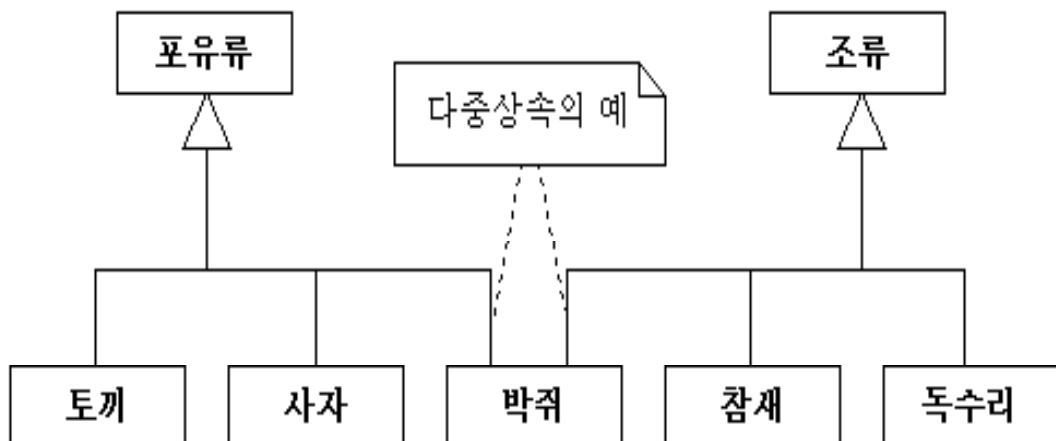
- 상속
  - 쉽게 확인 가능
- 다형 개념
  - 함수 중복 확인 : 상속 트리 내 클래스들 사이에 같은 이름의 함수가 존재하는지 확인
  - 폴리몰픽 리스트 확인 : List, Vector, Array로 끝나는 컨테이너 클래스들이 사용되었는지 확인
- 동적 바인딩
  - 상속 트리 상 중복 함수들이 가상 함수로 정의되었는지 확인

## 2.6 다중 상속

- 상속은 클래스의 재사용을 위한 필수적 기능
- 실세계에서 단순 상속만으로는 표현하기가 애매한 경우 존재
  - 예) 박쥐
    - 포유류 ?, 조류 ?
- 다중 상속
  - 어떠한 클래스가 여러 개의 상위 클래스로부터 자료 구조와 연산을 상속받을 수 있는 것
  - 실세계에 대한 모델이 단순해지며 이해하기 쉽게 됨

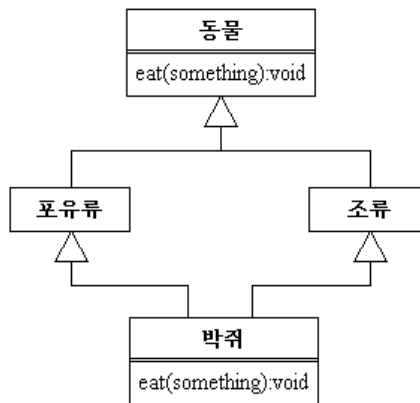
### ▪ 다중상속의 예

- 박쥐, 수륙 양용차, 트리 리스트(tree list) 컨트롤 등



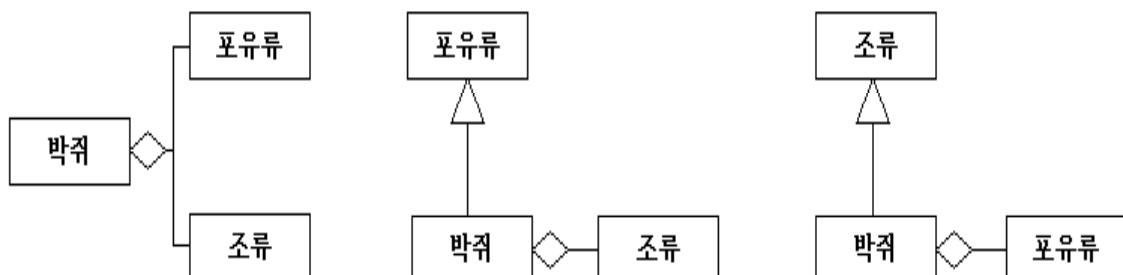
## □ 다중 상속의 장점과 단점

- 장점 1: 자연스러운 모델을 얻을 수 있음
- 단점 1: 프로그래밍 언어에 의존적임
  - 시스템간 이식성 문제 야기
- 단점 2: 부정확한 모델링의 가능성이 있음
  - 상위 클래스의 일부분만 상속받는 경우
- 단점 3: 다이아몬드형 다중 상속
  - (중복된 자원의 구분 필요)



## □ 다중 상속의 구현 방식

- Aggregation을 이용하는 경우
  - 관점에 따라 다르게 표현 가능 : 아래 그림
- Interface를 이용하는 경우 (2.9 절에서 다룸)





- 다중상속을 직접 이용하는 것은 바람직하지 않음
- 다중 상속이 문제를 추상화 시키는 데는 도움이 되지만 구현 시의 어려움 때문에 사용에 주의
- 논리적인 모델링 시에는 다중 상속을 이용하여 표현하고,
- 구현 시에는 중요하다고 생각되는 상위 클래스 한 쪽으로부터만 상속받고, 나머지 기능은 집합화를 이용하는 것이 바람직함
  - 단일 상속과 인터페이스로 구현

## 2.7 추상 클래스

- 추상 클래스 (Abstract class)
  - 직접적으로 객체를 만들지 않는 클래스
  - 클래스 상속 트리에서 비 단말 클래스 중의 일부
  - 하위 구체 클래스들의 사용법을 밝히는 목적을 가짐
  - 상속 트리에서 중복되는 코드를 책임지기 위한 목적
  - Java의 경우 abstract라는 modifier 명시하여 정의
  - C++의 경우 순수 가상 함수를 포함하는 클래스
- 구체 클래스 (Concrete class)
  - 객체를 만들기 위해 사용되는 클래스
  - 클래스 상속 트리에서 단말 노드에 해당하는 클래스

- [참고: C++] 가상 함수(Virtual Function)
  - 파생 클래스가 안전하게 재정의할 수 있는 함수
  - 상속 계층 내에서만 의미가 있다.
  - 가상 함수를 반드시 재정의해야만 하는 것은 아니다.  
기반 클래스의 동작을 그대로 쓰고 싶으면 단순히 상속만 받고 변경할 필요가 있을 때만 재정의하면 된다.
- [참고: C++] 순수 가상 함수(Pure Virtual Function)
  - 파생 클래스에서 반드시 재정의해야 하는 함수
  - 일반적으로 함수의 동작을 정의하는 본체를 가지지 않으며 따라서 이 상태에서는 호출할 수 없다.
  - 본체가 없다는 뜻으로 함수 선언부의 끝에 '=0'이라는 표기를 한다.
    - 이는 함수만 있고 코드는 비어 있다는 뜻이다.

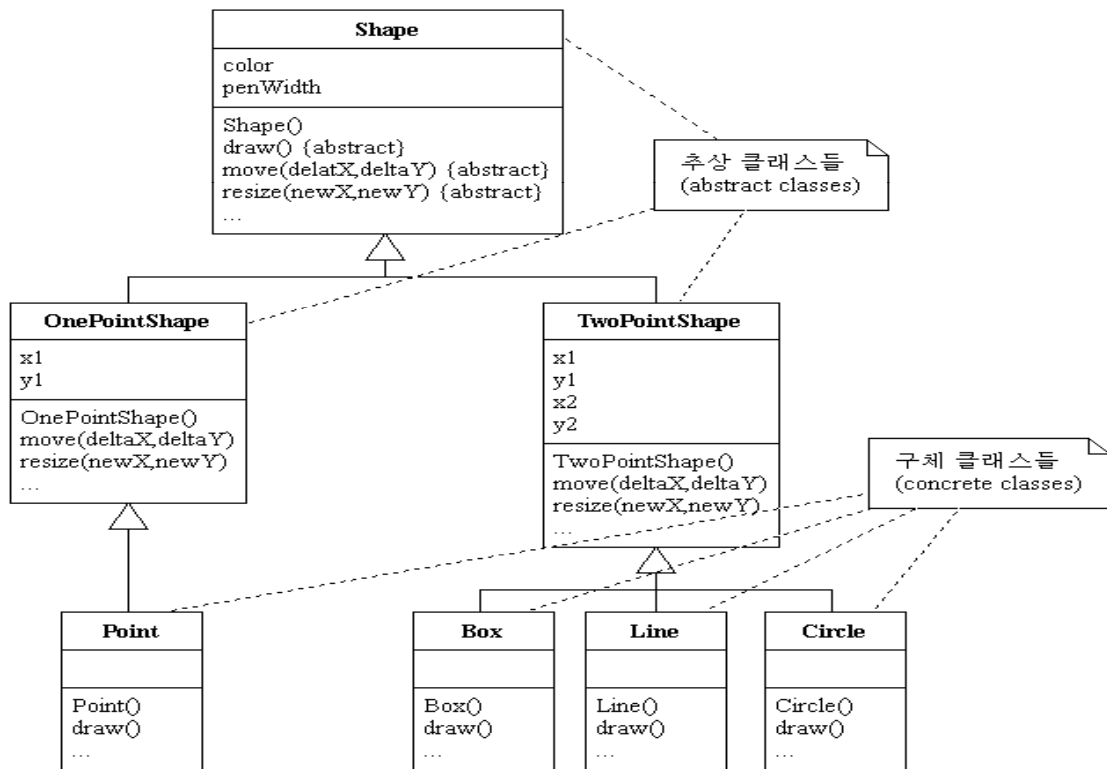
❖ C++에서 가상함수 & 순수가상함수 정의 예

```
class Graphic {           추상클래스
public:
    virtual void Draw()=0; // 순수가상함수
};

class Line : public Graphic {
public:
    virtual void Draw() { // 가상함수
        puts("선을 긋습니다.");
    }
};
```

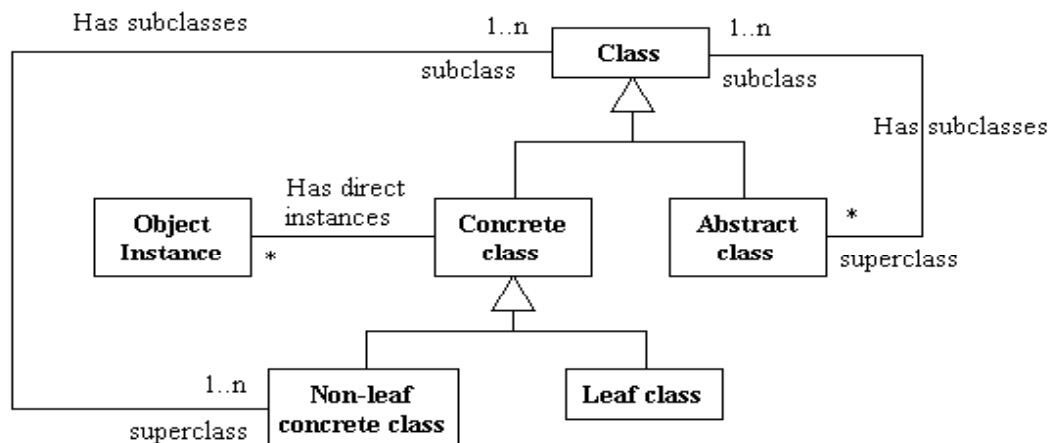
## ❖ 추상클래스와 구체 클래스의 구분

- p.72 주석 (67)



## ❖ 추상클래스와 구체 클래스에 대한 모델

- 객체 지향 시스템에서 클래스에는 구체 클래스나 혹은 추상 클래스가 있다.
- 구체 클래스에는 비단말 구체 클래스나 혹은 단말 클래스가 있다.
- 구체 클래스는 항상 인스턴스 객체를 직접 인스턴시에이션하는 관계를 갖고 있다.
- 추상 클래스와 비단말 구체 클래스는 상속 트리 상에서 한 개 이상의 하위 클래스들을 갖는다.



## ❑ 추상 클래스 (계속)

- 클래스를 인스턴시에이션 시키는 클래스
  - Ada에서 유래
    - 초기에는 상속기능이 없어서, 클래스 재사용을 위해 도입
  - C++의 **template** (= parameterized class, generic class, abstract data type)

## ❖ 추상 클래스의 특징

1. 형(type)을 인자(argument)로 취급함
2. 정적으로 다루어짐
  - 추상 클래스 관련 시스템 작업은 컴파일 시 모두 이루어짐

- [참고: C++] 템플릿 클래스의 객체를 생성하는 순간에 컴파일러 내부적으로 알맞은 클래스를 만든다.
  - 개발자가 만든 코드

```
template< typename A, typename B, int MAX >
class TwoArray
{
    // 중간 생략
    A arr1[ MAX ];
    B arr2[ MAX ];
};
```

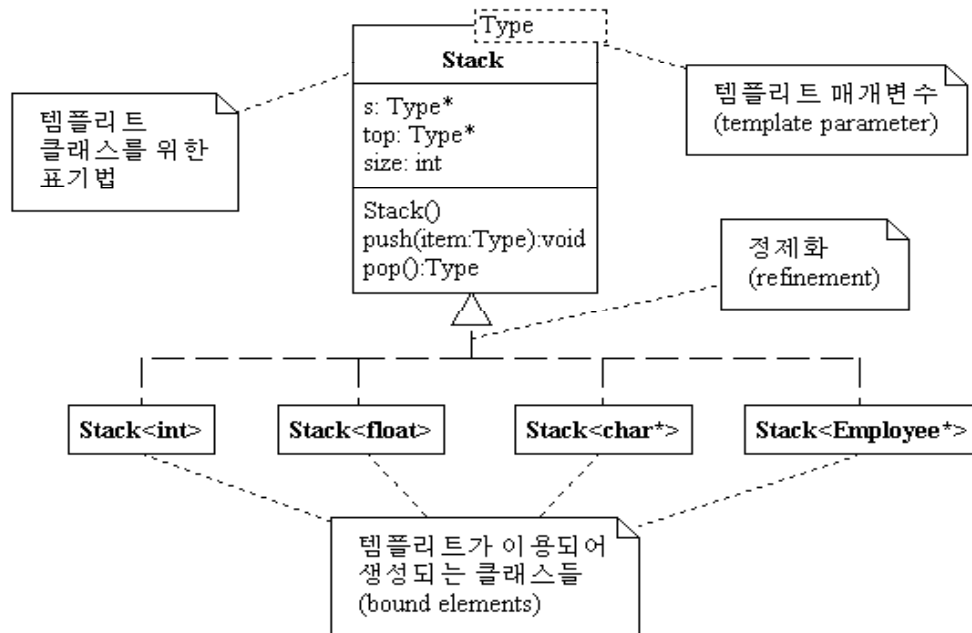
```
TwoArray< char, double, 20 > arr;
```

- 컴파일러에 의해 생성된 클래스

```
class TwoArray_char_double_20    // 이 이름은 임의로 만든 것
{
    // 중간 생략
    char arr1[ 20 ];
    double arr2[ 20 ];
};
```

## ❖ UML의 정제화 표기법

- C++: 템플릿이 클래스를 인스턴시에이션 시키는 경우
- Java: 어떠한 클래스가 인터페이스를 구현하는 경우



- C++에서 추상 클래스로서 템플릿이 도입된 이유는 코딩 분량을 감소시키기 위함
  - But, 목적 코드(object code)의 분량을 감소시키지는 않는다.
  - 매크로 확장 방식(정적인 특성)을 이용하기 때문
- 반드시 필요한 기능인가 ?
  - 동적 바인딩을 통한 시스템의 유연성을 확보하기 위해서는 템플릿을 사용하지 않는 것이 좋다.
  - Java에서는 상속과 인터페이스를 이용하면 됨

## 2.8 예외 처리 기능

- 예외 처리 (Exception Handling)
  - 객체 지향의 본질적 속성은 아님
  - Ada 초기 버전에서 신뢰성 향상을 위해 도입
- 방어적 프로그래밍 (Defensive Programming )
  - 고전적인 방법
  - 시스템 오작동을 방지하기 위해 많은 수의 비교문 추가
- 방어적 프로그래밍으로 인한 코드의 복잡성을 완화시키기 위한 방편이 예외 처리 기능

❑ 아래 C++ 코드에서 발생 가능한 실행 에러는?

```
...  
1: char *p = new char[100];  
2: strcpy(p, "hello");  
...
```

❑ 방어적 프로그래밍 스타일

```
...  
1: char *p = new char[100];  
2: if (p == NULL) {  
3:     cerr << "Fatal error: memory exhausted in this function.";  
4:     ... 이 부분에는 에러 복구를 위한 코드가 있을 수 있음 ...  
5:     return; // 리턴을 하거나 종료를 하거나 계속 진행할 수 있음  
6: }  
7: strcpy(p, "hello");  
...
```

- <문제점 1>

- 에러 발생 가능 장소에서 에러를 처리하는 많은 if 문의 사용
- 프로그램의 크기가 커진다.

- <문제점 2>

- 에러 복구와 관련된 제어가 복잡해 관련 문제 발생
  - 에러 복구 코드에서 에러가 발생한다면...
- 문제 해결과 관련된 제어문(시스템 구현 초기)과 에러 처리를 위한 제어문(시스템 구현 말기)의 구별이 없음
- 프로그램의 이해 및 유지보수를 어렵게 함

- <문제점 3>

- 에러 처리에 대한 사용자의 자율성이 없음
- 소프트웨어 공급자와 소비자 관점에서의 역할 분담
  - 공급자가 정한 방식대로만 에러 처리(시스템 정지 등)됨

→ 위 3가지 문제를 염두에 두고 제시된 기능이 예외 처리

- 물리적 에러, 논리적 에러 처리 가능

```
1: class OverflowException extends Exception {
2:     OverflowException() {
3:         super("Arithmetic Overflow Occurs");
4:     }
5:     OverflowException(int val) {
6:         super("Arithmetic Overflow Occurs :"+val);
7:     }
8: }
9: class IndexGenerator extends Object {
10:     private int index;
11:     IndexGenerator() {
12:         index = 0;
13:     }
14:     IndexGenerator(int i) {
15:         index = i;
16:     }
17:     int initIndex() {
18:         return index;
19:     }
20:     int nextIndex() throws OverflowException {
21:         index++;
22:         if (index < 0)
23:             throw new OverflowException(index);
24:         return index;
25:     }
26: }
```

```

27: public class ExceptionTest {
28:     public static void main(String[] args) {
29:         System.out.println("Hello !");
30:         try {
31:             IndexGenerator generator = new IndexGenerator();
32:             for(int i = generator.initIndex(); ; i = generator.nextIndex()) {
33:                 System.out.println("i = "+i);
34:             }
35:         } catch (OverflowException e) {
36:             System.err.println("In main() function: "+e.toString());
37:             java.lang.System.exit(1);
38:         }
39:         System.out.println("Good Bye !");
40:     }
41: }

```

Hello !

i = 0

i = 1

i = 2

i = 3

...

i = 2147483646

i = 2147483647

In main() function: **OverflowException: Arithmetic Overflow Occurs :-2147483648**

실행결과

## 2.8.1 <문제점 1>의 해결

### ❑ 문제점이 발생한 이유

- 에러 발생 장소에서 에러를 즉시 처리하도록 코딩함

### ❑ 해결 방법

- 에러 발생 장소와 에러 처리 장소를 분리함

- **에러 발생**: throw 명령어

특히 이 명령어가 매우 짧음

...

if (any error case)

throw new AnyException(some error information);

...

- **에러 처리**:

- 사용자가 처리 : try ... catch 블록



## 2.8.2 <문제점 2>의 해결

### □ 문제점이 발생한 이유

- 에러 처리를 위한 코드 작성 방법에 특별한 문법이 존재하지 않음

### □ 해결 방법

- 에러 처리를 위한 제어 구조를 명시하는 문법 구조 정의
- catch 블록에 에러처리 코드 위치
- Incremental development를 통한 점진적인 신뢰성 확보 가능
  - 개발 초기에는 문제 본질 해결하기 위해 코딩 수행하고, 점차 신뢰성 확보를 위해 에러처리 루틴 추가

❖ 예) 코드 2.6: 예외를 고려하지 않은 경우

```
1: class SampleArray {
2:     private int v[];
3:     SampleArray() {
4:         v = new int[10];
5:         for(int i = 0; i < v.length; i++) v[i] = 0;
6:     }
7:     void setData(int index, int data) {
8:         v[index] = data;
9:     }
10:    int getData(int index) {
11:        return v[index];
12:    }
13: }
```

- 다음과 같이 사용하게 되면

### ArrayIndexOutOfBoundsException 예외 발생

- 위 예외는 자바에서 RuntimeException의 하위 클래스이기 때문에 사용자 프로그램에서 catch 블록을 작성하지 않더라도 자동적으로 예외가 잡힌다.

```
...  
SampleArray testArray = new SampleArray();  
testArray.setData(20,100);  
...
```

### ❖ 예) 코드 2.7: 예외 발생 사실만을 알림

```
1: class SampleArray {  
2:     private int v[];  
3:     SampleArray() {  
4:         v = new int[10];  
5:         for(int i = 0; i < v.length; i++) v[i] = 0;  
6:     }  
7:     void setData(int index,int data) {  
8:         try {  
9:             v[index] = data;  
10:        } catch(ArrayIndexOutOfBoundsException ex) {  
11:            System.out.println(ex+" occurs in SampleArray.setData() method.");  
12:        }  
13:    }  
14:    int getData(int index) {  
15:        return v[index];  
16:    }  
17: }
```

예외 처리가 단순 디버깅 목적

## ❖ 예) 코드 2.8: 예외 처리를 통해 에러 복구

```
...
7: void setData(int index,int data) {
8:     try {
9:         v[index] = data;           // 문제의 본질을 해결하기 위한 코드
10:    } catch(ArrayIndexOutOfBoundsException ex) {
11:        int saved[] = v;           예외 처리 위한 코드
12:        v = new int[index+1];
13:        for(int i = 0; i < saved.length; i++)
14:            v[i] = saved[i];
15:        for(int i = saved.length; i < index; i++)
16:            v[i] = 0;
17:        v[index] = data;
18:        System.out.println(ex+" occurs in SampleArray.setData() method.");
19:        System.out.println("But the error is recovered by extending array size.");
20:    }
21: }
...
```

### 2.8.3 <문제점 3>의 해결

#### ❑ 문제점이 발생한 이유

- 라이브러리 함수에서는 예외 상황이 발생했을 때, 처리 방식이 미리 결정되어 있음
- 서비스 사용자 입장에서 선택의 여지가 없음

#### ❑ 해결 방법

- 사용자와 공급자의 역할 분담
- 공급자 – 예외 발생 사실만 보고
- 사용자 – 예외 발생에 대한 처리 담당
  - 3가지 대응방식
    - i) 예외 전달
    - ii) 예외를 인식하되 예외 처리는 하지 않기
    - iii) 예외 처리

❖ 예) 코드 2.9: 예외를 전달하는 경우

```
1: class TestArray {
2:     private int v[];
3:     private IndexGenerator generator = new IndexGenerator();
4:     ...                      setData()를 호출한 상위메소드에게 에러 처리를 넘김
5:     void setData(int data) throws OverflowException {
6:         int i = generator.nextIndex();
7:         v[i] = data;
8:     }
9: }
```

❖ 예) 코드 2.10: 예외를 인식하되 처리하지 않는 경우

```
1: class TestArray {
2:     private int v[];
3:     private IndexGenerator generator = new IndexGenerator();
4:     ...
5:     void setData(int data) {
6:         try {
7:             int i = generator.nextIndex();
8:         } catch (OverflowException e) {
9:             System.err.println("In TestArray.setData() function: "+e.toString());
10:            java.lang.System.exit(1);
11:        }
12:        v[i] = data;
13:    }
14: }
```

- 왜 에러가 발생했는지만 알 수 있음
- 프로그램의 신뢰성 향상에는 도움되지 않음
- 초보자의 코딩 스타일

## ❖ 예) 코드 2.11: 예외 처리를 하는 경우

```
1: class TestArray {
2:     private int v[];
3:     private IndexGenerator generator = new IndexGenerator();
4:     ...
5:     void setData(int data) {
6:         int i;
7:         try {
8:             i = generator.nextIndex();
9:         } catch (OverflowException e) {
10:             System.err.println("In TestArray.setData() function: "+e.toString());
11:             System.err.println("The IndexGenerator is reset to zero.");
12:             generator = new IndexGenerator();
13:             i = generator.initIndex();
14:         }
15:         v[i] = data;
16:     }
17: }
```

- 인덱스를 초기화 시킴
- 완벽한 예외 처리는 아니지만 상대적으로 안정된 처리 방법

## □ 기타 사항

- 소프트웨어 공학의 비용 산정 통계에 따르면, 신뢰성 있는 프로그램 작성 비용은 고려하지 않는 경우보다 2배의 생산비 소요됨
- 라이브러리로 제공되는 입출력 함수의 사용 시에는 exception handling 이 반드시 필요하다. 그 이유는 ?
  - 잘못된 데이터 가지고 처리해 봐야 잘못된 결과 나오므로..
- 예외처리의 오용 주의 (코드 2.12)
  - 예외처리는 시스템의 이상 상태 제어 목적으로만 사용해야 함
  - 정상 제어문은 포함해서는 안됨

❖ 예) 코드 2.12: 예외 처리를 남용하는 경우

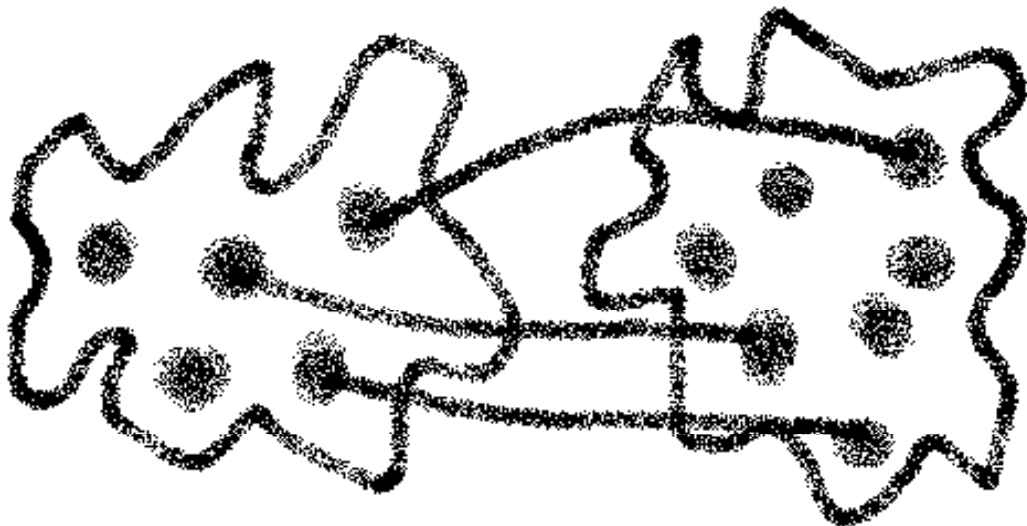
```
1: class EndOfDataException extends Exception {
2:     int count = 0;
3:     EndOfDataException(int val) {
4:         super("End Of Data:"+val);
5:         count = val;
6:     }
7: }
8: class ArrayElements extends Object {
9:     private int v[];
10:    private int n;
11:    ArrayElements(int n) {
12:        this.n = n;
13:        v = new int[n];
14:    }
15:    void setData(int i,int data) {
16:        v[i] = data;
17:    }
18:    int getData(int i) throws EndOfDataException {
19:        if (i == n) throw new EndOfDataException(n);
20:        return v[i];
21:    }
22: }
```

```
23: public class WrongExceptionUsage {
24:     public static void main(String[] args) {
25:         ArrayElements data = new ArrayElements(10);
26:         for (int i = 0; i < 10; i++)
27:             data.setData(i,i*10);
28:         int total = 0;
29:         try {
30:             for (int i = 0; ; i++)
31:                 total = total + data.getData(i);
32:         } catch (EndOfDataException e) {
33:             int n = e.count;
34:             System.out.println("Average = "+total/n);
35:         }
36:     }
37: }
```

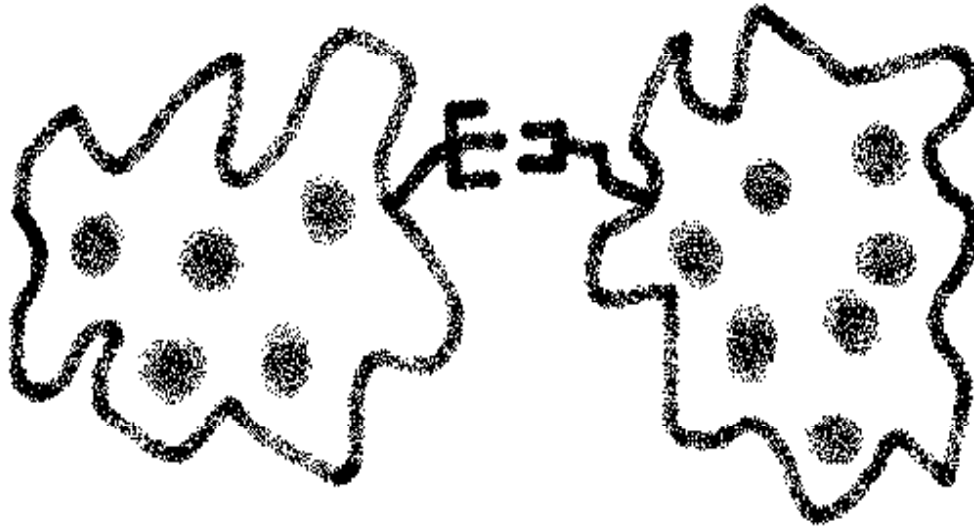
## 2.9 인터페이스

- 일반적인 용어(사전적인 의미)로서의 뜻
  - 컴퓨터 시스템을 구성하는 요소들을 연결하는 접속점
- 프로그래밍에서의 의미
  1. 사용자 관점에서 공급된 소프트웨어의 사용법을 나타내는 명세부로서의 의미를 가짐
    - 인터페이스 = 명세
    - API (Application Programming Interface)
      - 응용 프로그램을 개발하기 위한 함수의 집합
      - 라이브러리 형태로 제공되는 함수의 용도, 목적, 사용법 포함
      - 소프트웨어 라이브러리의 매뉴얼
    - 예) 클래스의 인터페이스가 잘 되어 있다는 의미는
      - 그 클래스에서 public으로 명시된 함수들의 시그니처가 잘 정의되어 있고, 그 함수들의 목적이 명확히 정의되어 있다는 의미

❖ 시스템의 연결(인터페이스를 이용하지 않는 경우)



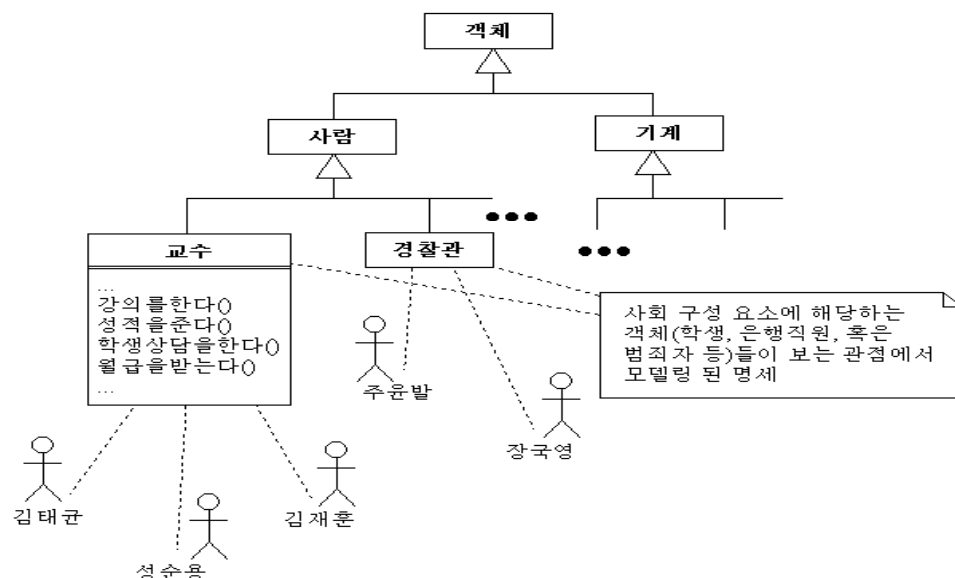
## ❖시스템의 연결(인터페이스를 이용하는 경우)



### 2.9.2 사물을 보는 관점

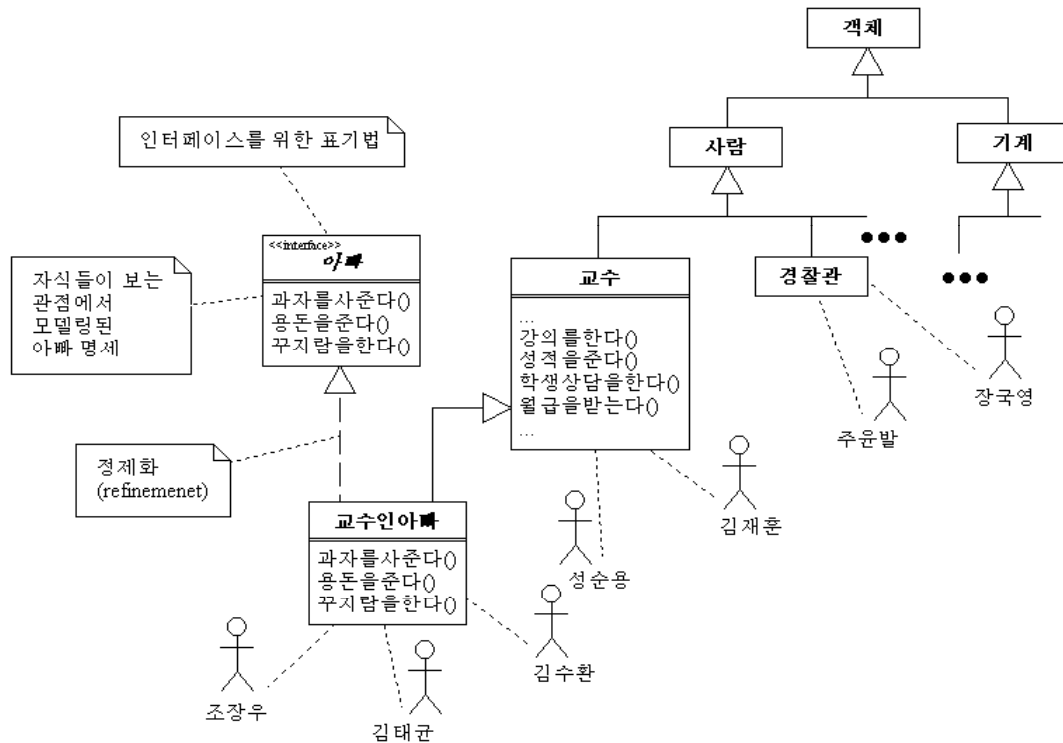
#### 2. 객체가 제공하는 다양한 특성을 표현하기 위한 수단으로 인터페이스가 사용됨

- 객체간의 관계에서 사물을 보는 다양한 관점을 프로그래밍에 반영하기 위한 방법
- 교수 객체를 모델링한 예





- 아빠로서의 인터페이스가 고려된 모델



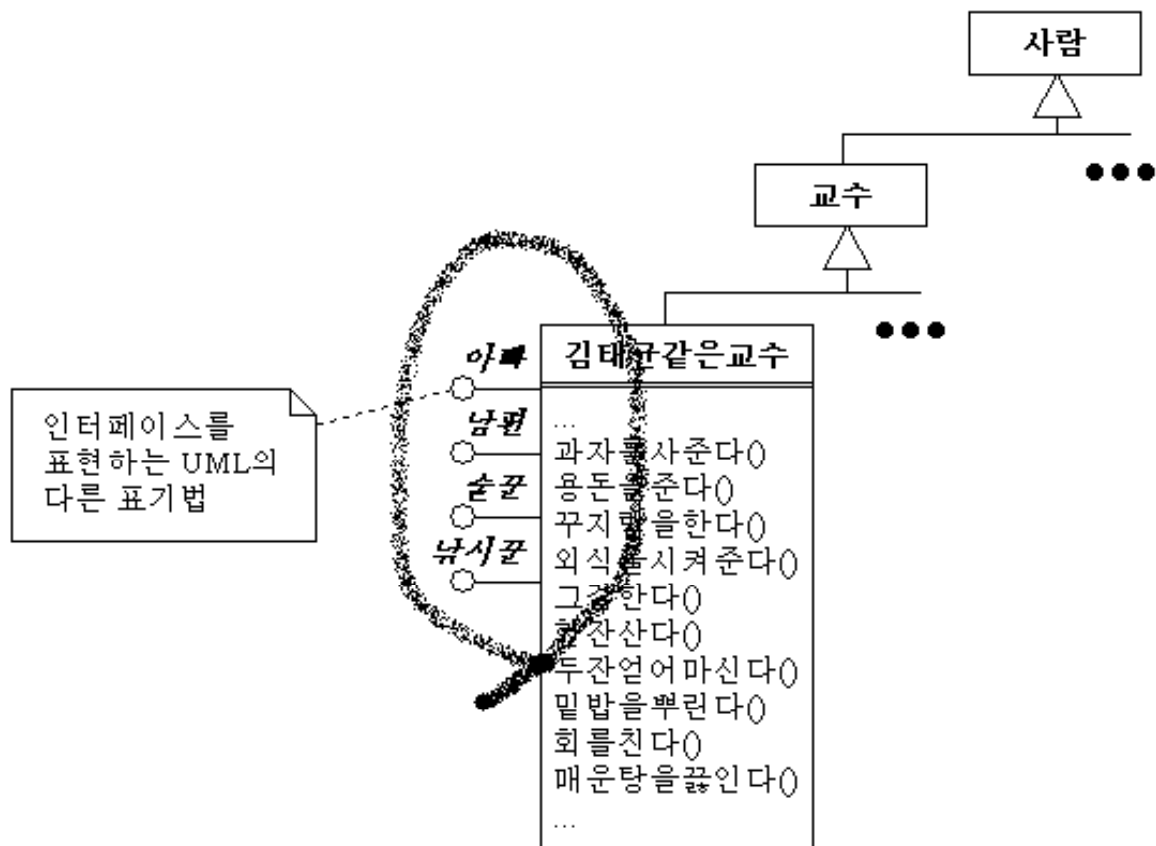
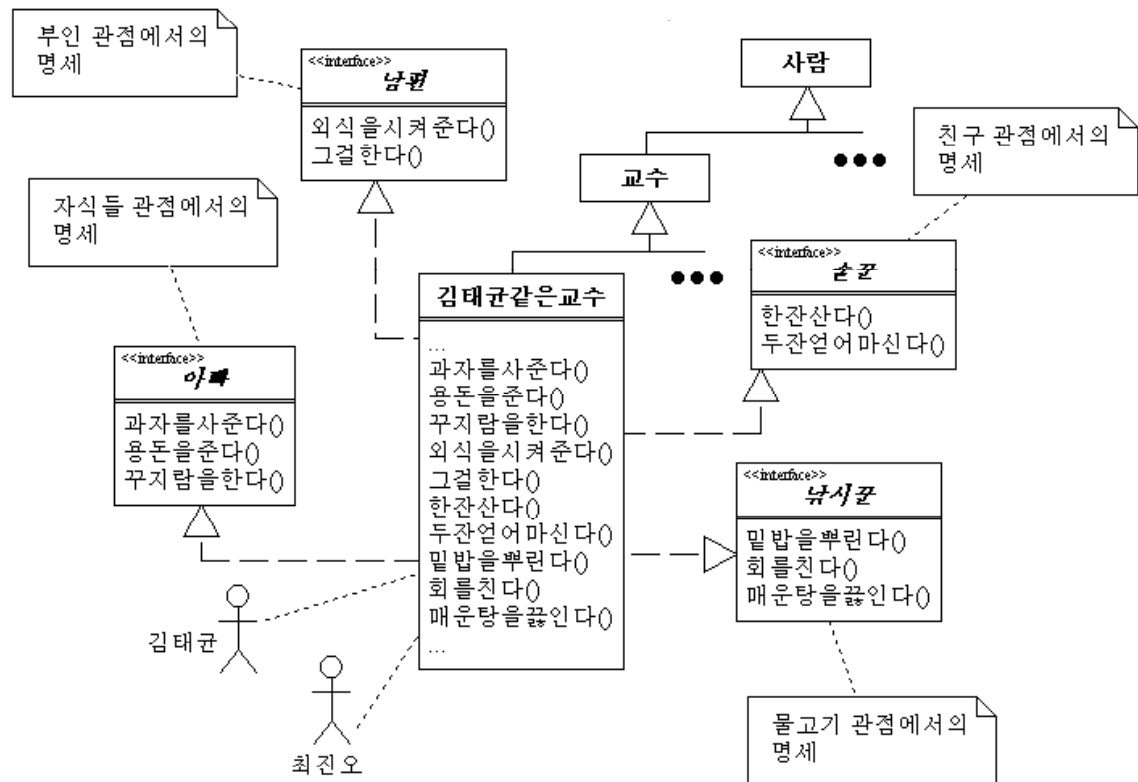
- 구현 상속

- 일반적인 상속을 통하여 데이터 멤버와 메소드 내용을 모두 물려받는 경우
- 앞 예에서 “교수” 클래스와 “교수인아빠” 클래스의 관계

- 인터페이스 상속

- 상위 클래스의 명세(함수의 시그너처)만 물려받는 것
- 하위 클래스에서는 반드시 상위 클래스에서 정의된 함수를 구현해야 함
- 앞 예에서 “아빠” 인터페이스와 “교수인아빠” 클래스의 관계

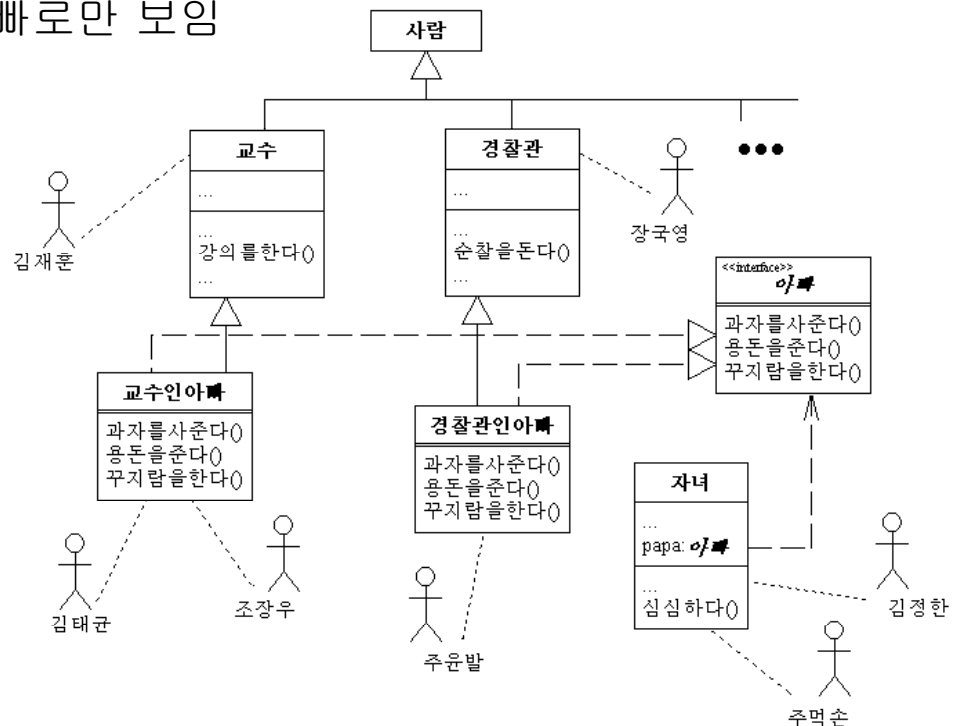
- 앞 예에서 “아빠” 인터페이스, “교수” 클래스, “교수인아빠” 클래스 간의 관계는 다중 상속은 아님



하나의 클래스에서 구현되어야 하는 인터페이스들이 많은 경우에 유용

- 인터페이스의 또 다른 용도

- 서로 다른 클래스에 속하는 객체들을 같은 부류의 객체로 인식하는 추상화 수단
- 자식의 관점에서 “김태균” 객체와 “주윤발” 객체는 아빠로만 보임



❖ 예) 코드 2.13: 아빠를 이용하는 자녀 클래스

```

1: class 자녀 extends 사람 {
2:     ...      // 여러 데이터 멤버에 대한 정의가 있을 수 있다.
3:     private 오락실 잘가는오락실; // 12 라인 코드에서 이용되는 객체
4:     private 아빠 papa;
5:     ...      // 다른 멤버 함수들의 구현이 있을 것이다.
6:     public void setFather(아빠 ptr) { // 아빠 레퍼런스를 설정하는 함수이다.
7:         papa = ptr;
8:     }
9:     public void 심심하다() {
10:         if (this.나이 > 10) {
11:             Money 얼마 = papa.용돈을준다(); // 아빠 인터페이스의 사용 예이다.
12:             잘가는오락실.오락한다(얼마); // 오락실 클래스에는 오락한다()는 함수가 있다.
13:         } else {
14:             과자 맛있는거 = papa.과자를사준다(); // 아빠 인터페이스의 사용 예이다.
15:             this.먹는다(맛있는거);
16:         }
17:     }
18: }

```

## 2.9.3 자바에서의 이용

- C++에서는 인터페이스를 몰라도 프로그래밍 가능하나, Java에서는 반드시 이용해야 함
  - Java가 C++보다 객체 지향 개념을 좀더 적극적으로 반영한 언어이기 때문

```
class ExampleComponent extends Component
{
    implements MouseMotionListener, KeyListener {
        ...
        /* MouseMotionListener 인터페이스를 위한 메소드들의 구현 */
        public void mouseDragged(MouseEvent e) {
            ...
        }
        public void mouseMoved(MouseEvent e)
        {
            ...
        }
        /* KeyListener 인터페이스를 위한 메소드들의 구현 */
        public void keyTyped(KeyEvent e) {
            ...
        }
        public void keyPressed(KeyEvent e) {
            ...
        }
        public void keyReleased(KeyEvent e) {
            ...
        }
    }
}
```

- 자바의 인터페이스의 특징 및 효용성
  - 아래의 3가지 경우로 나누어 설명

1. 윈도우 프로그래밍 시
2. MVC 컴포넌트 사용 시
3. 범용 자료구조 구현 시

## (가) 윈도우 객체(컴포넌트, 컨트롤, 위젯)에서 이벤트 처리를 위한 경우

### □ MFC 경우 (코드 2.14)

#### ■ 이벤트 처리기들을 등록하고 구현해 주어야 함

- 메시지 핸들러 혹은 가상 함수 이용
- 구현 상속 이용
- 메시지 핸들러 선언
- 메시지 연결을 위한 매크로 명시
- 메시지 핸들러 구현

### □ Java 경우 (코드 2.15)

#### ■ 인터페이스들을 구현해 주어야 함

- 인터페이스 상속 이용
- 인터페이스 선택
- 인터페이스 구현
- 인터페이스 구현 객체 등록

❖ 예) 코드 2.14: 다이얼로그에서 마우스버튼 누르면 “뽕” 소리는 C++ 프로그램

```
...
9: // CBeepDialog dialog
10: class CBeepDialog : public CDialog
11: {
...
27: protected:
28:     // Generated message map functions
29:     //{AFX_MSG(CBeepDialog)
30:     afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
31:     //}AFX_MSG
32:     DECLARE_MESSAGE_MAP()
33: };
...
64: BEGIN_MESSAGE_MAP(CBeepDialog, CDialog)
65:     //{AFX_MSG_MAP(CBeepDialog)
66:     ON_WM_LBUTTONDOWN()
67:     //}AFX_MSG_MAP
68: END_MESSAGE_MAP()
69: //////////////////////////////////////
70: // CBeepDialog message handlers
71: void CBeepDialog::OnLButtonDown(UINT nFlags, CPoint point)
72: {
73:     // TODO: Add your message handler code here and/or call default
74:     Beep(1000,1000);
75:     CDialog::OnLButtonDown(nFlags, point);
76: }
```

1. 메시지 핸들러 선언

2. 이벤트와 이벤트 처리기 연결

3. 메시지 핸들러 구현

## Java에서 이벤트 처리기를 구현하는 방법

- (1) 프레임 클래스가 이벤트 처리기도 함께 구현
- (2) 별도의 클래스로 이벤트 처리기를 작성
- (3) 내부 클래스로 이벤트 처리기를 작성

❖ 예) 코드 2.15: 다이얼로그에서 마우스버튼 누르면 “뽕” 소리나는 Java 프로그램 (1번 방법)

```
1: import java.awt.*;
2: import java.awt.event.*;
3: class BeepDialog extends Dialog implements MouseListener {
4:     BeepDialog(Frame frame) {
5:         super(frame);
6:         setSize(100,100);
7:         addMouseListener(this);
8:     }
9:     public void mousePressed(MouseEvent e) {
10:         getToolkit().beep();
11:     }
12:     public void mouseClicked(MouseEvent e) { }
13:     public void mouseReleased(MouseEvent e) { }
14:     public void mouseEntered(MouseEvent e) { }
15:     public void mouseExited(MouseEvent e) { }
16: }
```

인터페이스 구현 객체를  
컴포넌트에 등록

❖ 예) 다이얼로그에서 마우스버튼 누르면 “뽕” 소리는 **Java** 프로그램 (2번 방법)

```
import java.awt.*;
import java.awt.event.*;

class MouseListenerForBeepDialog implements MouseListener {
    Dialog owner;
    MouseListenerForBeepDialog(Dialog owner) {
        this.owner = owner;
    }
    public void mousePressed(MouseEvent e) {
        owner.getToolkit().beep();
    }
    public void mouseClicked(MouseEvent e) { }
    public void mouseReleased(MouseEvent e) { }
    public void mouseEntered(MouseEvent e) { }
    public void mouseExited(MouseEvent e) { }
}

class BeepDialog extends Dialog {
    BeepDialog(Frame frame) {
        super(frame);
        setSize(100,100);
        addMouseListener(new MouseListenerForBeepDialog(this));
    }
}
```

- 효율성은 떨어짐
- 멤버 변수들을 마음대로 접근 불가능
- 구현된 인터페이스를 다른 다이얼로그 객체에서 이용하고자 하는 경우에 유용

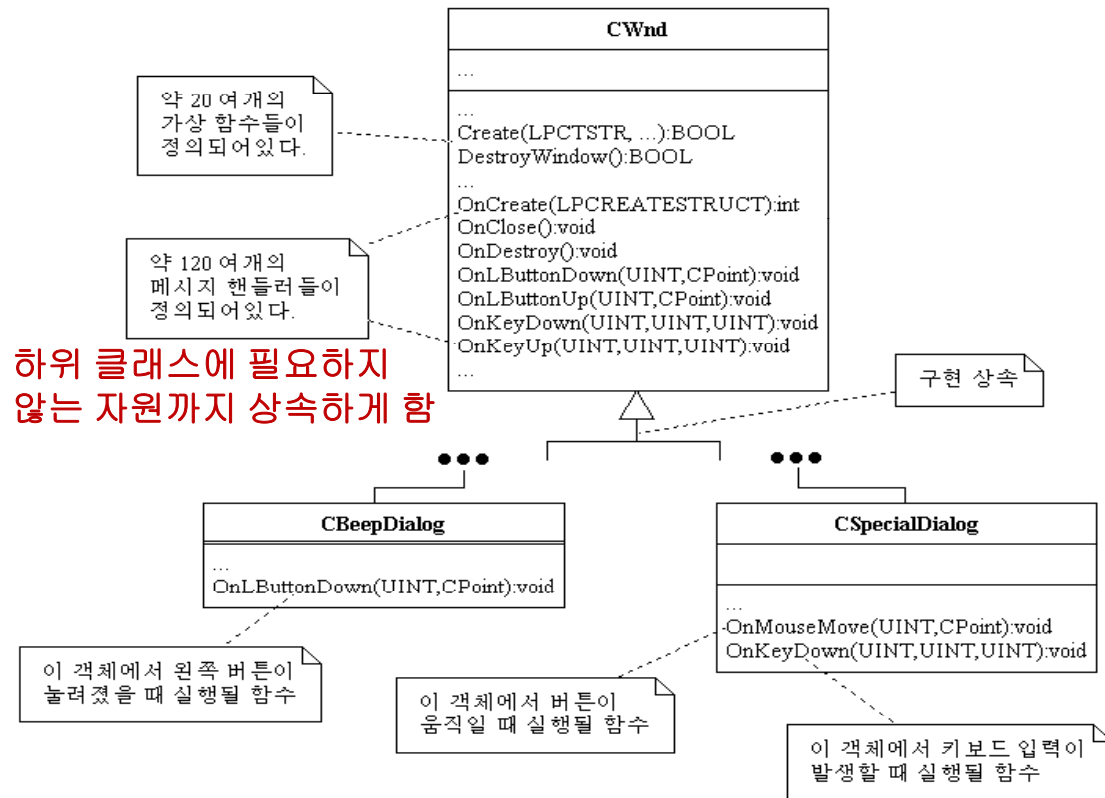
❖ 예) 다이얼로그에서 마우스버튼 누르면 “뽕” 소리는 **Java** 프로그램 (3번 방법)

```
import java.awt.*;
import java.awt.event.*;

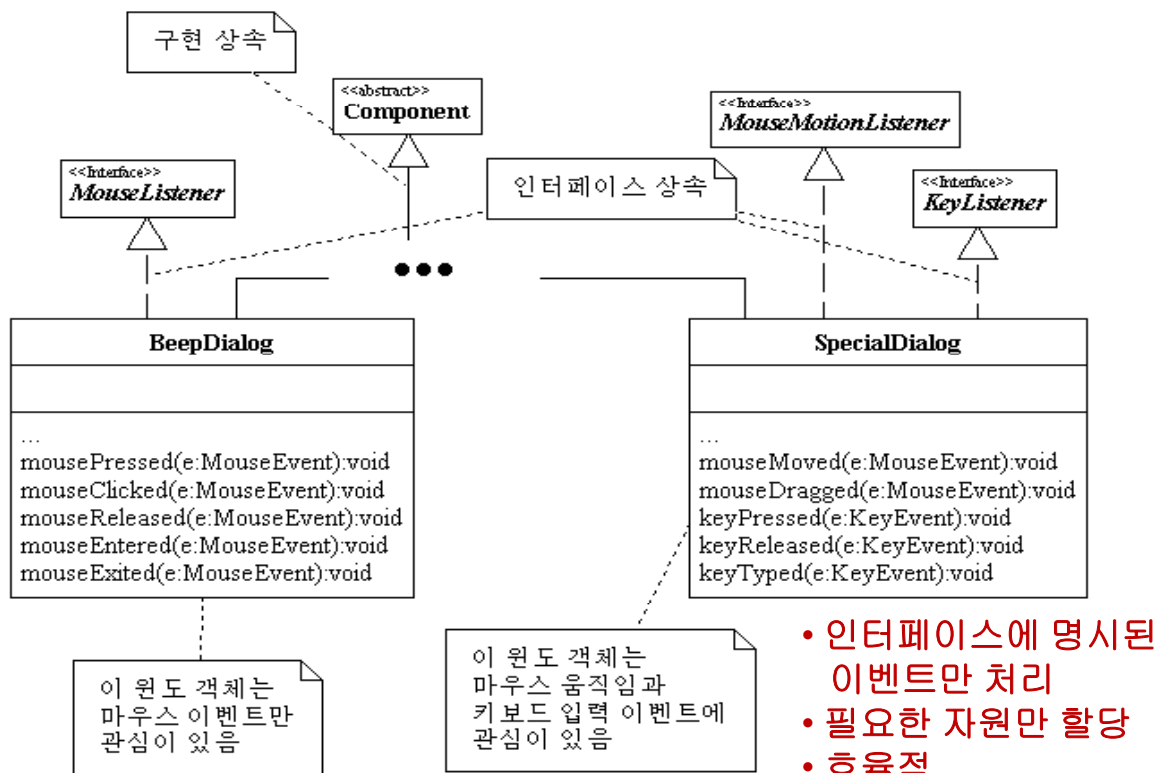
class BeepDialog extends Dialog {
    BeepDialog(Frame frame) {
        super(frame);
        setSize(100,100);
        addMouseListener(
            new MouseListener() {
                public void mousePressed(MouseEvent e) {
                    getToolkit().beep();
                }
                public void mouseClicked(MouseEvent e) { }
                public void mouseReleased(MouseEvent e) { }
                public void mouseEntered(MouseEvent e) { }
                public void mouseExited(MouseEvent e) { }
            }
        );
    }
}
```

- 효율적인 코드
- 재사용성, 이해력의 관점에서는 불리

## ❖ MFC (C++) 스타일



## ❖ Java 스타일





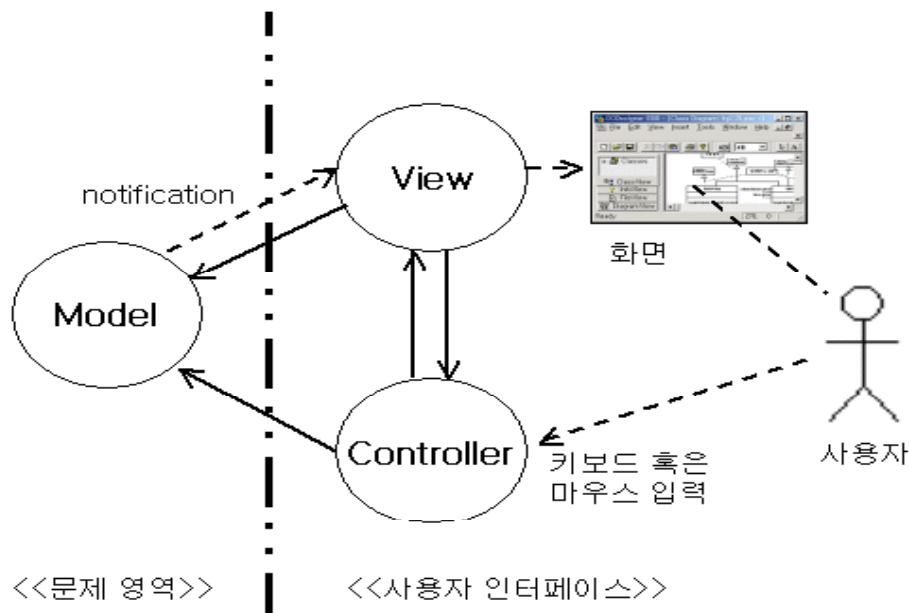
## (나) MVC 패러다임의 유용성 제공을 위한 경우

### □ MVC 모델

- 소프트웨어 설계 패턴의 한가지 방법
- GUI 포함하는 소프트웨어 설계 시 사용
- Smalltalk에서 유래
- Model, View, Controller 객체를 이용한 역할 분담
  - **Model** : 응용 영역의 객체(저장/로딩)
    - 응용 분야에서 다루고자 하는 데이터 자체를 모아놓은 것
  - **View** : 화면 출력 담당
    - 모델에 저장되어 있는 데이터를 화면을 통하여 출력
  - **Controller** : 이벤트 처리를 통한 모델의 상태 변화 담당
    - 키보드나 마우스 입력을 받아 모델과 뷰의 상태 변화를 수행

### □ MVC 모델

- MFC 경우에는 MDI, SDI application 차원에서 적용
  - 도큐먼트, 뷰, 프레임 클래스
- MVC 패러다임이 적용된 S/W 설계 구조



- MVC 구조의 장점
  - 시스템 구성 요소들 간 연결관계가 느슨해지기 때문에 각 요소간 독립성 유지하면서 구성 요소들을 병행적으로 개발 가능
- MFC
  - 비주얼 스튜디오에서는 전체 어플리케이션을 생성할 때 큰 덩어리로서 MVC 구조의 클래스들을 생성하지만 개별적인 컨트롤의 구현 시에는 MVC 패러다임이 이용되지 않는다.
- Java
  - UI 컴포넌트 중에서 데이터를 이용해야 하는 몇몇 Swing 컴포넌트 경우에 MVC 방식으로 운용되는 컴포넌트를 제공하고 있다.
  - (표 2.1)

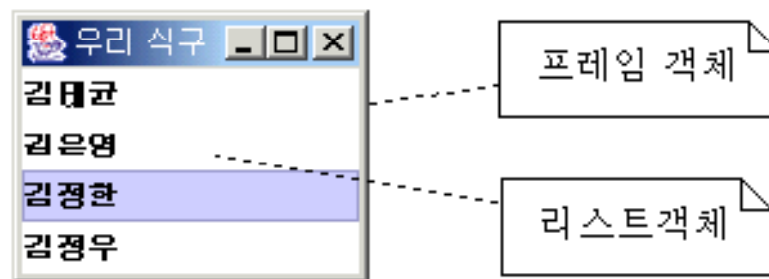
## ❖ 자바에서는 컨트롤 차원에서 적용 (표 2.1)

- 컴포넌트가 화면에 출력해야 하는 데이터들을 그에 해당되는 모델 객체로부터 얻음
  - 두 번째 열에 명시된 모델들은 Swing에서 정의된 인터페이스들로서 해당 컴포넌트가 사용할 수 있는 모델 명칭들
- 모델 객체를 액세스하기 위한 수단으로써 인터페이스 사용
  - 모델의 자료 구조를 구현하는 구현 기법의 다양성으로 인한 영향을 받지 않게 하기 위해 인터페이스를 사용

=> 자바에서 인터페이스는 모델 역할을 GUI 컴포넌트와 분리하여 구현하기 위해 사용됨

GUI 컴포넌트 이름	관련 모델 이름
JColorChooser	ColorSelectionModel
JComboBox	ComboBoxModel
JList	ListModel
JTable	TableModel
JTextArea	Document
JTextPane	StyledDocument
JTree	TreeModel

- JList가 ListModel 인터페이스를 이용하는 예
  - JFrame 객체 안에 하나의 JList 객체를 삽입한 것
  - **리스트 객체**에서는 스트링들을 나열
    1. 사용자 인터페이스로서의 리스트 (JList 객체) → View
    2. 데이터 자체로서의 리스트 ({김태균, 김은영, 김정한, 김정우} 배열을 의미) (ListModel 객체) → Model
      - 데이터들을 어떻게 조직하는가에 따라 다양하게 구현됨
      - **모델 인터페이스를 이용하여 구현과 명세를 분리**



(코드 2.16) TestFrame.java

```

1: import java.awt.*;
2: import java.util.*;
3: import javax.swing.*;
4: import javax.swing.event.*; // 순수하게 데이터만을 관리하는 클래스
5: class StringListModel implements ListModel {
6:     Vector strings;
7:     StringListModel() {
8:         strings = new Vector();
9:     }
10:    public Object getElementAt(int index) {
11:        return strings.elementAt(index);
12:    }
13:    public int getSize() {
14:        return strings.size();
15:    }
16:    public void addListDataListener(ListDataListener l) {
17:    }
18:    public void removeListDataListener(ListDataListener l) {
19:    }
20:    public void addString(String s) {
21:        strings.add(s);
22:    }
23: }
  
```

```

24: class FamilyList extends JList {
25:     FamilyList(ListModel model) {
26:         super(model);
27:         setSize(new Dimension(100,100));
28:     }
29: }
30: public class TestFrame extends JFrame {
31:     public static void main(String[] args) {
32:         TestFrame frame = new TestFrame();
33:         frame.setTitle("우리 식구");
34:         StringListModel familyNames = new StringListModel();
35:         familyNames.addString("김태균");
36:         familyNames.addString("김은영");
37:         familyNames.addString("김정한");
38:         familyNames.addString("김정우");
39:         frame.getContentPane().add(new FamilyList(familyNames););
40:         frame.pack();
41:         frame.setVisible(true);
42:     }
43: }

```

ListModel 객체와 JList 객체가 연관됨

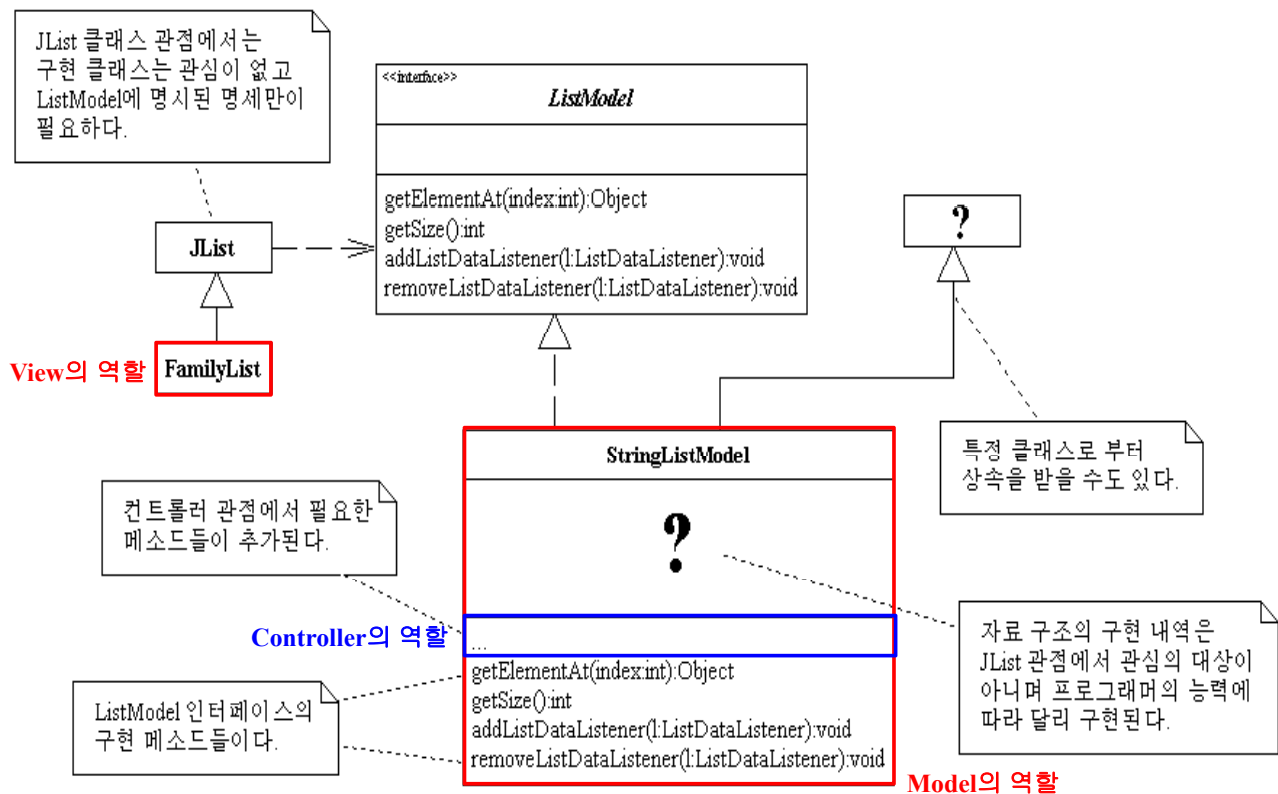
```

1: class StringListModel implements ListModel {
2:     String strings[];
3:     int lastPos;
4:     StringListModel() {
5:         strings = new String[100];
6:         lastPos = 0;
7:     }
8:     public Object getElementAt(int index) {
9:         return strings[index];
10:    }
11:    public int getSize() {
12:        return lastPos;
13:    }
14:    public void addListDataListener(ListDataListener l) {
15:    }
16:    public void removeListDataListener(ListDataListener l) {
17:    }
18:    public void addString(String s) {
19:        strings[lastPos] = s;
20:        lastPos++;
21:    }
22: }

```

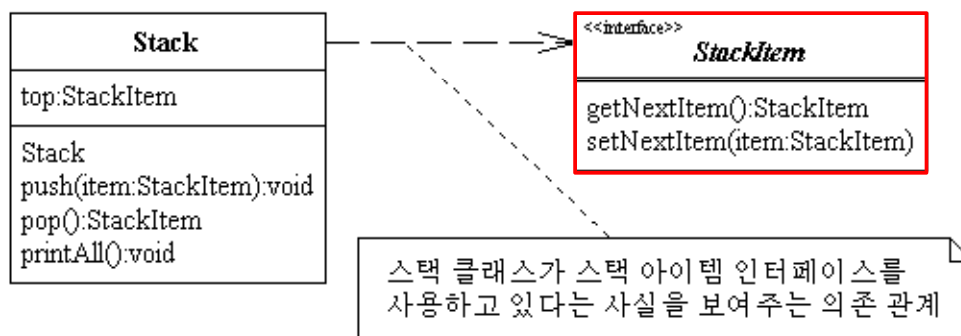
(코드 2.17) StringListModel 클래스의 다른 구현

## ❖ ListModel 인터페이스를 이용하는 StringListModel 클래스의 역할



## (다) 범용 자료구조 구현을 위한 경우

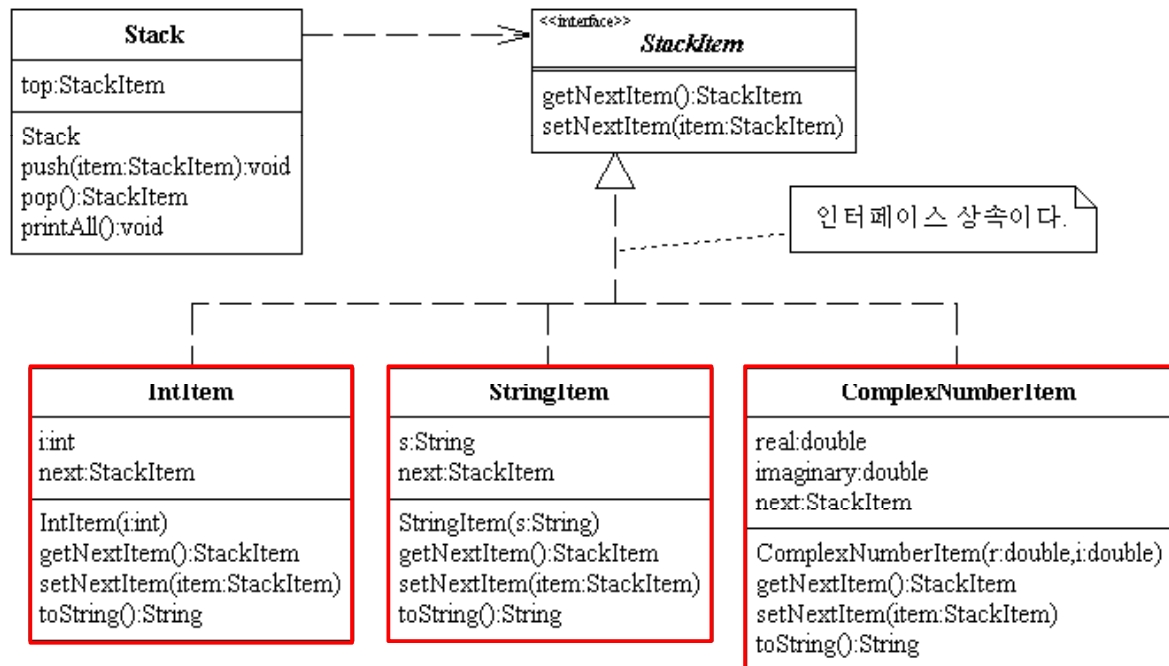
- 인터페이스는 좀더 유연성 있는 자료구조 클래스를 만들고자 할때 이용됨
- 예) 범용적인 스택을 만들 때



```
1: interface StackItem {
2:     public StackItem getNextItem();
3:     public void setNextItem(StackItem item);
4: }
5:
6: class Stack {
7:     StackItem top;
8:     public Stack() {
9:         top = null;
10:    }
11:    public void push(StackItem item) {
12:        if (top == null) top = item;
13:        else {
14:            item.setNextItem(top);
15:            top = item;
16:        }
17:    }
```

```
18:    public StackItem pop() {
19:        if (top == null) {
20:            System.out.println("stack is empty");
21:            java.lang.System.exit(-1);
22:        }
23:        StackItem topItem = top;
24:        top = top.getNextItem();
25:        return topItem;
26:    }
27:    public void printAll() {
28:        System.out.print("This stack has : ");
29:        StackItem item = top;
30:        while (item != null) {
31:            System.out.print(item);
32:            item = item.getNextItem();
33:        }
34:        System.out.println();
35:    }
36: }
```

## ❖ StackItem 인터페이스의 구현(1)



(코드 2.19) 스택 아이템들을 구현한 `Items.java` 파일

```

1: class IntItem implements StackItem {
2:     int i;
3:     StackItem next;
4:     public IntItem(int i) {
5:         this.i = i; next = null;
6:     }
7:     public StackItem getNextItem() {
8:         return next;
9:     }
10:    public void setNextItem(StackItem item) {
11:        next = item;
12:    }
13:    public String toString() {
14:        return i+"";
15:    }
16: }

```

```
- 17: class StringItem implements StackItem {  
- 18:     String s;  
- 19:     StackItem next;  
- 20:     public StringItem(String s) {  
- 21:         this.s = s; next = null;  
- 22:     }  
- 23:     public StackItem getNextItem() {  
- 24:         return next;  
- 25:     }  
- 26:     public void setNextItem(StackItem item) {  
- 27:         next = item;  
- 28:     }  
- 29:     public String toString() {  
- 30:         return s;  
- 31:     }  
- 32: }
```

```
- 33: class ComplexNumberItem implements StackItem {  
- 34:     double real;  
- 35:     double imaginary;  
- 36:     StackItem next;  
- 37:     public ComplexNumberItem(double r, double i) {  
- 38:         real = r; imaginary = i; next = null;  
- 39:     }  
- 40:     public StackItem getNextItem() {  
- 41:         return next;  
- 42:     }  
- 43:     public void setNextItem(StackItem item) {  
- 44:         next = item;  
- 45:     }  
- 46:     public String toString() {  
- 47:         return real+" "+imaginary+"i";  
- 48:     }  
- 49: }
```

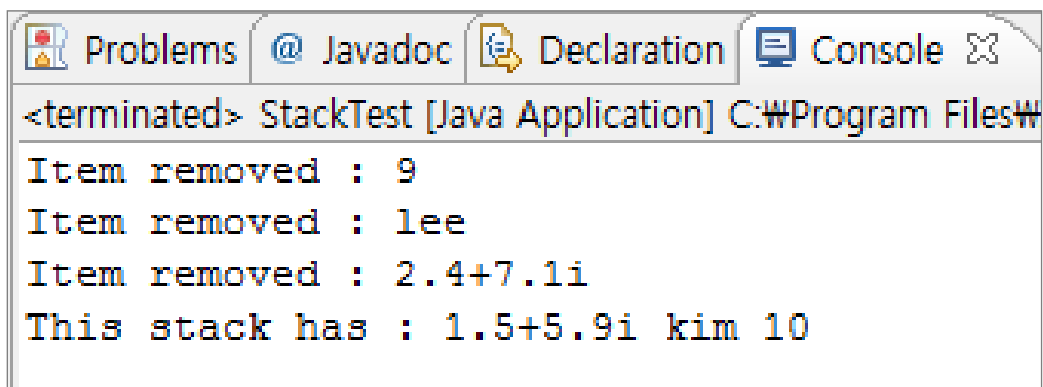


```

1: public class StackTest {
2:     public static void main(String[] args) {
3:         Stack aStack = new Stack();
4:         aStack.push(new IntItem(10));
5:         aStack.push(new StringItem("kim"));
6:         aStack.push(new ComplexNumberItem(1.5,5.9));
7:         aStack.push(new ComplexNumberItem(2.4,7.1));
8:         aStack.push(new StringItem("lee"));
9:         aStack.push(new IntItem(9));
10:        System.out.println("Item removed : "+aStack.pop());
11:        System.out.println("Item removed : "+aStack.pop());
12:        System.out.println("Item removed : "+aStack.pop());
13:        aStack.printAll();
14:    }
15: }

```

## • 실행결과



```

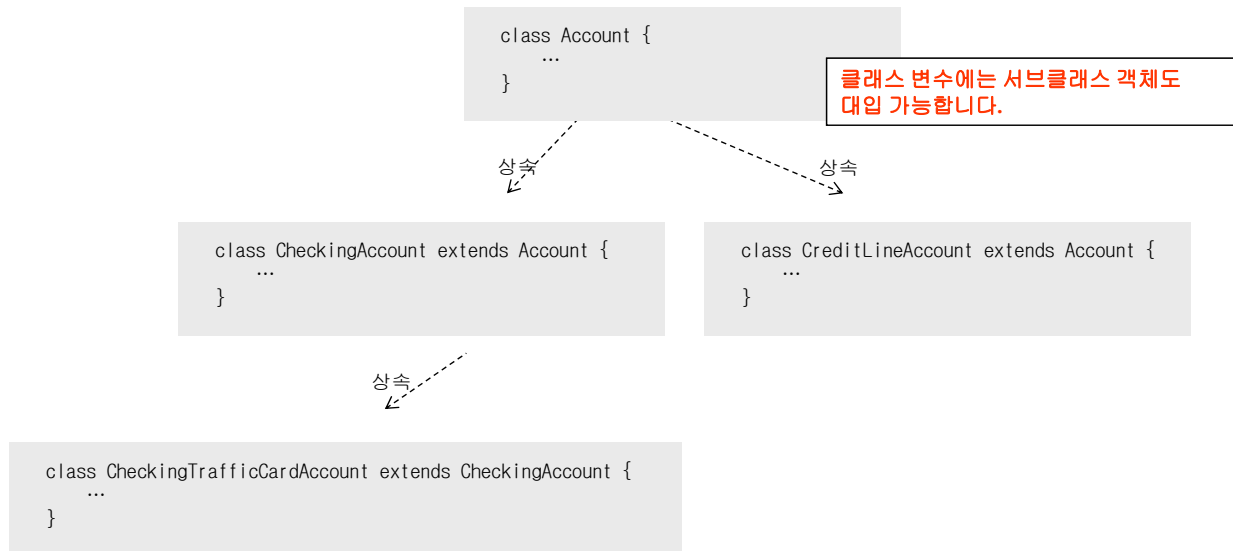
<terminated> StackTest [Java Application] C:\Program Files\#
Item removed : 9
Item removed : lee
Item removed : 2.4+7.1i
This stack has : 1.5+5.9i kim 10

```

# [참고자료]

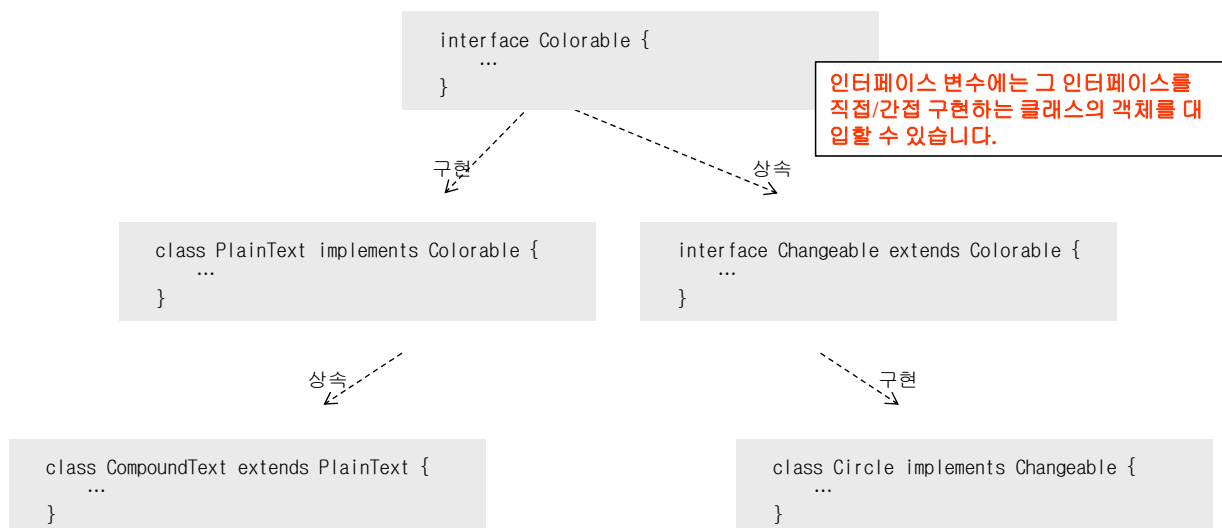
## • 변수의 타입과 객체의 타입

- 상속 관계를 갖는 클래스들



## • 변수의 타입과 객체의 타입

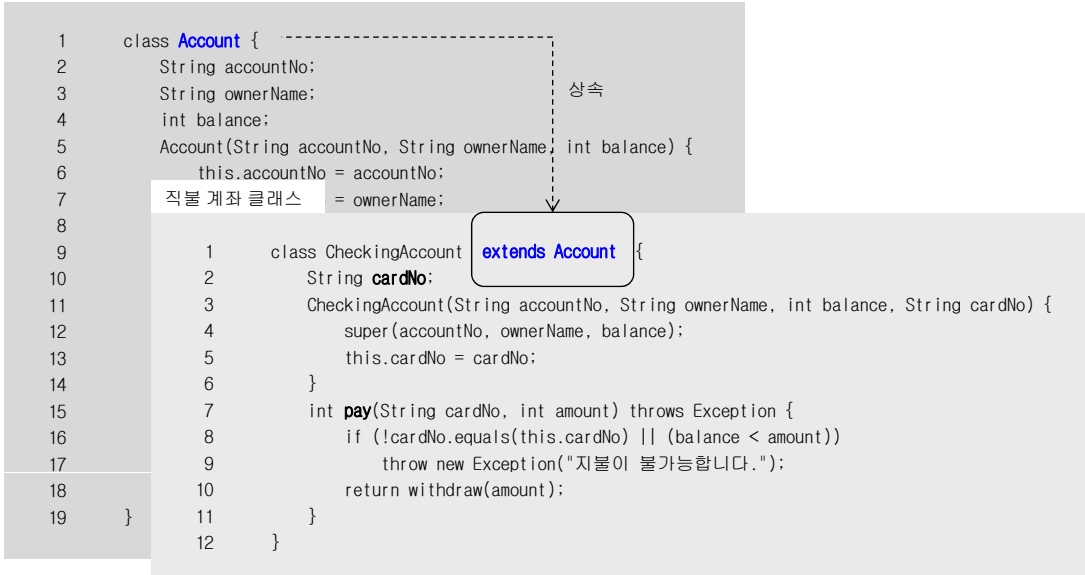
- 구현/상속 관계를 갖는 인터페이스와 클래스들



## • 변수에 의해 제한되는 객체의 사용 방법

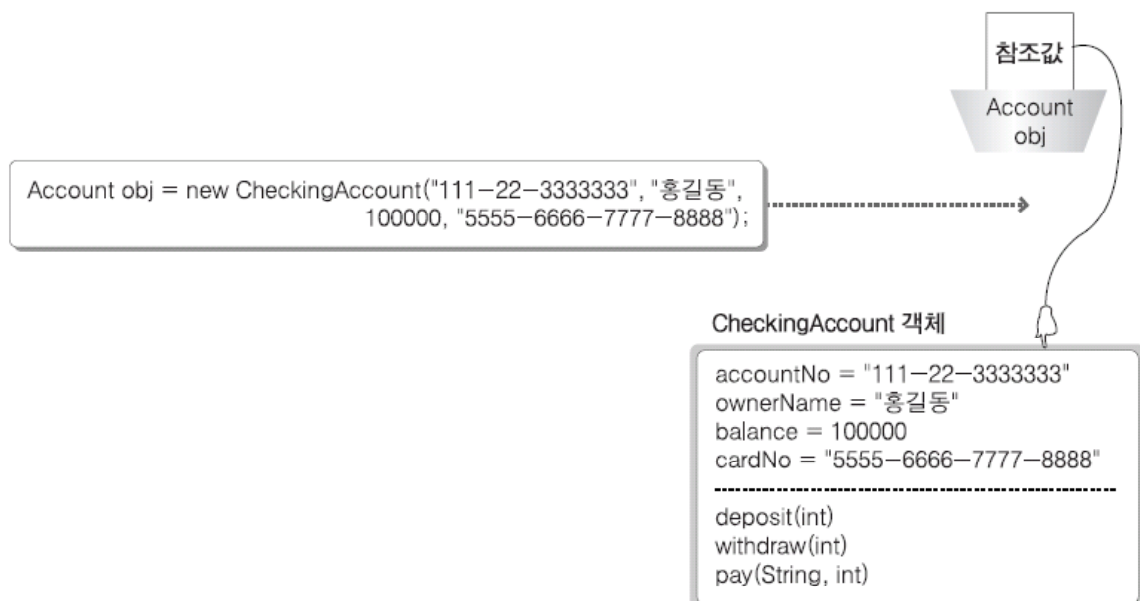
- Account 클래스와 CheckingAccount 클래스

은행 계좌 클래스



## • 변수에 의해 제한되는 객체의 사용 방법

- 다른 타입의 객체를 가리키는 클래스 변수

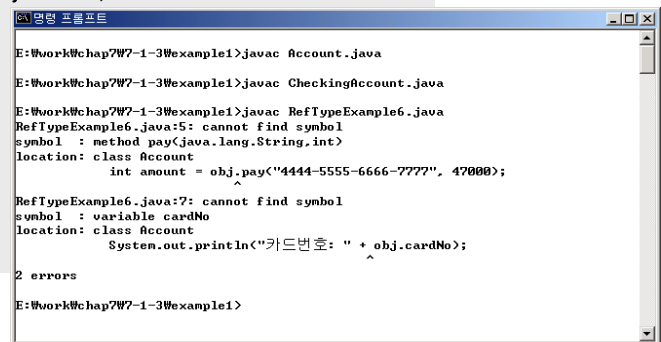


## • 변수에 의해 제한되는 객체의 사용 방법

- 변수 타입에 속하지 않는 필드와 메소드를 사용하는 잘못된 프로그램

```
1 class RefTypeExample6 {
2     public static void main(String args[]) {
3         Account obj = new CheckingAccount("111-22-33333333",
4             "홍길동", 10, "4444-5555-6666-7777");
5
6         try {
7             int amount = obj.pay("4444-5555-6666-7777", 47000);
8             System.out.println("인출액: " + amount);
9             System.out.println("카드번호: " + obj.cardNo);
10        }
11    }
12 }
13 }
```

Account 클래스에 없는  
메소드와 필드를  
사용하는 잘못된 명령문

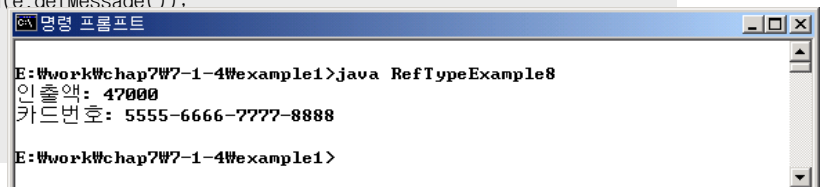


```
E:\work\chap7\7-1-3\example1>javac Account.java
E:\work\chap7\7-1-3\example1>javac CheckingAccount.java
E:\work\chap7\7-1-3\example1>javac RefTypeExample6.java
RefTypeExample6.java:5: cannot find symbol
symbol : method pay(java.lang.String,int)
location: class Account
    int amount = obj.pay("4444-5555-6666-7777", 47000);
                        ^
RefTypeExample6.java:7: cannot find symbol
symbol : variable cardNo
location: class Account
    System.out.println("카드번호: " + obj.cardNo);
                                   ^
2 errors
E:\work\chap7\7-1-3\example1>
```

## • 캐스트 연산자

- 슈퍼클래스 변수 값을 캐스트 해서 서브클래스 변수에 대입하는 프로그램

```
1 class RefTypeExample8 {
2     public static void main(String args[]) {
3         Account obj1 = new CheckingAccount("111-22-33333333",
4             "홍길동", 100000, "5555-6666-7777-8888");
5
6         CheckingAccount obj2 = (CheckingAccount) obj1;
7
8         try {
9             int amount = obj2.pay("5555-6666-7777-8888", 47000);
10            System.out.println("인출액: " + amount);
11            System.out.println("카드번호: " + obj2.cardNo);
12        }
13    }
14 }
```



```
E:\work\chap7\7-1-4\example1>java RefTypeExample8
인출액: 47000
카드번호: 5555-6666-7777-8888
E:\work\chap7\7-1-4\example1>
```

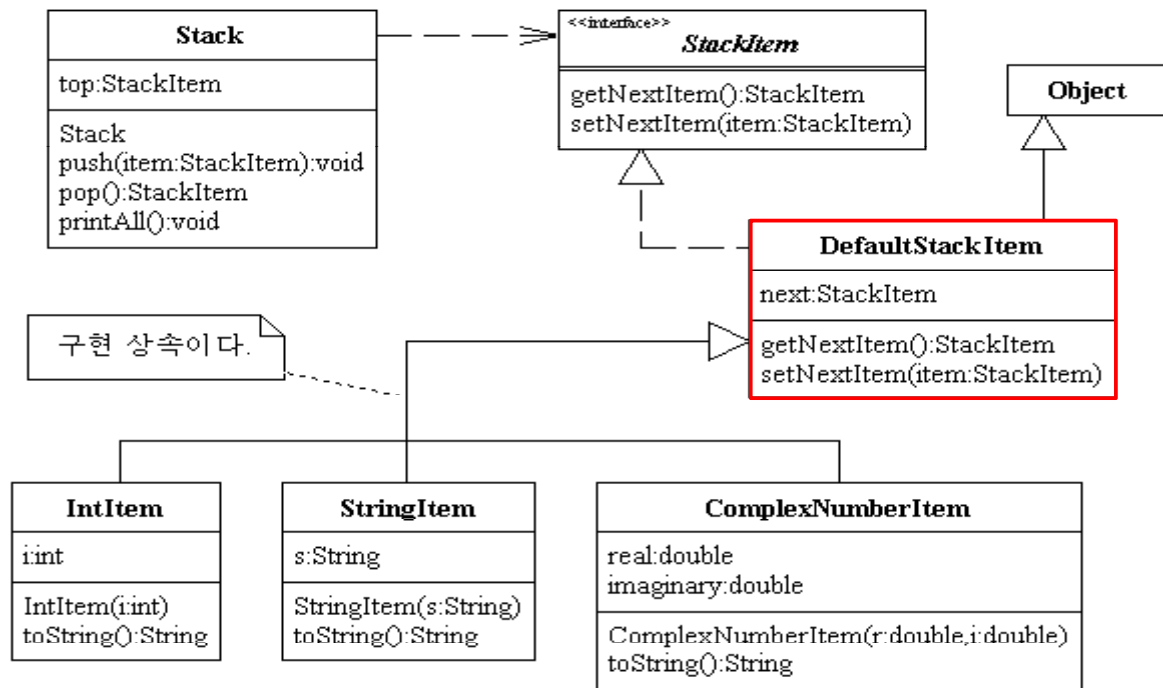
- 리팩토링 (refactoring)

- S/W 코드를 정리하여 추후 유지보수 작업에 대비하는 기법
- 대상
  - 특정 멤버의 명칭을 좋은 이름으로 바꾸는 경우
  - 특정 함수의 시그니처를 바꾸는 경우
  - 어떠한 클래스의 멤버를 상위 클래스나 혹은 다른 클래스로 옮기는 경우
  - 상속 트리의 같은 레벨에 있는 함수들의 메소드를 병합하여 부모 클래스 쪽으로 옮기는 경우
  - 같은 레벨에 있는 클래스들로부터 새로운 부모 클래스를 만드는 경우
- 리팩토링과 새로운 기능의 추가를 동시에 수행해서는 안 된다.
  - 기존 시스템을 리팩토링한 후에 새로운 기능을 추가하거나 혹은 새로운 기능을 추가 한 후에 리팩토링을 수행해야 한다.

- 리팩토링 (refactoring)

- 목적
  - 프로그램의 기능은 똑같으면서 유지보수가 쉬운 코드 작성하는 것
    - 코드 중복이 최소화되어야 함
- Principle of S/W entropy
  - 아무리 잘 설계된 소프트웨어라도 시스템 구현 이후에 점차로 잘 설계된 구조가 무너진다는 원리
- 아주 오래된 격언
  - “만약 소프트웨어가 제대로 작동하면 절대로 그 코드를 고치지 마라.”
- 프로그래머의 마음가짐
  - “빨리 원하는 기능을 구현하고 쉬자.” : X
  - “구현 후에 리팩토링하고 쉬자.” : O

## ❖ StackItem 인터페이스의 구현(2) – Refactoring



(코드 2.21) 코드 2.19를 리팩토링한 Items.java

```

1: class DefaultStackItem extends Object implements StackItem {
2:     StackItem next;
3:     public StackItem getNextItem() {
4:         return next;
5:     }
6:     public void setNextItem(StackItem item) {
7:         next = item;
8:     }
9: }
10: class IntItem extends DefaultStackItem {
11:     int i;
12:     public IntItem(int i) {
13:         this.i = i; next = null;
14:     }
15:     public String toString() {
16:         return i + "";
17:     }
18: }

```

```

19: class StringItem extends DefaultStackItem {
20:     String s;
21:     public StringItem(String s) {
22:         this.s = s; next = null;
23:     }
24:     public String toString() {
25:         return s;
26:     }
27: }
28: class ComplexNumberItem extends DefaultStackItem {
29:     double real;
30:     double imaginary;
31:     public ComplexNumberItem(double r, double i) {
32:         real = r; imaginary = i; next = null;
33:     }
34:     public String toString() {
35:         return real+" "+imaginary+"i";
36:     }
37: }

```

## 2.9.4 C++ 경우의 인터페이스

### □ C++ 에도 인터페이스가 있는가 ?

- C++에서는 구현 상속을 주로 사용함
- 인터페이스를 모방하여 코딩이 가능함

### □ 인터페이스를 모방하기 위해서는

- 순수 가상함수만을 포함하는 클래스와 다중 상속 이용

- 인터페이스는
  - C++에서 순수 가상함수만을 포함하는 클래스로 정의됨

Java

```
interface StackItem {
    public StackItem getNextItem();
    public void setNextItem(StackItem item);
}
```



C++

```
class StackItem {
public:
    virtual StackItem* getNextItem() = 0;
    virtual void setNextItem(StackItem* item) = 0;
};
```

- 인터페이스를 이용하는 클래스에서는 가상함수로 오버라이딩하여 구현

Java

```
class DefaultStackItem extends Object implements StackItem {
    StackItem next;
    public StackItem getNextItem() {
        return next;
    }
    public void setNextItem(StackItem item) {
        next = item;
    }
}
```



C++

```
class DefaultStackItem : public StackItem {
    StackItem *next;
public:
    virtual StackItem *getNextItem() {
        return next;
    }
    virtual void setNextItem(StackItem *item) {
        next = item;
    }
};
```



- 나머지 Java 코드를 변환한 C++ 코드는 아래 참조
  - 교재: p.127-130

### (코드 2.23) Stack 객체를 이용하는 StackTest.cpp

```
1: #include "StackTest.h"
2: void main(int argc, char* argv[]) {
3:     Stack* aStack = new Stack();
4:     aStack->push(new IntItem(10));
5:     aStack->push(new StringItem("kim"));
6:     aStack->push(new ComplexNumberItem(1.5, 5.9));
7:     aStack->push(new ComplexNumberItem(2.4, 7.1));
8:     aStack->push(new StringItem("lee"));
9:     aStack->push(new IntItem(9));
10:    printf("Item removed : %s\n", aStack->pop()->toString());
11:    printf("Item removed : %s\n", aStack->pop()->toString());
12:    printf("Item removed : %s\n", aStack->pop()->toString());
13:    aStack->printAll();
14: }
```

- 앞 예들을 살펴보면,
  - Java와 C++은 문법만 다를뿐 개별 클래스의 내용과 전체적인 구조는 똑같다.
- 인터페이스는
  - 사물을 보는 관점을 제공하는 추상화 도구
  - Java나 C++에 상관없이 보편성 있는 문제 해결 수단

## 2.9.5 컴포넌트 공학에서의 이용

- 컴포넌트 기술
  - COM, EJB, CORBA 등
- 컴포넌트 기술에서의 인터페이스
  - 클라이언트가 컴포넌트 객체의 서비스를 액세스할 수 있는 수단
  - 정보은닉의 수단으로 사용됨
- 상세내용은 생략
  - 관련도서 참고

## 2.10 시리얼라이제이션

- **Serialization** (직렬화)
  - 통신이나 저장을 목적으로 임의의 복잡한 자료구조를 장치 독립적인 방식으로 표현하는 기법
  - 포인터(레퍼런스)를 포함하는 임의의 자료구조를 저장하거나 로딩하기 위한 것
- 예) C 언어에서 리스트나 트리와 같은 자료구조 저장 시 어려움 발생
  - 리스트 내 데이터의 개수 저장 또는 끝을 나타내는 기호 추가 필요

- MVC의 Model 컴포넌트와 관련
  - 모델 관련 클래스들이 저장과 로딩의 대상
  - 모델 관련 클래스들에 직렬화 기능 구현
- MFC와 Java에서 직렬화를 지원함

## 1. Java의 경우

- **Serializable 인터페이스**를 구현해야 함
  - `writeObject()`, `readObject()` 함수가 정의되어 있음
  - 디폴트 함수가 호출되기 때문에 프로그래머가 반드시 구현하지 않아도 됨
- 예) 앞 예의 Stack, DefaultStackItem 클래스를 직렬화
  - IntItem, StringItem, ComplexItem 클래스도 직렬화 가능하게 됨

## ❖ Java에서 직렬화 사용하는 예제

```

0: import java.io.*;           // Serializable 인터페이스는 java.io 패키지에 정의되어 있음
1: interface StackItem {
2:     public StackItem getNextItem();
3:     public void setNextItem(StackItem item);
4: }
6: class Stack implements Serializable {
7:     StackItem top;
8:     ...
36: }

/* (코드 2.21) Items.java 파일의 변경 사항 */
0: import java.io.*;
1: class DefaultStackItem extends Object implements StackItem , Serializable {
2:     StackItem next;
3:     ...
37: }

```

(코드 2.26) Stack 객체 저장

```

public class StackTest {
    public static void main(String[] args) {
        Stack aStack = new Stack();
        aStack.push(new IntItem(10));
        aStack.push(new StringItem("kim"));
        aStack.push(new ComplexNumberItem(1.5, 5.9));
        aStack.push(new ComplexNumberItem(2.4, 7.1));
        aStack.push(new StringItem("lee"));
        aStack.push(new IntItem(9));
        System.out.println("Item removed : "+aStack.pop());
        System.out.println("Item removed : "+aStack.pop());
        System.out.println("Item removed : "+aStack.pop());
        aStack.printAll();

        try {
            FileOutputStream fos = new FileOutputStream("stack.dat");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(aStack);
            oos.close();
            fos.close();
        }
        catch (IOException ex) {
            System.out.println("Saving fail..");
        }
    }
}

```

(코드 2.27) Stack 객체 로딩

```

public class StackTestToLoad {
    public static void main(String[] args) {
        Stack aStack = null;

        try {
            FileInputStream fis = new FileInputStream("stack.dat");
            ObjectInputStream ois = new ObjectInputStream(fis);
            aStack = (Stack) ois.readObject();
            ois.close();
            fis.close();
        }
        catch (Exception ex) {
            System.out.println("Loading fail..");
        }

        aStack.printAll();
    }
}

```

## 2. MFC의 경우

- 자바 보다 약간 복잡
  - MFC의 모든 클래스는 CObject 클래스로부터 파생됨
  - CObject 클래스에 정의된 Serialize() 함수가 직렬화시키기 위한 목적으로 사용됨
  - 직렬화시키고 싶은 클래스에서 Serialize()를 오버라이드해야 함
- 직렬화는
    - 응용 프로그램 구현 시 데이터의 저장과 로딩에 관한 코딩 부담을 상당량 경감시켜줌

### ❖ MFC에서 직렬화 사용하는 예제

```
1: // SomeClass.h 파일
2: ...
3: class SomeClass : public AnySuperClass {
4:     DECLARE_SERIAL(SomeClass)
10:     ...
11:     virtual void Serialize(CArchive& ar);
12: };

1: // SomeClass.cpp 파일
2: ... // 해당 클래스가 직렬화 가능하다는 사실을 매크로로 명시
3: IMPLEMENT_SERIAL(SomeClass, AnySuperClass, 1)
4: ... // 해당 클래스에 맞게 Serialize() 함수를 오버라이드하여 구현
5: void SomeClass::Serialize(CArchive& ar) {
7:     if (ar.IsStoring()) {
13:         ...
19:     } else {
20:         ...

```

## 2.11 객체지향 프로세스의 특성

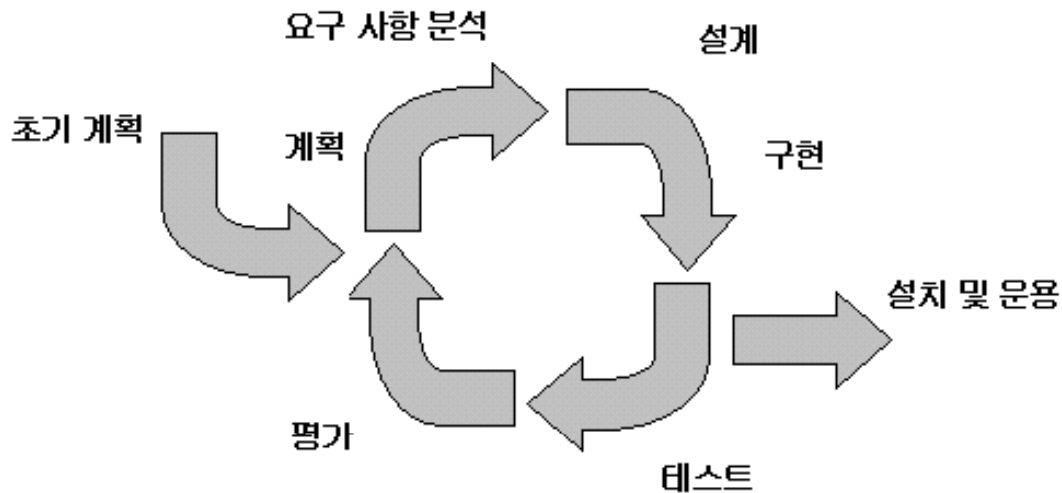
- 소프트웨어 공학의 목표
  - 성공적인 소프트웨어 제품 생산
    - 기능적으로 고객의 요구사항을 충족시키면서 성능이 뛰어난 제품을 만들자
  - 성공적인 프로세스 달성
    - 적절한 기간과 비용으로 만들자
- 객체 지향 S/W 개발 프로세스의 특징
  1. 반복적인(iterative) 프로세스
  2. 슬기없는(seamless) 프로세스
  3. 상향식(bottom up) 접근 방법
  4. 재사용(reuse) 고려

### 2.11.1 반복적인 프로세스

- 구조적 기법이 채택한 폭포수 모형
  - 요구사항 → 설계 → 구현 → 테스트 → 유지보수
  - 한 단계의 작업이 완전히 끝난 이후에 다음 단계로 진행 가능
  - 현 단계에서 오류가 발생하는 경우에는 역방향 진행도 가능해야
  - 개발 프로세스 초기에 시스템의 요구사항이 완벽하게 명시되어야 한다는 문제점 존재
    - 현실적으로 불가능

- 반복적인 객체 지향 프로세스

- 점진적인 기능 추가와 반복적인 공정을 전제
- 나선형 모델 (spiral model)
  - 우선 순위 높은 것 먼저 구현
  - 고객의 피드백을 받고 점차 우선 순위가 낮은 기능 구현



- 반복적인 프로세스의 장점

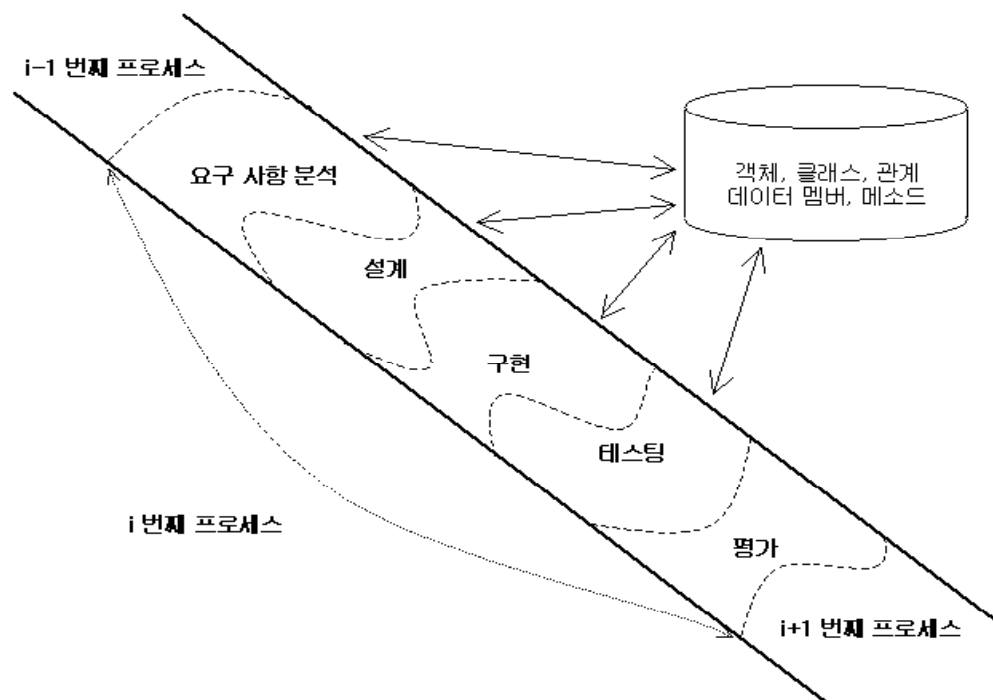
- 요구사항에 대한 잘못된 이해를 조기에 발견하여 수정할 수 있다.
- 고객과의 의사소통을 수월하게 하므로 시스템의 요구사항을 정확히 끌어낼 수 있다.
- 주어진 핵심 임무에만 전념할 수 있다.
- 결과물에 대한 객관적인 평가가 가능하다.
- 요구 사항, 설계 문서, 구현 코드간의 불일치를 조기에 파악할 수 있다.
- 테스트와 관련한 업무 부담이 팀 전체에 골고루 분산된다.
- 시행착오를 다음 공정에서의 교훈으로 활용할 수 있다.

## 2.11.2 솔기없는 공정

- 솔기없는 프로세스
  - 프로세스를 구성하는 각 단계간의 경계선이 불분명
  - 모든 단계에서 클래스 다이어그램을 표기법으로 사용
  - 분석 단계부터 작성되는 클래스 다이어그램이 시간이 지날수록 상세화, 구체화됨
- 구조적 기법의 경우
  - 각 단계간의 구분이 명확
  - 각 단계에서 사용하는 표기법과 작업 내용이 확연히 구별됨
  - 특정 단계의 결과물을 다음 단계의 문서로 변경하는 과정에서 오류가 발생할 여지 크다.

### ❖ 솔기없는 객체지향 프로세스

- 각 단계의 구분선이 점선 : 구분이 모호하다는 의미
- 각 단계에서의 고려대상은 항상 동일





- 솔기없는 프로세스가 주는 장점
  - 각 단계간의 정보 변환을 위한 부담이 적기 때문에 모델 변환 시에 발생하는 오류 가능성이 줄어든다.
  - 각 단계에서 사용하는 표기법에 일관성이 있으므로 문서의 양이 감소한다.
  - 프로그램으로부터 설계 문서를 추출하는 역공학이나 혹은 시스템을 다시 구축하는 재공학(restructuring)의 적용이 용이하다. 결과물에 대한 객관적인 평가가 가능하다.
  - 일관된 마음가짐으로 작업하므로 반복적인 프로세스의 적용이 가능하다.

## 2.11.3 상향식 프로세스

- 구조적 기법
  - 하향식(top-down) 프로세스
    - 어떤 문제를 큰 덩어리로 인식한 후, 작은 덩어리로 문제를 나누어 해결하는 방식
    - 잘 알려진, 많은 경험적 지식이 축적된 문제를 해결하는 기법
- 객체지향 기법
  - 상향식(bottom-up) 프로세스
    - 작은 덩어리 문제들을 해결한 후, 큰 문제를 해결하는 방식
    - 문제 영역에 대한 경험이 부족하여 잘 모르는 문제를 해결하는 기법
      - 예) 대동여지도 작성 방법
      - 예) 처음부터 클래스 상속 트리를 완벽하게 설계할 수는 없다.



- 상향식 프로세스의 장점

- 문제를 나누어서 해결할 수 있는 분할 정복(divide and conquer)이 가능하다.
- 부 프로젝트(sub-project)의 독립적인 동시 수행이 가능하다.
- 전체 시스템의 구현 이전에 일부 시스템의 구현이 이루어지므로 중간 단계에서의 성취감을 얻을 수 있다.
- 하드웨어 조립 방식과 유사하기 때문에 인적 물적 자원 배분을 위한 프로세스 관리가 체계적으로 이루어질 수 있다.
- 개별 클래스에 대한 단위 테스트를 심도 있게 시행할 수 있다.

## 2.11.4 재사용

- 재사용과 관련된 두 가지 시점

1. 시스템 설계가 끝난 후,

- 남이 작성한 코드는 그 동안 운용과 검증을 통해 신뢰성이 확보되어 있는 상태이므로 유용
- 재사용 후보가 되는 클래스 이해 후, 자기 요구사항에 맞게 수정해서 사용
  - 클래스 복사 후 수정
  - 또는, 상속받고 오버라이딩하기

2. 구현이 끝난 후,

- 현재 구현 컴포넌트를 차후에 재사용하기 위한 리팩토링

## ❖ 기존 클래스의 재사용

- 재사용 컴포넌트의 탐색
- 재사용 컴포넌트의 이해
- 재사용 컴포넌트의 수정
- 재사용 컴포넌트의 배치

## ❖ 리팩토링

- 클래스 자원의 수집
- 클래스 자원의 평가
- 클래스 자원의 수정
- 클래스 자원의 발표 및 분배

## 2.12 객체 지향의 기타 속성

- Uniformity
  - 객체 지향 시스템을 구성하는 객체들은 평등하게 다루어져야 한다.
- 객체들은 객체들로 구성된다.
  - UML의 집합화를 이용한 모델링의 시발점
- 모든 객체는 유일한 식별자를 갖는다.
  - 객체가 가지고 있는 속성의 조합은 객체마다 고유하다.
- 객체들은 시간이 지남에 따라 계속 변한다.
- 객체들은 생명주기를 갖는다