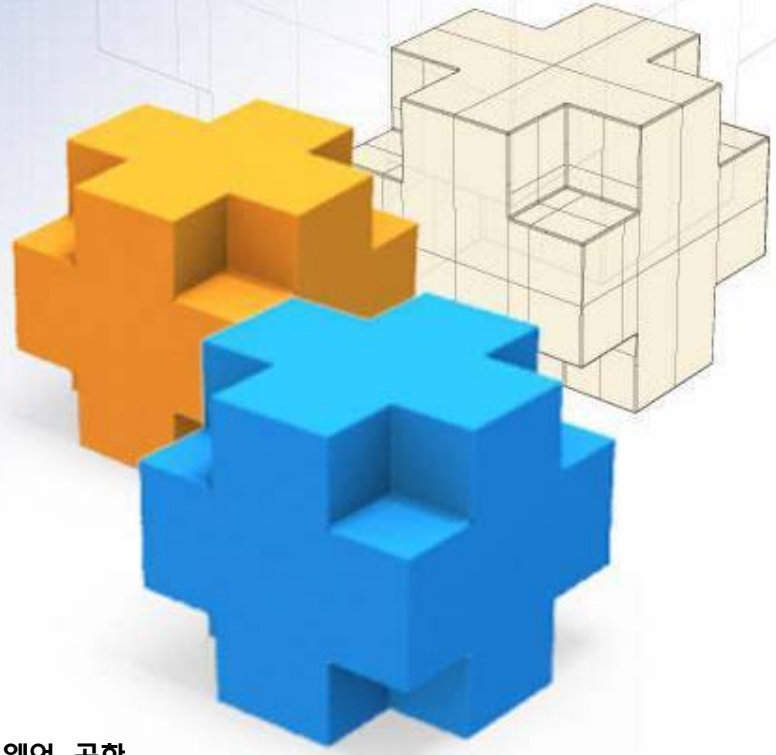
LECTURE

13

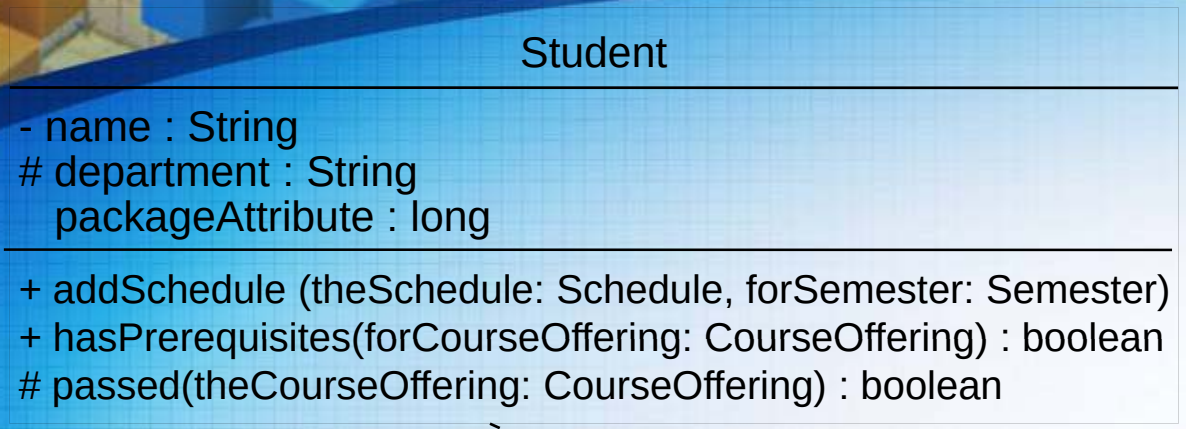
설계와 코딩



설계 구현 매핑

- UML 설계도로부터 Java 프로그래밍 언어로의 매핑 과정 설명
- 정적 다이어그램의 구현
- 동적 다이어그램의 구현

속성과 오퍼레이션의 구현

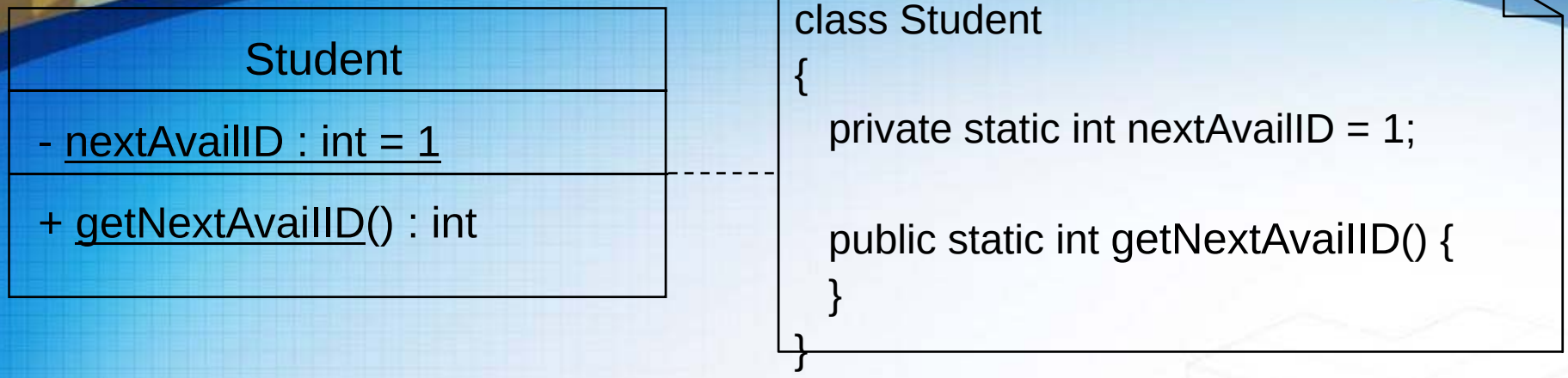


- +는 public, -는 private, #은 protected
- Public boolean은 true, false 리턴
- 메소드 바디는 빈칸

```
public class Student
{
private String name;
protected String department;
long packageAttribute;
public void addSchedule (Schedule theSchedule; Semester forSemester) {
        }

public boolean
        hasPrerequisites(CourseOffering forCourseOffering) {
        }
protected boolean
        passed(CourseOffering theCourseOffering) {
        }
```

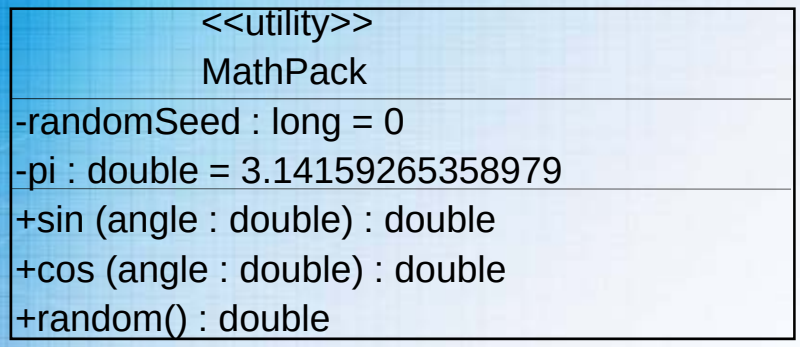
클래스 변수와 오퍼레이션



- Static 으로 선언되어야 할 클래스 변수와 클래스 오퍼레이션은 클래스에 속하는 어떤 다른 객체에서도 접근 가능
- 인스턴스 변수는 클래스의 객체가 생성될 때 객체 마다 생성되나
- Static으로 선언한 클래스 변수와 오퍼레이션은 클래스 수준의 정의

유틸리티 클래스

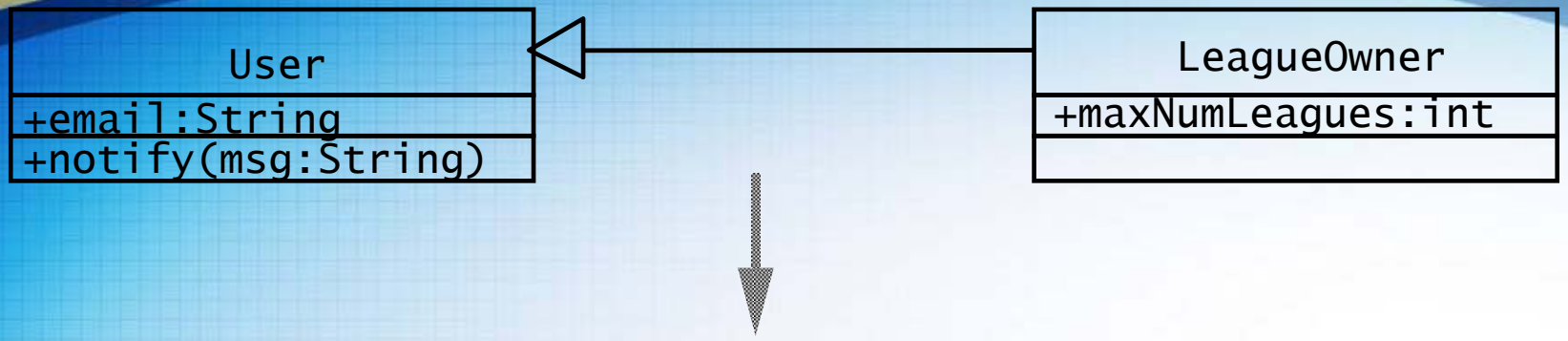
- 전역 변수와 오퍼레이션의 그루핑 – 인스턴스로 만들지 않아야 함. 따라서 Static
- 클래스 안의 여러 메소드를 외부에서 호출 할 것임(따라서 static)
- Static 선언은 메소드가 객체이름이 아니라 클래스 이름으로 호출 됨




```
void somefunction() {
. . .
      myCos = MathPack.cos(90.0);
. . .
```

```
import java.lang.Math;
import java.util.Random;
class MathPack
{
   private static randomSeed long = 0;
   private final static double pi =
         3.14159265358979;
    public static double sin(double angle) {
         return Math.sin(angle);
    }
    static double cos(double angle) {
         return Math.cos(angle);
    }
    static double random() {
              return new
                   Random(seed).nextDouble();
    }
```

상속

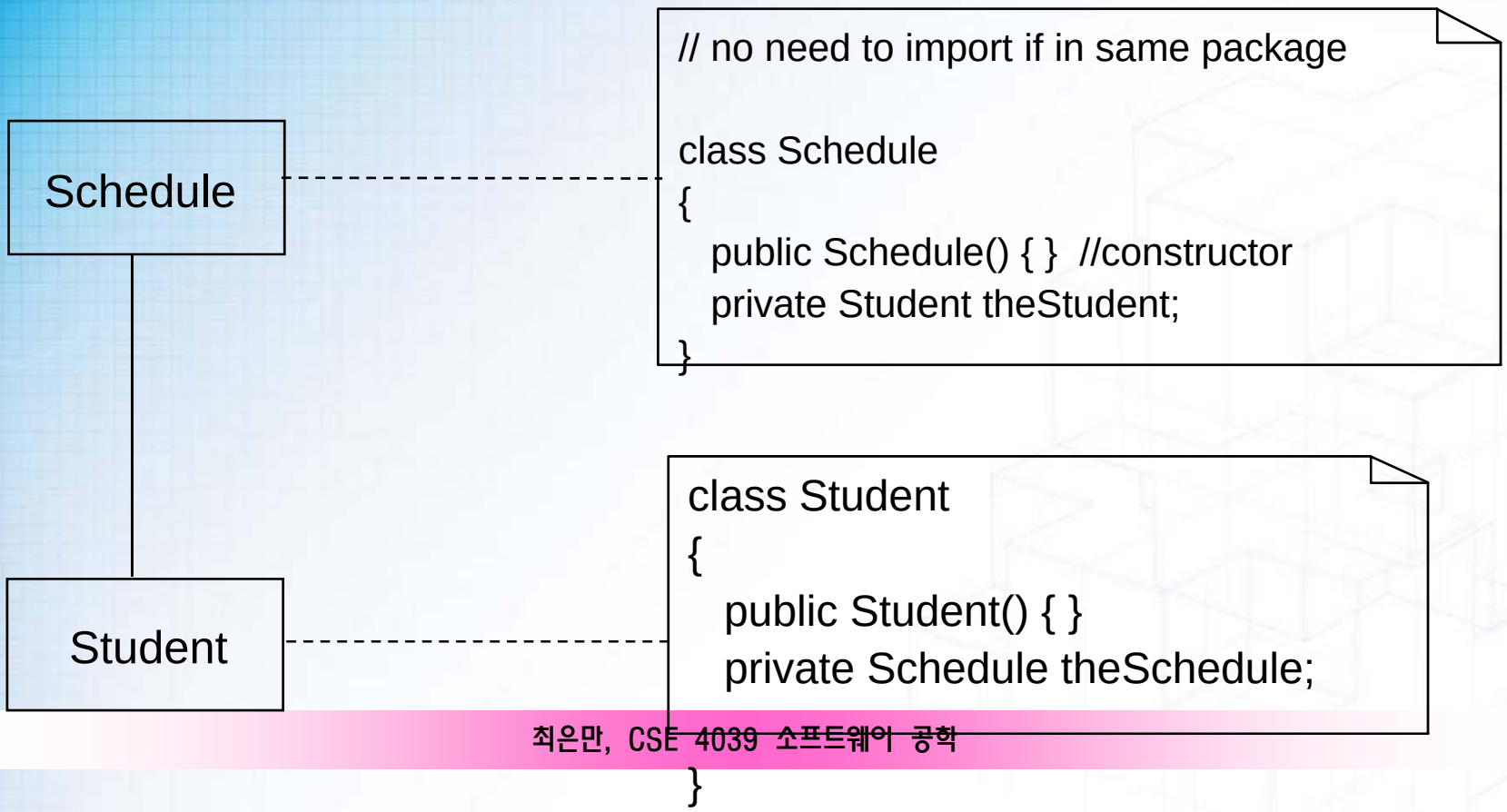



```
public class User {
   private String email;
   public String getEmail() {
         return email;
   }
   public void setEmail(String value){
         email = value;
   }
   public void notify(String msg) {
         // ....
   }
   /* Other methods omitted */
}
```

```
public class LeagueOwner extends
   User {
   private int maxNumLeagues;
   public int getMaxNumLeagues() {
         return maxNumLeagues;
   }
   public void setMaxNumLeagues
                  (int value) {
         maxNumLeagues = value;
   }
   /* Other methods omitted */
}
```

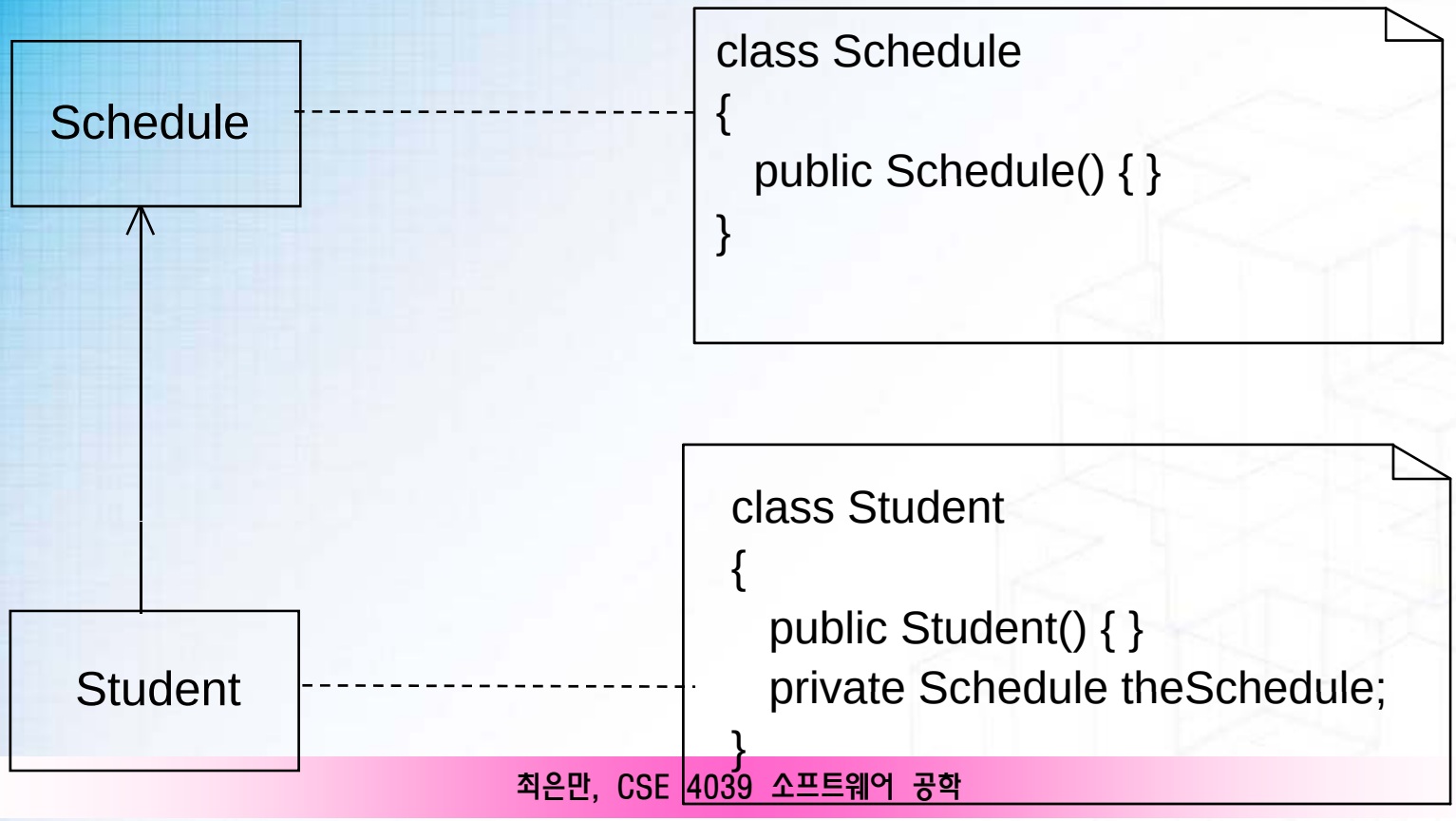
연관 관계

- 양방향 연관 – 두 클래스가 같은 패키지 안에 있어야
- 양방향은 두 클래스가 서로 상대편 객체를 알고 있어야 하므로 서로를 알기 위한 public 메소드와 private 변수가 있어야

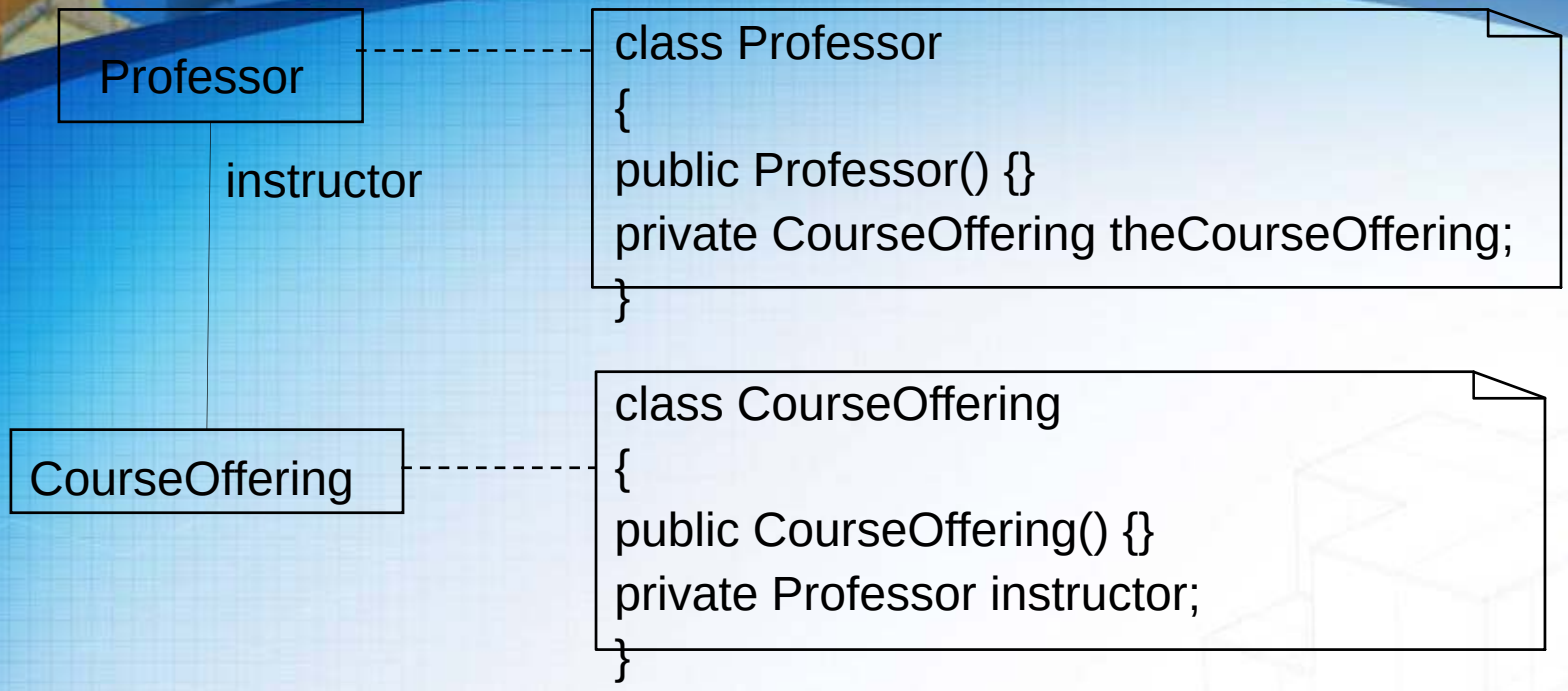


방향성 연관

- 단방향 연관
  - Student 객체가 Schedule 객체를 알아야. Student 객체는 Schedule 객체의 클라이언트
  - 다른 클래스의 객체를 만들 수 있어야

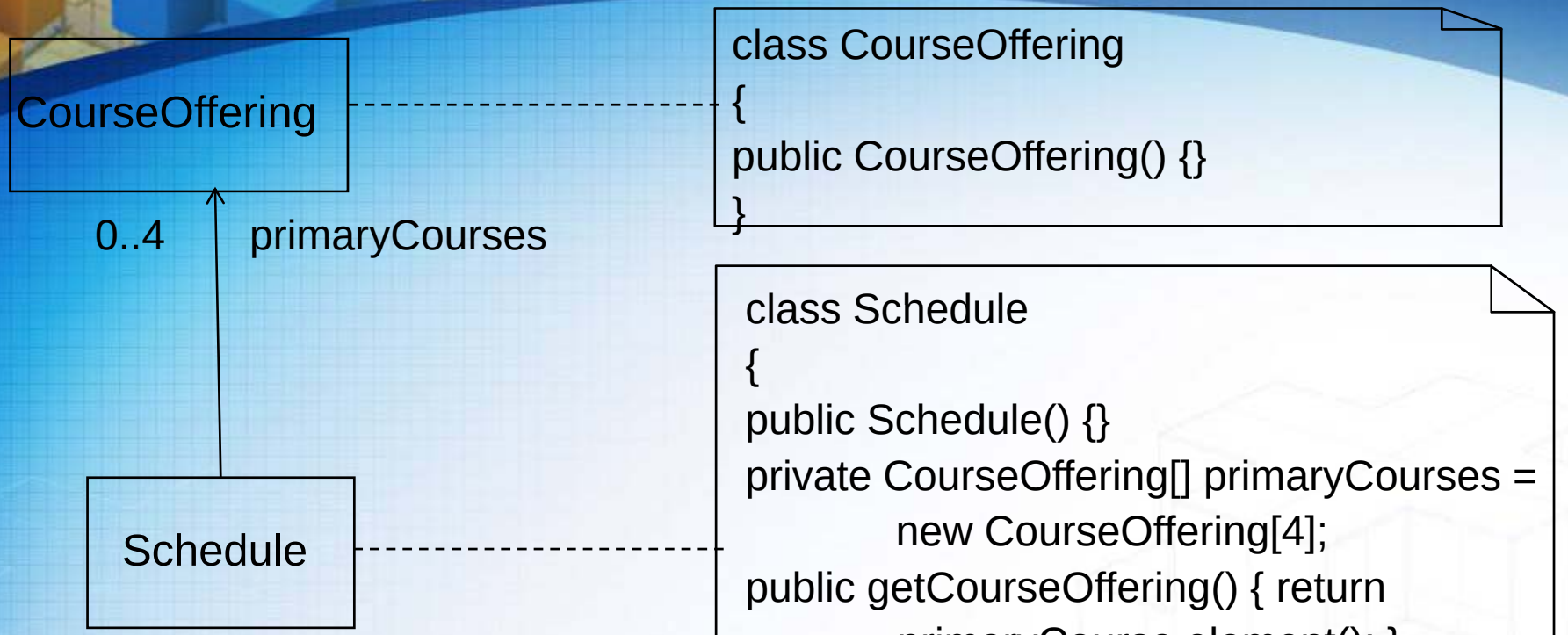


연관 Role



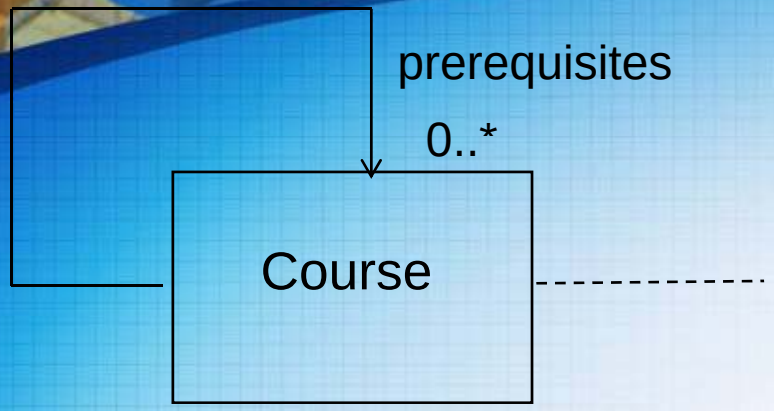
- Role 의 추가 – 각 클래스가 다른 클래스에 대하여 알고 있어야
- CourseOffering 클래스는 public 메소드 CourseOffering()가 있어야 하며 private 타입의 Professor 객체를 가지고 있어야
- Professor 클래스는 public 메소드 Professor()와 private 타입의 CourseOffering 객체를 알고 있어야 함

연관 다중도



- 다중도의 구현
- CourseOffering 클래스는 public method CourseOffering()을 가짐
- 클래스 Schedule에는 네 개의 CourseOffering 객체를 가질 수 있는 배열 또는 리스트가 있어야 함

재귀 연관

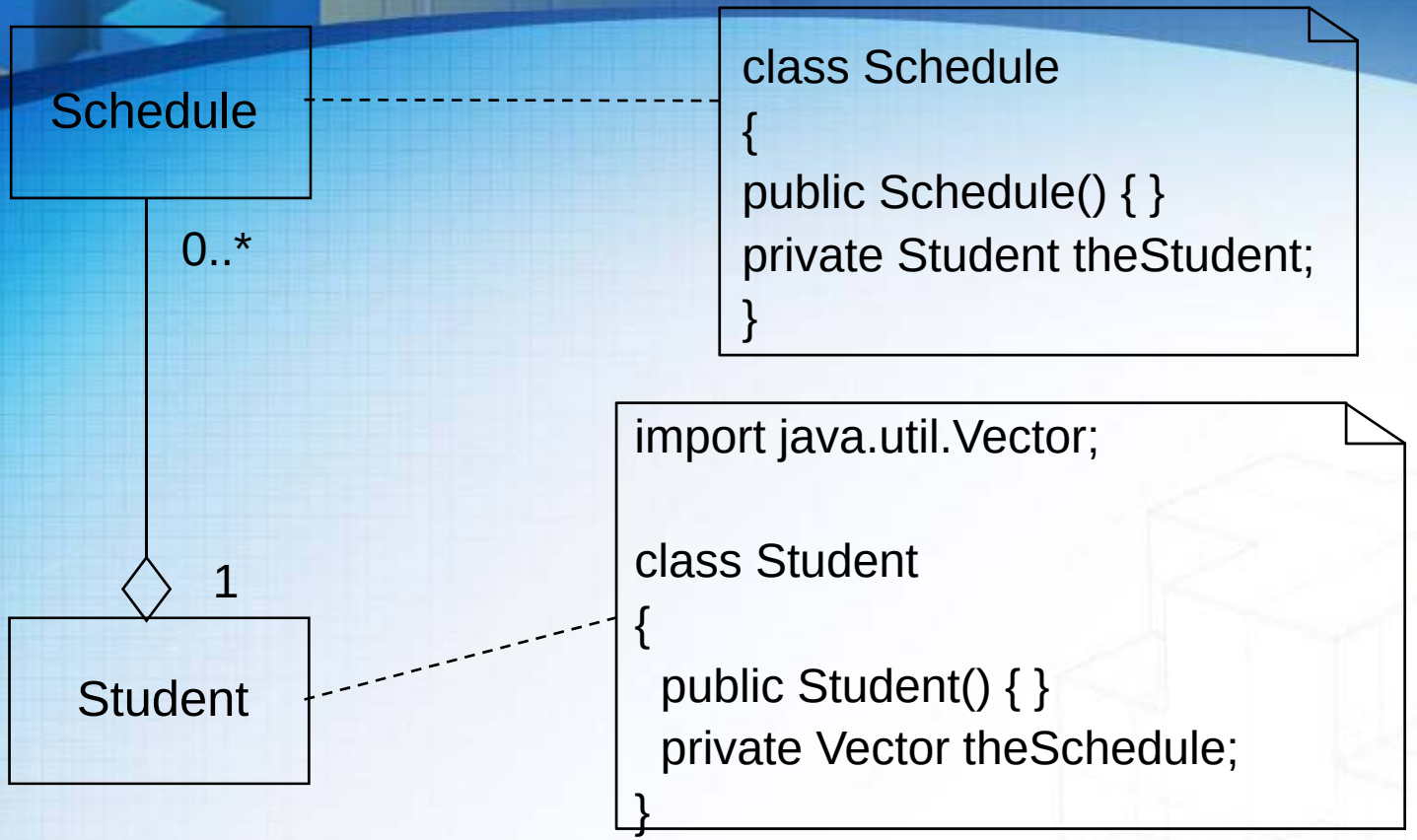



```
import java.util.Vector;

class Course
{
  public Course() {}
// The unbounded multiple association
//   is stored in a vector
  private Vector prerequisites;
}
```

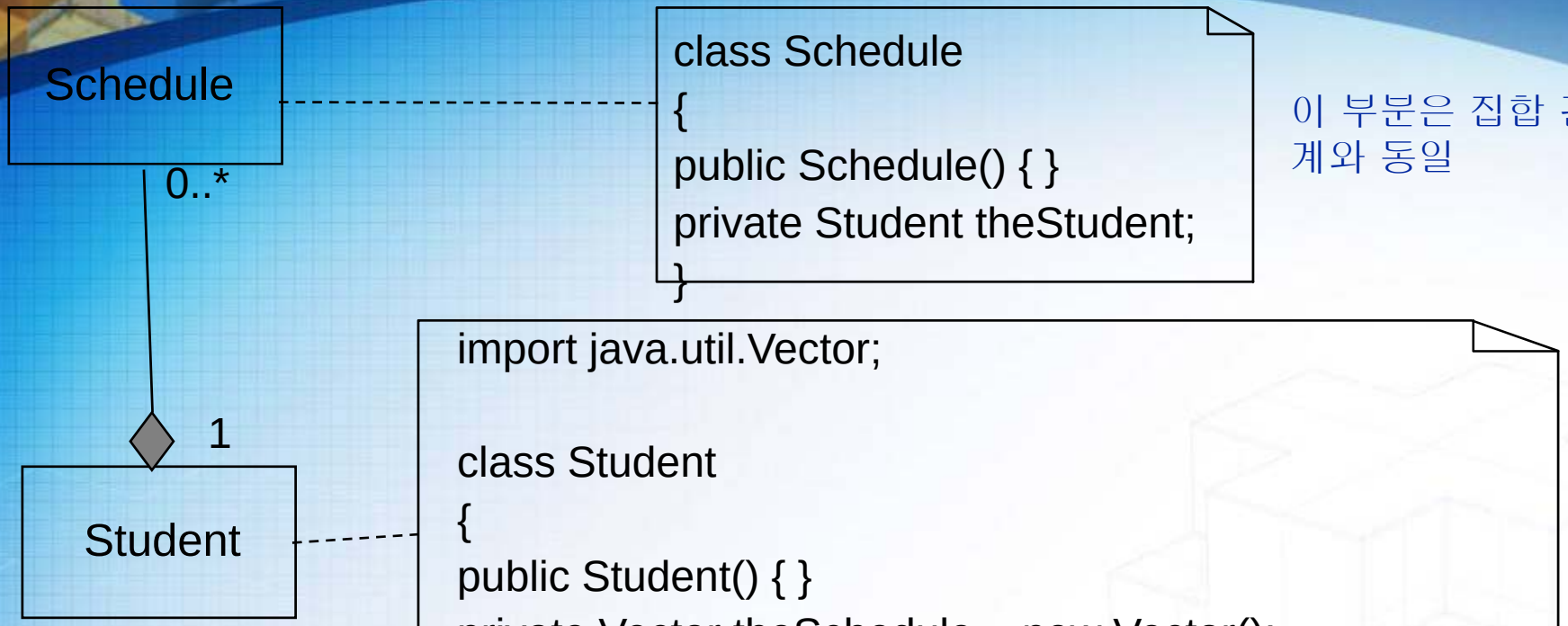
- 클래스 Course는 자기 자신에 속하는 객체와 연관
- Vector는 java.util에 있는 클래스로 객체의 배열을 관리
- Vector 타입의 객체에는 copyInto, elementAt, contains, insertElementAt, addElement 등의 메소드 적용

집합 관계



- ‘Student’ 클래스는 Vector 리스트 타입의 클래스를 선언
- 집합을 위한 생성자는 따로 없음. 객체의 배열로 취급
- 집합 관계의 구현은 연관 관계와 차이가 없음

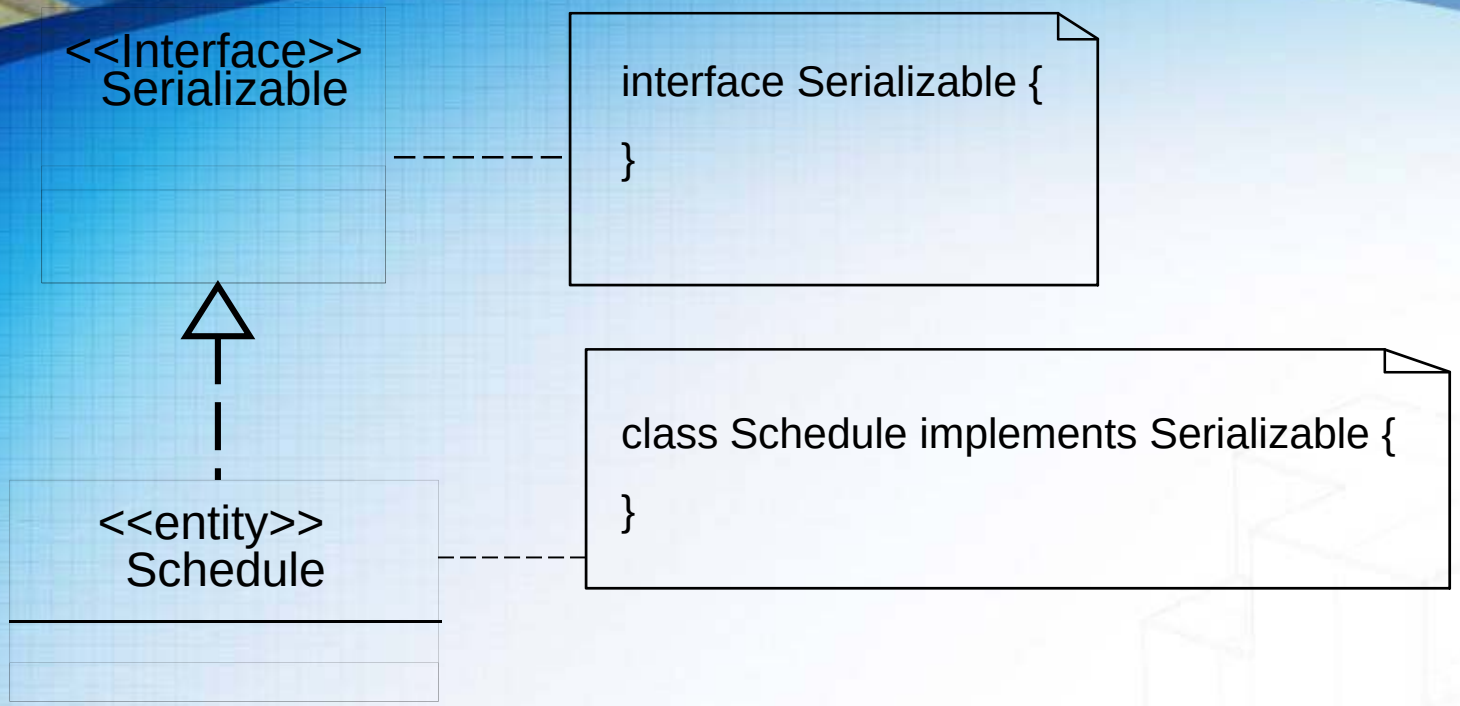
합성 관계



이 부분은 집합 관계와 동일

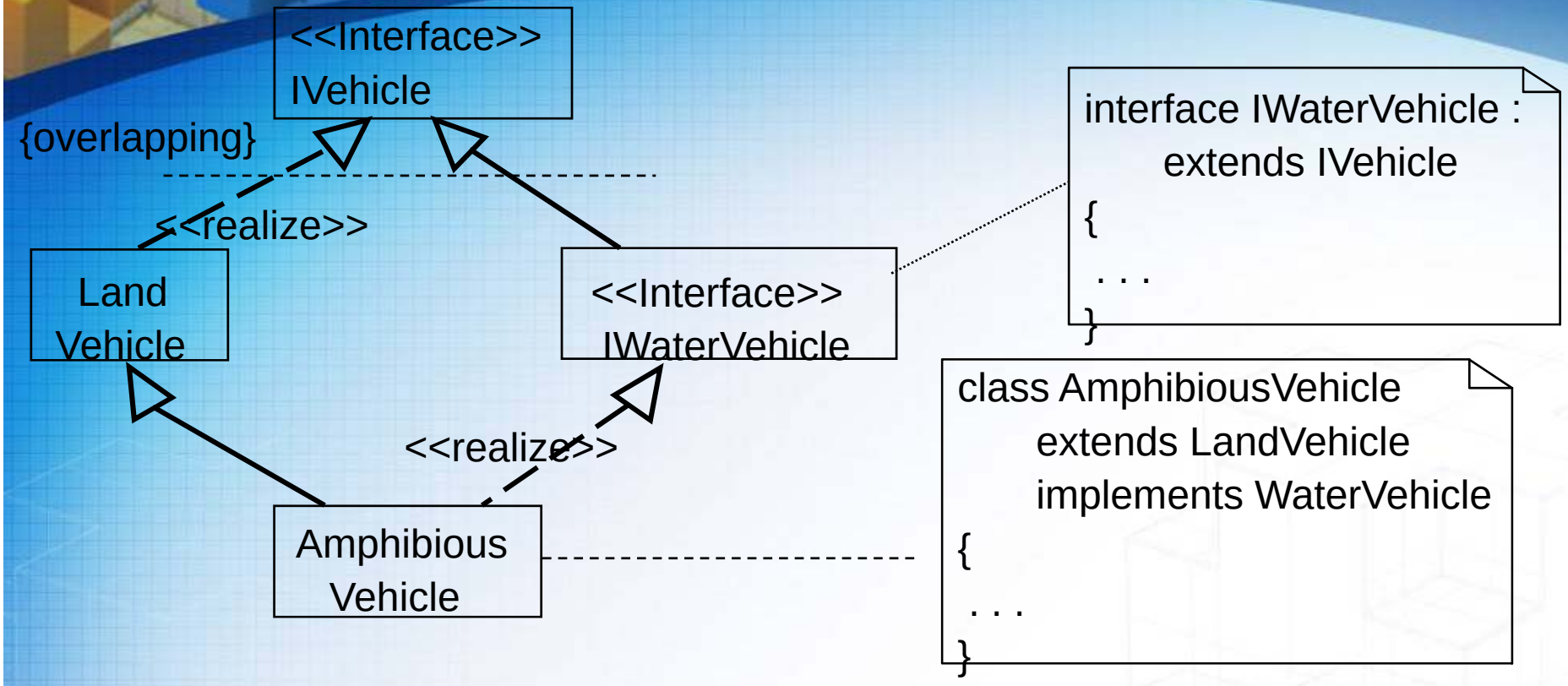
- Java는 containment by value 지원하지 않음
- theSchedule 클래스는 Student 클래스가 생성

Interface, Realize 관계



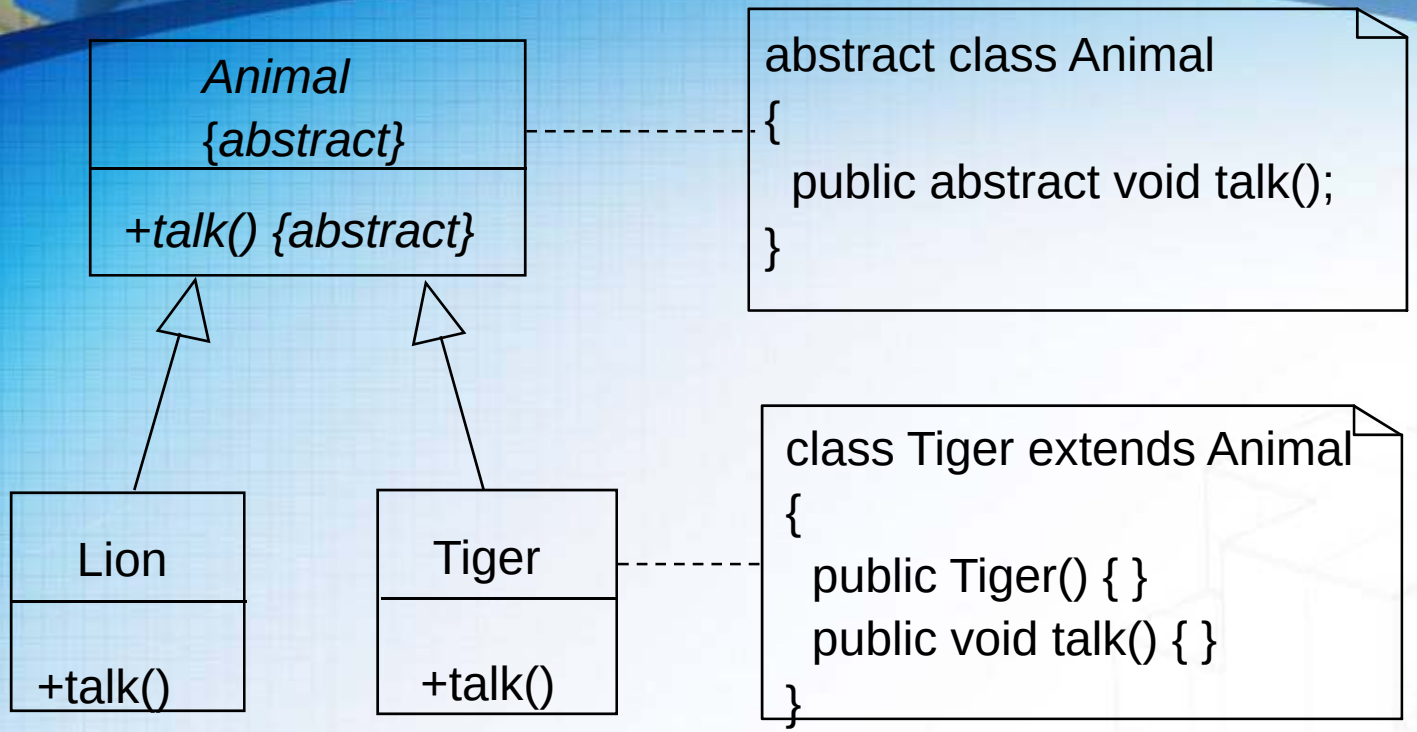
- UML의 realize 관계와 매칭되는 implement 구문이 있음
- Java 클래스는 여러 개의 인터페이스를 구현할 수도 있음
- Java의 경우 속성을 정의할 수도 있음

다중 상속



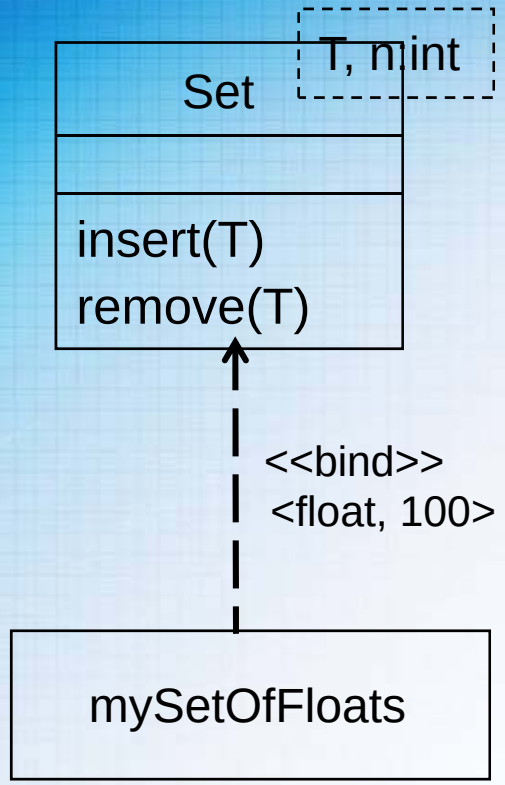
- Java는 단일 상속만 허용. 하지만 다중 인터페이스의 구현으로 같은 효과를 낼 수 있음

추상 클래스



- 추상 클래스는 인스턴스가 만들어지지 않는 클래스

파라미터 클래스



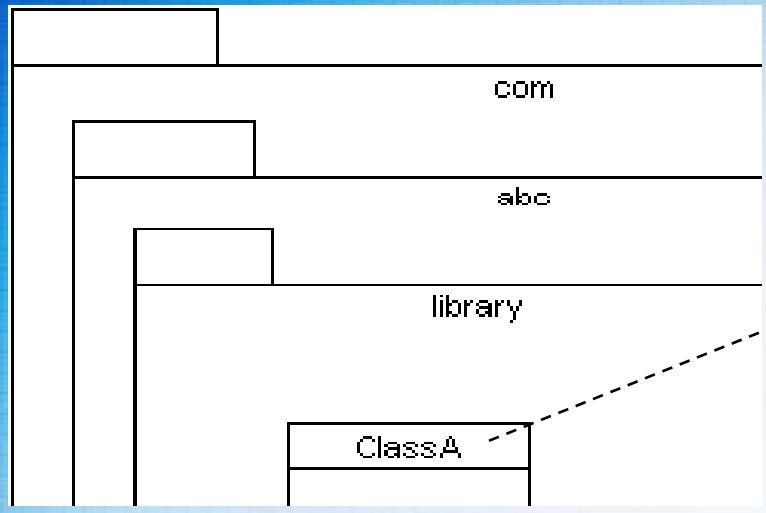

```
template <T, n>
class Set
{
  Vector T[100];
  public insert(T const &d);
  punlic remove(T const &d);
}
```

```
Set<float, 100> mySetOfFloats;
```

- Java는 파라미터 클래스를 지원하지 않음

패키지

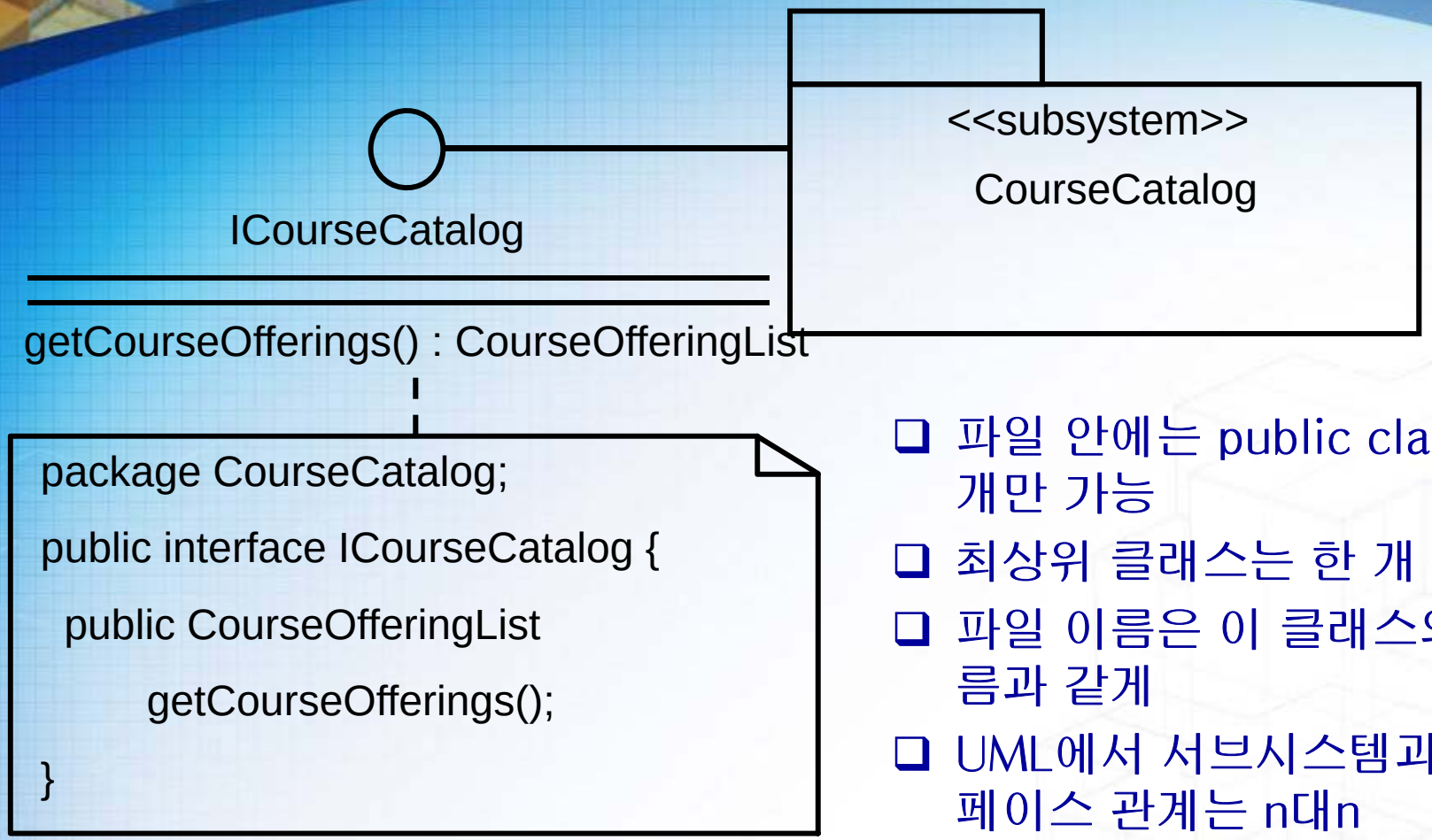
Package




```
package com.abc.library;
class ClassA …. {
…
}
```

- 패키지는 관련 있는 클래스 또는 인터페이스들을 그루핑한 것
  - 서브시스템이 될 수 있음
- 클래스들을 효율적으로 관리하기 위함
  - 계층적 구조
- 패키지 안에 선언된 다른 클래스는 import 하지 않아도 사용 가능

서브 시스템



- 파일 안에는 public class 한 개만 가능
- 최상위 클래스는 한 개
- 파일 이름은 이 클래스의 이름과 같게
- UML에서 서브시스템과 인터페이스 관계는 n대n
- Java는 1대 1

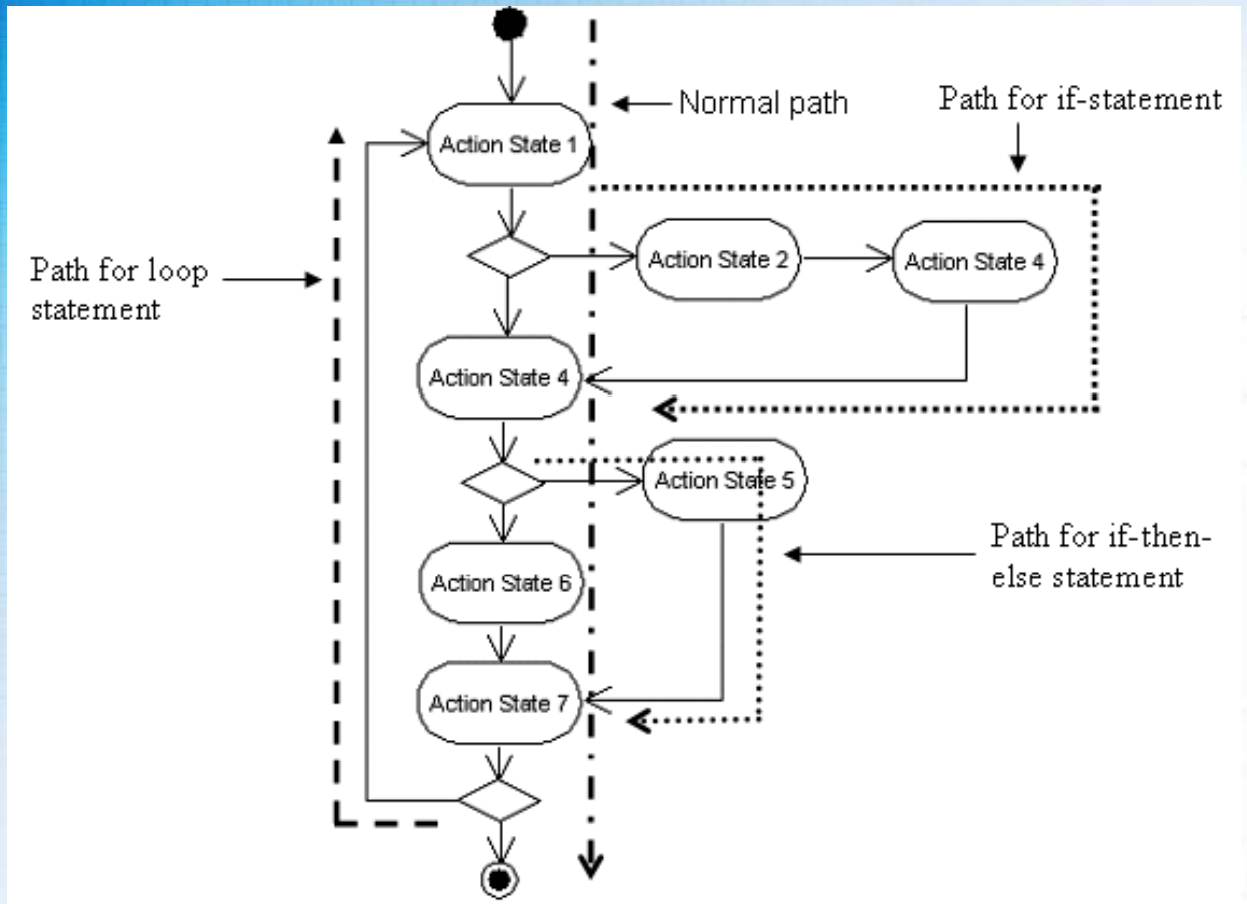
액티비티 다이어그램의 구현

- 액티비티나 프로시저 안에서 수행되어야 할 액션들의 순서를 나타냄
- 제어 객체나 서브시스템의 알고리즘이나 제어 흐름을 나타냄
- 두 가지 구현 방법
  - 프로그램의 위치, 즉 프로그램의 제어문, 반복문 으로 액티비티 다이어그램의 제어 흐름을 구현하는 방법
  - 제어 흐름을 상태 머신으로 구현하는 방법

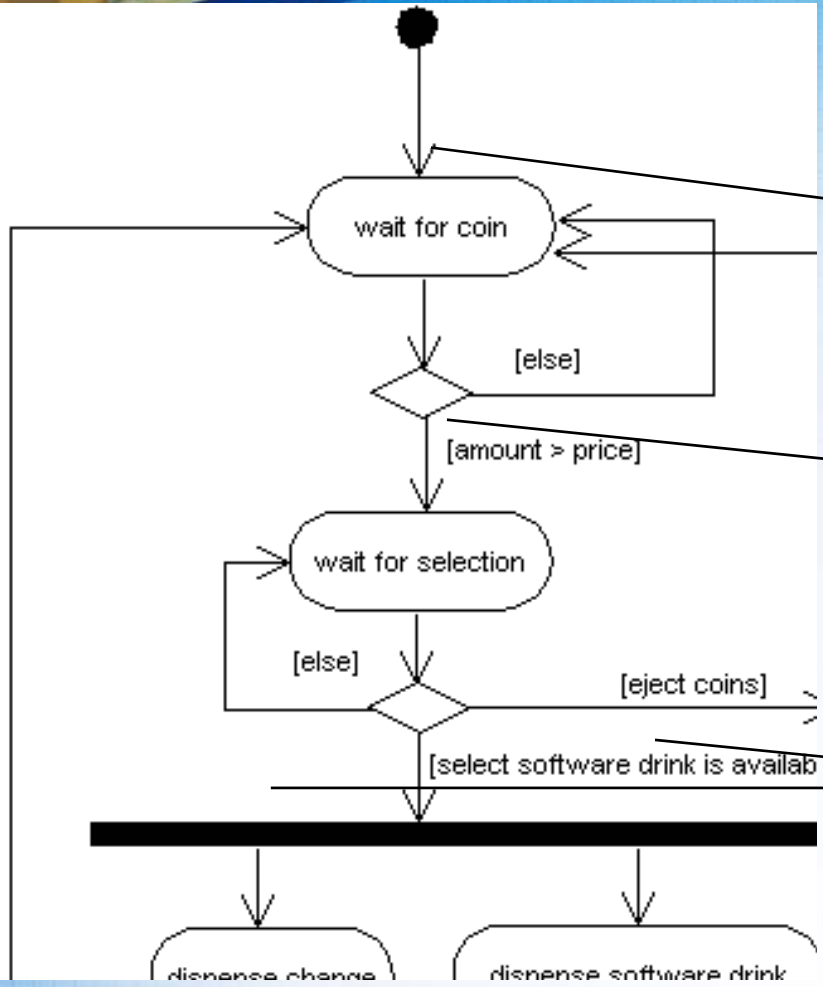
액티비티 다이어그램의 구현

- 액티비티 다이어그램을 코딩 하는 일반적인 규칙
- 액션 상태는 메소드 호출이나 일반 계산문장으로 구현
- 제어 노드는 if-then-else 문장으로 구현
- 병렬 노드는 스레드로 구현
- 반복 구조는 while 루프로 구현

액티비티 다이어그램의 구현



액티비티 다이어그램의 구현




```
while (true) {
  amount = 0.0;
  while (amount < price) {
    wait for a coin;
    add coin value to amount;
  }
  show all available soft drink;
  while (selection is not done) {
    wait for selection from user;
    if selection is “eject coins” {
      dispense coins;
      set selection to “done”;
    }
    else if selection is a valid soft drink {
      dispense change & soft drink concurrently;
      set selection to “done”
    }
  }
}
```

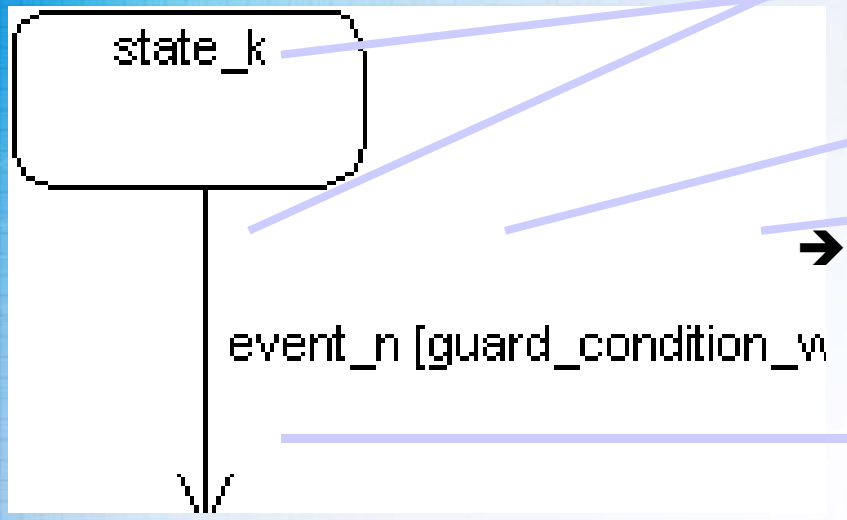
- Action은 메소드 호출, 분기 노드는 if, 포크 노드는 thread 로 코딩

상태 다이어그램의 구현

- Inactive 객체의 상태 다이어그램을 구현하는 방법
- 상태 다이어그램을 클래스로 매핑
- 상태정보를 저장하기 위한 속성 추가
- 이벤트는 메소드로 상태 변화나 이벤트의 액션은 메소드 안에 탑재

상태 다이어그램의 구현

- 모든 상태는 상태를 나타내는 속성의 값
- 상태 변화는 클래스의 메소드
- 가드는 메소드 안의 조건 체크




```
public void event_n(….) {
 switch (state) {
   case state_k:
     if (guard_condition_w) {
       state = state_m;
       perform actions of the transition;
     }
     break;
   case state_v:
     …
     …
 }
}
```

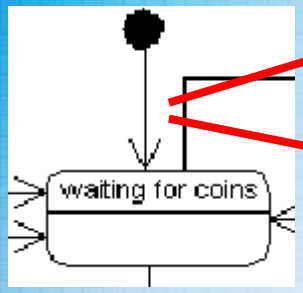
자판기 제어 객체

- 상태 다이어그램을 하나의 클래스로 구현
- 상태 정보를 저장하는 _state 속성 추가

```
class VendingMachineControl
{
  int _state;
  float _amount, _price;
  static final int WaitingCoin = 1;
  static final int WaitingSelection = 2;
  static final int DispensingSoftDrink = 3;
  static final int DispensingChange = 4;
  static final int EjectingCoins = 5;
```

상태 초기화

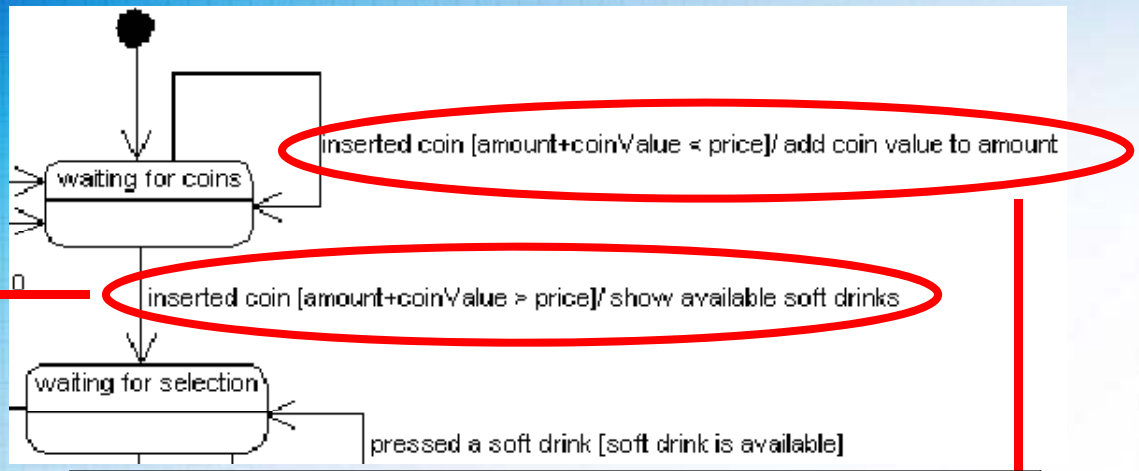
- 생성자 안에서 상태 변수 및 기타 변수 초기화




```
public VendingMachineControl(float price)
 {
   _amount = 0;
   _state = WaitingCoin;
   _price = price;
 }
```

이벤트는 메소드로

- 상태천이와 이벤트의 액션은 메소드 안에 구현




```
public void insertedCoin(float coinValue)
 {
   if (state == WaitingCoin)
   {
     amount += coinValue;
     if (amount >= price) {  // fire transition
       state = WaitingSelection;
       show available soft drinks;
     }
   }
 } // insertedCoin
```

순서 다이어그램

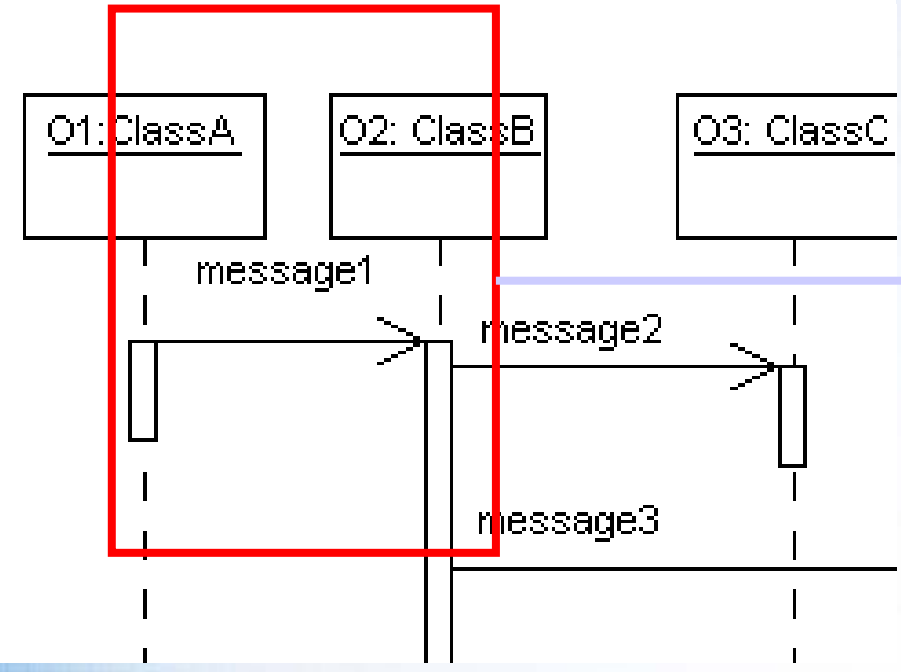
- 순서 다이어그램은 협력하는 객체들 사이의 메시지 교환을 나타낸 것
- 메시지는 화살표 나가는 객체에서 들어오는 객체로 메소드 호출
- 메시지를 받는 객체는 제 3의 객체에게 하나 이상의 메시지를 호출할 수 있음

순서 다이어그램의 구현

- 순서 다이어그램을 코딩하는 방법
- 메시지는 메소드의 호출로 코딩. 객체의 생성은 생성자(constructor)를 호출
- 메시지를 받는 객체의 클래스 안에 메소드 구현
- 분기구조는 if-else 문장과 같은 조건문으로 구현
- 병렬구조는 thread로 구현

순서 다이어그램의 구현

- 메시지가 호출 당하는 클래스 안에 메소드 구현




```
ClassB
{
  ClassC o3;
  ClassD o4;
  method1(…)
  {
    o3.method2(..);
    o4.method3(..);
  }
}
```