

소프트웨어 공학 개론

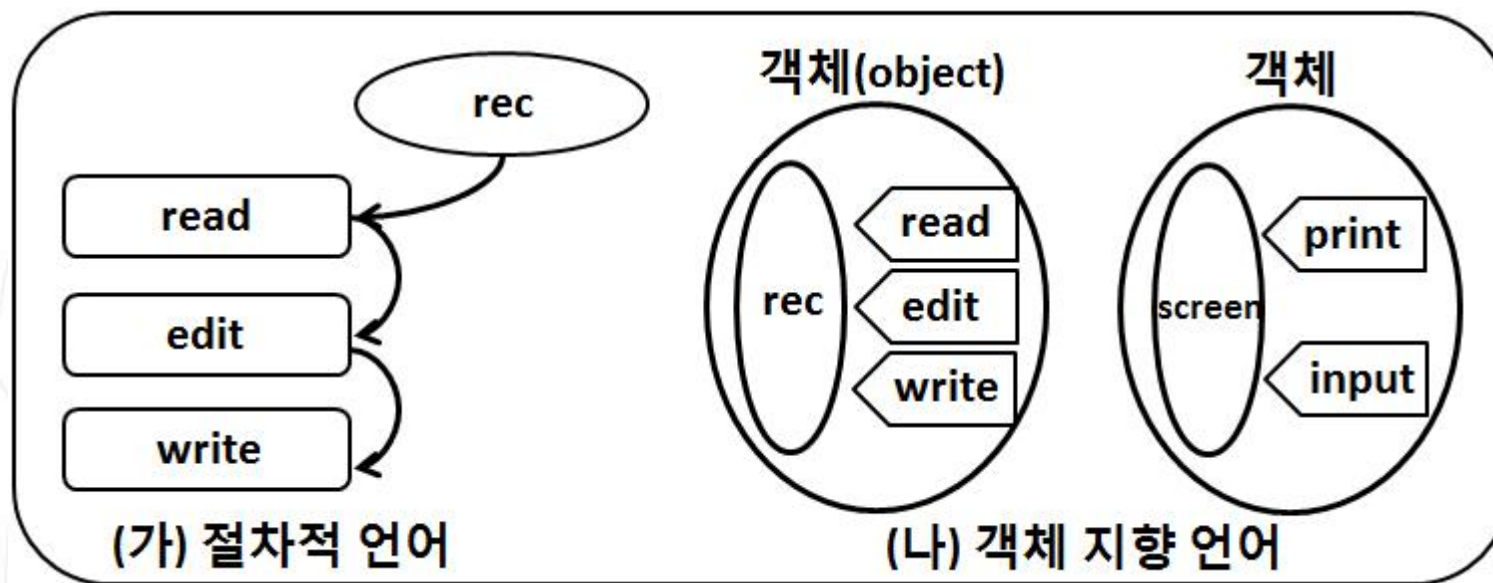
강의 5: 객체지향 개념

최은만
동국대학교 컴퓨터공학과



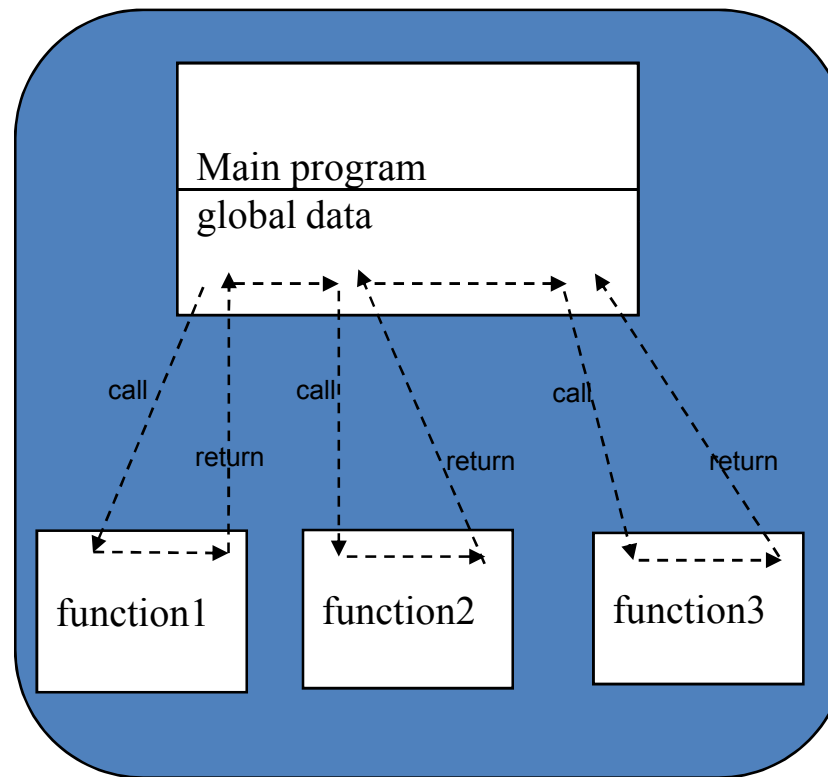
왜 객체지향인가?

- 절차적 패러다임 vs. 객체지향 패러다임
- 뭐가 다르지?



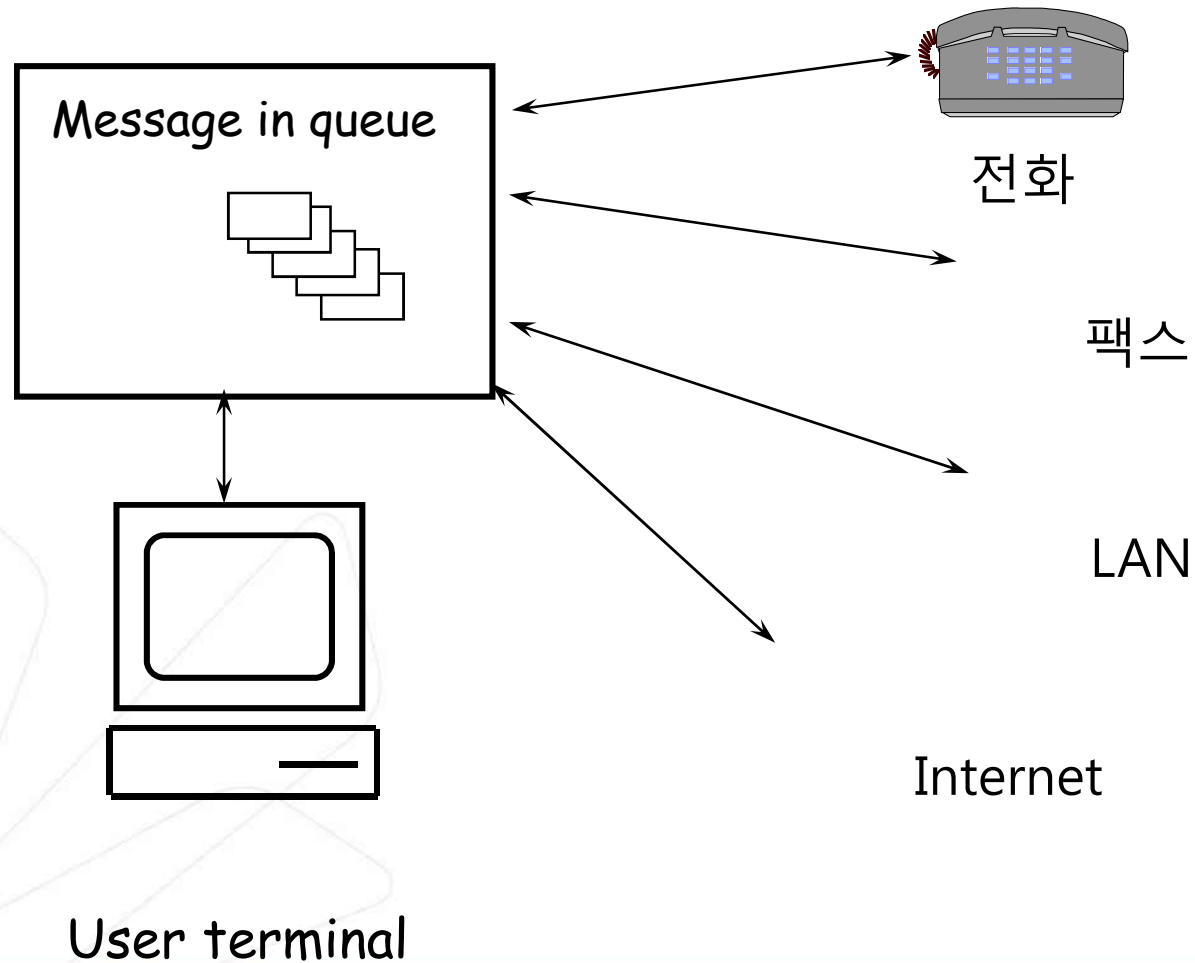
C 언어

- 프로그램은 데이터와 함수로 구성
- 함수는 데이터를 조작
- 프로그램을 조직화 하기 위해
 - 기능적 분할
 - 자료 흐름도
 - 모듈

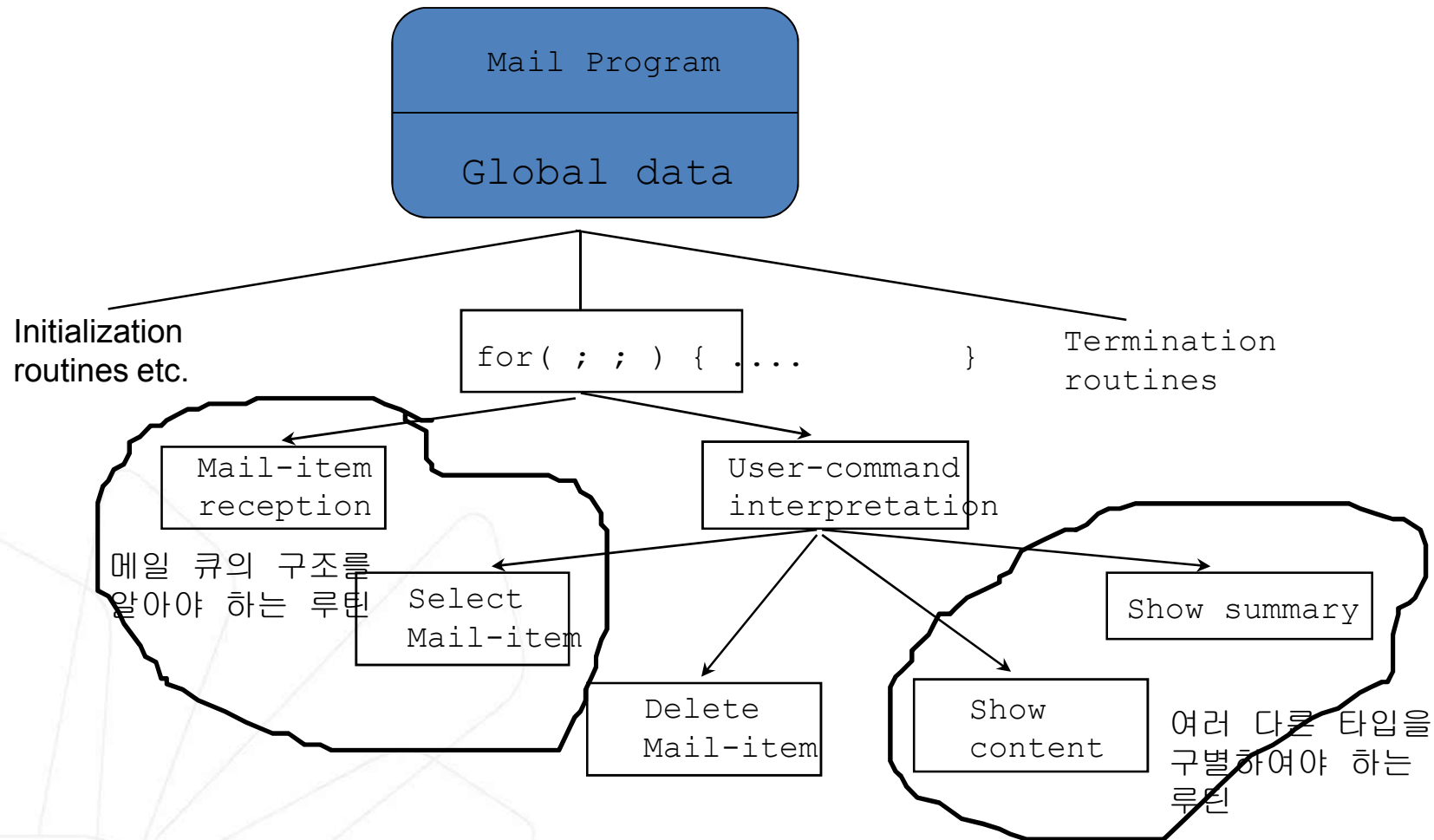


C 언어 - 절차적 프로그래밍

- 메일 처리 시스템



절차적 프로그램의 구조



너무 자세한 요구의 유입

```
void ShowContent(Mainitem* mailitem)
```

```
{
```

```
...
```

```
switch(mailitem->type)
```

```
{
```

```
case EMAIL:      ShowContentofEmail(mailitem); break;
```

```
case FAX:   DrawPictureofFax(mailitem); break;
```

```
...
```

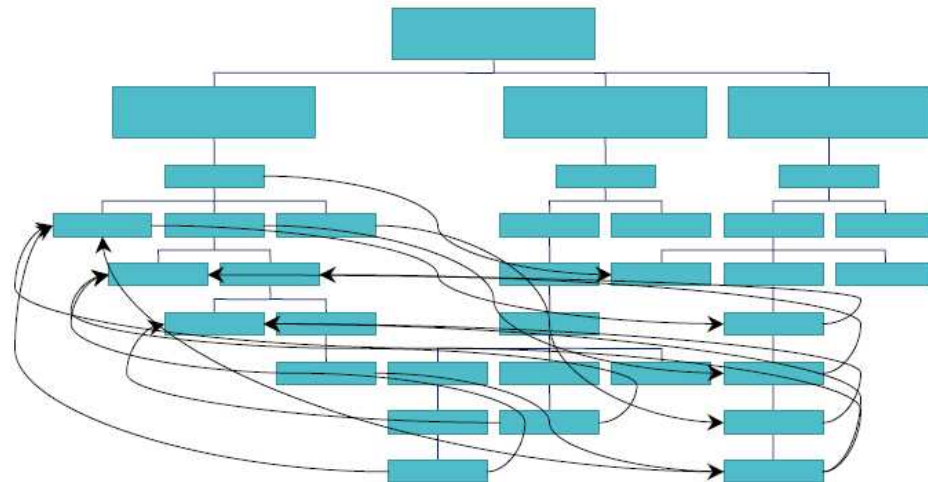
```
}
```

```
}
```

메일 타입 추가되면
구조 변경됨

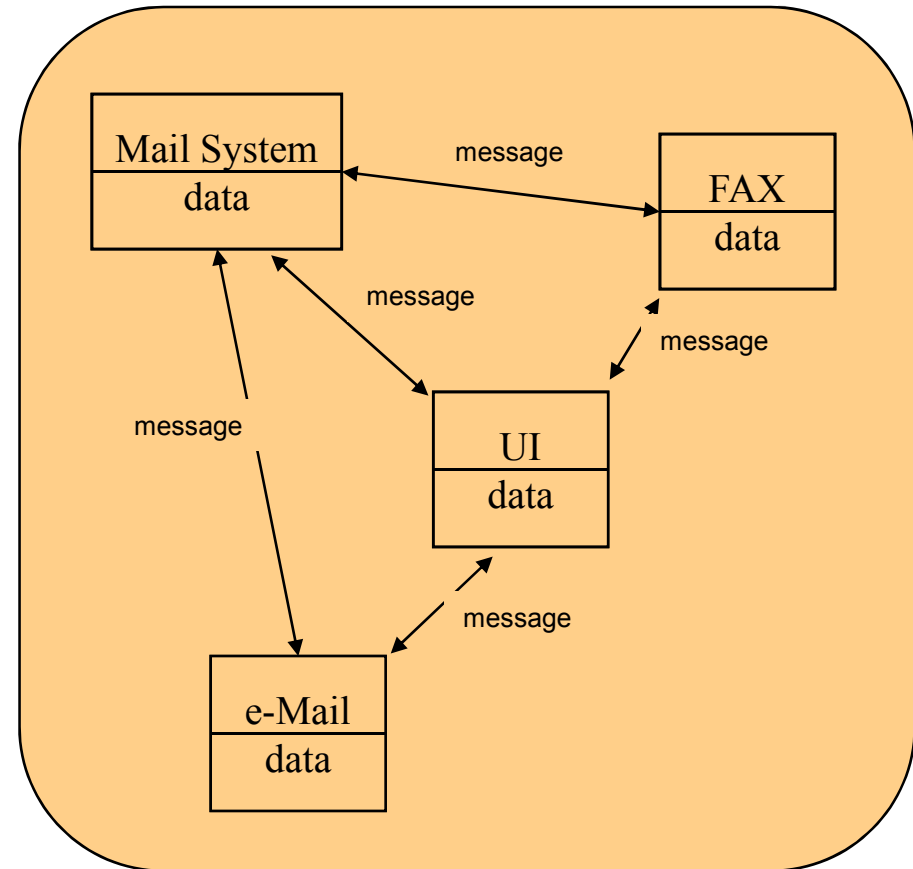
C 프로그램

- 프로그램 = 함수의 집합
 - 함수가 서로 자료를 주고 받음
- 함수와 자료가 분리
- 설계 작업
 - 자료, 함수 안의 알고리즘, 함수의 구조에 초점
- 함수의 재사용이 어려움
- 설계 방법
 - 함수와 자료를 별도로 생각
 - 모듈 사이의 연관 복잡

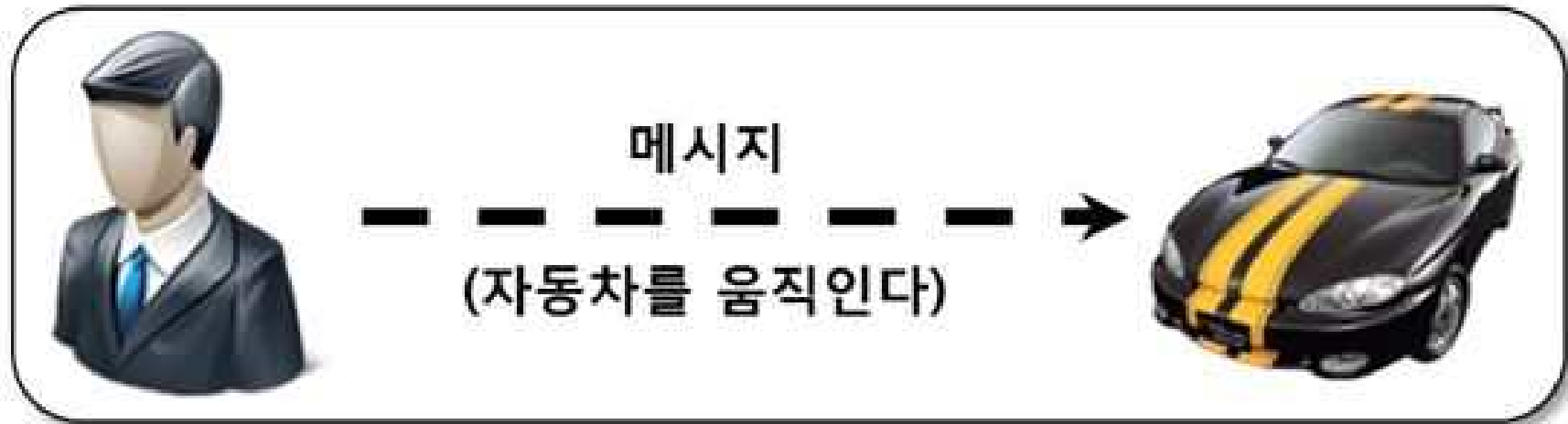


New Way - 객체지향

- 프로그램 = 클래스의 집합
- 프로그래밍
 - 어떤 객체가 필요하며
 - 어떤 **오퍼레이션**이 필요
 - 어떻게 서로 협력하여야 하는지를 결정하는 일
- 메일 타입 추가 용이
 - 다른 구조에 영향이 적음



메시지 호출



```
class Man {  
    public static void main(String[] args) {  
        Car car = new Car();  
        car.move();  
    }  
}
```

```
class Car {  
    public void move() {  
        // 차가 움직인다.  
    }  
}
```

복잡함을 잘 다루는 방법

- PC 하드웨어

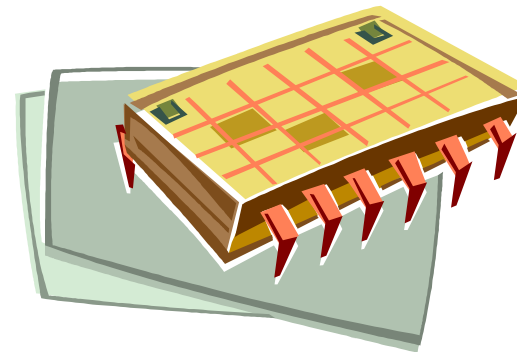
- 버스
- 메모리
- CPU
- 스크린
- 디스크 드라이브



내부는 복잡하지만
인터페이스는 간단
-업그레이드 간단
-새 PC 설계, 제조 간단해짐

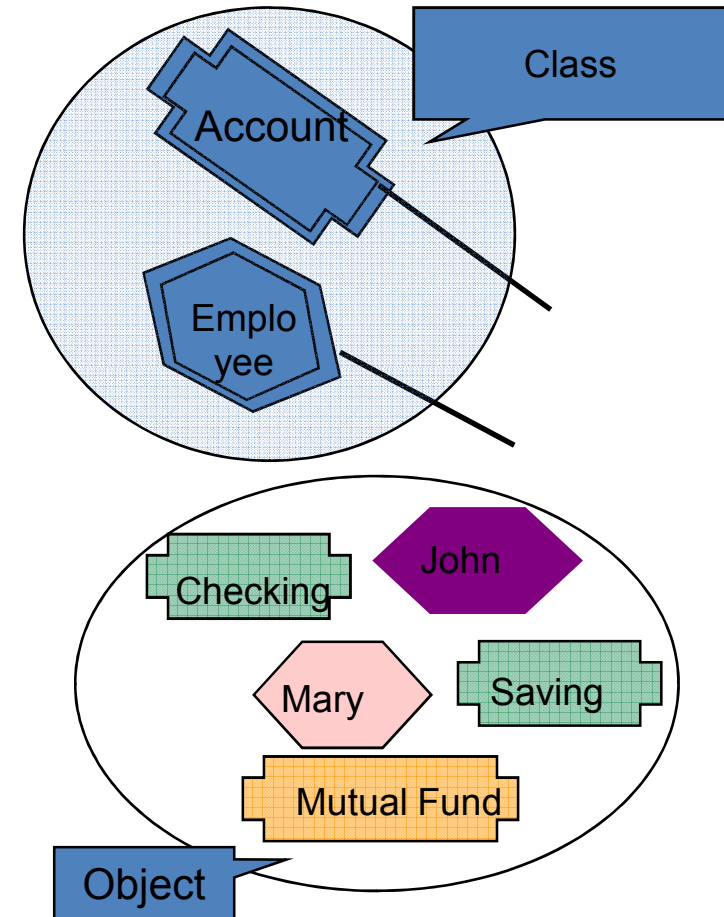
- 객체

- 내부 - 데이터와 이를 조작하는 함수가 있는 작은 단위의 프로그램
- 외부 - 함수 인터페이스, 내부의 복잡한 변수는 감추어짐



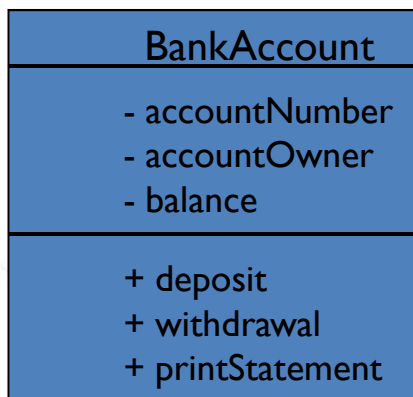
클래스와 객체

- 클래스: 객체를 정의한 **템플릿**
- 객체는 **생성자**로 생성
 - 같은 클래스에 속하는 메소드(예 deposit, withdraw, fundTransfer, ...)
 - 각 객체와 관련된 인스턴스 변수의 값은 다름 (예 accountOwner, balance, creditLimit)
- 메소드와 변수의 관계
 - 객체와 관련된 메소드(예 withdraw)가 실행되면 인스턴스 변수의 값을 바꾸어 놓음(예 balance)



클래스

- 클래스에 의하여 제공될 서비스를 정의한 것 – public 메소드
- 내부에서만 사용될 메소드 – private 메소드
- 인스턴스 변수 – 외부 조작이 불가능하도록 private 선언



```
public class BankAccount {  
  
    public void BankAccount {  
    }  
  
    public void deposit(int amount) {  
        balance += amount;  
    }  
    // additional methods such as withdrawal...  
  
    private String accountNumber;  
    private String accountOwner;  
    private int balance = 0;  
    ...  
}
```

Multiple constructors may be provided

Method definition

instance variables / attributes

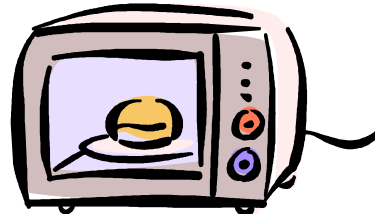
클래스:=자료+오퍼레이션

- 사원

- 이름
- 전화번호
- 직위
- 급여
- 경력
- ...

- 오퍼레이션

- 승진
- 전화번호 검색
- 경력 조회
- 직위 조회
- ...



- 타이머

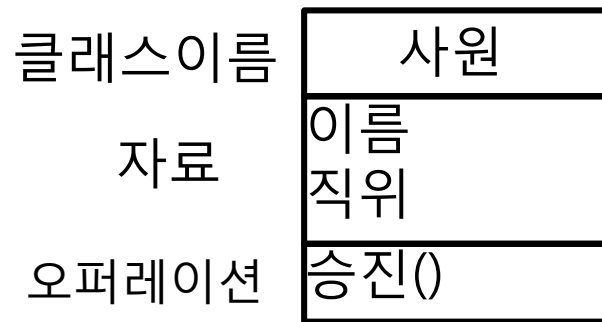
- 시각
- 세팅 시간
- ...
- 오퍼레이션
- 시각 변경
- 세팅
- Clear
- ...

- 전화번호

- 이름
- 전화번호
- 그룹
- 메일주소
- ...
- 오퍼레이션
- 추가
- 삭제
- 전화번호 검색
- ...



클래스와 객체



이름:김영희
직위:부장



이름:홍길동
직위:직원



이름:김동국
직위:팀장

- 클래스
 - 타입 선언
 - 객체들이 갖는 자료와 오퍼레이션을 **정의**한 것
 - 객체 생성을 위한 템플릿

- 객체
 - 클래스의 인스턴스
 - 구체적인 자료값(**실체**)을 가짐

Java - BankAccount.java - Eclipse SDK

File Edit Source Refactor Navigate Search Project Run Window Help

P RecursionDemo.java BankAccount.java BankAccountTest.java Forward

```
public class BankAccount{

    // Constructors
    public BankAccount() {
        accountNumber = "";
        accountOwner = "";
        balance = 0;
        printStatement("Generic Account Creation ");
    }

    public BankAccount(String accNo, String name, int amount){
        accountNumber = accNo;
        accountOwner = name;
        balance = amount;
        printStatement ("Specific Account Creation ");
    }

    // methods
    public void deposit (int amount) {
        balance += amount;
        printStatement("Deposit");
    }

    public void withdrawal (int amount) {
        if (balance >= amount)
            balance -= amount;
        else
            System.out.println ("Insufficient balance. No withdrawal.");
        printStatement("Withdrawal");
    }

    public void printStatement(String activity) {
        System.out.println ("After processing " + activity + " transaction request: ");
        System.out.println (" As of " + new java.util.Date());
        System.out.println ("     Account Number : " + accountNumber);
        System.out.println ("     Account Owner : " + accountOwner);
        System.out.println ("     Balance : " + balance + " won");
    }

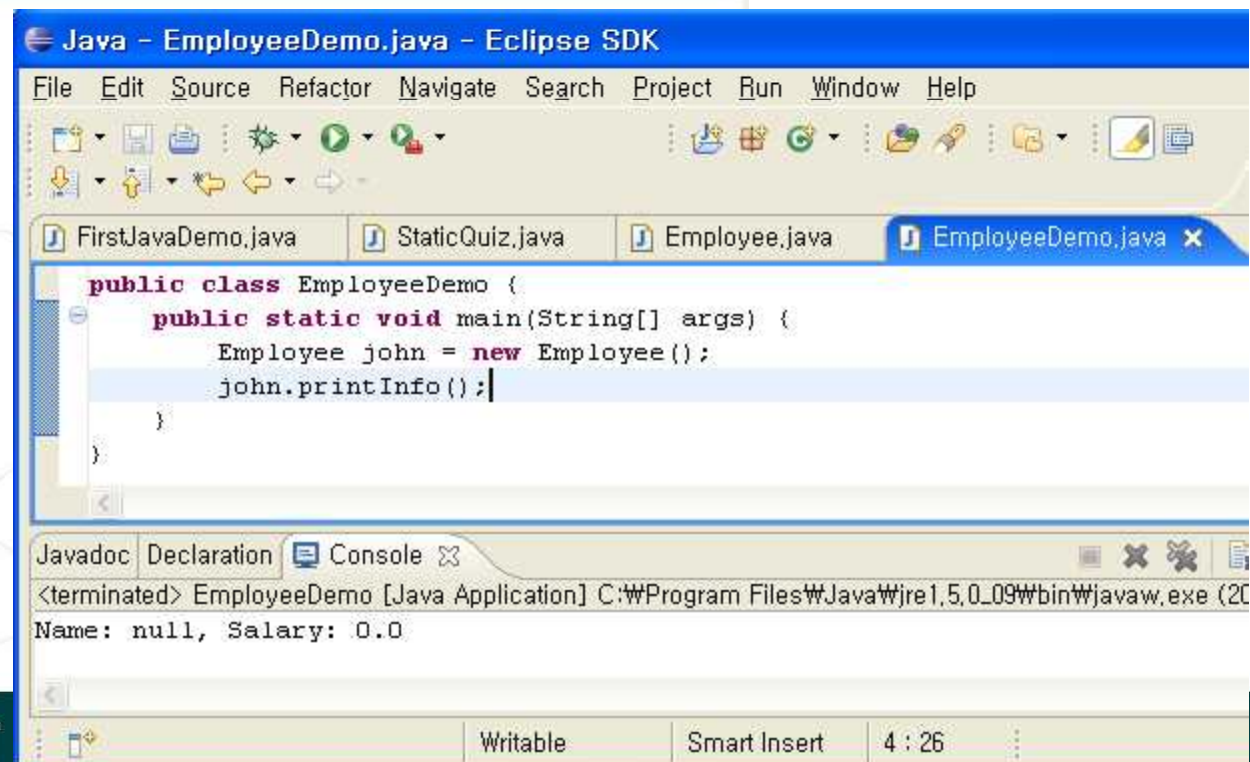
    private String accountNumber;
    private String accountOwner;
    private int balance = 0;
}
```


객체의 생성



The screenshot shows the Eclipse IDE with the file `Employee.java` open. The code defines a public class `Employee` with a `printInfo()` method that prints the employee's name and salary. The class has two private attributes: `String name` and `double Salary`.

```
public class Employee {  
    public void printInfo() {  
        System.out.println ("Name: " + name + ", Salary: " + Salary);  
    }  
    private String name;  
    private double Salary;  
}
```



The screenshot shows the Eclipse IDE with the file `EmployeeDemo.java` open. The code defines a public class `EmployeeDemo` with a `main` method that creates a new `Employee` object named `john` and calls its `printInfo()` method. The console output at the bottom shows the result of the program execution.

```
public class EmployeeDemo {  
    public static void main(String[] args) {  
        Employee john = new Employee();  
        john.printInfo();  
    }  
}
```

Console Output:

```
<terminated> EmployeeDemo [Java Application] C:\Program Files\Java\jre1.5.0_09\bin\javaw.exe (20  
Name: null, Salary: 0.0
```


메소드

```
Java - EmployeeDemo.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help

FirstJavaDemo.java StaticQuiz.java Employee.java EmployeeDemo.java x

public class EmployeeDemo {
    public static void giveTwice (Employee who) {
        who.giveRaise(100.0);
    }

    public static void main(String[] args) {
        Employee john = new Employee("John Public", 10000.0);
        john.printInfo();
        giveTwice (john);
        john.printInfo();
        giveTwice (john);
        john.printInfo();
    }
}
```

Javadoc Declaration Console x

<terminated> EmployeeDemo [Java Application] C

Name: John Public, Salary: 10000.0
Name: John Public, Salary: 20000.0
Name: John Public, Salary: 40000.0

```
Java - Employee.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help

FirstJavaDemo.java StaticQuiz.java Employee.java x EmployeeDemo.java

public class Employee {
    public Employee() {
        name = ""; salary = 0.0;
    }

    public Employee (String who, double howMuch) {
        name = who; salary = howMuch;
    }

    public void printInfo() {
        System.out.println ("Name: " + name + ", Salary: " + salary);
    }

    public void giveRaise (double percentage) {
        salary += salary * percentage / 100.0;
    }

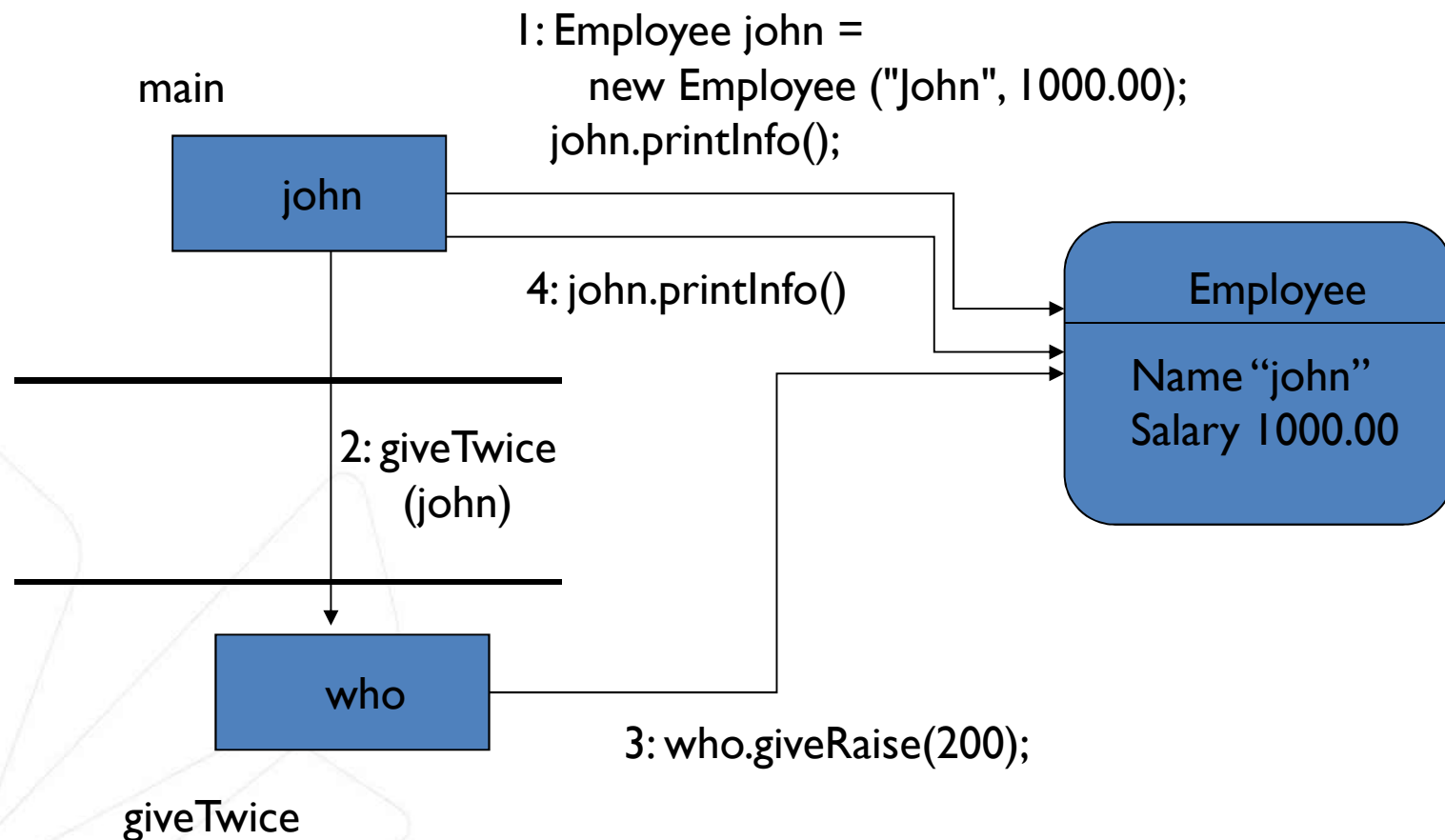
    private String name;
    private double salary;
}
```

메소드

- 메소드는 문장 블록과는 다르게 중첩(nested)될 수 없음
- 메소드는 호출되기 전에 선언할 필요가 없음
 - 정의와 호출 순서가 뒤바뀌어도 됨
- 실행은 항상 메인 메소드로부터

```
public class MyMainClass
{
    public static void main(String[] args)
    {
        <statements that define the main method>
    }
}
```

메소드의 호출



상속

- 상속의 의미
 - 상위 클래스의 속성과 연산을 물려 받음(Employee 객체는 name, age 인스턴스 변수와, birthday() 메소드를 가짐)
- 슈퍼클래스(superclass), 서브 클래스(subclass)
 - 예>직원 : 슈퍼클래스 , 관리자 : 서브클래스

```
class Person {  
    String name;  
    int age;  
    void birthday () {  
        age = age + 1;  
    }  
}
```

```
class Employee  
    extends Person {  
    double salary;  
    void pay () { ...}  
}
```

다형성(polymorphism)

- 다형성의 정의
 - 여러 형태를 가지고 있다 (=여러 형태를 받아들일 수 있다)
- 같은 이름의 메시지를 다른 객체 or 서브클래스에서 가질 수 있음

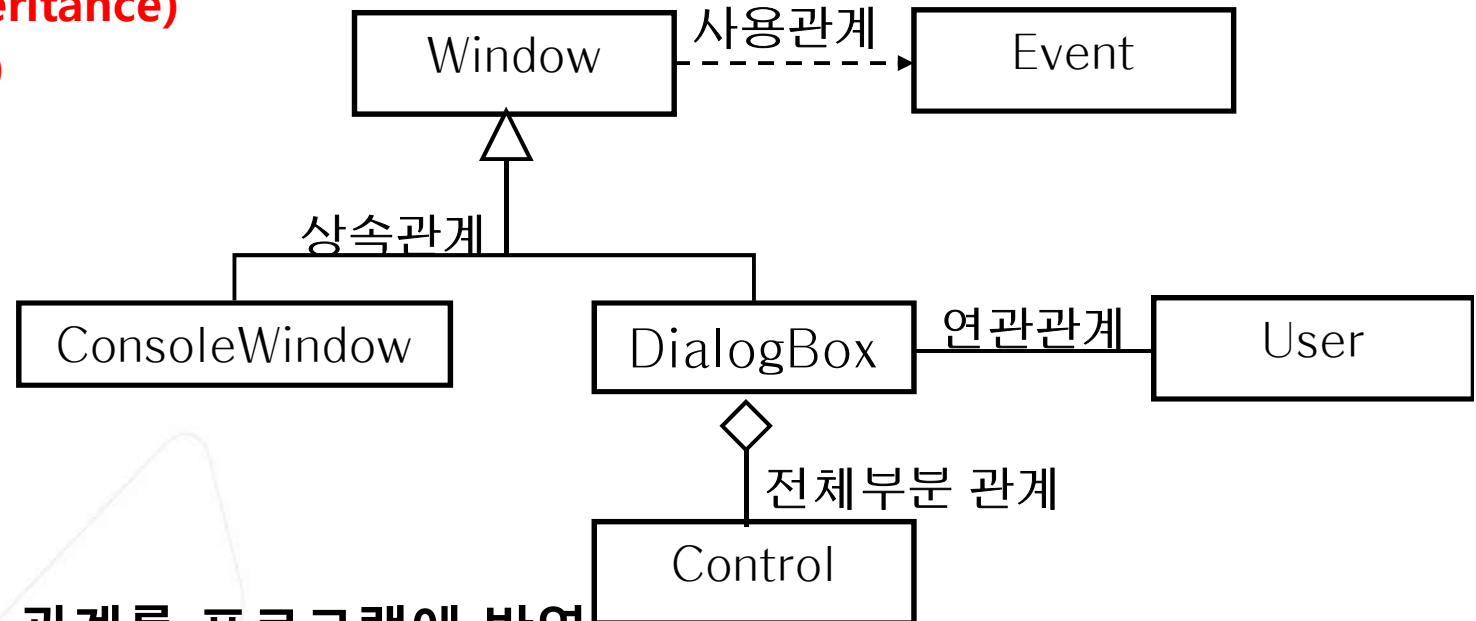
```
class FamilyMember extends Person {  
    void birthday () { // override birthday() in Person  
        super.birthday (); // call overridden method  
        givePresent ();    // and add your new stuff  
    }  
}  
  
FamilyMember Jennifer = new FamilyMember();  
Person P = new Person();  
Jennifer.birthday();  
P.birthday();
```

변수가 서브클래스의 객체를 가질 수 있음

- FamilyMember는 Person 클래스의 **서브 클래스**
 - FamilyMember 객체는 FamilyMember 변수의 값으로 지정될 수 있음
 - Person 객체는 Person 변수의 값으로 지정될 수 있음
 - FamilyMember 객체는 Person 변수의 값으로 지정될 수 있음
 - Person 객체는 FamilyMember 변수의 값으로 지정될 수 없음
 - 모든 FamilyMember는 Person이나 모든 Person이 FamilyMember는 아님
- **캐스트** 가능
 - `Person Rich = new Person();`
 - `cast son_in_law = (FamilyMember) Rich`

클래스의 관계

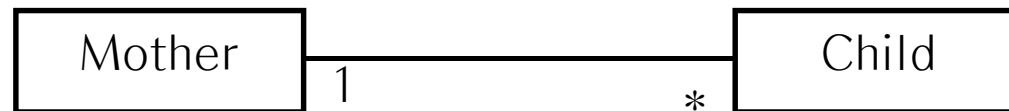
- 객체지향 모델링의 관계
 - 연관(association)
 - 전체 부분(whole/part)
 - 상속(inheritance)
 - 사용(use)



- 객체 사이의 관계를 프로그램에 반영

연관 관계

- 객체와 객체를 연결하는 구조적인 관계
- 방향성과 다중도를 고려



```
Class Mother{
    .....
    private Child[] theKids = new Child[20];
    public addChild(Child ch);
}
```

배열 theKids는 Child에 대한 레퍼런스를 저장
배열의 크기가 Child의 수를 제한

```
Class Child{
    .....
    private Mother mom;
    public setMom(mom);
}
```

변수 mom이 mother객체를 레퍼런스 함
Mom을 선언하므로 연관관계를 맺는다

연관 관계

- 연관관계 생성 코드
- **추적가능성(navigability)** 확인

- 서버클래스의 표현을 바꿀때
클라이언트 코드를 변경할
필요 없다

- 배열로 표현된 theKids를
벡터로 바꿀 수 있다
- 이때 theKids를 private로 선언하면 클라이언트 프로그램에
영향을 주지 않고 서버클래스(Mother)의 표현을 바꿀 수 있다.

```
Mother theMom = new Mother();  
Child jim = new Child();  
Child jennifer new Child();  
theMom.addChild(jim);  
theMom.addChild(jennifer);  
Jim.setMom(theMom);  
Jinnifer.setMom(theMom)
```

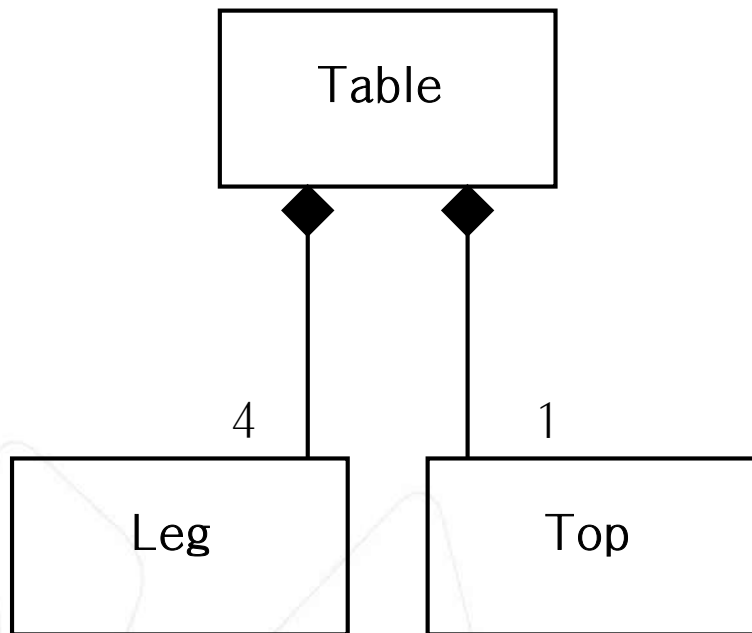
전체부분 관계 - 구성관계

- 전체부분(whole-part) 관계
 - 구성관계(composition), 집합관계(aggregation)
- 구성관계
 - 전체 개념 안에 구성요소 존재(테이블:4개의 다리+1개의 상판)
 - 연관관계의 일종으로 관계 표시는 없어도 된다
 - 방향성 ,추적가능성 고려, 컨테이너 객체 이용
 - 대부분 has, comprise, consist of의 의미
- 구성관계의 특성
 - 구성요소가 없이 전체가 **존재할 수 없다**
 - 구성요소는 언제나 하나의 전체객체에 대한 부품이다
 - 구성관계는 **이질적** 구성요소로 되어 있다
- UML표현: 검은 다이아몬드

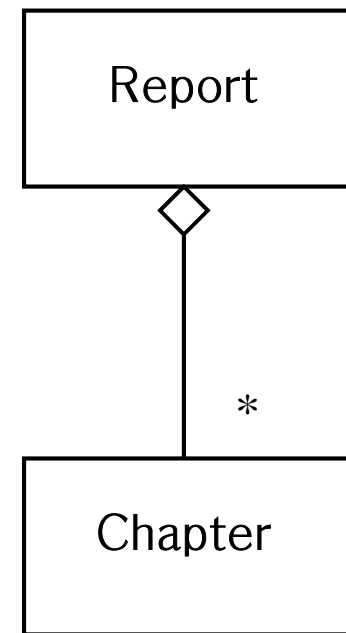
전체부분 관계 - 집합관계

- 집합관계
 - 예> 숲은 나무의 집합, 시/도는 군/구의 집합
 - 컨테이너 클래스 사용
 - 전체 개념의 클래스로부터 구성요소를 찾을 수 있음
- 집합관계의 특성
 - 구성요소가 없이 전체가 **존재할 수 있다**
 - 구성요소는 하나 이상의 전체집합에 소속 가능하다
 - 구성관계는 **동질적** 구성요소로 되어 있다
- UML표현: 흰 다이아몬드

전체부분 관계 UML



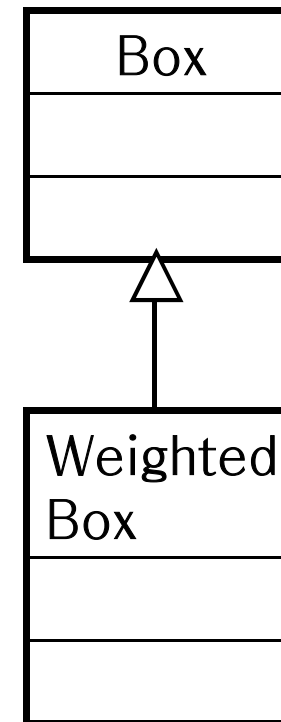
구성관계



집합관계

상속관계

- 상속관계(= 일반화 관계)
 - 일반적 개념의 클래스와 더 구체적 클래스의 관계
 - **A kind of**의 관계
- 명칭
 - 일반적인 클래스 : 베이스 클래스
 - 구체적인 클래스 : 파생된(derived) 클래스
- UML표현: 자식클래스에서 부모 쪽으로 화살표
 - 상향식 화살표로 베이스 클래스를 포인트
- 단일상속, 다중상속
 - 단일상속 : 하나의 베이스 클래스를 갖는다
 - 다중상속 : 두 개 이상의 베이스 클래스를 갖는다



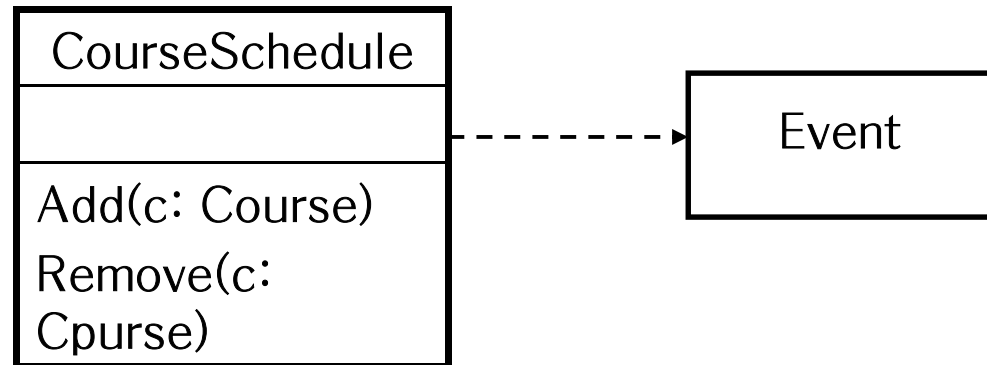
사용 관계

- 사용 관계

- 한 클래스가 다른 클래스를 코드 안에서 사용할 때

- 의미

- 코드상의 의존 관계
- 종속된 관계



- 일반적 유형

- 오퍼레이션의 매개변수로 다른 클래스를 사용하는 클래스 간의 연결
 - 예> **CourseSchedule** 클래스는 `add`과 `remove` 오퍼레이션을 위해 **Course** 클래스를 매개변수로 사용한다

- UML표현: 점선 화살표

관계의 비교

	연관관계	전체부분 관계	상속관계	사용관계
관계	클래스 사이에 영구적인 의미가 있는 관계	명확한 전체 부분 개념	일반적, 구체적 관계	한 클래스에서 다른 클래스 객체의 서비스를 사용
유지기간	클래스 상태의 일부분으로 객체가 살아있는 동안만 유지	클래스 상태의 일부분으로 클래스 객체가 살아있는 동안만 유지	서브 클래스가 정의 될 동안 영구적	클라이언트나 서버 메소드가 활성화된 경우만 관계 유지
구현	관련된 객체에 대한 인스턴스 변수를 정의, 다중도를 위하여 컨테이너 객체 사용	링크에 대한 레퍼런스를 인스턴스 변수로 정의, 다중도를 위하여 컨테이너 객체 사용	상속을 사용, java의 경우 서브클래스가 슈퍼클래스를 확장	클라이언트 클래스 메소드가 서버 클래스에 대한 레퍼런스를 매개변수로 가짐
UML				



Questions?

