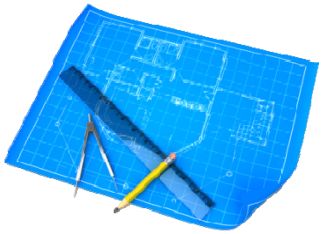
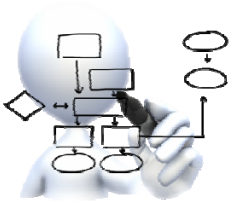


객체지향 프로그래밍 개념 재검토

NOTE 01



한국기술교육대학교 컴퓨터공학부 김상진

sangjin@koreatech.ac.kr
www.facebook.com/sangjin.kim.koreatech

강의목표

- 강의목표
  - 객체지향 개념 재검토
    - 객체, 클래스 개념
      - 캡슐화, 접근제어, 구조화 프로그래밍과의 차이점
    - 클래스 간의 관계
    - 객체 간의 관계
    - 상속, 인터페이스
  - 자바 프로그래밍 언어와 객체지향

객체지향 프로그래밍 (1/2)

- 컴퓨터가 어떤 유용한 일을 하기 위해서는 그것을 할 수 있도록 방법을 알려주어야 함
- 보다 편리하게 방법을 알려주기 위해 고급 프로그래밍 언어를 개발하여 사용하고 있음
- 방법이란 보통 절차를 말함. 따라서 초기 프로그래밍 기법은 절차 위주였음. 즉, 어떤 함수들을 만들어야 하는지가 초점이었음
- C언어로 대표되는 구조화 프로그래밍(structured programming)은 제공해야 하는 기능과 처리해야 하는 데이터가 복잡해진 오늘날 응용을 개발하는데 한계가 있음
- 이를 극복하기 위한 새로운 프로그래밍 패러다임이 필요하여 객체지향(object-oriented) 개념이 등장함
  - 객체지향 프로그래밍 (OOP: 데이터+절차를 하나로 묶음)
    - 구조화 프로그래밍에서는 모델링하고자 하는 실제를 바라보는 우리의 보편적 시각과 설계적 시각에 많은 차이가 있었음
    - 객체지향에서는 실제를 바라보는 우리의 시각과 유사하게 모델링하여 프로그래밍할 수 있음

객체지향 프로그래밍 (2/2)

- OOP에서는 주어진 문제를 해결하기 위해 객체로 모델링할 수 있는 것들을 도출하고, 객체들간의 상호작용을 통해 문제를 해결하는 프로그래밍 패러다임
  - 제공해야 하는 기능보다는 응용에서 필요로 하는 데이터에 보다 중점을 두는 방식

구조화 프로그래밍 (structured programming)

필요한 기능을 찾음
- 학생 검색 기능
- 성적 계산 기능
- 정렬 기능
- …

이런 기능들을 모듈화함 (함수로 구현)

객체지향 프로그래밍 (object-oriented programming)

객체로 모델링할 수 있는 것을 찾음 (데이터 중심)
- 학생
- 교수
- 과목
- …

그 다음 각 객체의 행위를 정의함 (클래스로 구현)

OOP vs. Structured Programming

```
typedef struct{
    int x;
    int y;
} Point;

void printPoint(Point *p){
    printf("(%d, %d)", p->x, p->y);
} /* printPoint */
```

```
void f(){
    Point x;
    ...
    printPoint(&x);
} /* f() */
```

구조화 프로그래밍에 비해 항상 인자가 하나 감소함

구조화 프로그래밍
- 함수(기능) 위주
- 데이터와 그것을 처리하는 함수가 분리
- 언어 문법적으로도 복잡

```
public class Point{
    private int x;
    private int y;
    public void print(){
        System.out.printf("(%d, %d)", x, y);
    } // print
} // class Point
```

```
void f(){
    Point x = new Point();
    ...
    x.print();
} /* f() */
```

객체지향 프로그래밍
- 데이터와 행위 결합
- 데이터 위주
- 실세계를 보다 가깝게 모델링

객체(Object)

- 객체는 행위(behavior), 상태(state), 식별자(identity)를 가짐
- 객체의 현재 상태에 따라 요청한 행위의 결과가 달라질 수 있음
  - 예) 문을 나타내는 객체의 현재 상태가 잠겨있는 상태이면 문을 열어라 하는 요청에 반응하지 않음
- 캡슐화(encapsulation): 데이터와 행위를 하나의 단위로 결합
  - 관리하기 편리한 작은 단위(small manageable unit)를 얻을 수 있으며, 이 단위는 재사용이 가능함
    - 확장, 변경이 용이하며, 그 파급효과를 지역적으로 제한 가능
  - 접근제어를 통해 외부에서 불필요한 요소에 대한 접근을 방지함
    - 프로그래밍 오류를 줄이는데 매우 효과적임
    - 기존 구조화 프로그래밍에서는 없던 기능
- 객체지향 프로그래밍에서는 프로그램의 목적을 달성하기 위해 필요한 객체들을 식별하고, 이들 간에 어떤 상호작용이 필요한지 설계하는 것이 가장 중요함

캡슐화 (1/2)

- 데이터의 형태를 숨기고 데이터의 접근을 위한 메소드를 제공하는 과정을 캡슐화(encapsulation)라 함
  - 다시 말하면, 데이터와 행위를 하나로 결합하고, 데이터 표현과 행위의 구현을 사용자로부터 숨기는 것을 말함
    - 예) 데이터 표현의 숨김: 내부적으로 사용자 이름을 하나의 문자열에 유지할 수 있고 성과 이름을 나누어 유지할 수 있음. 이 클래스의 객체를 사용하는 측에서는 이것을 알 필요가 없으며, 나중에 바뀌더라도 사용 방법에 변화가 없어야 함
    - 예) 행위 구현의 숨김: 내부적으로 어떤 알고리즘을 사용하는지 사용하는 측에서는 알 필요가 없음. 나중에 바뀌더라도 인터페이스만 유지하면 됨
  - 캡슐화는 정해지지 않은 방법으로 데이터가 조작되어 객체가 잘못된 상태로 진입하는 것을 방지해줌
    - 즉, 프로그램의 오류를 줄여주는 효과가 있음
    - 이를 위해 private, public과 같은 접근 권한 수식어를 사용함

기존 구조화 프로그래밍과 비교되는 중요한 차이점

캡슐화 (2/2)

- 요구사항 변화에 유연한 코드 제공

```
public class Member{
        private Date expiryDate;
        private boolean isMale;
        ...
        public Date getExpiryDate(){
                return expiryDate;
        }
}
```

```
if(member.getExpiryDate().getDate()
        < System.currentTimeMillis()){
        ...
}
```

```
public class Member{
        private Date expiryDate;
        private boolean isMale;
        ...
        public boolean hasExpired(){
                ...
        }
}
```

```
if(member.hasExpired()){
        ...
}
```

클래스(class)

- 클래스는 객체의 모습을 프로그래밍 환경에서 정의하는 틀임

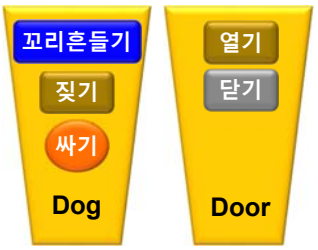
  - 클래스는 멤버변수(instance variable, field, attribute)와 메소드(멤버함수)로 구성됨
  - 멤버변수는 객체의 상태를 모델링하며, 메소드는 객체의 행위를 모델링함
  - 멤버변수와 메소드에 대해서는 외부의 접근을 제한할 수 있음
- 한 클래스가 정의되면 이를 이용하여 같은 종류의 객체를 필요한 만큼 생성하여 사용할 수 있음
  - 인스턴스(instance): 클래스의 한 객체를 말함

객체의 동작

- 프로그래밍 측면에서 객체는 프로그램이 실행되었을 때 수명을 가지는 것인 반면에 클래스는 객체가 어떤 상태를 가지고 어떤 행위를 할 수 있는지 정의하는 틀(template)일 뿐임
- 객체는 생성되어야 사용이 가능함
- 같은 종류의 객체에 대해 같은 메소드가 호출되더라도 해당 객체의 현재 상태에 따라 결과가 다를 수 있음
  - 객체의 메소드는 객체의 현재 상태를 이용해야 함
  - 객체의 상태를 이용할 필요가 없으면 해당 메소드는 객체 메소드로 적합하지 않을 수 있음
- 자바는 객체를 조작하기 위해 객체참조변수를 사용함
  - 자바의 모든 타입은 원시타입 아니면 참조타입
  - 자바는 쓰레기 수집 기능이 있어 동적 생성에 대한 부담이 없음

객체참조변수

- 특정 객체에 대한 리모컨(참조변수)을 만들면 같은 종류의 객체를 조작하기 위해서만 사용 가능
  - TV 리모컨으로 비데 리모컨을 조작할 수 없음
    - 설치되어 있는 버튼이 다름
  - 하지만 하나의 TV 리모컨으로는 다양한 TV를 조작할 수 있음
    - 물론, 특정 TV만 조작하도록 고정시킬 수도 있음 (상수포인터)



private vs. public (1/2)

- 멤버(변수, 함수)가 private으로 선언되어 있으면 외부에서 절대로 직접 접근할 수 없음. 반대로 public으로 선언되어 있으면 외부에서 직접 접근할 수 있음
- 멤버 변수가 private으로 선언되어 있으면 직접 접근할 수 없으므로 그것의 값을 외부로부터 숨기는 것이 목적이라고 생각할 수 있음. 즉, 보안에서 비밀성과 같은 개념으로 오해할 수 있음
- private으로 선언하는 것은 멤버 변수의 현재 값을 숨기고자 하는 것이 아니라 변수의 값을 임의로 조작하지 못하도록 하기 위함임
  - 이것은 객체가 잘못된 상태로 진입하는 것을 방지하여 프로그램의 오류가 발생하는 것을 줄이기 위함임
  - 예)

```
Student s = new Student("홍길동");
s.age = -1;
s.setAge(-1);
```

```
public void setAge(int a){
    if(a>0) age = a;
}
public int getAge(){
    return age;
}
```

private vs. public (2/2)

- 클래스의 멤버변수, 특히 인스턴스 변수(instance variable)는 private으로 지정하여 외부에서 정의되어 있지 않은 방법으로 조작할 수 없도록 해야 함 (오류를 줄이기 위한 목적)
- 클래스의 메소드는 보통 외부와 상호작용하기 위해 제공되는 수단이므로 대부분 public으로 지정됨
  - 만약 외부와 상호작용하기 위한 수단이 아니고, 내부적으로만 필요한 메소드일 경우에는 private으로 지정하는 것이 바람직함
  - 즉, 메소드도 public일 필요가 없으면 항상 private으로 지정함
  - 반대로 멤버변수도 문제가 없으면 public을 사용함
    - 예) 상수는 외부에 수정이 가능하지 않기 때문에 접근자 메소드보다는 직접 접근하도록 해주는 것이 바람직함

강아지 예 (1/3)

- 자바를 이용한 간단한 객체지향 프로그래밍의 예
  - 강아지는 어떤 상태를 가지고 있어야 하나?
    - 예) 이름, 종
  - 강아지는 어떤 행위를 해야 하나?
    - 예) 짖기, 먹기, 싸기
  - 이 둘 모두 개발하고자 하는 응용(예: 강아지를 판매하는 쇼핑몰, 강아지 캐릭터가 등장하는 게임)에 따라 다름

```
public class Dog{
        private String name;
        public void setName(String n){
                name = n;
        }
        public void bark(){
                System.out.printf("%s: 멍멍\n", name);
        } // bark()
} // class Dog
```

```
public class DogTest{
        public static void
                main(String[] args){
                Dog d = new Dog();
                d.setName("미미");
                d.bark();
        } // main
} // class DogTest
```

강아지 예 (2/3)

```
public class Dog{
        private String name;
        private String breed;
        public void setName(String n){
                name = n;
        }
        public void setBreed(String b){
                breed = b;
        }
        public void bark(){
                System.out.printf("%s: ", name);
                if(breed.equals("똥개"))
                        System.out.println("멍멍");
                else System.out.println("왈왈");
        } // bark()
} // class Dog
```

- 보통 경우에 따라 다르게 행동해야 하면 프로그래밍에서는 조건문을 사용함

```
public class DogTest{
        public static void main(String[] args){
                Dog d = new Dog();
                d.setName("미미");
                d.setBreed("시추");
                d.bark();
        } // main
} // class DogTest
```


강아지 예 (3/3)

- 상속: 조건식 제거, 확장성 제공, 코드 재사용성

```
public class Dog{
        private String name;
        public void setName(String n){
                name = n;
        }
        public void bark(){
                System.out.printf("%s이 ", name);
        } // bark()
} // class Dog
```

```
public class ShihTzu extends Dog{
        public void bark(){
                super.bark();
                System.out.println("왈왈");
        } // bark()
} // class ShihTzu
```

```
public class DogTest{
        public static void main(String[] args){
                Dog d = new ShihTzu();
                d.setName("미미");
                d.bark();
        } // main
} // class DogTest
```

의자 전쟁 (1/4)

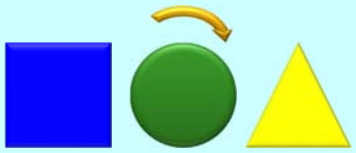
- 소프트웨어회사 팀장이 두 프로그래머(철수, 미미)의 실력을 테스트하기 위해 똑같은 스팩을 주고 프로그램을 만들도록 하였음
  - 경품으로 최고급 의자를…

구조화 프로그래밍으로 구현

```
rotate(shapeNum){
   // 도형을 회전함
}
playSound(shapeNum){
   // 해당 도형 WAV 사운드 파일을 재생함
}
```

객체지향 프로그래밍으로 구현
• 각 도형마다 하나의 클래스 작성

Square	Circle	Triangle
rotate() playSound()	rotate() playSound()	rotate() playSound()



GUI에 정사각형, 원, 삼각형이 있음. 사용자가 어떤 도형을 클릭하면 그 도형을 시계방향으로 360도 회전시키고 각 도형별로 WAV 사운드를 재생해야 함

- 구조화 프로그래밍: 조건문 필요
- 객체지향 프로그래밍
  - 조건문은 필요 없지만 코드가 중복됨 (실제는 아님)
  - 도형별로 독립적으로 수정 가능

의자 전쟁 (2/4)

- 둘 다 완성하였지만 팀장이 스팩을 수정하였음

구조화 프로그래밍으로 구현

```
playSound(shapeNum){
   // 도형이 아메바 모양이면 MP3
   // 나머지 도형은 해당 WAV
   // 사운드 파일을 재생함
} // 조건문으로 구현됨 (코드 변경)
```

객체지향 프로그래밍으로 구현
• 아메바 도형을 위한 클래스 추가
• 나머지 클래스는 코드 수정 없음

Amoeba
rotate() playSound()

기존 세 가지 도형에 아메바 모양의 도형을 추가해야 함. 기존과 달리 WAV 파일이 아니고 MP3 파일을 재생하여야 함

의자 전쟁 (3/4)

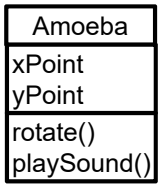
- 두 프로그래머는 모두 팀장이 정확한 스팩을 주지 않아 팀장이 원하는 대로 프로그램 결과를 제출하지 못함
  - 아메바 도형의 경우에는 회전 중심이 기존 도형과 다름

구조화 프로그래밍으로 구현

```
rotate(shapeNum, xPt, yPt){
  // 도형이 아메바 모양이면
  // 주어진 점을 중심으로 회전
  // 나머지 도형은 도형의 중심을
  // 계산하여 회전
} // 조건문으로 구현됨 (코드 변경)
```

객체지향 프로그래밍으로 구현

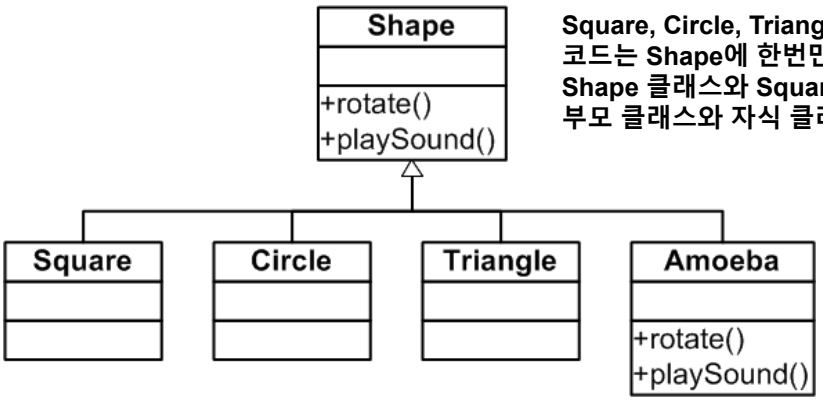
• 아메바 도형만 수정함



- 미미가 의자를 받았지만 철수는 동의하지 못함
  - 철수의 주장. 코드가 중복됐잖아!!!

의자 전쟁 (4/4)

- 미미가 실제 제출한 코드 형태는 상속 기반



Square, Circle, Triangle의 공통된 코드는 Shape에 한번만 정의됨
Shape 클래스와 Square 클래스는 부모 클래스와 자식 클래스 관계임

Amoeba는 기존 다른 도형과 다르게 동작하므로 메소드를 재정의함

클래스 선택하기 (1/2)

- 클래스의 종류
  - 종류1. 단일 개념을 나타내는 클래스
    - 예) 수학적 개념: Point, Rectangle, Ellipse
    - 예) 일상 개체: BankAccount, Dog, Student
  - 종류 2. Actor: 어떤 작업을 해주는 클래스
    - 예) Scanner, StringTokenizer, Random
  - 종류 3. 유틸리티 클래스: 객체를 전혀 생성하지 않는 클래스
    - 모든 메소드가 static 메소드임
    - 예) Math 클래스
  - 종류 4. 프로그램을 시작하기 위해 만든 클래스: main 메소드를 포함하는 클래스
    - 종류 3과 4는 실제 클래스로 보기 어려움

> - 앞으로 할 일이 이전에 수행한 일에 영향 받을 경우 이것은 객체로 모델링되어야 함
>   - 상태가 그 정보를 유지

클래스 선택하기 (2/2)

- 문제 정의, 요구사항 분석, 시나리오 분석 등에서 명사와 동사를 분석함
  - 명사는 클래스 또는 클래스의 멤버변수
  - 동사는 클래스 메소드
- 클래스는 높은 응집성(high cohesion), 낮은 결합성(low coupling)을 가져야 함
  - 중요한 소프트웨어공학 개념
  - 응집성: 단일 개념을 나타내야 한다는 것을 말함
  - 결합성: 다른 클래스와 관계를 말하는 것으로 낮은 결합성이란 서로에게 영향을 주는 것이 적다는 것을 말함
    - 결합성이 높은 두 클래스 중 한 클래스를 변경하게 되면 다른 클래스도 함께 변경해야 할 확률이 높음

클래스 수준 관계

- 상속: 기존 클래스를 이용하여 새 클래스를 정의하는 경우
  - 예) 주문시스템: 긴급주문은 일반 주문 클래스를 상속하여 정의함
  - 상속은 코드의 재사용을 가능하게 해줌
    - 긴급주문은 일반 주문이 가지고 있는 공통적인 상태와 행위는 다시 정의하지 않고 추가적으로 가지고 있어야 하는 상태와 기존 행위의 필요한 변화나 새 행위만 추가로 정의함
  - GUI의 경우 윈도우, 메뉴와 같은 요소가 필요하며, 이것을 무에서 새롭게 작성하고자 할 경우에는 힘듦. 하지만 이미 작성된 것을 바탕으로 그대로 사용할 수 있는 부분들은 모두 상속받고 내가 바꾸고 싶은 부분만 재정의하거나 추가하고 싶은 것을 새로 정의하여 사용할 수 있으면 매우 편리함
- 구체화: 한 클래스가 특정 인터페이스를 구현한 경우
  - 구체화는 같은 종류의 메소드를 제공하는 객체들을 그룹핑하여 줄 수 있음
  - 자바에서는 interface라는 개념을 통해 제공함

인스턴스 수준 관계 (1/2)

- 사용("uses-a"): 한 클래스가 다른 클래스를 사용하는 경우
  - 가장 일반적인 관계
  - 멤버 변수가 아닌 다른 클래스의 객체를 메소드의 인자로 받아 사용하거나 메소드 내에서 다른 클래스의 객체를 생성하여 사용하는 경우

```
public class Child {
    public void wash(Towel towel){
    }
}
// Child 클래스는 Towel을 사용
```

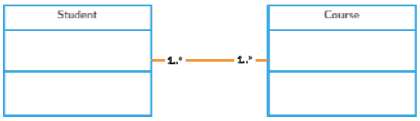
```
public class Camera {
    public Picture takePicture(){
        Picture picture = new Picture();
        ...
        return picture;
    }
}
// Camera 클래스는 Picture를 사용
```

인스턴스 수준 관계 (2/2)

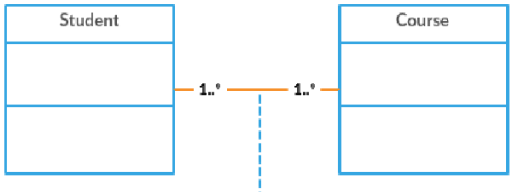
- 포함("has-a"): 한 클래스의 객체가 다른 클래스의 객체를 포함하고 있는 경우
  - 멤버 변수의 타입이 다른 클래스인 경우
  - 예) 주문에는 여러 상품이 포함됨
  - 종류: 연관(association), 집합(aggregation), 복합(composition)
  - 연관: 두 클래스의 객체 간에 관계가 형성되지만 그것이 부분-전체와 같은 관계가 아님
  - 집합/복합: 부분-전체 관계로서 복합이 보다 강한 개념. 복합의 경우에는 전체가 제거되면 부분도 함께 제거되어야 하는 경우, 집합은 전체가 제거되어도 부분은 남아 있는 경우로 구분됨
  - 연관(association): 예) 개인과 구독잡지
  - 집합(aggregation): 예) 연못과 오리
  - 복합(composition): 예) 빌딩과 방
  - 이들을 확실히 구분하는 것은 쉽지 않으며, 응용에 따라 다를 수 있음

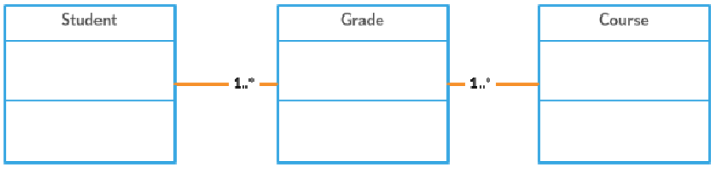
연관 클래스

- 두 클래스 간의 연관 관계가 복잡할 경우(다대다 관계) 연관 클래스를 활용 가능
  - 예) 학생과 수강 교과
    - 학생은 여러 개 교과를 수강할 수 있음
      - 한 교과도 여러 번 수강할 수 있음
    - 학 교과에는 여러 학생이 수강할 수 있음



  - 보통 연관 클래스의 구현은 어떻게?





집합/복합

- 구현 측면에서
  - 집합은 객체를 받아 내부 멤버를 초기화하고
  - 복합은 객체 내부에서 내부 멤버를 생성함

```
public class Computer {
    private CPU cpu;
    ...
    public Computer(String cpuType, ...){
        this.cpu = CPU.getInstance(cpuType);
    }
}
```

```
public class Computer {
    private CPU cpu;
    ...
    public Computer(CPU cpu, ...){
        this.cpu = cpu;
    }
}
```


상속과 인터페이스 비교 (1/2)

- 상속은 코드 재사용을 가능하게 하지만 인터페이스는 가능하지 않음
  - 상속은 멤버 변수 및 메소드의 공유
  - 인터페이스는 메소드 이름만 공유
    - 하지만 자바8부터는 인터페이스도 기본 메소드 제공 가능
- 상속은 논리적으로 관련 있는 것들을 그룹핑하여 주지만 인터페이스는 서로 논리적으로 관계가 없는 것들을 그룹핑하여 줌
  - 즉, 논리적으로 관계(is-a 관계)가 없는 것들을 코드 재사용 목적을 위해 상속 관계로 모델링하는 것은 바람직하지 않음
- 상속은 하나의 부모로부터 상속받을 수 있지만 한 클래스는 여러 개의 인터페이스를 구현할 수 있음

상속과 인터페이스 비교 (2/2)

- 상속과 인터페이스의 공통점
  - 여러 타입을 동일 리모콘을 사용하여 사용할 수 있게 해줌
    - 예) 상속에서 Pet 리모콘으로 Dog, Cat 객체 처리, interface에서 Flyable 리모콘으로 Airplane, Bee 객체 처리
  - 즉, 상속을 하여 자식 클래스를 만들 경우, 또는 특정 interface를 구현하는 클래스를 만들 경우, 해당 클래스의 객체는 해당 상위 타입을 이용하는 곳에 사용될 수 있어야 함
  - 문법적으로는 상위 리모콘을 이용하여 처리하는데 아무 문제가 없지만 논리적으로는 하위 클래스의 구현 결과에 따라 문제가 발생할 수 있음

자바 8 default method

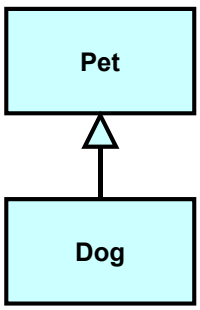
- 자바 8부터는 interface에 static 메소드를 정의할 수 있고, 기본 메소드를 정의할 수 있음
- 기본 메소드는 객체의 상태를 접근할 수 없으므로 한계가 분명히 있음
- C++에 존재한 다중 상속 문제(diamond problem)가 있음
  - 한 클래스는 interface를 여러 개 구현할 수 있음
  - 한 클래스가 동일한 기본 메소드를 가진 여러 개 interface를 구현하면?
    - 원칙 1. 부모클래스 우선
    - 원칙 2. 오류. 재정의를 통해 충돌을 해결할 수 있음
- 기본 메소드를 통해 얻어지는 장점
  - 기존에 interface를 변경하면 그것을 구현한 모든 클래스(상속과 달리 그 수가 매우 클 수 있음)를 무조건 수정해야 함
  - 기본 메소드를 사용하면 interface를 변경하더라도 기존 클래스을 수정 없이 사용할 수 있음 (물론 한계는 있음)

Why 인터페이스?

- 코드 공유도 가능하지 않는데 어떤 이유로 논리적으로 서로 관계가 없는 것들을 그룹핑할까?
  - 범용 프로그래밍과 다형성
    - 서로 논리적으로 관련 없는 것들을 하나의 코드에서 처리 가능
      - 상속처럼 interface도 해당 타입을 이용하여 해당 interface를 구현한 객체를 유지할 수 있음
      - 예) void simulate(Flyable f)라는 메소드가 있으면 이 메소드에는 Flyable 인터페이스를 구현한 모든 클래스 타입의 객체를 전달할 수 있음
    - Plug-and-play: 코드 수정 없이 호환되는 다른 객체 사용 가능
      - 상속, 인터페이스가 가지고 있는 공통 특성
      - 메소드 이름을 강제로 고정시켜 확장성 있는(수정이 쉬운) 코드 작성을 쉽게 해줌
        - 예) GUI 프로그래밍에서 버튼을 눌렀을 때 반응하는 메소드

상속 개요 (1/3)

- 상속은 기존 클래스를 이용하여 새 클래스를 만들 수 있게 해주며, 이를 통해 기존 코드를 재사용(반복하여 정의하지 않고)할 수 있음
- 상속계층도(inheritance hierarchy): 클래스 간에 상속 관계를 나타내는 도표 (UML에서 사용하는 표기법)
  - 옆 계층도에서 Dog은 Pet의 서브/자식/파생(sub/child/derived) 클래스라 하며, Pet은 Dog의 슈퍼/부모/기저(super/parent/base) 클래스라 함 (is-a 관계)
  - 상속할 때 자바에서 extends라는 키워드를 사용함



- 자식 클래스의 객체가 부모 클래스의 객체보다 큼
  - 자식 클래스의 객체는 자신만의 멤버변수뿐만 아니라 부모 클래스의 모든 멤버변수를 가지고 있음
- 자식 클래스의 객체가 부모 클래스의 객체보다 많은 기능을 가짐
  - 자식 클래스의 객체는 자신만의 공개 메소드를 제공할 뿐만 아니라 부모 클래스의 공개 메소드를 모두 제공함

상속 개요 (2/3)

- 상속을 통해 얻어지는 이점
  - 코드의 재사용이 가능함. 공통된 코드를 한 곳(상위 클래스)에 구현가능
  - 상위 타입을 통해 코드의 확장성을 높일 수 있음

```
public class People {
    private Dog dog;
    // private ShihTzu dog;
    public void setDog(Dog dog){
        this.dog = dog;
    }
}
```

```
public class DogTest{
    public static void main(String[] args){
        People p = new People();
        p.setDog(new ShihTzu("미미"));
        …
        p.setDog(new Maltiese("뽀송"));
    } // main
} // class DogTest
```

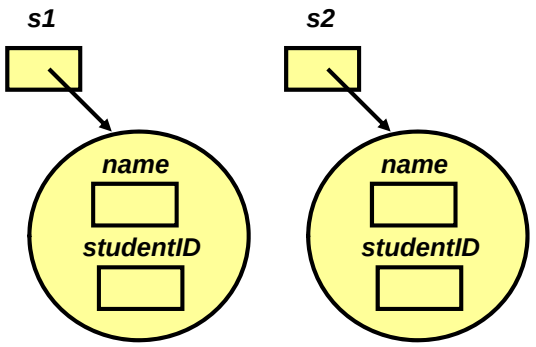
  - 이 예처럼 강아지가 바뀌어도 People 클래스의 수정은 불필요

상속 개요(3/3)

```
public class People {
    private String name;
    …
}
```

```
public class Student extends People {
    private String studentID;
    …
}
```

```
Student s1 = new Student();
Student s2 = new Student();
```



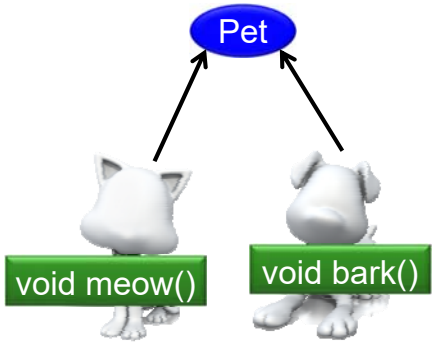
- 부모 클래스에 정의된 멤버는 자식 클래스 객체들이 공유하는 것이 아님
- 자식 객체를 생성하면 부모 객체들이 생성되는 것이 아님
- 자식 클래스는 부모 클래스를 포함하고 있는 개념

상속 구현

- 자식 클래스에 메소드를 정의할 때 가능한 시나리오
  - 경우 1. 부모 클래스의 메소드를 재정의(override) 함
  - 경우 2. 부모 클래스의 메소드를 상속 받음
    - 메소드를 오버라이드하지 않는 이상 접근 권한이 있는 모든 메소드(생성자 제외)는 자동으로 상속됨
  - 경우 3. 부모 클래스에 없는 새 메소드를 정의함
- 자식 클래스에 멤버변수를 정의할 때 가능한 시나리오
  - 경우 1. 부모 클래스의 멤버변수를 상속 받음
    - 부모 클래스의 모든 멤버변수는 자동으로 상속 받음
  - 경우 2. 새 멤버변수를 정의함
    - 메소드와 달리 부모 클래스에 있는 멤버변수와 같은 이름의 멤버변수를 정의하면 같은 이름의 멤버변수가 두 개 존재하게 됨. 이것은 여러 가지 부작용을 초래할 수 있음

상속 구현: 메소드

- 메소드를 재정의할 경우 보통 부모에 정의된 메소드를 활용하는 경우가 많음. 즉, 부모에 정의된 메소드를 호출하여 기본 작업을 한 후에 필요한 추가 작업을 하는 경우가 많이 있음
  - 이 때 super 키워드를 사용함
- 부모에 없는 새 메소드를 많이 정의해야 하면 상위 타입을 활용하기 힘들기 때문에 이 경우도 바람직하지 않을 수 있음




```
public static void main(String[] args){
        Pet p = new Dog();
        p.bark(); // error
} // main
```

- 호출할 수 있는 메소드는 객체참조변수 타입에 의해 결정
- 호출되는 메소드는 참조변수가 실제로 가리키는 객체 타입에 의해 결정

추상 클래스

- 상속 계층도에서 단말노드에 해당하지 않는 클래스들은 보통 해당 클래스의 객체를 생성할 필요가 없음
- 어떤 클래스의 객체를 생성할 필요가 없는 경우에는 해당 클래스를 추상 클래스(abstract class)라 함
- 상위 클래스를 정의할 때 다형성을 위해 메소드의 정의가 필요하지만 공유할 코드가 없는 경우가 있음. 이 경우 해당 메소드를 선언만 하게 되며, 이와 같은 메소드를 추상 메소드라 함
  - 추상 클래스와 추상 메소드는 모두 abstract이라는 수식어를 사용하여 정의 및 선언됨
- 추상 메소드를 가지는 클래스는 반드시 추상 클래스가 되어야 함
- 추상 메소드가 없어도 객체를 생성할 필요가 없으면 추상 클래스로 지정할 필요가 있음

상속 예: 문 예제 (1/7)

- 문의 상태: 문이 열려 있는지 닫혀 있는지?
- 문의 행위: 열기, 닫기

```
public class Door {
   private boolean isOpen = false;
   public Door() {}
   public void open(){
      if(isOpen) System.out.println("문이 열려 있음");
      else{
         System.out.println("문이 열림");
         isOpen = true;
      }
   }
   public void close(){
      if(isOpen){
         System.out.println("문이 닫힘");
         isOpen = false;
      }
      else System.out.println("문이 이미 닫혀 있음");
   }
}
```

왜 명백한 초기화를 했을까?

상속 예: 문 예제 (2/7)

- 잠금 장치가 있는 문을 만들고 싶음
- 이미 문 클래스가 정의되어 있는 상태?
  - 대안 1. 문 클래스를 수정함
  - 대안 2. 문 클래스를 상속받아 잠금 장치가 있는 문을 정의함
- 상속받는 것으로 구현하여 보자
  - 추가되어야 하는 상태: 잠겨있는지 여부
  - 추가되어야 하는 행위: 잠그고 잠금을 해제할 수 있어야 함

```
public static void main(String[] args){
        Door d = new Door();
        d.open();
        d.open();
        d.close();
        d.open();
} // main(): 테스트 프로그램
```

상속 예: 문 예제 (3/7)

```
public class DoorWithLock extends Door {
   private boolean isLocked = false;
   public DoorWithLock() { super(); }
   public void lock(){
      if(isLocked) System.out.println("이미 잠겨 있음");
      else{
         System.out.println("문을 잠금");
         isLocked = true;
      }
   }
   public void unlock(){
      if(isLocked){
         System.out.println("잠금을 해제함");
         isLocked = false;
      }
      else System.out.println("문이 잠겨 있지 않음");
   }
}
```

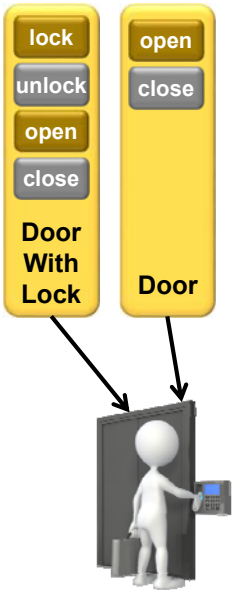
부모 생성자를 호출, 부모로부터 상속받은 멤버가 부모 생성자를 이용하여 초기화하여야 함

```
public static void main(String[] args){
        Door d = new DoorWithLock();
        d.open();
        d.open();
        d.close();
        d.lock(); // error
        d.open();
} // main(): 테스트 프로그램
```

Door 타입의 리모컨을 이용할 경우에는 lock 메소드를 사용할 수 없음

상속 예: 문 예제 (4/7)

- 여러 종류의 리모컨으로 한 객체를 제어할 수 있음
  - 조상 클래스 타입 또는 해당 객체가 구현하고 있는 인터페이스 타입의 리모콘으로 객체를 제어할 수 있음
  - 현재 객체를 가리키는 리모컨에 따라 할 수 있는 행위가 결정됨
  - 하지만 실제 호출되는 메소드는 리모컨이 실제 가리키는 객체에 의해 결정됨
- 따라서 앞 슬라이드 d 변수를 이용해서는 lock 메소드를 호출할 수 없음
  - 이것을 해결하기 위해 강제 타입 변환을 할 수 있음
    - 예) ((DoorWithLock)d).lock();
- 자식 클래스에 부모 클래스에 없는 메소드를 새로 추가할 경우 위와 같은 불편함이 생기며, 범용 메소드를 만들기 힘듦



상속 예: 문 예제 (5/7)

- 강제 타입 변환하여 테스트 프로그램 수행하면 논리 오류를 발견할 수 있음
- DoorWithLock은 open, lock을 재정의하여야 하며, 잠금도 문이 열려있는지 여부를 알아야 함

Door 클래스에 isOpen을 반환하여 주는 testOpen 메소드를 추가하여야 함

```
public class DoorWithLock extends Door {
   ...
   public void lock(){
      if(testOpened()) System.out.println("문이 열려 있어 잠글 수 없음");
      else if(isLocked) System.out.println("이미 잠겨 있음");
      else{
         System.out.println("문을 잠금");
         isLocked = true;
      }
   }
   ...
   public void open(){
      if(isLocked) System.out.println("잠겨있어 열 수 없음");
      else super.open();
   }
}
```

상속 예: 문 예제 (6/7)

- 이렇게 모델링하는 것이 올바른 것인가?
  - 이 경우 Door 객체도 생성하고 DoorWithLock 객체도 생성하겠다는 의미
  - 보통 클래스 계층도에서 단말만 보통 객체를 생성함
  - 따라서 이와 같이 모델링하는 것은 적절하지 않을 수 있음
- 클래스의 객체를 생성하지 못하도록 만들고 싶으면 abstract 키워드를 이용하여 추상 클래스로 정의할 수 있음
  - 추상 클래스는 추상 메소드를 가질 수도 있음
  - 보통 상속 계층도에서 단말이 아닌 클래스는 추상 클래스가 되는 것이 바람직함 (추상 메소드를 가지고 있지 않아도)

상속 예: 문 예제 (7/7)

```
public abstract class Door {
    private boolean isOpen = false;
    public Door() {}
    public void open(){ ... }
    public void close(){ ... }
    public abstract void lock();
    public abstract void unlock();
}
```

약간의 편법

추상메소드

```
public class DoorWithoutLock extends Door{
    public DoorWithoutLock() { super() }
    public void lock() {}
    public void unlock() {}
}
```

```
public class DoorWithLock extends Door{
    public DoorWithLock() { super() }
    public void open() { ... }
    public void close() { ... }
    public void lock() { ... }
    public void unlock() { ... }
}
```

상속 사용 시 고려사항 (1/2)

- 상속은 코드 재사용만을 목적으로 사용하는 것은 바람직하지 않음
- 상속은 조상과 후손 간에 논리적인 관계(is-a 관계)가 성립할 경우에만 사용되어야 함 (강아지(자식)는 애완동물(부모)임)
- 상속은 클래스간에 정적인 관계(고정된 관계)가 맺어지는 것이므로 조상에 특정 기능이 있으면 후손은 그것의 필요 여부와 상관없이 상속되는 문제점이 있음
  - 물론 이 경우에는 중간 클래스를 추가하거나 빈 메소드나 예외 처리 등을 통해 해결할 수는 있지만 이와 같은 경우가 많으면 모델링이 잘 못된 것일 수도 있음
  - 상속 관계가 형성된 후에 상위 클래스가 수정되면 이것은 많은 하위 클래스에 영향을 주게 되는 문제점도 있음
- 형제 클래스 간에 차이가 메소드 수의 차이가 아니라 메소드의 내부 내용의 차이만 있는 경우가 제일 좋음
  - 즉, 부모 없는 새 메소드를 자식 클래스에 많이 작성해야 하는 경우는 바람직하지 않음

상속 사용 시 고려사항 (2/2)

- 상속 관계로 모델링하였을 경우에는 가급적으로 상위 타입을 통해 처리하는 것이 바람직함. 이것은 코드의 확장성과 수정용이성에 도움이 됨

```
Dog d = new Dog();
d.eat();
```

```
Pet p = new Dog();
p.eat();
```

- 보통 상속 계층도에서 단말 클래스를 제외한 클래스들은 객체를 생성하지 않음. 즉, 상위 클래스 타입은 주로 다양한 하위 타입 객체를 유지하기 위한 목적으로 사용됨
- 클래스 상속의 깊이가 너무 깊으면 클래스 개수가 불필요하게 많아지게 되는 문제점이 있음. 적당한 레벨의 상속 깊이를 사용할 필요가 있음
- 상속을 꼭 해야 하는 것은 아님. 객체지향 설계 원리 중 포함관계가 상속관계보다 좋다는 원리(favor composition over inheritance)도 있음

상속 관련 한 가지 점검사항

```
public class A{
        protected int x;
        public void f1(){
                f2();
        }
        public void f2(){
                ++x;
        }
}
```

```
public class B extends A{
        @Override
        public void f2(){
                ++x;
                ++x;
        }
}
```

- 위 클래스 정의의 문제점은?
  - Can we safely replace object of A type with B type?

interface 개요 (1/3)

- interface는 interface를 구현하는 클래스가 반드시 제공해야 하는 메소드의 목록을 말함
  - 서로 논리적으로 관계가 없지만 동일 메소드를 제공하고 타입들을 하나로 그룹핑하여 주는 역할을 함
- 예) fly 기능: flyable interface
  - 곤충과 비행기: 논리적인 상속 관계는 아님
    - 상속과 달리 interface에는 코드를 정의할 수 없으므로 코드 공유 목적으로 사용할 수 없음
    - 보통 이름은 같지만 내부 구현 방식이 다른 것들을 묶는 목적으로 사용되기 때문에 코드 공유 기능이 없어도 큰 문제가 아님

```
public interface Flyable{
        void fly();
} //  interface Flyable
```

interface 개요 (2/3)

- interface는 클래스와 유사하게 정의하지만
  - interface는 멤버 변수를 가질 수 없으며,
  - 메소드의 내용을 정의할 수 없음. 이렇게 내용을 정의할 수 없는 메소드를 추상 메소드(abstract method)라 함
    - interface의 메소드는 자동적으로 public abstract임
  - interface에 상수는 정의할 수 있음
    - interface의 상수는 자동으로 public static final임
  - 따라서 인스턴스를 생성할 수 없음 (추상 메소드를 가지는 클래스를 추상 클래스라 하며, 추상 클래스도 인스턴스를 생성할 수 없음)
- 하지만 interface와 추상 클래스는 서로 다른 것임
  - 100% 순수한 추상 클래스 (추상클래스는 차후 설명)
    - 다만, 계층구조에 상위에 있는 클래스일 수록 추상 클래스가 될 확률이 높음

interface 개요 (3/3)

- 예)
- 클래스는 특정한 interface를 구현할 수 있음
  - 중요. 구현할 수 있는 interface 수에 제한은 없음
    - 비고. 오직 하나의 클래스만 상속 받을 수 있음
  - interface를 구현하면 그 interface에 정의되어 있는 모든 메소드를 무조건 구현해야 함. 이 때 implements 키워드를 사용함

```
public interface Flyable{
    void fly();
} //  interface Flyable
```

```
public class Duck implements Flyable{
    public void fly(){
        ...
    }
} // class Duck
```

```
public class Bee implements Flyable{
    public void fly(){
        ...
    }
} // class Bee
```

```
public static void main(String[] args){
    Flyable[] list = new Flyable[3];
    list[0] = new Duck();
    list[1] = new Bee();
    list[2] = new Airplane();
    for(Flyable f: list){
        list.fly();
    }
} // main
```

자바와 객체지향

- Object 클래스의 존재
  - 클래스 계층도에서 항상 최상위에 위치한 클래스가 존재
  - 범용 프로그래밍 및 일관성 유지(예: equals 메소드)에 매우 유용함
- 모든 객체는 동적으로 생성됨
  - 항상 쓰레기수집가능힙에서만 유지됨
- 복사생성자 대신에 clone 메소드
  - C++는 일반 클래스 변수의 대입, call-by-value 형태로 객체의 전달이 가능하기 때문에 이를 처리하기 위해 필요
  - 하지만 자바의 모든 객체는 동적 생성이며 참조변수로 처리함
- 인터페이스 제공
  - 보다 유연한 설계가 가능함
- 패키지 접근 권한이 있으며, protected 권한이 이 권한과 연관됨
- 열거형, 배열 등도 모두 객체로 모델링됨