**17회차(1)**

package overriding;

public class Student {

private String name;

private int kor;

private int math;

private int eng;

// 상속 관계인 클래스까지 접근 가능 => protected : 상속 가능

protected int tot;

protected double ave;

// 기본 생성자

Student() {

};

Student(String name, int kor, int math, int eng) {

this.name = name;

this.kor = kor;

this.math = math;

this.eng = eng;

// 총점과 평균!

set\_tot();

set\_ave();

}

// tot랑 ave 구하는 메소드 만들기!

// 메소드에서 사용하려는 속성의 접근제한자가 protected이면 메소드도 동일하게 작성

protected void set\_tot() {

this.tot = this.kor + this.eng + this.math;

}

protected void set\_ave() {

this.ave = this.tot / 3.0;

}

// private 속성으로 선언되어 있기 때문에

// 자식 클래스에서 사용할 수 있도록 public 속성으로 getter와 setter를 선언해줘야 한다

public int get\_kor() {

return this.kor;

}

public int get\_eng() {

return this.eng;

}

public int get\_math() {

return this.math;

}

public String get\_name() {

return this.name;

}

public void set\_kor(int kor) {

this.kor = kor;

}

public void set\_eng(int eng) {

this.eng = eng;

}

public void set\_math(int math) {

this.math = math;

}

public void set\_name(String name) {

this.name = name;

}

void info() {

System.out.println("이름 : "+this.name);

System.out.println("국어 : " + this.kor);

System.out.println("영어 : " + this.eng);

System.out.println("수학 : " + this.math);

System.out.println("총점 : "+this.tot);

System.out.println("평균 : "+this.ave);

}

}

package overriding;

public class StudentEx extends Student{

// 과학 과목 추가

private int sci;

// 생성자 앞에 public으로 해야 접근이 가능하다

public StudentEx(String name, int kor, int eng, int math, int sci) {

// 이름과 국어, 영어, 수학 점수는 부모 클래스의 속성 -> super 이용

super(name, kor, eng, math);

this.sci = sci;

set\_tot();

set\_ave();

}

// => 메소드 오버라이딩 : set\_tot()와 set\_ave()

// 총점과 평균을 계산할 때 과학 점수를 포함할 수 있도록 한다!

protected void set\_tot() {

// 국어, 영어, 수학 점수는 자식 클래스에서는 접근 불가능 (private이라서!)

// this.tot = this.tot + this.eng + this.math;

// 봄 클래스에 정의된 getter를 이용해서 점수 새로 계산

this.tot = get\_kor() + get\_eng() + get\_math() + this.sci;

}

protected void set\_ave() {

// this.ave = this.tot / 3.0;

this.ave = this.tot / 4.0;

}

// info 메소드도 새로 오버라이딩

// 과학 점수가 출력이 안되므로!

void info() {

System.out.println("이름 : " + get\_name());

System.out.println("국어 : " + get\_kor());

System.out.println("영어 : " + get\_eng());

System.out.println("수학 : " + get\_math());

System.out.println("과학 : " + this.sci);

System.out.println("총점 : " + this.tot);

System.out.println("평균 : " + this.ave);

}

int get\_sci() {

return this.sci;

}

void set\_card(int sci) {

this.sci = sci;

}

package overriding;

public class Main {

public static void main(String[] args) {

// 자식 클래스에서는 접근이 가능

// protected : 패키지가 달라지면 접근이 불가능하다

StudentEx stu1 = new StudentEx("홍길동", 90, 80, 100, 70);

// 같은 패키지까지는 접근을 허용

stu1.info();

System.out.println(stu1.tot);

}

}

}

**17회차(2)**

\*지금까지 메소드의 다형성(오버로딩, 오버라이딩)에 대하여 배웠다.

(1)

- 동물을 모델로 해서 클래스를 설계

- 모든 동물을 나타내는 Animal 클래스를 정의한다.

- 기본적으로 동물의 이름을 속성으로 한다.

- 그리고 동물의 울음소리를 기능으로 한다.

- Dog

- Animal 클래스를 상속받아서 강아지 클래스를 정의

- 이제는 어떻게 우는지 울음소리를 정의할 수 있을 것이다.

- 강아지 클래스는 먹는 기능을 추가

\* - 이때 부모클래스(기반클래스)는 Animal, 자식클래스(파생클래스)는 Dog 클래스이다

- 이때 Animal 클래스와 Dog 클래스는 ""Is-a" 관계가 성립한다

- 모든 Dog는 Animal이다.

=> "업 캐스팅"

# 모든 Dog는 모두 Animal이라고 할 수 있기 때문에

# 객체의 타입이 Dog가 아닌 Animal이어도, Dog 객체를 표현할 수 있다.

<=> Cat와 Dog은 Animal의 하위 클래스이므로

Animal이라는 하나의 타입으로 Cat과 Dog를 모두 처리할 수 있다!!

public static void bark(Animal a) {

a.bark();

}

(2) 추상 클래스

- 추상 메소드를 포함하는 클래스

- 추상 메소드 : 메소드 내부가 정의/구현되지 않은 메소드

- 클래스가 추상 메소드를 하나라도 포함하고 있으면 추상 클래스이다

- 인터페이스는 추상 클래스의 한 종류

- 클래스 내의 모든 메소드가 추상 메소드이면 인터페이스!

- (전부 다 구현되어 있지 않아야 한다)

- 완성되지 않은 클래스(미완성 클래스)

- 그렇기 때문에 추상 클래스나 인터페이스는 객체화할 수 없다!

=> 반드시 상속을 통해서 추상 메소드를 완성해야지만 객체화할 수 있다.

package test01;

public class Animal {

// 모든 동물은 이름을 가지고 있을 것이다.

private String name;

// 기본 생성자를 만들어주자 -> 초기화

public Animal() {}

public Animal(String name) {

this.name = name;

}

// 동물의 울음소리를 정의하고 싶지만

// 어떤 동물인지 모르기 때문에 불가능하다.. => 추상 클래스로!

public void bark() {

// System.out.println("???"); 생략

}

public void info() {

System.out.println("이름 : " + this.name);

}

}

package test01;

public class Dog extends Animal {

public Dog(String name) {

super(name); // name 속성 유지

}

// 메소드 오버라이딩 -> 동물의 울음소리를 정의

public void bark() {

System.out.println("멍멍");

}

// 동물 클래스에는 없는 메소드를 추가적으로 정의

public void eat() {

System.out.println("뼈다귀");

}

}

package test01;

public class Cat extends Animal {

private String name;

public Cat() {}

public Cat(String name) {

super(name);

}

public void bark() {

System.out.println("야옹");

}

public void eat() {

System.out.println("생선");

}

}

package test01;

public class Main {

// Dog와 Cat 클래스의 bark 메소드를 실행하는 메소드를 가정

// Dog와 Cat을 처리할 수 있도록 각 객체별로 각각의 메소드가 필요 (따로 만들어주어야 한다)

public static void bark(Dog d) {

d.bark();

}

public static void bark(Cat c) {

c.bark();

}

// 하지만 다형성을 이용한다면?

// Cat와 Dog은 Animal의 하위 클래스이므로

// Animal이라는 하나의 타입으로 Cat과 Dog를 모두 처리할 수 있다!!

public static void bark(Animal a) {

a.bark();

}

public static void main(String[] args) {

// 강아지 객체 생성

// 일반적인 타입이 일치하는 형태

Dog dog1 = new Dog("리트리버");

dog1.info();

dog1.bark();

dog1.eat();

// 다형성 is-a에 의해서 성립

// "업 캐스팅" 이라고도 한다

// 모든 Dog는 모두 Animal이라고 할 수 있기 때문에

// => 객체의 타입이 Dog가 아닌 Animal이어도, Dog 객체를 표현할 수 있다.

Animal dog2 = new Dog("사모예드"); // Animal은 자신의 하위 객체를 다 표현할 수 있다!

// 그러나 반대는 성립하지 않는다. (모든 Animal은 Dog이다 (x))

// => Animal 타입의 객체는 Animal의 하위 클래스인 모든 객체를 표현할 수 있다 : 다형성

dog2.info();

dog2.bark();

// 자식 클래스에서 정의된 고유 클래스는 사용할 수 없다.

// dog2.eat(); <- 오류

// 객체의 타입에 따라 실행시킨다.

// 메소드를 호출해서 사용할 때는 하나의 메소드처럼 보인다 (<- 오버로딩의 특징)

// 실제로는 객체별로 처리할 수 있는 메소드를 전부 작성할 필요가 있다.

Dog dog3 = new Dog("웰시코기");

Cat c1 = new Cat("코숏");

bark(dog3);

bark(c1);

}

}

**17회차(3)**

<예제>

동물원 클래스를 가정한다.

- 호랑이가 있으면 닭을 먹이로 주고

- 호랑이는 걷거나 뛸 수 있고

- 원숭이가 들어오면 바나나를 먹이로 주고

- 원숭이는 걷거나 나무를 타거나

- 독수리가 있으면 지렁이를 먹이로 주고

- 독수리는 날 수 있습니다.

- 타조

package test02;

public class Animal {

String name;

public Animal() {

};

public Animal(String name) {

this.name = name;

}

}

// 실습이므로 인터페이스와 추상클래스를 같이 사용해보자

// 인터페이스를 도입해보자 (땅에 사는 동물)

interface Ground {

public String getAction(); // <- 추상 메소드 : 아무것도 구현되어 있지 않다.

// 인터페이스에는 추상 메소드만 정의된다. 일반 메소드 들어갈 수 없다.

}

// 날아다니는 동물은 추상 클래스로 정의해보자

abstract class Birds extends Animal {

public abstract String getAction();

// 추상 클래스에는 추상 메소드 외에도 다른 일반 클래스들도 정의될 수 있기 때문에

// 반드시 abstract 붙여줘야 한다

public void get\_name() {

System.out.println("너의 이름은 : " + this.name);

}

}

// 모든 음식은 Foodable로 통일

interface Foodable {

public void food();

}

// 인터페이스 경우에는 extends를 통해서 여러 개의 인터페이스를 상속받을 수 있다.

// 기본적으로 자바에서 클래스는 반드시 1개만 상속이 가능하다

// 그러나, 인터페이스의 경우에 한해서 여러 개의 인터페이스를 상속받을 수 있다.

interface GroundFoodable extends Ground, Foodable {

}

// 이제는 동물을 하나씩 정의해보자

// 먼저 땅에 사는 동물부터 해보자

// 땅에 사는 동물은 Animal 클래스와 Ground, Foodable 인터페이스를 상속받아서

// 추상 메소드를 재정의한다

// 이때, Ground와 Foodable을 각각 상속받아서 구현해도 되고,

// GroundFoodable 을 상속받아서 구현해도 된다

// 중요한 것은 상속받은 인터페이스의 추상 메소드들을 정확하게 전부 구현해줘야 한다.

// 자바에서 클래스의 상속은 extends

// 여기서 인터페이스에서 인터페이스를 상속받는 경우에는 extend

// 클래스에서 인터페이스를 상속받는 경우에는 implements

// 인터페이스의 상속은 implements

class Tiger extends Animal implements Ground, Foodable {

// Ground, Foodable 인터페이스를 상속받았으므로

// 메소드 오버라이딩~

// Ground의 getAction() 추상 메소드와

// Foodable의 Food() 추상 메소드를 구현

public String getAction() {

return "walk";

}

public void food() {

System.out.println("닭");

}

}

// 원숭이는 GrounFoodable을 상속받아서 정의

// 똑같이 Ground와 Foodable의 추상 메소드를 정의해주면 된다

class Monkey extends Animal implements GroundFoodable {

public String getAction() {

return "climb";

}

public void food() {

System.out.println("바나나");

}

}

// 독수리는 Birds를 상속 받아서 구현

// Birds 추상 클래스에서 getAction 추상 메소드가 있으므로

// foodable 인터페이스만 구현을 해주면 된다

class Eagle extends Birds implements Foodable {

public String getAction() {

return "fly";

}

public void food() {

System.out.println("지렁이");

}

}

class Tazo extends Birds implements Foodable {

public String getAction() {

return "run";

}

public void food() {

System.out.println("사료");

}

}

package test02;

// 동물원 클래스 정의

public class Zoo {

// 두 가지 메소드를 처리

// 동물에 따라서 액션과 먹이를 구분할 수 있도록 처리

// => 객체의 타입에 따라서 타입에 맞는 액션과 먹이를 출력하도록 하자

// Birds와 Ground는 다형성이 적용되지 않는다

// => 그렇기 때문에 오버로딩을 통해서 구분

public void action(Ground animal) { // 땅에 사는 모든 동물 : 호랑이, 원숭이

// 객체의 다형성에 의해서 모든 육상동물에 맞는 액션이 출력될 것이다

System.out.println(animal.getAction());

}

public void action(Birds animal) { // 새 : 타조, 독수리

// 객체의 다형성에 의해서 모든 조류에 맞는 먹이가 출력될 것이다

System.out.println(animal.getAction());

}

// 모든 먹이는 Foodable 타입으로 확인 가능 [=> 다형성]

public void food(Foodable animal) {

// 이렇게만 해주면 객체의 다형성에 의해서 타입에 맞는 먹이가 출력될 것이다

animal.food();

}

}

package test02;

public class Main {

public static void main(String[] args) {

Zoo zoo = new Zoo();

// 각각의 동물 객체를 만들어서 동물원에 넣어보자

Tiger tiger = new Tiger();

Monkey monkey = new Monkey();

Eagle eagle = new Eagle();

Tazo tazo = new Tazo();

zoo.action(tiger);

zoo.action(monkey);

zoo.action(eagle);

zoo.action(tazo);

zoo.food(tiger);

zoo.food(monkey);

zoo.food(eagle);

zoo.food(tazo);

}

}