**15회차(1)**

package test01;

public class Mobile {

// 객체변수

// 외부에서 접근하지 못하도록 앞에 private 을 붙여 은닉성을 보장해준다

private String name;

private String Telecom;

private int price;

// 객체가 생성될 때 속성들이 초기화될 수 있도록

// 생성자 는 객체가 생성될 때 속성값을 매개변수(parameter)로 전달 받아서 속성을 초기화

Mobile(String name, String Telecom, int price) { // 지역변수

this.name = name;

this.Telecom = Telecom;

this.price = price;

}

// 내부 속성을 참조할 필요가 있다면

// 메소드를 정의해서 우회참조할 수 있도록 방법을 제공

// 그러한 역할을 하는 메소드를 getter(내부 속성 가져옴) 혹은

// setter(내부 속성 직접 설정, 바꿈)라고 한다

// 일반적으로 메소드 이름 앞에 get 또는 set을 붙여주는 게 일반적인 표현 방식

String get\_name() { // name을 가져온다!

return this.name;

}

// name 속성의 값을 외부에서 수정할 수 있도록 하고 싶다면

// 해당 속성에 대해서 setter를 정의

// 매개변수로 name을 받아와서 속성을 수정하는 메소드 정의

void set\_name(String name) { // name을 매개변수로 받아와 수정할 수 있게 한다!

this.name = name;

}

void info() {

// 객체의 속성을 전부 출력할 수 있도록

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

System.out.println("통신사 : " + this.Telecom);

System.out.println("가격 : " + this.price);

}

}

package test01;

public class Main {

public static void main(String[] args) {

// Mobile.java 클래스 파일을 생성한 후 Main 클래스에서

// 객체(m1, m2)를 생성해서 확인

Mobile m1 = new Mobile("Iphone", "SKT", 100);

Mobile m2 = new Mobile("Galaxy", "KT", 90);

m1.info();

m2.info();

// 객체지향을 따르는 이유(객체지향의 특징)

// 속성에 직접 접근해서는 안 된다

// System.out.println(m1.name); // 옳지 않다, 캡슐화/은닉성에 위배된다

// Mobile.java 객체변수 앞에 private 붙인 후 실행시키면 오류 발생

// getter(get 메소드) 를 통해서 속성의 값을 가져올 수 있다

// 일반적으로 이렇게 속성의 값을 가져올 수 있는 메소드를 getter라고 한다

// 실행시키면 가능하다!

System.out.println(m1.get\_name());

// 이제 정의된 setter를 통해서 name 속성을 변경해줄 수 있다

m1.set\_name("아이폰");

m1.info(); // 속성이 바뀌어 출력되는 것을 볼 수 있다

}

}

// 만약 scanner 이용하면

// input = 표준입력;

// m1.set\_name("input값");

**15회차(2)**

package test02;

//Player.java : 게임 캐릭터 클래스

//- 속성 : 아이디, 레벨, 공격력, 체력

//- 기능 : 속성을 출력(info) + @(필요하다면), 생성자는 알아서

// 특징 1) 캐릭터 생성 시 레벨은 1, 공격력은 5, 체력 20

// 2) 레벨이 증가할 경우 공격력은 3, 체력은 10씩 증가

// 3) 모든 속성은 private 로

public class Player {

// 객체변수

private String ID;

private int level;

private int attack;

private int health;

// 생성자

/\*

\* Player(String ID, int level, int attack, int health) { this.ID = ID;

\* this.level = level; this.attack = attack; this.health = health; }

\*/

Player(String ID) {

this.ID = ID;

this.level = 1;

this.attack = 5;

this.health = 20;

}

// setter

void set\_level(int level) {

// 전달 받은 레벨이 현재 레벨보다 낮은 경우에는?

// 이러한 문제 때문에 속성에 우회접근할 필요가 있다

if(this.level >= level) {

System.out.println("잘못된 레벨 입력");

return; // 메소드를 바로 종료

}

// 전달 받은 레벨만큼 공격력과 체력을 보정

int gap = level - this.level; // 레벨이 얼마나 올랐는가?

// 레벨이 오른 만큼 공격력과 체력을 보정

this.level = level;

this.attack += (gap \* 3);

this.health += (gap \* 10);

}

// info 메소드 : 정보 출력

void info() {

System.out.println("아이디 : " + this.ID);

System.out.println("레벨 : " + this.level);

System.out.println("공격력 : " + this.attack);

System.out.println("체력 : " + this.health);

}

}

package test02;

// Q2. 게임 캐릭터를 모델로 해서 클래스를 작성해보자

// Main.java : main() 메소드를 가지고 있는 클래스

public class Main {

public static void main(String[] args) {

/\*Player a1 = new Player("ID1", 1, 5, 20);

Player a2 = new Player("ID2", 1, 5, 20);

\*/

Player a1 = new Player("ID1");

Player a2 = new Player("ID2");

a1.info();

a2.info();

// 5레벨이 올랐다고 가정

a1.set\_level(6);

a1.info();

// 레벨이 잘못 입력되는 경우

// 이렇게 객체의 속성에 직접 참조하는 것보다

// 메소드를 통해서 우회 참조할 수 있도록 하는 것이 더 안전하다

// 객체지향프로그래밍에서는 은닉성(캡슐화)이라고 한다

a1.set\_level(5);

}

}

**15회차(3)**

package test03;

//1. 휴대폰의 이름은 한 번 설정하면 절대 변경 불가

//2. 가격은 아무리 싸게 해도 40만원 미만은 불가능

//- 최소 가격이 40만원, 만약에 20만원으로 설정했다면 40만원으로 설정

//3. 통신사나 가격은 언제든지 변경 가능

public class Mobile {

private String name;

private String telecom;

private int price;

Mobile(String name, String telecom, int price) {

this.name = name;

// 이렇게 작성하는 것도 가능하다

// this.set\_name(name);

this.telecom = telecom;

// set\_price을 총해서 가격 설정 <- 이걸 작성하기 전에는 맨 처음 가격이 40만원 아래여도 그대로 출력된다

// 굳이 다시 체크할 필요는 없다

// this.price = price;

this.set\_price(price);

}

// getter

// public : 접근 상관없이 언제든지 메소드를 통해 값에 접근할 수 있다

public String get\_name() {

return this.name;

}

public String get\_telecom() {

return this.telecom;

}

public int get\_price() {

return this.price;

}

// setter

// 같은 클래스 내에서는 호출 가능

// < 휴대폰의 이름은 한 번 설정하면 절대 변경 불가 >

// 1. set\_name을 지운다

// 2. set\_name 앞에 private을 붙인다

private void set\_name(String name) {

this.name = name;

}

public void set\_telecom(String telecom) {

this.telecom = telecom;

}

public void set\_price(int price) {

if (price < 400000) {

System.out.println("휴대폰 최소가격은 400,000");

this.price = 400000; // 지역변수!

return;

}

this.price = price;

}

// member method

void info() {

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

System.out.println("통신사 : " + this.telecom);

System.out.println("가격 : " + this.price);

}

}

package test03;

import test03.Mobile;

// Q3. 휴대전화를 모델로 해서 클래스를 작성하는 실습

// 단순히 객체를 생성하지 않고, 안전성을 높이기 위해서 강제적으로 규칙을 넣어보자

public class Main {

public static void main(String[] args) {

Mobile p1 = new Mobile("Iphone", "SKT", 100000);

p1.info();

// 통신사를 변경

p1.set\_telecom("KT");

p1.info();

// 가격을 변경

p1.set\_price(200000);

p1.info();

p1.set\_price(500000);

p1.info();

// 휴대폰 이름은 변경할 수 없다

// 속성뿐 아니라 메소드에서도 동일하게 접근제어자를 설정

// p1.set\_name("Galaxy"); -> 오류

}

}

**15회차(4)**

< 상속(inherit) >

- 부모로부터 유산을 물려받는 것을 상속이라고 한다

- 클래스 간에도 상속이라는 관계가 성립할 수 있다

- 그래서 상속되어지는 클래스를 부모 클래스 (기반 클래스)

- 상속받는 클래스를 자식 클래스 (파생 클래스)

- 클래스의 재사용성을 더 극대화한 개념

- 잘 정의된 클래스가 있다면, 다시 정의할 필요 없이 재사용하자는 개념

- 물려받은 클래스를 더 확장해서 사용

- 클래스 간의 관계를 표현하는 하나의 방법

- 클래스가 독립적으로 동작하지 않고, 여러 개의 클래스가 서로 유기적으로 연결되어서 동작

- 클래스들의 관계는 2가지 형태로 표현

- 상속(is-a)

- 다른 클래스의 객체를 속성으로 갖는 경우(has-a)

- 예를 들면, 모든 학생은 사람이다. (참)

- 학생 is a 사람. (그러나 반대는 성립하지 않는다 : 모든 사람은 학생이다(거짓))

- 그렇기 때문에 상속관계를 정의할 때는 반드시 큰 개념에서 작은 개념으로 상속이 될 수 있도록 한다

- 상속의 기본적인 형태

class Child extends Parent

**15회차(5)**

package test04;

public class Parent {

// 멤버 메소드

public void method() {

// 부모 클래스임을 알 수 있는 출력문

System.out.println("부모 클래스");

}

}

package test04;

// Child는 Parent를 상속받는다

public class Child extends Parent {

// 이때 Child 클래스는 정의하지 않는다

}

package test04;

public class Main {

public static void main(String[] args) {

// Parent 객체는 생성하지 않는다

// Child 객체만 생성 (Child 클래스는 구현하지 않았습니다)

// Parent에서 정의된 메소드를 자식 클래스에서 사용 가능 + 속성도 사용 가능!

Child c = new Child();

c.method(); // -> 부모한테 물려받은 메소드를 그래도 쓸 수 있다

}

}