1. 객체지향언어

\* 객체지향언어의 특징

1) 코드의 재사용성

2) 코드 관리가 용이

3) 신뢰성이 높은 프로그램을 가능하게한다.

\* 클래스

1) 정의

객체에 대해 정의해놓은 것. 객체의 설계도와 같음

기능을 수행하기 위한 데이터와 함수의 결합(데이터 저장개념의 발달에서 봤을 때)

- 변수와 함수가 유기적으로 연결되어 작업이 간단하고 명료해짐

예) Time[] t=new Time[3]; //Time 클래스는 hour, minute, second가 각각 다른형으로 선언됨

시간, 분, 초에서 한계값들을 체크하여 그 이상 이하 값들이 들어오지 못하도록 막는 기능

\* 인스턴스

1) 인스턴스 생성과 사용

Tv t = new Tv(); //클래스명 변수명 = new 클래스명();

t라는 참조변수는 Tv인스턴스가 저장된 주소값을 갖는다

t를 통해 Tv 인스턴스에 접근할 수 있다

참조변수 타입은 인스턴스 타입과 일치해야 한다.

\* 변수가 선언된 위치에 따른 변수 종류

1) 클래스변수(공유변수)

static이 붙은 멤버변수

인스턴스를 생성하지 않고도 사용가능(클래스이름.클래스변수)

2) 인스턴스 변수

static이 안붙은 멤버변수

3) 지역변수

메서드, 생성자, 초기화 블럭 내, main안에 선언된 변수

선언된 블록이 끝나면 소멸되어 사용 못함

\* 메서드

1) 반복적으로 사용되는 코드를 줄이기 위해서 사용한다

2) 사용방법 : 리턴타입 메서드이름 (타입 변수명, ….) {……}

(리터타입은 return result;에서 result와 같은 타입이어야 한다)

3) 호출방법

- 같은 클래스 내의 메서드끼리는 참조변수 사용하지 않고도 호출 가능

- static 클래스 메서드는 같은 클래스 내의 인스턴스 메서드를 호출할 수 없다

\* JVM 메모리 구조

1) 메소드영역(Method Area)

프로그램 실행 중 어떤 클래스가 사용되면, JVM은 해당 클래스를 읽어(\*.class) 분석하여 이곳에

저장

클래스 변수들도 이 영역에 함께 생성

2) 힙영역(Heal Area)

프로그램 실행 중 생성되는 모든 인스턴스는 모두 이곳에 저장

인스턴스 변수들도 이곳에서 생성

3) 호출영역(call stack or execution stack)

메서드의 작업에 필요한 공간이다

호출스택에 호출된 메서드를 위한 메모리가 할당

메서드가 작업을 수행하는 동안 지역변수(매개변수 포함)들과 연산의 중간결과 등을 저장

메서드가 작업을 마치면 할당되었던 메모리 공간은 반환되어 비워진다

호출스텍의 맨위는 현재 실행중인 메서드이며 나머지는 대기상태에 있게 된다.

\* 참조형 매개변수

인스턴스의 주소값이 복사된다

예) Data d = new Data();

change(d);

\* 재귀호출

1) factorial

static long factorial(int n) {

return n==1? 1 : n \* factorial(n-1);

- static 메서드는 생성하지 않고 직접호출 할 수 있다

2) x의 n제곱까지의 합

static long power(int x, int n) {

return n==1? X : x\*power(n-1);

\* static 메서드와 인스턴스메서드

1) static 메서드(=클래스메서드)

인스턴스 변수를 사용하지 않거나 클래스 변수만을 사용하는 메서드

작업하는 모든 값들을 매개변수로 받는 경우

2) 인스턴스 메서드(static이 안붙은 메서드)

인스턴스 변수와 관련된 작업을하는 메서드

\* 클래스멤버와 인스턴스 멤버간의 참조와 호출

같은 클래스에 속한 멤버간에는 별도의 인스턴스를 생성하지 않고도 서로 참조 또는 호출이 가능

단 클래스멤버가 인스턴스 멤버를 참조 또는 호출하고자 하는 경우에는 인스턴스를 생성해야 한다.

예) static int cv2=iv; 이렇게하면 에러가 나온다는 말

static int cv2 = new memberCall().iv; //memberCall 클래스에 iv는 인스턴스변수로 선언된 경우

int result = new memberCall().instanceMethod1();

대신 참조변수를 사용하지 않았기 때문에 memberCall() 인스턴스는 더 이상 사용할 수 없다

보통은

MemberCall c = new memberCall();

int result = c.instanceMethod1();

2. 객체지향 프로그래밍

1) 상속

정의와 장점

기존의 클래스를 재사용하여 새로운 클래스를 작성하는 것

코드의 재사용성 높이고 중복을 제거하여 프로그래밍의 생산성과 유지보수에 크게기여

\*\* 조상클래스 변경되면 자손은 자동으로 영향받으나 자손은 바뀌어도 조상은 영향 없음

\*\* 생성자나 초기화 블록은 상속되지 않고 오직 멤버변수와 메서드만 상속된다

\*\* 전체 프로그램을 구성하는 클래스들을 면밀히 설계, 분석하여, 클래스간의 상속관계를

적절히 맺어주는 것이 객체지향 프로그래밍에서 가장 중요한 부분이다

자손클래스의 인스턴스를 생성하면 조상클래스 멤버들도 함께 생성된다

클래스간의 관계 - 포함관계(composite relationships)

상속 이외에 클래스를 재 사용하는 방법이다

한 클래스를 작성하는데 다른 클래스를 멤버변수로 정의하여 포함시키는 것

예제

class Point {

int x;

int y;

}

class Circle {

Point p = new Point();

int r;

}

상속할것인가? 포함관계로 할것인가?

원은 점을 가지고 있다 - 포함관계

스포츠카는 car이다 - 상속관계

2) 오버라이딩(Overriding)

조상클래스에서 상속받은 메서드의 내용을 변경하는 것

오버라이딩 조건

(1) 조상클래스 매서드와 이름이 같아야 한다

(2) " 매개변수 개수, 타입, 순서가 같아야 한다.

(3) " 리턴타입이 같아야 한다

(4) 접근제어자는 조상클래스의 매서드보다 좁은 범위로 변경할 수 없다

(5) 예외는 조상클래스보다 많이 선언할 수 없다

\* 조상클래스를 가리킬 때 super.x와 super() 사용

3) package와 import

package

서로 관계가 있는 클래스들(인터페이스를 포함할 수 있음)의 묶음

패키지는 물리적으로는 하나의 directory이다

패키지 선언문(package 패키지명;)은 반드시 소스파일에서 주석과 공백 제외한 첫 번째 문장

하나의 소스파일에 단 한번만 선언될 수 있음

해당소스파일에 포함된 모든 클래스나 인터페이스는 선언된 패키지에 속하게 된다

패키지명은 클래스명과 구분하기 쉽게 소문자로 하는 것이 원칙

컴파일시 -d옵션을 추가하여 컴파일

패키지의 루트를 환경변수의 클래스패스에서 포함(p206)

import

import문으로 사용하고자 하는 클래스의 패키지를 미리 명시해주면 소스코드에 사용되는

클래스이름에서 패키지명은 생략할 수 있다

예제

import java.util.\*;

import java.text.\*;

위의 두 문장을 import java.\* 이렇게 쓸 수 없다

4 제어자

접근제어자

public, protected, default, private

그외 제어자

static, final, abstract, native,

transient, syncronized, volatile, strictfp

1) static

멤버변수, 매서드, 초기화블럭에 사용

공용변수가되며 생성하지 않고도 매서드 사용가능

static메서드는 인스턴스 멤버들을 사용할 수 없다

2) final

변수에 사용하면 값을 변경할 수 없는 상수가 된다 .

메서드에 사용하면 오버라이딩이 안되며 클래스에 사용되면 상속(extend)할 수 없다

3) abstract

클래스에 붙으면 클래스 내에 추상메서드가 선언되었음의 의미

메서드에 붙으며 선언부만 작성하고 구현부는 작성하지 않는 추상메서드임을 알린다

4) 접근제어자

public ㅇ ㅇ ㅇ ㅇ 같은클래스 같은패키지 자손클래스 전체

protected ㅇ ㅇ ㅇ 같은패키지, 자손에서접근가능

default ㅇ ㅇ 같은패키지에서 접근가능

private ㅇ 같은 클래스에서만 접근가능

private 클래스는 상속이 불가능. 왜냐하면 자손클래스의 인스턴스를 생성할 때

조상클래스의 생성자를 호출해야만하는데 생성자의 접근제어자가 private이므로

자손클래스에서 호출이 불가능하기 때문이다. 이 경우 클래스 앞에 final을 추가하여

상속할 수 없는 클래스라는 것을 알리는 것이 좋다

클래스 : public, (default), final, abstract

매서드 : 모든 접근제어자, final, abstract, static

멤버변수: 모든 접근제어자, final, static

지역변수: final

\* 접근제어자를 사용하는 이유

외부로부터 데이터보호

내부적으로 사용되는 부분을 감추기 위해

예) 시간 값을 가져오거나 셋팅 할 때

hour, minute, seond변수는 private으로 제한하여 그 클래스에서만 변경되도록하고

이 변수값을 읽고 변경하는 매서드는 public으로 하여 멤버변수값 다루는 것이 바람직

생성자의 접근자

생성자에 접근제어자를 사용하여 인스턴스 생성을 제한할 수 있다

보통 생성자의 접근제어자는 클래스의 접근제어자와 같으나 다르게 할 수도 있다

\* 생성자의 접근제어자를 private으로 지정하면 외부에서 생성자에 접근못하므로

인스턴스를 생성할수 없게된다. 그래도 클래스내부에서는 인스턴스 생성가능 함

대신 인스턴스를 생성해주는 public메소드를 제공하여 외부에서 이 클래스의 인스턴스

를 사용하도록 할 수 있다. 이메소드는 public인동시에 static이어야한다

예제

class Singleton {

private static Singleton s = new Singleton();

private Singleton() {

……

}

public static Singleton getInstance() {//여기서 Singletone은 리턴타입

return s;

}

}

제어자 사용시 주의사항

1) 메서드에 static과 abstract를 함께 사용 못함.static은 몸통이 있는 메서드에만 사용

2) 클래스에 abstract와 final을 함께 사용 못함. Final은 상속못함 의미 abstract는 상속

을 통해 완성해야함을 의미하기 때문

3) abstract 메서는 접근제어자가 private일수 없다.

자손클래스에서 구현해줘야 하는데 private이면 자손클래스에서 접근 못하므로

4) 메서드에 private과 final을 같이 사용할 필요 없다.

private인 메서드는 오버라이딩이 불가하기 때문에 둘 중 하나만 사용해도 충분

5) 다형성(Polymorphism)

여러 타입의 객체를 참조할 수 있도록 함으로써 다형성을 프로그램적으로 구현하였다

Tv t = new CaptionTv();

\* 참조변수의 타입이 참조변수가 참조하고 있는 인스턴스에서 사용할 수 있는 멤버의

개수를 결정한다

참조변수의 형변환

상속관계에 있는 클래스 사이에서만 형변환이 가능

예제

Car car =null;

FireEngine fe=new FireEngine();

FireEngine fe2 =null;

car=fe;car.water();는 안된다 car는 조상타입이기때문

fe2=(FireEngine)car;//Car car=new Car();로 Car 라면 fe2에는 car를 캐스팅할 수 없다

fe2.water();

형변환은 참조변수의 타입을 변환하는 것이지 인스턴스를 변환하는 것은 아니기에

참조변수의 형변환은 인스턴스에 아무런 영향을 미치지않는다

if c instanceof FireEngine이 참이라면

c는 FinreEngine으로 형변환이 가능하다는 것을 뜻한다

참조변수와 인스턴스의 연결

멤버변수가 조상클래스와 자손클래스에 중복으로 정의된 경우

조상타입의 참조변수를 사용했을 때는 조상클래스에 선언된 멤버변수가 사용된다

중복정의 되어있지 않다면

차이점이 없다

메서드는 참조변수의 타입에 상관없이 항상 실제인스턴스 타입이 호출

예제

Parent p= new Child(();

Child c= new Child();

p.method();//Child method가 프린트된다

c.method();

Parent {

x=1000;

void method(){System.out.println("Parent method");}

}

Child {

x=2000;

void method(){System.out.println("Child method");}

}

\* p.x는 1000이 들어가있고 c.x에는 2000이 들어가있다

\*

매개변수의 다형성

참조변수의 다형성은 메서드의 매개변수에도 적용된다

Tv, Computer, Audio의 조상클래스 Product를 매개변수로 이용

void buy(Tv t)

Void buy(Computer c) => void buy(Product p)

void buy(Audio a)

\* 매개변수가 Product타입의 참조변수라는 것은 메서드의 매개변수로 Product클래스의

자손타입의 참조변수나 인스턴스는 어느것이나 매개변수로 받아들일수 있다는뜻

\* 예제 7-21 : 여러 종류의 객체를 배열로 다루기

vector 클래스

내부적으로 Object타입의 배열을 가지고있어, 이 배열에 객체를 추가하거나 제거가능

배열의 크기를 동적으로 관리해주기때문에 객체의 개수에 신경쓰지않아도 됨

vector() 벡터 인스턴스 생성

boolean add(object o) 벡터에 객체추가. 성공하면 true 리턴

boolean remove(object o)

boolean isEmpty()

object get(int index) 지정된 위치(index)의 객체를 반환

반환타입이 Object타입이므로 적절한 형변환 필요

int size() 벡터에 저장된 객체의 개수를 반환

예제 7-22

6) 추상클래스

중요하지 않은 자세한 사항은 감추어 버리고, 가장 중요하고 필수적인 사항만 다룸으로써 복잡한 것들을 효율적으로 관리할 수 있는 개념

7) 인터페이스(interface)

일종의 추상클래스로 오직 추상메서드와 상수만을 맴버로 가질수 있다

다른 클래스를 작성하는데 도움을 줄 목적으로 작성된다

형식

interface 인터페이스이름 {

public static final 타입 상수이름= 값;

public abstract 메서드이름(매개변수 목록);

}

제약사항

(1) 모든 멤버변수는 public static final이어야하며 이를 생략할 수 있다

(2) 모든 메서드는 public abstract이어야하며 이를 생략할 수 있다

인터페이스의 상속

클래스와는 달리 여러 개의 인터페이스에서 다중상속이 가능

interface Fightable extends Movable, Attackable{ }

인터페이스의 구현

class Fighter implement Fightable {

class Fighter extends Unit implement Fightable {//상속과 구현을 동시에 할 수 있다

인터페이스를 이용한 다중상속

만일 두 개의 클래스로부터 상속을 받아야 할 상황이면, 두 조상클래스 중 비중이 높은 쪽을

상속받고 남은 한쪽은 클래스 내부에 포함시키거나 필요한 부분을 뽑아 인터페이스로

만든 다음 구현하는 방식으로 한다

인터페이스를 이용한 다형성

자손클래스의 인스턴스를 조상타입의 참조변수로 참조하는 것이 가능하다(다형성참조)

인터페이스 타입의 참조변수로 구현한 클래스의 인스턴스를 참조 할 수 있으며

인터페이스 타입으로 형변환도 가능하다

예제

Fightable f= (Fightable)new Fighter();

Fightable f= new Fighter();

인터페이스는 메서드의 매개변수의 타입으로 사용될 수 있다.

void Attack(Fightable f) {}

\*\* 인터페이스 타입의 매개변수가 갖는 의미는 메서드 호출시 해당 인터페이스를 구현한

클래스의 인스턴스를 매개변수로 제공해야 한다는 것이다

class Fighter extends Unit implements Fightable {

public void move(int x, int y) {……}

public void attack(Fightable f) {……}

}

\*\* 구현했을 때 attack(new Fighter())와 같이 할 수 있다

인터페이스는 메서드의 리턴타입으로 인터페이스 타입을 지정할 수 있다.

Fightable method() {

……

return new Fighter();

}

\*\* 리턴타입이 인터페이스라는 것은 메서드가 해당 인터페이스를 구현한 클래스의

인스턴스를 반환한다는 것을 의미한다

예제 7-24

인터페이스의 장점

1) 개발시간을 단축시킬 수 있다

인터페이스 작성하면 메서드를 호출하는 쪽에서는 선언부만 알면되기에

호출하는 쪽과 인터페이스를 구현하는 쪽을 동시에 진행할 수 있다

2) 표준화가 가능

프로젝트에 사용되는 기본틀을 인터페이스로 작성한 다음

개발자들이 인터페이스를 구현하여 프로그램 작성함으로 정형화된 프로그램 개발

3) 서로 관계없는 클래스들에게 관계를 맺어줄 수 있다

상속관계에 있지 않은 서로 관계가 없는 클래스에게 인터페이스를 공통적으로 구현

하도록 함으로써 관계를 맺어줄 수 있다.

4) 독립적인 프로그래밍이 가능

클래스의 선언과 구현을 분리함으로 프로그램 변경 없이 인터페이스만 변경하면 된다.

예제 7-25

인터페이스의 이해

먼저 고려해야 할 사항

클래스를 사용하는 쪽(User)과 클래스를 제공하는 쪽(Provider)이 있다

메서드를 사용하는 쪽(User)에서는 사용하려는 메서드의 선언부만 알면 된다.

메서드의 내용은 몰라도 된다

예제

Interface I {

public abstract void methodB();

}

class B implements I {

public void methodB() {

System.out.println(methodB in B class");

}

}

class A {

public void methodA (I i) {

i.methodB();

}

클래스 A는 인터페이스를 통해 실제로 사용하는 메서드의 이름을 몰라도 되고

심지어 실제로 구현된 메서드가 존재하지 않아도 문제되지 않는다

클래스 A는 직접적인 관계에 있는 인터페이스 I의 영향만을 받는다