**[ 14 ] 인터페이스 I**

목표 : 인터페이스의 이해와 문법

1. 인터페이스의 이해

객체지향 언어를 접하면서 '인터페이스'는 중요한, 쉽지 않는 개념입니다.

여기서 '쉽지 않다'라는 의미는 인터페이스 문법을 습득하는 것에 어려움이 있다는 것이 아니고, 인터페이스를 개발 중인 프로그래밍에 어떻게 접목시켜야 할지 고민이 많이 된다는 것입니다. 문법은 쉽습니다.

사실 오늘 수업을 통해 인터페이스를 완전히 이해하고, 실제 프로젝트에 적용한다는 것은 어렵습니다. 이번 강의를 통해서 인터페이스의 문법을 이해하고, 추후 살펴볼 디자인 패턴 강의 때 다시 한번 인터페이스를 공부한다면 이해에 도움이 될 것입니다. 시간이 지나면서 인터페이스를 내 소스에 녹여낼 수 있을 것입니다.

그리고, 실무에서 **규모가 어느 정도 큰 프로젝트**를 하게 된다면, 인터페이스의 필요성에 대해서 완전히 이해를 하게 될 것입니다. (수업 중에 하게 되는 짧은 예제에서는 인터페이스의 필요성을 거의 느끼지 못할 수도 있을 것입니다)

⑴ 인터페이스란?

① 작업명세서(작업지시서) - "앞으로 이렇게 만들어요"라고 표현해 놓은 것

- 실제 구현된 것이 전혀 없는 기본 설계도.(알맹이 없는 껍데기)

- 일종의 추상클래스. 추상클래스(미완성 설계도)보다 추상화 정도가 높다.

- 인스턴스를 생성할 수 없고, 클래스 작성에 도움을 줄 목적으로 사용된다

- 미리 정해진 규칙에 맞게 구현하도록 표준을 제시하는 데 사용된다

- 추상메서드와 상수만을 멤버로 가질 수 있다.

② 다형성을 가능하게 한다(하나의 객체를 다양하게 많은 type으로 만들 수 있다).

Class S{

...

public void method(){…}

}

Class C extends S {

...

public void method(){…}

}

C c = new C();

S c = new C();

S s = new S();

s.method(); c.method();

③ 객체를 **부속품화** -다양한 객체를 제품의 부속품처럼 개발자 마음대로 변경 할 수 있다.

④ 사용법은 어렵지 않지만, 실제 개발에 적용시키기는 쉽지 않다.

⑤ 인터페이스를 공부하는데 가장 좋은 방법은 패턴이나 프레임워크(ex. spring)를 통해 습득하는 것

⑥ 객체와 객체 간의 소통수단

2. 인터페이스의 문법

⑴ 'class'대신 'interface' 예약어를 사용한다는 점에서 클래스와 유사하다.

⑵ 실제 구현된 기능은 없어요 - 추상메소드와 상수만이 존재 합니다.

public interface 인터페이스이름 {

public static final 타입 **상수**이름 = 값;

public **abstract 메서드** 이름(매개변수 목록); //구현된 메소드는 갖을 수 없다

}

☞모든 멤버변수는 public static final이어야하며 이를 생략할 수 있다.'

모든 메서드는 public abstract 이어야 하며, 이를 생략할 수 있다.

⑵ private는 불가 - 상수나 메소드를 만들 때 private 접근 제한자는 불가

⑶ 추상화 - 메소드는 무조건 추상 메소드만 존재 합니다. (예외는 있으나 잠깐 생각하지 말자.

default 메소드 자바1.8부터 지원. Static 메소드는 가능)

⑷ 객체타입 - 인터페이스는 객체를 생성할 수 없다. 다만, 객체 타입(변수 타입)으로만 사용 됩니다.

(예외, 익명 구현 객체만이 가능은 하다. 안드로이드에서 주로)

⑸ 구현은 Implement 되는 클래스에서 합니다.

<예제>

**public** **interface** InterfaceEx1 {

**public** /\*static final\*/ **int** ***CONSTANT\_NUM*** = 100;

**public** **void** calculate();

}

**public** **interface** InterfaceEx2 {

**public** **static** **final** String ***CONSTANT\_STRING*** = "Hello World";

**public** String getStr();

}

**public** **class** InterfaceClass **implements** InterfaceEx1, InterfaceEx2{

@Override

**public** String getStr() {

System.***out***.println("실제 구현은 implements한 클래스에서 해요");

**return** **null**;

}

@Override

**public** **void** calculate() {

System.***out***.println("실제 구현은 implements한 클래스에서 해요");

}

}

**public** **class** TestMain {

**public** **static** **void** main(String[] args) {

InterfaceClass ifc = **new** InterfaceClass();

ifc.getStr();

ifc.calculate();

InterfaceEx1 if1 = **new** InterfaceClass();

InterfaceEx2 if2 = **new** InterfaceClass();

// 인터페이스는 객체를 생성하지는 못하지만 변수객체 타입으로는 가능하다.

// 그래서 접근을 제한하는 요도로 이렇게 사용하기도 한다

if1.calculate();

if2.getStr();

System.***out***.println("상수 InterfaceEx1.CONSTANT\_NUM : "+InterfaceEx1.***CONSTANT\_NUM***);

System.***out***.println("상수 InterfaceClass.CONSTANT\_NUM : "+InterfaceClass.***CONSTANT\_NUM***);

System.***out***.println("상수 InterfaceEx2.CONSTANT\_STRING : "+InterfaceEx2.***CONSTANT\_STRING***);

System.***out***.println("상수 InterfaceClass.CONSTANT\_STRING : "+InterfaceClass.***CONSTANT\_STRING***);

}

}

<예제2> 스마트폰 제작을 통한 Interface의 이해

요즘 스마트폰에는 아주 많은 기능이 있습니다. 제조사에서 출시할 때 기본적으로 설치되어 있는 어플 외에도 구글 스토어를 통해서 다양한 어플을 설치할 수 있습니다.

시나리오 : A전자에서 만든 다양한 스마트폰 모델은 아래와 같습니다.

a제품 : DMB송수신불가, 3G, TV리모콘 미탑재,

b제품 : DMB송수신가능, LTE, TV리모콘 탑재,

c제품 : DMB송수신가능, LTE, TV리모콘 미탑재,

**public** **interface** IAcor {

**public** **void** dmbReceive();

**public** **void** lte();

**public** **void** tvremoteControl();

}

**public** **class** AModel **implements** IAcor {

**private** String model = "A 모델";

@Override

**public** **void** dmbReceive() {System.***out***.println(model +"은 DMB 송신 가능한 모델");}

@Override

**public** **void** lte() {System.***out***.println(model +"은 3G 모델");}

@Override

**public** **void** tvremoteControl() {System.***out***.println(model +"은 TV리모콘 미탑재 모델");}

}

**public** **class** BModel **implements** IAcor {

**private** String model = "B 모델";

@Override

**public** **void** dmbReceive() {System.***out***.println(model +"은 DMB 송신 가능한 모델");}

@Override

**public** **void** lte() {System.***out***.println(model +"은 LTE 모델");}

@Override

**public** **void** tvremoteControl() {System.***out***.println(model +"은 TV리모콘 탑재 모델");}

}

**public** **class** CModel **implements** IAcor {

**private** String model = "C 모델";

@Override

**public** **void** dmbReceive() {System.***out***.println(model +"은 DMB 송신 가능한 모델");}

@Override

**public** **void** lte() {System.***out***.println(model +"은 LTE 모델");}

@Override

**public** **void** tvremoteControl() {System.***out***.println(model +"은 TV리모콘 미탑재 모델");}

}

**public** **class** TestMain {

**public** **static** **void** main(String[] args) {

/\*AModel a = new AModel();

BModel b = new BModel();

CModel c = new CModel();\*/

IAcor a = **new** AModel();

IAcor b = **new** BModel();

IAcor c = **new** CModel();

a.dmbReceive (); a.lte(); a.tvremoteControl();

b.dmbReceive (); b.lte(); b.tvremoteControl();

c.dmbReceive (); c.lte(); c.tvremoteControl();

IAcor[] phones = {a, b, c};

**for**(IAcor p : phones) {

p.dmbReceive();

p.lte();

p.tvremoteControl();

System.***out***.println("~~~~~~~~~~~~~~~~");

}

}

}

3. 다형성이란?

“여러 가지 형태를 가질 수 있는 능력”

말 그대로 객체가 다양하게 변할 수 있다고 생각하면 됩니다

“one interface, multiple implementation”

※ 하나의 인터페이스를 사용하여 다양한 구현 방법을 제공

※ 하나의 클래스나 함수가 다양하게 동작하는 것.

<예1> 오버로딩(overloading) : 같은 클래스 내에서만 오버로딩이라 할 수 있다.

메소드 overriding(다중정의) : 같은 이름을 가진 메소드를 다른 클래스에서 재정의 하는 법

☞일반함수(메소드) 오버로딩

class Print{

public void print(int i){

System.out.println("정수값 print : "+i);

}

public void print(String str){

System.out.println("문자열 print : "+str);

}

public void print(boolean bool){

System.out.println("boolean값 print : "+bool);

}

public void print(char c){

System.out.println("문자 print : "+c);

}

}

public class PrintMain {

public static void main(String[] args) {

Print p = new Print();

p.print(false);

p.print('A');

p.print(10);

p.print("문자열은 쌍따옴표로 묶여있어");

}

}

☞생성자함수 오버로딩과 일반 메소드 오버로딩

public class OverloadingTest {

private int i1;

private int i2;

public OverloadingTest() {

System.*out*.println("인자 없는 생성자함수");

}

public OverloadingTest(int i) {

System.*out*.println("인자를 "+i+"로 받은 생성자함수");

i1 = i2= i;

}

public OverloadingTest(String str) {

System.*out*.println("인자를 "+str+"로 받은 생성자함수");

i1 = i2 = Integer.*parseInt*(str);

}

public OverloadingTest(int i1, int i2) {

System.*out*.println("인자를 "+i1+"과 "+i2+"로 받은 생성자함수");

this.i1 = i1;

this.i2 = i2;

}

public void calculate() {

System.*out*.println(i1+"+"+i2+"="+(i1+i2));

}

public void calculate(char c) {

System.*out*.print(""+i1+c+i2+" = ");

switch (c) {

case '+': System.*out*.println(i1+i2);break;

case '-': System.*out*.println(i1-i2);break;

case '\*': System.*out*.println(i1\*i2);break;

case '/': System.*out*.println((double)i1/i2);break;

default:

System.*out*.println("유효하지 않는 연산자입니다");

}

}

}

public class TestMain {

public static void main(String[] args) {

OverloadingTest t1 = new OverloadingTest();

OverloadingTest t2 = new OverloadingTest(5);

OverloadingTest t3 = new OverloadingTest("3");

OverloadingTest t4 = new OverloadingTest(5,3);

t1.calculate();

t2.calculate('-');

t3.calculate('@');

}

}

<예2> 하나의 참조변수로 여러 타입의 객체를 참조할 수 있는 것.

즉, 부모클래스 타입의 참조변수로 자식클래스 타입의 객체를 다룰 수 있는 것

Class TV {

......

}

Class UltraHDTV extends TV {

......

}

UltraHDTV tv1 = new UltraHDTV();

TV tv1 = new UltraHDTV();

<예3> 오버라이딩(메소드 재정의) ch10\_abstract의 com.tj.ex02참조

circle.draw()

rectangle.draw()

triangle.draw()

※ 오버라이딩의 조건 : ① 선언부가 같아야 한다(이름, 매개변수, 리턴타입)

② 접근제어자를 좁은 범위로 변경할 수 없다.

조상클래스 메서드가 protected라면 범위가 같거나 넓은 protected나 public으로만

※ 오버로딩 vs. 오버라이딩

오버로딩(overloading) : 컴파일러 입장에서는 기존에 없는 새로운 메서드를 정의하는 것(new)

메소드 다중정의 (같은 class에서 동일한 메소드가 매개변수를 달리 여러 개 존재

오버라이딩(overriding): 상속받은 메서드의 내용을 변경하는 것(change, modify)

메소드 재정의 : 부모클래스와 자식클래스에 동일한 method 존재(틀만 가져와 재정의)

※ 인터페이스에서의 다형성 예제

말 그대로 객체가 다양하게 변할 수 있다고 생각하면 됩니다.

영화를 생각해 봅시다. 배우는 어떤 영화에서는

경찰관도 되었다가 소방관도 되었다가 요리사도 됩니다.

경찰관이 되서 범인도 잡고 잃어버린 물건도 찾습니다.

소방관이 되어서 불도 끄고 사람도 구합니다.

요리사가 되어서 피자도 만들고 스파게티도 만듭니다.

|  |  |  |  |
| --- | --- | --- | --- |
|  | 경찰관 | 소방관 | 요리사 |
| 배우 | 범인찾기 | 불끄기 | 피자 요리 |
| 물건찾기 | 사람구하기 | 스파게티요리 |

이러한 상황을 interface 를 통해 살펴보도록 하겠습니다

**public** **interface** PoliceMan {

**public** **void** canCatchCriminal();

**public** **void** canSearch();

}

**public** **interface** FireFighter {

**public** **void** outFire();

**public** **void** saveMan();

}

**public** **interface** Chef {

**public** **void** makePizza();

**public** **void** makeSpaghetti();

}

**public** **class** Actor **implements** PoliceMan, FireFighter, Chef{

**private** String name;

**public** Actor(String name) {**this**.name = name;}

@Override

**public** **void** makePizza() {System.***out***.println("피자를 만들 수 있습니다.");}

@Override

**public** **void** makeSpaghetti() {System.***out***.println("스파게티를 만들 수 있습니다.");}

@Override

**public** **void** outFire() {System.***out***.println("불을 끌 수 있습니다.");}

@Override

**public** **void** saveMan() {System.***out***.println("사람을 구할 수 있습니다.");}

@Override

**public** **void** canCatchCriminal() {System.***out***.println("범인을 잡을 수 있습니다.");}

@Override

**public** **void** canSearch() {System.***out***.println("물건을 찾을 수 있습니다.");}

}

**public** **class** AcotorMain {

**public** **static** **void** main(String[] args) {

Actor park = **new** Actor("박보검");

park.canCatchCriminal(); park.canSearch();

park.outFire(); park.saveMan();

park.makePizza(); park.makeSpaghetti();

PoliceMan parkPol = **new** Actor("박보검");

parkPol.canCatchCriminal();

parkPol.canSearch();

FireFighter parkFire = **new** Actor("박보검");

parkFire.outFire();

parkFire.saveMan();

Chef parkChef = **new** Actor("박보검");

parkChef.makePizza();

parkChef.makeSpaghetti();

}

}