**[ 11 ] 인터페이스**

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.

오늘 수업을 통해 인터페이스를 완전히 이해하고, 실제 프로젝트에 적용한다는 것은 어렵습니다. 이번 수업을 통해서 인터페이스의 문법을 이해하고, 여러 가지 패턴을 통해 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.

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

- 미리 정해진 규칙에 맞게 구현하도록 표준을 제시하는 데 사용된다

- 추상메서드와 상수 만을 멤버로 가질 수 있다.

② 객체의 다형성 구현

③ 사용법은 어렵지 않지만, 실제 개발에 적용시키기는 쉽지 않다.

④ 인터페이스를 공부하는데 가장 좋은 방법은 패턴이나 프레임워크(ex. Spring)를 통해 습득하는 것

2. 인터페이스의 문법

⑴ 'class'대신 'interface' 예약어를 사용한다는 점에서 클래스와 유사

⑵ 실제 구현된 기능 없이 **추상메소드와 상수**만이 존재

public interface 인터페이스이름 {

public /\* static final \*/ 타입 **상수**이름 = 값;

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

}

☞모든 멤버변수는 public static final이어야하며 static final은 생략한다.'

모든 메서드는 public abstract 이어야 하며, abstract를 생략한다.

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

⑶ 변수 타입 - 인터페이스는 객체를 생성할 수 없다. 다만, 변수 타입으로만 사용 됩니다.

(예외, 익명 구현 객체만이 가능은 하다. 안드로이드에서 주로)

⑷ 구현은 Implement 되는 클래스에서 합니다.

**(ex01)**

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

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

**public** /\*abstract\*/ **void** mehod1();

}

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

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

**public** String method2();

}

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

@Override

**public** String method2() {

System.***out***.println("2. 실제 구현은 implements한 클래스에서 해요");

**return** **null**;

}

@Override

**public** **void** method1() {

System.***out***.println("1. 실제 구현은 implements한 클래스에서 해요");

}

}

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

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

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

ifc.method1();

ifc.method2();

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

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

// 인터페이스는 객체를 생성하지는 못하지만 변수객체 타입으로는 가능하다.

// 그래서 접근을 제한하는 용도로 이렇게 사용하기도 한다

if1.method1();

if2.method2();

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***);

}

}

**(ex02) 앞장의** **HeadQuarterStore예제를 interface로 바꾸기**

**(ex03)**

볼륨을 올리거나 내리는 인터페이스를 만들어 그 인터페이스를 활용한 클래스(TV, Speaker)를 만들어봅시다. – 인터페이스를 기능정의서 용도로 사용

**public** **interface** IVolume {

**public** **void** volumeUp();

**public** **void** volumeDown();

**public** **default** **void** setMute(**boolean** mute) {

**if**(mute) {

System.***out***.println("무음");

}**else** {

System.***out***.println("무음 해제");

}

}

**public** **static** **void** changeBattery() {

System.***out***.println("건전지를 교환합니다");

}

}

**public** **class** Speaker **implements** IVolume {

**private** **final** **int** SPEAKER\_MAX\_VOLUME = 100;

**private** **final** **int** SPEAKER\_MIN\_VOLUME = 0;

**private** **int** volumeLevel;

**public** Speaker() {

volumeLevel = 5;

}

**public** Speaker(**int** volumeLevel) {

**this**.volumeLevel = volumeLevel;

}

@Override

**public** **void** volumeUp() {

**if**(volumeLevel<SPEAKER\_MAX\_VOLUME) {

volumeLevel ++;

System.***out***.println("스피커 볼륨을 1만큼 올려 현재"+volumeLevel);

}**else** {

System.***out***.println("스피커 볼륨이 최대치여서 올리지 못했습니다");

}

}

@Override

**public** **void** volumeDown() {

**if**(volumeLevel > SPEAKER\_MIN\_VOLUME) {

volumeLevel --;

System.***out***.println("스피커 볼륨을 1만큼 내려 현재 "+volumeLevel);

}**else** {

System.***out***.println("스피커 볼륨이 최소치여서 내리지 못했습니다.");

}

}

}

**public** **class** TV **implements** IVolume {

**private** **final** **int** SPEAKER\_MAX\_VOLUME = 30;

**private** **final** **int** SPEAKER\_MIN\_VOLUME = 0;

**private** **int** volumeLevel;

**public** TV() {

volumeLevel = 5;

}

**public** TV(**int** volumeLevel) {

**this**.volumeLevel = volumeLevel;

}

@Override

**public** **void** volumeUp() {

**if**(volumeLevel<SPEAKER\_MAX\_VOLUME) {

volumeLevel ++;

System.***out***.println("TV 볼륨을 1만큼 올려 현재"+volumeLevel);

}**else** {

System.***out***.println("TV 볼륨이 최대치여서 올리지 못했습니다");

}

}

@Override

**public** **void** volumeDown() {

**if**(volumeLevel > SPEAKER\_MIN\_VOLUME) {

volumeLevel --;

System.***out***.println("TV 볼륨을 1만큼 내려 현재 "+volumeLevel);

}**else** {

System.***out***.println("TV 볼륨이 최소치여서 내리지 못했습니다.");

}

}

}

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

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

IVolume[] devices = {**new** TV(5), **new** Speaker()};

**for**(**int** idx=0 ; idx<devices.length ; idx++) {

devices[idx].volumeDown();

devices[idx].setMute(**true**);

}

**for**(IVolume device : devices) {

device.volumeDown();

device.volumeDown(50);

device.setMute(**true**);

}

}

}

3. 다형성이란?

“여러 가지 형태를 가질 수 있는 능력” 객체가 다양하게 변할 수 있다.

“one interface, multiple implementation”

※ 하나의 인터페이스를 사용하여 다양한 구현 방법을 제공

※ 하나의 클래스나 함수가 다양하게 동작하는 것.

<예> 메소드 overloading(중복정의) : 같은 클래스 내, 매개변수에 따라 오버로딩이라 할 수 있다.

컴파일러 입장에서는 기존에 없는 새로운 메서드를 정의하는 것(new)

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

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

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

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

② 접근제어자를 좁은 범위로 변경할 수 없다.

조상클래스 메서드가 protected라면 범위가 같거나 넓은 protected나 public으로만

**(ex04)** 인터페이스에서의 다형성 예제

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** Chef, FireFighter, PoliceMan{

**private** String name;

**public** Actor(String name) {

**this**.name = name;

}

@Override

**public** **void** makePizza() {System.***out***.println(name+"은 요리사입니다 ♨ 피자를 만들 수 있다"); }

@Override

**public** **void** makeSpaghetti() {System.***out***.println(name+"은 요리사입니다 ♨ 스파게티를 만들 수 있다");}

@Override

**public** **void** outFire() {System.***out***.println(name+"은 소방관입니다 ♥ 불을 끌 수 있다");}

@Override

**public** **void** saveMan() {System.***out***.println(name+"은 소방관입니다 ♥ 사람을 구할 수 있다");}

@Override

**public** **void** canCatchCriminal() {System.***out***.println(name+"은 경찰입니다 ☞ 범인을 잡는다");}

@Override

**public** **void** canSearch() {System.***out***.println(name+"은 경찰관입니다 ☞ 물건을 찾는다");}

**public** String getName() {**return** name;}

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

}

**public** **class** ActorMain {

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

Actor park = **new** Actor("박보검");

park.makePizza();

park.makeSpaghetti();

park.outFire();

park.saveMan();

park.canCatchCriminal();

park.canSearch();

Chef chefPark = park;

chefPark.makePizza();

chefPark.makeSpaghetti();

// chefPark.outFire();

// chefPark.saveMan();

// chefPark.canCatchCriminal();

// chefPark.canSearch();

FireFighter firePark = park;

// firePark.makePizza();

// firePark.makeSpaghetti();

firePark.outFire();

firePark.saveMan();

// firePark.canCatchCriminal();

// firePark.canSearch();

PoliceMan policePark = park;

// policePark.makePizza();

// policePark.makeSpaghetti();

// policePark.outFire();

// policePark.saveMan();

policePark.canCatchCriminal();

policePark.canSearch();

}

}

4. 인터페이스의 장점

1. 개발시간을 단축시킬 수 있다.

일단 인터페이스가 작성되면, 이를 사용해서 프로그램을 작성하는 것이 가능하다. 메서드를 호출하는 쪽에서는 메서드의 내용에 관계없이 선언부만 알면 되기 때문이다. 그리고 동시에 다른 한 쪽에서는 인터페이스를 구현하는 클래스를 작성하도록 하여, 인터페이스를 구현하는 클래스가 작성될 때까지 기다리지 않고도 양쪽에서 동시에 개발을 진행할 수 있다.

1. 표준화가 가능하다

프로젝트에 사용되는 기본 틀을 인터페이스로 작성한 다음, 개발자들에게 인터페이스를 구현하여 프로그램을 작성하도록 함으로써 보다 일관되고 정형화된 프로그램의 개발이 가능하다

1. 독립적인 프로그래밍이 가능하다

인터페이스를 이용하면 클래스의 선언과 구현을 분리시킬 수 있기 때문에 실제 구현에 독립적인 프로그램을 작성하는 것이 가능하다. 클래스와 클래스간의 직접적인 관계를 인터페이스를 이용해서 간접적인 관계로 변경하면, 한 클래스의 변경이 관련된 다른 클래스에 영향을 미치지 않는 독립적인 프로그래밍이 가능하다.

6. 인터페이스와 추상클래스

(공통점)

1. 추상메소드를 가지고 있습니다 - 추상메소드를 가지고 있어 하위 클래스에서 구현해야 합니다.
2. 변수 타입이 목적 입니다 - 객체생성이 목적이 아닌 변수 타입을 정의하는 것이 목적입니다.
3. 객체 생성이 불가합니다. anonymose(익명클래스)는 이용할 수 있다..