JAVA

- 객체지향 3 : 추상클래스와 인터페이스

추상클래스와 인터페이스

Abstract & interface

abstract 클래스

추상클래스(abstract class)란?

- 클래스가 설계도라면 추상클래스는 '미완성 설계도'
- 추상메서드(미완성 메서드)를 포함하고 있는 클래스
  * 추상메서드 : 선언부만 있고 구현부(몸통, body)가 없는 메서드

```
abstract class Player {
     int currentPos;          // 현재 Play되고 있는 위치를 저장하기 위한 변수

     Player() {               // 추상클래스도 생성자가 있어야 한다.
          currentPos = 0;
     }

     abstract void play(int pos);      // 추상메서드
     abstract void stop();             // 추상메서드

     void play() {
          play(currentPos);            // 추상메서드를 사용할 수 있다.
     }
    ...
}
```

- 일반메서드가 추상메서드를 호출할 수 있다.(호출할 때 필요한 건 선언부)
- 완성된 설계도가 아니므로 인스턴스를 생성할 수 없다.
- 다른 클래스를 작성하는 데 도움을 줄 목적으로 작성된다.

추상클래스 (abstract class)

추상메서드(abstract method)란?

- 선언부만 있고 구현부(몸통, body)가 없는 메서드

```
/* 주석을 통해 어떤 기능을 수행할 목적으로 작성하였는지 설명한다. */
abstract 리턴타입 메서드이름();

Ex)
/* 지정된 위치(pos)에서 재생을 시작하는 기능이 수행되도록 작성한다.*/
abstract void play(int pos);
```

- 꼭 필요하지만 자손마다 다르게 구현될 것으로 예상되는 경우에 사용
- 추상클래스를 상속받는 자손클래스에서 추상메서드의 구현부를 완성해야 한다.

```
abstract class Player {
    ...
     abstract void play(int pos);      // 추상메서드
     abstract void stop();             // 추상메서드
    ...
}

class AudioPlayer extends Player {
    void play(int pos) { /* 내용 생략 */ }
    void stop() { /* 내용 생략 */ }
}

abstract class AbstractPlayer extends Player {
    void play(int pos) { /* 내용 생략 */ }
}
```

인스턴스의 생성을 허용 안 하는 abstract 클래스

```
class Friend {
    ..............
    public void showData() {
        System.out.println("이름 : "+name);
        System.out.println("전화 : "+phoneNum);
        System.out.println("주소 : "+addr);
    }
    public void showBasicInfo() {  }
}
```

앞서 상속 관련 예제에서 정의한 Friend 클래스!

이 클래스는 UnivFriend와 HighFriend를 상속의 관계로 연결하기 위해 정의한 클래스다.

즉, 인스턴스화에 목적이 없다!

달리 말해서 인스턴스화 된다면, 이는 실수다!

인스턴스의 생성을 허용 안 하는 abstract 클래스

추상화! 인스턴스 생성을 막음

```
abstract class Friend { // 하나 이상의 메소드가 abstract면, 클래스도 abstract
    ..............
    public void showData() {
        System.out.println("이름 : "+name);
        System.out.println("전화 : "+phoneNum);
        System.out.println("주소 : "+addr);
    }
    public abstract void showBasicInfo() {  }   메소드를 완성시키지 않는다는 선언
}
```

showBasicInfo 메소드는 비어있었다. 이렇듯 오버라이딩의 관계 유지를 목적으로 하는 메소드는 abstract로 선언이 가능하다.

* **하나 이상 abstarct 메소드를 포함하는 클래스는 abstract로 선언되어야 하며, 인스턴스 생성은 불가!**
* **인스턴스생성은 불가능하나, 참조변수 선언 가능하고, 오버라이딩의 원리 그대로 적용됨!**

인스턴스의 생성을 허용 안 하는 abstract 클래스

```
abstract class AAA
{
    void methodOne() { . . . }
    abstract void methodTwo();
}

class BBB extends AAA
{
    void methodThree() { . . . }
}
```

컴파일 에러 발생 BBB 클래스도 abstract로 선언 되어야 에러 발생 않는다!

위의 경우 BBB 클래스는 AAA 클래스의 abstract 메소드를 상속한다.

그런데 오버라이딩 하지 않았으므로, abstract 상태 그대로 놓이게 된다.

결국 BBB 클래스는 하나 이상의 abstract 메소드를 포함한 셈이니,

abstract로 선언 되어야 하며, 인스턴스의 생성도 불가능하게 된다.

interface

인터페이스(interface)란?

- 일종의 추상클래스. 추상클래스(미완성 설계도)보다 추상화 정도가 높다.
- 실제 구현된 것이 전혀 없는 기본 설계도.(알맹이 없는 껍데기)
- 추상메서드와 상수만을 멤버로 가질 수 있다.
- 인스턴스를 생성할 수 없고, 클래스 작성에 도움을 줄 목적으로 사용된다.
- 미리 정해진 규칙에 맞게 구현하도록 표준을 제시하는 데 사용된다.

인터페이스의 구성

- 'class'대신 'interface'를 사용한다는 것 외에는 클래스 작성과 동일하다.

```
interface 인터페이스이름 {
        public static final 타입 상수이름 = 값;
        public abstract 메서드이름(매개변수목록);
}
```

- 하지만, 구성요소(멤버)는 **추상메서드와 상수만 가능**하다.

- 모든 멤버변수는 public static final 이어야 하며, 이를 생략할 수 있다.
- 모든 메서드는 public abstract 이어야 하며, 이를 생략할 수 있다.

인터페이스의 구현

- 인터페이스를 구현하는 것은 클래스를 상속받는 것과 같다.
  다만, 'extends' 대신 'implements'를 사용한다.

```
class 클래스이름 implements 인터페이스이름 {
     // 인터페이스에 정의된 추상메서드를 구현해야한다.
}
```

- 인터페이스에 정의된 추상메서드를 완성해야 한다.

```
class Fighter implements Fightable {
     public void move() { /* 내용 생략*/ }
     public void attack() { /* 내용 생략*/ }
}

interface Fightable {
     void move(int x, int y);
     void attack(Unit u);
}
```

```
abstract class Fighter implements Fightable {
     public void move() { /* 내용 생략*/ }
}
```

- 상속과 구현이 동시에 가능하다.

```
class Fighter extends Unit implements Fightable {
     public void move(int x, int y) { /* 내용 생략 */}
     public void attack(Unit u) { /* 내용 생략 */}
}
```

문제의 상황

요구사항 1

이름과 주민등록 번호를 저장하는 기능 클래스가 필요하다.

이 클래스에는 주민등록 번호를 기반으로 사람의 이름을 찾는 기능이 포함되어야 한다.

요구사항 2

주민등록번호와 이름의 저장

→ void addPersnalInfor(String perNum, String name)

주민등록번호를 이용한 검색

→ String searchName(String perName)

문제의 상황

→ **문제 1**

"나도 프로젝트를 진행해야 하는데, A사가 클래스를 완성할 때까지 기다리고만 있을 수 없잖아! 그리고 나중에 내가 완성한 결과물과 A사가 완성한 결과물을 하나로 묶을 때 문제가 발생하면 어떻게 하지? A사와 나 사이에 조금 더 명확한 약속이 필요할 것 같은데"

→ **문제 2**

"내가 요구한 기능의 메소드들이 하나의 클래스에 담겨있지 않으면 어떻게 하지? A사에서 몇 개의 클래스로 기능을 완성하건, 나의 하나의 인스턴스로 모든 일을 처리하고 싶은데! 무엇보다도 나는 A사가 완성해 놓은 기능들을 활용만 하고 싶다고! 어떻게 구현했는지 관심 없다고!

인터페이스의 정의

→ **해결책**

"**클래스를 하나 정의해야겠다.** 그리고 A사에는 **이 클래스를 상속해서 기능을 완성해달라고 요구**하고, 난 이 클래스를 기준으로 프로젝트를 진행해야 겠다!!"

인터페이스의 정의

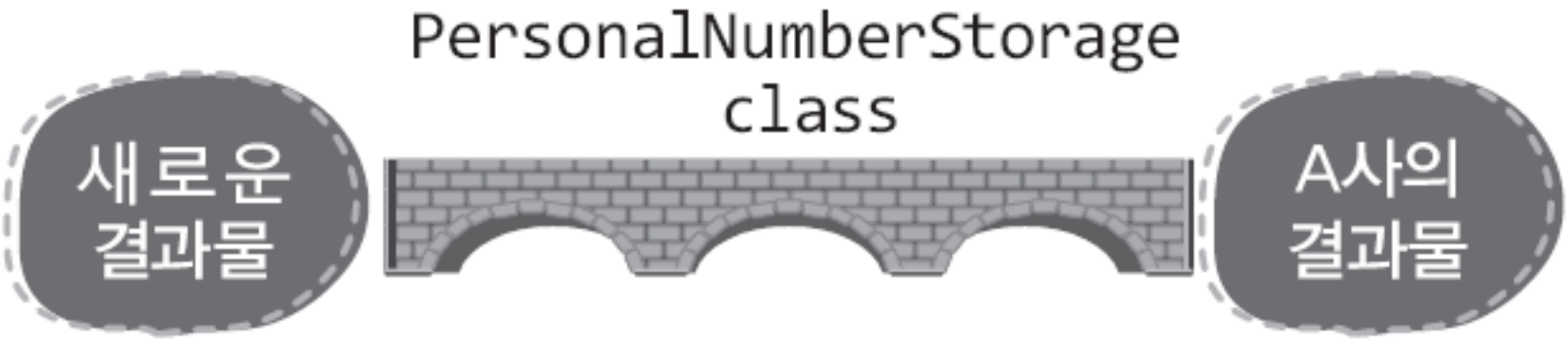
→ **새로 정의한 클래스**

```
abstract class PesonalNumberStorage
{
    public abstract void addPersnalInfor(String perNum, String name) ;
    public abstract String searchName(String perName) ;
}
```

인터페이스의 정의

```
class abstractInterface
{
    public static void main(String[] args)
    {
        PesonalNumberStorage storage = new (a사가 구현한 클래스 이름);
        storage.addPersnalInfor("홍길동","991111-1111111");
        system.out.println(storage.searchName("991111-1111111"));
    }
}
```

인터페이스의 정의



- 클래스 PersonalNumberStorage는 인터페이스의 역할을 하는 클래스이다.
- 인터페이스는 두 결과물의 연결고리가 되는 일종의 약속 역할을 한다.
- 인터페이스의 정의로 인해서 개발 하고는 있는 나는 나대로,

  A사는 A사대로 더 이상의 추가 논의 없이 프로젝트를 진행할 수 있었다.
- 인터페이스가 정의 되었기 때문에 프로젝트를 하나로 묶는 과정도 문제가 되지 않는다.

interface 예제

```
abstract class PersonalNumberStorage
{
        public abstract void addPersonalInfo(String perNum, String name);
        public abstract String searchName(String perNum);
}

class PersonalNumInfo
{
        String name;
        String number;

        PersonalNumInfo(String name, String number)
        {
                this.name=name;
                this.number=number;
        }

        String getName(){return name;}
        String getNumber(){return number;}
}
```

interface 예제

```
class PersonalNumberStorageImpl extends PersonalNumberStorage {
        PersonalNumInfo[] perArr;
        int numOfPerInfo;

        public PersonalNumberStorageImpl(int sz){
                perArr=new PersonalNumInfo[sz];
                numOfPerInfo=0;
        }

        public void addPersonalInfo(String name, String perNum){
                perArr[numOfPerInfo]=new PersonalNumInfo(name, perNum);
                numOfPerInfo++;
        }

        public String searchName(String perNum){

                for(int i=0; i<numOfPerInfo; i++){

                        if(perNum.compareTo(perArr[i].getNumber())==0)
                                return perArr[i].getName();
                }
                return null;
        }
}
```

interface 예제

```
class AbstractInterface
{
        public static void main(String[] args)
        {
                PersonalNumberStorage storage=
                                        new PersonalNumberStorageImpl(100);

                storage.addPersonalInfo( " 손흥민 " ,  " 950000-1122333 " );
                storage.addPersonalInfo( " 박지성", "970000-1122334");

                System.out.println(storage.searchName("950000-1122333"));
                  System.out.println(storage.searchName("970000-1122334"));
        }
}
```

Interface의 활용

```
abstract class PesonalNumberStorage
{
    public abstract void addPersnalInfor(String perNum, String name) ;
    public abstract String searchName(String perName) ;
}
```

모든 메소드가 abstract로 선언된 abstract 클래스는 다음과 같이 정의 가능하다!

```
interface  PesonalNumberStorage
{
    void addPersnalInfor(String perNum, String name) ;
    String searchName(String perName) ;
}
```

인터페이스간의 상속

- 인터페이스도 클래스처럼 상속이 가능하다.(클래스와 달리 다중상속 허용)

```
interface Movable {
     /** 지정된 위치(x, y)로 이동하는 기능의 메서드 */
     void move(int x, int y);
}

interface Attackable {
     /** 지정된 대상(u)을 공격하는 기능의 메서드 */
     void attack(Unit u);
}

interface Fightable extends Movable, Attackable { }
```

- 인터페이스는 Object클래스와 같은 최고 조상이 없다.

Interface의 특성

```
class Fighter extends Unit implements Fightable {
        public void move(int x, int y) { /* 내용 생략 */ }
        public void attack(Unit u) { /* 내용 생략 */ }
}

class Unit {
        int currentHP;      // 유닛의 체력
        int x;                          // 유닛의 위치(x좌표)
        int y;                          // 유닛의 위치(y좌표)
}

interface Fightable extends Movable, Attackable { }
interface Movable {         void move(int x, int y);     }
interface Attackable {      void attack(Unit u); }
```

Interface의 특성

```
class FighterTest {
        public static void main(String[] args) {
                Fighter f = new Fighter();

                if (f instanceof Unit)          {
                    System.out.println("f는 Unit클래스의 자손입니다.");
                }
                if (f instanceof Fightable) {
                    System.out.println("f는 Fightable인터페이스를 구현했습니다.");
                }
                if (f instanceof Movable) {
                    System.out.println("f는 Movable인터페이스를 구현했습니다.");
                }
                if (f instanceof Attackable) {
                    System.out.println("f는 Attackable인터페이스를 구현했습니다.");
                }
                if (f instanceof Object) {
                    System.out.println("f는 Object클래스의 자손입니다.");
                }
        }
}
```

Interface 기반의 상수표현

```
public class {
    public static final int MON=1;
    public static final int TUE=2;
    public static final int WED=3;
    public static final int THU=4;
    public static final int FRI=5;
    public static final int SAT=6;
    public static final int SUN=7;
}
```

인터페이스 내에 선언된 변수는 무조건 public static final로 선언이 되므로, 이 둘은 완전히 동일한 의미를 갖는다.

```
public interface week
{
    int MON=1, TUE=2, WED=3, THU=4, FRI=5, SAT=6, SUN=7;
}
```

Interface 기반의 상수표현

```
import java.util.Scanner;

interface Week
{
   int MON=1, TUE=2, WED=3, THU=4, FRI=5, SAT=6, SUN=7;
}

class MeaningfulConst
{
        public static void main(String[] args)
        {
                System.out.println("오늘의 요일을 선택하세요. ");
                System.out.println("1.월요일, 2.화요일, 3.수요일, 4.목요일");
                System.out.println("5.금요일, 6.토요일, 7.일요일");
                System.out.print("선택: ");

                Scanner sc=new Scanner(System.in);
                int sel=sc.nextInt();
```

Interface 기반의 상수표현

```
                switch(sel) {
                case Week.MON:
                        System.out.println("주간회의가 있습니다.");
                        break;
                case Week.TUE:
                        System.out.println("프로젝트 기획 회의가 있습니다.");
                        break;
                case Week.WED:
                        System.out.println("진행사항 보고하는 날입니다.");
                        break;
                case Week.THU:
                        System.out.println("사내 축구시합이 있는 날입니다.");
                        break;
                case Week.FRI:
                        System.out.println("프로젝트 마감일입니다.");
                        break;
                case Week.SAT:
                        System.out.println("가족과 함께 즐거운 시간을 보내세요");
                        break;
                case Week.SUN:
                        System.out.println("오늘은 휴일입니다.");
                }
        }
}
```

자바 interface의 또 다른 가치

```
interface UpperCasePrintable
{
        // 비어 있음
}

class ClassPrinter
{
        public static void print(Object obj)
        {
                String org=obj.toString();
                if(obj instanceof UpperCasePrintable)
                {
                        org=org.toUpperCase();
                }

                System.out.println(org);
        }
}
```

자바 interface의 또 다른 가치

```
class PointOne implements UpperCasePrintable
{
        private int xPos, yPos;

        PointOne(int x, int y)
        {
                xPos=x;
                yPos=y;
        }

        public String toString()
        {
                String posInfo="[x pos:"+xPos + ", y pos:"+yPos+"]";
                return posInfo;
        }
}
```

자바 interface의 또 다른 가치

```
class PointTwo
{
        private int xPos, yPos;

        PointTwo(int x, int y)
        {
                xPos=x;
                yPos=y;
        }

        public String toString()
        {
                String posInfo="[x pos:"+xPos + ", y pos:"+yPos+"]";
                return posInfo;
        }
}
```

자바 interface의 또 다른 가치

```
class InterfaceMark
{
        public static void main(String[] args)
        {
                PointOne pos1=new PointOne(1, 1);
                PointTwo pos2=new PointTwo(2, 2);
                PointOne pos3=new PointOne(3, 3);
                PointTwo pos4=new PointTwo(4, 4);

                ClassPrinter.print(pos1);
                ClassPrinter.print(pos2);
                ClassPrinter.print(pos3);
                ClassPrinter.print(pos4);
        }
}
```

```
[X POS:1, Y POS:1]
[x pos:2, y pos:2]
[X POS:3, Y POS:3]
[x pos:4, y pos:4]
```

인터페이스를 이용한 다형성

- 인터페이스 타입의 변수로 인터페이스를 구현한 클래스의 인스턴스를 참조할 수 있다.

```
class Fighter extends Unit implements Fightable {
     public void move(int x, int y)  { /* 내용 생략 */ }
     public void attack(Fightable f) { /* 내용 생략 */ }
}
```

```
Fighter   f = new Fighter();
Fightable f = new Fighter();
```

- 인터페이스를 메서드의 매개변수 타입으로 지정할 수 있다.

```
void attack(Fightable f) {   // Fightable인터페이스를 구현한 클래스의 인스턴스를
    //...                    // 매개변수로 받는 메서드
}
```

- 인터페이스를 메서드의 리턴타입으로 지정할 수 있다.

```
Fightable method() { // Fightable인터페이스를 구현한 클래스의 인스턴스를 반환
    // ...
    return new Fighter();
}
```

자바 interface의 또 다른 가치

- 무엇인가를 표시하는(클래스의 특성을 표시하는)용도로도 인터페이스는 사용된다.
- 이러한 경우, 인터페이스의 이름은 ~able로 끝나는 것이 보통이다.
- 이러한 경우, 인터페이스는 비어있을 수도 있다.
- instanceof 연산자를 통해서 클래스의 특성을 파악할 수 있다.

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.

2. 표준화가 가능하다.

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

3. 서로 관계없는 클래스들에게 관계를 맺어 줄 수 있다.

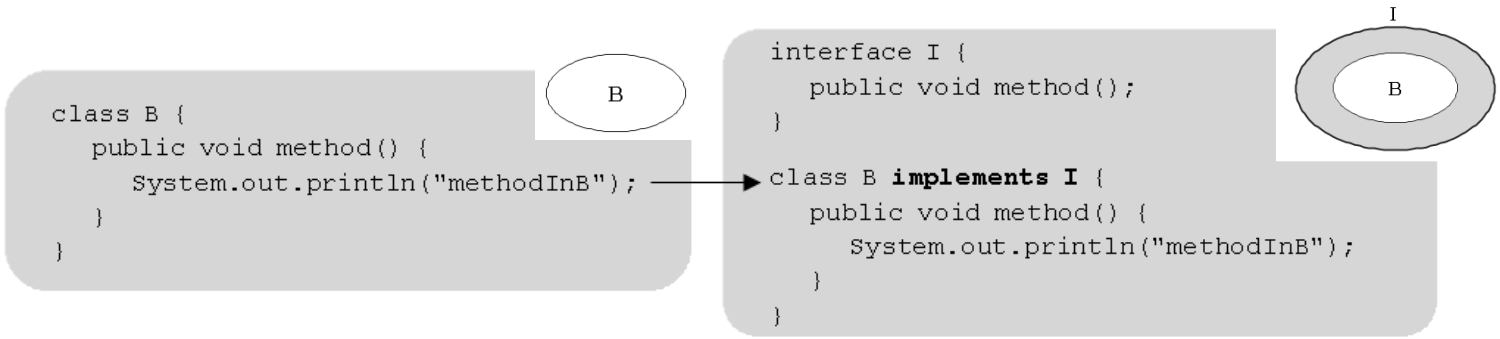
   서로 상속관계에 있지도 않고, 같은 조상클래스를 가지고 있지 않은 서로 아무런 관계도 없는 클래스들에게 하나의 인터페이스를 공통적으로 구현하도록 함으로써 관계를 맺어 줄 수 있다.

4. 독립적인 프로그래밍이 가능하다.

   인터페이스를 이용하면 클래스의 선언과 구현을 분리시킬 수 있기 때문에 실제구현에 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.

인터페이스의 이해

- 인터페이스는...
  - 두 대상(객체) 간의 '연결, 대화, 소통'을 돕는 '중간 역할'을 한다.
  - 선언(설계)와 구현을 분리시키는 것을 가능하게 한다.



- 인터페이스를 이해하려면 먼저 두 가지를 기억하자.
  - 클래스를 사용하는 쪽(User)과 클래스를 제공하는 쪽(Provider)이 있다.
  - 메서드를 사용(호출)하는 쪽(User)에서는 사용하려는 메서드(Provider)의 선언부만 알면 된다.

Project

Project ver 0.50

PhoneBookManager 클래스의 인스턴스수가 최대 하나를 넘지 않도록 코드를 변경.

‘initerface’기반의 상수 표현을 바탕으로 메뉴 선택과 그에 따른 처리가, 이름에 부여된 상 수를 기반으로 진행되도록 변경.

PhoneInfor 클래스의 구조에서 **“interface”**를 추가하여 **“추상 클래스”**로 구성해봅시다
이 인터페이스에는 오버라이딩을 위한 메서드만 포함하는 인터페이스로 구성합니다.