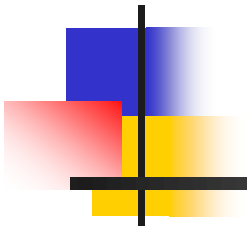
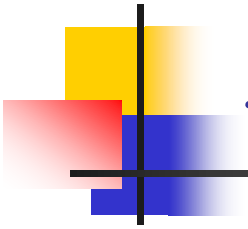


java 10강 추상클래스, 인터페이스

양 명 숙

[now4ever7@gmail.com]



목차

- 추상메서드
- 추상 클래스
- 인터페이스

추상 메서드(Abstract Method)

- 추상 메서드 - 몸체 없는 메서드, 미완성 메서드
  - **메서드의 구현부가 없다는 것**
    - 메서드 블럭{ }을 포함하고 있지 않고, 프로토타입(ProtoType)만 존재하는 메서드

```
public abstract int funcA(int a, int b);
```

  - 상속 계층의 부모 클래스에서 자식 클래스를 위해 메서드 시그니처만 정의해 놓고자 할 때 사용됨
  - 자식 클래스에서 재정의해야만 호출 가능한 메서드가 됨
  - 메서드의 내용이 상속받는 클래스에 따라 달라질 수 있기 때문에 **부모 클래스에서는 선언부만을 작성**하고, 주석을 덧붙여 어떤 기능을 수행할 목적으로 작성되었는지 알려주고, **실제 내용은 상속받는 클래스에서 구현**하도록 비워 두는 것임
  - 추상클래스를 상속받는 자식 클래스는 오버라이딩을 통해 부모의 추상 메서드를 상황에 맞게 적절히 구현해주어야 함

추상클래스(Abstract Class)

- **추상클래스**
  - **미완성 클래스,** 부분적으로만 완성된 '미완성 설계도'
  - **미완성 메서드를 포함**하고 있다는 의미
  - 완전한 클래스가 아니기 때문에 객체를 생성할 수 없음
  - 부모 클래스는 필요한 메서드를 추상적으로 정의할 뿐이며, 구체적으로 어떻게 구현할 것인가는 자식 클래스에게 결정권을 줌
  - 상속을 통해서 자식 클래스에 의해서만 완성될 수 있음
  - 추상 클래스 자체로는 클래스로서의 역할을 다 못하지만, 새로운 클래스를 작성하는 데 있어서 바탕이 되는 부모 클래스로서 중요한 의미를 갖는다.
- **추상클래스가 되는 방법**
  - 추상 메서드를 하나라도 포함하고 있는 클래스
  - 추상 메서드를 포함하고 있지 않더라도 클래스 선언할 때 abstract 키워드를 포함하고 있을 경우

추상클래스/추상 메서드

• **추상 클래스**
- 상속의 관계를 형성하기 위한 상위 클래스로
인스턴스화시키기 위해서 정의한 클래스가 아닌 경우 추상 클래스로 만든다

• **추상 메서드**
- 오버라이딩의 관계를 형성하기 위해 정의된 메서드, 비어있는 메서드는 추상 메서드로 만든다

추상클래스

- 추상 메서드의 선언 형태

```
public abstract void methodA();
```

- 추상클래스
  - 추상 메서드를 포함하고 있는 클래스
    - abstract 키워드를 이용해서 클래스 자신도 추상클래스라는 것을 명시해야 함
  - 추상 메서드를 포함하고 있지 않더라도 abstract 키워드만 포함하고 있다면 추상클래스가 됨

```
abstract class 클래스이름{
 ...
}
```

추상클래스

- 추상클래스 선언

```
abstract class Sample1
{
        public abstract void methodA();
}

abstract class Sample2
{
        public void methodB(){  }
}
```

추상클래스

- 추상클래스의 특징
  - 완전한 클래스가 아니기 때문에 완전하게 구현을 해야만 객체를 생성할 수 있음
    - **추상클래스를 상속한 후 모든 추상 메서드들을 구현했을 때 객체를 생성할 수 있음**
  - 객체가 가지는 특성들을 추상화 시켜 놓고 구체적인 **구현은** 이 추상클래스를 **상속 받는 자식클래스에서** 하도록 하는 것
  - 추상 메서드로 하는 대신, 아무 내용도 없는 메서드로 작성할 수도 있으나 추상 메서드로 선언하는 이유 - 자식 클래스에서 추상 메서드를 반드시 구현하도록 강요하기 위해서 사용
  - 프로그램의 설계 단계에서 많이 사용
    - 프로그램 상황에 맞는 계층적인 모델을 설계하도록 하는 역할을 함

예제

• sound() 메서드는 추상메서드이기 때문에 상속하여 구현해야 함 (메서드 재정의)

```
abstract class Animal{
	public abstract void sound();
}

class Dog extends Animal{
	public void sound() {
		System.out.println("멍멍");
	}
}

class Cow extends Animal{
	public void sound() {
		System.out.println("음메");
	}
}
```

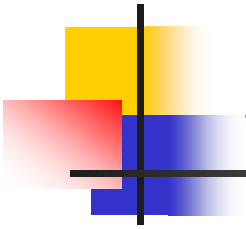
```
class Abstract1{
		public static void main(String[] args){
			Animal a;
			//a=new Animal();  에러
			a = new Dog();
			a.sound();

			a = new Cow();
			a.sound();
		}
}
```

• Animal은 일반적인 동물을 표현, 너무 일반적이어서 아무것도 할 수 없으며, 우는 소리조차 정의할 수 없다.

실습

```
도형을 선택하세요(1. 원, 2. 사각형)
1
원을 그립니다
```

- Shape 클래스(부모 클래스-추상 클래스)
  - draw() 메서드 - 추상 메서드
- Circle(자식)
  - draw() 메서드 오버라이딩
- Rect(자식)
  - draw() 메서드 오버라이딩
- Main()에서
  - 사용자로부터 원, 사각형 중 선택하게 하고, 각각 생성 후 draw()메서드 호출
    - 다형성 이용-Shape 클래스가 Circle, Rect를 참조하도록



추상클래스

- 추상메서드를 포함하고 있는 추상클래스를 상속받는 자식클래스는 추상클래스가 가지고 있는 모든 추상메서드를 구현해 주어야만 객체를 생성할 수 있음
- **자식클래스에서 추상메서드를 모두 구현해 주지 않으면** 자식클래스도 추상메서드를 포함하게 되므로 **추상클래스가 됨**
  - 이 자식클래스도 추상클래스로 선언해야 함

예제 1

- 자식 클래스가 추상 메서드를 재정의하지 않으면 이 클래스도 아직 추상 메서드임

```
abstract class Animal{
        public abstract void sound();
}
abstract class Mammal extends Animal{
        public void breed(int n){
                System.out.println( n + "마리 새끼를 낳는다");
        }
}
class Cat extends Mammal{
        public void sound() {
                System.out.println("야옹");
        }
}

class Abstract2{
        public static void main(String[] args){
                Animal a;
                //a=new Mammal();  에러
                a = new Cat();
                a.sound();
        }
}
```

sound()를 재정의하지 않았으므로 추상클래스

```
//Mammal ma = new Mammal(); //객체 생성 불가
Mammal ma = new Cat(); //다형성
ma.sound();
ma.breed(3);
```

예제2-에러 발생

```
abstract class  A {
    public abstract void f1();
    public abstract void f2();
}
class B extends A{   //에러발생 : 모든 추상메서드를 구현해야 한다.
    public void f1() {
        System.out.println("B클래스 f1()");
    }
}
class Abstract3_1{
    public static void main(String[] args){
        B b = new B();
        b.f1();
    }
}
```

예제2-개선

```
B클래스 F1()
C클래스 F2()
```

```
abstract class  A {
	public abstract void f1();
	public abstract void f2();
}
abstract class B extends A{    //모든 추상메서드를 구현하지 않았으므로 abstract 클래스로 지정
	public void f1() {
		System.out.println("B클래스 f1()");
	}
}
class C extends B{
	public void f2() {
		System.out.println("C클래스 f2()");
	}
}
class Abstract3{
	public static void main(String[] args){
		C c = new C();
		c.f1();
		c.f2();

		//B b = new B();  //에러
		//b.f1();
	}
}
```

추상 클래스 작성

```
abstract class Player{
    boolean pause;  //일시 정지 상태를 저장하기 위한 변수
    int currentPos;  //현재 play되고 있는 위치를 저장하기 위한 변수

    Player(){                                   //추상 클래스도 생성자가 있어야 한다
        pause=false;
        currentPos=0;
    }
    /** 지정된 위치 (pos)에서 재생을 시작하는 기능이 수행하도록 작성되어야 한다*/
    abstract void play(int pos);

    /** 재생을 즉시 멈추는 기능을 수행하도록 작성되어야 한다*/
    abstract void stop();

    void play(){
        play(currentPos);  //추상 메서드를 사용할 수 있다
    }
    void pause(){
        if (pause){                             //pause가 true일 때 (일시정지상태)에서 pause가 호출되면,
            pause=false;                        //pause의 상태를 false로 바꾸고,
            play(currentPos);                   //현재의 위치에서 play를 한다
        }else{                                  //play 상태에서 호출되면
            pause=true;
            stop();                             //play()를 멈춘다
        }
```

```
class CDPlayer extends Player{
    //부모 클래스의 추상 메서드를 구현한다
    void play(int currentPos){
            //구현
    }

    void stop(){
            //구현
    }

    //CDPlayer 클래스에 추가로 정의된 멤버
    int currentTrack;  //현재 재생 중인 트랙

    void nextTrack(){
            currentTrack++;
            //.....
    }

    void preTrack(){
            if (currentTrack>1){
                    currentTrack--;
            }
            //....
    }

}
```

부모 클래스의 추상 메서드를 CDPlayer 클래스의 기능에 맞게 완성해주고,
CDPlayer 만의 새로운 기능들을 추가한다

추상 클래스 만들기

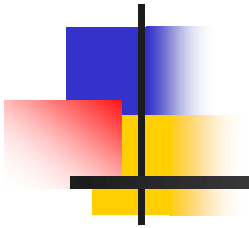
```
//기존의 클래스로부터 공통된 부분을 뽑아내어 추상클래스를 만들어 보자
class Marine{                       //보병
    int x, y;                       //현재 위치
    void move(int x, int y){/* 지정된 위치로 이동 */}
    void stop(){/* 현재 위치에 정지 */}
    void stimPack(){/* 스팀팩을 사용한다 */}
}

class Tank{                         //탱크
    int x, y;                       //현재 위치
    void move(int x, int y){/* 지정된 위치로 이동 */}
    void stop(){/* 현재 위치에 정지 */}
    void changeMode(){/* 공격모드를 변환한다 */}
}

class Dropship{                     //수송선
    int x, y;                       //현재 위치
    void move(int x, int y){/* 지정된 위치로 이동 */}
    void stop(){/* 현재 위치에 정지 */}
    void load(){/* 선택된 대상을 태운다 */}
    void unload(){/* 선택된 대상을 내린다 */}
}
```

- Starcraft에 나오는 유닛들을 클래스로 정의
이 유닛들은 각자 나름대로의 기능을 가지고 있지만
공통부분을 뽑아내어 하나의 클래스로 만들고,
이 클래스로부터 상속받도록 변경해보자

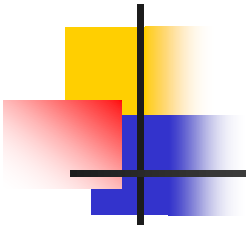
```
abstract class Unit{
    int x, y;                          //현재 위치
    abstract void move(int x, int y);  //Unit 클래스를 상속받아서 작성되는 클래스는
    //move 메서드를 자신의 클래스에 알맞게 반드시 구현해야 한다는 의미가 담겨 있다
    void stop(){/* 현재 위치에 정지 */}
}

class Marine extends Unit{         //보병
    void move(int x, int y){
        /* 지정된 위치로 이동 */
        System.out.println(x+","+y+"위치로 이동한다.");
    }
    void stimPack(){/* 스팀팩을 사용한다 */}
}

class Tank  extends Unit{                    //탱크
    void move(int x, int y){
                    System.out.println(x+","+y+"위치로 이동한다.");
    }
    void changeMode(){/* 공격모드를 변환한다 */}
}
```

- 각 클래스의 공통부분을 뽑아내서 Unit 클래스를 정의하고 이로부터 상속받도록 함
- 이 Unit 클래스는 다른 유닛을 위한 클래스를 작성하는데 재활용될 수 있다
- Marine,Tank는 지상유닛이고,Dropship은 공중유닛이기 때문에 이동하는 방법이 서로 달라서 move 메서드의 실제 구현 내용이 다름

```
class Dropship  extends Unit{                    //수송선
    void move(int x, int y){
        System.out.println("수송선의 위치를 "+x+","+y+"로 이동한다.");
    }
    void load(){/* 선택된 대상을 태운다 */}
    void unload(){/* 선택된 대상을 내린다 */}
}

class UnitTest {
    public static void main(String[] args) {
        Unit[] group = new Unit[4];
        group[0]=new Marine();
        group[1]=new Tank();
        group[2]=new Marine();
        group[3]=new Dropship();

        for (int i=0;i<group.length ;i++ ) {
            group[i].move(100, 200);
            //Unit 배열의 모든 유닛을 좌표(100, 200)의 위치로 이동한다
        }
    }
}
```

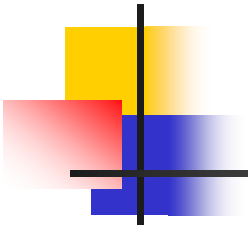


인터페이스



인터페이스(interface)

- 인터페이스
  - 일종의 추상 클래스
  - 추상메서드를 갖지만, 몸통을 갖춘 일반 메서드나 멤버변수를 구성원으로 가질 수 없음
  - **추상메서드와 상수만을 멤버**로 가질 수 있다
    - 메서드 목록만을 가지는 **특별한 타입**
  - 클래스의 뼈대만을 가지고 있는 것
  - 구현된 것은 아무것도 없고 밑그림만 그려져 있는 '기본 설계도'
  - 미리 정해진 규칙에 맞게 구현하도록 **표준을 제시**하는 데 사용됨



인터페이스(Interface)

- **인터페이스(Interface)**
  - 자신에게서 상속 받을 클래스가 구현해야 할 기능을 나열해 놓은 것
    - 자신은 직접 기능에 대한 구현을 가지지 않고, 자신의 자식클래스가 그 메서드를 구현하도록 하는 것
  - 다른 인터페이스나 클래스의 부모로만 사용되어 메서드를 물려주는 역할만 함
  - 인터페이스는 계약, 일종의 약속으로 최소한의 계약사항을 명시할 뿐이다.
  - 의무적으로 구현해야 하는 메서드의 목록을 인터페이스로 작성하여 자식 클래스들이 강제로 구현하도록 함

추상메서드와 상수만 멤버

인터페이스

- 인터페이스 선언

```
interface  인터페이스 이름
{
        //메서드 선언부;
        public static final 타입 상수이름 = 값;
        public abstract 메서드이름(매개변수목록);
}
```

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

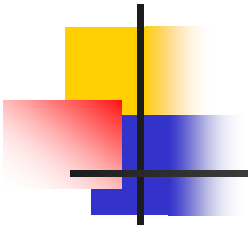
```
interface PlayingCard{
        public static final int SPADE = 4;
        final int DIAMOND = 3;
        static int HEART = 2;
        int CLOVER = 1;

        public abstract String getCardNumber();
        String getCardKind();
}
```

인터페이스의 구현

- 인터페이스도 추상 클래스처럼 그 자체로는 인스턴스를 생성할 수 없으며, 자신에 정의된 추상 메서드의 몸통을 만들어 주는 클래스를 작성해야 함
- 인터페이스는 구현한다는 의미의 키워드 'implements'를 사용

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

class Fighter implements Fightable {
        public void move(int x, int y){ ... }
        public void attack(unit u){ ... }
}
```

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

        /* 지정된 대상(u)를 공격하는 기능의 메서드*/
        void attack(unit u);
}
```



인터페이스의 구현

- 구현하는 인터페이스의 메서드 중 일부만 구현한다면, 추상클래스로 선언되어야 함

```
abstract class Fighter implements Fightable {
        public void move(int x, int y){ ... }
}
```

인터페이스 내의 메서드 구현

- 인터페이스 내의 메서드 구현
  - 인터페이스(Interface)는 프로토타입들의 선언만이 있을 뿐 실제로 이 기능을 구현 (Implementation) 하는 것은 이 인터페이스(Interface)를 상속받은 클래스가 하는 것

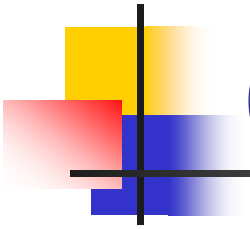
```
interface IMyInterface {
    void iMethod();
}
class MyClass implements IMyInterface {
    public void iMethod(){
      System.out.println("인터페이스의 메서드를 구현");
    }
}
class MainClass{
    public static void main(String[] args) {
      MyClass mc = new MyClass();
      mc.iMethod();
      IMyInterface imc = mc;
      imc.iMethod();
    }
}
```

```
인터페이스의 메서드를 구현
인터페이스의 메서드를 구현
```

- 인터페이스도 하나의 클래스이므로 인터페이스를 구현한 자식클래스 입장에서는 인터페이스가 부모클래스가 됨
- MyClass는 인터페이스를 상속했기 때문에 인터페이스로 업캐스팅 될 수 있음

예제

```
interface IAnimal
{
	public abstract void sound();
	void display();  //접근 제한자(public abstract) 생략 가능
}
class Cat implements IAnimal
{	//IAnimal 인터페이스를 구현하는 자식 클래스
	public void sound()
	{
		System.out.println("야옹~~");
	}
	public void display()
	{
		System.out.println("Cat 클래스!");
	}
}//class
abstract class Dog implements IAnimal
{
	//부모 인터페이스인  IAnimal의 추상메서드 중 display()는 구현하지 않았으므로
	//추상 클래스가 된다
	public void sound()
	{
		System.out.println("멍멍!!");
	}
}//class
```



예제

```
class Cow implements IAnimal
{
	public void sound()
	{
		System.out.println("음메~");
	}

	public void display()
	{
		System.out.println("Cow 클래스!!!!");
	}

	/*
	void display() //에러: 부모의 메서드를 오버라이딩할 때 접근 제한자는 부모보다 넓어야 함
	{	}
	*/

}//class
```

예제

```
//IAnimal obj = new IAnimal();  //인터페이스는 객체생성 불가
Cat c = new Cat();
c.sound();
c.display();

//Dog d = new Dog();  //추상 클래스는 객체 생성 불가
//다형성
IAnimal obj = new Cat();
obj.sound();
obj.display();
```

```
class InterfaceTest1
{
	public static void main(String[] args)
	{
		System.out.println("1. 고양이, 2. 소, 3. 종료");
		Scanner sc = new Scanner(System.in);
		int type = sc.nextInt();

		IAnimal ani=null;
		if (type==1){
			ani = new Cat();
		}else if (type==2){
			ani = new Cow();
		}else if (type==3){
			return;
		}else{
			System.out.println("잘못 입력!");
			return;
		}

		System.out.println("=====다형성 이용 : 사용자가 선택한 내용=======");
		ani.display();
		ani.sound();
	}
}
```

인터페이스의 구현

- 상속과 구현을 동시에 할 수도 있음

```
class Fighter extends Unit implements Fightable {
        public void move(int x, int y){ … }
        public void attack(unit u){ … }
}
```

```
class TV{
        public void onTV(){
                System.out.println("TV 영상 출력 중");
        }
}
interface Computer{
        public void dataReceive();
}

class IPTV extends TV implements Computer{
```

인터페이스의 상속

- 인터페이스는 인터페이스로부터만 상속받을 수 있으며, 클래스와는 달리 다중상속 가능
  - 여러 개의 인터페이스로부터 상속을 받는 것이 가능

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

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

interface Fightable extends Movable, Attackable{ }
```

인터페이스의 상속

- 인터페이스들 사이의 상속
  - 인터페이스끼리 상속가능
  - 인터페이스끼리 상속해서 더 큰 인터페이스를 만든다는 의미
  - 인터페이스들끼리는 다중상속과 단일 상속 가능
  - extends를 사용해서 인터페이스끼리 상속하면 인터페이스가 됨

```
interface IA{
        void sayA();
}
interface IB{
        void sayB();
}
interface IC extends IB{
        void sayC();    //인터페이스 단일 상속
}
interface ID extends IA, IC {
   //인터페이스들의 다중상속
}
```

인터페이스의 구현

- 인터페이스는 **구현(Implementaton)**을 목적으로 함
  - 인터페이스가 하나의 클래스가 되기 위해서는 반드시 모든 내부의 구현되지 않은 구성요소들을 전부 구현해야 함

```
interface IA{
        void sayA();
}
interface IB{
        void sayB();
}
interface IC extends IB{
        void sayC();  //인터페이스 단일 상속
}
interface ID extends IA, IC {    //인터페이스들의 다중상속
        void sayD();
}
interface IM{
        void sayM();
}
```

```
class Test implements ID, IM {
        //모든 인터페이스의 구현
        public void sayA(){  }
        public void sayB(){  }
        public void sayC(){  }
        public void sayD(){  }
        public void sayM(){  }
}
```

다중 상속

- 하나의 클래스가 여러 개의 인터페이스를 상속받아 구현할 수 있다

```
interface TV
{
        public abstract void onTV();
}

interface Computer
{
        public abstract void dataReceive();
}

class IPTV implements TV, Computer
{

}
```

예제 1

IPTV 는 일종의 TV 이다
IPTV 는 일종의 Computer 이다 => IPTV는 TV이자 Computer 이다
=> IPTV 클래스는 TV 클래스와 Computer 클래스를 동시에 상속하는 형태가 적절

```
class TV{
	public void onTV(){
		System.out.println("TV 영상 출력 중");
	}
}
interface Computer{
	public void dataReceive();
}

class IPTV extends TV implements Computer{
	public void dataReceive(){
		System.out.println("영상 데이터 수신 중");
	}
	public void powerOn(){
	//인터넷으로부터 방송 데이터를 입력 받아 TV에 출력
		dataReceive();
		onTV();
	}
}
```

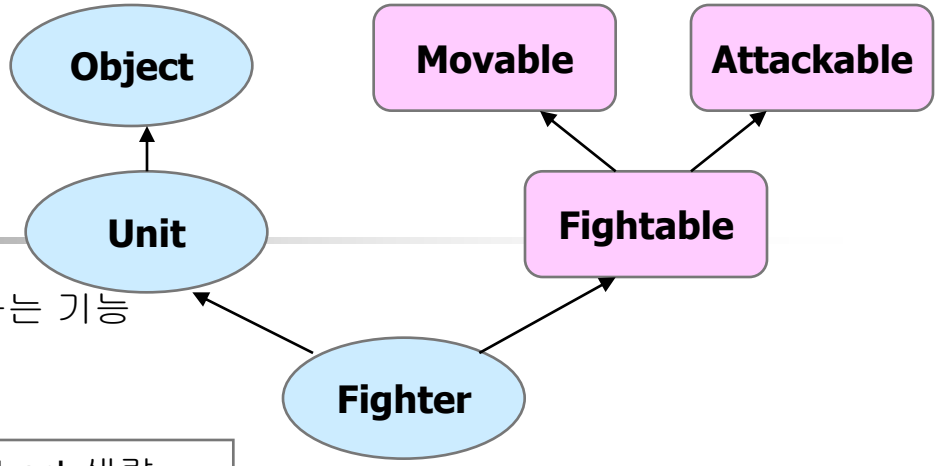
```
class MultiInheriAlternative1
{
	public static void main(String[] args)
	{
		IPTV iptv=new IPTV();
		iptv.powerOn();

		/*
		TV tv=new IPTV();
		tv.onTV();
		Computer comp=new IPTV();
		comp.dataReceive();*/
	}
}
```

```
영상 데이터 수신 중
TV 영상 출력 중
```

예제 2

```
interface TV{
    public abstract void onTV();
}

interface Computer{
    public abstract void dataReceive();
}

class IPTV implements TV, Computer{
    public void onTV(){
        System.out.println("TV 영상 출력 중");
    }

    public void dataReceive(){
        System.out.println("영상 데이터 수신 중");
    }

    public void powerOn(){
        dataReceive();
        onTV();
    }
}
```

예제




```
interface Attackable{//지정된 대상(unit)을 공격하는 기능
	public abstract void attack(Unit unit);
}

interface Movable{
	void move(int x, int y);  //지정된 위치(x,y)로 이동하는 기능
}

//인터페이스끼리의 상속도 가능-extends 키워드 이용, 다중상속이 가능함
interface Fightable extends Attackable, Movable{
}

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

	Unit(int x, int y, int currentHP){
		this.x=x;
		this.y=y;
		this.currentHP=currentHP;
	}
}//class
```

public abstract 생략

예제

오버라이딩할 때는 부모의 메서드보다 넓은 범위의 접근 제어자를 지정해야 한다

=> 인터페이스 구현시 메서드는 반드시 public 으로 해야 함

```
//클래스 상속과 인터페이스 구현을 동시에 할 수 있다
class Fighter extends Unit implements Fightable{
	Fighter(int x, int y, int currentHP){
		super(x, y, currentHP);
	}

	public void move(int x, int y){
		this.x=x;
		this.y=y;
		System.out.println(x+","+y+"위치로 이동합니다");
	}

	public void attack(Unit unit){
		System.out.println(unit.x +","+ unit.y +"위치에 있고, "
				+unit.currentHP+"의 체력을 갖는 유닛을 공격합니다");
	}

}//class
```

```
class FighterTest {
    public static void main(String[] args) {
        Fighter f = new Fighter(10, 20, 300);
        f.move(40, 50);

        Unit u = new Fighter(40,50,200);
        f.attack(u);
        f.move(70, 80);

        Fighter f2 = new Fighter(70, 80, 150);
        f.attack(f2);

        //자식은 부모의 인스턴스이기도 함  : 자식 instanceof 부모 => true
        System.out.println(f instanceof Unit);

        if (f instanceof Unit){
                System.out.println("f는 Unit의 인스턴스이다");
        }
        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.");
        }
    }
}
```

```
40,50위치로 이동합니다
40,50위치에 있고, 200의 체력을 갖는 유닛을 공격합니다
70,80위치로 이동합니다
70,80위치에 있고, 150의 체력을 갖는 유닛을 공격합니다
true
f는 Unit의 인스턴스이다
f는 Fightable인터페이스를 구현했습니다.
f는 Movable인터페이스를 구현했습니다.
f는 Attackable인터페이스를 구현했습니다.
f는 Object클래스의 자손입니다.
```

인터페이스를 이용한 다형성

- 다형성
  - 자식 클래스의 인스턴스를 부모 타입의 참조변수로 참조하는 것이 가능하다
- 인터페이스도 이를 구현한 클래스의 부모
  - 해당 인터페이스 타입의 참조변수로 이를 구현한 클래스의 인스턴스를 참조할 수 있다
  - 인터페이스 타입으로의 형변환도 가능

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

- 인터페이스는 메서드의 매개변수의 타입으로 사용될 수 있다.
- 또한 메서드의 리턴타입으로 인터페이스의 타입을 지정할 수 있다.

```
void attack(Fightable f){
...
}
```

```
public Fightable method(){
    .....
    return new Fighter();
}
```

인터페이스의 장점

- 1. 표준화가 가능하다.
  - 프로젝트에 사용되는 기본 틀을 인터페이스로 작성한 다음, 개발자들에게 인터페이스를 구현하여 프로그램을 작성하도록 함으로써 보다 일관되고 정형화된 프로그램의 개발이 가능하다.
- 2. 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.
- 3. 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

- 직접적인 관계의 두 클래스(A-B)

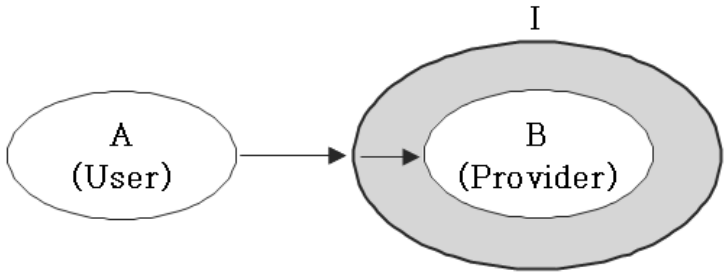
```
class A {
      public void methodA(B b) {
            b.methodB();
      }
  /*public void methodA(C c) {
            c.methodB();
      } */
}

class B {
      public void methodB() {
            System.out.println("B Class에서 구현한
methodB()");
      }
}

class C {
      public void methodB() {
            System.out.println("C Class에서 구현한
methodB()");
      }
}
```

- 클래스 A 는 클래스 B의 인스턴스를 생성하고 메서드를 호출한다
- 이 두 클래스는 직접적인 관계에 있다

단점 : 직접적인 관계의 두 클래스는 한쪽(Provider)이 변경되면, 이를 사용하는 다른 한 쪽(User)도 변경되어야 함

```
class InterfaceTest {
      public static void main(String args[]) {
            A a = new A();
            a.methodA(new B());
            //a.methodA(new C());
      }
}
```

인터페이스의 이해



- 간접적인 관계의 두 클래스(A-I-B)

```
interface I{
        public abstract void methodB();
}
class A {
    public void methodA(I i) {
        i.methodB();
    }
}

class B implements I{
    public void methodB() {
        System.out.println("B Class에서 구현한 methodB()");
    }
}

class C implements I{
    public void methodB() {
        System.out.println("C Class에서 구현한 methodB()");
    }
}
```

• 클래스 A 가 클래스 B를 직접 호출하지 않고 인터페이스를 매개체로 하는 경우

⇒클래스 A는 여전히 클래스 B의 메서드를 호출하지만, 클래스 A는 인터페이스 I 하고만 직접적인 관계에 있기 때문에 클래스 B의 변경에 영향을 받지 않음
⇒ 클래스 A는 오직 직접적인 관계에 있는 인터페이스 I의 영향만 받음

```
class InterfaceTest_2 {
    public static void main(String args[]) {
        A a = new A();
        a.methodA(new B());
        a.methodA(new C());
    }
}
```

interface 기반의 상수 표현

- 요일을 상수로 선언하는 경우

```
class Week
{
        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;
}

interface Week2
{
        int MON=1, TUE=2, WED=3, THU=4, FRI=5, SAT=6, SUN=7;
}
```

인터페이스 내에 존재하는 변수는 무조건 **public static final** 로 선언된다는 특성을 활용.
자바에서 사용하는 다수의 상수 선언 방식

```
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, 5.금요일, 6.토요일, 7.일요일");
            System.out.print("선택: ");
            Scanner sc=new Scanner(System.in);
            int sel=sc.nextInt();
            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.");
            } } }
```

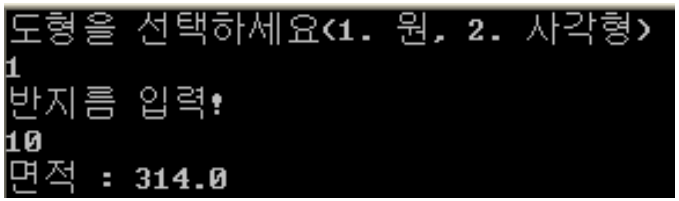
```
오늘의 요일을 선택하세요.
1.월요일, 2.화요일, 3.수요일, 4.목요일, 5.금요일, 6.토요일, 7.일요일
선택: 2
프로젝트 기획 회의가 있습니다.
```

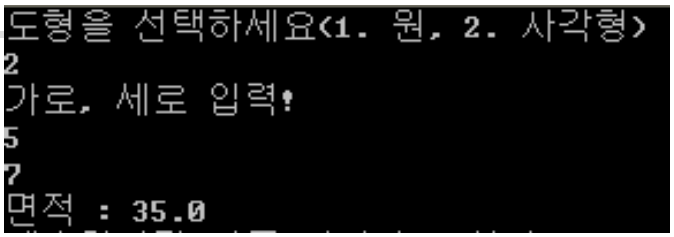
실습1-인터페이스

```
도형을 선택하세요(1. 원, 2. 사각형)
2
사각형을 그립니다
사각형을 지웁니다
```

- IShape 인터페이스
  - draw() 메서드
  - delete() 메서드
- Circle(자식 클래스)
  - draw() 메서드 구현
  - delete() 메서드 구현
- Rect(자식 클래스)
  - draw() 메서드 구현
  - delete() 메서드 구현
- Main()에서
  - 사용자로부터 원, 사각형 중 선택하게 하고, 객체 생성 후 draw(), delete() 메서드 호출
    - 다형성 이용 – IShape 가 Circle, Rect를 참조하도록

실습2





- Shape 인터페이스 (부모)
  - 메서드 : findArea()
- Circle 클래스 (자식 클래스)
  - 멤버변수 : 반지름 => 생성자에서 초기화
  - 메서드 : findArea() => 부모 클래스의 메서드 오버라이딩
    - 원의 면적을 구해서 return (원의 면적 : 3.14 * 반지름 * 반지름)
- Rectangle 클래스 (자식 클래스)
  - 멤버변수 : 가로, 세로 => 생성자에서 초기화
  - 메서드 : findArea() => 부모 클래스의 메서드 오버라이딩
    - 사각형의 면적을 구해서 return (가로*세로)
- 다형성 이용

- [1] 인터페이스 이용
- [2] abstract을 이용

실습3- 급여관리 시스템

- 직원의 고용형태
  - 고용직, 임시직
  - 급여계산방식의 차이
    - 고용직(Permanent) – 연봉제(매달 기본급여가 정해져 있다)
    - 임시직(Temporary) – 일한시간 * 시간당 급여
- 고용인들이 공통적으로 지니고 있어야 하는 멤버들을 모아서 Employee 클래스로 추상화시킨다

- 고용직 클래스 (Permanent) – Employee클래스를 상속받음
  - 직원의 이름과 급여정보를 저장하기 위한 클래스
  - 필드 : 이름, 기본급여, 보너스
  - 생성자, getter/setter
  - 메서드 : 급여계산 => 기본급여 + 보너스
- 임시직 클래스 정의 (Temporary) –Employee클래스를 상속받음
  - 필드 : 이름, 일한시간(time), 시간당 급여(pay)
  - 생성자, getter/setter
  - 메서드 : 급여계산 => time*pay

실습3-추상클래스 이용

- Employee 추상 클래스
  - 필드 : 이름
  - 생성자, getter/setter
  - 메서드 : 급여계산 getPay()
- Permanent – 고용직
  - 필드 : 기본급여(salary), 보너스
  - 생성자, getter/setter
  - 메서드 : 급여계산 getPay() 오버라이딩 =>기본급여+보너스
- Temporary – 임시직
  - 필드 : 일한시간(time), 시간당 급여(pay)
  - 생성자, getter/setter
  - 메서드 : 급여계산 getPay() 오버라이딩 => time*pay

- Main()
  - 다형성 이용
  - 결과 화면 출력시 getter/setter 이용하여 출력

실습3

```
고용형태 - 고용직(P), 임시직(T)을 입력하세요
p
이름, 기본급여, 보너스를 입력하세요
홍길동
2500000
300000
------------------------------------------

고용형태:고용직
이름:홍길동
급여:2800000
```

```
고용형태 - 고용직(P), 임시직(T)을 입력하세요
t
이름, 일한시간, 시간당급여를 입력하세요
김연아
20
65000
------------------------------------------------

고용형태:임시직
이름:김연아
급여:1300000
```