

1. 상속
2. 오버라이딩
3. package와 import

객체지향개념 II-1

4. 제어자
5. 다형성

객체지향개념 II-2

6. 추상클래스
7. 인터페이스

객체지향개념 II-3

6. 추상클래스(abstract class)

6.1 추상클래스(abstract class)란?

6.2 추상메서드(abstract method)란?

6.3 추상클래스의 작성

7. 인터페이스(interface)

7.1 인터페이스(interface)란?

7.2 인터페이스의 작성

7.3 인터페이스의 상속

7.4 인터페이스의 구현

7.5 인터페이스를 이용한 다형성

7.6 인터페이스의 장점

7.7 인터페이스의 이해

6. 추상클래스 (abstract class)

6.1 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.

6.2 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 */ }  
}
```

6.3 추상클래스의 작성

- 여러 클래스에 공통적으로 사용될 수 있는 추상클래스를 바로 작성하거나 기존클래스의 공통 부분을 뽑아서 추상클래스를 만든다.

```
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.*/ }
}
```

```
abstract class Unit {
    int x, y;
    abstract void move(int x, int y);
    void stop() { /* 현재 위치에 정지 */ }
}

class Marine extends Unit {    // 보병
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void stimPack()          { /* 스팀팩을 사용한다.*/ }
}

class Tank extends Unit {      // 탱크
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void changeMode()       { /* 공격모드를 변환한다. */ }
}

class Dropship extends Unit {  // 수송선
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void load()             { /* 선택된 대상을 태운다.*/ }
    void unload()           { /* 선택된 대상을 내린다.*/ }
}
```

```
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);
}
```

추상메서드가 호출되는 것이 아니라 각 자손들에 실제로 구현된 move(int x, int y)가 호출된다.

7. 인터페이스(interface)

7.1 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.

7.2 인터페이스의 작성

- '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 가능하다.

- 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;        // public static final int DIAMOND = 3;  
    static int HEART = 2;         // public static final int HEART = 2;  
    int CLOVER = 1;               // public static final int CLOVER = 1;  
  
    public abstract String getCardNumber();  
    String getCardKind(); // public abstract String getCardKind();  
}
```

7.3 인터페이스의 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.

7.4 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 */ }  
}
```

7.5 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();  
}
```

7.6 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.

7.6 인터페이스의 장점 - 예제

```

interface Repairable {}

class GroundUnit extends Unit {
    GroundUnit(int hp) {
        super(hp);
    }
}

class AirUnit extends Unit {
    AirUnit(int hp) {
        super(hp);
    }
}

class Unit {
    int hitPoint;
    final int MAX_HP;
    Unit(int hp) {
        MAX_HP = hp;
    }
}

```

```

public static void main(String[] args) {
    Tank tank = new Tank();
    Marine marine = new Marine();
    SCV scv = new SCV();
}

```

```

    scv.repair(tank); // SCV가 Tank를 수리한다.
    // scv.repair(marine); // 에러!!!
}

```

```

class Tank extends GroundUnit implements Repairable {
    Tank() {
        super(150); // Tank의 HP는 150이다.
        hitPoint = MAX_HP;
    }

    public String toString() {
        return "Tank";
    }
}

```

```

class Marine extends GroundUnit {
    Marine() {
        super(40);
        hitPoint = MAX_HP;
    }
}

```

```

class SCV extends GroundUnit implements Repairable {
    SCV() {
        super(60);
        hitPoint = MAX_HP;
    }

    void repair(Repairable r) {
        if (r instanceof Unit) {
            Unit u = (Unit)r;
            while(u.hitPoint != u.MAX_HP) {
                u.hitPoint++; // Unit의 HP를 증가시킨다.
            }
        }
    }

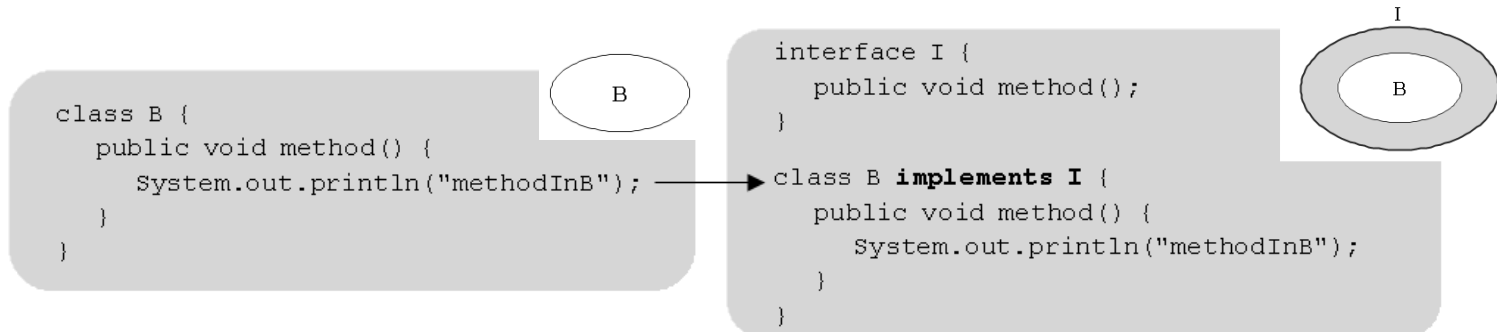
    // repair(Repairable r) {
}

```

7.7 인터페이스의 이해(1/3)

▶ 인터페이스는...

- 두 대상(객체) 간의 '연결, 대화, 소통'을 돕는 '중간 역할'을 한다.
- 선언(설계)과 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.



7.7 인터페이스의 이해(2/3)

- ▶ 직접적인 관계의 두 클래스(A-B) ▶ 간접적인 관계의 두 클래스(A-I-B)

```
class A {  
    public void methodA(B b) {  
        b.methodB();  
    }  
}
```

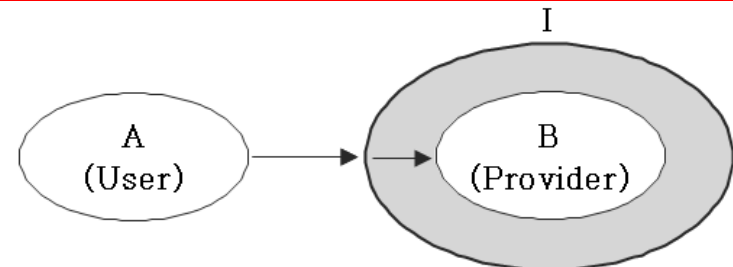
```
class B {  
    public void methodB() {  
        System.out.println("methodB()");  
    }  
}
```

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

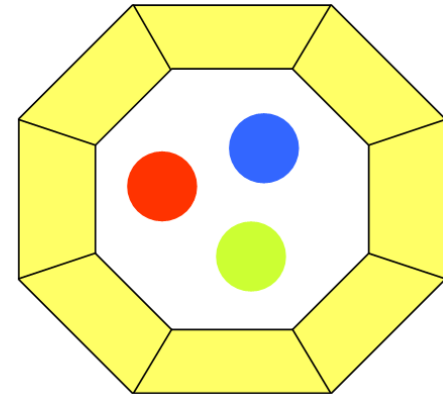
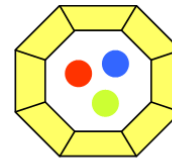
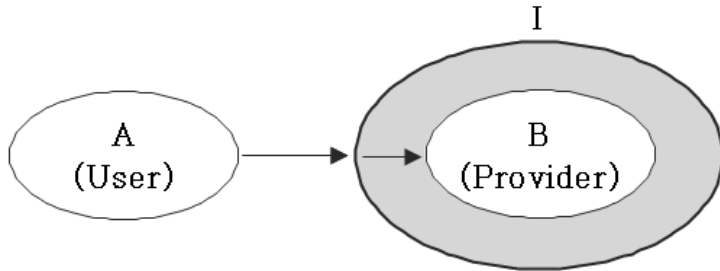
```
class A {  
    public void methodA(I i) {  
        i.methodB();  
    }  
}
```

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

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



7.7 인터페이스의 이해(3/3)



```
public class Time {  
    private int hour;  
    private int minute;  
    private int second;  
  
    public int getHour() { return hour; }  
    public void setHour(int h) {  
        if (h < 0 || h > 23) return;  
        hour=h;  
    }  
    public int getMinute() { return minute; }  
    public void setMinute(int m) {  
        if (m < 0 || m > 59) return;  
        minute=m;  
    }  
    public int getSecond() { return second; }  
    public void setSecond(int s) {  
        if (s < 0 || s > 59) return;  
        second=s;  
    }  
}
```

```
public interface TimeIntf {  
    public int getHour();  
    public void setHour(int h);  
  
    public int getMinute();  
    public void setMinute(int m);  
  
    public int getSecond();  
    public void setSecond(int s);  
}
```