

능동적 사고 방식의

java

강사 박주병

Park Ju Byeong

Park Ju Byeong



Part11 다형성

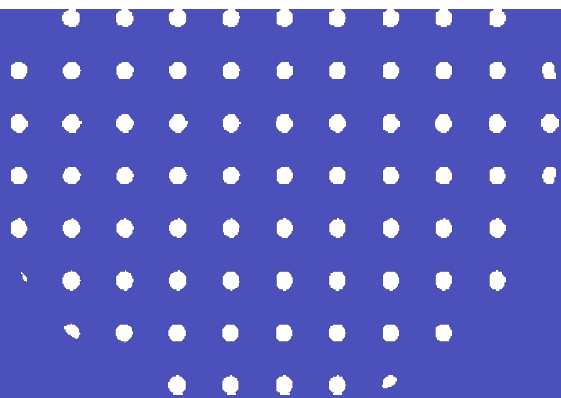


01 다형성

02 인터페이스

03 내부 클래스

04 실습 문제



01 —

다형성



```
Car car = new Car();  
OilCar oilCar = new OilCar();
```



```
Car car = new OilCar();
```

상속관계라면 자식 객체를 가리킬 수 있다.

자식은 부모 클래스의 모든 정보를 가지고 있기에 가능하다.

무슨 차이가 있을까?

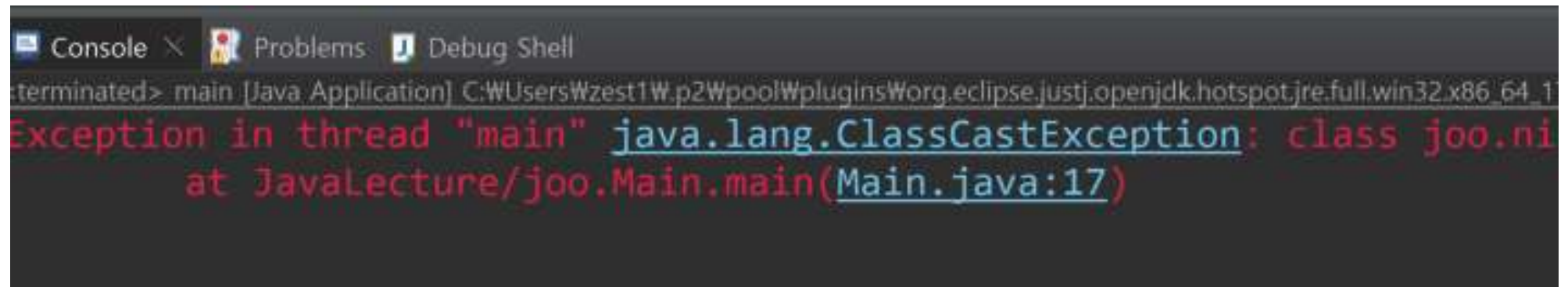
```
public class OilCar extends Car{  
    int Oil;  
    public OilCar()  
    {  
    }  
    boolean engineOilCheck()  
    {  
        return true;  
    }  
}
```

```
Car car = new OilCar();  
OilCar oilCar = new OilCar();  
car.Oil = 10;  
oilCar.Oil = 20;  
car.engineOilCheck();  
oilCar.engineOilCheck();
```

자식의 멤버들을 사용할수 없다.

```
OilCar oilCar = new Car();
```

```
OilCar car = (OilCar)new Car();
```



The screenshot shows the Eclipse IDE's console window. The title bar includes 'Console', 'Problems', and 'Debug Shell'. The console output shows a Java application running, followed by a red error message: 'Exception in thread "main" java.lang.ClassCastException: class joo.ni at JavaLecture/joo.Main.main(Main.java:17)'. The error message is partially cut off on the right side of the image.

```
terminated> main [Java Application] C:\Users\Wzest1W\p2Wpool\plugins\Worg.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_1
Exception in thread "main" java.lang.ClassCastException: class joo.ni
    at JavaLecture/joo.Main.main(Main.java:17)
```

OilCar의 입장에서는 필요한 정보가 충분하지 않다.

부모 -> 자식으로의 타입변환은 못하는가?

```
Car car = new OilCar();  
  
OilCar oilCar= (OilCar)car;  
  
oilCar.engineOilCheck();
```

← 부모로 형변환시 자식의 멤버를
사용하지는 못한다.
(기본형 처럼 데이터가 손실되는것은 아니다)

```
<terminated> Main [Java Application] C:\Users\USER5454\...  
엔진 오일을 점검 하였습니다.
```

Car

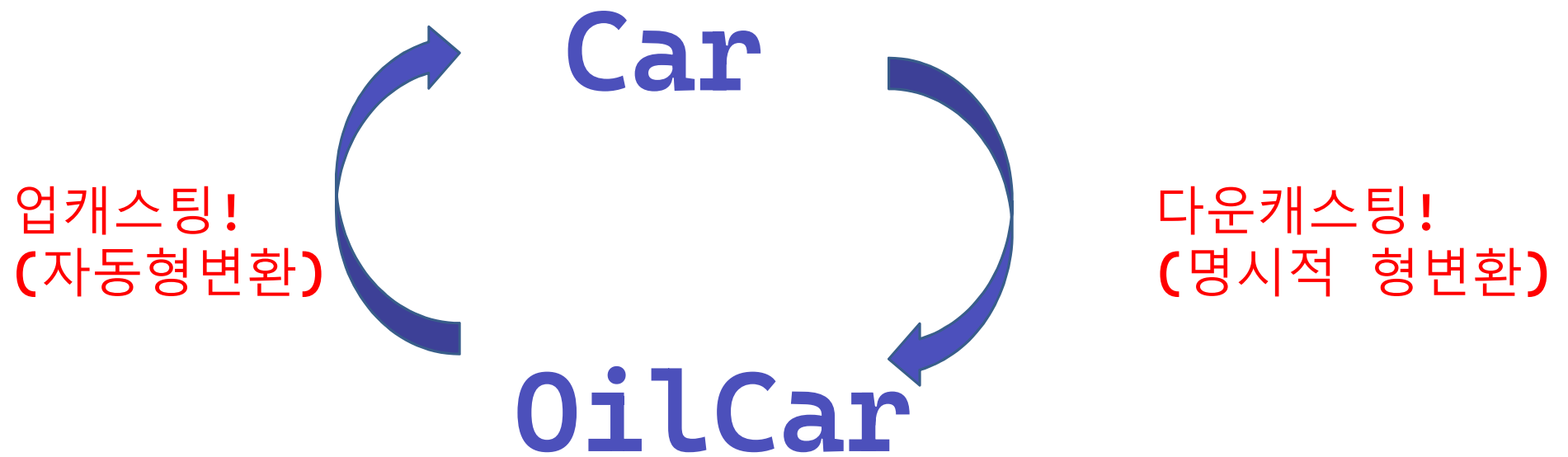
0X000A

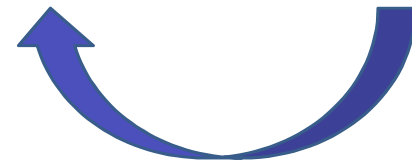
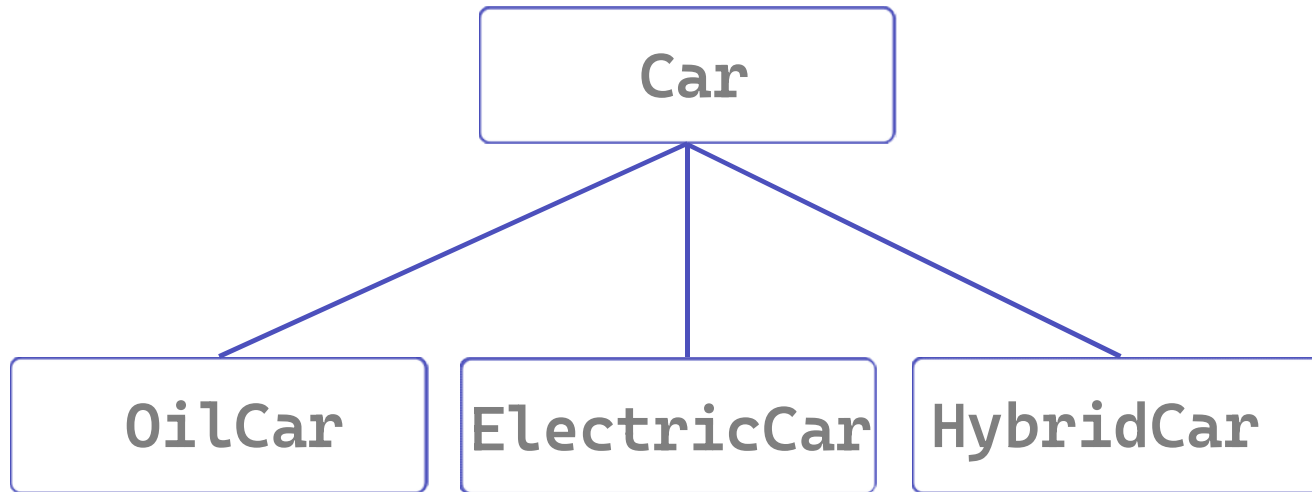
OilCar

0X000A

메모리주소	값
0x000A	speed
	Door[] doors
	Oil
	engineCheck()
	go()
	stop()

자식의 특성을
사용만 못할뿐





타입캐스팅 불가!

퀴즈

```
Car car1 = new OilCar();  
Car car2 = new Car();  
  
OilCar oilCar1 = (OilCar) car1;  
OilCar oilCar2 = (OilCar) car2;
```

실제 객체는 **Car**로 생성되어서 실행시 에러발생(런타임
에러)

instanceof 연산자

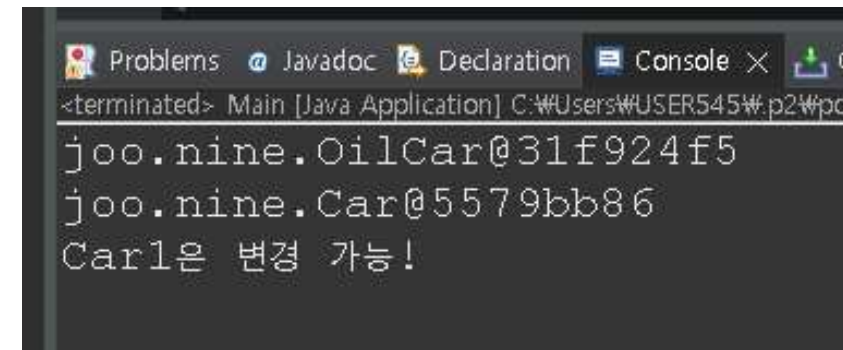
```
Car car1 = new OilCar();
Car car2 = new Car();

System.out.println(car1.toString());
System.out.println(car2.toString());

if(car1 instanceof OilCar)
{
    System.out.println("Car1은 변경 가능!");
    OilCar oilCar = (OilCar)car1;
}

if(car2 instanceof OilCar)
{
    System.out.println("Car2은 변경 가능!");
    OilCar oilCar = (OilCar)car2;
}
```

→ car1 이 OilCar로 형변환 해도
안전한가?

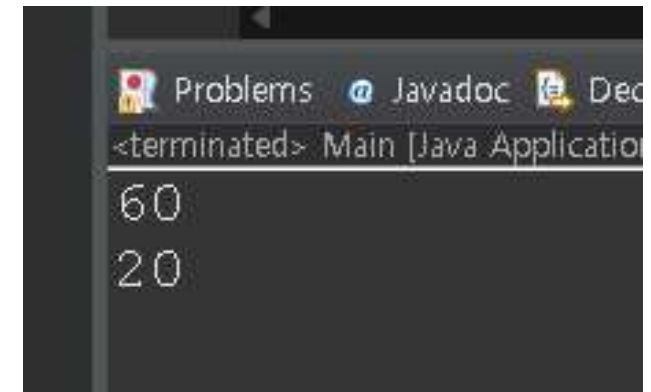


멤버변수 오버라이딩 후 참조변수의 타입에 따른 변화

```
3 public class Parent {  
4     public int age= 60;  
5 }
```

```
2  
3 public class Child extends Parent{  
4  
5     public int age=20;  
6 }  
7
```

```
Parent test1 = new Child();  
Child test2 = new Child();  
  
System.out.println(test1.age);  
System.out.println(test2.age);
```



Problems @ Javadoc Dec
<terminated> Main [Java Application]
60
20

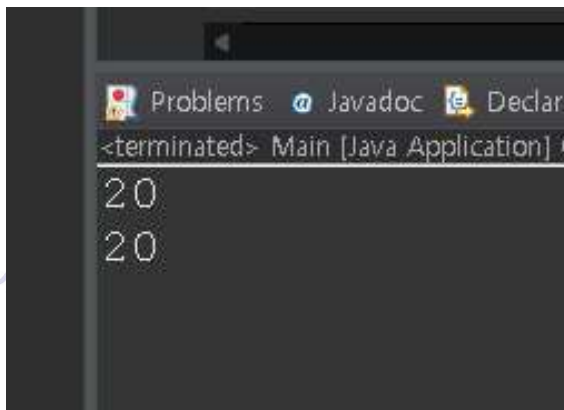
메서드의 경우 변수 타입에 관계 없이 무조건 실제
인스턴스의 메서드를 실행한다.

클래스를 만드는 개발자가 무조건 오버라이딩된 멤버변수를 쓰게 하고 싶다면?

```
2
3 public class Child extends Parent{
4
5     private int age=20;
6
7
8     public int getAge()
9     {
10         return this.age;
11     }
12 }
```

```
Parent test1 = new Child();
Child test2 = new Child();

System.out.println(test1.getAge());
System.out.println(test2.getAge());
```




```
Car car1 = new OilCar();
```

기능이 된다는 알겠는데...

그래서 이걸 왜 쓰는걸까..??

1. **OilCar**에 정의된 기능을 쓰지도 못하고

2. 참조변수 타입 = 객체타입 이 일치하지 않으니 실제 객체가 무슨 타입인지까지 고려해야 하는데..?

Marine의 공격 메서드

```
public void attack(Zergling target)
{
    target.hp -= (Power-target.armor);
    System.out.println("총을 쏩니다.");
}

public void attack(Marine target)
{
    target.hp -= (Power-target.armor);
    System.out.println("총을 쏩니다.");
}

public void attack(Zealot target)
{
    target.hp -= (Power-target.armor);
    System.out.println("총을 쏩니다.");
}
```



```
public void attack(Unit target)
{
    target.hp -= (Power-target.armor);
}
```

1. **Unit**을 상속받는 모든 클래스를 매개변수로 받을 수 있다.
2. **Unit**을 상속 받은 모든 클래스는 **HP**와 **armor**를 가지고 있다.

```

class Animal
{
    void move()
    {
        System.out.println("...");
    }
}

class Tiger extends Animal
{
    void move() {
        System.out.println("기어간다.");
    }
}

class Bird extends Animal
{
    void move() {
        System.out.println("날아간다.");
    }
}

```

```

35 // TODO Auto-generated method
36 Animal animal1 = new Bird();
37 Animal animal2 = new Tiger();
38
39 animal1.move();
40 animal2.move();
41
42
43 }
44
45 }
46

```

Problems Javadoc Declaration Console X Git Staging

<terminated> Main (2) [Java Application] C:\Users\zest1\p2\pool\plugins\org.ec
날아간다.
기어간다.

Park Ju Byeong

다형성

[polymorphism ]

다형성 (polymorphism)은 '여러가지 형태가 존재한다'라는 뜻이다. 이 용어는 학문 분야에 따라서 의미가 조금씩 다르다. 고전 생물학에서는 어떤 종의 군집 내에서 확연하게 다른 형태가 둘 또는 그 이상 존재하는 현상을 말한다. 즉 생명체의 다양성을 나타내는 말이다. 이에 비해 유전학에서 말하는 다형성은 어떤 군집 내에서 유전적 변이가 존재하는 현상을 의미한다. 분자생물학에서는 DNA 혹은 아미노산의 서열에서의 차이, 어떤 분자의 성질이나 기능에서의 차이를 말할 때 다형성이라는 용어를 사용한다.

목차

- 1.다형성의 어원
- 2.생물학에서 다형성
- 3.유전학과 분자생물학에서 다형성
- 4.참고문헌

다형성을 이용한 배열활용

```
2
3 public abstract class Unit {
4
5     int hp;
6     static int power;
7     static int armor;
8
9
10    public static int count=0;
11
12
13    Unit()
14    {
15        count++;
16    }
17
18    public void Attack(Unit target)
19    {
20        System.out.println("공격 합니다.");
21    }
22
```

```
Unit unitList[] = new Unit[10];
unitList[0]= new Marine();
unitList[1]= new Zergling();
unitList[2]= new Zealot();
for(Unit unit :unitList)
{
    unit.Attack(null);
}
```

서로 다른 객체타입 이더라도 다형성을 통해 일관성 있게 사용할 수 있다.

Unit 클래스에서 Attack의 역할은 무엇인가? 반드시 있어야 하는가?

```
2
3 public abstract class Unit {
4
5     int hp;
6     static int power;
7     static int armor;
8
9
10    public static int count=0;
11
12
13    Unit()
14    {
15        count++;
16    }
17
18    public void Attack(Unit target)
19    {
20        System.out.println("공격 합니다.");
21    }
22
```



```
public void Attack(Unit target)
{
}
}
```

공격 방식은 유닛마다 제각각 이다. 결국 **Unit** 클래스에서는
어떻게 공격할 것인지 구현 할수가 없다.

```
Unit unitList[] = new Unit[10];

unitList[0] = new Marine();
unitList[1] = new Zergling();
unitList[2] = new Zealot();
unitList[3] = new Tank();

for(Unit unit : unitList)
{
    unit.Attack(null);
}
}
```

```
// 팀장님이 만들라고 해서 만드는데... 어떤 기능들 넣어야 하지...?
// 아 몰라... 상속 받으라는것만 해놓고 내맘대로 해야지...
// 퇴근하고 싶다...
public class Tank extends Unit{

    public void Attack()
    {
        System.out.println("통통포 공격");
    }
}
```

Tank 객체의 차례일때 **Attack**이 정상적으로 되지 않는다.
게다가 컴파일은 되는 런타임에러이다.

Unit을 상속 받은 자식클래스에 내가 원하는
메서드 시그니처대로 강제로 오버라이딩 하게
만들순 없는가?


```

3 public abstract class Unit {
4
5     int hp;
6     static int power;
7     static int armor;
8
9
10    public static int count=0;
11
12
13    Unit()
14    {
15        count++;
16    }
17
18    public abstract void attack(Unit target);
19

```

```

// 팀장님이 만들라고 해서 만드는데... 어떤 기능들 넣어야 하지...?
// 아 상속받으니 시그니처가 다 정해져 있고 무조건 만들어야 하네?
public class Tank extends Unit{

    @Override
    public void attack(Unit target) {
        // TODO Auto-generated method stub

    }

}

```

Park Ju Byeong

Park Ju Byeong

추상화의 주된 이유는 다형성에서 안정적인 **100%**
보장된 코드를 구현하기 위함이다.

```
Unit unitList[] = new Unit[10];
```

```
unitList[0] = new Marine();
```

```
unitList[1] = new Zergling();
```

```
unitList[2] = new Zealot();
```

```
for(Unit unit : unitList)
```

```
{
```

```
    unit.Attack(null);
```

```
}
```

겉 모습은 같으나 객체에 따라 다르게 동작하게
만드는것이 다형성의 핵심이다.

자료구조 List

```
Unit unitList[] = new Unit[10];

unitList[0] = new Marine();
unitList[1] = new Zergling();
unitList[2] = new Zealot();

for (Unit unit : unitList)
{
    unit.Attack(null);
}
```



```
List<Unit> list = new ArrayList<Unit>();

list.add(new Marine());
list.add(new Zergling());
list.add(new Zealot());
list.add(new Tank());

for (Unit unit : list)
{
    unit.attack(null);
}

list.remove(2); // 질럿 삭제
```

1. 개수의 제한이 없으며 크기 변경이 용이하다.
2. 크기 변경시 성능이 우수하다.

```
List<Unit> linkedList = new LinkedList<Unit>();
```

실습문제1

1. 쇼핑몰 장바구니를 구현해보자

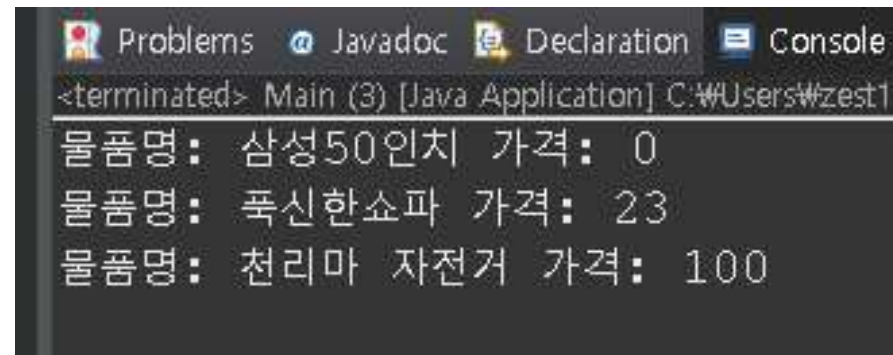
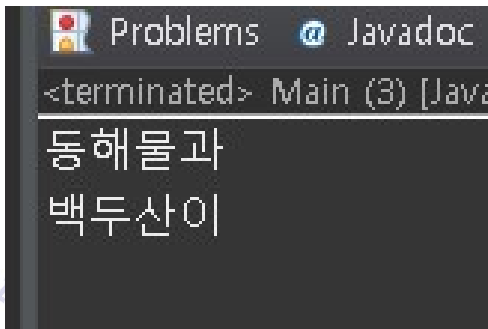
- 클래스 간의 관계를 생각하여 상속,포함 관계대로 구성하여 보자.
- 클래스 : ShoppingBasket , Item, TV, Sofa, Bicycle
- Item은 배열대신 List를 사용해보자.
ex) `List<Item> itemList = new ArrayList<Item>();`
- 장바구니는 여러 아이템들을 가진다.
- Tv,Sofa,Bicycle은 어떤 클래스를 상속 받아야 하는가?

```
ShoppingBasket basket = new ShoppingBasket();  
  
basket.itemList.add(new TV());  
basket.itemList.add(new Sofa());
```


2. 1번에서 선언한 클래스들의 내부를 구현해보자.

- 모든 물품들은 바코드 넘버(int), 이름(String), 가격(int), 마일리지 비율(int)을 가지고 있어야 한다.
- 멤버변수는 생성자를 통해 초기화 하자
- 물품 가격의 1% 만큼 마일리지가 적립된다. 단 TV는 이벤트로 인해 3%만큼 마일리지 적립된다.
- ShoppingBasket 클래스에 현재 장바구니에 있는 물품이름과 가격을 문자열로 반환하는 String getInfoList() 메서드를 만들자

```
ShoppingBasket sb = new ShoppingBasket();  
  
// TODO Auto-generated method stub  
System.out.println("동해물과" + System.lineSeparator() + "백두산이");  
sb.itemList.add(new TV(1, "삼성50인치", 0));  
sb.itemList.add(new Sofa(2, "폭신한쇼파", 23));  
sb.itemList.add(new Bicycle(3, "천리마 자전거", 100));  
  
System.out.println(sb.getInfoList());
```



3. 기능추가

- 현재 장바구니에 담겨 있는 물품들의 금액 합계를 반환 하는 메서드를 만들자

`int getTotalPrice()`

- 현재 장바구니 대로 구매를 한다면 마일리지가 얼마나 쌓이는지 반환하는 메서드를 만들자

`int getTotalMileage()`

- 현재 `Bicycle`은 재고가 없어 장바구니에 담을 수 없다. `ShoppingBasket`에 물건을 담을때 필터링하자

hint : itemList를 private으로 막아야 한다, instanceof 활용해야한다.

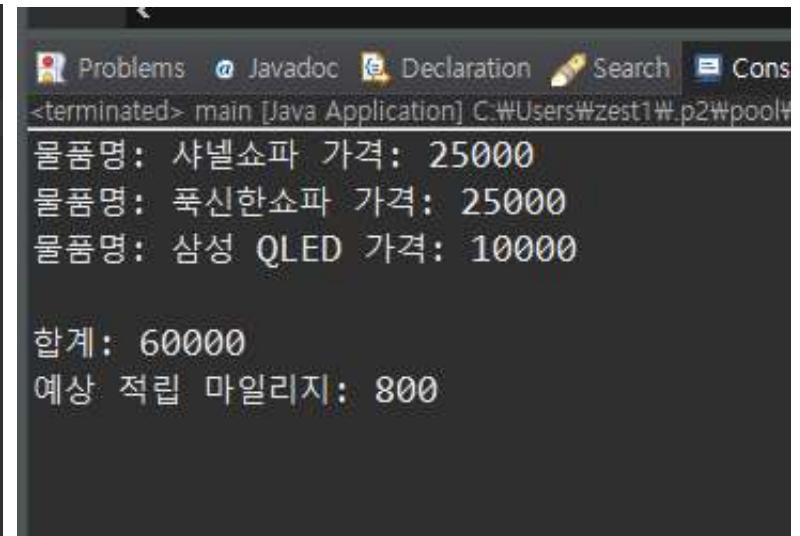
```
ShoppingBasket shopping = new ShoppingBasket();

shopping.itemList.add(new Sofa(1, "샤넬소파", 25000));
shopping.itemList.add(new Sofa(2, "죽신한소파", 25000));
shopping.itemList.add(new TV(3, "삼성 QLED", 10000));
shopping.itemList.add(new Bicycle(1, "빠른 자전거", 5000));
*/

shopping.addItem(new Sofa(1, "샤넬소파", 25000));
shopping.addItem(new Sofa(2, "죽신한소파", 25000));
shopping.addItem(new TV(3, "삼성 QLED", 10000));
shopping.addItem(new Bicycle(1, "빠른 자전거", 5000));

System.out.println(shopping.getInfoList());

System.out.println("합계: "+shopping.getTotalPrice());
System.out.println("예상 적립 마일리지: "+shopping.getTotalMileage());
```



```
<terminated> main [Java Application] C:\Users\zest1\p2\pool+
물품명: 샤넬소파 가격: 25000
물품명: 죽신한소파 가격: 25000
물품명: 삼성 QLED 가격: 10000

합계: 60000
예상 적립 마일리지: 800
```

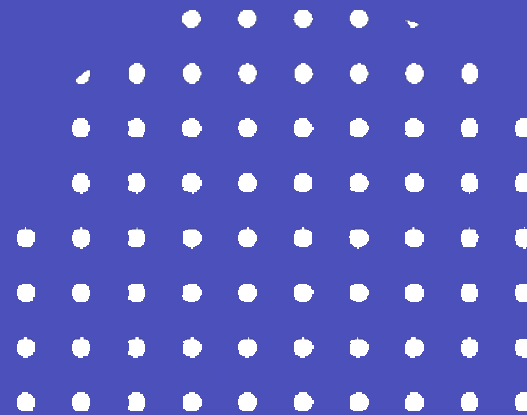
Park Ju Byeong

```
}  
    200m200.000(200m);
```

Part

— 02

인터페이스



인터페이스

1. 클래스와 동일하나 키워드만 **interface**로 바뀐다.
2. 전부 추상메서드이다. 따라서 객체 생성 불가
3. **JDK1.8** 버전 이상부터 상수를 허용한다.(그 이하는 오직 추상메서드만 존재할수 있다)

```
3 public interface interfaceTest {  
4  
5     // 멤버변수는 public static final 이어야 한다.  
6     public static final int TEST = 30;  
7  
8     //제어자를 생략해도 public static final 이 자동적용 된다.  
9     int MAX = 30;  
10  
11     //추상메서드 여야 한다.  
12     public abstract void test();  
13     //제어자를 생략해도 public abstract가 적용 된다.  
14     void MAX();  
15  
16 }
```

인터페이스의 상속

인터페이스는 인터페이스만 상속 받을수 있다.
(따라서 **Object**를 상속받지 않는다)

```
3 public interface interfaceTest extends Sofa{  
4
```

```
3 public interface ParentImpl {  
4  
5 }  
6
```

```
3 public interface interfaceTest extends ParentImpl{  
4
```



인터페이스의 상속

클래스와 다르게 다중 상속을 허용 한다.

```
public interface interfaceTest extends ParentImpl, GrandParentImpl {
```

인터페이스의 구현

인터페이스를 클래스가 구현 할때는 **implements**를 사용한다.

```
public class ClassTest implements ParentImpl{  
  
}  
|
```

implements = 인터페이스를 구현할때 (상속+내부구현)
extends = 상속받을때

인터페이스의 구현

일부분만 구현하면 해당 자식은 추상클래스가 되어야 한다.

```
2
3 public class ClassTest implements ParentImpl{
4
5
6     @Override
7     public void a() {
8         // TODO Auto-generated method stub
9
10    }
11 }
```

```
2
3 public abstract class ClassTest implements ParentImpl{
4
5
6     @Override
7     public void a() {
8         // TODO Auto-generated method stub
9
10    }
11 }
```

인터페이스의 구현

```
public class ClassTest extends Student implements ParentImpl {  
    |  
    @Override  
    public void a() {  
        // TODO Auto-generated method stub  
    }  
    @Override  
    public void b() {  
        // TODO Auto-generated method stub  
    }  
    @Override  
    public void c() {  
        // TODO Auto-generated method stub  
    }  
}
```

클래스를 상속 받는다. 인터페이스를 구현한다.

인터페이스의 활용

```
interface Moveable
{
    void Move();
}
```

```
class People implements Moveable
{
    @Override
    public void Move() {
        System.out.println("걷는다.");
    }
}
```

```
4 class Tiger implements Moveable
5 {
6     @Override
7     public void Move() {
8         // TODO Auto-generated method stub
9         System.out.println("기어간다");
10    }
11 }
```

```
class Bird implements Moveable
{
    @Override
    public void Move() {
        System.out.println("날아간다");
    }
}
```

```
List<Moveable> list = new ArrayList<Moveable>();

list.add(new People());
list.add(new Tiger());
list.add(new Bird());

for(Moveable obj : list)
    obj.Move();
```

추상메서드만 가지고 있는 추상클래스랑 차이가 없는데?

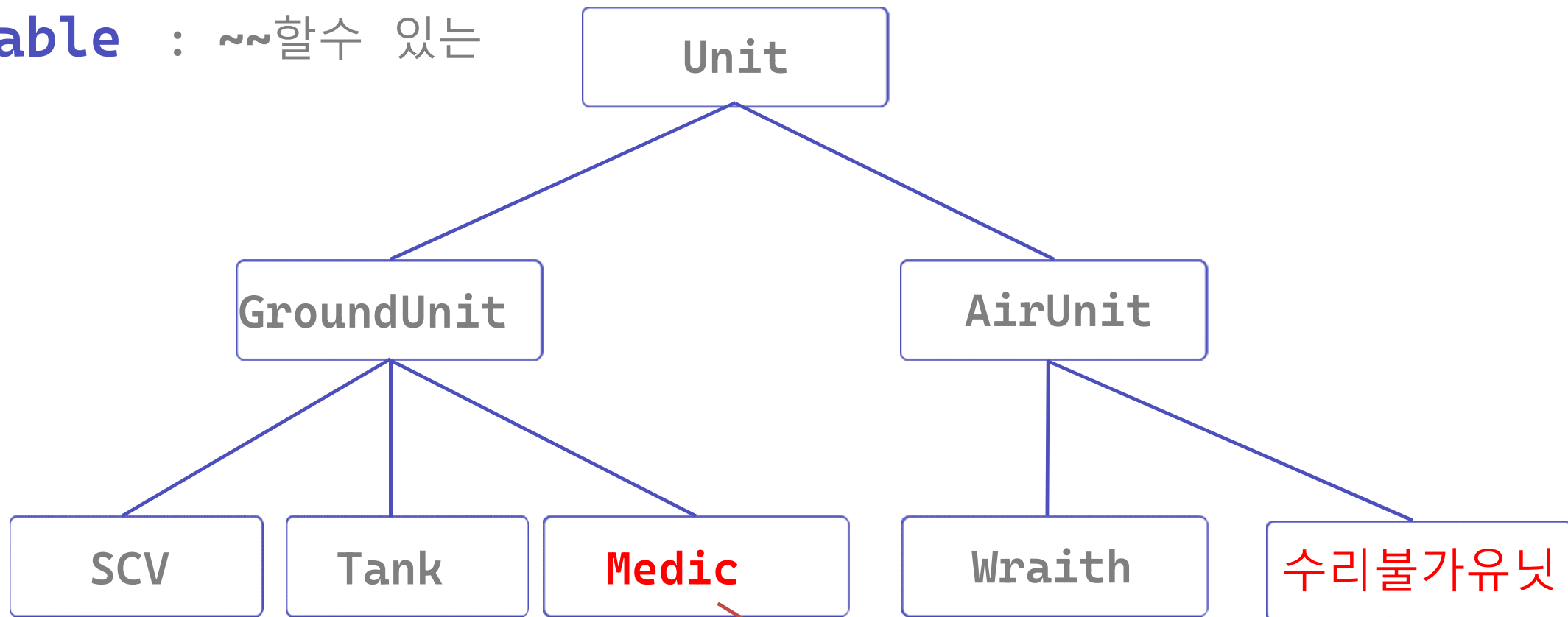
인터페이스의 네이밍

1.able 로 끝나는 경우가 많다. (문법적으로 정해진것은 아님)

```
1 public class ClassTest extends Student implements Serializable ,Closeable ,Appendable {
2
3     @Override
4     public Appendable append(char c) throws IOException {
5         // TODO Auto-generated method stub
6         return null;
7     }
8
9     @Override
10    public Appendable append(CharSequence csq) throws IOException {
11        // TODO Auto-generated method stub
12        return null;
13    }
14
15    @Override
16    public Appendable append(CharSequence csq, int start, int end) throws IOException {
17        // TODO Auto-generated method stub
18        return null;
19    }
20
21    @Override
22    public void close() throws IOException {
23        // TODO Auto-generated method stub
24    }
25 }
```

왜 **able**을 접미사로 붙이는걸까?

able : ~~할수 있는



```
void repair(Unit unit)
{
}
```

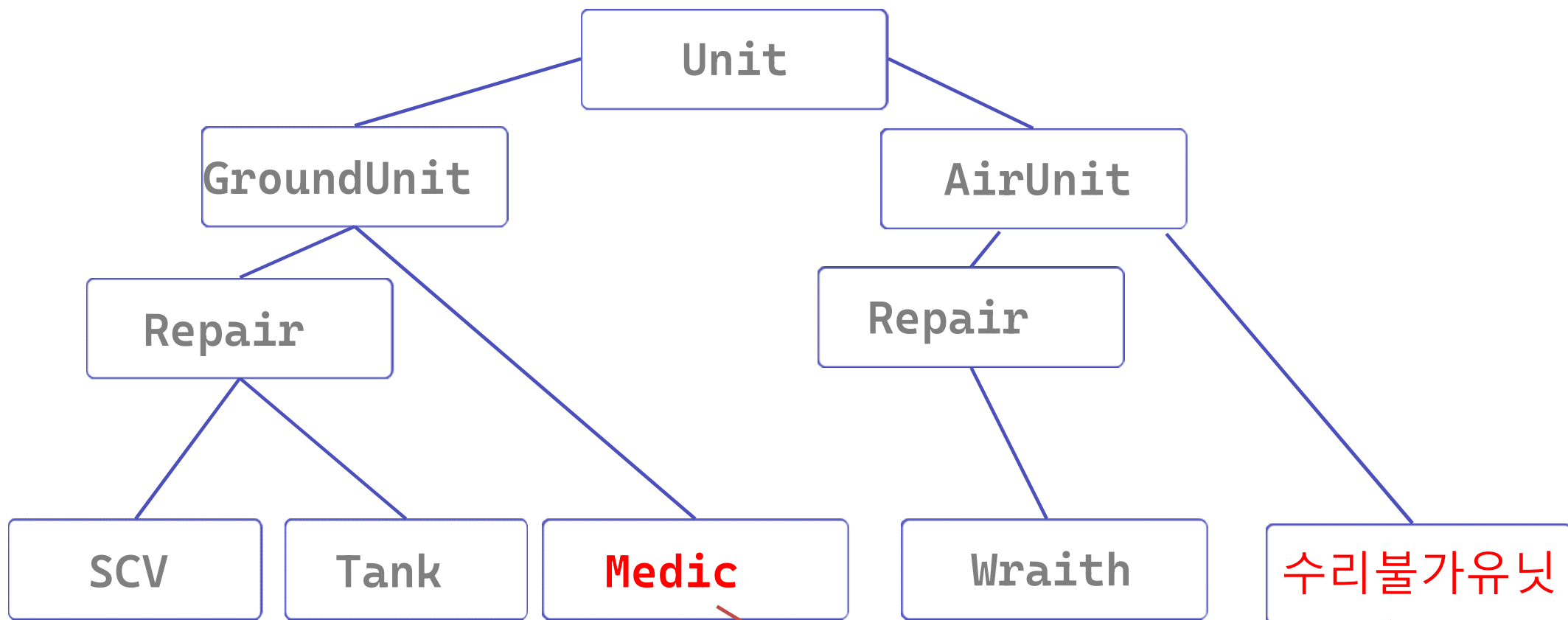
수리 불가능

Park Ju Byeong

```
void repair(Unit unit)
{
    if(unit instanceof Medic)
    {
        System.out.println("수리가 불가능 합니다!!");
        return ;
    }
}
```

수리 불가능한 유닛이 생길때마다 **if**문이 늘어난다..

유닛이 **100**개라면..??



```
void repair(Unit unit)
{
}
```

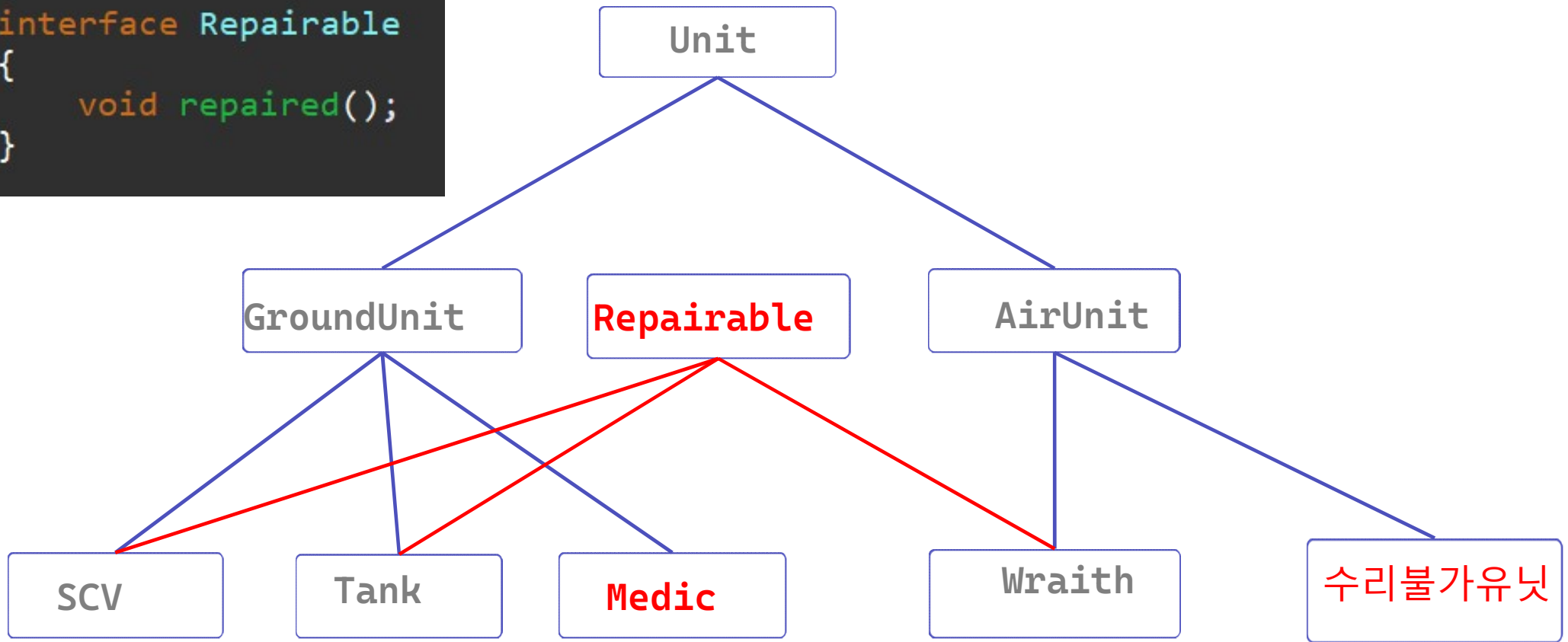
수리
불가능

Park Ju Byeong

```

4 interface Repairable
5 {
6     void repaired();
7 }
8

```




```

public void repair(Repairable unit)
{
    //해당 유닛은 Repairable 인터페이스를 상속받아 repaired 메서드를 구현해냈다.
    //유닛마다 각자의 수리속도와 방법이 다를텐데 SCV입장에선 고려할 필요가 없다.이것이 객체지향!
    unit.repaired();
}

```

기존의 클래스 관계도를 유지하면서
특정 기능별로 묶어 분류 하는것이
가능.

```
public class Tank extends Unit implements Repairable{  
  
    @Override  
    public void attack(Unit target) {  
        // TODO Auto-generated method stub  
    }  
  
    @Override  
    public void repaired() {  
        this.hp++;  
    }  
}
```



기존의 **extends**를 그대로 두기에 클래스 관계를 유지한채로
기능 위주로 다시 그룹을 묶을수 있다.

1. 왜 **able**을 접미사로 붙이는걸까?

클래스 마다 **repairable** 특수 기능을 덧붙이는 효과이다.

2. 추상메서드만 가지고 있는 추상클래스랑 차이가 없는데?

클래스를 상속 받는(**extends**) 와 인터페이스를 구현하는(**implement**)는 차이가 아주 크다
상속은 **딱 한번** 할 수 있고 **실질적인 기능을** 상속해줘야 할 때 쓰인다.

즉 클래스 관계에서 중심축이 되는 뼈대가 되는 연결고리이다.

반면 인터페이스는 클래스 상속을 통해 이뤄진 뼈대에 기능별로 느슨하게 분류를 덧붙이는 것이다.

즉 추상이 아닌 구현된 메서드를 물려 줘야 하는 강한 의존관계라면 상속을 해주고 그게 아니라면 인터페이스를 붙이는것 이 좋다.

인터페이스의 **static** 메서드

JDK1.8 버전부터 가능하다.

```
interface Moveable
{
    void Move();

    static void eat()
    {
        System.out.println("먹는다");
    };
}
```

인터페이스 default 메서드

```
interface Moveable
{
    void Move();
    void eat();
}

class People implements Moveable
{
    @Override
    public void Move() {
        System.out.println("걷는다.");
    }
}

class Tiger implements Moveable
{
    @Override
    public void Move() {
        // TODO Auto-generated method stub
        System.out.println("기어간다");
    }
}
```

사용중 나중에 추가된
메서드

기존에 개발해놓은 클래스들이 구현을
안했으니 에러가 발생한다.

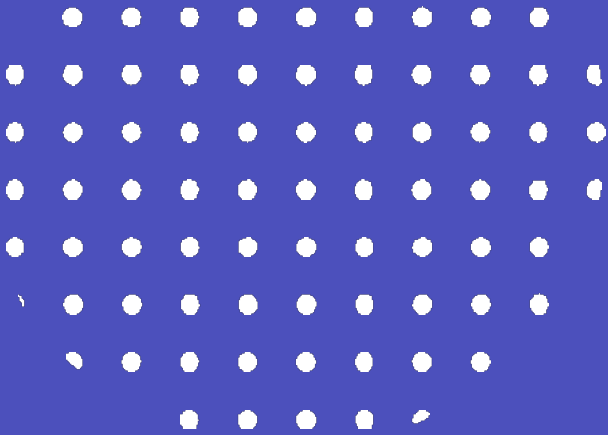
인터페이스 default 메서드

```
interface Moveable
{
    void Move();
    default void eat(){};
}
```

default 키워드 , {} 중괄호
추가

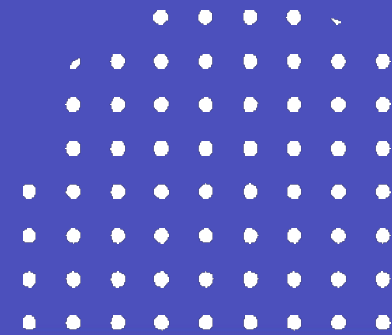
```
class People implements Moveable
{
    @Override
    public void Move() {
        System.out.println("걷는다.");
    }
}
```

```
class Tiger implements Moveable
{
    @Override
    public void Move() {
        // TODO Auto-generated method stub
        System.out.println("기어간다");
    }
}
```



03

내부클래스



내부 클래스

```
7
8 class animal
9 {
10     class InnerClass
11     {
12
13     }
14
15     static class InnerClass2
16     {
17
18     }
19
20     public void eat()
21     {
22         class InnerClass
23         {
24
25         }
26
27         System.out.println("동물이 먹는다.");
28     }
29 }
30 }
31
```

클래스 내부에서 클래스를 선언할 수 있다.

```
1 class animal
2 {
3
4     int a;
5     class InnerClass
6     {
7
8         void test()
9         {
10             a= 10;
11             System.out.println("내부 클래스 메서드");
12         }
13     }
14
15     void exMethod()
16     {
17         InnerClass innerClass = new InnerClass();
18         innerClass.test();
19     }
20 }
```

→ 내부 클래스 선언

→ 내부 클래스 사용

내부 클래스

1. 라이프사이클과 스코프는 변수와 동일하게 적용된다.
2. 일반적인 클래스와 동일하게 **abstract**, **final** 적용가능
3. 변수 처럼 **static** 선언도 가능, (내부 클래스를 객체 생성 없이 클래스 이름으로 사용한다)

익명 클래스

```
class test
{
    Object t = new Object()
    {
        void method()
        {
            System.out.println("익명클래스 내부");
        }
    };
}
```

→ 이름이 없는 익명 클래스 선언과
동시에 생성

???? 이름 있는데요? **Object** 클래스를 객체 생성한거 아닌가요?

```

class 이름있는클래스 extends Object
{
    void method()
    {
        System.out.println("이 클래스의 이름은 이름있는 클래스");
    }
}

class test
{
    Object t = new 이름있는클래스();
}

```

```

class test
{
    Object t = new Object()
    {
        void method()
        {
            System.out.println("익명클래스 내부");
        }
    };
}

```

새가 이동하는 방법을 외부에서 결정해줬으면 좋겠어!

```
class Bird implements Moveable
{
    Moveable howToMove;

    Bird(Moveable move)
    {
        howToMove = move;
    }

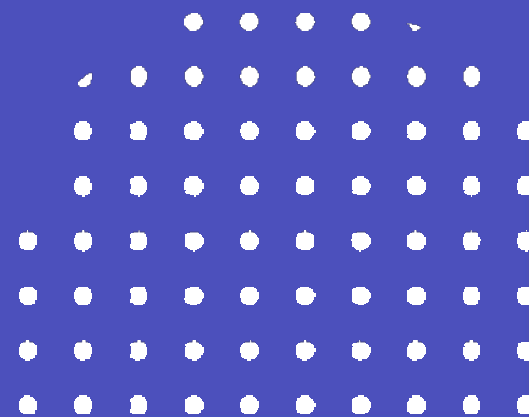
    @Override
    public void Move() {
        howToMove.Move();
    }
}
```

```
Bird chicken = new Bird(
    new Moveable()
    {
        @Override
        public void Move() {
            System.out.println("걸어간다.");
        }
    }
);

Bird duck = new Bird(
    new Moveable()
    {
        @Override
        public void Move() {
            System.out.println("둥둥 떠다닌다.");
        }
    }
);
```

— 04

실습문제



실습문제2

1. **User, Weapon, Repairable, Sword, Gun, Punch** 클래스를 만들고 각각의 관계에 맞게 클래스 설계를 하자.

Weapon: 추상 클래스 **Repairable** : 인터페이스

- **User**는 **Weapon**을 가진다.
- **Sword, Gun**은 수리가 가능하다.

2. 아래의 요구사항대로 클래스를 만들어 보자

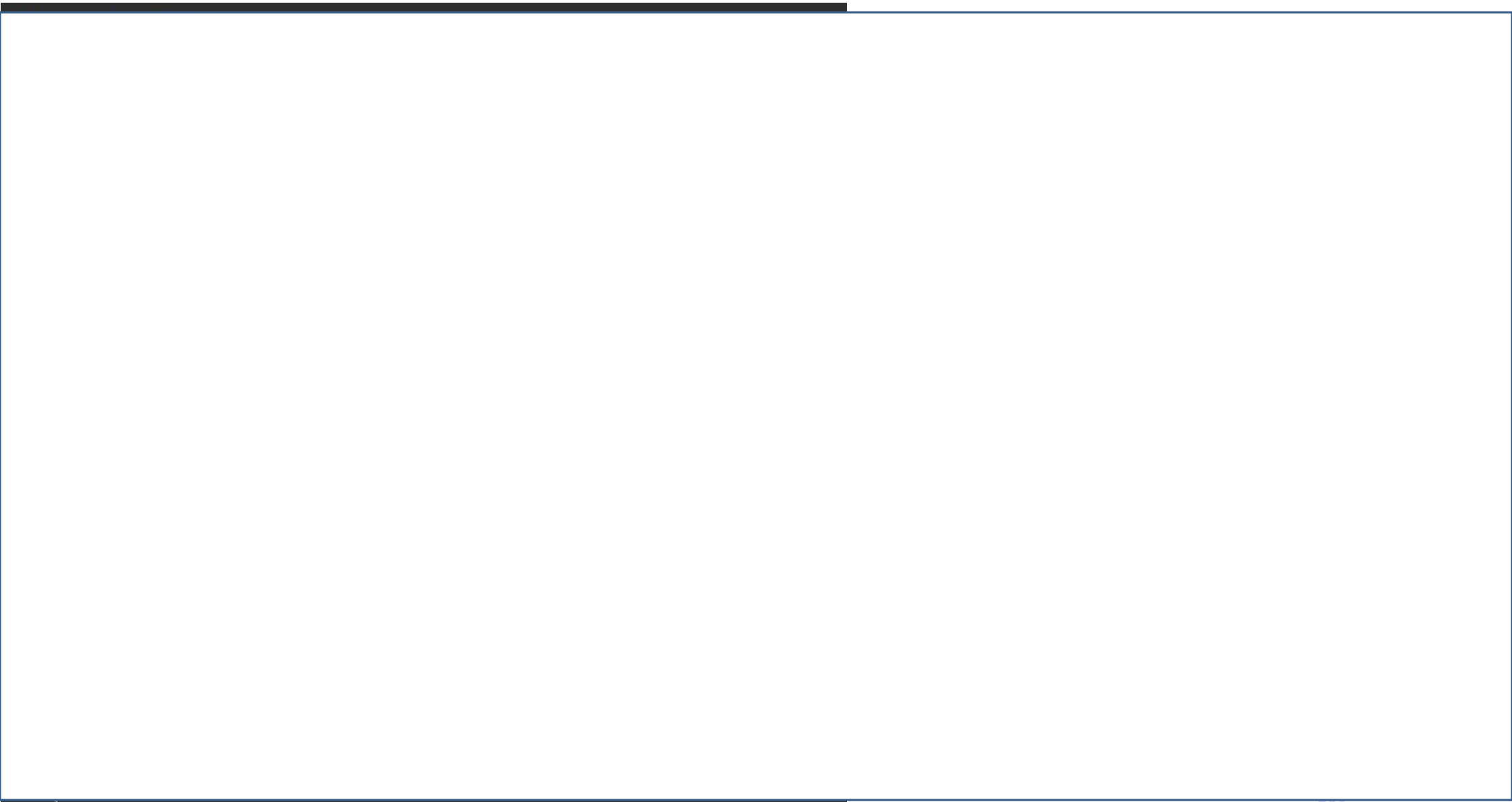
- User는 String id, int hp, void attack(User), toString() 을 구현한다.
- User는 생성자로 멤버변수 및 무기 객체를 받는다.
- User attack 메서드 내부에서 누가 누구를 공격했는지 출력하자.
- Weapon은 int damage, int durability, toString() 그리고 생성자로 멤버변수를 초기화한다.
- 모든 무기들은 한번 공격시 내구도가 1씩 차감된다.
- 내구도가 없다면 공격할 수 없고 그럴 경우 **sysout**으로 내구도가 없음을 알려주자.

```
User user1 = new User("랭킹1위가자",100,new Gun(10,5));
User user2 = new User("똥겜망해라",70,new Sword(15,10));

user1.attack(user2);
user1.attack(user2);
user1.attack(user2);
user1.attack(user2);
user1.attack(user2);
user1.attack(user2);

System.out.println(user1.toString());
System.out.println(user2.toString());
```

```
랭킹1위가자님이 똥겜망해라님을 공격하였습니다.
랭킹1위가자님이 똥겜망해라님을 공격하였습니다.
랭킹1위가자님이 똥겜망해라님을 공격하였습니다.
랭킹1위가자님이 똥겜망해라님을 공격하였습니다.
랭킹1위가자님이 똥겜망해라님을 공격하였습니다.
랭킹1위가자님이 똥겜망해라님을 공격하였습니다.
내구도가 없어 데미지가 0이 들어갔습니다.
아이디: 랭킹1위가자 체력: 100 무기 : {데미지: 10 내구도: 0}
아이디: 똥겜망해라 체력: 20 무기 : {데미지: 15 내구도: 10}
```



```
}  
}
```

Part

3. BlackSmith(대장장이) 클래스를 추가하여 무기를 수리해보자.

- BlackSmith 클래스에 `void reaire(Weapon)` 메서드를 구현하자.
- Sword, Gun은 수리가 가능하다.
- 수리가 불가능한 무기가 매개변수로 오면 `sysout`으로 거부 메시지를 출력하자
- `Reparable` 인터페이스를 활용하자.

```
System.out.println(user1.toString());
System.out.println(user2.toString());

BlackSmith bm = new BlackSmith();

//유저1의 무기를 대장장이에게 맡긴다.
bm.reaire(user1.Weapon);
System.out.println(user1.toString());

bm.reaire(user2.Weapon);
System.out.println(user2.toString());
```

```
아이디: 랭킹1위가자  체력: 100  무기 : {데미지: 10  내구도: 0}
아이디: 동검망해라  체력: 20  무기 : {데미지: 15  내구도: 10}
수리가 불가능한 무기입니다.
아이디: 랭킹1위가자  체력: 100  무기 : {데미지: 10  내구도: 0}
아이디: 동검망해라  체력: 20  무기 : {데미지: 15  내구도: 20}
```


4. BlackSmith(대장장이) 를 개선해보자

현재 **BlackSmith**는 어떻게 수리할지에 대한 구체적인 코드가 들어가 있다. 그러나 무기마다 수리하는 방법이 다를것이고 내구도 또한 한번에 올라가는 양이 다를수 있다. 그러므로 해당 코드는 다른곳으로 옮겨야 한다.

무기마다 수리되는 양이 다르게 기능을 수정하시오!

Hint : `repaire`의 매개변수로 `Repairable` 타입을 받고 해당 인터페이스에 기능이 추가 되어야 한다.



THANK YOU



강사 박주병