

## 1. 다형성

다형성이란 사전적으로 동일종의 개체에서 다양한 특징이나 형질을 보이는 것을 뜻한다. 즉 쉽게 말해 겉은 똑같지만 내부적인 특성이 다르다는걸 뜻한다.

다형성은 객체지향 프로그래밍에서 아주 중요한 기능이다. 지금부터 배울 다형성을 이용하여 다양한 기능들을 구현하기 때문이다.

우선 다형성을 배우기에 앞서 다형성을 구현하기 위한 문법적 기능들을 하나씩 살펴보자.\

### 1.1 객체의 타입캐스팅

#### 1.1.1 부모참조변수 = 자식객체

지금까지 객체를 생성하고 그것을 담아둘 참조변수는 생성한 객체와 동일한 타입으로 만들었어야 했다.

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

그림 1

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

그림 2

하지만 그림2처럼 자식클래스로 만들어진 객체라면 부모타입의 참조변수에 넣는 것이 가능하다.

타입이 다른데 어떻게 가능한것일까? 기본형일 때 float, double 같은 데이터라도 int형에 넣을수 있는것과 같은 이유이다. 바로 참조변수의 타입이 필요로 하는 정보를 모두 가지고 있기 때문이다.

OilCar는 Car 클래스를 상속 받았다. 따라서 Car 클래스에 있는 모든 멤버변수와 멤버메서드를 OilCar 역시 가지고 있는 것이다. 오버라이딩 하였다고 super를 배울 때 오버라이딩 하기 이전의 원본을 가지고 있다고 배웠다.

즉 OilCar 객체가 Car 타입의 참조변수에 들어간다 하더라도 Car타입의 참조변수 입장에서는 필요한 모든 정보를 가지고 있기에 당연히 담을 수 있는 것이다.

이렇게 타입이 다른 참조변수에 들어가면 어떻게 되는지 그림3을 살펴보자

Car의 경우 실질적으로 들어가 있는 객체가 OilCar 일지라도 참조변수가 Car 이기에 OilCar에만 존재하는 멤버는 사용할 수가 없다.

```
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.

그림 3

이제 반대의 경우를 보자. 자식타입의참조변수에 부모객체를 넣는다면 가능할까?

그림 4

그림 5

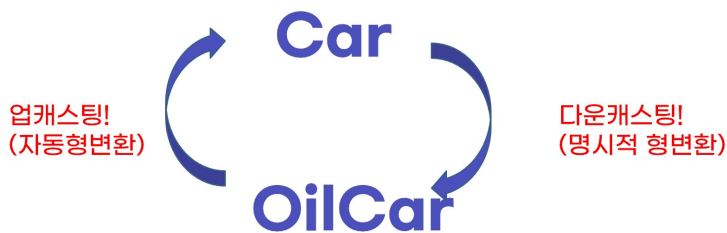
```
terminated> main [Java Application] C:\Users\Wzest1\p2\workspace\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.5.jdk\bin\java.exe
Exception in thread "main" java.lang.ClassCastException: class joo.ni
    at JavaLecture/joo.Main.main(Main.java:17)
```

그림 6

그렇다면 자식타입으로의 타입캐스팅은 무조건 안되는것인가? 그렇지 않다. 그림7 과 같이 실질적인 객체가 OilCar 이면 된다. 그렇다면 여기서 알수 있는 사실은 OilCar 객체가 Car타입의 참조변수에 의해 사용되어질 때 OilCar의 멤버를 사용만 못할 뿐이지 데이터는 그대로 라는 것이다. 이 상황에서 다시 자식타입으로 되돌아간다면 정보가 사라진 것이 아니기 OilCar만의 멤버들을 문제없이 사용 할 수 있는 것이다.

그림 7

지금까지 부모나 자식타입으로 타입캐스팅 되는 것은 봤다. 자식이 부모타입으로 캐스팅되는 것을 업캐스팅 이라고 부르고 그 반대는 다운캐스팅 이라고 부른다.



### 1.1.3 형제간의 타입캐스팅

그림9처럼 형제간의 타입캐스팅은 지원하지 않는다.

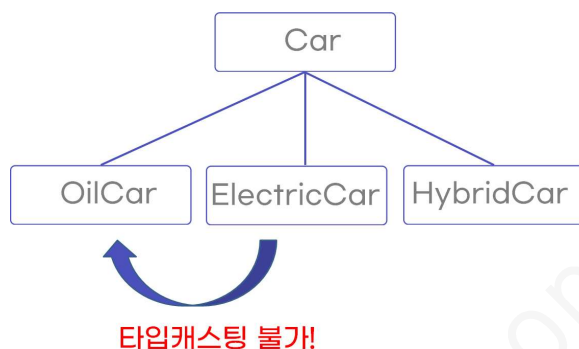


그림 9

### 1.1.4 instanceof 연산자

지금까지 내용을 보면 결국 참조변수보다도 실제 객체의 타입이 어떤지에 따라 런타임에러가 발생한다. 그런데 만약 코드의 길이가 길어진다면 수 많은 코드 중에 해당 참조변수가 가리키는 실질적인 객체의 타입이 무엇인지 파악하기가 힘들 것이다. 또한 프레임워크등을 사용한다면 의존주입으로 외부에서 객체를 생성하여 주는 경우가 많을 것이다.

따라서 참조변수가 가리키는 실제 객체의 타입이 무엇이 알아야될 필요가 있다.

이럴 때 사용 할 수 있는 키워드가 instanceof 이다. 그림10에서 car1이 가리키는 객체가 OilCar 로 형변환이 가능할 경우 true가 반환된다.

```
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?

그림 10

### 1.1.5 오버라이딩 후 참조변수 타입에 따른 변화

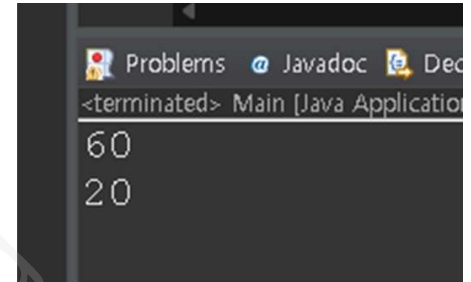
```
3 public class Parent {  
4     public int age= 60;  
5 }  
6
```

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

그림 11

그림11 과 같이 age 멤버변수를 상속받고 오버라이딩 하였다.그리고 참조변수를 다르게 사용했을 경우 어떤 멤버를 가리키는지 확인해보자.

```
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
```

test1의 경우 업캐스팅 이후 age 멤버변수를 출력 하였더니 60이 나왔다. 즉 참조변수가 부모의 타입이면 부모로부터 물려받은 멤버를 가리키게 된다. 마치 클래스내부에서 super를 쓰는 것과 비슷하다. 그리고 test2의 경우 자식에서 오버라이딩 해놓은 변수를 참조하고 있다.

메서드의 경우 참조변수의 타입에 관계없이 무조건 실제 인스턴스의 메서드를 실행한다.

그렇다면 클래스를 만드는 입장에서는 멤버변수를 오버라이딩 하였을 때 클래스를 사용하는쪽에서 어떤 참조변수를 쓰냐에 따라 상속받은 원본인지 오버라이딩된 변수인지가 달라진다. 이러면 클래스를 만든 사람의 의도와는 다르게 동작할수 있다. 그래서 원하는 멤버변수대로 유도하고 싶다면 getter, setter를 통해 this.멤버변수 혹은 super.멤버변수로 명확하게 값을 넘겨주면 된다.

### 1.1.6 업캐스팅의 활용

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

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

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



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

그림 13

그림13과 같이 다양한 객체를 받아서 처리 해야될 때 이들의 부모클래스를 매개변수로 받으면 코드중복을 막을수 있으며 해당 객체를 상속받은 모든 타입을 받을 수 있다.

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

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

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

또한 위의 그림처럼 객체배열을 사용할 때 업캐스팅을 이용한다면 하나의 배열에 여러 가지 타입의 객체를 받을수 있고 해당 배열을 사용할때도 코드의 변화가 필요없게 된다.

바로 이런 것 들을 두고 다형성 이라고 부른다. unit.Attack() 의 경우 코드는 동일한데 배열 안에 어떤 객체가 들어 있냐에 따라 액션이 달라지는 것이다.

### 1.1.7 메서드의 추상화

```
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)
{
}
}
```

그림 15

그림15를 보면 Unit의 Attack 메서드는 사실 메서드의 내용이 굳이 필요가 없다. 왜냐하면 자식들은 상속을 받아 자기에 맞게 오버라이딩하여 새롭게 만들기 때문이다. 그래서 오른쪽처럼 아무 내용도 없어도 영향을 받지 않는다.

그러나 위와 같이 그냥 내용만 비워둔다면 자식 클래스에서 Attack 메서드를 구현 하지 않아도 컴파일이 가능하다. 자식에서 구현하지 않았다면 부모의 비어있는 메서드를 상속받은것이고 다형성을 이용해 실행할 때 아무런 기능도 동작하지 않을 것이다. 물론 자식 클래스를 만드는 사람이 일부러 의도 하고 아무런 기능도 하지 않는 메서드를 만들었는것일수도 있다. 그러나 이게 의도한건지 까먹고 구현을 안한건지 알수가 없는 것이다.

Unit 클래스를 만드는 입장에서 자식들이 반드시 Attack을 구현했으면 한다면 앞에서 배웠던 추상메서드를 활용하면 된다.

```
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
```

이럴 경우 자식클래스는 반드시 오버라이딩해야지만 컴파일이 가능하며 만약 위처럼 아무기능도 안하는메서드를 만들고 싶었다면 오버라이딩하여 내용을 비워두면 된다. Unit 클래스를 만드는입장에선 자식에게 무조건 구현을 하라고 알려준 것이고 다형성을 통해 실행했을 때 자식 클래스를 만든 사람의도대로 동작 한다는걸 100% 확신 할 수 있다.



## 2. 자료구조 List

List 라는 자료 구조는 배열을 가변적으로 구현해놓은 것이다. 기존에 배열은 크기를 늘리거나 줄일려면 새로운 배열을 만들어서 요소를 하나씩 복사해줬어야 했다. 이러한 작업들을 직접 개발한다는 것은 매우 번거로운 일이다. Java에서는 자주 사용하는 자료구조들을 구현을 해 놓았다.



그림 17

그림17은 기존에 배열을 통해 다형성을 만들어 놓은 코드를 List를 이용하여 구현한 것이다.

List는 조금 뒤에서 배열 인터페이스라는 것이다. 지금은 클래스의 일종이라고만 알아두자.

그리고 `<Unit>` 이라고 되어 있는 것은 List의 요소안에 어떤 타입이 들어갈지를 지정하는 것이다. 이것 역시 뒤에서 배열 제네릭이라는 기능이다.(C++에서는 템플릿이라고 부른다.) 그리고 `List<Unit>` 타입의 참조변수 `list`에 `ArrayList<Unit>`이라는 객체를 만들어 초기화 해주고 있다. 그런데 가만히 보면 객체의 타입과 참조변수의 타입이 다르지 않은가? 그렇다 이 역시도 앞에서 배웠던 다형성인 것이다. `ArrayList` 클래스의 부모가 `List`인 것이다.

List를 상속받은 클래스는 `ArrayList`, `Vector`, `Stack`, `LinkedList`가 있다. 만약 여러분도 여러분만의 자료구조를 만들고 싶다면 List를 상속받아 다형성을 통해 사용해보자.

참조변수의 타입이 `ArrayList` 였다면 해당 코드는 `ArrayList`에 종속적일 수밖에 없다. 그런데 List를 사용했기에 객체를 주입하는 부분만 `ArrayList`에서 `LinkedList`로 변경되어도 전혀 영향이 없다. 다형성 덕분에 코드를 변경하지 않고도 객체를 바꿔가면서 데이터에 따라 유리한 자료구조를 선택하는 유연한 프로그램이 되는 것이다.

예를들어 `ArrayList`를 쓰다가 자료구조의 중간에 삽입, 삭제가 빈번하게 일어난다면 객체 주입부분만 `LinkedList`로 변경해주면 성능향상을 볼 수 있는 것이다.

List는 배열과 다르게 사이즈를 지정하지 않고 생성하였다. 그리고 `add`메서드를 통해 메모리가 되는 한 무한대로 요소들을 추가 할 수 있다. 그리고 `remove` 메서드를 통해 요소들을 삭제하면 알아서 인덱스를 맞춰주고 사이즈 또한 관리해준다.

### 3. 인터페이스

인터페이스는 한마디로 멤버메서드가 모두 추상메서드인 추상클래스이다. 이러한 클래스를 특별히 인터페이스라고 부르며 선언시에도 class 대신 interface 라고 적어준다. 그 외에는 추상클래스와 다르지 않다. 자세한 특징은 아래의 그림을 참조하자.

1. 클래스와 동일하나 키워드만 interface로 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 }
```

#### 3.1 인터페이스의 상속

인터페이스의 부모는 무조건 인터페이스 여야 한다. 추상클래스여도 안된다. 이러한 문법 요소들은 막연히 외우는 것이 아니라 이해를 해야 한다. 왜 인터페이스는 부모가 인터페이스여야만 하는가? 앞서 배운 인터페이스의 규칙을 다시 보자 2번 멤버메서드는 전부 추상 메서드여야 한다. 즉 일반클래스는 이미 내부가 구현된 메서드를 가지고 있다. 그리고 추상클래스 라고 하더라도 일부 클래스는 내부가 구현되어 있을 가능성이 있다. 하지만 인터페이스는 전부 추상메서드여야 한다. 따라서 인터페이스의 부모는 모두 인터페이스여야 한다.

인터페이스는 인터페이스만 상속 받을수 있다.  
(따라서 Object를 상속받지 않는다)

```
3 public interface interfaceTest extends Sofa{  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100
```

```
1 public interface ParentImpl {  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100
```

```
1 public interface interfaceTest extends ParentImpl{  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100
```



### 3.2 다중상속

인터페이스는 클래스와 다르게 다중상속을 지원한다.

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

### 3.3 인터페이스 구현

인터페이스를 상속받아 객체 생성이 가능한 클래스로 만드는 것을 구현이라고 한다. 구현을 할 때는 implements 키워드를 사용한다.

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

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

구현을 할 때 일부만 구현한다면 해당 자식은 추상클래스가 되어야 한다.

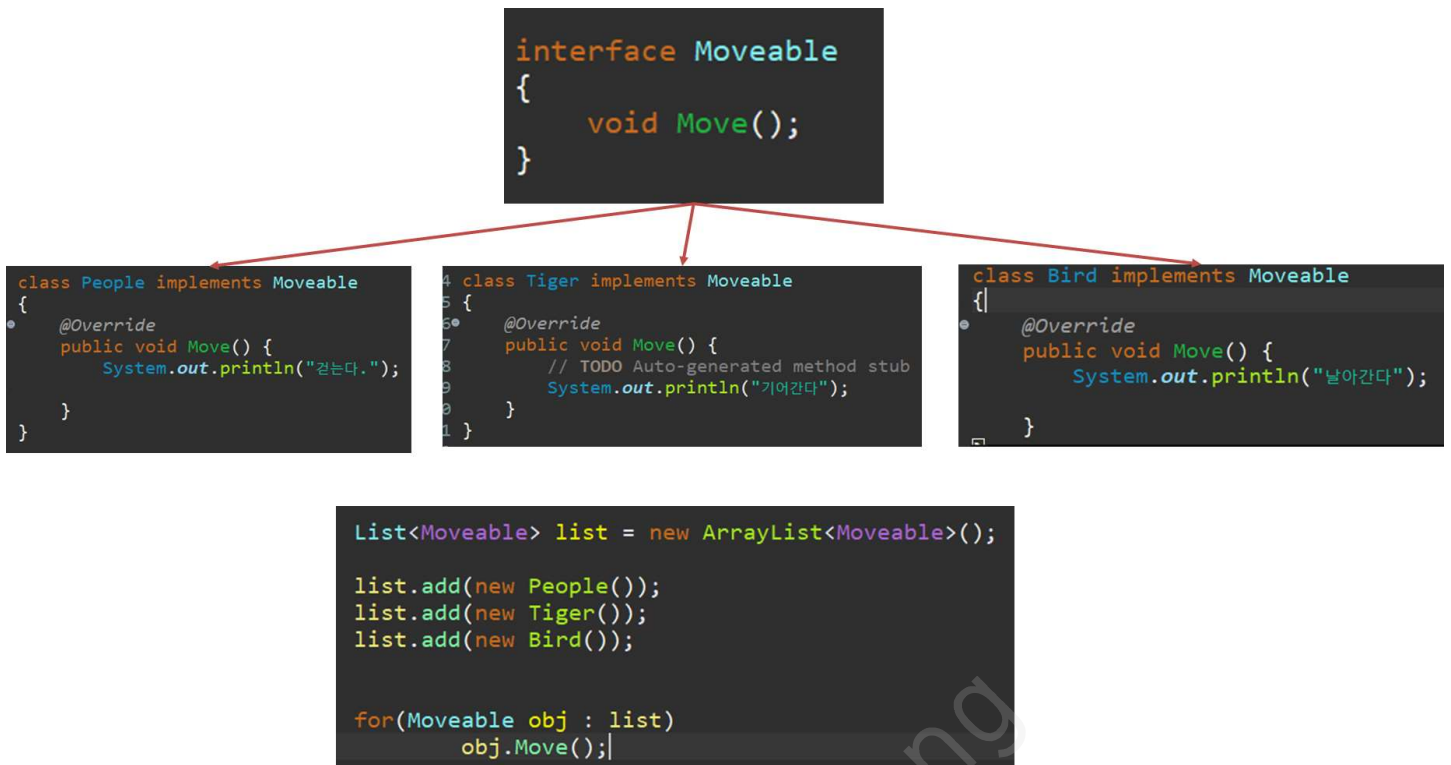
인터페이스의 구현은 상속과는 별개로 가능하다. 즉 그림22처럼 클래스를 상속받으며 동시에 인터페이스를 구현할 수 있다.

```
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.**

그림 22

### 3.4 인터페이스의 활용



위의 그림은 지금까지 배운 인터페이스를 활용하는 방법을 보여준다. Moveable을 구현하는 3개의 클래스가 있고 List에는 인터페이스의 타입을 요소로 지정하였다. 그리고 인터페이스를 상속받은 3개의 다른 객체를 생성하여 list로 관리를 하고 있다.

가만히 보면 이것은 이전에 배웠던 추상클래스를 활용한 다형성과 다를게 없다. 추상클래스 내부가 전부 추상메서드이면 인터페이스랑 뭐가 다른걸까? 이것을 보기전에 잠깐 인터페이스를 만들 때 일반적으로 사용하는 네이밍을 보자

### 3.5 인터페이스의 네이밍

```
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 }
```

문법적으로 정해진건 아니지만 일반적으로 인터페이스의 접미사로 able이 붙는걸 볼수 있다. able의 의미는 ~할수 있는 이다. 왜 이런 이름이 붙는것일까? 지금부터 추상클래스와 인터페이스의 차이점 그리고 왜 이러한 접미사를 붙여 쓰는지를 같이 보도록 하자.

### 3.6 인터페이스 VS 추상클래스

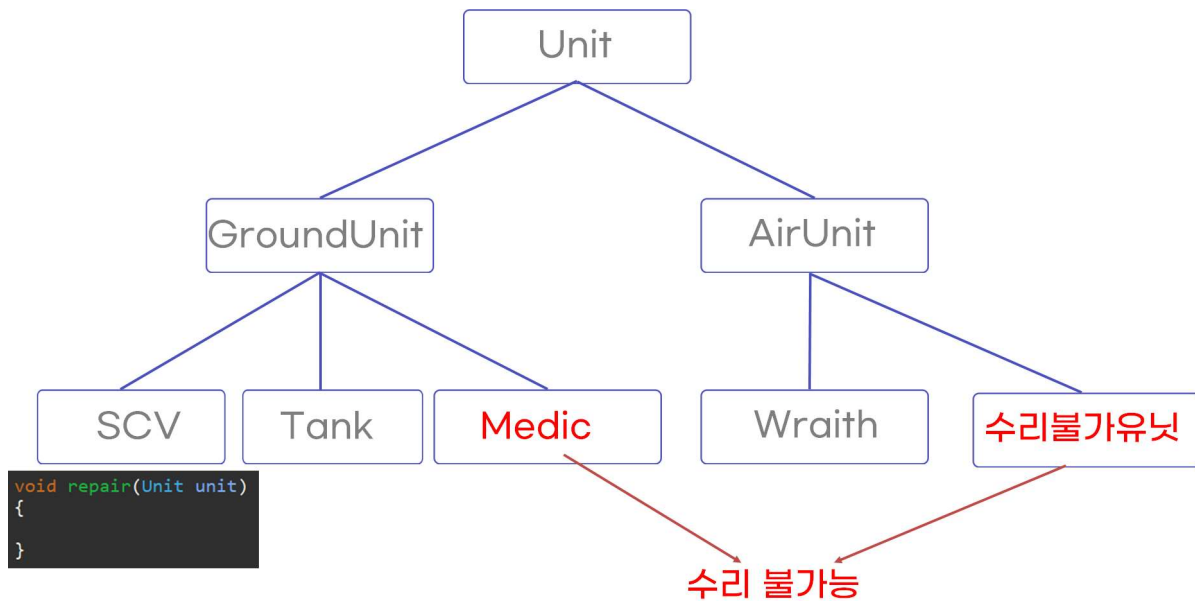


그림 25

그림25는 게임을 만들기 위한 클래스관계도 이다. SCV는 수리를 할수 있는 유닛이며 수리가 가능한 유닛은 지정되어 있다. 현재 SCV유닛에 repair 메서드는 매개변수로 Unit 타입을 받고 있다. 그런데 이러면 문제가 있다. 바로 Medic과 공중유닛중 수리가 불가능한 유닛등이 있다면 이는 repair 메서드 안에서 예외처리를 해줘야 한다. 예외처리만 한다고 끝인가? 만약 앞으로 Unit을 상속받은 클래스가 수리 불가능한 특성을 지닌다면 그러한 클래스가 추가 될 때마다 repair 메서드는 수정되어야 한다.

그러면 Unit을 받을게 아니라 수리가 가능한 유닛만을 묶어줄수 있는 부모 클래스를 중간에 만들면 되지 않는가? 그리고 그 클래스의 타입으로 매개변수를 받으면 해결될 것이다.

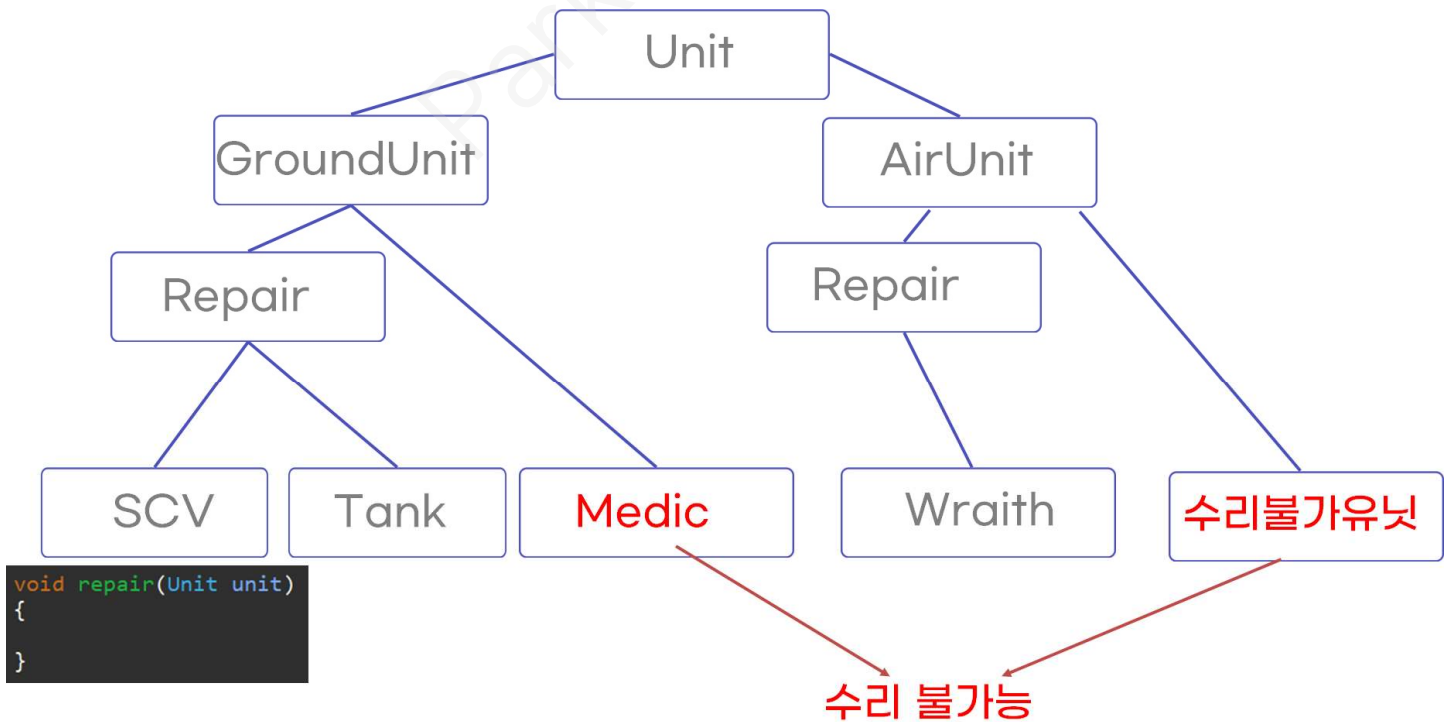


그림 26

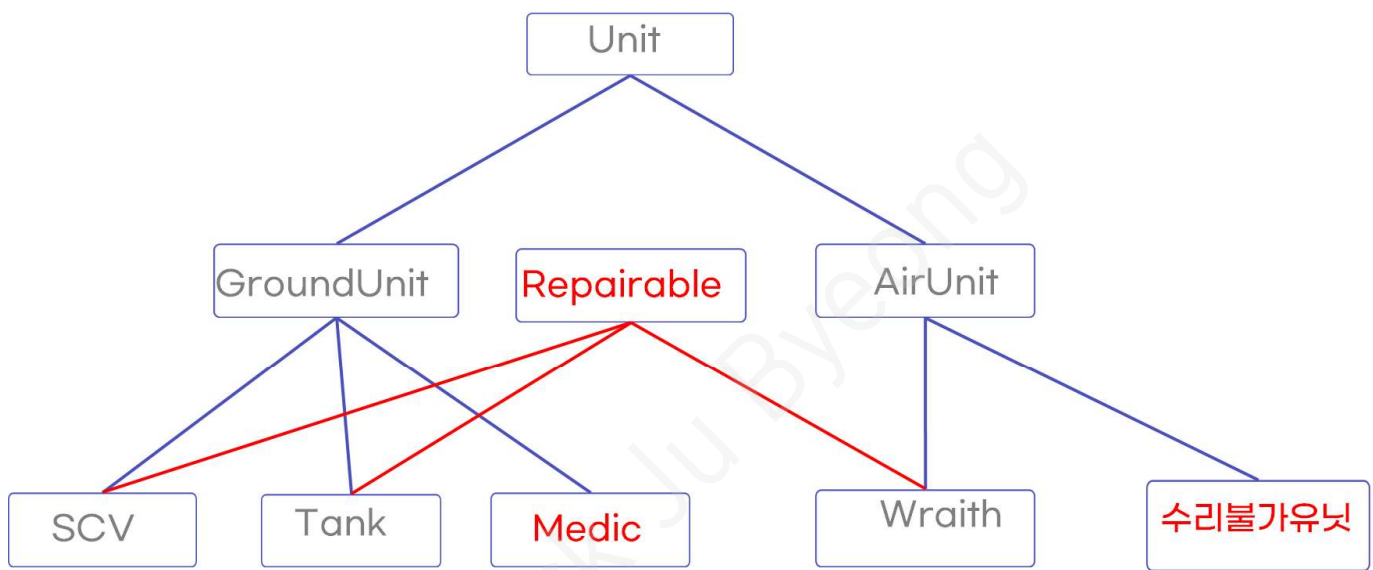
그림26은 중간에 수리가 가능한 클래스들을 묶어두기 위한 부모 클래스를 끼워둔 것이다.

그런데 Repair 클래스가 왜 2개 인 것일까? 일단 java는 다중상속을 허용하지 않기 때문에 지상군,공군을 동시에 상속 받을 수 없다. 또한 상속된다 하더라도 Repair를 상속받은 실제 유닛들은 또다시 지상군, 공군을 분류를 해야 한다.

즉 최하단의 유닛 하나의 분류는 다음의 4가지로 나뉘게 된다.

- 1.지상군이며 수리가 가능
- 2.지상군이며 수리가 불가능
- 3.공군이며 수리가 가능
- 4.공군이며 수리가 불가능

즉 중간에 어떤 순서로 부모 클래스의 관계를 잡든 무조건 4가지의 경우가 나오며 이는 부모 클래스의 중복을 야기한다. 그렇기에 상속을 이용한 클래스 관계에 영향을 주지 않고 또 다른 분류체계가 필요한 것이다.



기존의 클래스 관계도를 유지하면서 특정  
기능별로 묶어 분류 하는것이 가능.

그림 27

그림27은 기존의 클래스 관계도에 인터페이스를 추가해 놓은 그림이다. 인터페이스는 상속과 동시에 이뤄질수 있다고 하였다. 이렇게 한다면 SCV는 매개변수로 Repairable 타입으로 받으면 지상군, 공군 상관없이 수리가 가능한 유닛만을 매개변수로 받는 것이다. 즉 상속은 클래스들끼리 구체적인 특징들을 물려줌으로써 강한 의존성을 필요할 때 쓰이고 특정 기능 하나 하나를 부과할 때 인터페이스의 구현을 활용한다.

지금까지 설명을 보면 인터페이스는 클래스에 기능 하나를 부여 하는것과 비슷하다. 따라서 수리 할 수 있는 Tank를 표현하기 위해 able 이라고 붙이면 적절한 것이다.

### 3.7 인터페이스의 Static메서드

인터페이스는 모두 추상클래스여야 된다고 하였다. 그런데 생각해보면 Static메서드는 객체와는 무관한 메서드이다. 따라서 인터페이스도 Static 메서드를 가질 수 있고 이용할때도 큰 문제가 될 것이 없다. 그러나 그동안 인터페이스는 Static메서드를 사용 할 수 없었다. 문법의 심플함을 유지하기 위해서다. JDK1.8버전 이상부터는 인터페이스에서 Static메서드를 사용할수 있게 되었다.

### 3.8 default메서드

인터페이스는 구현을 할 때 추상메서드들을 무조건 구현해야지만 컴파일이 된다. 그러면 아래와 같은 상황은 어떤가?



그림 28

Moveable인터페이스를 구현해놓은 클래스들이 많은 상황에서 인터페이스에 메서드가 추가된다면 구현을 해놓은 수많은 클래스들 역시 해당 메서드는 구현해야지만 컴파일이 될 것이다.

이러한 작업은 쉽지가 않다. 다른회사와의 협업 상황일수도 있고 이미 개발된지 오래된 클래스일수도 있을 것이다. 혹은 그런 클래스들의 소스코드를 수정할 권한이 없을수도 있다.

이러한 상황에서 사용할수 있는 것이 default 이다.



위와 같이 default 라는 키워드를 적고 메서드의 내용을 종괄호로 만들어주면 해당인터페이스를 구현하는 클래스들은 영향을 받지 않는다. eat() 메서드는 그대로 자식클래스에 상속되어 객체 생성후 사용 할 수 있다. 인터페이스에서 실질적인 코드를 넣는것도 가능하다. 사실상 일반 메서드를 인터페이스에 추가해 놓은것과 다를 바 없다. 사실상 이러한 문법은 인터페이스의 목적과 맞지 않는 코드라 개인적으론 안 좋게 본다.

#### 4. 내부클래스

이름 그대로 클래스 안에 다시 클래스를 정의 하는 것이다. 선언 할 수 있는 위치는 변수를 선언 할 수 있는 위치 와동일하다. 라이프사이클과 스코프 역시 변수와 동일하다.

```
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
28         System.out.println("동물이 먹는다.");
29     }
30 }
31
```

클래스 내부에서 클래스를 선언할수 있다.

내부 클래스를 사용할때는 일반적인 클래스를 사용하는것과 동일하다. 객체를 생성하고 해당타입의 참조변수에 넣고 사용하는 것이다. 클래스 외부에서 사용할때는 new animal.InnerClass() 형태로 사용한다.

```
9 class animal
10 {
11
12     int a;
13     class InnerClass
14     {
15
16         void test()
17         {
18             a= 10;
19             System.out.println("내부 클래스 메서드");
20         }
21     }
22
23     void exMethod()
24     {
25         InnerClass innerClass = new InnerClass();
26         innerClass.test();
27     }
28 }
29
```

내부 클래스 선언

내부 클래스 사용



## 4.1 익명클래스

클래스를 선언과 동시에 객체 생성을 하는 특별한 방법이다. 특정 케이스 마다 클래스가 계속 추가 되어야 하는 형태가 있을수 있다. 예를 들어 버튼을 눌렀을 때 특정 코드를 수행해야 하는 기능을 만들고자 한다. 그런데 이러한 기능들은 버튼들마다 제 각각 일 것이다. 무조건 파일을 생성해야 하는 클래스로 이러한 기능들을 구현하고자 한다면 버튼 1개 당 클래스 파일 1개를 생성해야 하는 불편함이 따른다. 그래서 이럴 경우 버튼 이벤트에 특정 객체를 넘겨줄 때 무명 클래스를 이용하여 클래스 선언과 동시에 객체를 생성하여 버튼이벤트의 매개변수로 넘겨주는 형태이다.

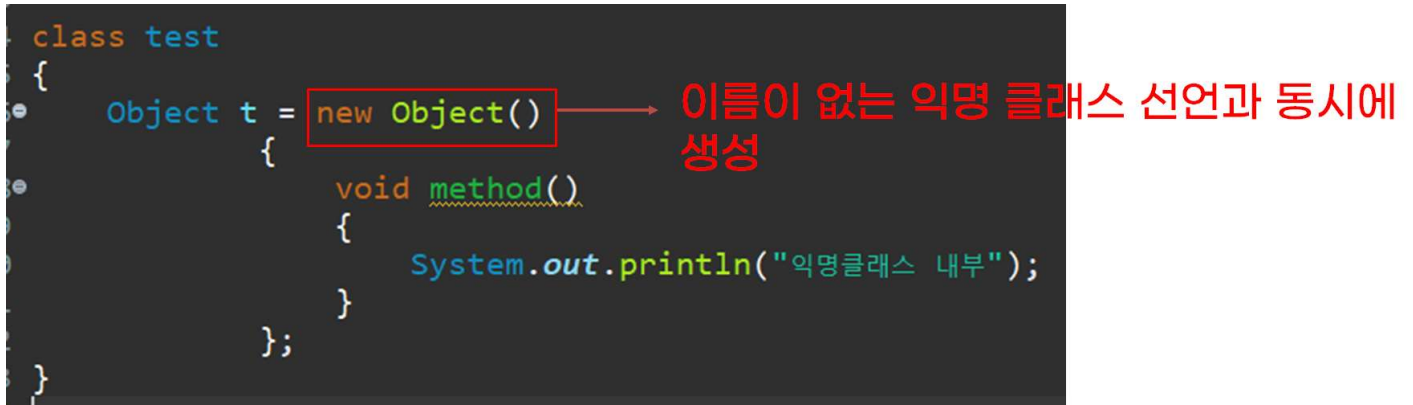
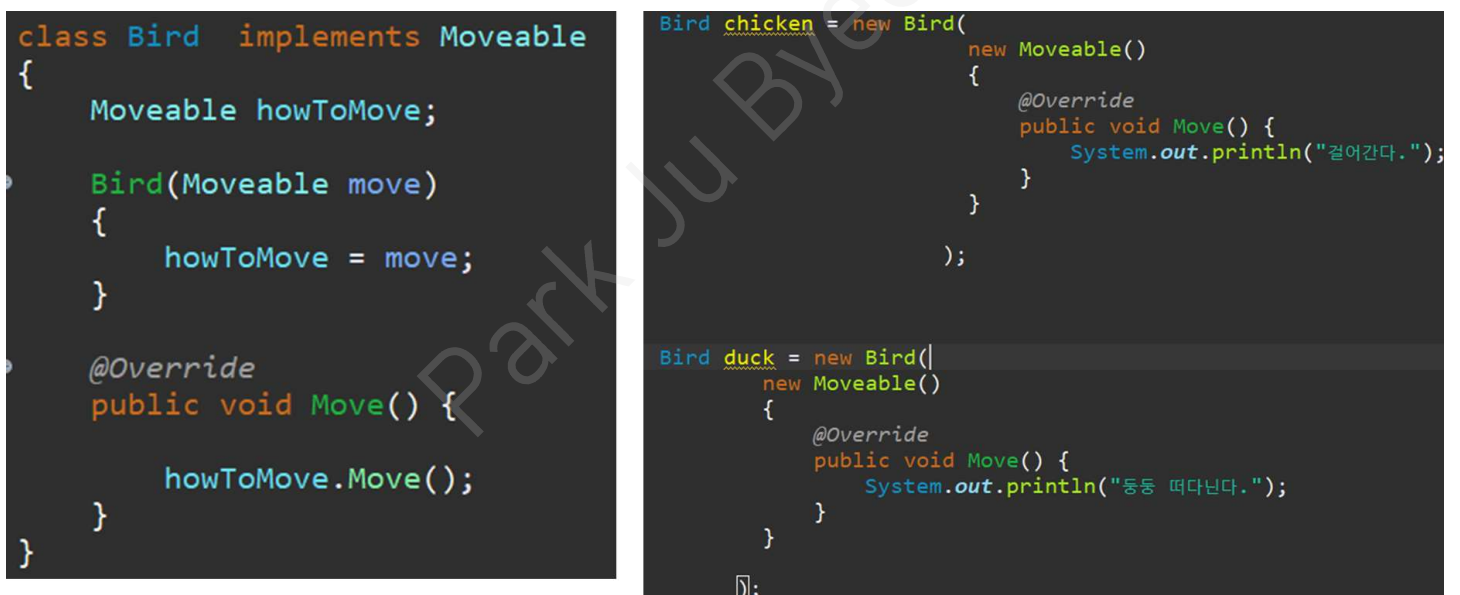


그림 32

기본적인 형태는 그림32와 같다. 그런데 주의해서 봐야 할 점은 new Object() 라는 것은 오브젝트 클래스를 생성하는 것이 아니다. Object클래스를 상속받은 자식클래스의 객체를 생성하는 것이다. 다음의 예시를 보도록 하자.



Bird 클래스는 Moveable 인터페이스를 구현하기만 하지만 정작 메서드 내부를 보면 단순히 외부로부터 Moveable 을 상속받은 객체를 전달 받아 그걸 그대로 실행시켜주는게 전부이다.

즉 Bird 객체가 이동하는 방법에 대해서 Moveable을 상속받아 구현하는 여러 가지 클래스들이 있을 것이다. 그중 에 어떠한것도 특정하지 않고 외부에서 주는 객체를 그대로 쓰겠다는 것이다.

물론 위의 예시는 이동할 수 있는 방법이 다 합쳐도 몇가지 경우가 없기에 클래스를 직접 선언해놓아도 괜찮은 방법이다. 하지만 이동하는 방법이 수백가지가 넘는 경우라면 수백개의 클래스를 모두 만들것인가?

이럴 때 무명클래스를 이용한다면 이름을 짓지 않고 Moveable 인터페이스를 구현하는 객체를 생성 할 수 있다.