

■ 다형성 (Polymorphism)

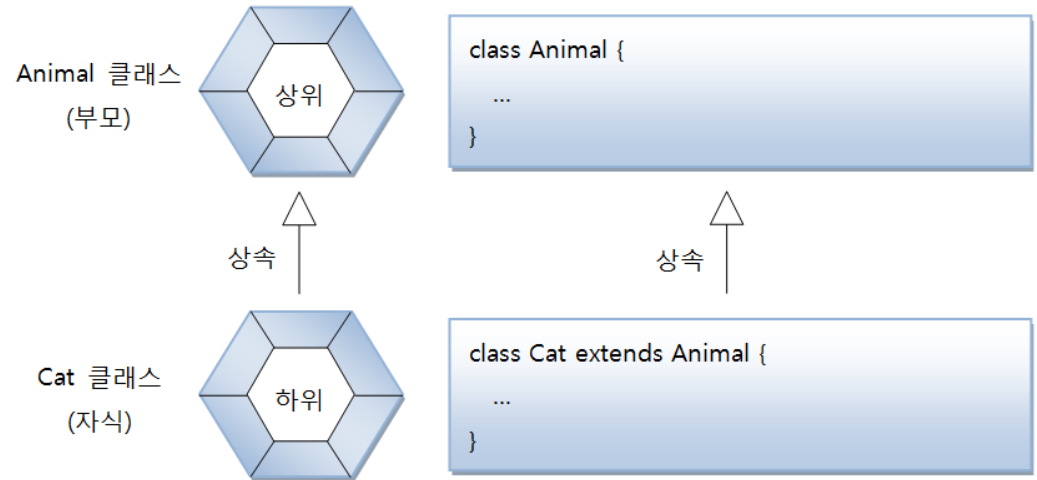
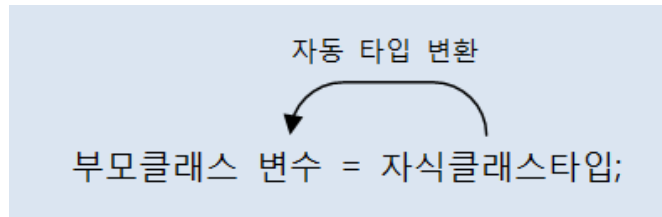
● 같은 타입이지만 다양한 객체 대입(이용) 가능한 성질

- 부모 타입에는 모든 자식 객체가 대입 가능
자식 타입은 부모 타입으로 자동 타입 변환
- 효과 : 객체 부품화 가능



■ 자동 타입 변환(Up Casting)

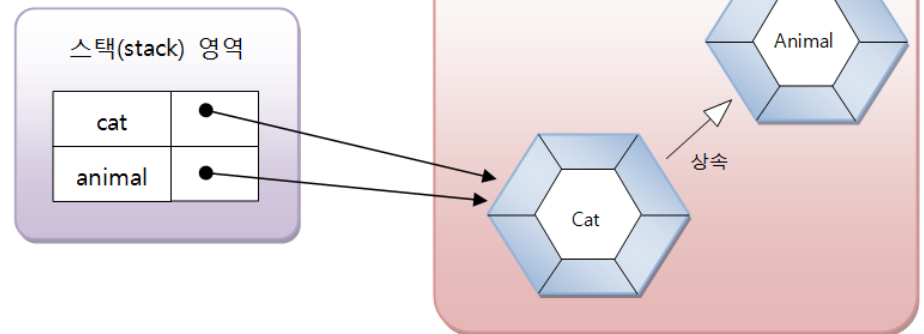
- 프로그램 실행 도중에 자동 타입 변환이 일어나는 것



```
Cat cat = new Cat();  
Animal animal = cat;
```

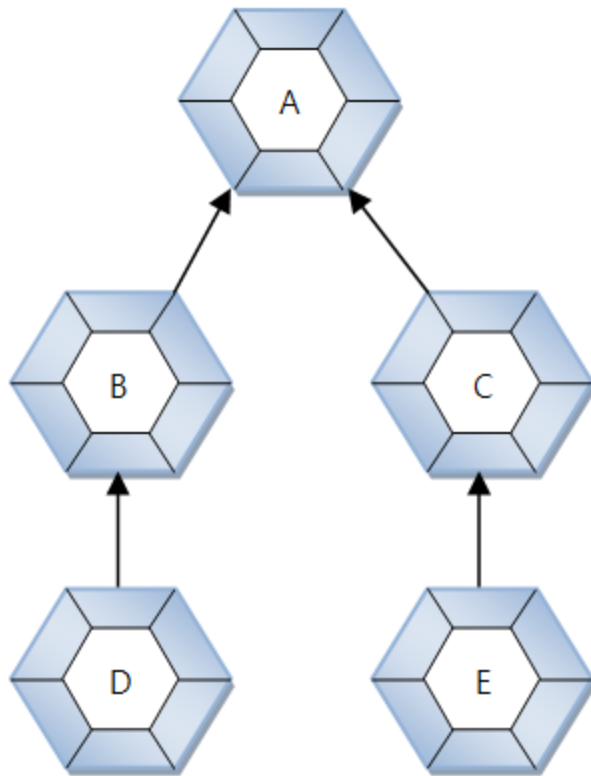
} Animal animal = new Cat(); 도 가능하다.

```
cat == animal //true
```



■ 자동 타입 변환(Up Casting)

- 바로 위의 부모가 아니더라도 상속 계층의 상위면 자동 타입 변환 가능
 - 변환 후에는 부모 클래스 멤버만 접근 가능



```
B b = new B();  
C c = new C();  
D d = new D();  
E e = new E();
```



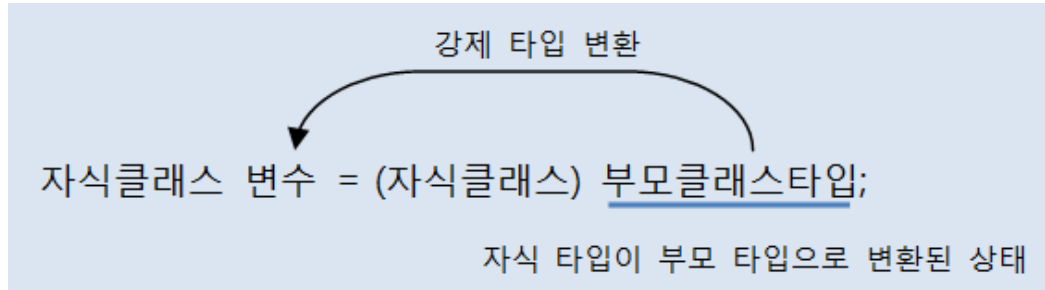
```
A a1 = b; (가능)  
A a2 = c; (가능)  
A a3 = d; (가능)  
A a4 = e; (가능)
```

```
B b1 = d; (가능)  
C c1 = e; (가능)
```

```
B b3 = e; (불가능)  
C c2 = d; (불가능)
```

■ 강제 타입 변환(Down Casting)

● 부모 타입을 자식 타입으로 변환하는 것



● 조건

- 자식 타입을 부모 타입으로 자동 변환 후, 다시 자식 타입으로 변환할 때

● 강제 타입 변환 이 필요한 경우

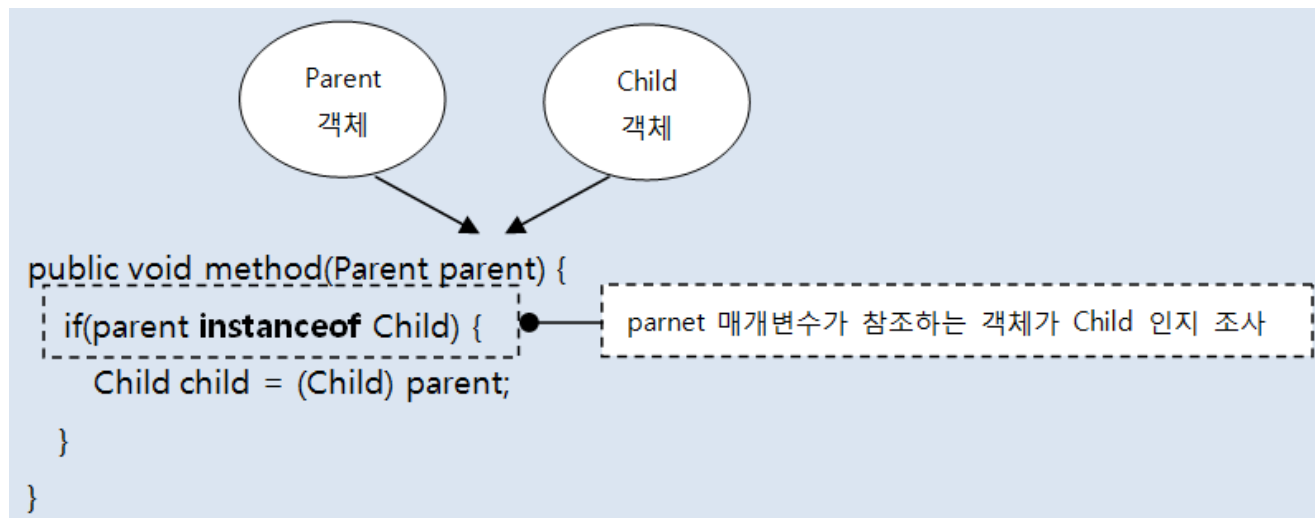
- 자식 타입이 부모 타입으로 자동 변환
(Up Casting 상태에서는 부모 타입에 선언된 필드와 메소드만 사용 가능)
- 자식 타입에 선언된 필드와 메소드를 다시 사용해야 할 경우

■ 객체 타입 확인 (instanceof)

- 부모 타입이면 모두 자식 타입으로 강제 타입 변환할 수 있는 것 아님
 - Child 클래스가 Parent 클래스의 자식이 아닌 경우
ClassCastException 예외 발생 가능

```
Parent parent = new Parent();  
Child child = (Child) parent;    //강제 타입 변환을 할 수 없다.
```

- 먼저 자식 타입인지 확인 후 강제 타입 실행해야 함



■ 추상 클래스(abstract class)

● 추상(abstract)

- 실체들 간에 공통되는 특성을 추출한 것

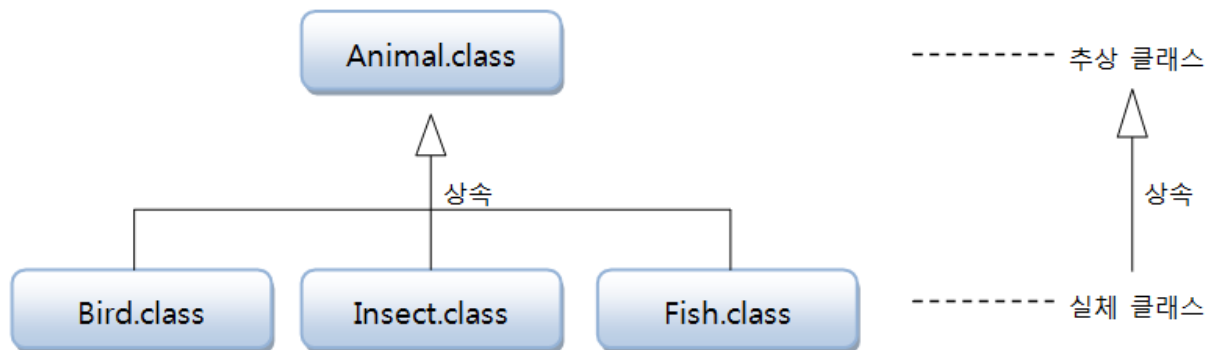
예1) 새, 곤충, 물고기 → 동물 (추상)

예2) 삼성, 현대, LG → 회사 (추상)

● 추상 클래스 (abstract class)

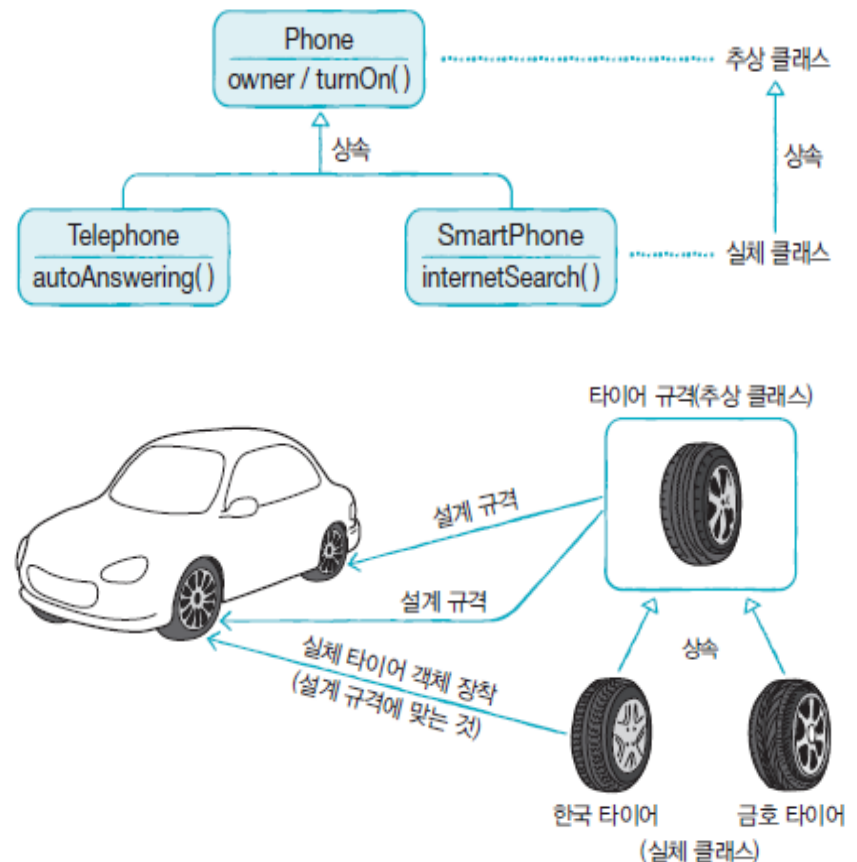
- 실체 클래스들의 공통되는 필드와 메소드 정의한 클래스
- 추상 클래스는 실체 클래스의 부모 클래스 역할 (단독 객체 X)

*실체 클래스: 객체를 만들어 사용할 수 있는 클래스



■ 추상 클래스(abstract class)

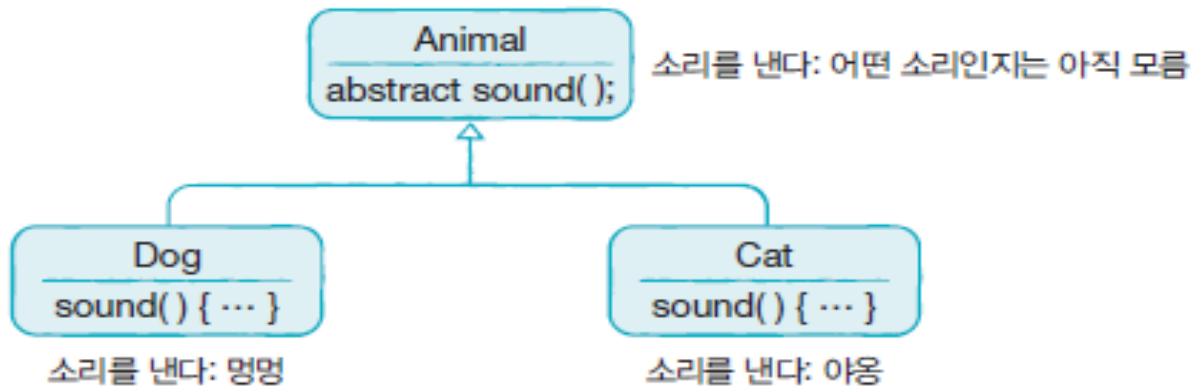
- 실체(구현) 클래스에 반드시 존재해야 할 필드와 메소드를 선언(설계규격)
- 공통된 내용은 상속받아 사용하고, **구현 클래스 내에서 다른 부분 수정**
- 일반 상속과의 차이점은 **각자 다른 부분을 반드시 수정해야 한다는 강제성**



■ 추상 클래스(abstract class)

- abstract 키워드를 사용하여 반드시 구현해야 하는 부분을 명시
- abstract 가 사용된 메소드는 중괄호 사용 불가 (미완성 상태로 남겨둠)
- 추상 클래스를 상속받을 구현 클래스에서 반드시 실행 내용을 완성해야 됨

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



■ 추상 클래스 사용 - 1

```
public abstract class Figure {  
    public abstract void area(int a, int b);  
}
```

```
public class Tetragon extends Figure{  
    @Override  
    public void area(int a, int b) {  
        System.out.println("사각형의 넓이 : " + (a * b));  
    }  
}
```

```
public class Triangle extends Figure {  
    @Override  
    public void area(int a, int b) {  
        System.out.println("삼각형의 넓이 : " + (a * b / 2));  
    }  
}
```

■ 추상 클래스 사용 - 2

```
public abstract class Car {  
    public abstract void move();  
}
```

```
public class SportsCar extends Car {  
    @Override  
    public void move() {  
        System.out.println("100km/h 이동");  
    }  
  
    public void openSunloof() {  
        System.out.println("썬루프 열림");  
    }  
}
```

```
public class Truck extends Car {  
    @Override  
    public void move() {  
        System.out.println("50km/h 이동");  
    }  
  
    public void load() {  
        System.out.println("짐 실음");  
    }  
}
```

■ 추상 클래스 사용 – 3

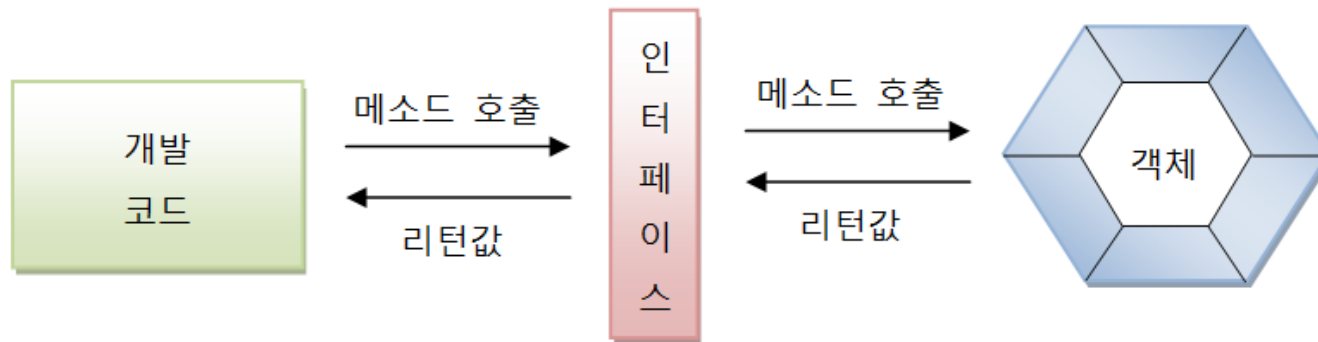
```
public abstract class AbstractParent {  
    abstract void walk();  
}
```

```
public abstract class AbstractChild extends AbstractParent {  
    abstract void run();  
}
```

```
public class AbstractMain extends AbstractChild {  
    @Override  
    void run() { /* 코드 작성 */ }  
  
    @Override  
    void walk() { /* 코드 작성 */ }  
}
```

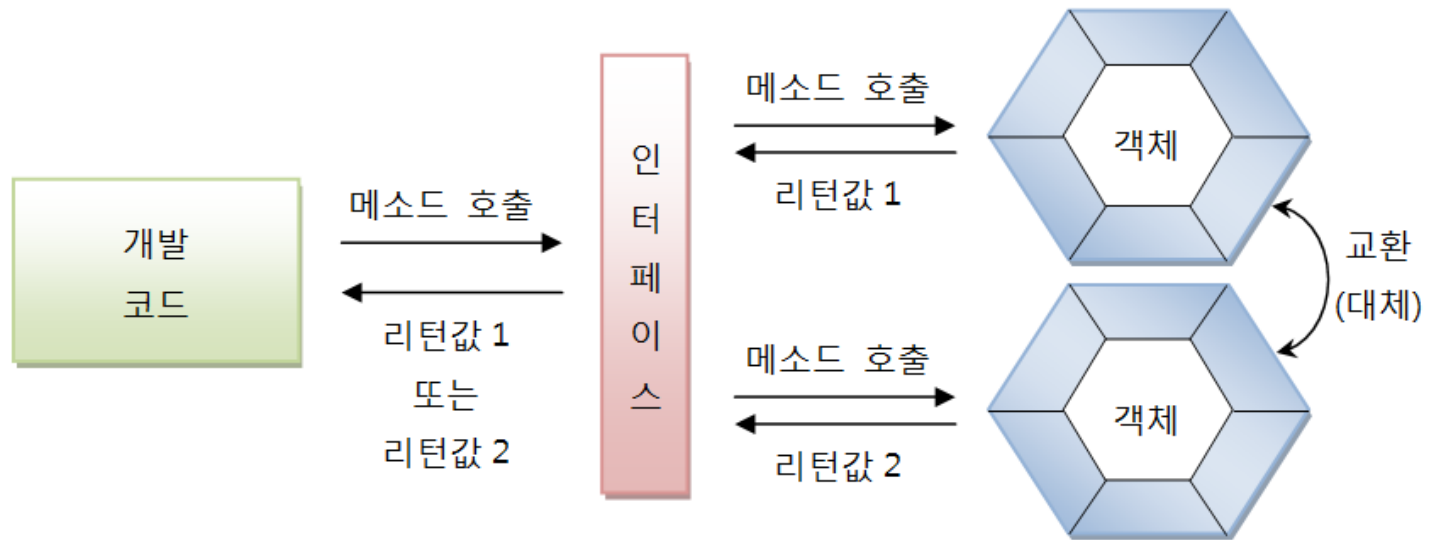
■ 인터페이스(interface)

- 개발 코드와 객체가 서로 통신하는 접점
- 일종의 추상클래스. 추상클래스(미완성 설계도)보다 추상화 정도가 높다.
- 인스턴스를 생성할 수 없고, 클래스 작성에 도움을 줄 목적으로 사용된다.
- 개발 코드 수정을 최소화 하면서 객체 사용(교체/추가)이 가능



■ 인터페이스의 역할

- 개발 코드가 객체에 종속되지 않게 -> 객체 교체할 수 있도록 하는 역할
- 개발 코드 변경 없이 리턴값 또는 실행 내용이 다양해 질 수 있음 (다형성)



■ 인터페이스의 작성

- 'class' 대신 'interface'를 사용

```
interface 인터페이스 이름{  
    public static final float pi = 3.14f;  
    public void add();  
}
```

```
public interface Calc {
```

```
    double PI = 3.14;  
    int ERROR = -999999999;
```

인터페이스에서 선언한 변수는 컴파일
과정에서 상수로 변환됨

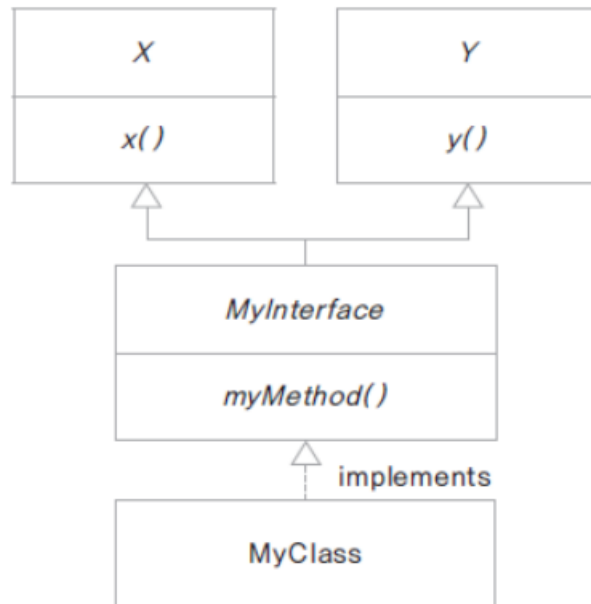
```
    int add(int num1, int num2);  
    int subtract(int num1, int num2);  
    int times(int num1, int num2);  
    int divide(int num1, int num2);
```

인터페이스에서 선언한 메서드는 컴파일
과정에서 추상 메서드로 변환됨

```
}
```

■ 인터페이스의 상속

- 인터페이스도 클래스처럼 상속이 가능하다. (클래스와 달리 다중상속 허용)
- 구현코드의 상속이 아니므로 형 상속(type inheritance) 라고 한다.



```
public interface MyInterface extends X, Y {  
    void myMethod( );  
}
```

인터페이스 여러 개를 상속받을 수 있음

■ 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 */}  
}
```


■ 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();  
}
```

■ 인터페이스의 다형성 사용 예

```
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) {
}
```

■ 인터페이스 사용 - 1

```
public interface IFigure {  
    void area(int a, int b);  
}
```

```
public class ITetragon implements IFigure {  
    @Override  
    public void area(int a, int b) {  
        System.out.println("사각형의 넓이 : " + (a * b));  
    }  
}
```

```
public class ITriangle implements IFigure {  
    @Override  
    public void area(int a, int b) {  
        System.out.println("삼각형의 넓이 : " + (a * b / 2));  
    }  
}
```

■ 인터페이스 사용 – 2 (1 / 3)

```
public interface Unit {  
    // public static final String COLOR_TANK = "BROWN"  
    String COLOR_TANK = "BROWN";  
  
    // public static final String COLOR_MARINE = "BLUE";  
    String COLOR_MARINE = "BLUE";  
  
    // public abstract void move(String area);  
    void move(String area);  
  
    // public abstract void attack(String target);  
    void attack(String target);  
}
```

■ 인터페이스 사용 – 2 (2 / 3)

```
public class Tank implements Unit {  
    private String color;  
  
    Tank(String color) {  
        this.color = color;  
    }  
  
    @Override  
    public void move(String area) {  
        System.out.println(area + " 위치로 탱크 이동");  
    }  
  
    @Override  
    public void attack(String target) {  
        System.out.println("대포를 사용하여 " + target + " 공격");  
    }  
}
```

■ 인터페이스 사용 – 2 (3 / 3)

```
public class Marine implements Unit {  
    private String color;  
  
    Marine(String color) {  
        this.color = color;  
    }  
  
    @Override  
    public void move(String area) {  
        System.out.println(area + " 위치로 마린 이동");  
    }  
  
    @Override  
    public void attack(String target) {  
        System.out.println("총을 사용하여 " + target + " 공격");  
    }  
}
```

■ 다형성 – 악기 연주 (1 / 2)

```
public interface Soundable {  
    String sound();  
}
```

```
public class Piano implements Soundable {  
    @Override  
    public String sound() {  
        return "도레미";  
    }  
}
```

```
public class Guitar implements Soundable {  
    @Override  
    public String sound() {  
        return "팅";  
    }  
}
```

■ 다형성 – 악기 연주 (2 / 2)

```
public class Player {  
    private Soundable soundable;  
  
    public Player(Soundable soundable) {  
        this.soundable = soundable;  
    }  
  
    public void printSound() {  
        System.out.println(soundable.sound());  
    }  
}
```

```
public class SoundLauncher {  
    public static void main(String[] args) {  
        Piano piano = new Piano();  
        Player player1 = new Player(piano); // up casting  
        player1.printSound();  
  
        Guitar guitar = new Guitar();  
        Player player2 = new Player(guitar); // up casting  
        player2.printSound();  
    }  
}
```


■ 다형성 – 휴대폰 사용 (1 / 3)

```
public interface Phone {  
    public void call(int number);  
    public void hangUp();  
}
```

```
public class LPhone implements Phone {  
    @Override  
    public void call(int number) {  
        System.out.println("LPhone " + number + " 전화 걸기");  
    }  
  
    @Override  
    public void hangUp() {  
        System.out.println("LPhone 전화 끊기");  
    }  
}
```

■ 다형성 – 휴대폰 사용 (2 / 3)

```
public class SPhone implements Phone {  
    @Override  
    public void call(int number) {  
        System.out.println("SPhone " + number + " 전화 걸기");  
    }  
  
    @Override  
    public void hangUp() {  
        System.out.println("SPhone 전화 끊기");  
    }  
}
```

■ 다형성 – 휴대폰 사용 (3 / 3)

```
public class PhoneMain {  
    public static void main(String[] args) {  
        // 다형성  
        SPhone sPhone = new SPhone();  
        LPhone lPhone = new LPhone();  
  
        Phone[] phone = new Phone[] {  
            sPhone, lPhone  
        };  
  
        // 전화걸기  
        for(int i = 0; i < phone.length; i++) {  
            phone[i].call(123);  
            phone[i].hangUp();  
        }  
    }  
}
```

■ 다형성 – 이동 수단 (1 / 3)

```
public interface Movable {  
    String move();  
}
```

```
public class Bicycle implements Movable {  
    @Override  
    public String move() {  
        return "자전거";  
    }  
}
```

```
public class Car implements Movable {  
    @Override  
    public String move() {  
        return "자동차";  
    }  
}
```

■ 다형성 – 이동 수단 (2 / 3)

```
public class MovableProxy {  
    private Movable movable;  
  
    public MovableProxy(Movable movable) {  
        this.movable = movable;  
    }  
  
    public void run() {  
        System.out.println(movable.move());  
    }  
}
```

■ 다형성 – 이동 수단 (3 / 3)

```
public class Main {  
    public static void main(String[] args) {  
        Car car = new Car();  
        MovableProxy proxy1 = new MovableProxy(car);  
        proxy1.run();  
  
        Bicycle bicycle = new Bicycle();  
        MovableProxy proxy2 = new MovableProxy(bicycle);  
        proxy2.run();  
    }  
}
```

■ 다형성 – 버튼 이벤트 (1 / 2)

```
public interface OnClickListener {  
    void onClick();  
}
```

```
public class Button {  
    public void setOnClickListener(OnClickListener onClickListener) {  
        onClickListener.onClick();  
    }  
}
```

```
public class MyClickListener implements OnClickListener {  
    @Override  
    public void onClick() {  
        System.out.println("버튼 클릭 - 전화걸기");  
    }  
}
```

■ 다형성 – 버튼 이벤트 (2 / 2)

```
public class Main {  
    public static void main(String[] args) {  
        Button btn = new Button();  
        MyClickListener mcl = new MyClickListener();  
        btn.setClickListener(mcl);  
    }  
}
```


■ 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.