

■ 다형성

● 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



■ 다형성 활용 예시

- 휴대폰 - 블루투스 이어폰
- 노트북 - WIFI 공유기
- 안드로이드폰 - 마이크로 5핀
- 아이폰, 아이패드 - 라이트닝 8핀
- (한국) 가전 제품 - 220V 콘센트
- 사람 - 안경, 옷, 신발, 마스크, 시계, 벨트
- 자동차 - 가솔린, 디젤, 가스, 엔진오일, 타이어

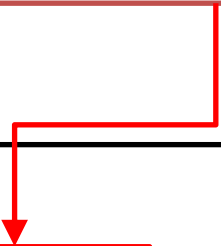
■ 다형성을 코드로 작성

ch13.Router

```
public String connect() {  
    return "connect";  
}  
public String disconnect() {  
    return "disconnect";  
}
```

ch13.Notebook

```
Router router = null;  
  
public void setRouter(Router router) { this.router = router; }  
  
public void connect() {  
    String result = this.router.connect();  
    System.out.println(result);  
}  
  
public void disconnect() {  
    String result = this.router.disconnect();  
    System.out.println(result);  
}
```



다형성을 코드로 작성

ch13.Router

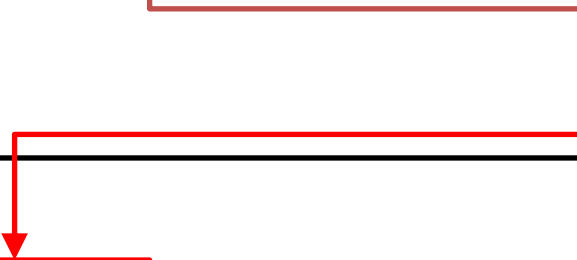
```
public String connect() {  
    return "connect";  
}  
public String disconnect() {  
    return "disconnect";  
}
```

ch13.IpTimeRouter

```
public class IpTimeRouter extends Router {  
    public String connect() {  
        return "iptime connect";  
    }  
    public String disconnect() {  
        return "iptime disconnect";  
    }  
}
```

ch13.Notebook

```
Router router = null;  
  
public void setRouter(Router router) { this.router = router; }  
  
public void connect() {  
    String result = this.router.connect();  
    System.out.println(result);  
}  
  
public void disconnect() {  
    String result = this.router.disconnect();  
    System.out.println(result);  
}
```



■ 다형성을 코드로 작성

ch13.PolymorphismMain

```
Notebook notebook = new Notebook();  
notebook.setRouter(new Router());  
notebook.connect();  
notebook.disconnect();
```

```
Notebook notebook2 = new Notebook();  
notebook2.setRouter(new IpTimeRouter());  
notebook2.connect();  
notebook2.disconnect();
```

```
Notebook notebook3 = new Notebook();  
notebook3.setRouter(new AsusRouter());  
notebook3.connect();  
notebook3.disconnect();
```

```
connect  
disconnect  
iptime connect  
iptime disconnect  
asus connect  
asus disconnect
```

■ 다형성을 코드로 작성

ch13.Router

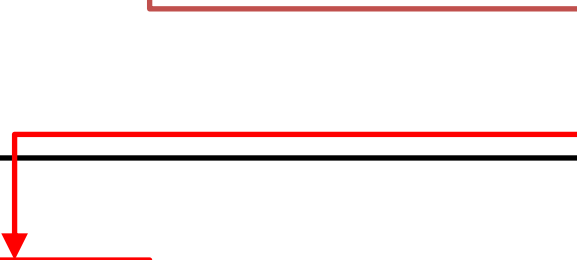
```
public String connect() {  
    return "connect";  
}  
public String disconnect() {  
    return "disconnect";  
}
```

ch13.AsusRouter

```
public class AsusRouter extends Router {  
    public String connect() {  
        return "asus connect";  
    }  
    public String disconnect() {  
        return "asus disconnect";  
    }  
}
```

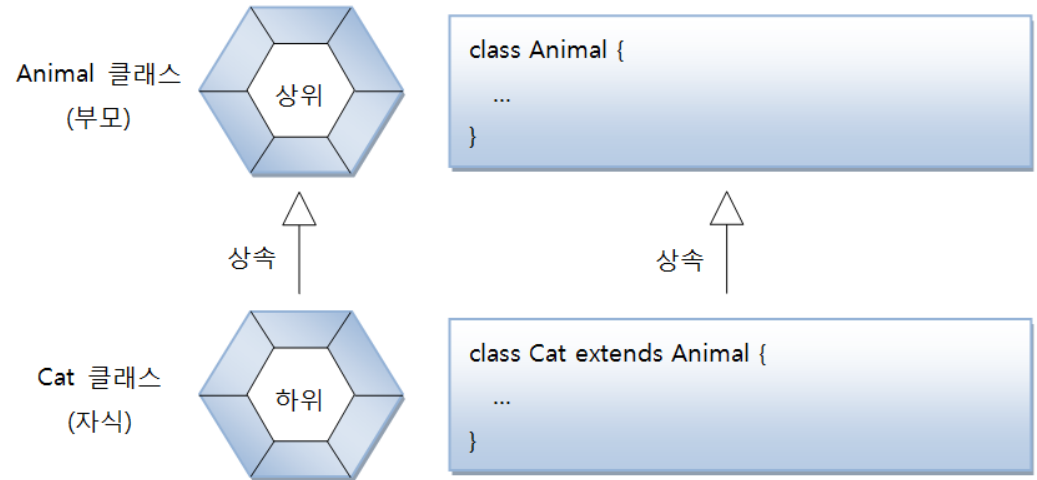
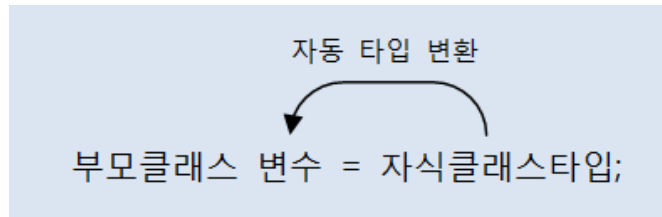
ch13.Notebook

```
Router router = null;  
  
public void setRouter(Router router) { this.router = router; }  
  
public void connect() {  
    String result = this.router.connect();  
    System.out.println(result);  
}  
  
public void disconnect() {  
    String result = this.router.disconnect();  
    System.out.println(result);  
}
```



■ 자동 타입 변환(Up Casting)

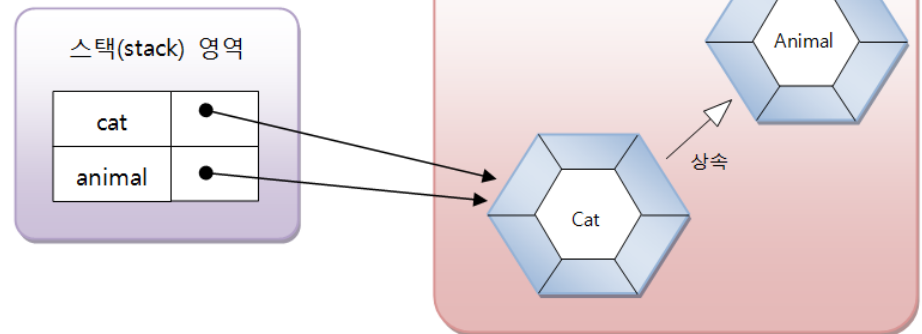
- 프로그램 실행 도중에 자동 타입 변환이 일어나는 것



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

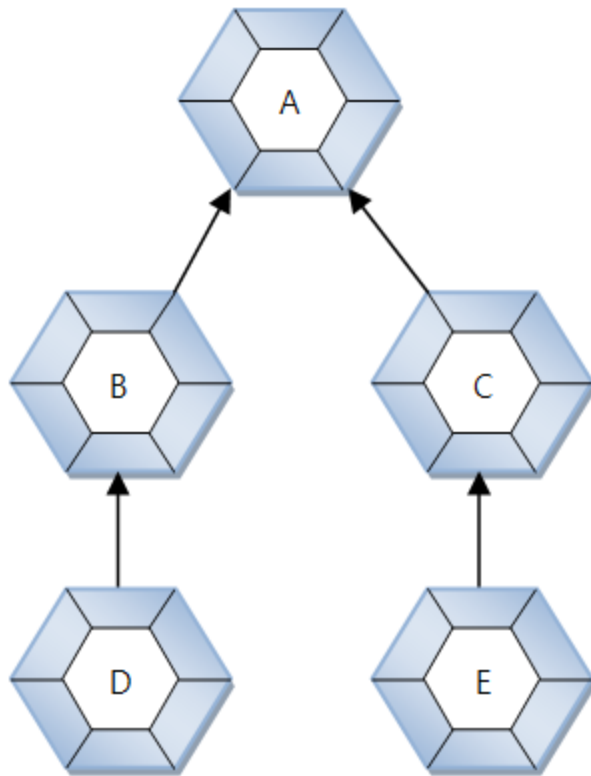
} Animal animal = new Cat(); 도 가능하다.

```
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();
```



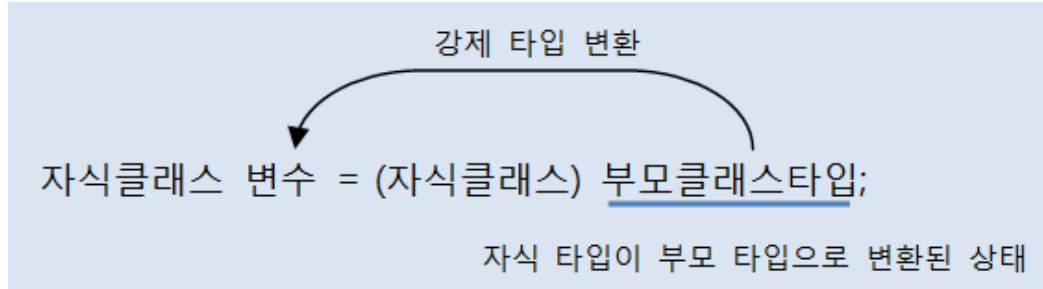
```
A a1 = b; (가능)  
A a2 = c; (가능)  
A a3 = d; (가능)  
A a4 = e; (가능)
```

```
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

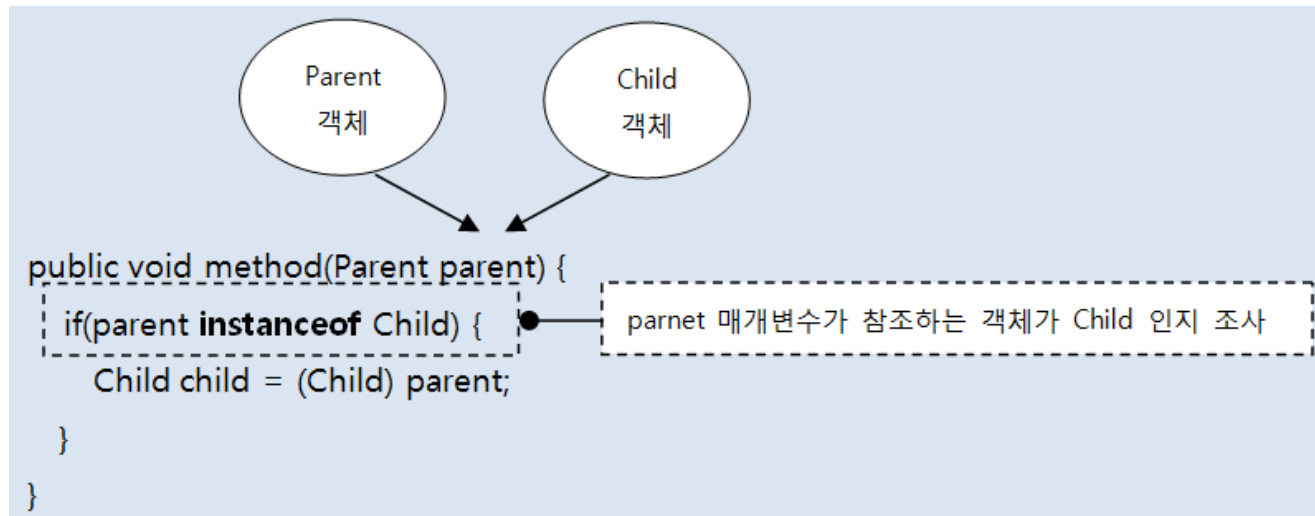
- 자식 타입이 부모 타입으로 자동 변환
(Up Casting 상태에서는 부모 타입에 선언된 필드와 메소드만 사용 가능)
- 자식 타입에 선언된 필드와 메소드를 다시 사용해야 할 경우

■ 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

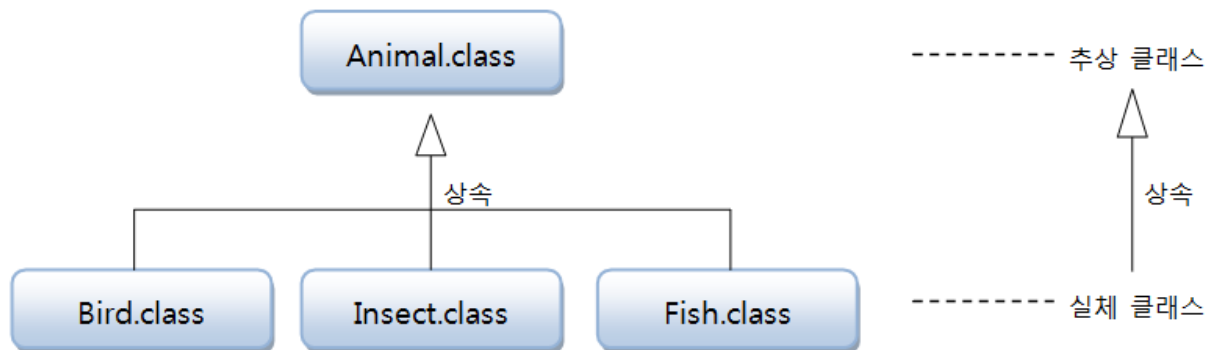
예1) 새, 곤충, 물고기 → 동물 (추상)

예2) 삼성, 현대, LG → 회사 (추상)

● 추상 클래스 (abstract class)

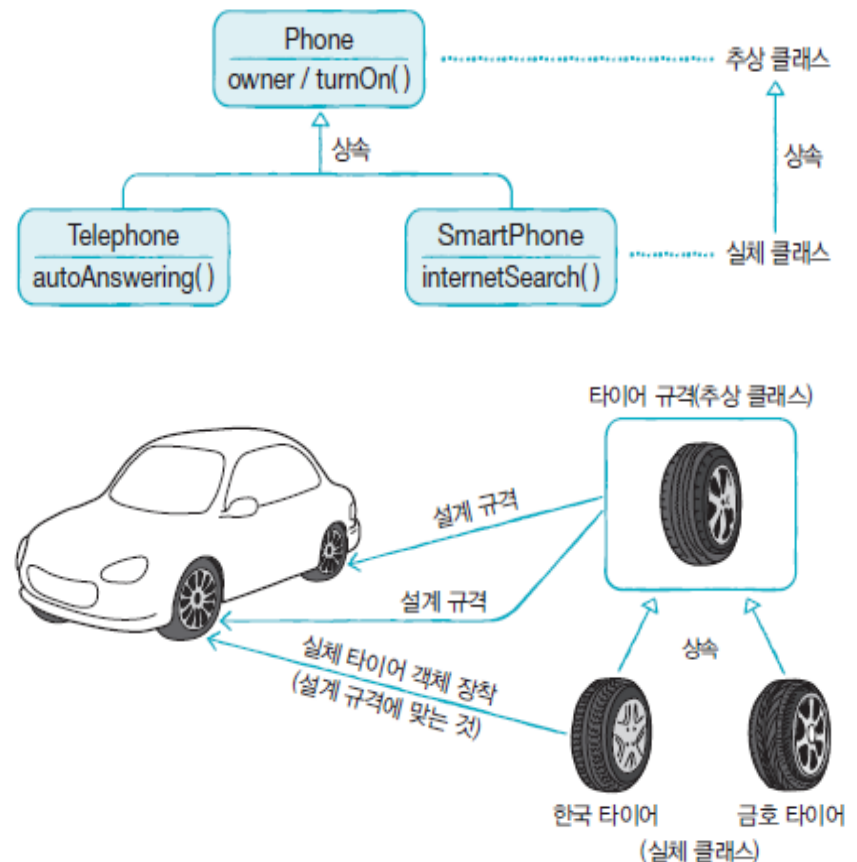
- 실체 클래스들의 공통되는 필드와 메소드 정의한 클래스
- 추상 클래스는 실체 클래스의 부모 클래스 역할 (단독 객체 X)

*실체 클래스: 객체를 만들어 사용할 수 있는 클래스



■ 추상 클래스(abstract class)

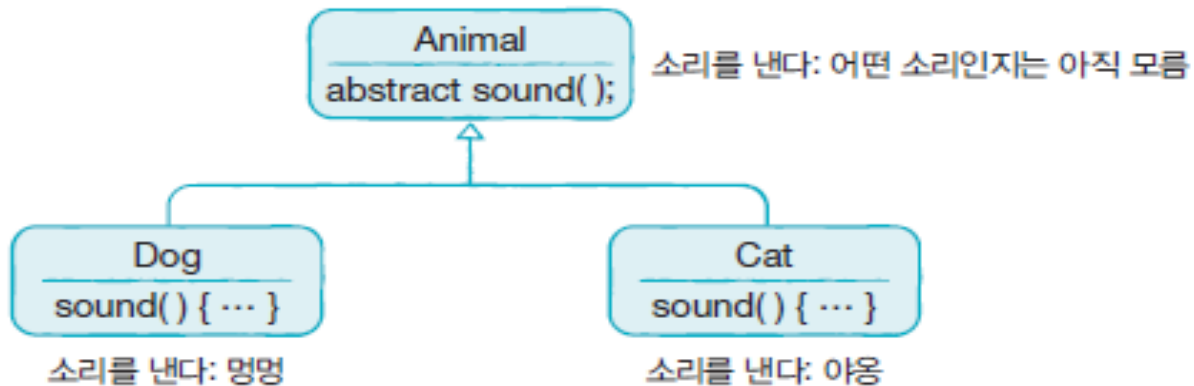
- 실체(구현) 클래스에 반드시 존재해야 할 필드와 메소드를 선언(설계규격)
- 공통된 내용은 상속받아 사용하고, **구현 클래스 내에서 다른 부분 수정**
- 일반 상속과의 차이점은 **각자 다른 부분을 반드시 수정해야 한다는 강제성**



■ 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

ch13.Animal

```
public abstract void sound();
```

ch13.Cat

```
@Override  
public void sound() {  
    System.out.println("야옹");  
}
```

ch13.Dog

```
@Override  
public void sound() {  
    System.out.println("멍멍");  
}
```

ch13.AnimalMain

```
Animal animal1 = new Cat();  
animal1.sound();  
  
Animal animal2 = new Dog();  
animal2.sound();
```

야옹
멍멍

■ 추상 클래스 사용 - 2

ch13.Figure

```
public abstract void area(int x, int y);
```

ch13.Triangle

```
@Override  
public void area(int x, int y) {  
    System.out.println(  
        "삼각형의 넓이 : " + (x * y / 2));  
}
```

ch13.Quadangle

```
@Override  
public void area(int x, int y) {  
    System.out.println(  
        "사각형의 넓이 : " + (x * y));  
}
```

ch13.FigureMain

```
Figure fig1 = new Triangle();  
fig1.area(200, 100);  
  
Figure fig2 = new Quadangle();  
fig2.area(200, 100);
```

```
삼각형의 넓이 : 10000  
사각형의 넓이 : 20000
```

■ 추상 클래스 사용 - 3

ch13.Car

```
public abstract void sound();
```

ch13.SportsCar

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

ch13.Truck

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

ch13.CarMain

```
Car car1 = new SportsCar();  
car1.move();  
((SportsCar) car1).openSunloof();  
  
Car car2 = new Truck();  
car2.move();  
((Truck) car2).load();
```

```
100km/h 이동  
썬루프 열림  
50km/h 이동  
짐 실음
```


■ 추상 클래스 사용 - 4

ch13.AbstractParent

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

ch13.AbstractChild

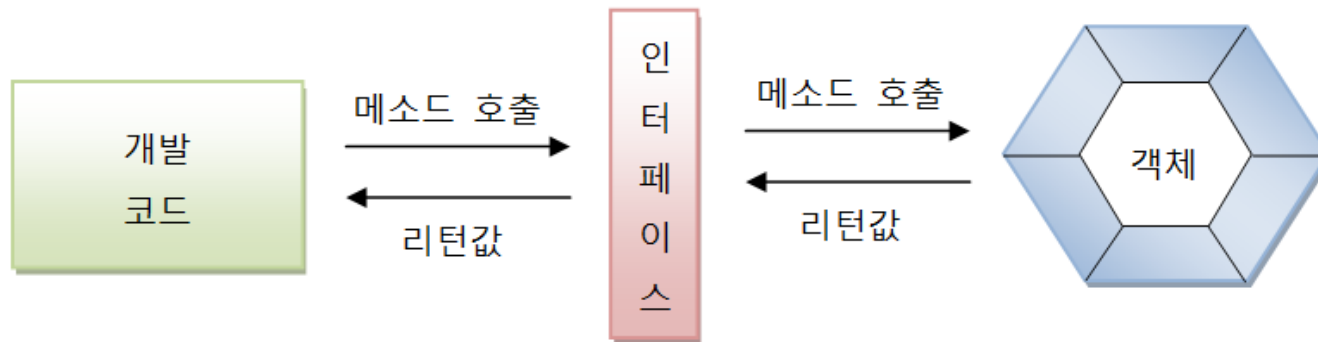
```
public abstract class AbstractChild extends AbstractParent {  
    public abstract void run();  
}
```

ch13.AbstractImpl

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

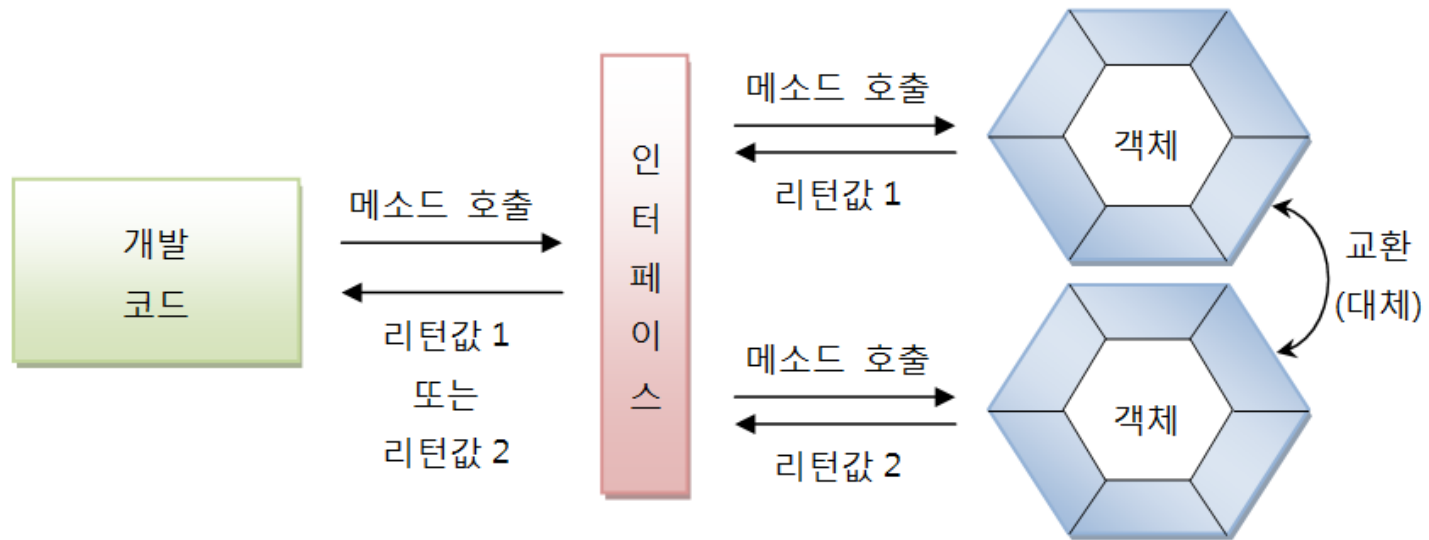
■ 인터페이스(interface)

- 개발 코드와 객체가 서로 통신하는 접점
- 일종의 추상클래스. 추상클래스보다 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();  
}
```

ch13.Calc

```
public interface Calc {
```

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

인터페이스에서 선언한 변수는 컴파일
과정에서 상수로 변환됨

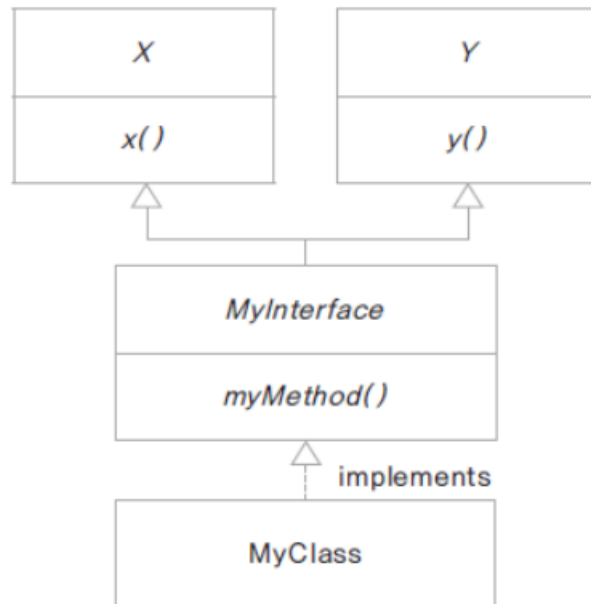
```
    int add(int num1, int num2);  
    int subtract(int num1, int num2);  
    int times(int num1, int num2);  
    int divide(int num1, int num2);
```

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

■ 인터페이스 사용 - 1

ch13.IFigure

```
public abstract void area(int x, int y);
```

ch13.TriangleImpl

```
@Override
public void area(int x, int y) {
    System.out.println(
        "삼각형의 넓이 : " + (x * y / 2));
}
```

ch13.QuadangleImpl

```
@Override
public void area(int x, int y) {
    System.out.println(
        "사각형의 넓이 : " + (x * y));
}
```

ch13.IFigureMain

```
IFigure fig1 = new TriangleImpl();
fig1.area(200, 100);

IFigure fig2 = new QuadangleImpl();
fig2.area(200, 100);
```

```
삼각형의 넓이 : 10000
사각형의 넓이 : 20000
```

■ 인터페이스 사용 - 2

ch13.Soundable

```
String sound();
```

ch13.Guitar

```
@Override  
public String sound() {  
    return "팅";  
}
```

ch13.Piano

```
@Override  
public String sound() {  
    return "도레미";  
}
```

ch13.Player

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

ch13.SoundableMain

```
Soundable piano = new Piano();  
System.out.println(piano.sound());  
  
Soundable guitar = new Guitar();  
System.out.println(guitar.sound());
```

도레미
팅

■ 인터페이스 사용 - 3

ch13.OnClickListener

```
void onClick();
```

ch13.MyClickListener

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

ch13.Button

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

ch13.ButtonMain

```
Button btn = new Button();  
MyClickListener myClickListener = new MyClickListener();  
btn.setOnClickListener(myClickListener);
```

버튼 클릭 - 전화걸기