

JAVA

인터페이스와 다형성





1 인터페이스와 다형성

1. 추상 클래스

2. 인터페이스

3. 다형성

4. 내부 클래스

5. 무명 클래스

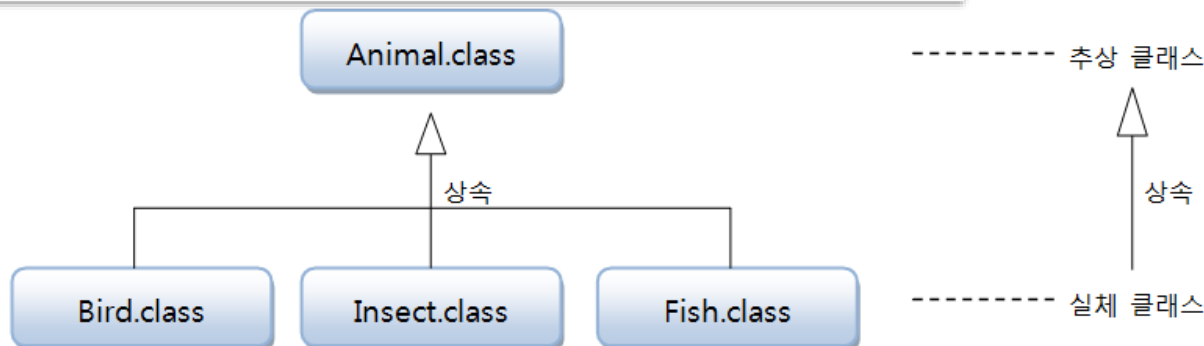


추상 클래스(Abstract Class)

❖ 추상 클래스 개념

- 추상(abstract) : 사전적 의미는 “여러 가지 사물이나 개념에서 공통되는 특성이나 속성 따위를 추출하여 파악하는 작용.”이다.
- 실체들 간에 공통되는 특성을 추출한 것
 - 예1: 새, 곤충, 물고기(실체) → 동물 (추상)
 - 예2: 삼성, 현대, LG (실체) → 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 직접 생성할 수 있는 클래스





추상 클래스(Abstract Class)

❖ 추상 클래스의 용도

- 실체 클래스의 공통된 필드와 메소드의 이름 통일할 목적
 - 실체 클래스를 설계자가 여러 사람일 경우,
 - 실체 클래스마다 필드와 메소드가 제각기 다른 이름을 가질 수 있음
- 실체 클래스를 작성할 때 시간 절약
 - 실체 클래스는 추가적인 필드와 메소드만 선언
- 실체 클래스 설계 규격을 만들고자 할 때
 - 실체 클래스가 가져야 할 필드와 메소드를 추상 클래스에 미리 정의
 - 실체 클래스는 추상 클래스를 무조건 상속 받아 작성



추상 클래스(Abstract Class)

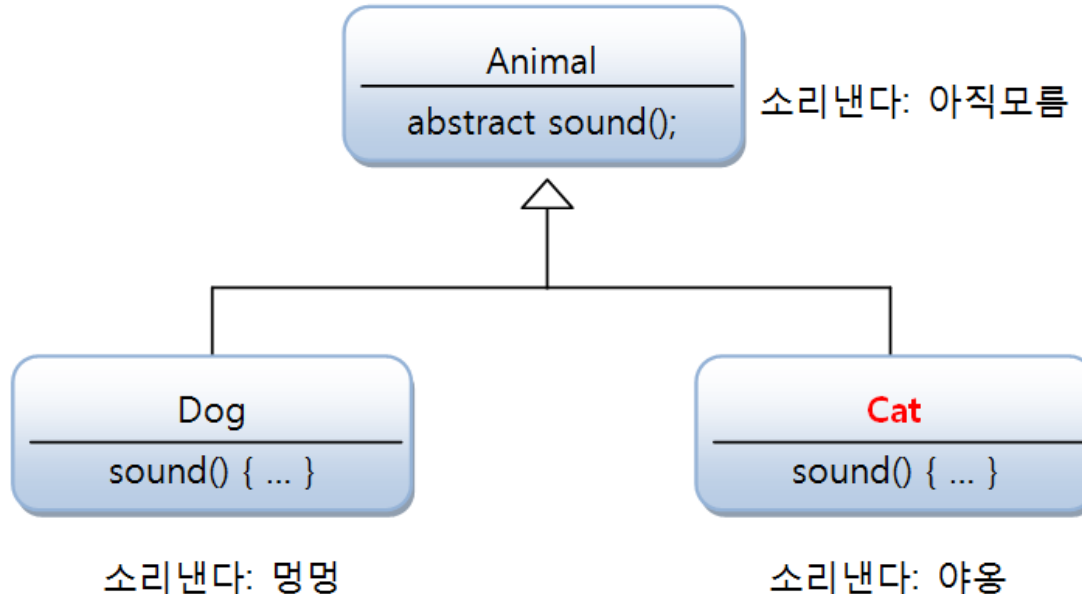
추상 클래스는 기능이 무엇(what)인지만을 정의하고 어떻게(how) 구현되는지는 정의하지 않는다. 그래서 하나의 추상 클래스에 정의된 기능을 여러 개의 하위 클래스에서 서로 다른 형태로 구현하여 사용할 수 있다.

추상 클래스(abstract class)는 완전하게 구현되어 있지 않은 메소드를 가지고 있는 클래스를 의미한다. 메소드가 미완성되어 있으므로 추상 클래스로는 객체를 생성할 수 없다.

추상 클래스(Abstract Class)

❖ 추상 메소드와 오버라이딩(재정의)

- 메소드 이름 동일하지만, 실행 내용이 실제 클래스마다 다른 메소드
- 예: 동물은 소리를 낸다. 하지만 실제 동물들의 소리는 제각기 다르다.
- 구현 방법
 - 추상 클래스에는 메소드의 선언부만 작성 (추상 메소드)
 - 실제 클래스에서 메소드의 실행 내용 작성(오버라이딩(Overriding))





추상 클래스(Abstract Class)

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

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

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



추상 클래스

자바에서 추상 클래스를 만들기 위해서는 클래스 선언 시에 앞에 **abstract**를 붙인다.

추상 클래스: 추상 메서드를 가지고 있는 클래스

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

추상 메소드 정의

추상 클래스는 일반적으로 추상 메소드를 가지고 있다. 추상 메소드란 몸체가 없는 메소드를 말한다. 추상 메소드는 항상 세미 콜론(;)으로 종료되어야 한다.



추상 클래스

- 추상 클래스의 특징은 다음과 같다
 - ✓ 추상 클래스 = 필드 + 일반 메서드 + 추상 메서드
 - ✓ `abstract`로 선언된 추상 클래스는 객체를 생성할 수 없다.
 - ✓ `abstract`로 선언된 메서드는 구현 코드를 갖지 않고 선언만 할 수 있다.
 - ✓ 추상 메서드를 하나라도 포함한 클래스는 추상 클래스가 된다.
 - ✓ 추상 클래스의 추상 메서드에는 선언부만 있기 때문에 이를 상속한 하위 클래스에서는 반드시 재정의하여야 한다.
 - ✓ 단일 상속이라는 제한을 받는다.



추상 클래스의 예

도형을 나타내는 클래스 계층 구조를 생각하여 보자. 각 도형은 공통적인 어떤 속성(위치, 회전 각도, 선 색상, 채우는 색 등)을 가지고 있다.

```
abstract class Shape {  
    int x, y;  
    public void move(int x, int y){  
        this.x = x;  
        this.y = y;  
    }  
    public abstract void draw();  
}
```



추상 클래스의 예

```
class Rectangle extends Shape {  
    int width, height;  
    public void draw(){  
        System.out.println("사각형 그리기 메소드");  
    }  
}
```

```
class Circle extends Shape {  
    int radius;  
    public void draw(){  
        System.out.println("원 그리기 메소드");  
    }  
}
```



추상 메소드와 추상 클래스

추상 메소드(abstract method)

- 선언되어 있으나 구현되어 있지 않은 메소드, `abstract`로 선언

```
public abstract String getName();  
public abstract void setName(String s);
```

- 추상 메소드는 서브 클래스에서 오버라이딩하여 구현해야 함

추상 클래스(abstract class)의 2종류

1. 추상 메소드를 하나라도 가진 클래스
 - 클래스 앞에 반드시 `abstract`라고 선언해야 함
2. 추상 메소드가 하나도 없지만 `abstract`로 선언된 클래스



2 가지 종류의 추상 클래스 사례

// 1. 추상 메소드를 포함하는 추상 클래스

```
abstract class Shape { // 추상 클래스 선언
    public Shape() { }
    public void paint() { draw(); }
    abstract public void draw(); // 추상 메소드
}
```

// 2. 추상 메소드 없는 추상 클래스

```
abstract class MyComponent { // 추상 클래스 선언
    String name;
    public void load(String name) {
        this.name = name;
    }
}
```


추상 클래스는 객체를 생성할 수 없다

```
abstract class Shape {  
    ...  
}  
  
public class AbstractError {  
    public static void main(String [] args) {  
        Shape shape;  
        shape = new Shape(); // 컴파일 오류. 추상 클래스 Shape의 객체를 생성할 수 없다.  
        ...  
    }  
}
```

Exception in thread "main" java.lang.Error: Unresolved compilation problem:
Cannot instantiate the type Shape

at chap5.AbstractError.main(AbstractError.java:4)



추상 클래스의 상속

추상 클래스의 상속 2 가지 경우

1. 추상 클래스의 단순 상속

- 추상 클래스를 상속받아, 추상 메소드를 구현하지 않으면 추상 클래스 됨
- 서브 클래스도 **abstract**로 선언해야 함

```
abstract class Shape { // 추상 클래스
    public Shape() { }
    public void paint() { draw(); }
    abstract public void draw(); // 추상 메소드
}
```

```
abstract class Line extends Shape { // 추상 클래스. draw()를 상속받기 때문
    public String toString() { return "Line"; }
}
```

2. 추상 클래스 구현 상속

- 서브 클래스에서 슈퍼 클래스의 추상 메소드 구현(오버라이딩)
- 서브 클래스는 추상 클래스 아님

추상 클래스의 구현 및 활용 예

Line, Rect, Circle은 추상클래스 Shape를 상속받아 만든 서브 클래스들로서, draw()를 오버라이딩하여 구현한 사례입니다. 그러므로 Line, Rect, Circle은 추상 클래스가 아니며 이들의 인스턴스를 생성할 수 있습니다.



```
class Shape {  
    public void draw() {  
        System.out.println("Shape");  
    }  
}
```

추상 클래스로 수정

```
abstract class Shape {  
    public abstract void draw();  
}
```

```
class Line extends Shape {  
    @Override  
    public void draw() {  
        System.out.println("Line");  
    }  
}
```

```
class Rect extends Shape {  
    @Override  
    public void draw() {  
        System.out.println("Rect");  
    }  
}
```

```
class Circle extends Shape {  
    @Override  
    public void draw() {  
        System.out.println("Circle");  
    }  
}
```

draw()라고 하면 컴파일 오류가 발생.
추상 메소드 draw()를 구현하지 않았기 때문



추상 클래스의 용도

•설계와 구현 분리

- 슈퍼 클래스에서는 개념 정의
 - 서브 클래스마다 다른 구현이 필요한 메소드는 추상 메소드로 선언
- 각 서브 클래스에서 구체적 행위 구현
 - 서브 클래스마다 목적에 맞게 추상 메소드 다르게 구현

•계층적 상속 관계를 갖는 클래스 구조를 만들 때

추상 클래스의 구현 연습

다음 추상 클래스 Calculator를 상속받은 GoodCalc 클래스를 구현하라.

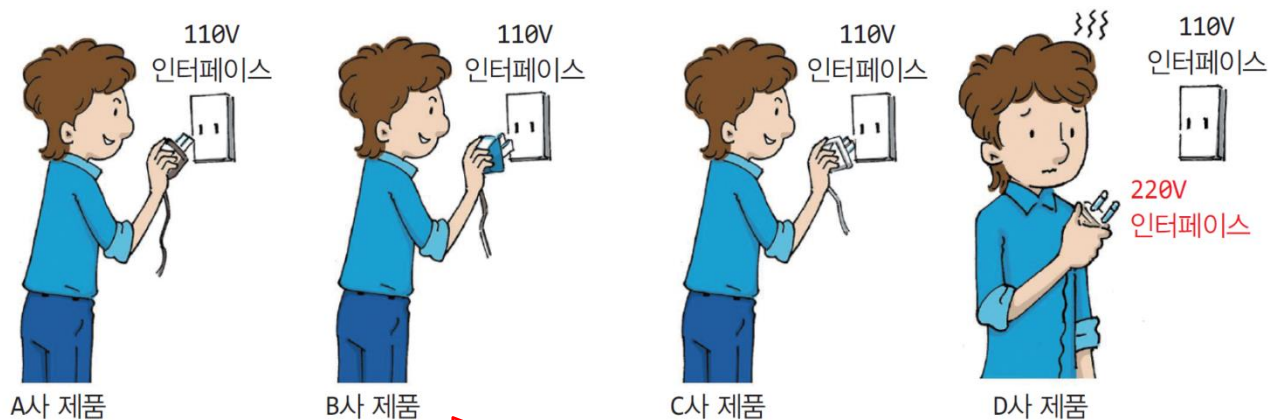
```
abstract class Calculator {  
    public abstract int add(int a, int b);  
    public abstract int subtract(int a, int b);  
    public abstract double average(int[] a);  
}
```

```
public class GoodCalc extends Calculator {  
    @Override  
    public int add(int a, int b) { return a + b; }  
    @Override  
    public int subtract(int a, int b) { return a - b; }  
    @Override  
    public double average(int[] a) { // 추상 메소드 구현  
        double sum = 0;  
        for (int i = 0; i < a.length; i++)  
            sum += a[i];  
        return sum/a.length;  
    }  
    public static void main(String [] args) {  
        GoodCalc c = new GoodCalc();  
        System.out.println(c.add(2,3));  
        System.out.println(c.subtract(2,3));  
        System.out.println(c.average(new int [] { 2,3,4 }));  
    }  
}
```


인터페이스

1) 인터페이스란?

인터페이스(interface)는 객체의 설계도인 클래스와는 약간 다르다. 인터페이스는 객체들 사이에 상호작용하기 위한 개념이다. 자바에서는 객체와 객체 사이의 상호 작용을 위한 인터페이스이다. 다시 말해 인터페이스는 여러 프로그램에서 사용할 멤버(필드, 메서드)를 일관되게 하기 위한 기술 명세이다.



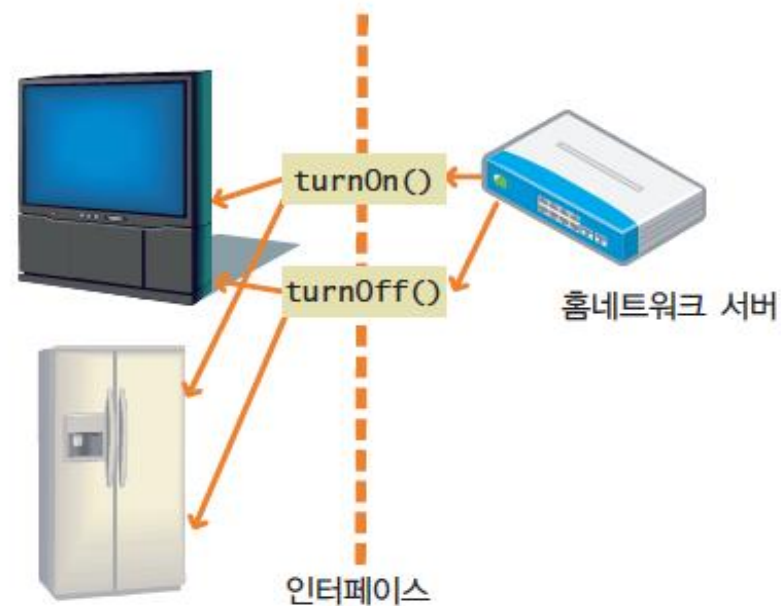
정해진 규격(인터페이스)에 맞기만 하면 연결 가능.
각 회사마다 구현 방법은 다름

정해진 규격(인터페이스)에
맞지 않으면 연결 불가

클래스와 클래스
사이의 상호
작용의 규격을
나타낸 것이 인
터페이스이다.

인터페이스의 필요성

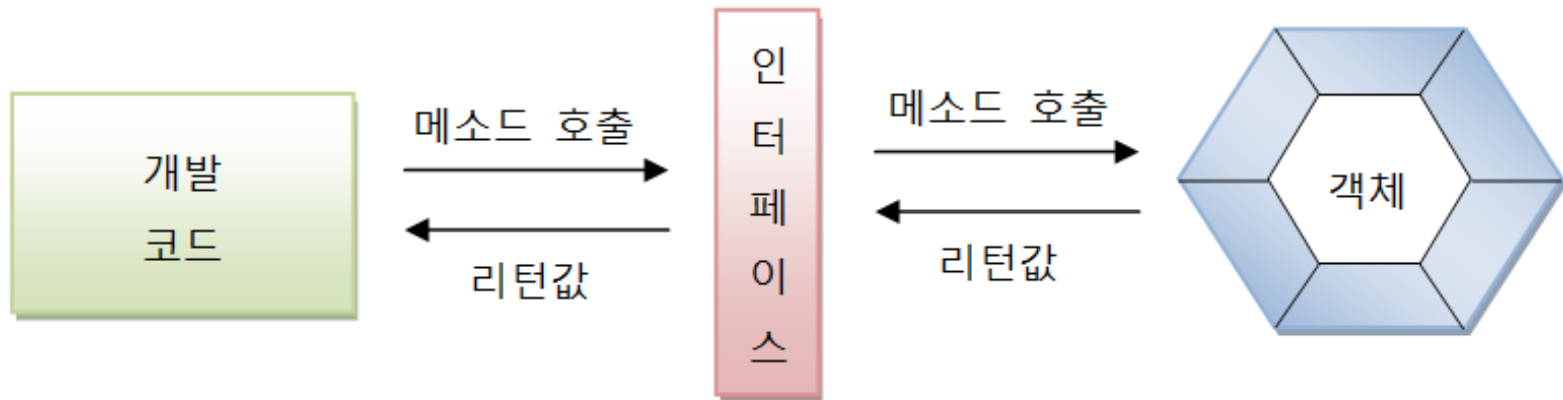
예를 들어서 홈 네트워크 시스템을 생각하여 보자. 홈 네트워크 시스템이란 가정에서 쓰이는 모든 가전 제품들이 유무선 하나의 시스템으로 연결, 쌍방향 통신이 가능한 미래형 가정 시스템을 말한다. 가전 제조사들과 홈 네트워크 업체 사이에는 가전 제품을 제어할 수 있는 일종의 표준규격이 필요하다. 자바로 작성한다면 이 표준규격은 인터페이스로 나타난다.



인터페이스의 역할

❖ 인터페이스란?

- 개발 코드와 객체가 서로 통신하는 접점
 - 개발 코드가 인터페이스의 메서드를 호출하면 인터페이스는 객체의 메서드를 호출 시킨다.
 - 개발 코드는 **인터페이스의 메소드만 알고 있으면 OK**

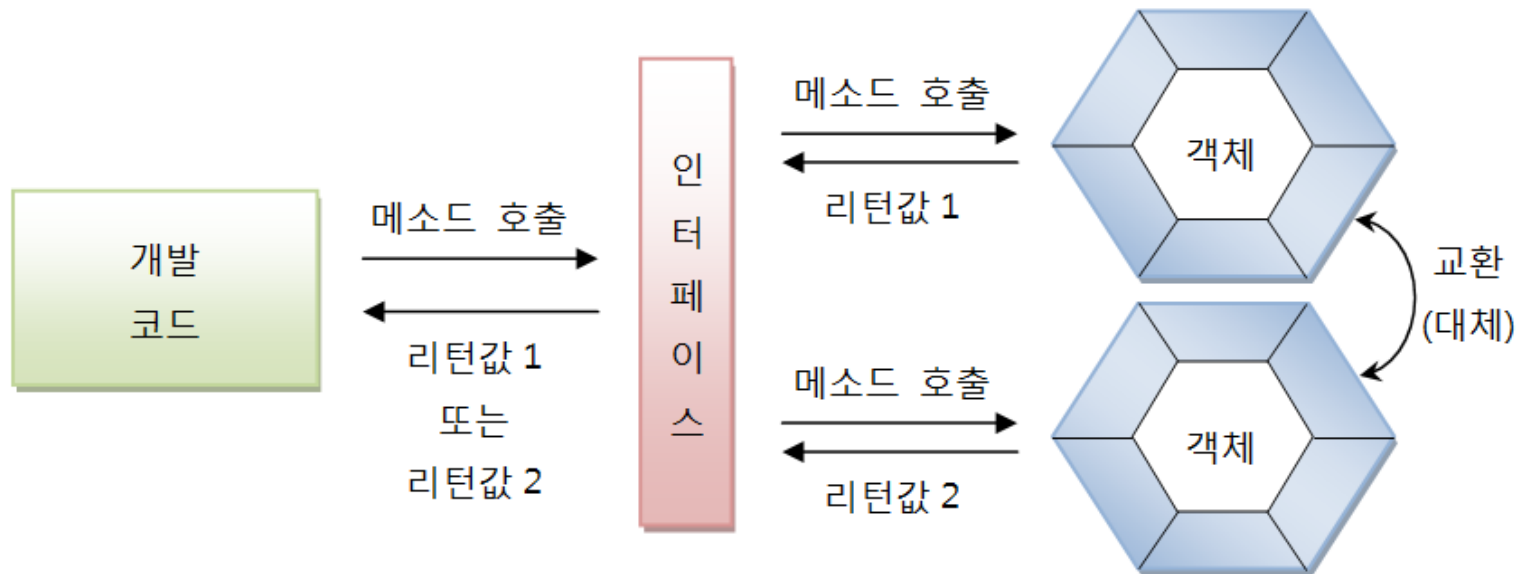


인터페이스의 역할

❖ 인터페이스란?

■ 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





인터페이스 선언

인터페이스를 정의하는 것은 클래스를 정의하는 것과 유사하다. 인터페이스는 추상 메소드(public abstract)와 상수(public static final)로 이루어진다.

```
public interface 인터페이스명 {  
    public static final 자료형 변수명 = 값;  
    public abstract 반환형 메소드명1();  
    public abstract 반환형 메소드명2();  
}
```

인터페이스에 선언된 필드는 모두 상수이기 때문에 public static final 생략할 수 있다.

인터페이스 내에는 추상 메소드만 정의할 수 있기에 public abstract는 생략할 수 있다.



인터페이스 선언

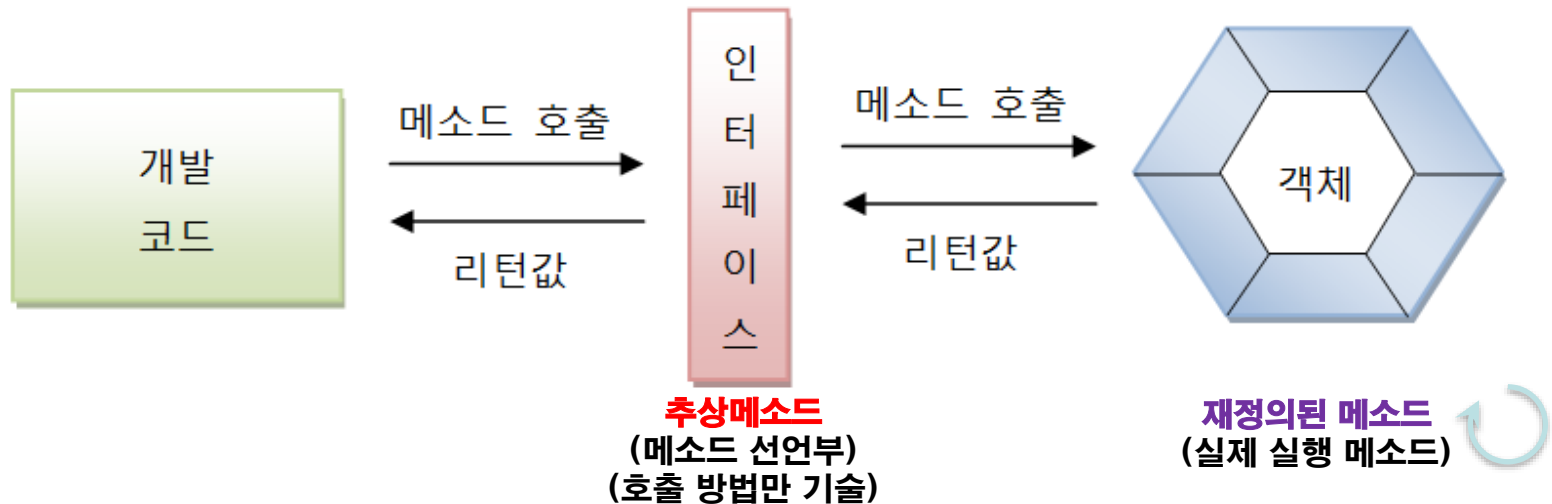
❖ 상수 필드 선언

- 인터페이스는 상수 필드만 선언 가능
 - 데이터 저장하지 않음
- 인터페이스에 선언된 필드는 모두 `public static final`
 - 자동적으로 컴파일 과정에서 붙음
- 상수명은 대문자로 작성
 - 서로 다른 단어로 구성되어 있을 경우에는 언더 바(_)로 연결
- 선언과 동시에 초기값 지정
 - `static { }` 블록 작성 불가 - `static { }` 으로 초기화 불가

인터페이스 선언

❖ 추상 메소드 선언

- 인터페이스 통해 호출된 메소드는 최종적으로 객체에서 실행
 - 인터페이스의 메소드는 기본적으로 실행 블록이 없는 추상 메소드로 선언
 - `public abstract`를 생략하더라도 자동적으로 컴파일 과정에서 붙게 됨



인터페이스 구현

❖ 구현 객체와 구현 클래스

- 인터페이스의 추상 메소드 대한 실제 메소드를 가진 객체 = 구현 객체



- 구현 객체를 생성하는 클래스 = 구현 클래스



인터페이스 구현

추상 클래스처럼 인터페이스도 객체를 생성할 수 없다. 다만 다른 클래스에 의하여 구현(implement)될 수 있다. 인터페이스를 구현한다는 말은 인터페이스에 정의된 추상 메소드의 재정의한다는 의미이다. (구현 클래스 선언)

```
public class 클래스명 implements 인터페이스명 {  
    반환형 추상메서드1(){ //재정의  
        ...  
    }  
    반환형 추상메서드2(){ //재정의  
        ...  
    }  
}
```

메소드의 선언부가 정확히 일치해야 한다.

인터페이스의 모든 추상 메소드를 재정의하여 작성해야 한다.



인터페이스의 선언

예를 들어 홈 네트워크의 가전 제품들을 원격 조정하기 위한 인터페이스를 정의하여 보자.

```
public interface RemoteControl {  
    public abstract void turnOn();  
    public abstract void turnOff();  
}
```

인터페이스에서 메소드는 시그너처만 있고 내용이 없으며 세미콜론으로 종료되어야 한다.

인터페이스 안에서 선언된 메소드들은 모두 묵시적으로 public abstract이다. 따라서 생략이 가능하다.



인터페이스의 구현

```
public class Television implements RemoteControl {  
    public void turnOn(){  
        //실제로 TV의 전원을 켜기 위한 코드 작성  
    }  
  
    public void turnOff(){  
        //실제로 TV의 전원을 끄기 위한 코드 작성  
    }  
}
```

인터페이스의 사용

TV 생산 업체마다 다르게 인터페이스를 구현할 것이다. 하지만 동일한 제어 인터페이스인 RemoteControl 인터페이스를 지원한다.

```
Television t = new Television();  
t.turnOn();  
t.turnOff();
```





인터페이스의 사용

냉장고를 나타내는 Refrigerator 클래스에서도 동일한 메소드로 제어할 수 있다.

```
Refrigerator r = new Refrigerator();  
r.turnOn();  
r.turnOff();
```



여러 개의 인터페이스 동시 구현

하나의 클래스는 하나 이상의 인터페이스를 구현할 수도 있다. TV는 시리얼 통신(SerialCommunication)을 할 수 있는 인터페이스도 가지고 있다고 가정하자.

```
public interface SerialCommunication {  
    void send(byte[] data); //데이터를 전송한다  
    byte[] receive();      //데이터를 받는다  
}
```



여러 개의 인터페이스 동시 구현

Television 클래스에서 RemoteControl과 SerialCommunication이라는 두 개의 인터페이스를 동시에 구현할 수 있다.

```
public class Television implements RemoteControl,
SerialCommunication {
    public void turnOn(){ ... }
    public void turnOff(){ ... }
    public void send(byte[] data) { ... }
    public byte[] receive() { ... }
}
```




인터페이스 상속

만약 다른 프로그래머들이 사용하고 있던 인터페이스 변경시키려면 이 인터페이스를 구현하였던 모든 클래스가 동작하지 않게 된다. 이런 경우를 대비하여서 인터페이스도 상속을 받아서 확장시킬 수 있도록 되어 있다.

```
public interface RemoteControl {  
    public void turnOn();  
    public void turnOff();  
    public void volumeUp();  
    public void volumeDown();  
}
```



인터페이스 상속

```
public interface RemoteControl {  
    public void turnOn();  
    public void turnOff();  
}
```

```
public interface AdvancedRemoteControl extends  
RemoteControl {  
    public void volumeUp();  
    public void volumeDown();  
}
```



여러 개의 인터페이스 동시 구현

```
interface TestA{
    int getValue();
}
interface TestB {
    int getValue();
}
interface TestC extends TestA, TestB {    }

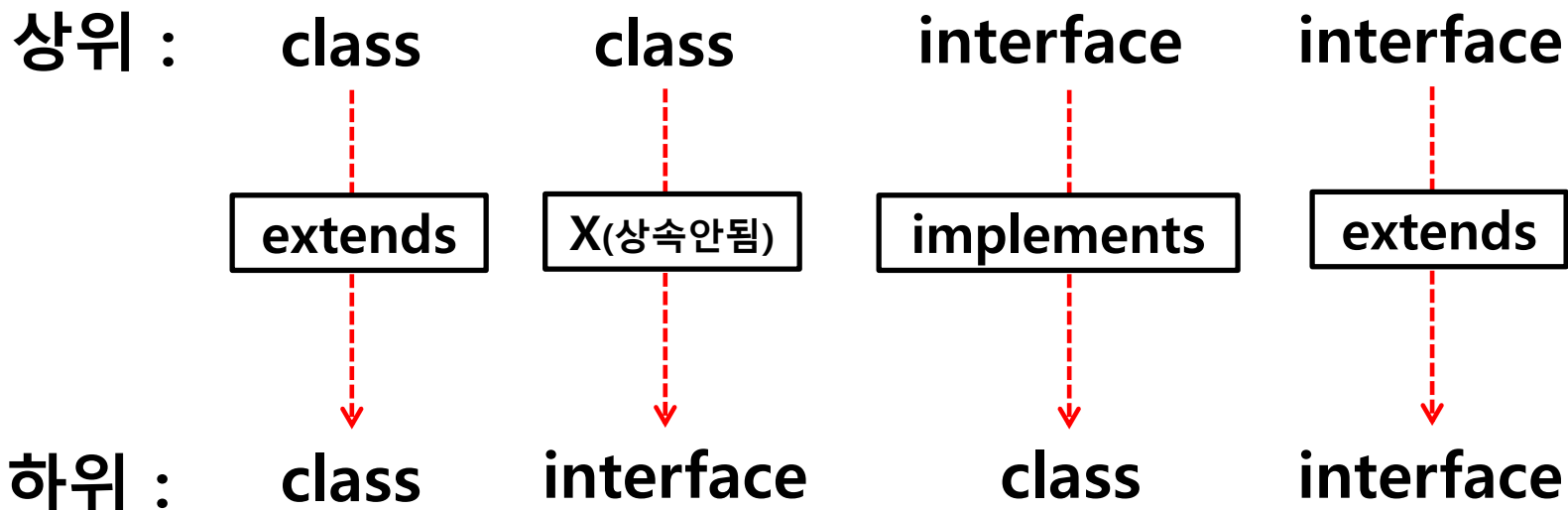
class TestD implements TestC {
    @Override
    public int getValue() {
        return 200;
    }
}
```

인터페이스 상속

클래스와는 다르게 인터페이스는 여러 개의 인터페이스로부터 상속을 동시에 받을 수 있다.

콤마로 분리하여서 부모 인터페이스를 적어주면 된다.

클래스에서는 하나의 부모로부터만 상속받을 수 있음을 유의하라.





자바의 인터페이스

자바 인터페이스에 대한 변화

- Java 7까지
 - 인터페이스는 상수와 추상 메소드로만 구성
- Java 8부터
 - 상수와 추상메소드 포함
 - default 메소드 포함 (Java 8)
 - private 메소드 포함 (Java 9)
 - static 메소드 포함 (Java 9)
- 여전히 인터페이스에는 필드(멤버 변수) 선언 불가

```
interface PhoneInterface { // 인터페이스 선언
    public static final int TIMEOUT = 10000; // 상수 필드 public static final 생략 가능
    public abstract void sendCall(); // 추상 메소드 public abstract 생략 가능
    public abstract void receiveCall(); // 추상 메소드 public abstract 생략 가능
    public default void printLogo() { // default 메소드 public 생략 가능
        System.out.println("** Phone **");
    }; // 디폴트 메소드
}
```



인터페이스 선언(JDK1.8)-디폴트메서드

❖ 디폴트 메소드 선언

- 자바8에서 추가된 인터페이스의 새로운 멤버

```
public default 리턴타입 메서드명(매개변수, ...) {  
    ...  
}
```

- 실행 블록을 가지고 있는 메소드이다.
- 인터페이스를 구현하는 클래스에 자동 상속되는 메소드이다.
- default 키워드를 반드시 붙여야 한다.
- 기본적으로 public 접근 제한
 - 생략하더라도 컴파일 과정에서 자동 붙음



디폴트 메서드

```
interface A {  
    public default int getValue() {  
        return 123;  
    }  
}  
  
class B implements A { }  
  
public class InterfaceTest {  
    public static void main(String[] args) {  
        B b = new B();  
        int n = b.getValue();  
        System.out.println("n=" + n);  
    }  
}
```



두 인터페이스의 디폴트 메서드가 중복되는 경우

```
interface Buy {  
    void buy();  
  
    default void order(){  
        System.out.println("구매 주문");  
    }  
}  
interface Sell {  
    void sell();  
  
    default void order(){  
        System.out.println("판매 주문");  
    }  
}
```



두 인터페이스의 디폴트 메서드가 중복되는 경우

```
public class Customer implements Buy, Sell{
```

```
public class Customer implements Buy, Sell{  
    @Override  
    public void buy() {  
        System.out.println("구매하기");  
    }  
    @Override  
    public void sell() {  
        System.out.println("판매하기");  
    }  
}
```



두 인터페이스의 디폴트 메서드가 중복되는 경우

```
public class Customer implements Buy, Sell{  
    @Override  
    public void buy() {  
        System.out.println("구매하기");  
    }  
    @Override  
    public void sell() {  
        System.out.println("판매하기");  
    }  
    @Override  
    public void order() {  
        System.out.println("고객 판매 주문");  
    }  
}
```



두 인터페이스의 디폴트 메서드가 중복되는 경우

```
public class Customer implements Buy, Sell{  
    @Override  
    public void buy() {  
        System.out.println("구매하기");  
    }  
    @Override  
    public void sell() {  
        System.out.println("판매하기");  
    }  
    @Override  
    public void order() {  
        Buy.super.order();  
    }  
}
```

인터페이스명.super.메서드명()으로 호출한다.



클래스와 인터페이스에 동일한 이름으로 메서드

```
class Person{  
    private String name;  
    private String phone;  
  
    public Person() {  
        this.name = "홍길동";  
        this.phone = "010-2345-6734";  
    }  
    public String getName() {  
        return name;  
    }  
    ...  
}
```


클래스와 인터페이스에 동일한 이름으로 메서드

```
interface Naming{
    String COMPANY_NAME = "future";
    default String getName() {
        return COMPANY_NAME;
    }
}

class Employee extends Person implements Naming{
}

public class InterfaceExample{
    public static void main(String[] args) {
        Employee emp = new Employee();
        System.out.println(emp.getName());
    }
}
```

결과?

클래스와 인터페이스에 동일한 이름으로 메서드

```
class Employee extends Person implements Naming{  
    @Override  
    public String getName() {  
        return "Java";  
    }  
}
```

상위 클래스의 getName() 메서드 호출

return super.getName();

디폴트 getName() 메서드 호출

return Naming.super.getName();



인터페이스 선언(JDK1.8)-정적메서드

❖ 정적 메소드 선언

- 자바8에서 추가된 인터페이스의 새로운 멤버

```
public static 리턴타입 메서드명(매개변수, ...) {  
    ...  
}
```



정적 메서드

```
interface TestA{  
    public static int getValue() {  
        return 123;  
    }  
}  
  
interface TestB {  
    public static int getValue() {  
        return 456;  
    }  
}
```

정적 메서드

```
class TestD implements TestA, TestB {
```

```
}
```

```
public class InterfaceTest{
```

```
    public static void main(String[] args) {
```

```
        System.out.println(TestD.getValue());
```

```
        System.out.println(TestA.getValue());
```

```
        System.out.println(TestB.getValue());
```

```
    }
```

```
}
```

결과?

결과?



인터페이스의 활용

1) 인터페이스와 다중 상속

인터페이스의 또 다른 중요한 사용처는 클래스에게 다중 인터페이스를 가지게 하는 능력이다. 자바에서는 지원되지 않는 다중 상속(multiple inheritance)의 개념을 변형시킨 것이다. 인터페이스는 일반적인 변수와 메소드를 가지지 않음을 유의하라.

인터페이스의 활용

```
class SuperA { int x; }  
class SuperB { int x; }  
class Sub extends SuperA , SuperB {  
    ...  
}
```



```
// main 메소드에서 다음과 같이 정의  
Sub obj = new Sub();  
obj.x = 10;
```



인터페이스의 활용

이런 복잡한 문제 때문에 자바에서는 하나의 클래스는 오직 하나의 클래스부터만 상속을 받을 수 있다. 하지만 클래스는 상속과 동시에 여러 개의 인터페이스를 구현할 수 있다. 여러 인터페이스를 구현하기 위해서는 클래스 정의 시에 키워드 `implements` 다음에 콤마(,)로 분리된 인터페이스 이름을 적으면 된다.

```
class Sub extends Super implements Interface1,  
Interface2 {  
    ...  
}
```



인터페이스의 활용

예를 들어 Rectangle 클래스는 Shape로부터 상속을 받고 동시에 Drawable 인터페이스를 구현한다.

```
class Shape {  
    protected int x, y;  
}  
interface Drawable {  
    void draw();  
}  
public class Rectangle extends Shape implements  
Drawable {  
    int width, height;  
    public void draw() {  
        System.out.println("사각형 그리기");  
    }  
}
```



인터페이스의 활용

2) 인터페이스와 추상 클래스

인터페이스는 추상 클래스와도 비슷하다. 추상 클래스는 추상 메소드 때문에 부분적으로 미완성된 클래스라고 했다. 인터페이스는 오로지 상수와 추상 메서드로만 구성되어 있기 때문에 역시 완전한 형태는 아니다. 인터페이스와는 다르게 추상 클래스는 일반적인 필드나 메소드를 가질 수 있다. 그러나 추상클래스는 일부의 구현을 제공한다는 점에서만 인터페이스와 차이가 있다. 왜 인터페이스라는 개념을 사용하는 이유는 자바가 다중 상속을 지원하지 않는다는 점에 있다



인터페이스의 활용

만약 다음과 같이 `MyComparable`을 추상 클래스로 정의하여 보자.

```
abstract class MyComparable {  
    public abstract int compareTo(Object other);  
}
```

그리고 `Circle` 클래스가 `MyComparable`로부터 상속받게 하려고 한다. 하지만 이미 `Rectangle` 클래스는 `Shape` 클래스로부터 상속을 받고 있다. 따라서 컴파일 오류가 발생한다.

```
public class Circle extends Shape, MyComparable {  
    ...  
}
```



인터페이스의 활용

이 문제를 해결하는 방법은 `MyComparable`을 인터페이스로 정의하는 것이다.

```
public class Circle extends Shape implements  
MyComparable {  
    ...  
}
```




인터페이스의 활용

3) 상수 정의

인터페이스의 또 다른 용도는 여러 클래스에서 사용되는 상수들을 정의하는 것이다.

```
interface Days {  
    public static final int SUNDAY=0, MONDAY=1,  
                           TUESDAY=2, WEDNESDAY=3,  
                           THURSDAY=4, FRIDAY=5,  
                           SATURDAY=6;  
}
```




추상 클래스와 인터페이스 비교

유사점

- 객체를 생성할 수 없고, 상속을 위한 슈퍼 클래스로만 사용
- 클래스의 다형성을 실현하기 위한 목적

다른 점

비교	목적	구성
추상 클래스	추상 클래스는 서브 클래스에서 필요로 하는 대부분의 기능을 구현하여 두고 서브 클래스가 상속받아 활용할 수 있도록 하되, 서브 클래스에서 구현할 수밖에 없는 기능만을 추상 메소드로 선언하여, 서브 클래스에서 구현하도록 하는 목적(다형성)	<ul style="list-style-type: none">• 추상 메소드와 일반 메소드 모두 포함• 상수, 변수 필드 모두 포함
인터페이스	인터페이스는 객체의 기능을 모두 공개한 표준화 문서와 같은 것으로, 개발자에게 인터페이스를 상속받는 클래스의 목적에 따라 인터페이스의 모든 추상 메소드를 만들도록 하는 목적(다형성)	<ul style="list-style-type: none">• 변수 필드(멤버 변수)는 포함하지 않음• 상수, 추상 메소드, 일반 메소드, default 메소드, static 메소드 모두 포함• protected 접근 지정 선언 불가• 다중 상속 지원



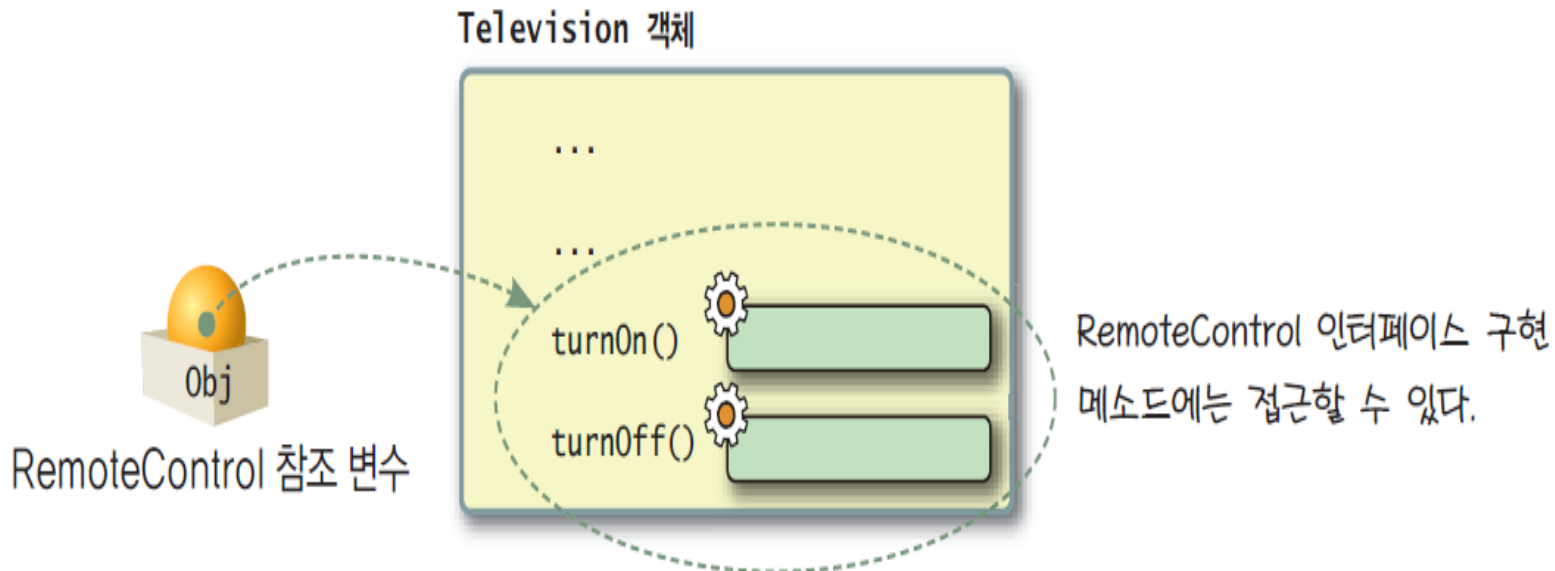
인터페이스와 타입

인터페이스는 하나의 타입(Type)으로 봐야 한다. 이 타입은 인터페이스를 구현한 클래스들을 하나로 묶는 역할을 한다. 인터페이스의 이름은 클래스의 이름과 마찬가지로 참조 변수를 정의하는데 사용될 수 있다. 만약 인터페이스 타입의 참조 변수를 정의하였다면 이 변수에 대입할 수 있는 값은 반드시 그 인터페이스를 구현한 클래스의 객체이어야 한다.

인터페이스와 타입

예를 들어 RemoteControl 인터페이스를 구현한 Television 클래스의 객체를 생성한다면

```
RemoteControl obj = new Television();  
obj.turnOn();  
obj.turnOff();
```





다형성

'다형성'은 여러 개의 서로 다른 형식과 모양을 가진다는 의미로, 다형성을 이용하면 객체의 유형에 따라 서로 다른 작업을 수행하도록 할 수 있다.

객체는 한가지 타입에 하나의 형태만을 생성하는 고정적인 형태보다, 한가지 타입으로 여러 형태를 생성할 수 있는 유동적인 형태가 프로그램을 더욱 편하게 작성하게 해준다



다형성

다형성은 동일한 부모 클래스에서 상속된 서브 클래스의 객체들을 하나의 타입(type)으로 취급할 수 있게 해준다. 따라서 서로 다른 타입들을 받아서 하나의 코드로 처리할 수 있다.

다형성은 코드 구조가 향상되고 가독성을 증가시키며, 확장이 가능한 프로그램을 작성하는 객체 지향의 중요한 개념 중의 하나이다.



다형성(정리)

- 정의

하나의 타입에 여러 가지 객체 대입해 다양한 실행 결과를 얻는 것

- 다형성을 구현하는 기술

- 상속 또는 인터페이스의 자동 타입 변환(Promotion)
- 오버라이딩(Overriding)

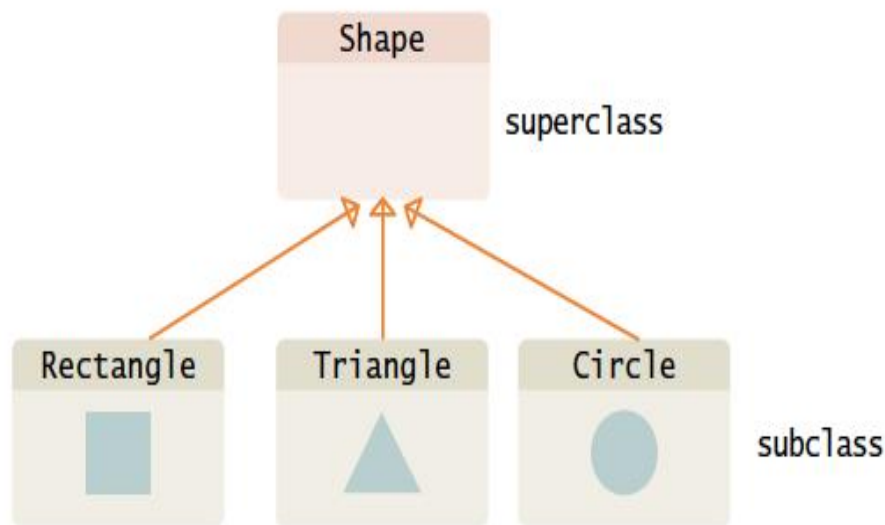
- 다형성의 효과

- 코드 구조가 향상되어 가독성을 증가
- 유지보수 용이 (메소드의 매개변수로 사용)

다형성

1) 상향 형변환

자바에서는 슈퍼 클래스 객체가 있는 곳을 서브 클래스 객체로 대치하는 것이 가능하다. 이것을 상향 형변환 (upcasting)이라고 한다. 자바는 이것을 자동적으로 한다. 예를들어 Rectangle, Triangle, Circle 클래스가 슈퍼 클래스인 Shape 클래스로부터 상속되었다고 가정한다.





다형성

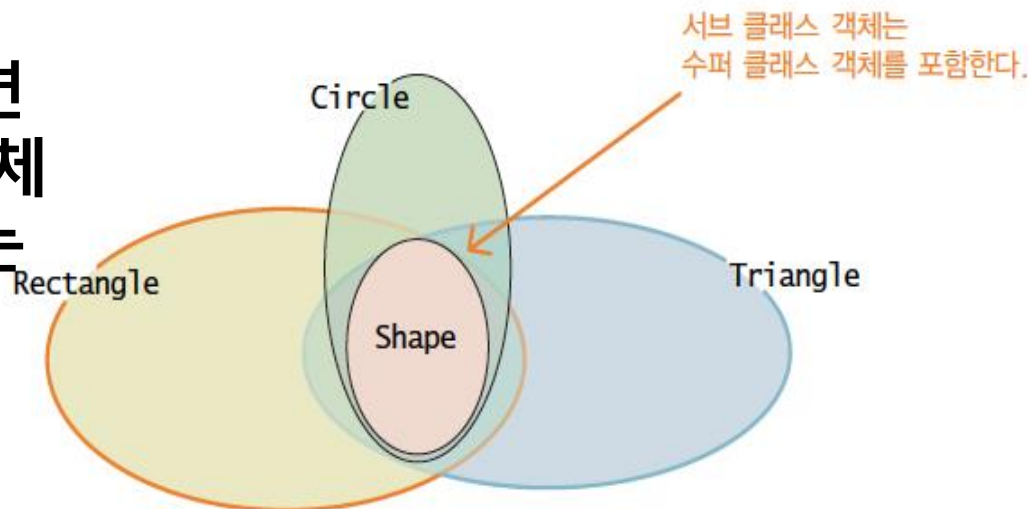
```
public class Shape {  
    private String color;  
    public void setColor(String color){ this.color = color;}  
    public String getColor(){return color;}  
}  
  
public class Rectangle extends Shape {  
    private double width;  
    private double height;  
    public String toString() {  
        return "color : " + getColor()  
            + "width : " + width  
            + "height" + height;  
    }  
}
```

다형성

```
Shape s = new Shape();  
Rectangle r = new Rectangle();
```

```
Shape obj = new Rectangle();
```

상위 클래스형의 참조변수가 하위 클래스의 객체를 가리키도록 변환하는 것을 업캐스팅(upcasting)이라 한다.



다형성

```
class Shape {  
    int x, y;  
}  
class Rectangle extends Shape {  
    int width, height;  
}
```

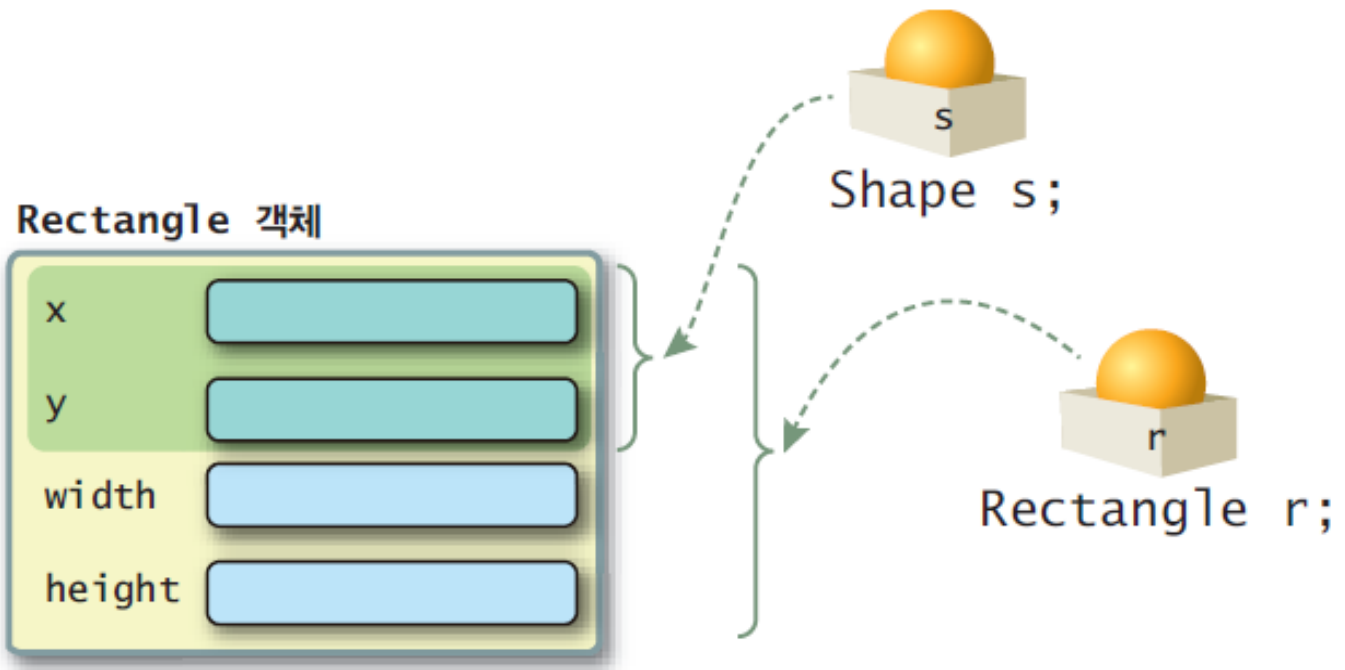
```
public class ShapeTest {  
    public static void main(String[] args){  
        Rectangle r = new Rectangle();  
        Shape s = r;  
        s.x = 0;  
        s.y = 0;  
        s.width = 100;  
        s.height = 100; }  
}
```

Shape s = new Rectangle();

결과는?

다형성

s를 통해서만 x, y만을 사용할 수 있고 r을 통해서
는 모든 필드를 사용할 수 있다.





다형성

```
class Circle extends Shape {  
    int radius;  
}
```

```
public class ShapeTest {  
    public static void main(String[] args){  
        Shape s = new Rectangle();  
        s.x = 0;  
        s.y = 0;  
  
        s = new Circle();  
        s.x = 0;  
        s.y = 0;  
    }  
}
```




다형성

```
class A {  
    int a = 10;  
    void b(){ System.out.println("A"); }  
}
```

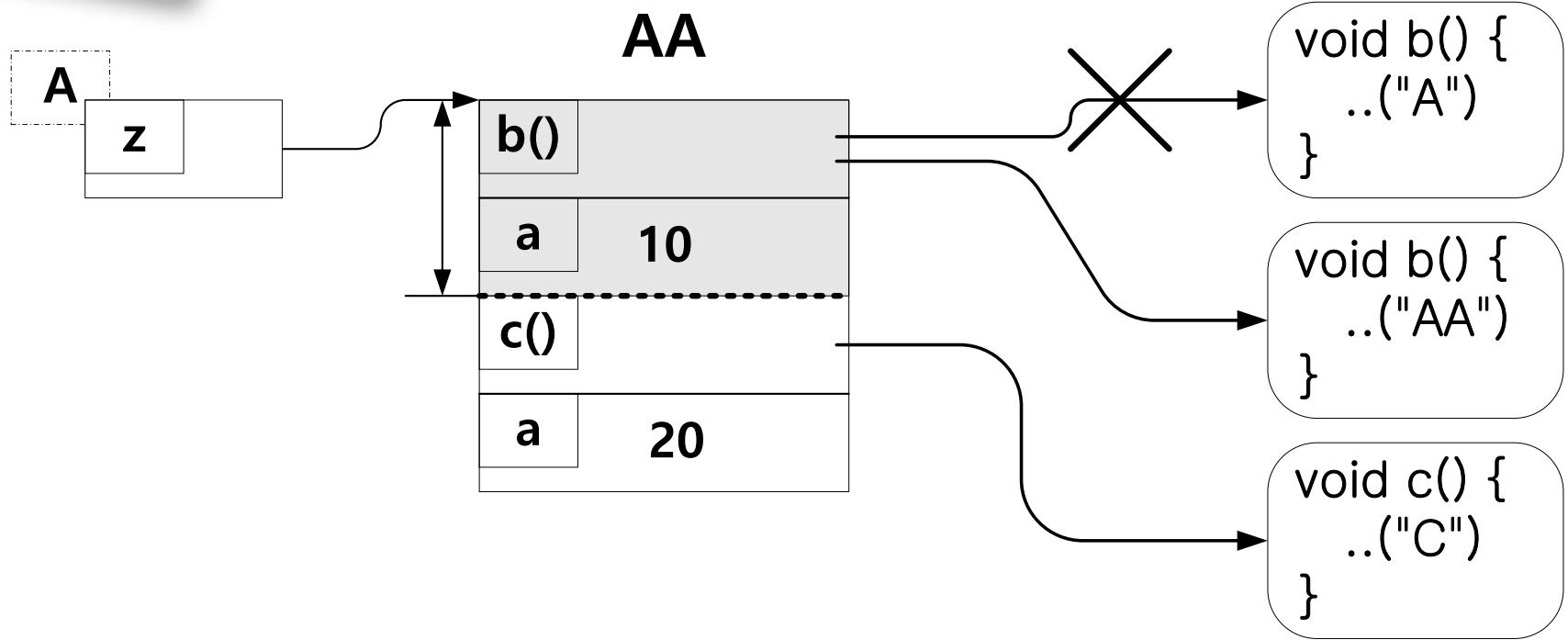
```
class AA extends A {  
    int a = 20;  
    void b(){ System.out.println("AA"); }  
    void c(){ System.out.println("C"); }  
}
```



다형성

```
class ClassTest {  
    public static void main(String[] arg){  
        //A a1 = new A();  
        //AA a2 = new AA();  
  
        A z = new AA();  
        System.out.println(z.a);  
        z.b();  
        z.c(); // 구문의 실행 결과는?  
    }  
}
```

다형성





다형성

업캐스팅을 정리하면 다음과 같다.

슈퍼 클래스 참조변수 = new 서브 클래스생성자();

- 상위 클래스로 형변환이 되는 것이다.
- 컴파일러에 의해 자동으로 형변환된다.
- 업캐스팅이 된 이후 상위 클래스형의 참조변수는 상위 클래스로부터 상속받은 멤버만 호출 할 수 있다.
- 단 업캐스팅이 되더라도 하위 클래스에서 재정의된 메서드는 참조할 수 있고, 상위 클래스로부터 상속받은 메서드는 은닉된다.

다형성

2) 하향 형변환

하향 형변환(downcasting)은 서브 클래스 참조 변수로 슈퍼 클래스 객체를 참조하는 것으로 본다면 컴파일 오류가 발생한다.

```
Rectangle r;  
r = new Shape();
```



다운 캐스팅을 정리하면 다음과 같다.

하위 클래스 = 상위클래스

- 하위 클래스로 형변환이 되는 것이다.
- 컴파일러에 의해 자동으로 형변환되지 않는다.
- 참조 가능한 영역이 확대되는 의미를 갖는다.

다형성

2) 하향 형변환(강제 타입 변환)

- 부모 타입을 자식 타입으로 변환하는 것.

강제 타입 변환

자식클래스 참조변수 = (자식클래스)부모클래스타입;

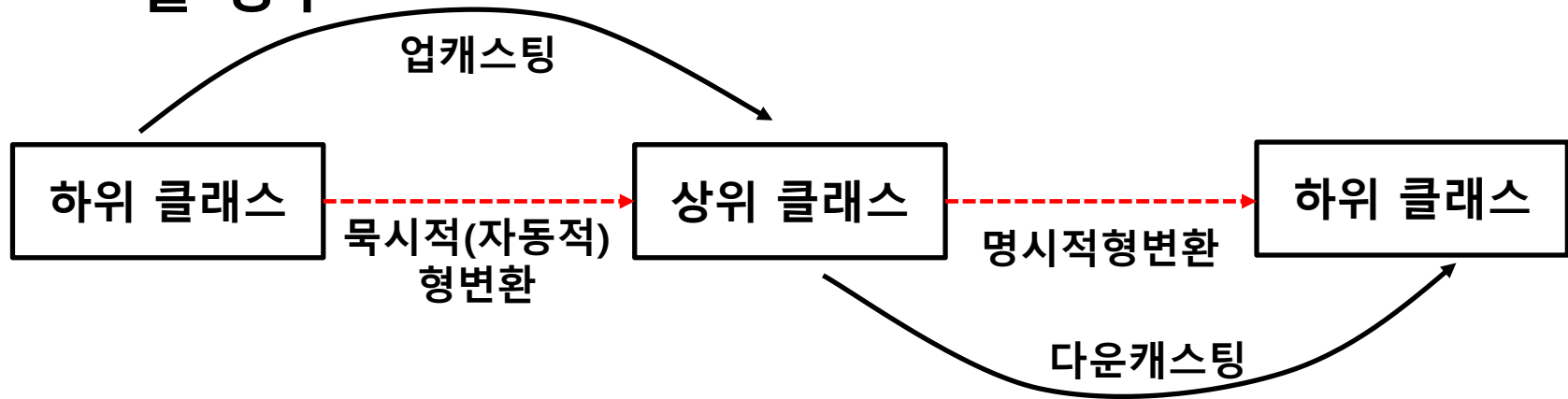
자식 타입이 부모 타입으로 변환된 상태

- 조건

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

다형성

- 강제 타입 변환이 필요한 경우
 - 자식 타입이 부모 타입으로 자동 변환
부모 타입에 선언된 필드와 메소드만 사용 가능
 - 자식 타입에 선언된 필드와 메소드를 다시 사용해야 할 경우





다형성

그러나 서브 클래스 객체인데 형변환에 의하여 일시적으로 수퍼 클래스 참조 변수에 참조되고 있는 경우에는 하향 형 변환을 통하여 원래 상태로 되돌릴 수 있다. 이 때에는 반드시 명시적으로 형변환 연산자를 적여주어야 한다.

```
class Shape { int x, y; }  
class Rectangle extends Shape { int width, height; }
```

```
Shape s = new Rectangle();  
((Rectangle)s).width=100;
```

```
Rectangle r = (Rectangle)s;  
r.width=100;
```



다형성

3) 객체의 타입을 알아내는 방법

객체가 특정 클래스로부터 생성된(상속된) 객체인지를 판별하기 위해 사용할 수 있는 연산자가 instanceof이다. 이때 반환값은 true 또는 false값이다.

```
Shape s = getShape();  
if (s instanceof Rectangle) {  
    System.out.println("Rectangle이 생성되었습니다.");  
} else{  
    System.out.println("Rectangle이 아닌 다른 객체가  
생성되었습니다.");  
}
```

참조변수 instanceof 클래스명



다형성

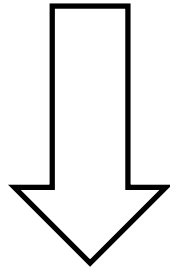
4) 다형성의 이용

첫 번째로 메소드의 매개변수를 선언할 때, 다형성을 많이 이용한다. 일반적으로 메소드 매개변수는 서브 클래스보다는 수퍼 클래스 타입으로 선언하는 것이 좋다.

예를 들어 파생 클래스인 Circle와 Rectangle 객체 타입으로 별도로 선언하는 것보다 수퍼 클래스인 Shape 객체 타입으로 메소드를 선언하는 것이 좋다. Shape 타입으로 선언하면 Shape에서 파생된 모든 객체를 받을 수 있기 때문이다.

다형성

```
public void area(Circle c) { }  
public void area(Rectangle r) { }  
public void area(Triangle t) { }
```



```
public void area(Shape s) { }
```



다형성

아래의 코드는 Shape 타입으로 정의된 매개변수를 이용해서 각 도형의 면적을 구한다.

```
public double area(Shape s){  
    double areaValue = 0.0;  
    if(s instanceof Rectangle) {  
        int w = ((Rectangle)s).getWidth();  
        int h = ((Rectangle)s).getHeight();  
        areaValue = (double)(w*h);  
    }  
    ...  
    return areaValue;  
}
```



다형성

만약 자바에서 생성되는 모든 객체를 전부 받는 메소드를 선언한다면 모든 객체는 Object 클래스를 상속받는다.

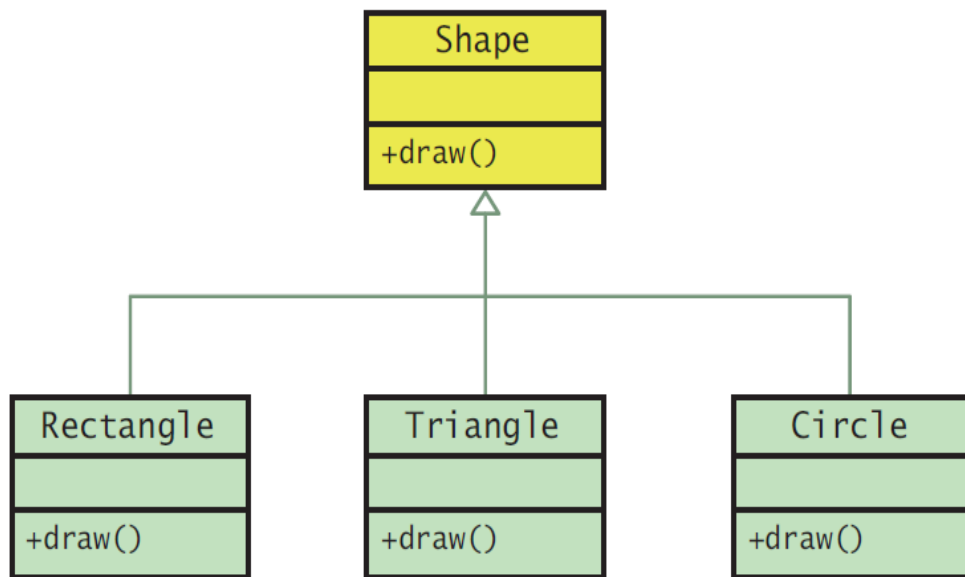
```
public double area(Object s){  
    ...  
}
```

모든 타입의 객체를 전부 받을 수 있다.

다형성

만약 자바에서 생성되는 모든 객체를 전부 받는 메소드를 선언한다면 모든 객체는 Object 클래스를 상속받는다.

모든 도형 클래스는 화면에 자신을 그리기 위한 메소드를 포함하고 있다. 이 메소드의 이름을 draw()라고 하자. 각 도형을 그리는 방법은 당연히 도형에 따라 다르다. 따라서 도형의 종류에 따라 서로 다른 draw()를 호출해야 한다.





다형성

Shape가 draw()메소드를 가지고 있고 Rectangle, Triangle, Circle 클래스들이 이 draw()메소드를 재정의 했을 경우, Shape 참조 변수를 통하여 각 객체들 어느 draw()메소드를 호출하는가?

```
Shape s = new Rectangle();  
s.draw();
```



클래스와 추상 클래스, 인터페이스

클래스와 특징을 표로 정리하면 다음과 같다.

	일반 클래스	추상 클래스	인터페이스
메서드의 재정의	선택적	강제적	강제적
상속, 구현시 예약어	extends	extends	implements
필드 선언 가능 여부	가능	가능	불가능
상수 선언 가능 여부	가능	가능	가능
생성자 선언 여부	가능	가능	불가능
추상 메서드 포함 여부	불포함	포함	포함
객체 생성 가능 여부	가능	불가능	불가능
다중 상속 가능 여부	불가능	불가능	가능
다중 구현 가능 여부	가능	불가능	가능



내부 클래스(중첩 클래스)

- 클래스가 여러 클래스와 관계를 맺는 경우에는 독립적으로 선언하는 것이 좋으나, 특정 클래스와 관계를 맺을 경우에는 관계 클래스를 클래스 내부에 선언하는 것이 좋다.
- 장점
 - 중첩 클래스를 사용하면 두 클래스의 멤버들을 서로 쉽게 접근할 수 있다는 장점.
 - 외부에는 불필요한 관계 클래스를 감춤으로써 코드의 복잡성을 줄일 수 있다는 장점.

중첩 클래스와 중첩 인터페이스란

❖ 중첩 클래스와 중첩 인터페이스란?

- 중첩 클래스: 클래스 멤버로 선언된 클래스

```
public class OuterClass {  
    class InnerClass {  
        ...  
    }  
}
```

중첩 클래스

- 중첩 인터페이스: 클래스 멤버로 선언된 인터페이스
 - UI 컴포넌트 내부 이벤트 처리에 많이 활용

```
public class OuterClass {  
    interface NestedInterface{  
        ...  
    }  
}
```

중첩 인터페이스

내부 클래스(중첩 클래스)

❖ 중첩 클래스의 분류

선언 위치에 따른 분류		선언위치	설명
멤버 클래스	인스턴스 멤버 클래스	<pre>class outer{ class inner{ ... } }</pre>	outer객체를 생성해야만 사용할 수 있는 inner 중첩 클래스
	정적 멤버 클래스	<pre>class outer{ static class inner{ ... } }</pre>	outer 클래스로 바로 접근할 수 있는 inner 중첩 클래스
로컬 클래스		<pre>class outer{ void method(){ class inner{ ... } } }</pre>	method()가 실행할 때만 사용할 수 있는 inner 중첩 클래스

- 클래스 생성시 바이트 코드 따로 생성



내부 클래스(중첩 클래스)

1) 내부 클래스 정의 및 선언 방법

자바에서는 하나의 클래스 안에 다른 클래스를 정의할 수 있다. 이것을 내부 클래스(inner class)라고 한다.

```
public class OuterClass {  
    //클래스의 필드와 메소드 정의  
    ...  
    private class InnerClass {  
        ...  
    }  
}
```

- 중첩 클래스: 클래스 멤버로 선언된 클래스



내부 클래스

2) 왜 내부 클래스를 사용하는가?

- 특정 멤버를 외부에서 자주 사용한다고 할 때 이것을 효율적으로 구현하기가 상당히 어렵다. → UI에서 이벤트 처리기를 구현할 때 많이 사용된다.
- 하나의 장소에서만 사용되는 클래스들을 한 곳에 모을 수 있다. → 만약 클래스가 하나의 장소에서만 필요하다면 클래스를 분리하는 것보다 클래스의 내부에 위치시키는 것이 가독력이 좋아진다.
- 보다 읽기 쉽고 유지보수가 쉬운 코드가 된다.



내부 클래스

1. 인스턴스 멤버 클래스(non-static 내부 클래스)
 - A. Inner 클래스는 Outer 클래스의 멤버이다.
 - B. 모든 멤버 변수와 메소드가 그렇듯이, Inner 클래스는 Outer의 모든 멤버를 참조할 수 있다.
 - C. 다른 클래스에서 Outer 클래스의 내부 클래스인 Inner 클래스에 접근하기 위해서 객체를 생성할 때는 먼저 외부 클래스의 객체를 생성해서 그 레퍼런스를 가지고 내부 클래스의 객체를 생성한다.
 - D. Outer 클래스의 객체가 있어야만 Inner 클래스의 객체를 만들 수 있기 때문에 Inner 클래스의 멤버들은 static이 될 수 없다. 하지만 상수(static final)는 가능하다.

인스턴스 멤버 클래스

```
A a = new A();  
A.B b = a.new B();  
b.field1 = 3;  
b.method1();
```

/외부 클래스**/**

```
class A {  
    A() { System.out.println("A 객체가 생성됨"); }  

```

/ 내부 클래스 - 인스턴스 멤버 클래스 **/**

```
    class B {  
        int field1;  
        B() { System.out.println("B 객체가 생성됨"); }  
        void method1(){  
            System.out.println("field1:" + field1);  
        }  
    }  
}
```



인스턴스 멤버 클래스

```
class OuterClass2{
    private int num;
    public OuterClass2() {
        this.num = 100;
    }
    class InnerClass2 {
        int num = 200;
        public void method() {
            int num = 300;
            int localNum = num;
            System.out.println("localNum = "+localNum);
            int innerNum = this.num;
            System.out.println("innerNum = "+innerNum);
            int outerNum = OuterClass2.this.num;
            System.out.println("outerNum = "+outerNum);
        }
    }
}
```



인스턴스 멤버 클래스

```
public class OuterClassTest2 {  
    public static void main(String args[]) {  
        OuterClass2.InnerClass2 inner  
                                = new OuterClass2().new InnerClass2();  
        inner.method();  
    }  
}
```



내부 클래스

2. 정적 멤버 클래스(static 내부 클래스)

A. Inner 클래스는 Outer의 static 멤버를 참조할 수 있다.
하지만 일반 멤버 변수는 참조할 수 없다.

B. Inner 클래스는 Outer 클래스의 멤버이면서 동시에 static으로 선언되었으므로 Outer 클래스 외부에서 Outer의 객체 없이도 Inner의 객체를 만들 수 있다

정적 멤버 클래스

```
A.C c = new A.C();  
c.field1 = 3;  
c.method1();  
A.C.field2 = 3;  
A.C.method2();
```

/외부 클래스**/**

```
class A {  
    A() { System.out.println("A 객체가 생성됨"); }
```

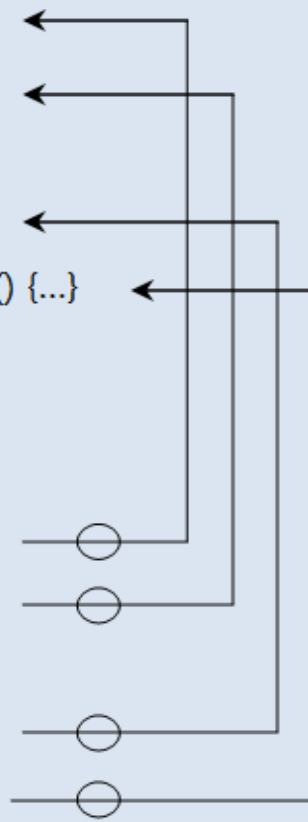
/내부 클래스 - 정적 멤버 클래스**/**

```
    static class C {  
        int field1;  
        static int field2;  
        C() { System.out.println("C 객체가 생성됨"); }  
        void method1() {  
            System.out.println("field1 : " + field1); }  
        static void method2(){  
            System.out.println("field2 : " + field2); }  
    }  
}
```

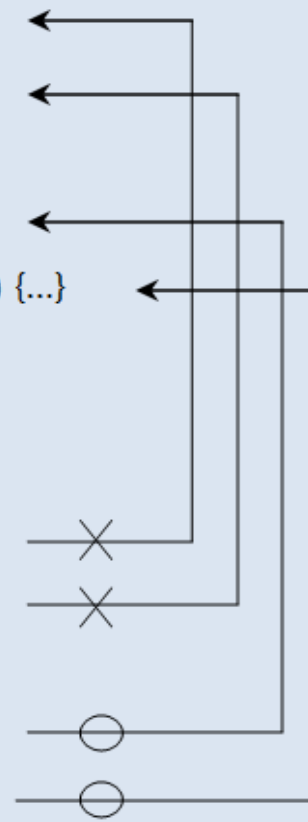
중첩 클래스의 접근 제한

❖ 멤버 클래스에서 사용 제한

```
class A {  
    int field1;  
    void method1() { ...}  
  
    static int field2;  
    static void method2() {...}  
  
    class B {  
        void method() {  
            field1 = 10;  
            method1();  
  
            field2 = 10;  
            method2();  
        }  
    }  
}
```



```
class A {  
    int field1;  
    void method1() { ...}  
  
    static int field2;  
    static void method2() {...}  
  
    static class C {  
        void method() {  
            field1 = 10;  
            methodA1();  
  
            fieldA2 = 10;  
            methodA2();  
        }  
    }  
}
```





내부 클래스

3. 지역 내부 클래스

인스턴스 멤버 클래스와 정적 멤버 클래스는 외부 클래스의 멤버의 위치에 선언되는 것에 비해, 지역 내부 클래스(local inner class)는 외부 클래스의 메소드 안에 선언된 내부 클래스이다.

- A. 메소드 내부에서만 사용 가능하고, 메소드의 실행이 끝나는 순간 메소드의 모든 멤버가 사라진다.
- B. 지역 내부 클래스는 접근 제한자(public, private) 및 static을 붙일 수 없다.
- C. 로컬 클래스 내부에는 인스턴스 필드와 메서드만 선언 가능하고 정적 필드와 메소드는 선언할 수 없다.



지역 내부 클래스

```
A a = new A();  
a.method();
```

```
class A {  
    A() { System.out.println("A 객체가 생성됨"); }  
    void method() {  
        /**내부 클래스 - 로컬 클래스**/  
        class D {  
            int field3;  
            D(){System.out.println("D 객체가 생성됨");}  
            void method1() {  
                System.out.println("field3 : " + field3);}  
        }  
        D d = new D();  
        d.field3 = 3;  
        d.method1();  
    }  
}
```



무명(익명) 클래스

무명(익명) 클래스(anonymous class)는 클래스 몸체는 정의되지만 이름이 없는 클래스이다.

다시말해 무명(익명) 클래스(anonymous class)는 정의부와 생성부가 하나로 묶여 있는 클래스로 선언과 동시에 객체가 생성된다.

```
new 상위클래스명(생성자 매개변수){  
    ← 클래스 정의  
};
```

상위 클래스란 일반 클래스와 추상 클래스, 인터페이스 모두 가능하다.



무명(익명) 클래스

익명 클래스는 이름이 없어 한번만 사용하며 재참조가 불가능하다. 즉 하나의 객체만을 생성하는 일회용 클래스이다.


익명 클래스의 특징은 다음과 같다

- ✓ 하나의 상위 클래스(일반 클래스, 추상 클래스, 인터페이스)로부터 상속받아 정의한다.

무명(익명) 클래스

- 이름이 있는 클래스의 경우

```
class TV implements RemoteControl {  
    ...  
}  
class Radio implements RemoteControl {  
    ...  
}  
  
RemoteControl obj = new TV();  
                obj = new Radio();
```

The diagram consists of two red dashed arrows. The first arrow originates from the **RemoteControl** text in the first class declaration and points to the RemoteControl text in the object creation line. The second arrow originates from the **RemoteControl** text in the second class declaration and also points to the RemoteControl text in the object creation line.



무명(익명) 클래스

- 무명(익명) 클래스의 경우

```
RemoteControl obj = new RemoteControl(){  
    ...  
};
```

- 명시적인 구현 클래스 작성 생략하고 바로 구현 객체를 얻는 방법
 - 이름 없는 구현 클래스 선언과 동시에 객체 생성

```
인터페이스 참조형 = new 인터페이스(){  
    // 인터페이스에 선언된 추상 메서드의 실제 메서드 선언  
    ...  
};
```

- 인터페이스의 추상 메소드들을 모두 재정의.
- 추가적으로 필드와 메소드 선언 가능하나 익명 객체 안에서만 사용



무명(익명) 클래스

• 익명(무명) 클래스 경우

// RemoteControl ac = new RemoteControl(); // 객체 생성 불가능

```
RemoteControl ac = new RemoteControl() { // 무명 클래스 정의  
    public void turnOn() {  
        System.out.println("TV 전원 켜기");  
    }  
    public void turnOff() {  
        System.out.println("TV 전원 끄기");  
    }  
    public void setVolume(int volume) { }  
};
```

```
ac.turnOn();  
ac.turnOff();
```




정적 멤버와 static

❖ 싱글톤(Singleton)

- 하나의 애플리케이션 내에서 단 하나만 생성되는 객체

❖ 싱글톤을 만드는 방법

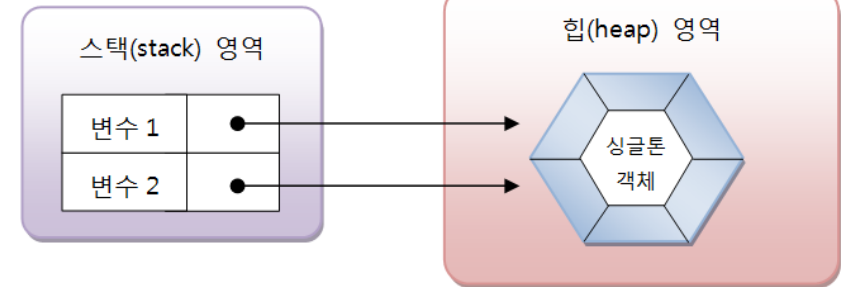
- 외부에서 new 연산자로 생성자를 호출할 수 없도록 막기
 - private 접근 제한자를 생성자 앞에 붙임
- 클래스 자신의 타입으로 정적 필드 선언
 - 자신의 객체를 생성해 초기화
 - private 접근 제한자 붙여 외부에서 필드 값 변경 불가하도록
- 외부에서 호출할 수 있는 정적 메소드인 getInstance() 선언
 - 정적 필드에서 참조하고 있는 자신의 객체 리턴

정적 멤버와 static

❖ 싱글톤 얻는 방법

```
클래스 변수1 = 클래스.getInstance();
```

```
클래스 변수2 = 클래스.getInstance();
```



```
Singleton obj1 = new Singleton();  
Singleton obj2 = new Singleton();
```

```
Singleton obj1 = Singleton.getInstance();  
Singleton obj2 = Singleton.getInstance();
```

```
if(obj1 == obj2)  
    System.out.println("같은 Singleton 객체입니다");  
else  
    System.out.println("다른 Singleton 객체입니다");
```



Thank You

