

### 2025년 상반기 K-디지털 트레이닝

# 인터페이스

상속 좋은 점 중 제일 좋은거
다형성 => 업캐스팅으로 참조 유연성
한 타입으로 여러 타입을 참조할 수 있고
다양하게 활용가능해짐

# [KB] IT's Your Life

자바의 단점 다중 상속 안돼 상속은 하나만 가능.

타파점 => 인터페이스 활용



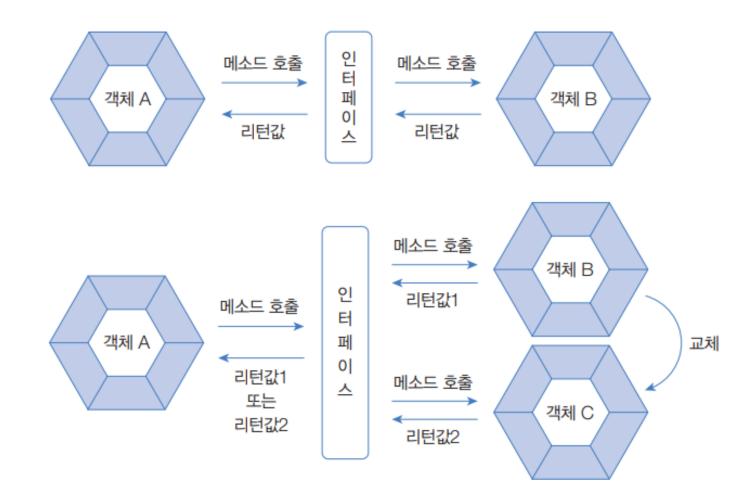
### 인터페이스 역할

### 🗸 인터페이스

- 두 객체를 연결하는 역할
- 다형성 구현에 주된 기술

메서드의 형태를 규정하는 메서드의 형태들을 규격화하면 좋은 점.

다른 데에서 만든 애들이랑도 인푸 아웃풋 메서드 형식이 같으니 좋아



### 🧿 인터페이스 선언

- 인터페이스 선언은 class 키워드 대신 interface 키워드를 사용
- 접근 제한자로는 클래스와 마찬가지로 같은 패키지 내에서만 사용 가능한 default, 패키지와 상관없이 사용하는 public을 붙일 수 있음

```
interface 인터페이스명 { ··· } //default 접근 제한 public interface 인터페이스명 { ··· } //public 접근 제한
```

```
public interface 인터페이스명 {

//public 상수 필드
//public 추상 메소드
//public 디폴트 메소드
//public 정적 메소드
//private 메소드
//private 정적 메소드
}
```

인터페이스는 거의 필드가 public으로 규정된다.

목적 자체가 그러니까

인터페이스에서 필드 범위의 디폴트는 public이다.

# RemoteControl.java

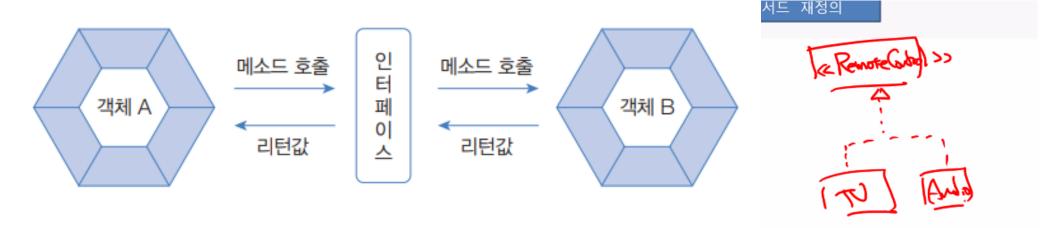
```
public interface RemoteControl {
    //public 추상 메소드
    public void turnOn();
}
```

### 2 인터페이스와 구현 클래스 선언

### 💟 구현 클래스 선언

○ 인터페이스에 정의된 추상 메소드에 대한 실행 내용이 구현

uml에서 인터페이스는 사각형 안에 <<인터페이스명>> 로 표현 실선은 상속 구현은 점선.



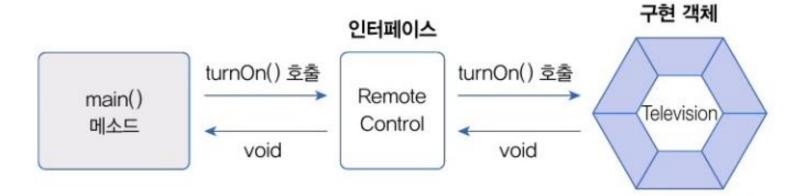
- 객체 B
  - 인터페이스에 선언된 추상 메서드와 동일한 선언부를 가진(재정의된) 메소드를 가지고 있어야 \_
- 객체 A
  - 인터페이스의 추상 메서드를 호출
  - 인터페이스 구현 객체 B의 메서드 실행

### 2 인터페이스와 구현 클래스 선언

### 💟 구현 클래스 선언

- o implements 키워드는 해당 클래스가 인터페이스를 통해 사용할 수 있다는 표시이며,
- 인터페이스의 추상 메소드를 재정의한 메소드가 있다는 뜻

public class B implements 인터페이스명 { … }



### Television.java

```
public class Television implements RemoteControl {
    @Override
    public void turnOn() {
        System.out.println("TV를 켭니다.");
    }
}
```

### 2 인터페이스와 구현 클래스 선언

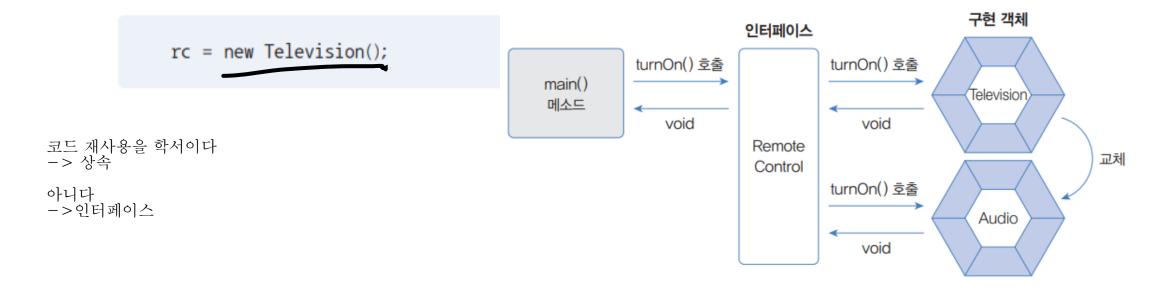
### 💟 변수 선언과 구현 객체 대입

○ <u>인터페이스는 찰조 타입에 속</u>하므로 인터페이스 변수에는 객체를 참조하고 있지 않다는 뜻으로 null을 대입할 수 있음

인터페이스 타입으로도 객체 참조 가능!

```
RemoteControl rc;
RemoteControl rc = null;
```

○ 인터페이스를 통해 구현 객체를 사용하려면, 인터페이스 변수에 구현 객체의 번지를 대입해야 함



### RemoteControlExample.java

```
package ch08.sec02;
public class RemoteControlExample {
 public static void main(String[] args) {
   RemoteControl rc;
   //rc 변수에 Television 객체를 대입
   rc = new Television();
   rc.turnOn();
   //rc 변수에 Audio 객체를 대입(교체시킴)
   rc = new Audio();
   rc.turnOn();
```

```
TV를 켭니다.
Audio를 켭니다.
```

### 상수 필드

○ 인터페이스는 public static final 특성을 갖는 불변의 상수 필드를 멤버로 가질 수 있음

[ public static final ] 타입 상수명 = 값;

- 인터페이스에 선언된 필드는 모두 public static final 특성
- public static final 생략 가능
- 상수명은 대문자로 작성하되, 서로 다른 단어로 구성되어 있을 경우에는 언더바(\_)로 연결

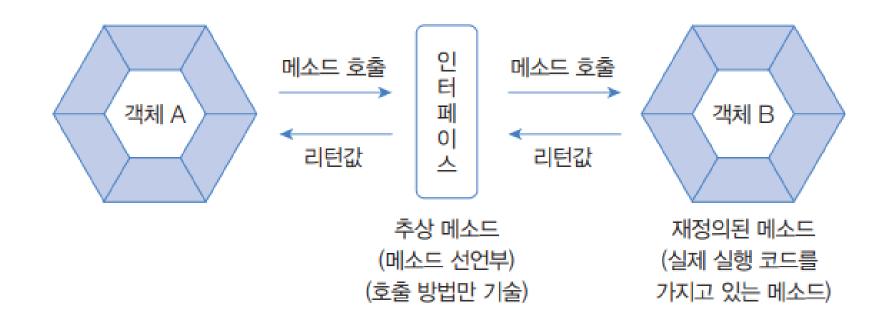
## RemoteControl.java

```
public interface RemoteControl {
  int MAX_VOLUME = 10;
  int MIN_VOLUME = 0;
} 상수 선언
}
```

### RemoteControlExample.java

```
public class RemoteControlExample {
   public static void main(String[] args) {
      System.out.println("리모콘 최대 볼륨: " + RemoteControl.MAX_VOLUME);
      System.out.println("리모콘 최저 볼륨: " + RemoteControl.MIN_VOLUME);
   }
}
```

- 리턴 타입, 메소드명, 매개변수만 기술되고 중괄호 { }를 붙이지 않는 메소드
- o public abstract를 생략하더라도 컴파일 과정에서 자동으로 붙음
- 추상 메소드는 객체 A가 인터페이스를 통해 어떻게 메소드를 호출할 수 있는지 방법을 알려주는 역할



## RemoteControl.java

```
package ch08.sec04;
public interface RemoteControl {
 //상수 필드
                         상수 선언
 int MAX_VOLUME = 10;
 int MIN_VOLUME = 0;
 //추상 메소드
 void turnOn();
 void turnOff();
 void setVolume(int volume);
```



## Television.java

```
package ch08.sec04;
                                                    //setVolume() 추상 메소드 오버라이딩
public class Television implements RemoteControl {
                                                    @Override
 //필드
                                                    public void setVolume(int volume) {
 private int volume;
                                                      if(volume>RemoteControl.MAX_VOLUME) {
                                                       this.volume = RemoteControl.MAX VOLUME;
 //turn0n() 추상 메소드 오버라이딩
                                                      } else if(volume<RemoteControl.MIN_VOLUME) {</pre>
 @Override
                                                       this.volume = RemoteControl.MIN VOLUME;
 public void turnOn() {
                                                      } else {
   System.out.println("TV를 켭니다.");
                                                       this.volume = volume;
                                                      System.out.println(
 //turn0ff() 추상 메소드 오버라이딩
                                                                 "현재 TV 볼륨: " + this.volume);
 @Override
 public void turnOff() {
   System.out.println("TV를 끕니다.");
```

# Audio.java

```
package ch08.sec04;
                                                    //setVolume() 추상 메소드 오버라이딩
public class Audio implements RemoteControl {
                                                    @Override
 //필드
                                                    public void setVolume(int volume) {
 private int volume;
                                                      if(volume>RemoteControl.MAX_VOLUME) {
                                                       this.volume = RemoteControl.MAX VOLUME;
 //turn0n() 추상 메소드 오버라이딩
                                                      } else if(volume<RemoteControl.MIN_VOLUME) {</pre>
 @Override
                                                       this.volume = RemoteControl.MIN VOLUME;
 public void turnOn() {
                                                      } else {
   System.out.println("Audio를 켭니다.");
                                                       this.volume = volume;
                                                      System.out.println("현재 Audio 볼륨: " + volume);
 //turn0ff() 추상 메소드 오버라이딩
 @Override
 public void turnOff() {
   System.out.println("Audio를 끕니다.");
```

# RemoteControlExample.java

```
package ch08.sec04;
public class RemoteControlExample {
 public static void main(String[] args) {
  //인터페이스 변수 선언
   RemoteControl rc;
   //Television 객체를 생성하고 인터페이스 변수에 대입
   rc = new Television();
   rc.turnOn();
   rc.setVolume(5);
   rc.turnOff();
   //Audio 객체를 생성하고 인터페이스 변수에 대입
   rc = new Audio();
   rc.turnOn();
                                     TV를 켭니다.
   rc.setVolume(5);
                                     현재 TV 볼륨: 5
   rc.turnOff();
                                     TV를 끕니다.
                                    Audio를 켭니다.
                                     현재 Audio 볼륨: 5
                                     Audio를 끕니다.
```

▽ 디폴트 메소드

일반적으로 인터페이스는 추상 타입이다. 하지만 디폴트 메소드는 구현되어있다. 이는 후에 자바 8에서 추가됨.

원래 인터페이스는 공통 코드, 재사용 가능 코드가 없다. 디폴트 메서드는 그걸 가능하게 해준다

- <u>인터페이스에는 완전한 실행 코드를 가진 디폴트 메소드를 선언할 수 있음</u>
- 추상 메소드는 실행부(중괄호 { })가 없지만 디폴트 메소드는 실행부 있음
- o default 키워드가 리턴 타입 앞에 붙음

[public] default 리턴타입 메소드명(매개변수, …) { … } 추상메소드가 아니라는 키워드

○ 디폴트 메소드의 실행부에는 상수 필드를 읽거나 추상 메소드를 호출하는 코드를 작성할 수 있음



## RemoteControl.java

```
package ch08.sec05;
public interface RemoteControl {
 //상수 필드
 int MAX_VOLUME = 10;
 int MIN VOLUME = 0;
 //추상 메소드
 void turnOn();
 void turnOff();
 void setVolume(int volume);
                                                 디폴트 메소드의 전제 조건.
 //디폴트 인스턴스 메소드
                                                 this키워드 사용 금지
 default void setMute(boolean mute) {
   if(mute) {
    System.out.println("무음 처리합니다.");
    //추상 메소드 호출하면서 상수 필드 사용
    setVolume(MIN_VOLUME);
  } else {
    System.out.println("무음 해제합니다.");
```

## RemoteControl.java

```
package ch08.sec05;
public class Audio implements RemoteControl {
 ... // 기존 코드
 //필드
 private int memoryVolume;
 //디폴트 메소드 재정의
 @Override
 public void setMute(boolean mute) {
   if(mute) {
    this.memoryVolume = this.volume;
    System.out.println("무음 처리합니다.");
    setVolume(RemoteControl.MIN_VOLUME);
   } else {
    System.out.println("무음 해제합니다.");
    setVolume(this.memoryVolume);
```

디폴트 메소드는 그냥 사용도 가능하지만 일반 함수처럼 오버라이딩도 가능

## RemoteControlExample.java

```
package ch08.sec05;
                                                    rc = new Audio(); // setMute 재정의 코드
public class RemoteControlExample {
                                                   rc.turnOn();
 public static void main(String[] args) {
                                                   rc.setVolume(5);
   //인터페이스 변수 선언
   RemoteControl rc;
                                                   //디폴트 메소드 호출
                                                    rc.setMute(true);
                                                                      audio는 오버라이딩 햇으니
   rc = new Television(); // 기존 코드
                                                    rc.setMute(false);
                                                                      재정의된 내용으로
   rc.turnOn();
   rc.setVolume(5);
   //디폴트 메소드 호출
                                       TV를 켭니다.
   rc.setMute(true);
                                        현재 TV 볼륨: 5
   rc.setMute(false);
                     오버라이드 안했으니
인터페이스 내용으로
                                       무음 처리합니다.
                                        현재 TV 볼륨: 0
   System.out.println();
                                       무음 해제합니다.
                                       Audio를 켭니다.
```

현재 Audio 볼륨: 5 무음 처리합니다. 현재 Audio 볼륨: 0 무음 해제합니다. 현재 Audio 볼륨: 5

### 💟 정적 메소드

- 구현 객체가 없어도 인터페이스만으로 호출할 수 있음
- 선언 시 public을 생략하더라도 자동으로 컴파일 과정에서 붙음

```
[public | private] static 리턴타입 메소드명(매개변수, …) { … }
```

○ 정적 실행부를 작성할 때 상수 필드를 제외한 추상 메소드, 디폴트 메소드, private 메소드 등을 호출할 수 없음

사수와 또다른 static멤버만 사용 가능. 당연. 인스턴스 관련 멤버는 사용 안되죠

## RemoteControl.java

```
package ch08.sec06;
public interface RemoteControl {
 //상수 필드
 int MAX_VOLUME = 10;
 int MIN_VOLUME = 0;
 //추상 메소드
 void turnOn();
 void turnOff();
 void setVolume(int volume);
 //디폴트 메소드
 default void setMute(boolean mute) {
   //이전 예제와 동일한 코드이므로 생략
 //정적 메소드
 static void changeBattery() {
   System.out.println("리모콘 건전지를 교환합니다.");
```

## RemoteControlExample.java

```
rc = new Audio(); // setMute 재정의 코드
package ch08.sec06;
                                                    rc.turnOn();
public class RemoteControlExample {
                                                    rc.setVolume(5);
 public static void main(String[] args) {
   //인터페이스 변수 선언
                                                    //디폴트 메소드 호출
                                                    rc.setMute(true);
   RemoteControl rc;
                                                    rc.setMute(false);
   rc = new Television(); // 기존 코드
   rc.turnOn();
                                                    System.out.println();
   rc.setVolume(5);
                                                    //정적 메소드 호출
                                                    RemoteControl.changeBattery();
   //디퐄트 메소드 호출
   rc.setMute(true);
   rc.setMute(false);
   System.out.println();
                                                            Audio를 켭니다.
                                                            현재 Audio 볼륨: 5
```

Audio를 켭니다.현재 Audio 볼륨: 5무음 처리합니다.현재 Audio 볼륨: 0무음 해제합니다.현재 Audio 볼륨: 5

### 7 private 메소드

### 🗸 private 메소드

- 인터페이스의 상수 필드, 추상 메소드, 디폴트 메소드, 정적 메소드는 모두 public 접근 제한을 가짐 public을 생략하더라도 항상 외부에서 접근이 가능
- 인터페이스에 외부에서 접근할 수 없는 private 메소드 선언도 가능

구분	설명
private 메소드	구현 객체가 필요한 메소드
private 정적 메소드	구현 객체가 필요 없는 메소드

- o private 메소<u>드는 디쏠트 메소드 안에서만 </u>오술이 가능
- o private 정적 메소드는 정적 메소드 안에서도 호줄이 가능

# Service.java

```
package ch08.sec07;
                                                   //정적 메소드
public interface Service {
                                                    static void staticMethod1() {
 //디폴트 메소드
                                                     System.out.println("staticMethod1 종속 코드");
 default void defaultMethod1() {
                                                     staticCommon();
   System.out.println("defaultMethod1 종속 코드");
   defaultCommon();
                                                    static void staticMethod2() {
                                                     System.out.println("staticMethod2 종속 코드");
                                                     staticCommon();
 default void defaultMethod2() {
   System.out.println("defaultMethod2 종속 코드");
   defaultCommon();
                                                   //private 정적 메소드
                                                   private static void staticCommon() {
                                                     System.out.println("staticMethod 중복 코드C");
 //private 메소드
                                                     System.out.println("staticMethod 중복 코드D");
 private void defaultCommon() {
   System.out.println("defaultMethod 중복 코드A");
   System.out.println("defaultMethod 중복 코드B");
```

# 7 private 메소드

# ServiceImpl.java

```
package ch08.sec07;
public class ServiceImpl implements Service {
}
```

## ServiceExample.java

```
package ch08.sec07;
public class ServiceExample {
 public static void main(String[] args) {
   //인터페이스 변수 선언과 구현 객체 대입
   Service service = new ServiceImpl();
   //디폴트 메소드 호출
   service.defaultMethod1();
   System.out.println();
   service.defaultMethod2();
   System.out.println();
   //정적 메소드 호출
   Service.staticMethod1();
   System.out.println();
   Service.staticMethod2();
   System.out.println();
```

defaultMethod1 종속 코드 defaultMethod 중복 코드A defaultMethod 중복 코드B

defaultMethod2 종속 코드 defaultMethod 중복 코드A defaultMethod 중복 코드B

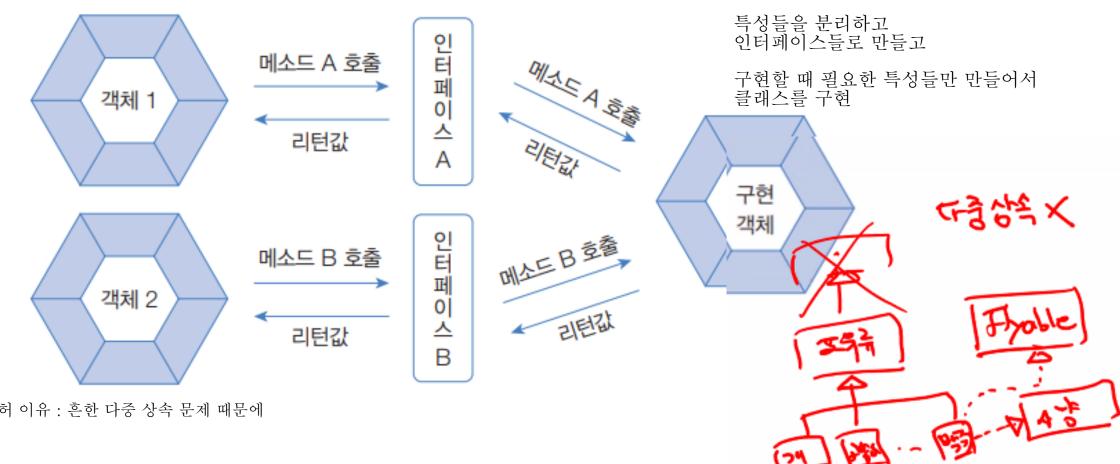
staticMethod1 종속 코드 staticMethod 중복 코드C staticMethod 중복 코드D

staticMethod2 종속 코드 staticMethod 중복 코드C staticMethod 중복 코드D

# 다중 인터페이스

다중 상속은 앋놰 다중 구현은 가능!

구현 객체는 여러 개의 인터페이스를 통해 구현 객체를 사용할 수 있음



다중 상속 불허 이유 : 흔한 다중 상속 문제 때문에

### ♡ 다중 인터페이스

○ 구현 클래스는 인터페이스 A와 인터페이스 B를 implements 뒤에 쉼표로 구분해서 작성해, 모든 인터페이스 가 가진 추상 메소드를 재정의

```
public class 구현클래스명 implements 인터페이스A, 인터페이스B {
//모든 추상 메소드 재정의
}
```

```
인터페이스A 변수 = new 구현클래스명(…);
인터페이스B 변수 = new 구현클래스명(…);
```

### 다중 인터페이스 구현

## RemoteControl.java

```
package ch08.sec08;

public interface RemoteControl {
    //추상 메소드
    void turnOn();
    void turnOff();
}

O터페이스면 관례

O터페이스는 특성을 많이 표현하므로
    접미사 able을 붙임.
    접두사로 I 붙임
```

### Searchable.java

```
package ch08.sec08;

public interface Searchable {
   //추상 메소드
   void search(String url);
}
```

## SmartTelevision.java

```
package ch08.sec08;
                                                                     ISP 원칙
public class SmartTelevision implements RemoteControl, Searchable {
                                                                     기능 단위로 쪼개라.
 //turn0n() 추상 메소드 오버라이딩
                                                                     interface seprate policy
 @Override
 public void turnOn() {
   System.out.println("TV를 켭니다.");
                                         RemoteControl 인터페이스구현
 //turnoff() 추상 메소드 오버라이딩
 @Override
 public void turnOff() {
   System.out.println("TV를 끕니다.");
                                                       single responsible policy
                                                       한 인스턴스, 클래스, 메소드는 하나의 일만 한다
 //search() 추상 메소드 오버라이딩
 @Override
 public void search(String url) {
                                             Searchable 인터페이스구현
   System.out.println(url + "을 검색합니다.");
```

## MultiInterfaceImplExample.java

```
package ch08.sec08;
public class MultiInterfaceImplExample {
 public static void main(String[] args) {
   //RemoteControl 인터페이스 변수 선언 및 구현 객체 대입
   RemoteControl rc = new SmartTelevision();
   //RemoteControl 인터페이스에 선언된 추상 메소드만 호출 가능
   rc.turnOn();
   rc.turnOff();
   //Searchable 인터페이스 변수 선언 및 구현 객체 대입
   Searchable searchable = new SmartTelevision();
   //Searchable 인터페이스에 선언된 추상 메소드만 호출 가능
   searchable.search("https://www.youtube.com");
```

```
TV를 켭니다.
TV를 끕니다.
https://www.youtube.com을 검색합니다.
```

### ♡ 인터페이스 상속

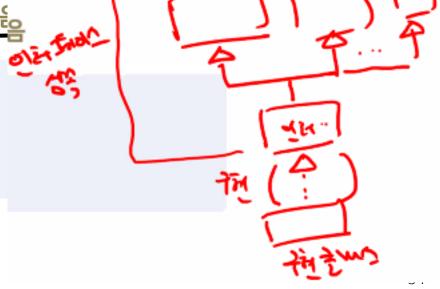
- **/**
- 인터페이스도 다른 인터페이스를 <u>상속할 수 있음. 다중 상속을</u> 허용
- extends 키워드 뒤에 상속할 인터페이스들을 나열

인터페이스간 상속 가능 인터페이스 상속시 다중상속 가능

public interface 자식인터페이스 extends 부모인터페이스1, 부모인터페이스2 { … }

- 자식 인터페이스의 구현 클래스는 자식 인터페이스의 메소드뿐만 아니라 부모 인터페이스의 모든 추상 메소드 를 재정의
- 구현 객체는 다음과 같이 자식 및 부모 인터페이스 변수에 대입될 수 있**는**

```
자식인터페이스 변수 = new 구현클래스(…);
부모인터페이스1 변수 = new 구현클래스(…);
부모인터페이스2 변수 = new 구현클래스(…);
```



## InterfaceA.java

```
package ch08.sec09;

public interface InterfaceA {
   //추상 메소드
   void methodA(); V
}
```

## InterfaceB.java

```
package ch08.sec09;

public interface InterfaceB {
   //추상 메소드
   void methodB();
}
```

## InterfaceC.java

```
package ch08.sec09;

public interface InterfaceC extends InterfaceA, InterfaceB {
    //추상 메소드
    void methodC(); 
}
```

```
InterfaceCImpl.java
package ch08.sec09;
public class InterfaceCImpl implements InterfaceC {
 public void methodA() {
   System.out.println("InterfaceCImpl-methodA() 실행");
 public void methodB() {
   System.out.println("InterfaceCImpl-methodB() 실행");
 public void methodC() {
   System.out.println("InterfaceCImpl-methodC() 실행")
```

인터페이스 구현 클래스시 이름 컨벤션 인터페이스명+Impl

모든 클래스에 해당

상속 배정된 부모 클래스가 없는 경우

자동으로 Object라는 클래스를 상촉 받음.

=>

자바의 모든 클래스는 Object 클래스를 상속 받고 있따. 모든 클래스의 조상은 Objec 클래스다.

# ExtendsExample.java

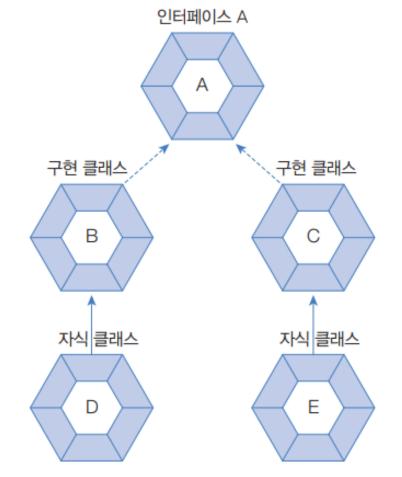
```
package ch08.sec09;
public class ExtendsExample {
 public static void main(String[] args) {
   InterfaceCImpl impl = new InterfaceCImpl();
   InterfaceA ia = impl;
   ia.methodA();
   //ia.methodB();
   System.out.println();
   InterfaceB ib = impl;
   //ib.methodA();
   ib.methodB();
                                             InterfaceCImpl-methodA() 실행
   System.out.println();
                                             InterfaceCImpl-methodB() 실행
   InterfaceC ic = impl;
   ic.methodA();
                                             InterfaceCImpl-methodA() 실행
   ic.methodB();
                                             InterfaceCImpl-methodB() 실행
   ic.methodC();
                                             InterfaceCImpl-methodC() 실행
```

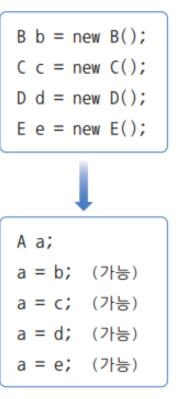
### 🗸 자동 타입 변환

○ 자동으로 타입 변환이 일어나는 것



부모 클래스가 인터페이스를 구현하고 있다면 자식 클래스도 인터페이스 타입으로 자동 타입 변환될 수 있음





## 10 <mark>타입 변환</mark>

# A.java

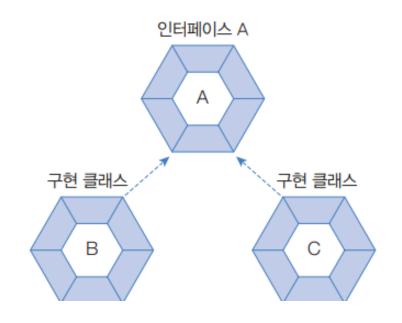
```
package ch08.sec10.exam01;
public interface A {
}
```

# **B.java**

```
package ch08.sec10.exam01;
public class B implements A {
}
```

# C.java

```
package ch08.sec10.exam01;
public class C implements A {
}
```



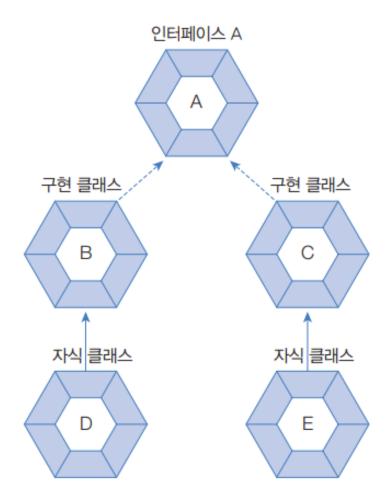
### 10 타입 변환

# D.java

```
package ch08.sec10.exam01;
public class D extends B {
}
```

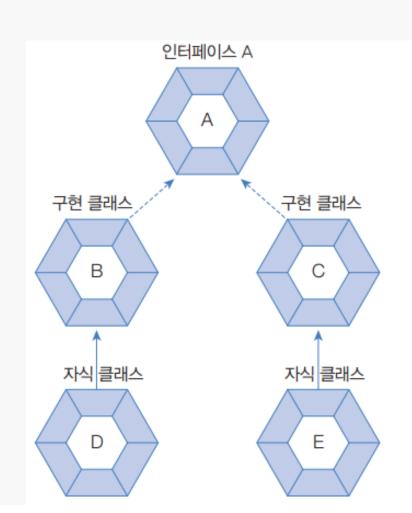
# **E**.java

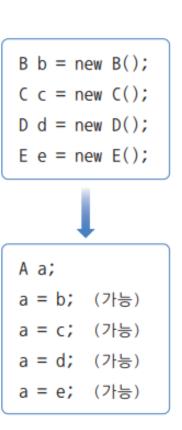
```
package ch08.sec10.exam01;
public class E extends C {
}
```



# ExtendsExample.java

```
package ch08.sec10.exam01;
public class PromotionExample {
 public static void main(String[] args) {
  //구현 객체 생성
   B b = new B();
  C c = new C();
  D d = new D();
   E e = new E();
   //인터페이스 변수 선언
   A a;
   //변수에 구현 객체 대입
   a = b; //A <-- B (자동 타입 변환)
   a = c; //A <-- C (자동 타입 변환)
   a = d; //A <-- D (자동 타입 변환)
   a = e; //A <-- E (자동 타입 변환)
```





#### ○ 강제 타입 변환

○ 캐스팅 기호를 사용해서 인터페이스 타입을 구현 클래스 타입으로 변환시키는 것



#### 💟 강제 타입 변환

○ 구현 객체가 인터페이스 타입으로 자동 변환되면, 인터페이스에 선언된 메소드만 사용 가능

```
Television

turnOn();
turnOff();
setVolume(int volume);

호출 가능
setVolume(int volume) { … }
setTime() { … }
record() { … }
```

```
RemoteControl rc = new Television();
rc.turnOn();
rc.turnOff();
rc.setVolume(5);

Television tv = (Television) rc;
tv.turnOn();
tv.turnOff();
tv.turnOff();
tv.setVolume(5);
tv.setTime();
tv.record();
```

#### ☑ 강제 타입 변환

```
interface Vehicle {
                                   Vehicle vehicle = new Bus();
 void run();
                                   vehicle.run(); //가능
                                  vehicle.checkFare(); //불가능
              구현
                                   Bus bus = (Bus) vehicle; //강제 타입 변환
class Bus implements Vehicle
 void run() { ... };
                                   bus.run(); //가능
 void checkFare() { ··· } '
                                   bus.checkFare(); //가능
```

## Vehicle.java

```
package ch08.sec10.exam02;

public interface Vehicle {
   //추상 메소드
   void run();
}
```

## Bus.java

```
package ch08.sec10.exam02;
public class Bus implements Vehicle {
 //추상 메소드 재정의
 @Override
 public void run() {
  System.out.println("버스가 달립니다.");
 //추가 메소드
 public void checkFare() {
  System.out.println("승차요금을 체크합니다.");
```

# CastingExample.java

```
package ch08.sec10.exam02;
public class CastingExample {
 public static void main(String[] args) {
   //인터페이스 변수 선언과 구현 객체 대입
   Vehicle vehicle = new Bus(); ∨
   //인터페이스를 통해서 호출
   vehicle.run();
   //vehicle.checkFare(); (x)
   //강제 타입 변환후 호출
   Bus bus = (Bus) vehicle;
   bus.run();
   bus.checkFare();
```

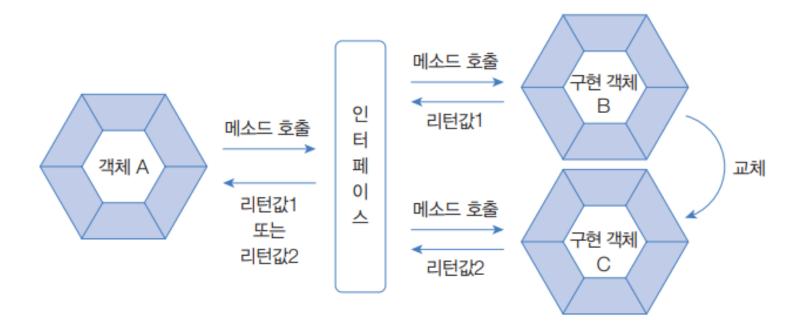
```
버스가 달립니다.
버스가 달립니다.
승차요금을 체크합니다.
```

### 11 다형성

#### 🗸 다형성

○ 사용 방법은 동일하지만 다양한 결과가 나오는 성질

인터페이스를 사용하는 또다른 이유 다형성



○ 인터페이스 역시 다형성을 구현하기 위해 재정의와 자동 타입 변환 기능을 이용



#### ☑ 필드의 다형성



```
public class Car {
   Tire tire1 = new HankookTire();
   Tire tire2 = new KumhoTire();
}
```

```
Car myCar = new Car();
myCar.tire1 = new KumhoTire();
```

# Tire.java

```
package ch08.sec11.exam01;

public interface Tire {
  //추상 메소드
  void roll();
}
```

# Tire.java

```
public class HankookTire implements Tire {
    //추상 메소드 재정의
    @Override
    public void roll() {
        System.out.println("한국 타이어가 굴러갑니다.");
    }
}
```

### KumhoTire.java

```
public class KumhoTire implements Tire {
    //추상 메소드 재정의
    @Override
    public void roll() {
        System.out.println("금호 타이어가 굴러갑니다.");
    }
}
```

# Car.java

```
public class Car {
   //필드
   Tire tire1 = new HankookTire();
   Tire tire2 = new HankookTire();

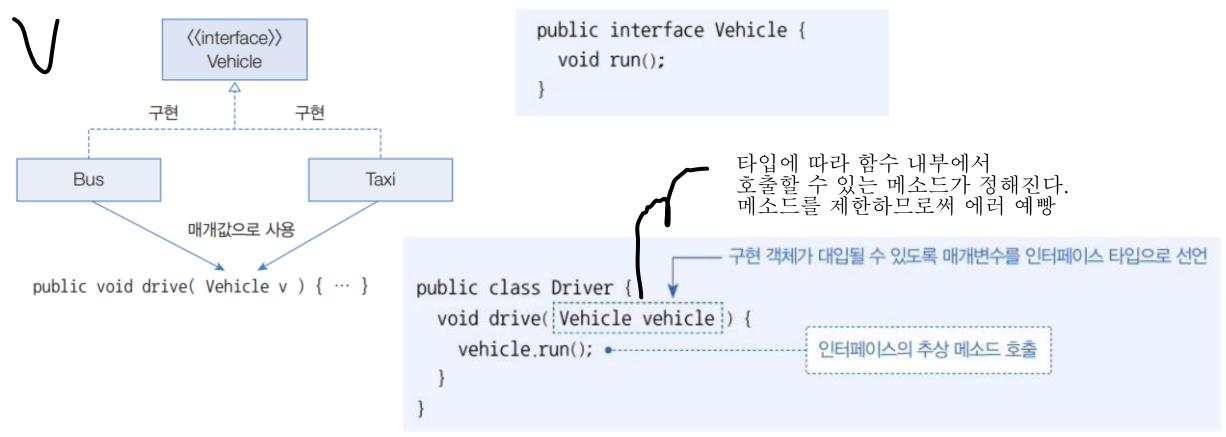
   //메소드
   void run() {
       tire1.roll();
       tire2.roll();
   }
}
```

# CarExample.java

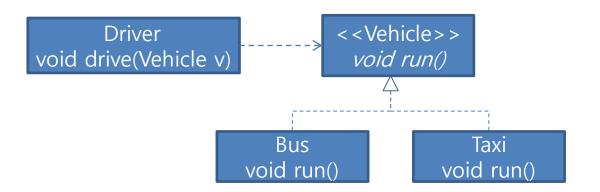
```
package ch08.sec11.exam01;
public class CarExample {
                                             리스코프 치환 워칙
 public static void main(String[] args) {
  //자동차 객체 생성
                                             언제든지 자식을 교체,치환해서 운용가능하게끔
  Car myCar = new Car();
                                            자식을 교체해도 언제든지 동작되어야 한다.
  //run() 메소드 실행
                                             =>
  myCar.run();
                                            메소드 정의시 실객체가 바뀌어도
로직 변화가 없게끔 정의해라
  //타이어 객체 교체
  myCar.tire1 = new KumhoTire();
  myCar.tire2 = new KumhoTire();
  //run() 메소드 실행(다형성: 실행 결과가 다름)
  myCar.run();
              한국 타이어가 굴러갑니다.
              한국 타이어가 굴러갑니다.
              금호 타이어가 굴러갑니다.
              금호 타이어가 굴러갑니다.
```

#### 💟 매개변수의 다형성

- 매개변수 타입을 인터페이스로 선언
- 메소드 호출 시 다양한 구현 객체를 대입할 수 있음



#### 🗸 매개변수의 다형성



```
Driver driver = new Dirver();

Bus bus = new Bus();

driver.drive( bus );

자동 타입 변환 발생

Vehicle vehicle = bus;
```

```
void drive(Vehicle vehicle) {
vehicle.run(); • 구현 객체가 재정의한 run() 메소드가 실행
}
```

# Vehicle.java

```
package ch08.sec11.exam02;

public interface Vehicle {
   //추상 메소드
   void run();
}
```

# **Bus.java**

```
public class Bus implements Vehicle {
    //추상 메소드 재정의
    @Override
    public void run() {
        System.out.println("버스가 달립니다.");
    }
}
```

## Taxi.java

```
package ch08.sec11.exam02;

public class Taxi implements Vehicle {
    //추상 메소드 재정의
    @Override
    public void run() {
       System.out.println("택시가 달립니다.");
    }
}
```

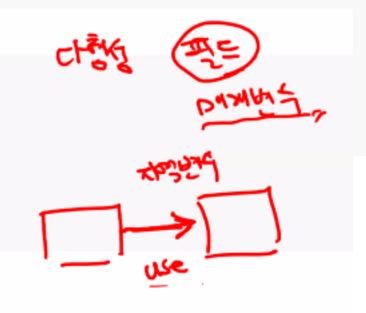
## Driver.java

```
package ch08.sec11.exam02;

public class Driver {
  void drive( Vehicle vehicle ) {
    vehicle.run();
  }
}
```

다른 클래스를 잠깐 지역변수로 사용할때

빈마름모 aggregation 다른 클래스를 필드 멤버로써 갖고 사용할때 그리고 다른 실체로 교체가능할때 한 클래스가 다른 클래스를 사용할 때 uml 표기법





마름모가 색채워져 있으면 composition 의미는 강결합. tightly coupled.

다른 클래스를 필드 멤버로 쓰지만 참조를 바꾸지 않을 때.

## DriverExample.java

```
package ch08.sec11.exam02;
public class DriverExample {
 public static void main(String[] args) {
   //Driver 객체 생성
   Driver driver = new Driver();
   //Vehicle 구현 객체 생성
   Bus bus = new Bus();
   Taxi taxi = new Taxi();
   //매개값으로 구현 객체 대입(다형성: 실행 결과가 다름)
   driver.drive(bus); // 자동 타입 변환: Bus → Verhicle
   driver.drive(taxi); // 자동 타입 변환: Taxi → Verhicle
```

#### 🧿 instanceof 연산자

○ 인터페이스에서도 객체 타입을 확인하기 위해 instanceof 연산자를 사용 가능

```
if( vehicle instanceof Bus ) {
  //vehicle에 대입된 객체가 Bus일 경우 실행
}
```

```
public void method( Vehicle vehicle) {
    if(vehicle instanceof Bus) {
        Bus bus = (Bus) vehicle;
        //bus 변수 사용
    }
}
```

## 12 객체 타입 확인

#### ♡ instanceof 연산자

○ Java 12부터는 instanceof 연산의 결과가 true일 경우
→ 우측 타입 변수를 사용할 수 있기 때문에 강제 타입 변환이 필요 없음

```
if(vehicle instanceof Bus bus) {
  //bus 변수 사용
}
```

# 12 객체 타입 확인

# Vehicle.java

```
package ch08.sec12;

public interface Vehicle {
  void run();
}
```

## Bus.java

```
public class Bus implements Vehicle {
    @Override
    public void run() {
        System.out.println("버스가 달립니다.");
    }

    public void checkFare() {
        System.out.println("승차요금을 체크합니다.");
    }
}
```

## Taxi.java

```
package ch08.sec12;

public class Taxi implements Vehicle {
    @Override
    public void run() {
        System.out.println("택시가 달립니다.");
    }
}
```

# InstanceofExample.java

```
public class InstanceofExample {
  public static void main(String[] args) {
    //구현 객체 생성
    Taxi taxi = new Taxi();
    Bus bus = new Bus();

    //ride() 메소드 호출 시 구현 객체를 매개값으로 전달
    ride(taxi);
    System.out.println();
    ride(bus);
}
```

## InstanceofExample.java

```
//인터페이스 매개변수를 갖는 메소드
public static void ride(Vehicle vehicle) {
 //방법1
 /*if(vehicle instanceof Bus) {
   Bus bus = (Bus) vehicle;
   bus.checkFare();
 }*/
 //방법2
 if(vehicle instanceof Bus bus) {
   bus.checkFare();
 vehicle.run();
```

#### sealed 인터페이스

○ Java 15부터 무분별한 자식 인터페이스 생성을 방지하기 위해 봉인된 인터페이스 사용

```
public sealed interface InterfaceA permits InterfaceB { ··· }
```

○ sealed 키워드를 사용하면 permits 키워드 뒤에 상속 가능한 자식 인터페이스를 지정. non-sealed는 봉인을 해제한다는 뜻

```
public non-sealed interface InterfaceB extends InterfaceA { ··· }
```