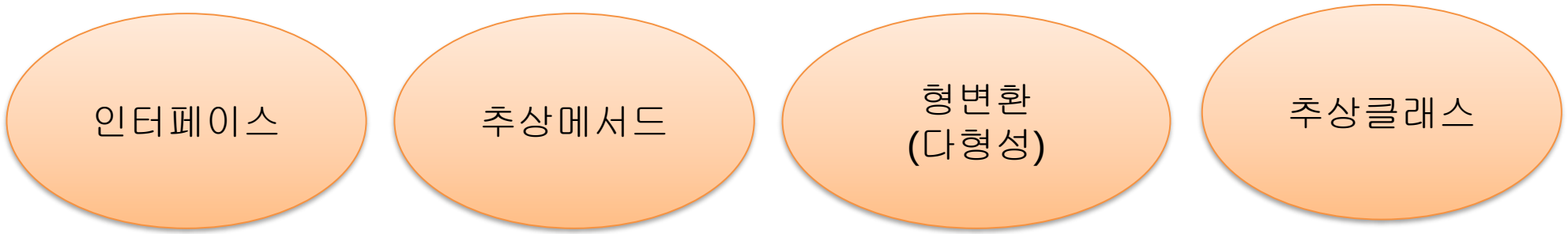
KB금융그룹 | 국민의 평생 금융파트너

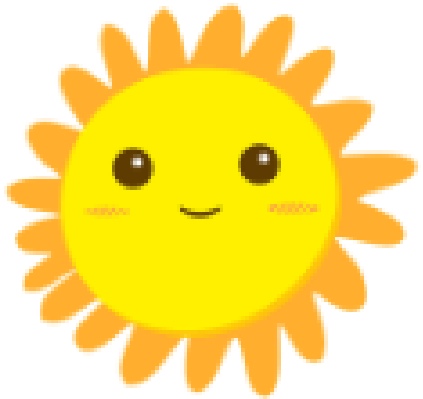
2025년 상반기 K-디지털 트레이닝

인터페이스 + 추상클래스

[KB] IT's Your Life

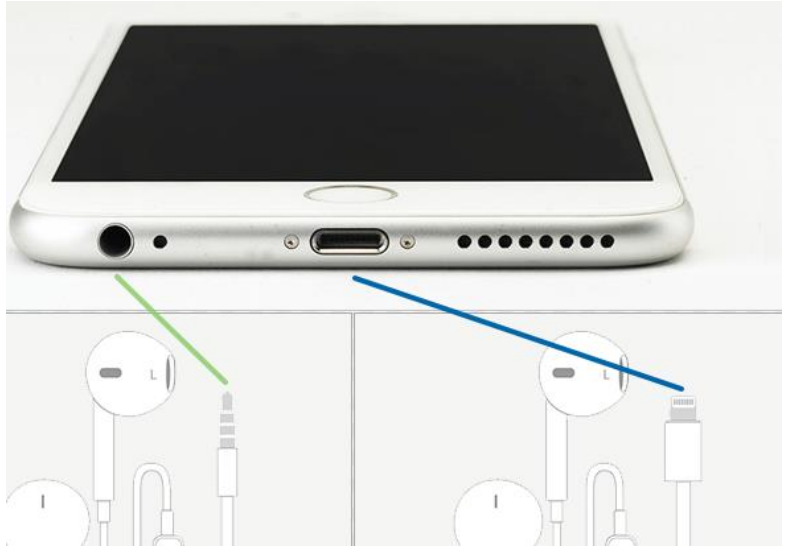


KB국민은행



인터페이스

1 인터페이스 역할



- 인터페이스 단어 의미 – 접점, 연결 지점
- It에서 인터페이스 의미 – 원하는 기능을 사용/처리하기 위한 방법(함수, 인터넷 주소 등)
- 자바에서 인터페이스 의미 – 접점이 되기 위한 필수 기능을 정의해 놓은 파일
  - 휴대폰의 이어폰이 되려면 “8핀이 되어야 한다.“, “C타입이어야 한다.”
  - 세탁기가 되려면 “빨래하는 기능이 있어야 한다.”, “건조하는 기능이 있어야 한다.”
  - 구체적으로 어떻게 기능을 처리할 지는 쓰지 않음.
  - 비어있는 메서드 선언(불완전한 메서드)

지디넷코리아

USB-C 타입, 국가표준으로

주문정 기자 | 입력 2022. 10. 23. 13:08 | 수정 2022. 10. 23. 14:10

1

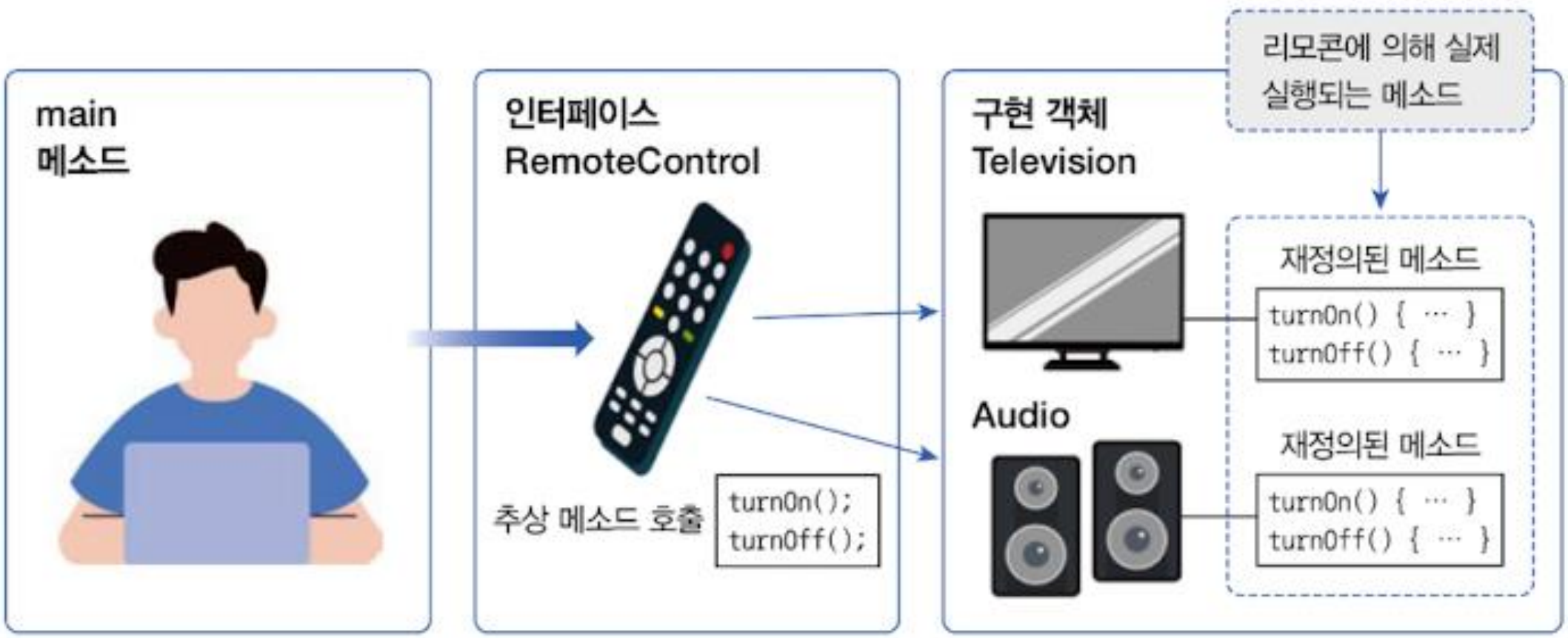
전자제품 전원·데이터접속 기준 통합..11월 말 가이드라인 발표

(지디넷코리아=주문정 기자)산업통상자원부 국가기술표준원은 전자제품 전원·데이터접속 커넥터 형상을 USB-C 타입으로 통합·호환해 산업경쟁력을 확보할 수 있도록 국가표준(KS)으로 제정한다고 23일 밝혔다.

표준안은 지난 8월 10일부터 10월 9일까지 예고 고시한 데 이어 18일 기술심의회를 통과했다. 11월 초 표준회의 등의 절차를 거쳐 이르면 다음달 중 국가표준으로 제정된다.

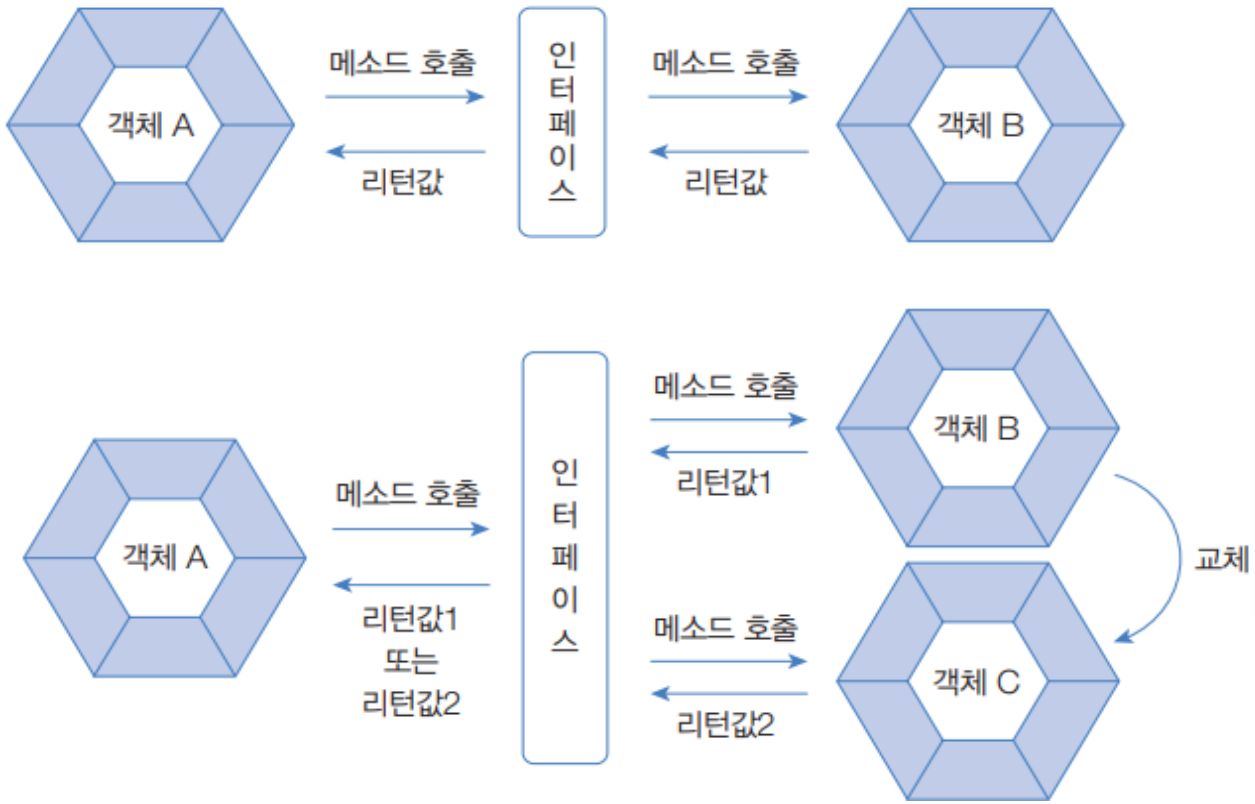
그동안 국내에서는 휴대폰·태블릿PC·휴대용 스피커 등 소형 전자제품에서 전원공급과 데이터 전송은 다양한 접속단자와 통신방식이 존재해 환경·비용문제와 소비자 사용 불편을 초래했다.

1 인터페이스 역할


main
메소드
인터페이스
RemoteControl
추상 메소드 호출
turnOn();
turnOff();
구현 객체
Television
Audio
리모콘에 의해 실제
실행되는 메소드
재정의된 메소드
turnOn() { … }
turnOff() { … }
재정의된 메소드
turnOn() { … }
turnOff() { … }

인터페이스

- 두 객체를 연결하는 역할
- 다형성 구현에 주된 기술



인터페이스 규격에 맞추어서 만든 클래스라면 어떤 클래스라도 쓸 수 있음.

대체도 가능함.

인터페이스 하나의 이름으로 여러 형태의 객체를 사용할 수 있게 되었음 —> 다형성

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
}
```

RemoteControl.java

```
package ch08.sec02;

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

추상메서드

- 구체적으로 메서드 내용을 구현하지 않은 메서드
- **추상 == 불완전** 의미
- 추상메서드를 하나라도 가진 클래스, 인터페이스는 객체 생성 불가능
- 메서드를 구체적으로 아직 구현하지 않고 구현할 기능만 써넣은 "메서드 이름"만 있는 메서드

- 차의 기능은 "시동 걸다, 멈추다" 기능이 있어야 한다라고 명시하고 싶은 경우

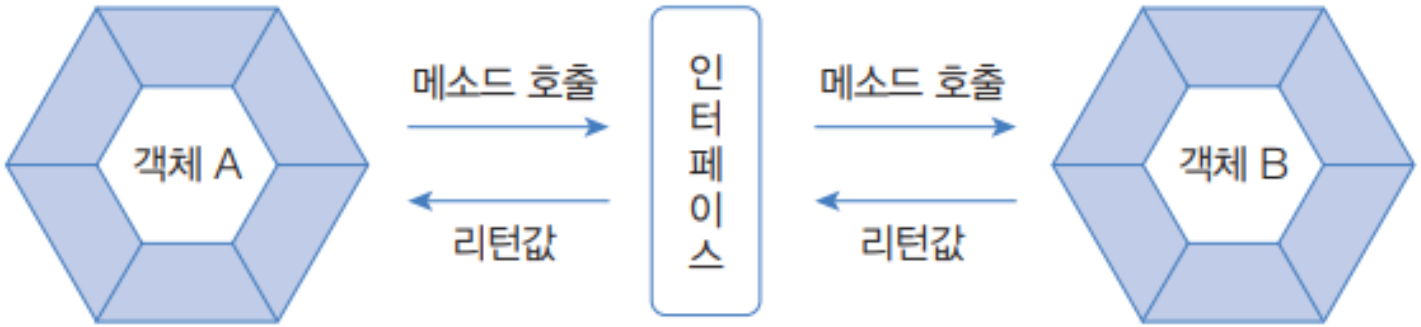
```
Car Interface {
        void run();
        void stop()
}
```

1.new를 이용한 객체 생성 불가
불완전한 메서드를 포함하고 있기 때문
RemoteControl rc = new RemoteControl(); //불가능

2.변수 타입으로는 쓸 수 있음
RemoteControl rc = null;
인터인터페이스 규격에 맞추어서 만든 클래스라면 어떤 클래스라도 쓸 수 있음.
인터페이스를 구현한 클래스 객체 할당 가능.
클래스다이어그램에서 위에서 위치하므로 부모역할을 함.
형변환 가능.

구현 클래스 선언

- 인터페이스에 정의된 추상 메소드에 대한 실행 내용이 구현



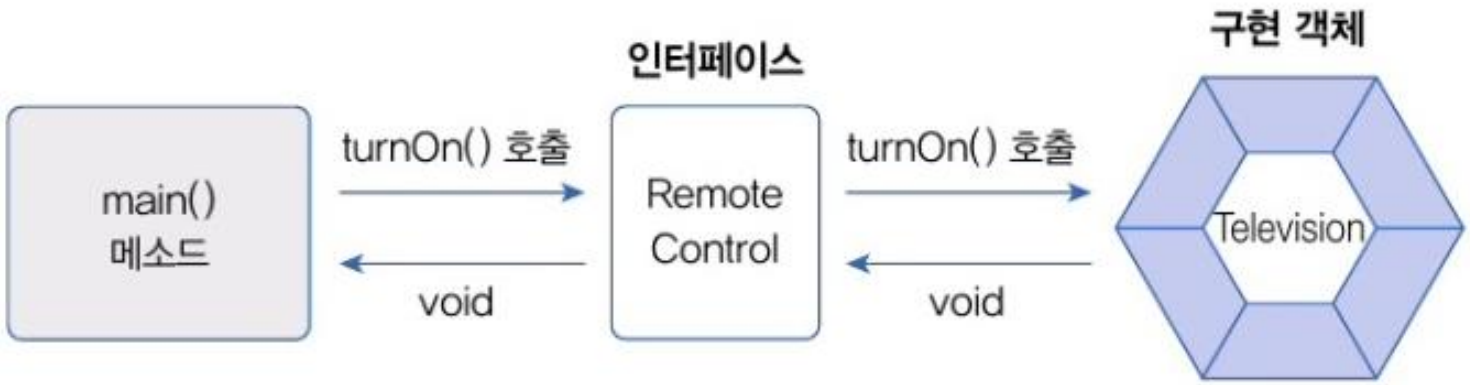
- 객체 B
  - 인터페이스에 선언된 추상 메서드와 동일한 선언부를 가진(재정의된) 메소드를 가지고 있어야 함

- 객체 A
  - 인터페이스의 추상 메서드를 호출
  - 인터페이스 구현 객체 B의 메서드 실행

구현 클래스 선언

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

```
public class B implements 인터페이스명 { … }
```



Television.java

```
package ch08.sec02;

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

인터페이스에 선언된
turnOn( ) 추상 메서드 재정의

Television.java

```
package ch08.sec02;

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

인터페이스에 선언된
turnOn( ) 추상 메서드 재정의

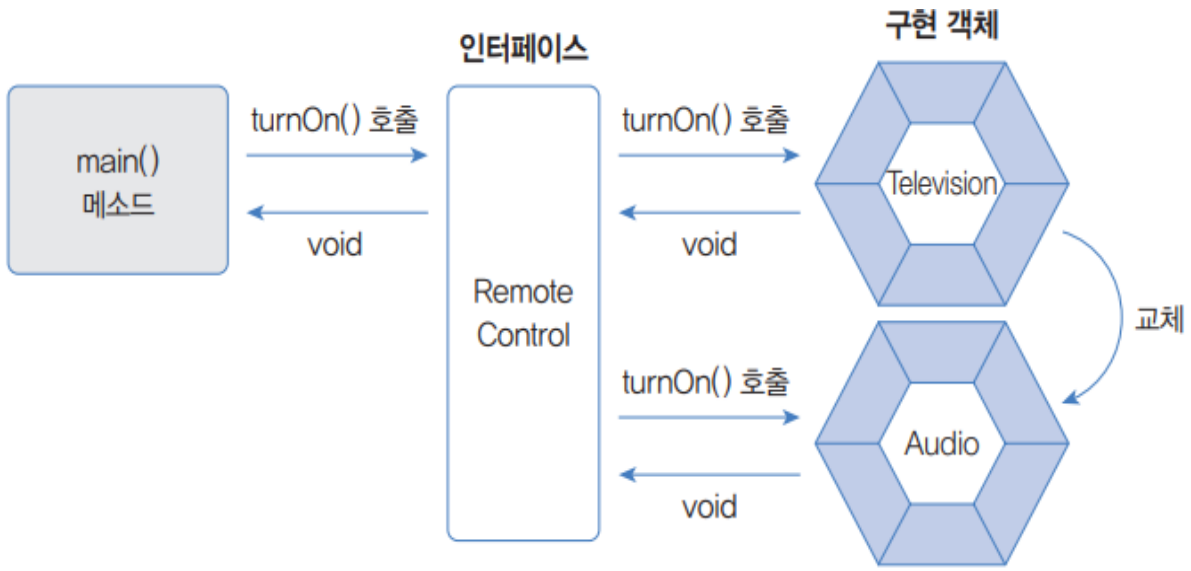
변수 선언과 구현 객체 대입

- 인터페이스는 참조 타입에 속하므로 인터페이스 변수에는 객체를 참조하고 있지 않다는 뜻으로 null을 대입할 수 있음

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

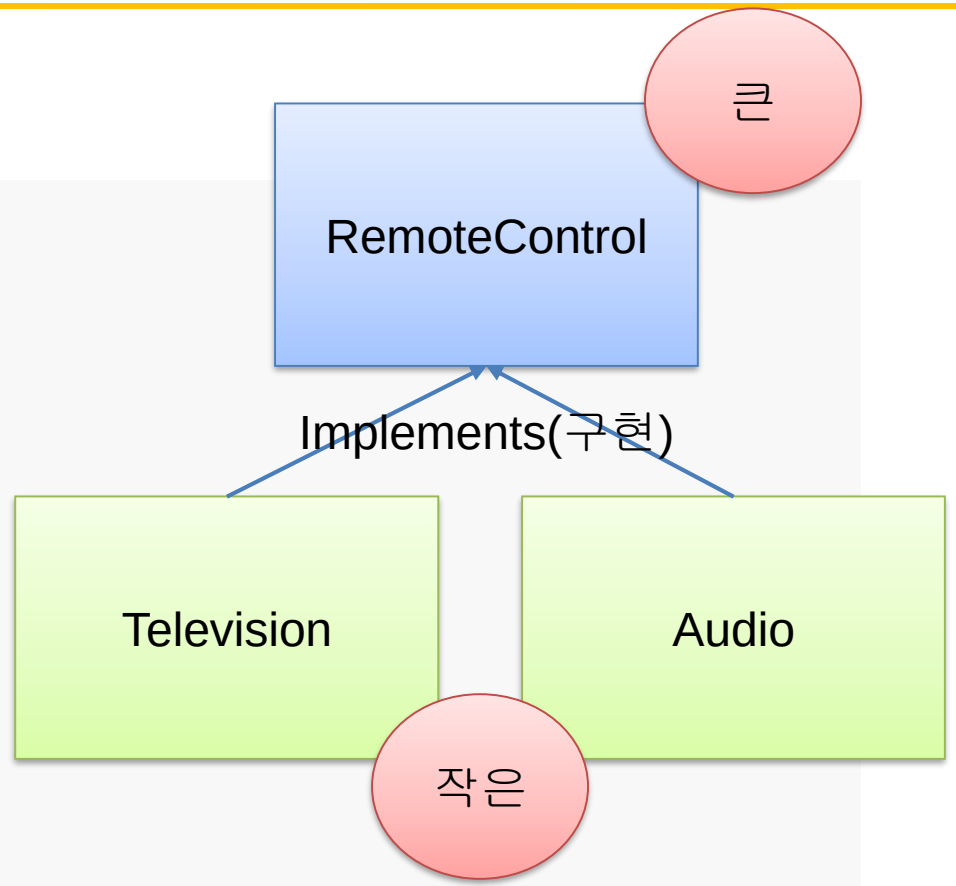
- 인터페이스를 통해 구현 객체를 사용하려면, 인터페이스 변수에 구현 객체의 번지를 대입해야 함

```
rc = new Television();
```



RemoteControlExample.java

```
package ch08.sec02;

public class RemoteControlExample {
	public static void main(String[] args) {
		RemoteControl rc;

		//rc 변수에 Television 객체를 대입
		rc = new Television(); //자동형변환
		rc.turnOn();

		//rc 변수에 Audio 객체를 대입(교체시킴)
		rc = new Audio(); //자동형변환
		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

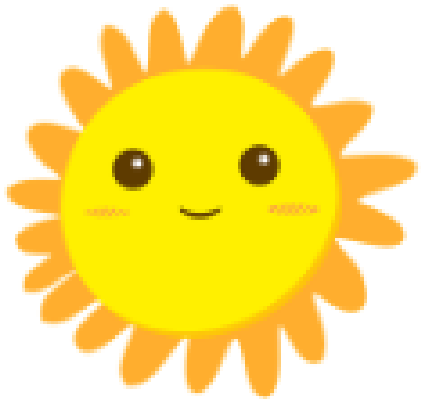
```
package ch08.sec03;

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

상수 선언

- public static final이 앞에 생략됨
- 구현된 클래스들에 공유할 목적으로 인터페이스에 상수 선언 가능

RemoteControlExample.java

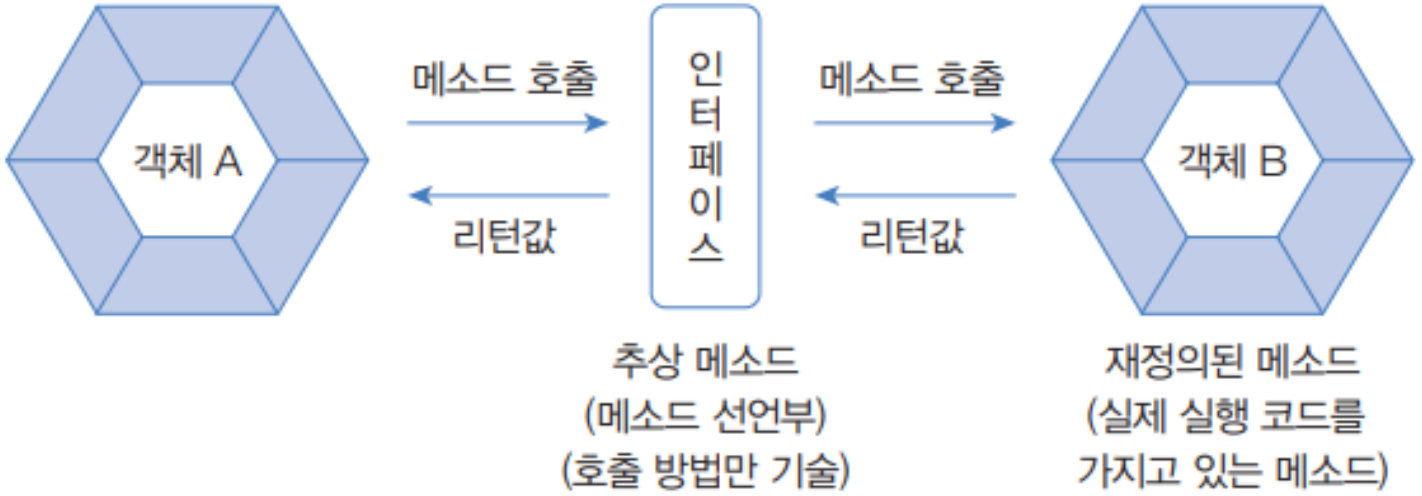
```
public class RemoteControlExample {
        public static void main(String[] args) {
                System.out.println("리모콘 최대 볼륨: " + RemoteControl.MAX_VOLUME);
                System.out.println("리모콘 최저 볼륨: " + RemoteControl.MIN_VOLUME);
        }
}
```



추상 메서드

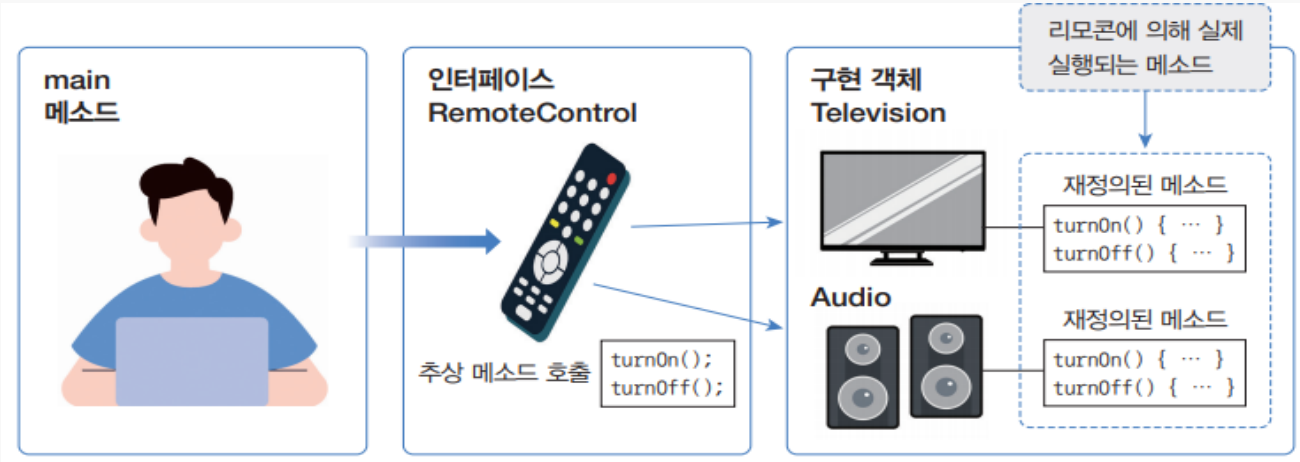
추상 메소드

- 리턴 타입, 메소드명, 매개변수만 기술되고 중괄호 { }를 붙이지 않는 메소드
- 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
        int MAX_VOLUME = 10;
        int MIN_VOLUME = 0;

        //추상 메소드
        void turnOn();
        void turnOff();
        void setVolume(int volume);
}
```



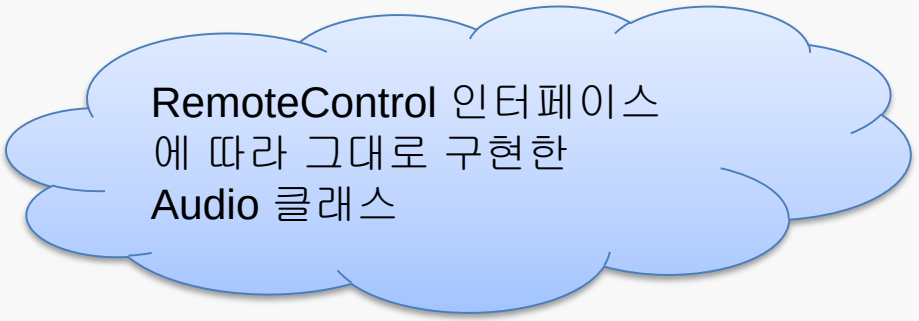
Television.java

```
package ch08.sec04;

public class Television implements RemoteControl {
		//필드
		private int volume;

		//turnOn() 추상 메소드 오버라이딩
		@Override
		public void turnOn() {
			System.out.println("TV를 켭니다.");
		}

		//turnOff() 추상 메소드 오버라이딩
		@Override
		public void turnOff() {
			System.out.println("TV를 끕니다.");
		}

		//setVolume() 추상 메소드 오버라이딩
		@Override
		public void setVolume(int volume) {
			if(volume>RemoteControl.MAX_VOLUME) {
				this.volume = RemoteControl.MAX_VOLUME;
			} else if(volume<RemoteControl.MIN_VOLUME) {
				this.volume = RemoteControl.MIN_VOLUME;
			} else {
				this.volume = volume;
			}
			System.out.println(
	"현재 TV 볼륨: " + this.volume);
		}
}
```

RemoteControl 인터페이스에 따라 그대로 구현한 Television 클래스

Audio.java

```
package ch08.sec04;

public class Audio implements RemoteControl {
		//필드
		private int volume;

		//turnOn() 추상 메소드 오버라이딩
		@Override
		public void turnOn() {
			System.out.println("Audio를 켭니다.");
		}

		//turnOff() 추상 메소드 오버라이딩
		@Override
		public void turnOff() {
			System.out.println("Audio를 끕니다.");
		}

		//setVolume() 추상 메소드 오버라이딩
		@Override
		public void setVolume(int volume) {
			if(volume>RemoteControl.MAX_VOLUME) {
				this.volume = RemoteControl.MAX_VOLUME;
			} else if(volume<RemoteControl.MIN_VOLUME) {
				this.volume = RemoteControl.MIN_VOLUME;
			} else {
				this.volume = volume;
			}
			System.out.println("현재 Audio 볼륨: " + volume);
		}
}
```



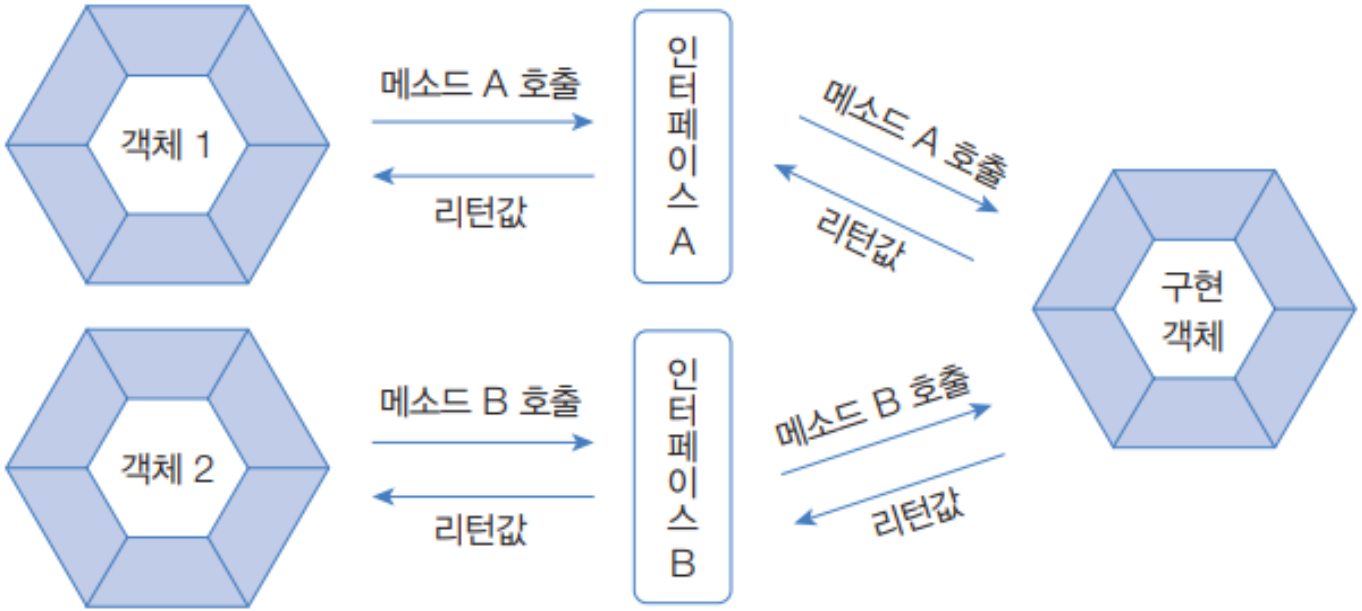
RemoteControlExample.java

```
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();
		rc.setVolume(5);
		rc.turnOff();
	}
}
```

```
TV를 켭니다.
현재 TV 볼륨: 5
TV를 끕니다.
Audio를 켭니다.
현재 Audio 볼륨: 5
Audio를 끕니다.
```

다중 인터페이스

- 구현 객체는 여러 개의 인터페이스를 통해 구현 객체를 사용할 수 있음



- 인터페이스는 다중 상속도 가능하다.
- 클래스는 여러 인터페이스를 구현할 수 있다.

다중 인터페이스

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

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

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

RemoteControl.java

```
package ch08.sec08;

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

Searchable.java

```
package ch08.sec08;

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

SmartTelevision.java

```
package ch08.sec08;

public class SmartTelevision implements RemoteControl, Searchable {
		//turnOn() 추상 메소드 오버라이딩
		@Override
		public void turnOn() {
			System.out.println("TV를 켭니다.");
		}

		//turnoff() 추상 메소드 오버라이딩
		@Override
		public void turnOff() {
			System.out.println("TV를 끕니다.");
		}

		//search() 추상 메소드 오버라이딩
		@Override
		public void search(String url) {
			System.out.println(url + "을 검색합니다.");
		}
}
```

RemoteControl 인터페이스구현

Searchable 인터페이스구현

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

인터페이스 상속

- 인터페이스도 다른 인터페이스를 상속할 수 있음. 다중 상속을 허용
- extends 키워드 뒤에 상속할 인터페이스들을 나열

```
public interface 자식인터페이스 extends 부모인터페이스1, 부모인터페이스2 { … }
```

- 자식 인터페이스의 구현 클래스는 자식 인터페이스의 메소드뿐만 아니라 부모 인터페이스의 모든 추상 메소드를 재정의
- 구현 객체는 다음과 같이 자식 및 부모 인터페이스 변수에 대입될 수 있음

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

InterfaceA.java

```
package ch08.sec09;

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

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();
}
```

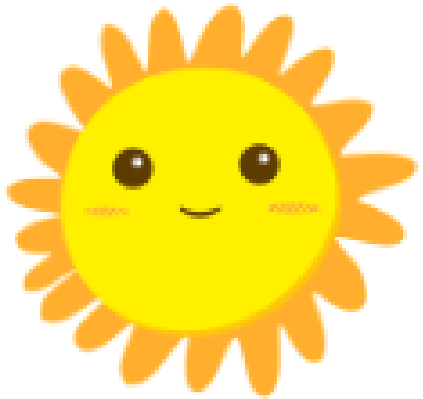
InterfaceCImpl.java

```
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");
	}
}
```

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();
		System.out.println();

		InterfaceC ic = impl;
		ic.methodA();
		ic.methodB();
		ic.methodC();
	}
}
```

```
InterfaceCImpl-methodA() 실행

InterfaceCImpl-methodB() 실행

InterfaceCImpl-methodA() 실행
InterfaceCImpl-methodB() 실행
InterfaceCImpl-methodC() 실행
```



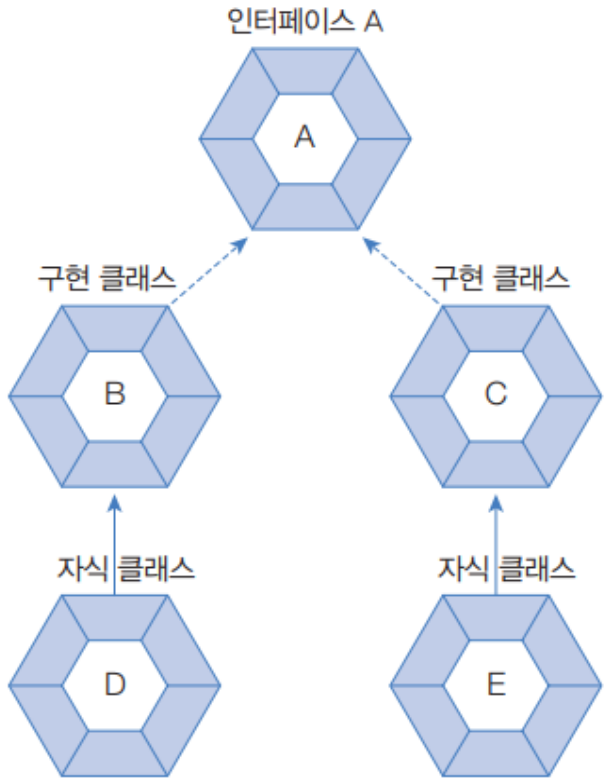
형변환(다형성)

자동 타입 변환

- 자동으로 타입 변환이 일어나는 것

```
인터페이스 변수 = 구현객체;
```
(자동 타입 변환)

- 부모 클래스가 인터페이스를 구현하고 있다면 자식 클래스도 인터페이스 타입으로 자동 타입 변환 될 수 있음




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

```
A a;
a = b;  (가능)
a = c;  (가능)
a = d;  (가능)
a = e;  (가능)
```

A.java

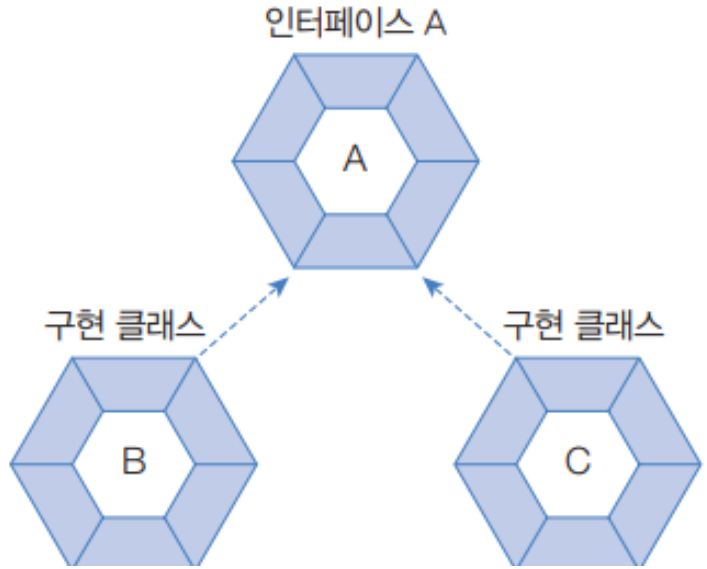
```
package ch08.sec10.exam01;

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 {
}
```

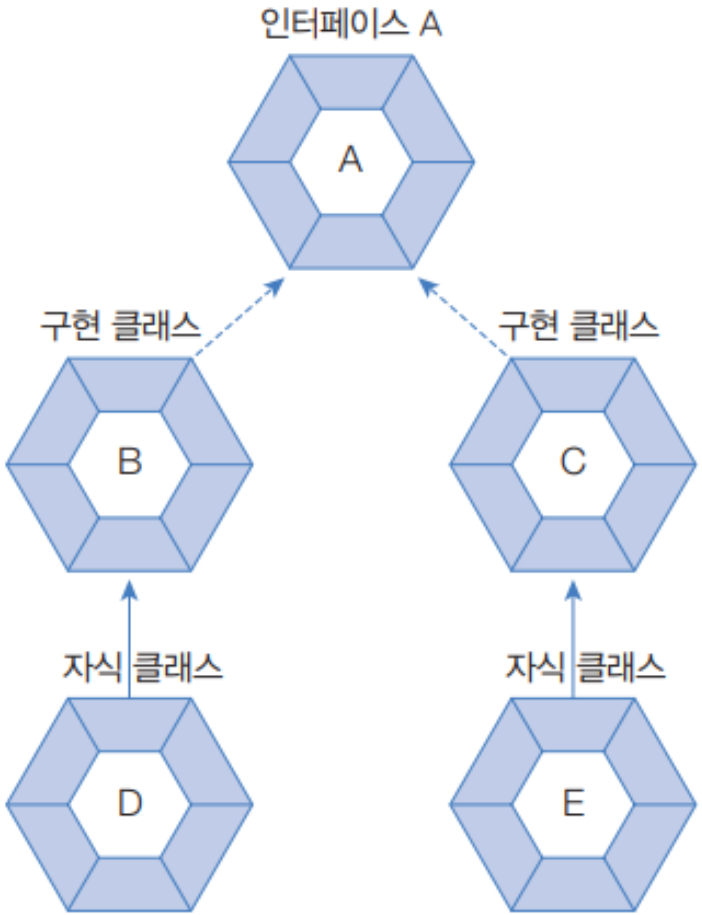


D.java

```
package ch08.sec10.exam01;

public class D extends B {
}
```

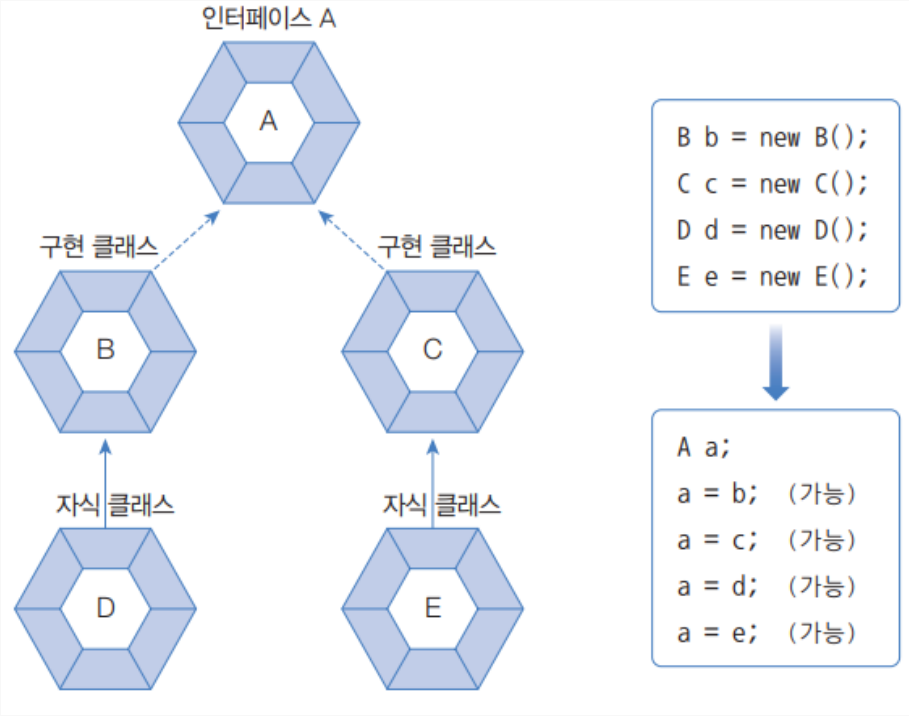
E.java

```
package ch08.sec10.exam01;

public class E extends C {
}
```



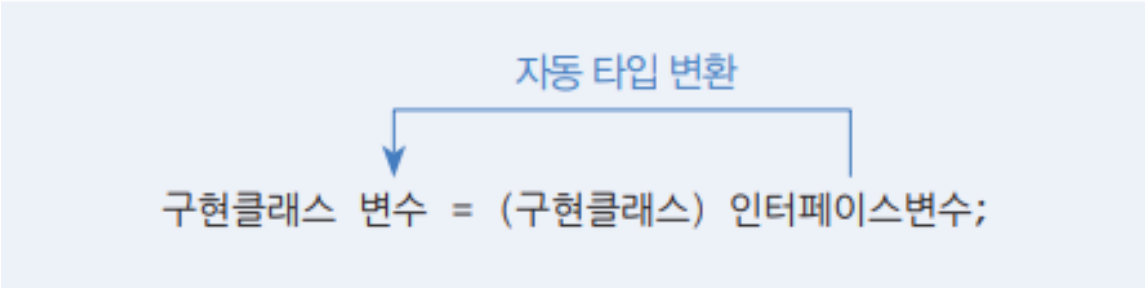
ExtendsExample.java

```
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

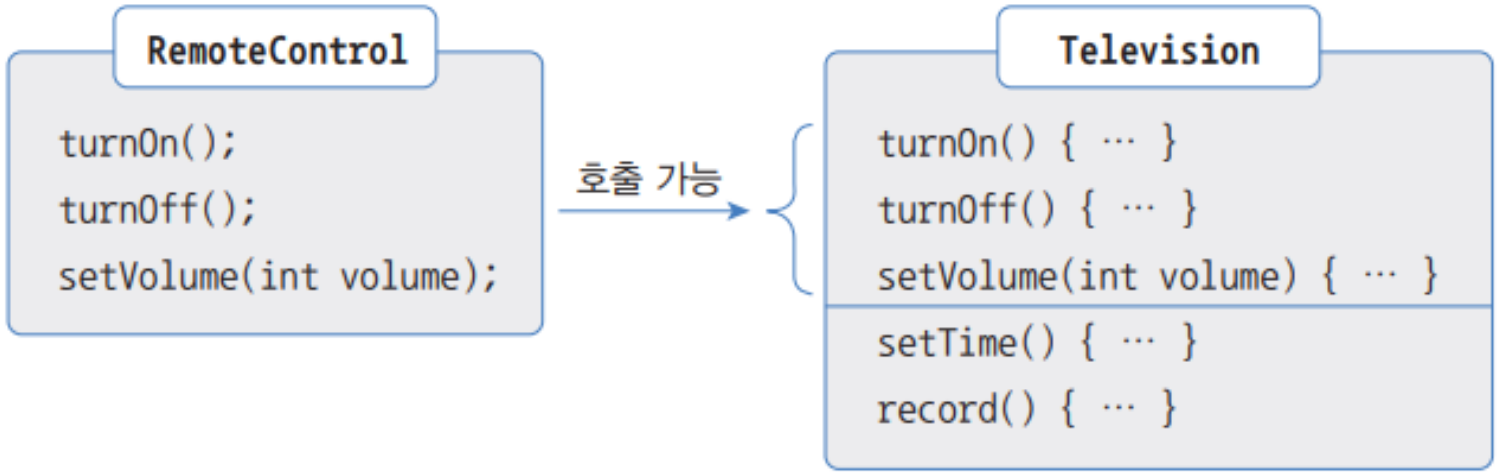
- 캐스팅 기호를 사용해서 인터페이스 타입을 구현 클래스 타입으로 변환시키는 것



하나의 이름(인터페이스 이름)으로 변수를 선언해 놓으면 그 변수에 인터페이스로 구현한 클래스, 그 클래스를 상속해서 만든 클래스의 객체를 모두 변수에 넣어서 사용할 수 있음.

강제 타입 변환

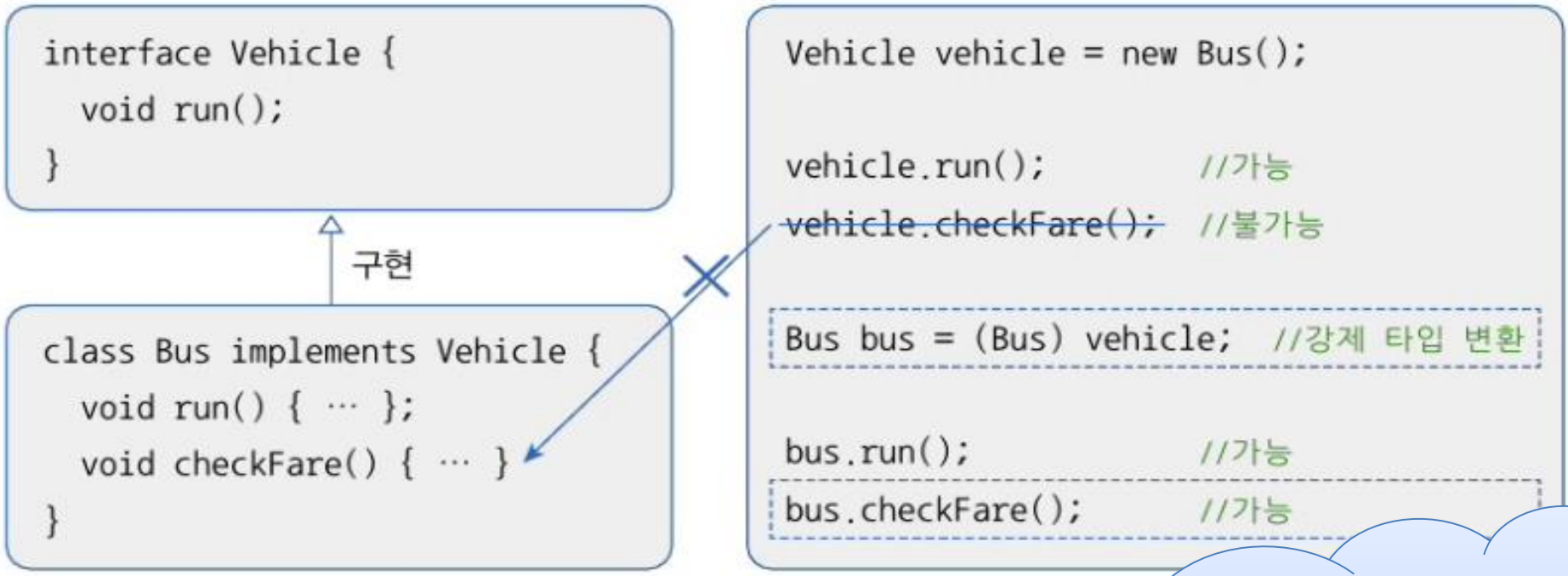
- 구현 객체가 인터페이스 타입으로 자동 변환되면, 인터페이스에 선언된 메소드만 사용 가능




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

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

강제 타입 변환



1. 버스나 기타 다른 이동수단이든 넣으려고 Vehicle타입의 변수를 선언해 둔 경우 → 부모 클래스인 Vehicle타입으로 변환되어 저장됨.
2. 버스의 checkFare() 기능을 사용할 수 없음.(Vehicle에는 없는 기능이므로)
3. 다시 버스의 모든 메서드를 사용하기 위해서는 강제 타입 변환을 해주어야 함.

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.");
		}
}
```

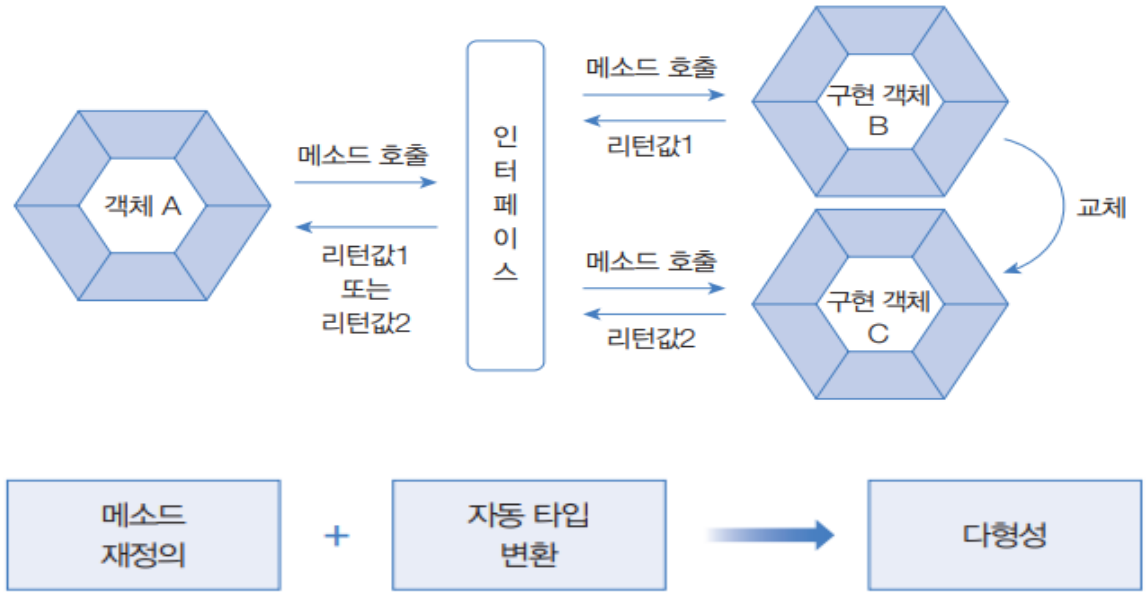
CastingExample.java

```
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.
```

다형성

- 사용 방법은 동일하지만 다양한 결과가 나오는 성질



- 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

```
package ch08.sec11.exam01;

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

KumhoTire.java

```
package ch08.sec11.exam01;

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

Car.java

```
package ch08.sec11.exam01;

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

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

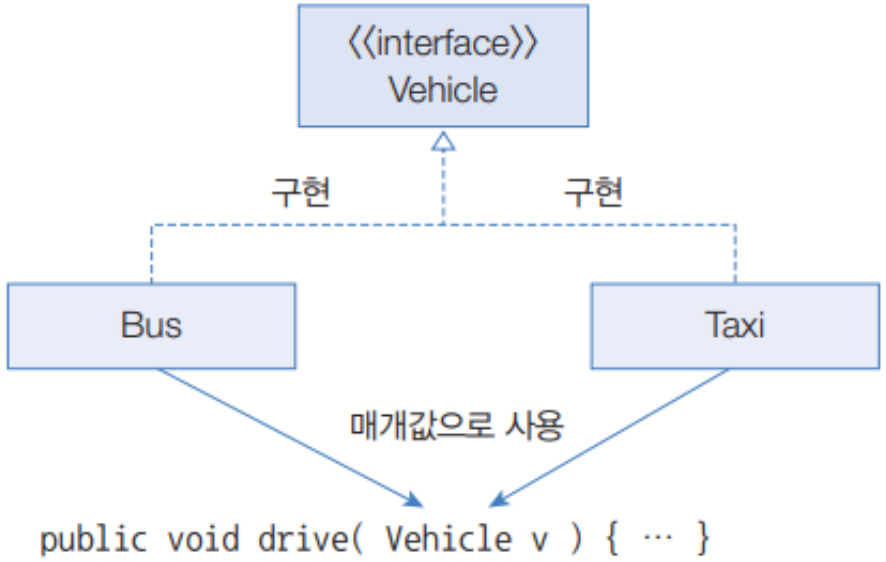
CarExample.java

```
package ch08.sec11.exam01;

public class CarExample {
	public static void main(String[] args) {
		//자동차 객체 생성
		Car myCar = new Car();

		//run() 메소드 실행
		myCar.run();

		//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



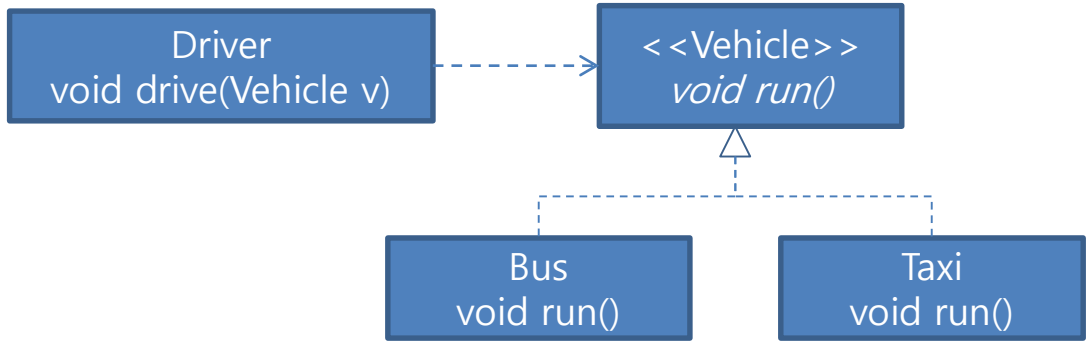

```
public interface Vehicle {
  void run();
}
```

구현 객체가 대입될 수 있도록 매개변수를 인터페이스 타입으로 선언

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

인터페이스의 추상 메소드 호출

매개변수의 다형성




```
Driver driver = new Dirver();

Bus bus = new Bus();

driver.drive( bus );
```

자동 타입 변환 발생
Vehicle vehicle = bus;

구현 객체

```
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

```
package ch08.sec11.exam02;

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();
	}
}
```

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 → Vehicle
		driver.drive(taxi);    // 자동 타입 변환: Taxi → Vehicle
	}
}
```

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
  }
}
```

vehicle에 대입된 객체가 Bus일 경우에만 Bus로 강제 타입 변환시킴

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
}
```

Vehicle.java

```
package ch08.sec12;

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

Bus.java

```
package ch08.sec12;

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

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

Taxi.java

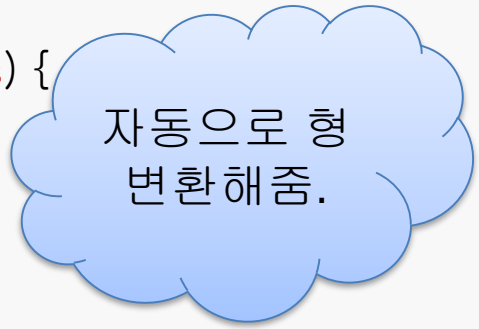
```
package ch08.sec12;

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

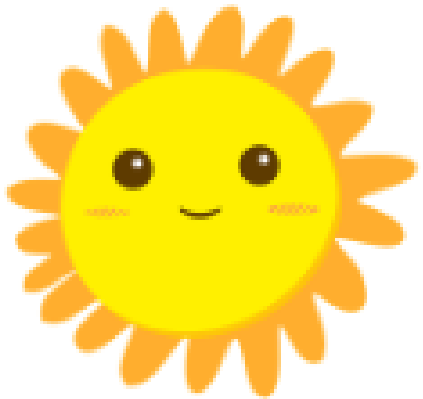
InstanceofExample.java

```
package ch08.sec12;

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();
        }
}
```

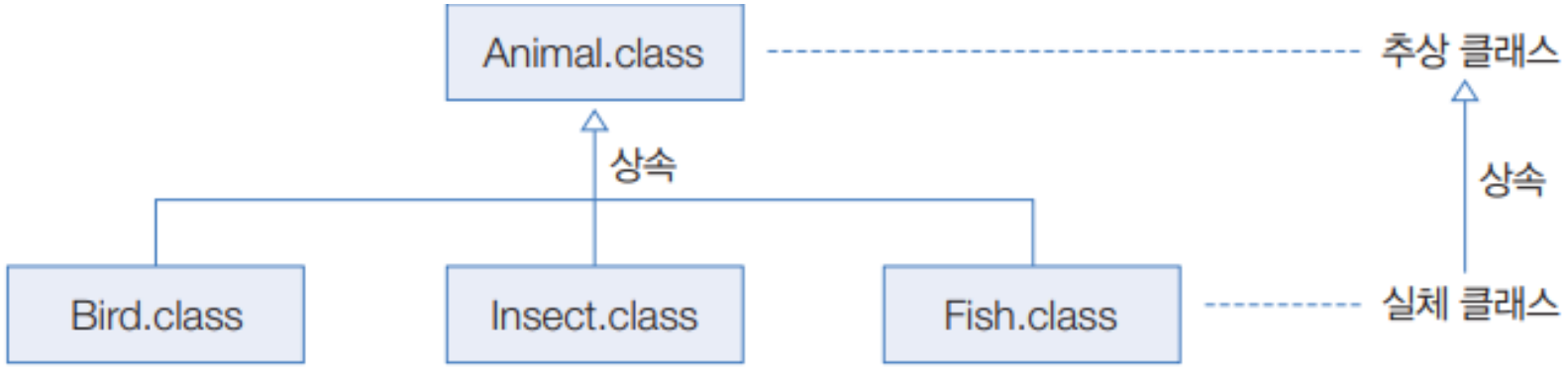




추상클래스

추상 클래스(불완전한 클래스, 객체 생성 불가)

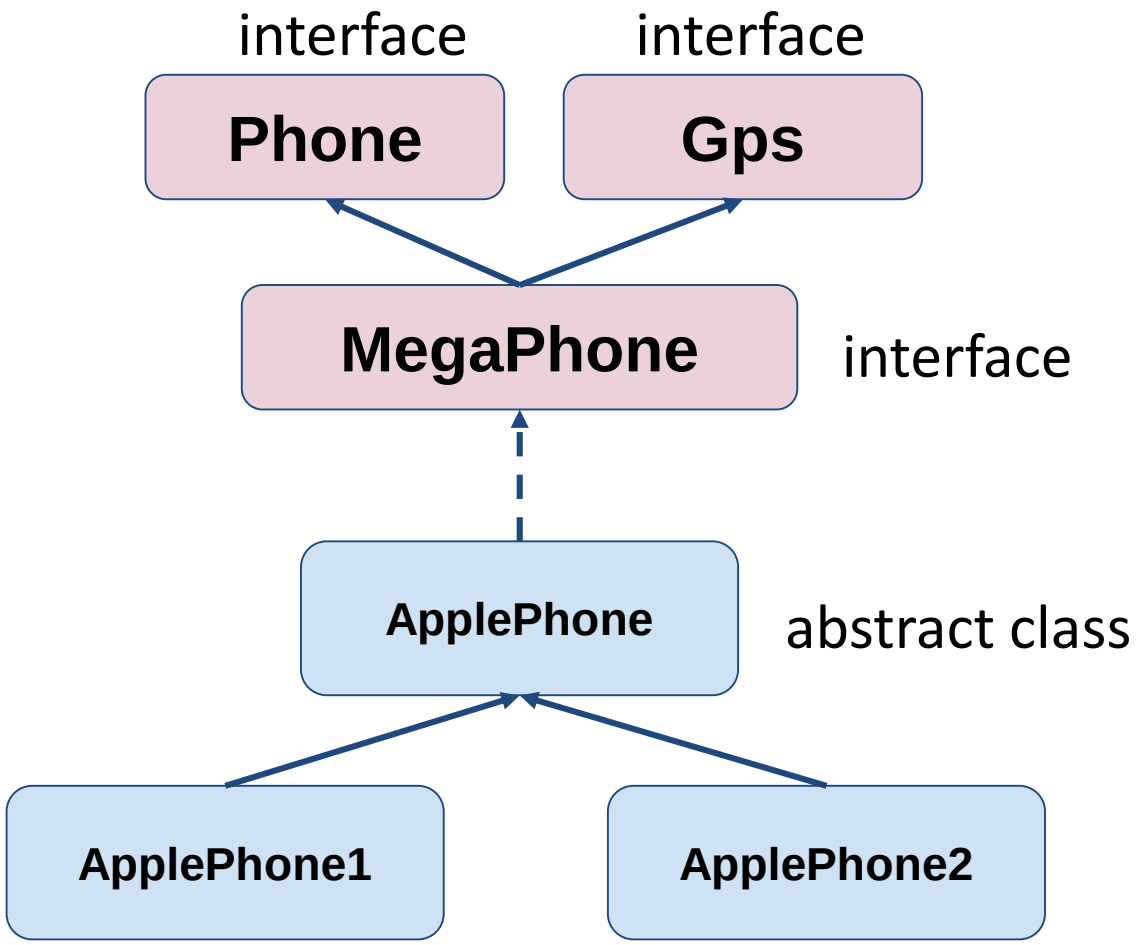
- 객체를 생성할 수 있는 실체 클래스들의 공통적인 필드나 메소드를 추출해서 선언한 클래스
- 추상 클래스는 실체 클래스의 부모 역할. 공통적인 필드나 메소드를 물려받을 수 있음




```
Animal animal = new Animal();     //x
```

```
class Fish extends Animal {
   ...
}
```

Phone, Gps interface



Gps.java

```java
package 인터페이스정의;

public interface Gps {
    public abstract void map();
}

```

Phone.java

```java
package 인터페이스정의;

public interface Phone {
    public abstract void internet();
    public abstract void call();
    public abstract void text();
    public abstract void kakao();
}
```

MegaPhone ← Phone, Gps (인터페이스 다중 상속)

```
public interface MegaPhone extends Phone, Gps{
    //인터페이스는 다중 상속을 할 수 있다.
    //일반 클래스는 단일 상속만 가능하다.
    //인터페이스는 객체생성을 할 수 없다.
    //추상메소드(abstract method)를 포함한 것은
    //객체생성불가
    //인터페이스는 일반 변수를 가질 수 없다.
    //인터페이스는 상수는 들어갈 수 있다.
    //인터페이스는 강제성을 가지고 있다.
    //=> 클래스에서 해당 메소드를 꼭 구현해주어야한다.
    //                          (꼭 오버라이드, 재정의)
    public final String COMPANY = "mega";
    public abstract void siri();
}
```

MegaPhone <- - ApplePhone

추상 클래스

- 추상메서드를 하나라도 가지면 무조건 추상클래스

```
public abstract class ApplePhone implements MegaPhone {
    int size;

    //추상메소드
    public abstract void camera();

    @Override
    public void internet() {
        System.out.println("인터넷하다.");
    }

    @Override
    public void call() {
        System.out.println("전화하다.");
    }

    @Override
    public void text() {
        System.out.println("문자하다.");
    }

    @Override
    public void kakao() {
```

ApplePhone ← ApplePhone1, ApplePhone2

ApplePhone2.java

```java
package 인터페이스구현;

public class ApplePhone2 extends ApplePhone {

    @Override
    public void camera() {
        System.out.println("인덕션 카메라로 찍다.");
    }

}
```

전화기구매.java | ApplePhone.java | ApplePhone1.java

```java
package 인터페이스구현;

public class ApplePhone1 extends ApplePhone {

    @Override
    public void camera() {
        System.out.println("렌즈 1개짜리 카메라로 찍다.");
    }

}
```

추상클래스를 상속받은 일반클래스는 무조건 추상메서드를 오버라이드해주어야 한다.

추상 클래스 선언

- 클래스 선언에 abstract 키워드를 붙임
- new 연산자를 이용해서 객체를 직접 만들지 못하고 상속을 통해 자식 클래스만 만들 수 있다.

```
public abstract class 클래스명 {
  //필드
  //생성자
  //메소드
}
```

Phone.java

```
package ch07.sec10.exam01;

public abstract class Phone {
	//필드 선언
	String owner;

	//생성자 선언
	Phone(String owner) {
		this.owner = owner;
	}

	//메소드 선언
	void turnOn() {
		System.out.println("폰 전원을 켭니다.");
	}

	void turnOff() {
		System.out.println("폰 전원을 끕니다.");
	}
}
```

SmartPhone.java

```
package ch07.sec10.exam01;

public class SmartPhone extends Phone {
	//생성자 선언
	SmartPhone(String owner) {
		//Phone 생성자 호출
		super(owner);
	}

	//메소드 선언
	void internetSearch() {
		System.out.println("인터넷 검색을 합니다.");
	}
}
```

PhoneExample.java

```
package ch07.sec10.exam01;

public class PhoneExample {
	public static void main(String[] args) {
		//Phone phone = new Phone();

		SmartPhone smartPhone = new SmartPhone("홍길동");

		smartPhone.turnOn();
		smartPhone.internetSearch();
		smartPhone.turnOff();
	}
}
```

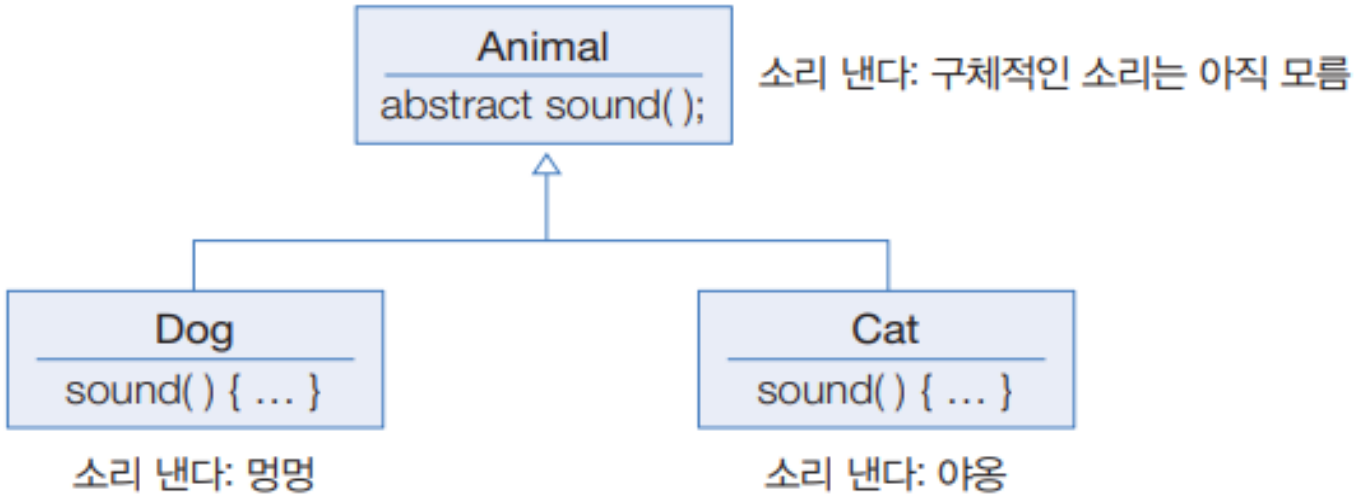
```
폰 전원을 켭니다.
인터넷 검색을 합니다.
폰 전원을 끕니다.
```

추상 메소드와 재정의

- 자식 클래스들이 가지고 있는 공통 메소드를 뽑아내어 추상 클래스로 작성할 때, 메소드 선언부만 동일하고 실행 내용은 자식 클래스마다 달라야 하는 경우 추상 메소드를 선언할 수 있음
- **일반 메소드 선언과의 차이점은 abstract 키워드가 붙고, 메소드 실행 내용인 중괄호 { }가 없다.**

```
abstract 리턴타입 메소드명(매개변수, …);
```

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



Animal.java

```
package ch07.sec10.exam02;

public abstract class Animal {
		//메소드 선언
		public void breathe() {
			System.out.println("숨을 쉽니다.");
		}

		//추상 메소드 선언
		public abstract void sound();
}
```

Dog.java

```
package ch07.sec10.exam02;

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

Cat.java

```
package ch07.sec10.exam02;

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

AbstractMethodExample.java

```
package ch07.sec10.exam02;

public class AbstractMethodExample {
	public static void main(String[] args) {
		Dog dog = new Dog();
		dog.sound();

		Cat cat = new Cat();
		cat.sound();

		//매개변수의 다형성
		animalSound(new Dog());
		animalSound(new Cat());
	}

	public static void animalSound( Animal animal ) {
		animal.sound(); // 재정의된 메소드 호출
	}
}
```

자동 타입 변환

```
멍멍
야옹
멍멍
야옹
```

인터페이스

- 필요한 기능만 추상적으로 정의하는 메서드로 구성(메서드 내부를 구현하지 않음.)
- 다중상속 가능
- 구현된 객체에서 공유할 목적으로 public static final 정적 상수 선언 가능
- 변수는 사용하지 않음.

추상메서드

- 필요한 기능만 추상적으로 정의하는 메서드

형변환

- 클래스간의 형변환은 상속관계에서만 가능
- 인터페이스를 구현한 클래스간 형변환가능
- 무조건 클래스 다이어그램상의 위(부모)클래스가 개념적으로 큼.
- 자동형변환, 강제형변환 가능

추상클래스

- 추상메서드를 하나라도 가지는 클래스
- 추상메서드는 구체적으로 구현되지 않은 추상적인 메서드
- 추상메서드는 상속받은 일반클래스에서 반드시 구체적으로 구현해주어야함.
- 상속받은 클래스에서 추상메서드를 재정의해서 사용하게 하고자 하는 경우 사용