Chapter 08. 인터페이스

8.1 인터페이스 역할

인터페이스(interface) 는 사전적인 의미로 두 장치를 연결하는 접속기를 말한다. 여기서 두 장치를 서로 다른 객체로 본다면, 인터페이스는 이 *두 객체를 연결하는 역할*을 한다. 객체 A가 인터페이스의 메소드를 호출하면 인터페이스가 객체 B의 메소드를 호출하고 그 결과를 받아 객체 A로 전달해준다고 할 때, 만약 객체 B가 객체 C로 교체된다고 하더라도 객체 A는 인터페이스의 메소드만 사용하므로 이 사실을 몰라도 된다. 이때 만약 인터페이스 없이 객체 A가 객체 B를 직접 사용한다면 객체 A의 소스 코드를 객체 B에서 객체 C로 변경하는 작업이 추가로 필요하게 된다.

이 특징으로 인해 인터페이스는 **다형성 구현**에 주된 기술로 이용된다.

8.2 인터페이스와 구현 클래스 선언

RemoteControl.java

```
package ch08.sec02;

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

Television.java

```
package ch08.sec02;

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

Audio.java

```
package ch08.sec02;

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

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();
    }
}
```

결과

```
TV를 켭니다.
Audio를 켭니다.
```

8.3 상수 필드

- 인터페이스에 선언된 필드는 모두 `public static final` 특성을 갖음. (생략하더라도 자동적으로 컴파일 과정에 붙는다.)
- 상수명은 대문자로 작성, 언더바(_)로 연결하는 것이 관례.
- 구현 객체와 관련 없는 인터페이스 소속 멤버이므로 인터페이스로 바로 접근해서 상수값 읽을 수 있음.

RemoteControl.java

```
package ch08.sec03;

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

RemoteControlExample.java

```
package ch08.sec03;

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);
        }
    }
```

8.4 추상 메소드

- 추상 메소드는 객체가 인터페이스를 통해 어떻게 메소드를 호출할 수 있는지 방법을 알려주는 역할.
- `public abstract`는 생략하더라도 컴파일 과정에 자동으로 붙음.
- 인터페이스 구현 객체는 추상 메소드의 실행부를 갖는 재정의된 메소드가 있어야 함.
- 구현 클래스에서 추상 메소드를 재정의할 때 `public`보다 더 낮은 접근 제한으로 재정의할 수 없음. (인터페이스 추상 메소드는 기본적으로 `public` 접근 제한을 갖기 때문)
- 인터페이스로 구현 객체를 사용하려면 인터페이스 변수 선언 후 구현 객체를 대입.
- 인터페이스 변수는 참조 타입이므로 구현 객체의 번지를 저장.
- 어떤 구현 객체가 대입되었는지에 따라 실행 내용 달라짐.

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) {
        //인터페이스 상수 필드를 이용해서 volume 필드의 값을 제한
        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);
    }
}
```

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();
    }
}
```

결과

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

8.5 디폴트 메소드

- 인터페이스에 선언할 수 있는 완전한 실행 코드를 가진 메소드
- `default` 키워드를 리턴 타입 앞에 붙인다.
- 구현 객체가 필요한 메소드로 호출하기 위해선 구현 객체를 인터페이스 변수에 대입한 후 호출해야 한다.

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 인스턴스 메소드
    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.");
        }
    }
}
```

RemoteControlExample.java

```
package ch08.sec05;

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);

        //디폴트 메소드 호출
        rc.setMute(true);
        rc.setMute(false);

        System.out.println();

        //Audio 객체를 생성하고 인터페이스 변수에 대입
        rc = new Audio();
        rc.turnOn();
        rc.setVolume(5);

        //디폴트 메소드 호출
        rc.setMute(true);
        rc.setMute(false);
    }
}
```

8.6 정적 메소드

- 선언 방법 클래스 메소드와 완전 동일
- 구현 객체가 없어도 인터페이스만으로 호출 가능
- `public` 생략해도 컴파일 과정에서 자동으로 붙음.
- 정적 메소드의 실행부에서는 상수 필드를 제외한 추상 메소드, 디폴트 메소드, `private` 메소드 등을 호출할 수 없음.

8.7 private 메소드

- 디폴트와 정적 메소드들의 중복 코드를 줄이기 위해 사용
- `private` 메소드는 디폴트 메소드 안에서만, `private` 정적 메소드는 디폴트 메소드와 정적 메소드 안에서 호출 가능

Service.java

```
package ch08.sec07;

public interface Service {
    //디폴트 메소드
    default void defaultMethod1() {
        System.out.println("defaultMethod1 종속 코드");
        defaultCommon();
    }

    default void defaultMethod2() {
        System.out.println("defaultMethod2 종속 코드");
        defaultCommon();
    }

    //private 메소드
    private void defaultCommon() {
        System.out.println("defaultMethod 중복 코드A");
        System.out.println("defaultMethod 중복 코드B");
    }

    //정적 메소드
    static void staticMethod1() {
        System.out.println("staticMethod1 종속 코드");
        staticCommon();
    }

    static void staticMethod2() {
        System.out.println("staticMethod2 종속 코드");
        staticCommon();
    }

    //private 정적 메소드
    private static void staticCommon() {
        System.out.println("staticMethod 중복 코드C");
```

```
        System.out.println("staticMethod 중복 코드D");
    }
}
```

8.8 다중 인터페이스 구현

- 구현 객체는 여러 개의 인터페이스를 `implements` 할 수 있음.
- `implements` 뒤에 쉼표로 구분해서 작성
- 모든 인터페이스가 가진 추상 메소드를 재정의해야 함.
- 구현 객체가 어떤 인터페이스 변수에 대입되느냐에 따라 변수를 통해 호출할 수 있는 추상 메소드가 결정

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.");
    }
}
```

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");
    }
}
```

결과

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

8.9 인터페이스 상속

- 다른 인터페이스 상속 가능, 클래스와 달리 다중 상속 허용함.
- `extends` 뒤에 상속할 인터페이스들을 나열
- 자식 인터페이스의 구현 클래스는 자식 인터페이스의 메소드뿐 아니라 부모 인터페이스의 모든 추상 메소드 재정의
- 구현 객체는 자식 및 부모 인터페이스 변수에 대입 가능

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();
    }
}
```

결과

```
InterfaceCImpl-methodA() 실행

InterfaceCImpl-methodB() 실행

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

8.10 타입 변환

- 인터페이스와 구현 클래스 간에 발생
- 인터페이스 변수에 구현 객체를 대입하면 구현 객체는 인터페이스 타입으로 자동 타입 변환
- 인터페이스 타입을 구현 클래스 타입으로 변환시킬 때는 강제 타입 변환 필요

자동 타입 변환

A.java

```
package ch08.sec10.exam01;

public interface A {
}
```

B.java

```java
package ch08.sec10.exam01;

public class B implements A {
}
```

C.java

```java
package ch08.sec10.exam01;

public class C implements A {
}
```

D.java

```java
package ch08.sec10.exam01;

public class D extends B {
}
```

E.java

```java
package ch08.sec10.exam01;

public class E extends C {
}
```

PromotionExample.java

```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

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();
    }
}
```

결과

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

8.11 다형성

- **다형성** : 사용 방법은 동일하지만 다양한 결과가 나오는 성질
- 구현 객체들 중 어느 객체가 인터페이스에 대입되었느냐에 따라 메소드 호출 결과 달라짐.

필드의 다형성

Tire.java

```
package ch08.sec11.exam01;

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

HankookTire.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();
    }
}
```

결과

```
한국 타이어가 굴러갑니다.
한국 타이어가 굴러갑니다.
금호 타이어가 굴러갑니다.
금호 타이어가 굴러갑니다.
```

매개변수의 다형성

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

Vehicle.java

```java
package ch08.sec11.exam02;

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

Driver.java

```java
package ch08.sec11.exam02;

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

Bus.java

```java
package ch08.sec11.exam02;

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

Taxi.java

```java
package ch08.sec11.exam02;
```

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

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);
        driver.drive(taxi);
    }
}
```

결과

```
버스가 달립니다.
택시가 달립니다.
```

8.12 객체 타입 확인

- 객체 타입을 확인하기 위해 `instanceof` 연산자 사용 가능

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);
    }

    //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();
    }
}
```

8.13 봉인된 클래스

- 무분별한 자식 인터페이스 생성 방지 위해 봉인된(`sealed`) 인터페이스 사용 가능
- `sealed` 키워드 사용 후 `permits` 키워드 뒤에 상속 가능한 자식 인터페이스를 지정
- 봉인된 인터페이스를 상속하는 인터페이스는 `non-sealed` 키워드로 봉인을 해제하거나 `sealed` 키워드를 사용해 또 다른 봉인 인터페이스로 선언