Chapter 08. Interface

8.1 인터페이스의 역할

- **인터페이스(interface)** : 객체의 사용 방법을 정의한 타입
  - **다형성**을 구현하는 매우 중요한 역할
  - 개발 코드와 객체가 서로 통신하는 접점 역할, 개발 코드가 인터페이스의 메소드를 호출하면 **인터페이스**는 객체의 메소드를 호출시킨다.
  - 개발 코드를 수정하지 않고, 사용하는 **객체를 변경**하기 위해서

    : 실행 내용과 리턴값을 **다양화**할 수 있다는 장점

8.2 인터페이스 선언

8.2.1 인터페이스 선언

: 인터페이스 선언은 class 키워드 대신에 interface 키워드를 사용한다.

```
[public] interface 인터페이스명 { ... }

// ex)
public interface RemoteControl { }
```

- **인터페이스 구성**
  - 상수와 메소드만을 구성 멤버로 가진다.
  - 객체로 생성할 수 없기 때문에 생성자를 가질 수 없다.
  - 추상메소드, 디폴트 메소드, 정적 메소드 선언 가능

```
interface 인터페이스명 {
    // 상수
    타입 상수명 = 값;

    // 추상 메소드
    타입 메소드명(매개변수, ...);

    // 디폴트 메소드
    default 타입 메소드명(매개변수, ...) { ... }

    // 정적 메소드
    static 타입 메소드명(매개변수) { ... }
}
```

상수 필드(Constant Field)

: 상수는 인터페이스에 고정된 값으로 런타임 시에 데이터를 바꿀 수 없다. 상수를 선언할 때에는 반드시 초기값 대입.

추상 메소드(Abstract Method)

: 객체를 가지고 있는 메소드를 설명한 것으로 호출할 때 어떤 매개값이 필요하고, 리턴 타입이 무엇인지만 알려준다. 실제 실행부는 객체가 가지고 있다.

디폴트 메소드(Default Method)

: 인터페이스에 선언이 되지만 객체(구현 객체)가 가지고 있는 인스턴스 메소드라고 생각해야 한다.

정적 메소드(Static Method)

: 디폴트 메소드와는 달리 객체가 없어도 인터페이스만으로 호출이 가능하다.

8.2.2 상수 필드 선언

: 인터페이스는 데이터를 저장할 수 없기 때문에 데이터를 저장할 인스턴스 또는 정적 필드를 선언할 수 없다. 대신 상수 필드만 선언할 수 있다.

- **선언 예시**

```
[public static final] 타입 상수명 = 값;
```

> public, static, final을 생략해도 자동적으로 컴파일 과정에서 붙는다.

- **예제**

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

8.2.3 추상 메소드 선언

: 인터페이스를 통해 호출된 메소드는 최종적으로 객체에서 실행된다. 그렇기 때문에 리턴 타입, 메소드명, 매개 변수만 기술되고 중괄호{}를 붙이지 않는 추상 메소드를 선언한다.

```
[public abstract] 리턴타입 메소드명(매개변수, ...);
```

> public abstract를 생략하더라도 자동적으로 컴파일 과정에서 붙게 된다.

- **예제**

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

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

8.2.4 디폴트 메소드 선언

: 형태는 클래스의 인스턴스 메소드와 동일한데, default 키워드가 리턴 타입 앞에 붙는다.

디폴트 메소드는 public 특성을 갖기 때문에 public을 생략하더라도 자동적으로 컴파일 과정에서 붙게 된다.

```
[public] default 리턴타입 메소드명(매개변수, ...) { ... }
```

- **예제**

  RemoteControl.java(**메소드 선언**)

```
package default_method_declare;

public interface RemoteContol {
    // 상수
    int MAX_VOLUME = 10;
    int MIN_VOLUME = 0;

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

    // 디폴트 메소드(실행 내용까지 작성)
    default void setMute(boolean mute) {
        if(mute) {
            System.out.println("무음 처리합니다");
        } else {
```

```
            System.out.println("무음 해제합니다");
        }
    }
}
```

8.2.5 정적 메소드 선언

: 형태는 클래스의 정적 메소드와 완전 동일하다. 정적 메소드는 public 특성을 갖기 때문에 public을 생략하더라도 자동적으로 컴파일 과정에서 붙게 된다.

```
[public] static 리턴타입 메소드명(매개변수, ...) { ... }
```

- **예제**

  RemoteControl.java(**메소드 선언**)

```
package static_method_declare;

public interface RemoteControl {
    // 상수
    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) {
        if (mute) {
            System.out.println("무음 처리합니다.");
        } else {
            System.out.println("무음 해제합니다.");
        }
    }

    // 정적 메소드
    static void changeBattery() {
        System.out.println("건전지를 교환합니다.");
    }
}
```

8.3 인터페이스 구현

: 객체는 인터페이스에서 정의된 추상 메소드와 동일한 실체 메소드를 가지고 있어야 한다.

이러한 객체를 인터페이스의 구현(implement) 객체라고 하고, 구현 객체를 생성하는 클래스를 구현 클래스라고 한다.

8.3.1 구현 클래스

: 보통의 클래스와 동일한데, 인터페이스 타입으로 사용할 수 있음을 알려주기 위해 클래스 선언부에 implements 키워드를 추가하고 인터페이스명을 명시.

- **예시**

```
public class 구현클래스명 implements 인터페이스명{
    // 인터페이스에 선언된 추상 메소드의 실체 메소드 선언
}
```

- **예제**

  Television.java(**구현 클래스**)

```
package implement_class;

import static_method_declare.RemoteControl;

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

    // turnOn() 추상 메소드의 실체 메소드
    public void turnOn(){
        System.out.println("TV를 켭니다.");
    }

    // turnOff() 추상 메소드의 실체 메소드
    public void turnOff(){
        System.out.println("TV를 끕니다.");
    }

    // setVolume() 추상 메소드의 실체 메소드
    // 인터페이스 상수를 이용해서 volume 필드의 값 제한
    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: " + volume);
        }
    }
```

- **실체 메소드를 작성할 때 주의할 점**
  - 인터페이스의 모든 메소드는 기본적으로 public 접근 제한을 갖기 때문에 **public보다 더 낮은 접근 제한으로 작성할 수 없다.**
  - 만약 인터페이스에 선언된 추상 메소드에 대응하는 실체 메소드를 구현 클래스가 작성하지 않으면 구현 클래스는 자동적으로 추상 클래스가 된다. 그렇기 때문에 클래스 선언부에 **abstract 키워드**를 추가해야 한다.

```
public abstract class Television implements RemoteControl {
    // setVolume() 실체 메소드가 없고 일부만 구현
    public void turnOn() {...}
    public void turnOff() {...}
}
```

- **예제**

RemoteControlExample.java(**인터페이스 변수에 구현 객체 대입**)

```
package implement_class;

import static_method_declare.RemoteControl;

public class RemoteControlExample {
    public static void main(String[] args) {
        RemoteControl rc;       // 인터페이스 선언
        rc = new Television();  // 구현 객체 Television 대입
        rc = new Audio();       // 구현 객체 Audio 대입
    }
}
```

> **인터페이스로 구현 객체 사용**
>
> - 인터페이스 변수를 먼저 선언한다.
> - 구현 객체를 대입하여 사용한다.

8.3.2 익명 구현 객체

: 소스 파일을 만들지 않고도 구현 객체를 만들 수 있는 방법.

- UI 프로그래밍에서 이벤트를 처리하기 위해
- 임시 작업 스레드를 만들기 위해

- **예시**

```
인터페이스 변수 = new 인터페이스() {
    // 인터페이스에 선언된 추상 메소드의 실체 메소드 선언
};
```

- **예제**

```
package anonymous_implement;

import static_method_declare.RemoteControl;

public class RemoteControlExample {
    public static void main(String[] args) {
        RemoteControl rc = new RemoteControl() {
            @Override
            public void turnOn() {
                // 실행문
            }
            @Override
            public void turnOff() {
                // 실행문
            }

            @Override
            public void setVolume(int volume) {
                // 실행문
            }
        };
    }
}
```

> 추가적으로 필드와 메소드를 선언할 수 있지만, 익명 객체 안에서만 사용할 수 있고 인터페이스 변수로 접근할 수 없다.

8.3.3 다중 인터페이스 구현 클래스

: 인터페이스 A와 인터페이스 B가 하나의 객체의 메소드를 호출할 수 있으려면 객체는 이 두 인터페이스를 모두 구현해야 한다.

- **예시**

```
public class 구현클래스명 implements 인터페이스A, 인터페이스B {
    // 인터페이스 A에 선언된 추상 메소드의 실체 메소드 선언
    // 인터페이스 B에 선언된 추상 메소드의 실체 메소드 선언
}
```

- **예제**

  Searchable.java(**인터페이스**)

```
package multiple_interface;

import static_method_declare.RemoteControl;

public class SmartTelevision implements RemoteControl, Searchable {
    // RemoteControl의 추상 메소드에 대한 실체 메소드
    private int volume;

    public void turnOn() {
        System.out.println("TV를 켭니다.");
    }
    public void turnOff() {
        System.out.println("TV를 끕니다.");
    }
    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: " + volume);
    }

    // Searchable의 추상 메소드에 대한 실체 메소드
    public void search(String url) {
        System.out.println(url + " 을 검색합니다.");
    }
}
```

8.4 인터페이스 사용

: 인터페이스 변수를 선언하고 구현 객체를 대입하여 사용한다.

- **예시**

```
RemoteControl = rc;          // 인터페이스 변수 선언
rc = new Television();       // 구현 객체 대입
rc = new Audio();            // 구현 객체 대입
```

- 개발 코드에서 인터페이스는 클래스의 필드, 생성자 또는 메소드의 매개 변수, 생성자 또는 메소드의 로컬 변수로 선언될 수 있다.

```
public class MyClass {
    // 필드
    RemoteControl rc = new Television();

    // 생성자
    MyClass(RemoteControl rc) {
        this.rc = rc;
    }

    // 메소드
    void methodA() {
        // 로컬 변수
        RemoteControl rc = new Audio();
    }

    void methodB(RemoteControl rc) { ... }
}
```

8.4.1 추상 메소드 사용

: 구현 객체가 인터페이스 타입에 대입되면 인터페이스에 선언된 추상 메소드를 개발 코드에서 호출할 수 있게 된다.

- **예시**

```
RemoteControl rc = new Television();
rc.turnOn();    // Television의 turnOn() 실행
rc.turnOff();   // Television의 turnOff() 실행
```

- **예제**

  RemoteControlExample.java(**인터페이스 사용**)

```
package abstract_method_using;

import implement_class.Audio;
import implement_class.Television;
import static_method_declare.RemoteControl;

public class RemoteControlExample {
    public static void main(String[] args) {
        // 인터페이스 변수 선언
        RemoteControl rc = null;

        // Television 객체를 인터페이스 타입에 대입
        rc = new Television();
```

```
        // 인터페이스의 turnOn(), turnOff() 호출
        rc.turnOn();
        rc.turnOff();

        // Audio 객체를 인터페이스 타입에 대입
        rc = new Audio();

        // 인터페이스의 turnOn(), turnOff() 호출
        rc.turnOn();
        rc.turnOff();
    }
}
```

실행 결과

```
TV를 켭니다.
TV를 끕니다.
Audio를 켭니다.
Audio를 끕니다.
```

8.4.2 디폴트 메소드 사용

: 디폴트 메소드는 추상 메소드가 아닌 인스턴스 메소드이므로 구현 객체가 있어야 사용할 수 있다.

- **예제**

  Audio.java(**구현 클래스**)

```
package default_method_using;

import default_method_declare.RemoteControl;

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

    // turnOn() 추상 메소드의 실체 메소드
    public void turnOn() {
        System.out.println("Audio를 켭니다.");
    }

    // turnOff() 추상 메소드의 실체 메소드
    public void turnOff() {
        System.out.println("Audio를 끕니다.");
    }

    // setVolume() 추상 메소드의 실체 메소드
```

```
    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);
    }

    // 디폴트 메소드 재정의
    public void setMute(boolean mute) {
        this.mute = mute;
        if(mute) {
            System.out.println("Audio 무음 처리합니다.");
        } else {
            System.out.println("Audio 무음 해제합니다.");
        }
    }
}
```

RemoteControlExample.java(**디폴트 메소드 사용**)

```
package default_method_using;

import static_method_declare.RemoteControl;
import implement_class.Television;

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

        rc = new Television();
        rc.turnOn();

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

        rc = new Audio();
        rc.turnOn();

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

실행 결과

```
TV를 켭니다.
무음 처리합니다.
Audio를 켭니다.
Audio 무음 처리합니다.
```

8.4.3 정적 메소드 사용

: 인터페이스의 정적 메소드는 인터페이스로 바로 호출이 가능하다.

- **예제**

  RemoteControlExample.java(**정적 메소드 사용**)

```java
package static_method_using;

import static_method_declare.RemoteControl;

public class RemoteControlExample {
    public static void main(String[] args) {
        // 인터페이스의 정적 메소드 호출
        RemoteControl.changeBattery();
    }
}
```

  실행 결과

```
건전지를 교환합니다.
```

8.5 타입 변환과 다형성

- **다형성** : 하나의 타입에 대입되는 객체에 따라서 실행 결과가 다양한 형태로 나오는 성질.
  - 인터페이스 타입에 어떤 구현 객체를 대입하느냐에 따라 실행 결과가 달라진다.
  - 인터페이스는 메소드의 매개 변수로 많이 등장한다. 인터페이스 타입으로 매개 변수를 선언하면 메소드 호출 시 **매개값으로 여러 가지 종류의 구현 객체를 줄 수 있기 때문에** 메소드 실행 결과가 다양하게 나온다.

8.5.1 자동 타입 변환(Promotion)

: 구현 객체가 인터페이스 타입으로 변환되는 것

```
인터페이스 변수 = 구현객체;    // 자동 타입 변환
```

> - 인터페이스 구현 클래스를 상속해서 자식 클래스를 만들었다면 자식 객체 역시 인터페이스 타입으로 자동 타입 변환시킬 수 있다.
> - 필드와 매개 변수의 타입을 인터페이스로 선언하고 다양한 구현 객체를 대입하면 실행 결과가 다양하게 나온다.

8.5.2 필드의 다형성

- **예제**

  Tire.java(**인터페이스**)

```
package field_polymorphism;

// 인터페이스 선언
public interface Tire {
    public void roll();
}
```

  HankookTire.java(**구현 클래스**)

```
package field_polymorphism;

public class HankookTire implements Tire {
    // 실체 메소드 구현
    public void roll() {
        System.out.println("한국 타이어가 굴러간다.");
    }
}
```

  KumhoTire.java(**구현 클래스**)

```
package field_polymorphism;

public class KumhoTire implements Tire {
    // 실체 메소드 구현
    public void roll() {
        System.out.println("금호 타이어가 굴러간다.");
    }
}
```

  Car.java(**필드 다형성**)

```
package field_polymorphism;

// 인터페이스를 이용해 Car 클래스 구현
public class Car {
    Tire frontLeftTire = new HankookTire();
    Tire frontRightTire = new HankookTire();
    Tire backLeftTire = new HankookTire();
```

```
    Tire backRightTire = new HankookTire();

    void run() {
        frontLeftTire.roll();
        frontRightTire.roll();
        backLeftTire.roll();
        backRightTire.roll();
    }
}
```

CarExample.java(**필드 다형성 테스트**)

```
package field_polymorphism;

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

        myCar.run();

        // Car 객체의 필드를 새로운 실체 객체를 대입한다.
        myCar.frontLeftTire = new KumhoTire();
        myCar.frontRightTire = new KumhoTire();

        myCar.run();
    }
}
```

실행 결과

```
한국 타이어가 굴러간다.
한국 타이어가 굴러간다.
한국 타이어가 굴러간다.
한국 타이어가 굴러간다.
금호 타이어가 굴러간다.
금호 타이어가 굴러간다.
한국 타이어가 굴러간다.
한국 타이어가 굴러간다.
```

8.5.3 인터페이스 배열로 구현 객체 관리

- **예시**

```
// 4개의 타이어 필드를 인터페이스 배열로 선언
Tire[] tires = {
    new HankookTire(),
    new HankookTire(),
    new HankookTire(),
```

```
    new HankookTire()
};

// 인덱스를 통해 앞오른쪽 타이어를 KumhoTire로 교체
tires[1] = new KumhoTire();

// 전체 타이어의 roll() 메소드를 호출하는 for문
void run() {
    for(Tire tire : tires) {
        tire.roll();
    }
}
```

8.5.4 매개 변수의 다형성

: 매개 변수를 인터페이스 타입으로 선언하고 호출할 때에는 구현 객체를 대입한다.

- **예시**

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

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

Driver dirver = new Driver();
Bus bus = new Bus();
driver.drive(bus);      // 자동 타입 변환 발생, Vehicle vehicle = bus;
```

8.5.5 강제 타입 변환(Casting)

: 구현 객체가 인터페이스 타입으로 자동 변환하면, 인터페이스에 선언된 메소드만 사용 가능하다는 제약 사항이 따른다. 이때 **강제 타입 변환을 해서 다시 구현 클래스 타입으로 변환**한 다음, 구현 클래스의 필드와 메소드를 사용할 수 있다.

- **예시**

```
구현클래스 변수 = (구현클래스) 인터페이스변수;      // 강제 타입 변환
```

8.5.6 객체 타입 확인(instanceof)

: 강제 타입 변환은 구현 객체가 인터페이스 타입으로 변환되어 있는 상태에서 가능하다. 그러나 어떤 구현 객체가 변환되어 있는지 알 수 없는 상태에서 무작정 변환을 할 경우 ClassCastException이 발생할 수도 있다.

- **예시**

```
Vehicle vehicle = new Taxi();
Bus bus = (Bus) vehicle;    // 컴파일 에러!!
```

> 인터페이스 타입으로 자동 변환된 매개값을 메소드 내에서 다시 구현 클래스 타입으로 강제 타입 변환해야 한다면 반드시 매개값이 어떤 객체인지 **instanceof 연산자**로 확인하고 안전하게 강제 타입 변환을 해야 한다.

수정한 코드

```
public class Driver {
    public void drive(Vehicle vehicle) {
        // vehicle 매개 변수가 참조하는 객체가 Bus인지 조사
        if(vehicle instanceof Bus) {
            // Bus 객체일 경우 안전하게 강제 타입 변환시킴
            Bus bus = (Bus) vehicle;
            bus.checkFare();
        }
        vehicle.run();
    }
}
```

8.6 인터페이스 상속

: 인터페이스는 클래스와는 달리 다중 상속을 허용한다.

- **예시**

```
public interface 하위인터페이스 extends 상위인터페이스1, 상위인터페이스2 { ... }
```

> 하위 인터페이스는 하위 인터페이스의 메소드뿐만 아니라 상위 인터페이스의 모든 추상 메소드에 대한 실체 메소드를 가지고 있어야 한다. 그러므로 구현 클래스로 부터 객체를 생성하고 나서 하위 및 상위 인터페이스 타입으로 변환이 가능하다.

- **예제**

InterfaceA.java(**부모 인터페이스**)

```java
package interface_inherit;

public interface InterfaceA {
    public void methodA();
}
```

InterfaceB.java(**부모 인터페이스**)

```java
package interface_inherit;

public interface InterfaceB {
    public void methodB();
}
```

InterfaceC.java(**하위 인터페이스**)

```java
package interface_inherit;

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

Implementation.java(**하위 인터페이스 구현**)

```java
package interface_inherit;

public class ImplementationC implements InterfaceC{
    // InterfaceA와 InterfaceB의 실체 메소드도 있어야 한다.
    public void methodA() {
        System.out.println("ImplementationC-methodA() 실행");
    }

    public void methodB() {
        System.out.println("ImplementationC-methodB() 실행");
    }

    public void methodC() {
        System.out.println("ImplementationC-methodC() 실행");
    }
}
```

Example.java(**호출 가능 메소드**)

```java
package interface_inherit;

public class Example {
    public static void main(String[] args) {
        ImplementationC impl = new ImplementationC();

        // InterfaceA 변수는 methodA()만 호출 가능
```

```
        InterfaceA ia = impl;
        ia.methodA();
        System.out.println();

        // InterfaceB 변수는 methodB()만 호출 가능
        InterfaceB ib = impl;
        ib.methodB();
        System.out.println();

        // InterfaceC 변수는 methodA, B, C() 모두 호출 가능
        InterfaceC ic = impl;
        ic.methodA();
        ic.methodB();
        ic.methodC();
    }
}
```

실행 결과

```
ImplementationC-methodA() 실행

ImplementationC-methodB() 실행

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

8.7 디폴트 메소드와 인터페이스 확장

- **디폴트 메소드** : 인터페이스에 선언된 인스턴스 메소드이기 때문에 구현 객체가 있어야 사용할 수 있다.

8.7.1 디폴트 메소드의 필요성

: 기존 인터페이스를 확장해서 새로운 기능을 추가하기 위해서.

- **예제**

  MyInterface.java(**인터페이스**)

```java
package default_method_necessity;

public interface MyInterface {
    public void method1();

    // 디폴트 메소드
    public default void method2() {
        System.out.println("MyInterface-method2 실행");
    }
}
```

MyClassA.java(**구현 클래스**)

```java
package default_method_necessity;

public class MyclassA implements MyInterface{
    public void method1() {
        System.out.println("MyclassA-method1() 실행");
    }
}
```

MyClassB.java(**새로운 구현 클래스**)

```java
package default_method_necessity;

public class MyClassB implements MyInterface {
    public void method1() {
        System.out.println("MyClassB-method1() 실행");
    }

    // 디폴트 메소드 재정의
    public void method2() {
        System.out.println("MyClassB-method2() 실행");
    }
}
```

DefaultMethodExample.java(**디폴트 메소드 사용**)

```java
package default_method_necessity;

public class DefaultMethodExample {
    public static void main(String[] args) {
        MyInterface mi1 = new MyclassA();
        mi1.method1();
        mi1.method2();

        MyInterface mi2 = new MyClassB();
        mi2.method1();
        mi2.method2();
    }
}
```

실행 결과

```
MyclassA-method1() 실행
MyInterface-method2 실행
MyClassB-method1() 실행
MyClassB-method2() 실행
```

8.7.2 디폴트 메소드가 있는 인터페이스 상속

- 부모 인터페이스에 디폴트 메소드가 정의되어 있을 경우, 자식 인터페이스에서 디폴트 메소드를 활용하는 방법 세가지
  1. **디폴트 메소드를 단순히 상속만 받는다.**
  2. **디폴트 메소드를 재정의(Override)해서 실행 내용을 변경한다.**
  3. **디폴트 메소드를 추상 메소드로 재선언한다.**

- **예제1**

  ParentInterface.java(**부모 인터페이스**)

```java
package default_method_interface_inherit;

public interface ParentInterface {
    public void method1();

    // 디폴트 메소드
    public default void method2() {
        // 실행문
    }
}
```

  ChildInterface1.java(**자식 인터페이스**)

```java
package default_method_interface_inherit;

public interface ChildInterface1 extends ParentInterface {
    // ParentInterface를 상속받고 추상 메소드 method3() 선언
    public void method3();
}
```

  main 예제

```
ChildInterface ci1 = new ChildInterface1() {
    // method1()과 method3()의 실체 메소드를 가지고 있어야 한다.
    public void method1(){
        // 실행문
    }
    public void method3(){
        // 실행문
    }
};

ci1.method1();  // ChildInterface1의 method1() 호출
ci1.method2();  // ParentInterface의 method2() 호출
ci1.method3();  // ChildInterface1의 method3() 호출
```

- **예제2**

  ChildInterface2.java(**자식 인터페이스**)

```
package default_method_interface_inherit;

public interface ChildInterface2 extends ParentInterface{
    // ParentInterface의 디폴트 메소드 method2()를 재정의함
    public default void method2() {
        // 실행문
    }
    public void method3();
}
```

  main 예제

```
ChildInterface2 ci2 = new ChildInterface2() {
    public void method1(){
        // 실행문
    }
    public void method3(){
        // 실행문
    }
};

ci2.method1();  // ChildInterface2의 method1() 실행
ci2.method2();  // ChildInterface2의 재정의된 method2() 호출
ci2.method3();  // ChildInterface2의 method3() 실행
```

- **예제3**

  ChildInterface3.java(**자식 인터페이스**)

```java
package default_method_interface_inherit;

public interface ChildInterface3 extends ParentInterface {
    public void method2();  // 추상 메소드 재선언
    public void method3();
}
```

main 예제

```java
ChildInterface ci3 = new ChildInterface3() {
    public void method1() {
        // 실행문
    }
    public void method2() {
        // 실행문
    }
    public void method3() {
        // 실행문
    }
};

ci3.method1();  // ChildInterface3 구현 객체의 method1() 호출
ci3.method2();  // ChildInterface3 구현 객체의 method2() 호출
ci3.method3();  // ChildInterface3 구현 객체의 method3() 호출
```

확인문제

1. 인터페이스에 대한 설명으로 틀린 것은 무엇입니까?
    1. 인터페이스는 객체 사용 설명서 역할을 한다.
    2. 구현 클래스가 인터페이스의 추상 메소드에 대한 실체 메소드를 가지고 있지 않으면 추상 클래스가 된다.
    3. **인터페이스는 인스턴스 필드를 가질 수 있다. (X, 가질 수 없다.)**
    4. 구현 객체는 인터페이스 타입으로 자동 변환된다.

2. 인터페이스의 다형성과 거리가 먼 것은?
    1. 필드가 인터페이스 타입일 경우 다양한 구현 객체를 대입할 수 있다.
    2. 매개 변수가 인터페이스 타입일 경우 다양한 구현 객체를 대입할 수 있다.
    3. **배열이 인터페이스 타입일 경우 다양한 구현 객체를 저장할 수 있다. (X, 배열 저장은 다형성과 거리가 멀다.)**
    4. 구현 객체를 인터페이스 타입으로 변환하려면 강제 타입 변환을 해야 한다.

3. 다음은 Soundable 인터페이스입니다. sound() 추상 메소드는 객체의 소리를 리턴합니다.

```
public interface Soundable {
    String sound();
}
```

SoundableExample 클래스에서 printSound() 메소드는 Soundable 인터페이스 타입의 매개 변수를 가지고 있습니다. main() 메소드에서 printSound()를 호출할 때 Cat과 Dog 객체를 주고 실행하면 각각 "야옹"과 "멍멍"이 출력되도록 Cat과 Dog 클래스를 작성해보세요.

```
public class SoundableExample {
    private static void printSound(Soundable soundable) {
        System.out.println(soundable.sound());
    }

    public static void main(String[] args) {
        printSound(new Cat());
        printSound(new Dog());
    }
}
```

코드

```
class Cat implements Soundable {
    public String sound() {
        return "야옹";
    }
}

class Dog implements Soundable {
    public String sound() {
        return "멍멍";
    }
}
```

실행 결과

```
야옹
멍멍
```

4. DaoExample 클래스의 main() 메소드에서 dbWork() 메소드를 호출할 때 OracleDao와 MySqlDao 객체를 매개값으로 주고 호출했습니다. dbWork() 메소드는 두 객체를 모두 매개값으로 받기 위해 DataAccessObject 타입의 매개 변수를 가지고 있습니다. 실행 결과를 보고 DataAccessObject 인터페이스와 OracleDao.MysqlDao 구현 클래스를 각각 작성해보세요.

```
public class DaoExample {
    public static void dbWork(DataAccessObject dao) {
        dao.select();
        dao.insert();
        dao.update();
        dao.delete();
    }

    public static void main(String[] args) {
        dbWork(new OracleDao());
        dbWork(new MySqlDao());
    }
}
```

코드

```
interface DataAccessObject {
    void select();
    void insert();
    void update();
    void delete();
}

class OracleDao implements DataAccessObject {
    public void select() {
        System.out.println("Oracle DB에서 검색");
    }
    public void insert() {
        System.out.println("Oracle DB에 삽입");
    }
    public void update() {
        System.out.println("Oracle DB를 수정");
    }
    public void delete() {
        System.out.println("Oracle DB에서 삭제");
    }
}

class MySqlDao implements DataAccessObject {
    public void select() {
        System.out.println("MySql DB에서 검색");
    }
    public void insert() {
        System.out.println("MySql DB에 삽입");
    }
    public void update() {
        System.out.println("MySql DB를 수정");
    }
    public void delete() {
        System.out.println("MySql DB에서 삭제");
    }
}
```

> 구현 클래스에서 메소드 앞에 public을 붙이는 이유는?

실행 결과

```
Oracle DB에서 검색
Oracle DB에 삽입
Oracle DB를 수정
Oracle DB에서 삭제
MySql DB에서 검색
MySql DB에 삽입
MySql DB를 수정
MySql DB에서 삭제
```

5. 다음은 Action 인터페이스입니다. work() 추상 메소드는 객체의 작업을 시작시킵니다.

```
public interface Action {
    void work();
}
```

ActionExample 클래스의 main() 메소드에서 Action의 익명 구현 객체를 만들어 다음과 같은 실행 결과가 나올 수 있도록 박스 안에 들어갈 코드를 작성해보세요.

```
public class ActionExample {
    void work();
}
```

코드

```
public class ActionExample {
    public static void main(String[] args) {
        Action action = new Action() {
            @Override
            public void work() {
                System.out.println("복사를 합니다.");
            }
        };
        action.work();
    }
}
```

실행 결과

```
복사를 합니다.
```