
8장. 인터페이스

한림대학교 소프트웨어융합대학 신미영.

8장. 인터페이스

- 안녕하세요? 여러분!
- 오늘은 자바의 인터페이스 단원을 학습 합니다.
- 이번 장에서는
 - 인터페이스의 개념과 사용법에 대해 알아보도록 하겠습니다.
 - 자바에서는 클래스, 상속, 인터페이스가 가장 중요한 부분입니다.
- 지난 시간에 학습한 내용을 리뷰한 후 학습을 시작하도록 하겠습니다.

지난 시간 Review

- 상속 개념
- 클래스 상속(extends)
- 부모 생성자 호출(super(...))
- 메소드 재정의(Override)
- final 클래스와 final 메소드
- protected 접근 제한자
- 타입 변환과 다형성(polymorphism)
- 추상 클래스(abstract Class)

학습 목차

- 1절. 인터페이스의 역할
- 2절. 인터페이스 선언
- 3절. 인터페이스 구현
- 4절. 인터페이스 사용
- 5절. 타입변환과 다형성
- 6절. 인터페이스 상속
- 7절. 디폴트 메소드와 인터페이스 확장
- 8절. 랴다식
- 9절. 중첩 클래스와 중첩 인터페이스
- 10절. 익명 객체

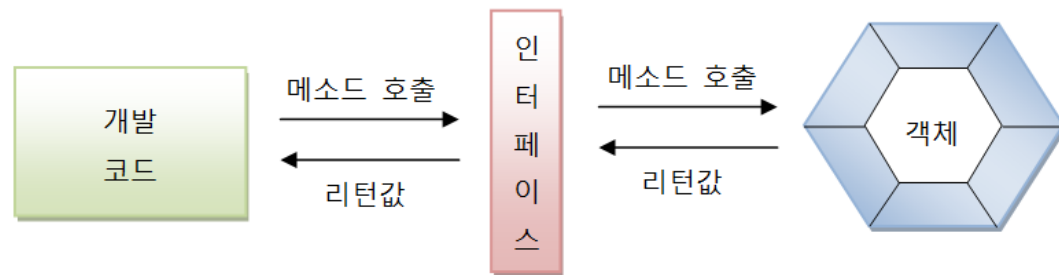
학습 목표

- 인터페이스의 의미와 사용법을 안다.
- 인터페이스를 이용해서 다형성을 구현할 수 있다.
- 인터페이스의 다형성을 이용해 다양한 객체를 일관된 방법으로 사용할 수 있다.
- 람다식을 이해하고 활용할 수 있다
- 익명 객체를 이해하고 활용할 수 있다
- 중첩 클래스와 중첩 인터페이스의 개념을 이해하고 활용할 수 있다

인터페이스의 역할

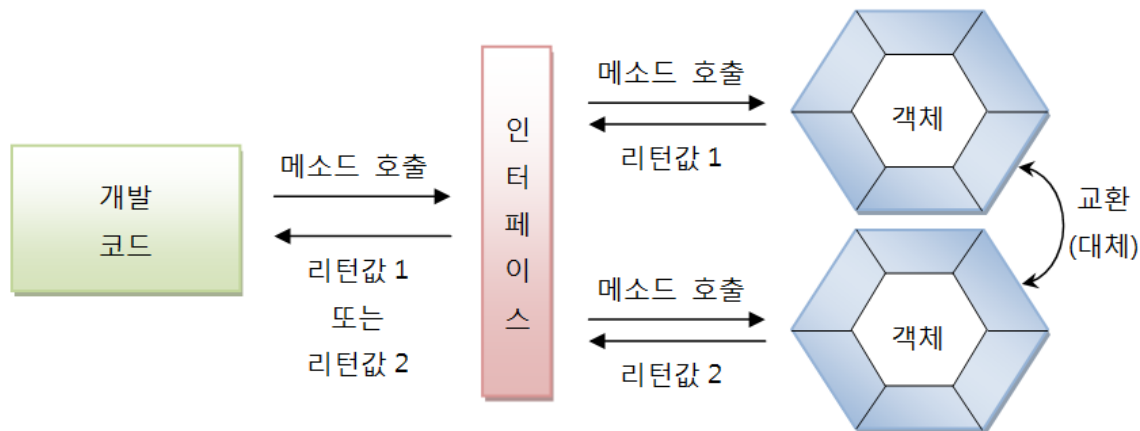
■ 인터페이스란?

- 개발 코드와 객체가 서로 통신하는 접점



■ 인터페이스의 역할

- 인터페이스의 메소드를 호출하면 객체의 메소드가 호출된다.
- 개발 코드가 객체에 종속되지 않게 함-> 개발 코드를 수정하지 않으면서 객체의 교환이 가능
- 개발 코드 변경 없이 리턴값 또는 실행 내용이 다양해 질 수 있음 (다형성)



인터페이스 선언

- 인터페이스 선언

```
[ public ] interface 인터페이스명 { ... }
```

```
interface 인터페이스명 {  
    //상수  
    타입 상수명 = 값;  
    //추상 메소드  
    타입 메소드명(매개변수,...);  
    //디폴트 메소드  
    default 타입 메소드명(매개변수,...) {...}  
    //정적 메소드  
    static 타입 메소드명(매개변수) {...}  
}
```

- 인터페이스 이름

- 자바 식별자 작성 규칙에 따라 작성
- 인터페이스 이름과 대소문자가 동일한 소스 파일 생성 : ~.java 형태

- 인터페이스는 객체를 생성할 수 없으므로 생성자를 가질 수 없다.

상수 필드 선언(constant field)

- 인터페이스는 데이터를 저장할 인스턴스 혹은 정적 필드 선언 불가
- 상수 필드만 선언 가능
 - 인터페이스에 선언된 필드는 모두 public static final
 - 자동적으로 컴파일 과정에서 붙음

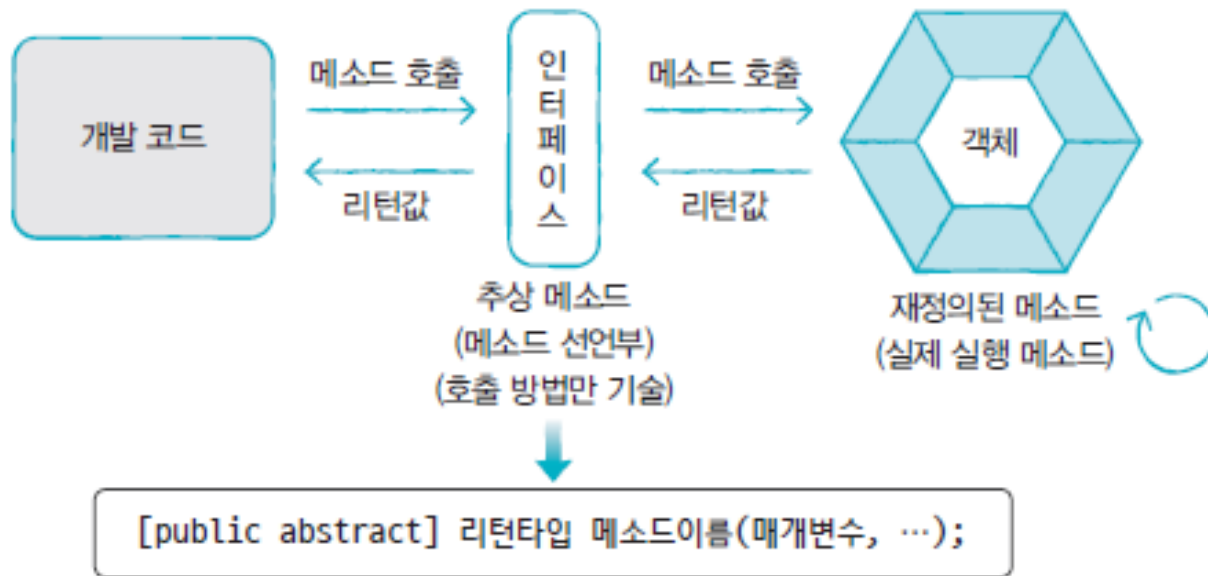
```
[public static final] 타입 상수이름 = 값;
```

- 상수명은 대문자로 작성
- 서로 다른 단어로 구성되어 있을 경우에는 언더스코어(_)로 연결
- 선언과 동시에 초기값 지정
- static { } 블록 작성 불가 - static {} 으로 초기화 불가

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


abstract 메소드 선언

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



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

default 메소드 선언

- 자바8에서 추가된 인터페이스의 새로운 멤버
 - 실행 블록을 가지고 있는 메소드

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

- 모든 구현 객체가 가지고 있는 기본 메소드로 사용 가능
 - 필요에 따라 구현 클래스가 디폴트 메소드 재정의해 사용
- default 키워드를 반드시 붙여야 함
- 기본적으로 public 접근 제한
 - 생략하더라도 컴파일 과정에서 자동 붙음
- 인터페이스만으로는 사용 불가
 - 구현 객체가 인터페이스에 대입되어야 호출할 수 있는 인스턴스 메소드

정적 메소드 선언

- 자바8에서 추가된 인터페이스의 새로운 멤버
- 정적 메소드 사용
 - 인터페이스로 바로 호출 가능

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

```
public interface RemoteControl {  
    static void changeBattery() {  
        System.out.println("건전지를 교환합니다.");  
    }  
}
```

인터페이스 구현

- 구현 (implement) 클래스
 - 인터페이스에서 정의된 추상 메소드를 재정의해서 실행 내용을 가지고 있는 클래스

```
public class 구현클래스이름 implements 인터페이스이름 {  
    //인터페이스에 선언된 추상 메소드의 실제 메소드 선언  
}
```

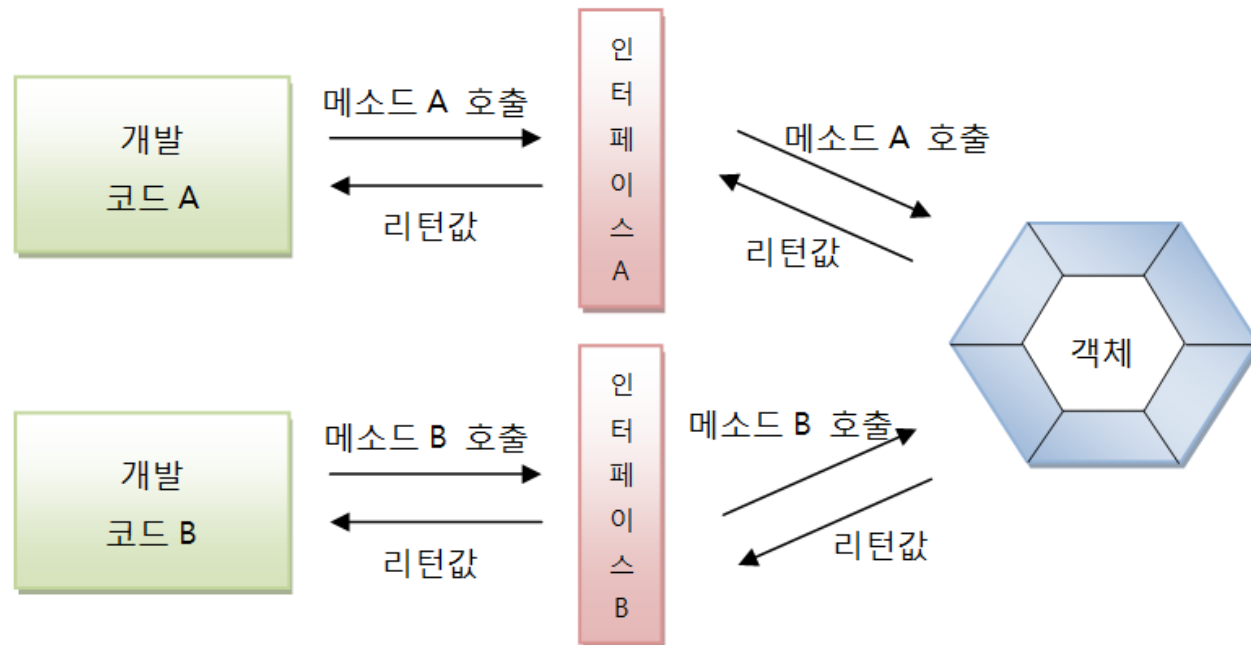
```
public interface RemoteControl {  
    //추상 메소드  
    public void turnOn();  
    public void turnOff();  
    public void setVolume(int volume);  
}
```

```
public class Television implements RemoteControl {  
    //turnOn() 추상 메소드의 실제 메소드  
    public void turnOn() {  
        System.out.println("TV를 켭니다.");  
    }  
    //turnOff() 추상 메소드의 실제 메소드  
    public void turnOff() {  
        System.out.println("TV를 끕니다.");  
    }  
}
```

- 추상 메소드의 실제 메소드를 작성하는 방법
 - 클래스 선언부에 implements 키워드 추가하고 인터페이스 이름 명시
 - 메소드의 선언부가 정확히 일치해야
 - 인터페이스의 모든 추상 메소드를 재정의하는 실제 메소드 작성해야 함
 - 일부만 재정의할 경우, 추상 클래스로 선언 + abstract 키워드 붙임

다중 인터페이스 구현 클래스

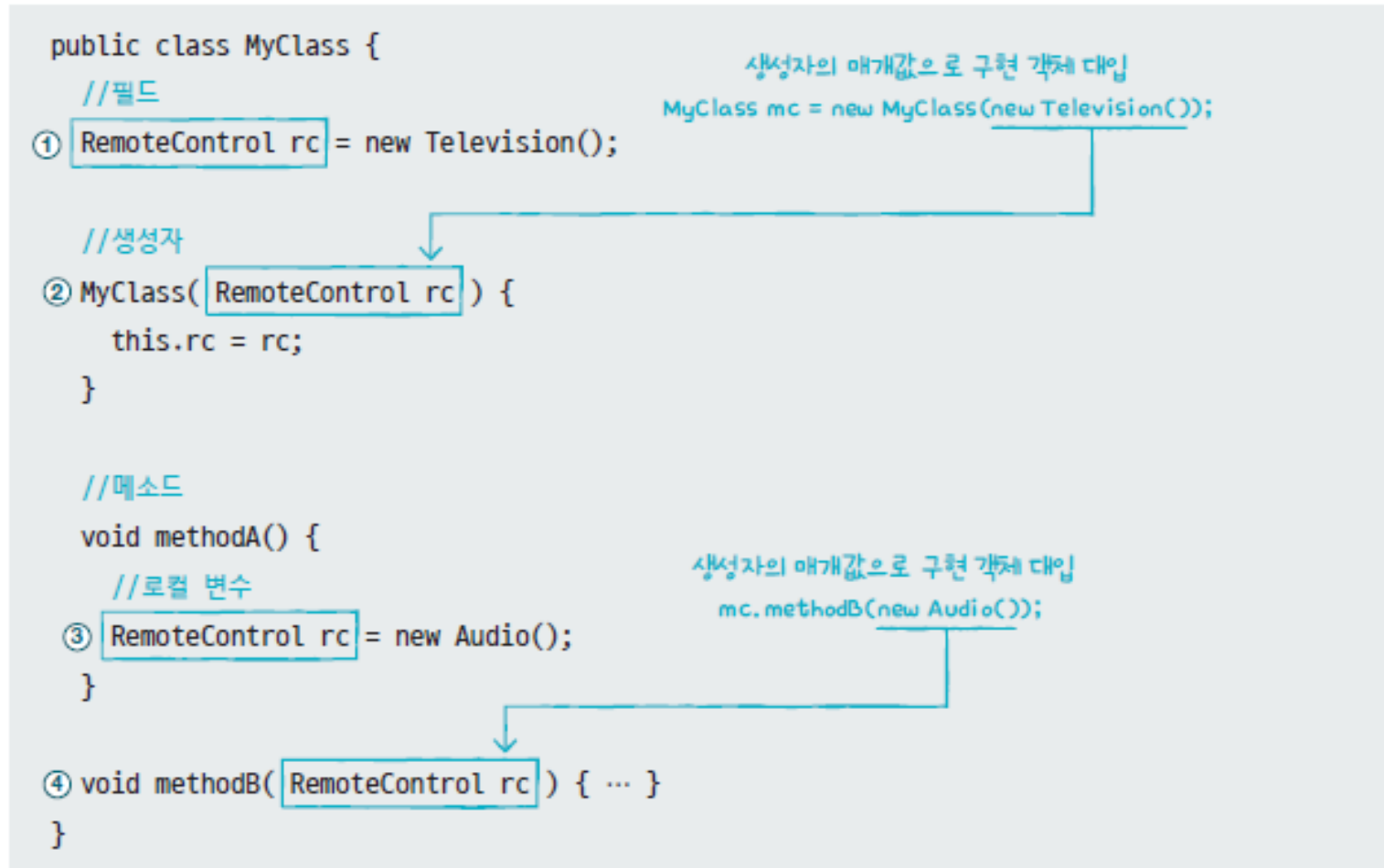
- 객체는 다수의 인터페이스 타입으로 사용 가능



```
public class 구현클래스명 implements 인터페이스 A, 인터페이스 B {  
    //인터페이스 A 에 선언된 추상 메소드의 실제 메소드 선언  
    //인터페이스 B 에 선언된 추상 메소드의 실제 메소드 선언  
}
```

인터페이스 사용

- 인터페이스는 필드, 매개 변수, 로컬 변수로 선언 가능



인터페이스 사용 예

```
RemoteControl rc = new Television();  
rc.turnOn();    → Television 의 turnOn() 실행  
rc.turnOff();   → Television 의 turnOff() 실행
```

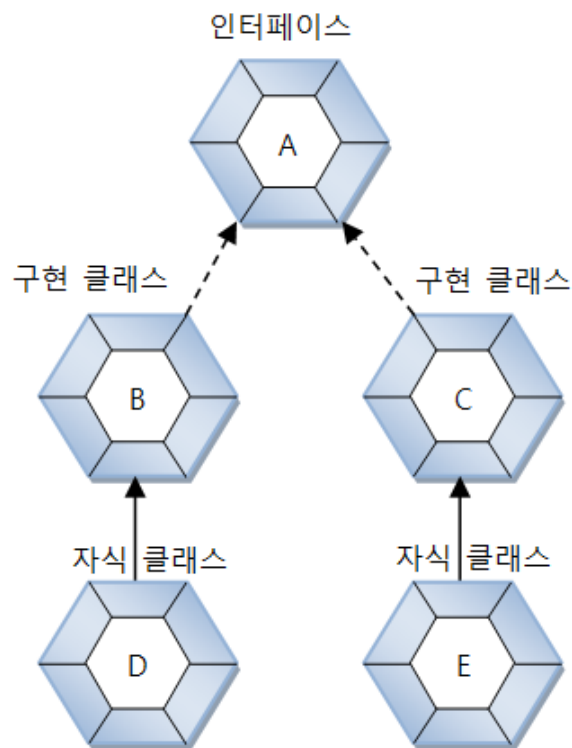
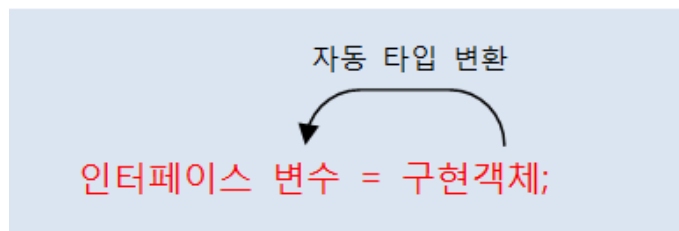


타입 변환과 다형성(Polymorphism)

- 다형성
 - 하나의 타입에 여러 가지 객체를 대입해 다양한 실행 결과를 얻는 것
- 다형성을 구현하는 기술
 - 상속 또는 인터페이스의 자동 타입 변환(Promotion)
 - 오버라이딩(Overriding)
- 다형성의 효과
 - 다양한 실행 결과를 얻을 수 있음
 - 객체를 부품화 시킬 수 있어 유지보수 용이 (메소드의 매개변수로 사용)

자동 타입 변환(Promotion)

- 구현 객체와 자식 객체는 인터페이스 타입으로 자동 타입 변환 가능



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



```
A a1 = b; (가능)  
A a2 = c; (가능)  
A a3 = d; (가능)  
A a4 = e; (가능)
```

필드의 다형성

[다형성은 객체를 부품화시킨다]



```
public interface Tire {  
    public void roll();  
}
```

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

```
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();  
    }  
}
```

```
Car myCar = new Car();  
myCar.frontLeftTire = new KumhoTire();  
myCar.frontRightTire = new KumhoTire();
```

```
myCar.run();
```

인터페이스 배열로 구현한 객체 관리

```
public interface Tire {  
    public void roll();  
}
```

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

```
Tire[] tires = {  
    new HankookTire(),  
    new HankookTire(),  
    new HankookTire(),  
    new HankookTire()  
};
```

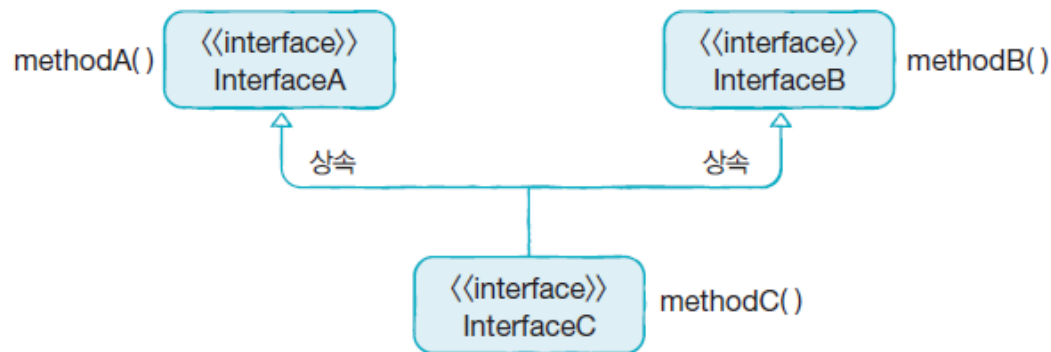
```
tires[1] = new KumhoTire();
```

```
void run() {  
    for(Tire tire : tires) {  
        tire.roll();  
    }  
}
```

인터페이스 상속

- 인터페이스는 다중 상속을 할 수 있다.

```
public interface 하위인터페이스 extends 상위인터페이스 1, 상위인터페이스 2 { ... }
```



```
public interface InterfaceC extends InterfaceA, InterfaceB {
```

```
public class ImplementationC implements InterfaceC {  
    ImplementationC impl = new ImplementationC();  
    InterfaceA ia = impl;  
    InterfaceB ib = impl;  
    InterfaceC ic = impl;
```

- 인터페이스 자동 타입 변환
- 하위 인터페이스의 구현 클래스는 상위 인터페이스의 모든 추상 메소드를 재정의해야 함

인터페이스 상속

```
public interface InterfaceA {      public void methodA(); }
public interface InterfaceB {      public void methodB(); }
public interface InterfaceC extends InterfaceA, InterfaceB { public void methodC(); }
public class ImplementationC implements InterfaceC {
    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"); }}

public class Example {
    public static void main(String[] args) {
        ImplementationC impl = new ImplementationC();
        InterfaceA ia = impl;      ia.methodA();
        InterfaceB ib = impl;      ib.methodB();
        InterfaceC ic = impl;      ic.methodA();      ic.methodB();      ic.methodC();
    }
}
```

다중 상속

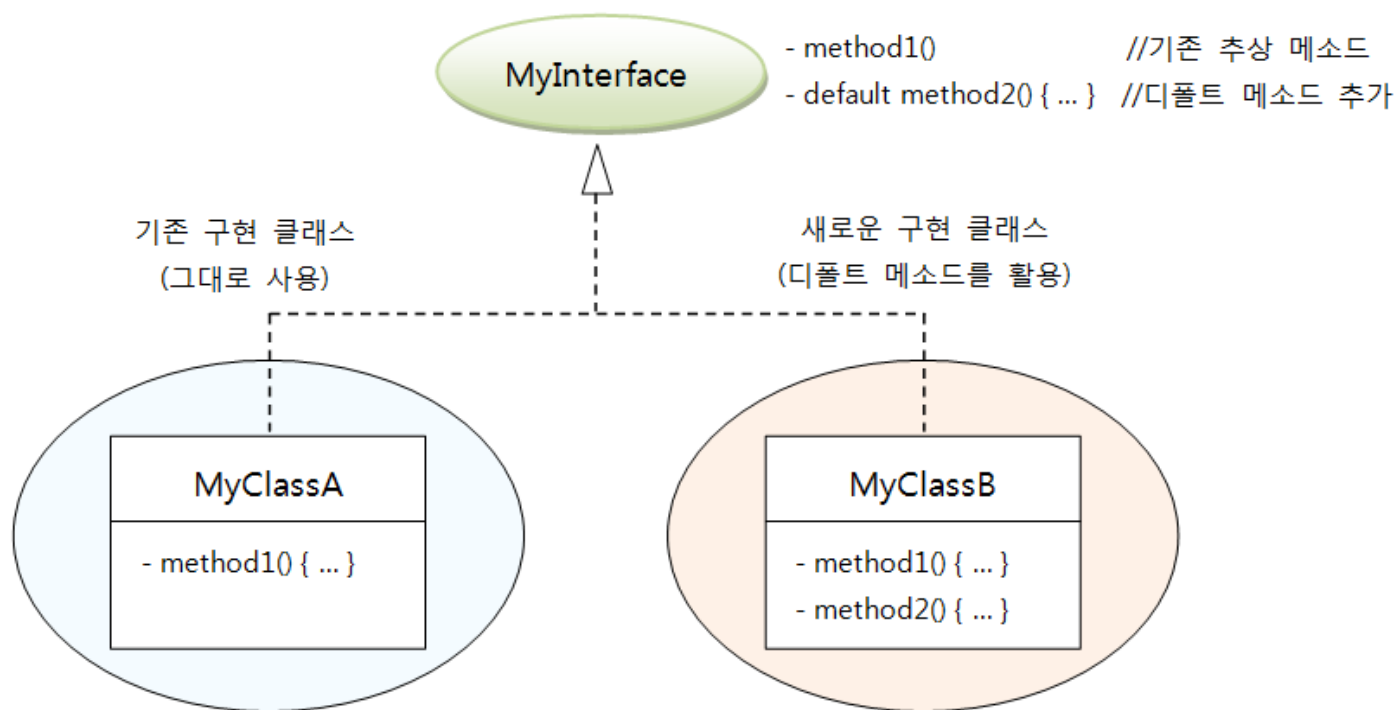
- 인터페이스를 이용하면 다중 상속의 효과를 낼 수 있다.

```
class Shape {  
    protected int x, y;  
    Shape(int x, int y){ this.x=x; this.y=y; }  
}  
  
interface Drawable { void draw(); }  
  
class Rectangle extends Shape implements Drawable {  
    Rectangle(int x, int y){ super(x,y); }  
    public void draw() {  
        System.out.println("중심 좌표 (" + x + ", " + y + ")");  
        System.out.println("Rectangle Draw");  
    }  
}
```

```
Drawable rec=new Rectangle(34,20);  
rec.draw();
```

디폴트 메소드와 인터페이스 확장

- 디폴트 메소드가 있는 인터페이스를 상속했을 경우 자식 인터페이스에서 활용하는 방법
 - 부모 인터페이스의 디폴트 메소드를 단순히 상속만 받을 수도 있음
 - 디폴트 메소드를 재정의(Override)해서 실행 내용 변경도 가능
 - 디폴트 메소드를 추상 메소드로 재 선언도 가능



활용 1. 인터페이스 구현

- 제시된 실행 결과를 보고 DataAccessObject 인터페이스와 OracleDB와 MySQLDB 구현 클래스를 작성하시오

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

```
public class DaoExample {  
    public static void dbWork(DataAccessObject db) {  
        db.select();  
        db.insert();  
        db.update();  
        db.delete();  
    }  
    public static void main(String[] args) {  
        dbWork(new OracleDB("Oracle DB"));  
        dbWork(new MySQLDB("MySQL DB"));  
    }  
}
```


활용 2. 인터페이스와 다형성(1/2)

- 제시된 두 개의 클래스가 공통으로 갖는 필드와 메소드를 인터페이스와 다형성을 사용하여 도형 넓이를 계산하는 프로그램으로 완성 하시오

```
class Circle {
    static final double PI = 3.14;
    double r;

    public Circle(double r) {
        this.r=r;
    }

    double area() {
        return r * r * PI;
    }

    void write() {
        System.out.print("Circle [radius= " + r);
        System.out.printf("]area = %.2f ]\n", area());
    }
}
```

```
class Rectangle {
    private double d1, d2;

    public Rectangle(double d1, double d2) {
        this.d1=d1;
        this.d2=d2;
    }

    double area() {
        return d1 * d2;
    }

    void write() {
        System.out.print("Rectangle [d1 = " + d1 + " d2= " + d2);
        System.out.printf("]area = %.2f ]\n", area());
    }
}
```

활용 2. 인터페이스와 다형성(2/2)

```
public class Interface_test {  
    public static void main(String[] args) {  
        Scanner in = new Scanner(System.in);  
        Shape obj = null;  
        System.out.print("1. 원넓이 2. 사각형넓이 >> ");  
        int choice = in.nextInt();
```

```
        System.out.println("프로그램 종료");
```

```
    }  
}
```

```
1. 원넓이 2. 사각형넓이 >> 1  
반지름 입력 >> 2.4  
Circle [radius= 2.4      area = 18.09 ]  
프로그램 종료
```

```
1. 원넓이 2. 사각형넓이 >> 2  
가로와 세로 값을 순차적으로 입력 >> 2.3 5.2  
Rectangle [ga = 2.3      se= 5.2      area = 11.96 ]  
프로그램 종료
```

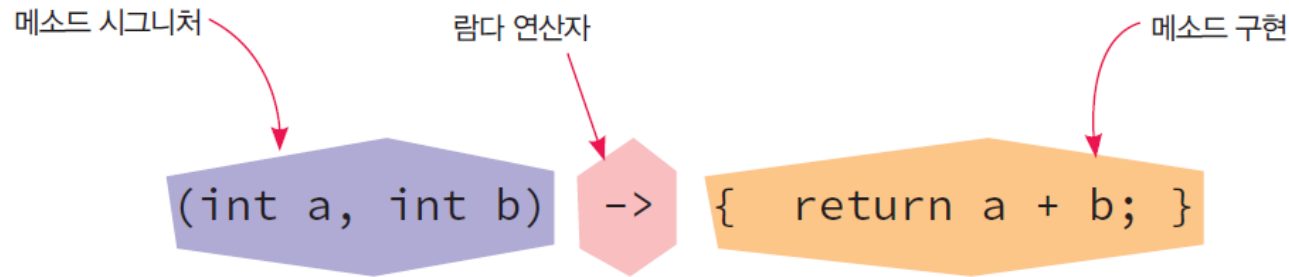
람다식이란

- 자바 8부터 함수적 프로그래밍 위해 람다식 지원
 - 익명 함수(anonymous function)을 생성하기 위한 식
- 자바에서 람다식을 수용한 이유
 - 코드가 매우 간결 해진다.
 - 컬렉션 요소(대용량 데이터)를 필터링 또는 매핑해 쉽게 집계
- 자바는 람다식을 함수적 인터페이스의 익명 구현 객체로 취급
 - 어떤 인터페이스를 구현 할 지는 대입되는 인터페이스에 달려있음

```
Runnable runnable = () -> { ... }; ● ----- 람다식
```

람다식 기본 문법

- 함수적 스타일의 람다식 작성법



- 매개변수 타입은 런타임 시에 대입 값에 따라 자동 인식 → 생략 가능
- 하나의 매개변수만 있을 경우에는 괄호() 생략 가능
- 하나의 실행문만 있다면 중괄호 { } 생략 가능
- 매개변수 없다면 괄호 () 생략 불가
- 리턴값이 있는 경우, return 문 사용
- 중괄호 { }에 return 문만 있을 경우, 중괄호 생략 가능

타겟 타입과 함수적 인터페이스

- 타겟 타입(target type)
 - 랴다식이 대입되는 인터페이스
 - 익명 구현 객체를 만들 때 사용할 인터페이스

```
인터페이스 변수 = 랴다식;
```

- 함수적 인터페이스(functional interface)
 - 하나의 추상 메소드만 선언된 인터페이스가 타겟 타입
 - @FunctionalInterface 어노테이션
 - 하나의 추상 메소드만을 가지는지 컴파일러가 체크
 - 두 개 이상의 추상 메소드가 선언되어 있으면 컴파일 오류 발생

타겟 타입과 함수적 인터페이스

- 매개변수와 리턴값이 없는 람다식
 - method()가 매개 변수를 가지지 않는 경우

```
@FunctionalInterface
public interface MyFunctionalInterface {
    public void method();
}
```

```
MyFunctionalInterface fi = () -> { ... }
```

```
fi.method();
```

- 매개변수가 있는 람다식

```
@FunctionalInterface
public interface MyFunctionalInterface {
    public void method(int x);
}
```

```
MyFunctionalInterface fi = (x) -> { ... } 또는 x -> { ... }
```

```
fi.method(5);
```

타겟 타입과 함수적 인터페이스

- 리턴값이 있는 람다식

```
@FunctionalInterface
public interface MyFunctionalInterface {
    public int method(int x, int y);
}
```

```
MyFunctionalInterface fi = (x, y) -> { ...; return 값; }
```

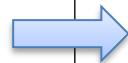
```
int result = fi.method(2, 5);
```

```
MyFunctionalInterface fi=(x, y) -> {
    return x+y;
}
```



```
MyFunctionalInterface fi = (x,y) -> x+y;
```

```
MyFunctionalInterface fi=(x, y) -> {
    return sum(x,y);
}
```



```
MyFunctionalInterface fi = (x,y) -> sum(x, y);
```

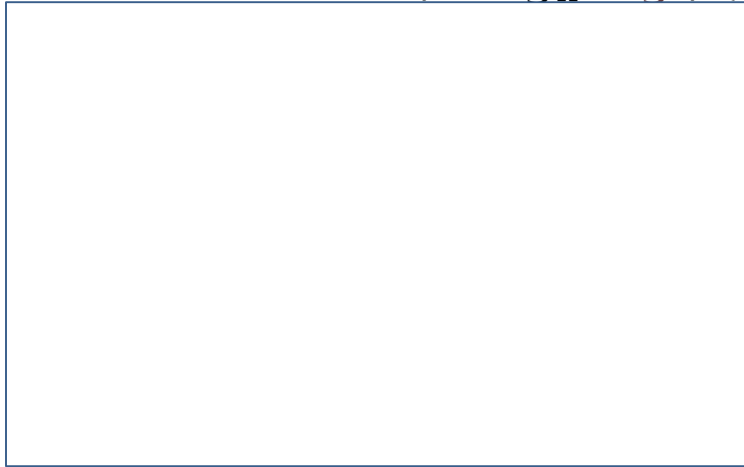
활용 3. 람다식

- 제시된 인터페이스를 람다식으로 구현한 후 테스트하는 프로그램으로 완성하시오

```
@FunctionalInterface
interface Func1{
    int calc(int a, int b);
}
```

```
@FunctionalInterface
interface Func2{
    void sayHello();
}
```

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




    }
}
```

중첩 클래스와 중첩 인터페이스

- 중첩 클래스: 클래스 멤버로 선언된 클래스

```
class ClassName{  
    class NestedClassName{ //중첩 클래스  
    }  
}
```

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

```
class ClassName{  
    interface NestedInterfaceName{ //중첩 인터페이스  
    }  
}
```

중첩 클래스

- 중첩 클래스의 분류

| 선언 위치에 따른 분류 | | 선언 위치 | 설명 |
|--------------|----------------|--|---------------------------------------|
| 멤버 클래스 | 인스턴스 멤버 클래스 | <pre>class A { class B { ... } }</pre> | A 객체를 생성해야만 사용할 수 있는 B 중첩 클래스 |
| | 정적 멤버 클래스 | <pre>class A { static class B { ... } }</pre> | A 클래스로 바로 접근할 수 있는 B 중첩 클래스 |
| 로컬 클래스 | | <pre>class A { void method() { class B { ... } } }</pre> | method()가 실행할 때만 사용할 수 있는 B 중첩 클래스 |

- 클래스 생성시 바이트 코드 파일(*.class) 별도 생성

중첩 클래스 - 인스턴스 멤버 클래스

```
class Out{
    private int outfield;
    Out(){
        System.out.println("Out 객체 생성");
    }
    class In{
        int infield;
        // static int sinfield; ---- 정적 필드 불가
        // static void inmethod() { } ---- 정적 메소드 불가
        In(){
            System.out.println("in 객체 생성");
            outfield = 50;
        }
        void inmethod() {
            System.out.println("inmethod 실행");
            System.out.println("바깥 클래스 outfield : " + outfield);
            System.out.println("안쪽 클래스 infield : " + infield);
        }
    }
}
```

```
public class NestedClass1 {
    public static void main(String[] args) {
        Out out = new Out();
        Out.In in = out.new In();
        in.infield = 3;
        in.inmethod();
    }
}
```

Out 객체 생성
in 객체 생성
inmethod 실행
바깥 클래스 outfield : 50
안쪽 클래스 infield : 3

중첩 클래스 – 정적 멤버 클래스

- static 키워드로 선언된 클래스, 모든 종류의 필드, 메소드 선언 가능

```
class Out{
    private int outfield;
    Out(){
        System.out.println("Out 객체 생성");
    }
    static class In {
        int infield; //인스턴스 필드
        static int sinfield; //정적 필드
        In() {
            System.out.println("in 객체 생성");
            sinfield = 50;
            // outfield= 50; --- non static
        }
        void inmethod() { //인스턴스 메소드
            System.out.println("inmethod 실행");
            System.out.println("안쪽 클래스 infield : " + infield);
        }
        static void smethod() { //정적 메소드
            System.out.println("정적 멤버 클래스 sinfield : " + sinfield);
        }
    }
}
```

```
public class NestedClass2 {
    public static void main(String[] args) {
        Out.In in = new Out.In(); //Out 객체를 생성할 필요 없음
        in.infield = 3;           //인스턴스 필드
        Out.In.smethod();         //클래스 이름으로 정적 메소드 호출
    }
}
```

in 객체 생성
정적 멤버 클래스 sinfield : 50

중첩 클래스 – 로컬 클래스

- 메소드 내에서 선언되는 클래스 – 접근 제한자 사용 불가
- 로컬 클래스는 메소드가 실행될 때 메소드 내에서 객체를 생성하고 사용
- 비동기 처리를 위한 스레드 객체 생성시 사용

```
class Out {  
    Out() {  
        System.out.println("Out 객체 생성");  
    }  
    void method() {  
        class Local { //로컬 클래스  
            int lfield; //인스턴스 필드  
            //static int s; - 정적필드와 정적 메소드 사용 불가  
            Local(){  
                System.out.println("Local 클래스 객체 생성");  
            }  
            void localMethod() {  
                System.out.println("Local 클래스 메소드\n로컬 클래스 인스턴스 필드 => "+lfield);  
            }  
        }  
        Local local=new Local(); //로컬 클래스 객체 생성  
        local.lfield=60; local.localMethod();  
    }  
}
```

```
public class NestedClass3 {  
    public static void main(String[] args) {  
        Out out = new Out();  
        //로컬 클래스 객체 생성을 위한 메소드 호출  
        out.method();  
    }  
}
```

| |
|--------------------------------|
| Out 객체 생성 |
| Local 클래스 객체 생성 |
| Local 클래스 메소드 |
| 로컬 클래스 인스턴스 필드 => 60 |

중첩 클래스의 접근 제한

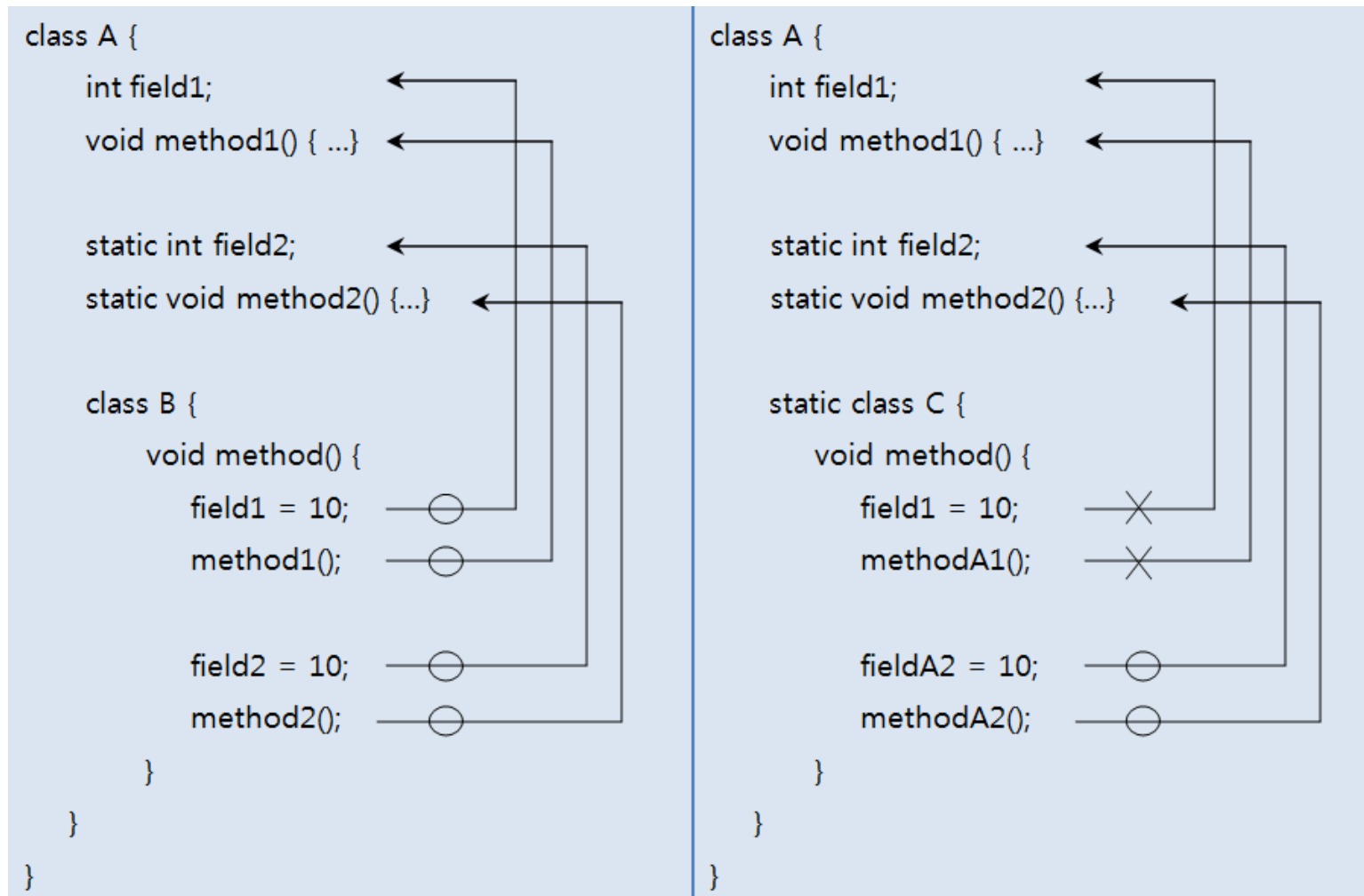
- 바깥 필드와 메소드에서 사용 제한

```
public class A {  
    //인스턴스 멤버 클래스  
    class B {}  
  
    //정적 멤버 클래스  
    static class C {}  
}
```

```
public class A {  
    //인스턴스 필드  
    B field1 = new B();          ----- (o)  
    C field2 = new C();          ----- (o)  
  
    //인스턴스 메소드  
    void method1() {  
        B var1 = new B();        ----- (o)  
        C var2 = new C();        ----- (o)  
    }  
  
    //정적 필드 초기화  
    //static B field3 = new B();   ----- (x)  
    static C field4 = new C();    ----- (o)  
  
    //정적 메소드  
    static void method2() {  
        //B var1 = new B();       ----- (x)  
        C var2 = new C();         ----- (o)  
    }  
}
```

중첩 클래스의 접근 제한

- 멤버 클래스에서 사용 제한



중첩 클래스의 접근 제한

- 로컬 클래스에서 사용되는 메소드의 매개변수나 로컬 변수는 로컬 클래스 내부에 복사
- 매개변수와 로컬 변수가 수정되면 로컬 클래스내부에 복사된 값과 달라짐
- 이러한 문제를 해결하기 위해 값 변경 불가

```
public class Outer {  
    //자바7 이전  
    public void method1(final int arg) {  
        final int localVariable = 1;  
        //arg = 100; (x)  
        //localVariable = 100; (x)  
        class Inner {  
            public void method() {  
                int result = arg + localVariable;  
            }  
        }  
    }  
}
```

final 매개변수와 로컬 변수에 final 붙이지 않으면 컴파일 오류 발생

```
//자바8 이후  
public void method2(int arg) {  
    int localVariable = 1;  
    //arg = 100; (x)  
    //localVariable = 100; (x)  
    class Inner {  
        public void method() {  
            int result = arg + localVariable;  
        }  
    }  
}
```


중첩 클래스의 접근 제한

- 중첩 클래스에서 바깥 클래스 참조 얻기

바깥클래스.this.필드

바깥클래스.this.메소드();

```
public class Outer {  
    String field = "Outer-field";  
    void method() {  
        System.out.println("Outer-method");  
    }  
    class Nested {  
        String field = "Nested-field";  
        void method() {  
            System.out.println("Nested-method");  
        }  
        void print() {  
            System.out.println(this.field);  
            this.method();  
            System.out.println(Outer.this.field);  
            Outer.this.method();  
        }  
    }  
}
```

● ----- 중첩 객체 참조

● ----- 바깥 객체 참조

중첩 인터페이스

- 중첩 인터페이스
 - 클래스의 멤버로 선언된 인터페이스
 - 해당 클래스와 긴밀한 관계 맺는 구현 클래스 만들기 위함

```
class A {  
    [static] interface I {  
        void method();  
    }  
}
```

← 중첩 인터페이스

- 인스턴스 멤버 인터페이스와 정적 멤버 인터페이스 모두 가능함

중첩 인터페이스

메시지를 보냅니다
전화를 겁니다

```
class Button{
    OnClick onclick;
    void setOnClick(OnClick onclick) { this.onclick = onclick; }
    void touch() { onclick.onClick(); }
    static interface OnClick{ void onClick(); }
}
class Call implements Button.OnClick{
    public void onClick() { System.out.println("전화를 겁니다"); }
}
class Message implements Button.OnClick{
    public void onClick() { System.out.println("메시지를 보냅니다"); }
}
public class Lec_02 {
    public static void main(String[] args) {
        Button btn=new Button();
        btn.setOnClick(new Message());
        btn.touch();
        btn.setOnClick(new Call());
        btn.touch();
    }
}
```

익명 객체

- 익명 객체: 이름이 없는 객체
 - 익명 객체는 단독 생성 불가
 - 클래스 상속하거나 인터페이스 구현해야만 생성 가능
 - 사용 위치
 - 필드의 초기값, 로컬 변수의 초기값, 매개변수의 매개값으로 주로 대입
 - UI 이벤트 처리 객체나, 스레드 객체를 간편하게 생성할 목적으로 주로 활용
 - 사용 형식

```
인터페이스 변수 = new 인터페이스(){  
    .....  
};
```

```
부모 클래스 변수 = new 부모 클래스(){  
    .....  
};
```

- 익명 객체에 새롭게 정의된 필드와 메소드
 - 익명 객체 내부에서만 사용
 - 외부에서는 익명 객체의 필드와 메소드에 접근할 수 없음
 - 이유: 익명 객체는 부모 타입 변수에 대입되므로 부모 타입에 선언된 것만 사용 가능

익명 구현 객체 – 인터페이스 구현

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

```
public class AnonymousClassTest1 {  
    public static void main(String args[]) {  
        // 익명 구현 객체 생성  
        RemoteControl ac = new RemoteControl() {  
            int volume;  
            public void turnOn() {  
                System.out.println("TV turnOn()");  
            }  
            public void turnOff() {  
                System.out.println("TV turnOff()");  
            }  
            public void setVolume(int volume) {  
                this.volume = volume;  
                System.out.println("현재 TV 볼륨: " + this.volume);  
            }  
        }; // ; 반드시 기입  
        ac.turnOn();  
        ac.turnOff();  
        ac.setVolume(5);  
        ac.volume; //error  
    }  
}
```

익명 자식 객체 - 부모 클래스 상속

```
class Person { //부모 클래스
    void wake() { System.out.println("7시에 일어납니다."); }
}
class Ano {
    Person field=new Person() { //익명 자식 객체 생성1 - 필드 초기값으로 대입
        void study() { System.out.println("열심히 자바를 공부합니다"); }
        @Override
        void wake() { System.out.println("8시에 일어납니다."); }
    };
    void method1(Person per) { per.wake(); }
}
public class Lec_02 {
    public static void main(String[] args) {
        Ano ano = new Ano();
        ano.method1(new Person() { //익명 자식 객체 생성2 - 매개변수 값으로 대입
            @Override
            void wake() { System.out.println("6시에 일어납니다."); }
        });
        ano.field.wake();
        ano.field.study(); //error
    }
}
```

활용 4. 중첩 인터페이스

- 제시된 main()을 실행하였을 때 다음과 같이 실행될 수 있도록 프로그램을 완성 하시오

```
class CheckBox{
    OnSelectListener listener;

    void setOnSelectListener(OnSelectListener listener) {
        this.listener = listener;
    }

    void change(String color) {
        listener.onChange(color);
    }
    static interface OnSelectListener{
        void onChange(String color);
    }
}
```

체크 박스 레이블을 blue 색상으로 변경합니다
체크 박스 배경을 green 색상으로 변경합니다
체크 박스 전경색을 red 색상으로 변경합니다

```
public class CheckBoxExample {
    public static void main(String[] args) {
        CheckBox check=new CheckBox();
        check.setOnSelectListener(new TextChange());
        check.change("blue");
        check.setOnSelectListener(new BackgroundChange());
        check.change("green");
        check.setOnSelectListener(new ForegroundChange());
        check.change("red");
    }
}
```

학습 정리

- 인터페이스
 - 객체의 사용 방법 정의한 것
 - 인터페이스의 상수 필드 : 인터페이스의 필드는 기본적으로 public static final 특성 가짐
 - 인터페이스의 메소드는 public abstract이 생략된 메소드 선언부만 있는 추상 메소드이다.
 - 인터페이스는 다중 상속 허용한다.
- implements
 - 구현 클래스에서 어떤 인터페이스를 사용 가능한지 기술하기 위해 사용한다.
 - 인터페이스 사용 : 클래스 선언 시 필드, 매개 변수, 로컬 변수로 선언 가능. 구현 객체를 대입.
 - 자동 타입 변환 : 구현 객체는 인터페이스 변수로 자동 타입 변환된다.
- 다형성
 - 인터페이스도 재정의와 타입 변환 기능을 제공하므로 다형성을 구현할 수 있다.
 - 강제 타입 변환 : 인터페이스에 대입된 구현 객체를 다시 원래 타입으로 변환하는 것을 말한다.
 - instanceof : 객체가 어떤 타입인지 조사할 때 사용한다. 강제 타입 변환 전에 사용.

학습 정리

- 람다식
 - 익명 함수(anonymous function)을 생성하기 위한 식
- 중첩 클래스
 - 클래스 내부에 선언
 - 멤버 클래스
 - 로컬 클래스
- 중첩 인터페이스
 - 클래스 멤버로 선언
- 익명 객체
 - 익명 자식 객체
 - 익명 구현 객체

Q & A

- “인터페이스”에 대한 학습이 모두 끝났습니다.
 - 모든 내용을 이해 하셨나요?
 - 아직 이해가 안되는 내용이 있다면 다시 한번 복습하시기 바랍니다.
 - 질문은 한림 SmartLEAD 쪽지 또는 e-mail 또는 전화상담을 이용하시기 바랍니다.
-
- 퀴즈와 과제가 출제되었습니다. 마감시간에 늦지 않도록 주의해 주세요.
 - 다음주에는 예외 처리에 대하여 알아보도록 하겠습니다.
-
- 수고하셨습니다.^^