

JAVA 프로그래밍

인터페이스

인터페이스(Interface)

- 인터페이스는 클래스가 아니다.
 - **interface** 키워드로 선언한다.
- 인터페이스는 객체와 객체 사이의 상호 작용을 나타낸다.
- **다중 상속의 기능**을 **구현**하려고 할 경우, 인터페이스를 사용한다.
- 인터페이스는 **추상 메서드만** 가질 수 있고, 일반 메서드는 가질 수 없다.
- 인터페이스는 **static final** 형태의 **상수만** 가질 수 있다. 변수는 가질 수 없다.

[인터페이스 정의 형식]

```
Interface 인터페이스명 {  
    public static final 자료형 변수명 = 변수 값;  
    public abstract 반환자료형 함수명(매개변수, 매개변수, ...);  
}
```

인터페이스(Interface)

- 인터페이스는 데이터는 표현할 수 없고, **메서드(함수)만 표현이 가능**하다.
- 이미 클래스 상속을 받고 있더라도 인터페이스의 상속을 받는 것이 가능하다.
- 인터페이스도 추상 클래스처럼 객체를 생성할 수 없다.
 - 다른 클래스에 의해 구현(**implements**)하여 사용한다.
- **추상 메서드는 자식 클래스에서 반드시 구현해야 한다.**

<인터페이스 vs 추상클래스>

- **인터페이스** : 모든 메서드가 추상 메서드이면 인터페이스로 구현, 다중 상속
- **추상클래스** : 여러 개의 메서드 중 일부가 추상 메서드이면 추상 클래스로 구현, 단일 상속

인터페이스의 사용

- 인터페이스의 선언 : **interface** 인터페이스이름
- 인터페이스의 구현 : **class** 클래스이름 **implements** 인터페이스이름

```
public interface 인터페이스_이름 {
```

```
    반환형 추상 메소드1(...);
```

```
    반환형 추상 메소드2(...);
```

```
    ...
```

```
}
```

← 인터페이스 안에는 추상 메소드들이 정의된다.

```
public class 클래스_이름 implements 인터페이스_이름 {
```

```
    반환형 추상 메소드1(...) {
```

```
        ....
```

```
    }
```

```
    반환형 추상 메소드2(...) {
```

```
        ....
```

```
    }
```

```
}
```

← 인터페이스를 구현하는 클래스는 추상 메소드의
몸체를 구현하여야 한다.

인터페이스 만들기

```
public interface Calc {
```

```
    double PI = 3.14;  
    int ERROR = -999999999;
```

인터페이스에서 선언한 변수는 컴파일
과정에서 상수로 변환됨

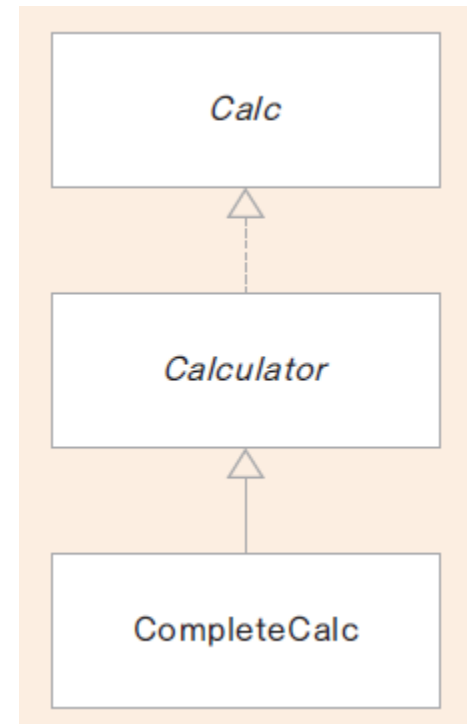
```
    int add(int num1, int num2);  
    int subtract(int num1, int num2);  
    int times(int num1, int num2);  
    int divide(int num1, int num2);
```

인터페이스에서 선언한 메서드는 컴파일
과정에서 추상 메서드로 변환됨

```
}
```

클래스에서 인터페이스 구현하기

- Calc 인터페이스를
Calculator 클래스에서 구현
- Calc의 모든 추상 메서드를 구현하지 않으면
Calculator는 추상 클래스가 됨
- CompleteCalc 클래스가 Calculator를
상속받은 후, 모든 메서드를 구현
- CompleteCalc는 생성 가능한 클래스



*Calculator*와 CompleteCalc 클래스

```
public abstract class Calculator implements Calc { //추상 클래스
    @Override
    public int add(int num1, int num2) {
        return num1 + num2;
    }

    @Override
    public int subtract(int num1, int num2) {
        return num1 - num2;
    }
}
```

인터페이스 구현

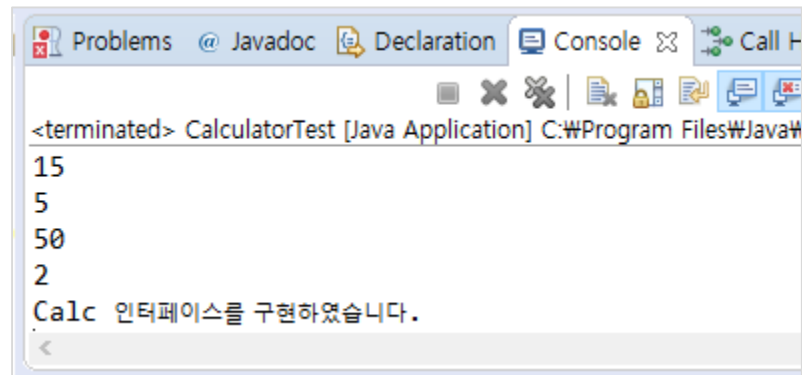
```
public class CompleteCalc extends Calculator {
    @Override
    public int times(int num1, int num2) {
        return num1 * num2;
    }

    @Override
    public int divide(int num1, int num2) {
        if(num2 != 0)
            return num1/num2;
        else
            return ICalc.ERROR; // 나누는 수가 0인 경우에 대해 오류 반환
    }
}
```

클래스 완성

CompleteCalc 클래스 실행하기

```
public class CalculatorTest {  
    public static void main(String[] args) {  
        int num1 = 10;  
        int num2 = 5;  
  
        CompleteCalc calc = new CompleteCalc();  
        System.out.println(calc.add(num1, num2));  
        System.out.println(calc.subtract(num1, num2));  
        System.out.println(calc.times(num1, num2));  
        System.out.println(calc.divide(num1, num2));  
        calc.showInfo();  
    }  
}
```

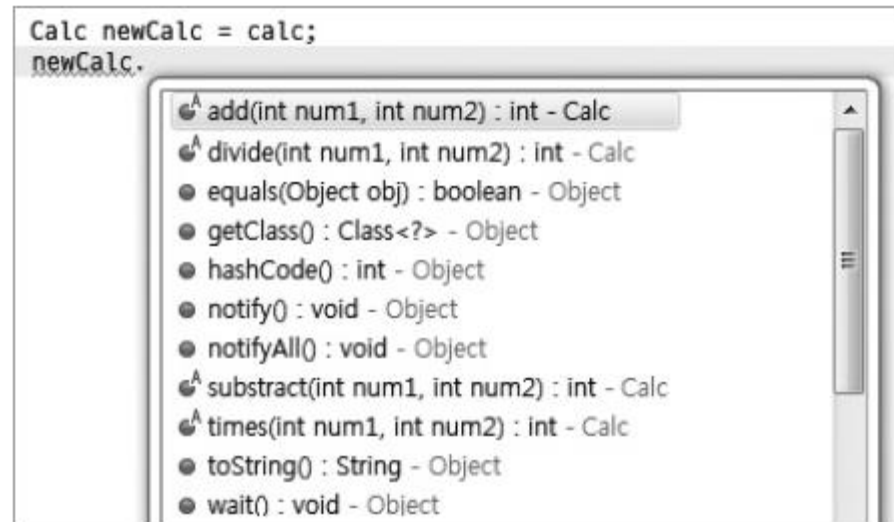


인터페이스 구현과 형 변환

- 인터페이스를 구현한 클래스는 인터페이스 형(타입)으로 선언한 변수로 형 변환할 수 있음
- 상속에서의 형 변환과 동일한 의미
- 단, 클래스 상속과 달리 구현 코드가 없기 때문에 여러 인터페이스를 구현할 수 있음
- 형 변환 시 사용할 수 있는 메서드는 인터페이스에 선언된 메서드만 사용할 수 있음

```
Calc calc = new CompleteCalc( );
```

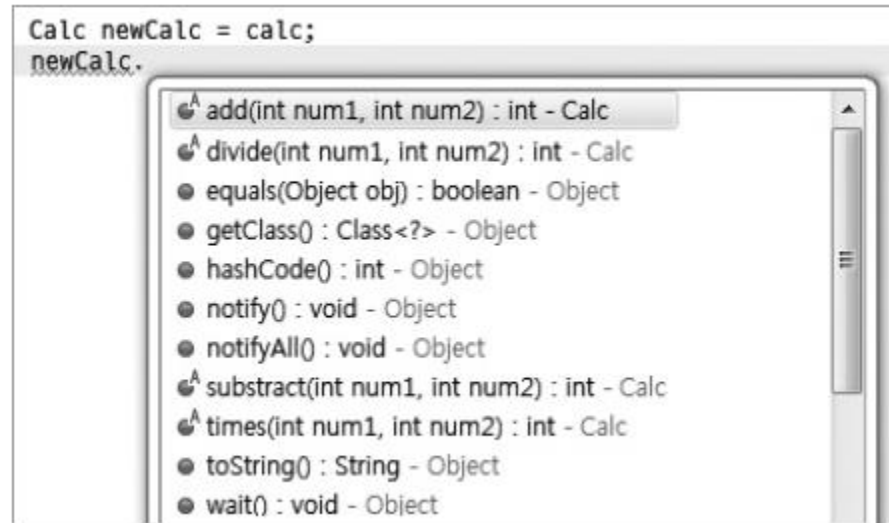
CompleteCalc가 Calc형으로
대입된 경우 사용할 수 있는 메서드



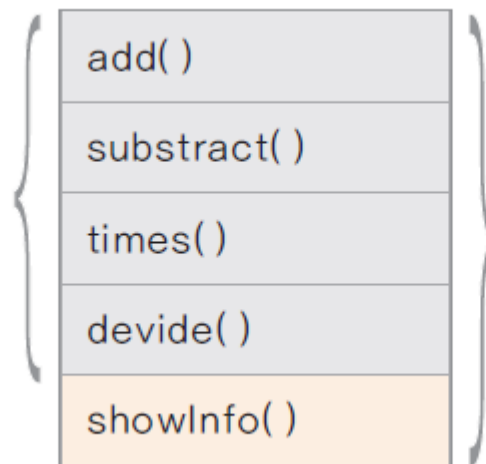
인터페이스 구현과 형 변환

```
Calc calc = new CompleteCalc( );
```

CompleteCalc가 Calc 형으로
대입된 경우 사용할 수 있는 메서드



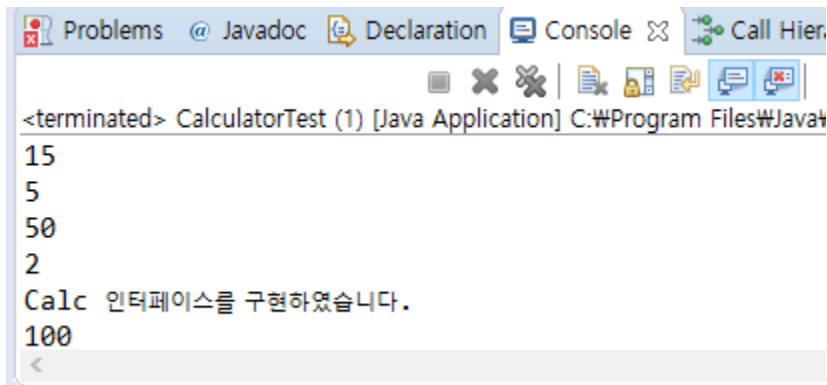
Calc형 변수와 Calculator형
변수에서 사용 가능



CompleteCalc형 변수에서 사용 가능

Quiz 인터페이스

- Calc 인터페이스에 square() 메서드를 추가로 선언
 - **square() 메서드** : 매개변수로 전달된 값의 제곱을 반환하는 메서드
- 인터페이스에 메서드를 추가하고 CompleteCalc에서 구현한 후, CompleteCalcTest 클래스에서 메서드를 호출해 보세요.
- 실행 결과

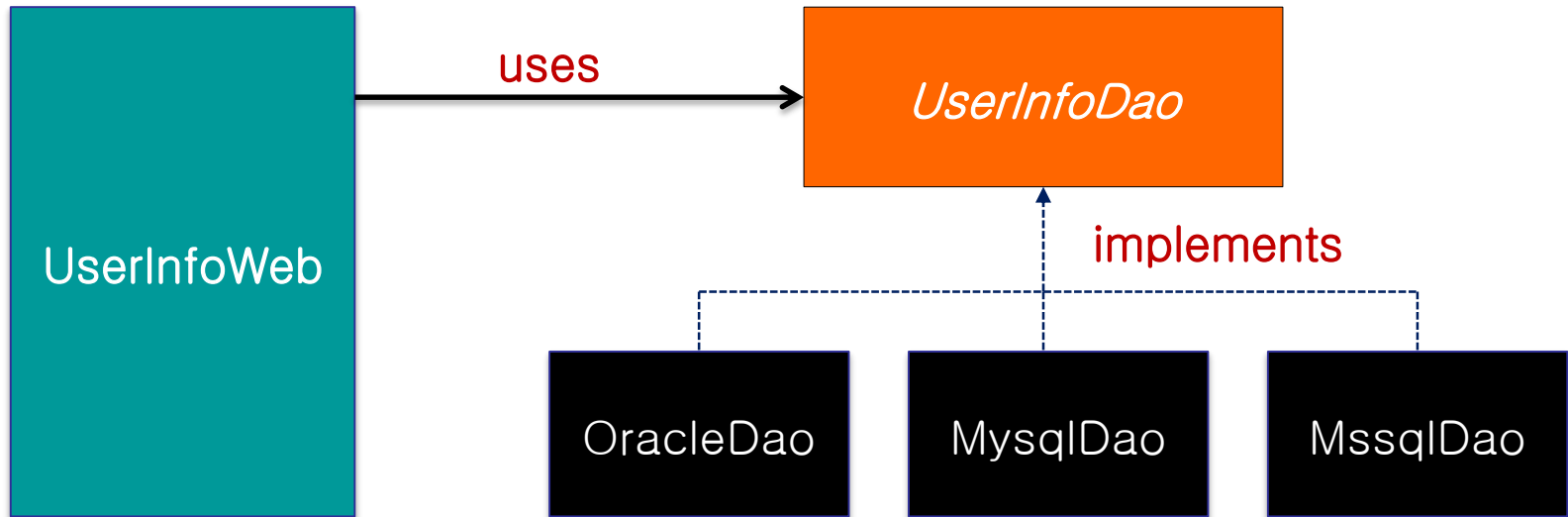


```
<terminated> CalculatorTest (1) [Java Application] C:\Program Files\Java\
15
5
50
2
Calc 인터페이스를 구현하였습니다.
100
<
```

인터페이스와 다형성

- 인터페이스는 “Client Code” 와 서비스를 제공하는 “객체” 사이의 약속이다.
- 어떤 객체가 어떤 **interface 타입**이라 함은 그 interface가 제공하는 메서드를 구현했다는 의미임
- Client는 어떻게 구현되었는지 상관없이 interface의 정의만을 보고 사용할 수 있음 (ex : **JDBC**)
- 다양한 구현이 필요한 인터페이스를 설계하는 일은 매우 중요한 일
 - ※ 인터페이스의 역할은 ?
인터페이스를 구현한 클래스가 어떤 기능의 메서드를 제공하는지를 명시하는 것

왜 인터페이스를 사용하는가?



UserInfoWeb은 *UserInfoDao*에 정의된 메서드 명세만 보고 Dao를 사용할 수 있고, Dao 클래스들은 *UserInfoDao*에 정의된 메서드를 구현할 책임이 있다.

인터페이스의 요소

- 상수 : 모든 변수는 상수로 변환됨
- 추상 메서드 :
모드 메서드는 추상 메서드로 구현코드가 없음
- 디폴트 메서드 :
기본 구현을 가지는 메서드, 구현 클래스에서 재정의할 수 있음
- 정적 메서드 :
인스턴스 생성과 상관 없이 인터페이스 타입으로 사용할 수 있는 메서드
- private 메서드 :
인터페이스를 구현한 클래스에서 사용하거나 재정의할 수 없음
인터페이스 내부에서만 기능을 제공하기 위해 구현하는 메서드

```
public interface Calc {  
    double PI = 3.14;  
    int ERROR = -999999999;  
    ...  
}
```

디폴트 메서드 재정의

- Calc 인터페이스에 디폴트 메서드 정의

```
public interface Calc {  
    ...  
    default void description( ) {  
        System.out.println("정수 계산기를 구현합니다");  
    }  
}
```

- CompleteCalc 에서 재정의하기

```
public class CompleteCalc extends Calculator {  
    ...  
    @Override  
    public void description( ) {  
        // TODO Auto-generated method stub  
        super.description( );  
    }  
}
```

디폴트 메서드 description()을 CompleteCalc 클래스에서 원하는 기능으로 재정의

정적 메서드 사용하기

- static 키워드로 정적 메서드 구현

```
public interface Calc {  
    ...  
  
    static int total(int[] arr) {  
        int total = 0;  
  
        for(int i : arr) {  
            total += i;  
        }  
        return total;  
    }  
}
```

인터페이스에 정적 메서드 total() 구현

- 인터페이스 이름으로 정적 메서드 호출

```
int[] arr = {1, 2, 3, 4, 5};  
System.out.println(Calc.total(arr));
```

정적 메서드 사용하기

private 메서드

```
public interface Calc {
```

```
...
```

```
default void description( ) {
```

```
    System.out.println("정수 계산기를 구현합니다");
```

```
    myMethod( );
```

디폴트 메서드에서 private 메서드 호출

```
}
```

```
static int total(int[] arr) {
```

```
    int total = 0;
```

```
    for(int i: arr){
```

```
        total += i;
```

```
    }
```

```
    myStaticMethod( );
```

정적 메서드에서 private static 메서드 호출

```
    return total;
```

```
}
```

```
private void myMethod( ) {
```

```
    System.out.println("private 메서드입니다.");
```

```
}
```

private 메서드

```
private static void myStaticMethod( ) {
```

```
    System.out.println("private static 메서드입니다.");
```

```
}
```

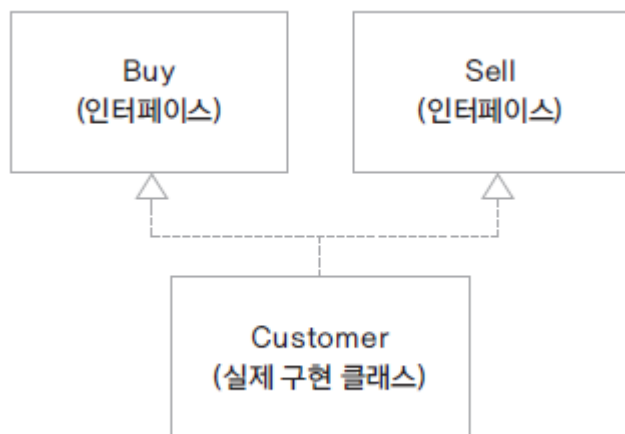
private static 메서드

```
}
```

- 인터페이스 내부에 private 혹은 private static으로 선언한 메서드 구현

- private static은 정적 메서드에서 사용 가능

두 개의 인터페이스 구현하기



```
package interfaceex;
```

```
public interface Buy {  
    void buy( );  
}
```

```
package interfaceex;
```

```
public interface Sell {  
    void sell( );  
}
```

두 인터페이스를 구현한 클래스

```
package interfaceex;
```

Customer 클래스는 Buy와 Sell
인터페이스를 모두 구현함

```
public class Customer implements Buy, Sell {
```

```
    @Override
```

```
    public void sell( ) {
```

```
        System.out.println("구매하기");
```

```
    }
```

```
    @Override
```

```
    public void buy( ) {
```

```
        System.out.println("판매하기");
```

```
    }
```

```
}
```

구현한 클래스 사용하기

```
public class CustomerTest {  
    public static void main(String[] args) {  
        Customer customer = new Customer( );
```

```
        Buy buyer = customer;  
        buyer.buy( );
```

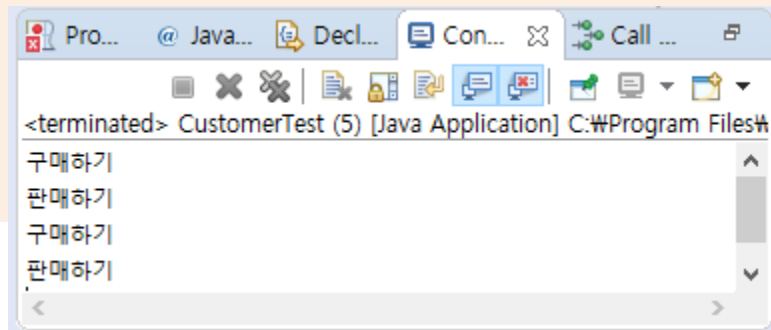
Customer 클래스형인 customer를 Buy 인터페이스형인 buyer에
대입하여 형 변환. buyer는 Buy 인터페이스의 메서드만 호출 가능

```
        Sell seller = customer;  
        seller.sell( );
```

Customer 클래스형인 customer를 Sell 인터페이스형인 seller에
대입하여 형 변환. seller는 Sell 인터페이스의 메서드만 호출 가능

```
        if(seller instanceof Customer) {  
            Customer customer2 = (Customer)seller;  
            customer2.buy( );  
            customer2.sell( );  
        }  
    }  
}
```

seller를 하위 클래스형인 Customer로
다시 형 변환



두 인터페이스의 디폴트 메서드가 중복되는 경우

```
package interfaceex;

public interface Buy {
    void buy( );

    default void order( ) {
        System.out.println("구매 주문");
    }
}
```

```
package interfaceex;

public interface Sell {
    void sell( );

    default void order( ) {
        System.out.println("판매 주문");
    }
}
```

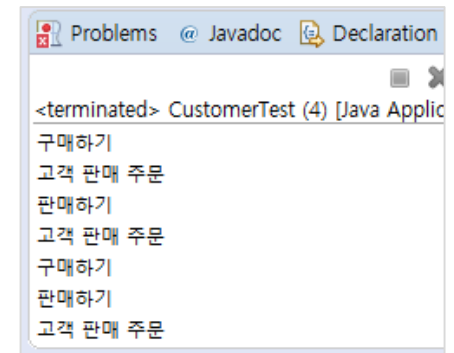
- 구현한 클래스에서 중복된 디폴트 메서드를 재정의

```
public class Customer implements Buy, Sell {
    ...
    @Override
    public void order( ) {
        System.out.println("고객 판매 주문");
    }
}
```

디폴트 메서드 order()를
Customer 클래스에서 재정의함

두 개의 인터페이스 구현 클래스 실행하기

```
public class CustomerTest {  
    public static void main(String[] args) {  
        Customer customer = new Customer();  
  
        Buy buyer = customer;  
        buyer.buy();  
        buyer.order();  
  
        Sell seller = customer;  
        seller.sell();  
        seller.order();  
  
        if (seller instanceof Customer) {  
            Customer customer2 = (Customer) seller;  
            customer2.buy();  
            customer2.sell();  
        }  
  
        customer.order();  
    }  
}
```

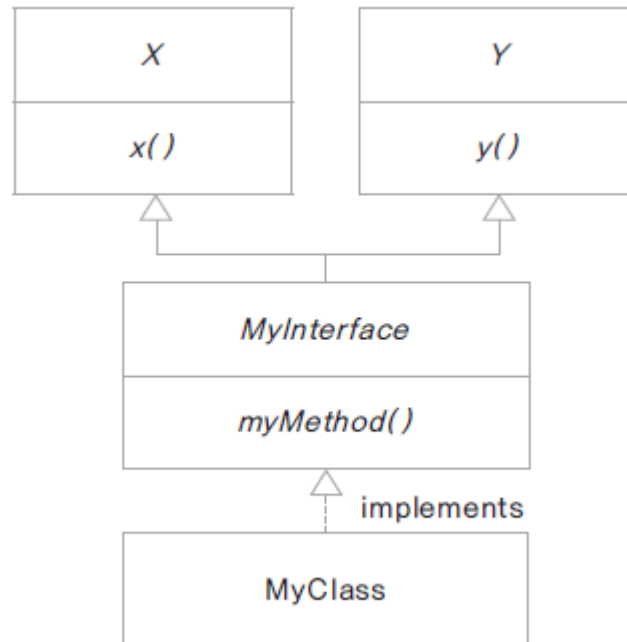


인터페이스 상속

- 인터페이스 간에도 상속이 가능
- 구현코드의 상속이 아니므로 **형 상속(type inheritance)**이라고 함

```
public interface MyInterface extends X, Y {  
    void myMethod( );  
}
```

인터페이스 여러 개를 상속받을 수 있음



인터페이스 상속

- 여러 인터페이스를 상속한 인터페이스(MyInterface)를 구현하는 클래스는 선언된 모든 추상 메서드를 구현해야 함

```
public class MyClass implements MyInterface {
```

```
    @Override
```

```
    public void x( ) {  
        System.out.println("x( )");  
    }
```

X 인터페이스에서 상속받은
x() 메서드 구현

```
    @Override
```

```
    public void y( ) {  
        System.out.println("y( )");  
    }
```

Y 인터페이스에서 상속받은
y() 메서드 구현

```
    @Override
```

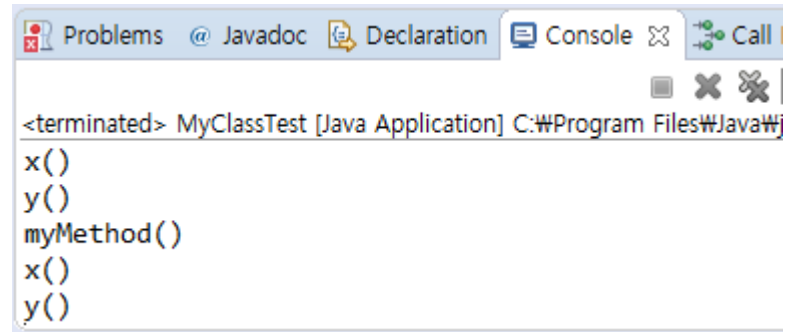
```
    public void myMethod( ) {  
        System.out.println("myMethod( )");  
    }
```

MyInterface 인터페이스의
myMethod() 메서드 구현

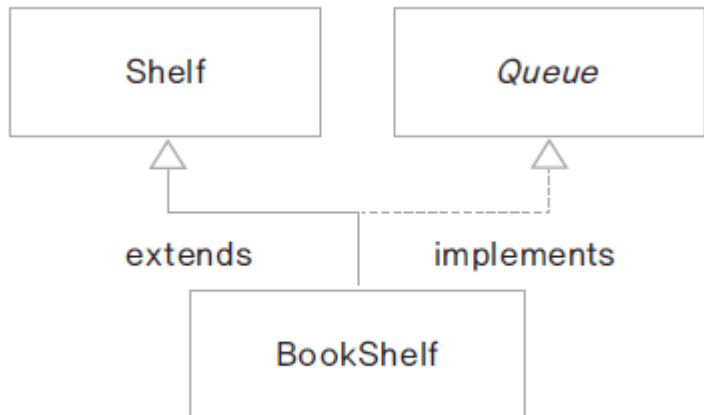
```
}
```


인터페이스 상속

```
public class MyClassTest {  
    public static void main(String[] args) {  
        MyClass mClass = new MyClass();  
        X xClass = mClass;  
        xClass.x();  
  
        Y yClass = mClass;  
        yClass.y();  
  
        MyInterface iClass = mClass;  
        iClass.myMethod();  
        iClass.x();  
        iClass.y();  
    }  
}
```



인터페이스 구현과 클래스 상속 함께 사용하기



- 실제 프레임워크(스프링, 안드로이드)를 사용하면 클래스를 상속 받고 여러 인터페이스를 구현하는 경우가 종종 있음

```
public class BookShelf extends Shelf implements Queue {  
    @Override  
    public void enqueue(String title) {  
        shelf.add(title);  
    }  
  
    @Override  
    public String dequeue( ) {  
        return shelf.remove(0);  
    }  
  
    @Override  
    public int getSize( ) {  
        return getCount( );  
    }  
}
```

배열에 요소 추가

맨 처음 요소를 배열에서 삭제하고 반환

배열 요소 개수 반환

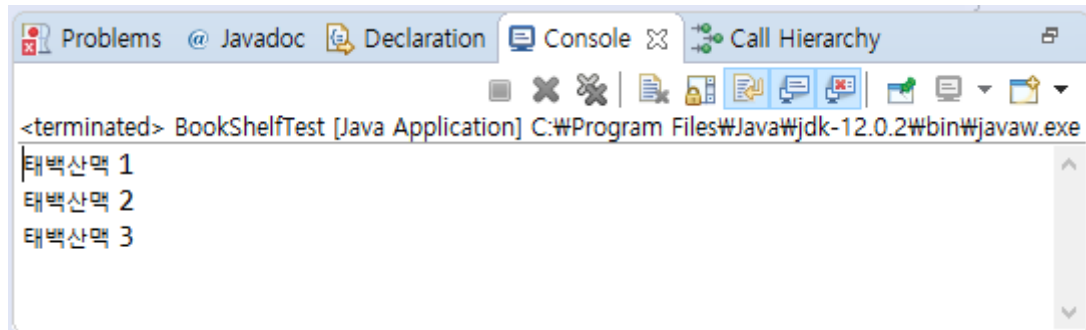
Shelf 클래스, Queue 인터페이스 정의하기

```
public class Shelf {  
    protected ArrayList<String> shelf;  
  
    // 디폴트 생성자로 Shelf 클래스를 생성하면 ArrayList도 생성됨.  
    public Shelf() {  
        shelf = new ArrayList<String>();  
    }  
  
    public ArrayList<String> getShelf() {  
        return shelf;  
    }  
  
    public int getCount() {  
        return shelf.size();  
    }  
}
```

```
public interface Queue {  
    void enqueue(String title); // 배열의 마지막에 추가  
    String dequeue();           // 배열의 맨 처음 항목을 반환  
    int getSize();              // 현재 Queue에 있는 개수 반환  
}
```

BookShelf 테스트 하기

```
public class BookShelfTest {  
    public static void main(String[] args) {  
  
        Queue shelfQueue = new BookShelf();  
        shelfQueue.enqueue("태백산맥 1");  
        shelfQueue.enqueue("태백산맥 2");  
        shelfQueue.enqueue("태백산맥 3");  
  
        // 입력 순서대로 요소를 꺼내서 출력  
        System.out.println(shelfQueue.dequeue());  
        System.out.println(shelfQueue.dequeue());  
        System.out.println(shelfQueue.dequeue());  
    }  
}
```



Quiz

〈DrawShape.java〉

각 클래스들이 Movable과 Drawable 인터페이스를 동시에 상속받도록 하여 아래와 같이 구현하세요.

[인터페이스] Movable { 추상 메서드 move(int x, int y) }

[인터페이스] Drawable { 추상 메서드 draw() }

[클래스] Rectangle, Triangle, Circle

[각 클래스에서 추상 메서드 완성]

move() { "포인터 (x, y)로 이동" }

draw() { “OOO을 그렸습니다.” }

[실행 결과]

포인터 (4, 8)로 이동

사각형을 그렸습니다.

포인터 (2, 3)로 이동

삼각형을 그렸습니다.

포인터 (5, 5)로 이동

원을 그렸습니다.

Quiz

〈RobotTest.java〉

[부모 클래스] Robot { 내용 없음 }

[자식 클래스] DanceRobot{ void dance() }, SingRobot{ void sing() }, DrawRobot{ void draw() }

[action() 메서드] main 클래스 내부에 정의 : **static void action(Robot r) { }**

- 기능 : 생성된 객체의 메서드를 호출한다.
instanceof 연산자를 사용하여 실제 인스턴스의 타입을 확인한 이후 메서드를 호출한다.
 - 생성된 객체가 DanceRobot인 경우, dance() 메서드 호출
 - 생성된 객체가 SingRobot인 경우, sing() 메서드 호출
 - 생성된 객체가 DrawRobot인 경우, draw() 메서드 호출
 - 그 외 생성된 Robot 객체인 경우

[main() 메서드] 각 객체를 생성하여 action() 메서드 호출

[실행 결과]

로봇이 춤을 춥니다.
로봇이 노래를 합니다.
로봇이 그림을 그립니다.
로봇이 정지한 상태입니다.