인터페이스

- 자바 언어에서는 다중 상속을 지원하지 않기 때문에 한 번에 하나의 클래스만 상속 가능
    - 그러나 인터페이스의 경우 다중 "구현"이 가능하므로 **여러 개의 인터페이스를 동시에 구현하는 것이 가능**함
- **interface 키워드**를 통해서 인터페이스 정의 가능

문법

```
interface MyInterface {}
```

- (자바 1.7까지는) 인터페이스(interface) 내부에는 추상 메소드와 상수만을 포함 시킬 수 있음

```
interface MyInterface {
    // 인터페이스 이름으로 접근할(static) 상수(final)만 선언 가능
    public static final int a = 1000;
    // 모든걸 다 생략하면 자동으로 "public static final"을 붙여줌
    int b = 3000;

    // 추상 메서드 포함 가능
    abstract public void abstractMethod1();
    // 생략하면 자동으로 "abstract public"을 붙여줌
    void abstractMethod2();
}
```

- (자바 1.8버전부터) **default 키워드**를 붙여서 구현 메서드도 포함 가능

```
interface MyInterface {
    // ...

    // default 키워드를 통해서 구현 메서드 추가 가능
    default public void concreteMethod() {
        System.out.println("from concrete method");
    }
}
```

- 인터페이스는 추상 클래스와 마찬가지로 **객체 생성이 불가능**하고, 인터페이스가 포함하고 있는 **추상 메소드를 모두 구현해 줄 클래스를 작성하고 해당 타입의 객체를 생성**해야 함

문법

```
class 클래스이름 implements 인터페이스이름 { ... }
```

클래스 정의

```
// 인터페이스를 "구현"하는 개념이므로 extends 대신에 implements 키워드 사용
class MyInterfaceImpl implements MyInterface {
    // 모든 추상 메서드를 재정의해야 함
    @Override
    public void abstractMethod1() {
        System.out.println("from abstract method 1");
    }
    @Override
    public void abstractMethod2() {
        System.out.println("from abstract method 2");
    }
}
```

- 인터페이스는 다중 상속 가능

```
interface Interface1 {
    public void abstractMethod1();
}

interface Interface2 {
    public void abstractMethod2();
}

// 두 개의 인터페이스 구현 가능 (일종의 다중 상속)
class MyMultipleInterfaceImpl implements Interface1, Interface2 {
    // 추상 메서드 두 개 구현 필요
    @Override
    public void abstractMethod1() {}
    @Override
    public void abstractMethod2() {}
}
```

- 인터페이스에서 다른 인터페이스를 상속 가능
    - 여기서는 extends 키워드를 사용함을 유의

```
interface SuperInterface {
    public void superMethod();
}

// SuperInterface를 상속받는 SubInterface 정의
// (superMethod 추상 메서드가 추가됨)
interface SubInterface extends SuperInterface {
    public void subMethod();
}

class SubInterfaceImpl implements SubInterface {
    @Override
    public void superMethod() {
        System.out.println("from super method");
    }
    @Override
    public void subMethod() {
        System.out.println("from sub method");
    }
}
```

인터페이스 활용 사례 살펴보기

인터페이스 활용 사례

```
// 화면에 무언가를 표시하는데 사용할 캔버스 객체가 있다고 가정
class Canvas {
    public void draw(Drawable d, int x, int y) {
        d.draw(this, x, y);
    }
}

interface Drawable {
    void draw(Canvas c, int x, int y);
}

class Line implements Drawable {
    // 생성자가 있다고 가정하고 다음의 값을 초기화하면 p1, p2의 위치에 선을 그릴 수 있음
    private double p1x, p1y, p2x, p2y;
    @Override
    public void draw(Canvas c, int x, int y) {
        System.out.println(c + "의 " + x + ", " + y + " 위치에 선을 그립니다.");
    }
}

class Rectangle implements Drawable {
    // 생성자가 있다고 가정하고 다음의 값을 초기화하면 width, height 크기 만큼의 사각형을
그릴 수 있음
    private double width, height;
    @Override
    public void draw(Canvas c, int x, int y) {
        System.out.println(c + "의 " + x + ", " + y + " 위치에 사각형을 그립니다.");
    }
}

class Circle implements Drawable {
    // 생성자가 있다고 가정하고 다음의 값을 초기화하면 radius 반지름 크기의 사각형을 그릴
수 있음
    private double radius;
    @Override
    public void draw(Canvas c, int x, int y) {
        System.out.println(c + "의 " + x + ", " + y + " 위치를 중앙으로 하는 원을 그
립니다.");
    }
}

class Image implements Drawable {
    // 생성자가 있다고 가정하고 다음의 값을 초기화하면 이미지 파일을 불러와 이미지를 그릴 수
있음
    private File imageFile;
    @Override
    public void draw(Canvas c, int x, int y) {
        System.out.println(c + "의 " + x + ", " + y + " 위치를 이미지의 좌상단 시작
위치로 사용하는 이미지를 그립니다.");
```

```
    }
}
```

사용 예시

```
public static void main(String[] args) {
    Canvas c = new Canvas();
    // 인터페이스를 이용한 다형성 구현
    Drawable[] drawables = {
        new Line(),
        new Rectangle(),
        new Circle(),
        new Image()
    };

    for(Drawable d : drawables) {
        // 캔버스에 모두 그리기 (단, d의 그리는 동작은 다 달라지게 됨)
        c.draw(d, 0, 0);
    }
}
```

- 자주 사용되는 인터페이스들

```
List, Set, Map, Collection, Iterable, Comparable, Runnable, Serializable
```

직접 Iterable 인터페이스 구현해보기

```
class FruitsIterable implements Iterable<String> {

    @Override
    public Iterator<String> iterator() {
        String[] fruits = { "apple", "banana", "orange" };

        return new Iterator<String>() {
            private int currentIndex = 0;
            @Override
            public boolean hasNext() {
                return fruits.length > currentIndex;
            }

            @Override
            public String next() {
                String fruit = fruits[currentIndex];
                currentIndex++;
                return fruit;
            }
        };
    }
}

class IntRange implements Iterable<Integer> {
    private int start, end;
```

```
    public IntRange(int start, int end) {
        this.start = start;
        this.end = end;
    }

    @Override
    public Iterator<Integer> iterator() {
        return new Iterator<Integer>() {
            @Override
            public boolean hasNext() {
                return start <= end;
            }

            @Override
            public Integer next() {
                return start++;
            }
        };
    }
}
```

사용 예시 1

```
public static void main(String[] args) {
    FruitsIterable fruitsIterable = new FruitsIterable();
    IntRange intRange = new IntRange(1, 10);
    // Iterable 인터페이스를 구현한 객체는 향상된 for 문에 대상으로 사용 가능
    for(String fruit : fruitsIterable) {
        System.out.println(fruit);
    }
    for(Integer i : intRange) {
        System.out.println(i);
    }
}
```

사용 예시 2

```
public static void main(String[] args) {
    iterateSomething(fruitsIterable);
    iterateSomething(intRange);
    ArrayList<Double> arr = new ArrayList<>();
    arr.add(1.0); arr.add(2.0); arr.add(3.0);
    iterateSomething(arr);
}

static void iterateSomething(Iterable<? extends Object> something) {
    for(Object o : something) {
        System.out.println(o.toString());
    }
}
```