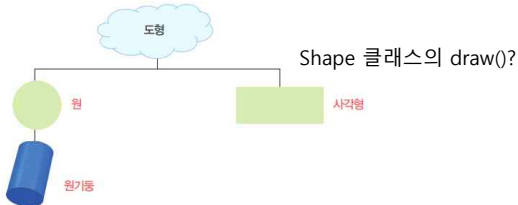




인터페이스와 특수 클래스

추상 클래스



■ 추상 메서드

- 메서드 본체를 완성하지 못한 메서드. 무엇을 할지는 선언할 수 있지만, 어떻게 할지는 정의할 수 없음
- 선언부만 있고 구현부(몸통, body)가 없는 메서드

■ 추상 클래스 : 클래스는 설계도, 추상 클래스는 미완성 설계도

- 보통 하나 이상의 추상 메서드(미완성 메서드)를 포함하지만 없을 수도 있음
- 주로 상속 계층에서 자식 멤버의 이름을 통일하기 위하여 사용
- 다른 클래스를 작성하는데 도움을 줄 목적으로 작성

추상클래스 s = new 추상클래스(); // 추상 클래스는 인스턴스를 생성하지 못한다.

추상 클래스

■ 추상 클래스 선언

```
abstract class 클래스이름 {  
    // 필드  
    // 생성자  
    // 메서드  
}
```

추상 클래스라는 것을 나타낸다.

일반적으로 하나 이상의 추상 메서드를 포함한다.

■ 추상 메서드 선언 : 꼭 필요하지마 자손마다 다르게 구현될 것으로 예상되는 경우에 사용

- 추상클래스를 상속받는 자손클래스에서 추상메소드의 구현부를 완성해야 함

```
abstract 반환타입 메서드이름() ;
```

추상 메서드라는 것을 나타낸다.

항상 세미콜론으로 끝나야 한다.

메서드 본체가 없다.

■ 예제

- [sec01/Shape](#),
- [sec01/Circle](#),
- [sec01/AbstractClassDemo](#)

원을 그리다.
원의 넓이는 28.3
사각형을 그리다.
사각형의 넓이는 12.0

인터페이스 기초

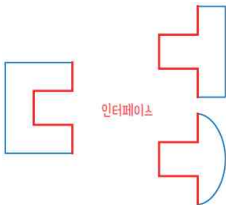
- 인터페이스 의미 : 일종의 추상 클래스, 추상클래스(미완성설계도)보다 추상화 정도가 높음

인터페이스만
맞으면
건축한 후에
어떤 가전제품들이
들어올지 신경 쓸
필요 없다.



인터페이스만
맞으면
어떤 가전제품도
사용할 수 있다.

현실 세계의 인터페이스



인터페이스



자바의 인터페이스

인터페이스 기초

■ 인터페이스에 의한 장점

- 인터페이스만 준수하면 통합에 신경 쓰지 않고 다양한 형태로 새로운 클래스를 개발할 수 있다.
- 클래스의 다중 상속을 지원하지 않지만, 인터페이스로 다중 상속 효과를 간접적으로 얻을 수 있다.

■ 인터페이스 vs. 추상 클래스

분류	인터페이스	추상 클래스
구현 메서드	포함 불가(단, 디폴트 메서드와 정적 메서드는 예외)	포함 가능
인스턴스 변수	포함 불가능	포함 가능
다중 상속	가능	불가능
디폴트 메서드	선언 가능	선언 불가능
생성자와 main()	선언 불가능	선언 가능
상속에서의 부모	인터페이스	인터페이스, 추상 클래스
접근 범위	모든 멤버를 공개	추상 메서드를 최소한 자식에게 공개

인터페이스 기초

■ 인터페이스의 예

- 자바가 기본적으로 제공하는 인터페이스는 다양하다.
- 대표적인 인터페이스
 - java.lang 패키지의 CharSequence, Comparable, Runnable 등
 - java.util 패키지의 Collection, Comparator, List 등
- 예를 들어, 객체의 크기를 비교하는 Comparable 인터페이스는 다음과 같다.

```
public interface Comparable {  
    int compareTo(Object other);  
}
```

객체 other보다 크면 양수, 같으면 0,
작으면 음수를 반환한다.

```
public interface Comparable<T> {  
    int compareTo(T o);  
}
```

인터페이스 기초

■ 인터페이스 구조

```
interface 인터페이스이름 {  
    // 상수 필드      → 상수만 가능하기 때문에 public static final 키워드 생략 가능  
  
    // abstract 메서드 → 인터페이스의 모든 메서드(아래 3가지 종류를 제외)가  
                        public abstract이기 때문에 public abstract 키워드 생략 가능  
  
    // default 메서드  → JDK 8부터 가능  
  
    // static 메서드   → JDK 8부터 가능  
  
    // private 메서드  → JDK 9부터 가능  
}
```

■ 디폴트 메서드와 정적 메서드

- 디폴트 메서드는 오버라이딩될 수 있지만, 정적 메서드는 오버라이딩될 수 없다.
- 디폴트 메서드는 인스턴스 메서드이므로 객체를 생성한 후 호출하지만, 정적 메서드는 인터페이스로 직접 호출한다.

인터페이스 기초

■ 인터페이스의 구조

- 인터페이스 멤버에 명시된 `public`, `static`, `final`, `abstract` 키워드는 생략 가능
- 생략한 키워드는 컴파일 과정에서 자동으로 추가
- 인터페이스 파일 확장자도 `java`
- 컴파일하면 확장자가 `class`인 파일을 생성

```
interface MyInterface {  
    int MAX = 10;  
  
    void sayHello();  
}
```

MyInterface.java

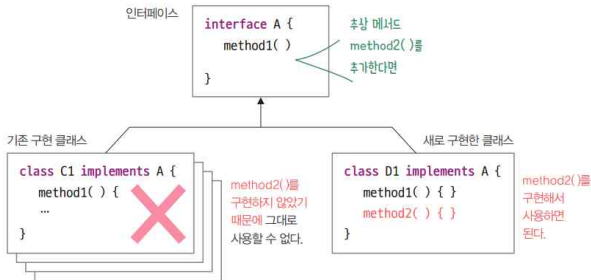


```
interface MyInterface {  
    public static final int MAX = 10;  
  
    public abstract void sayHello();  
}
```

MyInterface.class

인터페이스 기초

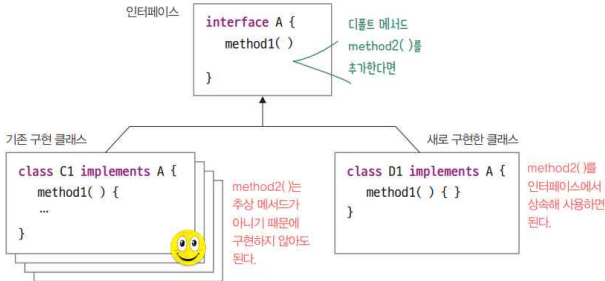
■ 인터페이스 수정과 기존 구현 클래스



인터페이스 기초

■ 인터페이스 수정과 기존 구현 클래스 : 디폴트 메서드

```
default 반환타입 디폴트메서드이름( ) {  
    // 본체를 구성하는 코드  
}
```



인터페이스 기초

■ 인터페이스 상속

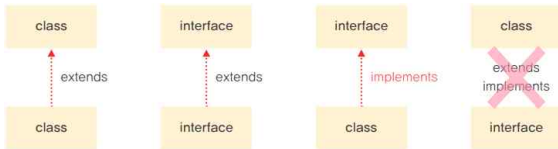
// 인터페이스를 상속하려면 extends 키워드를 사용한다.

```
interface 자식인터페이스 extends 부모인터페이스 {  
}
```

// 인터페이스를 구현하려면 implements 키워드를 사용한다.

```
class 자식클래스 implements 부모인터페이스 {  
}
```

■ 클래스와 인터페이스의 관계



인터페이스 기초

■ 인터페이스 상속

// 상속할 인터페이스가 여러 개라면 쉼표(,)로 연결한다.

```
interface 자식인터페이스 extends 부모인터페이스1, 부모인터페이스2 {  
}  
  
class 자식클래스 implements 부모인터페이스1, 부모인터페이스2 {  
}
```

// 인터페이스는 다중 상속할 수 있다.

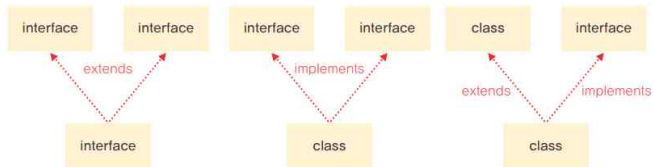
```
class 자식클래스 extends 부모클래스 implements 부모인터페이스1, 부모인터페이스2 {  
}
```

// 클래스는 다중 상속할 수 없다.

```
class 자식클래스 extends 부모클래스1, 부모클래스2 {  
}
```

인터페이스 기초

■ 인터페이스를 이용한 다중 상속 효과



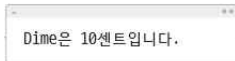
인터페이스 응용

■ 인터페이스와 상수

● 예제 :

[sec03/Coin1Demo](#)

[sec03/Coin2Demo](#)



인터페이스 응용

■ 인터페이스의 상속과 구현 클래스

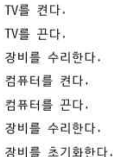
- 예를 들어, 전자제품에 포함되어야 하는 제어부의 요구 조건
 - 모든 전자제품에는 전원을 온.오프하는 기능이 있으며, 수리 및 공장 초기화를 할 수 있다.
 - 전자제품 객체는 turnOn() 메서드, turnOff() 메서드로만 전원을 조절할 수 있어야 한다.
 - 수리 및 공장 초기화 기능을 미리 구현해 놓아서 필요할 때 사용할 수 있어야 한다.
 - 수리 기능은 자식 클래스에서 오버라이딩할 수도 있다.

- 예제 :

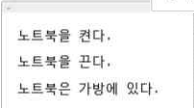
[sec03/Controllable](#)
[sec03/RemoteControllable](#)
[sec03/TV](#)
[sec03/ControllableDemo](#)

- 예제 :

[sec03/Notebook](#)



TV를 켜다.
TV를 끄다.
장비를 수리한다.
컴퓨터를 켜다.
컴퓨터를 끄다.
장비를 수리한다.
장비를 초기화한다.



노트북을 켜다.
노트북을 끄다.
노트북은 가방에 있다.

인터페이스와 다형성

■ 인터페이스 타입

- 인터페이스도 클래스처럼 하나의 타입이므로 변수를 인터페이스 타입으로 선언 가능
- 인터페이스의 구현 클래스는 그 인터페이스의 자식 타입

인터페이스타입 변수 = 구현객체

구현 객체는 인터페이스 타입이므로 자동 변환된다.

- 인터페이스 타입 변수가 구현 객체를 참조한다면 강제 타입 변환 가능

구현클래스타입 변수 = (구현클래스타입) 인터페이스타입변수

타입 변환 연산자이다.

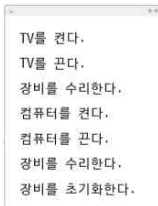
인터페이스 구현 객체를 참조하는 변수이다.

인터페이스와 다형성

■ 타입 변환과 다형성

● 예제 :

[sec04/ControllableDemo](#)



[sec04/AnimalDemo](#)



인터페이스와 다형성

■ 타입 변환과 다형성

- 예제 : [sec04/MovableDemo](#)

15m 이동했습니다.

```
interface Movable {  
    void move();  
}
```

Movable 타입에는
move() 메서드만 있고
show() 메서드는 없다.

```
class Car implements Movable {  
    public void move() { ... }  
    public void show() { ... }  
}
```

Car 타입에는
move(), show()
메서드 둘 다 있다.

MovableDemo.java

```
Movable m = new Car();  
m.move();  
  
// m.show();  
Car c = (Car) m; // 강제 타입 변환  
c.move();  
c.show();
```

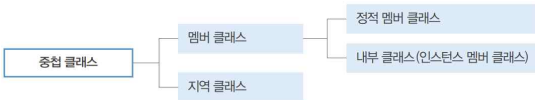
중첩 클래스와 중첩 인터페이스

■ 의미



중첩 클래스는 외부 클래스를
상속할 필요 없이
외부 클래스의 private 멤버까지
사용할 수 있다.

■ 종류



중첩 클래스와 중첩 인터페이스

■ 중첩 클래스의 구조

```
class 외부클래스 {  
    class 멤버클래스 {  
    }  
  
    interface 중첩인터페이스 {  
    }  
}
```

외부 클래스의
멤버로 선언된
클래스이다.

```
class 외부클래스 {  
    void 메서드() {  
        class 지역클래스 {  
        }  
    }  
}
```

외부 클래스의
메서드 내부에
선언된
클래스이다.

■ 컴파일 후 생성 파일

외부클래스\$멤버클래스.class

외부클래스\$중첩인터페이스.class

외부클래스\$1지역클래스.class

이름이 동일한 지역 클래스가 있다면 \$2 등을 사용한다.

중첩 클래스와 중첩 인터페이스

■ 외부 클래스 접근

외부클래스.this

■ 중첩 클래스의 객체 생성

외부클래스.내부클래스 변수 = 외부클래스의객체변수.new 내부클래스생성자();

외부클래스.정적멤버클래스 변수 = new 외부클래스.정적멤버클래스생성자();

■ 예제

- 내부 클래스 사용 : [sec05/MemberClassDemo](#)
- 지역 클래스와 지역 변수 관계 : [sec05/LocalClassDemo](#)
- 중첩 인터페이스 사용 : [sec05/InnerInterfaceDemo](#)

내부
내부 클래스
비공개
내부
외부

내부
지역 클래스

아이콘을 터치한다.

익명 클래스

■ 소개

- 중첩 클래스의 특수한 형태로 코드가 단순해지기 때문에 이벤트 처리나 스레드 등에서 자주 사용

```
class OnlyOnce  $\left[ \begin{array}{l} \text{extends} \\ \text{implements} \end{array} \right]$  Parent {  
    // Parent가 클래스라면 오버라이딩한 메서드  
    // Parent가 인터페이스라면 구현한 메서드  
}
```

```
Parent p = new OnlyOnce();
```



```
Parent p = new Parent() {
```

```
    // Parent가 클래스라면 오버라이딩한 메서드  
    // Parent가 인터페이스라면 구현한 메서드
```

```
};
```

하나의 실행문이므로 세미콜론(;)으로 끝낸다.

무명 클래스 본체로서 OnlyOnce 클래스의 본체와 동일하다.

익명 클래스

■ 활용

- 익명 클래스의 부모로 사용할 클래스 : [sec06/Bird](#)
- 기명 멤버 클래스 : [sec06/MemberDemo](#)
- 익명 멤버 클래스 : [sec06/Anonymous1Demo](#)
- 기명 지역 클래스 : [sec06/LocalDemo](#)
- 익명 지역 클래스 : [sec06/Anonymous2Demo](#)

■ 동기

- 객체 간에 불변 데이터를 전달하는 것은 Java 애플리케이션에서 가장 흔하지만 일상적인 작업
- 즉, 대부분의 경우 판에 박은 형태의 필드, 메서드 등을 가진 클래스를 정의하는 작업이지만 사소한 실수 등에 취약
- 이와 같은 경우 다음과 같은 멤버를 포함하는 클래스를 정의
 - private, final 멤버 필드
 - 멤버 필드를 위한 getter 메서드
 - 필드를 포함하는 생성자
 - 모든 필드 내용이 동일하면 true를 반환하는 equals 메서드
 - 모든 필드 내용이 동일하면 동일한 값을 반환하는 hashCode 메서드
 - 클래스 이름과 각 필드의 이름 및 값을 포함하는 toString 메서드
- 문제점
 - 판에 박은 형태의 코드가 너무 많다. 또한 약간의 변화에도 변경할 코드가 많다.
 - Circle 클래스의 목적을 모호하게 한다.
- 자바 16부터 레코드(record), 즉 순수하게 데이터를 보유하기 위한 특수한 종류의 클래스를 도입하여 이러한 문제를 해결

■ 기초

- 레코드는 필드의 유형과 이름만 필요한 불변 데이터를 위한 특별한 클래스
- 형식

```
record 레코드_이름 ( 컴포넌트_필드... ) { }
```

헤더 본체

- 레코드는 다음과 같은 클래스로 컴파일
 - 레코드_이름을 가진 final class 파일을 생성
 - 0개 이상의 컴포넌트_필드는 클래스의 private final인 멤버 필드로 사용
 - 접근자가 자동으로 생성되며, 접근자 이름은 컴포넌트 필드의 이름과 동일
 - 정규 생성자(canonical constructor),
 - 모든 필드 내용이 동일하면 true를 반환하는 equals() 메서드
 - 모든 필드 내용이 동일하면 동일한 값을 반환하는 hashCode() 메서드
 - 클래스 이름과 각 필드의 이름 및 값을 포함하는 toString() 메서드
- 클래스와 레코드 비교 예제 : [sec07/CircleDemo](#), [sec07/recordtype1/CircleDemo](#)

■ 정규 생성자(canonical constructor)

- 헤더와 동일한 시그니처를 가진 생성자
- 형식

```
record 레코드_이름 ( 컴포넌트_필드... ) {  
    public 레코드_이름 ( 컴포넌트_필드... ) {  
        // 컴포넌트_필드로 컴포넌트_필드 초기화 코드  
        // 기타 코드  
    }  
}
```

→ 정규 생성자

- 문제점 : 레코드 클래스의 구성 요소를 반복하기 때문에 지루하고 오류가 발생하기 쉽다

■ 축약 생성자(compact constructor)

- 정규 생성자의 문제점을 해소
- 형식

```
record 레코드_이름 ( 컴포넌트_필드... ) {  
    public 레코드_이름 {  
        // 필요하다면 컴포넌트_필드 초기화를 제외한 코드  
    }  
}
```

→ 축약 생성자

- 예제 : [sec07/recordtype2/CircleDemo](#), [sec07/recordtype3/CircleDemo](#)



■ 동기

- 예제 : [sec08/sealed1/ShapeDemo](#)
- Shape의 모든 구현체를 점검하지 않으면 컴파일 오류

■ 목적

- 확장하거나 구현할 수 있는 class나 interface를 제한한다.
- 부모클래스의 사용을 제한하는 접근 제어자보다 더 강력한 기능을 제공
- 패턴 매칭을 이용하여 서브클래스를 명시적으로 확인 가능

■ 형식

```
sealed class 클래스_이름 permits 자식클래스_목록 { }
```

- sealed, non-sealed와 permits는 키워드가 아니라 예약어. 따라서 식별자로 사용 가능
- 예제 : [sec08/sealed2/ShapeDemo](#)

Eclipse 2022-03, IntelliJ Idea 2022.1 버전에서 정상 작동.
이전 버전에서는 봉인 클래스를 지원하지 않을 수도 있음

■ 주의

- 봉인 클래스의 자식은 자식_클래스 목록에 있는 것만 허용
- 봉인 클래스의 자식은 기본적으로 봉인 클래스이지만 non-sealed로 수식하면 비봉인 클래스
- 봉인 클래스의 자식은 final, sealed, non-sealed 중 하나로 수식 필수
- 봉인 클래스와 자식 클래스는 동일 모듈 소속. 모듈이 없으면 동일 패키지 소속
- 봉인 인터페이스도 봉인 클래스와 유사