● 인터페이스(interface): 추상메서드와 상수만을 멤버로 가질 수 있는 기본 설계도 (추상화 정도가 높은 일종의 추상클래스). 다른 클래스를 작성하는데 도움 줄 목적으로 작성된다.

```
interface 인터페이스이름 {
    public static final 타입 상수이름 = 값;
    public abstract 메서드이름(매개변수목록);
}
```

키워드로 interface을 사용하며 접근제어자로 public 또는 default을 사용할 수 있다.
모든 멤버변수는 public static final이어야 하며, 이를 생략할 수 있다.
모든 메서드는 public abstract이어야 하며, 이를 생략할 수 있다. (단, JDK1.8부터 static메서드와 디폴트 메서드도 허용된다)
인터페이스에 정의된 모든 멤버에 예외없이 적용되는 사항이기 때문에 제어자를 생략할 수 있다. 컴파일 시에 컴파일러가 자동적으로 추가해 준다.

```
interface PlayingCard {
    public static final int SPADE = 4;
    final int DIAMOND = 3; // public static final int DIAMOND = 3;
    static int HEART = 2; // public static final int HEART = 2;
    int CLOVER = 1; // public static final int CLOVER = 1;

    public abstract String getCardNumber();
    String getCardKInd(); // public abstract String getCardKind();
}
```

▶ 인터페이스의 상속:

키워드 extends을 사용해 인터페이스로부터만 상속받을 수 있다. 자손 인터페이스는 조상 인터페이스에 정의된 멤버를 모두 상속받는다.
클래스와는 달리 다중상속, 즉 여러 개의 인터페이스로부터 상속을 받는 것이 가능하다.
인터페이스는 클래스와는 달리 Object클래스와 같은 최고 조상이 없다
인터페이스의 이름에는 ~able (~할 수 있는)로 끝나는 것들이 많은데, 그 이유는 어떠한 기능 또는 행위를 하는데 필요한 메서드를 제공한다는 의미를 강조하기 위해서이다. 또한 그 인터페이스를 구현한 클래스는 '~할 수 있는' 능력을 갖추었다는 의미이기도 한다.
인터페이스 추상 메서드 작성 시 선언부 전에 /** */ 사이 어떤 기능을 할 메서드로 작성됐는지 먼저 적어준다

```
interface Movable {
    //** 지정된 위치(x, y)로 이동하는 기능의 메서드 */
    void move(int x, int y);
}

interface Attackable {
    //** 지정된 대상(u)를 공격하는 기능의 메서드 */
    void attac(Unit u);
}

interface Fightable extends Movable, Attackable {
}

// 자손 인터페이스 Fightable은 조상 인터페이스 Movable과 Attackable에
// 정의된 멤버를 모두 상속 받는다.
```

▶ 인터페이스의 구현:

키워드 implements을 사용해 인터페이스에 정의된 추상메서드의 구현부(몸통)을 만들어주는 클래스를 작성한다.
만일 구현하는 인터페이스의 메서드 중 일부만 구현한다면, abstract을 붙여서 추상 클래스로 선언해야 한다.
상속과 구현을 동시에 할 수도 있다.
자손 인터페이스에서 조상 인터페이스에 정의된 추상메서드를 구현할 때 접근 제어자를 반드시 public으로 해야 한다. 오버라이딩 할 때는 조상의 추상메서드(public abstract)보다 넓은 범위의 접근 제어자를 지정해야 하기 때문이다.

키워드 implements을 사용해 인터페이스에 정의된 추상메서드의 구현부(몸통)을 만들어주는 클래스를 작성한다.

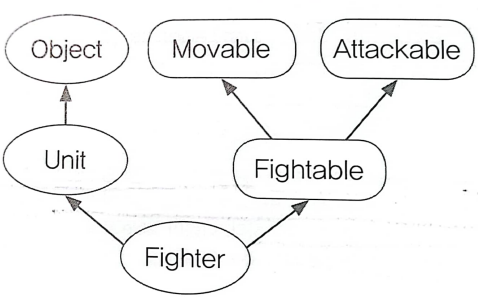
```
class 클래스이름 implements 인터페이스이름 {
    // 인터페이스에 정의된 추상메서드를 구현해야 한다.
}
```

```
class Fighter implements Fightable {
    public void move(int x, int y) {
        내용구현;
    };
    public void attack(Unit u) {
       내용구현;
    };
}
```

```
// 구현하는 인터페이스의 메서드 중 일부만 구현한다면, abstract을 붙여서 추상 클래스로 선언
abstract class Fighter implements Fightable {
    public void move(int x, int y) {
        내용구현;
    };
}
```

```java
// 상속과 구현을 동시에
class Fighter extends Unit implements Fightable {
    public void move(int x, int y) {
        내용구현;
    }
    public void attack(Unit u) {
        내용구현;
    }
}
```

```java
//  예제
class FighterTest {
    public static void main(String[] args) {
        Fighter f = new Fighter();

        if (f instanceof Unit)  {
            System.out.println("f는 Unit클래스의 자손입니다.");
        }
        if (f instanceof Fightable) {
            System.out.println("f는 Fightable인터페이스를 구현했습니다.");
        }
        if (f instanceof Movable) {
            System.out.println("f는 Movable인터페이스를 구현했습니다.");
        }
        if (f instanceof Attackable) {
            System.out.println("f는 Attackable인터페이스를 구현했습니다.");
        }
        if (f instanceof Object) {
            System.out.println("f는 Object클래스의 자손입니다.");
        }
    }
}

class Fighter extends Unit implements Fightable {
    public void move(int x, int y) {
        /* 내용 생략 */
    }

    public void attack(Unit u) {
        /* 내용 생략 */
    }
}

class Unit {
    int currentHP;  // 유닛의 체력
    int x;          // 유닛의 위치(x좌표)
    int y;          // 유닛의 위치(y좌표)
}

interface Fightable extends Movable, Attackable { }

interface Movable {
    void move(int x, int y);
}

interface Attackable {
    void attack(Unit u);
}
```



Fighter클래스는 Unit클래스로부터 상속받고 Fightable인터페이스만을 구현했지만, Unit클래스는 Object클래스의 자손이고, Fightable인터페이스는 Attackable과 Movable인터페이스의 자손이므로 Fighter클래스는 이 모든 클래스와 인터페이스의 자손이 된다.

자손 인터페이스에서 조상 인터페이스에 정의된 추상메서드를 구현할 때 접근 제어자를 반드시 public으로 해야 한다. 오버라이딩 할 때는 조상의 메서드보다 넓은 범위의 접근 제어자를 지정해야 하기 때문이다. 위의 예에서 Movable인터페이스에 'void move(int x, int y)'와 같이 정의되어 있지만 사실 'public abstract'가 생략된 것이기 때문에, 이를 구현하는 Fighter클래스에서는 접근 제어자를 반드시 public으로 해야 하는 것이다.

▶ 인터페이스를 이용한 다중상속: 인터페이스는 static상수만 정의할 수 있으므로 조상클래스의 멤버변수와 충돌하는 경우는 거의 없고 충돌된다고 하더라도 클래스 이름을 붙여서 구분이 가능하다. 그리고 추상메서드는 구현내용이 전혀 없으므로 조상클래스의 메서드와 선언부가 일치하는 경우에는 당연히 조상 클래스 쪽의 메서드를 상속받으면 되므로 문제되지 않는다. 만일 두 개의 클래스로부터 상속을 받아야 할 상황이라면, 두 조상클래스 중에서 비중이 높은 쪽은 선택하고 다른 한쪽은 클래스 내부에 멤버로 포함시키는 방식으로 처리하거나, 어느 한쪽의 필요한 부분을 뽑아서 인터페이스로 만든 다음 구현하도록 한다.

```
// 예시 - Tv클래스는 상속받고, VCR클래스는 인터페이스로 만들어서 구현 시에
// 코드를 작성하는 대신 VCR인스턴스의 메서드를 호출한다(내부에 멤버로 포함시킨다)
// 이렇게 하면 VCR클래스의 내용이 변경되어도 변경된 내용이 TVCR클래스에도 자동적으로 반영된다

public class Tv {
    protected boolean power;
    protected int channel;
    protected int volume;

    public void power() { power =! power; }
    public void channelUp() { channel++; }
    public void channelDown() { channel--; }
    public void volumeUp() { volume++; }
    public void volumeDown() { volume--; }

}

public class VCR {
    protected int counter;

    public void play() {};
    public void stop() {};
    public void reset() {
        counter = 0;
    }
    public int getCounter() {
        return counter;
    }
    public void setCounter(int c) {
        counter = c;
    }
}

// VCR 클래스에 정의된 메서드와 일치하는 추상메서드를 갖는 인터페이스 작성
public interface IVCR {
    public void play();
    public void stop();
    public void reset();
    public int getCounter();
    public void setCounter(int c);
```

```
}

// TVCR클래스 작성: IVCR을 구현하고 Tv클래스로부터 상속받는 클래스
// IVCR을 구현할 때, 코드를 작성하는 대신 VCR인스턴스의 메서드를 호출한다.
public class TVCR extends Tv implements IVCR {
    VCR vcr = new VCR();

    public void play() {
        vcr.play();
    }

    public void stop() {
        vcr.stop();
    }

    public void reset() {
        vcr.reset();
    }

    public int getCounter() {
        return vcr.getCounter();
    }

    public void setCounter(int c) {
        vcr.setCounter(c);
    }
}
```

▶ 인터페이스를 이용한 다형성: 인터페이스의 타입의 참조변수로 인터페이스를 구현한 클래스의 인스턴스를 참조할 수 있으며, 인터페이스 타입으로의 형변환도 가능하다.

```
// 예제: 클래스 Figther가 인터페이스 Fightable을 구현했을 때,
// 다음과 같이 Fighter인스턴스를 Fightable타입의 참조변수로 참조하는 것이 가능하다.

Fightable f = new Figther(); // Fightable f = (Fightable)new Fighter();
// 이때 f로는 인터페이스 Fightable에 정의된 멤버들만 호출이 가능하다.
```

따라서 **인터페이스는 메서드의 매개변수의 타입으로 사용될 수 있다. 메서드 호출 시 해당 인터페이스를 구현한 클래스의 인스턴스를 매개변수로 제공한다.**

```
class Fighter extends Unit implements Fightable {
    public void move(int x, int y) { /* 내용 생략 */ }
    public void attack(Fightable f) { /* 내용생략 */ }
    // attack메서드의 매개변수로 Fighter인스턴스를 넘겨준다.
    // attack(new Fighter())와 같다.
}
```

또한 메서드의 리턴타입으로 인터페이스의 타입을 지정할 수 있다. 리턴타입이 인터페이스라는 것은 메서드가 해당 인터페이스를 구현한 클래스의 인스턴스를 반환하는 것을 의미한다.

```
Fightable method() {
    Fighter f = new Fighter();
    return f;
    // 위의 두 문장을 한 문장으로; return new Fighter();
}

// method()의 리턴타입이 Fightable인터페이스이기 때문에 메서드의 return문에서
// Fightable인터페이스를 구현한 Fighter클래스의 인스턴스를 반환한다.
```

```java
// 예시

interface Parseable {
    // 구문 분석작업을 수행한다.
    public abstract void parse(String fileName);
}

class XMLParser implements Parseable { // Parseable 구현
    public void parse(String fileName) {
        /* 구문 분석작업을 수행하는 코드를 적는다. (구문분석을 수행하는 기능 구현)*/
        System.out.println(fileName + "- XML parsing completed.");
    }
}

class HTMLParser implements Parseable {// Parseable 구현
    public void parse(String fileName) {
        /* 구문 분석작업을 수행하는 코드를 적는다. */
        System.out.println(fileName + "- HTML parsing completed.");
    }
}

class ParserManager {
    public static Parseable getParser(String type) {
    // 리턴타입이 Parseable인터페이스이다. 매개변수로 넘겨받는 type의 값에 따라
    // 이 인터페이스를 구현한 클래스(XMLParser 또는 HTMLParser) 인스턴스를 생성하여 반환한다.
        if(type.equals("XML")) {
            return new XMLParser();
        } else {
            Parseable p = new HTMLParser();
            return p;
            // return new HTMLParser();
        }
    }
}


class ParserTest {
    public static void main(String args[]) {
        Parseable parser = ParserManager.getParser("XML"); // new XMLParser() 생성, 반환
        // 이는 마치 Perseable parser = new XMLParser();이 수행된 것과 같음.
        parser.parse("document.xml"); // document.xml - XML parsing completed.
        parser = ParserManager.getParser("HTML"); // new HTMLParser() 생성, 반환
        parser.parse("document2.html"); // document2.html - HTML parsing completed.
    }
}

// 만일 나중에 새로운 XML구문분석기 NewXMLPaser클래스가 나와도 ParserTest클래스는 변경할
// 필요 없이 ParserManager클래스의 getParser메서드에서 'return new XMLParser();' 대신
// 'return new NewXMLParser();'로 변경하기만 하면 된다.
```

이런 장점은 특히 분산환경 프로그래밍에서 위력을 발휘한다. 사용자 컴퓨터에 설치된 프로그램을 변경하지 않고, 서버측의 변경만으로도 사용자가 새로 개정된 프로그램을 사용하는 것이 가능하다.

Resource: 자바의 정석