▶ 인터페이스의 장점

1. 개발시간을 단축시킬 수 있다 일단 인터페이스가 작성되면, 이를 사용해서 프로그램을 작성하는 것이 가능하다. 메서드를 호출하는 쪽에서는 메서드의 내용에 관계없이 선언부만 알면 되고, 동시에 다른 한쪽에서는 인터페이스를 구현하는 클래스를 작성하게 하면, 인터페이스를 구한하는 클래스가 작성될 때까지 기다리지 않고도 양쪽에서 동시에 개발을 진행할 수 있다.
2. 표준화가 가능하다. 프로젝트에 사용되는 기본 틀을 인터페이스로 작성한 다음 개발자들에게 인터페이스를 구현하여 프로그램을 작성하도록 하면 보다 일관되고 정형화된 프로그램의 개발이 가능하다.
3. 서로 관계없는 클래스들에게 관계를 맺어줄 수 있다. 하나의 인터페이스를 공통적으로 구현하도록 함으로써 서로 관계없는 클래스들 사이에 관계를 맺어줄 수 있다.
4. 독립적인 프로그래밍이 가능하다. 인터페이스를 이용하면 클래스의 선언과 구현을 분리시킬 수 있기 때문에, 실제구현에 독립적인 프로그램을 작성하는 것이 가능하다. 클래스와 클래스간의 직접적인 관계를 인터페이스를 이용해서 간접적인 관계로 변경하면, 한 클래스의 변경이 다른 클래스에 영향을 미치지 않는 독립적인 프로그래밍이 가능하다.
예) 한 데이트베이스가 회사가 제공하는 특정 데이터베이스를 사용할 때 데이터베이스 관련 인터페이스를 정의하고 이를 이용해서 프로그램을 작성하면, 데이터베이스 종류가 변경되더라도 프로그램을 변경하지 않도록 할 수 있다. 단, 데이터베이스 회사에서 제공하는 클래스도 인터페이스를 구현하도록 요구해야 한다. 데이터베이스를 이용한 응용프로그램을 작성하는 쪽에서는 인터페이스를 이용해서 프로그램을 작성하고, 데이터베이스 회사에서는 인터페이스를 구현한 클래스를 작성해서 제공해야 한다. 실제로 자바에서는 다수의 데이터베이스와 관련된 다수의 인터페이스를 제공하고 있으며, 프로그래머는 이 인터페이스를 이용해서 프로그래밍하면 특정 데이터베이스에 종속되지 않는 프로그램을 작성할 수 있다.

만약 공통된 조상이 없는 두 클래스에 동일한 메서드를 정의해야 한다면, 인터페이스를 정의하고 두 클래스에게 인터페이스를 구현하도록 하면 기존의 상속체계를 유지하면서 두 클래스에게 공통점을 부여할 수 있다.

```
// 예를 들면 SCV에게 Tank와 Dropship같은 기계화 유닛을 수리할 수 있는 기능을 제공하기 위해
// repair이라는 메서드를 정의할 때, Repairable이라는 빈 인터페이스를 정의하고 수리가 가능한
// 기계화 유닛에게(SCV, Tank, Dropship) 이 인터페이스를 구현하도록 하면 된다.
```

```
class Unit {
    int hitPoint;
    final int MAX_HP;
    Unit(int hp) {
        MAX_HP = hp;
    }
    //...
}

class GroundUnit extends Unit {
    GroundUnit(int hp) {
        super(hp);
    }
}
```

```
}

class AirUnit extends Unit {
    AirUnit(int hp) {
        super(hp);
    }
}

class Marine extends GroundUnit {
    Marine() {
        super(40);
        hitPoint = MAX_HP;
    }
    //...
}

interface Repairable {}

class Tank extends GroundUnit implements Repairable {
    Tank() {
        super(150);     // Tank의 HP는 150이다.
        hitPoint = MAX_HP;
    }

    public String toString() {
        return "Tank";
    }
    //...
}

class Dropship extends AirUnit implements Repairable {
    Dropship() {
        super(125);     // Dropship의 HP는 125이다.
        hitPoint = MAX_HP;
    }

    public String toString() {
        return "Dropship";
    }
    //...
}

class SCV extends GroundUnit implements Repairable {
    SCV() {
        super(60);
        hitPoint = MAX_HP;
    }

    void repair(Repairable r) {
        if (r instanceof Unit) {
            Unit u = (Unit)r;
            while(u.hitPoint!=u.MAX_HP) {
                /* Unit의 HP를 증가시킨다. */
                u.hitPoint++;
            }
        // repair의 매개변수 r은 Repairable타입이기 때문에 인터페이스 Repairable에
        // 정의된 멤버만 사용할 수 있다. 그러나 Repairable에는 정의된 멤버가 없으므로
        // 이 타입의 참조변수로는 할 수 있는 일은 아무것도 없다.
        // 그래서 instanceof연산자로 타입을 체크한 뒤 캐스팅하여 Unit클래스에 정의된
        // hitPoint와 MAX_HP을 사용할 수 있도록 하였다.
```

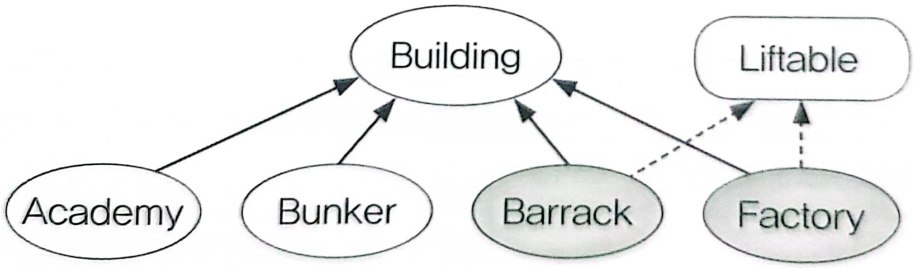
```
                System.out.println( u.toString() + "의 수리가 끝났습니다.");
            }
        }
        //...
}

class RepairableTest{
    public static void main(String[] args) {
        Tank tank = new Tank();
        Dropship dropship = new Dropship();

        Marine marine = new Marine();
        SCV scv = new SCV();

        scv.repair(tank);   // SCV가 Tank를 수리하도록 한다.
        scv.repair(dropship);
//      scv.repair(marine); 에러. marine은 Repairable인터페이스를 구현하지 않았다.
    }
}
```

```
// 또다른 예시로 조상 Building 클래스의 자손 Academy, Bunker, Barrack, Factory가 있을 때,
// Barrack클래스와 Factory클래스에 건물을 이동할 수 있는 새로운 메서드를 추가하고자 한다면,
// 조상 클래스 Building에 코드를 추가하면 다른 자손인 Academy와 Bunker클래스도 추가된 코드를
// 상속받으므로 안 된다. 이런 경우에도 인터페이스를 이요해서 새로 추가하고자 하는 메서드를
// 정의하는 인터페이스를 정의하고 이를 구현하는 클래스 LiftableImpl을 작성하면 된다.
// 이러게 함으로써 같은 내용의 코드를 Barrack클래스와 Factory클래스에서 각각 작성하지 않고
// LiftableImpl클래스 한 곳에서 관리할 수 있다. 이 클래스는 후에 다시 재사용될 수도 있다.
```



```
interface Liftable {
   /** 건물을 들어 올린다. */
   void liftOff();
   /** 건물을 이동한다. */
   void move(int x, int y);
   /** 건물을 정지시킨다. */
   void stop();
   /** 건물을 착륙시킨다. */
   void land();
}

class LiftableImp1 implements Liftable {
    public void liftOff() { /* 메서드 구현 */ }
    public void move(int x, int y) { /* 메서드 구현 */ }
    public void stop() { /* 메서드 구현 */ }
    public void land() { /* 메서드 구현 */ }
}

// Barrack과 Factory클래스가 Liftable인터페이스를 구현하도록 하고,
// 인터페이스를 구현한 LiftableImpl클래스를 클래스 내부에 포함시켜서
// 내부적으로 호출해서 사용하도록 한다.
// 이렇게 작성하므로써 같은 내용의 코드를 Barrack클래스와 Factory클래스에서
// 각각 작성하지 않고, LiftableImpl클래스 한곳에서만 관리할 수 있다.
// 그리고 작성된 Liftable인터페이스와 이를 구현한 LiftableImpl클래스는
// 후에 다시 재사용될 수 있을 것이다.

class Building {
    // ...
}

class Barrack extends Building implements Liftable {
    LiftableImp1 l = new LiftableImp1;
    public void liftOff() { l.liftOff(); }
    public void move(int x, int y) { l.move(x, y); }
    public void stop() { l.stop(); }
    public void land() { l.land(); }
    void trainMarine() { /* 내용 생략 */ }
}

class Factory extends Building implements Liftable {
    LiftableImp1 l = new LiftableImp1;
    public void liftOff() { l.liftOff(); }
    public void move(int x, int y) { l.move(x, y); }
    public void stop() { l.stop(); }
    public void land() { l.land(); }
    void makeTank() { /* 내용 생략 */ }
}
```

▶ 인터페이스의 이해

클래스를 사용한 쪽(User)와 클래스를 제공하는 쪽(Provider)이 있다.
메서드를 사용(호출)하는 쪽(user)에서는 사용하려는 메서드(Provider)의 선언부만 알면 된다(내용은 몰라도 된다).

직접적인 관계의 두 클래스는 한 쪽(Provider)이 변경되면 다든 한 쪽(User)도 변경되어야 한다는 단점이 있다.

```
// 클래스A(User)는 클래스B(Provider)의 인스턴스르 생성하고 메서드를 호출한다.
// 이 두 클래스는 서로 직접적인 관계에 있다.
// 이 경우 클래스A를 작성하려면 클래스B가 이미 작성되어 있어야 하고,
// 클래스B의 선언부가 변경되면 이를 사용하는 클래스A도 변경되어야 한다.

class A {
    public void methodA (B b) {
        b.methodB();
    }
}

class B {
    public void methodB() {
        System.out.println("methodB()");
    }
}

class InterfaceTest {
    public static void main(String args[]) {
        A a = new A();
        a.methodA(new B());     // methodB()
    }
}
```

그러나 User가 Provider을 직접 호출하지 않고 인터페이스를 매개체로 해서 User가 인터페이스를 통해서 Provider의 메서드에 접근하도록 하면, Provider에 변경사항이 생기거나 Provider와 같은 기능의 다든 클래스로 대체되어도 User는 전혀 영향을 받지 않도록(변경하지 않아도 되도록)하는 것이 가능하다. 두 클래스간의 관계를 직접적으로 변경하기 위해서는, 먼저 인터페이스를 이용해서 Provider의 선언과 구현을 분리해야 한다. 먼저 Provider에 정의된 메서드를 추상메서드로 정의하는 인터페이스를 정의한다. 그 다음에는 Provider가 인터페이스를 구현하도록 한다. User클래스를 인터페이스를 사용해서 작성한다.

```java
// 두 클래스간의 관계를 간접적으로 변경하기 위해 먼저 인터페이스를 이용해서
// 클래스B의 선언과 구현을 분리한다.


// 클래스B에 정의된 메서드를 추상메서드로 정의하는 인터페이스 I를 정의한다.
interface I {
    public abstract void methodB();
}

// 그 다음에는 클래스 B가 인터페이스 I를 구현하도록 한다.
class B implements I {
    public void methodB() {
        System.out.println("methodB in B class");
    }
}

// 이제 클래스A는 인터페이스I를 사용해서 작성할 수 있다.
// methodA가 호출될 때, 인터페이스 I를 구현한 클래스의 인스턴스(클래스B의 인스턴스)
// 를 제공받아야 한다.
class A {
    public void methodA(I i) {
        i.methodB();
    }
}

// 이제 클래스A와 클래스B는 'A-I-B'의 간접적인 관계로 바뀌었다.
// 클래스A는 인터페이스 I하고만 직접적인 관간계에 있기 때문에, 실제로 사용되는
// 클래스의 이름을 몰라도 되고 심지어는 실제로 구현된 클래스가 존재하지 않아도
// 문제되지 않는다. 클래스A는 오직 직접적이 관계에 있는 인터페이스 I의 영향만 받는다.
```

User가 인터페이스를 사용해서 작성되긴 하였지만, 매개변수를 통해서 인터페이스를 구현한 클래스의 인스턴스를 동적으로 제공받아야 한다. (예시 - 클래스 Thread의 생성자인 Thread(Runnable target))

```
class A {
   void autoPlay(I i) {
         i.play();
    }
}

interface I {
     public abstract void play();
}

class B implements I {
    public void play() {
         System.out.println("play in B class");
    }
}

class C implements I {
    public void play() {
         System.out.println("play in C class");
    }
}

class InterfaceTest2 {
    public static void main(String[] args) {
        A a = new A();
        a.autoPlay(new B()); // void autoPlay(I i)호출
        a.autoPlay(new C()); // void autoPlay(I i)호출
    }
}
```

또는 다음과 같이 제 3의 클래스를 통해서 제공받을 수도 있다. (예시 - JDBC의 DriverManager클래스) 인스턴스를 직접 생성하지 않고, 제 3의 클래스의 메서드를 통해 제공받는다. 이렇게 하면, 나중에 다른 클래스의 인스턴스로 변경되어도 User의 변경 없이 메서드만 변경하면 된다는 장점이 있다.

```
class A {
   void methodA() {
         // 제3의 클래스의 메서드를 통해서 인터페이스 I를 구현한 클래스의 인스턴스를 얻어온다.
         I i = InstanceManager.getInstance();
         i.methodB();
         System.out.println(i.toString()); // i로 Object클래스의 메서드 호출가능
    }
}

interface I {
     public abstract void methodB();
}

class B implements I {
    public void methodB() {
         System.out.println("methodB in B class");
    }

     public String toString() { return "class B";}
}

class InstanceManager {
   public static I getInstance() {
       return new B();
   }
}

class InterfaceTest3 {
    public static void main(String[] args) {
        A a = new A();
        a.methodA();     //methodB in B class
                         //class B
    }
}
```

그리고 인터페이스 I 타입의 참조변수 i로도 Object클래스에 정의된 메서드들을 호출할 수 있다는 것도 알아두자. i에 toString()이 정의되어 있지는 않지만, 모든 객체는 Object클래스에 정의된 메서드를 가지고 있을 것이기 때문에 허용하는 것이다.

▶ 디폴트 메서드와 static 메서드: JDK1.8부터 디폴트 메서드와 static메서드도 인터페이스에 추가할 수 있게 되었다. 인터페이스의 static메서드 역시 접근 제어자가 항상 public이며, 생략할 수 있다.

☞ 디폴트 메서드(default method): 추상 메서드의 기본적인 구현을 제공하는 메서드로, 추상 메서드가 아니기 때문에 디폴트 메서드가 새로 추가되어도 해당 인터페이스를 구현한 클래스를 변경하지 않아도 된다. 키워드 default을 붙이며, 추상 메서드와는 달리 일반 메서드처럼 몸통{}이 있어야 한다. 디폴트 메서드 역시 접근 제어자가 public이며, 생략 가능하다. 새로 추가된 디폴트 메서드가 기존의 메서드와 이름이 중복되어 충돌하는 경우, 이 충돌을 해결하는 규칙은 다음과 같다.

1. 여러 인터페이스의 디폴트 메서드 간의 충돌: 인터페이스를 구현한 클래스에서 디폴트 메서드를 오버라이딩해야 한다.
2. 디폴트 메서드와 조상 클래스 메서드 간의 충돌: 조상 클래스 메서드가 상속되고, 디폴트 메서드는 무시된다.

```java
class Child extends Parent implements MyInterface, MyInterface2 {
    public void method1() {
        System.out.println("method1() in Child"); // 오버라이딩
    }
}

class Parent {
    public void method2() {
        System.out.println("method2() in Parent");
        // default메서드와 조상 클래스 메서드 간의 충돌:
        // 조상 클래스의 메서드가 상속되고, default메서드는 무시된다.
    }
}

interface MyInterface {
    default void method1() {
        System.out.println("method1() in MyInterface");
        // 여러 인터 페이스의 default메서드 간의 충돌:
        // 인터페이스를 구현한 클래스에서 디폴트 메서드를 오버라이딩 해야 한다.
    }

    default void method2() {
        System.out.println("method2() in MyInterface");
        // default메서드와 조상 클래스 메서드 간의 충돌:
        // 조상 ㅋ르래스의 메서드가 상속되고, default메서드는 무시된다.
    }

    static void staticMethod() {
        System.out.println("staticMethod() in MyInterface");
    }
}

interface MyInterface2 {
    default void method1() {
        System.out.println("method1() in MyInterface2");
        // 여러 인터 페이스의 default메서드 간의 충돌:
        // 인터페이스를 구현한 클래스에서 디폴트 메서드를 오버라이딩 해야 한다.
    }

    static  void staticMethod() {
        System.out.println("staticMethod() in MyInterface2");
    }
}

class DefaultMethodTest {
    public static void main(String[] args) {
        Child c = new Child();
        c.method1();                    // method1() in Child
        c.method2();                    // method2() in Parent
        MyInterface.staticMethod();     // staticMethod() in MyInterface
        MyInterface2.staticMethod();    // staticMethod() in MyInterface2
    }
}
```

Resource: 자바의 정석