CHAPTER 11

인터페이스 꼭 알아야 하나?

객체와 객체의 중간에서 하나의 스펙으로 동작하는 인터페이스를 배워봅니다.

Java를 공부하면서 가장 넘기 어려운 고비 중의 하나가 인터페이스이라고 생각합니다. 제 경우에도 인터페이스를 제대로 이해하는데 꽤 많은 시간이 걸렸다고 생각합니다. 왜 이렇게 인터페이스를 이해하는 것이 어려웠을까요? 우선은 인터페이스를 모른다고 해서 프로그램을 아주 못 만드는 것은 아니기 때문일 겁니다. 상속이나 인터페이스를 모른다고 해서 프로그램 자체를 못 만드는 것이 아니고, 다만 좀 더 귀찮고 수고스러울 뿐이니까요. 어떻게든 결과만 나오게 한다면 굳이 상속이나 인터페이스 개념은 필요하지 않을 때가 더 많더군요.

두 번째는 인터페이스 자체를 써서 더 좋아지는 경우는 시스템이나 프로그램의 규모가 어느 정도 규모가 있고 확장성을 요구하는 상황인데, 프로그래밍을 처음 배우면서 그런 상황을 접하기는 쉽지 않기 때문입니다. 객체지향 프로그래밍을 한마디로 요약하자면 '재사용성(Reuse)'입니다. 기존의 코드를 수정하지 않으면서 점점 더 늘어나는 요구 사항과 문제를 해결해 나가면서 점점 눈을 떠야 하는데, 초급자들에게는 이런 기회가 열려 있지 않은 경우가 더 많습니다. 따라서 인터페이스를 사용할 만한 상황이 없으므로 실제로 사용해야 하는 상황에서도 쓰지 못하는 경우가 많습니다.

인터페이스에 대해서 우선 얘기할 만한 사항은 앞으로 여러분이 엔터프라이즈급 환경에서 개발하려면 인터페이스에 대한 이해가 필수라는 겁니다. 대규모 시스템의 프로그램 설계에서는 인터페이스가 적극적으로 쓰이기 때문에 인터페이스를 적절하게 사용할 수 없다면 프로그램의 설계 자체가 유연하지 못하게 되는 경우가 많습니다. 또한, 고급 서적들일수록 인터페이스를 많이 취급하고 있습니다. 디자인 패턴이라든가, 프레임워크라는 것들을 공부하고 제대로 이해하려면 적어도 인터페이스에 대한 기본 지식은 갖추고 있어야만 합니다.

개인적으로 인터페이스에 대한 학습은 두 가지 단계를 거치는 것이 좋다고 생각합니다. 우선은 인터페이스에 대한 기본적인 개념만 이해한 다음 Java의 API에서 인터페이스가 어떻게 사용되는지를 열심히 익히는 단계입니다. 이것을 통해서 인터페이스가 여러분이 코드를 만들 때 도움을 줄 수 있는 존재라는 것을 확신하는 단계입니다. 이 단계가 지나고 나면 자신이 필요한 인터페이스를 이해하는 단계에 접어들면 됩니다. 자신이 필요한 인터페이스를 정의하고 이것을 구현해보면서 더욱 깊이 이해하는 단계라고 할 수 있습니다. 따라서 이번 장에서는 여러분이 어떤 의미로 인터페이스를 받아들여야 하는가에 대해서 집중적으로 설명할 생각입니다.

1 인터페이스란 무엇인가?

인터페이스에 대해서 가장 많이 받는 질문 중 하나입니다. 인터페이스라는 것이 뭐 할 때 쓰이는 것이고 어떻게 쓰는 것인가라는 질문입니다. 자주 받는 질문에 대해서 간단히 대답하면 다음과 같습니다. 우선 사전적인 의미를 살펴보겠습니다.

참고

in · ter · face n.

1_ 경계면, 접점, 공유[접촉] 영역

▶ the interface between the scientist and society 과학자와 사회의 접점

2_ (이종 간의) 대화, 연락, 의사 소통

3_【컴퓨터】 인터페이스《CPU와 단말 장치와의 연결 부분을 이루는 회로》

— vt.

1_ 조화시키다, 조정하다

2_【컴퓨터】 인터페이스로 접속[연결]하다

— vi. 결부하다, 조화되다

출처: 엔싸이버 영어사전

사전적인 의미를 보시면 주로 어떤 객체와 객체의 중간에 놓이는 것을 인터페이스라고 한다는 것을 알 수 있습니다. 즉 인터페이스라는 것을 그림으로 보면 객체와 객체의 중간에 놓이는 통신 채널이라고 이해하시면 좋겠습니다.

이제 인터페이스에 흔히 하는 질문들에 대해서 몇 가지 살펴보도록 합니다.

- **인터페이스는 뭐 할 때 쓰는 건가?**
  - 둘이서 서로 다른 클래스를 만들 때 서로 "이렇게 만들자, 이렇게 만들어 드릴게요."라면서 약속할 때 쓰이는 것이 인터페이스입니다. 즉 클래스를 만들기 전에 앞으로 이런 기능을 만들겠다, 이런 기능이 있었으면 좋겠다고 먼저 합의해놓는 것이 인터페이스입니다.
- **인터페이스는 어떻게 쓰는 것인가?**
  - 먼저 필요한 약속을 먼저 잡고, 한쪽에서는 약속대로 호출하고, 한쪽에서는 약속대로 구현해서 사용합니다. 인터페이스를 구현하는 입장에서는 약속한 대로 기능(메소드)을 만들어 주고, 사용하는 입장에서는 실제 객체가 어떻든 간에 약속을 이행할 수 있는 객체이기만 하다면 마음대로 약속된 기능을 호출하는 방식입니다.
- **인터페이스는 새로운 기능인가? 새로운 문법인가?**
  - 인터페이스가 네트워크 연결을 하거나, 데이터베이스 처리를 하는 기능은 절대 아닙니다. 다만, 프로그램을 설계하고 조금 더 유연한 프로그램을 만드는 기법입니다. 인터페이스는 상속과 더불어서 다형성(Polymorphism)이라는 객체지향 프로그래밍의 특징을 구현하는 방식입니다.
- **인터페이스를 알면 어떤 이로움이 있는가?**
  - 다른 객체지향 언어를 공부할 때 상당히 편리합니다. 예를 들어 닷넷 진영의 C#이나, Flex의 ActionScript와 같은 것을 공부할 때에도 인터페이스와 상속은 완전히 같은 내용입니다. 따라서 요즘 나오는 언어들이 객체지향이라는 것을 생각해 보면 꽤 많은 도움을 줄 수 있다고 생각합니다.
- **인터페이스를 공부하는 데 어느 정도의 시간이 걸리는가? 어떤 목표를 가지고 공부해야 하는가?**
  - 정보와 지식은 그 수준이 다릅니다. 정보가 체계적으로 쌓여서 자신이 마음대로 응용할 수 있는 수준을 지식이라고 할 수 있습니다. 인터페이스에 대해서 정보를 얻는 것은 어렵거나 시간이 오래 걸리는 일은 아닙니다. 다만, 그 개념이 정확히 본인의 코드에 녹아서 반영되기까지는 상당한 시간이 요구된다고 할 수 있습니다.
- **인터페이스를 알면 웹이나 다른 개념을 공부할 때 도움이 되는가?**
  - 물론입니다. 웹이나 프레임워크를 제대로 이해하는 데 있어 어떻게 인터페이스가 쓰였는지 아는 것과 모르는 것은 큰 차이가 납니다. 즉 단순히 사용법을 배우는가? 아니면 개념과 원리를 이해하면서 사용하는가의 문제입니다.

인터페이스는 다른 용어들과는 달리 프로그램이 아닌 다른 곳에서도 많이 사용되기 때문에 프로그래밍에서 어떤 의미로 사용하는지에 대한 이해가 필수적입니다.

2 인터페이스는 실생활에서 많이 사용됩니다.

인터페이스의 문법을 배우는 것은 아주 간단합니다. 오히려 제일 어려운 것은 인터페이스가 실제로 우리의 생활 속에서 어떻게 반영되어 있는지를 이해하는 것입니다. 이 작업이 필요한 이유는 여러분이 코드에 인터페이스를 언제 적용할 것인지 판단하는 데 있어서 도움을 줄 수 있기 때문입니다.

■ **상황① 당신과 친구가 서로 약속을 한다면: 객체의 스펙**

여러분과 제가 같이 프로그래밍을 작성한다고 합시다. 제가 여러분이 사용할 객체를 만들어준다면 아마도 여러분은 제가 프로그래밍이 다 완료될 때까지 기다리고 있어야만 할 겁니다. 모든 작업이 끝나고 나서야 코딩할 수 있을 겁니다. 제가 코딩을 다 마칠 때까지 여러분은 집에 갈 수도 없고, 애인과의 약속을 지킬 수 없는 상황도 발생할 겁니다. 이것은 상당히 낭비가 아닐까요? 그래서 아예 이렇게 해주면 어떨까요?

처음부터 제가 어떤 메소드를 만들 것인지를 먼저 정하는 겁니다. 어떤 메소드에는 어떤 파라미터를 받게 할 것이고, 어떤 메소드에는 어떤 리턴 값을 이용할 것이다 등 이런 논의를 먼저 하게 되면 여러분은 적어도 제가 만든 클래스와 객체가 어떤 메소드를 가졌는지 알게 될 것이고, 바로 당장은 아니더라도 프로그램을 개발하는 데 있어서 분명히 도움이 되지 않을까요?

그림 1

논의를 해서 얻을 수 있는 이익이 하나 더 있습니다. 만일 제가 갑자기 어떤 일이 생겨서 프로그램을 만들 수 없게 되면 다른 사람에게 논의된 자료를 알려주면 됩니다. 즉 논의된 대로만 작업을

진행한다면 나머지는 다른 사람이 만들어도 여러분의 코드는 크게 변경될 필요가 없을 것입니다. 인터페이스는 이럴 때 사용될 수 있습니다. 즉 여러분이 해야 하는 일을 정확하게 구분하고, 상대방에 해야 할 일을 명시해주는 작업을 해야 한다면 인터페이스를 이용해야 한다고 생각하시면 됩니다. 이러한 의미에서 보면 인터페이스는 일종의 객체의 계약서(스펙)라는 의미가 있습니다.

■ 상황② 오늘 점심 뭐 먹지?: 스펙을 만족하는 객체

밖에서 식사를 자주 하는 분이라면 점심 메뉴로 고민해보신 적이 있을 겁니다. 이런 고민은 주로 식당이 많을 때 일어납니다. 그리고 나서 여러분은 다양한 판단의 기준을 만들어 결정하는 데 사용합니다. 예를 들어 어제는 중국집에 갔으니 오늘은 제외하고, 오늘은 날씨가 쌀쌀하니 국물 요리를 먹었으면 좋겠고, 몸이 피곤하니 음식이 빨리 나오는 집으로 갔으면 좋겠고, 가격은 어느 정도 하는 집 등등 많은 기준이 있습니다.

프로그래밍에서도 마찬가지입니다. 여러 객체 중에서 여러분이 원하는 기능을 가진 객체의 스펙을 만족하게 한다는 객체를 만들어주고자 하는 것이 인터페이스입니다. 즉 어떤 객체가 여러분이 선택한 기준의 기능들을 다 구현하고 있다고 생각하시면 됩니다. 따라서 인터페이스를 구현한다는 의미는 어떤 객체가 어떤 기준을 만족하게 한다는 의미로 해석할 수 있습니다.

> 입사 기준에 만족하는 사람은 여러 명 있을 수 있습니다. 실생활에서 어떤 기준을 만족하듯이 객체들이 어떤 기준을 만족하게 하는 장치가 바로 인터페이스의 용도 중의 하나입니다.

■ 상황③ 꿩 대신 닭: 현재 객체의 대체물

만일 여러분이 다른 이성에게 친구를 소개해주기로 약속했다고 생각해봅시다. 키가 얼마나 되고, 얼굴은 어떻고, 기타 등등의 묘사를 해 놓고 난 후에 막상 소개팅 날 소개해주기로 한 친구가 일이 있어서 문제가 생길 수 있습니다. 이럴 때 소위 '땜빵' 역할을 할 친구가 필요합니다.

프로그래밍에서도 어떤 기준에 만족한 객체를 이용해서 프로그래밍을 만들다가 새로운 버전이나 다른 객체로 변경하는 일도 자주 일어납니다. 이럴 때 기존의 메소드를 전면 수정하게 되면 결국은 모든 코드의 내용을 수정해야 하는 일이 발생합니다.

인터페이스를 이용한다는 것은 인터페이스를 구현한 객체들을 하나의 부속품처럼 마음대로 변경해서 사용하는 것이 가능하다는 것입니다.

자동차의 순정부품이 있긴 하지만, 때로는 다른 부품을 결합하기도 하는 것처럼, 인터페이스는 시스템이 확장되면서 발생하는 잦은 변경의 문제를 해결하는 데 사용합니다.

> 인터페이스를 코드에 적용하게 되면 실제 객체의 클래스를 몰라도 프로그램을 코딩할 수 있습니다. 따라서 더 유연한 프로그램을 설계할 수 있습니다.

■ **상황④ 호텔 떡볶이와 길거리 떡볶이: 전혀 다른 객체의 같은 기능**

실제로 시내의 모 호텔에 갔더니 떡볶이를 파는 것을 보고 한참을 재밌게 여긴 기억이 있습니다. 떡볶이는 길거리 음식이라고 생각했는데 버젓이 호텔의 메뉴판에 있는 것을 보니 신기하기도 했습니다. 호텔 요리사와 노점에서 장사하는 사람, 이 두 사람은 서로 다른 객체이지만 같은 기능을 할 수 있는 존재들입니다. 즉 필요하다면 위에서 말한 하나의 부속품처럼 두 객체를 시스템에서 마음대로 사용할 수 있어야만 합니다.

인터페이스는 하나의 기능에 대한 약속이기 때문에 중요한 점은 어떤 객체이든 간에 그 약속을 지키기만 한다면 필요한 곳에서 사용할 수 있게 한다는 것을 의미합니다. 몇 가지의 상황에 인터페이스라는 것이 적용되는 것을 설명해보았습니다. 이런 상황에 대해 이해하는 것이 우선이고, 코드와 문법은 그 뒤에 고려해도 충분합니다. 이제 각각의 상황에 맞추어 인터페이스를 설명해 보도록 합니다.

3 문법으로 알아보는 인터페이스

■ **인터페이스는 실제 객체를 의미하지 않습니다.**

우선 여러분이 가장 먼저 기억해야 하는 사실은 인터페이스는 그 자체가 객체를 의미하지 않는다는 겁니다. 앞에서 예를 든 상황을 보면 인터페이스라는 것은 결국은 어떤 객체가 할 수 있는 기능 자체를 의미하고, 그 기능을 하나의 스펙으로 모은 것에 불과합니다. 따라서 인터페이스가 실제 기능을 가지는 것이 아닙니다. 즉 실제로 구현된 코드를 가지지 않았다는 겁니다. 인터페이스는 실제로 구현된 코드 대신에 오로지 추상 메소드와 상수만을 가지고 있게 됩니다.

■ **인터페이스의 상수는 private으로 만들 수 없습니다.**

인터페이스는 실제 객체는 아니지만 서로 간의 약속으로 사용됩니다. 간단히 예를 들어 생각해보면 중식당에서 주방장과 주인아주머니 사이에는 몇 가지 암묵적인 약속이 존재합니다. 예를 들

어 '짐'이라고 말하면 짐뽕을 의미한다든가 하는 약속들입니다. 만일 이 단어로 정해진 약속을 한쪽에서 일방적으로 수정하게 되면 문제가 발생할 수 있습니다. 인터페이스는 객체와 객체의 중간에 놓이기 때문에 인터페이스에 선언하는 변수는 자동으로 'public static final'이 됩니다. 즉 완벽한 상수로 정의되게 됩니다.

반면에 private은 객체를 만들 때 외부 클래스에서는 접근할 수 없게 하려고 사용하기 때문에 외부로 공개되기 위해서 사용하는 인터페이스에는 맞지 않습니다. 그래서 실제로 인터페이스에 private으로 시작하는 변수는 선언할 수 없습니다.

■ **인터페이스에는 추상 메소드만 존재합니다.**

인터페이스는 실제 객체로 만들어 지지 않습니다. 즉 우리가 원하는 어떤 기능들을 모아서 하나의 인터페이스로 선언하는 것이 가장 일반적으로 사용되는 용도이기 때문에 인터페이스는 '기능의 묶음'이라고 해석하는 것이 편리합니다.

Java의 API 안에 있는 인터페이스를 보면 유독 뒤의 단어가 '-able'로 끝나는 경우가 많은 데 이 역시 우연이 아닙니다. 상속을 영어로 해석할 때 'is a relation'이라고 한다면, 인터페이스는 'has a relation'으로 해석됩니다. 즉 어떤 객체가 어떤 인터페이스를 구현했다는 것은 인터페이스에서 정의한 기능들을 그 객체가 모두 구현해두었다는 것을 의미합니다. 따라서 실제 객체는 모든 메소드를 구현했겠지만, 인터페이스 자체에는 메소드의 선언만 들어 있게 됩니다.

인터페이스의 문법적인 구성

- 인터페이스의 선언
- 상수 선언부
- 추상 메소드 선언부

■ **인터페이스는 객체의 타입으로만 사용됩니다.**

인터페이스는 실제 객체로 사용되지는 않지만, 객체의 타입으로는 사용됩니다. 이 말은 상속에서와같이 변수 선언 시에 객체의 타입으로 인터페이스를 사용할 수 있다는 것을 의미합니다.

```
인터페이스 a = new 인터페이스 구현 객체( );
```

이 경우에 컴파일러는 실제로 변수를 사용할 때 변수의 타입만을 보기 때문에 a라는 변수를 이용해서 객체의 메소드를 호출하는 작업을 실행하면 컴파일러는 인터페이스에 호출하려는 메소드가 있는지만을 따지게 됩니다. 상속에서 타입이 부모 클래스일 때 컴파일러가 부모 클래스에 선언된 메소드만 따지는 것과 같은 방식이라고 생각하시면 됩니다. 인터페이스 역시 객체의 타입으로 선언될 수 있기 때문에 컴파일러는 변수의 타입에 선언된 메소드가 존재하는지만 따지게 되고, 실제로 호출되는 것은 실행되고 있는 객체의 메소드가 호출됩니다.

4 인터페이스 직접 만들기: '당신이 원하는 기능'을 정의합니다.

우리가 원하는 MP3의 기능들을 다음과 같이 추려냈다고 가정해봅니다.

- mp3 파일을 들을 수 있어야 한다.
- 사진을 볼 수 있어야 한다.
- FM도 들을 수 있어야 한다.

이 기능들을 인터페이스라는 것으로 선언하는 겁니다. 다시 한번 말씀드리지만, 인터페이스 그 자체는 객체가 아닙니다. 따라서 기준에 맞는 제품들은 어떤 것인지 고려하지 말고, 선언만 합니다. 그 자체가 객체가 아니므로 선언 시에도 역시 추상 메소드처럼 빈 메소드만 선언합니다.

예제 | 원하는 기준을 인터페이스 만든 MyMP3 인터페이스

```
package org.thinker.mp3;

public interface MyMP3 {
        public void playMp3();
        public void listenFM();
        public void viewPhoto();
}
```

4.1 기능을 구현한 객체를 만들 때는 'implements 인터페이스 이름'

위의 기준을 만족하는 객체를 만들어봅니다. 만들어 볼 제품은 TONY, BPPLE사의 제품이라고 생각해봅니다. 어떤 클래스를 만들 때 원하는 기준을 만족시키기 위해서 만든다면 클래스의 선언 뒤에 implements 기준(인터페이스)으로 선언하시면 됩니다. implements의 뜻이 '구현하다'는 뜻이기 때문에 어떤 기준을 구현한다고 생각하시면 됩니다.

예제 MyMP3라는 기준을 구현한 TonyMP3 클래스

```
package org.thinker.mp3;

public class TonyMP3 implements MyMP3 {
	@Override
	public void listenFM() {
		System.out.println("tony 제품은 FM 수신 가능");
	}

	@Override
	public void playMp3() {
		System.out.println("tony 제품은 mp3 지원");
	}

	@Override
	public void viewPhoto() {
		System.out.println("tony 제품은 이미지 뷰어 제공");
	}
}
```

하나 더 BPPLE사의 제품도 만들어 볼까 합니다.

예제 마찬가지로 MyMP3 기능을 만족하는 Bpple사의 제품

```
package org.thinker.mp3;

public class BppleMP3 implements MyMP3 {
	@Override
	public void listenFM() {
		System.out.println("Bpple 제품은 FM 일부 가능");
	}

	@Override
	public void playMp3() {
		System.out.println("Bpple 제품은 mp3 가능");
	}

	@Override
	public void viewPhoto() {
		System.out.println("Bpple 제품은 사진 보기 가능");
	}
}
```

이제 여러분의 기준에 맞는 제품이 두 가지나 있습니다. 이제 마지막으로 제품을 사용해 볼까요? 실제로 테스트를 해보도록 합니다.

예제

```
package org.thinker.mp3;

public class MP3Test {
        public static void main(String[] args) {
                MyMP3 mp3 = new TonyMP3();
        }
}
```

위의 코드는 에러가 발생할까요? 'MyMP3 mp3 = new TonyMP3( );'라는 구문은 성립될까요? 위의 구문을 이렇게 해석해보시면 됩니다. 'MyMP3 기준을 만족하는 제품 TonyMP3 객체' 이런 해석으로 보면 성립되는 구문입니다. 이것이 바로 인터페이스를 변수 타입으로 선언한다는 의미입니다.

> 반드시 기억해야 하는 사실은 컴파일러가 따지는 것은 단순히 변수의 타입이라는 겁니다. 즉 'MyMP3 mp3 = new TonyMP3();'에서 컴파일러는 뒤의 객체가 MyMP3의 기능이 구현된 객체이기만 하면 됩니다. 때문에 mp3.xxx를 호출할 때 컴파일러가 따지는 것은 인터페이스인 MyMP3에 호출하려는 xxx가 존재하는지만이 중요할 뿐입니다.

■ 컴파일러는 변수의 타입만을 봅니다.

컴파일러가 따지는 것은 변수의 타입이라는 사실을 기억하십니까? 실제 객체가 무언지가 중요한 것이 아닙니다. 중요한 것은 변수의 타입입니다. 즉, 위의 선언 이후의 코드는 MyMP3에 선언된 메소드이기만 하면 된다는 겁니다.

코드

```
MyMP3 mp3 = new TonyMP3();
mp3.listenFM();
mp3.playMp3();
mp3.viewPhoto();
```

```
tony 제품은 FM 수신 가능
tony 제품은 mp3 지원
tony 제품은 이미지 뷰어 제공
```

컴파일러는 MyMP3라는 변수의 타입으로 선언된 것을 따지지만, 실행 시점에 실제로 움직이는 것은 TonyMP3 클래스의 객체이므로 실제로 실행을 하게 되면 실행되는 것은 TonyMP3 클래스의 구현된 메소드의 내용이 실행됩니다.

■ **컴파일러는 MyMP3만 보고 있으니...**

컴파일러는 MyMP3라는 인터페이스만 보고 있습니다. 그렇다면 'new TonyMP3( );' 대신 'new BppleMP3( );'는 가능할까요?

코드 변수의 선언은 그대로 두고 실제 객체를 변경해도 아무런 문제가 없습니다.

```
MyMP3 mp3 = new BppleMP3();
mp3.listenFM();
mp3.playMp3();
mp3.viewPhoto();
```

```
Bpple 제품은 FM 일부 가능
Bpple 제품은 mp3 가능
Bpple 제품은 사진 보기 가능
```

어떻습니까? BppleMP3 제품으로 변경되어도 아무런 문제가 없습니다. **인터페이스는 어떤 객체를 직접 알고서 호출하는 방식이 아니라, 어떤 객체가 어떤 기능적인 스펙을 만족한다**는 것을 더 중요하게 생각합니다. 따라서 인터페이스를 이용해서 어떤 호출을 할 때 중요한 것은 실제 객체가 무엇인가가 중요한 것이 아니라 인터페이스에 해당하는 메소드가 존재하는지가 더 중요합니다.

4.2 인터페이스를 이용할 때에는 다음 순서대로 만들어주세요.

흔히 초급 개발자들을 보면 인터페이스의 문법을 몰라서 인터페이스를 제대로 활용하지 못하는 경우는 거의 없습니다. 인터페이스를 어떤 시점에 어떻게 적용해야 할지 모른다는 것이 가장 정확한 원인이라고 생각합니다. 그럼 여러분이 인터페이스를 이용해서 코딩하려면 어떤 내용을 가지고 어떤 순서에 맞춰야 하는지를 설명해볼까 합니다.

4.2.1 단계① 기능을 인터페이스로 정의한다.

우선 인터페이스를 제대로 활용하려면 객체 위주의 생각에서 조금 더 발전해서 기능 위주의 생각으로 시작해야 합니다. 무슨 말인지 어렵다면 다음과 같은 상황을 생각해보시면 됩니다.

■ **계약 방식으로 일하는 패스트푸드 음식점**

'버거퀸'이라는 가게에 가보면 여러 직원이 때로는 점원으로, 때로는 주방에서 일하는 모습을 볼 수 있습니다. 기회가 되시면 지점에서 어떻게 일을 하게 되는지를 살펴보시면 인터페이스를 이해하는 데 많은 도움이 됩니다. 자세히 살펴보시면 고객이 주문했을 때 점원이 직접 주방에 얘기하지는 않습니다. 점원은 지정된 방식으로 쪽지를 붙여주거나, 시스템을 통해서 데이터를 입력해줍니다. 입력된 데이터에 따라 주방에서 일하는 주방 직원이 이미 약속한 대로의 지정된 메뉴를 만들어 주게 됩니다. 이때 중요한 점은 손님을 만나는 점원은 실제로 주방에서 어떤 점원이 어떻게 만드는지 전혀 모른다는 겁니다. 엄밀하게 말하자면 알 필요가 없습니다. 점원은 지정된 방식으로 주방에 있는 어떤 객체에 일을 시키고, 주방은 약속된 대로 일을 해주는 겁니다. 이 경우 직원과 주방 사이에 있는 존재를 인터페이스라고 생각해주면 됩니다.

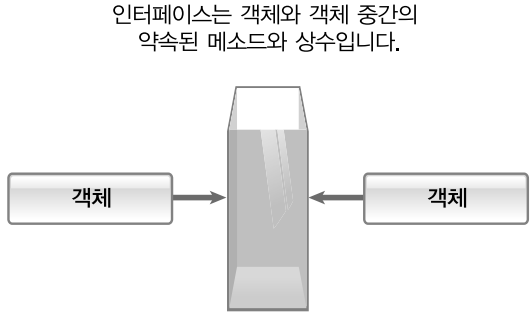


그림 2

■ **진짜 객체가 아니라 어떤 객체가 했으면 하는 기능을 정의하세요.**

인터페이스는 객체와 객체의 중간 역할을 하기 때문에 우선은 양쪽에서 호출하고, 실행할 기능을 명확히 구분하는 작업에서 시작하는 겁니다. 즉 햄버거 가게를 예로 들자면 인터페이스를 다음과 같은 기능으로 설계할 수 있습니다.

- 주방은 원하는 햄버거를 제공한다.
- 주방은 샐러드를 제공한다.

점원의 요구 사항은 위의 두 가지입니다. 이것을 인터페이스로 구현해보도록 합니다.

예제 | 주방에서 일하는 객체들에 대한 요구 사항을 정의한 계약서인 BurgerCook 인터페이스

```
package org.thinker.burgerqueen;

public interface BurgerCook {
        public void makeBurger();
        public void makeSalad();
}
```

인터페이스는 거듭 강조하지만 '~ 기능'에 대해서만 정의하는 겁니다.

4.2.2 단계② 인터페이스를 구현한 실제 클래스를 만든다.

인터페이스를 이용해서 하나의 기능에 대한 정의를 내렸다면 그다음으로 할 일은 실제로 그 기능을 구현해서 실행할 수 있는 객체를 만들어 내는 클래스를 정의하는 겁니다. 여기서는 BurgerCook 인터페이스의 기준을 만족하는 객체 HotelCook과 InstantCook이라는 클래스가 어떻게 BurgerClerk과 연결되는지를 만들어보도록 합니다.

예제 | BurgerCook 기준을 만족하는 HotelCook 클래스와 InstanceCook

```
package org.thinker.burgerqueen;

public class HotelCook implements BurgerCook {
        @Override
        public void makeBurger() {
                System.out.println("5성급 호텔 주방장의 햄버거");
        }

        @Override
        public void makeSalad() {
                System.out.println("호텔급 샐러드");
        }
}
```

```
package org.thinker.burgerqueen;

public class InstantCook implements BurgerCook {
	@Override
	public void makeBurger() {
		System.out.println("냉동 햄버거");
	}

	@Override
	public void makeSalad() {
		System.out.println("냉동 샐러드");
	}
}
```

너무 극단적인가요? HotelCook 객체는 호텔급 음식을 만드는 반면에 InstantCook은 주로 냉동 식품이군요. 하지만, 중요한 점은 두 객체 모두 BurgerCook이라는 기준을 만족하는 객체라는 겁니다. 따라서 이제 최종적으로 BurgerClerk과 연결해서 실행하는 코드를 만들어보도록 합니다.

예제 | BurgerTest

```
package org.thinker.burgerqueen;

public class BurgerTest {
	public static void main(String[] args) {
		BurgerCook cook = new HotelCook();
		cook.makeBurger();
		cook.makeSalad();
	}
}
```

```
5성급 호텔 주방장의 햄버거
호텔급 샐러드
```

실행된 결과를 보면 지금 현재 주방 뒤편에는 호텔 주방장이 있기 때문에 훌륭한 요리가 나오는 것을 보실 수 있습니다. 그림으로 현재 코드의 관계를 보면 다음과 같이 만들어질 수 있습니다.

호출하는 쪽에서 인터페이스만 알 수 있도록 하면
뒤에 있는 실제 객체에는 신경 쓰지 않는다.

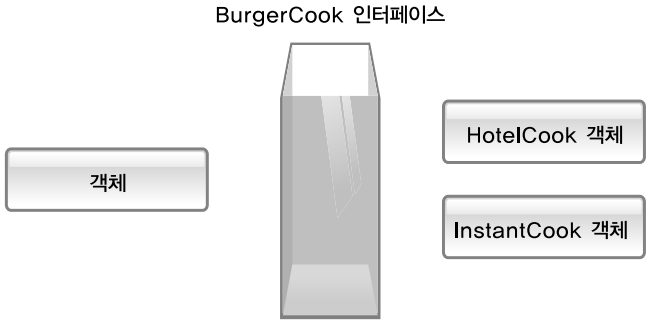


그림 3

■ **변수의 선언을 유심히 보세요.**

앞의 코드에서 변수의 선언을 유심히 보면 변수의 타입이 BurgerCook으로 선언된 것이 보입니다. 변수의 타입이 BurgerCook이라는 것은 다른 의미로는 컴파일러가 따지는 변수에 선언된 메소드가 BurgerCook에 정의된 메소드이기만 하면 된다는 뜻입니다.

그러므로 실제 객체가 어떤 객체이든 간에 그 객체가 기능의 스펙인 인터페이스를 구현하고 있다면 얼마든지 올 수 있다는 것을 의미합니다. 이것은 시스템이 점점 더 많은 클래스를 가지고 점점 더 복잡해질 때 코드를 수정하는 부분을 최소화할 수 있습니다. 즉, 앞의 코드에서 HotelCook 대신에 InstantCook을 넣어주어도 변수의 타입이나 메소드의 호출은 수정할 필요가 없다는 점에 주목해주시기 바랍니다.

4.2.3 단계③ 객체가 필요한 곳에서는 인터페이스만 보고 코딩하자.

인터페이스의 설계, 인터페이스의 구현 클래스 작성 이후는 이제 객체를 사용하는 코드에서 실제 객체를 사용하지 않고, 인터페이스만 보고 코드를 작성해주는 겁니다.

예제 | BurgerCook의 약속을 최대한 이용하는 점원 BurgerClerk 클래스

```
package org.thinker.burgerqueen;

public class BurgerClerk {

        private BurgerCook cook = new HotelCook();
```

```
        public void orderBurger(){
                System.out.println("주방에 햄버거를 주문합니다.");
                cook.makeBurger();
        }

        public void orderSalad(){
                System.out.println("주방에 샐러드를 주문합니다.");
                cook.makeSalad();
        }
}
```

클래스의 이름에서 보시는 것과 같이 햄버거 가게의 점원 정도 되는 클래스입니다. 위의 코드를 자세히 보시면 변수 선언 시에 인터페이스만을 사용하고 있습니다. 즉 나중에 코드를 수정하고 싶다면 HotelCook( ) 부분을 InstantCook처럼 원하는 클래스로 대체할 수 있습니다. 중요한 사실은 대체한 후에는 나머지의 코드는 전혀 손댈 필요가 없다는 점이 가장 중요합니다.

> 클래스와 클래스의 관계가 정확한 클래스로 묶이는 것이 아니라 상속이나 인터페이스를 이용한 다형성을 구현해서 묶는 것을 느슨한 커플링(Loose Coupling)이라고 합니다. 느슨한 커플링으로 관계를 구성해두면 나중에 시스템상에서 클래스를 변경할 때 더 유연해지도록 구성할 수 있습니다.

■ **인터페이스를 이용하면 나중에 시스템이 변경될 때 유연해질 수 있습니다.**

여러분의 시스템이 나중에 새로운 종류의 객체가 나온다고 해도 인터페이스를 이용한다면 너무 걱정할 필요 없습니다.

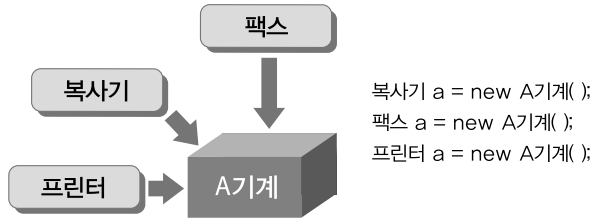
- 새로 만들어지는 클래스 역시 인터페이스를 구현해 주었다면 바로 적용할 수 있을 것이다.
- 어떤 객체이든 인터페이스의 규약에 맞는 객체는 사용할 수 있다.

인터페이스는 주로 현재를 보고 만들기보다는 시스템의 규모가 커지거나, 다른 사람들과 같이 협업을 해야 할 때 효과를 발휘하는 것이 일반적입니다.

5 인터페이스는 하나의 클래스에 여러 개를 붙여줄 수 있습니다.

인터페이스는 객체와 관련된 것이 아니라 객체가 할 수 있는 기능에 대한 정의이기 때문에 상속과는 달리 하나의 객체가 여러 가지의 기능의 스펙을 구현하는 것도 이상한 일이 아닙니다. 즉 Java에서는 단일 상속만을 지원했던 것과는 달리 하나의 객체는 여러 가지 종류의 기능 스펙인 인터페이스를 구현할 수 있습니다.

조금 더 쉽게 이해하려면 복합기를 생각해보시면 됩니다. 복합기는 프린터도 되고, 팩스도 되고, 복사도 됩니다. 즉 객체는 하나인데 이 객체가 여러 개의 기능 스펙을 구현하고 있다는 겁니다.



하나의 객체를 여러 타입(모습)으로 보는 것이 다형성의 핵심입니다.

그림 4

■ **인터페이스는 변수 타입으로 가능하니까 하나의 객체를 여러 가지 타입으로?**

위의 그림으로 보고 변수의 선언을 생각해 보면 '프린터 a = new 마이복합기( );' 혹은 '팩스 a = new 마이복합기( );', '복사기 a = new 마이복합기( );'가 가능한 것을 보실 수 있습니다. 이런 때에는 객체는 여러 가지의 인터페이스를 구현하는 형태가 됩니다. 실제로 인터페이스를 적극적으로 사용하게 되면 인터페이스의 개수가 점점 더 많아질 겁니다. 인터페이스 위주로 프로그램을 설계한다는 것은 만들어야 하는 기능에 대한 기준을 미리 잡는다는 것과 유사어 정도라고 생각합니다. 또 다른 예제로 요즘 나오는 3G 휴대전화의 기능들을 목록으로 뽑아보자면 다음과 같이 만들어 볼 수 있을 듯합니다.

- 음성통화 기능
- 화상통화 기능

다들 아시겠지만, 3G폰이라고 하는 것들은 위에서 화상통화의 기능까지 제공되어야 3G폰이라 부릅니다.

■ **각 기능을 인터페이스로 정의해봅니다.**

음성통화 기능과 화상통화 기능을 인터페이스로 정의해보자면 다음과 같이 만들어집니다. 음성통화 기능을 VoiceCall이라고 하고, 화상통화 기능을 VisualCall이라고 하겠습니다.

예제

```
package org.thinekr.phone3g;

public interface VoiceCall {
        public void sendVoiceMsg();
        public void getVoiceMsg();
}
```

VisualCall 인터페이스는 다음과 같이 만들었습니다.

예제

```
package org.thinekr.phone3g;

public interface VisualCall {
        public void sendVisualMsg();
        public void getVisualMsg();
}
```

■ **Phone3G 클래스는 두 인터페이스의 모든 기능이 필요한데요?**

만일 3G폰을 만들려면 두 인터페이스의 기능이 둘 다 필요합니다. 이 경우에 할 수 있는 방법은 두 가지가 있습니다.

- Phone3G 클래스가 인터페이스를 두 개 구현하는 방법
- VoiceCall과 VisualCall 인터페이스를 Call3G라는 인터페이스 하나로 묶고, Phone3G 클래스가 Call3G라는 인터페이스만 구현하는 방법

둘 다 설명할 내용이 많으므로 하나씩 설명해야 할 듯합니다.

■ **Phone3G 클래스가 인터페이스 두 개를 구현하게 하는 방법: 다중 구현**

Java는 C++의 다중 상속의 애매모호함을 제거하고자 단일 상속 체계만을 지원합니다. 따라서 Java에서는 클래스의 extends 뒤에는 하나의 부모 클래스만 올 수 있습니다.

> 상속은 구체적으로 구현된 메소드를 물려주기 때문에 다중 상속을 하게 되면 문제가 생깁니다. 하지만, 인터페이스는 스펙(추상 메소드)만을 물려주기 때문에 여러 개의 인터페이스의 같은 메소드를 물려받아도 구현은 실제 구현 클래스 한 곳에서만 합니다.

상속을 통해서 부모의 속성과 동작을 물려받는 것은 개발자에게는 상당히 매력있는 방법입니다. 조금 더 솔직히 말해서 강력한 유혹입니다. 코딩을 해야 할 필요가 없어지기 때문입니다. 하지만, 불행히도 Java는 이 방식을 취하지 않았기 때문에 다른 형태로 다중 상속과 비슷하게 구현하는 수밖에 없습니다.

상속과 달리 인터페이스는 하나의 기준이면서 '~를 할 수 있는 기능을 가진 객체'의 의미로 해석될 수 있습니다. 인터페이스 자체가 하나의 클래스가 아니고, 그저 객체를 판단하는 기준이기 때문에 하나의 객체가 여러 가지 기능을 가질 때에는 클래스의 선언 뒤에 여러 개의 인터페이스를 ','를 이용해서 사용할 수 있습니다.

예제 | Phone3G1 클래스가 두 개의 인터페이스를 구현하도록 만드는 방식

```
package org.thinekr.phone3g;

public class Phone3G1 implements VoiceCall, VisualCall{
	@Override
	public void getVoiceMsg() {
		System.out.println("음성 메시지 듣기");
	}

	@Override
	public void sendVoiceMsg() {
		System.out.println("음성 메시지 보내기");
	}

	@Override
	public void getVisualMsg() {
		System.out.println("영상 보내기");
	}
```

```
        @Override
        public void sendVisualMsg() {
                System.out.println("영상 보기");
        }
}
```

Phone3G1 클래스 선언 뒤에 인터페이스가 2개 나온 것이 보입니다. 이런 경우 Phone3G1의 선언은 VoiceCall 기능을 구현하고 있으면서, VisualCall 기능도 구현하고 있다고 해석하시면 됩니다. 각 인터페이스는 고유의 메소드들이 있기 때문에 Phone3G1은 모든 인터페이스의 모든 메소드를 구현해주어야 합니다.

■ **다중 구현은 다중 타입으로 선언할 수 있습니다.**

클래스의 선언에 implements 인터페이스 구문이 나온다는 얘기는 해당 클래스가 인터페이스의 메소드들을 실제로 구현하고 있다는 선언입니다. 그렇다면, 인터페이스가 여러 개 나온다는 얘기는 여러 개의 인터페이스의 메소드들이 구현되어 있다고 생각할 수 있습니다. 위의 코드를 보시면 Phone3G1이 두 개의 인터페이스를 구현하겠다고 선언되어 있기 때문에 실제로 두 인터페이스 안에 있는 모든 메소드가 오버라이드된 것을 보실 수 있습니다.

클래스가 여러 개의 인터페이스를 구현한다는 것은 또 다른 의미로는 하나의 객체를 여러 가지의 타입으로 선언하는 것이 가능하다는 것을 의미합니다. 즉 인터페이스가 타입으로 사용된다는 것을 설명 드린 적이 있습니다. 'MyMP3 mp3 = new TonyMP3();'에서처럼 어떤 클래스가 인터페이스를 구현하게 되면 변수 선언 시에 인터페이스를 타입으로 사용할 수 있다는 얘기입니다. 클래스가 여러 개의 인터페이스를 구현하게 되면 결과적으로 변수의 타입으로도 다양하게 쓰일 수 있다는 것을 의미하게 됩니다. 실제로 Phone3G1을 객체로 만들고 다양한 타입으로 선언해보면 알 수 있습니다.

예제 | 여러 개의 인터페이스를 구현한 클래스는 여러 개의 타입으로 볼 수 있다.

```
package org.thinekr.phone3g;

public class PhoneTest {
        public static void main(String[] args) {
                Phone3G1 phone = new Phone3G1();
                VisualCall v1 = new Phone3G1();
                VoiceCall v2 = new Phone3G1();
```

```
        }
}
```

변수의 타입을 보시면 실제로 만들어지는 객체는 마찬가지로 Phone3G1이지만 VoiceCall 타입으로도, VisualCall 타입으로도 사용되는 것을 보실 수 있습니다.

6 다형성이라는 것

인터페이스를 보고 코드를 작성하고 나중에 쉽게 수정한다는 점을 보고, 상속을 떠올리셨다면 환영할 만한 일이라고 생각합니다. 사실 인터페이스와 상속은 둘 다 다형성이라는 객체지향 프로그래밍의 특징을 구현하는 방식이기 때문입니다.

■ 다형성: 하나의 객체를 여러 개의 타입으로, 하나의 타입으로 여러 종류의 객체를

다형성(Polymorphism)이라는 용어에 주눅이 들 필요는 없습니다. 그냥 쉽게 풀어 보자면 'poly'는 다양한, 많은(多)의 의미이고 'morp'라는 것은 형태라는 것을 의미하니, 여러 가지 모습으로 해석될 수 있는 성격이라고 생각하시면 됩니다.

■ 다형성은 하나의 객체를 여러 가지 타입으로 선언할 수 있다는 뜻입니다.

다형성은 개발자들에게는 간단히 말해서 하나의 사물(객체)을 다양한 타입으로 선언하고 사용할 수 있다는 의미로 해석해주면 됩니다. 일반적으로 어떤 객체가 하나의 분류에만 포함되는 것은 아닙니다. 대한민국의 국민인 동시에, 남자인 동시에, 서울에 사는 사람 등과 같이, 만일 저라는 사람을 보자면 국적으로 판단할 때, 성별로 판단할 때, 거주지로 판단할 때의 기준은 각각 다르지만, 저는 그 기준에 모두 해당합니다. 이처럼 다형성은 어떤 사물을 여러 가지 시선으로 바라보는 모습을 생각하시면 쉽게 이해하실 수 있습니다.

프로그램을 만들면서 하나의 객체를 여러 가지 타입의 변수로 선언할 수 있게 되면 그만큼 뭔가 얻는 이득이 있을 겁니다. 그 이익에 대한 얘기를 먼저 시작해보도록 합니다.

■ Java에서의 다형성은 상속과 인터페이스를 통해 이루어집니다.

가끔은 기초 책 중에서 상속과 다형성을 분리해서 설명한 책들을 자주 보곤 합니다만, 사실 Java는 상속이라는 방식과 인터페이스 이렇게 두 가지를 이용해서 다형성을 지원하고 있습니다.

예를 들어 상속을 생각해 보시면 'Mouse m = new OpticalMouse( )'가 가능합니다. 또한 'Wheel Mouse m = new OpticalMouse( )'나 'OpticalMouse m = new OpticalMouse( )' 모든 선언이 가능합니다. 즉 객체는 하나인데 객체를 선언해둔 변수의 타입이 세 가지나 됩니다. 사실은 하나 더 있습니다. 'Object obj = new OpticalMouse( );'까지 가능하기 때문에 실제로는 4가지의 타입으로 하나의 객체를 가리키게 할 수 있는 겁니다. 다형성의 의미는 이렇게 하나의 객체를 다양한 시선(타입)으로 바라볼 수 있게 한다는 의미입니다. 중요한 것은 다양한 타입으로 본다는 사실 자체가 아니라 다양한 타입으로 객체를 바라보게 되면 호출할 수 있는 메소드 역시 타입에 따라 달라진다는 겁니다. 상속의 오버라이딩을 설명하면서 오버라이딩을 하게 되면 컴파일러는 실제 객체의 메소드를 바라보는 것이 아니라, 변수 선언 타입의 메소드를 본다고 설명했습니다.

```
Mouse m = new WheelMouse( );
```

실제 객체가 WheelMouse이지만 컴파일러는 Mouse 타입의 메소드가 정상적으로 호출되고 있는지에만 관심을 두게 됩니다.

■ **인터페이스가 상속보다 다형성에 더욱 유연함을 제공합니다.**

상속의 경우 Java에서는 단일 상속을 지원합니다. 따라서 부모 클래스부터 내려오는 상속 구조에 의해서만 표현할 수 있습니다.

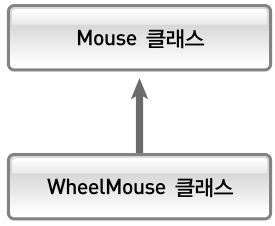


그림 5

하지만, 인터페이스는 클래스의 선언 뒤에 여러 개의 인터페이스를 구현할 수 있게 할 수 있습니다.

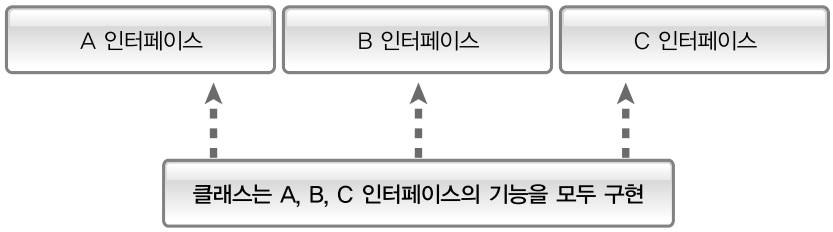


그림 6

이런 이유 때문에 하나의 객체를 여러 개의 타입으로 바라보는 다형성에는 상속보다는 인터페이스가 더 큰 유연함을 제공한다고 할 수 있습니다.

6.1 인터페이스가 다중 상속으로 오인되는 이유

인터페이스에 대해서 흔히들 단일 상속의 단점을 보완하는 Java의 현실적인 다중 상속의 도구로 설명하는 이유는 간단히 말해서 클래스 선언 뒤에 인터페이스가 여러 개 올 수 있기 때문입니다. 인터페이스가 여러 개 올 수 있다는 의미는 다시 말해 "여러 가지 타입으로 변수를 선언할 수 있다."라는 것입니다.

> 인터페이스를 상속과 결부시키지 말고 다형성의 측면에서 이해해야만 합니다. 인터페이스는 다중 구현이라는 말이 더 정확합니다.

인터페이스는 그 목적상 기능을 의미하게 할 수 있습니다. 즉 어떤 객체가 어떤 기능을 할 수 있는가로 설계할 경우에 기능에 초점을 두고 인터페이스로 설계할 수 있다는 얘기입니다. 따라서 이렇게 되면 어떤 객체는 여러 가지 기능을 가지게 됩니다. 이해가 잘 안 되시면 여러분이 가지고 다니는 휴대전화나 PMP를 생각해 보시면 됩니다. 요즘 휴대전화는 그냥 단순히 휴대전화 기능만 있는 경우는 거의 없습니다. 기본으로 카메라가 들어가 있고, MP3기능도 됩니다. 때로는 DMB 기능이 추가되어 있어서 TV 시청도 가능하고, 무선 인터넷도 지원됩니다. 그럼 만일 위의 각각의 기능을 하나의 인터페이스로 정의했다고 생각해 보면 휴대전화라는 것은 다음과 같이 여러 개의 인터페이스를 구현한 객체가 됩니다.

예제

```
public interface Camera {
	public void takePhoto();
}

public interface MP3 {
	public void playMp3();
}

public interface DMB {
	public void viewTV();
}

public class MyPhone implements Camera, MP3, DMB {
	@Override
	public void takePhoto() {
		// TODO Auto-generated method stub
	}

	@Override
	public void playMp3() {
		// TODO Auto-generated method stub
	}

	@Override
	public void viewTV() {
		// TODO Auto-generated method stub
	}
}
```

MyPhone 클래스를 보시면 Camera, MP3, DMB라는 각 각의 기능을 인터페이스로 정의하고 MyPhone 클래스에서 각 인터페이스의 내용이 구현되도록 되어 있는 것을 보실 수 있습니다. 중요한 사실은 이 경우 MyPhone은 다음과 같이 다양한 타입으로 선언될 수 있다는 겁니다.

```
Camera a = new MyPhone( ); ← 카메라 기능을 가진 객체 MyPhone
MP3 a = new MyPhone( ); ← MP3 기능을 가진 객체 MyPhone
DMB a = new DMB( ); ← DMB 기능을 가진 객체 MyPhone
```

결론적으로 생각해보면 인터페이스를 이용하면 하나의 객체가 여러 개의 기능을 가지는 형태로 보이게 만들어줄 수 있습니다. 마치 상속에서 부모 타입으로 변수를 선언하고 자식 타입으로 객체를 생성하는 코드와 유사하긴 하지만 인터페이스는 더 다양한 형태로 객체를 정의해줄 수 있습

니다. 이것은 마치 부모 클래스의 기능을 물려받는 모습처럼 선언되기는 하지만 상속보다는 더 많은 종류를 보여줄 수 있게 됩니다. 이런 모습 때문에 일반적으로 다중 상속의 기능을 활용하기 위해서 인터페이스를 사용한다고 설명되는 경우가 많은 겁니다.

6.2 인터페이스가 타입으로 쓰일 수 있기에 가능한 일들

인터페이스는 하나의 타입으로 선언되기 때문에 변수의 선언으로 가능한 모든 작업이 고스란히 가능해집니다. 우리가 일반적으로 변수를 쓰는 경우는 다음과 같습니다.

- 객체를 나중에 다시 사용하기 위해서 변수로 선언하는 경우
- 메소드의 파라미터나 메소드의 리턴 타입으로 사용하는 경우
- 배열이나 자료구조를 선언하기 위해서 사용하는 경우

인터페이스를 이용하면 이런 작업들이 모두 가능해집니다.

■ 인터페이스를 변수로 선언하는 경우

인터페이스를 변수로 선언하는 경우는 앞에서 많은 예제를 보았기 때문에 어떤 장점을 얻게 되는지만 설명해볼까 합니다. 우선 인터페이스로 변수를 선언하게 되면 사용하는 입장에서는 보이는 메소드는 인터페이스의 메소드들만 보이게 됩니다.

'MP3 a = new MyPhone( );'을 생각해보시면 a라는 레퍼런스를 이용한 뒤에는 호출한 쪽에서 보이는 메소드는 오로지 playMp3( ) 메소드밖에 없습니다. 물론 이 경우엔 상속에서 사용하는 다운 캐스팅을 이용해서 작업할 수도 있겠지만, 근본적인 해결책은 아니라는 겁니다.

인터페이스 변수를 선언하게 되면 뒤에 오는 모든 객체는 간단히 인터페이스만 구현한 객체이면 되기 때문에 실제로 new 뒤에 올 수 있는 클래스에 대한 제한이 없게 됩니다. 따라서 좀 더 시스템이 유연해지는 계기를 마련하게 됩니다.

■ 메소드의 파라미터나 메소드의 리턴 타입으로 사용되는 경우

메소드의 파라미터나 메소드의 리턴 타입으로 사용되는 경우에는 추상 클래스나 상속을 이용하는 것과 같아집니다. 역시 이런 경우에는 상속과는 달리 전혀 무관하지만, 인터페이스에 선언된 기능을 가진 객체이면 되기 때문에 좀 더 확장성 있는 구조가 됩니다.

■ **배열이나 자료구조로 사용되는 경우**

배열이나 자료구조에서 인터페이스 타입으로 선언되는 경우에는 상속의 단점을 보완하는 방식의 설계가 가능해집니다. 좀 더 어려운 말로 하자면 객체지향 프로그래밍의 격언 중에는 "상속보다는 조합을 이용하라."라는 말이 있습니다. 이것에 대해서 조금 더 깊이 알아두시면 나중에 시스템을 설계하실 때 도움이 됩니다.

7 인터페이스는 전혀 다른 객체를 하나의 타입으로 인식할 수 있게 합니다.

지금까지 앞에서 설명한 내용은 두 가지입니다. 우선은 **인터페이스가 실제 객체를 의미하는 것이 아니라 원하는 객체의 스펙을 의미하는 것**이라는 점, 두 번째는 **어떤 객체가 인터페이스를 구현함에 따라 여러 가지 타입으로 선언될 수 있다는 점**입니다. 인터페이스는 그 자체가 실제 기능 구현을 의미하지는 않기 때문에 때로는 조금 색다른 용도로 사용될 수 있습니다. 인터페이스 자체가 의미하는 것은 어떤 기능만을 의미하기 때문에 우리가 만든 여러 가지의 클래스에 원하는 인터페이스를 붙여주면 전혀 다른 객체이지만 같은 타입으로 인식할 수 있게 됩니다. 좀 설명이 어려우니 역시 실제 생활을 한번 생각해보겠습니다.

예를 들어 어떤 회사에 직원의 형태가 정규직, 계약직, 일용직 세 가지가 있다고 생각해봅시다.

사번	이름	구분	연봉/일당	수당/년
E001	홍길동	R	3000	400
E002	임꺽정	T	4000	
E003	황진이	A	5	
E004	어우동	A	10	

표

위의 표는 앞에서 상속을 설명했을 때도 나왔던 문제입니다. 공통적인 데이터가 있는 경우, '~is a kind of'의 관계를 충실하게 구현하면 상속으로 처리하는 것도 좋습니다. 다만, 상황이 약간 달라지면 어떨까요? 이 회사에서 월급이나 수당 말고도 다른 지출도 계산에 포함해야 한다고 생각해봅시다.

- 장비 임대료도 계산되어야 한다.
- 건물 임대료도 계산되어야 한다.
- 계약직이나 아르바이트가 아닌 프로젝트당 고용하는 인력의 월급도 계산되어야 한다.

위와 같은 문제를 처리하다 보면 상속이 그다지 적합하지 않은 상황이라는 것을 알 수 있습니다. 즉 공통적인 속성은 거의 존재하지 않고, 오로지 월급을 계산하는 기능만이 유일하게 같이 나오는 기능이기 때문에 상속 관계로는 부적합하다고 생각됩니다. 만일 클래스를 구성했다면 다음과 같을 겁니다.

예제 정규직은 연봉의 1/12를 받습니다.

```
public class RegularWorker {
	private float yearSalary;

	public float getMonthSal(){
		return yearSalary/12;
	}
}
```

계약직은 지정된 월급을 받기 때문에 다음과 같이 구성됩니다.

예제 계약직은 지정된 월급을 받습니다.

```
public class ContractWorker {
	private float monthSalary;

	public float getMonthPay(){
		return monthSalary;
	}
}
```

일용직은 근무일 수와 일당이 필요합니다.

예제 | 일용직은 근무 일수 × 일당의 월급을 받습니다.

```
public class PartTimer {
        private int days;
        private float pay;

        public float getMonthPay(){
                return days * pay;
        }
}
```

아, 그리고 매달 지급해야 하는 것 중에서 장비의 임대료가 있다고 방금 통보가 왔습니다. 실제로 개발하다 보면 가장 많이 겪는 상황이 바로 이렇게 예상치 못한 새로운 기능이 추가될 때입니다.

예제 | 장비 임대료는 무조건 매달 지정된 금액 나갑니다.

```
public class RentalPay {
        private float rentalFee;

        public float getMonthFee(){
                return rentalFee;
        }
}
```

이제 매달 나가는 금액을 계산하려면 네 가지 클래스의 객체들을 이용해서 계산해줘야 합니다. 이렇게 전혀 다른 객체들의 공통적인 문제를 해결하는 데 있어서 인터페이스가 도움을 줄 수 있습니다.

> 인터페이스는 하나의 타입으로 이전에 전혀 관계가 없는 클래스를 하나의 타입으로 볼 수 있게 합니다. 따라서 전혀 엉뚱한 데이터를 가진 객체들을 하나의 자료구조와 같은 타입으로 묶어줄 수 있습니다.

7.1 공통적인 기능만을 인터페이스로 정의해버립니다.

가장 먼저 해야 하는 일은 공통적인 기능을 인터페이스로 정의하는 겁니다. 위의 4가지의 클래스들을 보면 결국 매달 지급해야 하는 돈을 계산해주는 기능이 필요하다고 판단됩니다. 따라서 IPayable이라는 인터페이스를 정의하도록 합니다(인터페이스의 이름 앞에 I를 붙여서 인터페이스라는 것을 쉽게 알아보도록 명명규칙을 사용하는 경우도 흔합니다).

예제 | 공통적인 기능을 찾아서 인터페이스로 추출합니다.

```
public interface IPayable {
	public float getMonthPay();
}
```

7.2 공통 기능을 필요한 클래스가 정의하도록 클래스의 선언을 수정합니다.

이제 필요한 클래스가 공통 기능을 할 수 있도록 수정해주어야 합니다. 즉 전혀 관계없는 객체들을 공통된 기능을 구현한 객체로 볼 수 있게 한다는 겁니다. 우선 RegularWorker와 RentalPay를 수정해보았습니다.

예제 |

```
public class RegularWorker implements IPayable{
	private float yearSalary;

	public float getMonthSal(){
		return yearSalary/12;
	}

	@Override
	public float getMonthPay() {
		return getMonthSal();
	}
}
```

```
public class RentalPay implements IPayable{
	private float rentalFee;

	public float getMonthFee(){
		return rentalFee;
	}

	@Override
	public float getMonthPay() {
		// TODO Auto-generated method stub
		return 0;
	}
}
```

나머지 ContractWorker, PartTimer의 경우는 아예 메소드 자체가 getMonthPay( )로 작성되어 있기 때문에 추가적인 메소드 없이 단순히 인터페이스 선언만 추가해주면 됩니다.

예제

```
public class PartTimer implements IPayable{
        private int days;
        private float pay;

        public float getMonthPay(){
                return days * pay;
        }
}
```

```
public class ContractWorker implements IPayable {
        private float monthSalary;

        public float getMonthPay(){
                return monthSalary;
        }
}
```

7.3 인터페이스 타입으로 여러 종류의 객체를 처리할 수 있습니다.

이제 RegularWorker, ContractWorker, PartTimer, RentalPay 이 네 개의 클래스에서 나온 객체는 동일한 인터페이스를 구현했기 때문에 다음과 같이 선언할 수 있습니다.

코드

```
IPayable[] arr = new IPayable[4];

arr[0] = new RegularWorker();
arr[1] = new ContractWorker();
arr[2] = new PartTimer();
arr[3] = new RentalPay();
```

만일 네 객체의 이번 달 지급해야 하는 비용을 계산해야 한다면 각 객체의 getMothPay( )를 호출해주면 됩니다. 이처럼 인터페이스는 전혀 관계가 없는 객체들을 하나의 타입으로 볼 수 있도록 하는 중간적인 역할이 필요할 때에도 사용할 수 있습니다.

8 인터페이스는 일종의 접근 제한 역할도 할 수 있습니다.

어떤 클래스는 여러 개의 스펙(인터페이스)을 맞춰서 개발할 수 있습니다. 예를 들어 요즘 나오는 MP3들은 많은 경우에 동영상 재생 기능과 사진 보기 기능을 가지고 있습니다. 즉 동영상을 보는 인터페이스와 사진 보기 인터페이스가 구현되어 있다고 보시면 됩니다.

예제

```
package org.thinker.mp4;

public interface PlayMovie {
	public void playMovie();
}
```

```
package org.thinker.mp4;

public interface ViewPhoto {
	public void viewImage();
}
```

```
package org.thinker.mp4;

public class MP3 implements PlayMovie, ViewPhoto{
	@Override
	public void playMovie() {
		// TODO Auto-generated method stub
	}

	@Override
	public void viewImage() {
		// TODO Auto-generated method stub
	}
}
```

이때에는 실제 MP3객체가 가진 기능은 동영상 기능과 사진 보기 기능이지만, 외부에 인터페이스만 호출하게 되면 호출하는 쪽에서는 이 객체를 어떤 타입으로 보는지에 따라서 사용할 수 있는 메소드가 제한됩니다. 즉 'PlayMovie m = new MP3( );'로 보면 실제로 MP3 객체가 가진 기능은 이미지를 보는 기능 viewImage( )와 동영상을 보는 기능 playMovie( )이지만 변수의 선언에 의해서 사용할 수 있는 메소드는 playMovie( )만 보이게 됩니다. 이런 이유 때문에 클래스에 여러 가지 메소드를 만들어 둔 다음 인터페이스로 분리하는 작업을 진행하는 경우가 가끔 있습니다.

A, B, C라는 인터페이스를 구현한 클래스를 반환할 때 A 타입으로 반환하게 되면 외부에서는 A 인터페이스의 메소드만 보이게 됩니다. 따라서 별도의 접근 제한을 이용하지 않고도 접근 제한과 마찬가지 효과를 보게 하는 방법입니다.

9 인터페이스는 다중 상속 문법도 됩니다.

인터페이스를 공부하면서 잘 와 닿지 않는 것 중의 하나가 바로 인터페이스의 다중 상속입니다. 앞에서 언급했지만, 인터페이스는 다중 구현을 통해서 하나의 객체가 여러 가지의 인터페이스를 구현할 수 있도록 하고 있습니다. 그렇다면, 인터페이스에서 다중으로 상속한다는 것도 굳이 필요해 보이지 않습니다.

인터페이스의 다중 상속은 여러 개의 스펙을 모아서 하나의 스펙을 만들어내는 일종의 조합(Composition) 방식입니다.

또 다른 의문은 다중 상속을 한다는 것이 과연 Java에서 올바른 선택인가라는 점입니다. 클래스에서는 상속할 때 단일 상속만을 지원하면서 다중 상속을 지원하는 실제 의도는 무엇일까요?

9.1 인터페이스의 다중 상속은 '스펙+스펙=새로운 스펙'으로 이해하세요.

인터페이스의 다중 상속은 문법적으로 이해하지 마시고, 좀 더 클래스와 인터페이스가 많은 상황에서 생각해보시면 도움이 될 것입니다. 여러분에게 새로운 휴대전화 객체를 만들라는 임무가 주어졌다고 생각해봅니다. 여러분이 개발하는 휴대전화는 다음과 같은 기능이 있어야만 한다고 생각해보겠습니다.

- Camera 기능
- MP3 기능
- DMB 기능
- Widget 기능

기능을 보니 대부분이 예전에 앞에서 다룬 인터페이스들입니다. 그럼 간단히 어떤 객체가 이런 기능을 가질 수 있도록 인터페이스를 구현한 클래스를 만들어주면 되겠군요. 여러분이 만든 PhoneX는 4개의 인터페이스를 구현해서 프로그램을 완성했습니다. 성공적으로 만들어진 클래스 때문에 여러분은 앞으로 새로운 핸드폰의 가장 기본이 되는 스펙을 완성했습니다. 그런데 이렇게 했더니만 가끔은 이 규격을 지키지 않은 클래스들이 등장할 수도 있겠네요. 예를 들어 PhoneY는 위의 4가지 기능 중에 3가지(Camera, MP3, DMB)만 구현한다고 가정해봅니다. 그리고 PhoneZ는 위의 기능 중에서 2가지(Camera, MP3)만 구현한다고 가정해봅니다. 여러분이 시장에 PhoneX, PhonY, PhoneZ를 내 놓으면 사람들이 다음과 같이 인터페이스를 이용해서 사용할 수 있습니다.

```
MP3 a = new PhoneX( );
혹은
Camera a = new PhoneX( );
```

개발자로는 조금 아쉬운 상황입니다. 왜냐하면, PhoneX는 Camera, MP3, DMB, Widget 기능을 모두 갖춘 멋진 기기인데 겨우 Camera나 MP3로만 치부되기 때문입니다. 이럴 때는 진짜 PhoneX가 가진 모든 기능을 활용하려면 'PhoneX a = new PhoneX( );' 밖에는 답이 없습니다. 이렇게 쓸 바에는 아예 인터페이스를 만들지 않는 편이 더 나아 보이기까지 하네요. 만일 여러분이 이 상황을 그대로 방관하면 PhoneX의 후속 제품인 PhoneXX라는 제품을 개발해도 인터페이스는 별로 쓸모가 없게 됩니다.

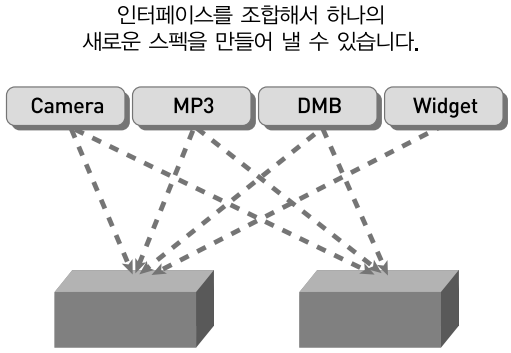


그림 7

이럴 때는 PhoneX가 구현한 4가지의 기능을 하나의 규격으로 정의해야 하는 상황이 발생합니다. 실제로 이런 식으로 제품을 규정하는 경우가 또 있습니다. 아마도 컴퓨터나 노트북에 관심을 두고 계셨다면 센트리노라는 용어를 들어 보셨을 겁니다. 센트리노라는 것은 어떤 종류의 CPU와 어떤 종류의 무선 인터넷 방식을 지원하는 하나의 스펙입니다. 어떤 회사가 어떤 제품을 만들든 간에 센트리노라는 스펙을 정확히 구현해주면 센트리노 제품으로 인정한다는 뜻입니다. 이런 상황이 바로 인터페이스를 하나의 규격으로 보는 상황입니다.

객체의 스펙으로 인터페이스를 설계하는 경우는 주로 아예 처음부터 실제 객체를 염두에 두지 않습니다. 그저 스펙을 정의해둘 뿐입니다. 어떠한 기능들을 한다는 인터페이스들을 정의하고, 그 인터페이스들을 모아서 하나의 스펙을 만들어두는 방식입니다.

이제 여러분은 위의 4가지 인터페이스를 하나의 제품의 스펙으로 규정하길 원합니다. 이런 상황이 바로 인터페이스가 제품의 규격을 정의할 때 다중 상속을 하는 경우입니다. 저는 4가지의 기능이 모두 지원되는 것은 PerfectPhone이라는 인터페이스로 정의하겠습니다.

코드 | 인터페이스는 다른 인터페이스들을 조합해서 하나의 새로운 스펙을 만들 수 있습니다.

```
public interface PerfectPhone extends Camera, MP3, DMB, Widget {

}
```

보시면 extends라는 키워드 뒤에 Java의 상속 법칙을 위반(?)하는 것을 보실 수 있습니다. Java는 기본적으로 단일 상속만을 지원하고 있습니다만 이 경우엔 extends 키워드 뒤에 무려 4개나 상속하는 것을 보실 수 있습니다.

9.2 인터페이스의 extends는 상속으로 보시면 안 됩니다.

인터페이스 뒤에 나오는 extends를 상속의 키워드이긴 하지만 상속으로 이해하시면 안 됩니다. 인터페이스 자체가 객체가 아니라는 것부터 다시 떠올려 봅니다. 인터페이스는 하나의 타입이나 규격이 될 뿐이지 그 자체가 하나의 객체가 되는 것이 아닙니다. 따라서 인터페이스의 상속은 클래스의 상속처럼 부모의 속성과 동작을 물려받는 것이 아닙니다. 인터페이스의 상속은 규격이나 스펙 자체 혹은 기능 자체의 선언을 물려받는 것입니다. 규격이나 스펙을 물려받아서 새로운 스펙을 만든다면 기존 여러 개의 스펙을 조합해서 하나로 묶거나 기존의 스펙을 고스란히 물려받은 후에 다시 추가적인 기능을 가지게 하는 것입니다.

9.3 새로운 규격을 만들게 되면 스펙의 모든 기능을 사용할 수 있습니다.

새로운 규격을 만들어 주었다면 이제 PhoneX는 무려 4개의 인터페이스를 물려받을 필요가 없고 단지 하나의 인터페이스 PerfectPhone을 물려받으면 끝납니다. 물론 이 경우에는 새로운 PhoneXX도 마찬가지입니다. 이제 PerfectPhone 이라는 규격을 물려받는 모든 휴대전화는 자동으로 4가지의 인터페이스 기능을 다 물려받게 되면서 변수의 타입도 다음처럼 선언할 수 있게 되었습니다.

```
PerfectPhone a = new PhoneX( );
PerfectPhone b = new PhoneXX( );
```

변수의 타입이 PerfectPhone이기 때문에 사용자는 그 안에 있는 4개의 인터페이스를 알 수는 없지만 4개의 인터페이스가 가지는 모든 메소드가 공개됩니다.

9.4 다중 상속 인터페이스는 언제 만들지?

객체지향 개념을 알고 있다는 것과 객체지향 개념이 코드에 녹아서 코딩에 적용되는 것은 조금 다르다고 생각합니다. 안다고 해서 바로 코드가 만들어지는 것이 아니라는 얘기입니다. 거의 모든 스포츠를 보면 자세를 상당히 중요시합니다. 야구, 축구, 농구 상관없이 바른 자세를 가지면 나중에 더 좋은 성적을 낼 수 있기 때문일 겁니다. 프로그래밍도 마찬가지입니다. 제가 지금 여러분에게 설명하는 인터페이스도 바른 자세에 관련된 내용이므로 이것을 보고 바로 코드에 적용할 것을 기대하지는 않습니다. 다만, 나중에 참고는 되실 수 있을 거라고 위안 삼고 있습니다.

인터페이스를 다중 상속하는 것은 철저한 인터페이스 기반의 설계에서만 나올 수 있는 방식입니다. 즉 모든 기능에 대해서 인터페이스로 적절하게 이미 분배가 되어 있는 상황에서 만들어지는 장치입니다. 이런 예제를 더 만들어보려고 이번에는 분식집을 한번 만들어볼까 합니다. 분식집에서는 떡볶이도 팔고, 김밥도 팔고, 라면도 팝니다. 하지만, 김밥과 라면은 편의점에서 팔기도 하고, 떡볶이와 김밥은 포장마차에서도 많이 팔지 않나요? 그럼 이러한 상황을 인터페이스로 만들어볼까요?

9.4.1 우선 각 기능은 인터페이스로 정의합니다.

가장 먼저 할 일은 각 기능을 인터페이스로 만들어 놓는 작업에서 시작해야 합니다. 떡볶이(DukBokGi), 김밥(KimBob), 라면(InstanceNoodle)을 각각의 인터페이스로 정의하는 작업에서 시작

해봅니다.

예제

```
public interface DukBokGi {
	public void makeDukBokGi();
}
```

```
public interface KimBob {
	public void makeKimBob();
}
```

```
public interface InstanceNoodle {
	public void makeInstanceNoodle();
}
```

9.4.2 원하는 기능을 묶어서 하나의 규격으로 만들어줍니다.

이제 원하는 기능들을 묶어서 규격을 만들어주어야 합니다. 예를 들어 모든 분식집에서는 기본적으로 떡볶이, 김밥, 라면을 팔고 있습니다. 그러니 모든 분식집이 지켜야 하는 인터페이스는 다음과 같은 형태가 됩니다.

예제

```
public interface BunSikHouse extends DukBokGi, KimBob, InstanceNoodle{

}
```

이제 BunSikHouse라는 인터페이스를 지키는 '김밥헤븐'이라는 클래스를 만들어 보면 다음과 같은 형태가 됩니다. 실제 메소드의 구현은 생략하겠습니다.

예제

```
public class KimbobHeaven implements BunSikHouse {
	@Override
	public void makeDukBokGi() {
		// TODO Auto-generated method stub
	}

	@Override
	public void makeKimBob() {
```

```
            // TODO Auto-generated method stub
        }
        @Override
        public void makeInstanceNoodle() {
            // TODO Auto-generated method stub
        }
}
```

누군가 이제 객체를 만든다면 'BunSikHouse a = new KimbobHeaven( );'이 될 수 있고, BunSikHouse라는 인터페이스를 구현하는 모든 클래스는 분식집이라고 할 수 있게 되었습니다.

9.4.3 인터페이스를 다중 상속으로 만들어서 좋아지는 것은?

인터페이스를 다중 상속으로 만들어서 좋아지는 것은 무엇일까요? 단일 상속도 아니고 여러 개의 인터페이스를 상속해가면서 만들어내는 이득은 주로 다음과 같습니다.

■ 기존 인터페이스들을 묶어서 하나의 새로운 스펙을 만들어낼 수 있습니다.

우선은 기존 인터페이스를 손상하지 않으면서 여러 개의 인터페이스를 모아서 새로운 스펙을 만들어낼 수 있다는 데 장점이 있습니다. 여러분의 후임이 있다면 후임들에게 인터페이스 A, B, C, D를 구현해주면 된다고 설명하는 것이 아니라, 인터페이스를 묶어둔 E라는 인터페이스 하나만 구현해주면 된다고 얘기할 수 있게 됩니다. 물론 이렇게 되면 자동으로 몇 개의 인터페이스 내용을 구현해주기 때문에 실제 클래스 작성 시에 필요한 인터페이스를 빠뜨리는 것을 방지할 수 있습니다.

■ 변수의 타입을 인터페이스로 그대로 유지할 수 있습니다.

인터페이스를 여러 개 구현하는 경우는 어떤 인터페이스로 바라보는지에 따라서 사용할 수 있는 메소드가 제한됩니다. 예를 들어 'MP3 a = new PhoneX( );'를 생각해 보시면 PhoneX가 가진 모든 기능을 활용하지 못하고, MP3 인터페이스에 정의된 메소드밖에 사용하지 못했습니다. 만일 PhoneX가 가진 모든 기능을 활용하고자 하면 'PhoneX x = new PhoneX( );'로 만들어 주는 수밖에 없습니다. 하지만, 인터페이스를 여러 개 모은 것을 사용하게 되면 'PerfectPhone a = new PhoneX( );'인 경우는 가진 모든 기능을 호출할 수 있게 된 것을 보실 수 있습니다.

■ **더욱더 풍부한 다형성을 제공합니다.**

PhoneX는 PerfectPhone 인터페이스를 구현해주었고, PerfectPhone 인터페이스는 4개의 인터페이스를 구현한 상태입니다. 따라서 필요하다면 다음과 같은 선언이 가능해집니다. 아래의 코드를 보면 하나의 객체를 인터페이스를 이용해서 다양한 타입으로 볼 수 있게 됨을 알 수 있습니다.

```
PerfectPhone x = new PhoneX();
MP3 a = x;
DMB b = x;
Widget c = x;
Camera d = x;
```

객체는 하나이지만 필요하다면 원하는 인터페이스 타입만으로도 볼 수 있는 더욱 풍부한 다형성을 제공합니다.

10 인터페이스를 이용하는 Advanced 코딩 가이드

사실 앞에서 여러 가지의 인터페이스의 용도나 활용법에 대해서 설명했지만, 이제는 그중에서 가장 중요하고, 실무에서 많이 적용하는 내용을 알려 드릴까 합니다. 요즘은 Java를 공부한다는 것이 단순한 문법을 공부한다는 것만은 아닙니다. Java를 공부한다는 것은 결국은 엔터프라이즈 시스템에서 사용될 수 있는 프로그램을 만드는 방법을 배우는 것까지 학습하는 경우가 대부분입니다. 따라서 기업용 시스템에서 인터페이스가 가지는 의미를 좀 알아두실 필요가 있습니다.

■ **인터페이스는 실제 객체가 무엇인지 몰라도 코드를 작성할 수 있습니다.**

인터페이스에서 가장 중요한 사실은 인터페이스를 통해서 객체와 객체 간의 메시지를 주고받는다는 겁니다. 이때 호출을 하는 입장('주'라고 표현하겠습니다)과 호출을 받는 입장('객'이라고 표현하겠습니다) 사이에 인터페이스가 쓰이게 됩니다. 이 구조는 마치 우리가 현실적으로 보는 음식점이나 패스트푸드점과 유사하다고 생각하시면 됩니다.

현실 세계에서는 인터페이스 뒤에는 실제 객체가 무언지 몰라도 된다는 상황이 전개됩니다. 즉 인터페이스 뒤 편에 있는 객체가 어떤 클래스에서 만들어진 객체인지 몰라도 원하는 기능을 만족하고 있기 때문에 호출해서 사용하면 된다는 겁니다. 이런 의미에서 인터페이스는 '원하는 스펙'

입니다. 실제 생활에서 이렇게 주와 객 사이에 인터페이스가 쓰이듯이 코드를 작성할 때에도 이런 식으로 되면 어떨까요? 즉 뒤편에 있는 실제 객체를 모르는 상태에서 코드를 완성할 때 인터페이스를 쓰는 겁니다. 그림에서 보듯이 주방장은 몇 가지 음식을 만드는 기준을 만족하고 있으면 됩니다.

인터페이스 방식의 코드를 작성하면 실제 객체를 모르는 상태에서도 코드 작성이 가능합니다.

그림 8

예제

```
package org.thinker.food;
public interface ICook {
	public void makeRice();
	public void makeSoup();
	public void makeSalad();
	public void makeSource();
}
```

간단히 이 인터페이스를 구현한 객체를 만들겠습니다.

예제

```
public class KoreanCook implements ICook {
	@Override
	public void makeRice() {
		System.out.println("한식 밥을 짓습니다.");
	}
	@Override
	public void makeSalad() {
		System.out.println("한식 무침을 만듭니다.");
```

```
        }
        @Override
        public void makeSoup() {
                System.out.println("고기국물을 만듭니다.");
        }
        @Override
        public void makeSource() {
                System.out.println("고추장 양념장을 만듭니다.");
        }
}
```

이제 KoreanCook을 매번 호출하는 주가 되는 점원을 만들어 볼까 합니다.

예제

```
package org.thinker.food;

public class CookManager {
        //밥을 주문받으면 주방장에게 makeRice()를 요청한다.
        //국물을 주문받으면 주방장에게 makeSoup()을 요청한다.
}
```

아직 코드가 완성되지는 않았지만, 대강 이런 기능이 요청될 겁니다. 자세히 보면 CookManager가 원하는 기능은 결국은 주방에서 일하는 KoreanCook이 가진 기능이기 때문에 CookManager는 이 기능을 KoreanCook에게 의뢰하게 될 겁니다. 이 때문에 코드를 작성할 때에는 'KoreanCook cook = new KoreanCook( );'이나 인터페이스를 이용한 'ICook cook = new KoreanCook( );'이라는 코드가 필요해집니다.

예제

```
package org.thinker.food;
public class CookManager {
        private ICook cook = new KoreanCook();
        //밥을 주문받으면 주방장에게 makeRice()를 요청한다.
        public void orderRice(){
                cook.makeRice();
        }
        //국물을 주문받으면 주방장에게 makeSoup()을 요청한다.
        public void orderSoup(){
                cook.makeSoup();
```

```
        }
}
```

■ **이게 최선일까요? 좀 더 유연한 방법을 찾아봅시다.**

위의 코드는 실행될 때에 아무런 문제가 없고, 만일 주방장이 한식 주방장에서 양식이나 일식 주방장이 되면 코드 'ICook cook = new JapeneseCook( );'과 같이 코드를 한 줄만 수정해주면 됩니다. 하지만, 좀 더 생각해보시면 CookManager 코드를 작성할 때 반드시 KoreanCook 클래스나 호출하는 클래스가 있어야만 합니다. 즉 아직 KoreanCook 클래스를 만들지 않았다면 위의 코드는 컴파일할 수 없는 코드입니다. 이것을 다른 말로 표현하자면 CookManager는 KoreanCook에 종속적이라고 합니다. 즉 KoreanCook이 없으면 코드를 컴파일할 수 없다는 얘기입니다.

■ **인터페이스만 보고 우선은 코드를 컴파일할 수 있게 합시다.**

이런 종속적인 문제를 매번 겪는다면 여러분은 다른 사람들이 클래스를 만들어주기 전까지 코드를 완성할 수 없을 겁니다. 따라서 우리들의 선배격의 프로그래머들은 "인터페이스에 의존해서만 코드를 작성합시다."라고 말하게 됩니다. 이 말은 결국 현재 작성되는 코드에서는 종속적인 문제를 일으키는 코드(위에서는 KoreanCook)를 사용하지 말고, 인터페이스만 보고 코드를 만들어 두자는 것을 의미합니다. 코드를 수정하자면 다음과 같습니다.

예제

```
package org.thinker.food;
public class CookManager {
        //인터페이스만 봅니다.
        private ICook cook;
        //밥을 주문받으면 주방장에게 makeRice()를 요청한다.
        public void orderRice(){
                cook.makeRice();
        }
        //국물을 주문받으면 주방장에게 makeSoup()을 요청한다.
        public void orderSoup(){
                cook.makeSoup();
        }
}
```

앞의 코드에서 KoreanCook의 객체 생성 코드를 없애니, 인터페이스만 보고 코드를 만들었는데도 컴파일이 되었습니다.

■ **에이, 실행이 안 되잖아요?**

하지만, 실제로 이 CookManager를 실행하면 NullPointerException이 발생합니다. 왜냐하면, 실제 사용되는 객체에 대한 언급이 없으니까 'ICook cook;'이기 때문에 cook이라는 변수에는 null이라는 빈 리모컨만 들어 있게 되는 겁니다. 아무리 인터페이스를 보고 유연하게 코드를 만든들 실행이 안 되면 무슨 의미가 있겠습니까?

■ **의존적인 문제를 일으키는 객체를 외부에서 넣어줄 수 있다면?**

지금의 CookManager 코드에서 의존적인 문제를 일으키는 것은 'ICook cook'입니다. 실제 동작하는 객체가 들어 있지 않으니 문제가 발생합니다. 따라서 이런 의존적인 객체를 외부에서 객체를 만들어줘서 넣어주게 코드를 작성하면 어떨까요?

예제

```
package org.thinker.food;
public class CookManager {
        //인터페이스만 봅니다.
        private ICook cook;
        //의존성의 문제 있는 객체를 생성 시에 주입합니다.
        public CookManager(ICook cook){
                this.cook = cook;
        }
        //의존적인 객체를 setter를 통해 주입합니다.
        public void setCook(ICook cook){
                this.cook  = cook;
        }
        //밥을 주문받으면 주방장에게 makeRice()를 요청한다.
        public void orderRice(){
                cook.makeRice();
        }
        //국물을 주문받으면 주방장에게 makeSoup()을 요청한다.
        public void orderSoup(){
                cook.makeSoup();
        }
}
```

앞의 코드를 보면 외부에서 CookManger가 필요한 ICook 인터페이스를 구현한 객체를 세팅하도록 하는 데 두 가지 방법을 사용하고 있습니다. 생성자와 setter 메소드를 이용해서 어떤 객체이든 간에 ICook이라는 인터페이스를 구현한 객체를 넣어주면 사용할 수 있게 만든 겁니다.

11 의존성 주입(Dependency Injection)

기존의 코드와 비교해보면 CookManager가 직접 KoreanCook과 같이 자신에 필요한 객체를 직접 생성한 반면에 위의 코드에서는 외부에서 ICook 인터페이스를 구현한 객체를 넣어주도록 변경된 것을 보실 수 있습니다. 이것은 마틴 파울러라는 유명한 개발자가 제시한 '의존성 주입'이라는 개념입니다. 즉 코드를 작성할 때 의존적인 코드는 만들지 말고, 외부에서 필요한 객체를 넣어주게 되면 코드를 보다 유연하게 작성할 수 있다는 얘기입니다. 용어라는 것은 만들어지면 그 자체로 힘을 가지게 됩니다. '의존성 주입'이라는 이 용어는 예상보다 엄청난 파급 효과를 가져오게 됩니다. 요즘 나오는 소위 프레임워크(Framework)라는 것들이 바로 이런 원리를 이용해서 만들어지고 있습니다.

12 인터페이스 vs. 추상 클래스

사실 이 부분은 추가로 넣은 부분입니다만, 가끔 강의를 하면서 많이 받는 질문이라 한번 정리하고 가는 것도 좋을 듯합니다. 보통 제가 많이 받는 질문은 다음과 같습니다.

- 인터페이스와 추상 클래스가 뭐가 다른가요?
- 어느 게 더 좋은 건가요? 인터페이스? 추상 클래스?
- 인터페이스는 어려우니 상속으로 만들면 안 될까요? 상속은 조금 더 쉽던데.

12.1 인터페이스와 추상 클래스의 비교

우선은 첫 번째 질문부터 생각해 봅니다. 인터페이스와 추상 클래스는 다음과 같은 공통점이 있습니다.

- **둘 다 모두 추상 메소드라는 것을 가집니다.**
  - 둘 다 추상 메소드를 가지는 데 추상 메소드라는 것이 무언지 생각해 보면 결국은 실제로 구현하는 클래스에게는 일종의 빚(부채)입니다. 즉 구현 클래스에서는 반드시 만들어야만 하는 강제성을 가지게 됩니다.
  - 추상 메소드는 결국 컴파일러를 속입니다. 추상 클래스는 변수를 인터페이스나 부모 클래스 타입으로 보았을 때 아무런 호출에 문제를 일으키지 않습니다. 따라서 컴파일러는 실제로 메소드가 어떻게 동작할지는 모르고, 아무 문제 없이 실행해줍니다. 즉 컴파일러를 속여서 내가 원하는 동작을 마음대로 조종하게 한다는 겁니다.
- **둘 다 객체 생성은 불가능하고, 타입으로만 사용됩니다.**
  - 추상 클래스와 인터페이스 둘 다 객체의 생성이 목적인 클래스가 아니라, 변수나, 파라미터, 리턴 타입, 자료구조를 유연하게 쓰기 위한 도구입니다.

반면에 추상 클래스와 인터페이스는 다음과 같은 차이가 있습니다.

- **인터페이스는 스펙이나 원하는 기능을 정의하고자 쓰지만, 추상 클래스는 '상속 + 약간의 강제성이 목적'입니다.**
  - 이 이야기는 인터페이스가 객체 간의 통신을 위해서 사용되는 반면에 추상 클래스는 본연의 상속 기능에 강제성을 추가하는 것이 목표로, 만들어진 시작점이 다르다는 겁니다.
  - Java에서는 구현 클래스의 상속이 단일 상속만을 지원하고 있다는 점을 생각해보면 만일 추상 클래스가 좀 더 유연한 구조로 사용되려면 인터페이스처럼 여러 개를 사용할 수 있게 설계되었어야 한다는 얘기입니다.
- **인터페이스는 상수, 추상 메소드만 존재하지만, 추상 클래스는 상속이 원래 목적이므로 실제 변수나 메소드를 그대로 가지고 있습니다.**
  - 인터페이스는 상호 간의 규칙이기 때문에 private으로 변수를 선언할 수도 없고, 선언된 변수는 자동으로 상수로 처리됩니다. 반면에 추상 클래스는 상속의 수단이기 때문에 상속의 본연의 기능을 그대로 가지고 있습니다. 따라서 공통적인 메소드나 변수를 부모 클래스로 모아서 사용할 수 있습니다.
- **인터페이스는 부채만 남겨주지만, 추상 클래스는 재산도 남겨 줍니다.**
  - 이 말은 앞의 설명과 좀 비슷합니다만, 인터페이스는 추상 메소드만 존재하기 때문에 부채만 잔뜩 던져주게 되지만 추상 클래스는 상속의 원래 기능을 그대로 사용할 수 있다는 얘기

입니다.

- **인터페이스는 다중 상속(?)도 가능하지만, 추상 클래스는 단일 상속만 됩니다.**
  - 인터페이스는 기존에 선언된 인터페이스를 여러 개 모아서 하나의 새로운 스펙을 만드는 것이 가능합니다. 이것을 정상적인 용어로는 인터페이스의 다중 상속이라고 합니다만, 실제로는 다중 스펙이나 다중 기능이라는 표현이 더 정확하다고 생각합니다. 반면에 추상 클래스는 상속이 본 목적이므로 기존 상속처럼 단일 상속만 가능합니다.

12.2 인터페이스와 추상 클래스? 어느 게 더 좋은가요?

Java를 사용하는 사람들에게 가장 난감한 질문일 듯하군요. 사실 솔직한 제 정답은 "그때그때 달라요."입니다. 그러니 차라리 언제 추상 클래스 기반으로 만들고, 언제 인터페이스를 고려하는지에 대한 기준을 정리해드리는 것이 더 좋을 듯합니다.

■ 추상 클래스(상속)가 더 나은 상황

추상 클래스는 거의 모든 기능을 그대로 물려주면서 부분적으로 한 가지나 두 가지의 기능이 다를 때만 사용하는 것이 정석입니다. 가장 대표적인 예가 나중에 공부할 InputStream이나 OutputStream입니다. InputStream은 간단히 말해서 데이터를 읽어들이는 역할을 하는 클래스입니다. 하지만, 실제로 데이터를 읽는 대상에 따라 읽어들이는 방법 자체가 다양합니다. 키보드에서 들어오는 데이터를 읽어들이는 방법과, 파일 시스템에서 들어오는 데이터를 읽어들이는 방법, 네트워크를 통해서 데이터를 읽어들이는 방법 등이 있을 수 있습니다. 이런 다양한 상황에서 InputStream 계열의 하위 클래스들은 자신만의 방식으로 데이터를 읽어냅니다.

데이터를 읽어 내고 난 후의 처리는 어떤 방식이든 문제가 되지 않습니다. 즉 어떻게든 데이터만 읽어 내면 그다음은 똑같은 방식으로 처리됩니다. 추상 클래스는 바로 이러한 문제에 가장 적합합니다. 거의 모든 기능이 그대로 사용되지만, 핵심적인 부분 한 곳 정도가 다른 경우에 가장 좋은 선택이라고 생각하면 됩니다.

■ 인터페이스가 더 나은 상황

인터페이스가 더 나은 상황은 간결하게 말하자면 상속과 추상 클래스가 완벽하게 좋은 경우를 제외한 모든 경우라고 할 수 있습니다만, 여기서는 조금 더 구체적인 기준을 잡아주는 것이 더 좋을 듯합니다.

인터페이스는 추상 메소드와 상수부분밖에 없습니다. 따라서 인터페이스를 쓴다는 것은 뒤에서 동작할 객체의 메소드를 미리 알고 있다는 것과 일맥상통합니다. 이렇게 되면 각 객체가 내부적으로 어떤 방식으로 동작하는지는 각 객체의 사정에 맞추어 주고, 단순히 메소드만 통일해서 쉽게 호출하자는 겁니다. 좀 더 간결하게 정리하자면 어차피 뒤에서 각 객체가 다르게 동작하겠지만, 난 메소드들을 미리 알고 있다는 겁니다.

상속이나 추상 클래스가 거의 모든 것을 물려받는 상황에 적합한 선택이라고 한다면, 인터페이스는 반대의 상황입니다. 즉 실제 움직이는 각 객체는 자신만의 방식을 고수할 수 있기 때문에 전혀 다르게 동작하겠지만, 외부에서 호출하는 메소드는 인터페이스를 통해서 호출하면 나중에 객체가 A에서 B와 같은 방식으로 변경되더라도 아무런 문제가 없이 사용할 수 있는 상황에서 최적화된다고 생각하시면 됩니다.