CHAPTER 10

상속을 통해서 코드를 절약한다

Java 언어가 상속이라는 기법을 통해 반복적인 작업을 줄이고 유연한 구조를 만들어 내는 방식을 배워봅니다.

"프로그램을 확장하고, 객체지향 개념을 이용해서 상속을 한다?" 이런 구호가 여러분의 코드에 영향을 줄까요? 제 생각에는 별로 그렇지 못한 것 같습니다. 그보다는 "상속을 이용하면 여러분이 이렇게 편해집니다."가 더 설득력 있지 않을까요? 무언가 새로운 개념들은 기존에 불편함을 해결하기 위해서 나오는 겁니다. 상속이나 인터페이스 같은 개념도 마찬가지입니다. 새로운 것을 배우면 좀 더 편해지고 여러분의 코드는 간결해져야 합니다.

예전에 모 회사의 컨설팅을 할 때였습니다. 소스를 보고 있는데 옆에 개발자가 와서 저에게 "우리 코드는 너무 복잡해서 보시기 어려워요. if ~ else가 250개가 넘어요."라고 말하는 것을 들었습니다. 그분은 너무 복잡하다는 것을 설명해 주려고 그런 말씀을 해주셨겠지만, 그 순간 제 머릿속을 스치는 생각은 '프로그램 설계가 잘못되었군.'이라는 느낌이었습니다. 객체지향 설계가 올바르게 적용된 코드에서는 if ~ else가 수 백번 반복이 되는 코드를 만들어 내지는 않기 때문입니다. 이번 장에 배우는 상속과 다음 장에 배우는 인터페이스가 제가 왜 그런 생각을 하는지를 설명해 줄 수 있기를 기대하면서 시작해 봅니다.

1 기본적인 객체지향 프로그래밍 다음에 배워야 하는 내용

Java 언어를 배우는 사람들 대다수가 가장 어려워하는 고비는 객체지향의 개념을 받아들이는 부분입니다. 지금 현재 전 세계에서 가장 많이 쓰이는 언어가 Java라는 사실을 생각해보면 전 세계의 사람들이 객체지향 프로그래밍의 개념을 배울 때 사용하는 언어도 Java라는 간단한 결론이 나옵니다. 그렇다면, 객체지향 프로그래밍에 대한 기념 개념 이후에 왜 갑자기 '상속'이나 '인터페이스'에 대한 얘기가 모든 Java 서적에서 다루는 주제가 되었을까요?

객체지향 프로그래밍의 개념을 보고, 클래스를 여러 개 구성하다가 보면 "대체 이렇게까지 클래스를 나누어야 하는 이유는 뭐야? 그냥 하나의 소스에 다 코드를 작성해도 될 텐데..."와 같은 의문을 가지는 분들을 봅니다. 분명한 것은 그분들의 생각이 결코 틀린 생각은 아니라는 겁니다.

하지만, 정말로 수많은 사람이 그런 생각을 했다면 이미 프로그래밍의 방식이 변경되어야만 했어야 정상적인 순서입니다. 마치 시장에서 공급이 부족하면 가격이 상승하는 것과 같은 방식입니다. 이렇게 불합리하다고 생각된 방식에는 뭔가가 그 이상의 장점이 있다고 가정해볼 수 있습니다. 그리고 그 가정은 여러분이 규모가 큰 시스템을 구성할 때 그 효과를 알 수 있습니다. 다른 많은 요인이 있겠지만, Java 언어를 처음 접하는 사람들이 상속이나 앞으로 배울 다형성을 제대로 이해하지 못하는 가장 큰 원인은 이런 개념들 자체가 규모가 작은 시스템에서는 느껴지지 않는 면이 있기 때문이라고 생각합니다.

1.1 상속(Inheritance)이란 무엇인가?

상속이라는 것에 대해서 정의를 내린다면 다음과 같이 정리해볼 수 있습니다.

> **상속**이란 특정 클래스를 구성할 때 기존 클래스의 데이터(속성)와 메소드를 상위(부모) 클래스에서 그대로 물려받아서 중복적인 코드를 줄인다는 장점과, 하나의 변수 타입으로 여러 종류의 객체를 의미하는 추상화된 방식의 프로그램이 가능하게 하는 객체지향 기법입니다.

상속이라는 용어는 오히려 현실 세계에서는 여러분에게도 매우 익숙한 용어입니다. 현실 세계의 상속이란 '부모의 재산을 자식이 물려받는 것'을 의미합니다. 현실 세계에서야 그렇다고 해도 객체지향 프로그래밍에서 왜 상속이라는 것을 사용하기로 했을까요?

1.2 상속의 용도: 이미 작성된 코드를 물려받는다.

상속이 초급 개발자들에게 느껴지는 가장 큰 장점은 한마디로 코드의 양을 줄여 준다는 겁니다. 상속이라는 것이 부모의 것을 물려받는다는 의미인데 간단히 설명해서 어떤 코드에 있는 메소드나 변수를 그대로 물려받아서 코드의 양을 줄이겠다는 의미입니다. 예를 들어 계산기를 한 번 생각해 봅니다. 계산기에는 더하기, 빼기, 곱하기, 나누기의 기능이 있습니다. 여러분은 다른 사람들이 쉽게 사용할 수 있도록 계산기를 클래스로 만들어서 객체를 생성할 수 있도록 만들 수 있습니다. 그런데 가끔은 기존에 만든 것을 업그레이드해서 추가적인 기능을 가지도록 만들고 싶을

때가 있습니다. 상속은 이 경우 기존 코드를 그대로 활용할 수 있기 때문에 다시 처음부터 재작성하는 수고를 줄여 줄 수 있습니다.

그림 1 일반 계산기와 공학용 계산기

우리가 주로 앞에서 만든 것을 다시 만들어야 하는 경우는 객체가 가지는 속성(데이터)과 기능(메소드)이 있습니다.

■ **비슷비슷한 속성이 필요할 때가 있습니다.**

흔히 교육할 때 가장 많이 쓰이는 예가 바로 상품 정보입니다. 예를 들어 컴퓨터 상품, 식료품이 있다고 생각해봅시다.

상품명	가격
연필	500
노트	1000
색연필	300
볼펜	250

상품명	가격	사용전압
세탁기	300000	110V
TV	3090000	200V

상품명	가격	유통기한
라면	800	2011-01-01
우유	1200	2010-10-10
통조림	3500	2015-01-01

표

위에서 보는 것처럼 둘 다 몇 가지는 같은 데이터가 필요합니다. 제품의 코드번호나, 제조년월일, 가격, 상품명같이 공통적인 정보도 있고, 가끔은 특화된 정보도 있습니다. 식료품일 경우에는 보관방법이 좀 다를 수 있고, 전자제품에는 온도나 동작 환경 같은 데이터가 추가되는 경우도 많습니다. 위의 도표에 나오는 각 행(Row)을 객체로 구성하고 위의 열(Column)을 클래스로 구성한다면 상품명과 가격 데이터를 표현하는 코드를 매번 작성해주어야 합니다.

■ **아까 봤던 그 메소드가 필요할 때도 있습니다.**

이런 경우는 계산기의 예제에 해당할 듯합니다. 기존에 있었던 기능에 새로운 기능을 덧붙여서 무언가 만들어 두고 싶은 경우입니다. 일반 계산기와 공학용 계산기를 생각해 보면 공학용 계산기는 일반 계산기가 가진 모든 기능을 다 가진 상태에 추가적인 기능을 더 가진 형태입니다. 이 경우에 상속이라는 것을 이용하면 공학용 계산기를 만들 때 별도의 작업이 없이 기존의 기능을 그대로 물려받게 해줄 수 있습니다.

그림 2

위의 그림은 윈도우 7에서 제공하는 계산기 프로그램입니다. 보시는 것처럼 같은 사칙연산이라는 기능을 가지고 있기는 하지만 공학용 계산기나 프로그래밍용 계산기라는 원래의 계산기에 추

가적인 기능이 더 붙는 형태로 구성됩니다.

1.3 부모 클래스, 자식 클래스: 상속과 관련된 용어

상속이란 객체지향 개념을 학습할 때 비교적 쉽게 적응할 수 있는 개념이라고 생각합니다. 기존에 있는 것을 물려받는다, 즉 기존의 코드를 그대로 활용하면서 필요한 만큼 새로운 기능을 붙여줄 수 있다는 것이 상속입니다. 이제 상속을 하는 데 있어서 사용되는 용어를 좀 알아둘 필요가 있습니다.

■ 부모 클래스, Super 클래스, Parent: 코드를 물려주는 클래스

우선은 부모 클래스에 대한 얘기입니다. 상속을 해주는 대상이라고 할까요? 마치 부모님 입장에 놓이는 클래스를 부모 클래스라고 합니다. 부모 클래스는 다른 말로는 Super 클래스, parent라고 합니다. 예를 들어 공학용 계산기와 일반 계산기의 관계는 일반 계산기가 부모 클래스라고 볼 수 있습니다.

일반 계산기의 기능을 그대로 물려받고, **공학용 계산기**의 추가적인 기능이 더 붙는다.

일반 계산기(부모)
공학용 계산기(자식)

그림 3 일반 계산기와 공학용 계산기

■ 물려 주는 코드가 많다고 해서 꼭 좋은 것만은 아닙니다.

상속이 무조건 편리하고 편한 것만은 아닙니다. 때로는 하위에서 개발하는 개발자가 더 힘들어질 수도 있습니다. 우리가 현실에서도 재산뿐 아니라 채무 역시 상속이 되는 것처럼, 상속이라는 것도 사용 용도에 따라서는 구현하는 개발자를 괴롭힐 수도 있습니다. 이에 관해서는 나중에 추상 메소드, 추상 클래스에서 살펴보도록 합니다.

■ **자식 클래스, Sub 클래스, Child: 코드를 물려 받는 클래스**

자식 클래스는 반대의 개념입니다. 자식 클래스는 부모 클래스로부터 데이터와 기능을 물려받은 클래스라고 생각하면 됩니다. 그럼 자식 클래스는 부모 클래스를 물려받기 때문에 별도의 속성과 동작이 없어도 부모의 속성과 동작을 그대로 물려받습니다. 즉 기능으로 보면 자식 클래스 쪽은 별도로 무언가 하지 않아도 부모의 인스턴스 변수와 메소드를 물려받습니다.

■ **자식 클래스는 부모 클래스의 모든 것을 물려받습니다.**

자식 클래스의 존재는 부모 클래스의 모든 선언과 메소드를 물려받습니다. 게다가 원한다면 자신만의 확장된 기능을 추가할 수도 있습니다. 이런 의미에서 보면 자식 클래스는 기본적으로 부모의 모든 메소드 + (자신의 고유 기능)이라는 공식이 성립할 수 있습니다. 따라서 기능면으로 본다면 부모 클래스의 기능보다는 자식 클래스의 기능이 더 많은 경우가 많습니다.

1.4 extends: 상속을 사용하는 문법

이제 본격적으로 상속을 어떻게 하고 사용하는지 알아보도록 합니다. 상속은 개념적으로는 어려운 개념이 아닙니다. 오히려 객체지향 설계 시에는 상속의 단점들이 드러나면서 어려워지는 경향이 있습니다만, 우선은 가벼운 예제로 시작하도록 합니다. 예를 들어 부모 클래스를 재현하기 위해서 가상의 가게 '자장루(가칭)'를 만들어 보도록 하겠습니다. 자장루는 설정상 자장면으로 아주 유명한 가계입니다. 따라서 자장루에는 '자장면을 만든다'는 메소드가 있습니다.

예제 자장면을 만드는 기능을 가진 ZaZangRu 클래스

```
public class ZaZangRu {
      public void makeZaZang(){
            System.out.println("감자와 돼지고기, 자장을 볶습니다.");
            System.out.println("자장면을 만듭니다.");
      }
}
```

간단한 클래스니까 별도의 설명은 생략하도록 합니다. 그럼 이제 ZaZangRu를 상속하는 개념을 한번 보도록 합니다.

1.4.1 클래스 뒤의 extends + 상속받을 클래스

ZaZangRu의 사장님에게 아들들이 있다고 생각해봅시다. 이 아들 중 한 명이 서울에 와서 분점을 내서 SeoulZaZangRu를 개업한다고 가정해봅시다. 어떤 클래스로부터 상속하고자 할 때는 'extends [부모 클래스 이름]' 구문을 사용합니다.

예제 | ZaZangRu를 상속한 SeoulZaZangRu클래스

```
public class SeoulZaZangRu extends ZaZangRu {
      public static void main(String[] args) {
            SeoulZaZangRu sz = new SeoulZaZangRu();
            sz.makeZaZang();
      }
}
```

```
감자와 돼지고기, 자장을 볶습니다.
자장면을 만듭니다.
```

위의 코드에서는 extends라는 키워드를 유심히 보시면 됩니다. extends 뒤에는 부모 클래스의 이름이 들어옵니다. SeoulZaZangRu의 경우에는 ZaZangRu 클래스를 상속한 경우입니다. 재미있는 사실은 정작 SeoulZaZangRu라는 클래스는 아무런 메소드가 없다는 점입니다. 아무것도 메소드가 없는 상태인데 makeZaZang( ) 메소드가 호출되는 것을 보실 수 있습니다. 상속이라는 것은 부모 클래스가 가진 속성과 동작을 그대로 물려받을 수 있기 때문에 호출하려는 메소드가 부모 쪽에 있으면 그대로 물려받아서 사용할 수 있습니다.

1.4.2 기능으로 보면 부모보다는 자식이 더 많을 수 있습니다.

상속에서 보면 오히려 부모 클래스보다는 자식 클래스 쪽이 더 많은 기능을 가질 수 있습니다. 예를 들어 지금 SeoulZaZangRu가 ZaZangRu를 물려받았으니 당연히 makeZaZang( )은 할 수 있습니다. 하지만, 자식 클래스는 자신만의 추가적인 기능을 만들어 낼 수 있습니다. 여기서는 '짬뽕을 만든다'는 기능을 SeoulZaZangRu에 추가해 보도록 할까요?

예제 | makeZamBong( ) 기능이 추가된 SeoulZaZangRu

```
public class SeoulZaZangRu extends ZaZangRu {
      public void makeZamBong(){
            System.out.println("짬뽕을 만들 수 있습니다.");
      }

      public static void main(String[] args) {
            SeoulZaZangRu sz = new SeoulZaZangRu();
            sz.makeZaZang();
            sz.makeZamBong();
      }
}
```

```
감자와 돼지고기, 자장을 볶습니다.
자장면을 만듭니다.
짬뽕을 만들 수 있습니다.
```

이제 SeoulZaZangRu 클래스는 자신만의 메소드인 makeZamBong( ) 메소드를 가지고 있기 때문에 부모 클래스보다 기능이 더 많은 형태가 됩니다.

1.5 단일 상속: Java에서는 C++과 달리 하나의 부모 클래스만 가질 수 있습니다.

요즘 방송에는 '막장 드라마'라는 말이 참 많이 나옵니다. 광업에 종사하시는 분들에게는 비하적인 표현이라 자제하고 싶습니다만, 이해를 돕기 위해서 그냥 이 용어 쓰도록 하겠습니다. 막장 드라마에는 늘 아이들이 부모를 몰라서 갈팡질팡합니다. 시청자들도 모르니 뭐 본인들이라고 알겠습니까? 뭔 출생의 비밀들은 그리도 많은지.

Java 이전의 C++의 경우는 객체지향의 개념을 구현한 언어이기 때문에 상속이라는 것도 똑같이 만들었습니다. 그런데 C++에서는 부모 클래스가 여러 개입니다. 아래의 그림은 다중 상속을 표현하기 위해서 고양이와 호랑이를 다중 상속을 하는 구조를 만들어 보도록 합니다.

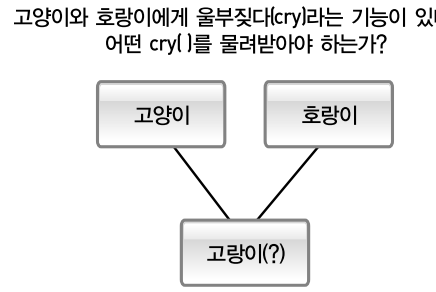

그림 4

위의 그림처럼 보면 자식 객체에서 cry( )라는 메소드를 호출하는 순간, 자식은 누구에게서 물려받은 cry( )를 호출해야 할까요? 두 개의 cry( ) 메소드를 물려받기 때문에 실제로 어떤 메소드를 호출해야 하는지 결정하기가 어려워집니다. 이 문제를 겪지 않기 위해서 Java에서는 단일 상속만을 허용합니다. 단일 상속만을 허용하게 되면 이런 애매모호한 호출의 문제를 해결할 수 있기 때문입니다.

1.6 상속을 이용해봅시다: Mouse, WheelMouse, OpticalMouse

변수의 타입이라는 개념을 정확히 이해해야 합니다. 변수 선언 앞에 있는 변수의 타입이라는 것은 객체 자료형의 변수인 경우에는 어떤 타입의 객체의 리모컨(레퍼런스)을 담는 용도로 사용됩니다.

자장루가 상속이 어떤 것인지 살짝 맛보기 수준이었다면 조금 더 깊이 가기 위해서 우리가 사용하는 컴퓨터의 마우스를 예로 들어 봅니다. 변수의 선언과 실제의 클래스를 주의해서 보시기 바랍니다.

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

위 선언의 해석은 앞으로 남은 '다형성'이라는 개념과 더불어 너무나 중요한 개념입니다. 변수의 타입을 해석할 때에는 '~계열, 종류, 직원' 등의 의미를 가지고 바라보면 조금 더 적용이 쉬워집니다. WheelMouse 객체를 마우스 계열로 생각하시면 됩니다. 그럼 이 경우에는 상속이 어떻게 적용되었는지 알아보도록 합니다.

그림 5 2 버튼 마우스

원래 마우스는 처음 나왔을 때 위의 그림처럼 2 버튼이었습니다. '쥐'를 닮았다고 해서 마우스라는 이름이 붙었습니다. 물론 지금은 2 버튼 마우스는 역사 속으로 사라졌지만, 아직은 간혹 오래된 시스템의 서버용 컴퓨터에 붙어 있기는 합니다.

예제 | Mouse 클래스

```
public class Mouse {
	public void clickLeft(){
		System.out.println("왼쪽 클릭");
	}

	public void clickRight(){
		System.out.println("오른쪽 클릭");
	}
}
```

2 버튼의 마우스는 시간이 지나면서 중간에 휠(Wheel)이 있어서 편하게 스크롤링이 가능한 휠 마우스가 나왔습니다. 이런 경우가 상속을 써먹기에 가장 좋은 곳입니다. 부모의 기능을 그대로 물려받으면 되니까요. 상속을 할 때에는 extends라는 키워드가 사용됩니다.

예제 | scroll( )이 추가된 WheelMouse 클래스 (Mouse 상속)

```
public class WheelMouse extends Mouse{
	public void scroll(){
		System.out.println("스크롤 기능 추가");
	}
}
```

WheelMouse 클래스는 Mouse 클래스를 상속받았기 때문에 별도의 clikcLeft( )와 clickRight( )를 구현할 필요가 없다는 사실을 아실 수 있어야 합니다. 두 개의 클래스에서 객체를 만들어서 테스트를 진행해보면 다음과 같이 만들 수 있습니다.

예제 Mouse를 테스트하는 클래스

```
public class MouseTest {
      public static void main(String[] args) {
            Mouse m = new Mouse();
            m.clickLeft();
            m.clickRight();

            System.out.println("--------------------");

            WheelMouse wm = new WheelMouse();
            wm.clickLeft();
            wm.clickRight();
            wm.scroll();
      }
}
```

```
왼쪽 클릭
오른쪽 클릭
--------------------
왼쪽 클릭
오른쪽 클릭
스크롤 기능 추가
```

실행 결과를 보면 WheelMouse 객체는 scroll( )이라는 고유의 메소드는 하나밖에 없지만, 부모의 clickLeft( ), clickRight( )를 물려받은 것이 보입니다. 여기서 주의 깊게 봐야 하는 부분은 현재까지는 변수의 타입 (Mouse m, WheelMouse wm)과 실제의 객체의 클래스가 정확히 일치하고 있다는 점입니다.

■ 부모와 자식 클래스 구분법: 부분 집합과 전체 집합

상속에서 편리하고도 주의해야 하는 것이 바로 이 내용입니다. 지금까지는 항상 변수의 타입과 뒤에 나오는 클래스가 동일했지만, 이제부터는 변수의 타입과 실제 클래스가 맞지 않는 현상이 일어납니다. 변수를 선언할 수 있는 경우의 수를 한번 따져보도록 합니다.

그림 6

■ **경우의 수: 타입은 Mouse이고 객체는 WheelMouse??? 부분 집합이 되나?**

'Mouse m = new WheelMouse( );'입니다. 역시나 집합 관계의 그림으로 보시면 모든 WheelMouse는 Mouse의 일종이라는 관계가 성립됩니다. 따라서 변수의 타입으로 Mouse를 선언할 수 있습니다.

예제 | 실제 객체와 변수의 타입이 다를 수 있는 상속

```
public class MouseTest {
      public static void main(String[] args) {
            Mouse m = new WheelMouse();
            m.clickLeft();
            m.clickRight();
      }
}
```

```
왼쪽 클릭
오른쪽 클릭
---------------------
```

■ **'is a kind of'의 관계로 보면 됩니다.**

이제 여러분이 배울 내용에서는 변수의 타입과 실제 클래스가 일치하지 않는 경우를 빈번하게 보게 됩니다. 따라서 항상 부분 집합의 관계가 성립되는지, 아니면 'is a kind of~'의 관계가 되는지를 정확히 따져보면 됩니다.

그림 7 Mouse라고 칭하는 것들

■ **경우의 수: 타입은 WheelMouse이고 객체는 Mouse??? 이거 될까요?**

예를 들어 코드로는 'WheelMouse m = new Mouse( );'의 선언은 가능할까요? 조금이라도 혼란스럽다고 판단되면 무조건 집합을 생각해 주시면 됩니다. 관계를 보면 모든 WheelMouse의 객체인 경우에는 모두 다 Mouse의 일종으로 속합니다. 부분 집합으로 포함된다는 의미입니다. 하지만, 반대로 Mouse라고 해서 모두 WheelMouse에 속하는 것은 아닙니다. 따라서 변수의 타입이 WheelMouse이고 실제 객체가 Mouse인 경우에는 논리적으로 맞지 않는 관계가 되기 때문에 컴파일러가 오류를 발생시킵니다.

코드

```
//컴파일러 오류
WheelMouse m = new Mouse(); //Mouse 중에는 WheelMouse가 아닌 것도 있으니까
```

1.7 변수의 타입은 '~계열', '~종류', '~상표'라는 의미가 있습니다.

변수의 타입이라는 것을 이해하기 위해서는 두 가지 방식의 이해를 수반하면 좋습니다. 우선은 변수의 타입을 논리적인 것으로 이해하는 것과 변수의 타입을 메모리상으로 이해하는 방식을 겸하면 가장 좋다고 생각합니다. 논리적으로 변수의 타입을 이해하자면 변수의 타입이 객체 자료형은 경우에는 '~계열이나 ~종류', 혹은 '브랜드나 ~ 직원' 같은 것으로 생각하시면 됩니다.

- 건축가 a = new 아무개( ); ← 만일 아무개의 직업이 건축가라면 논리적으로 성립한다.
- 자동차 a = new 포르셰( ); ← 포르셰 역시 자동차의 일종이므로 성립 가능하다.
- MP3 a = new 아이팟( ); ← 아이팟이 MP3로 분류될 수 있다면 가능하다.
- Food a = new 햄버거( ); ← 전체집합과 부분집합의 관계이므로 가능하다.

변수의 타입을 메모리로 이해하면 조금 독특한 방식으로 구성될 수 있습니다. 우리가 new라는 키워드를 이용해서 어떠한 객체를 만들면 메모리 공간에 실제 객체가 만들어지게 됩니다. 변수의 타입과 실제 객체가 일치하는 'Mouse m = new Mouse( );'의 코드의 경우 변수 안에는 Mouse의 리모컨이 담기기 때문에 아무런 문제가 없이 실행됩니다.

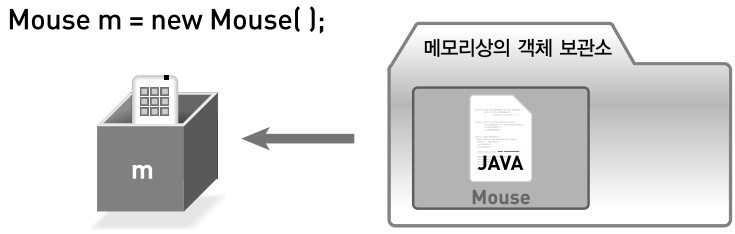

그림 8

만일 Mouse m = new WheelMouse( ) 라는 방식을 사용하게 되면 다음과 같은 메모리 구조가 됩니다.

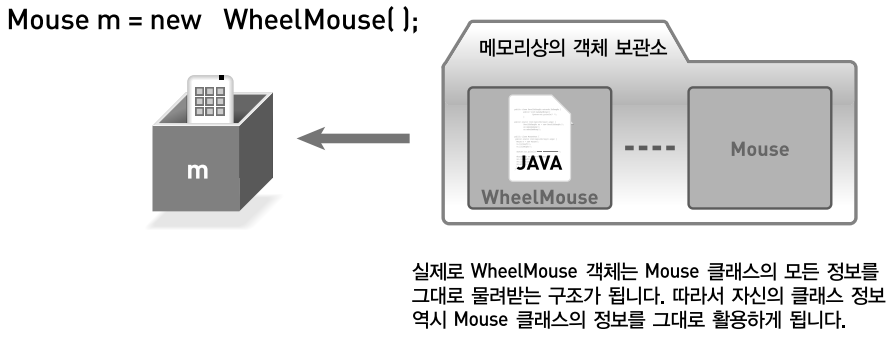

그림 9

WheelMouse 클래스의 객체를 Mouse 타입으로 볼 수 있는 것은 Mouse 클래스의 정보를 WheelMouse가 그대로 물려받으면서 타입의 정보마저도 물려받게 됩니다.

■ **변수의 타입은 컴파일러를 위한 겁니다.**

> 컴파일러는 변수의 타입을 보고 메모리상에 상자를 만들어줍니다. 따라서 실제 객체가 현재 선언된 타입의 정보를 가지고 있는가만 신경 씁니다.

변수를 최초로 선언할 때 앞에 변수의 타입이라는 것을 붙여 줍니다.

```
int a ;
```

이 경우에는 컴파일러가 a라는 변수 상자에 int에 속하는 데이터를 담겠다는 의미입니다.

```
Mouse m;
```

컴파일러는 변수 m이라는 상자에 앞으로 Mouse 계열의 리모컨이 담긴다고 생각합니다. 따라서 컴파일러는 이하의 코드가 Mouse라는 클래스에 정의된 데이터이거나 메소드가 문법적으로 올바르면 정상적이라고 생각해주는 겁니다.

■ **컴파일 타임이라는 것이 있습니다.**

프로그램을 만들다 보면 컴파일이라는 것이 있고, 실행이라는 것이 있습니다. 때로는 전자를 컴파일 타임이라고 하고, 후자를 런타임(Runtime)이라고 합니다. Java는 컴파일 시점에 로직을 점검하는 것이 아닙니다. 컴파일이라는 작업 자체가 결국은 기계어로 잘 번역이 될 수 있는 유효한 코드인지를 살펴보는 가장 중요한 작업입니다. 반면에 런타임은 실행하면서 필요하다면 메모리상에 객체를 만들거나 기존의 객체를 활용하게 됩니다.

변수의 선언 앞쪽에 있는 타입이라는 것은 사실을 컴파일러를 위한 겁니다. 컴파일러가 이 소스를 실행하기 위해서 문법적인 문제가 없는지를 미리 살펴보는데 변수의 타입이 필요합니다.

```
Mouse m = null;
m.leftClick( );
```

예를 들어 앞과 같은 코드는 m이라는 상자에 null이 할당되었기 때문에 실제로 실행할 수 없는 코드이지만 컴파일러는 이것을 고려하지 않고 있습니다. 컴파일러는 위의 코드를 정상적으로 컴파일시켜줍니다. 즉 컴파일러는 m이라는 변수의 타입이 Mouse이고, 그 안에 leftClick( )이라는 메소드만 존재한다면 실제로 실행 시점에 어떻게 되는지는 생각하지 않습니다.

1.8 변수의 타입에 상관없이 실행되는 것은 결국은 객체

변수의 타입의 상자의 종류를 의미하지만 담기는 리모컨(레퍼런스)은 결국은 어떤 객체의 리모컨입니다. 따라서 실제 리모컨을 이용해서 어떤 메소드를 실행하면 실제 움직이는 것은 리모컨이 가리키는 객체입니다.

WheelMouse가 참 많은 인기를 누리다 보니 요즘 나오는 모든 마우스의 기본은 WheelMouse입니다. 여기에 추가적인 기능이 있긴 하지만 기본은 WheelMouse입니다. 그런데 예전에는 마우스 안쪽에 커다란 고무 공이 들어 있었습니다. 요즘 나오는 마우스들은 거의 광학 센서를 이용합니다. 따라서 동작 자체는 완전히 동일한 기능을 수행하는데, 동작이 실행되는 방식이 완전히 다릅니다. 그럼 WheelMouse를 상속해서 OpticalMouse를 만들어 볼까요?

예제 | WheelMouse를 상속한 OpticalMouse

```
public class OpticalMouse extends WheelMouse {

}
```

1.8.1 OpticalMouse? 기존의 기능을 대체할 뿐

OpticalMouse는 뭔가 새로운 기능이 있는 것이 아닙니다. 다만, 그 실행되는 방식이 내부적으로만 다를 뿐입니다. 이런 경우는 어떻게 표현해야 할까요? 간단합니다. 그냥 다시 만들고 싶은 메소드를 그대로 가져다가 고쳐 주시면 됩니다.

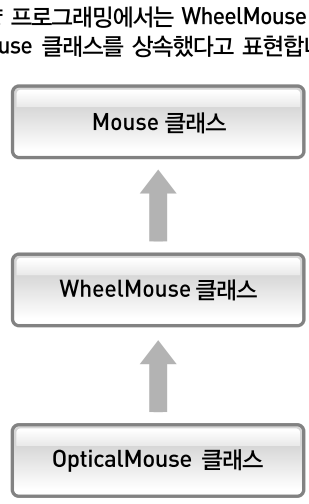

그림 10

상속 구조를 보면 OpticalMouse는 저 멀리 Mouse에서부터 clickLeft( )와 clickRight( )의 기능을 그대로 물려받았습니다. 따라서 내부적으로 다르게 동작하는 clickLeft( )와 clickRight( )를 그대로 OpticalMouse 클래스에 구현해둘까 합니다.

예제 물려받은 기능을 그대로 다시 구현한 clickLeft( ), clickRight( )

```
public class OpticalMouse extends WheelMouse {
      public void clickLeft(){
            System.out.println("광센서로 왼쪽 클릭");
      }

      public void clickRight(){
            System.out.println("광센서로 오른쪽 클릭");
      }
}
```

이제 MouseTest에서 이것을 다시 테스트해 보도록 할까요?

1.8.2 'Mouse op = new OpticalMouse( );'는 가능합니다만….

변수의 타입이라는 것은 '~계열'을 의미할 수 있기 때문에 당연히 'Mouse op = new OpticalMouse( );' 선언은 가능합니다. 그런데 다음 코드의 실행 결과는 어떻게 될까요?

```
op.clickLeft();
op.clickRight();
```

이렇게 호출하면 어떤 결과가 출력될까요? '오른쪽 클릭, 왼쪽 클릭'일까요? 아니면 '광센서로 오른쪽 클릭, 광센서로 왼쪽 클릭'일까요? 객체를 만들어서 실험해봅니다.

예제 항상 호출되는 것은 결국 실제 객체의 메소드

```
public class MouseTest {
      public static void main(String[] args) {
            System.out.println("--------------------");
            Mouse op  = new OpticalMouse();
            op.clickLeft();
            op.clickRight();
      }
}
```

```
--------------------
광센서로 왼쪽 클릭
광센서로 오른쪽 클릭
```

실행된 결과를 보시면 '광센서~'가 출력되는 것을 보실 수 있습니다. 변수의 타입이 아무리 Mouse 타입이라고 해도 결과적으로 메소드를 실행하는 객체는 OpticalMouse 클래스의 객체입니다. clickLeft( ) 메소드와 clickRight( ) 메소드를 실행하는 코드를 작성할 때에는 변수의 타입을 중요하게 보지만(컴파일 타임), 실행될 때에는(런타임) 실제 객체가 해당 메소드가 있는지를 더 중요하게 생각합니다.

1.8.3 실행되는 메소드는 실제 객체의 메소드가 호출

변수의 타입이 무엇이든 간에 실제로 움직이는 것은 리모컨(레퍼런스)에 연결된 객체입니다.

- 컴파일러는 변수 타입으로 선언된 클래스에 해당하는 메소드가 있는지만 확인합니다.
- 실제로 움직이는 것은 연결된 객체입니다.
- 만일 실제 객체에 해당 메소드가 없다면 부모 클래스의 정보를 추적해서 실행합니다

상속을 하면 변수의 타입과 실제 객체가 일치하지 않을 수 있습니다. 하지만, 그렇다고 해서 실제로 객체가 바뀌치기 당하는 것은 아닙니다. 즉 아무리 변수의 타입이 부모가 아니라 조부모, 증조부모라고 해도 실제로 움직이는 대상은 결국 실제 객체입니다. 즉 컴파일러는 변수의 선언 시에 사용된 클래스의 메소드를 호출하는 것만이 중요한 반면에 실행 시점에는 실제 객체가 그 메소드를 가졌는지가 중요합니다.

■ **원하는 메소드가 없다면 부모 클래스를 찾아서 실행하려 합니다.**

WheelMouse는 Mouse를 상속했기 때문에 Mouse 안에 있는 두 개의 메소드를 찾을 수 있는 구조입니다. 따라서 WheelMouse 클래스의 객체라고 해도 clickLeft( ) 메소드를 호출하게 되면 우선은 현재 클래스에 clickLeft( )라는 메소드가 있는지를 먼저 검사하고, 만일 메소드가 없다면 부모 클래스의 메소드를 호출하는 방식으로 동작하게 됩니다.

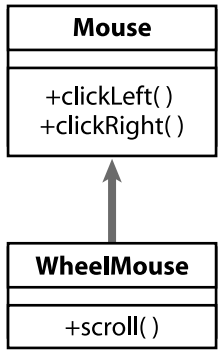

그림 11

■ **컴파일러가 보는 것은 단순한 클래스의 정보만 바라봅니다.**

'Mouse m = new WheelMouse( );'라고 선언되었을 때 컴파일러가 따져 보는 것은 실제 어떤 동작을 하는지를 보는 것이 아니라, 정확한 타입으로 기술이 되는지만을 살펴보게 됩니다. 여러분이 변수 m.xxx( ) 메소드를 호출하는 코드에 대한 문법적인 오류만을 점검하기 때문에 실행 시와는 전혀 다르게 동작한다고 생각해주어야 합니다.

1.9 오버라이드(Override): 자식 클래스에서 부모 클래스의 메소드를 다시 정의

상속을 하면 메모리상에서 해당 클래스와 해당 클래스의 부모 클래스가 서로 연결되는 구조가 만들어집니다. 따라서 실제 클래스에서부터 해당 메소드를 찾아서 실행 → 없는 경우에 부모 클래스를 찾아 올라가는 방식으로 실행되게 됩니다. 반면에 컴파일러는 변수의 타입 정보를 우선으로 보기 때문에 해당 클래스에 실행하려는 메소드가 존재하는 것만으로 충분합니다. 이런 경우에 만일 실행되는 클래스가 부모 클래스의 메소드를 동일하게 정의하게 되면 내부적으로 부모 클래스의 메소드를 실행하지 않고, 현재 클래스의 메소드를 실행하게 되는데, 이것을 오버라이드(Override)라고 합니다.

오버라이드(Override)라는 것은 부모 클래스의 메소드를 동일하게 자식 클래스에서 다시 작성한 것을 의미합니다.

```
Mouse a = new WheelMouse( );
a.clickLeft( );
```

컴파일러는 Mouse 클래스에 clickLeft( ) 메소드만 확인하지만, 실제 a라는 변수에 들어 있는 리모컨(레퍼런스)는 WheelMouse 클래스에 작성된 clickLeft( )를 실행합니다.

오버라이드는 간단히 말하면 부모 클래스에 어떤 메소드가 있었는데, 그 메소드를 자식 클래스에서 다시 재정의한다는 것입니다(흔히 '오버라이드한다'고 표현합니다). 따라서 컴파일러는 재정의되었는지 어쩐지 모르고, 그냥 문법적인 부분만 형식적으로 검사합니다. 하지만, 실행 시점에는 움직이는 객체 자체의 메소드가 호출되기 때문에 재정의된 메소드가 호출되는 겁니다.

1.10 부모 클래스, 자식 클래스 그리고 오버라이드

상속을 이용하다 보면 부모 클래스와 자식 클래스 간에 서로 약간의 계약이나 규칙이 존재합니다. 몇 가지 상황을 고려해봅시다.

- 특정 메소드를 자식 클래스에서 오버라이드를 하면 부모 클래스의 메소드는 없어져 버리는데 가끔은 부모 클래스의 메소드를 다시 사용하려고 하는 경우에는 어떻게 해야할까?
- 부모 클래스가 객체를 생산하는 방법을 제한하기 위해서 생성자를 사용했다면 자식 클래스는 생성자가 없어도 괜찮을까?

각각의 상황을 해결하는 데 여러분에게 도움이 될 만한 키워드가 super입니다.

1.10.1 super는 부모 클래스를 부르는 키워드

super라는 키워드는 부모 클래스를 호출하거나 이용할 때 사용하는 키워드입니다. this라는 키워드가 현재 객체를 의미한다면 super라는 것은 자신의 부모를 소환(?)하는 키워드입니다.

■ 자장과 옛날 자장의 관계

다시 한번 자장루의 문제로 돌아가 볼까요? 만일 부모 클래스에서 물려받은 makeZaZang( )을 오버라이드하면 원래 부모 클래스에서 물려받은 makeZaZang( )은 더는 사용할 수 없게 됩니다.

예제 | ZaZangRu의 makeZaZang( )을 override한 SeoulZaZangRu

```
public class SeoulZaZangRu extends ZaZangRu {
      public void makeZaZang(){
            System.out.println("돼지고기와 식물성 기름을 사용합니다.");
            System.out.println("현대식 자장면을 만듭니다.");
      }
      public static void main(String[] args) {
            ZaZangRu sz = new SeoulZaZangRu();
            sz.makeZaZang();
      }
}
```

```
돼지고기와 식물성 기름을 사용합니다.
현대식 자장면을 만듭니다.
```

결과를 보시면 아시겠지만, 부모의 makeZaZang( )은 더는 호출되지 않습니다.

■ 옛날 자장이라는 메뉴로 부활하는 부모 클래스의 makeZaZang( )

요즘 중국집에 가 보면 옛날 자장이라는 메뉴로 과거처럼 감자가 들어간 자장을 판매하는 곳을 간혹 볼 수 있습니다. 이것을 프로그램으로 만들면 어떻게 될까요? 지금 문제는 부모 클래스의 메소드를 다시 살려내는 겁니다. 앞에서 제가 super라는 키워드에 소환이라는 단어를 사용한 이유도 이것 때문입니다. 오버라이드하면 기존의 부모가 가진 메소드 대신에 현재 오버라이드 된 메소드가 그 기능을 대신하게 되는데, super라는 키워드를 이용해주면 다시 살려낼 수가 있습니다. SeoulZaZangRu에 makeOldZaZang( )이라는 메소드를 통해서 살려보도록 합니다.

예제 | super라는 키워드를 이용하면 부모 클래스 쪽을 호출할 수 있습니다.

```
public class SeoulZaZangRu extends ZaZangRu {
	public void makeOldZaZang(){
		super.makeZaZang();
	}

	public void makeZaZang(){
		System.out.println("돼지고기와 식물성 기름을 사용합니다.");
		System.out.println("현대식 자장면을 만듭니다.");
	}

	public void makeZamBong(){
		System.out.println("짬뽕을 만들 수 있습니다.");
	}
	public static void main(String[] args) {
		SeoulZaZangRu sz = new SeoulZaZangRu();
		sz.makeOldZaZang();
		sz.makeZaZang();
		sz.makeZamBong();
	}
}
```

```
감자와 돼지고기, 자장을 볶습니다.
자장면을 만듭니다.
돼지고기와 식물성 기름을 사용합니다.
현대식 자장면을 만듭니다.
짬뽕을 만들 수 있습니다.
```

오버라이드하면 자식 클래스는 부모 클래스의 원래의 메소드를 잃어버리는 단점을 가지게 됩니다. 즉 재정의하면서 기존 부모 클래스의 메소드의 호출할 수 있는 고리를 잃어버리는 겁니다. 하지만, super라는 키워드를 활용하면 부모에서 물려받은 원래의 메소드를 사용할 수 있도록 합니다.

1.10.2 super의 또 다른 용도: 부모의 생성자를 소환

super라는 키워드의 또 다른 사용처는 바로 생성자에서 사용됩니다. 우선 생성자라는 것에 대해서 논리적으로 생각해볼 필요가 있습니다. 부모 클래스가 객체를 생성하기 위해서 생성자를 사용했다고 가정해봅시다. 생성자를 사용했다는 의미는 결국 객체를 만들 때 반드시 어떤 데이터를 넣고 싶거나, 객체가 만들어지면서 어떤 메소드를 실행하겠다는 것을 의미합니다. 만일, 부모 클

래스에서 생성자를 사용했다면 자식 클래스에서는 이와 관계된 어떤 조치가 있는 것이 더 합당하지 않을까요?

> 만일 부모 클래스에서 생성자를 만들면 반드시 자식 클래스 역시 생성자를 사용해야 하는 불편함이 있습니다. 이것은 생성자가 객체를 만드는 조건을 의미하기 때문입니다.

간단히 예제에서는 일반 Ball과 CannonBall의 관계를 살펴볼까 합니다. 일반 공에는 반지름이 있다는 속성을 줄 것이고, CannonBall에는 반지름과 같이 무게를 데이터로 갖도록 해봅니다.

예제 | 반지름 데이터를 가지는 Ball 클래스

```
public class Ball {
	private double radius;

	public Ball(double radius){
		this.radius = radius;
	}

	public String toString(){
		return "이 공의 반지름은"+radius;
	}
}
```

모든 Ball은 반지름이 필수 데이터라는 생각에 생성자를 추가해 주었습니다. 문제는 Ball을 부모 클래스로 하는 CannonBall에서 발생합니다. 우선은 클래스를 선언해봤는데 컴파일이 안 되는 문제가 발생합니다.

예제 | 부모 클래스에 생성자가 있으면 자식 클래스에서도 필요합니다. CannonBall 클래스

```
public class CannonBall extends Ball {
//컴파일 오류

}
```

생성자라는 것이 객체의 생산에 제약을 거는 방법이기 때문에 당연히 부모 클래스가 생성자를 이용하게 되면 자식은 무언가 선택의 여지를 가져야 합니다. 부모의 생성자를 그대로 유지할 것인가? 아니면 자신이 스스로 갈 것인가? 이클립스를 이용하면 위와 같은 경우에 전자를 선택합니다. 즉 부모의 생성자를 그대로 따라서 자식 클래스에서도 생성자를 만들라고 도와줍니다.

예제 | 이클립스가 도와준 자식 클래스의 생성자 코드

```
public class CannonBall extends Ball {
      public CannonBall(double radius) {
            super(radius);
      }
}
```

이클립스에서 에러 메시지를 보고 마우스 오른쪽 버튼을 누르면 추가적인 옵션으로 만든 생성자입니다. 보시면 super( )라는 부분이 들어가 있습니다. 이것은 부모 클래스 생성자의 내용을 그대로 따르겠다는 의미입니다. this라는 키워드와 super라는 키워드의 관계를 정리해보면 다음과 같습니다.

	'.'를 이용해서 사용할 때	생성자에서 사용할 때
this	현재 코드를 실행하는 객체의 데이터나 메소드를 호출할 때 사용 ex) this.doA( );	현재 클래스의 또 다른 생성자를 호출하고자 할 때 사용하는 방식. 즉 같은 클래스의 또 다른 생성자를 호출할 때 사용한다.
super	현재 코드에서 명시적으로 부모 클래스의 속성이나 메소드를 호출할 때 사용 ex) super.doA( );	상속 관계에서 부모 클래스에 정의된 생성자를 사용하는 방식, 실제 호출되는 생성자는 부모 클래스 안에 정의해 둔 생성자가 호출된다.

표

1.11 private은 상속으로 보지 마시고, 접근 제한자라고만 생각합시다.

부모 클래스에 private이 붙은 메소드나 변수는 기본적으로 접근할 수 없는 것이지 상속할 수 없는 것은 아닙니다. 그냥 private은 상속의 개념으로 이해하지 마시고, 외부 클래스에서는 접근할 수 없다는 접근 제한의 원리로만 파악해 주시면 됩니다.

> private을 이용하면 상속이 안 되는 것이 아니라 접근을 못 한다는 개념으로 이해해야만 합니다.

2 상속을 이용하면 빠르게 개발할 수 있을까?

객체지향 패러다임을 이용하는 가장 중요한 이유는 나중에 시스템의 확장이나 유지보수에 유리하기 때문입니다만, 그냥 현실적으로는 소스를 작성할 때 내용이 줄어들고, 기존 소스 코드를 수정하는 부분이 적어져서 개발자들의 더 빠른 개발이 가능해집니다. 그럼 앞에서는 새로운 코드를 작성하는 사람이 좀 더 편리해지는 모습을 보았으니 이제는 코드를 이용하는 사람의 입장에서도 생각해 보도록 합니다. 유치하긴 하지만 상황에 맞는 예제를 구성해보도록 하겠습니다.

■ 인형 뽑기 기계로 본: 상속을 모르는 개발자의 하루

"안녕하세요. 뽑기 맨 아저씨. 전 학교가 끝나고 매일 집에 오면서 뽑기를 해요. 근데 한 한 달 동안 만날 같은 인형들만 나오니까 심심해요. 다른 기계들 보면 열쇠고리나 풍선껌도 나오던데…." 만일 여러분이 뽑기 기계를 개발하는 사람인데 이런 메일을 받았다고 생각해 봅니다(물론 여러분이 부지런한 개발자가 아니라면 이 편지는 살포시 접혀서 쓰레기통으로 가겠지만). 뽑기 기계에는 분명히 인형(Toy)이 나오는 메소드가 하나 있을 겁니다.

예제 이해를 돕고자 구성한 Toy 클래스

```
public class Toy {
      private String name;

      public String getName() {
            return name;
      }

      public void setName(String name) {
            this.name = name;
      }
}
```

예제 | 이해를 돕고자 구성한 인형 뽑기 기계

```
public class CatchToyMachine {
      private Toy[] toys;

      public CatchToyMachine(){
            toys = new Toy[10];
            for(int i = 0; i < toys.length; i++){
                  Toy toy = new Toy();
                  toy.setName("인형"+i);
            }
      }

      public Toy catchToy(){
            Toy toy = null;
            //인형을 뽑는 로직
            return toy;
      }
}
```

객체로 선언하는 것은 배열로 선언할 수 있다는 개념을 활용해서 기계 안에 여러 개의 Toy 객체를 넣어주는 형태로 구성한 클래스입니다.

2.1 변수의 타입과 실제 객체가 일치하지 않아도 된다는 장점

상속은 변수의 타입과 실제 객체의 클래스가 일치하지 않아도 됩니다. 즉 'Mouse m = new WheelMouse( );'처럼 실제와 타입은 다르게 할 수 있습니다. 위의 개발자에게 상속을 알려주시면 다음과 같은 구조를 만들 수 있습니다.

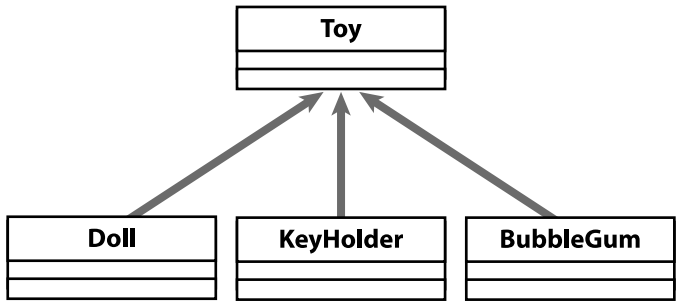

그림 12

Toy라는 하나의 타입으로 세 가지 종류의 객체를 나타낼 수 있는 유연한 방식의 표현이 가능해 집니다.

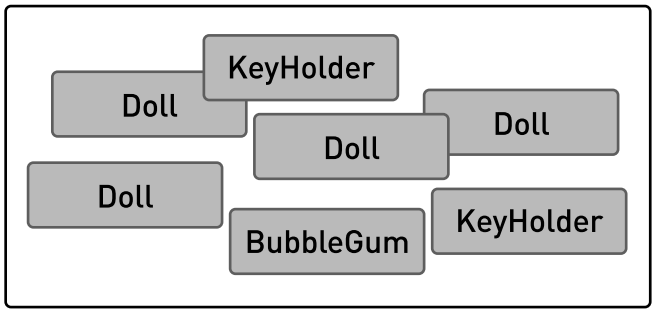

그림 13 인형 뽑기 기계 안의 객체들

변수로 선언할 수 있는 것이 1) 메소드의 파라미터로 사용된다, 2) 메소드의 리턴 타입으로 사용될 수 있다, 3) 자료구조로 선언할 수 있다는 것을 생각해 본다면 위의 그림처럼 인형 뽑기 기계가 Toy 타입의 배열을 가지는 방식의 설계가 가능해 집니다. 우선은 각각의 클래스를 작성해보도록 합니다.

```
Toy a = new Doll( );
Toy b = new KeyHolder( );
Toy c = new BubbleGum( );
```

예제 Toy를 상속한 Doll 클래스

```
public class Doll extends Toy{
      //인형 사이즈
      private String size;

      public String getSize() {
            return size;
      }

      public void setSize(String size) {
            this.size = size;
      }
}
```

예제 | Toy를 상속한 KeyHolder 클래스

```
public class KeyHolder extends Toy{
	//재질
	private String material;

	public String getMaterial() {
		return material;
	}

	public void setMaterial(String material) {
		this.material = material;
	}
}
```

예제 | Toy를 상속한 BubbleGum 클래스

```
public class BubbleGum extends Toy{
	private String color;

	public String getColor() {
		return color;
	}

	public void setColor(String color) {
		this.color = color;
	}
}
```

위와 같은 선언이 가능하다면 개발자는 'public Toy catchToy( ){ ... }'을 만들어 주면 메소드를 여러 개 만들지 않아도 될 듯합니다. 사실 이 부분이 상속을 이용하는 가장 큰 이유입니다. 하나의 부모 타입만 알면 실제 자식 클래스는 뒤에 여러 가지가 올 수 있습니다. 아직은 한 번에 이해하실 필요는 없습니다.

코드 |

```
Toy a = new Doll( ); //Toy가 부모 클래스이기 때문에 가능한 코드입니다.
Toy a = new KeyHolder( ) ;
Toy a = new  BubbleGum( );
```

앞과 같은 코드가 가능하면 Toy[ ] 역시 가능해집니다.

예제 인형 뽑기 기계 안에 여러 가지 장난감을 넣어주지만 변경되는 부분은 적게 할 수 있습니다.

```
public class CatchToyMachine {
      private Toy[] toys;

      public CatchToyMachine(){
            toys = new Toy[10];
            for(int i = 0; i < toys.length; i++){
                  Toy toy = null;
                  if(i % 3 == 0){
                        toy = new Doll();
                  }else if(i % 3 == 1 ){
                        toy = new KeyHolder();
                  }else if(i % 3 == 2){
                        toy = new BubbleGum();
                  }
                  toys[i] = toy;
            }
      }
      // 아직 미구현
      public Toy catchToy(){
            Toy toy = null;
            //인형을 뽑는 로직
            return toy;
      }
}
```

코드에서 유심히 봐야 하는 부분은 'Toy[ ] toys' 코드입니다. 생성자 안에서는 루프를 돌면서 여러 가지 Toy의 하위 클래스로부터 객체를 생성하고 있습니다만 기존의 다른 부분은 변경이 전혀 없습니다.

상속을 사용하면 하나의 변수 타입으로 여러 가지 종류의 객체를 가리킬 수 있기 때문에 기존 코드를 고치지 않고도 처리할 수 있는 부분이 많아집니다. 이것을 전문 용어로 다형성(Polymorphism)이라고 합니다. Java에서는 상속과 인터페이스를 이용해서 다형성을 구현할 수 있습니다. 흔히들 상속과 다형성이라고 나누어서 설명하지만 실제로 상속을 통해서 다형성을 구현할 수 있다는 표현이 더 정확한 표현입니다.

2.2 부모 타입으로 파라미터를 선언할 수 있습니다.

상속을 이용하면 메소드의 파라미터를 부모 타입으로 선언해줄 수 있습니다. 예를 들어 MouseTest2를 새로 아래와 같이 만들어봅시다.

예제 | 부모 타입으로 파라미터를 선언할 수 있습니다.

```
public class MouseTest2 {
	public void testMouse(Mouse m){
		m.clickLeft();
		m.clickRight();
	}

	public static void main(String[] args) {
		MouseTest2 mt2 = new MouseTest2();
		Mouse m = new Mouse();
		mt2.testMouse(m);
	}
}
```

```
왼쪽 클릭
오른쪽 클릭
```

testMouse( )를 보면 파라미터로 Mouse 타입의 객체의 리모컨을 요구합니다. main 메소드에서 Mouse 객체를 만들어서 testMouse( )에 넣어주면 정상적으로 실행되는 것이 보입니다.

■ **testMouse(Mouse m)에 WheelMouse나 OpticalMouse 객체의 리모컨을 넣어도 전혀 문제 없습니다.**

testMouse( )의 파라미터로 WheelMouse 객체나 OpticalMouse 객체를 넣어봅시다.

코드 |

```
MouseTest2 mt2 = new MouseTest2();
Mouse m = new OpticalMouse();
mt2.testMouse(m);
```

```
광센서로 왼쪽 클릭
광센서로 오른쪽 클릭
```

결과를 테스트해보면 기존의 메소드를 전혀 수정하지 않고도 기존 메소드를 사용할 수 있는 것을 보실 수 있습니다.

2.3 부모 타입으로 배열도 선언할 수 있습니다.

배열이란 결국 변수의 탑이기 때문에 부모 타입의 변수를 여러 개 쌓아서 배열로 만들어 줄 수도 있습니다.

예제 | Mouse의 배열에는 Mouse, WheelMouse, OpticalMouse 객체의 리모컨들이 들어갈 수 있다.

```
public class MouseTest3 {
      public static void main(String[] args) {
            Mouse[] arr = new Mouse[3];
            arr[0] = new Mouse();
            arr[1] = new WheelMouse();
            arr[2] = new OpticalMouse();

            for(int i = 0; i < arr.length; i++){
                  arr[i].clickLeft();
                  arr[i].clickRight();
                  System.out.println("--------------------");
            }
      }
}
```

```
왼쪽 클릭
오른쪽 클릭
--------------------
왼쪽 클릭
오른쪽 클릭
--------------------
광센서로 왼쪽 클릭
광센서로 오른쪽 클릭
--------------------
```

■ 부모 타입으로 리턴 타입을 설정할 수도 있습니다.

메소드의 파라미터, 리턴 타입, 배열을 변수로 선언할 수 있으므로 당연히 메소드의 리턴 타입으로 Mouse를 이용할 수도 있습니다.

변수로 어떤 객체를 선언한다는 것이 다음과 같은 의미가 있다는 것을 생각하시면 됩니다.

- 변수로 선언하는 것은 메소드의 파라미터로 사용될 수 있습니다. 따라서 상속도 변수 선언 시에 사용하는 방식을 그대로 활용할 수 있습니다. 파라미터로 'Mouse m'을 사용하게 되면 WheelMouse 객체를 파라미터로 넣어줄 수 있습니다.
- 변수로 선언할 수 있는 것은 메소드의 리턴 타입으로 사용할 수 있습니다. 따라서 리턴 타입은 Mouse이지만 실제로는 WheelMouse 객체를 반환할 수 있습니다.
- 변수로 선언할 수 있는 것은 배열과 같은 자료구조로 선언할 수 있습니다. 결국, 배열은 변수들이 쌓인 형태이므로 Mouse의 배열에 Mouse 객체뿐 아니라, WheelMouse 객체나 OpticalMouse 객체를 배열에 넣어줄 수 있습니다.

2.4 상속이라는 것은 개발 시간을 단축시킵니다.

■ **새로운 기능이나 새로운 클래스를 작성하는 개발자**

extends를 이용해서 어떤 클래스를 상속해주면 중복되는 코드를 다시 만들 필요가 없습니다. 그냥 extends라는 키워드만 쓰면 됩니다. 또 필요하면 부모의 메소드를 다시 스스로 재정의(Override)해서 실행될 때 자신이 다시 정의한 메소드가 실행될 수 있도록 해주면 됩니다.

■ **기존 코드를 유지보수하는 개발자**

기존 코드를 유지보수하는 개발자들은 상속 구조를 이용하면 자신의 코드는 손대지 않고, 시스템에 새로운 객체를 파라미터나, 리턴 타입, 자료구조(배열) 등에 사용할 수 있게 됩니다. 즉 나중에 파라미터에 새로운 자식 클래스를 던지더라도 자신은 아무런 일도 안 해도 됩니다.

3 상속을 이용하면 if ~ else를 없앨 수도 있습니다.

상속을 제대로 이해했다면 그 결과는 간단합니다. 즉 코드가 다양한 종류의 객체를 처리할 수 있기 때문에 예전보다 유연해져서 특별히 새로운 내용이 없어도 부모 클래스를 상속한 어떠한 객체도 처리할 수 있어야 합니다. 이것은 '개방과 폐쇄의 원칙(Open Close Principle)'이라고 하는데, 즉 시스템이 새로운 확장이나 변경에는 얼마든지 열려 있지만, 기존의 코드를 수정하는 데는 닫혀 있다는 겁니다. 지금부터 몇 개의 예제를 통해서 이런 목표를 어떻게 이루는 것인지 알아보도록 합시다.

3.1 '뚜벅이가 부산 여행가기'를 만들어 봅시다.

이 아이디어는 모 TV프로그램을 보고 얻은 아이디어입니다. 만일 서울에 사는 뚜벅이인 제가 부산에 여행을 가려면 어떻게 갈 수 있을까요?

- 부산행 기차를 타고 간다.
- 부산행 버스를 타고 간다.
- 지인이 부산 가는 편에 눈치껏 끼어 타고 간다.

다양한 방법이 있을 듯합니다. 목표는 하나입니다. 부산까지 가는 것. 만일 이런 경우라면 다음과 같은 그림으로 교통편이 표현될 수 있을 듯합니다.

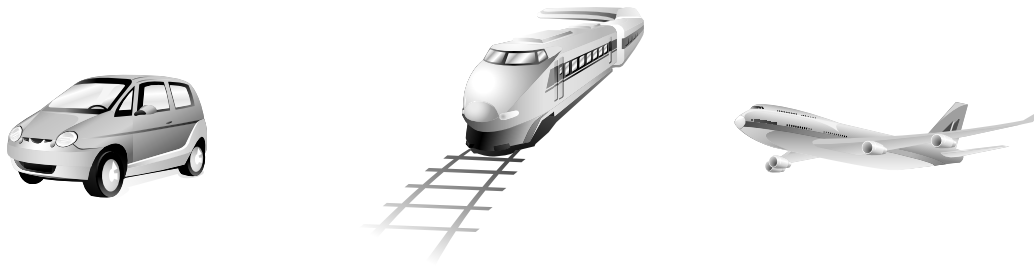

그림 14 자동차, 기차, 비행기

우선은 상속이 없는 방법으로 해볼까요? 개념을 돕기 위한 예제이니 굳이 실습하실 필요는 없습니다.

예제

```
public class Bus{
      public void goBus(){
            System.out.println("부산행 버스에 몸을 싣고");
      }
}
```

```
public class Train{
      public void goTrain(){
            System.out.println("부산행 기차에 몸을 싣고...");
      }
}
```

이제 부산 가는 것을 클래스로 표현해 보면 아래처럼 만들어야 할 듯합니다.

예제

```
public class BusanTravel {
      public void goBusan(Bus b){
            b.goBus();
      }

      public void goBusan(Train t){
            t.goTrain();
      }
}
```

아, 이렇게 했다가는 비행기를 타거나, 오토바이를 타거나, 자전거 여행을 할 때마다 너무 힘들어 지겠습니다. 매번 클래스에 메소드를 교통편에 따라서 완전히 새로 만들어야 하니까요.

■ **상속을 써서 메소드를 간단하게 만들 수는 있습니다.**

우선은 지금은 교통편에 따라서 goBusan( ) 메소드를 매번 새로 만들어 줘야 하니 환장할 노릇입니다. 이제 Bus와 Train이 상속 구조를 가지게 수정해봅니다. 모든 교통편의 부모 클래스를 Transportation이라는 클래스로 삼고자 합니다. 아직은 아무 내용도 없이 그냥 상속만 하겠습니다.

예제 | 상속 구조를 이용하면 여러 종류의 객체를 하나의 타입으로 볼 수 있습니다.

```
public class Transportation {
      //아무 내용도 없음
}
```

```
public class Bus extends Transportation {
      public void goBus(){
            System.out.println("부산행 버스에 몸을 싣고");
      }
}
```

```
public class Train extends Transportation{
      public void goTrain(){
            System.out.println("부산행 기차에 몸을 싣고...");
      }
}
```

이제 부모 클래스로 goBusan( )을 다시 만들면 좀 나아집니다.

코드

```
public void goBusan(Transportation t){

}
```

흠, 이제 메소드가 좀 간단해졌네요. 그런데 문제는 여전히 해결되지 않았습니다. 왜냐하면 goBusan( ) 메소드 안에는 다음과 같은 로직이 들어가야 하기 때문입니다.

"만일 버스라면 goBus( )를 호출하고, 만일 기차라면 goTrain( )을 호출한다(if ~ else 코드 추가)."

이렇게 되면 새로운 교통편이 나왔을 때 결국 if ~ else 코드가 더 늘어나는 수밖에 없게 됩니다. 따라서 이렇게 하지 말고 아예 컴파일러를 속이는 게 어떨까요?

3.2 오버라이드를 잘 이용하면 간단한 코드로 if ~ else를 없앨 수 있습니다.

Transportation에 go( )라는 메소드를 하나 선언합니다. 그런데 이 메소드는 그냥 텅 비어 있거나 아니면 간단한 실행만 되는 로직만 만들어봅니다.

예제 go( ) 메소드가 하나 추가된 Transportation 클래스

```
public class Transportation {
      public void go(){
            System.out.println("알아서 갑니다.");
      }
}
```

이제는 부산으로 가는 메소드를 다음과 같이 수정할 수 있습니다.

코드

```
public void goBusan(Transportation t){
      t.go();
}
```

오버라이드의 효과를 이용해 보았습니다. 상속에서 오버라이드(Override)를 이용하면 컴파일러는 타입으로 선언된 메소드를 따지지만 실제로 실행될 때에는 실제 객체의 메소드가 실행된다는 사실 말입니다. 이제 Train이나 Bus는 부모 클래스의 go( )라는 메소드를 오버라이드해서 다시 구성해주면 됩니다.

예제 | Bus에 go( ) 메소드를 오버라이드 / Train에 go( ) 메소드를 오버라이드

```
public class Bus extends Transportation {
      //override
      public void go(){
            System.out.println("부산행 버스에 몸을 싣고");
      }
}
```

```
public class Train extends Transportation{
      //override
      public void go(){
            System.out.println("부산행 기차에 몸을 싣고...");
      }
}
```

이제 부산으로 갈 때의 메소드를 마지막으로 한 번 더 살펴봅니다.

코드 |

```
public void goBusan(Transportation t){
      t.go();
}
```

컴파일러는 Transportation 클래스에 go( )라는 메소드가 있으니 컴파일을 통과시킵니다. 그런데 메소드에 Bus의 객체나 Train의 객체를 넣어주면 어떤가요? 두 객체 모두 go( )라는 메소드가 있기 때문에 실행 타임에는 진짜 객체의 go( )가 호출되게 되는 겁니다. goBusan( ) 메소드에는 Transportation 클래스를 상속한 모든 클래스의 객체를 넣어주어도 됩니다. 그리고 그 객체에는 go( )라는 메소드를 오버라이드해주면 됩니다.

상속을 이용하고, 하나의 타입으로 여러 종류의 객체를 처리할 수 있다면 if ~ else로 복잡하게 처리할 로직이 아주 간단하게 처리되는 것을 볼 수 있습니다. 이런 것이 바로 상속이라는 다형성

의 가장 큰 위력이라고 할 수 있습니다. 다만, 다형성에 대한 자세한 설명은 인터페이스를 설명할 때까지 조금 뒤로 미루도록 하겠습니다.

3.3 다른 사람들이 반드시 오버라이드하게 만드는 방법

만일 비행기를 통해서 부산에 간다면 어떻게 구현될까요? 우선은 Transportation 클래스를 상속해서 만들어 주고, go ( ) 메소드를 오버라이드하면 됩니다. 하지만, 아직은 이런 단순한 오버라이드에는 몇 가지 문제가 있습니다.

- 만일 누군가 Transportation 클래스에서 객체를 생성해서 go( )를 호출하면 '알아서 갑니다'가 실행되어 버린다.
  - Transportation a = new Transportation( ); a.go( ); 를 실행하지 말라는 법은 없습니다.
- Airplane 같은 새로운 교통편을 만들어도 go( ) 메소드를 오버라이드하지 않아도 컴파일러의 문제도 없고 실행도 된다. 다만 "알아서 갑니다."로 출력된다.
  - 상속이라는 게 물려받는 개념이라 오버라이드하지 않으면 '알아서 갑니다'가 go( ) 메소드의 의미가 될 겁니다.

이런 문제를 내버려 두고, 새로운 교통편을 만드는 개발자에게 반드시 go( )를 오버라이드하게끔 주의를 줄 수도 있겠지만 보다 적극적으로 다른 개발자들이 오버라이드를 할 수 있게 하는 방법이 있습니다. 그것은 바로 추상 클래스를 이용하는 방법입니다.

4 추상 클래스(Abstract Class): 객체가 아니라 타입으로만 존재하고 추상 메소드를 가지는 존재

추상 클래스라는 것은 간단히 말해서 객체 생성은 안 되고 부모 클래스로만 존재하는 클래스를 의미합니다. 즉 변수의 타입이 될 수는 있지만, 객체 생성을 못 하는 겁니다.

> **추상 클래스**라는 것은 클래스의 용도가 객체 생성 용도로 사용하지 않는 경우에 사용합니다. 클래스를 객체 생성의 용도가 아니라면 변수의 타입으로 선언하는 용도와 상속의 본연의 기능으로 특정한 코드를 하위 클래스로 물려주는 기능을 활용할 때 사용합니다.

Transportation을 생각해 보시면 Transportation a = new Bus( ); 혹은 Transportation a = new Train( ); 이 될 수는 있지만, Transportation a = new Transportation( );이 될 가능성은 없습니다. 지금의 경우라면 Transportation 클래스는 단순히 타입이 되기 위해서만 존재하는데 이런 클래스를 추상 클래스라는 것으로 선언합니다. 즉 변수의 타입으로만 의미가 있고, 객체로 생성될 가능성이 없는 것은 클래스 선언 앞에 'abstract'라는 키워드를 붙여서 추상 클래스라는 것으로 만들어 줄 수 있습니다.

예제 추상 클래스가 된 Transportation 클래스

```
public abstract class Transportation {
      public void go(){
            System.out.println("알아서 갑니다.");
      }
}
```

추상 클래스에 대해서 제가 가장 적절한 예제로 생각하는 것은 라면입니다. 생각해보시면 '라면'이라는 이름을 정확히 가지는 상품은 없습니다. 하지만 '신라면 객체 하나는 라면','너구리 라면 객체 하나는 라면' 즉, 라면이라는 클래스에서 실체가 나오지는 않지만, 분명히 라면이라는 개념 자체는 존재합니다. 아니면 주차타워를 생각해 보시면 됩니다. 주차타워를 굳이 자료구조로 보자면 배열과 같습니다. 예를 들어 주차타워를 다음과 같이 선언한다고 생각해 봅시다. 'Car[ ] arr = new Car[10];' 여기서 주목할 사실은 실제로 Car[ ] 안에는 Car 클래스에서 나온 객체가 들어가는 것이 아니라(즉 자동차라는 구체적인 것이 들어가는 것이 아니라), 자동차 클래스를 상속받은 객체(특정 상표의 자동차)만 들어간다는 겁니다.

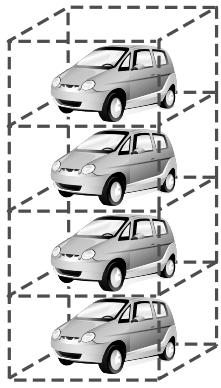

그림 15 주차타워

추상 클래스를 굳이 그림으로 표현하자면 점선으로 표현되는 그림이라고 생각하시면 좋겠습니다. 점선 안에 실제 클래스에서 나온 객체가 들어가기는 하지만 실제로 점선 자체가 객체로 만들어지는 것은 아니기 때문입니다.

그림 16 점선과 실체

■ **추상 클래스는 객체를 못 만들 뿐이지 상속의 기능은 그대로 있습니다.**

객체를 못 만든다니까 마치 아무런 의미가 없는 듯 생각하시는데 그렇지는 않습니다. 추상 클래스는 상속의 본연의 기능, 즉 속성과 동작을 그대로 자식 클래스에게 물려주는 기능은 살아 있습니다. 다만, 스스로 객체가 되지 못하는 것뿐입니다. 따라서 실제로 클래스를 만들 때 하위 클래스들에게 물려줄 만한 기능은 추상 클래스인 부모 클래스 안쪽에 선언되면 됩니다. 다시 한번 강조하지만, 상속의 최대의 기능은 물려주기입니다.

추상 클래스의 용도

- 객체를 생성하지는 않으나 변수 선언 용도로 사용됩니다.
- 상속의 고유 기능인 하위 클래스에 같은 데이터와 기능을 가지도록 물려주는 기능

■ **추상 클래스가 타입으로 선언된다면?**

추상 클래스의 주된 용도는 타입이라고 말씀드렸습니다. 그리고 컴파일러는 타입에 호출하려는 메소드가 존재하는지를 따진다고 설명했습니다. 따라서 추상 메소드 타입으로 어떤 변수를 선언하면 컴파일 시에는 해당 클래스에 정확한 메소드가 있는지 만이 유일한 관심사가 됩니다. 반대로 실행 시에는 추상 클래스를 물려받은 하위 클래스가 오버라이드한 메소드가 실행되는 방식으로 동작합니다.

추상 클래스는 추상 메소드라는 것을 가질 수 있습니다. 추상 메소드라는 것은 하위에서 '반드시 오버라이드하라'는 강제성을 가지는 메소드 입니다.

컴파일 시점에는 변수의 타입만을 보기 때문에 변수의 선언이 추상 클래스인 경우에는 추상 클래스가 가진 추상 메소드를 이용한 코드를 작성할 수 있습니다.

컴파일러가 변수의 타입 안에 선언된 메소드만 따진다는 것은 이미 말씀을 드렸던 내용입니다. 즉 컴파일러는 변수의 타입만을 보고 컴파일하지만, 실제로 실행될 때에는 같은 메소드가 있는 실제 객체가 동작하기 때문에 컴파일러는 실제 객체를 모르고 그냥 컴파일을 시켜줍니다. 이때 컴파일러를 속이기 위해서 메소드를 선언해주는 데 이 메소드를 만드는 것은 컴파일러를 속이고, 실제 실행되는 내용은 자식 클래스(실제 구현 클래스)에 만들어 주는 방식입니다.

■ 추상 메소드는 부모의 부채, 빚

추상 메소드는 추상 클래스가 부모로서 자식에게 남겨놓은 빚이나 부채, 혹은 부모님의 원수라고 생각하시면 됩니다. 즉 자식이 반드시 해야만 하는 의무이면서 책임이 되는 겁니다. 문법적으로 말씀드리자면 자식 클래스에서 반드시 오버라이드해야 하는 메소드를 의미합니다.

추상 메소드를 사용해서 얻는 이득은 간단한 겁니다. 우선은 모든 하위 클래스가 강제적으로 지정된 추상 메소드를 구현해야 하기 때문에 강제성이 생깁니다. 즉 정말 빈 껍데기라고 해도 반드시 자신에게 맞게끔 메소드를 정의해야 하는 것이 의무가 되니까요. 하지만, 제가 느끼는 추상 메소드의 실제 위력은 다른 곳에 있습니다. 실제 객체의 메소드를 몰라도 된다는 점입니다. 컴파일러는 부모 클래스를 보면서 go( )를 허락해 주지만, 실제로 실행되는 메소드는 Bus나 Train 클래스의 go( )가 된다는 얘기입니다.

■ 추상 메소드가 있으면 그 클래스는 추상 클래스가 되어야 합니다.

앞에서 추상 클래스라는 것은 객체를 만들 수 없고, 타입으로만 존재하는 경우에 사용한다고 말씀드렸습니다. 그런데 여기서 조금만 더 생각해 보시면 추상 메소드가 클래스 안에 있는데 그 클래스에서 객체를 생성하고 추상 메소드를 호출하면 어떻게 되나요?

만일 어떤 클래스에 추상 메소드가 있어서 실체가 없는 메소드가 존재하는데 그 상태로 객체를 생성해서 추상 메소드로 선언된 메소드를 만들어진 객체를 통해서 호출하게 되면 실행될 수 없게 됩니다. 따라서 하나라도 추상 메소드를 가지도록 만들어진 클래스는 추상 클래스로 선언해서 객체를 생성할 수 없게 합니다.

즉 추상 메소드가 있는 클래스가 실제 객체가 되면 난감한 상황이 발생합니다. 메소드의 실체가 없으므로 호출하면 정상적인 결과가 나올 수 없게 됩니다. 이런 문제를 방지하기 위해서 추상 메소드를 가지는 모든 클래스는 반드시 추상 클래스가 되어야만 합니다.

5 상속 활용 방법: 추상 클래스와 추상 메소드를 어떻게 쓰는가?

이제는 조금 더 상속이라는 것을 실제 문제를 해결할때 어떻게 적용해야 하는가를 고민해보도록 합니다. 사실 상속을 제대로 사용하기 위해서는 제약적인 조건이 있습니다. 상속을 정확하게 쓰려면 정확한 부분 집합인 동시에 속성이나 메소드의 상당 부분을 물려받아야만 적합해지는데, 적당한 예제가 별로 없군요. 여러분에게는 아직 그런 내용이 중요한 것은 아니고 상속을 어떤 기준으로 잡아서 사용하는지를 정확히 봐주시면 됩니다.

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

A라는 회사의 직원정보가 위와 같습니다. 구분을 보시면 정규직: R, 임시직: T, 일용직: A입니다. 당연히 보시면 일용직은 월급의 개념이 없고 일당의 개념만 있습니다. 여러분이 해야 할 일은 이런 데이터에서 매월 지급해주어야 할 금액을 계산할 수 있는 프로그램을 작성하는 일입니다. 다만, 여기서 나오지 않는 내용도 있습니다. 예를 들어 이 도표는 기준표이기 때문에 아르바이트를 하는 경우에는 근무 일수에 따라서 금액이 달라집니다. 이것을 객체지향이 아닌 방식으로 설계했다면 어떻게 될까요?

```
//if 파라미터로 들어온 직원이 정직원이면...
//else if 계약직이면
//아르바이트생이면
```

기존의 코딩 방식이라면 위와 같은 식의 계산이 if ~ else로 구분되는 코딩을 했을 겁니다. 맨 앞에서 회사의 코드에 if ~ else가 250개라는 말은 과장이 아닙니다. 하지만, 객체라면 객체가 알아서 월급을 계산하지 않을까요? 제가 이들의 데이터를 처리하는 프로그램을 의뢰받았다면 프로그램을 어떻게 설계하는지를 봐주시면 좋겠습니다. 전 다음과 같은 순서로 프로그램을 만들어볼 생각입니다.

1_ 우선은 해당 데이터 들을 몇 개의 클래스로 만들어야 할지 결정합니다.
2_ 상속 구조가 가능한지를 결정합니다.
3_ 일반 상속이 나은가? 추상 클래스를 쓰는 상속이 나은가 결정해야 합니다.
4_ 부모 클래스에 있어야 할 속성과 메소드를 조절합니다.
5_ 데이터를 활용하는 계산은 객체가 알아서 하도록 합니다.

그럼 단계별로 어떻게 설계를 잡아가는지 살펴보도록 하겠습니다.

5.1 데이터를 몇 개의 클래스로 만들어야 할까를 결정합니다.

가장 중요하고도 어려운 내용입니다. 프로그래밍을 하면서 제일 어려운 일이 바로 이런 판단입니다. 허접하게 대강 만들어서 결과를 만들어낼 것인가? 조금 더 복잡하게 프로그램을 설계해서 갈 것인가입니다. 엄밀히 그리고 정확히 말해서 이에 대해서는 절대로 정답이 있을 수 없습니다. 더 나은 설계가 있을 수는 있어도 항상 옳다는 절대적인 가치는 없다고 생각합니다.

클래스를 나누는 것도 그렇습니다. 보시면 그냥 도표 하나로 만들어질 수 있는 데이터 이므로 하나의 클래스로 설계하고 월급을 계산하는 부분은 if ~ else로 판단할 수 있다고 생각할 수도 있고, 계약에 따른 지급 금액의 형태나 보너스의 형태가 다르므로 별도의 클래스로 갈 수도 있습니다. 위의 경우에는 거의 많은 데이터가 일치하고, 약간의 if ~ else가 나옵니다. 따라서 하나의 클래스로 가도 무방할 듯합니다. 도표가 하나인 경우라면 클래스를 생각하시되 상속을 고려하면 꽤 높은 확률로 맞아 들어갑니다.

5.2 상속 구조 적용이 가능한지를 결정합니다.

데이터를 보고 클래스를 판단할 때 하나 혹은 몇 개의 클래스를 결정한 후에는 상속을 고려해보도록 합시다. 상속의 또 다른 이름이 '일반화(Generalization)'입니다. 즉 일반적인 내용은 모아서 하나의 부모 클래스로 만들어 줄 수 있다는 얘기입니다. 위의 문제를 보면 몇 가지 공통적인 속성이 보입니다.

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

- 사번
- 이름
- 고용형태(정규직, 계약직, 일용직)
- 받는 돈(연봉이든, 월급이든, 일당이든)

반면에 다른 데이터는 정규직 직원의 보너스 항목이 하나 보입니다. 상속을 활용하는 구조로 만들면 나중에 물려받는 것을 많이 가져갈 수 있도록 만들어 줄 수 있을 듯합니다. 그리고 또 하나의 중요한 데이터인 금액에서는 결국, 월급을 계산할 때 계약 형태에 따라 받는 돈을 다르게 계산하기 위해서 if ~ else를 이용하면 될 듯합니다.

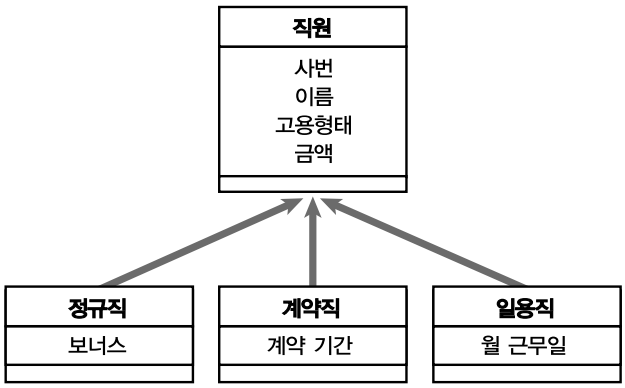

그림 17

여러 종류의 객체를 만들어야 하고 앞으로도 얼마든지 추가될 만한 클래스를 만들 것이라고 판단되면 여러 객체를 구성해 보고 공통된 데이터나 동작을 모아서 상속을 고려하는 것이 정석입니다.

상속을 UML(Unified Modeling Language, 통합 모델링 언어) 쪽에서 표기법으로 표시할 때의 이름은 '일반화(Generalization)'입니다. 즉 일반적이고 공통적인 것들을 모아서 추상화시킨다는 개념인데, 이런 의미에서 보면 상속을 올바르게 사용하는 것은 분석을 통해서 객체들의 데이터들을 모아서 하나로 다시 정의하는 형태이지, 처음부터 부모 클래스를 결정하는 스타일의 방식은 아닙니다. 어떤 개발자들은 처음부터 부모 클래스를 정의하고, 하위 클래스를 결정하는 것이 맞다고 생각하지만, 개인적으로는 여러 개의 클래스를 보면서 상속 구조로 변경하는 것이 더 정확한 결과를 만들어낸다고 생각합니다.

5.3 일반 상속이 나을까요? 추상 클래스를 쓰는 상속이 나을까요?

코딩을 물려받을 것이 많다고 생각하면 상속을 생각하고, 그다음으로 고려해야 하는 것은 그럼 일반 상속을 쓸 것인가? 아니면 추상 클래스를 이용하는 상속을 이용할 것인가의 문제입니다. 일반 상속을 쓰면 클래스의 숫자가 줄기 때문에 좀 편해 보이긴 합니다만, 제가 생각하기에는 다음 기준을 정확히 만족할 때에만 일반 상속을 쓰기를 권장하고 싶습니다.

■ 오버라이드가 필요 없을 때만 일반 상속을 쓰세요.

오버라이드할 필요가 전혀 없고, 새로운 속성과 메소드만 추가되는 경우에는 일반 상속을 그대로 쓰셔도 괜찮습니다. 하지만, 하위에서 반드시 오버라이드해야 하는 내용이 있다면 일반 상속보다는 추상 클래스의 형태로 가시는 것이 좋습니다. 그런 데이터의 특징은 나중에 새로운 데이터가 들어올 가능성이 많은 경우가 상당수입니다. 프로그램을 만들다 보면 if ~ else에 대한 부분이 필

요하다고 생각이 들면 이것은 오버라이드해야 한다는 '경고 신호'라고 보시면 됩니다. 지금의 경우에는 월급을 계산할 때 반드시 필요한 if ~ else가 추가되어야 할 것으로 보입니다. 이런 경우에는 일반 상속이 더 낫다고 판단하기는 어렵습니다.

■ **부모 클래스도 객체화될 때만 일반 상속을 씁니다.**

우선은 위의 기준을 정확히 만족했다면 그다음은 부모 클래스가 객체화되어도 되는가의 문제입니다. 부모 클래스도 객체로 갈 것이 확실하다면 그때는 일반 상속을 쓰셔도 됩니다. 위의 판단 기준으로 보면 결국 위의 문제는 매월 지급할 돈을 계산하는 부분에서 if ~ else의 로직이 들어가기 때문에 일반 상속을 포기하고 추상 클래스의 구조를 선택하는 것이 더 나아 보입니다. 거듭 말씀드립니다만 객체지향 프로그래밍 설계에 정답은 없습니다. 그저 좀 더 나은 설계가 있을 뿐입니다.

그럼 이제 상속 구조를 판단해 볼까요? 추상 클래스는 일반 상속의 역할도 충실하게 할 수 있기 때문에 다음과 같은 구조로 직원이라는 추상 클래스, 정규직, 임시직, 일용직(아르바이트)과 같이 4개의 클래스를 구성할 수 있을 것 같습니다.

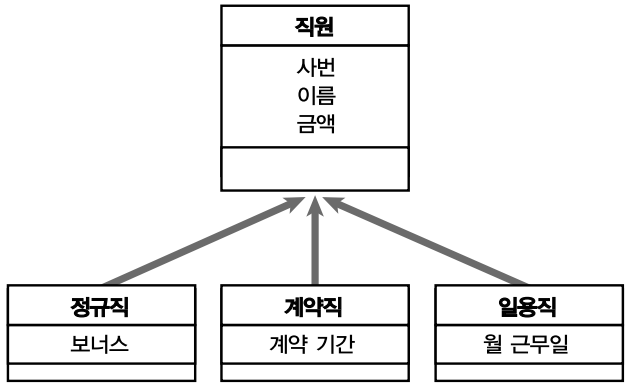

그림 18

사실 상속이라는 것 자체의 성격이 기존에 무언가 작업을 하고 나서 공통적인 속성이나 내용이 많으니 상속의 형태로 가자는 방식의 역방향으로 진행되는 경우가 많습니다. 전문용어로는 리팩터링(Refactoring)이라고 하는데, 유연해 질수록 클래스는 조금 더 늘어나는 경우가 많습니다.

5.4 공통적인 속성과 동작을 부모 클래스로 분리합니다.

이제 직원(Employee)이라는 클래스를 부모 클래스로 가기로 했다면 그 안에 넣어주어야 하는 속성과 메소드를 조절해봅니다. Employee는 다음과 같은 속성을 가지는군요.

- 사번 (empno)
- 이름 (name)
- 임금 (pay)

고용 형태는 조금 다릅니다. 고용 형태는 이미 클래스에 의해서 결정되어 버립니다. Employee는 다음과 같은 메소드를 만들어주면 될 것 같습니다.

- getMonthPay( ): 각 직원의 종류에 따라서 다르게 동작할 메소드, 즉 오버라이드가 필요한 메소드

이제 이것을 클래스로 만들면 아래와 같은 형태가 됩니다.

예제 | 추상 클래스로 선언된 Employee 클래스

```
public abstract class Employee {
      protected String empno;
      protected String name;
      protected int pay;
      public abstract double getMonthPay();
}
```

5.5 인스턴스 변수에 protected라는 것을 사용했다?

인스턴스 변수를 만들어 줄 때에는 private을 붙여서 만드는 것이 일반적이라고 말씀드렸습니다만, 위의 코드를 보면 protected라는 키워드를 사용하고 있습니다. private은 현재 코드가 있는 클래스의 바깥쪽에서는 변수나 메소드를 직접 접근할 수 없게 하는 키워드인 반면 protected는 상속 관계에서 변수나 메소드의 접근을 허용합니다. 유명한 객체지향 프로그래밍 전문가들은 거의 한결같이 "모든 인스턴스 변수는 private으로 만들어라."라고 가이드를 주고 있습니다만 상속을 배우는 입장이므로 protected를 사용해 보는 것도 괜찮을 듯합니다.

private은 같은 클래스 내에서만 메소드나 변수로의 접근이 가능하지만 protected는 상속 관계에서 하위 클래스가 별도의 제약 없이 부모에서 물려받은 변수의 선언이나 메소드에 접근할 수 있습니다.

■ **toString( )이나 생성자를 고려합시다.**

위의 구성을 보시면 전형적인 데이터 전용 객체라는 것이 보입니다. 따라서 이런 경우라면 toString( )을 만들어 주는 것이 좋습니다(toString( ) 메소드에 관해서는 조금 더 뒤에서 설명하겠습니다).

코드 | Employee 클래스에 추가

```
public String toString(){
      return empno+":"+name+":"+":"+pay;
}
```

■ **생성자를 고려합시다.**

모든 종류의 직원이 3가지 데이터(사번, 이름, 임금)를 다 물려받을 것이고, 3가지 데이터 모두 필수적인 데이터입니다. 이런 경우라면 생성자를 쓰는 것이 좋겠습니다.

예제 | Employee 클래스 안에 추가되는 생성자

```
public Employee(String empno, String name, int pay){
      this.empno = empno;
      this.name = name;
      this.pay = pay;
}
```

Employee 클래스는 상속만을 위해서 존재해야 하는 클래스, 즉 추상 클래스입니다. 실제 직원들은 정규직, 아르바이트, 계약직 클래스의 객체입니다. 추상 클래스라고 해도 일반 상속의 기능을 그대로 가진다는 사실을 잊지 마시길 바랍니다.

예제 | 추상 클래스로 완성된 Employee 클래스

```
public abstract class Employee {
	protected String empno;
	protected String name;
	protected int pay;

	public Employee(String empno, String name, int pay){
		this.empno = empno;
		this.name = name;
		this.pay = pay;
	}

	public abstract double getMonthPay();

	public String toString(){
		return empno+":"+name+":"+":"+pay ;
	}
}
```

5.6 중요한 계산은 객체가 알아서 하도록 합니다.

이제 실제 각각의 클래스를 구현해야 할 차례가 되었습니다. 우선은 정규 직원(RegularEmployee)을 만들어 보도록 합니다. 당연히 Employee를 상속해야 합니다. 이클립스를 이용해서 부모 클래스를 Employee로 지정했더니 아래와 같은 코드가 생깁니다.

5.6.1 부모의 생성자를 이용해서 코딩을 간단하게

상속에서는 부모 클래스가 생성자를 가지는 경우에는 조금 더 신경을 써줘야 합니다.

예제 | RegularEmployee는 자동으로 override가 되면서 생성자 때문에 컴파일 에러가 발생합니다.

```
public class RegularEmployee extends Employee {
	@Override
	public double getMonthPay() {
		// TODO Auto-generated method stub
		return 0;
	}
}
```

Employee 클래스를 추상 클래스로 선언해두고, getMonthPay( )가 추상 메소드가 되면 RegualarEmployee 클래스 내에는 자동으로 오버라이드가 된 코드가 생성되는데, 문제는 컴파일

러가 컴파일할 수 없다고 얘기한다는 것입니다. 이것을 부모 클래스가 생성자가 있기 때문입니다. 따라서 생성자를 추가해주어야만 온전히 컴파일됩니다.

예제 RegularEmployee: 이클립스의 자동완성 기능을 이용해서 만든 생성자

```
public class RegularEmployee extends Employee {
      public RegularEmployee(String empno, String name,
                              int pay, int bonus) {
            super(empno, name, pay);
      }

      @Override
      public double getMonthPay() {
            return 0;
      }
}
```

상속을 이용했더니만 변수에 대한 선언이나 생성자의 처리도 참 간단해졌습니다. 이제 여기서 RegularEmployee만의 데이터를 생각해봅시다. RegularEmployee는 매년 지정된 금액만큼의 수당(bouns)이 있습니다. 따라서 이 데이터를 인스턴스 변수에 추가해야 합니다. RegularEmployee는 Employee의 속성을 그대로 물려받을 수 있고, 고유하게 가지는 데이터가 바로 extra입니다. 따라서 저는 extra를 인스턴스 변수로 추가해주었습니다. 모든 직원은 수당 정보가 있다고 하면 생성자는 결국 empno(사원번호), name(이름), pay(임금), extra(수당) 이라는 4가지 정보를 생성자를 통해서 받는 것이 좋을 듯합니다. 수정된 생성자의 형태는 다음과 같습니다.

예제 bonus가 추가되고, 생성자를 수정해 반쯤 완성한 RegularEmployee

```
public class RegularEmployee extends Employee {
      private int bonus;

      public RegularEmployee(String empno, String name,
                                    int pay, int bonus) {
            super(empno, name, pay);
            this.bonus = bonus;
      }
~이하 생략~
```

수정된 생성자를 보면 super( )를 통해서 코드를 절약하고 필요한 데이터만 'this.bonus = bonus'로 추가하였습니다.

5.6.2 getMonthPay( )의 작성

getMonthPay( )는 Employee 클래스의 추상 메소드이므로 하위 클래스는 반드시 구현해야만 하는 메소드입니다. 우선 구현해야 하는 내용을 살펴보면 '(연봉/12)+(수당/12)'와 같이 됩니다. 상속이라는 것이 당연히 부모의 데이터를 물려받으니 그냥 '(pay/12)+(extra/12)'면 충분할 것 같습니다. 여기서 주의해서 봐야 하는 것은 부모 클래스의 인스턴스 변수를 하위 클래스에서도 마음대로 사용할 수 있는 접근 레벨인 protected라는 것으로 주었다는 겁니다. 완성된 RegularEmployee의 코드는 다음과 같습니다.

예제 | 완성된 RegularEmployee 클래스

```
public class RegularEmployee extends Employee {
      private int bonus;

      public RegularEmployee(String empno, String name,
                                    int pay, int bonus) {
            super(empno, name, pay);
            this.bonus = bonus;
      }

      @Override
      public double getMonthPay() {
            return (pay/(double)12)  + (bonus/(double)12);
      }
}
```

5.7 계약 직원과 아르바이트 직원의 코딩

계약 직원은 정규 직원과는 달리 수당이 없습니다. 또한 getMonthPay( )의 계산은 단순히 '연봉/12'면 충분합니다.

예제 | 계약 직원 TempEmployee 클래스

```
public class TempEmployee extends Employee{
      //고용기간
      private int hireYear;
```

```
	public TempEmployee(String empno, String name,
				int pay,   int hireYear) {
		super(empno, name, pay);
		this.hireYear = hireYear;
	}

	@Override
	public double getMonthPay() {
		// TODO Auto-generated method stub
		return pay/(double)12;
	}
}
```

아르바이트의 경우에는 일하는 날짜만큼의 돈을 받기 때문에 workDay라는 데이터를 하나 추가해 주도록 하겠습니다. 아르바이트의 경우에는 일당으로 계산되기 때문에 getMonthPay( )는 'pay×workDay'가 됩니다.

예제 아르바이트 PartTimeEmployee

```
public class PartTimeEmployee extends Employee{
	private int workDay;

	public PartTimeEmployee(String empno, String name,
				int pay,  int workDay) {
		super(empno, name, pay);
		this.workDay = workDay;
	}
	@Override
	public double getMonthPay() {
		return pay*workDay;
	}
}
```

현재 4개의 클래스를 작업했습니다만 사실 작업량은 Employee라는 추상 클래스일 경우가 가장 많은 내용이었고, 나머지는 간단한 추가적인 내용밖에 없습니다. 만일 유사한 다른 형태의 직원이 있다면 간단하게 상속을 해주고 필요한 변수만 만들어주면 됩니다.

5.8 객체를 만들어서 월급 계산해보기

추상 클래스를 이용했을 때 사용하는 데 있어서 편리함을 느껴봐야 합니다. EmployeeTest를 만들고 printPay( )를 static 메소드로 만들어 보았습니다.

예제

```
public class EmployeeTest {
      public static void printPay(Employee emp){
            System.out.println("======================");
            System.out.println(emp);
            System.out.println("----------------------");
            System.out.println(emp.getMonthPay());
            System.out.println("----------------------");
      }
}
```

소스를 보시면 Employee 클래스로만 코드를 작성하는 것이 보입니다. 파라미터는 Employee 타입의 객체이면 상관이 없습니다.

예제 | EmployeeTest에 추가한 main 메소드

```
public static void main(String[] args) {
      Employee[] emps = new Employee[4];
      emps[0] = new RegularEmployee("E001","홍길동",3000,500);
      emps[1] = new TempEmployee("E002","임꺽정",4000);
      emps[2] = new PartTimeEmployee("E003","황진이",5,10);
      emps[3] = new PartTimeEmployee("E004","어우동",10,7);

      for(int i = 0; i < emps.length ; i++){
            printPay(emps[i]);
      }
}
```

```
======================
E001:홍길동::3000
----------------------
291.6666666666667
----------------------
======================
E002:임꺽정::4000
```

```
----------------------
333.3333333333333
----------------------
======================
E003:황진이::5
----------------------
50.0
----------------------
======================
E004:어우동::10
----------------------
70.0
----------------------
```

5.9 상속, 다운 캐스팅, instanceof

상속을 통해서 여러분이 알았으면 하는 가장 중요한 내용은 "코딩의 양을 줄여준다."입니다. 객체지향 분석설계라는 분야가 있습니다, 이 분야에서 절차적인 언어와 객체지향 언어와의 차이점의 하나로 드는 것이 바로 조금 전에 작성된 예제에서처럼 조금 더 추상화된 레벨에서 코드를 작성할 수 있기 때문에 나중에 유지보수나 프로그램의 확장 시에 유연하다는 겁니다. 하지만, 살다가 보면 피치 못할 사정이라는 것도 생기게 마련입니다. 부모의 타입을 보고 메소드를 만들고, 부모의 타입을 보고 배열을 선언해서 나중에 자식 클래스가 많이 생겨도 작업할 내용이 없어져서 좋기는 하지만 반드시 자식 클래스를 알아야 하는 경우가 생깁니다.

앞에서 얘기한 중국집의 얘기를 해볼까 합니다. 아시겠지만 자장루는 본점과 서울분점이 있습니다. 본점은 자장면, 분점은 물려받은 자장과, 짬뽕을 팝니다. 만일 제가 어떤 사람 A를 만나서 "나 어제 자장루에 가서 짬뽕 먹었어."라고 말한다면 거짓말을 하는 건가요? 아닙니다. 제가 어제 서울자장루에 갔다면 짬뽕을 먹을 수 있으니까요.

그런데 문제는 이 얘기를 우연히 들은 B에게 있습니다. B는 자장루 본점을 알고 있기 때문에 제가 하는 말을 의심합니다. "이봐 자장루라는 가게는 자장면만 파는 가게야 그러니 당신 말은 틀렸어."라고 얘기를 했습니다. 그럼 B의 말은 틀린 말인가요? B의 얘기도 거짓말이 아니고, 제가 하는 말도 거짓말이 아닙니다. 그럼 둘 다 거짓말을 하지 않았다면 오해를 풀어야만 합니다. 제가 B에서 "어제 제가 간 자장루는 서울자장루입니다. 거기서는 짬뽕을 팔거든요."라고 말을 해주면 됩니다. 프로그래밍에서도 이렇게 말을 해줄 수 있습니다. 즉 지금의 상황을 정리해보면 다음과 같습니다.

• B는 부모 클래스의 객체라고 생각한다.
• 나는 타입은 부모 클래스이지만 객체는 자식 클래스의 객체라고 생각한다.

■ **컴파일러는 변수의 타입만 보고 메소드가 적합한지 판단합니다.**

위의 얘기에서 B라는 존재는 컴파일러를 의미합니다. 코드로 이해하시는 것도 괜찮을 겁니다.

```
ZaZangRu a = new SeoulZaZangRu();
```

변수의 타입이 '계열, 브랜드, 상표, 직원'과 같은 개념이기 때문에 위의 코드는 '자장루 브랜드인 서울자장루'로 해석하거나 '자장루의 분점 서울자장루'로 해석할 수 있습니다. 문제는 'a.makeZamBong( );' 시에 발생합니다. 컴파일러는 단순해서 a의 변수의 타입을 보고 a가 가리키는 객체가 ZaZangRu의 객체일 것으로 판단합니다. 즉 ZaZangRu 클래스 안에 선언된 변수나 메소드인지를 살펴본다는 뜻입니다. 따라서 이런 코드는 에러를 발생시킵니다.

예제 | 컴파일러는 변수의 타입만을 보고 판단합니다.

```
public class ZaZangRu {
	public void makeZaZang(){
		System.out.println("감자와 돼지고기, 자장을 볶습니다.");
		System.out.println("자장면을 만듭니다.");
	}
}
```

```
public class SeoulZaZangRu extends ZaZangRu {
	public void makeZamBong(){
		System.out.println("짬뽕을 만들 수 있습니다.");
	}

	public static void main(String[] args) {
		ZaZangRu sz = new SeoulZaZangRu();
		sz.makeZaZang();
		sz.makeZamBong(); //컴파일에러 발생
	}
}
```

■ **다운 캐스팅: 부모로 판단하지 마시고 줄여서 자식으로 봐주세요.**

위의 코드에는 에러가 있습니다. 하지만, 논리적인 내용만 놓고 판단해 보자면 잘못된 내용이 아닙니다. SeoulZaZangRu에 가서 짬뽕을 시킨 내용이니까요. 이런 경우는 컴파일러에 딱 집어서

얘기해 주어야 합니다. "이 봐 컴파일러 내가 간 곳은 ZaZangRu 중에서도 SeoulZaZangRu라는 곳이야. 그러니 오해하지 말고 컴파일을 잘 생각해봐."라고 얘기해 주어야 합니다. 이렇게 컴파일러가 바라보던 타입을 부모 타입이 아니라 자식 타입으로 보는 것을 다운 캐스팅(Down Casting)이라고 합니다. 그냥 캐스팅이라고 하기도 합니다. 캐스팅이라는 용어는 이미 넓은 범위의 변수를 작은 범위의 상자에 담을 때 나온 적이 있었습니다. 캐스팅이라는 것은 결국 넓은 범위를 좁은 범위로 좁힐 때 사용한다는 겁니다. 따라서 위의 자장루의 문제도 명확하게 컴파일러에 범위를 좁혀서 서울자장루라는 것을 얘기해주어야 합니다.

```
((SeoulZaZangRu)sz).makeZamBong( );
```

작은 순서는 먼저 sz의 타입을 명확하게 '(SeoulZaZangRu)sz'로 줄여주는 작업에서부터 시작합니다. 이후에 makeZamBong( )을 호출하는 방식입니다.

■ 객체 A instanceof B 클래스: 객체 A가 B 클래스의 객체인가?

위의 경우에는 아예 우리가 변수의 실제 객체가 SeoulZangRu라는 사실을 알고 있었기 때문에 가능합니다. 하지만, 만일 Mouse[ ]의 배열 안에는 여러 개의 마우스가 들어가 있는 상태일 텐데 모든 마우스의 scroll( ) 기능을 호출하면 안 될 겁니다. Mouse 배열 안에 아무 생각 없이 WheelMouse로 캐스팅해서 scroll( )을 시켜 보면 다음과 같이 제대로 실행이 안 됩니다.

예제 잘못된 캐스팅을 하는 경우

```
public class MouseTest4 {
      public static void main(String[] args) {
            Mouse[] arr = new Mouse[3];
            arr[0] = new Mouse();
            arr[1] = new WheelMouse();
            arr[2] = new OpticalMouse();

            for(int i = 0; i < arr.length; i++){
                  ((WheelMouse)arr[i]).scroll();
            }
      }
}
```

```
Exception in thread "main" java.lang.ClassCastException: Mouse cannot be cast to
WheelMouse
      at MouseTest4.main(MouseTest4.java:12)
```

앞의 코드는 실행하자마자 에러가 납니다. 배열의 가장 먼저 들어 있는 것이 Mouse의 객체의 리모컨이기 때문에 WheelMouse로 다운 캐스팅할 수 없거니와 만일 할 수 있다고 해도 Mouse의 객체에는 scroll( )이라는 메소드가 없기 때문입니다. 상속에서 어떤 클래스의 존재인지를 살펴보는 간단한 방법은 instanceof라는 연산자를 이용한 처리입니다.

예제

```
for(int i = 0; i < arr.length; i++){
      if(arr[i] instanceof WheelMouse){
            ((WheelMouse)arr[i]).scroll();
      }
}
```

instanceof 연산자는 리모컨이 해당 클래스의 타입인지를 따져보는 연산자입니다. 따라서 위의 코드는 배열에 들어 있는 리모컨이 WheelMouse 타입의 객체를 가리키고 있는지를 따져 봅니다.

■ instaceof 연산자의 단점

국내 Java 책들을 보면 상속, 캐스팅, instanceof 연산자를 중점적으로 설명하더군요. 물론 저도 처음에 그렇게 공부를 했는데 조금 더 개발을 하다 보니 이 연산자가 우리 생각처럼 완벽하지 않다는 생각이 들었습니다. 우선 아래 코드의 결과를 짐작해 보도록 합시다.

예제

```
Mouse[] arr = new Mouse[3];
arr[0] = new Mouse();
arr[1] = new WheelMouse();
arr[2] = new OpticalMouse();

for(int i = 0; i < arr.length; i++){
      if(arr[i] instanceof Mouse){
            System.out.println(i+"번째는 일반 마우스" );
      }else if(arr[i] instanceof WheelMouse){
            System.out.println(i+"번째는 휠 마우스" );
      }
}
```

루프를 돌면서 Mouse인지 WheelMouse인지를 구분하는 코드입니다. 그런데 이것을 그대로 실행하면 어떻게 될까요?

```
0번째는 일반 마우스
1번째는 일반 마우스
2번째는 일반 마우스
```

결과를 보시면 분명히 1번째와 2번째의 마우스는 WheelMouse와 OpticalMouse인데 제대로 검사를 하지 않고 있습니다. 또한, 더 큰 문제는 문법적으로나 실행할 때 아무런 제약 없이 실행될 수 있다는 겁니다. instanceof라는 연산자에 대해서 오해하지 않도록 하는 것이 가장 정답입니다. instanceof 연산자는 해당 변수를 원하는 타입으로 볼 수 있는지를 판단해주는 연산자입니다. 즉 'Mouse m = new WheelMouse( );'일 때 'if(m instanceof WheelMouse) { ... }'라는 연산자의 의미는 'WheelMouse m'이라는 선언이 가능한가에 대한 의미라는 겁니다. 따라서 위의 루프를 돌 때에는 Mouse arr[i]와 같은 의미입니다. 당연히 arr이라는 배열에는 마우스만 있으니 '일반 마우스'라는 결과가 나오게 됩니다. instanceof 대신에 사용할 대안은 잠시 후에 나오는 getClass( )라는 것으로 해결할 수 있습니다.

5.10 java.lang.Object: Java 객체의 숨겨진 부모

Java는 문법상 단일 상속만을 지원합니다. 그런데 이 단일 상속을 하다 보니 전혀 관계가 없는 객체를 하나의 타입으로 묶어서 처리할 경우에 적절한 방법이 없습니다. 즉 String 객체와 Mouse 객체, SeoulZazangRu 객체들을 하나의 배열로 처리한다고 생각해봅시다. 전혀 다른 타입의 객체들이기 때문에 하나의 배열로 만들어 낼 방법이 없습니다.

> **java.lang.Object 클래스**는 모든 클래스의 최상위 부모 클래스입니다. Java에서는 모든 객체는 자동으로 java.lang.Object 클래스를 상속합니다. 따라서 별도의 extends를 사용하지 않습니다.

프로그래밍을 하다 보면 자료구조라는 것을 배우게 됩니다. 자료구조란 데이터를 저장하는 방식에 대한 문제인데, Java와 같은 객체지향 프로그래밍에서는 객체 자체가 데이터를 의미합니다. 따라서 이런 문제를 처리하려면 범용적인 타입이 하나 필요합니다. java.lang.Object는 모든 클래스의 부모 클래스입니다. Java에서는 모든 클래스는 묵시적으로 Object라는 이름의 클래스를 상속(extends)하게 됩니다. 따라서 다음과 같은 선언이 가능합니다.

```
Object obj = new String("AAA");
Object obj = new Mouse( );
Object[ ] arr = { new String( ), new Mouse( ) };
```

Java로 객체를 만들어서 API 문서를 작성해보면 이것을 더 정확하게 알 수 있습니다.

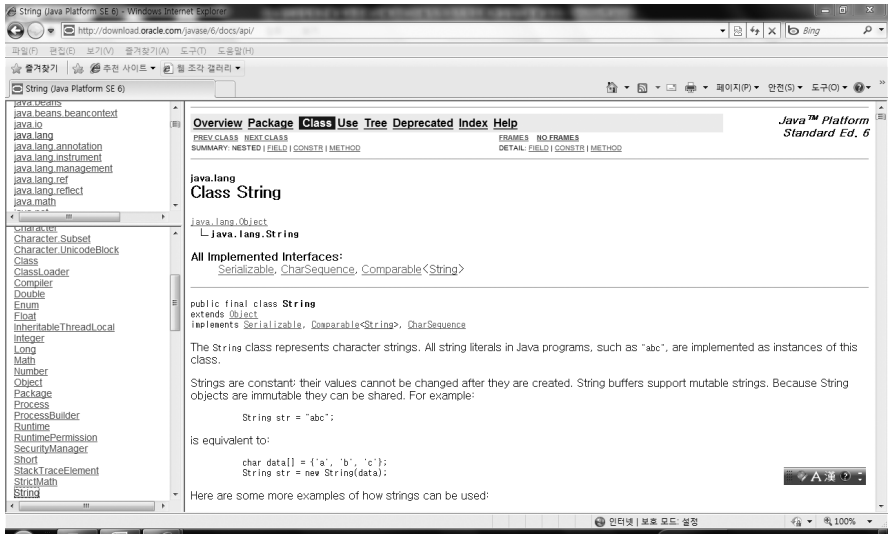

그림 19 API 문서

API를 보시면 최고 상속 구조가 나온 곳에서 가장 상위에 java.lang.Object가 존재하게 됩니다. 즉 여러분이 어떤 식으로 클래스를 만들든 간에 java.lang.Object 클래스를 상속합니다. 예를 들어 문자열인 String 클래스의 API 문서를 보면 다음과 같이 구성됩니다.

```
java.lang.Object
  └ java.lang.String
```

그림 20

때로는 어떤 클래스들은 조금 더 복잡한 상속 구조로 되어 있지만, 결론적으로 java.lang.Object 클래스를 상속해서 이루어집니다. 예를 들어 조금 뒤에 배우게 될 NullPointerException 클래스의 경우는 아래와 같이 복잡한 구조를 가지지만 최상위는 역시 java.lang.Object임을 알 수 있습니다.

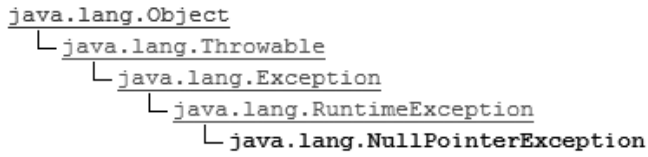

그림 21

5.11 모든 클래스는 java.lang.Object 클래스의 메소드를 상속받습니다.

모든 클래스의 부모 클래스이기 때문에 당연히 java.lang.Object가 가지는 메소드들을 상속되거나 하위 클래스에 반영되거나 필요한 경우에 오버라이드되어서 적용됩니다.

어떤 클래스의 객체이든 관계없이 진행해야 하는 작업이 있습니다. 가장 대표적인 것이 바로 가비지 컬렉션입니다. java.lang.Object 클래스는 모든 객체가 물려받아야 하는 공통적인 메소드를 하위로 물려줍니다. 물론 이 메소드들은 필요하다면 오버라이드하는데 가장 빈번하게 오버라이드하는 메소드는 equals( ) 와 hashCode( ), toString( ) 메소드입니다.

protected Object	clone( ) 이 객체의 카피를 작성해, 돌려줍니다.
boolean	equals(Object obj) 이 객체와 '등가'가 되는 객체가 있을지 어떨지를 나타냅니다.
protected void	finalize( ) 이 객체에의 참조는 이제 없으면 가비지 컬렉션에 의해 판단되었을 때 가비지 컬렉터에 의해 불려 갑니다.
Class 〈?〉	getClass( ) 이 Object의 실행 시 클래스를 돌려줍니다.
int	hashCode( ) 객체의 해시 코드 값을 돌려줍니다.
void	notify( ) 이 객체의 모니터로 대기 중의 thread를 1 개 재개합니다.
void	notifyAll( ) 이 객체의 모니터로 대기 중의 모든 thread를 재개합니다.
String	toString( ) 객체의 캐릭터 라인 표현을 돌려줍니다.
void	wait( ) 다른 thread가 이 객체의 notify( ) 메소드 또는 notifyAll( ) 메소드를 호출할 때까지, 현재의 thread를 대기시킵니다.

void	**wait(long timeout)** 다른 thread가 이 객체의 notify( ) 메소드 또는 notifyAll( ) 메소드를 호출하는지, 지정된 시간이 경과할 때까지, 현재의 thread를 대기시킵니다.
void	**wait(long timeout, int nanos)** 다른 thread가 이 객체의 notify( ) 메소드 또는 notifyAll( ) 메소드를 호출하는지, 다른 thread가 현재의 thread에 인터럽트를 걸거나 지정된 양의 실시간이 지날 때까지 현재의 thread를 대기시킵니다.

표 메소드의 개요

■ toString( )의 비밀을 풀어봅니다.

여러분이 어떤 클래스에서 객체를 만들고 나서 System.out.println( ) 같은 구문을 이용해서 출력하고 나면 '클래스 이름@16진수'로 표현되는 값이 출력되는 것을 보신 적이 있습니다. 반면에 그 클래스에 'pbulic String toString( ){ }' 메소드를 작성해 주면 출력되는 내용이 달라지는 것을 보신 적이 있을 겁니다.

예제 | toString( ) 메소드는 System.out.println( )에서 사용됩니다.

```
public class SampleObj {
      public String toString(){
            return "AAA";
      }

      public static void main(String[] args) {
            SampleObj obj = new SampleObj();
            System.out.println(obj);
      }
}
```

```
AAA
```

지금까지 배운 내용을 보자면 변수의 타입이 어떤 클래스인지가 중요하지 않습니다. 중요한 것은 변수를 이용할 때 실제 움직이는 객체가 무엇인지가 더 중요하다는 겁니다. 위의 코드를 이렇게 해석하시면 됩니다.

- toString( )은 원래 java.lang.Object 클래스에 있었던 것이다.
- 모든 클래스는 java.lang.Object를 상속한다. 그러므로 별도의 toString( )을 오버라이드하지 않으면 java.lang.Object 클래스의 toString( )을 그대로 사용하는 것이 된다.

- 'System.out.println(obj);' 코드는 실제로는 객체의 toString( ) 메소드를 호출하는 것이다.
- toString( ) 메소드를 오버라이드하게 되면 어차피 실제 객체는 SampleObj 클래스의 객체이므로 실제 객체의 toString( ) 메소드가 호출된다.

그동안 클래스를 만들고 나면 toString( )을 만들곤 했는데 그 비밀이 바로 java.lang.Object 클래스입니다. 즉 java.lang.Object의 toString( )을 오버라이드한 메소드였던 겁니다. java.lang.Object의 toString( )은 다음과 같은 몇 가지 규칙이 있습니다.

- 객체가 null일 때는 null을 반환한다.
- 기본적으로 [클래스 이름 + '@' + 16진수]로 구성된 Hashcode 값을 반환한다.

toString( )은 여러분이 주로 디버깅을 하거나 객체를 확인하기 위해서 많이 사용하기 때문에 내용을 깊이 있게 아실 필요는 없지만, 가능하면 클래스를 만들어주면 toString( )을 오버라이드하는 습관을 가지는 것이 좋습니다.

오버라이드라는 것은 결국 실행되는 것이 실제 객체의 메소드이기 때문에 toString( )을 오버라이드하게 되면 실행 시점에 진짜 호출되는 메소드는 실제 객체의 toString( )이 되면서 우리가 오버라이드한 메소드가 실행되는 원리입니다.

■ equals( )와 '==' 연산자

문자열에서 가장 강조한 내용이 바로 equals( )와 '==' 연산자입니다. String 클래스는 사실은 java.lang.Object의 equals( ) 메소드를 오버라이드해서 자신이 내용에 맞게 만들어진 메소드입니다. 원래의 java.lang.Object의 equals( ) 연산자는 '=='과 유사합니다. 즉 비교하는 두 변수에 담겨 있는 리모컨이 같은 객체를 의미할 때에만 true라는 결과가 나오게 되어 있습니다.

예제 equals( )는 원래 '=='와 동일했습니다.

```
public class Ball {
      private double radius;

      public Ball(double radius){
            this.radius = radius;
      }
```

```
	public String toString(){
		return "이 공의 반지름은"+radius;
	}

	public static void main(String[] args) {
		Ball a = new Ball(30);
		Ball b = new Ball(30);
		System.out.println(a == b);
		System.out.println(a.equals(b));
	}
}
```

```
false
false
```

결과를 보시면 둘 다 false입니다. 위의 코드 '=='은 두 변수안에 있는 리모컨이 같은 객체를 가리키는 것인지를 따져보는 연산자입니다. 물론 위의 예제에서는 아예 객체가 다르니 같은 결과가 나올 수 없습니다. 다음의 코드는 equals( )를 이용해 봅니다. 지금 현재 오버라이드가 되지 않았기 때문에 '=='과 동일하다고 생각하시면 됩니다.

■ **두 객체를 논리적으로 비교할 때는 equals( )를 오버라이드**

위의 두 개의 Ball 객체는 분명히 다른 객체이지만, 사실상 반지름이 같다면 같다고 논리적으로 판단하고 싶은 경우가 발생할 수 있습니다.

> equals( ) 메소드는 실제 객체의 물리적인 비교가 아닌 논리적인 비교를 합니다. 즉 '==' 연산의 경우 같은 메모리를 가리키는지를 단순히 비교한다면, equals( ) 메소드는 상황에 맞게 이러면 같다고 판단하도록 한다는 겁니다.

문자열의 equals( )를 기억해보시면 됩니다. 사실상 객체는 다르지만, 문자열 안의 내용물이 같아서 equals( )의 결과가 true가 나왔습니다. equals( )를 오버라이드해야 할 때가 이처럼 물리적으로는 분명히 다른 객체이지만 같은 객체라고 판단하고 싶은 경우에 사용합니다.

예제 | Ball 클래스에 오버라이드한 equals( ) 메소드

```
public boolean equals(Object obj){
```

```
        return this.radius == ((Ball)obj).radius;
}
```

위의 equals는 Java.lang.Object 클래스의 메소드를 상속한 메소드입니다. 코드의 내용은 같은지를 계산할 때 다른 Ball 객체의 반지름이 같은지를 가지고 판단하겠다는 겁니다. 즉 두 개의 객체가 전혀 다르다고 해도 개발자가 중요하게 생각하는 것은 반지름의 값이 같은가라는 겁니다. Ball 클래스의 equals( )를 위와 같이 오버라이드해준 후에 결과를 보면 이전과는 다른 결과를 볼 수 있습니다.

```
false ← 물리적으로 다르다.
true ← 논리적으로는 같다고 볼 수 있다.
```

나중에 자료구조 관련 API를 공부할 때 보시면 equals( ) 메소드의 위력을 아시게 됩니다.

■ 절대로 오버라이드하지 말아야 하는 finalize( )

finalize( )는 가비지 컬렉션이 일어나서 현재의 객체가 사라지게 될 때 하는 작업입니다. 그런데 이것을 오버라이드해서 사용하게 되면 성능상에도 치명적인 문제가 발생하고, 시스템이 비정상적인 경우에는 아예 동작을 보장할 수도 없습니다. 따라서 SCJP와 같은 자격증 시험문제에나 나오는 메소드라고 생각하시면 되고, 이 메소드는 절대로 오버라이드하지 않도록 합니다.

■ Class를 반환해 주는 getClass( )

모든 객체는 반드시 어떠한 클래스가 있기 마련입니다. getClass( )라는 것은 어떤 객체가 있을 때 그 객체가 어떤 클래스에서 나왔는지를 정확하게 알려줍니다.

코드 정확한 객체의 클래스를 알려준다.

```
Mouse[] arr = new Mouse[3];
arr[0] = new Mouse();
arr[1] = new WheelMouse();
arr[2] = new OpticalMouse();
for(int i = 0; i < arr.length; i++){
        System.out.println(arr[i].getClass());
}
```

```
class Mouse
class WheelMouse
class OpticalMouse
```

위의 코드를 보시면 오히려 instanceof보다 더 정확하게 특정 객체의 클래스의 정보를 가져옵니다.

코드 | getClass()를 instanceof 대신 사용하는 경우

```
Mouse[] arr = new Mouse[3];
arr[0] = new Mouse();
arr[1] = new WheelMouse();
arr[2] = new OpticalMouse();
for(int i = 0; i < arr.length; i++){
      //System.out.println(arr[i].getClass());
      if(arr[i].getClass().equals(Mouse.class)){
            System.out.println("Mouse");
      }else if(arr[i].getClass().equals(WheelMouse.class)){
            System.out.println("WheelMouse");
      }else if(arr[i].getClass().equals(OpticalMouse.class)){
            System.out.println("OpticalMouse");
      }
}
```

```
Mouse
WheelMouse
OpticalMouse
```

이번에는 getClass( )를 이용해서 instanceof의 대체 효과를 내 보았습니다. 오히려 instanceof보다 더 명확한 결과가 나오는 것이 보입니다.

5.12 변하는 것을 막는 final 키워드

상속을 공부하면서 마지막으로 알아두면 좋은 내용은 final이라는 키워드입니다. final 키워드를 한마디로 얘기하자면 고정한다는 의미입니다. 즉 어떤 곳에 final 키워드가 쓰이는 곳에서는 그 대상이 더 이상은 변경될 수 없다는 의미로 생각해야 합니다. final 키워드는 세 가지 용도로 사용되고, 그 각각의 용도만 정확히 알고 있으면 충분합니다.

■ **메소드 선언 시에 쓰이는 final**

클래스 안에 메소드를 작성할 때 final이라는 단어를 넣어 줄 수 있습니다. 이것은 자식 클래스가 현재의 메소드를 수정할 수 없도록 합니다. 즉 오버라이드해서 스스로 재정의하는 작업을 막습니다. 프로그램에서 부모 클래스가 물려주는 메소드 중에서 가장 중요한 메소드들은 final로 선언해서 사용합니다.

```
public final void doA( ) { .. }
```

final 키워드는 추상 메소드에는 사용할 수 있을까요? 잠깐만 생각해 보시면 추상 메소드의 목적 자체가 하위 클래스에서의 오버라이드를 목표로 해서 나왔다는 사실을 잊으시면 안 됩니다.

■ **클래스 선언 시에 쓰이는 final**

final 키워드는 클래스 선언 시에 사용될 수 있습니다. 이때의 final 키워드의 의미는 아주 간단합니다. "나를 더는 상속하지 못한다."라는 의미입니다. 그렇다면, 왜 상속을 더는 못 하게 할까요? 상속의 최대의 이익을 포기하는 것이니까요. 상속을 못 하게 되면 우선 당연한 얘기지만 자식 클래스를 만들어낼 수 없게 됩니다. 이렇게까지 해서 얻을 수 있는 최대의 이익은 우선은 더 이상의 수정을 불가능하게 만든다는 데에 있습니다. 즉 클래스에 선언된 모든 메소드의 오버라이드가 금지됩니다. 메소드가 10개이건, 100개이건 간에 아예 상속 자체가 불가능하기 때문에 오버라이드를 할 수 없게 됩니다. 또한, final로 선언된 클래스는 타입을 제한하는 역할도 할 수 있습니다. 즉 Mouse를 선언했는데 Mouse가 상속을 금지하도록 선언되었다면 이제 프로그램에서 Mouse[ ] 안으로 들어갈 수 있는 객체 종류는 오직 Mouse 클래스에서 생산되는 객체만을 사용할 수 있게 됩니다. 마지막으로 얻는 이익은 실행 시점에 어떠한 객체의 클래스가 final인 경우에는 더는 하위 클래스를 검색하지 않기 때문에 속도 면에서 미세하게 더 빠른 작동이 가능해집니다.

■ **값을 변경할 수 없는 변수 선언 시에 final**

final 키워드의 마지막 용도는 final이라는 키워드를 변수에 이용해서 변수의 값이 변경될 수 없게 하는 역할을 합니다. 기본 자료형과 객체 자료형의 경우 모두 살펴보면 좋습니다.

예제

```
public class FinalEx {

	public static void main(String[] args) {

		//final 변수는 대상을 변경할 수 없다.
		final int value = 10;
		//컴파일 에러
		value = 20;

	}
}
```

위의 코드는 기본 자료형의 변수가 고정되는 모습입니다. final은 가리키는 대상 불변이 아니라 변수의 내용 불변입니다. 객체의 경우는 어떤지 보도록 합니다.

예제

```
public class FinalEx {

	public static void main(String[] args) {

		final String[] arr = new String[3];

		arr[0] = "aaa";
		arr[1] = "bbb";
		arr[2] = "ccc";

	}
}
```

객체의 경우라면 변수 상자 안에는 리모컨이 들어가기 때문에 변경될 수 없는 것은 변수 상자 안의 리모컨의 대상이 됩니다. 즉, 위의 경우에는 arr 상자 안의 리모컨은 String[ ]을 가리키기 때문에 다른 배열의 리모컨을 넣어줄 수 없게 됩니다.

예제

```
final String[] arr = new String[3];

arr[0] = "aaa";
arr[1] = "bbb";
arr[2] = "ccc";

//다른 배열의 리모컨을 넣어줄 수 없다.
//컴파일 에러
arr = new String[5];
```

6 상속을 쓸까 말까? 상속을 사용하는 체크포인트

상속에 대해서 기존의 개발자나 입문자들은 상당히 너그럽다는 생각을 합니다. 사실 이런 기조가 된 것은 기존의 서적들의 영향이 큽니다. 상속이라는 개념에 대해서 자세히 설명해주다 보니 마치 상속을 많이 써야 할 것 같고, 상속이 만병통치약처럼 느껴지게 되는 상황이 옵니다. 그래서 이번에는 상속을 조금은 비관적인 눈으로 바라볼까 합니다.

6.1 상속을 쓰세요: '반드시 ~의 일종'이 성립할 때만 쓰세요.

상속을 써서 얻을 수 있는 최대의 이익은 역시나 기존의 코드를 그대로 물려받는다는 겁니다. 즉 기존의 코드를 물려받았는데도 불구하고, 더 많은 신경을 써야 한다면 사실 상속이라는 것은 의미가 없는 겁니다. 의미가 없다기보다는 잘못 사용되고 있다는 겁니다. 그래서 기존의 객체지향 프로그래밍 주의자들은 상속을 가능하면 정확하게 'is a kind of'의 관계가 성립할 때만 쓰기를 권장하고 있습니다. 그 외의 상황은 '상속보다는 조합(Composition)'을 이용하기를 권장합니다. 우선은 상속을 오용 혹은 남용하는 예제를 간단하게 회원 관리 시스템으로 살펴볼까 합니다.

> **"우리 회사에는 일반회원, 우대회원, 관리자로 분류됩니다. 일반회원은 커뮤니티에 가입을 하고 자유로이 이용할 수 있습니다. 이용 시마다 마일리지가 쌓여서 어느 수준이 되면 직접 커뮤니티를 개설할 수 있습니다. 자신이 개설한 커뮤니티에 대해서는 마음대로 운영할 수 있어 다른 사람의 글을 삭제하는 것도 가능합니다. 마지막으로 관리자는 이 모든 권한을 다 가지고 있습니다."**

이 문장을 가지고 프로그램을 설계하라고 하면 일반적인 개발자들이 "상속을 이용하면 되겠군. 일반회원, 우대회원, 관리자 이렇게 뼈대를 잡아주고 기능을 추가하면 되겠네."라고 많이 생각을 하더군요. 그러나 조금 문제가 있습니다.

■ **kind of 위배: 회원은 동시에 일반회원일 수도, 우대회원일 수도 있습니다.**

조금만 생각해 보면 어떤 회원은 다른 커뮤니티에서는 일반회원으로 활동하고 있고, 어떤 커뮤니티에서는 우대회원으로 커뮤니티를 운영할 수 있습니다. 즉 회원 한 사람당 여러 가지 역할을 가지게 됩니다. 이 회원은 자신이 개설한 커뮤니티에서는 마음대로 글을 삭제할 수 있지만, 자신이 참여만 하는 커뮤니티인 경우에는 그냥 마일리지가 높은 회원에 불과합니다. 이렇게 한 객체가 다양한 역할로 등장해야 할 경우에는 상속을 사용하면 복잡해집니다.

■ **안타까운 상속의 설계: 회사별 관리 시스템**

제가 본 모 회사의 권한 시스템에 대해 얘기해드릴까 합니다. 이 회사의 경우 고객사별로 직원이 담당하는 구조로 되어 있습니다. 그러니 각 직원은 어떤 고객사에 대해서는 모든 권한을 가지고 있지만, 어떤 고객사에 대해서는 제한된 기능만을 가지고 있었습니다. 우선 전체적인 기능의 목록은 단순합니다.

- 등록
- 수정
- 조회
- 삭제
- 검색

자꾸 담당자들 사이에서 사고가 생기자 이제는 등록, 조회, 검색만 하는 일반 담당자와 삭제, 수정 권한이 있는 관리 담당자로 시스템을 세분화시켰습니다. 그랬더니 처음에는 문제가 없었는데 직원 중에 임시직으로 일하거나, 외부에서 직접 자기네 회사의 상황을 보기 위한 고객사의 직원들이 조회 기능만이라도 달라는 얘기를 하게 됩니다. 뭐 어쩔 수 없이 새로운 사용자를 만들어서 처리해야 했습니다. 이제 시스템은 점점 커지고, 각 사람은 점점 권한이 세분화되어 갑니다. 누군가는 검색과 조회만, 누군가는 검색, 조회, 수정, 삭제가 필요합니다. 또 누군가는 등록만 필요할 겁니다. 이 경우에는 그림을 그리면 아주 복잡한 경우의 수가 나오게 됩니다.

이런 경우에 상속을 쓰게 되면 아주 골치 아파집니다. 상속이란 부모가 가지는 90% 이상을 그대로 물려받는 것이기 때문입니다. 따라서 이런 경우는 각 사용자에게 권한의 목록을 가지게 하고 그 권한을 넣어주는 형태의 설계가 더 나은 형태입니다. 이런 것을 조합(Composition)이라고 합니다. 일반적으로 상속보다는 조합이 더 나은 경우가 훨씬 많습니다.

6.2 단순 기능을 하나로 모으려고 상속을 쓰지는 않습니다.

상속을 잘못 사용하는 또 다른 경우 중의 하나는 유틸리티성 기능의 클래스를 상속하는 것입니다. 예를 들어 우리 회사의 회계 공식을 하나의 클래스로 모아서 이 클래스만 이용하면 될 수 있게 만들어 주는 습관은 좋은 습관입니다. 하지만, 그 클래스를 상속해서 하위에서 마음대로 오버라이드하게 되면 어떻게 될까요? 오버라이드하게 되면 기존의 로직을 재정의하기 때문에 필요한 경우가 아니라면 가능하면 오버라이드는 금지하는 것이 좋습니다. 오버라이드를 금지하는 데 있어서 가장 좋은 방법은 클래스 자체를 final 클래스로 선언해버리는 겁니다. 즉 이것은 상속을 금지하는 방법입니다.

혹 중요한 것만 오버라이드를 금지하면 되지 않을까 생각하시는 분들도 있을 겁니다. 그런데 메소드가 많은 경우에는 원하지 않는 곳에서 부모로부터 물려받은 데이터를 손상시키는 경우가 발생할 수 있습니다. 이것을 '깨지기 쉬운 상속'이라고 합니다만 이해를 돕고자 예를 하나만 들어 보도록 하겠습니다. "자장루에서 배달을 할 때 일회용 그릇을 이용했습니다. 굳이 회수 필요가 없으니 배달 후에 그릇을 회수하는 메소드도 없습니다. 하지만, 서울자장루에서는 배달을 할 때 일회용 용기를 사용할 수 없는 법령에 걸려서 가게 내에 있는 그릇을 사용합니다."

앞의 상황을 보면 배달이라는 기능을 오버라이드를 하지만 자식 쪽에서는 다른 데이터를 같이 수정하게 됩니다. 오버라이드는 정확한 로직을 대체해주어야 하는데 위의 경우에는 배달이라는 메소드가 오버라이드되면서 다른 데이터에게도 영향을 주게 됩니다. 따라서 오버라이드라는 작업 자체가 부모와는 완전히 다르게 동작할 수 있다는 가능성을 배제하지는 못한다는 겁니다.

6.3 하위 클래스에서 오버라이드를 많이 할수록 위험한 신호입니다.

상속이라는 것을 정확하게 쓰고 있는지, 아니면 잘못 사용하고 있는지 알 수 없는 경우도 종종 있습니다. 이럴 때는 상속을 이용하면서 새로운 코드의 작성량이 점점 줄어드는지, 늘어나는지로 판단해주면 됩니다.

- 만들어 두고 나니 if ~ else를 이용하는 다운 캐스팅을 열심히 해주어야 한다면 무언가 잘 못된 것으로 판단해야 합니다. 이런 코드는 여러분을 편리하게 할 수 없습니다. 추상 클래스처럼 객체의 타입에 상관없이 동작하게 해야 합니다.

- 억지로 끼워 맞추어 상속을 이용하지는 마세요. 삼각형은 밑변과 높이, 원은 반지름과 같이 약간의 융통성이 있을 수는 있지만, 부모 클래스에는 인스턴스 변수가 많고, 자식에서는 그 일부만 사용하는 방식은 상속으로 적당하지 않습니다.
- 항상 조합이 가능하다면 상속을 포기하는 것이 좋습니다. 다양한 경우의 수로 조합이 가능한 경우라면 상속을 쓰는 것보다는 조합을 이용하는 방식의 프로그래밍을 선택하는 것이 정답입니다.