CHAPTER 07

객체지향이라는 패러다임

객체지향은 기존 방식의 불편함을 해결하기 위해 등장한 방식입니다.
객체지향 프로그래밍이라는 것이 무엇을 해결하기 위해 나온 것인지 이해하도록 합시다.

앞에서 배운 변수나 제어문, 배열의 내용은 여러분이 거의 모든 프로그래밍을 공부할 때 먼저 접하는 내용입니다. 즉 그만큼 기본이 되는 내용이라고 할 수 있습니다. 이제는 조금 더 어려운 프로그램으로 가기 위한 선배들의 노하우를 배워야 하는 단계라고 할 수 있습니다. 이 단계를 통해서 앞으로 객체지향 프로그래밍을 배우면서 알아야 하는 내용에 대해서 좀 더 깊이 생각해보는 계기를 마련했으면 합니다.

사실 이번 절은 다음부터 시작하고 싶은 객체지향 프로그래밍으로 바로 진행하기 전에 살짝 소개를 먼저 하는 것이 좋다고 생각했기 때문에 마련한 부분입니다. 실제로 많은 Java나 객체지향 언어를 다루는 책들을 보면 마치 '잃어버린 고리'처럼 갑작스럽게 클래스와 객체에 대한 소개가 나오는 것 같습니다. 그 때문에 약간은 억지스럽게 객체지향 프로그램을 소개하는 듯한 느낌을 강하게 받았던 것도 사실입니다. 거창하게 프로그래밍 패러다임이 어떻게 변화되었는지를 설명하는 것보다는 좀 더 현실적으로 사람들의 생각이 어떤 식으로 발전했는가를 살펴보는 것이 배울 때 더 도움이 될 듯합니다.

1 1에서 100까지의 합을 구하는 프로그램을 만든다면?

객체지향 프로그래밍을 연습하는 데 있어서 가장 좋은 것은 여러분이 이미 만들어 본 프로그램을 다양한 형태로 분석해서 다시 객체지향의 원리에 맞게 재구성해보는 것입니다. 우선은 프로그램이 어떻게 실행되는지를 정확히 알고 있으면 가장 중요한 마음의 안정을 얻게 되고, 그 이후에 프로그램을 객체지향으로 변경해보는 연습을 통해서 자신만의 노하우를 쌓는 것이 좋습니다.

객체지향 프로그래밍을 무조건 외우지 않도록 노력해야 합니다. 객체지향 프로그래밍이 현재까지 인기를 얻게 된 이유를 이해하는 것이 더 중요합니다. C 언어와 같은 절차지향적인 언어와 객체지향 언어의 차이를 이해해야 합니다.

1.1 실행 가능한 프로그램에서 시작하세요.

초급자들이 보통 프로그래밍하면서 가장 중요하게 생각하는 것은 바로 프로그램이란 우선은 돌아가야만 한다는 겁니다. 설명을 위해서 1에서 100까지의 합을 계산한 프로그램을 보도록 하겠습니다.

예제 | 1에서 100까지의 합을 구하는 프로그램

```
public class SumEx {
      public static void main(String[] args) {
            int start = 0;
            int end = 100;
            int sum = 0;

            for(int i = start; i <= end ; i++){
                  sum = sum + i;
            }//end for
      }//end main
}
```

코드를 보면 아무런 문제도 없어 보이고 실제로 동작도 잘됩니다.

1.2 정말 잘 만들어진 프로그램이라면 나중에 변경 없이 다시 사용할 수 있을 겁니다.

조금 더 나은 프로그램과 그렇지 못한 프로그램의 차이는 뭘까요? 좋은 프로그램은 시간이 지나도 계속 쓸모 있는 프로그램이 좋은 프로그램이 아닐까요? 지금 당장 기능이 돌아가도록 만들면 본인이 혼자 사용하는 데에는 문제가 없지만, 다른 사람들이 프로그램을 이용하기는 힘들게 됩니다. 좀 더 실력이 있는 프로그래머라면 다른 사람이 지금 만든 기능을 다시는 개발하지 않도록 하는 겁니다. 그 자체가 완벽해서 더 이상은 손대지 않고 사용하는 겁니다. "좋은 디자인이란 더할 것이 없는 것이 아니라 더는 뺄 것이 없는 것이다."라는 말처럼 잘 만들어진 프로그램은 다른 사람들이 같은 기능을 다시 개발하지 않도록 심플하고 완전한 하나의 단위를 만드는 것을 의미합니다.

프로그래밍에서의 영원한 화두는 "어떻게 하면 한 번 개발한 코드를 다시 재사용할 수 있게 하는가?"입니다. 객체지향이나 함수와 같은 많은 대안이 그러한 고민의 해결책으로 제시된 방법들이라고 할 수 있습니다.

1.2.1 Copy & Paste를 이용한 재사용

이미 개발한 소스나 프로그램을 다시 개발하지 않는 가장 간단한 방법은 아마도 Copy & Paste를 하는 방법입니다. 복사하는 방법은 사실 현실적으로 시간이 없을 때 쓸 수 있는 최고의 방법이기도 합니다. 하지만, 이 방법에는 몇 가지 치명적인 문제가 있습니다.

- 만일 복사해 둔 곳에 이미 동일한 이름의 변수가 선언되어 있다면?
  - 기존에 있는 변수들 때문에 프로그램의 충돌이 발생할 수 있습니다. 만일 나중에 다시 복사해서 붙여 넣기를 하면 또다시 변수명이 충돌하게 됩니다.
- 기존에 작성된 코드를 복사해서 작업했는데 원래의 코드를 수정해야 한다면 일은 더 커집니다.
  - 기존의 로직을 100곳에다가 적용해 두었다면 100곳을 다 수정해야 하니까요.

기존 코드를 복사해서 사용하지 마세요. 언젠가는 터지는 시한폭탄과 같습니다. 한번 변경이 일어나면 뇌관을 건드리듯 사방에서 문제를 일으킵니다.

여러분이 Java를 조금 더 오랜 시간 공부하게 된다면 DRY(Don't Repeat Yourself)라는 말을 들어볼 기회가 있을 겁니다. 이 말이 뜻하는 복사하는 방법이 끼치는 해악에 대해서 공부하실 일이 있을 겁니다만 우선 여기서는 복사해서 붙여 넣는 행위가 결과적으로 문제를 해결해주지 않는다는 사실을 기억해야 합니다.

1.2.2 그럼 재사용을 하려면 어떻게 하는 게 좋을까?

기존의 코드를 복사하는 방법은 만병통치약이 아닙니다. 복사하는 방법은 중복된 코드를 여러 번 만들어내고, 이 코드들은 나중에 문제를 수정할 때 참으로 복잡한 문제를 발생시킵니다. 그렇다면, 이제는 현실적으로 어떤 코드를 만들어내야만 할까요? 어떤 코드를 작성해두고 이것을 필요할 때 재사용하는 좋은 방법이 없을까요? 아마도 여러분보다 앞서서 프로그래밍을 했던 사람들도 이 문제를 고민하지 않았을까요?

2 함수의 출현: 로직이나 기능을 하나의 { }으로 묶어서 한 덩어리로 만드는 방식

사람들은 자신들이 만든 소스를 보면서 '여기서부터 여기까지는 나중에 다시 써먹으면 좋겠는데' 라고 생각하기 시작합니다. 그리고 그것을 문법에서는 '{}'를 이용해서 경계선을 만들어내는 겁니다(이것을 블록으로 묶는다고 표현합니다). 이렇게 구분된 블록을 알아보기 쉽게 별도로 소스에서 분리해 내고, 이름을 붙여서 알아보기 쉽게 하려는 시도가 바로 함수(Function)라는 것의 시작이 됩니다. 함수란 어떤 로직을 묶어서 입력된 데이터를 처리해서 결과를 만들어내는 로직 처리의 단위를 의미합니다.

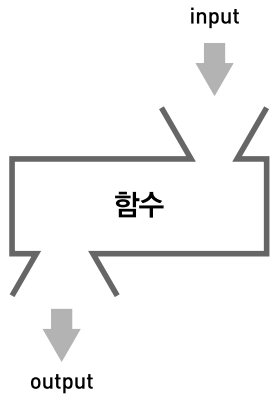


그림 1 함수라는 로직 처리의 덩어리

함수라는 존재 덕분에 많은 로직을 재사용하는 데 성공하게 됩니다. 나중에 변경이 일어나면 함수만 수정하는 방식으로 개발하여 개발자들의 생산성에 이바지하게 됩니다.

2.1 함수란 로직이나 기능을 구성하는 소스를 하나의 { }로 묶는 것

사람들이 {}로 묶는 것은 몇 라인 혹은 몇백 라인의 소스 코드입니다. 이렇게 블록으로 만들어지는 소스 코드는 당연한 얘기지만 소스 코드의 여러 라인들이 묶여서 하나의 의미를 구성할 때 하게 됩니다. 주로 이런 것들은 어떤 복잡한 계산을 하는 로직을 의미하거나 변환이나 계산을 하는 기능 혹은 전체의 로직의 흐름에서의 하나의 단계 등을 의미하게 됩니다. 이 책에서는 이런 단위로 로직/기능/동작이라는 단어를 같이 혼용해서 사용하도록 하겠습니다.

2.2 Java에서는 함수라는 표현 대신에 Method(메소드)라고 합니다.

Java에서는 함수라는 표현 대신 메소드라는 용어를 사용합니다. 이 표현이 가지는 의미에 대해서는 뒤에 설명하기로 하고, 우선은 어떻게 해서 이런 코드를 재사용할 수 있는지 그 요령을 먼저 알아보도록 합니다.

2.3 더하는 로직을 하나의 메소드로 만들어보기

실제로 주어진 숫자들 사이의 모든 합을 만드는 로직을 분리해서 어떻게 구성할 것인지를 생각해 봅니다. 보통 수학에서 사용하는 것처럼 만들려고 하는 메소드를 f(x)라고 하겠습니다.

1_ 변수 start, end를 준비한다.
2_ 합을 구하는 메소드 f(x)에 start와 end 값을 던져 준다.
3_ f(x)는 연산 작업을 하고 나서 결과물을 반환해준다.
4_ f(x)를 호출한 곳에서는 결과물을 특정 변수를 통해서 저장한다.

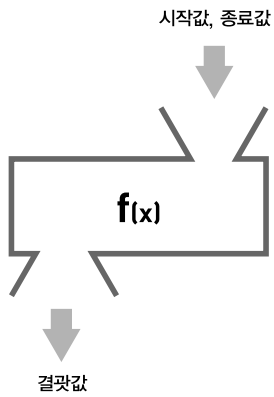


그림 2

여러분이 f(x)라는 함수에 시작값과 종료값을 넣어주기만 하면 결과 데이터가 나온다고 가정하면 100곳에서 합을 구해야 하는 로직이 필요할 때 굳이 작성할 필요 없이 만들어진 f(x)를 이용해 주기만 하면 되기 때문에 같은 코드를 중복으로 사용하지는 않을 겁니다.

■ **아마도 이름은 좀 구분하기 쉽게 지어야 하지 않을까요?**

f(x)의 이름은 말 그대로 이 메소드가 어떤 기능을 하는지를 의미해야 합니다. 그래야만 나중에 다시 사용할 때 이름만 보고 찾아낼 수 있습니다. 여러 명의 사람이 각자 자신의 마음대로 이름을 붙이는 것도 문제가 있을 수 있습니다. 따라서 실제 프로젝트에서는 '명명 규칙(Naming Conventions)'이라는 것을 미리 정해서 사용합니다.

> Naming Conventions는 프로그램을 작성할 때 사용하는 '명명 규칙'입니다. 주로 어떤 로직을 처리하는 경우에는 영어의 '동사'로 표현하는 것이 일반적입니다. 메소드를 만들 때 로직에 걸맞는 이름을 부여해 주면 다른 사람들이 메소드를 이용할 때 많은 도움이 됩니다. 실무에서는 회사마다 고유한 명명 규칙을 가진 경우가 많습니다. 그리고 이런 규칙들이 정리된 내용은 '코딩 가이드'라는 문서로 개발자들이 참고하게 합니다.

■ **동일한 기능을 실행할 때마다 다른 데이터를 입력할 수 있습니다.**

f(x)에는 1과 100이라는 값이 들어올 수도 있고, 10, 200이라는 값이 들어올 수도 있습니다. 그러니 이 데이터는 마음대로 넣어 줄 수 있어야 할 겁니다. 다만, 숫자를 두 개 받는다든가, 문자 하나와 숫자 하나를 받는다든가 등의 사항은 미리 순서나 타입을 맞춰 주어야만 할 겁니다. 이처럼 어떤 메소드를 실행할 때 필요한 데이터를 파라미터 혹은 인자, 인수라고 하는데 조금 더 뒤에서 자세히 공부하도록 합니다.

■ **결과 데이터 역시 필요할 수 있습니다.**

결과 데이터 역시 로직이나 기능을 수행하고 나면 어떤 형태로 데이터가 나올지 정해져 있어야만 할 겁니다. 그래야만 호출한 쪽에서 결과가 어떤 타입의 데이터로 나올지를 판단하고 대처할 수 있습니다. 위의 내용을 종합해 보면 함수라는 것은 3가지 형태로 구성되는 것을 알 수 있습니다.

- 메소드(함수)는 반드시 자신을 구분할 수 있는 이름을 가진다.
- 메소드(함수)는 동작하는 데 있어서 필요한 데이터를 가질 수 있다.
- 메소드(함수)는 필요하다면 결과를 반환해주는 기능도 있어야 한다.

3 메소드: Java에서 어떤 로직이나 기능을 만들어내는 방법

이제 실제로 Java에서 이런 기능을 어떻게 만들어내는지 알아보도록 합니다. 사실 Java에서는 이런 기능을 만들 때 가장 많이 쓰이는 방식은 다음과 같습니다.

- static이라는 키워드와 함께 작성되는 진짜 함수와 비슷한 방법
- 클래스와 객체를 이용해서 동작하는 객체를 만들어내는 방법

이 두 가지 방법이 공존하지만 실제로 여러분이 우선 알아야 하는 방식은 클래스와 객체를 이용하는 방식입니다. static에 대한 얘기는 객체와 클래스에 대한 설명이 끝난 다음에 별도의 장에서 추가할 생각입니다.

> 함수라는 로직을 단위로 묶는 기능은 객체지향 주의자들에게도 매력적으로 보였습니다. 이 사람들은 함수를 조금 더 수정해서 조금은 차별화된 형태로 함수를 사용하기로 하고 이름을 메소드(Method)로 부릅니다.

3.1 백 번 보느니 한 번 만들어보는 게 낫습니다.

그럼 이제 Java에서 어떤 식으로 만들어 내는지 알아보고 개념적인 설명을 하는 것이 좋을 듯합니다.

3.1.1 우선은 가장 먼저 할 일은 어떤 것을 만들지를 결정하는 겁니다.

여러분이 가장 먼저 할 일은 어떤 것을 만들려고 하는지를 결정하는 겁니다. 숫자들의 더하기를 하는지, 빼기를 하는지를 결정하는 것이 시작입니다. 어떤 것을 만들지 결정했다면 그런 기능이나 로직을 어떤 이름으로 알아볼 수 있게 할 것인가를 결정해주어야 합니다. 예를 들어 "숫자들의 합을 구한다."와 같은 구문을 add라는 기능(로직)으로 부르기 쉽게 만드는 겁니다. 이런 이름을 붙일 때는 간단히 몇 가지 규칙을 가집니다.

- 동작이나 기능(혹은 로직)에 관련된 일이기 때문에 동사로 시작한다.
- 한 단어로 표현하기 때문에 복합적으로 구성할 때(즉 단어를 두 개 이상 결합할 때에는) 구분하기 좋게 단어와 단어 사이에서 연결되는 단어는 대문자로 시작한다.

예제 | 우선 프로그램을 만들기 위해서 클래스를 하나 만들어줍니다.

```
public class SumMachine {

      public static void main(String[] args) {

      }
}
```

위의 코드는 아무런 내용도 없는 그저 'main 메소드'만 존재한다고 부릅니다. 이제 여러분은 이 코드를 조금씩 수정해서 작성해 나갈 것입니다.

3.1.2 메소드는 우선 'public void 메소드 이름( ) { }'으로 작성해두고 생각합니다.

'public static void main ...'과 객체지향 프로그래밍을 혼동하지 않는 작업이 바로 객체지향 프로그래밍을 배우는 가장 첫 단계입니다. main 메소드는 객체지향 프로그래밍과 무관합니다.

Java에서 메소드 선언의 가장 첫 번째 단계는 그냥 앞에서 만든 프로그램에 'public void 메소드 이름( ) { }'라는 코드를 넣어 주는 작업에서 시작합니다. 예를 들어 지금 만드는 것이 어떤 숫자들의 합을 구하는 기능이라면 'makeSum'이라는 이름을 이용하도록 하면 될 것 같습니다.

예제 | 메소드 이름을 이용해서 public void 이름( ) { } 코드를 프로그램에 추가합니다.

```
public class SumMachine {

      public void makeSum(){

      }


      public static void main(String[] args) {

      }
}
```

아직은 앞의 예제를 실행해도 아무런 동작도 하지 않을 겁니다. 다만, 위치를 유심히 보도록 합니다(SumMachine이라는 하나의 큰 블록과 makeSum이라는 이름을 가진 블록, main이라는 이름을 가진 블록으로 구분해볼 수 있습니다). 그리고 main 메소드와는 단독으로 { }를 이용해서 분리되어 있다는 점도 알아야 합니다. 메소드의 해석은 이름에서 항상 먼저 시작합니다. 'makeSum이라는 메소드는 ~'이라고 해석을 시작해주면 되는 겁니다.

■ **public은 '누구나 사용할 수 있는'이라는 뜻입니다.**

public은 말 그대로 누구나 이 기능을 사용하게 했으면 할 때 사용하는 단어입니다. 간단히 말하자면 지금 만드는 makeSum이라는 기능은 누구나 사용하게 해주겠다는 뜻으로 해석하시면 됩니다.

■ **void라는 단어의 뜻은 메소드를 호출한 쪽에서 결과를 얻을 수 없다는 표현입니다.**

영어사전에서 'void'라는 단어의 뜻을 찾으면 '텅 빈'이라는 의미로 해석되는 것을 볼 수 있습니다. 즉 이 makeSum이라는 메소드를 실행하고 나면 결과를 호출한 쪽에서 받아 봤자 아무것도 없다는 뜻입니다. 이에 대한 자세한 얘기는 조금 뒤에 결괏값(리턴 타입)에 대한 얘기를 할 때까지 미루겠습니다.

■ **( ) 안에는 필요한 데이터를 넣어 줍니다.**

마지막으로 메소드의 ( ) 안쪽에는 이 기능을 실행하기 위해서 필요한 input이 무언지를 얘기해 줍니다. 예를 들어 필요한 데이터가 숫자인지, 숫자 하나인지, 둘인지와 같은 정보를 알려주기 위해서 사용합니다.

3.2 메소드를 만들었으면 무조건 실행되게 하세요.

메소드를 하나라도 만들어 놨다면 그다음에 할 일은 무조건 메소드가 실행되도록 하는 겁니다. 실행되도록 하기 위해서는 실행되는 모습을 볼 수 있게 코드를 약간은 수정해줄 필요가 있습니다.

> 코드의 작은 부분이라도 반드시 실행해서 결과를 확인하도록 합니다. 이를 통해서 작성된 코드의 문제점을 찾는데 이런 작업을 '디버깅(Debugging)'이라고 합니다.

코드 | 선언해 둔 메소드가 실행되는 모습을 볼 수 있게 수정해줍니다.

```
public void makeSum(){

      System.out.println("make sum.......");
}
```

makeSum 메소드의 { } 안에 System.out.println( )을 추가해서 위의 메소드가 실행되면 어떤 결과를 만들어 내게 수정해 두었습니다.

3.2.1 main 메소드를 수정해서 실행되도록 합니다.

앞에서 변수나 배열들의 예제를 만들 때 보셨듯이 main 메소드라는 곳에 작성하는 코드는 프로그램을 실행시키면 무조건 실행됩니다. 아직 클래스와 객체에 대한 개념을 배우지 않았기 때문에 우선은 코드에 나오는 모습을 보시고 실행되는 결과를 보도록 합니다.

예제 | 실행되는 SumMachine 코드

```
public class SumMachine {
      public void makeSum(){
            System.out.println("make sum.......");
      }

      public static void main(String[] args) {
            SumMachine m = new SumMachine();
            m.makeSum();
      }
}
```

```
make sum.......
```

위의 코드를 대소문자를 틀리지 않고 작성했다면 실행되는 모습을 보실 수 있습니다(Java 언어가 대소문자를 엄격하게 구분한다는 사실을 잊지 마시기 바랍니다).

3.2.2 기존 방식과 메소드 방식의 차이

여기까지의 모습을 보고 '뭐야? 이거 예전에 main 메소드에 코딩하는 것과 별다른 차이가 없잖아'라고 생각하시는 분들이 있을지도 모르겠습니다. 만일 이 글을 읽는 자신이 그런 생각을 하셨

다면 조금만 더 설명을 읽어주시기 바랍니다. 여러분이 보시기에는 기존에 main 메소드의 안쪽에서 직접 코딩하는 것보다 코드만 더 복잡해졌고 결과는 동일해 보입니다만 사실 지금의 코드는 완전히 새로운 방식의 접근입니다. 단언하건데 **이 코드야말로 여러분이 Java를 공부하는데 있어서 가장 중요한 코드**라고 할 수 있습니다. 우선 main 메소드 안의 코드를 보면서 차근차근 설명해보도록 하겠습니다.

```
SumMachine m = new SumMachine( );
```

아마도 가장 낯설어하실 만한 부분이라고 생각하는데 우선은 좀 쉽게 하고 갑니다. 간단히 말해서 makeSum( )이라는 기능을 보유하고 있는 실제 기계(존재)를 만들어낸다고 생각하시면 됩니다. 이 구문에서 주의하실 것은 여러분이 처음에 프로그램을 만들 때 작성한 SumMachine이라는 단어의 대소문자가 틀리면 절대로 실행되지 않는다는 겁니다.

■ **SumMachine m = new SumMachine( ); 자세히 보면 마치 변수 선언 같습니다.**

좀 더 자세히 코드를 보시면 앞에 무언가가 정의되어 있고 뒤에 선언되는 부분의 방식이 마치 변수를 선언할 때 사용한 것과 유사하다고 생각하시면 됩니다. 사실 이전에 앞에서 이런 방식의 변수를 선언해 사용한 적도 있는데 그때의 코드는 Scanner였습니다.

```
Scanner scanner = new Scanner(System.in);
```

'new Scanner(System.in);' 이라는 코드를 기억해 보시면 위의 코드와 몇 가지의 공통점이 있습니다.

- Scanner s, SumMachine m과 같이 변수의 선언에 앞글자가 대문자로 되어 있다는 점
- 'new Scanner(System.in);'과 'new SumMachine( );'처럼 변수에 값을 할당하는 부분에 new라는 키워드가 쓰였다는 점

아직 Scanner 혹은 SumMachine라는 기계(존재)를 생각하지는 마시고 지금은 우선 메소드라는 존재를 만들고 사용하는 데에만 집중하도록 합니다. 그리고 뒤에서 조금 더 자세히 살펴보도록 합니다. 지금은 이런 코드를 넣어주었다고 생각만 해 주시고 Scanner와의 유사성만을 눈으로 봐두도록 합니다.

■ m.makeSum( )은 마치 scanner.nextInt( )와 비슷하지 않나요?

마지막으로 작성된 코드는 m.makeSum( )입니다. 그런데 특이한 것이 이 코드 역시 예전에 Scanner를 사용할 때 쓰던 코드와 유사하다는 겁니다.

```
Scanner scanner = new Scanner(System.in);
int a = scanner.nextInt( );
```

생각해 보면 Scanner라는 장치를 하나 만든 후에 그 장치를 이용할 때 변수 scanner 뒤에 '.'을 이용했습니다. 아마도 Java 코드를 이해하는 데 가장 중요한 표현일 것입니다. 그리고 지금 'SumMachine m = new SumMachine( );' 이라는 기계(존재)를 사용할 때에는 m.makeSum( ); 처럼 '.'를 이용했다는 사실을 봐 두시기 바랍니다.

3.2.3 실행된다면 추가 코드를 넣어봅니다.

여러분이 위에 작성된 코드가 실행되었다면 조금 더 코드를 넣어서 실행되는지 확인해보도록 합니다.

예제 | 코드가 실행된다면 조금 더 코드를 넣어봅니다.

```
public class SumMachine {

	public void makeSum(){
		int start = 0;
		int end = 100;
		int sum = 0;

		for(int i = start; i <= end ; i++){
			sum = sum + i;
		}//end for

		System.out.println("시작값: " + start);
		System.out.println("종료값: " + end);
		System.out.println("총 합: "+ sum);
	}//end makeSum메소드


	public static void main(String[] args) {
		SumMachine m = new SumMachine();
```

```
            m.makeSum();
        }//end main

}
```

```
시작값: 0
종료값: 100
총 합: 5050
```

3.2.4 실행되는 방식이 완전히 다른 코드

맨 앞에서 작성한 코드와 지금의 코드가 결과를 보면 차이가 전혀 없어 보인다고 생각하실 겁니다. 그럼 이제는 차별성을 한번 자세히 보는 것도 좋겠습니다. 몇 가지 설명해야 하는 내용이 있지만 가장 대표적인 것은 다음과 같은 점입니다.

- 메소드를 한 번 만들면 나중에 그 코드를 다시 작성하지 않는다.
- 메소드가 데이터를 입력받을 수 있도록 해주면 매번 다른 데이터를 처리할 수 있다.

각 내용을 조금 더 자세히 보도록 하겠습니다.

3.3 메소드를 만들어 주면 재사용이 가능합니다.

여러분이 앞으로 어떤 로직을 작성하고 나서 그것을 메소드의 형태로 작성해야 하는 가장 큰 이유는 바로 앞서 말씀드렸던 것처럼 재사용을 목표로 하기 때문입니다. 예를 들어 여러분이 프로그램을 작성하다가 1에서 100까지의 합을 구하는 데이터가 필요하다면 이제는 매번 for 루프를 반복할 필요없이 makeSum()을 호출해주기만 하면 됩니다.

코드

```
public static void main(String[] args) {

        SumMachine m = new SumMachine();
        m.makeSum();
        m.makeSum();
        m.makeSum();
}
```

```
시작 값: 0
종료 값: 100
총 합: 5050
시작 값: 0
종료 값: 100
총 합: 5050
시작 값: 0
종료 값: 100
총 합: 5050
```

실행된 결과를 보니 단지 코드를 몇 줄 추가해서 반복적으로 호출했을 뿐인데 실제로는 for 루프가 여러 번 실행되어 결과를 만들어 내는 것이 보입니다. 물론 아직은 처리하는 데이터가 1과 100으로 고정되어 있기 때문에 재사용을 하는 효과는 미미합니다.

3.4 메소드의 파라미터(Parameter), 인자(Argument): 메소드가 필요한 데이터를 다른 값으로 받을 수 있다면 어떨까요?

메소드를 만들어서 한 번만 코드를 작성하고, 필요한 만큼 실행하는 방식을 취했지만 이런 메소드가 조금 더 다양한 상황에 맞게 사용되려면 매번 처리해야 하는 데이터가 다른 경우에도 사용할 수 있어야만 합니다. 이렇게 메소드를 실행할 때마다 조금 다른 데이터를 처리하기 위해서 어떤 데이터를 입력할 수 있게 하는데 이것을 메소드의 파라미터(Parameter)라고 합니다.

코드

```
SumMachine m = new SumMachine();
m.makeSum();
m.makeSum();
m.makeSum();
```

위의 코드를 다음과 같은 형태로 바꿀 수 있다면 어떨까요? 아직은 코드를 수정하지 말고 이런 모습으로 바꾸는 것을 상상해봅시다.

코드

```
SumMachine m = new SumMachine();
m.makeSum(1,100);
m.makeSum(100,1000);
m.makeSum(200,300);
```

예를 들어 처음에는 1에서 100까지의 합을, 두 번째는 100에서 1,000까지의 합을 계산하도록 하는 겁니다. 이런 식으로 바꿔 준다면 다양한 상황에서 계산을 해줄 수 있기 때문에 좀 더 쓸모가 있어 보입니다.

■ **메소드를 재사용하려면 매번 다른 값을 줄 수 있게 합시다.**

예를 들어 여러분의 휴대전화에는 "전화를 건다."라는 기능이 있습니다. 그런데 전화를 매번 같은 곳으로만 건다면 그런 휴대전화를 구매할 필요는 없을 겁니다. 지금의 makeSum( ) 메소드 역시 마찬가지입니다. 여러분이 로직을 잘라내서 메소드라는 것을 구성했다고 해도 그것이 완전히 똑같은 값만 만들어 낸다면 별다른 의미가 없을 겁니다. 그래서 우리는 메소드를 만들 때 들어오는 input 값에 다른 값을 줄 수 있도록 만들어 주는데 이것을 바로 메소드의 파라미터(Parameter) 혹은 인자(Argument)라고 합니다.

3.5 메소드의 파라미터를 정하는 방법

여러분이 만든 makeSum( )이라는 메소드의 재사용성을 조금 더 높이고자 파라미터를 받아들이게 결정했다면 이제는 그 파라미터로 무엇을 받을 수 있도록 해야 하는지 결정해야 하는 차례가 되었습니다. 이런 기준은 어떻게 잡는 것일까요?

3.5.1 메소드를 녹즙기라고 생각해 주세요.

농담 반 진담 반으로 말씀드리는 것이지만 어릴 때 산수를 공부할 때 녹즙기 비슷한 그림을 보신 기억이 나실지 모르겠습니다.

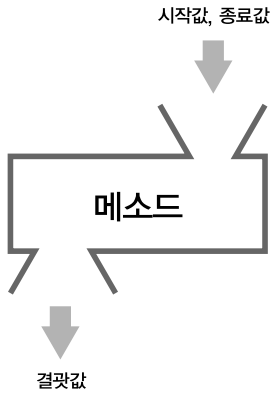


그림 3

녹즙기에는 당근을 넣으면 당근즙이 나오고, 사과를 넣으면 사과즙이 나오게 될 겁니다. 그렇다면, 녹즙기에 들어가는 재료를 파라미터(Parameter)라고 생각해보면 안 될까요? makeSum( )이라는 메소드를 녹즙기로 보시면 어떤 데이터를 넣어서 처리하면 좋을지도 좀 더 쉽게 눈에 보입니다. 즉 우리가 원하는 것은 시작값과 종료값을 넣어 줄 수 있으면 좋겠다는 겁니다.

3.5.2 메소드에 넣어주는 데이터의 종류를 결정합니다.

메소드를 실행할 때 필수적으로 필요한 데이터를 파라미터라고 한다면, 파라미터가 문자인지, 숫자인지를 명확하게 해서 정확하게 호출할 수 있도록 '파라미터의 타입(type)'을 지정하도록 합니다.

메소드에 어떤 데이터를 넣어 줄 수 있도록 하려면 가장 먼저 어떤 데이터를 넣을 수 있게 할 것인지를 결정해주어야 하겠지만, 그다음으로 결정할 사항은 그럼 대체 '어떤 데이터 타입의 자료를 넣어줄 수 있도록 하는가'입니다. 즉 녹즙기에 쇳덩어리를 넣고 돌리는 무식한 일은 하면 안 되며 녹즙기에는 채소나 과일만을 넣어 주어야 하는 것처럼 메소드에는 그 메소드가 처리했으면 하는 적합한 종류의 데이터 타입만 받아들이게 하는 겁니다.

코드 | 수정된 makeSum( ) 메소드: 소스를 수정하면 컴파일 에러가 발생하지만 잠시만 지켜봅니다.

```
public void makeSum(int startValue, int endValue){
```

안에 들어가는 int startValue, int endValue는 이 메소드에 데이터를 던지려면 반드시 int 타입의 데이터여야만 한다는 것을 의미한다고 보시면 됩니다. makeSum( ) 메소드를 위와 같은 형태로 작성하면 여러분의 눈에 가장 먼저 띄는 것은 에러가 난다는 사실일 겁니다만 잠시만 참고 남은 설명을 봐야 합니다.

3.5.3 도대체 startValue, endValue라는 변수는 어디서 나온 겁니까?

메소드의 파라미터는 남들이 호출할 때 식별이 가능하도록 이름을 붙여서 알아보기 쉽게 합니다. 다시 말하자면 메소드의 파라미터를 선언한다는 것은 '변수를 선언'하는 것과 같은 방식입니다.

메소드를 실행할 때 필요한 데이터는 고유한 이름을 부여해줍니다. 이것은 입력되는 데이터에게 이름을 부여해서 쉽게 사용할 수 있게 하는 방식인데 좀 더 쉬운 예를 들자면 우리가 식당에서 테이블 번호를 이용해서 점원들이 계산하는 것과 유사한 방식이라고 할 수 있습니다.

■ 메소드의 파라미터 이름은 음식점의 테이블 번호와 같습니다.

누군가 makeSum( )을 실행하는 데 20에서부터 200까지의 합을 구한다고 생각해봅시다. 그럼 메소드의 입장에서는 두 개의 데이터를 받아들여야 할 것이고, 이 작업을 할 때 자신만의 규칙대로 들어오는 데이터에게 이름을 붙여야만 처리하기 좋을 겁니다.

코드 | 수정된 makeSum( ) 메소드: 컴파일 에러가 발생하지만 잠시만 지켜봅니다.

```
public void makeSum(int startValue, int endValue){
```

위의 경우라면 입력받는 두 데이터 중에서 앞의 데이터를 startValue라는 변수를 이용하겠다는 뜻이고, 두 번째로 들어오는 데이터는 endValue라는 변수를 이용해서 담겠다는 뜻입니다. 따라서 메소드 내부에서는 첫 번째 들어온 데이터를 startValue라는 변수로 이용하고, 두 번째 들어오는 변수를 endValue라는 변수로 활용합니다. 마치 식당에서 여러분이 하는 주문을 기록할 때 점원들이 테이블 번호를 기록하는 것처럼 어떤 데이터를 변수로 처리합니다.

■ 파라미터는 사실 메소드에 던지는 값을 변수에 담는다는 겁니다.

메소드의 파라미터는 메소드에 들어오는 데이터를 구분하기 위해서 사용하지만, 문법적으로 보면 변수에 데이터를 담는 것과 마찬가지입니다. 우선은 코드를 수정해서 돌아갈 수 있게 한 뒤에 이어서 가도록 합니다.

예제 | 정상적인 결과는 아니지만 컴파일은 되는 SumMachine

```
public class SumMachine {
      public void makeSum(int startValue, int endValue){
            int start = 0;
            int end = 100;
            int sum = 0;

            for(int i = start; i <= end ; i++){
                  sum = sum + i;
```

```
            }//end for

            System.out.println("시작값: " + start);
            System.out.println("종료값: " + end);
            System.out.println("총 합: "+ sum);
      }

      public static void main(String[] args) {
            SumMachine m = new SumMachine();
            m.makeSum(1,100); // 수정 된 부분
            m.makeSum(20,200); // 수정 된 부분
            m.makeSum(30,300); // 수정 된 부분
      }
}
```

아직은 실행된 결과를 보지 말고 호출하는 인과관계만 보도록 합니다. 우선은 main 메소드에서 makeSum( )을 호출하는 코드에 정수 1, 100 같은 데이터가 들어가는 모습을 봐 주시면 됩니다. 이 코드가 실행될 때에는 다음과 같은 코드가 실행된다고 생각하시면 됩니다. 이 코드에서 가장 중요한 내용은 바로 데이터가 들어오는 모습이 메소드의 파라미터로 선언된 변수에 할당된다는 개념입니다. Java에서는 '=' 연산을 변수의 개념을 복사하는 방법을 사용하기 때문에 'int startValue = 200;'이라는 코드가 실행되는 것과 동일하게 실행되는 겁니다.

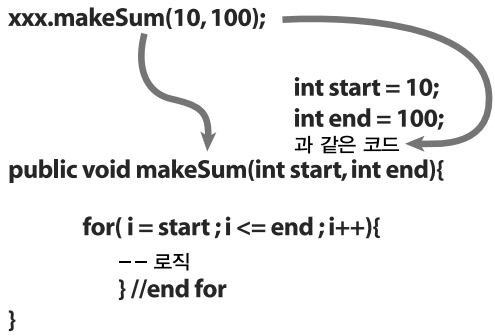


그림 1

■ **','는 메소드의 파라미터가 여러 개일 때 순서를 지정하기 위해서 사용됩니다.**

이제 마지막으로 파라미터를 사용할 때의 ','의 의미입니다. ','는 그 자체가 순서를 의미합니다. 예를 들어 입력되는 데이터가 3개인 경우라면 ','는 두 개가 필요하게 됩니다.

■ **메소드에 선언하는 파라미터와 for 루프의 i 값은 유사하지 않나요?**

메소드의 파라미터는 메소드의 { } 범위 안으로 한정됩니다. 따라서 파라미터의 영향력(범위)도 그 안으로 한정됩니다.

좀 뜬금없는 얘기처럼 들릴 수도 있겠지만 메소드의 선언과 for 루프의 선언 사이에서도 하나 배울 점이 있습니다.

```
public void makeSum(int startValue, int endValue){
```

```
for(int i = 0; i < 3 ; i++){
```

두 구문을 자세히 보면 몇 가지 공통점이 보입니다.

- 둘 다 {}를 이용해서 영역을 표시하고 있다는 점
- 둘 다 ( ) 안에서 변수를 사용하고 있다는 점

for 루프를 설명할 때 얘기했듯이 for 루프 안에 선언된 i라는 변수는 루프의 안쪽에서만 사용됩니다. 그렇다는 얘기는 결국 makeSum( ) 안에 선언된 startValue, endValue라는 변수 역시 메소드 안에서만 사용된다는 겁니다. Java 언어에서 '{ }'의 위력은 절대적입니다. 즉 메소드 선언 안에 사용된 statValue, endValue라는 변수는 결국에는 메소드의 { } 안에서만 효력을 가지는 변수라는 뜻입니다. 이것을 흔히 스코프(Scope)라고 하기도 하는데 변수의 범위라고 생각하시면 됩니다. 이에 대한 자세한 설명은 뒤쪽에서 지역 변수(Local Variables)라는 용어와 같이 다시 설명하도록 합니다.

3.5.4 이제 정상적으로 실행되게 소스를 수정해봅니다.

이제 정상적으로 실행될 수 있게 소스를 조금 더 수정해 보도록 합니다. 메소드에 들어오는 파라미터는 메소드 내에서는 유효한 데이터이므로 이 데이터를 기초로 해서 프로그램을 수정해주어야 합니다.

코드

```
public void makeSum(int startValue, int endValue){

      int start = startValue;
      int end = endValue;
```

원래의 코드를 수정해서 메소드 안에 선언된 변수 start와 end라는 변수에 파라미터로 들어온 데이터를 받도록 수정해 주었습니다. 완성된 소스는 다음과 같습니다.

예제 | 메소드를 온전하게 만든 SumMachine 코드

```
public class SumMachine {
      public void makeSum(int startValue, int endValue){
            int start = startValue;
            int end = endValue;
            int sum = 0;

            for(int i = start; i <= end ; i++){
                  sum = sum + i;
            }//end for

            System.out.println("시작값: " + start);
            System.out.println("종료값: " + end);
            System.out.println("총 합: "+ sum);
      }

      public static void main(String[] args) {
            SumMachine m = new SumMachine();
            m.makeSum(1,100);
            m.makeSum(20,200);
            m.makeSum(30,300);
      }
}
```

```
시작값: 1
종료값: 100
총 합: 5050
시작값: 20
종료값: 200
총 합: 19910
시작값: 30
종료값: 300
총 합: 44715
```

실행된 결과를 보면 메소드를 호출할 때 매번 다른 데이터를 넣어서 다른 결과물을 만들어낼 수 있게 된 것이 확인됩니다. 즉 메소드의 파라미터라는 것은 특정한 로직에 필요한 약간씩 다른 데이터의 값을 넣어서 조금 더 재사용을 높이기 위해서 사용한다는 겁니다.

3.6 메소드의 실행 결과를 예고해주는 리턴 타입(Return Type)

메소드를 실행하고 특정 타입의 결과를 반환해주겠다고 선언하는 것을 '리턴 타입'이라고 하고, 그 결과 데이터를 '리턴 값(Return Value)'라고 합니다.

- 리턴 타입은 메소드를 실행한 결과 데이터를 지정된 타입으로 반환해준다는 표시입니다.
- 리턴 타입이 'void'라는 선언은 결과 데이터로 반환할 것이 없다는 뜻입니다(반환하지 않겠다는 뜻).
- 결과 데이터는 매번 달라지므로 메소드의 선언에서는 어떤 유형의 데이터가 나오는지만을 리턴 타입으로 지정해줍니다.

이제 마지막으로 메소드의 실행 결과를 반환하는 작업에 대해서 생각해보도록 합니다.

3.6.1 메소드의 실행 결과를 왜 반환해줄 때가 있는가?

앞의 코드에서는 우리가 만들어진 결과를 System.out.println( )을 이용해서 메소드의 내부에서 화면에 출력하는 방식을 사용해주었습니다. 그런데 나오는 메시지를 영어로 해야 한다면 어떻게 해야 할까요? 이제 메소드를 호출하는 곳을 100곳에 적용해 두었다면 일이 좀 많아집니다. 결과를 한글로 출력하도록 했기 때문에 우리는 메소드에서 System.out.println( )을 이용하는 부분을 영어로 수정해주어야 합니다.

문제는 여기에 있습니다. 애써 만들어 준 메소드를 영어와 한글을 동시에 처리하지 못하는 것은 조금 아쉬움이 남는 일입니다. 사실 메소드야 결과를 제대로 연산해서 결과를 만들어내면 되는 것인데, 영어와 한글 때문에 메소드를 고쳐야만 하다니요. 그래서 사람들은 "메소드는 로직을 처리한 결과만을 호출한 쪽으로 다시 전달해주고, 반환된 결과물에 대한 처리는 호출한 쪽에서 알아서 하도록 분리해주자."라고 생각하게 됩니다.

메소드에 파라미터라는 장치를 통해서 매번 다른 상황에 대처하도록 했다면, 메소드의 리턴 타입이라는 것은 결과를 매번 다른 결과가 나오는 것을 처리하기 위한 장치입니다.

3.6.2 메소드의 결과를 반환해줄 때 리턴 타입을 'void'가 아닌 타입으로

메소드의 리턴값의 가장 중요한 판단 기준은 특정한 메소드를 실행한 후에 즉각적으로 그 메소드의 실행 결과를 알게 해주는 것이 좋은가라는 기준입니다. 예를 들어 자판기에서 커피를 마시려고 버튼을 눌렀는데 10분 후에 반응이 온다거나, 물건 수리를 맡겼더니 다음 주에나 찾아오는 것은 즉각적이 피드백이 아닌 경우라고 할 수 있습니다.

메소드를 통해서 어떤 로직이나 기능을 수행시켜놓고 결과를 바로 원하는 경우는 무척이나 흔한 일입니다. 이때 여러분이 즉각적으로 피드백해주기로 했다면 그 메소드에 반환되는 결과의 타입을 명시해주는 것이 좋습니다(리턴 타입). 예를 들어 지금 makeSum( )을 호출했을 때 한국어나 영어와 같은 메시지를 처리하고 싶다면 메소드는 실행된 결과만을 반환해주고 호출한 쪽에서 메시지를 한국어나 영어로 처리하는 것이 더 나은 선택일 것입니다.

코드 | makeSum( )을 호출하면 즉시 int 타입의 결과를 반환하도록 수정한 코드(컴파일 에러)

```
public int makeSum(int startValue, int endValue){
      int start = startValue;
      int end = endValue;
      int sum = 0;
      ...이하 생략
```

앞의 코드처럼 소스를 수정하면 실행 에러 메시지가 나옵니다. 우선은 변경된 메소드의 의미부터 정확히 알아보도록 합니다.

> **public void makeSum(int startValue, int endValue){**
> void로 선언되었을 때의 의미는 이 메소드는 실행해도 결과를 알려주지 않는다는 뜻입니다.

> **public int makeSum(int startValue, int endValue){**
> int로 선언되었다는 것은 메소드를 실행하면 반드시 int 타입의 데이터를 피드백해 주겠다는 의미입니다.

makeSum( ) 메소드에 int를 리턴 타입으로 명시해주었다는 뜻은 결과적으로 이 코드에서 반드시 int에 해당하는 데이터를 반환해주겠다는 것을 의미합니다. 소스를 수정하고 나서 컴파일이 되지 않는 것은 코드 내에서 결괏값을 반환해주는 부분이 없기 때문입니다.

3.7 return이라는 키워드를 이용해서 데이터를 반환합니다.

어떤 메소드가 리턴해주기로 했다면 반드시 메소드의 코드는 return이라는 코드를 작성한 부분이 실행되도록 작성되어야 합니다. 그렇지 않으면 컴파일이 되지 않습니다.

만일 어떤 메소드가 실행된 결과를 피드백해주기로 결정했다면 여러분이 반드시 사용해야 하는 키워드는 바로 'return'이라는 구문입니다. 예를 들어 makeSum( )이 정상적이라면 아마도 주어진 숫자들 사이의 모든 합을 구해서 만들어진 합을 반환해줄 것입니다. 따라서 정상적으로 반환될 수 있는 코드의 결과를 return이라는 키워드를 이용해서 반환해주어야 합니다.

코드 | 완전해진 makeSum( ) 메소드

```
public int makeSum(int startValue, int endValue){
      int start = startValue;
      int end = endValue;
      int sum = 0;

      for(int i = start; i <= end ; i++){
            sum = sum + i;
      }//end for
      return sum;
}
```

makeSum( ) 메소드가 int 타입의 결과를 반환해주기로 약속했으니 어떻게든 int 타입의 데이터를 반환해주어야 합니다. 코드를 보면 'int sum = 0;'이라는 변수를 하나 선언해두고 그 변수가 결과적으로 'return sum'이라는 코드를 통해서 실행되는 것을 볼 수 있습니다.

■ 사실 return이라는 키워드의 진짜 의미는 '실행을 여기까지만'이라는 뜻입니다.

사실 return이라는 키워드의 진짜 의미는 "현재 코드의 실행 제어권을 반환한다."라는 뜻입니다. 너무 어렵게 생각하지 마시고, 우선은 실행되면 여기까지만 실행된다는 의미로 이해하시면 됩니다. 간단한 코드와 결과를 보는 것이 더 나은 설명이 될 듯합니다.

예제 return이라는 키워드는 실행을 끝낼 때 사용합니다.

```
public class ReturnEx {

	public void test(int a){

		if(a == 10){
			System.out.println("a 는 10 ");
			return; // 아래쪽은 실행하지 않고 종료합니다.
		}

		if (a == 20){
			System.out.println("a 는 20 ");
			return;
		}

	}


	public static void main(String[] args) {

		ReturnEx ex = new ReturnEx();
		ex.test(10);
	}
}
```

```
a 는 10
```

코드를 보면 test( ) 라는 메소드에서 입력되는 값에 따라서 다른 결과를 만들어 내도록 작성되어 있습니다. 눈여겨보실 것은 메소드의 리턴 타입이 'void'라는 사실과 'return' 키워드가 실행된 이후의 코드는 실행되지 않는다는 겁니다. 본래 'return' 키워드의 의미는 이처럼 '여기까지만 실행하고 다시 돌아간다'는 의미였습니다.

■ **리턴한 결과에 대한 처리는 전적으로 받는 사람의 몫입니다.**

원래의 makeSum( ) 메소드를 보도록 합니다. 메소드를 실행해 주면 결과를 반환해 주도록 되어 있고, 정상적으로 결과도 만들어집니다. 다만, 메소드의 리턴 값에 대해서 여러분이 마지막으로 알아 두어야 하는 사실은 결과를 반환한 이후의 작업은 메소드를 호출한 쪽의 몫이라는 겁니다. 예를 들어 여러분이 편의점에서 900원짜리 콜라를 산 후에 1,000원을 냈다고 가정해 봅니다. 그런데 여러분이 깜빡해서 거스름돈을 챙기지 않았다면 그것은 전적으로 여러분의 잘못(?) 혹은 선택(?)입니다. 다시 말해서, 메소드가 어떤 리턴 값을 반환해주었다고 해도 그 결과를 받아서 처

리하는 것은 호출한 쪽의 선택이라는 겁니다.

메소드의 리턴 값과 연산자에는 중대한 차이가 존재하는데 그것은 바로 다음과 같이 정리할 수 있습니다.

- 연산에 대한 결과는 ++/--와 같은 자동 증감을 제외한 경우에 반드시 받거나 처리해야 한다.
- 반면에 메소드의 리턴 값은 결과 데이터를 받는 것을 선택으로 할 수 있다.

> 연산자가 연산 결과를 반드시 처리해야 하는 것에 비해서 메소드의 리턴 값은 메소드를 호출한 사람이 리턴 값을 어떻게 사용할 것인지를 자유롭게 선택한다는 점에서 차이가 있습니다.

이제 결과를 반환해주고 그 결과 데이터를 호출한 쪽에서 다시 사용할 수 있도록 소스를 수정해 보면 다음과 같은 형태가 됩니다.

예제 | 온전하게 데이터를 사용하는 SumMachine 코드

```
public class SumMachine {
	public int makeSum(int startValue, int endValue){
		int start = startValue;
		int end = endValue;
		int sum = 0;

		for(int i = start; i <= end ; i++){
			sum = sum + i;
		}//end for
		return sum;
	}

	public static void main(String[] args) {
		SumMachine m = new SumMachine();
		int result1 = m.makeSum(1,100);
		int result2 = m.makeSum(20,200);
		int result3 = m.makeSum(30,300);

		System.out.println(result1);
		System.out.println(result2);
		System.out.println(result3);
	}
}
```

```
5050
19910
44715
```

메소드를 세 번 호출하고 그 결과는 makeSum( ) 메소드를 호출한 쪽에서 result1, result2, result3와 같은 변수로 받는 모습을 볼 수 있습니다.

4 메소드의 구성 요소를 정리해봅시다.

이제 메소드에 대해서 배운 내용을 좀 정리해볼 필요가 있을 것 같습니다. 우선은 메소드의 세 가지 구성 요소부터 다시 살펴보도록 합니다.

구성 요소	의미
메소드 이름	외부에서 호출할 때 알아보기 쉬운 이름으로 만들어서 다른 사람들이 편하게 알아볼 수 있게 한다.
파라미터	메소드를 실행할 때 필수적인 데이터가 있는 경우에 사용되는 타입에 따라서 변수를 선언해서 메소드를 좀 더 유연하게 만든다.
리턴 타입	메소드의 결과를 호출한 쪽에 실행 직후 결과를 피드백해줄 필요가 있을 때에 void가 아닌 것으로 선언해 준다. 이 경우에는 return이라는 키워드를 이용해서 결과 데이터(리턴 값)를 반환해준다.

표

여러분이 메소드에 대해서 배우기는 했지만 아직까지는 좀 더 연습이 필요할 겁니다. 몇 가지 주어진 상황에서 메소드를 어떻게 설계해야 하는지를 살펴보도록 합니다.

■ 계산기를 만든다면 어떻게 될까요?

아마도 가장 무난한 예제는 여러분이 계산기를 만든다고 하면 어떤 식으로 메소드를 설계할 것인지 생각해보는 겁니다. 우선은 계산기라면 적어도 사칙연산 기능을 가질 것이 분명합니다. 따라서 각각의 메소드를 선언해 보는 연습을 해 보면 좋을 듯합니다.

■ **더하기는 어떻게 선언할까?**

더하기 메소드부터 생각해 보겠습니다. 메소드를 생각할 때에는 다음의 세 가지 순서에 따라서 구성하시면 됩니다.

1_ 만들려고 하는 메소드의 이름을 결정한다.
2_ 메소드를 실행하는 데 있어서 필수적인 파라미터를 결정한다.
3_ 메소드를 실행한 후에 즉각적으로 피드백을 받을 것인지를 결정한다.

위의 순서대로 구성한다면 더하기 메소드의 모습은 다음과 같이 변경될 겁니다.

```
public void add( ) {  }
```

우선은 메소드의 이름만을 결정하고 나머지는 신경 쓰지 않습니다. add( )는 적어도 두 숫자를 더하는 기능을 가지게 될 것이기 때문에 add( ) 메소드가 정상적으로 실행되기 위해서는 두 개의 숫자가 필요할 것이고, 그 숫자는 메소드 내에서만 쓰이는 변수이므로 이름은 알아보기 쉽게 만들어주기만 하면 됩니다.

```
public void add(int x, in y) {  }
```

마지막으로 메소드를 실행하고 나면 그 결과를 즉각적으로 피드백해줄 것인지를 결정합니다. 아마도 add( )라는 메소드를 실행하고 나면 더한 결과를 반환해주어야 할 것이므로 결과 타입을 결정해서 반환하는 데이터를 명시해줍니다.

```
public int add(int x, int y) { ... }
```

이때 int를 반환해주기로 했기 때문에 {} 안의 코드는 반드시 return이라는 구문을 통해서 int 타입의 데이터를 반환해주어야 합니다.

순서를 정확하게 판단할 수만 있다면 생각보다 별로 어려운 일은 아닐 겁니다. 위의 정해진 순서만 정확히 기억해 둔다면 앞으로 메소드를 여러 개 작성할 때에도 도움이 될 것입니다.

5 메소드를 이용해 봅시다: 카우프 지수 구하기

혹시 '카우프 지수'라는 것을 들어 보셨나요? 간단하게 말해서 유아들의 비만 정도를 측정하는 겁니다. 이 비만을 판단하는 기준이 되는 데이터는 다음과 같은 공식을 이용해서 구합니다.

카우프 지수 공식

$$\text{카우프 지수} = \frac{\text{체중(kg)}}{\{\text{신장(m)}\}^2}$$

지수가 30 이상	비만
24~29	과체중
20~24	정상
20 미만	저체중
13~15	여윔
10~13	영양 실조증
10	이하는 소모증

주변에 아이들이나 조카들이 있으시다면 한번 만들어 두시면 좋을 듯합니다. 카우프 지수를 Body Mass Index라고 하기도 합니다.

5.1 작업 ① 클래스를 선언하고 메소드를 결정

가장 먼저 할 일은 역시 클래스를 최초 선언하고 메소드를 만들어서 실행 가능하도록 만들어주는 겁니다. 우선은 main 메소드를 만들어준 다음 카우프 지수를 구하는 메소드의 이름을 calculate( )라고 해봅시다.

예제 | 클래스 선언 + main 메소드 + 메소드 결정하기

```
public class BodyMassIndexMachine {
      public void calculate(){

      }

      public static void main(String[] args) {

      }
}
```

■ **우선은 무조건 실행되도록 만들어줍니다.**

앞에서와 마찬가지로 코드를 작성하며 우선은 실행될 수 있게 만들어주는 겁니다. 이제 main 메소드에 약간의 코드를 넣어서 실행될 수 있도록 작성합니다.

예제 실행 가능하도록 수정한 BodyMassIndexMachine 클래스 코드

```
public class BodyMassIndexMachine {
	public void calculate(){
		System.out.println("비만 지수를 구합니다.");
	}

	public static void main(String[] args) {
		BodyMassIndexMachine m = new BodyMassIndexMachine();
		m.calculate();
	}
}
```

```
비만 지수를 구합니다.
```

main 메소드의 소스를 보면 마치 Scanner를 만드는 것처럼 BodyMassIndexMachine을 new를 이용해서 작성한 후에 m이라는 것을 이용해서 calculate( )를 실행하고 있습니다. 우선은 작성한 calculate( ) 메소드가 동작하는 것을 확인했기 때문에 메소드의 내부에 적절한 코드를 추가해주면 될 듯합니다.

5.2 작업 ② 메소드의 파라미터를 결정

다음 할 일은 메소드의 파라미터를 결정하는 겁니다. 카우프 지수는 체중을 신장의 제곱으로 나눈 데이터에 100을 곱한 결과입니다. 따라서 제대로 계산을 하기 위해서는 당연히 신장 데이터와 체중 데이터가 필요하다고 생각됩니다. 신장과 체중 데이터는 아마도 정수로만 표현하기에는 좀 아쉬운 데이터이므로 float이나 double 타입을 이용해주는 것이 좋을 듯합니다. 간단한 소수의 계산에는 float이 더 나을 듯하니 이제 calculate( ) 메소드는 다음과 같이 수정해주도록 하겠습니다.

예제 | 파라미터를 추가하고 main 메소드를 수정한 코드

```
public class BodyMassIndexMachine {
      public void calculate(float height, float weight){
            System.out.println("비만 지수를 구합니다.");
      }

      public static void main(String[] args) {
            BodyMassIndexMachine m = new BodyMassIndexMachine();

            float h = 1.8F;
            float w = 77F;

            m.calculate(h, w);
      }
}
```

실제로 위의 코드에서 키와 몸무게 데이터는 다음과 같은 과정을 통해서 실행된다고 생각할 수 있습니다.

> 메인 메소드에서 데이터의 변수 선언
> float h = 1.8F; (키 데이터)
> float w = 77F; (몸무게 데이터)
>
> 계산해 주는 장치 BodyMassIndexMachine m을 이용해서 calculate( ) 메소드 실행
> m.calculate(h, w);
>
> 실행하면 calculate( ) 메소드에 데이터 전달
> public void calculate(float height, float weight)에 전달 이때의 연산은 변수의 할당(복사)작업
> float height = h;
> float weight = w;
>
> 실제 메소드의 실행

5.3 작업 ③ 메소드가 즉각적으로 피드백할 것인지 결정하고 로직 완성하기

이제 남은 작업은 메소드의 피드백이 즉각적인지를 판단해서 결과를 반환해주는 완전한 코드를 작성하는 작업입니다. 여러분은 아마도 비만 지수를 구하기 위해서 데이터를 넣으면 바로 결과를 얻기를 바랄 겁니다. 따라서 결과를 바로 반환해주기로 하고, 리턴타입은 소수이니 float를 사용할까 합니다.

실제로 완성된 코드는 다음과 같이 만들어집니다.

예제 완성된 카우프 지수 구하는 프로그램

```
public class BodyMassIndexMachine {
      public float calculate(float height, float weight){
            //키의 제곱
            float hData = height * height;
            //몸무게를 나눈다.
            float result = weight/hData;
            return result ;
      }

      public static void main(String[] args) {
            BodyMassIndexMachine m = new BodyMassIndexMachine();

            float h = 1.8F;
            float w = 77F;

            float index = m.calculate(h, w);
            System.out.println("비만 지수 : " + index);
      }
}
```

```
비만 지수 : 23.765434
```

로직을 분리해서 메소드라는 것을 선언해 보았습니다. 이제 main 메소드 안에 선언된 h, w와 같은 변수의 값을 필요한 데이터의 수치로만 변경할 수 있도록 하면 누구나 자신의 카우프 지수를 구할 수 있게 되었습니다.

6 프로그램이 점점 커지면서 발생하는 상황

실제로 프로그램을 작성하다 보면 메소드 하나로 충분한 경우는 별로 없습니다. 가장 쉽게 생각해 보면 메소드를 하나 만들고 나면 거기에 더욱더 복잡하거나 추가적인 기능을 붙이고 싶은 경우가 많다는 겁니다. 이렇게 추가적인 개발이 요구될 때 개발자들의 선택은 두 가지 중 하나입니다.

- 기존에 개발해 둔 메소드를 수정해서 사용하는 방법
- 기존 메소드를 수정하지 않고 새로운 메소드를 작성해서 같이 연동하도록 하는 방법

이제 앞에서 만들어진 카우프 지수를 구하는 기능에 조금 더 추가적인 기능을 생각해보도록 합니다.

예제 | 완성된 카우프 지수 구하는 프로그램

```
public class BodyMassIndexMachine {
	public float calculate(float height, float weight){
		//키의 제곱
		float hData = height * height;
		//몸무게를 나눈다.
		float result = weight/hData;
		return result ;
	}

	public static void main(String[] args) {
		BodyMassIndexMachine m = new BodyMassIndexMachine();
		float h = 1.8F;
		float w = 77F;
		float index = m.calculate(h, w);
		System.out.println("비만 지수 : " + index);
	}
}
```

```
비만 지수 : 23.765434
```

위의 코드에서 보는 것처럼 메소드를 통해서 로직을 분리해두었더니 실제로 main 메소드에서 호출할 때에는 아주 간결하게 호출만 해주면 됩니다. 하지만, 개인적으로 이 프로그램에 추가적인 기능이 좀 더 나왔으면 좋겠습니다.

지수가 30 이상	비만
24~29	과체중
20~24	정상
20 미만	저체중
13~15	여윔
10~13	영양 실조증
10	이하는 소모증

표 카우프 지수에 의한 건강 측정도

아무래도 좀 더 프로그램이 기능을 가지려면 위의 표를 매번 찾아보지 않고도 그러한 결과를 알아보는 정도는 되어야 하지 않을까요? 예를 들어 위의 예제에서 비만 지수 23.76 정도면 아주 정상적인 몸 상태인데 이것을 매번 이 표를 보면서 확인하니까 좀 불편합니다.

사람들에게 카우프 지수를 손쉽게 만들어 준다고 해도 그다음에는 여러분에게 '그래서 난 30이 넘었는데 그럼 난 건강한 거야?' 혹은 '난 17 나왔는데 그럼 어떤 상태야?'라는 질문을 받게 될 것입니다. 그럼 이제 더 나은 선택은 이런 판단을 프로그램이 자동으로 해주도록 만들어 주는 것이 아닐까요?

■ **그럼 알아서 건강 상태 메시지를 알려주는 기능도 추가합시다.**

기존에 만들어 둔 프로그램에 새로운 메소드를 하나 추가하는 작업은 단순합니다. 그냥 메소드를 추가했던 것처럼 새로운 메소드를 작성하고 { } 를 이용해서 여기서부터 여기까지는 메소드라고 표시해 주는 겁니다.

■ **메소드의 이름은 getResult( )라고 하겠습니다.**

메소드를 작성하는 이름은 getResult( )라고 하겠습니다. 이제 BodyMassIndexMachine은 다음과 같은 형태가 됩니다.

예제 getResult( ) 메소드를 기존 소스에 추가합니다.

```
public class BodyMassIndexMachine {
      public void getResult(){

      }

      public float calculate(float height, float weight){
            //중간 생략
      }

      public static void main(String[] args) {
            //중간 생략
      }
}
```

■ **메소드의 파라미터를 고민해봅시다.**

이제 메소드 작성의 두 번째 단계인 메소드의 파라미터를 고민해봅시다. 사실 메소드의 파라미터는 이미 결정되어 있는 것과 다름없습니다. 당신이 건강한 상태인지 비만의 상태인지를 정확히 파악하기 위한 필수 데이터는 역시 계산된 카우프 지수일 것이기 때문입니다. 따라서 메소드의 파라미터로 구해진 카우프 지수를 사용하도록 해봅시다.

코드

```
public void getResult(float indexValue){

}
```

개인적으로 이건 좀 이상합니다. 왜 그렇게 제가 생각하는지는 잠시 후에 말씀드리도록 하겠습니다.

■ **메소드의 리턴 타입은 어떤 것으로 결정할까?**

getResult( )를 호출하면 결과를 반환해줄까요? 말까요? 즉시 피드백을 할 것이면 리턴 타입을 명시해주는 것이 좋다고 했습니다. 여러분이 문자와 관련되어서 지금 아는 지식은 char밖에 없으므로 리턴 값은 char를 이용하도록 하겠습니다. 생각해 보면 동일한 결과라고 해도 여학생들에게는 좀 더 친절한 메시지를 보여주고, 아저씨들에게는 좀 더 자극적인 메시지를 보여 줄 수도 있기 때문에 결과 메시지에 대한 처리는 현재 프로그램에서 해주는 것은 좋은 선택으로 보이지 않습니다.

지수	결과 메시지
30이상 (비만)	'A'
24 ~ 29 (과체중)	'B'
20 ~ 24 (정상)	'C'
15 ~ 20 (저체중)	'D'
13 ~ 15 (마름)	'E'
10 ~ 13 (영양 실조증)	'F'
10 미만 (소모증)	'G'

표

이제 위의 메시지를 이용해서 메소드를 완성해 보면 다음과 같은 형태가 됩니다.

코드 완성된 getResult( ) 메소드

```
public char getResult(float indexValue){
      if(indexValue > 30){
            return 'A'; //비만
      }else if(indexValue > 24){
            return 'B'; //과체중
      }else if(indexValue > 20){
            return 'C'; //정상
      }else if(indexValue > 15){
            return 'D'; //저체중
      }else if(indexValue > 13){
            return 'E'; //마름
      }else if(indexValue > 10){
            return 'F'; //영양실조
      }else {
            return 'G'; //소모증
      }
}
```

만일 위의 getResult( ) 메소드에 27이라는 데이터를 넣어주면 위에서부터 true가 되는 조건을 하나 실행하기 때문에 'B'라는 결과물을 만들어 낼 겁니다. if ~ else는 위에서부터 조건을 검사하기 때문에 계단처럼 범위를 만들어 주면 'if(indexValue 〉 20 && indexValue 〈24)...'와 같은 코드를 사용하지 않아도 됩니다.

예제 getResult( ) 기능이 완성된 BodyMassIndexMachine 프로그램

```
public class BodyMassIndexMachine {
      public char getResult(float indexValue){

            if(indexValue > 30){
                  return 'A'; //비만
            }else if(indexValue > 24){
                  return 'B'; //과체중
            }else if(indexValue > 20){
                  return 'C'; //정상
            }else if(indexValue > 15){
                  return 'D'; //저체중
```

```
            }else if(indexValue > 13){
                  return 'E'; //마름
            }else if(indexValue > 10){
                  return 'F'; //영양실조
            }else {
                  return 'G'; //소모증
            }
      }

      public float calculate(float height, float weight){

            //키의 제곱
            float hData = height * height;
            //몸무게를 나눈다.
            float result = weight/hData;

            return result ;
      }

      public static void main(String[] args) {

            BodyMassIndexMachine m = new BodyMassIndexMachine();

            float h = 1.8F;
            float w = 77F;

            float index = m.calculate(h, w);
            System.out.println("비만 지수 : " + index);
            System.out.println("건강지수 : " + m.getResult(index));
      }
}
```

```
비만 지수 : 23.765434
건강 지수 : C
```

소스가 좀 길어 보이긴 하지만 실제로 내용은 별로 특이한 점은 없습니다. 자 이제 여러분은 두 가지의 기능을 가지는 프로그램을 작성해 주었습니다.

- 키와 몸무게를 넣으면 자동으로 비만 지수를 구하는 기능
- 비만 지수를 넣으면 건강 등급을 알려주는 기능

그동안 배운 것을 잘 써먹었으니 뿌듯한 마음으로 자리를 털고 일어나려 합니다. 그런데 그 순간 지나가던 친구가 한마디 합니다. "키랑 몸무게 넣어주면 알아서 지수 구한다며? 그럼 건강 결과도 기계가 알아서 구하면 되잖아?"

7 다른 메소드의 실행 결과를 항상 참고할 수 있다면?

순간 여러분은 그 친구에게 살기를 띤 눈빛을 보내시겠지만 조금만 더 깊이 생각해보면 결코 틀린 말이 아니라는 것을 깨닫게 될 겁니다. "맞아. 이전에 분명히 이 프로그램에서 알아서 계산하라고 키와 몸무게를 넣었잖아? 그러니 다음에 그 데이터를 가지고 알아서 결과만 알려주면 되지, 왜 내가 매번 호출할 때마다 모든 필요한 데이터를 알아야만 하지?" 이 개념을 좀 더 쉽게 이해하기 위해서 외국여행을 생각해 봅시다. 자유 여행으로 갈 수도 있을 것이고, 여행사를 통한 패키지 여행으로 갈 수도 있을 겁니다.

자유 여행
돈은 덜 들지만, 신경 쓸 일이 많다.
모든 일은 스스로
항공권 예약도 알아서
숙소도 알아서 구매
여행 계획도 모두 스스로

패키지 여행
돈은 비싸지만, 여행사가 알아서
필요한 개인정보와 금액만 지불
모든 것은 여행사가
난 몸만 가면 됨

기존의 다른 프로그래밍언어에서 함수를 만들어서 구성하는 방식을 자유 여행에 비유하자면 여행사를 통하는 패키지 여행은 객체지향 프로그램으로 비유할 수 있습니다.

> 기능이나 절차가 많아지면 사용하기 불편해지고, 그것을 전문적으로 하는 사람들이 생기듯, 메소드가 많아지고 데이터가 많아지면 그것을 알아서 처리할 수 있는 존재를 생각해볼 수 있는데 객체지향에서 바로 이런 존재를 객체(Object)라고 합니다.

■ 한번 입력한 데이터나 결과 데이터를 가지고 있다면 좋을 텐데

프로그램은 자신이 원하는 것을 마음대로 만들어 낼 수 있는 아주 창조적인 작업입니다. 그러니 여러분도 이런 작업을 상상해 보시면 됩니다. 우선 제가 제시하는 가장 이상적인 시나리오는 이겁니다.

• 프로그램에 내 몸무게와 키를 입력한다.
• 내가 원한다면 비만 지수를 확인할 수 있다.
• 내가 원한다면 건강 정도를 확인할 수 있다.

어떻습니까? 이렇게 만들어 낼 수 있다면 여러분은 그냥 처음에 프로그램에 키와 몸무게를 넣고 나서 나머지는 원하는 대로 입맛에 맞게 선택해 주면 좋을 듯합니다. 예를 들어 인터넷 쇼핑몰을 생각하시면 상대방이 우리가 필요한 데이터를 보관하고 있다는 것이 얼마나 편리한지 알 수 있습니다. 만일 쇼핑몰이 개인정보를 보관하고 있지 않다면 매번 사용할 때마다 배송이나 주문에 필요한 모든 정보를 새로 입력해주어야 합니다. 하지만, 데이터를 보관하고 있으면 훨씬 간소해지는 것을 알 수 있습니다.

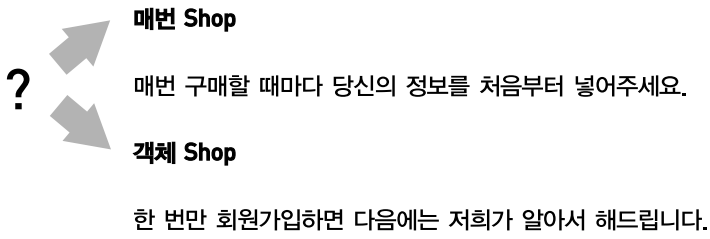


그림 5

■ **로직을 가진 쪽에서 데이터도 보관하면 좋을 텐데...**

지금 우리가 고민하는 이 문제의 근본적인 원인은 우리가 메소드라는 존재를 너무 독립적으로 생각했기 때문입니다. 사실 이렇게 메소드를 완벽하게 독립적인 단위로 보는 것은 전혀 객체지향스럽지 않은 방식입니다. 소위 다른 언어들에서 함수라고 말하는 것이 바로 이 역할이었습니다. 객체지향에서 객체는 로직(기능)뿐 아니라 데이터도 스스로 관리합니다.

> 객체지향 프로그래밍에서 말하는 객체(Object)라는 것이 편리하다고 생각하는 가장 중요한 이유는 코드를 호출하는 입장에서 편하다는 겁니다. 알아서 데이터를 보관하기 때문에 그만큼 신경 써야 하는 일이 줄어듭니다.

객체지향 언어에서는 메소드와 데이터가 분리되는 것보다는 메소드와 데이터가 같이 사용될 수 있다면 조금 더 유연하고 편리한 프로그램을 작성할 수 있을 것이라고 판단했습니다.

■ **로직과 데이터를 따로 분리하지 않기로 합니다.**

"데이터와 로직이 하나로 결합된 형태로 프로그램을 작성한다." 아마도 아직은 이 문장 자체가 쉽게 이해되지 않을 겁니다. 하지만, 조금씩 풀어서 접근해보면 결코 어려운 얘기가 아니라는 것을 아시게 될 겁니다. 사실 우리가 프로그램에 원하는 것은 조금 더 편리해지는 겁니다. 조금만 더, 조금만 더 때문에 기능을 추가하기로 했습니다. 문제는 기능을 추가하다 보니 사실 '이전에 전달됐던 데이터나 이전에 실행했던 결과 데이터를 다시 호출받는 쪽이 보관만 했더라면 상당히 편리했을 텐데'라는 생각에서 시작하는 겁니다. 데이터와 로직(메소드)이 서로 유기적으로 묶인 구조를 객체라고 생각하면 됩니다.

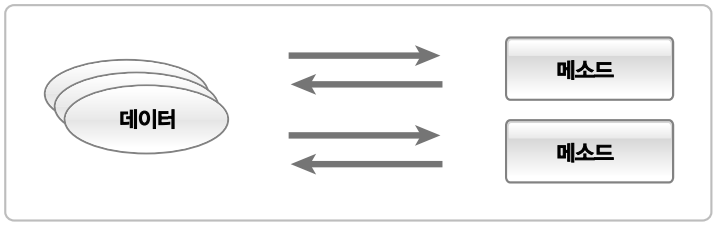


그림 6 데이터와 로직이 하나로 묶인 특별한 구조(객체)

아쉽다는 생각, 더 나은 것을 찾아보자는 생각, 좀 더 편해지자는 생각, 이런 생각에서 사람들은 생각하는 방법을 바꾸게 됩니다. 더 나은 아이디어가 나오고, 그것이 유행을 타게 됩니다. 따라서 우리가 만들 프로그램도 마찬가지로 이런 과정을 거치게 됩니다.

8 데이터와 로직을 결합했더니... 여러 명이 사용하면 문제가 됩니다.

같은 말이지만 어려운 말로 보이는 것들이 꽤 많습니다. 아마도 IT 관련 용어들도 그런 부류라고 생각합니다만, 사실 "데이터와 로직을 결합한다."라는 말도 여행사를 통한 패키지 여행을 예를 들어 보면 쉽게 이해할 수 있는 경우라고 생각됩니다.

• 여러분은 필요한 개인정보와 금액만을 지불합니다.
• 모든 일정과 가이드는 여행사가 알아서 처리합니다.
• 여러분은 식당을 알아볼 필요도, 교통편을 알아볼 필요도 없습니다.

8.1 우리 생활의 거의 모든 기계가 바로 데이터와 로직의 결합체입니다.

좀 갑작스러운 얘기지만 휴대전화가 있으실 겁니다. 휴대전화의 데이터를 생각해보도록 합니다. 우리가 휴대전화를 쓰면서 많은 번호를 기억하지 않게(기억할 필요가 없게) 되었습니다. 그냥 주로 쓰는 번호는 단축키를 지정하고, 그 단축키를 외우는 정도가 더 많습니다(가끔은 제 번호를 잊어버릴 때도 있습니다). 휴대전화가 데이터를 보관해주니 그만큼 편리해졌다고 생각해 볼 수 있습니다. 게다가 휴대전화는 당연히 어떤 로직을 수행할 수 있습니다. 전화를 걸고 받거나, 메시지를 받는 등은 바로 이런 기능들을 의미하는 겁니다. 휴대전화가 데이터를 저장하고 있어서 편리한 점은 너무나 많습니다. MP3나 TV 역시 마찬가지입니다.

8.2 데이터와 로직을 결합해 놨더니만... 막 꼬이네요.

예를 들어 제가 최신형 MP3를 구입했다고 가정해 보겠습니다. 그런데 어느 날 자고 일어났더니 다른 사람이 제 MP3를 가져가 버렸습니다. 문제는 다른 사람이 제가 가진 노래들을 다 지우고, 자신이 좋아하는 노래들로 MP3를 가득 채웠다는 겁니다. 좀 막연한 얘기지만 이 문제는 하나의 MP3를 두 사람이 사용하니까 발생하는 문제입니다. 즉 좀 더 추상화하자면 하나의 로직과 데이터를 가진 존재를 여러 곳에서 같이 사용하다 보니 원하지 않게도 가끔은 데이터가 훼손된다는 겁니다.

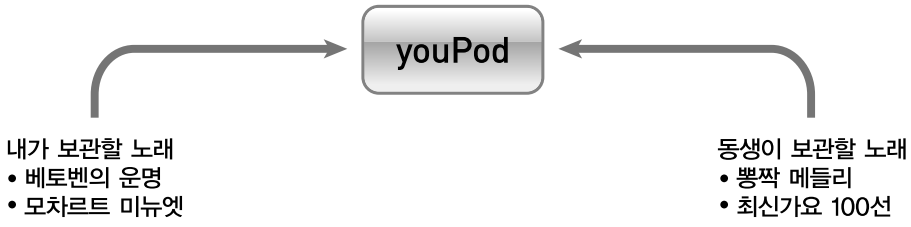


그림 7

프로그램도 마찬가지입니다. A가 키와 몸무게를 이용해서 비만 지수를 구해두었는데, B가 자신의 키와 몸무게를 입력해 주면 A의 결과 데이터는 사라지게 되는 겁니다. 이렇게 하면 애써 데이터와 로직을 하나로 묶은 것이 의미가 없게 됩니다.

8.3 MP3를 가지고 두 형제가 싸운다면? 가장 좋은 해결책은 각자 사주는 것

한 개의 MP3를 두 사람이 사용하게 되면 데이터가 훼손됩니다. 이 경우 가장 쉬운 부모님의 해결책은 '같은 것을 각자 하나씩' 갖도록 하는 겁니다. 여기서 중요한 것은 '각자'라는 것이 의미하는 '동일한 기능을 가진 다른 데이터'라는 겁니다. 즉 같은 기종의 MP3라면 같은 기능을 가지게 되지만, 그 안에 있는 데이터는 사용자마다 다를 수 있다는 겁니다.

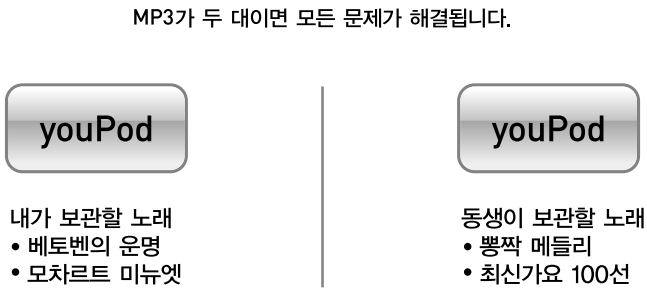


그림 8

프로그래밍에서도 이런 방법을 생각하게 됩니다. 같은 로직을 가지지만 때에 따라서 다른 데이터를 보관할 수 있는 방식의 프로그램을 만드는 것, 이것이 바로 객체지향 프로그래밍의 핵심입니다.

9 클래스와 객체의 개념: 데이터와 로직을 하나의 덩어리로 만들고, 각자 데이터를 쓸 수 있게 하자!

클래스(Class)는 데이터와 로직을 묶어서 누구나 동일한 기능을 사용할 수 있게 하고, 누구나 필요한 자기만의 데이터를 보관하기 위해서 스펙을 결정해 둔 것을 의미합니다. 예를 들어 MP3라면 노래라는 데이터는 각자 다르지만, 노래를 듣는 기능은 모든 MP3가 동일한 기능이 지원됩니다.

프로그래밍에서 사람들은 이 방법을 실현하기 위해서 하나의 새로운 방법을 생각해냅니다. 방법의 요지는 아주 간단합니다. '**어떤 데이터와 로직을 하나로 결합하고, 필요하다면 그런 존재를 복사해서 각자 필요한 데이터를 따로 보관할 수 있게 하자**'는 겁니다.

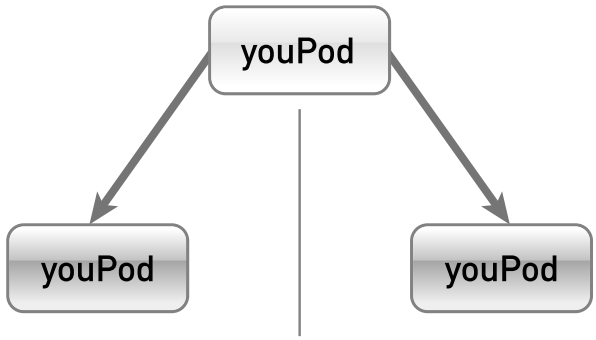


그림 9

각자가 휴대폰을 가지고 있다면 그 안에 있는 데이터는 각자의 것이 됩니다. 따라서 우리가 데이터를 따로 보관하지 않아도 자신의 휴대폰이 있다면 얼마든지 예전에 보관해둔 번호를 사용할 수 있게 됩니다. 물론 다른 사람의 휴대폰 역시 마찬가지입니다.

지금까지의 프로그램 발전 순서를 보면 다음과 같다고 볼 수 있습니다.

1_ 동일한 로직을 여러 번 사용하려다 보니 하나의 묶음으로 묶는 것이 좋다고 생각합니다. 그리고 이런 묶음을 '함수'나 '메소드'라고 하고, Java에서는 메소드라는 용어로 사용합니다.
2_ 그런데 이 메소드들이 많아지니까 점점 그 사이를 오가는 데이터도 많아지게 되어서 불편해 보이기 시작했습니다.
3_ 그래서 가능하다면 호출할 때 필요한 데이터가 보관된다면 더욱 좋을 것 같다는 생각을 하게 되었습니다. 예를 들어 한번 호출할 때 입력했었던 데이터를 보관해주거나, 어떤 기능을 호출했을 때의 결과 데이터를 호출받은 쪽에서 보관하면 좋을 듯합니다.
4_ 이제 사람들이 함수와 데이터를 하나의 덩어리로 묶는 것이 좋겠다고 생각하기 시작합니다.
5_ 문제는 이렇게 묶어준 덩어리를 여러 명이 같이 사용하면 기존 데이터가 사라지거나 하는 문제가 발생한다는 겁니다. 따라서 사람들은 필요한 만큼 덩어리를 만들어 내서 사용할 수 있으면 좋겠다고 생각합니다.

9.1 클래스란? 객체지향에서 로직과 데이터를 결합해서 남들이 사용하도록 하는 스펙

객체지향 프로그래밍(Object-oriented Programming, OOP, 이하 객체지향)에서는 로직과 데이터를 묶어둔 하나의 단위를 클래스(Class)라고 합니다. 그렇다면, 클래스가 어떤 모습을 가질

것인지 상상해보도록 하겠습니다.

클래스(Class)는 복사본을 만들기 위한 원형(Prototype)과 같습니다. 따라서 다음과 같은 특징을 가집니다.
- 각자 따로 보관하고 싶어하는 데이터 혹은 속성들을 정의
- 데이터에 관계없이 제공되는 같은 기능이나 로직

■ **아마도 덩어리에 이름이 있겠지요: 클래스도 이름이 있습니다.**

데이터와 로직을 결합해 두었는데, 로직 하나하나도 이름을 붙이니 당연히 이런 존재에도 이름이 있을 겁니다.

■ **아마도 { }를 사용하지 않을까요?**

Java에서 의미를 구분하는 가장 보편적인 경계선은 바로 {}를 이용하는 겁니다. 그러니 클래스라는 것을 만들어도 {}를 이용할 겁니다.

■ **클래스 안에는 변수가 선언되어 있지 않을까요?**

'데이터와 로직'을 결합한다고 했습니다. 그러니 당연한 얘기지만 그 안에 데이터를 보관하기 위한 장치가 있을 것이고, 그런 장치라면 당연히 변수입니다(객체마다 보관하는 변수라는 의미에서 인스턴스 변수라고 합니다 자세한 내용은 좀 더 뒤에서 설명하도록 합니다). 만일 여러 개의 데이터를 보관하기 바란다면 이런 변수의 선언이 여러 개일 수 있을 겁니다.

■ **클래스 안에는 메소드 선언도 있을 겁니다.**

이미 앞에서 메소드라는 것을 본 적이 있으니 자세한 설명은 생략해도 될 듯합니다. 필요하다면 메소드가 여러 개 있을 겁니다. 이런 모습으로 클래스의 모습을 상상해 보면 다음과 같은 형태가 되지 않을까 합니다.

```
클래스 이름 {
        데이터를 담을 변수1
        데이터를 담을 변수2
        ...
        데이터를 담을 변수n
```

```
        어떤 로직을 수행하는 메소드1
        어떤 로직을 수행하는 메소드2
        ...
        어떤 로직을 수행하는 메소드n
}
```

어떻습니까? 조금은 억지스러운가요? 하지만, 데이터는 변수로 처리될 것이고, 로직이나 기능은 메소드이니 이것을 결합한 형태는 적어도 {}를 이용해서 처리될 겁니다.

9.2 클래스의 복사본을 객체(Object, Instance)라고 합니다: 실제로 사용할 때에는 충돌이 안 나게 대량생산합니다.

앞서 얘기했듯이 '데이터와 로직을 하나의 덩어리'로 묶었다면 실제에서는 이렇게 구성된 존재를 복사해서 사용할 겁니다.

> 객체지향 프로그래밍에서의 프로그래밍의 순서는 다음과 같습니다.
> ① 만들고 싶은 존재(Object)를 만들어 내기 위한 틀(Class)을 먼저 작성합니다.
> ② 틀(Class)을 선언한 다음에는 필요할 때마다 복사본(Instance 혹은 Object)을 만들어서 각자 필요에 맞게 사용합니다.
>
> 객체지향 프로그래밍에서 인스턴스(Instance)라는 말과 객체(Object)라는 용어는 같은 의미로, 어떤 틀(스펙)의 역할을 하는 클래스(Class)에서 나온 복사본의 존재를 의미합니다.

실제로 여러분이 클래스를 작성했다면 사용해야 할 때에는 이와 비슷한 방법으로 사용하게 될 것입니다. 이전의 설명에서 '복사'라고 했던 방법을 사용하듯이 말입니다. 객체지향 언어에서는 이렇게 클래스에서 복사본의 존재들을 객체(Object)라고 합니다. 즉 객체는 클래스에 선언한 데이터와 로직을 필요한 순간에 원하는 대로 사용하기 위해서 만들어지는 실체라고 볼 수 있습니다. 우선 간단히 데이터와 로직을 가진 존재를 생각해 내기 위해서 간단히 저금통을 예제로 들어서 설명해볼까 합니다.

10 저금통을 클래스로 만들고 객체로 만들기

객체지향 프로그래밍에서 객체는 마치 굉장히 단단한 철골 구조물을 만드는 것과 유사하게 만들어집니다.
- 우선 원하는 존재를 만들기 위해서 거푸집과 같은 존재를 클래스로 작성합니다.
- 거푸집의 모양은 한번 결정되면 모든 결과물 존재들은 거푸집 내용물과 동일한 모습입니다.
- 거푸집이 완성되면(클래스를 만들어 내면) 원하는 만큼 실제 존재(객체)를 생산해서 사용합니다.

실제로 프로그래밍을 해봅시다. 저금통이라는 존재가 세상에 있다고 가정합니다.

- 저금통이 있어서 나는 매번 금액이 어떻게 되는지 알 필요가 없습니다.
- 저금통에 돈을 넣으면 알아서 금액 데이터가 수정됩니다.
- 나중에 저금통을 깨면 전체 금액을 알 수 있게 됩니다.

저금통의 데이터
- 금액 데이터

저금통의 기능
- 돈을 넣는다.
- 모든 금액을 찾는다.

그림 10

저금통에는 어떤 기능이 필요할까요? 우선은 저금통에는 당연히 돈(데이터)을 넣는 기능이 필요할 듯합니다. 이 기능은 로직이 되고, 이를 함수나 메소드라는 이름으로 부르는 것입니다. 돈을 넣어주면 원래 저금통 안에 데이터에 추가된 만큼 금액 데이터를 추가하면 됩니다. 돈을 빼는 기능도 넣을까요? 돈을 꺼내는 기능을 하게 되면 저금통 안의 잔액은 0원이 되면 됩니다. 이렇게 하기로 하면 결국 메소드는 두 개를 만들기로 결정합니다.

- 돈을 넣는다: deposit
- 모든 돈을 찾는다: withdraw

이제 다음으로 이런 메소드들이 동작할 때 어떤 데이터를 보관할지 생각해봅니다. 당연히 저금통이기 때문에 금액 데이터가 사용될 것입니다.

• 입금된 모든 금액 데이터

이 모습을 가지고 프로그램을 설계하게 되면 다음과 같은 모습으로 만들어집니다.

예제 | 간단하게 구조만 잡은 PigSave 프로그램

```
public class PigSave {
	public int total;

	public void deposit(){
		System.out.println("저금통 입금");
	}

	public void withdraw(){
		System.out.println("저금통 배 따기");
	}
}
```

프로그램을 보면 약간 이상한 점이 몇 가지 있습니다. 이에 대해서도 조금 더 설명하는 것이 좋을 듯합니다.

10.1 main 메소드는 어디로?

여러분이 본격적으로 객체지향 프로그래밍을 하기 위해서 클래스를 만들고 객체를 만들면서 가장 먼저 느끼는 당혹감은 main 메소드가 없다는 겁니다. main 메소드가 없으므로 어떻게 실행해야 하는지가 우선 좀 난감합니다.

■ main 메소드를 안 만든 이유

객체지향 프로그래밍에서는 가장 중점을 두는 것은 로직보다는 객체입니다. 무슨 말이냐면 객체는 나중에 재사용할 것인지를 따지는 것이기 때문에 로직을 고려하지 않고, 어떤 존재가 해야 하는 일을 고려하는 것이 가장 중요한 특징이라고 할 수 있습니다. 우리는 main 메소드로 주로 어떤 로직을 실행하는 순서대로 만드는 것에 익숙합니다. 하지만, 지금 만드는 것은 클래스와 객체이지 main 메소드에 로직을 코딩하는 방식에서 시작하는 것이 아니기 때문입니다.

10.2 다른 곳에서 main을 만들어서 객체를 생산합시다.

방금 여러분이 만든 것을 클래스는 결국 객체를 생산하기 위한 장치입니다. 그러니 이제는 객체를 생산해야 하는데 이 생산하는 방법이 이전 방식과 약간 다릅니다.

예제 PigSave 클래스에서 객체를 만들어서 사용하는 프로그램 PigSaveTest

```
public class PigSaveTest {
      public static void main(String[] args) {
            PigSave save = new PigSave();
            System.out.println(save);
      }
}
```

```
PigSave@1fb8ee3 ← @뒤의 결과는 다를 수 있습니다.
```

PigSaveTest의 코드에는 main 메소드를 만들어 주었습니다. main 메소드를 만들었다는 것은 실행해서 결과를 보겠다는 뜻입니다. 여러분이 가장 눈여겨봐야 하는 사실은 PigSaveTest 프로그램을 작성하는 사람은 PigSave 안의 프로그램이 어떻게 작성되었는지는 모른다는 겁니다. 즉 누군가 PigSave를 만들었다면 그것을 활용하는 수준으로 프로그램을 작성하고 있다는 겁니다.

10.3 PigSave save = new PigSave( ); 클래스에서 객체를 만드는 코드

PigSave save = new PigSave( ); 코드 해석

- new PigSave( ): 'PigSave라는 클래스에서 새로(new) PigSave 객체를 생산해라.
- PigSave save: save라는 이름의 상자를 하나 만든다. save 상자에는 PigSave 클래스에서 생산된 객체의 리모컨(레퍼런스)이 들어갈 것이다.

Java 프로그래밍에서 가장 중요한 구문을 하나 뽑으라면 지체 없이 'PigSave save = new PigSave( );'를 들 수 있습니다. 이 코드야말로 객체지향의 가장 기본적인 사용 방법을 얘기해주는 코드이기 때문입니다.

■ new PigSave( ); PigSave 클래스에서 객체를 만들어내라는 얘기

이 코드를 해석할 때에는 뒤에서부터 해석하는 것이 좋습니다. 앞서 객체는 데이터들과 로직들을 하나의 단위로 보관하는 것이고, 이것을 따로따로 사용하기 위해서 클래스라는 개념과 복사본인 객체라는 개념으로 발전했다는 것을 설명했습니다.

new PigSave( );라는 코드가 하는 일이 바로 이겁니다. PigSave 클래스에서 데이터와 로직의 덩어리를 하나의 복사본으로 만들어내는 코드입니다. 이렇게 만들어진 객체를 이용해서 객체 안에 보관된 데이터를 사용하거나, 객체가 가진 로직을 처리할 때 사용합니다.

■ PigSave save = ... 변수의 선언

PigSave save = ...의 코드의 의미는 말 그대로 변수의 선언입니다. 그런데 변수의 선언이 조금 이상합니다. 지금까지 PigSave라는 것을 변수 선언에서 사용해 본 적이 없으니까요. 조금씩 나누어서 설명하도록 하겠습니다.

■ save = new PigSave( )일 때 변수 상자에는 무엇이 담길까?

가장 먼저 처리해야 할 것은 'save = new PigSave( );'에서부터 입니다. 대체 이 코드의 의미는 무엇일까요? save라는 이름이 붙은 상자에는 무엇이 들어갈까요? 앞에서 배열을 다룰 때 왜 변수 안에 모든 데이터를 직접 넣지 않고, 리모컨(레퍼런스)을 넣었는지 설명한 적이 있습니다. 변수 상자에 데이터를 넣지 않고 리모컨을 넣는 이유는 간단히 말해서 데이터가 너무 크다는 겁니다. 객체라는 것도 생각해 보면 여러 개의 데이터와 로직을 가진 존재입니다. 그러니 이런 존재를 실제로 메모리상의 상자에 담아서 주고받는 것은 상식적으로 생각해 봐도 좋은 방법이 아닙니다. 따라서 객체를 만들고 변수로 처리한다는 것은 간단히 말해서 배열처럼 리모컨이 들어가는 개념으로 봐야만 합니다.

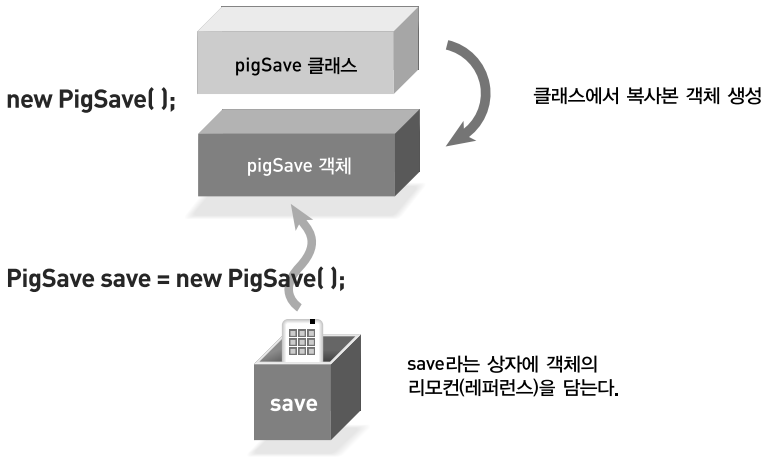


그림 11

■ PigSave save = ...의 변수의 타입

Java에서 변수라는 것은 처음 선언할 때 상자의 종류를 지정합니다. 'int a'라는 변수는 정수용 상자라는 의미입니다. 그럼 위 코드의 의미는 해석하자면 'PigSave용 상자'라는 뜻입니다. 배열과 비교해 보도록 합니다. 'int[ ] arr'인 경우에는 arr이라는 이름이 붙은 상자는 int[ ]의 리모컨이 담긴다는 뜻입니다. 그러니 PigSave save는 PigSave의 리모컨을 담는다는 겁니다.

■ 출력되는 결과는 대체 무엇인가?

'System.out.println(변수);'는 변수 안의 내용물을 출력합니다. 그렇다면, 지금 나오는 것은 무엇을 의미할까요? 상자 안에는 리모컨이 들어가 있으니 당연히 출력되는 것은 리모컨과 관련된 정보라고 생각하시면 됩니다. 조금 더 구체적으로 설명하자면 리모컨이 가리키는 객체와 관련된 정보라고 하는 것이 더 정확한 표현입니다(흔히 말하는 C 언어의 주솟값은 아니지만 비슷하다고 할 수는 있습니다).

■ Java에서는 레퍼런스(Reference)라고 합니다.

Java에서는 "save라는 변수에 PigSave 객체의 레퍼런스가 담겨 있다."라고 표현합니다. C 언어에 익숙한 분들을 위해서 조금 설명을 하자면 Pointer는 가리키는 대상을 변경할 수 있지만, 레퍼런스의 경우에는 레퍼런스가 가리키는 메모리상의 존재를 변경할 수 없습니다. 사실 레퍼런스라는 용어로 표현해주어야 하지만 여러분의 이해를 돕기 위해서 이 책에서는 계속해서 리모컨이라는 표현을 사용하도록 하겠습니다.

■ **코드를 조금 바꾸면 어떨까요?**

리모컨의 개념은 무척 중요합니다. 코드를 아래와 같이 약간 수정한 후의 실행 결과를 보면서 해석해보도록 합시다.

코드 | 변수를 추가한 main 메소드

```
public static void main(String[] args) {
      PigSave save = new PigSave();
      System.out.println(save);
      PigSave save2 = save;
      System.out.println(save2);
}
```

```
PigSave@61de33
PigSave@61de33
```

위의 코드에서는 변수의 처리와 실행 결과를 정확히 해석할 수 있어야 합니다. 우선은 코드를 보면서 알아봅니다.

코드 |

```
PigSave save = new PigSave();
PigSave save2 = save;
```

변수 save2가 추가되었습니다. 그런데 이 save2의 변수는 상자 안에 save 변수의 내용물을 복사하고 있습니다. 덕분에 save라는 상자와 save2라는 상자는 다음 그림처럼 됩니다.

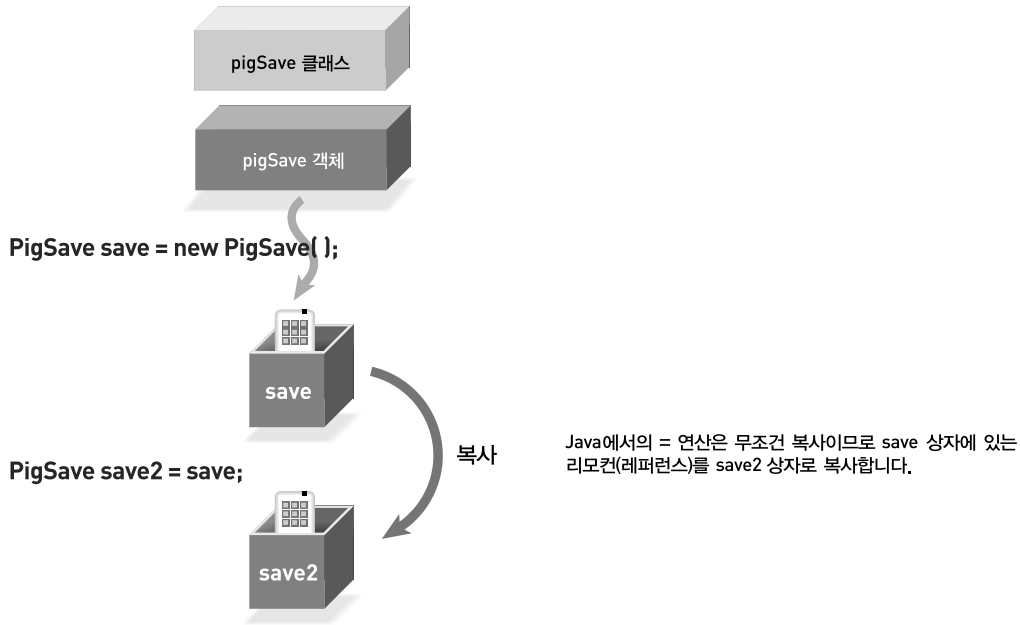


그림 12

그림을 보시면 두 상자에 있는 리모컨들이 같은 대상을 가리키는 리모컨임을 알 수 있습니다. 이렇게 되면 save라는 변수를 출력해서 알아본 리모컨의 정보와 save2를 출력해서 알아본 결과는 동일하게 됩니다(같은 리모컨을 복사해서 save2에 담았기 때문에).

■ **이번에는 클래스에서 객체를 두 개 만들어봅니다.**

이제는 클래스에서 객체를 두 개 만들어보도록 합니다.

예제 | 하나의 PigSave 클래스를 이용해서 객체를 두 개 생성하기

```
public class PigSaveTest {
      public static void main(String[] args) {
            PigSave save = new PigSave();
            System.out.println(save);

            //PigSave save2 = save;
            PigSave save2 = new PigSave();

            System.out.println(save2);
      }
}
```

```
PigSave@c17164
PigSave@1fb8ee3
```

코드를 보면 new PigSave( )를 두 번 실행합니다. 이 말은 결국 PigSave 클래스에서 객체를 두 개 만들어서 각 변수에 그 리모컨을 담는다는 겁니다. 그러니 각 변수에 담긴 리모컨은 다른 존재(객체)의 리모컨이고 이 리모컨을 출력해보면 결과가 다른 것을 볼 수 있습니다.

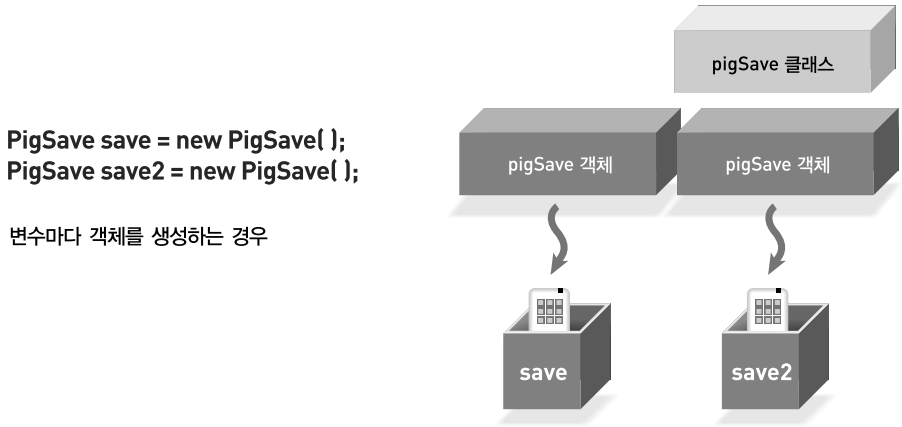


그림 13

11 객체 자료형: 객체의 리모컨을 담는 변수

이제야 제대로 Java 언어의 변수에 대해서 설명할 수 있게 되었습니다. 사실 Java 언어의 변수는 두 가지 스타일로 나누어서 구분합니다.

- 변수에 데이터가 직접 담기는 스타일: 기본 자료형이라고 합니다.
- 변수에는 리모컨만 담는 스타일: 객체 자료형이라고 합니다.

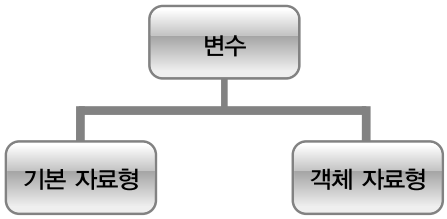


그림 14

기본 자료형은 여러분이 공부했던 byte, short, int, long과 같은 변수들을 의미합니다. 이런 변수들은 변수의 상자를 만들 때 아예 데이터를 다 담아 둡니다. 따라서 상자의 크기가 바로 내용물이 담길 수 있는 크기가 됩니다.

■ **상자 안에 들어갈 내용물이 너무 커지니까 리모컨(레퍼런스)만을 담는 스타일로 변합니다.**

지금까지는 클래스라는 것이 '데이터와 로직'을 하나로 묶는 단위를 의미하는 것이고, 이를 원하는 만큼 복사해 내서 객체라는 것으로 사용한다는 개념을 막 완성했습니다. 데이터를 여러 개를 하나의 묶음으로 묶는 것은 사실 배열에서도 했던 작업입니다. 배열과 클래스에서 생산된 객체는 둘 다 여러 개의 데이터를 가질 수 있는 구조라는 공통점이 있습니다. 따라서 배열처럼 객체를 처리할 때에도 변수에는 리모컨만 담고 실체는 메모리상에 존재하도록 하는 방식을 사용합니다.

11.1 기본형 자료는 상자의 내용물로 데이터가 들어갑니다.

기본형 자료의 선언을 봅니다.

```
int a;
혹은
int a = 10;
```

위의 방식은 변수를 선언만 한 것이고, 아래의 것은 변수를 선언하자마자 변수 안에 데이터를 넣어준 형태입니다. 두 코드 모두 실행될 때 메모리에 상자를 만듭니다. int인 경우라면 4byte이므로 32bit의 공간이 상자로 만들어진다고 볼 수 있습니다. 생각해 보시면 이 각각의 bit는 0과 1이라는 데이터 중의 하나만 가지는데 기본은 0이라는 값으로 채워집니다. 따라서 기본형 자료는 상자가 만들어지면 자동으로 값을 가지는 형태가 됩니다. 이런 이유로 기본형 자료들은 다음과 같은 값들을 기본값으로 가집니다.

• byte, short, int, long: 숫자이므로 0
• double, float: 소수이지만 데이터가 없으므로 0.0
• char: 아스키코드에 따라서 ' ' (공백 문자)
• boolean: false

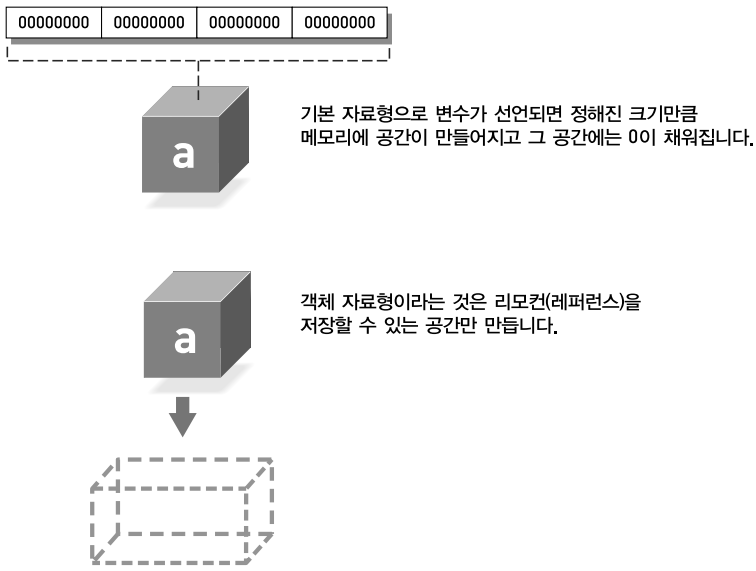


그림 15

11.2 객체 자료형의 변수는 만들어질 때 어떤 모습일까?

기본형 자료는 메모리의 상자 자체가 데이터를 담는 공간이지만 객체 자료형인 경우에는 상자는 리모컨을 담는 곳입니다. 변수를 선언하면 다음과 같이 됩니다.

```
PigSave a;
혹은
PigSave a = new PigSave( );
```

아래쪽의 코드는 명확히 객체를 만들고 리모컨을 넣은 것을 확인했지만 대체 위의 코드가 실행되면 어떤 일이 벌어질까요?

■ **변수가 선언되었으니 메모리에 상자가 만들어질 겁니다.**

우선은 변수가 선언되었으니 메모리에 상자가 만들어질 겁니다. 문제는 이 상자의 내용물입니다. 이 상자에는 특이하게도 'null'이라는 것이 들어갑니다. null이라는 것을 이해하는 가장 쉬운 방법은 리모컨은 리모컨인데 그냥 껍데기만 리모컨이고 실제로 동작하지 않는 존재라고만 생각하면 됩니다.

11.3 null이라는 껍데기 리모컨

```
PigSave a;
혹은
PigSave a = null;
```

위의 코드는 상자를 만들기는 합니다. 다만, 그 상자에 아직 실제의 리모컨은 들어가지 않은 상태라고 생각하시면 됩니다.

객체 자료형이라는 것은 리모컨(레퍼런스)을
저장할 수 있는 공간만 만듭니다.
이때 변수 상자에는 텅빈 리모컨이 생성됩니다.
이 텅빈 리모컨을 null이라 합니다.

그림 16

가끔 어떤 물건을 사면 종이 모형을 끼워주는 경우를 보신 적이 있을 겁니다. null을 그런 것으로 생각해 주시면 좋습니다. 아니면 집에서 쓰던 TV를 버리고 난 후에 리모컨만 남아 있는 경우가 있습니다. 이런 경우가 null이라고 생각하시면 됩니다. 이 두 비유의 공통점은 리모컨을 눌러봤자 실제로 움직이는 것은 아무것도 없다는 겁니다. null이라는 존재는 상당히 애매모호합니다(객체 자료형도 아니고 기본 자료형도 아닌). 우선은 여러분이 편하기 이해하기 위해서 그냥 껍데기뿐인 리모컨이라고 생각하시는 것이 편리합니다. null에 대해서는 다음과 같은 사항을 알면 프로그램을 작성할 때 조금 도움이 됩니다.

• 객체 자료형 변수의 상자에 내용물이 없을 때 사용한다.

• 제어문에서 변수에 리모컨이 실제 객체를 가리키는 데 사용한다.
  – if(a == null)이라는 연산을 한다는 것은 a가 객체 자료형이라는 것이고, a 안의 리모컨이 가리키는 객체가 있는지 없는지를 따질 때 사용합니다.

11.4 NullPointerException이라는 에러

프로그램을 실행하다가 보면 여러 종류의 에러를 보게 됩니다. 역시나 가장 많이 겪는 것은 문법이 잘못되어 발생하는 컴파일 에러지만, Java를 처음 공부할 때 가장 많이 만나는 에러 중의 하나가 바로 NullPointerException입니다. 아래의 코드를 보면서 설명하도록 하겠습니다.

코드

```
PigSave save = new PigSave();
save.deposit();

save = null;
save.deposit();
```

```
저금통 입금
Exception in thread "main" java.lang.NullPointerException
      at PigSaveTest.main(PigSaveTest.java:10)
```

위의 소스를 실행한 결과를 보면 처음에는 제대로 실행되었지만, 아래쪽의 코드를 실행할 때 에러가 발생합니다. 그림으로 한번 자세히 살펴볼까요? 우선은 소스의 위쪽입니다. 중요한 코드는 'save = null;'이라는 코드입니다.

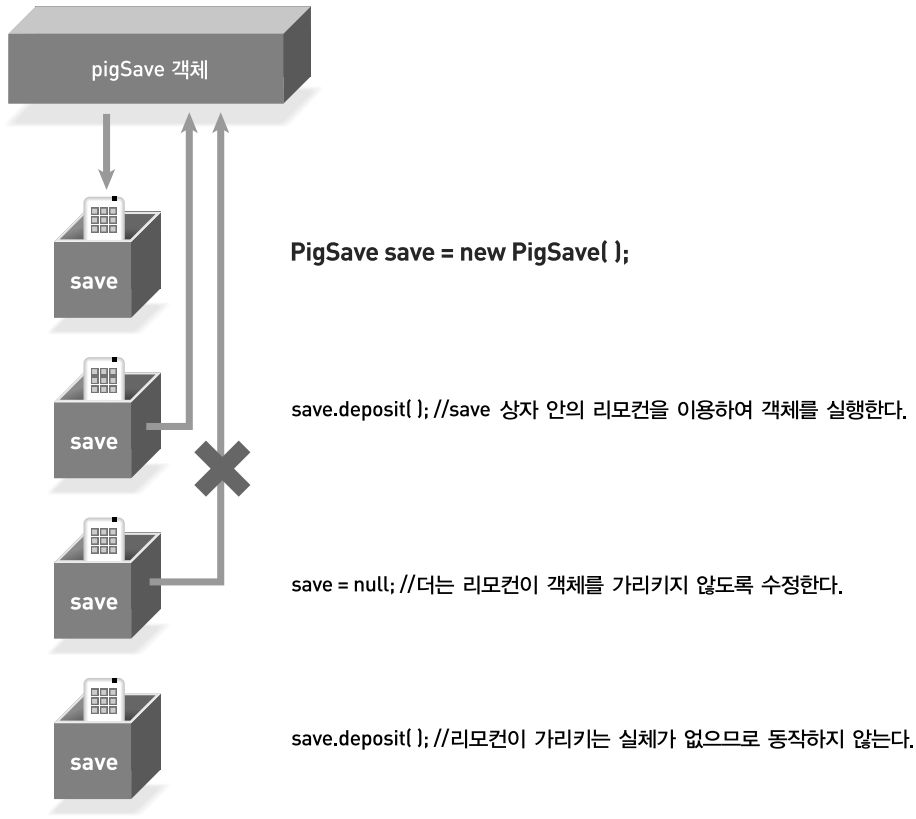


그림 17

결과적으로 실제로 실행될 녀석이 없으므로 발생하는 겁니다.

11.5 NullPointerException을 해결하는 가장 쉬운 비법

여러분이 작성한 코드에서 NullPointerException을 만나게 되었을 때 가장 쉬운 방법은 발생하는 위치의 소스에서 '.'를 사용하는 부분을 찾아내는 겁니다. NullPointerException은 상자 안에 담긴 리모컨이 종이 리모컨인데 이것을 누르면서 실제로 움직이기를 원할 때 발생하는 겁니다. 그리고 리모컨을 누르는 동작은 바로 '.'을 이용할 때밖에 없으므로 어떤 코드인지를 찾아내고 '.'으로 이용되는 상자(변수)를 확인해주시면 됩니다.

12 객체와 데이터의 관계

객체지향 프로그래밍에서 말하는 객체(Object)란 결국 메소드라고 하는 기능들 + 데이터들의 결합체입니다. 그리고 객체는 클래스라는 존재에서 필요한 만큼 생산해서 사용합니다. 메소드를 통해서 데이터를 저장하고, 메소드는 데이터를 이용해서 제어됩니다.

앞에서 객체라는 것이 어떤 배경에서 나오게 되었는지를 설명했다면 이제는 구체적인 몇 가지의 내용을 살펴볼 차례입니다.

12.1 객체의 데이터(인스턴스 변수)

인스턴스 변수는 클래스의 복사본인 각각의 객체마다 가지는 데이터를 의미합니다. 같은 MP3가 여러 대 있다면 MP3마다 가지는 노래 데이터가 바로 인스턴스 변수가 됩니다.

다시 앞에서 만들고 있던 PigSave에서 프로그램을 시작해봅니다.

예제

```
public class PigSave {
	public int total;
	public void deposit(){
		System.out.println("저금통 입금");
	}

	public void withdraw(){
		System.out.println("저금통 배 따기");
	}
}
```

PigSave는 돈을 입금하는 기능과 돈을 다 비우는 기능 두 가지 기능을 가지도록 설계되었습니다. 누군가 필요하다면 이 기능과 더불어서 total이라는 데이터도 별도로 보관하기 위해서 객체로 생성하게 되어 있습니다. 자 그럼 이제 total이라는 변수를 좀 더 살펴보도록 합니다.

12.2 클래스의 { } 안쪽에 선언한 변수는 '객체마다 가지는 데이터'

가장 우선 알아야 하는 것은 메소드의 들여쓰기 레벨과 동등하게 클래스의 { } 안쪽에 선언되는 변수를 볼 때는 '객체마다 가진 ~ 데이터'로 해석해 주시면 좋다는 점입니다. 이것은 클래스에서 생산된 객체마다 독립적으로 그 데이터를 가지고 있다는 뜻입니다. 즉 예를 들어 저금통이 3개라면 저금통마다 금액은 다른 것과 같은 원리입니다.

■ 저금통마다 다른 금액이 있기에: 미리 데이터를 정해야 할까요?

소스를 보면 변수 선언 부분에 선언부만 있고 초기화는 하지 않은 것을 볼 수 있습니다.

코드

```
public int total;
```

클래스의 선언 내부에 선언된 변수를 선언만 해두는 이유는 간단합니다. 어차피 이 변수는 객체마다 다른 값을 가질 것이기 때문에 굳이 어떤 값을 지정하는 것이 의미가 없다는 겁니다. 즉 이 변수는 우리에게는 각자의 휴대전화 번호와도 같습니다. 모든 휴대전화에 번호라는 데이터가 있기는 하지만 그 값은 모두 다르듯이 객체마다 어떤 데이터가 있는데 그 값은 다르게 보일 때 사용하는 것이 바로 객체의 데이터입니다. 따라서 그냥 이런 데이터를 가질 것이라고 선언만 해두어도 됩니다.

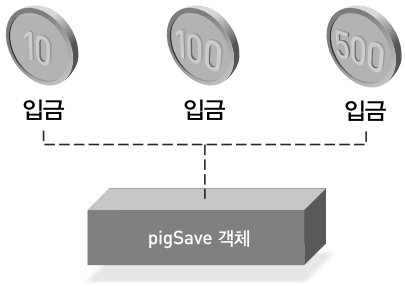
> 인스턴스 변수는 필요하다면 클래스에서 나온 모든 객체마다 다른 값을 가질 수도 있습니다(예를 들어 여러분의 휴대전화 번호를 생각하시면 됩니다). 따라서 인스턴스 변수는 필요하지 않다면 굳이 변수의 값을 지정하지 않아도 됩니다.

12.3 저금통에 돈을 넣어주면 total 데이터에 누적됩니다.

객체가 가지는 데이터를 '인스턴스 변수(Instance Variable)'라고 합니다. 지금부터는 이 변수를 언제 사용하게 되는가에 대해서 좀 더 살펴볼 생각입니다. 우선 가장 먼저 기억해야 하는 것은 메소드의 실행 결과를 누적해서 보관하는 용도로 사용한다는 겁니다.

> **인스턴스 변수의 용도**
> 메소드를 통해서 실행된 데이터를 보관하는 역할을 합니다.

저금통은 그런 개념을 설명하는 데 있어서 아주 좋은 예입니다. 생각해 보면 저금통에 넣는 금액만큼 저금통 내부의 금액 데이터는 증가하게 됩니다. 만일 처음에 저금통에 100원을 넣고 그다음에 500원을 넣었다면 600원이라는 것이 최종 total 값이 됩니다. 즉 어떤 메소드를 호출했을 때의 그 결과 데이터가 계속해서 누적됩니다.



객체의 total 금액은 10, 110, 610으로 누적해서 증가합니다.

그림 18

이렇게 누적해서 쌓이는 데이터 덕분에 호출하는 입장에서는 데이터를 매번 들고 다니지 않아도 됩니다. 이런 모습을 보면 마치 로그인하는 인터넷 쇼핑몰 같기도 합니다. 이제 저금통 안의 deposit이라는 메소드를 조금 수정해서 입금이라는 행위를 할 때 일정 금액을 받도록 수정해봅니다. 물론 이렇게 입금된 금액은 total이라는 데이터에 누적되도록 해야 합니다.

예제 | PigSave의 total 변수와 deposit 메소드

```
public int total;

public void deposit(int amount){
      System.out.println("저금통 입금");
      total = total + amount;
}
```

소스를 위와 같이 수정해봅니다. 이것은 결과적으로 매번 외부에서 입금하는 데이터를 'total = total+ amount;'라는 코드를 이용해서 누적되는 결과를 만들어 냅니다. total이라는 변수는 객체마다 존재하는 데이터를 의미하기 때문에 결과적으로 하나의 저금통에 100원, 500원을 두 번에 나누어 입금하게 되면 전체 금액은 600원이 됩니다. 이제 이 결과를 실행하는 코드도 만들어

서 실행해봅니다.

예제 | PigSaveTest를 이용해서 입금을 테스트해봅니다.

```
public class PigSaveTest {
      public static void main(String[] args) {
            PigSave save = new PigSave();
            save.deposit(100);
            save.deposit(500);
      }
}
```

```
저금통 입금
저금통 입금
```

코드를 실행했더니만 입금되는 로직이 두 번 실행되는 것이 보입니다.

12.4 인스턴스 변수는 객체의 리모컨(레퍼런스)으로 사용할 수 있습니다.

객체의 리모컨(레퍼런스)을 이용해서 가장 많이 하는 작업은 객체의 메소드를 실행하는 것과 객체의 데이터(속성)에 접근하는 작업입니다.

Java에서 어떤 변수에 객체의 리모컨을 담아 두었다면 그 변수와 '.'의 접근 방식을 이용해서 어떤 동작을 시킬 수 있습니다. 물론 이 예제로 가장 좋았던 것은 Scanner를 이용하는 코드였습니다.

예제 |

```
Scanner scanner = new Scanner(System.in);
int a = scanner.nextInt( );
```

이처럼 '.(dot)'은 Java 프로그래밍에서 어디엔가 접근하거나 메소드를 실행할 때 사용하는데, 그 대상은 두 가지입니다.

- 객체가 가진 메소드
- 객체가 가진 데이터

즉, 위에서 실행했던 소스를 봐도 어떤 로직을 실행하는 것을 볼 수 있습니다.

코드

```
PigSave save = new PigSave();
save.deposit(100);
save.deposit(500);
```

객체의 리모컨(레퍼런스)을 이용해서 객체의 데이터와 객체의 메소드를 사용할 수 있습니다.

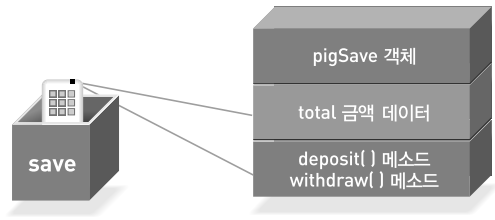


그림 19

그럼 이번에는 객체의 데이터에도 '.'을 이용해서 접근해 볼까요? 코드를 아래와 같이 수정하면 실제 PigSave 클래스에 생산된 객체가 가진 데이터에도 접근할 수 있는 것을 볼 수 있습니다.

코드 '.'를 이용하면 데이터에 접근할 수 있습니다.

```
PigSave save = new PigSave();
save.deposit(100);
save.deposit(500);
System.out.println(save.total); // 직접 데이터에 접근
```

```
저금통 입금
저금통 입금
600
```

12.5 정보 은닉: 객체의 데이터는 함부로 공개하지 않는다.

'.'을 이용했더니 객체가 가진 로직과 데이터에 마음대로 접근해서 참 편하게 사용할 수 있을 듯합니다. 하지만, 절대로 이렇게 프로그램을 작성하면 안 됩니다.

> 객체의 데이터를 마음대로 접근할 수 있다면 메소드를 통해서 만들어진 데이터는 의미가 없게 되는 문제가 생깁니다. 예를 들어 입금하지 않아도 저금통 안의 데이터를 변경할 수 있다면 total 데이터의 값을 사용자 마음대로 조작할 수 있게 되는 문제가 생깁니다.

12.5.1 데이터에 접근해서 금액을 바꾸면 어쩌죠?

지금의 돼지 저금통 프로그램을 생각해 보면 입금이라는 메소드를 통해서만 금액 데이터가 변경 가능해야 합니다. 객체의 데이터를 보호한다는 의미는 이 금액 데이터에 직접 접근하면 안 된다는 겁니다. 예를 들어 다음과 같은 코드가 가능하다는 겁니다.

코드 | 객체의 데이터를 공개하면 누구나 수정할 수 있다는 문제가 있습니다.

```
PigSave save = new PigSave();

save.deposit(100);
save.deposit(500);

//private으로 변경하면 접근할 수 없다.
System.out.println(save.total);

//금액을 몰래 바꾼다.
save.total = 10000;

//입금을 안 해도 금액이 변경된다.
System.out.println(" 변경 "+save.total);
```

```
저금통 입금
저금통 입금
600
변경 10000
```

코드를 보면 원래의 입금한 금액은 100원과 500원이므로 현재 돼지 저금통의 금액은 600원이 되는 것이 올바른 상황입니다. 그러나 '입금(deposit( ))'이라는 기능(메소드)을 굳이 하지 않아도

데이터에 직접 접근이 가능하다면 'save.total = 10000;'과 같은 방식으로 바로 접근해서 데이터를 수정하는 일도 가능합니다. 실제로 이런 일이 발생하면 당연히 안 됩니다. 이처럼 객체에서 데이터는 어떤 메소드의 실행 결과를 누적해서 보존하는 경우가 많기 때문에 함부로 누구나 사용할 수 있게 해주면 안 됩니다.

12.5.2 객체의 데이터는 메소드를 통해 변경해야 합니다.

> 객체지향 프로그래밍의 방식은 필요한 로직이나 기능을 수행할 수 있는 객체에게 내가 원하는 일을 부탁한다(ask)라는 형태로 이루어집니다. 따라서 객체지향에서 객체가 가진 데이터가 필요하다면 객체에 데이터를 알고 싶다고 부탁하는 방식의 프로그래밍이 가장 적합합니다.

돼지 저금통의 금액을 외부에서 마음대로 변경하면 안 된다면 남은 수단은 객체가 가진 기능(메소드)을 통해서 데이터를 조정하는 방법만 남습니다. 객체지향 프로그래밍에서 객체가 가지는 데이터는 앞에서 설명했듯이 호출한 쪽에서 데이터를 보관하지 않도록 하기 위한 장치입니다. 데이터가 외부에 공개되면 데이터의 순수성을 보장할 수 없게 됩니다. 즉 밖으로 한번 나갔다 왔던 데이터는 누군가 조작했을 가능성도 있게 된다는 뜻입니다. 그래서 객체지향 프로그래밍에서 데이터는 감추어 두고 사용해야 한다고 주장합니다. 이것을 '캡슐화(Encapsulation)'나 '정보 은닉'이라는 거창한 용어로 부르는 것뿐입니다.

■ **그래서 객체 안의 데이터는 외부에서 보이지 않게 합니다.**

돼지 저금통에서 데이터를 외부에 공개하지 않도록 하기 위한 가장 좋은 방법은 아예 외부에서 데이터에 접근하는 것을 막는 겁니다. 이때 등장하는 개념이 바로 '접근 제한자(Access Modifier)'라는 개념입니다. 우선은 여기서는 이 정도의 개념만 잡고 나중에 뒤에서 중점적으로 다루도록 합시다. 외부에서 마음대로 접근할 수 없게 코드를 수정해줄 때에는 객체가 가진 데이터 앞에 'private'이라는 단어를 사용하도록 합니다.

12.6 private: 외부에 공개하지 않는다.

private을 이용해서 PigSave의 코드를 수정해보면 외부에서 저금통의 데이터에 접근할 때 에러가 발생하는 것을 볼 수 있습니다.

예제 객체의 데이터는 함부로 접근할 수 없게 보호해 주어야 합니다.

```
public class PigSave {
      private int total; // public에서 변경
      public void deposit(int amount){
            System.out.println("저금통 입금");
            total = total + amount;
      }

      public void withdraw(){
            System.out.println("저금통 배 따기");
      }
}
```

```
public class PigSaveTest {
      public static void main(String[] args) {
            PigSave save = new PigSave();
            save.deposit(100);
            save.deposit(500);
            System.out.println(save.total); // 컴파일 에러
      }
}
```

위의 코드에서 수정된 부분은 PigSave 코드 내의 total이라는 변수 앞에 'private'이라는 키워드가 붙은 것뿐입니다. 코드에서 private으로 변경하자마자 PigSaveTest에서는 에러가 발생하게 됩니다.

> 클래스를 작성할 때 인스턴스 변수를 private으로 작성한다는 의미는 클래스의 { }를 벗어나서는 접근할 수 없게 한다는 뜻으로 해석하면 됩니다. 클래스 선언 시에 사용한 { }의 바깥쪽 코드에서는 private으로 선언된 인스턴스 변수를 사용할 수 없습니다.

객체지향 프로그래밍에서는 데이터의 존재가 여러 개의 클래스로 구분되어 나누어져 있습니다. 따라서 이 데이터를 공개하는 정도의 수준을 '접근 제한'이라고 하는데 가장 기본은 'private'으로 외부에서 접근할 수 없게 합니다.

12.6.1 그럼 객체가 가진 데이터는 어떻게 알아내야 하나요?

여기까지 보면 객체가 가진 데이터를 어떻게 나타내야 할지 참으로 막막해집니다.

"객체를 이용해서 데이터를 보관해 놨더니만 이제는 접근이 안 되네."

객체의 데이터는 특별한 경우가 아닌 이상 모두 private으로 처리해 둡니다. 그리고 외부에서 필요하다면 메소드를 이용해서 알 수 있도록 하는데 이런 메소드를 보통 getter 메소드라고 합니다.

이제 객체에 보관된 데이터에 접근하기 위해서 객체지향의 원칙대로 특정한 메소드를 통해서 데이터를 얻어오게 메소드를 수정해봅니다. 주로 이렇게 객체가 가진 데이터를 접근하는 메소드는 일반적으로 getXXX와 같은 이름을 가지는 것이 보통입니다. 즉 PigSave에 getTotal( )이라는 메소드를 통해서 객체가 가진 데이터를 가져오게 해주는 것이 좋습니다.

예제 total 데이터를 알려주는 getTotal( ) 메소드를 추가한 PigSave

```
public class PigSave {
	private int total;
	//public int total;
	public int getTotal(){
		return total;
	}

	public void deposit(int amount){
		System.out.println("저금통 입금");
		total = total + amount;
	}

	public void withdraw(){
		System.out.println("저금통 배 따기");
		total = 0;
	}
}
```

이제 메소드를 통해서 total 데이터를 얻도록 해 보면 다음과 같이 코드를 작성할 수 있습니다. 가장 주의해서 볼 것은 save.total이라는 코드를 save.getTotal( ); 이라는 것으로 수정한 곳입니다.

예제 | 객체가 가진 데이터는 메소드를 통해서 사용합니다.

```
public class PigSaveTest {
      public static void main(String[] args) {
            PigSave save = new PigSave();
            save.deposit(100);
            save.deposit(500);
            int totalMoney = save.getTotal(); // get 메소드를 통한 접근
            System.out.println("총액: " + totalMoney);
      }
}
```

```
저금통 입금
저금통 입금
총액: 600
```

12.6.2 변수에 접근하는 대신에 굳이 get 메소드를 쓰는 이유

실행된 결과를 보면 "뭐야? 아무런 차이가 없잖아!"라고 생각하실 수 있습니다. 굳이 getTotal()(get 메소드)를 사용했을 뿐 결과는 동일합니다. 하지만, 정말 그럴까요? 정말 그렇다면 굳이 이런 방식으로 private으로 만들고 메소드를 작성하지 않아도 되는 것 아닐까요? Java를 공부하면서 이런 의문을 가지는 것은 지극히 정상적입니다. 그럼 이제 같지 않다는 것을 설명하도록 하겠습니다.

■ **return한 결과를 받는 작업은 또다시 복사입니다.**

코드를 보면서 설명합니다.

코드 |

```
//PigSave의 코드
private int total;

public int getTotal(){
      return total;
}
//PigSaveTest의 코드
int totalMoney = save.getTotal(); // total 변수의 값을 totalMoney 변수로 복사

System.out.println("총액: " + totalMoney);
```

당연한 얘기지만 save.getTotal( )을 호출하면 PigSave 객체는 자신이 가진 total 데이터를 반환합니다. 그런데 코드를 보면 'int totalMoney = getTotal( )'이 됩니다. 그럼 위의 코드는 'int totalMoney = 객체의 total;'이 성립할 수 있습니다. 이 성립에서 가장 중요한 것은 Java에서의 변수 할당은 '복사(Copy)'라는 개념입니다. 데이터에 직접 접근하지 않고 데이터를 복사하기 때문에 원본은 안전해집니다.

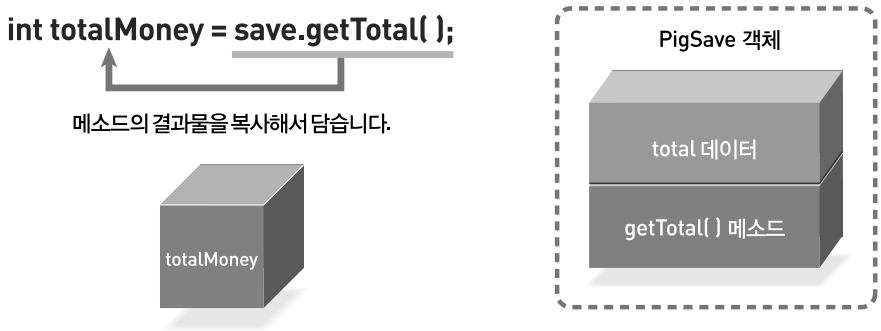


그림 20

즉 int totalMoney 변수는 객체의 진짜 객체의 total 값 자체를 의미하는 것이 아니라, 객체의 totalMoney 데이터를 복사해서 int totalMoney라는 자신의 변수로 복사하게 됩니다. 이 복사라는 사실 때문에 재밌는 현상이 나타나게 되는데 메소드의 결과로 받은 변수는 복사본 데이터일 뿐 절대 원본 데이터가 아니라는 겁니다. 조금 어려울지도 모르니 예를 들어서 설명해봅니다.

- 우선 외부에서는 PigSave 객체가 가진 total이라는 데이터에 직접 접근할 수 없다.
- 외부에서는 getTotal( )이라는 메소드를 실행하고 그 결과를 변수 totalMoney에 담는다.
- 이때 PigSave 객체의 데이터는 totalMoney에 복사된다.
- totalMoney 변수는 복사본 데이터만을 가지고 있으므로 만일 totalMoney 값을 변경하면 복사한 데이터가 변하는 것이지 PigSave 객체의 데이터가 변경되는 것은 아니다.

백문이 불여일견입니다. 실제로 변경해 본 후에 다시 PigSave 객체의 getTotal( )을 호출해보면 알 수 있습니다.

코드 | 객체의 데이터에 직접 접근 못 하게 하면 데이터의 안정성을 보장할 수 있습니다.

```
PigSave save = new PigSave();

save.deposit(100);
save.deposit(500);

int totalMoney = save.getTotal(); // total을 복사해서 totalMoney에 저장

System.out.println("총액: " + totalMoney);

//강제로 변경
totalMoney = 10000; // 저장된 데이터만 수정되고 원래 데이터는 안전

//PigSave도 변했을까?
int afterMoney = save.getTotal();

System.out.println("변경 되었나?" + afterMoney);
```

```
저금통 입금
저금통 입금
총액: 600
변경 되었나? 600
```

결과를 보면 메소드를 통해서 데이터를 받았더니만 실제 PigSave 객체의 데이터는 변경되지 않고 복사본 데이터만 변경되는 것을 알 수 있습니다.

12.7 객체의 데이터는 반드시 안전하게!

지금까지 여러분에게 설명한 내용은 객체가 데이터를 가질 수 있는데 이 데이터는 메소드를 통해서 접근하도록 하고, 가능하면 외부에는 공개하지 않는다는 겁니다. 객체지향에서는 어차피 모든 데이터는 필요하다면 객체로 만들어서 보관합니다. 그리고 이런 객체 안에 보관하는 데이터는 안전하게 보관해야 하기 때문에 외부에서 사용할 수 없게 private이라는 키워드를 이용해서 처리해 줍니다.

■ 데이터를 얻는 메소드는 getXXX( )로 만들어 줍니다.

메소드를 통해서 데이터를 얻게 되면 데이터를 안전하게 보관할 수 있습니다. 따라서 이런 메소드를 일반적인 getter라고 하는데, 말 그대로 '데이터를 get'하는 기능을 한다는 뜻으로 보시면 됩니다.

■ **그럼 메소드를 통해서 데이터를 변경한다면 setter?**

getXXX으로 시작하는 메소드가 주로 데이터를 얻어오는 기능을 한다면 데이터를 바꿀 수 있게 해줄 때에도 역시 메소드를 이용합니다. 이런 메소드를 setter 메소드라고 하는데 메소드를 통해서만 데이터에 접근하게 합니다. setter 메소드는 신중하게 생각해야 합니다. setter를 이용해서 데이터를 변경할 수 있기 때문에 반드시 필요한 경우에만 작성해주도록 합니다. 이런 예제들은 좀 더 뒤쪽에서 객체지향 방식으로 프로그램을 설계할 때 자주 보게 될 겁니다.

■ **자동 코드 생성 기능: 이클립스가 도와줍니다.**

이클립스는 여러분의 이런 작업을 아주 효과적으로 도와줍니다. 간단히 말해서 getter, setter 메소드를 자동으로 만들어주는 기능을 가지고 있습니다. 따라서 필요하다면 모든 데이터는 private으로 선언하고 필요한 getter나 setter 메소드는 버튼만 클릭해주면 됩니다. 이 방법 역시 조금 뒤에서 객체 설계 요령을 공부할 때 사용해보도록 합니다.

13 나만의 클래스, 객체를 만드는 순서

객체지향 프로그래밍을 배우는 작업을 어려워하는 가장 큰 이유는, 남들이 만든 소스는 보면 알 수 있지만, 자신이 어떻게 만들어야 하는지는 잘 모르기 때문입니다. 따라서 자신만의 클래스를 만드는 순서와 기준을 명확히 잡아두어야만 합니다. 이것을 보다 발전시키면 자신만의 분석과 설계 방식이 자리 잡게 됩니다.

새로운 클래스를 만들기 전에 앞에서 구성한 저금통의 소스를 보는 것에서 시작할까 합니다. 이렇게 하는 이유는 여러분이 우선은 완성된 소스를 보면서 정확한 이해를 할 수 있어야만 유사한 프로그램을 만들어 낼 수 있기 때문입니다.

예제 | PigSave의 소스 코드

```
public class PigSave {
	private int total;

	public int getTotal(){
		return total;
	}

	public void deposit(int amount){
		System.out.println("저금통 입금");
		total = total + amount;
	}

	public void withdraw(){
		System.out.println("저금통 배 따기");
		total = 0;
	}
}
```

클래스를 만드는 순서는 다음과 같이 작업하는 것이 일반적입니다.

① 먼저 이름을 결정하고 클래스를 선언한다.
② 원하는 메소드를 선언해둔다.
③ 보관되어야 하는 데이터를 인스턴스 변수로 선언한다.
④ 메소드를 구현해 가면서 인스턴스 변수를 추가하거나 수정한다.
⑤ main 메소드를 이용해서 작업한 내용을 테스트한다.

13.1 우선은 PigSave 클래스를 선언하는 작업이 시작입니다.

객체지향 프로그래밍에서는 데이터와 로직을 처리하기 위해서 객체라는 독특한 존재를 사용하는 방식입니다. 그리고 이 객체라는 것은 클래스(Class)의 복사본이기도 합니다. 따라서 여러분이 어떤 객체를 만들고 싶다면 가장 먼저 할 일은 클래스를 선언하는 일입니다. 클래스는 데이터와 로직을 묶어둔 단위라고 생각하면 됩니다. 클래스는 주로 다음과 같은 기준을 가지는 것이 일반적입니다.

• **클래스의 이름은 주로 명사인 경우가 많다.**
  – 우리가 어떤 사물을 칭할 때는 주로 명사의 이름을 쓰는 경우가 많습니다. 따라서 연극에 나오는 것도 명사로 나오는 경우가 많습니다. 예를 들어 회원정보, 주문, 상품 등과 같은 이름들이 사용될 수 있습니다.

• **클래스의 이름은 주로 역할이나 직업으로 구분되는 경우가 많다.**
  – 클래스는 어떤 기능들을 가지고 있기 때문에 하나의 직업에 비유되는 경우가 많습니다. 요리사, 건축가, 관리자 등의 이름들이 사용될 수 있습니다.

• **클래스의 이름은 실생활의 이름을 쓰는 경우도 많다.**
  – 클래스 이름은 적당하다면 실생활의 이름을 그대로 쓰는 경우도 많습니다. TV, 계산기, MP3 플레이어 등과 같은 이름을 사용할 수 있습니다.

클래스를 선언할 때 가장 좋은 습관은 바로 하나의 그림으로 표현하는 겁니다.

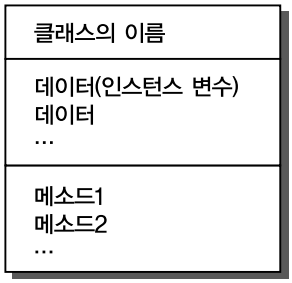


그림 21

거의 모든 객체지향 설계에서 위와 같은 그림을 이용해서 클래스의 구조를 표현합니다.

13.2 클래스가 아닌 객체라 생각하면서 원하는 메소드를 정합니다.

어떤 클래스를 만들어야 할지 결정했다면 다음에 해야 하는 일은 이제 이런 클래스가 어떤 로직이나 기능을 가질 것인지를 결정하는 겁니다. 물론 이것은 문법적으로는 메소드라는 것으로 형상화됩니다. 클래스를 표현해 둔 그림에 아래쪽의 영역에 만들려고 하는 클래스가 가져야 하는 기능을 작성합니다. 이때 가능하면 클래스라고 단정하지 마시고, 그냥 만들어질 객체라고 간주하는 것이 생각하기에 조금 더 편리할 겁니다. 지금의 경우라면 '저금통'이 실제 있다고 가정하고 저금통이 가져야 하는 기능만을 생각하는 겁니다. 저금통에 금액 데이터를 추가하는 기능과 금액 데

이터를 초기화하는 기능이 있다면 그 기능을 그림에 추가해줍니다. 다이어그램의 아래쪽은 메소드를 기술하는 영역입니다.

객체가 가져야 하는 기능(메소드)을 고민할 때 한 가지 팁을 드리자면 가능하면 문장의 주어를 만들려고 하는 객체로 시작하라는 겁니다. "돼지 저금통은 xxx를 할 수 있다."라는 문장을 만들어 두면 현재 만들려고 하는 클래스 안에 어떤 메소드가 선언되어야 할지를 조금 더 쉽게 결정할 수 있습니다.

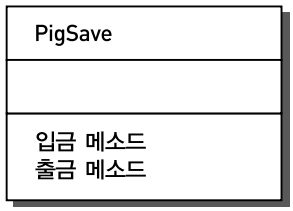


그림 22

13.3 로직이나 기능은 무조건 'public void'로 시작하는 메소드로 선언해두고 시작합니다.

이제 기능을 결정했다면 이제는 메소드를 구현하는 단계에 들어갑니다. 이때 메소드의 선언을 우선 'public void'로 시작하는 메소드로 작성하도록 합니다.

메소드를 작업하는 순서

① 먼저 이름을 결정하고 무조건 'public void 이름( ) { }'로 코드를 작성해둡니다.
② 메소드의 파라미터를 결정해봅니다.
③ 메소드의 리턴 타입을 결정해봅니다.

예제

```
public class PigSave {
      public void deposit(){

      }

      public void withdraw(){

      }
}
```

메소드를 구현할 때 너무 많은 생각을 하게 되면 점점 더 이상한 방식의 코드를 작성하게 됩니다. 메소드의 파라미터와 리턴 타입은 생각하지 말고 우선은 메소드를 선언하는 일에만 집중하는 것이 좋습니다.

■ **메소드를 작성할 때에는 파라미터를 먼저 판단합니다.**

메소드를 선언해 두었다면 그다음에 생각할 것은 이 메소드를 실행하기 위한 필수 데이터인 파라미터(Parameter)입니다. 이때에는 아주 단순한 판단 기준을 하나 가지시면 좋습니다.

메소드 파라미터의 판단 기준

- 메소드를 실행할 때마다 매번 다르게 들어오는 데이터는 파라미터로 선언합니다.
- 메소드를 실행할 때 필수적인 데이터는 메소드의 파라미터로 선언합니다.

■ **메소드를 실행할 때마다 필요한 데이터가 달라진다면 무조건 파라미터입니다.**

저금통의 deposit( ) 메소드를 봅시다.

예제

```
public void deposit(int amount){
      System.out.println("저금통 입금");
      total = total + amount;
}
```

deposit( ) 메소드는 입금 로직을 구현한 메소드입니다. 그런데 곰곰이 생각해 보면 입금할 때마다 데이터가 일정하지 않다는 것을 알 수 있습니다. 때로는 100원, 때로는 500원처럼 실행될 때마다 다른 금액을 처리해야 한다는 겁니다. 따라서 이런 경우에는 데이터를 파라미터로 받아들이게 해야 합니다.

■ **즉각적으로 피드백해주기로 했다면 리턴을 이용합니다.**

파라미터를 결정했다면 그다음은 메소드의 실행 후에 즉시 어떤 결과를 알려주는 것인지를 판단하는 작업입니다. 이 판단이 바로 리턴 타입이 됩니다.

메소드의 리턴 값 판단 기준

- 메소드의 실행 결과를 즉각적으로 피드백해 주어야 한다면 void 대신에 반환할 타입을 명시해줍니다.
- 메소드의 실행 결과를 보관할 때 메소드는 void로 해두고, 인스턴스 변수를 활용해서 데이터를 보관하도록 합니다.

13.4 지역 변수(Local Variables): 때로는 메소드 내부에서 변수를 사용할 때도 있습니다.

여러분이 메소드를 구현하다 보면 실제로 메소드의 내부를 구현하기 위해서 가끔은 변수를 사용할 일이 있을 겁니다. 이런 변수를 지역 변수(Local Variables)라고 합니다. 지역 변수는 메소드의 안쪽에 선언되는 변수입니다. 개인적으로 지역 변수라는 말보다는 좀 더 한국식으로 이름을 붙이자면 '임시 변수', '일회성 변수', '휘발성 변수'라는 말이 더 좋을 듯합니다. 제가 왜 이런 이름을 사용하는지 설명해 보도록 하겠습니다.

지역 변수(Local Variables)라는 것은 메소드 안에서 메모리상에 잠깐동안 만들어져 사용되는 변수를 의미합니다. 변수의 선언이 메소드의 { }안으로 국한되기 때문에 메소드 내에서만 효력이 있는 임시적인 변수입니다.

■ 지역 변수는 매번 실행할 때마다 만들어지는 일회성 변수

메소드 안쪽에 선언되는 변수는 메소드가 실행될 때 잠깐씩 필요한 데이터를 담아두는 변수입니다. 말 그대로 메소드를 호출할 때마다 메소드가 실행되는 동안에만 잠깐씩 사용되는 변수라는 겁니다. 변수를 최초로 선언할 때만 변수의 앞에 타입이 붙는다는 사실을 기억해 보시기 바랍니다. 즉, 메소드 안쪽에 선언되는 변수는 메소드가 실행되는 순간에 매번 임시로 만들어져서 메소드가 실행되는 동안에만 그 데이터를 유지했다가 메소드의 실행이 끝나면 버려지는 존재입니다. 따라서 메소드를 여러 번 실행한다는 의미는 메소드 안쪽에 있는 변수가 여러 번 만들어진다는 의미입니다. 이런 의미로 생각해보면 메소드 안쪽의 선언되는 변수는 '임시적이고, 일회적'이라 할 수 있습니다.

■ 지역 변수는 반드시 사용되기 전에 초기화되어야 합니다.

인스턴스 변수가 객체를 대표하는 좋은 위치에 오른 변수라면, 지역 변수는 일회용 변수라고 할 수 있습니다. 그냥 한번 쓰이고 버려지는 신세입니다. 게다가 지역 변수는 발생할 수 있는 모든

문제를 막고자 아예 사용하기 전에는 반드시 초기화를 해야만 하는 변수입니다. 이것은 인스턴스 변수에 비유하자면 엄청나게 차별적인 내용입니다. 인스턴스 변수는 메소드 등을 통해서 객체가 살아있는 동안 스스로 진화하면서 발전할 수 있는 데이터입니다. 메소드의 실행 결과에 따라 영향을 받기도 하고 주기도 하니까요.

반면에 매번 메소드를 실행할 때마다 필요한 임시적인 데이터를 담는 변수는 다음에 실행할 때에는 그 이전에 실행한 결과가 데이터로 남아 있으면 안 됩니다. 메소드가 동일한 상태에서 마찬가지로 실행되는 것을 보장하려면 메소드가 사용하는 임시 데이터도 동일한 상황이어야 하는 겁니다. 이런 상황을 보장해 주고자 메소드 안에 선언되는 변수는 사용하기 전에 반드시 초기화를 시켜줍니다.

13.5 인스턴스 변수

■ 유지되는 객체의 데이터

만일 여러분이 어떤 메소드를 실행했는데 그 결과를 즉시 알려줄 필요가 없거나, 결과를 보관해 두는 것이 좋다고 판단했다면 그것은 문법에서는 인스턴스 변수로 선언됩니다.

인스턴스 변수	지역 변수
객체의 데이터	메소드 실행 시 잠깐 필요한 임시 데이터
객체마다 다른 값을 가질 수 있으므로 선언만 해도 된다.	매번 실행 시마다 같은 값으로 실행되기 때문에 변수 선언 시에 초기화한다.
주로 private로 보호	메소드의 { }이므로 외부에서 사용 자체가 불가능

표 인스턴스 변수와 지역 변수의 비교

인스턴스 변수는 원래의 의미가 '로직에서 분리되어서 보관되는 데이터'라고 볼 수 있습니다. 그리고 이 데이터라는 것을 예전에 함수라고 말하던, 지금은 우리가 메소드라고 하는 기능을 통해서 만들어진 데이터입니다. 따라서 인스턴스 변수는 메소드에서 만들어진 결과 데이터를 지속적으로 보관한다고 생각해야 합니다.

예제 | 인스턴스 변수는 결과를 누적합니다.

```
public class PigSave {
      private int total;
      public void deposit(int amount){
            System.out.println("저금통 입금");
            total = total + amount;
      }
```

위의 소스를 보시면 deposit( )이라는 동작을 했는데 즉각적인 피드백은 하지 않았습니다. 그런데 이 데이터는 저금통의 원리처럼 계속해서 누적되는 데이터입니다. 따라서 두 번 입금(deposit( )) 하고, 그 결과를 누적해서 보관하기 위해서 인스턴스 변수가 쓰인 것을 볼 수 있습니다.

인스턴스 변수의 판단 기준

- 객체마다 보관해야 하는 데이터는 무조건 인스턴스 변수
- 메소드의 실행 결과를 누적해서 보관하는 경우
- 동일한 메소드를 여러 번 실행할 때마다 분기(if ~ else)의 기준이 되는 데이터
- 메소드끼리 공유하는 데이터

■ 메소드 실행 기준

때로는 같은 메소드를 실행해도 다른 결과가 나오는 경우가 있습니다. 예를 들자면 액션 슈팅 게임을 하는데 매번 폭탄 버튼을 누른다고 해서 항상 폭탄이 나가는 것은 아닌 것처럼 말입니다. 일반적으로 비행기 게임들은 각 유닛이 총알을 쏘고, 폭탄을 터트리는 기능을 가지고 있습니다. 만일 비행기 게임에 나오는 비행기를 소스로 만든다면 다음과 같은 형태가 될 겁니다.

예제 | 총알을 쏘고, 폭탄을 터트리는 Airplane을 위한 클래스

```
public class Airplane {
      public void shootBullet(){
            System.out.println("총알을 쏩니다.");
      }
      public void useBomb(){
            System.out.println("폭탄을 터트립니다.");
      }
}
```

액션슈팅 게임에서는 아시다시피 폭탄이라는 게 아주 유용합니다. 위급한 순간에 한 번씩 터트리면 주변이 깨끗해지니까요. 그럼 폭탄을 계속 터트리면 아주 쉽게 게임을 할 수 있지 않을까요? 하지만, 폭탄의 개수가 정해져 있다는 사실을 알고 계실 겁니다. 그렇다면, 비행기가 useBomb( ) 을 호출할 때 폭탄이 없다면 아무런 효과가 없어야만 합니다. 이것을 어떻게 표현할까요?

게임에서 비행기는 일반적으로 3개의 폭탄을 가지는데 이것은 이 비행기가 살아 있는 동안 계속 유지되는 데이터라는 의미입니다. 휴대전화라면 우리가 그 휴대전화를 사용하는 동안에 번호가 변경되지 않는 것과 같은 원리입니다. 객체가 가진 전 재산이 데이터와 기능이라는 사실을 생각해 보면 비행기 게임에서는 폭탄을 터트릴수록 사용 가능한 폭탄의 개수가 줄어드는 사실을 알고 있습니다. 따라서 이런 경우는 폭탄을 사용할수록 객체가 가진 데이터가 변경되고, 만일 폭탄의 개수가 0이라면 더 이상은 폭탄 사용 버튼을 눌러도 폭탄이 나가지 않게끔 됩니다.

인스턴스 변수와 메소드는 클래스 안에서 들여쓰기하는 레벨이 동일합니다. 쉽게 생각하시려면 레벨이 동일하면 서로 영향을 줄 수 있다고 생각하시면 됩니다. 그럼 Airplane에다가 비행기가 가진 폭탄의 수를 3으로 해서 인스턴스 변수를 선언할까 합니다. 인스턴스 변수를 'int bombCount = 3;'으로 클래스 안에 선언하게 되면 클래스의 모든 복사본인 객체들은 생성될 때 3개의 폭탄을 가지고 만들어지게 됩니다.

코드

```
public class Airplane {
      int bombCount = 3;
      //이하 생략
```

이제 useBomb이라는 기능을 실행하게 되면 사용 가능한 폭탄의 수가 점점 줄어들어야 하고, 폭탄의 수가 0이 되면 더는 폭탄을 사용할 수 없도록 변경되어야 합니다. 로직으로 설계하면 useBomb( ) 메소드는 다음과 같습니다.

> **만일 폭탄의 수가 0보다 크다면 "폭탄을 터트립니다."를 출력하고 남은 폭탄의 수도 출력한다.**
> **폭탄 수가 0이라면 "폭탄이 없네요."를 출력한다.**

이것을 useBomb( ) 안쪽에 구현해보도록 합니다.

예제 인스턴스 변수와 상호작용하는 useBomb( ) 메소드

```
public class Airplane {

	int bombCount = 3;

	public void shootBullet(){
		System.out.println("총알을 쏩니다.");
	}

	public void useBomb(){
		if(bombCount > 0){
			System.out.println("폭탄을 터트립니다.");
			//폭탄을 사용했으므로 사용할 수 있는 폭탄 수가 줄어야 한다.
			bombCount--;
			System.out.println("남은 폭탄의 수" +bombCount);
		}else{
			System.out.println("폭탄이 없네요.");
		} // end if
	}
}
```

객체 지향 프로그래밍에서의 데이터의 보관이라는 것이 개발자들에게는 로직의 간소화를 가져옵니다. 여러분이 휴대전화로 메시지를 보낼 때 발신자 번호를 매번 일일이 지정하는 방식이 함수형 언어를 사용하는 방식이라고 비유하자면 객체지향 프로그래밍에서는 휴대전화가 자신의 번호를 저장하고 있기 때문에 그만큼 작업이 수월하게 처리되는 원리와 같습니다.

■ **인스턴스 변수는 객체의 상태, 속성이라고 합니다.**

클래스에 선언하는 인스턴스 변수는 정리해 보면 주로 다음과 같은 특징이 있다고 할 수 있습니다.

- 객체마다 데이터의 값이 다르게 유지된다: 저금통이 여러 개라면 저금통마다 금액의 값은 다르다는 겁니다.
- 메소드의 결과를 누적한다: 입금하면 누적된 결과로 사용됩니다. 이 때문에 누적된 결과를 다른 곳에서 굳이 보관할 필요가 없게 됩니다.
- 메소드의 판단기준이 될 수도 있다: 비행기 게임에서처럼 폭탄의 수가 메소드를 실행할 때의 판단기준이 됩니다.

그리고 여기에 한 가지 기준을 굳이 더 추가하자면 다음과 같습니다.

- 메소드끼리 공유하는 데이터: 저금통의 경우 입금하면 금액 데이터가 증가하고, 출금하면 금액 데이터가 없어지는 경우

인스턴스 변수는 이런 이유 때문에 '객체의 상태(State)' 혹은 '객체의 속성(Property)'이라고 표현합니다.

14 복불복 기계 만들기

요즘 TV를 보면 '복불복'이라는 형태의 게임이 있더군요. 이 게임은 아주 단순합니다. 여러 개의 선택 항목 중의 하나는 아주 불행한 결과를 만들어내는 겁니다. 그럼 이것을 프로그램으로 작성해보도록 하겠습니다.

14.1 복불복 기계 설계하기

종이 하나에 '복불복 기계'를 설계해봅시다. 우선은 사각형 하나에 'BokBulBok'이라는 이름으로 작성해보도록 합니다.

■ **복불복 기계의 메소드 설계하기**

복불복 기계가 어떤 기능이나 로직을 가질 것인지 생각해봅시다. 이때에는 가상의 상자를 하나 앞에 두면 참 편리합니다. 간단히 기계가 어떨지 생각해볼까요?

- 우선 기계 안에는 하나의 '잘못된 데이터'와 다수의 '괜찮은 데이터'가 있을 것이다.
- 한 명씩 기계에서 뽑아내면 누군가 한 명은 '잘못된 데이터'를 결과로 얻게 된다.

아마도 가장 중요한 기능은 위와 같이 되지 않을까 싶습니다. 이제 종이에 이 기능을 적어두고 소스 코드를 작성하기 시작합니다.

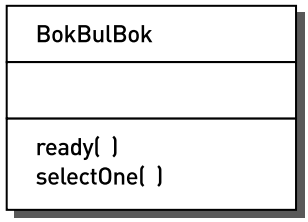


그림 23 복불복 기계 설계

예제 | BokBulBok 클래스의 선언과 메소드의 선언 형태

```
public class BokBulBok {

      public void ready(){

      }

      public void selectOne(){

      }
}
```

14.2 각각의 메소드를 구현하면서 소스를 수정하기

이제 다음 단계는 실행되는 모습을 그려보면서 데이터가 있는지를 확인하는 단계입니다. 지금의 상황이라면 복불복 재료를 준비하는 ready( )라는 메소드가 실행되면, 그 재료들을 selectOne( )에서 사용하는 시나리오가 나옵니다. 그렇다면, 이렇게 '실행 결과를 누적하고, 메소드 간의 공유되는 데이터'라는 기준으로 보면 인스턴스 변수로 복불복 재료들을 선언하는 것이 맞을 듯합니다.

■ **결정된 데이터가 '~들'인 경우라면 무조건 배열이나 자료구조라고 생각하시면 됩니다.**

배열은 '이름을 가지지 못한 변수들의 탑'입니다. 그러니 여러분이 데이터를 생각할 때, 여러 개의 데이터가 '~들'로 나온다면 지금의 상태에서는 배열을 생각하시는 것이 좋습니다. 간단하게 10개의 종이에 'O'가 9개이고, 'X'가 하나라고 생각해 보도록 하겠습니다. ready( )에서는 임의의 순서에 'X'를 가지게 하고, 나머지는 'O'를 가지는 형태로 구현하게 될 겁니다. 그렇다면, 아마도 내부는 다음과 같은 모습으로 만들어질 겁니다.

예제 | 복불복 기계 안의 10개 글자 중 임의의 글자 하나에 'X' 표시를 합니다.

```
public class BokBulBok {

	private char[] items;

	public void ready(){

		items = new char[10];
		//복불복 당첨 인덱스 번호
		int index = (int)(Math.random()* items.length);

		for(int i = 0; i < items.length ; i++){

			if(i == index){
				items[i] = 'X';

			}else{
				items[i] = 'O';
			}//end if
		}//end for


	public void selectOne(){

	}
}
```

메소드의 실행 결과를 저장하기 위해서, 그리고 selectOne( ) 메소드가 이 데이터를 활용할 것이므로 소스는 위와 같은 형태로 변경됩니다. 다만, 좀 맘에 안 드는 부분은 if ~ else를 쓰다 보니 { }가 많아져서 보기 좋지 않습니다. 이런 식으로 변경해도 좋을 듯합니다.

코드 |

```
for(int i = 0; i < items.length ; i++){

	if(i == index){
		items[i] = 'X';
		continue;
	}
	items[i] = 'O';
}//end for
```

다시 루프의 위로 올라가는 continue를 이용하면 조금 더 소스가 간결해집니다. ready( )라는 메소드와는 달리 selectOne( )은 이제 사용자들이 한 명씩 순서대로 상자의 내용물을 뽑는 것으로 볼 수 있습니다. 물론 TV에서는 자기가 원하는 번호를 골라서 뽑기는 하지만, 지금은 그냥 순서대로 뽑는다고 가정하고 작성해보도록 하겠습니다. select( )의 구현을 파라미터나 리턴 타입으로 생각해 보면 다음과 같은 특징이 있음을 알 수 있습니다.

- 순서대로 뽑는다면 굳이 뽑는 사람이 순서를 지정하지 않을 것이다. 그러니 메소드를 실행할 때 특별히 필요한 파라미터는 필요 없을 듯하다.
- 하나를 뽑으면 뽑은 사람은 결과가 무척이나 궁금할 듯하다. 그러니 결과는 알려주는 것이 좋겠다.

다만, 구현하는 데 있어서 한 가지 여러분이 더 생각해야 하는 부분은 매번 selectOne( ) 메소드를 실행하게 되면 items에서 다음 순서의 아이템이 나와야 한다는 겁니다. 이런 식으로 구성하게 된다면 결과적으로 동일한 메소드가 실행될 때마다 어떤 기준에 의해 다르게 동작하는 인스턴스 변수에 해당하는 데이터가 하나 더 필요할 듯합니다. 보통 이런 순서들은 인덱스 번호라고 합니다.

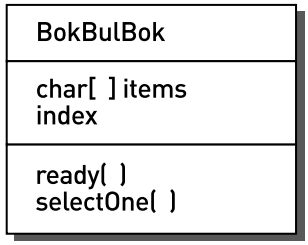


그림 24 복불복 기계의 설계

예제 | selectOne( )의 구현

```
public class BokBulBok {

	private char[] items;

	private int nextIndex = 0;

	public void ready(){

		items = new char[10];
		//복불복 당첨 인덱스 번호
		int index = (int)(Math.random()* items.length);

		for(int i = 0; i < items.length ; i++){

			if(i ==  index){
				items[i] = 'X';
				continue;
			}
			items[i] = 'O';
		}//end for
	}

	public char selectOne(){
		return items[nextIndex++];
	}
}
```

인스턴스 변수는 가끔은 위와 같이 메소드를 구현하다가 발견하기도 합니다. 프로그래밍에 대한 경험이 없을수록 처음에 모든 것을 완벽하게 결정하려는 경향이 있는데, 코드를 변경하는 것은 사람이 완벽할 수 없기 때문에 너무나 자연스러운 현상입니다.

14.3 클래스와 객체의 테스트를 진행해봅니다.

여러분이 어떤 클래스를 만들었다면 가능하면 반드시 어떤 형태로든 메소드가 정상적으로 동작하는지를 확인하는 것이 좋습니다.

예제 | 복불복의 테스트를 위해 작성한 별도의 클래스와 main 메소드

```
public class BokBulBokTest {

	public static void main(String[] args) {

		BokBulBok bok = new BokBulBok();
		//복불복 기계 안에 준비작업
		bok.ready();

		//10번 결과를 확인해 본다.
		//그중에 한번은 X가 나와야 한다.
		for(int i = 0; i < 10 ; i++){

			char result = bok.selectOne();
			System.out.println(i+"번째: " + result);
		}
	}
}
```

```
0번째: O ← 결과는 매번 달라질 수 있습니다.
1번째: O
2번째: O
3번째: X
4번째: O
5번째: O
6번째: O
7번째: O
8번째: O
9번째: O
```

물론 결과는 임의의 순번을 뽑기 때문에 매번 실행할 때마다 다르게 나올 수 있습니다.

14.4 화면은 나중에 생각해봅니다.

객체지향 프로그래밍을 할 때 처음 배우는 사람들의 나쁜 버릇 중의 하나는 '클래스를 작성할 때 프로그램이 어떻게 실행되는지를 소스에 넣으려고' 하는 행위입니다. 예를 들어서 설명하자면 여러분은 위의 클래스를 이용해서 친구와 게임을 만들 수도 있습니다. 키보드를 이용해서 입력하고 그 결과를 BokBulBok 기계를 이용해서 동작시키게 하는 겁니다. 이런 생각에서 프로그램을 작성하면 가장 먼저 작성하는 코드는 사실 제가 작성한 코드와 전혀 다르게 나옵니다. 이것은 프로

그래밍 경력이나 실력과는 아무런 관계가 없습니다. 왜냐하면, 화면이 어떻게 실행되는지를 중심으로 생각하다 보니 그러한 기능을 메소드로 만들려고 생각하기 때문에 발생하는 문제니까요. 따라서 개인적으로 이런 훈련을 할 때 가장 추천하고 싶은 방법은 웹이나 특정 프로그램이 우리가 원하는 화면을 가지고 있다고 생각하는 겁니다. 예를 들어 지금의 경우라면 아주 간단히 다음과 같은 화면이 있으면 될 것입니다.

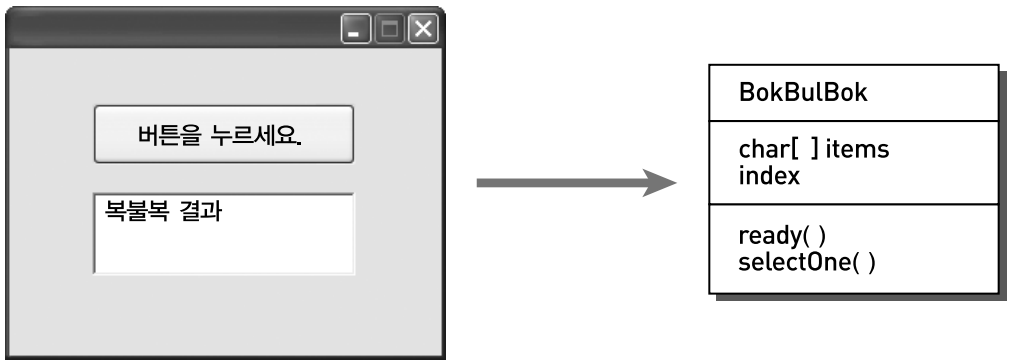


그림 25 화면이 있다고 가정하고 클래스를 설계

이 그림 하나를 그린다고 해서 시간이 오래 걸리는 것이 아닙니다. 시간이 많이 안 걸리는 일은 많이 해보고, 시간이 오래 걸리는 일은 가능하면 짧게 하는 것이 지혜라고 생각합니다.

15 객체의 데이터란?

객체지향 프로그래밍 이전의 방식에서는 프로그래밍이라는 것이 그림에서처럼 로직과 로직 사이에 데이터가 흐르는 형태로 작성되는 것이 일반적이었습니다. 그러나 객체지향 프로그래밍에서는 이제 데이터 자체를 객체가 가지고 있기 때문에 데이터가 로직 사이를 흐르는 것이 아니라, 객체와 객체 간의 의사소통을 통해서 어떤 작업이 일어나는 구조로 인식하게 됩니다. 따라서 객체지향 프로그램을 제대로 구성한다는 것은 결과적으로 어떤 객체들이 필요하고, 어떻게 이 객체들과의 관계를 정해주는 것이 거의 전부입니다. 그리고 이 작업을 위해서는 올바른 클래스를 만들어 내는 몇 가지의 추가적인 노하우가 필요합니다.

인스턴스 변수와 메소드, 그리고 클래스와 객체를 이해했다면 이제 생성자(Constructor)라는 표현법을 익히면 클래스에서 객체를 만들어 낼 때 조금 더 다양한 방식을 사용할 수 있게 됩니다.

■ **객체(Object)는 결과적으로 데이터를 개별적으로 보관하는 장치**

객체는 '데이터와 로직'으로 구성됩니다. 데이터는 보관되어야 하는 데이터를 의미합니다. 이 데이터는 때로는 어떤 로직의 결과일 수도 있고, 어떤 로직을 수행하기 위해서 사용자가 입력한 파라미터일 수도 있으며, 어떤 로직을 수행하기 위해서 적절한 판단기준이 되는 데이터일 수도 있습니다. 이 때문에 객체가 가진 데이터와 로직은 처음부터 어떻게 사용하게 할 것인가를 제어할 수 있는 방법도 제공되는데 이런 장치를 '생성자'라고 합니다.

결국 '생성자'라는 것은 객체의 데이터나 로직을 처리하는 약간의 변형된 방식입니다. 생성자를 이용하면 객체 생성 시에 어떤 제약(반드시 필요한 데이터를 지정한다든가)을 줄 수 있고, 객체가 생성되자마자 어떤 기능을 수행하게 할 수 있습니다.

본격적으로 생성자에 대해 공부하기 전에 우선 객체에 대해서 조금만 더 생각해보기로 합니다.

■ **객체는 데이터를 보관할 수 있다? 데이터가 필요하지 않다면?**

클래스에서 객체를 생산한다는 말은 객체마다 다른 데이터를 가지게 하는 경우라고 할 수 있습니다. 그렇다면, 우리가 개별적으로 보관할 필요가 없다면 굳이 클래스에서 객체를 생산하는 작업을 하지 않아도 괜찮다는 얘기가 성립됩니다. 데이터를 보관하지 않는 아주 단순 기능들은 굳이 객체를 생성하지 않아도 괜찮을 듯합니다.

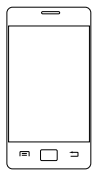
■ **Math.random( )이 좋은 예입니다.**

아무리 생각해도 데이터를 개별적으로 보관하지 않는다면, 그리고 앞으로도 그렇게 될 일이 별로 없다면 굳이 객체를 찍어내는 형태의 프로그래밍은 의미가 없어 보입니다. 사실 Java에서도 이런 개념을 static이라는 키워드로 구현하고 있습니다. 지금은 객체에 대해 설명하기 때문에 예만 들어서 설명하도록 하겠습니다.

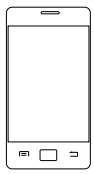
여러분이 변수나 제어문을 공부할 때 Math.random( )을 이용해 보았습니다. Math.random( )이라는 것을 생각해 보면 new를 이용해서 새로이 무언가 작업하는 내용이 없이 실행하는 코드입니다. 그리고 이 결과는 매번 다르게 나옵니다. Math.random( )은 실행될 때 발생하는 데이터를 보관하지 않기 때문에 중복된 데이터가 나올 수 있다는 것을 알 수 있습니다. 즉 이렇게 Java에서도 데이터를 보관하지 않는다면 굳이 객체의 필요성을 고집하지 않습니다. 이것은 뒤에 static에 대해 설명할 때 다시 다루도록 합니다.

■ **객체란 데이터의 묶음**

결과적으로 객체가 데이터를 개별적으로 다르게 보관하지 않는다면 굳이 어떤 데이터를 객체로 만들 필요는 없습니다. 그리고 클래스 선언 안에는 여러 개의 인스턴스 변수를 선언할 수 있다는 것은 여러 개의 데이터를 각각 객체마다 보관하는 형태가 됩니다.



010-AAAA-BBBB



010-CCCC-DDDD

그림 26 각각의 휴대전화와 다른 번호

16 객체: 데이터들의 묶음

프로그램을 작성하기 위해서 변수들을 선언하고 사용하다 보면 가끔은 떨어지면 의미가 없는 데이터들을 많이 보게 됩니다. 예를 들어 성적에 대한 데이터는 '과목과 점수'라는 구성 요소로 이루어졌음을 알 수 있습니다. 만일 프로그래밍을 만들어야 하는데 "100점을 받은 과목은?", "국어 과목의 성적은?"과 같은 데이터를 알아내려 할 때 여러분이 가장 먼저 알아야 하는 사실은 '과목과 점수'라는 두 가지 데이터가 분리되지 않는 하나의 의미가 있는 데이터라는 것입니다. 전 이런 데이터를 '단위 데이터'라고 표현하기도 하는데, 이런 데이터의 특징은 바로 도표와 같은 형태로 표현할 수 있다는 것입니다.

과목명	점수
국어	80
영어	70
수학	60

각 행의 데이터는 서로 분리되면
의미 없는 데이터에 불과합니다.

그림 27 떨어지면 의미가 없는 과목별 점수 데이터

아주 간단한 예를 들어서 우리가 쉽게 계산할 수 있는 것들도 클래스로 만들어 보면 좀 색다르게 느껴집니다. 제가 주로 사람들에게 제시하는 예제는 바로 도형의 문제입니다.

■ **분리되면 의미가 없는 데이터를 도표로 그려봅니다.**

가로와 세로의 데이터, 그리고 삼각형인지, 사각형인지를 의미하는 데이터를 가지고 여러분이 도형의 넓이를 계산한다고 생각해봅시다. 여러분이 이 문제를 풀기 위해 가장 먼저 할 일은 데이터를 아래와 같이 정리해보는 작업입니다.

도형 종류	가로	세로
삼각형	100	200
사각형	250	130
삼각형	300	100

표

각각의 도형마다 다른 데이터를 가지고 있고, 그 데이터는 떨어지면 의미가 없게 됩니다. 위와 같은 형태로 정리할 수 있는 데이터는 우리 실생활에 너무나도 많습니다. 여러분이 물건을 구매했을 때 받는 영수증도 결국은 이런 형태입니다. 이 데이터의 가로 라인이 하나의 객체라고 보시면 됩니다. 즉 하나의 도형 객체는 삼각형이라는 데이터와, 가로 100, 세로 250이라는 데이터로 구성되었다고 보시면 되는 겁니다.

예제 | 도형이 클래스가 되는 모습

```
public class Shape {
      private char type;
      private int width;
      private int height;
}
```

아직 메소드의 구현은 없습니다만 이런 식으로 분리되면 안 되는 데이터는 하나의 단위인 객체로 구성하게 되고, 이를 위해서 하나의 클래스를 작성하게 됩니다. 또 다른 예를 들어 음식점에 먹은 음식의 값을 계산하는 것은 어떨까요? 친구들과 음식점에서 여러 종류의 음식을 주문해서 먹었다면, 각 음식의 가격이 있고, 수량이 있습니다. 그리고 이런 데이터가 여러 개 존재하게 됩니다. 이때에도 도표를 활용할 수 있을 겁니다.

가격	수량
3,000	2
4,000	5

표

클래스의 구성은 아마도 다음과 같은 형태가 될 듯합니다.

예제

```
public class FoodPrice {
      private int menuPrice;
      private int quantity;
}
```

이런 방식으로 분리되지 않는 데이터를 하나의 클래스로 모아주면 나중에 객체를 생산해서 각 객체가 하나의 묶음 데이터를 표현하도록 사용됩니다.

■ **객체가 데이터를 가지고 있다면 잘 알거나, 잘할 수 있는 일들이 많습니다.**

객체가 데이터를 가질 수 있다는 사실을 제대로 인지한다면 여러분의 객체지향 프로그래밍은 정말 많은 성장을 하게 됩니다. 그 가장 큰 이유가 바로 객체가 데이터를 가지니 객체가 주인공이 되어서 로직을 처리하기 시작하기 때문입니다. 앞의 도형을 생각해보시면 아주 좋을 듯합니다. 하나의 도형 객체가 여러 개의 필요한 데이터를 가지고 있기 때문에 이 데이터를 가지고 무언가 처리해야 하는 로직은 도형 객체가 가장 잘할 수 있다는 겁니다. 가장 좋은 예는 역시는 도형의 넓이가 되지 않을까 싶네요.

예제

```
public class Shape {
      private char type;
      private int width;
      private int height;

      public float getArea(){
            float area = 0F;
            //내부 구현은 생략
            return area;
      }
}
```

가령 넓이를 구하는 게 아주 복잡하고, 난해해서 몇백 라인의 소스로 구성된다고 생각해봅시다. 여러분이 위와 같이 어떤 도형의 객체가 스스로 넓이를 구하도록 할 수 있다면 상당히 프로그램을 작성하는 데 편해집니다.

■ **객체지향이라고 해서 무조건 현실을 그대로 베끼지는 않습니다.**

흔히들 객체지향 프로그래밍을 현실 세계를 모델로 한다고 합니다. 그래서 현실 세계에 있는 어떤 존재를 프로그래밍으로 옮긴다고 하는데, 과연 그럴까요? 조금 전의 예를 생각해 보면 도형이라는 것은 데이터만을 의미합니다. 그러니 이것은 단순히 하나의 명사일 뿐이고, 여기에 로직이나 기능 따위는 존재하지 않아야만 정상입니다. 하지만, 실제로는 그렇지 않은 것이 보시는 바와 같이 어떤 객체가 데이터를 가지고 있고, 그 데이터를 가공하는 일을 알아서 하는 구조로 가는 것이 객체지향 프로그래밍입니다.

■ **한국어 자체가 조금 불리합니다.**

한국어는 무생물 존재가 주어가 되는 경우가 없습니다. "삼각형의 넓이를 구한다."라는 문장을 보면 "(내가 혹은 우리가) 삼각형의 넓이를 구한다."가 됩니다. 반대로의 위의 코드는 "삼각형이 넓이를 계산한다."라는 문장이 됩니다. 영어는 무생물 주어를 사용하기 때문에 이런 표현이 더욱 자연스럽습니다. "삼각형이 자신의 넓이를 계산한다."와 같은 문장이 성립할 수 있기 때문에 객체위주의 사고에 조금 유리하다고 할 수 있습니다.

■ **데이터를 가지고 잘할 수 있는 일은 데이터를 정의한 클래스에 넣어줍니다.**

제가 제시해 드리고 싶은 가이드는 간단히 말해서 어떤 데이터를 가진 존재를 생각했다면, 그 데이터를 가장 잘 활용할 수 있는 것도 그 존재라고 생각하는 겁니다. 즉 도형 객체가 도형의 종류와 가로, 세로 등의 필요한 데이터를 가지고 있으니 이제 넓이를 구하는 최적화된 모든 데이터는 도형이 가지고 있다고 판단하는 방식을 생각해보시면 좋겠습니다.

17 데이터가 주인공인 클래스와 생성자(Constructor)

데이터를 묶어둔 클래스에 대해 얘기하면서 굳이 예제를 완성하지 않은 이유는 바로 생성자(Constructor)를 이해해야 하기 때문입니다. 앞의 내용을 찬찬히 보시면 **데이터들이 흩어지는 것을 막고자 객체로 묶어주고, 이를 위해서 하나의 클래스를 선언한다**는 논리가 사용됩니다. 이런 식의 데이터를 중심으로 두고 클래스를 설계하게 되면 가끔은 '객체가 반드시 특정 데이터를 가져야 하는 상황'이 발생합니다. 즉 경우에 따라서는 객체가 데이터를 반드시 가져야 의미가 있는 경우가 있다는 겁니다. 예를 들어 앞의 도형의 경우에도 넓이를 구하기 위해서는 반드시 도형의 종류가 명시되어야만 합니다. 도형의 종류를 명시하지 않으면 어떤 도형 객체는 아무런 의미가 없습니다.

> 데이터를 보관하기 위해서 객체를 만들 때 반드시 어떠한 필수적인 데이터가 있어야만 객체로 만들고 싶은 경우에 생성자(Constructor)라는 문법을 이용합니다. 예를 들어 회원 가입을 할 때 반드시 아이디와 패스워드가 필요한 것처럼, 어떤 객체를 만들 때 필수적인 데이터가 있다면 생성자를 고려하는 것이 좋습니다.

17.1 생성자라는 강제 옵션

이런 이유에서 객체를 생성할 때 반드시 어떤 데이터를 넣지 않으면 객체를 생성할 수 없도록 하는 방안이 나오는 데 이것을 생성자라고 합니다. 생성자는 오로지 클래스에서 객체를 생성할 때만 사용됩니다. 오직 new 뒤에 객체를 만들 때만 사용한다는 겁니다. 메소드와는 차원이 다릅니다. 메소드는 객체가 할 수 있는 동작을 의미하지만, 생성자는 아예 객체의 생성 여부와 관계가 있습니다. 지금까지 여러분이 만든 클래스에는 메소드라는 것은 있었지만, 생성자라는 것은 존재하지도 않았습니다. 그런데 이 생성자라는 것은 어디서 나온 걸까요?

17.2 컴파일러가 끼워 넣은 기본 생성자(Default Constructor)

그동안 아무런 의심 없이 받아들였던 일이 때로는 비밀을 가질 수가 있습니다. 아마도 지금 설명하는 내용도 그런 비슷한 맥락이 아닐까 합니다. 예전에는 클래스를 선언하고 나면 객체를 만들 수 있다는 사실을 알고 있습니다. 하지만, 기계어로 어떻게 만들어졌는지를 살펴보면 정말 우리가 소스에서 만든 부분만이 존재할까요? 제가 간단한 클래스를 하나 만들어 보도록 하겠습니다.

예제 | 아무것도 선언하지 않는 SampleObj 클래스

```
public class SampleObj {

}
```

위의 코드를 보면 아무것도 없이 클래스를 선언만 해 두었습니다. 그렇다면, 이것을 컴파일한 클래스 파일 결과물도 있을 겁니다. '역 컴파일러'라고 얘기하기도 합니다만, 기계어로 만들어진 것을 반대로 사람의 눈이 알아볼 수 있는 코드로 변환하는 소프트웨어들이 있습니다. JDK가 설치된 bin 폴더 밑에 있는 javap라는 명령어는 자세히는 아니지만, 기본적인 모습을 기계어에서 인간이 볼 수 있는 소스로 변환해주는 명령어입니다.

예제 | javap 명령어를 이용한 SampleObj.class 파일의 조회 결과

```
Compiled from "SampleObj.java"
public class SampleObj extends java.lang.Object{
      public SampleObj();
}
```

위의 코드를 보시면 이상하게 약간의 코드가 더 붙어 있는 것이 보일 겁니다. 우선은 클래스의 선언 부분에 extends java.lang.Object가 붙은 부분이 보입니다. 이것은 상속이라는 것입니다만, 다음에 학습할 내용이니 그때까지 미루도록 합니다. 그다음 라인에 나오는 'public SampleObj();'라는 코드가 얘기하고 싶은 대상입니다. 분명히 이 코드는 제가 프로그램을 만들면서 작성한 코드도 아닌데 클래스 파일이 가지고 있습니다. 이 코드의 정체는 무엇일까요?

17.3 기본 생성자(Default Constructor)라는 자동 코드

기본 생성자는 Java 클래스 소스를 컴파일할 때 자동으로 만들어주는 생성자입니다. 따라서 개발자가 생성자를 만들지 않아도 사실은 new라는 표현을 실행하면 기본 생성자를 통해서 객체가 생성됩니다.

클래스 파일을 만들면 컴파일러는 앞에서처럼 약간의 코드를 추가해줍니다. 이 코드의 의미는 여러분이 main 메소드에서 객체를 만들 때 아무런 의심 없이 사용한 코드입니다.

코드

```
SampleObj obj = new SampleObj();
```

객체를 만들 때 가장 기본으로 사용하는 코드가 위와 같습니다. 그런데 javap 명령어를 사용했을 때도 이런 식으로 만들어진 코드가 있다는 게 보입니다. 자세히 살펴보면 일반적인 메소드와는 약간의 규칙이 다릅니다.

- 일반적인 메소드와는 달리 리턴 타입에 대한 언급이 없다.
- 메소드의 이름이 클래스의 이름과 같다.

클래스를 만들면 자동으로 만들어지는 위의 2가지 규칙을 지키는 함수가 자동으로 만들어지는데 이것을 우리는 기본 생성자(Default Constructor)라고 합니다. 기본 생성자가 있다는 얘기는 다른 방식의 생성자가 있다는 얘기입니다. 우선 여기서 알고 넘어갈 사항은 단 하나입니다. 기본 생성자라는 것은 컴파일러가 자동으로 만들어준다는 겁니다.

17.4 사용자 정의 생성자(UserDefined Constructor): 생성자를 프로그래머 마음대로

사용자 정의 생성자(생성자 함수라고 하기도 함)는 개발자가 객체를 만들 때 꼭 이렇게 하고 싶다고 의사표현을 하는 것입니다. 따라서 컴파일러는 기본 생성자를 포기하고, 개발자의 의견대로 객체를 만들 수 있게 합니다.

기본 생성자와는 달리 프로그래머가 직접 클래스를 만들 때 생성자라는 것을 정의할 때도 있습니다. 이것을 '사용자 정의 생성자'라고 합니다. 사용자 정의 생성자는 다음과 같은 규칙만 알면 만들어 낼 수 있습니다.

- 메소드처럼 선언하지만 메소드의 리턴 타입이 없다.
- 메소드의 이름이 클래스의 이름과 같다.

사용자 정의 생성자의 문법은 잠시 후에 봐도 괜찮습니다. 오히려 생성자를 사용자가 정의한다는 것이 어떤 의미인지를 명확하게 아셔야만 합니다.

예제 사용자 정의 생성자를 추가한 SampleObj

```
public class SampleObj {

      //사용자 정의 생성자 추가
      public SampleObj(int value){

      }

}
```

기존의 SampleObj 클래스에 사용자 정의 생성자를 추가해 주었습니다. 이 코드가 기계어로 번역된 모습을 보면 다음과 같습니다.

예제 컴파일된 기계어를 다시 변환해본 SampleObj

```
Compiled from "SampleObj.java"
public class SampleObj extends java.lang.Object{
      public SampleObj(int);
}
```

■ **사용자 정의 생성자를 사용하면 컴파일러가 만드는 기본 생성자는 없어집니다.**

결과를 자세히 보시면 이전과는 달리 기본 생성자가 사라지는 것을 볼 수 있습니다. 이것은 사용자 정의 생성자의 용도를 정확히 이해하면 당연한 결과입니다. 사용자 정의 생성자의 경우는 주로 객체를 생성하는데 제약 조건을 겁니다. 예를 가격이 없는 메뉴는 없는 것처럼, 객체를 만들

때 반드시 특정 조건을 만족해야지만 객체를 만들어 낼 수 있게 하려고 생성자 함수를 사용하는 겁니다. 도형의 경우도 마찬가지로 여러분이 도형의 종류와 가로, 세로의 데이터가 필수라고 생각할 때 사용하는 겁니다. 도형의 타입과 가로, 세로의 데이터를 주지 않으면 도형 객체를 만들어 줄 수 없게 하려고 사용자 정의 생성자를 이용하게 됩니다.

■ **객체 생성 시 필수적인 데이터가 있도록 제약하는 것이 사용자 정의 생성자입니다.**

기본 생성자의 경우는 클래스를 만들면 객체를 생성할 수 있게 해주기 위해서 컴파일러가 자동으로 만들어 줍니다. 하지만, 개발자가 직접 "난 이렇게 해야지만 객체를 생성할 수 있도록 하겠어." 라는 의미로 사용자 정의 생성자를 작성하게 되면 이제는 컴파일러의 입장에서는 개발자의 의도를 충분히 반영해주어야만 합니다. 따라서 개발자가 하나라도 사용자 정의 생성자 함수를 사용하게 되면 컴파일러는 자신의 의도였던 기본 생성자 함수를 더는 작성하지 않고, 개발자의 의도를 존중해서 개발자가 원하는 대로만 객체를 생성할 수 있게 해주도록 합니다. 이런 의미에서 사용자 정의 생성자 함수는 객체를 생성하는 데 있어서 제약 조건처럼 동작합니다.

> 사용자 정의 생성자는 주로 개발자가 객체에 필요한 필수적인 데이터를 주려고 할 때 만들기 때문에 객체 생성 시에 강제적인 데이터를 넣도록 하는 장치로 많이 사용됩니다.

17.5 생성자와 this라는 키워드

> this를 해석하는 가장 편리한 해석은 self입니다. 즉 현재 코드를 실행하는 현재 객체 자체를 가리킵니다.

도형 데이터를 주고 계산한다고 하면 이제는 사용자 정의 생성자를 이용해서 객체를 만들도록 수정해봅시다.

예제 | 사용자 정의 생성자를 추가한 Shape 소스

```
public class Shape {
      private char type;
      private int width;
      private int height;
```

```
	public Shape(char t, int w, int h){
		type = t;
		width = w;
		height = h;
	}

	public float getArea(){
		float area = 0F;
		return area;
	}
}
```

Shape를 위와 같이 수정하게 된다면 이제는 객체를 만들 때 반드시 char, int, int 타입의 데이터를 주어야만 객체를 생성할 수 있게 됩니다. 이제 도형을 테스트하기 위해서 main 메소드를 가진 소스를 하나 만들어 보도록 합니다.

예제 Shape 클래스에서 객체를 만들어서 테스트하기 위한 소스

```
public class ShapeTest {
	public static void main(String[] args) {
		Shape s1 = new Shape('T', 100,200);
	}
}
```

가장 중요한 점은 new Shape( )를 할 때 사용자 정의 생성자를 사용했기 때문에 반드시 필요한 데이터를 다 넣어 주어야만 Shape 객체를 만들어 낼 수 있다는 사실입니다. 이제 삼각형일 경우에는 도형의 넓이를 제대로 계산해주도록 수정해서 실제로 결과를 확인해보도록 합니다.

예제

```
public class Shape {
	private char type;
	private int width;
	private int height;

	public Shape(char t, int w, int h){
		type = t;
		width = w;
		height = h;
	}
```

```
	public float getArea(){
		float area = 0F;

		if(type == 'T'){
			area = (width * height)/(float)2;
		}
		return area;
	}
}
public class ShapeTest {
	public static void main(String[] args) {
		Shape s1 = new Shape('T', 100,200);
		System.out.println(s1.getArea());
	}
}
```

```
10000.0
```

■ **this 키워드는 현재 실행되고 있는 객체를 의미합니다.**

다시 한번 사용자 정의 생성자를 봅시다.

코드

```
public Shape(char t, int w, int h){
	type = t;
	width = w;
	height = h;
}
```

위의 코드를 보면 사용자 정의 생성자에 파라미터를 세 개 던지고, 그것을 객체의 데이터인 type, width, height라는 데이터의 값으로 지정하고 있습니다. 이 코드는 다음과 같이 사용할 수도 있습니다.

코드

```
public Shape(char t, int w, int h){
	this.type = t;
	this.width = w;
	this.height = h;
}
```

달라진 점이라면 this.xxx 과 같은 방식의 코드가 사용되었다는 점 정도입니다.

■ **'.'이라는 표기법은 리모컨을 눌러서 객체를 움직일 때 사용하였습니다.**

생각해 보시면 '.' 표시는 우리가 주로 어떤 객체를 동작시킬 때 주로 사용하였습니다. 예를 들어 'Scanner s = new Scanner(System.in);'과 같은 코드에서 s.nextInt( );와 같은 형태로 사용했습니다. 그렇다면, 여기서 사용한 this라는 키워드는 무엇일까요? this는 한마디로 '현재 이 코드를 실행하는 객체'를 의미합니다. 따라서 해석을 해보자면 파라미터로 들어온 width와 height라는 변수의 값을 현재 이 코드를 실행하는 객체의 인스턴스 변수로 삼겠다는 의미입니다.

■ **현재 객체의 데이터나 메소드에 접근할 때 사용하는 this**

this라는 키워드는 좀 더 편하게 'self'로 이해하셔도 됩니다. 이 코드를 '현재 실행하는 객체'라는 뜻입니다. 사실 주로 실무에서는 생성자를 작성할 때 다음과 같이 만들어 주는 것이 관례입니다.

예제

```
public Shape(char type, int width, int height){
      this.type = type;
      this.width = width;
      this.height = height;
}
```

소스를 보면 사용자 정의 생성자에 파라미터의 이름을 일부러 객체의 인스턴스 변수의 이름과 동일하게 작성한 것이 보입니다. 굳이 이렇게 번잡한 작업을 하는 이유는 사실 가독성 때문입니다. 즉 파라미터가 많으면 입력하는 파라미터가 객체의 안에서 어떻게 사용될지 알 수 없는 일이 발생하기 때문에 그냥 파라미터를 아예 데이터의 이름과 통일시켜 버리는 겁니다. 이렇게 되면 width로 던지는 파라미터가 현재 객체의 width 데이터가 될 것이라는 것이 잘 보이기 때문입니다. 솔직히 성능으로 따지자면 this.width와 같은 방식은 오히려 성능에 저해되는 요소이지만 여러분은 "빠른 코드보다는 쉬운 코드가 더 낫다."라는 격언대로 this를 이용해주는 것이 좋습니다.

■ **사용자 정의 생성자는 객체를 만드는 옵션: 때로는 여러 개가 있을 수도 있습니다.**

사용자 정의 생성자는 여러 개 만들 수 있습니다. 이 의미는 객체를 생산할 수 있는 방식이 여러 가지라는 겁니다. 사용자 정의 생성자가 세 개라면 세 가지 방식으로 객체를 만들어 낼 수 있게 됩니다.

앞에서 우리는 사용자 정의 생성자라는 것이 객체를 생성하는 데 있어서 하나의 옵션처럼 동작한다는 사실을 배웠습니다. 그리고 사용자 정의 함수는 주로 객체에 필수적인 데이터를 채우고 싶을 때 사용한다고 설명했습니다. 그런데 때로는 10개의 데이터가 있는데 그중에 5개만 반드시 필요하고, 나머지는 필요 없거나 기본적인 어떤 값을 가지는 경우도 있습니다. 굳이 예를 들자면 햄버거 가게에서 "치즈버거 주세요."라고 말하면 당연히 햄버거 하나를 얘기하는 것과 비슷한 원리입니다. 사용자 정의 생성자는 객체를 만드는 옵션이기 때문에 이런 상황을 대비해서 여러 개를 작성해 줄 수도 있습니다. 예를 들어 다음의 코드를 봅시다.

예제 | 사용자 정의 생성자가 두 개인 FoodPrice 클래스

```
public class FoodPrice {
      private int menuPrice;
      private int quantity;

      public FoodPrice(int menuPrice){
            this.menuPrice = menuPrice;
            this.quantity = 1;
      }

      public FoodPrice(int menuPrice, int quantity){
            this.menuPrice = menuPrice;
            this.quantity = quantity;
      }

      public int getTotalPrice(){
            return menuPrice * quantity;
      }
}
```

소스를 보면 사용자 정의 생성자가 두 개입니다. 즉 객체를 생성하는 옵션이 두 가지라는 것을 의미합니다. 하나는 그냥 메뉴의 가격만을 넣는 것이고, 또 하나는 메뉴의 가격과 수량을 같이 지정하는 겁니다. 다만, 메뉴의 가격만을 넣으면 자동으로 메뉴의 수량은 1이 되도록 작성되어 있습니다. 그런데 다시 한번 생각해 보면 굳이 반복적으로 메뉴의 이름과 수량을 매 생성자에서 처리하고 있을 필요는 없을 듯합니다.

17.6 this 키워드의 또 다른 용도는 다른 사용자 정의 생성자를 호출하는 것

사용자 정의 생성자 함수를 여러 개 작성할 때 하나 고려해주어야 하는 것이 바로 중복적인 코드가 발생한다는 겁니다.

코드 | 거의 비슷한 코드가 두 번 사용된 생성자

```
public FoodPrice(int menuPrice){
      this.menuPrice = menuPrice;
      this.quantity = 1;
}

public FoodPrice(int menuPrice, int quantity){
      this.menuPrice = menuPrice;
      this.quantity = quantity;
}
```

위의 두 개의 사용자 정의 생성자를 보시면 거의 같은 코드가 사용되고 있는 것이 보입니다. 여러분이 소스를 작성할 때 중복된 코드를 보면 무언가 고칠 부분이 있다고 생각하시는 것이 좋습니다. 위의 소스는 아래처럼 간단하게 수정해 주는 것이 좋습니다.

예제 |

```
public FoodPrice(int menuPrice){
      this(menuPrice, 1);
}

public FoodPrice(int menuPrice, int quantity){
      this.menuPrice = menuPrice;
      this.quantity = quantity;
}
```

소스를 보면 this( )로 사용된 부분이 보입니다. this( )의 경우는 또 다른 생성자를 호출할 때 사용하는 옵션입니다. 즉, 위의 경우라면 생성자 중에서 파라미터를 두 개 받는 생성자를 찾아서 실행하라는 의미입니다. 생성자가 하나의 클래스에 여러 개 정의되는 경우에는 위와 방식을 이용해서 중복적인 코드를 최소화합니다.

17.7 사용자 정의 생성자와 오버로딩(Overloading)

사용자 정의 생성자의 경우에는 이름이 다 똑같이 클래스의 이름과 같습니다. 다만, 그 안의 파라미터의 값이 경우에 따라 다르게 작성됩니다. 객체지향에서는 오버로딩(Overloading)이라는 용어로 이런 코드를 설명합니다.

■ **동일 소스 내에서 이름이 같습니다.**

오버로딩은 동일한 소스 내에서 이름이 같은 메소드나 사용자 정의 함수가 여러 개인 모습을 의미합니다. 오버로딩은 다음과 같은 규칙을 따릅니다.

- 호출하는 이름은 같지만, 파라미터들은 다르다.
- 파라미터의 개수가 같다면 파라미터들의 타입이 다르다.
- 리턴 타입은 다를 수 있다(생성자의 경우에는 리턴 타입이 없으므로 신경 쓰지 않아도 된다).

오버로딩은 Java에서는 생성자와 메소드에서 사용 가능합니다. 아마도 가장 많이 쓰이는 곳은 역시 생성자이고, 메소드의 경우라면 동일한 기능의 메소드이지만, 경우에 따라서 다른 파라미터들을 처리해야 하는 경우에 사용됩니다.

17.8 객체가 만들어지면서 어떤 메소드를 실행할 때도 생성자

생성자가 사용되는 경우 중에 또 다른 가장 빈번한 경우는 객체가 만들어진 직후에 어떤 메소드를 실행하고 싶은 경우에 사용되는 경우라고 할 수 있습니다. 클래스에서 객체가 만들어지면서 처리되는 메소드가 있다면 이런 메소드들은 외부에서 호출되기 전에 생성자에서 호출될 수 있습니다. 예를 들어 여러분이 만든 클래스 안에 인터넷과의 연결이 필요하다고 생각해보겠습니다. 객체를 생성해서 동작하기 전에 반드시 인터넷 가능 여부가 확인되어야 한다면 다음과 같은 가상의 코드를 생각해볼 수 있습니다.

예제 | 객체가 생성될 때 어떤 메소드를 호출해야 하는 상황

```
public class InternetData {

      private String domain;

      public InternetData(String domain){
            this.domain = domain;
```

```
        }

        public void checkInternet(){
                System.out.println("인터넷 부터 확인합니다.");
        }
}
```

코드의 내용은 별다른 것이 아닙니다. 문제는 위와 같은 방식으로 코드를 만들어 두면 클래스에서 객체를 만드는 사람은 사실 객체에게 다른 메소드를 실행시키기 전에 반드시 checkInternet( ) 메소드를 실행해서 인터넷 접속이 가능한지를 확인하는 코드가 들어가야 한다는 점입니다.

예제 생성자에서 처리하지 않는 경우

```
        public static void main(String[] args) {

                InternetData obj = new InternetData("http://www.google.com");
                //인터넷 확인
                obj.checkInternet();
                //나머지 작업
        }
```

그런데 만일 객체가 생성되면서 인터넷이 체크될 수 있도록 수정하면 객체를 생성해서 사용하는 사용자로서는 인터넷이 체크되기 때문에 별도의 확인 작업 없이 바로 원하는 작업을 진행할 수 있게 됩니다.

예제 생성자에서 객체 생성 시 필요한 메소드를 바로 실행

```
public class InternetData {

        private String domain;

        public InternetData(String domain){
                this.domain = domain;
                checkInternet();
        }

        private void checkInternet(){
                System.out.println("인터넷 부터 확인합니다. ");
        }

}
```

수정된 코드를 보면 생성자 내에서 바로 checkInternet( ) 메소드를 실행하게 합니다. 이렇게 되면 외부에서는 매번 객체를 만들 때마다 checkInternet( ) 메소드를 실행해주어야 하는 불편함을 없앨 수 있습니다. 자세히 보면 checkInternet( ) 메소드 역시 public에서 private으로 외부에서 호출할 필요가 없어지면서 접근 제한이 변경되었습니다. 생성자의 기능을 크게 두 가지로 구분합니다. 하나는 '객체 생성 시에 절대적으로 필요한 데이터를 넣도록 하는 강제성'의 기능이고, 다른 하나는 '객체가 생성되면 자동으로 어떤 메소드를 호출하는 기능'을 위한 장치입니다.