CHAPTER 09

접근 제한, 패키지, static

클래스의 변수나 메소드에 보안 레벨을 적용하는 접근 제한 키워드를 배워봅니다. 아울러 공유되는 데이터나 함수 선언 시에 사용되는 static 키워드를 알아봅니다.

객체지향 프로그래밍의 기본과는 관계가 별로 없습니다만, 프로그램을 만들다 보면 다른 사람들과의 지켜야 하는 예의가 있습니다. 이번 장에서는 접근 제한이라는 것에 대해서 알아보도록 하고, 프로그램의 코드들을 좀 더 구조화하는 패키지라는 것에 대해서도 알아보도록 하겠습니다. 이번 장의 내용은 프로그램을 만드는 데 있어서 필수불가결한 것은 아니지만 다른 사람들이 작성하는 코드를 볼 경우나, 잘못된 사용을 막는 역할을 합니다.

1 패키지 폴더 구조

Java와 C 언어는 프로그램의 규모를 얘기할 때에도 말하는 방식이 다릅니다. 예를 들어 C 언어인 경우에는 라인의 수로 얘기를 하는 것이 보통입니다. 몇만 라인의 코드로 작성된 부분이라는 방식입니다. 반면에 Java의 경우는 그렇게 얘기하지 않습니다. Java의 경우는 컴파일된 클래스 파일의 개수로 얘기하는 것이 일반적입니다. 이렇게 클래스 파일의 수가 많으면 우선은 클래스 이름의 충돌을 조심해야 합니다. 여러 명이 같이 개발하는 경우에는 더욱 그렇습니다. 그럼 어느 정도의 수가 프로젝트의 일반적인 규모가 될까요? 작은 경우에는 몇백 개의 클래스, 큰 경우에는 몇천 개 혹은 몇만 개의 클래스 파일로 만들어지게 됩니다. 따라서 이 파일들을 좀 더 체계적으로 관리하기 위해서 폴더를 만들어 주게 됩니다. 이런 클래스의 폴더들을 패키지라고 생각하시면 됩니다.

■ **JDK와 패키지**

JDK 안쪽에는 5,000개에 가까운 클래스 파일들이 존재합니다. 따라서 이 클래스 파일들을 체계적으로 관리하도록 패키지라는 구조를 사용합니다. 일반적으로 JDK의 패키지는 기능에 따라 묶는 것이 일반적입니다.

그림 1 패키지(http://download-llnw.oracle.com/javase/7/docs/api/index.html)

예를 들어 java.io 패키지는 입출력(Input-Output) 프로그래밍과 관련된 클래스 파일들의 묶음이고, java.util은 유틸리티 기능과 자료구조들과 날짜, 시간 처리에 대한 클래스들의 묶음입니다. JDK에서 제공하는 패키지는 다음과 같은 규칙을 따르는 것이 일반적입니다.

- 시작은 java 혹은 javax로 시작한다.
  - javax인 경우는 추가(Extended) 내용으로 기존에 있었던 패키지보다 많은 기능을 만들어 두었을 때 사용하는 이름입니다.
- 중간 이름은 소문자로 만들어진다.
  - 중간의 모든 이름은 아무리 길어도 대문자를 사용하지 않습니다. 이것은 클래스와 구분하기 위해서입니다.

Java를 처음 접하는 사람들은 엄청나 패키지와 클래스의 양을 보고 "저걸 다 공부해야 하나?"고 겁부터 먹는 경우를 종종 보는데 아마 저 많은 API를 다 아는 사람은 없다고 생각합니다. 거의 개발자들은 자신이 주로 사용하는 부분에 대해서만 알기 때문에 전체의 5~ 10% 정도만 주로 사용하게 됩니다.

■ **import와 Scanner 클래스**

Scanner 클래스를 이용하는 코드에서 import java.util.Scanner; 라는 코드가 추가되었던 것을 기억하십니까? Scanner는 java.util 패키지 안에 있는 클래스이기 때문에 코드에서 사용하려면 반드시 import 구문을 선언해서 사용해주어야 합니다. import 문은 이렇게 현재 작성하는 코드에서 다른 패키지의 클래스를 이용하기 위해서 코드의 맨 앞부분에 추가해주는 것이 일반적입니다. 다만, 다른 패키지들과는 달리 java.lang 패키지는 자동으로 임포트되기 때문에 별도의 import 문이 필요하지 않습니다. 하지만, 나머지 패키지는 반드시 import 코드를 붙여 주어야만 합니다.

2 자신만의 패키지를 설계하는 방법

실무에서는 회사에서 개발 업무가 진행되면서 많은 클래스 파일들이 만들어지게 됩니다. 따라서 이 역시 패키지의 형태로 문서화하고, 관리하게 되어 있습니다. 개발자들이 회사의 지침을 참고로 하는 '코딩 가이드'라는 문서에 따라서 클래스를 작성할 때 정해진 패키지에 맞게 만들어 줍니다.

패키지를 만드는 작업은 의외로 단순합니다. 이클립스의 경우에는 처음 클래스를 만들 때 간단히 패키지 이름을 지정해 주는 것만으로도 작업이 끝납니다. 패키지 이름은 일반적으로 도메인의 역순으로 만드는 것이 보통입니다. 예를 들어 회사의 도메인이 'www.company.co.kr'라면 'kr.co.company'와 같은 이름으로 만들어 주는 것이 일반적입니다.

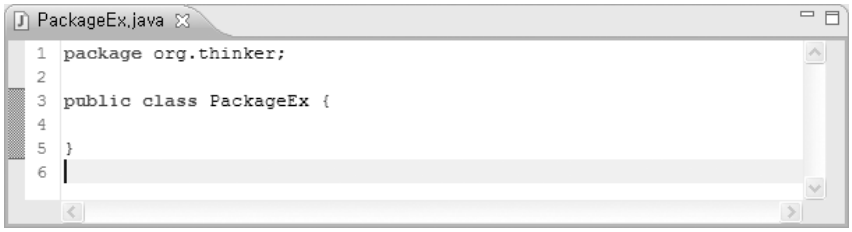

그림 2

패키지를 가장 단순하게 생각하는 방법은 하나의 폴더로 간주하는 방법입니다. 사실 패키지라는 것이 수많은 클래스를 하나의 구분 단위로 묶기 위한 목적도 있었던 것만큼 그 사용법 역시 지극히 단순하게 사용될 수 있습니다.

만일 위와 같은 클래스를 작성해서 정상적으로 컴파일하거나, 컴파일된 상태라면 실제로는 org라는 폴더 밑에 thinker라는 폴더가 있고, 그 안에 PackageEx.class 파일이 존재하게 됩니다. 이클립스는 컴파일된 소스를 workspace 안에 해당 프로젝트의 bin 폴더 내에 보관하고 있습니다.

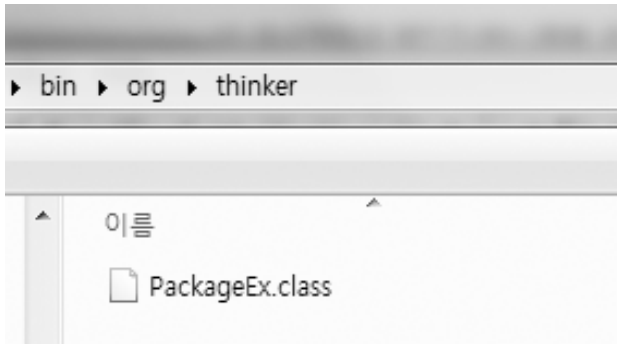

그림 3

2.1 패키지를 설계하는 일반적인 방식

패키지를 설계할 때는 일반적으로 '모듈(Module)'이라는 추상적인 단위를 이용합니다. 즉 시스템의 주된 기능을 단위로 패키지를 구분한다는 뜻입니다. 예를 들어 회원(member), 상품(product), 주문(order)이라는 단위로 시스템이 운영된다면 kr.co.company.member와 같은 방식으로 설계하게 됩니다. 자주는 아니지만, 클래스의 기능별로 패키지를 구분하는 때도 있습니다. 예를 들어 데이터베이스와 관련된 작업을 하는 클래스들만 모아서 별도의 패키지로 관리하거나, 공통적인 유틸리티 관련 클래스들만 모아서 하나의 패키지를 만들 때도 있습니다. 어떤 방식을 이용하는지는 각 개인이나 팀 혹은 회사의 판단입니다. 다만, 중요한 것은 규칙에는 일관성이 있어야 한다는 점입니다.

■ **패키지의 시작은 회사의 도메인으로 합니다.**

패키지의 시작은 앞서 말씀드린 내용처럼 도메인의 역순으로 시작합니다. 보통은 'org.aaa' 혹은 'kr.co.aaa'로 작성을 시작합니다. 비영리 기관일 때 보통 org로 시작하고 그 외에는 com이나

net으로 시작하는 경우가 많습니다.

■ **패키지의 중간의 약어 혹은 모듈의 이름으로 작성합니다.**

패키지의 중간은 보통은 네이밍 룰을 문서로 만들어 정해진 약어로 작성하는 경우가 많습니다. 가끔은 대문자로 된 약어를 사용할 때도 있습니다만 거의 일반적으로 소문자로 된 코드를 이용합니다. (예: 주문관리(OM): kr.co.aaa.om)

■ **패키지의 맨 뒤는 주로 패키지 안에 있는 클래스들의 역할로 작성되는 경우가 많습니다.**

패키지의 마지막 이름은 주로 model, dao, business, ui, util처럼 조금 더 역할별로 구분된 경우가 많습니다.

- model이나 dao의 경우는 주로 데이터베이스 관련 처리를 담당하는 클래스들을 하나의 패키지로 묶을 때 많이 사용한다.
- business나 service라는 패키지 명은 비즈니스 로직을 처리하는 클래스를 모아둔 패키지라고 생각하면 된다.
- 아예 패키지 명이 common이나 util을 쓰는 경우가 종종 있다. 이런 패키지의 경우는 개발 시에 모든 곳에서 같이 사용하는 공용 클래스가 들어가게 된다.

2.2 import는 다른 패키지의 클래스들을 사용할 때 씁니다.

소스 코드에서 가장 먼저 나오는 것은 항상 패키지의 선언입니다. 그리고 다음 나오는 것이 바로 import 구문들입니다. import는 두 가지 형태로 사용합니다. 우선은 import java.util.*;과 같이 '*'를 이용하는 방식과 import java.util.Scanner와 같이 '패키지 이름 + 클래스 이름'을 다 적어주는 방식을 사용합니다. 하지만, 엄밀하게 말하자면 아예 import 구문 없이 모든 패키지 명을 붙여서 클래스를 선언할 수도 있습니다. 예를 들어 'java.util.Scanner s = new java.util.Scanner(System.in);'과 같이 전체 이름을 사용해서 클래스를 선언할 수도 있습니다.

■ **java.lang 패키지와 현재 패키지는 임포트하지 않습니다.**

현재 자신의 패키지와 java.lang 패키지는 별도의 import 구문 없이 사용합니다.

이클립스의 자동 import 기능: Ctrl + ⇧Shift + O

이클립스는 코드를 작성할 때 개발자들이 어떤 패키지를 사용해야 하는지 잘 몰라도 자동완성 기능을 활용해서 import 코드를 자동으로 만들어 줍니다. 예를 들어 아래와 같은 코드를 작성하게 되면 이클립스는 컴파일이 불가능하다는 메시지를 보여 줍니다.

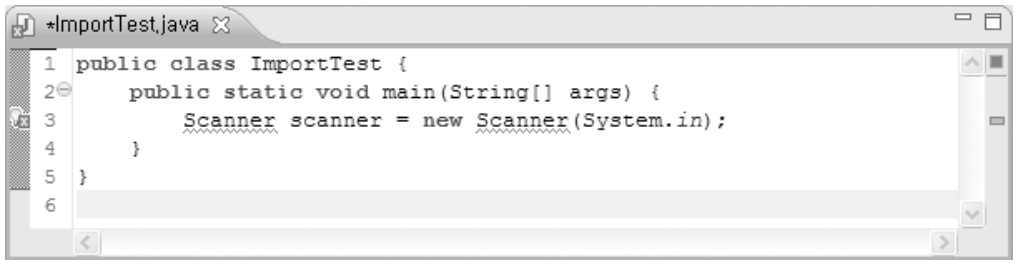

그림 4

이때 화면에서 마우스 오른쪽 버튼을 누르고 [Source] → [Organize Imports] 기능을 선택하거나, 단축키 Ctrl + ⇧Shift + O 를 누르게 되면 자동으로 import 코드가 생성됩니다. 만일 Scanner라는 클래스가 여러 곳에 있다면 사용자가 선택하도록 해줍니다.

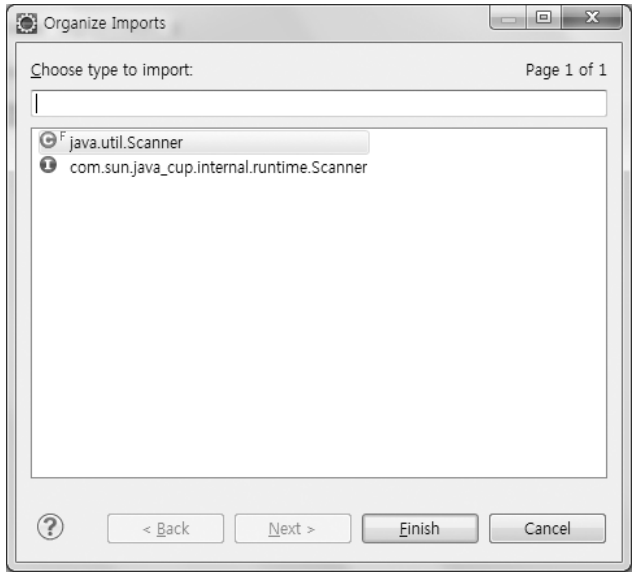

그림 5

3 접근 제한이라는 것이 왜 필요한가?

패키지가 단순히 문법적이고, 사용법 위주였다면 접근 제한 개념은 조금 더 이해가 필요한 부분입니다. 우선은 접근 제한(Access Modifier)이라는 것이 왜 필요한지부터 생각해봅시다.

3.1 접근 제한은 외부에 공개되는 수준을 의미합니다.

프로그램을 만들다 보면 외부에 공개해주어야 하는 것이 있고, 그렇지 않은 것도 있습니다. 이때 그 수준을 결정할 때 접근 제한자를 이용합니다. 메소드의 경우에는 메소드에서 또 다른 메소드를 호출하거나, 여러 개의 메소드를 호출해서 결과를 조합할 때가 있으므로 이럴 때 내부적으로 동작하는 메소드를 외부에 굳이 공개할 필요가 없을 때가 많이 있습니다. 인스턴스 변수의 경우 메소드를 통해서 동작하고 그 결과를 인스턴스 변수의 값으로 저장하거나 제어하는 경우가 많은데 이 경우 외부에서 데이터에 직접 접근하는 것에 대한 방지책이 필요할 수가 있습니다.

Java 언어에서 문법적인 접근 제한자라고 하면 public, protected, default(아무것도 없음), private과 같이 4가지가 있습니다만 가장 많이 쓰는 접근 제한자는 public과 private입니다.

- public: 모든 외부에서 직접 접근하거나 호출할 수 있습니다.
- protected: 현재 클래스와 동일 패키지이거나 다른 패키지이더라도 상속 시에는 접근하거나 호출할 수 있습니다.
- default: 현재 클래스와 동일한 패키지 내에서만 접근하거나 호출할 수 있습니다.
- private: 현재 클래스의 { } 바깥쪽에서는 절대로 보이지 않습니다.

3.2 접근 제한 키워드를 사용하는 곳

접근 제한 키워드를 사용하는 곳은 클래스를 선언하는 경우, 변수를 선언하는 경우, 메소드를 선언하는 경우 그리고 내부 클래스를 선언하는 경우로 나누어 볼 수 있습니다.

■ 클래스의 접근 제한: public 혹은 default

클래스를 선언할 때에는 접근 제한자 중에서 public과 default만을 선택할 수 있습니다. public으로 만들면 외부에 클래스를 노출하겠다는 의미이고, default는 현재 패키지 내에서만 보이는 클래스로 만들겠다는 의미입니다.

■ **인스턴스 변수의 접근 제한: public, protected, default, private**

인스턴스 변수에 대한 접근 제한은 위의 네 가지를 다 이용할 수 있습니다. 하지만, 일반적으로 인스턴스 변수는 private으로 만드는 경우와 프로그램에서 완벽한 상수를 표현할 때 public static으로 시작하는 변수를 선언해주는 것이 유일한 용도입니다.

protected와 default는 간혹 사용되기는 합니다만, 레퍼런스를 통해서 변수에 직접 접근하는 방식보다는 메소드를 이용해서 접근하는 방식으로 처리되는 경우가 많습니다. 메소드를 이용해서 접근하는 방식은 잠시 후에 다시 설명합니다.

■ **메소드의 접근 제한: public, protected, default, private**

메소드 역시 네 가지 접근 제한자를 다 이용할 수 있지만 간단한 제약이 있습니다. 추상 메소드의 형태로 메소드를 만들 때에는 'private'으로 선언할 수 없습니다.

4 인스턴스 변수와 getter, setter

Java를 처음 공부하는 사람들에게 있어서 좀 아리송한 부분 중의 하나가 "인스턴스 변수는 어째서 private으로 선언하는 것이 정석이고, getter, setter라는 것을 이용해서 접근할 수 있게 한다면 아예 public으로 선언해버리는 것이 더 낫지 않은가?"입니다. 저 역시 그랬던 기억이 있습니다.

4.1 인스턴스 변수는 private?

인스턴스 변수는 특별하지 않은 이상 무조건 private으로 선언합니다. private이기 때문에 외부에서는 접근 자체가 불가능합니다. 따라서 이렇게 만들어 주면 외부 클래스에서는 객체가 가진 데이터에 접근하기 위해서는 별도의 다른 방법을 생각해야 합니다. 역시 그 대답은 메소드가 될 것입니다. 객체가 인스턴스 변수를 private으로 선언하는 데에는 일반적으로 '캡슐화(Encapsulation)'라는 객체지향 프로그래밍의 원리를 구현하기 위해서라고 알려졌습니다. 우리말로는 '정보 은닉'이라고 할 수 있습니다만 이것이 어떻게 우리에게 도움을 주는지에 대해서는 아직 논란의 여지가 있습니다.

객체는 레퍼런스(리모컨)을 이용해서 접근하게 됩니다. 즉 객체의 레퍼런스를 여러 곳이 가지고 있게 되면 여러 곳에서 객체를 조정할 수 있게 됩니다. 이렇게 되면 예상하지 못한 문제가 발생하게 되는데 가장 문제가 되는 것은 정보의 변경입니다. 주방에 있는 라면을 생각해보시면 됩니다.

주방에 라면이 5개 있었는데, 주방에 갈 수 있는 내가 한 개, 동생이 한 개를 먹는다면 내가 기억하는 라면의 숫자는 4개이지만, 실제로 라면의 수는 3개만 남아 있게 됩니다. 객체라는 것은 데이터와 기능으로 구성되는데 이것은 객체의 리모컨을 여러 곳에서 사용하면서 발생하는 문제점이라고 할 수 있습니다. 따라서 객체의 데이터를 누구나 손댈 수 없게 할 필요가 있다는 생각을 하기 시작합니다. 외부에서 아예 볼 수조차 없게 하자는 방식입니다. private이라는 것은 이런 의도로 나온 접근 제한자입니다.

이제 사람들은 데이터를 private으로 보호하기 시작했습니다. 그런데 문제는 주로 데이터성 객체(Value Object, Transfer Object, Data Transfer Object)의 경우에는 데이터를 은닉시켜 버리면 사용하기가 어려워진다는 겁니다. 데이터의 초기화의 경우에는 생성자를 이용할 수 있겠지만, 객체가 가진 데이터를 조회하고, 사용하기에는 너무 어려워지게 됩니다. 인스턴스 변수가 소스에서 뻔히 눈에 보여도 호출할 방법이 없고, 접근이 안 되면 무척 불편합니다. 따라서 아예 조회만 할 수 있게 하는 메소드(getter)와 데이터를 세팅할 수 있는 메소드(setter)를 만들어 두고 사용하자는 임시방편을 생각해냅니다.

예제 | setter, getter 메소드를 적용한 클래스

```
package org.thinker;

class AccessTest {
      private String val1;
      private int val2;

      public String getVal1() {
            return val1;
      }
      public void setVal1(String val1) {
            this.val1 = val1;
      }
      public int getVal2() {
            return val2;
      }
      public void setVal2(int val2) {
            this.val2 = val2;
      }
}
```

4.2 setter, getter

getter, setter를 만들어 내는 것은 너무나 규격화되어 있기 때문에 특별히 설명할 필요도 없고, 아예 이클립스가 이 기능을 지원하기도 합니다(그리고 이런 방식으로 작성된 코드들을 Java Beans라고 하기도 합니다). 하지만, getter/setter를 어떻게 이용하는지에 대한 가이드도 좀 필요할 듯합니다. 우선 제가 지금부터 설명드릴 내용은 인스턴스 변수가 기본 자료형이거나, 문자열 같은 기본적인 데이터에 한정하도록 합니다.

4.2.1 getter는 데이터 복사본을 던져주기 때문에 원래 객체의 데이터를 손상시키지 않습니다.

만일 외부에서 AccessTest의 객체를 만들어서 사용하는 예를 만들어 보면 다음과 같이 만들어 줄 수 있습니다.

예제 | 동일한 패키지에서 AccessTest 객체의 setter, getter를 이용하는 경우

```
package org.thinker;

public class AccessObjTest {
      public static void main(String[] args) {
            AccessTest at = new AccessTest();
            at.setVal2(100);
            int value = at.getVal2();
            System.out.println(value);
      }
}
```

우선은 AccessTest 클래스에서 객체를 만들고 setVal2( )를 이용해서 객체에 100이라는 데이터를 넣어주었습니다. 그리고 나서 at.getVal2( )를 이용해서 데이터를 가져오게 하였습니다. 그런데 자세히 보시면 'int value = at.getVal2( );'를 해서 객체의 데이터를 가져오는 경우 메소드의 리턴 값이라는 것은 실제의 데이터를 손상시키지 않습니다. 그림으로 표현하자면 다음과 같습니다.

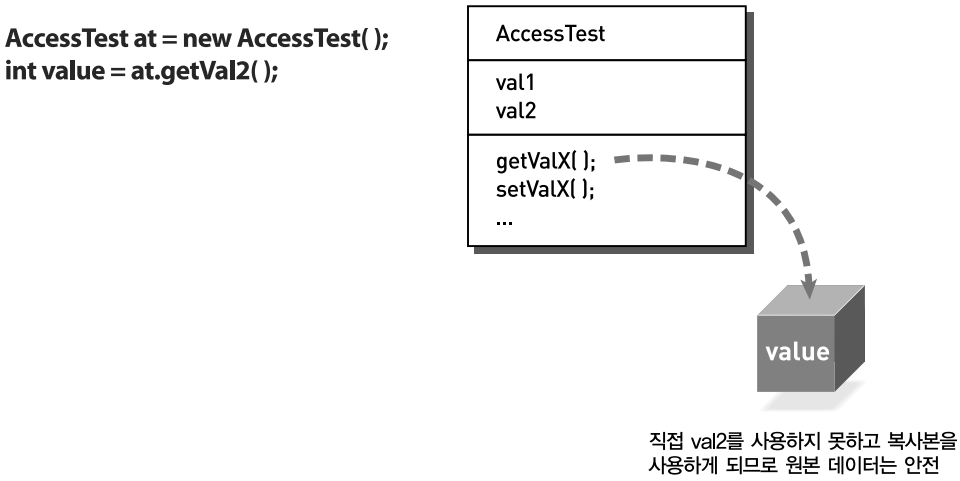

그림 6

코드

```
public int getVal2() {
      return val2;
}
```

Java에서는 변수의 할당이 데이터의 복사라고 말씀드렸습니다. 따라서 지금의 경우도 객체가 가진 데이터의 복사본이 'int value = at.getVal2( );' 실행 시에 전달됩니다. 따라서 getVal2( )를 호출한 쪽에서는 원래 객체의 데이터를 얻어가기는 하지만 원래 객체의 데이터 자체를 얻어가지는 못합니다. 따라서 getter 메소드를 이용하게 되면 객체의 원래의 데이터의 복사본을 얻어 갈 수는 있지만 원래 데이터 자체를 얻어가는 것은 아닙니다. 이렇게 되면 만일 setter를 만들지 않고 getter 메소드만을 사용할 수 있는 쪽에서는 원래 객체의 데이터를 손상시킬 수 없게 되는 겁니다.

간단하게 생성자를 통해서만 데이터를 넣어줄 수 있고(데이터의 injection이라고 표현합니다만), getter 메소드만 있다면(물론 인스턴스 변수가 기본 자료형이나 String인 경우) 외부에서는 객체의 원본 데이터를 변경할 수 없게 됩니다. 이런 경우를 불변(Immutable)이라고 합니다. 앞에서 문자열을 설명하는 부분에서 자세히 설명한 적이 있습니다.

4.2.2 setter 메소드를 이용하면 파라미터를 검증할 수 있다.

setter 메소드는 그렇다면 어떤 작업을 하기 위해서 만드는 것일까요? 당연한 얘기지만 setter 메소드는 객체가 가진 데이터를 변경할 수 있도록 하려고 선언하는 것이 일반적입니다. 문제의 핵심은 인스턴스 변수에 직접 접근하는 것과 무슨 차이가 있느냐는 겁니다.

우선 객체가 가진 인스턴스 변수에 직접 접근하게 되면 어떤 문제가 있는지부터 생각해볼 필요가 있습니다. 인스턴스 변수에 직접 접근하게 되면 실제로 비즈니스 로직에서 허락되지 않는 데이터가 적용될 수 있습니다. 따라서 이런 작업을 허용하게 되면 객체의 데이터를 적절히 보장해줄 수가 없습니다. 예를 들어 계좌라는 객체를 생각해봅시다.

예제 | setter 메소드가 있는 Account 클래스

```
package org.thinker;

public class Account {
	private double amount;

	public double getAmount() {
		return amount;
	}

	public void setAmount(double amount) {
		this.amount = amount;
	}
}
```

만일 위의 경우에 setter 메소드 안에 사용자의 정보를 검사하거나, 영업시간 등의 로직을 추가하는 상황을 생각해 봅니다. 인스턴스 변수에 직접 접근하는 경우라면 아무 때나 어떤 데이터라도 세팅할 수 있겠지만, setter 메소드를 만들어 주면 객체에 대한 데이터 변경 시 일정 로직을 검사할 수 있습니다. 또 필요하다면 setter를 만들 때 접근 제한을 줄 수도 있습니다. 이렇게 되면 동일 패키지라든가, 상속 구조에서만 접근 제한을 한다든지 등의 제한을 걸어줄 수 있습니다.

4.3 setter, getter는 가능하면 피하는 것이 좋습니다.

아무래도 이 글을 쓰면 많은 논쟁의 빌미를 제공할지도 모르지만 가능하면 setter, getter는 피해주는 것이 좋습니다. 사실 setter, getter는 C 언어의 구조체나 C++의 흔적으로 인식되는 것이 일반적인 견해입니다. 이클립스에서 제공하는 setter를 만들게 되면 어쩔 수 없이 객체는 데이터

의 수정에 대해서 무방비 상태가 되는 것을 보실 수 있습니다. 이것은 객체지향의 객체의 데이터를 보호하자는 원칙에서 위반되는 것이 됩니다(Allen Holub의 "Why getter and setter methods are evil"이라는 글을 참고하시기 바랍니다. 마틴 파울러의 "ThoughtWorks Anthology" 역시 같은 견해를 피력합니다).

getter 메소드를 생각해봅시다. getter 메소드의 경우에는 기본 자료형이나 String과 같은 불변(Immutable)한 객체의 getter 메소드일 경우에는 문제가 없습니다만, 객체의 리모컨 자체를 리턴해주게 되면 문제가 심각합니다. 예를 들어 회원 정보라는 객체가 회원 주소라는 객체를 가지는 경우를 생각해 봅시다. Address getAddress( )와 같은 getter 메소드를 만들게 될 것이고, 이렇게 되면 getAddress( ) 메소드를 호출했을 때 회원 정보 객체가 가진 Address 객체의 리모컨과 같은 리모컨이 복사되게 됩니다. 이럴 때 누군가 Address 객체의 리모컨으로 정보를 수정해 버리면 회원 정보 객체가 가진 Address 객체의 정보도 같이 변경되는 문제점이 발생합니다.

요즘 객체지향 프로그래밍과 관련된 이론 책들을 보면 가능하면 getter, setter를 사용하지 않는 구조를 만들어 보자는 움직임이 활발한 경우가 많습니다. 메소드의 공개를 신중하게 결정해야 한다는 겁니다. 이에 대해 언급하자면 너무 길어지기 때문에 우선은 이 정도에서 설명을 멈추는 것이 좋을 듯합니다. 다음은 static에 대해 공부해보도록 하겠습니다.

5 static이라는 특별한 의미

객체지향이라는 패러다임이란 데이터와 기능(로직, 메소드)을 가진 객체들의 커뮤니케이션으로 어떤 작업을 완료하는 것을 의미합니다. 지금까지 객체지향 프로그래밍의 기본적인 구조를 익혔다면 이제부터는 조금 더 부가적인 기능을 알아볼 차례가 되었습니다. 그중에서 가장 먼저 살펴보려고 하는 것이 바로 static이라는 키워드입니다.

■ 객체마다 데이터를 가져도 불편한 때도 있습니다.

Java 언어에서는 객체라는 것이 각자의 데이터를 가질 수 있는 구조가 가능합니다. 이 때문에 가끔은 불편한 점도 발생하게 됩니다. 예를 들어 여러분이 어떤 쇼핑몰을 운영한다고 가정해봅니다. 여러분의 시스템에서 발생하는 매출 현황은 비단 여러분뿐 아니라 여러분의 회사의 모든 직원이 다 알아야 한다고 생각해봅시다.

이 상황을 해결하는 방법에는 쉽게 두 가지를 생각해볼 수 있습니다. 우선은 여러분이 매출 현황을 매일 전 직원에게 발송해주는 방법입니다. 만일 여러분이 이런 작업을 하려면 여러분이 모든 직원의 메일 주소를 알고 있어야만 합니다. 이 방법에는 여러 가지 문제가 있습니다. 예를 들어 여러분이 모르는 매출 데이터가 발생하면 반대로 그 직원은 여러분에게 알려주어야만 하고, 이 결과를 여러분이 다시 보내는 방식으로 진행되게 되어서 무척이나 불편해집니다. 게다가 신입사원이 여러분의 회사에 들어오게 되면 여러분은 반드시 그 직원의 메일 주소를 알아야 합니다. 따라서 뭔가 좀 더 편한 방법이 필요합니다.

이때 고려해보실 만한 다른 방법은 여러분이 모두가 볼 수 있도록 매출 현황을 공개하는 겁니다. 모든 사람이 매출 현황을 볼 수 있고, 매출 현황을 갱신할 수 있는 구조로 간다면 여러분의 입장에서는 무척이나 편리해집니다. 여러분이 모든 직원에게 일일이 메일을 발송해야 하는 번거로움도 없을뿐더러, 각 직원이 매출이 발생하면 매출 데이터를 갱신할 수 있게 됩니다. 따라서 여러분은 별도의 작업이 필요하지 않게 됩니다. 이 비유는 객체지향 프로그래밍에도 마찬가지로 적용할 수 있습니다. 우선 가정은 다음과 같습니다.

- 모든 객체가 동일한 데이터를 참고해야 할 필요가 있다.
- 모든 객체는 데이터에 영향을 줄 수 있다.

이 상황에서 프로그램으로 대처하는 방법을 생각해보면 좋겠습니다. 우선 여러분이 모든 직원에게 메일을 보내는 상황을 예로 들자면 다음과 같이 구성하게 됩니다.

예제

```
public class Employee {

      private String name;
      private int companyIncome;

      public String getName() {
            return name;
      }
      public void setName(String name) {
            this.name = name;
      }
      public int getCompanyIncome() {
            return companyIncome;
```

```
	}
	public void setCompanyIncome(int companyIncome) {
		this.companyIncome = companyIncome;
	}
}
```

예를 들어 위와 같은 클래스가 있다고 가정해봅니다. 이 클래스의 구조로는 각 객체가 company Income이라는 변수로 회사의 매출 데이터를 가지게 되어 있습니다. 그럼 만일 이 클래스에서 객체가 100개 만들어져 있는 상황에서 매출 데이터를 동일하게 맞추어야 하는 상황이라면 어떻게 될까요? 별다른 방법을 쓰지 않는 이상은 그냥 모든 객체의 메소드를 호출해서 변경하는 방법을 사용할 수밖에 없습니다.

> static이라는 의미는 '정적인, 움직이지 않는다'는 뜻입니다. 메모리에서 고정되기 때문에 붙은 이름이지만, 여러분이 실제로 소스에서 static을 사용한다는 의미는 모든 객체가 '공유'한다는 의미입니다.

5.1 static의 의미 ① 공유

이제 static이 이런 문제를 해결할 때 얼마나 도움이 되는지 느껴보도록 합시다. 여러분이 어떤 변수에 static을 붙이게 되면 이 변수는 모든 객체가 다 같이 공유하는 변수가 됩니다. 즉, 위와 같은 문제에서 어떤 데이터를 모든 객체가 같이 사용하게 하고 싶다면 static이라는 키워드가 바로 방법입니다. 좀 더 예제를 간단히 하기 위해서 은행에 있는 '번호표 기계'를 생각해보면 좋겠습니다. 은행에 있는 번호표 기계는 매 버튼을 누를 때마다 다른 숫자를 반환해줍니다. 이 문제를 예제로 만든다면 다음과 같이 만들 수 있습니다.

예제 | 인스턴스 변수를 사용하는 번호표 기계

```
public class OrderUtil {

	private int count = 0;

	public void pressButton(){
		count++;
		System.out.println("고객님의 번호는 " + count+" 입니다.");
	}
}
```

코드를 보시면 이해하시겠지만, 이 클래스에서 객체를 하나 만들고 pressButton( ) 메소드를 실행할 때마다 count 값은 증가하여 출력됩니다.

■ 객체가 하나일 때는 문제가 없습니다만...

위의 코드는 은행마다 번호표 기계를 하나씩만 두는 경우에는 아무런 문제가 없습니다. 하지만, 번호표 기계를 여러 대 만들어서 사용하게 되면 어떻게 될까요?

예제 | 객체를 두 개 만들면 각 객체가 데이터를 가지게 됩니다.

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

      OrderUtil u1 = new OrderUtil();
      OrderUtil u2 = new OrderUtil();

      u1.pressButton();
      u2.pressButton();
}
```

```
고객님의 번호는 1 입니다.
고객님의 번호는 1 입니다.
```

위의 main 메소드에서는 객체를 두 개 생성해서 처리하고 있습니다. 중요한 점은 기계가 두 대일 경우에는 1번이라는 데이터가 동일하게 나온다는 겁니다. 만일 번호표 기계 100대를 놓는다는 좀 억지스러운 상상을 해봅시다. 100명의 손님이 자신이 1번이라고 주장하는 웃지 못할 사태가 벌어집니다. 이 사태의 가장 근본적인 문제는 각 객체가 자신만의 데이터를 갖는다는 것에 있습니다. 그렇다면, 반대로 이 문제를 해결하기 위한 가장 손쉬운 해결책 역시 모든 객체가 같은 데이터를 공유할 수 있게 하면 될 듯합니다.

■ static이 붙은 변수는 객체 간에 공유됩니다.

여러분이 어떤 변수를 모든 객체가 공유하게 하는 가장 손쉬운 방법은 공유되기를 바라는 변수에 'static'이라는 키워드를 붙이는 겁니다. 위의 코드를 static을 붙여서 변경해보도록 합니다.

예제 | static이 붙은 변수는 객체 간에 공유됩니다.

```
public class OrderUtil {
      //static이 붙으면 공유
```

```
        private static int count = 0;

        public void pressButton(){
                count++;
                System.out.println("고객님의 번호는 " + count+" 입니다.");
        }

        public static void main(String[] args) {

                OrderUtil u1 = new OrderUtil();
                OrderUtil u2 = new OrderUtil();
                u1.pressButton();
                u2.pressButton();
        }
}
```

```
고객님의 번호는 1입니다.
고객님의 번호는 2 입니다.
```

위의 소스에서 변경된 부분은 단지 변수를 선언하는 부분에 'static'이라는 키워드가 붙은 것뿐입니다. 그런데 결과를 보면 이전과는 다르게 값이 증가한 것이 보입니다. static은 이처럼 어떤 변수를 모든 객체가 공유하는 값으로 변경해버리기 때문에 아무리 많은 객체가 있더라도 같은 데이터를 사용하게 하는 공유의 개념을 완성할 수 있습니다.

5.2 static이 붙은 변수는 어째서 공유가 될까?

static이라는 키워드가 붙으면 어떤 원리에 의해서 변수가 모든 객체가 같은 값을 사용하게 될까요? 그리고 언제 이런 static을 이용해서 소스를 작성해야 할까요? 이러한 문제를 좀 더 생각해보도록 합시다.

■ static이 붙은 변수는 클래스 변수라고 부릅니다.

우리는 객체가 만들어지려면 반드시 클래스가 있어야만 한다는 사실을 알고 있습니다. 따라서 이 원칙을 기본으로 추론해보면 다음과 같은 사실이 가능합니다.

• 모든 객체는 자신이 어디에서 생산되었는지 해당 클래스를 알 수 있을지도 모른다.

아직 가설이기는 하지만 솔직히 먼저 결론을 말씀드리면 위의 가설은 맞습니다. 즉 어떤 객체는 반드시 자신이 어떤 클래스에 속하는지를 알아낼 수 있습니다. 모든 객체가 자신이 속한 클래스

를 알 수 있다는 것을 다른 말로 하면 모든 객체는 자신이 속한 클래스의 정보를 공유한다는 뜻입니다. 사실 static의 비밀은 바로 이곳에 있습니다. static의 가장 중요한 내용은 객체와 묶이는 데이터가 아니라 클래스와 묶이는 데이터라는 겁니다. 따라서 static이 붙은 변수를 클래스 변수라고 부릅니다.

> static이 붙은 변수는 객체의 리모컨(레퍼런스)을 이용해서 사용하는 일반적인 객체지향 프로그래밍과는 달리 클래스의 변수이기 때문에 그냥 클래스 '이름.클래스' 변수라는 방식으로 사용하게 됩니다.

static을 사용하는 변수는 객체가 아닌 클래스의 변수이기 때문에 별도의 객체를 사용할 필요 없이 그냥 사용할 수 있습니다.

코드

```
System.out.println(OrderUtil.count);
```

■ **메모리의 구조도 조금 더 알아보면 좋습니다.**

static을 조금 더 깊이 알아보기 위해서 메모리의 구조를 보시면 조금 더 도움이 됩니다. 일반적인 Java 책들과는 달리 메모리의 구조를 아주 극단적으로 단순화시켜서 보면 Java의 메모리는 크게 객체 영역과 비객체 영역으로 볼 수 있습니다.

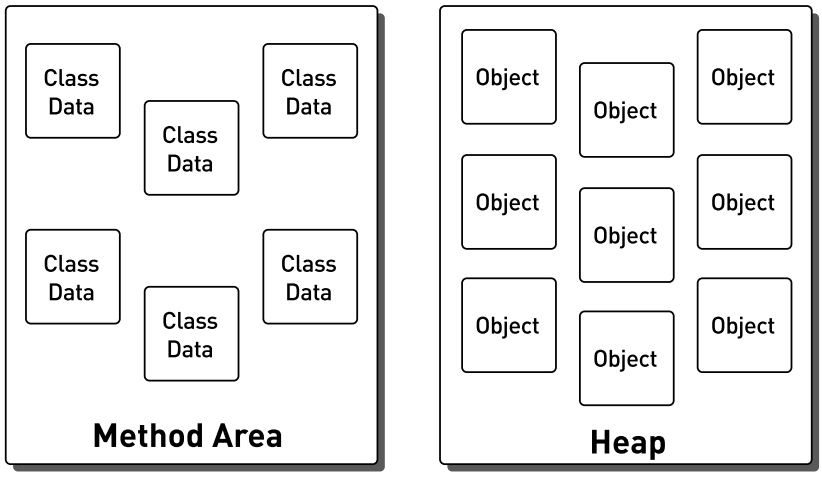

그림 7

앞의 그림은 Java 가상 머신(이하 JVM)의 Runtime Data Area의 구조입니다. 간단히 그려서 Heap이라는 영역은 객체의 영역으로 선언된 것이 보입니다. 그리고 클래스는 Heap이 아닌 Method Area 안으로 들어가게 되어 있습니다만 편의상 비객체 영역이라고만 표현하겠습니다.

객체의 영역이란 말 그대로 객체들이 만들어지고 살다가 죽는 영역입니다. 이 영역을 Heap 영역이라고 합니다만 이 영역에서 가장 중요한 존재는 다름 아닌 가비지 컬렉터(Garbage Collector)입니다. Java 언어는 개발자가 스스로 메모리를 관리하는 것이 아니라 가비지 컬렉터가 동작하면서 더는 사용하지 않는 메모리를 회수하게 됩니다. 때문에 어느 한 편으로는 개발자가 메모리를 해제하는 불편함이 없지만, 다른 한편으로는 최소한의 메모리를 사용해야 하는 경우에 개발자가 직접 컨트롤 할 수 없다는 문제도 있습니다.

비객체 영역인 Method Area는 클래스가 메모리상에 올라가는 영역입니다. 이 영역은 가비지 컬렉터의 영향을 받지 않고 메모리에 상주합니다. 간단히 생각해보면 모든 객체는 클래스에서 나오는데 이 클래스가 나도 모르게 메모리상에서 지워지면 난감한 일이 발생할 수 있다는 겁니다. 비객체 영역은 가비지 컬렉터의 영향으로부터 자유롭고, 메모리에 상주하게 되어 있는데 이런 상주의 의미를 'static(정적인, 고정된)'이라는 뜻으로 사용합니다.

■ **static이 붙은 부분은 클래스가 메모리상에 로딩되면서 같이 올라갑니다.**

앞에서 나온 내용을 종합해보면 'static' 키워드가 붙으면 메모리상에서 다르게 처리된다는 겁니다. static이 붙은 변수를 클래스 변수라고 하는 것은 변수가 존재하는 영역이 클래스가 존재하는 영역과 같기 때문입니다. Java에서 모든 객체는 결과적으로는 동일한 클래스에서부터 생성되기 때문에 동일한 클래스에서 나온 모든 객체는 자신이 속한 클래스에 대해서 접근할 수 있습니다. static 변수는 이런 방식을 통해서 모든 객체가 동일한 데이터를 사용할 수 있는 공유의 개념을 완성시킵니다.

5.3 static의 의미 ② 객체와 무관한 메소드와 함수라는 개념

static에 대해서 가장 먼저 알아야 하는 개념이 바로 공유라는 개념이라면 그다음으로 중요한 개념은 "메소드 앞에 static이 붙으면 어떻게 되는가?"라는 겁니다. static이 붙으면 공유된다고 생각해보면 static이 붙은 메소드는 모든 객체가 공유하는 메소드라는 공식이 가능합니다. 문제는 여기에 있을 겁니다. 대체 왜 모든 객체가 공유하는 메소드를 작업하게 되는가? 어떤 때 모든 객체가 공유하는 메소드를 만드는가에 대해서 생각해볼 필요가 있습니다.

객체지향 프로그래밍에서는 객체라는 독립적인 존재가 각자의 데이터를 가지고 있습니다. 메소드라는 것은 어찌 보면 이런 데이터를 어떻게 변경하거나 사용하는 로직의 단위라고 볼 수가 있습니다. 객체의 모든 메소드는 객체의 데이터에 따라서 좌지우지되도록 하는 것이 가장 일반적이라고 한다면 100개의 객체를 만들어서 다른 데이터를 가지고 있다면 100가지의 결과가 나올 수도 있다는 것을 의미합니다. 조금 더 구체적인 예를 들어서 설명해보는 것이 좋을 듯합니다.

만일 여러분이 메일을 발송하는 메소드를 하나 만들었다고 가정해봅니다. 각 객체가 다른 데이터를 가질 수 있기 때문에 메일을 발송하는 메소드는 각자 다른 상대방에서 메일을 전송하는 것이 가능해집니다. 여러분이 이 상황에서 모든 객체가 동일한 대상으로 메일을 보내게 하고 싶다면 어떻게 해야 할까요? 간단하게 소스를 보면서 생각해보도록 합니다.

예제 | 인스턴스 변수와 객체의 메소드를 이용한 방식

```
public class MailSender {

      private String destination = "aaa@aaa.co.kr";

      public void sendMail(){

            System.out.println("메일을 전송합니다.");
            System.out.println("목적지 :" + destination);

      }
}
```

우선 위의 소스 방식이라면 가장 먼저 생각할 수 있는 방법은 destination 변수를 static으로 변경하는 겁니다. static은 모든 객체가 동일한 데이터를 사용하기 때문에 MailSender 클래스를 아무리 많이 객체 생성해도 동일한 값을 사용할 수 있게 됩니다. 그런데 이 방법보다 더 나은 방법은 없을까요? 단지 변수를 static으로 변경하는 것에는 어떤 단점은 없을까요?

■ **메소드는 객체의 데이터를 사용하기 때문에 객체의 현재 상황에 의해 좌지우지됩니다.**

어떤 메소드를 객체의 메소드로 만들게 되면 그 안에 객체의 인스턴스 변수에 의해서 변경되는 상황이 발생할 수 있습니다. 예를 들어 가장 쉽게 생각해볼 수 있는 상황은 만일 어떤 객체가 이미 한번 메일을 보냈다면 다시 보낼 수 없게 한다고 가정해봅시다.

예제 | 객체의 메소드는 결국 객체의 데이터에 의해서 결정됩니다.

```
public class MailSender {

      private String destination = "aaa@aaa.co.kr";

      private boolean sendFlag = false;

      public void sendMail(){

            if(sendFlag == false){
                  System.out.println("메일을 전송합니다.");
                  System.out.println("목적지 :" + destination);
                  sendFlag = true;
            }else {
                  System.out.println("메일을 이미 보냈습니다.");
            }
      }
}
```

위의 소스를 보면 객체의 인스턴스 변수인 sendFlag라는 변수에 의해서 결과적으로 메소드가 좌지우지되는 것이 보입니다. 이 문제는 결국 객체마다 데이터를 가지는 객체지향의 가장 근본적인 문제일지도 모릅니다. 모든 객체가 자신만의 데이터를 가지고 있으니 발생하는 문제니까요. 이처럼 메소드 역시 객체의 데이터에 의해서 좌우되는 상황을 벗어나기 위해서 Java에서의 static은 메소드에도 적용할 수 있게 합니다.

■ **static 메소드는 결과적으로 모든 객체가 완벽하게 똑같이 동작하는 메소드를 위해서 사용합니다.**

클래스에서 나온 객체의 데이터에 영향받지 않고 완벽히 동일하게 동작하는 메소드를 만들고 싶다면 static을 이용해서 객체의 메소드를 처리하면 됩니다. 메소드 앞에 static이 붙으면 모든 객체가 클래스에 정의된 방법대로만 사용하게 됩니다. 이 문제는 좀 더 현실적인 해결책으로 생각해보면 객체가 가진 데이터에 의해서 좌우되고 싶지 않다는 뜻입니다. 객체가 어떤 데이터를 가지든 객체의 데이터의 영향 없이 동작하게 만든다면 언제나 완전하게 동일하게 동작하는 것을 보장해줄 수 있습니다.

■ Integer.parseInt( )는 static 메소드입니다.

예를 들어 Integer.parseInt( ) 메소드가 바로 이런 static이 붙은 메소드의 가장 흔한 예입니다. 이 메소드의 의미는 "문자열을 int 값으로 변경한다."라는 뜻입니다. 그런데 이 메소드를 조금만 더 곰곰이 생각해보면 언제 어디서나 완전히 동일하게 동작한다는 겁니다. 즉 Integer.parseInt( )는 아예 메소드 자체가 어느 상황에서든 객체의 데이터에 의해서 영향받지 않고 같은 기능만을 제공합니다.

■ static은 C 언어의 함수 개념과 유사합니다.

C 언어의 함수라는 개념을 기억하십니까? 단순하게 입력 값이 있고, 출력하는 결과가 있는 함수라는 존재는 객체지향 프로그래밍에서 데이터를 객체가 가지도록 변경하는 가장 큰 이유가 되었던 존재입니다.

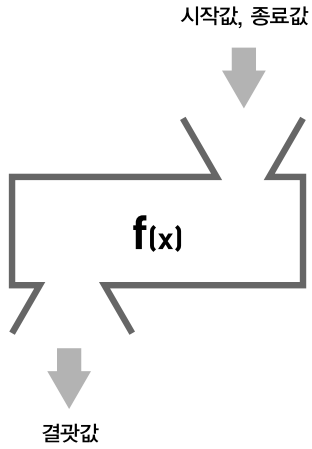

그림 8

static 메소드는 마치 C 언어의 함수와 유사합니다. 완벽하게 언제나 동일하게 동작하는 것이 보장되기 때문에 사용자는 객체의 데이터를 신경 쓰지 않아도 됩니다. 이런 의미에서 보면 static 메소드라는 것은 객체가 가지는 메소드라기 보다는 클래스 자체의 메소드라는 의미로 생각해볼 수 있습니다.

5.4 static의 의미 ③ 단 한 번만 동작하게 하는 static { }

지금까지 static을 변수에 사용할 때와 메소드에 static을 이용하는 경우를 봤지만, 한 가지 용도

를 더 알아두면 좋습니다. static을 이용해서 {}를 작성하게 되면 조금 특이하게 동작하게 됩니다.

예제 static 블록은 단 한 번만 동작합니다.

```
public class BlockEx {

	static{
		System.out.println("static 블럭");
	}

	public void doA(){
		System.out.println("AAAA");
	}
}
```

위의 코드를 보면 static으로 처리된 {}이 보입니다. Java에서 모든 영향력의 경계선이 {}라는 것을 생각해보면 static {}는 static이라는 키워드의 영향을 받는 부분이라는 겁니다. 그럼 어떤 방식으로 이 영향을 받게 될까요? static이라는 키워드가 객체가 아닌 클래스와 관련이 있다는 것을 생각해보면 static {} 역시 클래스와 무슨 관련이 있다고 생각하시면 맞습니다. static {}는 클래스가 메모리상에 올라가는 시점에만 동작하게 됩니다. 위의 소스에 main 메소드를 추가해서 여러 번 객체를 생성하는 코드를 작성해보면 어떨까요?

예제 클래스 로딩 후에 static 블록은 바로 실행됩니다.

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

	BlockEx ex1 = new BlockEx();
	BlockEx ex2 = new BlockEx();
	BlockEx ex3 = new BlockEx();

	ex1.doA();
	ex2.doA();
	ex3.doA();
}
```

```
static 블럭
AAAA
AAAA
AAAA
```

소스의 실행 결과를 보면 객체 생성을 3번 했고, 메소드의 호출을 했지만 중요한 점은 BlockEx의 static {} 이 가장 먼저 실행되었다는 겁니다. static {}의 실행되는 시점은 클래스가 메모리상에 올라갈 때입니다. 즉 우리가 프로그램을 실행하면 필요한 클래스가 메모리상에 로딩되는 과정을 거치게 됩니다. 그리고 한번 로딩된 클래스는 특별한 일이 발생하지 않는 이상 메모리상에서 객체를 생성할 수 있도록 메모리에 상주하게 됩니다. static {}은 바로 이 시점에 동작합니다. 클래스가 메모리상에 올라가자마자 실행되면서 필요한 처리를 하게 됩니다. 따라서 static{}은 객체의 생성과는 관계없이 클래스가 로딩되는 시점에 단 한 번만 실행하고 싶은 처리를 하기 위해서 사용합니다.

5.5 객체의 영향을 받지 않기 때문에 static은 굳이 객체를 통해서 사용할 이유가 없습니다.

어떤 변수나 메소드 앞에 static이 나온다는 의미는 결과적으로 객체의 데이터나 메소드의 영향을 받지 않고 완벽하게 동일한 공유의 개념을 완성하는 겁니다. 따라서 굳이 지금껏 사용하던 것처럼 객체의 리모컨(레퍼런스)을 이용해야 하는 필요성은 조금 부족합니다. 어차피 객체보다는 클래스와 더 가깝기 때문입니다.

■ static이 붙은 변수와 메소드는 클래스의 이름만으로 충분합니다.

이제 static이 붙은 메소드를 하나 만들어 보고 사용해보려 합니다.

예제 | static이 붙은 calcCircle 메소드

```
public class AreaEx {
      public static double calcCircle(double radius){
            return (radius * radius) * Math.PI;
      }
}
```

calcCircle( ) 메소드를 유심히 살펴봅시다. 우선은 메소드의 선언 앞에 static이 붙어 있습니다. 여러분이 어떤 메소드 앞에 static이 붙어 있는 것을 보시면 무조건 "객체와 무관하다." 혹은 "객체의 영향 받지 않는 메소드이다."라고 생각하시면 됩니다. 위의 소스의 내부는 들어온 반지름의 제곱에 Math.PI라는 3.14xxx 값을 곱하는 기능입니다. 생각해보시면 calcCircle( ) 메소드가 다르게 동작해야 할 가능성은 거의 없습니다. 모든 경우에 완벽하게 동일하게 동작해주어야만 합니다. 이런 상황에서는 static을 이용해서 경우에 따라서 다르게 동작할 수 없게 하는 것이 정상적인 static의 사용법입니다.

■ **static을 사용할 때 객체는 필요 없습니다.**

앞에서 만든 AreaEx를 이용하는 화면을 작성해봅시다.

예제 'static 클래스 이름.함수(메소드) 이름'으로 사용합니다.

```
import java.util.Scanner;

public class AreaUI {
      public static void main(String[] args) {
            Scanner scanner = new Scanner(System.in);
            System.out.println("반지름을 입력해주세요.");
            double r = scanner.nextDouble();
            double area = AreaEx.calcCircle(r);
            System.out.println("넓이는 " + area);
      }
}
```

```
반지름을 입력해주세요.
10 Enter
넓이는 314.1592653589793
```

위의 코드를 보시면 중간에 AreaEx.calcCircle( ) 메소드를 활용하는 부분이 보입니다. 그런데 여기서 재미있는 사실은 객체를 생성하지 않고, 바로 calcCircle( ) 메소드를 활용하는 것이 보입니다. staic은 객체마다 다른 데이터를 가지고 동작하는 것을 막고 완벽하게 동일하게 동작하는 것을 보장하기 위해서 사용하기 때문에 굳이 객체를 사용해야 하는 이유가 없으므로 바로 '클래스 이름.변수'나 '클래스 이름.메소드'를 사용할 수 있습니다. 굳이 객체를 사용하지 않아도 된다는 의미는 객체를 사용하는 기존의 방식도 가능하기는 하다는 뜻입니다.

코드

```
//double area = AreaEx.calcCircle(r);
AreaEx ex = new AreaEx();
double area = ex.calcCircle(r);
```

위의 코드는 static의 전형적으로 호출하는 방법을 굳이 객체를 활용하는 방법으로 사용해본 것입니다. 실제로 이클립스에서 위와 같은 코드로 변경해보면 노란색 경고창만 하나 뜨게 되는데 이것은 "굳이 staic한 메소드를 객체를 통해서 사용할 필요 없다."라는 뜻입니다.

5.6 static은 이런 기준으로 사용합니다.

static에 대해서 얘기하고 있지만, 현실적으로 언제 static을 사용해야 하는지 감을 못 잡을 때가 더 많기 때문에 제가 개인적으로 사용하는 가이드를 몇 가지 제시해드릴까 합니다.

■ 객체가 데이터를 보관할 필요가 없다면 staitc으로 메소드를 만들어도 됩니다.

Math.random( )을 기억하실겁니다. 0에서 0.99999...에 해당하는 소수를 발생시키는 기능입니다. 그런데 사실 Java에는 Random( )이라는 클래스가 있습니다. 또한, Random 클래스에는 nextDouble( )이라는 메소드도 있습니다.

예제 | 데이터를 따로 보관하는 경우에는 static을 쓰지 않습니다.

```
import java.util.Random;

public class RandomEx {
      public static void main(String[] args) {
            double a = Math.random();
            Random random = new Random();
            double b = random.nextDouble();
            System.out.println(a);
            System.out.println(b);
      }
}
```

위의 코드는 Math.random( )을 이용하는 방식(static 방식)과 Random 클래스의 nextDouble( ) 메소드를 활용하는 방식의 예를 보여줍니다. 결과는 마찬가지로 임의의 소수를 발생시킵니다. 그렇다면, 왜 이런 동일한 기능을 static 방식으로도 제공하고, Random이라는 클래스로도 제공하고 메소드도 제공할까요? 가장 중요한 원인은 바로 Random 클래스는 객체를 생성할 때 데이터를 가지게 할 수 있다는 점입니다. seed 값이라고 하는데 그 의미를 아실 필요는 없고, 다만 소수를 생성하는 판단 기준을 객체마다 다르게 지정할 수 있다는 겁니다.

'Random r1 = new Random(11), Random r2 = new Random(22);'와 같은 방식은 각 객체가 자신만의 seed 값을 가지게 합니다. 따라서 객체가 데이터를 가지기 때문에 동일한 메소드를 호출해도 다른 결과를 만들어 낼 수 있다는 것을 의미합니다. 반면에 Math.random( )은 특정한 데이터를 기준으로 하는 것이 아니라, 완전히 매번 동일한 방법으로 동작하게 됩니다. 결론적으로 말하자면 어떤 메소드가 완전히 객체의 데이터와 독립적인 결과를 만들어내야 하는 상황에서는

static 메소드로 만드는 것이 더 나은 선택이라는 겁니다.

■ **계산기를 만든다면 static을 이용해야 할까? 객체로 만들어야 할까?**

흔히 객체지향 프로그래밍을 처음 배우면서 만들어 보는 것 중의 하나가 바로 계산기입니다. 계산기를 만들 때 다음과 같이 구성합니다.

예제 | 객체를 이용하는 방식으로 계산기

```
public class Calculator {
      public double add(double x, double y){
            return x + y;
      }
      public double minus(double x, double y){
            return x - y;
      }
      public double multi(double x, double y){
            return x * y;
      }
      public double divide(double x, double y){
            return x / y;
      }
}
```

위의 코드를 곰곰이 보면 한 가지 의문이 생깁니다. 매번 모든 객체가 동일한 메소드만 제공하고, 굳이 객체마다 따로 데이터를 누적하지 않는 이상 계산기를 객체로 만들어야 하는 이유가 있을까요? 계산기마다 다른 데이터를 가지는 상황이라면 계산기의 모든 메소드는 인스턴스 변수를 활용하게 될 겁니다. 그런데 굳이 완벽하게 객체가 가진 데이터의 영향을 받지 않는 기능이라면 그냥 static으로 만들어도 아무런 문제가 없어 보입니다.

예제 | static 방식을 이용하는 계산기

```
public class Calculator {
      public static double add(double x, double y){
            return x + y;
      }
      public static double minus(double x, double y){
            return x - y;
      }
      public static double multi(double x, double y){
```

```
            return x * y;
      }
      public static double divide(double x, double y){
            return x / y;
      }
}
```

제가 실무에서 static을 이용할 것인지 아닌지를 판단하는 데 있어서 가장 중요한 것은 객체마다 가지는 데이터를 활용해야 하는 경우인가, 그렇지 않은가입니다. Integer.parseInt( ), Math.random( )을 생각해보면 객체가 가진 데이터와는 완전히 무관합니다. 그냥 단순히 어떤 로직을 묶어둔 존재에 가깝습니다.

객체마다 가질 수 있는 데이터를 활용하지 않기 때문에 static 메소드를 흔히들 '클래스 메소드'라고 하거나 C 언어의 전역 함수의 개념으로 보기도 합니다. 개인적으로 전역 함수라고 보기에는 약간의 무리가 있다고 생각합니다만 static이 붙은 메소드는 함수처럼 그저 단순히 input과 output으로만 구성된다는 겁니다.

■ **객체들이 공유하는 데이터를 사용할 때에도 static 메소드를 이용합니다.**

객체의 데이터와 무관한 메소드는 static 메소드로 만든다는 기준 외에 굳이 하나의 기준을 더 들자면 static한 변수를 사용할 때에는 static 메소드를 이용한다는 겁니다. 하지만, 이것은 이미 컴파일 시점에 포착되기 때문에 굳이 기준이라고 하기에는 좀 무리가 있습니다. 앞에서 만든 MailSender 클래스의 메소드를 static으로 변경하면 아래와 같이 에러가 발생합니다.

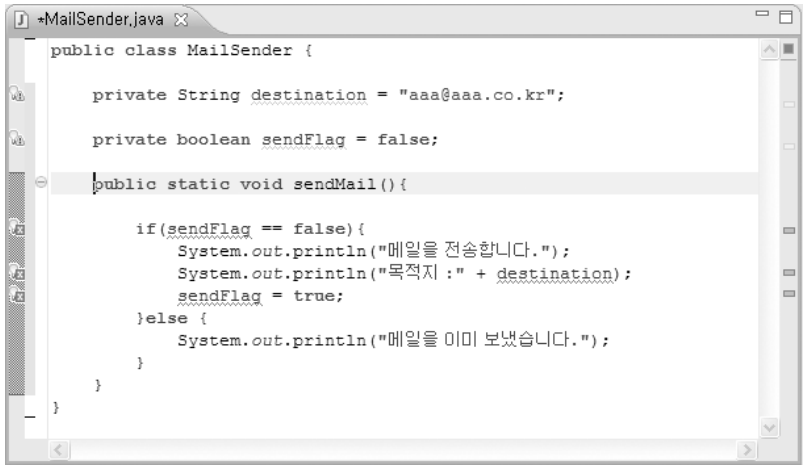

그림 9

앞에서 컴파일 에러가 발생하는 부분을 보면 전부 sendFlag라는 객체의 변수를 활용하는 모든 코드에서 발생합니다. static이라는 키워드가 붙은 존재는 객체와 무관해지기 위해서 사용합니다. 그런데 static한 메소드에서 어떤 특정한 객체의 데이터를 활용하게 되면 몇 가지 문제가 생깁니다. 가장 중요한 한 가지만 생각해보도록 합시다.

> **만일 static한 공통된 기능의 메소드에서 어떤 하나의 객체의 데이터를 사용하게 된다면 대체 어떤 객체의 데이터를 활용해야 할까?**

static이 붙으면 객체와는 무관해집니다. 따라서 그 안에서 특정한 하나의 객체의 데이터를 활용하게 되면 문제가 되는데, 가장 근본적인 것은 만들어진 수많은 객체 중에서 어떤 객체의 데이터를 활용해야 할지 알 수 없게 된다는 점입니다. 이런 이유 때문에 static이 붙은 메소드 안에서는 인스턴스 변수를 활용할 수 없도록 컴파일러가 확인하게 됩니다. static은 다음과 같은 몇 가지 법칙이 존재합니다.

- static이 붙은 변수들은 객체들이 다 같이 공유하는 데이터를 의미한다.
- static이 붙은 메소드는 객체들의 데이터와 관계없는 완벽하게 공통적인 로직을 정의할 때 사용한다.
- 따라서 static 메소드에서는 인스턴스 변수나 객체의 메소드를 사용할 수 없다.

5.7 static은 속도는 빠르지만, 메모리가 회수되지 않기 때문에 주의해야 합니다.

static은 앞에서 설명한 내용을 기반으로 보면 객체를 이용하는 방식이 아니라, C 언어처럼 그냥 클래스 안에 공통으로 사용하는 데이터와 메소드를 활용하는 형태입니다. 그런데 이 점을 냉정하게 생각해보면 속도 면에서 static을 이용하는 것이 오히려 객체를 이용하는 방법보다는 더 나아보입니다. 클래스가 매번 필요할 때마다 객체를 생산할 수 있다는 점의 최대의 무기는 바로 객체마다 그 값이 다른 데이터를 보관한다는 데에 있습니다. 반면에 사용할 때마다 객체를 생산해야 한다는 단점이 존재하는 것도 사실입니다. Java에서 객체의 생성 문제는 장단점을 가지는 미묘한 문제입니다. 객체 생성을 하기 때문에 분명히 편해지는 것도 있지만, 반대로 객체 생성이라는 것 자체가 메모리를 차지하는 필수적인 과정을 거치기 때문에 어쩔 수 없이 메모리를 많이 사용하게 되는 문제를 피할 수 없습니다.

반면에 static을 유심히 보면 객체와 관련 있기보다는 오히려 클래스와의 관련이 있습니다. 객체의 리모컨(레퍼런스)을 통해서 접근하는 방식이 아니고 바로 클래스가 로딩되는 영역과 관계가 있기 때문에 별도의 객체를 만들어야 하는 메모리의 소비를 줄일 수 있습니다. 이 때문에 흔히 static을 '양날의 검'에 비유하기도 하는데 이는 속도 면에서는 객체를 생성해서 호출하는 방식보다 월등히 빠를 수 있기 때문입니다. static이 객체 생성이 없어서 속도를 향상시킬 수 있는 반면에 반대급부로 단점도 존재하는데 이 단점은 바로 static이라는 키워드가 붙은 변수는 가비지 컬렉션의 대상이 아니라는 데에서 시작합니다.

static이 붙은 변수 자체는 객체와 같이 생성되지 않습니다. static을 클래스 변수라고 말하듯이 static이 붙은 변수는 클래스와 같은 영역에 생기고, 클래스와 동일하게 메모리에서 항상 상주하게 됩니다. 이 때문에 static으로 어떤 데이터나 객체를 연결해서 사용하게 되면 메모리 회수가 안 되는데, 더 중요한 것은 이 문제가 단순하게 개발해서 결과만 보는 동안에는 느껴지지 않는다는 겁니다. 실제로 시스템이 며칠, 몇 주, 몇 달 정도 가동되면서 점점 느려지는 현상을 보이면 그제야 눈에 들어오기 때문에 가능하면 static으로 사용하는 것은 조심해야만 합니다.

5.8 static과 상수

static에 대해서는 클래스 변수, 클래스 메소드라는 개념으로 알아 두시면 좋지만, 굳이 하나 더 필요한 것을 말씀드리자면 static을 이용해서 상수를 처리하는 기법을 알아두시면 좋습니다. 아직 final이라는 키워드는 설명하지 않았지만, 이 기회에 조금 미리 보면 도움이 될 듯합니다. static은 클래스 변수로 객체의 상관없이 유지되는 데이터이고, 별도의 객체 생성 없이도 마음대로 사용할 수 있기 때문에 외부에서 누구나 사용하는 데이터를 정의하기도 합니다.

예제 | static final을 이용하는 상수의 정의

```
public class ConstEx {

      public static final int RECTANGLE = 1;
      public static final int TRIANGLE = 2;

      public double getArea(int type, int width, int height){

            double area = 0.0;

            if(type == RECTANGLE ){
                  area = width * height;
```

```
            }else if(type == TRIANGLE){
                  area = (width * height)/2;
            }
            return area;
      }
}
```

앞의 코드를 보면 getArea(타입, 가로, 세로)의 값을 입력해주면 자동으로 계산되게 됩니다. 소스의 맨 위를 보면 지금까지와 달리 모든 변수를 대문자로 선언한 점도 조금 특이해 보입니다. 만일 위와 같은 어떤 코드를 의미하는 데이터를 매번 숫자로 처리하려면 모든 사용자가 1이라는 값이 사각형인지, 삼각형인지 알아야 한다는 것을 의미합니다. 하지만, 위와 같은 코드로 작성하게 되면 외부에서 사용할 때 굳이 코드의 의미를 알지 못해도 사용할 수 있게 됩니다.

예제 | static 변수는 상수로 많이 사용합니다.

```
import java.util.Scanner;

public class ConstUI {
      public static void main(String[] args) {
            ConstEx ex = new ConstEx();
            int w = 100;
            int h = 50;
            System.out.println(ex.getArea(ConstEx.RECTANGLE, w, h));
            System.out.println(ex.getArea(ConstEx.TRIANGLE, w, h));
      }
}
```