

# [Java의 정석]제1장 자바를 시작하기 전에

## 자바의정석

2012/12/22 17:01

<http://blog.naver.com/gphic/50157739427>

### 1.자바(Java Programming Language)

#### 1.1 자바란?

자바는 썬 마이크로시스템즈(Sun Microsystems, Inc 이하 썬)에서 개발하여 1996년 1월에 공식적으로 발표한 객체지향 프로그래밍 언어이다.

자바의 가장 중요한 특징은 운영체제(Operating System, 플랫폼)에 독립적이라는 것이다. 자바로 작성된 프로그램은 운영체제의 종류에 관계없이 실행이 가능하기 때문에, 운영체제에 따라 프로그램을 전혀 변경하지 않고도 실행이 가능하다.

이 러한 장점으로 인해 자바는 다양한 기종의 컴퓨터와 운영체제가 공존하는 인터넷 환경에 적합한 언어로써 인터넷의 발전과 함께 많은 사용자층을 확보할 수 있었다. 또한 객체지향개념과 기존의 다른 프로그래밍언어, 특히 C++의 장점을 채택하는 동시에 잘 사용되지 않는 부분은 과감히 제외시킴으로써 비교적 배우기 쉽고 이해하기 쉬운 간결한 표현이 가능하도록 했다.

자바는 풍부한 클래스 라이브러리(Java API)를 통해 프로그래밍에 필요한 요소들을 기본적으로 제공하기 때문에 자바 프로그래머는 단순히 이 클래스 라이브러리만을 잘 활용해도 강력한 기능의 자바 프로그램을 작성하는 것이 가능하다.

지금도 자바는 썬의 전폭적인 지원 하에 꾸준히 자바의 성능을 개선하여 새로운 버전을 발표하고 있으며, 모바일(J2ME)이나 대규모 기업환경(J2EE), XML 등의 다양한 최신 기술을 지원함으로써 그 활동영역을 넓혀 가고 있다.

#### 1.2 자바의 역사

자바의 역사는 1991년에 썬의 엔지니어들에 의해서 고안된 Oak라는 언어에서부터 시작되었다. 제임스 고슬링과 아서 밴 호프와 같은 썬의 엔지니어들의 원래 목표는 가전제품에 탑재될 소프트웨어를 만드는 것이었다. C++을 확장해서 사용하려 했지만 C++로는 그들의 목적을 이루기에 부족하다는 것을 깨달았다.

그 래서 C++의 장점을 도입하고 단점을 보완한 새로운 언어를 개발하기에 이르렀다. Oak는 처음에는 가전제품이나 PDA와 같은 소형기기에 사용될 목적이었으나 인터넷이 등장하자 운영체계에 독립적인 Oak가 이에 적합하다고 판단하여 인터넷 쪽으로 그 개발 방향을 바꾸면서 이름을 자바(Java)로 변경하였으며, 자바로 개발한 웹브라우저인 핫자바(Hot java)를 발표하기하고 그 다음에인 1996년 1월에 Java의 정식 버전을 발표했다.

그 당시만 해도 자바로 작성된 애플릿(Applet)은 정적인 웹페이지에 사운드와 애니메이션 등의 멀티미디어적인 요소들을 제공할 수 있는 유일한 방법이었기 때문에 많은 인기를 얻고 단 기간에 많은 사용자층을 확보할 수 있었다.

그 러나 속웨이브(shockwave)와 같은 뛰어난 멀티미디어 플러그인(plugin)에 밀려 애플릿은 사용범위가 많이 줄어들었고 대신 서버 쪽 프로그래밍을 위한 서블릿(Servlet)과 JSP(Java Server Pages)가 더 많이 사용되고 있다.

앞으로는 자바의 원래 목표였던 소규모 가전제품과 대규모 기업환경을 위한 소프트웨어개발 분야에 활발히 사용될 것이다.

### 1.3 자바언어의 특징

자바는 최근에 발표된 언어 답게 기존의 다른 언어에는 없는 많은 장점들을 갖고 있다. 그 중 대표적인 몇 가지에 대해서 알아보도록 하자.

#### 1. 운영체계에 독립적이다.

기 존의 언어는 한 운영체계에 맞게 개발된 프로그램을 다른 종류의 운영체계에 적용하기 위해서는 많은 노력이 필요하였지만, 자바에서는 더 이상 그런 노력을 하지 않아도 된다. 이것은 일종의 에뮬레이터인 자바가상머신(JVM)을 통해서 가능한 것인데, 자바 응용프로그램은 운영체제나 하드웨어가 아닌 JVM하고만 통신하고 JVM이 자바 응용프로그램으로부터 전달받은 명령을 해당 운영체제가 이해할 수 있도록 변환하여 전달한다. 자바로 작성된 프로그램은 운영체계에 독립적이지만 JVM은 운영체계에 종속적이어서 쉘에서는 여러 운영체제에 설치할 수 있는 서로 다른 버전의 JVM을 제공하고 있다. 그래서 자바로 작성된 프로그램은 운영체제와 하드웨어에 관계없이 실행 가능하다.(Write once, run anywhere)

#### 2. 객체지향언어이다.

자 바는 프로그래밍의 대세로 자리잡은 객체지향 프로그래밍언어(Object-oriented programming language) 중의 하나로 객체지향개념의 특징인 상속, 캡슐화, 다형성이 잘 적용

된 순수한 객체지향언어라는 평가를 받고 있다.

### 3. 배우기 쉽다.

자바의 연산자와 기본구문은 C++에서, 객체지향관련 구문은 smalltalk이라는 객체지향언어에서 가져왔다. 이 둘 언어의 장점은 취하면서 복잡하고 불필요한 부분은 과감히 제거하여 단순화함으로써 쉽게 배울 수 있으며 단순하고 이해하기 쉬운 코드를 작성할 수 있도록 하였다. 객체지향언어의 특징인 재사용성과 유지보수의 용이성 등의 많은 장점에도 불구하고 배우기가 어렵기 때문에 많은 사용자층을 확보하지 못했으나 자바의 간결하면서도 명료한 객체지향적 설계는 사용자들이 객체지향개념을 보다 쉽게 이해하고 활용할 수 있도록 하여 객체지향 프로그래밍의 저변확대에 크게 기여했다.

### 4. 자동 메모리 관리(Garbage Collection)

자바로 작성된 프로그램이 실행되면, 가비지컬렉터(Garbage Collector)가 자동적으로 메모리를 관리해주기 때문에 프로그래머는 메모리를 따로 관리 하지 않아도 된다. 가비지컬렉터가 없다면 프로그래머가 사용하지 않는 메모리를 체크하고 반환하는 일을 수동적으로 처리해야 할 것이다. 자동으로 메모리를 관리한다는 것이 다소 비효율적인 면도 있지만, 프로그래머가 보다 프로그래밍에 집중할 수 있도록 도와준다.

### 5. 네트워크와 분산처리를 지원한다.

인터넷과 대규모 분산환경을 염두에 둔 까닭인지 풍부하고 다양한 네트워크 프로그래밍 라이브러리를 통해 보다 짧은 시간에 쉽게 네트워크 관련 프로그램을 개발할 수 있도록 지원한다.

### 6. 멀티쓰레드를 지원한다.

일반적으로 멀티쓰레드(Multi-thread)의 지원은 사용되는 운영체계에 따라 구현방법도 상이하며, 처리 방식도 다르다. 그러나 자바에서 개발되는 멀티쓰레드 프로그램은 시스템과는 관계없이 구현가능하며, 관련 API가 제공되므로 구현이 쉽다. 그리고 여러 쓰레드에 대한 스케줄링을 자바 인터프리터가 담당하게 된다.

### 7. 동적 로딩(Dynamic Loading)을 지원한다.

자바는 클래스 단위의 언어로서 소스에 정의되어 있는 클래스 단위로 실행파일이 각각 생성된다. 일반적으로 각 클래스들은 다른 클래스들을 활용하여 생성되는데 소스에서 활용한 클래스들에 대한 binding은 컴파일시 처리되나 실제 메모리 할당은 프로그램 수행 중간에 필요한 시점에 인스턴스화를 통해 메모리 영역에 할당된다.

[참고] 자바의 단점으로는 속도문제가 가장 대표적인 것인데 바이트코드(bytecode)를 하드웨어

의 기계어로 바로 변환해주는 JIT컴파일러와 Hotspot과 같은 신기술의 도입으로 JVM의 기능이 향상됨으로써 속도문제가 상당히 개선되었다.

## 2. 자바개발환경 구축하기

### 2.1 자바 개발도구(J2SDK)설치하기

자바로 프로그래밍을 하기 위해서는 먼저 J2SDK를 설치해야한다. J2SDK를 설치하면, JVM과 자바클래스 라이브러리(Java API)외에 자바를 개발하는데 필요한 프로그램들이 설치된다.

[참고] 1996 년 1월에 처음으로 JDK1.0을 발표한 이후로 꾸준히 기능을 추가 개선해가고 있으며, 2003년 3월 현재 J2SDK1.4.1이 최신 버전이다. 자바 1.2이전의 버전을 Java, 자바 1.2이후의 버전을 Java2라고 부른다.

J2SDK는 부록CD에 첨부되어 있으므로 바로 실행시켜서 설치할 수 있다.

j2sdk- 1\_4\_1\_02-windows-i586.exe를 실행시키고 기본설정으로만 선택해서 설치하면 된다. 다만 가능하면 설치디렉토리를 c:\wj2sdk1.4.1로 하자. 그리고 서브디렉토리 work를 생성해서 앞으로 실습할 예제파일들을 저장하는데 사용하자.

[참고] J2SDK - CD\wj2sdk-1\_4\_1\_02-windows-i586.exe <http://java.sun.com/j2se/1.4.1/download.html>에서 다운 받을 수 있다.

설치가 끝났으면 설치된 디렉토리의 bin디렉토리(예:c:\wj2sdk1.4.1\bin)를 path에 추가해주어야 한다. 이 디렉토리에는 자바로 프로그램을 작성하는데 필요한 파일들이 들어있다.

[참고] 클래스패스는 따로 설정해 주지 않아도 된다. 기본적으로 현재디렉토리(.)가 클래스패스로 설정되어 있다. 클래스패스 설정방법은 후에 자세히 다룰 것이다.

[참고] 각 명령어에 대한 옵션을 보려면 명령어만을 실행시키면 된다.

J2SDK의 bin디렉토리에 있는 주요 실행파일들은 다음과 같다.

- ▶ javac.exe - 자바컴파일러, 자바소스코드를 바이트코드로 컴파일한다.

```
c:\wj2sdk1.4.1\work>javac Hello.java
```

- ▶ java.exe - 자바인터프리터, 컴파일러가 생성한 바이트코드를 해석하고 실행한다.

```
c:\wj2sdk1.4.1\work>java Hello
```

- ▶ **javap.exe** - 역어셈블러, 컴파일된 클래스파일(.class)를 원래의 소스(.java)로 변환한다.

```
c:\wj2sdk1.4.1\work>javap Hello > Hello.java
```

위와 같이 하면 Hello.class파일이 변환되어 Hello.java에 저장된다. '-p'옵션을 이용하면, 바이트코드로 컴파일된 내용도 볼 수 있다.

[참고]바이트코드 - JVM이 이해할 수 있는 기계어, JVM은 바이트코드를 다시 해당 OS의 기계어로 변환하여 전달한다.

- ▶ **appletviewer.exe** - 애플릿 뷰어, HTML문서에 삽입되어 있는 애플릿을 실행시킨다.

```
c:\w>appletviewer Hello.html
```

- ▶ **javadoc.exe** - 자동문서생성기, 소스파일에 있는 주석(/\*\* \*/)을 이용하여 Java API문서와 같은 형식의 문서를 자동으로 생성한다.

```
c:\wj2sdk1.4.1\work>javadoc Hello.java
```

- ▶ **jar.exe** - 압축프로그램, 클래스파일과 프로그램의 실행에 관련된 파일을 하나의 jar파일(.jar)로 압축하거나 압축해제한다.

\* 압축할 때 : c:\wj2sdk1.4.1\work>jar cvf Hello.jar Hello1.class Hello2.class

\* 압축풀 때 : c:\wj2sdk1.4.1\work>jar xvf Hello.jar

[참고] J2SDK와 JRE

J2SDK - 자바개발도구(Java2 Standard Development kit)

JRE - 자바실행환경(Java Runtime Environment), 자바로 작성된 응용프로그램이 실행되기 위한 최소환경.

J2SDK = JRE + 개발에 필요한 실행파일(javac.exe 등)

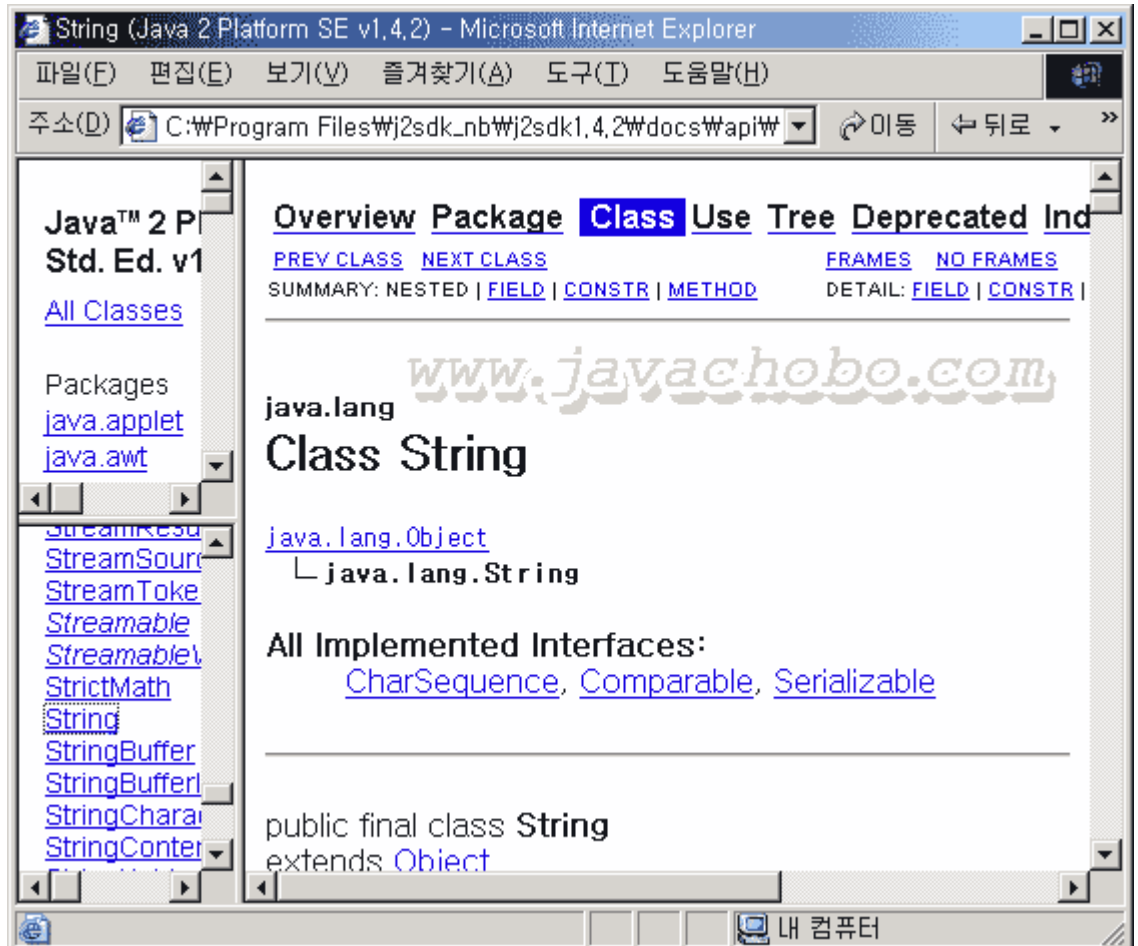
JRE = JVM + 클래스라이브러리(Java API)

## 2.2 Java API문서 설치하기

자바에서 제공하는 클래스 라이브러리(Java API)를 잘 사용하기 위해서는 Java API문서가 필수적이다. 이 문서에는 클래스라이브러리의 모든 클래스에 대한 설명이 자세하게 나와있다. 아마도 자바에서 제공하는 다양하고 방대한 양의 클래스라이브러리에 감탄하게 될 것이다.

Java API문서의 설치를 위해서는 부록CD(j2sdk-1\_4\_1-doc.zip)나 <http://java.sun.com/>에서 다운받을 수 있다. 이 문서는 ZIP방식으로 압축되어 있으므로 알집이나 winzip으로 압축을 풀어야 된다. 모두 html문서로 되어 있으며 자바와 관련된 여러가지 유용한 내용이 수록되어 있으므로 참고하도록 하자.

Java API문서가 C:\wj2sdk1.4.1에 설치되었다고 할 때,  
C:\wj2sdk1.4.1\docs\api\index.html를 웹브라우저로 열면 자바의 클래스라이브러리를 클래스의 자세한 설명을 볼 수 있을 것이다. 앞으로 자주 참고하게 될 문서이므로 바로가기를 만들어서 쉽게 열어 볼 수 있도록 하자.



### 3. 자바로 프로그램작성하기

#### 3.1 Hello.java

자 바로 프로그램을 개발하려면 J2SDK이외에 메모장(notepad.exe)나 editplus와 같은 편집기  
가 필요하다. 그 외에도 좋은 개발도구들이 많지만, 일단 처음 자바를 배우는 사람들에게겐  
editplus와 같이 가벼우면서도 성능이 뛰어난 편집기가 좋다.

[참고]editplus는 [www.editplus.com](http://www.editplus.com)에 가면 평가판을 무료로 제공한다.

##### [예제1-1] Hello.java

```
class Hello
{
    public static void main(String[] args)
    {
        System.out.println("Hello World!");
    }
}
```

이 예제는 화면에 'Hello, World!'를 출력하는 아주 간단한 프로그램이다. 이 예제를 통해서 화  
면에 글자를 출력하는 방법을 알 수 있다. 예제1-1을 작성한 다음 Hello.java로 저장하자. 이 프  
로그램을 실행시키기 위해서는 먼저 컴파일러(javac.exe)를 통해 클래스파일(Hello.class)을 생  
성해야한다. 그 다음에 자바인터프리터(java.exe)로 실행한다.





[그림1-2] Hello.java를 컴파일하고 실행한 화면

이처럼 매번 콘솔에서 컴파일하고 실행하는 것은 좀 불편하다. editplus에 javac.exe와 java.exe를 사용자도구로 등록해 놓으면 editplus내에서 컴파일과 실행을 손쉽게 할 수 있다. 설정방법은 부록CD에 있다.

### 3.2 자바프로그램의 실행과정

아래 같이 자바프로그램을 실행시켰을 때



내부적인 진행순서는 다음과 같다.

1. 프로그램의 실행에 필요한 클래스를 로드한다.
2. 클래스파일을 검사한다.
3. 지정된 클래스(Hello)에서 main(String args[])를 찾아서 호출한다.

만일 지정된 클래스에 main(String args[])가 없다면 다음과 같은 에러메시지가 나타날 것이다.



```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

자바로 작성된 프로그램은 클래스 단위로 이루어져 있으며 보통 하나 이상의 클래스 파일로 이루어져 있다. 애플릿이 아닌 응용프로그램이라면 `main(String args[])`를 가지고 있는 클래스가 반드시 있어야 한다.

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

자바 프로그램은 `main`메서드의 호출로 시작해서 `main`메서드의 마지막문장이 수행을 마치면 종료된다.

[참고]하나의 응용프로그램에서 `main(args[])`를 가진 클래스가 여럿 있을 수도 있지만, 보통 하나이다.

# [Java의 정석]제2장 변수 - 1.변수

자바의정석

2012/12/22 17:02

<http://blog.naver.com/gphic/50157739509>

## 1.변수(Variable)

### 1.1 변수란?

컴퓨터 언어에서 변수(variable)란, 값을 저장할 수 있는 메모리상의 공간을 의미한다. 계산을 하기 위해서 변수를 사용하지 않고 값을 직접 사용할 수도 있지만, 의미있는 이름의 변수에 저장하여 사용하는 것이 더 바람직하다.

변수의 값은 바뀔 수 있으며, 하나의 변수에는 단 하나의 값만을 저장할 수 있다. 그래서 값을 여러 번 저장하면 마지막에 저장한 값을 갖게 된다.

### 1.2 변수의 선언

변수를 사용하기 위해서는 먼저 변수를 선언해야한다. 변수가 선언되면 메모리 공간에 변수의 타입에 알맞은 크기의 메모리공간이 확보되어, 값을 저장할 준비가 되는 것이다.

변수를 선언하는 방법은 다음과 같다.

변수타입 변수이름;

```
int number;           // 정수형 변수 number를 선언한다.
```

변수를 선언할 때는 변수의 타입과 이름을 함께 써주어야 한다. 위의 예는 number라는 이름의 정수형 변수를 선언한 것이다. 변수타입은 변수에 담을 값의 종류와 범위를 충분히 고려하여 결정해야한다.

변수를 선언한 후부터는 변수를 사용할 수 있으며, 변수를 사용하기에 앞서 적절한 값을 저장해주는 것이 필요하다. 이것을 변수의 초기화라고 하는데, 보통 아래와 같이 변수의 선언과 함

께 한다.

```
// 정수형 변수 number를 선언하고 변수의 값을 10으로 초기화 했다.  
int number = 10;  
  
// 위 문장은 아래의 두 문장과 동일하다.  
int number;  
number=10;
```

변수의 종류에 따라 변수의 초기화를 생략할 수 있는 경우도 있지만, 변수는 사용되기 전에 적절한 값으로 초기화 하는 것이 좋다.

[참고] 지역변수는 사용되기 전에 초기화를 반드시 해야 하지만 클래스변수와 인스턴스변수는 초기화를 생략할 수 있다. 변수의 초기화에 대해서는 후에 자세히 학습하게 될 것이다.

### 1.3 명명규칙(Naming Convention)

변수의 이름, 메서드의 이름, 클래스의 이름 등 모든 이름을 짓는 데는 반드시 지켜야 할 공통적인 규칙이 있으며 다음과 같다.

1. 대소문자가 구분되며 길이에 제한이 없다.
  - True와 true는 서로 다른 것으로 간주된다.
2. 예약어를 사용해서는 안 된다.
  - true는 예약어라서 사용할 수 없지만, True는 가능하다.
3. 숫자로 시작해서는 안 된다.
  - top10은 허용하지만, 7up는 허용되지 않는다.
4. 특수문자는 '\_'와 '\$'만을 허용한다.
  - \$harp은 허용되지만, S#arp은 허용되지 않는다.

[참고]예약어는 keyword 또는 reserved word라고 하는데, 프로그래밍언어에서 구문에 사용되는 단어를 뜻한다. 그래서, 예약어는 이름으로 사용될 수 없다.

예약어는 앞으로 차차 배워 나가게 될 것이므로 지금은 간단히 훑어보는 보는 정도면 충분하다.

abstract	do	implements	private	this
boolean	double	import	protected	throw
break	else	instanceof	public	throws
byte	extends	int	return	transient
case	false	interface	short	true
catch	final	long	static	try
char	finally	native	strictfp	void
class	float	new	super	volatile
continue	for	null	switch	while
default	if	package	synchronized	

[표2-1]자바에서 사용되는 예약어

[참고] 이 밖에도 goto와 const가 더 있지만 사용되지 않는다.

그 외에 필수적이지는 않지만, 자바프로그래머들에게 권장하는 규칙들은 다음과 같다.

1. 클래스 이름의 첫 글자는 항상 대문자로 한다.
  - 변수나 메서드의 이름의 첫 글자는 항상 소문자로 한다.
2. 여러 단어로 이루어진 이름은 단어의 첫 글자를 대문자로 한다.
  - lastIndexOf, StringBuffer
3. 상수의 이름은 모두 대문자로 한다. 여러 단어로 이루어진 경우 '\_'를 사용하여 구분한다.
  - PI, MAX\_NUMBER

위의 규칙들은 반드시 지켜야 하는 것은 아니지만, 코드를 보다 이해하기 쉽게 하기 위한 자바 프로그램들간의 암묵적인 약속이다. 가능하면 지키도록 노력하자.

[참고] 자바에서는 모든 이름에 유니코드에 포함된 문자들을 사용할 수 있지만, 클래스이름은 ASCII코드(영문자)로 하는 것이 좋다. 유니코드를 인식하지 못하는 운영체제(OS)도 있기 때문이다.

# [Java의 정석]제2장 변수 - 2.변수의 타입

자바의정석

2012/12/22 17:03

<http://blog.naver.com/gphic/50157739548>

## 2. 변수의 타입(Type)

모든 변수에는 타입(Type 또는 형形)이 있으며, 변수의 타입 따라 변수에 저장할 수 있는 값의 종류와 범위가 달라진다. 변수를 선언할 때 저장하고자 하는 값을 고려하여 가장 알맞은 타입을 선택하면 된다.

변수의 타입은 크게 기본형과 참조형, 2가지로 나눌 수 있는데, 기본형 변수는 실제 값(Data)을 저장하는 반면에, 참조형 변수는 어떤 값이 저장되어 있는 주소를 값으로 갖는다.

자바는 C언어와는 달리 String을 제외한 참조형 변수간의 연산을 할 수 없으므로 실제 연산에 사용되는 것은 모두 기본형 변수이다.

### 기본형(Primitive Type)

- boolean, char, byte, short, int, long, float, double 계산을 위한 실제 값을 저장한다.

### 참조형(Reference Type)

- 8개의 기본형을 제외한 나머지 타입, 객체의 주소를 저장한다.

[참고]참조형 변수는 null 또는 객체의 주소(4 byte, 0x0~0xffffffff)를 값으로 갖는다. null은 어떤 값도 갖고 있지 않음, 즉 어떠한 객체도 참조하고 있지 않다는 것을 뜻한다.

기본형의 개수는 모두 8개이고, 참조형은 프로그래머가 직접 만들어 추가할 수 있으므로 그 수가 정해져 있지 않다.

참조형 변수를 선언할 때는 변수의 타입으로 클래스의 이름을 사용하므로 클래스의 이름이 변수의 타입이 된다. 그러므로 새로운 클래스를 작성한다는 것은 새로운 참조형을 추가하는 셈이다.

다음은 참조변수를 선언하는 방법이다.

```
클래스이름 변수명;
```

```
Date today;
```

Date클래스 타입의 참조변수 today를 선언한 것이다. 참조형 변수는 null 또는 객체의 주소를 값으로 갖으며 참조변수의 초기화는 다음과 같이 한다.

```
Date today = null;
```

또는

```
Date today = new Date();
```

객체를 생성하는 연산자 new의 연산결과는 생성된 객체의 주소이다. 이 주소가 대입연산자(=)에 의해서 참조변수 today에 저장되는 것이다.

이제 참조변수 today를 이용해서 생성된 객체를 사용할 수 있게 된다.

[참고]기본형은 저장할 값(Data)의 종류에 따라 구분되므로 기본형 변수의 종류를 얘기할 때는 자료형(Data Type)이라는 용어를 쓰고, 모든 참조형은 종류에 관계없이 4 byte의 주소(0x0 ~ 0xffffffff 또는 null)을 저장하기 때문에, 참조형 변수들은 값(Data)이 아닌, 어떤 객체의 주소를 담을 것인가에 따른 객체의 종류에 의해서 구분되므로, 참조형 변수의 종류를 구분할 때는 자료형(Data Type)대신 타입(Type)이라는 용어를 사용한다. 타입이 자료형을 포함하는 보다 넓은 의미의 용어이므로 반드시 구분해서 사용할 필요는 없다.

## 2.1 기본형(Primitive Types)

기본형에는 모두 8개의 타입이 있으며, 크게 논리형, 문자형, 정수형, 실수형, 4가지로 구분된다.

논리형 - true와 false 중 하나를 값으로 갖으며, 조건식과 논리적 계산에 사용된다.

문자형 - 문자를 저장하는데 사용되며, 변수 당 하나의 문자만을 저장할 수 있다.

정수형 - 정수 값을 저장하는 데 사용된다. 주로 사용되는 것은 int와 long이며,  
     byte는 이진데이터를 다루는데 주로 사용되며,  
     short은 C언어와의 호환을 위해서 추가하였다.

실수형 - 실수 값을 저장하는데 사용된다. float와 double밖에 없다.

종류 \ 크기	1 byte	2 byte	4 byte	8 byte
논리형	boolean			
문자형		char		
정수형	byte	short	int	long
실수형			float	double

[표2-2]기본형의 종류와 크기

[참고] 4개의 정수형(byte, short, int, long)중에서 int형이 기본(default) 자료형이며, 실수형(float, double)중에서는 double형이 기본 자료형이다.

논리형인 boolean은 나머지 7개의 자료형과 연산이 가능하지 않지만, char는 문자를 내부적으로 정수값 코드로 저장하고 있기 때문에 정수형과 밀접한 관계가 있다.

정수형과 실수형의 경우에는 여러 개의 변수형이 있지만, 특별히 큰 값을 다루어야 하지 않는 한 int 와 float를 주로 사용한다.

자료형	저장 가능한 값의 범위	크 기	
		bit	byte
boolean	false, true	8	1
char	\u0000 ~ \uffff(0~2 <sup>16</sup> -1, 0~65535)	16	2
byte	-128 ~ 127(-2 <sup>7</sup> ~2 <sup>7</sup> -1)	8	1
short	-32768 ~ 32767(-2 <sup>15</sup> ~2 <sup>15</sup> -1)	16	2
int	-2147483648 ~ 2147483647(-2 <sup>31</sup> ~2 <sup>31</sup> -1)	32	4
long	-9223372036854775808~9223372036854775807(-2 <sup>63</sup> ~2 <sup>63</sup> -1)	64	8
float	1.4E-45 ~ 3.4028235E38	32	4
double	4.9E-324 ~ 1.7976931348623157E308	64	8

[표2-3] 8가지 기본형(Primitive Type)과 저장 가능한 값의 범위

[참고] float와 double의 음의 최대값과 최소값은 양의 최대값과 최소값에 각각 음수 부호를 붙이면 된다. 실수형은 소수점이하의 자리수, 즉 정밀도가 중요하기 때문에, 얼마나 0에 가까운 값을 표현할 수 있는지도 큰 의미를 갖는다.

각 자료형이 가질 수 있는 값의 범위를 정확히 외울 필요는 없고, 정수형(byte, short, int, long)의 경우  $-2^{n-1} \sim 2^{n-1}-1$  (n: bit수)이라는 정도만 기억하고 있으면 된다.

예를 들어 int형의 경우 32bit(4byte)이므로  $-2^{31} \sim 2^{31}-1$ 의 범위를 갖는다.

$$2^{10}=1024 \div 10^3 \text{이므로 } 2^{31}=2^{10} \times 2^{10} \times 2^{10} \times 2 = 1024 * 1024 * 1024 * 2 \div 2 * 10^9$$

따라서, int형은 대략 9자리수(약 2,000,000,000)의 값을 저장할 수 있다는 것을 알 수 있다. 9자리수에 가까운 자리수(7자리나 8자리)의 수를 계산할 때는 넉넉하게 long형(약 19자리)을 사용하는 것이 좋다. 연산 중에 저장범위를 넘어서게 되면 원하지 않는 값을 결과로 얻게 될 것이기 때문이다.

정수형에서 int를 실수형에서는 float를 주로 사용하므로, int와 float의 범위를 기억해서, int와 float의 범위를 넘는 값을 다뤄야 할 때 long과 double을 사용하면 된다.

기본 자료형의 크기를 쉽게 외우는 방법은 다음과 같다.



- boolean은 true와 false 두 가지 값만 표현할 수 있으면 되므로 가장 작은 크기인 1 byte.
- char은 자바에서 유니코드(2 byte 문자체계)를 사용하므로 2 byte.
- byte는 크기가 1 byte라서 byte.
- int(4 byte)를 기준으로 짧아서 short(2 byte), 길어서 long( 8byte). (short <-> long)
- float는 실수값을 부동소수점(floating-point)방식으로 저장하기 때문에 float.
- double은 float보다 두 배의 크기(8 byte)와 두 배의 정밀도(double-precision)를 갖기 때문에 double.

## 2.2 논리형 - boolean

논리형에는 boolean, 한가지 밖에 없다. boolean형 변수에는 true와 false 중 하나를 저장할 수 있으며 기본값(default)은 false이다.

boolean 형 변수는, 대답(yes/no), 스위치(on/off) 등의 논리구현에 주로 사용된다. 그리고, boolean형은 true와 false, 두 가지의 값만을 표현하면 되므로 기본형 중에서 가장 크기가 작은 1 byte이다.

[참고] 1 byte는 8 bit이므로 2의 8제곱, 256가지의 값을 표현할 수 있다. 따라서 boolean형은 크기가 1byte이므로 256가지의 값을 표현할 수 있으나, true와 false, 2가지의 값만을 표현하는데 사용되고 있다.

아래 문장은 power라는 boolean형 변수를 선언하고 true로 변수를 초기화 했다.

```
boolean power = true;
```

[주의] Java에서는 대소문자를 구별하기 때문에 TRUE와 true는 다른 것으로 간주하므로 주의하도록 한다.

## 2.3 문자형 - char

문자형 역시 char 한가지 밖에 없다. 기존의 많은 프로그래밍의 언어에서 문자형의 경우 1 byte(ASCII코드)의 크기를 갖지만, Java에서는 유니코드(Unicode)문자 체계를 사용하기 때문에 크기가 2byte이다.

[참고] Unicode는 세계 각 국의 언어를 통일된 방법으로 표현할 수 있게 제안된 국제적인 코드 규약이다.

미국에서 개발되어진 컴퓨터는 그 구조가 영어를 바탕으로 정의되어 있기에 26자의 영문 알파벳과 몇 가지 특수 문자를 표현하기에는 1바이트로 충분하였기 때문에 모든 정보(문자)가 1바이트를 단위로 표현되고 있었으나 동양3국의 언어 표현인 한글, 한자 또는 일어 등과 같은 문자는 그 구조가 영어와 달라서 1 바이트로는 표현이 불가능하기에 2바이트로 조합하여 하나의 문자를 표현하는 컴퓨터의 구조적 문제점을 바탕으로 유니코드가 만들어 졌다. 유니코드에 대한 보다 자세한 내용은 원한다면, <http://www.unicode.org/standard/translations/korean.html>을 방문해보도록 하자.

[참고]아스키는 128개의 가능한 문자조합을 제공하는 7비트(bit) 부호로, 처음 32개의 부호는 인쇄와 전송 제어용으로 사용된다. 보통 기억장치는 8비트(1바이트, 256조합)이고, 아스키는 단지 128개의 문자만 사용하기 때문에 나머지 비트는 특수문자에 사용된다.

char형의 크기는 2 byte이므로 16진수로 0000부터 ffff까지, 문자를 표현하는데 65536개(2의 16제곱)의 코드값을 사용할 수 있으며, char형 변수는 이 범위 내의 코드값 하나를 저장할 수 있다.

예를 들어 알파벳 A의 유니코드값은 0041이다. char형 변수에 문자 A를 저장하려면 아래와 같이 한다.

```
char firstLetter = 'A' ;  
또는  
char firstLetter = '\u0041' ;    // 16진수 41은 10진수로 65
```

char형 변수 firstLetter를 선언하고, 문자 A를 저장했다. char형 변수에 문자를 저장할 때는 "(홀따옴표)로 문자를 둘러싼다. 두 번째는 문자의 코드를 이용해서 문자형 변수 firstLetter에 값을 저장했다.

문자형 변수에 값을 저장하는 데는 위의 두 가지 모두 가능하지만, 주로 첫 번째 방식으로 문자

를 저장한다.

다음은 char형이 저장되는 방식을 short형과 함께 비교해 보았다.

자료형	2 진수	10 진수
char	0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1	65
short	0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1	65
char	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	65535
short	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	-1

[표2-4]char형과 short형의 값 비교

char형이나 short형은 크기가 모두 2 byte(16 bit)지만, 범위가 다르다. char형은 0~65535이고 short형은 -32768~32767이다.

하지만, 둘 다 2 byte이기 때문에 표현할 수 있는 수는 65536개로 같다. char 형은 문자의 코드값을 저장하므로 음수를 필요로 하지 않기 때문에 2진수로 표현했을 때의 첫 번째 자리를 부호에 사용하지 않는다. 반면에 short형은 첫 번째 자리를 부호를 표현하는데 사용하기 때문에 서로 다른 범위를 갖게 되는 것이다. 이처럼 char형은 정수형과 표현방식이 같기 때문에 정수형과 깊은 관계가 있다.

만 일 어떤 문자의 코드값을 알고 싶으면, char형 변수를 정수형(int)으로 변환하면 된다. 어떤 타입(Type, 형)을 다른 타입으로 변환하는 것을 형변환(캐스팅, casting)이라고 하는데, 형변환에 대해서는 후에 자세히 설명하도록 하겠다. 지금은 문자의 코드값을 알아내는 방법과, 어떤 코드가 어떤 문자를 나타내는가를 알아내는 방법이 있다는 정도만 알면 된다.

#### [예제2-1] CharToCode.java

```
class CharToCode {  
    public static void main(String[] args) {  
        char ch = 'A';          // char ch = '\u0041';로 바꿔 써도 같다.  
        int code = (int)ch;  
        System.out.println(ch);  
        System.out.println(code);  
    }  
}
```

<pre>         }     } </pre>
<b>[실행결과]</b>
<pre> A 65 </pre>

위의 예제를 실행하면 65가 화면에 출력되는데, 문자 A의 코드가 10진수로 65임을 뜻한다.(16진수로는 41)

<b>[예제2-2] CodeToChar.java</b>
<pre> class CodeToChar {     public static void main(String[] args) {         int code = 65; // 또는 int code = 0x0041;         char ch = (char)code;          System.out.println(code);         System.out.println(ch);     } } </pre>
<b>[실행결과]</b>
<pre> 65 A </pre>

이 예제는 코드값 65(16진수로 41)가 어떤 문자를 뜻하는지를 알아낼 수 있는 방법을 알려 준다.

**[참고]** char형은 크기가 2byte이고, 2byte는 16bit이므로 2의 16제곱(65536)개의 값을 저장할 수 있다. 0을 포함하므로 범위는 0~65535, 16진수로는 0x0000~0xffff가 된다.

두 예제에서 볼 수 있듯이 char형의 변수를 정수형(int)으로 변환(casting)하면, 변수에 저장되어 있는 문자의 코드값을 10진수로 얻을 수 있다(CharToCode.java). 반대로 한 코드가 어떤 문자를 나타내는지 알고 싶으면, 코드를 정수형 변수에 저장한 다음, char형으로 변환하여 출력하면 된다(CodeToChar.java).

임의로 0~65535사이의 값을 하나 선택한 다음, CodeToChar.java를 이용해서 선택한 코드가 어떤 문자를 뜻하는지를 알아보거나 반복문을 이용해서 유니코드 전체를 출력해 보는 것도 흥미로울 것이다.

10진수로 66(16진수로 42)을 문자형으로 변환하여 출력하면 어떤 문자가 출력될까?

[참고]유니코드(Unicode)는 ASCII코드와의 호환을 위해서, 유니코드 코드번호 1부터 128까지의 문자를 ASCII코드의 1부터 128까지의 문자와 동일하게 지정하였다.

영문자 이외에 Tab이나 space 등의 특수문자를 저장하려면 아래와 같이 하도록 한다.

```
char tab = '\t';
```

\t는 실제로는 두 문자로 이루어져 있지만, 단 한 문자 Tab을 의미한다.

아래의 표는 Tab과 같이 특수한 문자를 어떻게 표현할 수 있는지 알게 해준다.

특수문자	리터럴
Tab	\t
Backspace	\b
Form Feed	\f
new line	\n
carriage return	\r
역슬래쉬(\)	\\
홀따옴표	\'
겹따옴표	\"
유니코드 문자(16진수)	\u16진수 (예: char a='\u0041')

# [Java의 정석]제2장 변수 - 3.형변환

자바의정석

2012/12/22 17:03

<http://blog.naver.com/gphic/50157739587>

## 3. 형변환

### 3.1. 형변환(Casting)이란?

모든 리터럴과 변수에는 타입이 있다는 것을 배웠다. 프로그램을 작성하다 보면, 서로 다른 타입의 값으로 연산을 수행해야하는 경우가 자주 발생한다.

모든 연산은 기본적으로 같은 타입의 피연산자(Operand)간에만 수행될 수 있으므로, 서로 다른 타입의 피연산자간의 연산을 수행해야하는 경우, 연산을 수행하기 전에 형변환을 통해 같은 타입으로 변환해주어야 한다.

형변환이란, 변수 또는 리터럴의 타입을 다른 타입으로 변환하는 것이다.

예를 들어 int형 값과 float형 값의 덧셈연산을 수행하려면, 먼저 두 값을 같은 타입으로 변환해야하므로, 둘 다 int형으로 변환하던가 또는 둘 다 float형으로 변환해야한다.

### 3.2 형변환 방법

기본형과 참조형 모두 형변환이 가능하지만, 기본형과 참조형 사이에는 형변환이 성립되지 않는다. 기본형은 기본형으로만 참조형은 참조형으로만 형변환이 가능하다.

형변환 방법은 매우 간단하며 다음과 같다.

(타입이름)피연산자

피연산자 앞에 변환하고자 하는 타입의 이름을 괄호에 넣어서 붙여 주기만 하면 된다. 여기에 사용되는 괄호는 특별히 캐스트연산자(형변환 연산자)라고 하며, 형변환을 캐스팅(Casting)이라고도 한다.

캐스트연산자는 수행결과로 피연산자의 값을 지정한 타입으로 변환하여 반환한다. 이 때, 형변환은 피연산자의 원래 값에는 아무런 영향도 미치지 않는다.

#### [예제2-7] CastingEx1.java

```
class CastingEx1
{
    public static void main(String[] args)
    {
        double d = 100.0;
        int i = 100;
        int result = i + (int)d;

        System.out.println("d=" + d);
        System.out.println("i=" + i);
        System.out.println("result=" + result);
    }
}
```

#### [실행결과]

```
d=100.0
i=100
result=200
```

double변수 d와 int변수 i의 덧셈 연산을 하기 위해 캐스트연산자를 이용해서 d를 int형으로 변환하여 덧셈연산을 수행하였다. 그리고 그 결과를 int변수 result에 저장하였다.

int형과 int형의 연산결과는 항상 int형 값을 결과로 얻으므로 result의 타입을 int형으로 하였다.

결 과의 첫째 줄을 보면 d=100.0으로 형변환 후에도 d에 저장된 값에는 변함이 없다. 캐스트 연산자를 이용해서 d를 형변환 하였어도 d에 저장된 값에는 변함이 없음을 알 수 있다. 단지, 형변환 한 결과를 덧셈에 사용했을 뿐이다.

[참고]캐스트 연산자는 우선순위가 매우 높기 때문에 덧셈연산 전에 형변환이 수행되었다.

### 3.3 기본형의 형변환

8개의 기본형 중에서 boolean을 제외한 나머지 7개의 기본형 간에는 서로 형변환이 가능하다.

변 환	수 식	결 과
int → char	(char) 65	'A'
char → int	(int) 'A'	65
float → int	(int) 1.6f	1
int → float	(float) 10	10.0f

[표2-10]기본형간의 형변환

[참고]char형은 십진수로 0~65535의 코드값을 갖는다. 문자 'A'의 코드는 십진수로 65이다.

각 자료형마다 표현할 수 있는 값의 범위가 다르기 때문에 범위가 큰 자료형에서 범위가 작은 자료형으로의 변환은 값 손실이 발생할 수 있다.

예를 들어 실수형을 정수형으로 변환하는 경우 소수점 이하의 값은 버려지게 된다. 위의 표에서도 알 수 있듯이 float리터럴인 1.6f를 int로 변환하면 1이 된다.

반대로 범위가 작은 자료형에서 큰 자료형으로 변환하는 경우에는 절대로 값 손실이 발생하지 않으므로 변환에 아무런 문제가 없다.

[참고]실수형을 정수형으로 변환할 때 소수점 이하의 값은 반올림이 아닌 버림으로 처리된다는 점에 유의하도록 하자.

#### [예제2-8] CastingEx2.java

```
class CastingEx2
{
```



```

public static void main(String[] args)
{
    byte b = 10;
    int i = (int)b;
    System.out.println("i=" + i);
    System.out.println("b=" + b);

    int i2 = 300;
    byte b2 = (byte)i2;
    System.out.println("i2=" + i2);
    System.out.println("b2=" + b2);
}
}

```

#### [실행결과]

```

i=10
b=10
i2=300
b2=44

```

byte 형 값을 int형으로, int형 값을 byte형으로 변환하고 그 결과를 출력하는 예제이다. byte와 int는 모두 정수형으로 각각 1 byte(8 bit)와 4 byte(32 bit)의 크기를 갖으며, 표현할 수 있는 값의 범위는 byte가  $-2^7 \sim 2^7-1$  (-128~127), int가  $-2^{32} \sim 2^{32}-1$ 이다.

byte의 범위를 int가 포함하고 있으며, int가 byte보다 훨씬 큰 표현 범위를 갖고 있다. 아래 그림에서 볼 수 있는 것과 같이 byte값을 int값으로 변환하는 것은 1 byte에서 4 byte로 나머지 3 byte(24자리)를 단순히 0으로 채워 주면 되므로 기존의 값이 그대로 보존된다.

하지만, 반대로 int값을 byte값으로 변환하는 int값의 상위 3 byte(24자리)를 잘라내서 1 byte로 만드는 것이므로 기존의 int값이 보존될 수도 있고 그렇지 않을 수도 있다.



자동형변환이 되는 변환이며, 그 반대 방향으로의 변환은 반드시 캐스트 연산자를 이용한 형변환을 해야 한다.

보통 자료형의 크기가 큰 것일 수록 값의 표현범위가 크기 마련이지만, 실수형은 정수형과는 값을 표현하는 방식이 다르기 때문에 같은 크기일지라도 실수형이 정수형보다 훨씬 더 큰 표현범위를 갖기 때문에 float와 double이 같은 크기인 int와 long보다 오른쪽에 위치한다.

short과 char은 모두 2 byte의 크기를 갖지만, short의 범위는  $-2^{15} \sim$

$2^{15}-1$ (-32768~32767)이고 char의 범위는  $0 \sim 2^{16}-1$ (0~65535)이므로 서로 범위가 달라서 둘 중 어느 쪽으로의 형변환도 값 손실이 발생할 수 있으므로 자동적으로 형변환이 수행될 수 없다.

위의 그림을 참고하여 형변환 시에 캐스트 연산자를 생략할 수 있는지 또는 캐스트 연산자를 사용해야하는지를 결정하도록 한다.

#### << 요약정리 >>

1. boolean을 제외한 나머지 7개의 기본형들은 서로 형변환이 가능하다.
2. 기본형과 참조형 간에는 서로 형변환이 되지 않는다.
3. 서로 다른 타입의 변수간의 연산에는 형변환이 요구되지만, 값의 범위가 작은 타입에서 큰 타입으로의 변환은 생략할 수 있다.

# [Java의 정석]제3장 연산자 - 4.비교연산자, 5.그 외의 연산자

자바의정석

2012/12/22 17:04

<http://blog.naver.com/gphic/50157739677>

## 4. 비교 연산자

두 개의 변수 또는 리터럴을 비교하는 데 사용되는 연산자로, 주로 조건문과 반복문의 조건식에 사용되며, 연산결과는 true 또는 false이다.

비교연산자 역시 이항연산자이므로 비교하는 피연산자의 자료형이 서로 다를 경우에는 자료형의 범위가 큰 쪽으로 형변환을 하여 피연산자의 타입을 일치시킨 후에 비교한다.

### 4.1 대소비교연산자 - <, >, <=, >=

두 피연산자의 크기를 비교하는 연산자이다. 기본형 중에서는 boolean형을 제외한 나머지 자료형에 다 사용할 수 있고, 참조형에는 사용할 수 없다.

### 4.2 등가비교연산자 - ==, !=

두 피연산자에 저장되어 있는 값이 같은지, 또는 다른지를 비교하는 연산자이다.

대 소비교연산자(<,>, <=, >=)와는 달리, 기본형은 물론 참조형 모든 자료형에 사용할 수 있다. 기본형의 경우 변수에 저장되어 있는 값이 같은지를 알 수 있고, 참조형의 경우 객체의 주소값을 저장하기 때문에 두 개의 피연산자(참조변수)가 같은 객체를 가리키고 있는지를 알 수 있다. 기본형과 참조형간에는 서로 형변환이 가능하지 않기 때문에 등가비교 연산자(==,!=)의 피연산자로 기본형과 참조형을 함께 사용할 수는 없다.

[참고] 참조형 변수에 사용할 수 있는 연산자는 "=="와 "!=" 그리고 캐스트 연산자 뿐이다.

String에는 예외적으로 문자열결합에 "+"을 사용하는 것을 허용한다.

www.javachoco.com	
$x > y$	x보다 y가 작을 때 true, 그 외에는 false
$x < y$	x보다 y가 클 때 true, 그 외에는 false
$x >= y$	x보다 y가 작거나 같을 때 true, 그 외에는 false
$x <= y$	x보다 y가 크거나 같을 때 true, 그 외에는 false
$x == y$	x와 y가 같을 때 true, 다를 때 false
$x != y$	x와 y가 다를 때 true, 같을 때 false

[표3-10]비교연산자

#### [예제3-21] OperatorEx21.java

```

class OperatorEx21
{
    public static void main(String[] args)
    {
        if(10 == 10.0f) {
            System.out.println("10과 10.0f은 같다.");
        }

        if('0' != 0) {
            System.out.println("'0'과 0은 같지 않다.");
        }

        if('A' == 65) {
            System.out.println("'A'는 65와 같다.");
        }

        int num = 5;

        if( num > 0 && num < 9) {
            System.out.println("5는 0보다 크고, 9보다는 작다.");
        }
    }
}

```

#### [실행결과]

비 교연산자에서도 연산을 수행하기 전에 형변환을 통해 피연산자의 타입을 맞추는 다음 피연산자를 비교한다. 10==10.0에서 10은 int이고 10.0f는 float이므로, int값인 10을 보다 큰 범위를 갖는 float로 변환한 다음 비교한다. 두 값이 10.0f로 같으므로 결과로 true를 얻게 된다. 마찬가지로 'A'==65는 'A'가 int로 변환되어 65가 된 다음 65==65를 계산하므로 역시 true를 얻는다.

[참고]문자 'A'의 문자코드는 10진수로 65이다.

[illegible]

비트 연산자는 이진 비트연산을 수행한다. 값을 이진수로 표현했을 때의 각 자리수를 아래의 표의 규칙에 따라 연산을 수행한다. 실수형인 float와 double을 제외한 모든 기본형에 사용가능 하다.

[참고]boolean형의 경우 1은 true로 0은 false로 생각하면 된다. 예를 들면 true & true의 연산 결과는 true이고 true ^false의 결과는 true이다.

[참고]연산자 "^"는 배타적 OR(Exclusive OR)라고 하며, 피연산자의 값이 서로 다른 경우, 즉 배타적인 경우에만 1 또는 true를 결과로 얻는다.

x	y	x   y	x & y	x ^ y
1	1	1	1	0
1	0	1	0	1
0	1	1	0	1
0	0	0	0	0

[표3-12]논리연산자의 연산결과

식	2진수	10진수
3   5 = 7	$\begin{array}{r} 00000011 \\ 00000101 \\ \hline 00000111 \end{array}$	3 5 7
3 & 5 = 1	$\begin{array}{r} 00000011 \\ 00000101 \\ \hline 00000001 \end{array}$	3 5 1
3 ^ 5 = 6	$\begin{array}{r} 00000011 \\ 00000101 \\ \hline 00000110 \end{array}$	3 5 6

[표3-13]비트연산자의 연산결과

#### [예제3-22] OperatorEx22.java

```
class OperatorEx22
{
    public static void main(String[] args)
    {
        int x = 3;
```

```

        int y = 5;

        System.out.println("x는 " + x + "이고, y는 " + y + "일 때, ");
        System.out.println("x | y = " + (x|y));
        System.out.println("x & y = " + (x&y));
        System.out.println("x ^ y = " + (x^y));

        System.out.println("true | false = " + (true|false));
        System.out.println("true & false = " + (true&false));
        System.out.println("true ^ false = " + (true^false));
    }
}

```

#### [실행결과]

```

x는 3이고, y는 5일 때,
x | y = 7
x & y = 1
x ^ y = 6
true | false = true
true & false = false
true ^ false = true

```

[참고]비트연산자는 덧셈연산자(+)보다 연산우선순위가 낮기 때문에 괄호를 사용해야한다.

## 5. 그 외의 연산자

### 5.1 논리연산자 - &&, ||

논리 연산자는 피연산자로 boolean형 또는 boolean형 값을 결과로 하는 조건식만을 허용한다. 조건문과 반복문에서 조건식간의 결합에 사용된다.

그리고, "&&"가 "||" 연산보다 우선순위가 높으므로 한 조건식에 "&&"와 "||" 가 함께 사용될 때는 괄호를 사용해서 우선순위를 명확히 해주는 것이 좋다.



|| (OR결합) - 피연산자 중 한 쪽만 true이면 true을 결과로 얻는다.

&&(AND결합) - 피연산자 양쪽 모두 true이어야 true을 결과로 얻는다.

x	y	x    y	x && y
true	true	true	true
true	false	true	false
false	true	true	false
false	false	false	false

[표3-14]논리연산자의 연산결과

논리연산자의 또 다른 특징은 효율적인 연산을 한다는 것이다. OR 연산(||)의 경우, 두 개의 피연산자중 어느 한 쪽만 true이어도 전체 연산결과가 true이므로 좌측의 피연산자가 true이면, 우측의 피연산자의 값은 검사하지 않는다. AND연산(&&)의 경우도 마찬가지로 어느 한쪽만 false이어도 전체 연산결과가 false이므로 좌측의 피연산자가 false이면, 우측의 피연산자의 값은 검사하지 않는다.

같은 조건식이라도 피연산자의 위치에 따라서 연산속도가 달라질 수 있는 것이다. AND연산의 경우에는 연산결과가 false일 확률이 높은 피연산자를 연산자의 좌측에 놓아야 더 빠른 연산결과를 얻을 수 있을 것이다.

[참고]비트연산자 "&"와 "&" 역시 피연산자로 boolean형을 허용하므로 조건식간의 연결에 사용할 수 있지만, "||"나 "&&"와는 달리 항상 양 쪽의 피연산자를 모두 검사한다.

#### [예제3-23] OperatorEx23.java

```
class OperatorEx23
{
    public static void main(String[] args)
    {
        char x='j';

        if((x>='a' && x <='z') || (x>='A' && x <='Z')) {
            System.out.println("유효한 문자입니다.");
        }
    }
}
```

```

    } else {
        System.out.println("유효하지 않은 문자입니다.");
    }
}
}

```

#### [실행결과]

유효한 문자입니다.

"&&" 와 "||"를 조합하여 문자 x가 영문자인지를 검사하는 조건식을 만들었다. 사용자로부터 입력받은 문자를 검사하는데 응용될 수 있을 것이다. 대문자보다 소문자를 입력할 확률이 높기 때문에 소문자인지를 검사하는 식을 좌측에 두었다.

문자형 변수 x에 저장된 문자가 'j'이므로 OR연산자(||)의 좌측 피연산자인 조건식(x>='a' && x <='z')의 결과가 true이다. 그래서 나머지 우측 피연산자는 검사하지 않고 전체 조건식을 true로 판단한다.

[참고]영 문자 또는 숫자인지를 검사하려면 조건식을 ((x>='a' && x <='z') || (x>='A' && x <='Z') || (x>='0' && x <='9'))와 같이 하면 된다.

## 5.2 삼항 연산자 - ? :

삼항 연산자는 세 개의 피연산자를 필요로 하기 때문에 삼항연산자로 이름지어졌다. 조건식과 조건식이 참(true)일 때와 거짓(false)일 때 반환되는 값 이 세가지가 삼항연산자의 피연산자이다. 삼항연산자의 조건식에는 연산결과가 true 또는 false인 식이 사용되어야 한다.

삼항연산자는 if문으로 바꿔 쓸 수 있으며, 간단한 if문 대신 삼항연산자를 사용하면 코드를 보다 간단히 할 수 있다.

조건식의 연산결과가 true이면 식1을 결과로 얻고 false이면 식2를 결과로 얻는다.

(조건식) ? 식1 : 식2

result = (x > 0) ? x : -x

삼항연산자를 if문으로 변경하면 아래와 같다.

```
if (x > 0) {  
    result = x;  
} else {  
    result = -x;  
}
```

#### [예제3-24] OperatorEx24.java

```
class OperatorEx24  
{  
    public static void main(String[] args)  
    {  
        int x = 10;  
        int y = -10;  
  
        int absX = (x >= 0) ? x : -x;  
        int absY = (y >= 0) ? y : -y;  
  
        System.out.println("x=10일 때, x의 절대값은 "+absX);  
        System.out.println("y=-10일 때, y의 절대값은 "+absY);  
    }  
}
```

#### [실행결과]

```
x=10일 때, x의 절대값은 10  
y=-10일 때, y의 절대값은 10
```

삼항연산자를 이용해서 변수의 절대값을 구하는 예제이다. 삼항연산자 대신 조건문을 사용하

면 다음과 같다.

```
int abs_x=0;

if (x > 0) {
    abs_x = x;
} else {
    abs_x = -x;
}
```

[참고] if문 안에 선언한 변수는 if문 내에서만 유효하므로 변수 abs\_x를 if문 외부에 선언하였다.

### 5.3 대입 연산자 =, op=

대입연산자는 변수에 값 또는 수식의 연산결과를 저장하는데 사용된다. 대입연산자의 왼쪽에는 반드시 변수가 위치해야하며, 오른쪽에는 리터럴이나 변수 또는 수식이 올 수 있다.

```
int i = 0;
i = 3;
i = i + 3;
3 = i + 3;           // 대입연산자의 왼쪽 피연산자가 변수가 아니다.
final MAX = 3;
MAX = 10;           // 대입연산자의 왼쪽 피연산자가 변수가 아니다.
```

대입 연산자는 모든 연산자들 중에서 가장 낮은 연산순위를 가지고 있기 때문에 제일 마지막에 수행된다. 그리고 연산진행방향이 오른쪽에서 왼쪽이기 때문에 x=y=3;에서 y=3이 먼저 수행되고 그 다음에 x=y가 수행된다.

또한 대입연산자는 다른 연산자와 결합하여 "op="와 같은 방식으로 사용될 수 있다. 예를 들

면,  $i = i + 3$ 은  $i += 3$ 과 같이 표현될 수 있다.

op=	=
$i += 3;$	$i = i + 3;$
$i -= 3;$	$i = i - 3;$
$i *= 3;$	$i = i * 3;$
$i /= 3;$	$i = i / 3;$
$i \% = 3;$	$i = i \% 3;$
$i << = 3;$	$i = i << 3;$
$i >> = 3;$	$i = i >> 3;$
$i >>> = 3;$	$i = i >>> 3;$
$i \& = 3;$	$i = i \& 3;$
$i \wedge = 3;$	$i = i \wedge 3;$
$i \mid = 3;$	$i = i \mid 3;$
$i *= 10 + j;$	$i = i * (10 + j);$

[표3-15] 대입연산자 op=

# [Java의 정석]제3장 연산자 - 1.연산자, 2.단항연산자

자바의정석

2012/12/22 17:04

<http://blog.naver.com/gphic/50157739613>

---

## 1. 연산자(Operator)

연산자는 모든 프로그래밍언어에서 가장 기본적이면서도 중요한 요소이다. 각 연산자의 특징과 수행결과, 그리고 우선순위에 대해서 아주 잘 알고 있어야 한다.

자바는 연산자의 대부분과 조건문과 반복문 등의 기본 구문을 C언어에서 가져왔다. 그 것이 C언어를 배운 사람이 자바를 쉽게 배우는 이유이기도 하다.

하지만, 프로그래밍 언어를 처음 배우는 사람이 자바를 배우기 위해 C언어를 배울 필요는 없다. 그 시간에 자바를 배우는 데 투자하는 것이 프로그래밍실력을 향상시키는데 더 도움이 되기 때문이다.

종 류	연산방향	연 산 자	우선순위
단항연산자	←	++ -- + - ~ ! (타입)	높음
산술연산자	→	* / %	
	→	+ -	
	→	<< >> >>>	
비교연산자	→	< > <= >= instanceof	
	→	== !=	
논리연산자	→	&	
	→	^	
	→		
	→	&&	
	→		
삼항연산자	→	?:	낮음
대입연산자	←	= *= /= %= += -= <<= >>= >>>= &= ^=  =	

[표3-1 연산자의 종류와 우선순위]

[참고] instanceof 연산자는 인스턴스의 타입을 알아내는데 사용되는 연산자이다. 후에 자세하게 다룰 것이므로 이에 대한 설명은 생략하겠다.

위의 표에서 같은 줄에 있는 연산자들은 우선순위가 같다. 우선순위가 같은 연산자들 간에는 연산방향에 의해서 연산순서가 정해진다.

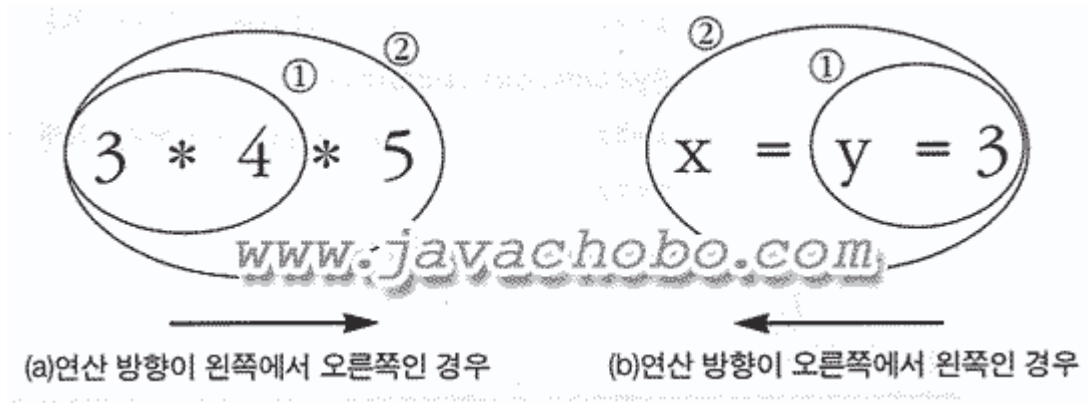
1. 산술 > 비교 > 논리 > 대입. 대입은 제일 마지막에 수행된다.
2. 단항(1) > 이항(2) > 삼항(3). 단항 연산자의 우선순위가 이항 연산자보다 높다.
3. 단항연산자와 대입연산자를 제외하고는 연산의 진행방향이 왼쪽에서 오른쪽이다.

[참고] 연산자가 연산을 하는데 필요로 하는 피연산자의 개수에 따라서 단항, 이항, 삼항 연산자라고 부른다. 덧셈 연산자(+)는 두 개의 피연산자를 필요로 하므로 이항연산자이다.

[참고] 표3-1에서 산술연산자 중 단항연산자의 '(자료형)'은 형변환에 사용되는 캐스트 연산자

이다.

연산의 진행방향을 설명하기위해 다음과 같은 두 개의 식을 예로 들어보자. 수식  $3*4*5$ 는 곱셈연산자(\*)의 연산방향이 왼쪽에서 오른쪽이므로 수식의 왼쪽에 있는  $3*4$ 가 먼저 계산되고, 그 다음  $3*4$ 의 연산결과인 12와 5의 곱셈을 수행한다.



대입연산자는 연산방향이 오른쪽에서 왼쪽으로 진행하므로, 수식  $x=y=3$ 의 경우 제일 오른쪽에서부터 계산을 시작해서 왼쪽으로 진행해 나간다.

따라서,  $y=3$ 이 가장 먼저 수행되어서  $y$ 에 3이 저장되며, 그 다음에  $x=y$ 가 수행되어  $y$ 에 저장되어 있는 값이 다시  $x$ 에 저장되어  $x$  역시  $y$ 와 같은 값을 갖게 된다.

즉,  $x=y=3$ ;은 아래 두 문장을 한 문장으로 줄여 쓴 것과 같다.

```
y=3;    // 먼저 y에 3이 저장되고
x=y;    // y에 저장되어 있는 값(3)이 x에 저장된다.
```

[Tip] 연산우선순위가 확실하지 않을 경우에는 괄호를 사용하면 된다. 괄호 안의 계산식이 먼저 계산될 것이 확실하기 때문이다.

이제 위의 표에 나와 있는 연산자들을 하나씩 자세히 살펴보도록 하자.

## 2. 단항연산자



## 2.1 증감연산자 - ++, --

일반적으로 단항연산자는 피연산자의 오른쪽에 위치하지만, ++와 --연산자는 양쪽 모두 가능하다. 연산자를 어느 위치에 놓는가에 따라서 연산결과가 달라질 수 있다.

++ : 피연산자(operand)의 값을 1 증가 시킨다.  
-- : 피연산자(operand)의 값을 1 감소 시킨다.

boolean형을 제외한 모든 기본형(Primitive Type) 변수에 사용 가능하며, 피연산자의 왼쪽에 사용하는 전위형과 오른쪽에 사용하는 후위형이 있다.

### [예제3-1] OperatorEx1.java

```
class OperatorEx1 {  
    public static void main(String args[]) {  
        int i=5;  
        i++;      // i=i+1과 같은 의미이다. ++i; 로 바꿔 써도 결과는 같다.  
        System.out.println(i);  
        i=5;      // 결과를 비교하기 위해 i값을 다시 5로 설정.  
        ++i;  
        System.out.println(i);  
    }  
}
```

### [실행결과]

6  
6

i의 값을 증가시킨 후 출력하는데, 한번은 전위형(++i)을 사용했고, 또 한번은 후위형(i++)을 사용했다. 결과를 보면 두 번 모두 i의 초기값 5에서 1이 증가된 6이 출력됨을 알 수 있다.

이 경우에는 어떤 수식에 포함된 것이 아니라 단독적으로 사용된 것이기 때문에, 증감연산자(++i)를 피연산자의 오른쪽에 사용한 경우(i++)와 왼쪽에 사용한 경우(++i)의 차이가 전혀 없

다.

[참고] 증감연산자를 피연산자의 앞에 사용하는 것을 전위형(prefix)이라 하고, 피연산자 다음에 사용하는 것을 후위형(postfix)이라고 한다.

그러나, 다른 수식에 포함되거나 함수의 매개변수로 쓰여진 경우, 즉 단독으로 사용되지 않은 경우 전위형과 후위형의 결과는 다르다.

#### [예제3-2] OperatorEx2.java

```
class OperatorEx2 {  
    public static void main(String args[]) {  
        int i=5;  
        int j=0;  
        j = i++;  
        System.out.println("j=i++; 실행 후, i=" + i + ", j="+ j);  
  
        i=5;    // 결과를 비교하기 위해, i와 j의 값을 다시 5와 0으로 변경  
        j=0;  
        j = ++i;  
        System.out.println("j=++i; 실행 후, i=" + i + ", j="+ j);  
    }  
}
```

#### [실행결과]

j=i++; 실행 후, i=6, j=5

j=++i; 실행 후, i=6, j=6

i의 값은 어느 경우에서나 1이 증가되어 6이 되지만 j의 값은 전위형과 후위형의 결과가 다르다는 것을 알 수 있다.

전위형은 변수(피연산자)의 값을 먼저 증가시킨 후에 변수가 참조되는데 반해, 후위형은 변수의 값이 먼저 참조된 후에 값이 증가된다.

따라서, j=i++;(후위형)에서는 i값인 5가 참조되어 j에 5가 저장된 후에 i가 증가한다.

j=++i;(전위형)에서는 i가 5에서 6으로 먼저 증가한 다음에 참조되어 6이 j에 저장된다.

다음은 함수의 매개변수에 증감연산자가 사용된 예이다.

#### [예제3-3] OperatorEx3.java

```
class OperatorEx3 {  
    public static void main(String args[]) {  
        int i=5, j=5;  
        System.out.println(i++);  
        System.out.println(++j);  
        System.out.println("i = " + i + ", j = " + j);  
    }  
}
```

#### [실행결과]

5

6

i = 6, j = 6

i 는 값이 증가되기 전에 참조되므로 println메서드에 i에 저장된 값 5를 넘겨주고 나서 i의 값이 증가하기 때문에 5가 출력되고, j의 경우 j에 저장된 값을 증가 시킨 후에 println메서드에 값을 넘겨주므로 6이 출력된다. 결과적으로는 i, j 모두 1씩 증가되어 6이 된다.

감소연산자(-- )는 피연산자의 값을 1 감소시킨다는 것을 제외하고는 증가연산자와 동일하다.

#### [알아두면 좋아요] ++i 와 i = i+1의 비교

두 수식의 결과는 같지만, 실제로 연산이 수행되는 과정은 다르다. ++i가 i = i + 1보다 더 적은 명령만으로 작업을 수행하기 때문에 더 빠르다. 그리고, ++i를 사용하면, 수식을 보다 더 간략히 할 수 있다.

수 식	<code>i = i + 1;</code>	<code>++i;</code>
컴파일된 코드	<pre> istore_1 iload_1 iconst_1 iadd istore_1 </pre>	<pre> istore_1 iinc 1 1 </pre>

[표3-3] `i=i+1`과 `++i`의 바이트코드 비교

위의 표에서는 `i = i + 1`과 `++i`를 컴파일 했을 때 생성되는 클래스 파일(\*.class)의 바이트코드 명령어를 비교한 것이다.

`i = i + 1`은 5개의 명령어로 이루어져 있지만, `++i`는 단 2개의 명령어로 이루어져 있다. 같은 결과를 얻지만, `i = i + 1`에 비해 `++i`가 훨씬 적은 명령만으로 수행된다는 것을 알 수 있다. 그리고 덧셈연산자(+)는 필요에 따라 피연산자를 형변환하지만 증감연산자는 형변환 없이 피연산자의 값을 변경한다.

## 2.2 부호 연산자 - +, -

부호연산자는 피연산자의 부호를 변경하는데 사용되며, boolean형과 char형을 제외한 나머지 기본형에 사용할 수 있다.

부호연산자 '+'의 경우는 피연산자에 양수 1을 곱한 결과를, 그리고 '-'의 경우에는 피연산자에 음수 1을 곱한 결과를 얻는다.

### [예제3-4] OperatorEx4.java

```

class OperatorEx4
{
    public static void main(String[] args)
    {
        int i = -10;
        i = +i;
    }
}

```

```

        System.out.println(i);
        i=-10;
        i = -i;
        System.out.println(i);
    }
}

```

#### [실행결과]

-10

10

### 2.3 비트전환 연산자 - ~

'~'는 정수형과 char형에만 사용될 수 있으며, 피연산자를 2진수로 표현했을 때, 0은 1로 1은 0으로 바꾼다. 그래서, 연산자 '~'에 의해 비트전환 되고 나면, 피연산자의 부호가 반대로 변경된다.

[주의]byte, short, char형은 int형으로 변환된 후에 전환된다.

#### [예제3-5] OperatorEx5.java

```

class OperatorEx5
{
    public static void main(String[] args)
    {
        byte b = 10;
        System.out.println("b = " + b );
        System.out.println("~b = " + ~b);
        System.out.println("~b+1 = " + (~b+1));
    }
}

```

#### [실행결과]

```

b = 10
~b = -11
~b+1 = -10

```

결과를 보면, 어떤 양의 정수에 대한 음의 정수를 얻으려면 어떻게 해야 하는 지를 알 수 있다. 양의 정수  $b$ 가 있을 때,  $b$ 에 대한 음의 정수를 얻으려면,  $\sim b + 1$ 을 계산하면 된다. 이 사실을 통해서  $-10$ 을 2진수로 어떻게 표현할 수 있는지 알 수 있을 것이다.

2진수	10진수
0 0 0 0 1 0 1 0	10
1 1 1 1 0 1 0 1	-11
1 1 1 1 0 1 0 1	-11
0 0 0 0 0 0 0 1	+ ) 1
1 1 1 1 0 1 1 0	-10

[표3-5] 음수를 2진수로 표현하는 방법

먼저 10을 2진수로 표현한 다음 0은 1로, 1은 0으로 바꾸고 그 결과에 1을 더한다. 그러면  $-10$ 의 2진 표현을 얻을 수 있다.

#### [예제3-6] OperatorEx6.java

```

class OperatorEx6
{
    public static void main(String[] args)
    {
        byte b = 10;
//        byte result = ~b; // '~'연산의 결과가 int이기 때문에 byte형 변수에 저장할 수 없다.
        byte result = (byte)~b;    // 또는 int result = ~b;와 같이 해야 한다.

        System.out.println("b = " + b);
    }
}

```

```

        System.out.println("~b = " + result );
    }
}

```

#### [실행결과]

```

b = 10
~b = -11

```

연산자 '~'는 피연산자의 타입이 int형 보다 작으면, int형으로 변환한 다음 연산을 하기 때문에 위의 예제에서는 byte형 변수 b가 int형으로 변환된 다음에 연산이 수행되어 연산결과가 int형 이 된다.

그래서 연산자 '~'의 연산결과를 저장하기 위해서는 int형 변수에 담거나, 캐스트 연산자를 사용해야한다.

## 2.4 논리부정 연산자 - !

이 연산자는 boolean형에만 사용할 수 있으며, true는 false로 false는 true로 변경한다. 조건문과 반복문의 조건식에 사용되어 조건식을 보다 효율적으로 만들어 준다.

연산자 '!'를 이용해서 한번 누르면 켜지고, 다시 한번 누르면 꺼지는 TV의 전원버튼과 같은 토글버튼(Toggle button)을 논리적으로 구현할 수 있다.

#### [예제3-7] OperatorEx7.java

```

class OperatorEx7 {
    public static void main(String[] args) {
        boolean power = false;
        System.out.println(power);
        power = !power;        // power의 값이 false에서 true로 바뀐다.
        System.out.println(power);
        power = !power;        // power의 값이 true에서 false로 바뀐다.
        System.out.println(power);
    }
}

```

[실행결과]

false

true

false



# [Java의 정석]제3장 연산자 - 3.산술연산자

자바의정석

2012/12/22 17:04

<http://blog.naver.com/gphic/50157739642>

## 3. 산술 연산자

산술연산자인 사칙연산자(+, -, \*, /), 나머지연산자(%), 쉬프트연산자(<<, >>, >>>)는 모두 두개의 피연산자를 취하는 이항연산자이며, 이항 연산자는 피연산자의 크기가 4 byte보다 작으면 4byte(int형)로 변환한 다음에 연산을 수행한다는 점을 명심해야한다.

이항연산자는 연산을 수행하기 전에 피연산자들의 타입을 일치시킨다는 사실 또한 매우 중요하다.

이항연산자는 연산을 수행하기 전에

- 크기가 4 byte이하인 자료형을 int형으로 변환한다.
- 피연산자의 타입을 일치시킨다.

### 3.1 사칙연산자 - +, -, \*, /

이 연산자들이 프로그래밍에 가장 많이 사용되어지는 연산자들 일 것이다. 여러분들이 이미 알고 있는 것처럼, 곱셈(\*), 나눗셈(/), 나머지(%) 연산자가 덧셈(+), 뺄셈(-)연산자보다 우선순위가 높다.

- int형(4 byte)보다 크기가 작은 자료형(byte, short, char)은 int형으로 변환된 후에 연산을 수행한다.

byte + short → int + int → int

- 두 개의 피연산자중 자료형의 표현범위가 큰 쪽에 맞춰서 형변환 된 후 연산을 수행한다.

int + float → float + float → float

- 정수형간의 나눗셈에서 0으로 나누는 것은 금지되어 있다.

피연산자1	피연산자2	연산결과
byte, short, char	byte, short, char	int
byte, short, char, int	int	int
byte, short, char, int, long	long	long
byte, short, char, int, long, float	float	float
byte, short, char, int, long, float, double	double	double

[표3-6]피연산자의 타입에 따른 이항연산결과

```

byte + byte -> int +int -> int
byte + short -> int + int -> int
char + char -> int + int -> int
int +int -> int
float + int -> float + float -> float
long + float -> float + float -> float
float + double -> double + double -> double

```

[참고]위의 결과는 덧셈연산자를 포함한 모든 이항 연산자에 공통적으로 해당한다.

피연산자가 정수형인 경우, 나누는 수로 0을 사용할 수 없다. 만일 0으로 나누면, 컴파일은 정상적으로 되지만 실행 시 오류(ArithmeticException)가 발생한다.

부 동소수점값인 0.0f, 0.0d으로 나누는 것은 가능하지만 그 결과는 NaN(Not A Number, 숫자 아님)이다. 나눗셈 연산자(/)와 나머지 연산자(%)의 피연산자가 무한대(Infinity) 또는 0.0인 경우의 결과를 표로 정리해 놓았다. 중요한 것은 아니니 참고만 하도록 하자.

x	y	x / y	x % y
유한수	$\pm 0.0$	$\pm \text{Infinity}$	NaN
유한수	$\pm \text{Infinity}$	$\pm 0.0$	x
$\pm 0.0$	$\pm 0.0$	NaN	NaN
$\pm \text{Infinity}$	유한수	$\pm \text{Infinity}$	NaN
$\pm \text{Infinity}$	$\pm \text{Infinity}$	NaN	NaN

[표3-7]피연산자가 유한수가 아닌 경우의 연산결과

#### [예제3-8] OperatorEx8.java

```
class OperatorEx8
{
    public static void main(String[] args)
    {
        byte a = 10;
        byte b = 20;
        byte c = a + b;
        System.out.println(c);
    }
}
```

#### [컴파일결과]

```
OperatorEx8.java:7: possible loss of precision
found   : int
required: byte
    byte c = a + b;
    ^
1 error
```

이 예제를 컴파일하면 위와 같은 에러가 발생한다. 발생한 위치는 7번째 줄(byte c=a+b;)이다. a와 b는 모두 int형보다 작은 byte형이기 때문에 +연산자는 이 두 개의 피연산자들의 자료형을 int형으로 변환한 다음 연산(덧셈)을 수행한다.

따라서 a+b의 결과는 int형(4byte)인 것이다. 4 byte의 값을 1 byte의 변수에 형변환 없이 저장하려고 했기 때문에 에러가 발생하는 것이다. 크기가 작은 자료형의 변수를 큰 자료형의 변수

수에 저장할 때는 자동으로 형변환(type conversion, casting)되지만, 반대로 큰 자료형의 값을 작은 자료형의 변수에 저장하려면 명시적으로 캐스트 연산자를 사용해서 변환해주어야 한다.

에러가 발생한 7번째 줄 `byte c = a + b;`를 `byte c = (byte)(a + b);`와 같이 변경해야 한다.

[참고] `byte c = (byte)a + b;`와 같이 하면, 역시 에러가 발생한다. 왜냐하면, 캐스트 연산자는 단항연산자이므로 연산순위가 이항 연산자인 덧셈연산자보다 높다. 그렇기 때문에, `(byte)a`가 먼저 수행된 다음 덧셈이 수행되므로, 캐스트 연산자에 의해서 byte형으로 형변환된 후에 다시 덧셈 연산자에 의해서 int형으로 변환된다.

#### [예제3-9] OperatorEx9.java

```
class OperatorEx9
{
    public static void main(String[] args)
    {
        byte a = 10;
        byte b = 30;
        byte c = (byte)(a * b);
        System.out.println(c);
    }
}
```

#### [실행결과]

44

위 의 예제를 실행하면 44가 화면에 출력된다.  $10 * 30$ 의 결과는 300이지만, 형변환(캐스팅, casting)에서 배운 것처럼, 큰 자료형에서 작은 자료형으로 변환하면 데이터의 손실이 발생하므로, 값이 바뀔 수 있다. 300은 byte형의 범위를 넘기 때문에 byte형으로 변환하면 데이터 손실이 발생하여 결국 44가 byte형 변수 c에 저장된다.

아래의 표에서 알 수 있듯이 byte형(1byte)에서 int형(4byte)으로 변환하는 것은 2진수 8자리에서 32자리로 변환하는 것이기 때문에 자료 손실이 일어나지 않는다.(원래 8자리는 그대로 보존하고 나머지는 모두 0으로 채운다. 음수인 경우에는 부호를 유지하기 위해 0 대신 1로 채운다.)

반대로 int형을 byte형으로 변환하는 경우 앞의 24자리를 없애고 하위 8자리(1byte)만을 보

저 장된 값이 10인 경우 값이 작아서 상위 24자리를 잘라내도 원래 값을 유지하는데 지장이 없지만, byte형의 범위인 -128~127의 범위를 넘는 int형의 값을 byte형으로 변환하면, 원래의 값이 보존되지 않고 byte형의 범위 중 한 값을 가지게 된다. 이러한 값 손실을 예방하기 위해서는 충분히 큰 자료형을 사용해야 한다.

[표3-8] byte와 int간의 형변환

```
{
    public static void main(String[] args)
    {
        int a = 1000000;           // 1,000,000 1백만
        int b = 2000000;           // 2,000,000 2백만
        long c = a * b;             // 2,000,000,000,000 2 * 10의 12제곱
        System.out.println(c);
    }
}
```

-1454759936

53 · 신입사원

바꾸어야 한다.

[참고] 변수 a, b중 어느 한쪽만 long형으로 바꾸어도 a \* b연산을 수행하면서 둘 다 long형으로 바뀌어서 연산하므로 결과는 long형이 된다.

[참고]long형의 변수가 표현할 수 있는 값의 범위는  $-2^{63} \sim 2^{63}-1$ 이므로, 양수는 약  $9 * 10$ 의 18제곱까지 표현 가능하므로  $2 * 10$ 의 12제곱 값을 저장할 수 있다.

#### [예제3-11] OperatorEx11.java

```
class OperatorEx11
{
    public static void main(String[] args)
    {
        char c1 = 'a';        // c1에는 문자 "a"의 문자코드(유니코드)값이 저장된다.
        char c2 = c1;         // c1에 저장되어 있는 값이 c2에 저장된다.
        char c3 = '\u0000';    // c3를 null문자로 초기화 한다.

        int i = c1 + 1;        // char형이 덧셈 연산 전에 int형으로 변환되어 97+1이 수행된다.

        // 덧셈연산의 결과가 int형이므로 c3에 담기 위해서는 char형으로의 형변환이 필요하다.
        c3 = (char)(c1 + 1);
        c2++;
        c2++;

        System.out.println("i=" + i);
        System.out.println("c2=" + c2);
        System.out.println("c3=" + c3);
    }
}
```

#### [실행결과]

```
i=98
c2=c
c3=b
```

c1 + 1을 계산할 때, c1이 char형이므로 int형으로 변환한 후 덧셈연산을 수행하게 된다. c1에 저장되어 있는 코드값이 변환되어 int형 값이 되는 것이다. 따라서 c1 + 1은 97 + 1이 되고 결과적으로 int형 변수 i에는 98이 저장된다.

[참고] 소문자 a는 코드값이 10진수로 97이고 16진수로 61이다. 그래서 c1 + 1의 연산결과가 98이다. 이 값을 char형으로 변환하면, char형 변수 c3에 저장할 수 있다. 그리고 이 값은 문자의 코드값으로 인식된다. 98은 16진수로 62이며, 소문자 b를 뜻한다. 그러므로 "b"를 문자코드를 이용해서 표현하면"\u0062"이다.

c2++ 은 어떠한 형변환도 없이 c2에 저장되어 있는 값을 1 증가시키므로, 예제에서는 원래 저장되어 있던 값인 97이 1씩 두 번 증가되어 99가 된다. 코드값이 10진수로 99인 문자는 "c"이다. 따라서, c2를 출력하면, "c"가 화면에 나타나는 것이다.

[참고] c2++;대신에 c2=c2+1;을 사용하면 에러가 발생할 것이다. c2+1의 연산결과는 int형이며, 그 결과를 다시 c2에 담으려 하면 캐스트 연산자를 사용하여 char형으로 형변환을 해야 하기 때문이다.

따라서 c2++;대신 c2=(char)(c2+1);과 같이 하면 똑같은 결과를 얻을 수 있을 것이다.

#### [예제3-12] OperatorEx12.java

```
class OperatorEx12
{
    public static void main(String[] args)
    {
        char c = 'a';
        for(int i=0; i<26; i++) {           // 블록} 안의 문장을 26번을 반복한다.
            System.out.print(c++); // "a"부터 시작해서 26개의 문자를 출력하게 된다.
        }

        System.out.println();

        c = 'A';
        for(int i=0; i<26; i++) {           // 블록} 안의 문장을 26번을 반복한다.
```

```

        System.out.print(c++); // "A"부터 시작해서 26개의 문자를 출력하게 된
다.
    }

    System.out.println();

    c='0';
    for(int i=0; i<10; i++) {        // 불럭{} 안의 문장을 10번을 반복한다.
        System.out.print(c++); // "0"부터 시작해서 10개의 문자를 출력하게 된
다.
    }
    System.out.println();
}
}

```

#### [실행결과]

```

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789

```

[참고] println메서드는 값을 출력하고 줄을 바꾸지만, print메서드는 줄을 바꾸지 않고 출력한다. 매개변수없이 println메서드를 호출하면, 아무 것도 출력하지 않고 단순히 줄을 바꾸고 다음 줄의 처음으로 출력위치를 이동시킨다.

위 의 예제를 실행하면, 문자 a부터 시작해서 26개의 문자를 출력하고, 또 문자 A부터 시작해서 26개의 문자, 0부터 9까지 10개의 문자를 출력한다. 소문자 a부터 z까지, 그리고 대문자 A부터 Z까지, 숫자 0부터 9까지 연속적으로 코드가 지정되어 있기 때문에 이런 결과가 나타난다.

문자 a의 코드값은 10진수로 97, b의 코드값은 98, c의 코드값은 99, ... , z의 코드값은 122이며, 문자 A의 코드값은 10진수로 65, B의 코드값은 66, C의 코드값은 67, ... , Z의 코드값은 90이다. 그리고 문자 0의 코드값은 10진수로 48이다.

이 사실을 이용하면 대문자를 소문자로 소문자를 대문자로 변환하는 프로그램을 작성할 수 있다.

[참고]대문자와 소문자간의 코드값 차이는 10진수로 32이다.



#### [예제3-13] OperatorEx13.java

```
class OperatorEx13
{
    public static void main(String[] args)
    {
        char lowerCase = 'a';
        char upperCase = (char)(lowerCase - 32);
        System.out.println(upperCase);
    }
}
```

#### [실행결과]

A

소문자를 대문자로 변경하려면, 대문자 A가 소문자 a보다 코드값이 32가 적으므로 소문자 a의 코드값에서 32를 빼면 되고, 반대로 대문자를 소문자로 변환하려면, 대문자의 코드값에 32를 더해주면 된다.

[참고]char형과 int형간의 뺄셈연산(-) 결과는 int형이므로, 연산 후 char형으로 다시 형변환해야 한다는 것을 잊지 말자.

#### [예제3-14] OperatorEx14.java

```
class OperatorEx14
{
    public static void main(String[] args)
    {
        float pi = 3.141592f;
        float shortPi = (int)(pi * 1000) / 1000f;

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

#### [실행결과]

3.141

int 형간의 나눗셈 int / int을 수행하면 결과가 float나 double가 아닌 int임에 주의하라. 그리고 나눗셈의 결과를 반올림을 하는 것이 아니라 버린다는 점에 유의 하도록 한다. 예를 들어 3 / 2의 결과는 1.5 또는 2가 아니라 1이다.

이 예제는 나눗셈 연산자의 이러한 성질을 이용해서 실수형 변수 pi의 값을 소수점 셋째 자리까지만 빼내는 방법을 보여 준다.

```
(int)(pi * 1000) / 1000f;
```

위의 수식에서 제일 먼저 수행되는 것은 괄호 안의 pi \* 1000이다. pi가 float이고 1000이 정수형이니까 연산의 결과는 float인 3141.592f가 된다.

```
(int)(3141.592f) / 1000f;
```

그 다음으로는 단항연산자인 캐스트연산자의 형변환이 수행된다. 3141.592f를 int로 변환하면 3141를 얻는다.

```
3141 / 1000f;
```

int와 float의 연산이므로, int가 float로 변환된 다음, float와 float의 연산이 수행된다.

```
3141.0f / 1000f
```

float와 float의 나눗셈이므로 결과는 float인 3.141f가 된다.

[참고] 1000f는 float형 접미사가 붙었으므로 float이며 1000.0f와 같다.

[참고] DecimalFormat클래스를 사용해서 소수점자리를 맞추는 것도 좋은 방법이다.

추가로 원하는 자릿수에서 반올림을 하고 나머지를 버리는 예를 하나 더 보자.

#### [예제3-15] OperatorEx15.java

```
class OperatorEx15
{
    public static void main(String[] args)
    {
        float pi = 3.141592f;
        float shortPi = Math.round(pi * 1000) / 1000f;

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

#### [실행결과]

3.142

[참고] Math.round(3141.592f)의 결과는 3142이다. 소수점 첫 째 자리에서 반올림을 하기 때문이다.

이 예제의 결과는 pi의 값을 소수점 넷째 자리인 5에서 반올림을 해서 3.142가 출력되었다. round메서드는 매개변수로 받은 값을 소수점 첫째자리에서 반올림을 하고 그 결과를 정수로 돌려주는 메서드이다.

여기서는 소수점 넷 째 자리에서 반올림 하기 위해 1000을 곱했다가 다시 1000f로 나누었다. 만일 1000f가 아닌 1000으로 나누었다면, 3.142가 아닌 3을 결과로 얻었을 것이다.

[참고]일반적으로 클래스의 메서드는 그 클래스의 객체를 만든 후에 호출할 수 있지만, round 메서드와 같은 static메서드는 객체를 만들지 않고도 직접 호출할 수 있다.

### 3.2 나머지 연산자 - %

왼쪽의 피연산자를 오른쪽 피연산자로 나누고 난 나머지 값을 돌려주는 연산자이다. boolean 형을 제외하고는 모든 기본형 변수에 사용할 수 있다. 나머지 연산자는 주로 짝수, 홀수 또는 배수 검사 등에 주로 사용된다.

나눗셈에서와 같이 피연산자가 정수형인 연산에서는 나누는 수(오른쪽 피연산자)로 0을 사용할 수 없고, 나머지 연산자 역시 0.0이나 0.0f로 나누는 것은 허용한다.

#### [예제3-16] OperatorEx16.java

```
class OperatorEx16
{
    public static void main(String[] args)
    {
        int share = 10 / 8;
        int remain = 10 % 8;
        System.out.println("10을 8로 나누면, ");
        System.out.println("몫은 " + share + "이고, 나머지는 " + remain + "입니다.");
    }
}
```

#### [실행결과]

10을 8로 나누면,  
몫은 1이고, 나머지는 2입니다.

#### [예제3-17] OperatorEx17.java

```
class OperatorEx17
{
    public static void main(String[] args)
    {
        for(int i=1; i <=10; i++) { // i가 1부터 10이 될 때까지, {}안의 문장을 반복 수행한다.
            if(i%3==0) {           // i가 3으로 나누어 떨어지면, 3의 배수이므로 출력한다.
                System.out.println(i);
            }
        }
    }
}
```

<pre>         }     } }</pre>
[실행결과]
3
6
9

조건문과 반복문을 사용해서, 1과 10사이의 정수에서 3의 배수인 숫자만 출력하는 예제이다. 반복문for는 i값을 1부터 10까지 1씩 증가시키면서 괄호} 안의 문장들을 반복해서 수행한다. 조건문 if는 조건이 만족하는 경우만 괄호}안의 문장들을 수행한다. 따라서, i%3의 결과가 0인 경우만 i의 값을 화면에 출력하는 것이다.

[예제3-18] OperatorEx18.java
<pre> class OperatorEx18 {     public static void main(String[] args)     {         System.out.println(-10%8);         System.out.println(10%-8);         System.out.println(-10%-8);     } }</pre>
[실행결과]
-2
2
-2

피연산자 중에 음의 부호가 있는 경우에 어떤 결과를 얻게 되는지를 보여 주는 예제이다. 결과에서 알 수 있듯이, %연산자의 왼쪽에 있는 피연산자(나눠지는 수)의 부호를 따르게 된다. 간단히 말해서 피연산자의 부호를 모두 무시하고 나머지 연산을 한 결과에 나눠지는 수의 부호를 붙이면 된다.

### 3.3 쉬프트연산자 - <<, >>, >>>

쉬프트연산자는 정수형 변수에만 사용할 수 있는데, 피연산자의 각 자리(2진수로 표현했을 때)를 오른쪽 또는 왼쪽으로 이동(shift)한다고 해서 쉬프트연산자(shift operator)라고 한다. 오른쪽으로 n자리를 이동하면, 피연산자를  $2^n$ 로 나눈 것과 같은 결과를 얻을 수 있고, 왼쪽으로 n자리를 이동하면  $2^n$ 으로 곱한 것과 같은 결과를 얻을 수 있다.

$x \ll n$ 는  $x * 2^n$ 의 결과와 같다.

$x \gg n$ 는  $x / 2^n$ 의 결과와 같다.

"<<" 연산자의 경우, 피연산자의 부호에 상관없이 자리를 왼쪽으로 이동시키며 빈칸을 0으로만 채우면 되지만, ">>" 연산자는 오른쪽으로 이동시키기 때문에 음수인 경우 부호를 유지시켜 주기 위해서 음수인 경우 빈자리를 1로 채우게 된다. 반면에 ">>>" 연산자는 부호에 상관없이 항상 0으로 빈자리를 채운다.

그렇기 때문에, 음수에 ">>>" 연산을 행한 후, 10진수로 변환해 보면, 뜻밖의 결과를 얻게 될 것이다. "<<", ">>" 연산자와는 달리 ">>>" 연산자의 결과는 10진수 보다는 2진수로 표현했을 때 기호로서 더 의미를 가지므로, 10진 연산보다는 비트연산(2진 연산)에 주로 사용된다.

[참고]여기서 숫자 n의 값이 자료형의 bit수 보다 크면, 자료형의 bit수로 나눈 나머지 만큼만 이동한다. 예를 들어 int형의 경우, 4byte(32bit)이므로 자리수를 32번 바꾸면 결국 제자리로 돌아오기 때문에,

$1000 \gg 32$ 는 수행하면 아무 일도 하지 않아도 된다.  $1000 \gg 35$ 는 35를 32로 나눈 나머지인 3만큼만 이동하는  $1000 \gg 3$ 을 수행한다.

수 식	자리아동	연 산 결 과	
		2 진수	10 진수
8 >> 0	없음	00000000 00000000 00000000 00001000	8
8 >> 1	오른쪽으로 한 번	00000000 00000000 00000000 00000100	4
8 >> 2	오른쪽으로 두 번	00000000 00000000 00000000 00000010	2
-8 >> 0	없음	11111111 11111111 11111111 11111000	-8
-8 >> 1	오른쪽으로 한 번	11111111 11111111 11111111 11111100	-4
-8 >> 2	오른쪽으로 두 번	11111111 11111111 11111111 11111110	-2

수 식	자	연 산 결 과	
		2 진수	10 진수
8 << 0	없음	00000000 00000000 00000000 00001000	8
8 << 1	왼쪽으로 한 번	00000000 00000000 00000000 00010000	16
8 << 2	왼쪽으로 두 번	00000000 00000000 00000000 00100000	32
-8 << 0	없음	11111111 11111111 11111111 11111000	-8
-8 << 1	왼쪽으로 한 번	11111111 11111111 11111111 11110000	-16
-8 << 2	왼쪽으로 두 번	11111111 11111111 11111111 11100000	-32

수 식	자	연 산 결 과	
		2 진수	10 진수
8 >>> 0	없음	00000000 00000000 00000000 00001000	8
8 >>> 1	오른쪽으로 한 번	00000000 00000000 00000000 00000100	4
8 >>> 2	오른쪽으로 두 번	00000000 00000000 00000000 00000010	2
-8 >>> 0	없음	11111111 11111111 11111111 11111000	-8
-8 >>> 1	오른쪽으로 한 번	01111111 11111111 11111111 11111100	2147483644
-8 >>> 2	오른쪽으로 두 번	00111111 11111111 11111111 11111110	1073741822

[표3-9]쉬프트연산의 예

[참고]쉬프트연산자 역시 int보다 작은 타입의 변수는 int로 변환한 다음에 연산을 수행하므로 8은 4byte(32bit), 따라서 32자리의 2진수로 표현된다.

곱셈이나 나눗셈 연산자를 사용하면 같은 결과를 얻을 수 있는데, 굳이 쉬프트연산자를 제공하는 이유는 속도 때문이다.

예를 들어 8 >> 2 의 결과는 8 / 4의 결과와 같다. 하지만, 8 / 4를 연산하는데 걸리는 시간보다 8 >> 2를 연산하는데 걸리는 시간이 더 적게 걸린다. 다시 말하면, ">>" 또는 "<<" 연산자를 사용하는 것이 나눗셈(/) 또는 곱셈(\*) 연산자를 사용하는 것 보다 더 빠르다.

쉬프트연산자보다 곱셈 또는 나눗셈연산자를 주로 사용하고, 보다 빠른 실행속도가 요구되어

지는 곳에만 쉬프트연산자를 사용하도록 한다.

[예제3-20] OperatorEx20.java

```
class OperatorEx20 {
    public static void main(String args[]) {
        int temp;                // 계산 결과를 담기 위한 변수

        System.out.println(-8);
        System.out.println(Integer.toBinaryString(-8)); // -8을 2진수 문자열로 변경한
다.

        System.out.println();    // 줄바꿈을 한다.

        temp = -8 << 1;
        System.out.println( "-8 << 1 = " + temp);
        System.out.println(Integer.toBinaryString(temp));
        System.out.println();

        temp = -8 << 2;
        System.out.println( "-8 << 2 = " + temp);
        System.out.println(Integer.toBinaryString(temp));
        System.out.println();

        System.out.println();
        System.out.println(-8);
        System.out.println(Integer.toBinaryString(-8));
        System.out.println();

        temp = -8 >> 1;
        System.out.println( "-8 >> 1 = " + temp);
        System.out.println(Integer.toBinaryString(temp));
        System.out.println();

        temp = -8 >> 2;
        System.out.println( "-8 >> 2 = " + temp);
```



```

        System.out.println(Integer.toBinaryString(temp));
        System.out.println();

        System.out.println();
        System.out.println(-8);
        System.out.println(Integer.toBinaryString(-8));
        System.out.println();

        temp = -8 >>> 1;
        System.out.println( "-8 >>> 1 = " + temp);
        System.out.println(Integer.toBinaryString(temp));
        System.out.println();

        temp = -8 >>> 2;
        System.out.println( "-8 >>> 2 = " + temp);
        System.out.println(Integer.toBinaryString(temp));
        System.out.println();
    }
}

```

#### [실행결과]

```

-8
11111111111111111111111111111000

-8 << 1 = -16
111111111111111111111111111110000

-8 << 2 = -32
1111111111111111111111111111100000

-8
11111111111111111111111111111000

-8 >> 1 = -4
11111111111111111111111111111100

```

-8 >> 2 = -2

11111111111111111111111111111110

-8

11111111111111111111111111111000

-8 >>> 1 = 2147483644

1111111111111111111111111111100

-8 >>> 2 = 1073741822

111111111111111111111111111110

[참고] -8 >>> 1과 -8 >>> 2의 결과에서 맨 앞의 0은 생략된 것이다.

# [Java의 정석]제4장 조건문과 반복문 - 1.조건문

자바의정석

2012/12/22 17:05

<http://blog.naver.com/gphic/50157739723>

## 1. 조건문 - if, switch

조건문은 조건식과 문장을 포함하는 블록{}으로 구성되어 있으며, 조건식의 연산결과에 따라 실행될 문장을 달리 할 수 있다.

처리해야할 경우의 수가 많을 때는 switch문을 사용해서 표현할 수 있는지 살펴봐야 한다.

[참고] 모든 switch문은 if문으로 변경이 가능하지만, 모든 if문이 switch문으로 변경 가능한 것은 아니다.

### 1.1 if문

if문은 널리 사용되는 조건문이며, 기본구조는 다음과 같다.

```
if (조건식) {  
    // 조건식의 연산결과가 true일 때 수행될 문장들을 적는다.  
}
```

if 다음에 오는 조건식에는 연산의 최종결과 값이 true 또는 false인 수식만을 사용할 수 있다. 조건식의 결과가 false이면, 블록{} 내의 문장이 실행되지 않는다.

[참고] C언어에서는 조건식의 최종결과 값으로 true 또는 false 이외의 값을 허용하지만, 자바에서는 이를 허용하지 않는다.

if 문의 변형인 if-else문의 기본 구조는 다음과 같다. if문에 else블럭을 추가한 것이다. else블럭 내의 문장들은 if문의 조건식의 결과가 false일 때 수행되는 문장들이다. 조건식의 결과에 따라 이 두 개의 블럭 중 어느 한 블럭의 내용만 수행하고 전체 if문을 벗어나게 된다.

```

if (조건식) {
    // 조건식의 연산결과가 true일 때 수행될 문장들을 적는다.
} else {
    // 조건식의 연산결과가 false일 때 수행될 문장들을 적는다.
}

```

if문의 또 다른 변형으로 if-else if가 있는데 기본 구조는 다음과 같다. 여러 개의 블록 중 조건식을 만족하는 하나의 블록만을 수행하고 if문 전체를 빠져나가게 된다.  
if-else if의 기본구조는 다음과 같다.

```

if (조건식1) {
    // 조건식1의 연산결과가 true일 때 수행될 문장들을 적는다.
} else if (조건식2) {
    // 조건식2의 연산결과가 true일 때 수행될 문장들을 적는다.
} else if (조건식3) {    // 여러 개의 else if를 사용할 수 있다.
    // 조건식3의 연산결과가 true일 때 수행될 문장들을 적는다.
    //...
} else {    // 보통 else블록으로 끝나며, else블록은 생략이 가능하다.
    // 위의 어느 조건식도 만족하지 않을 때 수행될 문장들을 적는다.
}

```

블록{}은 여러 개의 문장을 하나로 묶기 위해 사용되는 것이며, 함수, 조건문, 반복문 등에 사용된다. 조건문과 반복문에서는 수행될 문장이 하나인 경우 블록을 생략할 수 있으나, 가능하면 생략 않고 사용하는 것이 바람직하다.

수행될 문장이 한 문장이라서 블록을 생략하고 적었을 때, 나중에 새로운 문장들이 추가되면 블록으로 문장들을 감싸 주어야 하는데 이 때 블록을 추가하는 것을 잊기 쉽기 때문이다. 그리고, 여러 개의 if문이 중첩되어 사용되었을 때 if문과 else블록의 관계가 의도한 바와 다르게 형성될 수도 있다.

[예제4-1] FlowEx1.java

```

class FlowEx1
{
    public static void main(String[] args)
    {
        int visitCount = 0;
        if (visitCount < 1) {
            System.out.println("처음 오셨군요. 방문해 주셔서 감사합니다.");
        }
    }
}

```

#### [실행결과]

처음 오셨군요. 방문해 주셔서 감사합니다.

#### [예제4-2] FlowEx2.java

```

class FlowEx2
{
    public static void main(String[] args)
    {
        int visitCount = 5;
        if (visitCount < 1) {    // 5 < 1의 연산결과는 false.
            System.out.println("처음 오셨군요. 방문해 주셔서 감사합니다.");
        } else {
            System.out.println("다시 방문해 주셔서 감사합니다.");
        }
        System.out.println("방문횟수는 " + ++visitCount + "번 입니다.");
    }
}

```

#### [실행결과]

다시 방문해 주셔서 감사합니다.

방문횟수는 6번 입니다.

#### [예제4-3] FlowEx3.java

```

class FlowEx3
{
    public static void main(String[] args)
    {
        int score = 45;
        char grade = 'Wu0000';

        if (score >= 90)           // score가 90점 보다 같거나 크면 A학점(grade)
        {
            grade = 'A';
        } else if (score >=80){    // score가 80점 보다 같거나 크면 B학점(grade)
            grade = 'B';
        } else {                  // 나머지는 C학점(grade)
            grade = 'C';
        }

        System.out.println("당신의 학점은 " + grade + "입니다.");
    }
}

```

#### [실행결과]

당신의 학점은 C입니다.

if문은 삼항연산자(? :)로 바꿀 수 있는 경우가 많이 있는데, 간단한 if문의 경우 삼항연산자를 사용한 문장들로 변경하는 것을 고려해보도록 한다.

위의 예제에서 if문 대신 삼항연산자를 사용하면 다음과 같다.

#### [예제4-4] FlowEx4.java

```

class FlowEx4
{
    public static void main(String[] args)
    {
        int score = 45;

```

```

char grade = 'Wu0000';
grade = (score >=90) ? 'A' : ((score >=80) ? 'B' : 'C');

System.out.println("당신의 학점은 " + grade + "입니다.");
}
}

```

#### [실행결과]

당신의 학점은 C입니다.

두 예제는 모두 같은 결과를 얻지만, 삼항연산자를 사용한 두 번째 예제가 보다 더 간결하다. 이렇듯 삼항연산자를 활용하면, if문을 간단히 표현할 수 있다.

## 1.2 중첩 if문

if문의 블록 내에 또 다른 if문을 사용하는 것이 가능하며, 이것을 중첩 if문이라고 부른다.

```

if (조건식1) {
    // 조건식1의 연산결과가 true일 때 수행될 문장들을 적는다.
    if (조건식2) {
        // 조건식1과 조건식2가 모두 true일 때 수행될 문장들
    } else {
        // 조건식1이 true이고, 조건식2가 false일 때 수행되는 문장들
    }
} else {
    // 조건식1이 false일 때 수행되는 문장들
}

```

#### [예제4-5] FlowEx5.java

```
class FlowEx5
```

```

{
    public static void main(String[] args)
    {
        int score = 82;
        String grade = "";           // 두 문자를 저장할 것이므로 String으로 했음

        System.out.println("당신의 점수는 " + score + "입니다.");
        if (score >= 90)              // score가 90점 보다 같거나 크면 A학점(grade)
        {
            grade = "A";
            if ( score >= 98) { // 90점 이상 중에서도 98점 이상은 A+
                grade += "+";    // grade = grade + "+";와 같다.
            } else if ( score < 94)    {
                grade += "-";
            }
        } else if (score >= 80){      // score가 80점 보다 같거나 크면 B학점(grade)
            grade = "B";
            if ( score >= 88) {
                grade += "+";
            } else if ( score < 84)    {
                grade += "-";
            }
        }

        } else {                     // 나머지는 C학점(grade)
            grade = "C";
        }

        System.out.println("당신의 학점은 " + grade + "입니다.");
    }
}

```

#### [실행결과]

당신의 점수는 82입니다.

당신의 학점은 B-입니다.

위 예제에서 보듯이, 모두 3개의 if문으로 이루어져 있으며 if문 내의 블록에 또 다른 2개의 if문



을 포함하고 있는 모습을 하고 있다. 제일 바깥쪽에 있는 if문에서 점수에 따라 학점(grade)을 결정하고, 내부의 if문에서는 학점을 더 세부적으로 나누어서 평가를 하고 그 결과를 출력한다.

### 1.3 switch문

조건의 경우의 수가 많을 때는 if문 보다 switch문을 사용하는 것이 더 간결하고 알아보기 쉽다. if문은 조건식의 결과가 true, false 두 가지 밖에 없기 때문에 경우의 수가 많아 질수록 계속 else-if를 추가해야하므로 조건식이 많아져서 복잡해지고, 여러 개의 조건식을 계산해야하므로 수행시간도 많이 걸린다. 하지만, switch문의 조건식은 결과값으로 int형 범위의 정수값을 허용하므로, 하나의 조건식만 계산하면 그 결과에 따라서 해당 문장들을 수행하면 되므로 같은 기능의 if문보다 속도가 빠르다.

하지만, switch문은 if문 보다 제약조건이 많기 때문에, 조건식을 switch문으로 작성할 수 없는 경우가 많다.

switch문의 기본구조는 아래와 같다.

```
switch (조건식) {
    case 값1 :
        // 조건식의 결과가 값1과 같을 경우 수행될 문장들
        //...
        break;
    case 값2 :
        // 조건식의 결과가 값2와 같을 경우 수행될 문장들
        break;
        //...
    //...
    default :
        // 조건식의 결과와 일치하는 case문이 없을 때 수행될 문장들
        //...
}
```

switch 문의 조건식은 연산결과가 int형 범위의 정수값이어야한다. byte, short, char, int 타입의 변수나 리터럴을 사용할 수 있다. 그리고, case문에는 반드시 상수값만을 허용한다. 변수는 허용되지 않으므로 유의해야한다.

switch문의 조건식을 먼저 계산한 다음, 그 결과와 일치하는 case문으로 이동한다. 이동한 case문 이하에 있는 문장들을 수행하며, break문을 만나면 전체 switch문을 빠져나가게 된다. 만일 case문 아래에 break문을 생략하면, 다른 break문을 만나거나 switch문 블록의 끝을 만날 때까지 나오는 모든 문장들을 수행한다.

#### [예제4-6] FlowEx6.java

```
class FlowEx6
{
    public static void main(String[] args)
    {
        // Math클래스의 random()함수를 이용해서 1~10사이의 수를 임의로 얻어낼 수
        // 있다.
        int score = (int)(Math.random() * 10) + 1;

        switch(score*100) {
            case 100 :
                System.out.println("당신의 점수는 100이고, 상품은 자전거입니다.");

                break;
            case 200 :
                System.out.println("당신의 점수는 200이고, 상품은 TV입니다.");

                break;
            case 300 :
                System.out.println("당신의 점수는 300이고, 상품은 노트북 컴퓨터입니
다.");
                break;
            case 400 :
                System.out.println("당신의 점수는 400이고, 상품은 자동차입니다.");

                break;
```

```

        default :
            System.out.println("죄송하지만 당신의 점수에 해당하는 상품이 없습니
다.");
        }
    }
}

```

#### [실행결과]

당신의 점수는 100이고, 상품은 자전거입니다.

Math클래스의 random메서드를 사용해서 1과 10사이의 임의의 수를 얻은 다음, 그 값을 score에 저장한다. score에 100을 곱한 값과 일치 하는 case문을 찾아서 이동한다.

예 를 들어서 score값이 1이라면, score \* 100의 결과는 100이 되고, 첫 번째 case문인 case 100:으로 이동하여 System.out.println("당신의 점수는 100이고, 상품은 자전거입니다.")문장을 수행한 후, break문을 만나서 switch문을 빠져 나오게 된다.

이 예제에서는 switch문의 조건식인 score \* 100의 결과가, 100, 200, 300, 400인 경우에 대해서만 다루고 있기 때문에 그 외의 값인 500, 600, 700, 800, 900, 1000의 경우에는 default문 이하의 문장들 수행된다.

[참고]random메서드를 사용했기 때문에 매번 수행할 때마다 다른 결과를 얻게 될 것이다.

random메서드는 0.0과 1.0사이의 값 중 하나의 double값을 생성한다.(0.0은 범위에 포함되고 1.0은 포함되지 않는다.)

$$0.0 \leq \text{Math.random}() < 1.0$$

예를 들어 1 과 10 사이의 정수를 구하기를 원한다면 아래와 같은 순서를 따르도록 한다.

1. 각 번에 10을 곱한다.

$$\begin{aligned}
 0.0 * 10 &\leq \text{Math.random}() * 10 < 1.0 * 10 \\
 0.0 &\leq \text{Math.random}() * 10 < 10.0
 \end{aligned}$$

2. 각 변을 int형으로 변환한다.

```
(int)0.0 <= (int)(Math.random() * 10) < (int)10.0  
0 <= (int)(Math.random() * 10) < 10
```

지금까지는 0과 9사이의 정수 중 하나를 가질 수 있다.(0은 포함, 10은 포함 안됨)

3. 각 변에 1을 더한다.

```
0+1 <= (int)(Math.random() * 10) +1 < 10 +1  
1 <= (int)(Math.random() * 10) +1 < 11
```

자, 이제는 1과 10사이의 정수 중 하나를 얻을 수 있다.(1은 포함, 11은 포함 안됨)

[참고]순서 2와 3을 바꿔서, 각 변에 1을 먼저 더한 다음 int형으로 변환해도 같은 결과를 얻는다.

위와 같은 방식으로 식을 변환해가며 범위를 조절하면 원하는 범위의 값을 얻을 수 있다. 주사위를 던졌을 때 나타나는 임의의 값을 얻기 위해서는 10대신 6을 곱하면 된다. 그렇게 하면 1과 6사이의 값을 얻어낼 수 있을 것이다.

random메서드의 활용 예를 한가지 더 소개하자면, 숫자 대신 임의의 문자를 얻을 수 있도록 하는 것이다. 대문자 A~Z사이의 문자 중 임의의 한 문자를 얻도록 하려면 다음과 같이 한다.

[참고]대문자 26개의 코드값은 65부터 90까지(A~Z)이다.

```
0.0 <= Math.random() < 1.0
```

1. 발생시키려는 수의 개수가 26개 이므로 각 변에 26을 곱한다.

```
0.0 * 26 <= Math.random() * 26 < 1.0 * 26  
0.0 <= Math.random() * 26 < 26.0
```

2. 65부터 시작하므로 각 변에 65를 더한다.

```
65.0 <= Math.random() * 26 + 65 < 91.0
```

3. 각 변을 문자형(char형)으로 형변환을 한다.

```
(char)65.0 <= (char)(Math.random() * 26 + 65) < (char)91.0  
'A' <= (char)(Math.random() * 26 + 65) < '['
```

[참고] '['는 대문자 Z다음에 오는 문자.(코드값이 Z보다 1이 더 큰 문자).

여 기서 주의해야할 것은 위의 예에서와는 달리 char형으로 형변환한 후에 65를 더하면 전체 결과가 int형이 되어 버리기 때문에 char형 값을 얻을 수 없다. 그렇기 때문에 65를 먼저 더하고 그 다음에 형변환을 해야 하는 것이다.

#### [예제4-7] FlowEx7.java

```
class FlowEx7  
{  
    public static void main(String[] args)  
    {
```

```

        char ch = (char)(Math.random() * 4 +65); // A, B, C, D중의 하나를 얻을 수 있
다.

        int score = 0;

        switch (ch)
        {
            case 'A':
                score = 90;
                break;
            case 'B':
                score = 80;
                break;
            case 'C':
                score = 70;
                break;
            case 'D':
                score = 60;
                break;
        }

        System.out.println("당신의 점수는 "+ score +"점 이상 입니다.");
    }
}

```

#### [실행결과]

당신의 점수는 80점 이상 입니다.

[참고]Math.random()를 사용했기 때문에 실행결과가 이와 다를 수 있다.

#### [예제4-8] FlowEx8.java

```

class FlowEx8
{
    public static void main(String[] args)
    {

```

```

int score = 1;

switch(score*100) {
    case 100 :
        System.out.println("당신의 점수는 100이고, 상품은 자전거입니다.");

    case 200 :
        System.out.println("당신의 점수는 200이고, 상품은 TV입니다.");

    case 300 :
        System.out.println("당신의 점수는 300이고, 상품은 노트북 컴퓨터입니
다.");
    case 400 :
        System.out.println("당신의 점수는 400이고, 상품은 자동차입니다.");

    default :
        System.out.println("죄송하지만 당신의 점수에 해당하는 상품이 없습니
다.");
}
}
}

```

#### [실행결과]

당신의 점수는 100이고, 상품은 자전거입니다.

당신의 점수는 200이고, 상품은 TV입니다.

당신의 점수는 300이고, 상품은 노트북 컴퓨터입니다.

당신의 점수는 400이고, 상품은 자동차입니다.

죄송하지만 당신의 점수에 해당하는 상품이 없습니다.

위의 예제는 FlowEx6.java를 변형한 것으로 score의 값을 1로 고정시키고, switch문에 있는 break문을 모두 뺀 것이다.

score의 값이 1이므로 switch문의 조건식 결과 값은 100이 된다. 그래서 case 100:으로 이동한 후 그 이하의 문장들을 수행한다. FlowEx6.java과는 달리, break문이 없으므로 case 100:이후부터 switch문의 블록 끝까지 모든 문장들을 수행하게 된다.

그렇기 때문에 매 case문마다 break문을 사용하는 것을 잊지 않도록 한다. 때로는 switch문의

이러한 성질을 이용하기도 한다.

#### [예제4-9] FlowEx9.java

```
class FlowEx9
{
    public static void main(String[] args)
    {
        int score = (int)(Math.random() * 10) + 1;
        String msg = "";

        score *= 100;          // score = score * 100;

        msg = "당신의 점수는 " + score + "이고, 상품은 ";

        switch(score) {
            case 1000 :
                msg += "자전거, ";          // msg = msg + "자전거, ";
            case 900 :
                msg += "TV, ";
            case 800 :
                msg += "노트북 컴퓨터, ";
            case 700 :
                msg += "자전거, ";
            default :
                msg += "볼펜";
        }

        System.out.println( msg + "입니다.");
    }
}
```

#### [실행결과]

당신의 점수는 800이고, 상품은 노트북 컴퓨터, 자전거, 볼펜입니다.



Math 클래스의 random()을 사용했기 때문에 위 예제의 결과는 실행할 때마다 결과가 다를 것이다. 위의 결과는 점수가 800점이 되었을 때의 결과이다. 점수(score)는 100부터 1000점까지 100점 단위의 값을 갖을 수 있는데, 높은 점수를 얻을수록 많은 상품을 주도록 되어 있다. 예를 들어 1000점을 얻은 사람은 모든 상품을 다 가질 수 있도록 되어 있다. 이 예제에서 알 수 있듯이 switch문에 각 case문마다 반드시 break문을 사용해야하는 것은 아니며, 이 성질을 잘 이용하면 보다 간결하고 논리적으로 명확한 코드를 작성할 수 있게 된다.

한 가지 더 간단한 예를 들어 보겠다. 아래의 코드는 전체 코드가 아닌 코드의 일부를 발췌한 것인데, 회원제로 운영되는 인터넷 사이트에서 많이 사용될 만한 코드이다.

```
switch (level) {
    case 3 :
        grantDelete();    // 삭제권한을 준다.
    case 2 :
        grantWrite();     // 쓰기권한을 준다.
    case 1 :
        grantRead();      // 읽기권한을 준다.
}
```

로 로그인한 사용자의 등급(level)을 체크하여, 등급에 맞는 권한을 부여하는 방식으로 되어 있다. 제일 높은 등급인 3을 가진 사용자는 grantDelete, grantWrite, grantRead메서드가 모두 수행되어 읽기, 쓰기, 삭제 기능까지 모두 갖게 되고, 제일 낮은 등급인 1을 가진 사용자는 읽기 권한만을 갖게 된다.

[참고]위의 코드는 사용자에게 읽기, 쓰기, 삭제권한을 주는 기능의 grantRead(), grantWrite(), grantDelete()가 존재한다는 가정 하에 작성되었다.

#### [예제4-10] FlowEx10.java

```
class FlowEx10
{
    public static void main(String[] args)
```

```

{
    int score = 88;
    char grade = '\u0000';
    switch(score) {
        case 100: case 99: case 98: case 97: case 96:
        case 95: case 94: case 93: case 92: case 91:
        case 90 :
            grade = 'A';
            break;
        case 89: case 88: case 87: case 86:
        case 85: case 84: case 83: case 82: case 81:
        case 80 :
            grade = 'B';
            break;
        case 79: case 78: case 77: case 76:
        case 75: case 74: case 73: case 72: case 71:
        case 70 :
            grade = 'C';
            break;
        case 69: case 68: case 67: case 66:
        case 65: case 64: case 63: case 62: case 61:
        case 60 :
            grade = 'D';
            break;
        default :
            grade = 'F';
    } // end of switch
    System.out.println("당신의 학점은 " + grade + "입니다.");

} // end of main
} // end of class

```

#### [실행결과]

당신의 학점은 B입니다.

위의 예제는 예제4-3을 switch문을 이용해서 변형한 예제이다. 위의 예제를 if문을 이용해서

구현하려면, 조건식이 4개가 필요하며, 최대 4번의 조건식을 계산해야한다. 하지만, switch문은 조건식을 1번만 계산하면 되므로 더 빠르다. 하지만, case문이 너무 많아지므로 좋지 않다.

반드시 속도를 더 향상시켜야 한다면 복잡하더라도 switch문을 선택해야겠지만, 그렇지 않으면 이런 경우 if문이 더 적합하다.

#### [예제4-11] FlowEx11.java

```
class FlowEx11
{
    public static void main(String[] args)
    {
        int score = 88;
        char grade = '\u0000';
        switch(score/10) {
            case 10:
            case 9 :
                grade = 'A';
                break;
            case 8 :
                grade = 'B';
                break;
            case 7 :
                grade = 'C';
                break;
            case 6 :
                grade = 'D';
                break;
            default :
                grade = 'F';
        }
        System.out.println("당신의 학점은 " + grade + "입니다.");
    }
}
```

#### [실행결과]

당신의 학점은 B입니다.

이전 예제에 기교를 부려서 보다 간결하게 작성한 예제이다. score를 10으로 나누면, 전에 배운 것과 같이  $\text{int} / \text{int}$ 의 결과는  $\text{int}$ 이기 때문에, 예를 들어  $88/10$ 은 8.8이 아니라 8을 얻는다. 따라서 80과 89사이의 숫자들은 10으로 나누면 결과가 8이 된다. 마찬가지로 70~79사이의 숫자들은 10으로 나누면 7이 된다.

# [Java의 정석]제4장 조건문과 반복문 - 2.반복문

자바의정석

2012/12/22 17:05

<http://blog.naver.com/gphic/50157739752>

## 2. 반복문 - for, while, do-while

반복문은 어떤 작업이 반복적으로 수행되도록 할 때 사용되며, 반복문의 종류로는 for문과 while문, do-while문이 있다.

for문이나 while문에 속한 문장은 조건에 따라 한 번도 수행되지 않을 수 있지만 do-while문에 속한 문장은 최소한 한 번 이상 수행될 것을 보장한다.

반복문은 주어진 조건을 만족하는 동안 주어진 문장들을 반복적으로 수행하므로 조건식을 포함하며 switch문을 제외한 if, for, while문에 사용되는 조건식은 연산결과가 반드시 boolean형, 즉 true 또는 false이어야 한다.

[참고]C언어에서는 true와 false이외의 값도 허용한다.

for문과 while문은 구조와 기능이 유사하여 어느 경우에도 서로 변환이 가능하기 때문에 반복문을 작성해야 할 때 for문과 while문 중 어느 쪽을 선택해도 좋으나 for문은 주로 반복횟수를 알고 있을 때, 그리고 카운터가 반복문 내에 필요한 경우에 사용되고, 단순히 조건에 따른 반복만이 필요한 경우 while문을 사용하도록 한다.

[참고]조건식을 잘못 작성하면, 한번도 수행되지 않거나 무한히 반복하게 되므로 주의해야 한다.

### 2.1 for문

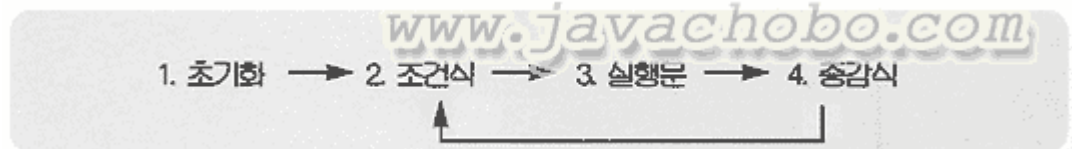
for문의 기본구조는 다음과 같다.

```
for (초기화;조건식;증감식) {  
    // 조건식의 연산결과가 true일 때 수행될 문장들을 적는다.
```

}

[참고]반복하려는 문장이 단 하나일 때는 중괄호{}를 생략할 수 있다.

초기화, 조건식, 증감식, 실행문과 같이 모두 4부분으로 이루어져 있으며, 실행순서는 아래와 같다.



초기화는 처음에만 한번 수행되고, 그 이후부터는 조건식을 만족하는 한 2-3-4의 순서로 계속 반복된다. 조건식의 결과가 false가 되면, for문 전체를 빠져나가게 된다. 초기화, 조건식, 증감식은 모두 생략이 가능하며, 조건식이 생략되면 true로 간주된다.

식	설 명
<code>for(;;) { /* 반복 수행할 문장 */ }</code>	조건식이 없기 때문에 결과가 true로 간주되어 무한반복을 하게 된다.
<code>for(int i=0;;) { /* 반복 수행할 문장 */ }</code>	for문에 int형 변수 i를 선언하고 0으로 초기화 했다. 변수 i는 for문 내에 선언되었기 때문에 for문 내에서만 유효하다.
<code>for(int i=1, j=1; i&lt;10 &amp;&amp; i*j&lt;50; i++, j+=2) { /* 반복 수행할 문장 */ }</code>	쉼표(,)를 이용해서 하나 이상의 변수를 선언하고 초기화 할 수 있다. 단, 같은 타입이어야 한다. 증감식 역시 쉼표(,)를 이용해서 여러 문장이 수행되도록 할 수 있다. 매 반복마다 i는 1씩, j는 2씩 증가한다.

[표4-1]for문의 작성예

#### [예제4-12] FlowEx12.java

```
class FlowEx12
{
    public static void main(String[] args)
    {
        int sum =0;           // 합계를 저장하기 위한 변수.
    }
}
```

```

        for(int i=1; i <= 10; i++) {
            sum += i;          //    sum = sum + i;
            System.out.println( i + " 까지의 합: " + sum);
        }
    }
}

```

#### [실행결과]

```

1 까지의 합: 1
2 까지의 합: 3
3 까지의 합: 6
4 까지의 합: 10
5 까지의 합: 15
6 까지의 합: 21
7 까지의 합: 28
8 까지의 합: 36
9 까지의 합: 45
10 까지의 합: 55

```

1 부터 10까지의 합을 구하는 예제인데, 가장 일반적인 for문의 형태를 사용하고 있다. i가 1부터 10이 될 때까지(i가 10보다 같거나 작을 조건이 만족되는 동안), i를 1씩 증가시켜가면서 블록{}내의 문장들을 반복 수행한다.

sum +=i;는 sum = sum +i;를 줄여 쓴 것이며, sum에 저장되어 있는 값에 i의 값을 더해서 다시 sum에 저장한다.

이 수식은 i의 값의 변화에 따라 다음과 같이 계산된다.

i가 1일 때 :  $1 = 0 + 1$

i가 2일 때 :  $3 = 1 + 2$

i가 3일 때 :  $6 = 3 + 3$

...

i가 10일 때 :  $55 = 45 + 10$

#### [예제4-13] FlowEx13.java

```

class FlowEx13
{
    public static void main(String[] args)
    {
        int sum =0;           // 합계를 저장하기 위한 변수.

        for (int i=1; i <= 10; i++) {
            sum += i;         // sum = sum + i;
        }
        System.out.println(i + " 까지의 합: " + sum);    // error발생!!!
    }
}

```

예제4-12를 변경한 것으로 최종 결과만을 출력하도록 하기 위해서 System.out.println(i + " 까지의 합: " + sum);을 for문 밖으로 빼낸 것이다. 별문제 없을 것처럼 보이지만, 컴파일시에 라인10에서 에러가 발생한다.

그 이유는 변수 i를 for문의 초기화부분에서 선언을 했기 때문에 i가 유효한 영역은 오직 for문 내부 뿐인데, for문을 벗어난 곳에서 i를 사용했기 때문이다.

[참고]for문 외에 다른 반복문이나 조건문에 선언된 변수들도 각 구문의 내부에서만 유효하다.

원하는 결과를 얻기 위해서는 위의 예제를 다음과 같이 변경해야한다.

#### [예제4-14] FlowEx14.java

```

class FlowEx14
{
    public static void main(String[] args)
    {
        int sum =0;           // 합계를 저장하기 위한 변수.
        int i;                // 선언부분을 for문 밖으로 옮겼다.
        for(i=1; i <= 10; i++) {
            sum += i;         // sum = sum + i;
        }
    }
}

```



```

    }
    System.out.println(i-1 + " 까지의 합: " + sum);
}
}

```

#### [실행결과]

10 까지의 합: 55

변수 `i`를 for문 밖에서도 사용할 수 있도록, `i`의 선언부분을 for문 밖으로 빼내었다. `i`는 이제 main메서드 내에 선언된 변수이므로 for문 이후에도 유효하다.

for 문의 카운터로 사용되는 변수는 주로 for문의 블록 내에서만 사용되기 때문에, for문 내에 선언해서 for문의 실행이 끝나고 나면 없어지도록 하는 것이 다음에 또 다른 반복문에서 다시 재사용할 수도 있어서 코드를 보다 단순화 하는데 도움이 된다.

#### [예제4-15] FlowEx15.java

```

class FlowEx15
{
    public static void main(String[] args)
    {
        int sum =0;
        for(int i=0; i <=10; i+=2) {
            sum += i;        // sum = sum + i;
            System.out.println(i + " : " + sum);
        }
    }
}

```

#### [실행결과]

```

0 : 0
2 : 2
4 : 6
6 : 12
8 : 20
10 : 30

```

0 과 10사이에 있는 짝수들의 합을 구하는 프로그램이다. for문의 증감식 부분에 카운터 i의 값을 2씩 증가시키기 위해 i+=2를 사용했다. 이처럼 카운터의 값을 원하는 만큼 값을 증가시키거나 감소시킬 수 있다. 증감식 부분 역시 for문의 초기화 부분처럼 실행표(,)를 이용해서 여러 문장들을 넣을 수 있다.

#### [예제4-16] FlowEx16.java

```
class FlowEx16
{
    public static void main(String[] args)
    {
        for(int i=0, sum=0;i<10;System.out.println((i+=2)+ ":" + (sum+=i)));
    }
}
```

이전 예제의 for문을 한 줄로 작성해 보았다. 간략한 코드도 좋지만 이해하기 쉬운 코드가 더 좋은 코드이므로 별로 바람직하지는 않으니 참고만 하기 바란다.

if문 내에 또 다른 if문을 넣을 수 있는 것처럼, 반복문 안에 또 다른 반복문을 포함시키는 것 역시 가능하다.

#### [예제4-17] FlowEx17.java

```
class FlowEx17
{
    public static void main(String[] args)
    {
        for(int i=2; i <=9; i++) {
            for(int j=1; j <=9; j++) {
                System.out.println( i + " * " + j + " = " + i*j );
            }
        }
    }
}
```

```

    }
}

```

반복문에 반복문을 사용해서 구구단을 출력하는 예제이다. i는 단을 출력하는 데 사용되며, j는 1부터 9까지의 곱에 사용된다. 2단부터 출력하기 위해 첫 번째 for문의 i는 2로 초기화 되었다.

#### [예제4-18] FlowEx18.java

```

class FlowEx18
{
    public static void main(String[] args)
    {
        for(int i=2; i <=9; i++)
            for(int j=1; j <=9; j++)
                System.out.println( i + " * " + j + " = " + i*j );
    }
}

```

구 구단 예제를 보다 간단하게 바꾼 것이다. for문, while문, if문의 수행할 문장이 하나일 경우 중괄호{}를 생략할 수 있다. 두 번째 for문은 첫 번째 for문에 속한 한 문장으로, 출력문은 두 번째 for문에 속한 한 문장으로 간주된다.

되도록이면 중괄호{}를 사용하는 것이 좋지만 너무 많아도 복잡하므로 경우에 따라서는 이처럼 간략하게 생략하는 것도 좋다.

#### [예제4-19] FlowEx19.java

```

class FlowEx19
{
    public static void main(String[] args)
    {
        long startTime = System.currentTimeMillis();
        for(int i=0; i < 10000000000; i++) {
            ;
        }
    }
}

```

```

        long endTime = System.currentTimeMillis();

        System.out.println("시작시간 : " + startTime);
        System.out.println("종료시간 : " + endTime);
        System.out.println("소요시간 : " + (endTime - startTime));
    }
}

```

#### [실행결과]

```

시작시간 : 1042517111350
종료시간 : 1042517125520
소요시간 : 14170

```

currentTimeMillis() 를 이용해서 for문의 이전과 이후의 시간을 얻은 다음 그 차이를 계산함으로써 for문을 수행하는데 걸린 시간을 측정하는 예제이다. 결과는 컴퓨터의 성능에 따라 다르며, 속도가 빠른 컴퓨터일수록 Start와 End간의 시간차가 적을 것이다.

이 결과에서 알 수 있듯이 저자의 컴퓨터에서는 for문을 수행하는데 14.170초가 소요되었다. for문 내의 블록{}에 아무런 작업도 하지 않는 빈 문장(;)만 들어있기 때문에, 조건식 (i<1000000000)과 증감식(i++)만 10억 번을 반복해서 처리하는데 그 만큼의 시간이 걸린 것이다.

이처럼 단순히 반복만 하는 for문을 사용하면 작업들 사이에 시간간격을 줄 수 있다.

[참고]currentTimeMillis()는 1970년 1월 1일부터 현재까지의 시간을 천 분의 일초로 계산한 결과를 long형의 정수로 반환하는 메서드이다.

#### [예제4-20] FlowEx20.java

```

class FlowEx20
{
    public static void main(String[] args)
    {
        System.out.println("자, 이제 카운트 다운을 시작합니다.");
        for(int i=10; i >= 0; i--) {
            for(int j = 0; j < 1000000000; j++) {
                ;
            }
        }
    }
}

```

```

        System.out.println(i);
    }
    System.out.println("GAME OVER");
}
}

```

#### [실행결과]

자, 이제 카운트 다운을 시작합니다.

10

9

8

7

6

5

4

3

2

1

0

GAME OVER

10부터 1까지 1씩 감소시켜가면서 출력을 하되 매출력마다 약 1초의 시간이 지연된다.

위에 사용된 for문은 반복할 문장이 하나 밖에 없으므로 중괄호를 생략할 수 있으므로 다음과 같이 할 수 있다.

```
for (int i=0; i < 1000000000; i++) ;
```

예제에서는 for문의 중괄호{}내에 빈 문장(;)을 써주긴 했지만, 중괄호가 있을 때는 문장이 없어도 되므로 다음과 같이 쓸 수도 있다.

```
for (int i=0; i < 1000000000; i++) {}
```

## 2.2 while문

for문과는 달리, 조건식과 수행해야할 문장블럭{}만으로 구성되어 있지만 카운터로 사용할 변수와 증감식을 함께 사용함으로써 for문과 같이 구성할 수 있다. 그래서 for문과 while문은 항상 서로 대신 사용할 수 있다.

```
while (조건식) {  
    // 조건식의 연산결과가 true일 때 수행될 문장들을 적는다.  
}
```

### [예제4-21] FlowEx21.java

```
class FlowEx21  
{  
    public static void main(String[] args)  
    {  
        int i=10;  
        while(i >=0) {  
            System.out.println(i--);  
        }  
    }  
}
```

### [실행결과]

```
10
9
8
7
6
5
4
3
2
1
0
```

10부터 1씩 감소시켜가면서 출력하는 예제이다. 출력문의 `i--`는 후위형이기 때문에 `i`가 출력된 후에 감소한다. 이와 같은 구조의 `while`문에서 주의할 점은 카운터로 사용되는 변수의 선언과 초기화 위치이다.

만일 실수로 다음과 같이 `i`가 `while`문 내에 초기화된다면, 매 반복마다 `i`의 값이 10으로 초기화 되므로 `while`문은 무한 반복에 빠지게 된다.

```
int i = 0 ;
while (i >=0) {
    i=10 ;
    System.out.println(i--);
}
```

그래서, 반복문에 카운터가 필요한 경우는 `while`문보다는 `for`문을 사용하는 것이 좋다.

#### [예제4-22] FlowEx22.java

```
class FlowEx22
{
    public static void main(String[] args)
    {
```

```

int i=2;
while(i<=9) {
    int j=1;
    while(j <=9) {
        System.out.println( i + " * " + j + " = " + i*j );
        j++;
    }
    i++;
}
}

```

while문을 사용해서 구구단을 출력하는 예제를 작성해보았다. for문으로 작성된 예제와 한번 비교해 보도록 하자.

#### [예제4-23] FlowEx23.java

```

class FlowEx23
{
    public static void main(String[] args)
    {
        int sum =0;
        int i = 0;

        while(sum + i <= 100) {
            sum += ++i;    // sum = sum + ++i;
            System.out.println(i + " - " + sum);
        }
    }
}

```

#### [실행결과]

```

1 - 1
2 - 3
3 - 6

```



```
4 - 10
5 - 15
6 - 21
7 - 28
8 - 36
9 - 45
10 - 55
11 - 66
12 - 78
13 - 91
```

1부터 몇까지 더하면 누적합계가 100에 가까워지는지 계산하는 예제이다.

### 2.3 do-while문

while문의 변형으로 기본적인 구조는 while문과 같으나 블록{}이 먼저 수행한 후에 조건식을 판단한다는 것이 while과의 유일한 차이점이다.

while문은 조건식의 결과에 따라 블록{}이 한번도 수행되지 않을 수 있지만, do-while문은 최소한 한번은 수행될 것을 보장한다.

```
do {
    // 조건식의 연산결과가 true일 때 수행될 문장들을 적는다.
} while (조건식);
```

#### [예제4-24] FlowEx24.java

```
class FlowEx24
{
    public static void main(String[] args) throws java.io.IOException
```

```

{
    int input=0;

    System.out.println("문장을 입력하세요.");
    System.out.println("입력을 마치려면 x를 입력하세요.");
    do {
        input = System.in.read();
        System.out.print((char)input);
    } while(input!=-1 && input !='x');
}
}

```

#### [실행결과]

C:\Wj2sdk1.4.1\work>java FlowEx24

문장을 입력하세요.

입력을 마치고 싶으면 x를 입력하세요.

What do you want?

What do you want?

Tell me the truth.

Tell me the truth.

x

x

C:\Wj2sdk1.4.1\work>

System.in.read() 를 이용해서 화면을 통해 사용자로부터 입력을 받은 다음, 입력받은 내용을 다시 화면에 출력한다. 일단 사용자로부터 입력을 받은 후 입력받은 문자를 검사하므로 while 문 보다는do-while문이 적합하다. 사용자가 'x'를 입력하거나 Ctrl+z를 입력하면 do-while문을 빠져나가 프로그램이 종료된다.

System.in.read()는 사용자가 입력한 문자를 int형으로 반환하므로 문자가 아닌 문자의 코드 값이 반환된다. 그래서 화면으로부터 입력받은 문자를 다시 출력하기 위해서는 char형으로 변환해야 한다.

그리고 사용자가 입력한 문자가 Ctrl+z(키보드의 Ctrl키와 z키를 동시에 누름)이면 -1을 결과로 반환한다.

[참고]OS가 윈도우인 경우에는 Ctrl+z이지만 유닉스에서는 Ctrl+d이며, 이는 파일의 끝(EOF)을 의미한다.

## 2.4 break문

switch문에서 이미 배운 것과 같이 break문은 현재 위치에서 가장 가까운 switch문 또는 반복문을 벗어나는데 사용된다. 주로 if문과 함께 사용되어 특정 조건을 만족하면 반복문을 벗어나도록 한다.

**예제4-25** /ch4/FlowEx25.java

```
class FlowEx25
{
    public static void main(String[] args)
    {
        int sum =0;
        int i = 1;

        while(true) {
            if(sum > 100)
                break;
            sum += i;
            i++;

            System.out.println("i=" + i);
            System.out.println("sum=" + sum);
        }
    }
}
```

*www.javachobo.com*

break문이 실행되면 이부분은 실행되지 않고 while문을 완전히 벗어난다.

### [실행결과]

i=15

sum=105

숫자를 1부터 계속 더해 나가서 몇까지 더하면 합이 100을 넘는지 알아내는 예제이다. i의 값을 1부터 1씩 계속 증가시켜가며 더해서 sum에 저장한다. sum의 값이 100을 넘으면 if문의 조건이 true이므로 break문이 수행되어 자신이 속한 반복문을 즉시 벗어난다.

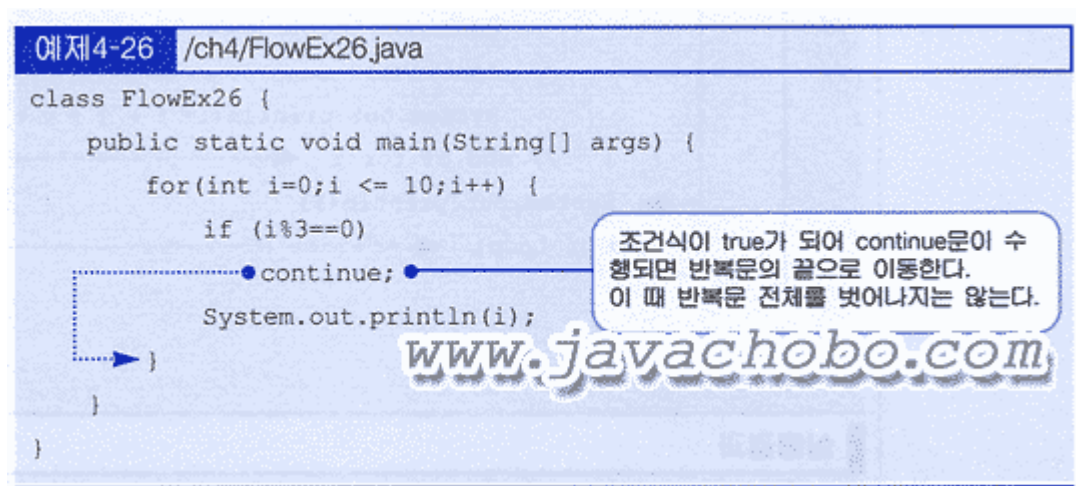
이처럼 무한 반복문에는 조건문과 break문이 항상 같이 사용된다. 그렇지 않으면 무한히 반복되기 때문에 프로그램이 종료되지 않을 것이다.

[참고] `sum += i;`와 `i++;` 두 문장을 `sum += i++;`와 같이 한 문장으로 줄여 쓸 수도 있다.

## 2.5 continue문

`continue` 문은 반복문 내만 사용될 수 있으며, 반복이 진행 중에 `continue` 문을 만나게 되면 반복문의 끝으로 이동하여 다음 반복으로 넘어간다. `for`문의 경우 증감식으로 이동하며, `while` 문과 `do-while`문의 경우 조건식으로 이동한다.

`continue` 문은 반복문 전체를 벗어나지 않고 다음 반복을 계속 수행한다는 점이 `break`문과 다르다. 주로 `if`문과 함께 사용되어 특정 조건을 만족하는 경우에 `continue`문 이후의 문장들을 수행하지 않고 다음 반복으로 넘어가서 계속 진행하도록 한다. 전체 반복 중에 특정조건을 만족하는 경우를 제외하고자 할 때 유용하다.



### [실행결과]

1  
2  
4  
5  
7  
8  
10

1 과 10사이의 숫자를 출력하되 그 중에서 3의 배수인 것은 제외하도록 하였다. 나머지 연산자를 사용해서 3으로 나눴을 때 나머지가 없으면 3의 배수이므로 이 경우 `continue`문이 실행되

어 그 이후의 문장이 실행되지 않고 다음 반복으로 넘어간다.

## 2.5 이름 붙은 반복문

여러 반복문이 중첩되어 있을 때 반복문 앞에 이름을 붙이고 break문과 continue문에 이름을 지정해 줌으로써 하나 이상의 반복문을 벗어나거나 반복을 건너뛸 수 있다.

**예제4-27** /ch4/FlowEx27.java

```
class FlowEx27
{
    public static void main(String[] args)
    {
        // for문에 Loop1이라는 이름을 붙였다.
        Loop1 : for(int i=2; i <=9; i++) {
            for(int j=1; j <=9; j++) {
                if(j==5)
                    break Loop1;
                // break;
                // continue Loop1;
                // continue;
                System.out.println(i+"*"+j+"="+i*j);
            } // end of for i
            System.out.println();
        } // end of Loop1
    }
}
```

[www.javachobo.com](http://www.javachobo.com)

### [실행결과]

```
2*1=2
2*2=4
2*3=6
2*4=8
```

구 구단을 출력하는 예제이다. 제일 바깥에 있는 for문에 Loop1이라는 이름을 붙였다. 그리고 j가 5일 때 break문을 수행하도록 했다. 반복문의 이름이 지정되지 않은 break문은 자신이 속한 하나의 반복문만 벗어날 수 있지만, 지금처럼 반복문에 이름을 붙여 주고 break문에 반복문

이름을 지정해주면 하나 이상의 반복문도 벗어날 수 있다.

j가 5일 때 반복문 Loop1을 벗어나도록 했으므로 2단의 4번째 줄까지 밖에 출력되지 않았다.

만일 반복문의 이름이 지정되지 않은 break문이었다면 2단부터 9단까지 모두 네 줄씩 출력되었을 것이다.

3개의 주석처리된 break문과 continue문을 바꿔 가면서 어떤 결과를 얻을지 예측해보고 실행 결과와 비교해 보도록 하자.

# [Java의 정석]제5장 배열

자바의정석

2012/12/22 17:06

<http://blog.naver.com/gphic/50157739793>

## 1. 배열(Array)

### 1.1 배열(Array)이란?

같은 타입의 여러 변수를 하나의 묶음으로 다루는 것을 "배열"이라고 한다. 많은 양의 데이터를 저장하기 위해서, 그 데이터의 숫자만큼 변수를 선언해야 한다면 매우 혼란스러울 것이다.

이런 경우에 배열을 사용하면 하나의 변수로 많은 양의 데이터를 손쉽게 다룰 수 있다.

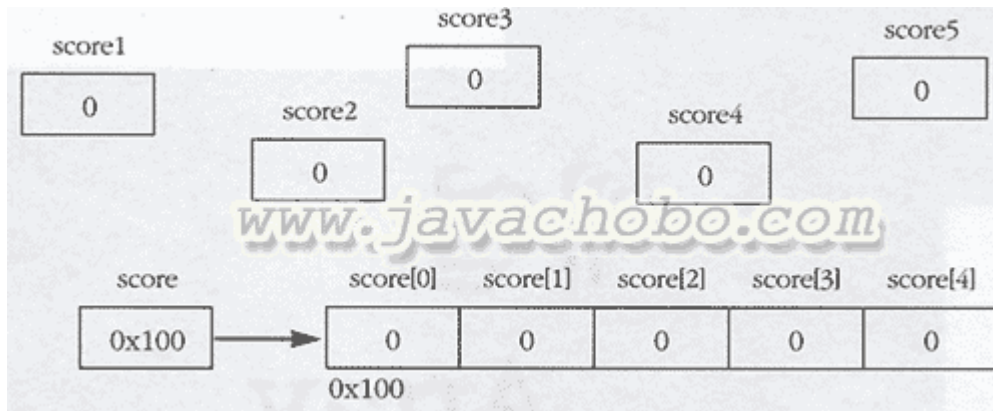
[참고]서로 다른 타입의 데이터를 하나로 묶어서 다루려면, 클래스를 정의해서 사용하면 된다.

한 학급의 시험점수를 저장하고자 할 때가 배열을 사용하기 좋은 예이다. 만일 배열을 사용하지 않는다면 5명의 학생의 점수를 저장하기 위해서 아래와 같이 해야 할 것이다.

```
int score1=0, score2=0, score3=0, score4=0, score5=0 ;
```

하지만, 배열을 사용하면 다음과 같이 간단히 할 수 있다.

```
int[] score = new int[5]; // 5개의 int 값을 저장할 수 있는 배열을 생성한다.
```



[그림5-1] 메모리에 생성된 변수들과 배열

## 1.2 배열의 선언

배열을 선언하는 방법은 간단하다. 원하는 타입의 변수를 선언하고 변수 또는 타입에 배열임을 의미하는 대괄호[]를 붙이면 된다.

선언 방법	선언예
타입[] 변수이름;	int[] score; String[] name;
타입 변수이름[];	int score[]; String name[];

## 1.3 배열의 생성

배열을 선언한 다음에는 배열을 생성해야 한다. 배열을 선언하는 것은 단지 생성된 배열을 다루기 위한 참조 변수를 위한 공간이 만들어지는 것 뿐이다. 배열을 생성해야만 비로소 데이터를 저장할 수 있는 공간이 만들어지는 것이다.

배열을 생성하기 위해서는 new 연산자를 사용하고 배열의 타입과 크기를 지정해 주어야 한다.



```
int[] score; // 배열을 선언한다.(생성된 배열을 다루는데 사용될 참조변수 선언)
score = new int[5]; // 배열을 생성한다.(5개의 int값을 저장할 수 있는 공간생성)
```

[참고] 위의 두 문장은 `int[] score = new int[5];`와 같이 한 문장으로 줄여 쓸 수 있다.

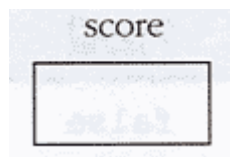
사실 배열도 객체이기 때문에 멤버변수와 메서드를 갖고 있으며, 이 중 멤버변수 `length`는 배열의 크기에 대한 정보를 담고 있다. 위의 예제코드에서 `score`의 크기가 5이므로 `score.length`의 값은 5가 된다.

[참고] 배열은 한번 생성되면 크기를 변경할 수 없다.

배열의 선언과 생성과정을 단계별로 그림과 함께 자세히 살펴보도록 하자.

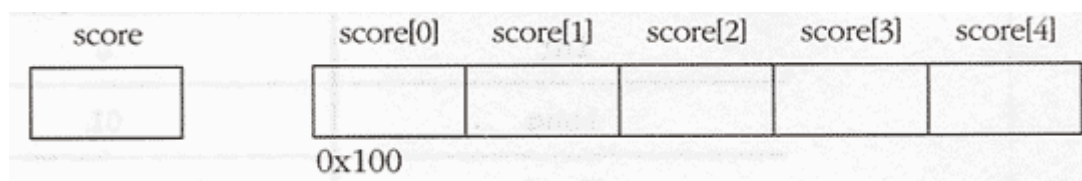
1. `int[] score;`

`int`형 배열 참조변수 `score`를 선언한다. 데이터를 저장할 수 있는 공간은 아직 마련되지 않았다.

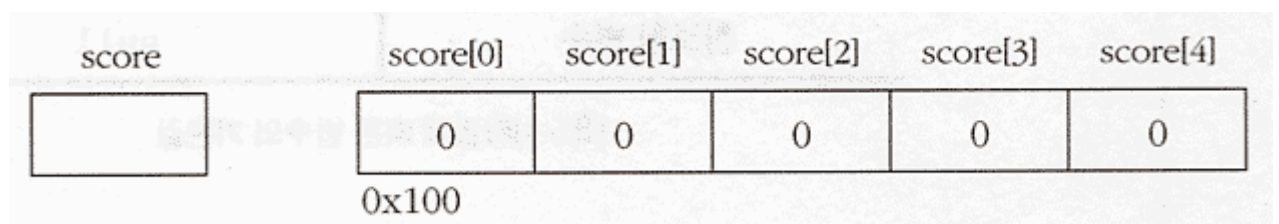


2. `score = new int[5];`

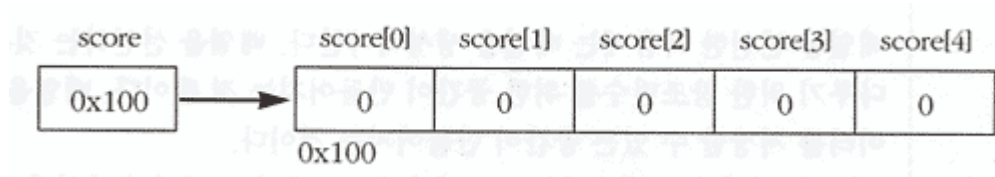
`new`연산자에 의해서 메모리의 빈공간에 5개의 `int`형 데이터를 저장할 수 있는 공간이 마련된다.



그리고 각 배열요소는 자동적으로 `int`의 기본값(default)인 0으로 초기화 된다.



마지막으로 할당연산자(=)에 의해서 배열의 주소가 int형 배열 참조변수 score에 저장된다.

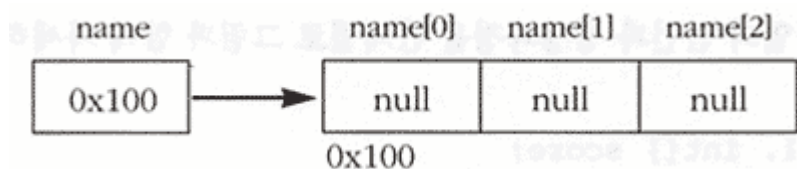


[참고] 배열이 주소 0x100번지에 생성되었다고 가정한 것이다.

이번엔 참조변수 배열의 예를 들어보자.

```
String[] name;      // String타입의 참조변수 배열을 선언한다.
name = new String[3]; // String인스턴스의 참조변수를 담을 수 있는 배열을 생성한다.
```

위의 두 문장을 수행한 결과를 그림으로 표현하면 다음과 같다. 3개의 String타입의 참조변수를 저장하기 위한 공간이 마련되고 참조변수의 기본값은 null이므로 각 배열요소의 값은 null로 초기화 된다.



참고로 변수의 타입에 따른 기본값은 다음과 같다.

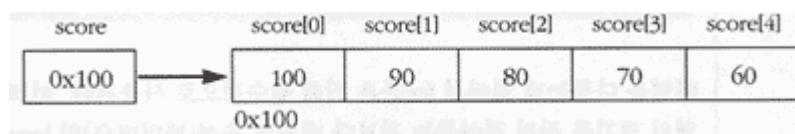
자료형	기본값
boolean	false
char	'\u0000'
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
참조형 변수	null

[표5-1] 타입에 따른 변수의 기본값

#### 1.4 배열의 초기화

배열은 생성과 동시에 자동적으로 자신의 타입에 해당하는 기본값으로 초기화되므로 배열을 사용하기 전에 초기화를 해주지 않아도 되지만, 원하는 값으로 초기화 하기 위해서 다음과 같이 한다.

```
int[] score = new int[5];    // 크기가 5인 int형 배열을 생성한다.
score[0] = 100;              // 각 요소에 직접 값을 저장한다.
score[1] = 90;
score[2] = 80;
score[3] = 70;
score[4] = 60;
```

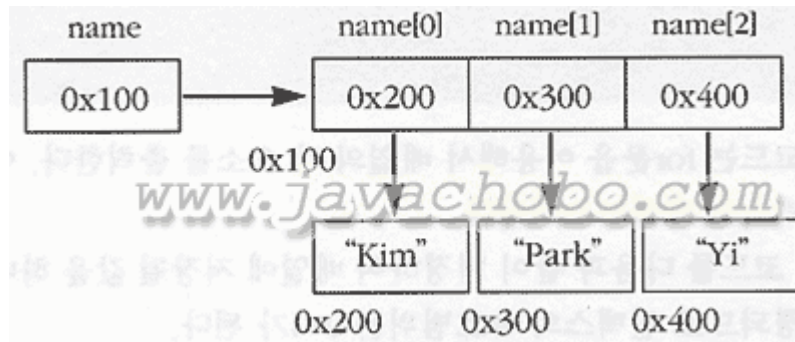


```
String[] name = new String[3];
```

```

name[0] = new String("Kim");
name[1] = new String("Park");
name[2] = new String("Yi");

```



이처럼 배열의 각 요소를 하나씩 초기화하는 것은 좀 불편하다. 그래서 자바에서는 보다 간편한 초기화 방법들을 제공한다. 이 때 배열의 크기는 따로 지정해주지 않으며 주어진 값의 개수에 따라 크기가 결정된다.

```

int[] score = { 100, 90, 80, 70, 60};
int[] score = new int[]{ 100, 90, 80, 70, 60};

String[] name = { new String("Kim"), new String("Park"), new String("Yi")};
String[] name = { "Kim", "Park", "Yi"};
String[] name = new String[]{ new String("Kim"), new String("Park"), new
String("Yi")};

```

[참고]String은 클래스이므로 new연산자를 통해 인스턴스를 생성해야하지만, "Kim"과 같이 쌍따옴표를 사용해서 간략히 표현하는 것을 특별히 허용한다.

## 1.4 배열의 활용

배열의 각 저장공간에 값을 저장하고 또는 저장된 값을 읽어오기 위해서는 배열 참조변수와 인덱스를 이용한

다. 배열의 인덱스는 배열의 각 저장공간에 자동적으로 주어지는 일련 번호인데, 0부터 시작해서 1씩 증가하는 연속적인 값이다. 크기가 5인 배열에서는 index의 범위가 0~4까지 모두 5개가 된다. 배열의 값을 읽거나 저장하기 위해서는 다음과 같이 배열 참조변수와 배열의 인덱스를 사용하면 된다.

```
score[3] = 100;    // 배열 score의 4번째 요소에 100을 저장한다.  
int value = score[3]; // 배열 score의 4번째 요소에 저장된 값을 읽어서 value에 저장한다.
```

배열을 다루는데 있어서 for문은 거의 필수적으로 사용된다. 이 때 for문의 조건식으로 배열의 크기를 직접 적어주는 것보다 배열의 속성인 length를 사용하는 것이 더 견고한 코드를 만든다.

```
int[] score = { 100, 90, 80, 70, 60, 50 };  
  
for (int i=0; i < 6; i++) {  
    System.out.println(score[i]);  
}
```

위의 코드는 배열의 각 요소를 for문을 이용해서 출력하는 일을 한다. 여기서 score배열의 크기는 6이며 인덱스의 범위는 0~5이다.

이 때 코드를 다음과 같이 변경하여 배열에 저장될 값을 하나 줄인다면, 배열의 크기가 5로 변경되었으므로 유효한 인덱스의 범위는 0~4가 된다.

```
int[] score = { 100, 90, 80, 70, 60 };  
  
for (int i=0; i < 6; i++) {  
    System.out.println(score[i]);  
}
```

배열의 크기가 변경되었으니 for문에 사용되는 조건의 범위도 변경해주어야 하는데, 만일 이 것을 잊고 실행한다면 for문은 배열의 유효한 인덱스 범위인 0~4를 넘어 0부터 5까지 반복하기 때문에 5번째 반복에서 `ArrayIndexOutOfBoundsException`이라는 예외(일종의 에러)가 발생하여 비정상적으로 종료될 것이다.

그래서 이러한 경우에는 for문의 조건식에 배열의 크기를 직접 적어주는 것보다 배열의 멤버변수인 `length`를 사용하는 것이 좋다. 위의 for문을 `length`를 사용해서 변경하면 다음과 같다.

```
for(int i=0; i < score.length; i++) {  
    System.out.println(score[i]);  
}
```

`length`는 배열의 크기가 변경됨에 따라 자동적으로 변경된 배열의 크기를 갖기 때문에, 배열의 처리에 사용되는 for문의 조건식을 일일이 변경해주지 않아도 된다.

이처럼 배열의 크기를 직접 적어주는 것보다 배열의 멤버변수인 `length`를 사용하는 것이 더 편리하고 안전하다.

#### [예제5-1] ArrayEx1.java

```
class ArrayEx1  
{  
    public static void main(String[] args)  
    {  
        int sum =0;           // 총점을 저장하기 위한 변수  
        float average = 0f;    // 평균을 저장하기 위한 변수  
  
        int[] score = {100, 88, 100, 100, 90};  
  
        for (int i=0; i < score.length ; i++ ) {  
            sum += score[i];    // 반복문을 이용해서 배열에 저장되어 있는 값들을 더한  
다.  
        }  
  
        average = sum / (float)score.length ; // 계산결과를 float로 얻기 위함.
```

```

        System.out.println("총점 : " + sum);
        System.out.println("평균 : " + average);
    }
}

```

#### [실행결과]

총점 : 478

평균 : 95.6

for문을 이용해서 배열에 저장된 값을 모두 더한 결과를 배열의 개수로 나누어서 평균을 구하는 예제이다. 평균을 구하기 위해 전체 합을 배열의 크기인 score.length로 나누었다.

이 때 int와 int간의 연산은 int를 결과로 얻기 때문에 정확한 평균값을 얻지 못하므로 score.length를 float로 변환하여 나눗셈을 하였다.

#### [예제5-2] ArrayEx2.java

```

class ArrayEx2
{
    public static void main(String[] args)
    {
        int[] number = new int[10];

        for (int i=0; i < number.length ; i++ ) {
            System.out.print(number[i] = i);    // 배열을 0부터 9의 숫자로 초기화한
다.
        }
        System.out.println();

        for (int i=0; i < 100; i++ ) {
            int n = (int)(Math.random() * 10);    // 0~9중의 한 값을 임의로 얻는다.
            int temp = number[0]; // 배열의 첫 번째 값과 임의로 선택된 위치의 값과 바
꾼다.
            number[0] = number[n];
            number[n] = temp;
        }
    }
}

```

```

    }
    for (int i=0; i < number.length ; i++ ) {
        System.out.print(number[i]);      // 배열의 내용을 출력한다.
    }
}
}

```

#### [실행결과]

0123456789

5827164930

크기가 10인 배열을 생성하고 0~9의 숫자로 차례대로 초기화하여 출력한다. 그 다음 random메서드를 이용하여 배열의 임의의 위치에 있는 값과 배열의 첫 번째 값을 교환하는 일을 100번 반복한다.

그리고 그 결과를 화면에 출력한다. 이 예제를 응용하면 카드게임에서 카드를 한 벌을 생성하여 초기화한 다음 카드를 섞는 것과 같은 일을 할 수 있을 것이다.

[참고]random메서드를 이용했기 때문에 실행 할 때 마다 결과가 다를 수 있다.

#### [예제5-3] ArrayEx3.java

```

class ArrayEx3
{
    public static void main(String[] args)
    {
        int[] number = new int[10];

        for (int i=0; i < number.length ; i++ ) {
            System.out.print(number[i] = (int)(Math.random() * 10));
        }
        System.out.println();

        for (int i=0; i < number.length ; i++ ) {
            boolean changed = false;    // 자리바꿈이 발생했는지를 체크한다.
            for (int j=0; j < number.length-1-i ; j++ ) {
                if(number[j] > number[j+1]) { // 옆의 값이 크면 서로 바꾼다.
                    int temp = number[j];

```



```

        number[j] = number[j+1];
        number[j+1] = temp;
        changed = true;    // 자리바꿈이 발생했으므로 changed를 true로.
    } // end if
} // end for j
for(int k=0; k < number.length; k++)
    System.out.print(number[k]);    // 매 반복마다 정렬된 결과를 출력한
다.

    System.out.println();
    if (!changed) break;    // 자리바꿈이 없으면 반복문을 벗어난다.
} // end for i
}
}

```

#### [실행결과]

```

1344213843
1342134438
1321344348
1213343448
1123334448
1123334448

```

크기가 10인 배열에 0과 9사이의 임의의 값으로 채운다음, 버블정렬 알고리즘을 통해서 크기순으로 정렬하는 예제이다. 이 알고리즘의 정렬방법은 아주 간단하다. 배열의 크기가 n일 때, 배열의 첫 번째부터 n-1까지의 요소에 대해, 근접한 값과 크기를 비교하여 자리바꿈을 반복하는 것이다.

보다 효율적인 작업을 위해 changed라는 boolean형 변수를 두어서 자리바꿈이 없으면 break문을 수행하여 정렬을 마치도록 했다. 자리바꿈이 없다는 것은 정렬이 완료되었음을 뜻하기 때문이다.

버블정렬은 다소 비효율적이긴 하지만 가장 간단한 정렬방법이다.

#### [예제5-4] ArrayEx4.java

```

class ArrayEx4
{
    public static void main(String[] args)
    {

```

```

int[] number = new int[10];
int[] counter = new int[10];

for (int i=0; i < number.length ; i++ ) {
    System.out.print(number[i] = (int)(Math.random() * 10));
}
System.out.println();

for (int i=0; i < number.length ; i++ ) {
    counter[number[i]]++;
}

for (int i=0; i < number.length ; i++ ) {
    System.out.println( i +"의 개수 :"+ counter[i]);
}

}
}

```

#### [실행결과]

4446579753

0의 개수 :0

1의 개수 :0

2의 개수 :0

3의 개수 :1

4의 개수 :3

5의 개수 :2

6의 개수 :1

7의 개수 :2

8의 개수 :0

9의 개수 :1

이전 예제에서와 같이 크기가 10인 배열을 만들고 0과 9사이의 임의의 값으로 초기화 한다. 그리고 이 배열에 저장된 각 숫자가 몇번 반복해서 나타나는지를 세어서 배열 counter에 담은 다음 화면에 출력한다.

[플래시동영상]자료실의 [Array.swf](#)을 보면 예제를 설명과 함께 순차적으로 실행과정을 볼수 있다.

#### [예제5-5] ArrayEx5.java

```
class ArrayEx5
{
    public static void main(String[] args)
    {
        char[] hex = { 'C', 'A', 'F', 'E' };

        String[] binary = {"0000", "0001", "0010", "0011"
                           , "0100", "0101", "0110", "0111"
                           , "1000", "1001", "1010", "1011"
                           , "1100", "1101", "1110", "1111" };

        String result="";

        for (int i=0; i < hex.length ; i++ ) {
            if(hex[i] >='0' && hex[i] <='9') {
                result +=binary[hex[i]-'0'];    // '8'-'0'의 결과는 8이다.
            } else {    // A~F이면
                result +=binary[hex[i]-'A'+10]; // 'C'-'A'의 결과는 2
            }
        }

        System.out.println("hex:"+ new String(hex));
        System.out.println("binary:"+result);
    }
}
```

#### [실행결과]

hex:CAFE

binary:1100101011111110

16 진수를 2진수로 변환하는 예제이다. 먼저 변환하고자 하는 16진수를 배열 hex에 나열한다. 16진수에는 A~F까지 6개의 문자가 포함되므로 char배열로 처리하였다. 그리고 String배열 binary는 이진수 0000 부터 1111(16진수로 0~F)까지 모두 16개의 값을 String으로 저장하였다.

for문을 이용해서 배열 hex에 저장된 문자를 하나씩 읽어서 그에 해당하는 이진수 표현을 배열 binary에서

얻어 result에 덧붙이고 그 결과를 화면에 출력한다.

[참고] 16진수 1자리는 2진수 4자리에 해당된다. 16진수 A는 십진수로 10이고 B는 11이다.

#### [예제5-6] ArrayEx6.java

```
class ArrayEx6
{
    public static void main(String[] args)
    {
        String source = "SOSHELP";
        String[] morse = {".-", "-...", "-.-.", "-..", ". "
            , ".--", "--.", "...", "..", "---"
            , "-.-", ".-.", "--", "-.", "---"
            , ".--", "--.-", ".-.", "...", "-"
            , "..-", "...-", ".--", "-..-"
            , "-.-", "--.."};

        String result="";

        for (int i=0; i < source.length() ; i++ ) {
            result+=morse[source.charAt(i)-'A'];
        }

        System.out.println("source:"+ source);
        System.out.println("morse:"+result);
    }
}
```

#### [실행결과]

```
source:SOSHELP
morse:...---.....-...--.
```

문자열(String)을 모르스(morse)부호로 변환하는 예제이다. 이전의 16진수를 2진수로 변환하는 예제와 같지만, char배열 대신 이번엔 String을 사용했다.

String의 문자의 개수는 length()를 통해서 얻을 수 있고, charAt(int i)메서드는 String의 i번째 문자를 반환한다. 그래서 for문의 조건식에 length()를 사용하고 charAt(int i)메서드를 통해서 source에서 한 문자씩 차례대로 읽어 올 수 있다.

[참고]String클래스는 char배열을 내부 데이터로 갖고 char배열을 다루는데 필요한 다양한 메서드를 제공한다.

## 1.4 다차원 배열

자바에서는 1차원 배열 뿐만 아니라 2차원 이상의 다차원 배열도 허용한다. 그러나 특별한 경우를 제외하고는 2차원 이상의 배열은 잘 사용되지 않는다.

그리고 2차원 배열을 잘 이해하면 2차원 이상의 배열에 응용하는 것은 그리 어렵지 않다. 본서에서는 2차원 배열에 대해서만 설명하도록 하겠다. 우선 2차원 배열의 선언방법은 다음과 같다.

선언 방법	선언예
타입[][] 변수이름;	int [][] score;
타입 변수이름[][];	int score [][];
타입[] 변수이름[];	int [] score [];

[표5-2]2차원 배열의 선언

[참고]3차원 이상의 고차원 배열의 선언은 대괄호[]의 개수를 차원 수 만큼 추가해 주기만 하면 된다.

2차원 배열은 주로 테이블 형태의 데이터를 담는데 사용되며, 만일 5행 3열의 데이터를 담기 위한 배열을 생성하려면 다음과 같이한다.

```
int [][] score = new int[5][3];    // 5행 3열의 2차원 배열을 생성한다.
```

위 문장이 수행되면 score[0][0]부터 score[4][2]까지 15개의 저장공간이 마련된다.

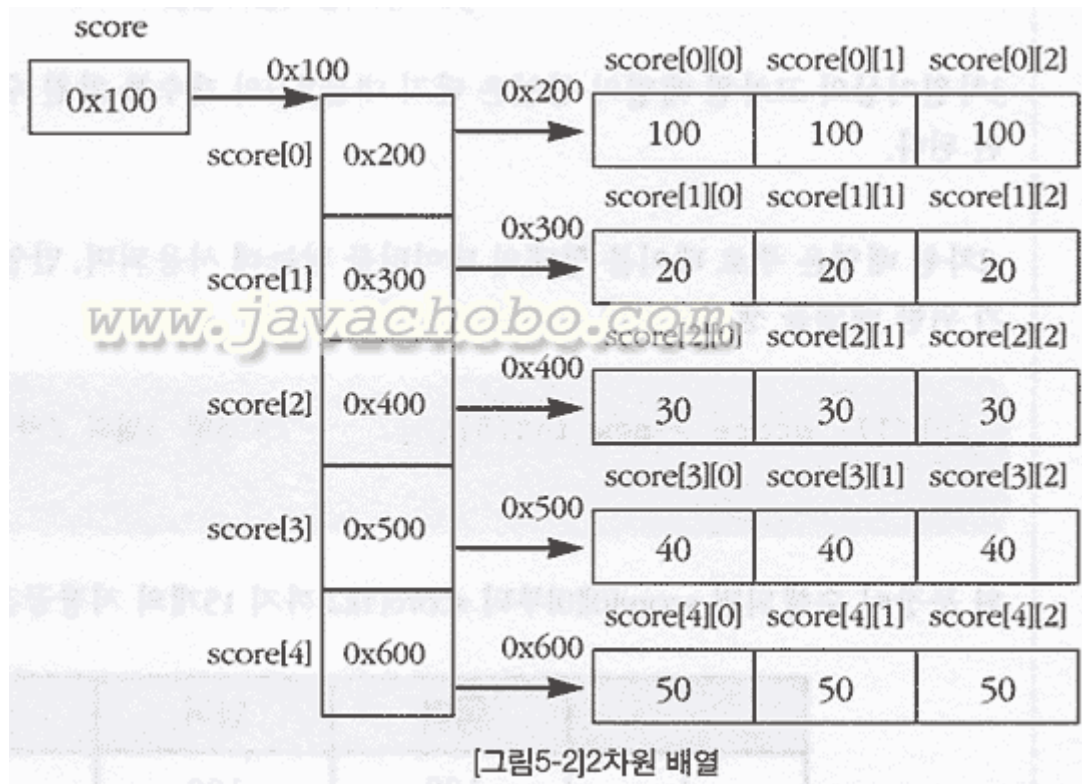
	국어	영어	수학
1	100	100	100
2	20	20	20
3	30	30	30
4	40	40	40
5	50	50	50

위와 같은 테이블형태의 데이터를 배열에 저장하기 위해서는 다음과 같이 한다.

```
score[0][0]=100 ;
score[0][1]=100 ;
score[0][2]=100 ;
score[1][0]=20 ;
score[1][1]=20 ;
...
score[4][2]=50 ;
```

1차원 배열에서와 같이 중괄호{}를 이용해서 2차원 배열의 생성과 초기화를 동시에 할 수도 있다.

```
int[][] score = {{100, 100, 100}, {20, 20, 20}, {30, 30, 30}, {40, 40, 40}, {50, 50, 50}};
```



5 행 3열의 2차원 배열을 생성하고 초기화한 결과를 그림으로 나타낸 것이다. 그림에서 알 수 있듯이 2차원 배열은 "배열의 배열"로 구성되어 있음을 알 수 있다. 즉, 여러 개의 배열이 모여서 또 하나의 배열을 이루고 있는 것이다.

여기서 `score.length`의 값은 얼마일까? 배열 참조변수 `score`가 참조하고 있는 배열의 크기가 얼마인가를 세어보면 될 것이다. 그래서 `score.length`의 값은 5이다. `score[0].length`은 배열 참조변수 `score[0]`이 참조하고 있는 배열의 크기이므로 3이고 `score[1].length`, `score[2].length`, `score[3].length`, `score[4].length`의 값 역시 모두 3이라는 것을 쉽게 알 수 있을 것이다.

만일 for문을 이용해서 2차원 배열을 초기화 한다면 다음과 같을 것이다.

```
for (int i=0; i < score.length; i++) {
    for (int j=0; j < score[i].length; j++) {
        score[i][j] = 10;
    }
}
```

위의 코드는 2차원 배열 score의 모든 요소를 10으로 초기화 한다.

[예제5-7] ArrayEx7.java

```
class ArrayEx7
{
    public static void main(String[] args)
    {
        int[][] score = {{ 100, 100, 100}
                        , { 20, 20, 20}
                        , { 30, 30, 30}
                        , { 40, 40, 40}
                        , { 50, 50, 50}};

        int koreanTotal = 0;
        int englishTotal = 0;
        int mathTotal = 0;

        System.out.println("번호 국어 영어 수학 총점 평균 ");
        System.out.println("=====");

        for(int i=0;i < score.length;i++) {
            int sum=0;
            koreanTotal += score[i][0];
            englishTotal += score[i][1];
            mathTotal += score[i][2];
            System.out.print(" " + (i + 1) + " ");
            for(int j=0;j < score[i].length;j++) {
                sum+=score[i][j];
                System.out.print(score[i][j]+" ");
            }
            System.out.println(sum + " " + sum/(float)score[i].length);
        }

        System.out.println("=====");
        System.out.println("총점:" + koreanTotal + " " +englishTotal + " " +mathTotal);
    }
}
```



```

    }
}

[실행결과]

번호 국어 영어 수학 총점 평균
=====
1 100 100 100 300 100.0
2 20 20 20 60 20.0
3 30 30 30 90 30.0
4 40 40 40 120 40.0
5 50 50 50 150 50.0
=====
총점:240 240 240

```

5명의 학생의 세 과목 점수를 더해서 각 학생의 총점과 평균을 계산하고 과목별 총점을 계산하는 예제이다.

## 1.7 가변 배열

자바에서는 2차원 이상의 배열에 대해서 "배열의 배열"의 형태로 처리한다는 사실을 이용하면 보다 자유로운 형태의 배열을 구성할 수 있다.

2차원 이상의 다차원 배열을 생성할 때 전체 배열 차수 중 마지막 차수의 크기를 지정하지 않고, 추후에 각기 다른 크기의 배열을 생성함으로써 고정된 형태가 아닌 보다 유동적인 가변 배열을 구성할 수 있다.

만일 다음과 같이 5 \* 3크기의 2차원 배열 score를 생성하는 코드가 있을 때,

```
int[][] score = new int[5][3];
```

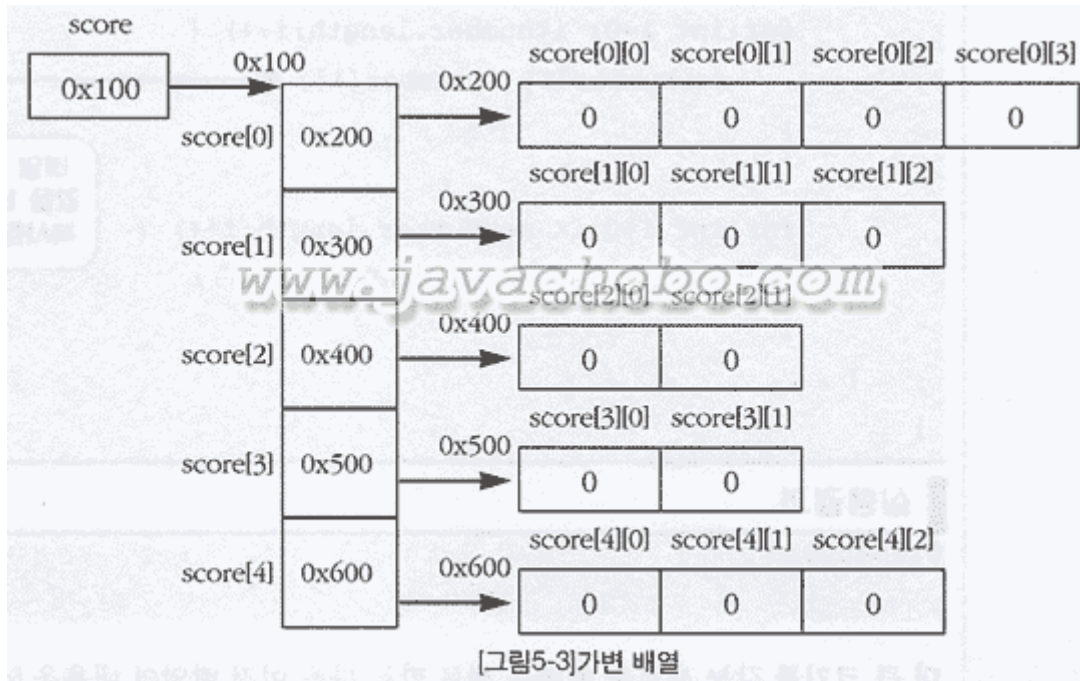
위 코드를 다음과 같이 표현 할 수 있다.

```
int[][] score = new int[5][];    // 두 번째 차원의 크기는 지정하지 않는다.  
score[0] = new int[3];  
score[1] = new int[3];  
score[2] = new int[3];  
score[3] = new int[3];  
score[4] = new int[3];
```

첫 번째 코드와 같이 2차원 배열을 생성하면 직사각형 테이블 형태의 고정적인 배열만 생성할 수 있지만, 두 번째 코드와 같이 2차원 배열을 생성하면 다음과 같이 각 열마다 다른 크기의 배열이 생성하는 것이 가능하다.

```
int[][] score = new int[5][];  
score[0] = new int[4];  
score[1] = new int[3];  
score[2] = new int[2];  
score[3] = new int[2];  
score[4] = new int[3];
```

위의 코드에 의해서 생성된 2차원 배열을 그림으로 표현하면 다음과 같다.



score.length의 값은 여전히 5지만, 전과는 달리 score[0].length의 값은 4이고 score[1].length의 값은 3으로 서로 다르다.

가변배열 역시 중괄호{}를 이용해서 다음과 같이 생성과 초기화를 동시에 하는 것이 가능하다.

```
int[][] score = {{100, 100, 100, 100}, {20, 20, 20}, {30, 30}, {40, 40}, {50, 50, 50}};
```

[플래시동영상]자료실의 [MultiDim.swf](#)을 보면 예제를 설명과 함께 순차적으로 실행과정을 볼수 있다.

## 1.8 배열의 복사

배열은 한번 생성하면 그 크기를 변경할 수 없기 때문에 더 많은 저장공간이 필요하다면 보다 큰 배열을 새로 만들고 이전 배열로부터 내용을 복사해야한다.

배열 간의 내용을 복사하려면 for문을 사용하거나 System클래스의 arraycopy메서드를 사용하면 된다. 예제를 통해서 배열을 복사하는 방법에 대해서 알아보도록 하자.

[예제5-8] ArrayEx8.java

```

class ArrayEx8
{
    public static void main(String[] args)
    {
        int[] number = {0,1,2,3,4,5};
        int[] newNumber = new int[10];

        for(int i=0; i < number.length;i++) {
            // 배열 number의 값을 newNumber에 저장한다.
            newNumber[i] = number[i];
        }

        for(int i=0;i < newNumber.length;i++) {
            System.out.print(newNumber[i]);
        }
    }
}

```

#### [실행결과]

0123450000

더 큰 크기의 새로운 배열을 새로 만든 다음 이전 배열의 내용을 for문을 사용해서 복사하는 예제이다. 배열은 생성과 동시에 자동적으로 자신의 타입의 기본값으로 초기화 되므로 배열 newNumber의 요소들이 int의 기본값인 0으로 초기화 되었다는 것을 알 수 있다.

System 클래스의 arraycopy메서드를 사용하면 보다 간단히 배열을 복사할 수 있다. arraycopy메서드는 배열에 저장되어 있는 값만을 복사하기 때문에 참조변수 배열인 경우에는 단지 주소값만을 복사할 뿐 참조변수가 가리키고 있는 객체를 복사하지는 않는다.

배열 abc와 number가 존재한다고 가정하고, abc의 내용을 number로 모두 복사하려면 다음과 같이 하면 된다.

```
System.arraycopy(abc, 0, number, 0, abc.length);
```

배열 abc의 내용을 배열 number로, 배열 abc에서 인덱스 0의 위치부터 시작해서 abc.length 만큼을 number의 인덱스 0인 위치에 복사한다.

이때 복사하려는 배열의 위치가 적절하지 못하여 복사하려는 내용보다 여유공간이 적으면 `ArrayIndexOutOfBoundsException`이 발생한다.

#### [예제5-9] ArrayEx9.java

```
class ArrayEx9
{
    public static void main(String[] args)
    {
        char[] abc = { 'A', 'B', 'C', 'D' };
        char[] number = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' };
        System.out.println(new String(abc));
        System.out.println(new String(number));

        // 배열 abc와 number를 붙여서 하나의 배열(result)로 만든다.
        char[] result = new char[abc.length+number.length];
        System.arraycopy(abc, 0, result, 0, abc.length);
        System.arraycopy(number, 0, result, abc.length, number.length);
        System.out.println(new String(result));

        // 배열 abc을 배열 number의 첫 번째 위치부터 배열 abc의 크기만큼 복사
        System.arraycopy(abc, 0, number, 0, abc.length);
        System.out.println(new String(number));

        System.arraycopy(abc, 0, number, 6, 3);    // number의 인덱스6 위치에 3개
를 복사
        System.out.println(new String(number));
    }
}
```

#### [실행결과]

ABCD

0123456789

```
ABCD0123456789
```

```
ABCD456789
```

```
ABCD45ABC9
```

## 1.9 커맨드라인을 통해 입력받기

System.in.read() 이외에 화면을 통해 사용자로 부터 값을 입력받을 수 있는 간단한 방법이 있다. 바로 커맨드라인을 이용한 방법인데, 프로그램을 실행할 때 클래스이름 뒤에 공백문자로 구분하여 여러 개의 문자열을 프로그램에 전달 할 수 있다.

만일 실행할 프로그램의 main메서드가 담긴 클래스의 이름이 MainTest라고 가정하면 다음과 같이 실행할 수 있을 것이다.

```
c:\wj2sdk1.4.1\work>java MainTest abc 123
```

커맨드라인을 통해 입력된 두 문자열은 String 배열에 담겨서 MainTest클래스의 main메서드의 매개변수 (args)에 전달된다. 그리고는 main메서드 내에서 args[0],args[1]과 같은 방식으로 커맨드라인으로 부터 전달받은 문자열에 접근할 수 있다. 여기서 args[0]은 "abc"이고 args[1]은 "123"이다.

### [예제5-10] MainTest.java

```
class MainTest
{
    public static void main(String[] args)
    {
        System.out.println("매개변수의 개수:"+args.length);
        for(int i=0;i< args.length;i++) {
            System.out.println("args[" + i + "] = " + args[i] + " ");
        }
    }
}
```

```

    }

}
}

```

#### [실행결과]

```
C:\₩j2sdk1.4.1\work>java MainTest abc 123 "John Lim"
```

매개변수의 개수:3

args[0] = "abc"

args[1] = "123"

args[2] = "John Lim"

커맨드라인에 입력된 매개변수는 공백문자로 구분하기 때문에 입력될 매개변수값에 공백이 있는 경우 겹따옴표(")로 감싸 주어야 한다. 그리고 커맨드라인에서 숫자를 입력해도 숫자가 아닌 문자열로 처리된다는 것에 주의해야한다.

#### [예제5-11] MorseConverter.java

```

class MorseConverter
{
    public static void main(String[] args)
    {
        if (args.length !=1) {
            System.out.println("usage: java MorseConverter WORD");
            System.exit(0);
        }

        System.out.println("source:"+ args[0]);
        String source = args[0].toUpperCase(); // 대문자로 변환한다.

        String[] morse = {".-", "-...", "-.-.", "-..", "."
            , "...", "--", "...", "..", "---"
            , "-.-", ".-..", "--", "-.", "---"}
    }
}

```

```

        , ".--.", "--.", ".-.", "...", "-"
        , ".-.", "...-", ".-.", ".-."
        , ".-.", "--.", "--." };

String result="";

for (int i=0; i < source.length() ; i++ ) {
    result+=morse[source.charAt(i)-'A'];
}

System.out.println("morse:"+result);
}
}

```

#### [실행결과]

```

C:\wj2sdk1.4.1\work>java MorseConverter
usage: java MorseConverter WORD

C:\wj2sdk1.4.1\work>java MorseConverter sos
source:sos
morse:...---...

```

예 제5-6을 변경하여 모리스부호로 변환할 문자열을 커맨드라인으로부터 입력받도록 변경하였다. 대문자만 변경 가능하도록 작성되어 있기 때문에 String클래스의 toUpperCase()를 이용해서 대문자로 변경한 다음에 모리스부호로 변환한다.

그리고 실행 시 매개변수를 입력하지 않거나 둘 이상 입력하게 되면 사용법을 출력한다.

[참고]커맨드라인에 매개변수를 입력하지 않으면 크기가 0인 배열이 생성되어 args.length의 값은 0이 된다. 이처럼 크기가 0인 배열을 생성하는 것도 가능하다.



# [Java의 정석]제6장 객체지향개념 1 - 1.객체지향언어, 2.클래스와 객체

자바의정석

2012/12/22 17:06

<http://blog.naver.com/gphic/50157739818>

## 1. 객체지향언어

### 1.1 객체지향언어의 역사

요즘은 컴퓨터의 눈부신 발전으로 활용 폭이 넓고 다양해져서 컴퓨터가 사용되지 않는 분야가 없을 정도지만, 초창기에는 주로 과학실험이나 미사일 발사실험과 같은 모의실험(Simulation)을 목적으로 사용되었다.

이 시절의 과학자들은 모의실험을 위해 실제 세계와 유사한 가상 세계를 컴퓨터 속에 구현하고자 노력하였으며, 이러한 노력은 객체지향이론을 탄생시켰다.

객체지향이론의 기본 개념은 '실제 세계는 사물(객체)로 이루어져 있으며, 발생하는 모든 사건들은 사물간의 상호작용이다.'라는 것이다.

실제 사물의 속성과 기능을 분석한 다음, 데이터(변수)와 함수로 정의함으로써 실제 세계를 컴퓨터 속에 옮겨 놓은 듯한 가상세계를 구현하고 이 가상세계에서 모의실험을 함으로써 많은 시간과 비용을 절약할 수 있었다.

객체지향이론은 상속, 캡슐화, 추상화 개념을 중심으로 점차 구체적으로 발전되었으며, 1960년대 중반에 객체지향이론을 프로그래밍언어에 적용한 Simula라는 최초의 객체지향언어가 탄생하였다.

그 당시 FORTRAN이나 COBOL과 같은 절차적 언어들이 주류를 이루었으며, 객체지향언어는 널리 사용되지 못하고 있었다.

1980년대 중반에 C++을 비롯하여 많은 수의 객체지향언어가 발표되면서, 객체지향언어가 본격적으로 개발자들의 관심을 끌기 시작하였지만 여전히 사용자 층이 넓지 못했다.

그러나 프로그램의 규모가 점점 커지고 사용자들의 요구가 빠르게 변화해가는 상황을 절차적 언어로는 극복하기 어렵다는 한계를 느끼고 객체지향언어를 이용한 개발방법론이 대안으로 떠오르게 되면서 조금씩 입지를 넓혀가고 있었다.

자바가 1995년에 발표되고 1990년대 말에 인터넷의 발전과 함께 크게 유행하면서 객체지향언어는 이제 프로그래밍의 주류로 자리잡았다.

## 1.2 객체지향언어

객체지향언어는 기존의 프로그래밍언어와 다른 전혀 새로운 것이 아니라, 기존의 프로그래밍 언어에 몇가지 새로운 규칙을 추가한 보다 발전된 형태의 것이다. 이러한 규칙들을 이용해서 코드간에 서로 관계를 맺어 줌으로써 보다 유기적으로 프로그램을 구성하는 것이 가능해졌다. 기존의 프로그래밍 언어에 익숙한 사람이라면 자바의 객체지향적인 부분만 새로 배우면 될 것이다. 다만 절차적언어에 익숙한 프로그래밍 습관을 객체지향적으로 바꾸도록 노력해야 할 것이다.

객체지향언어의 주요특징은 다음과 같다.

### 1. 코드의 재사용성이 높다.

- 새로운 코드를 작성할 때 기존의 코드를 이용하여 쉽게 작성할 수 있다.

### 2. 코드의 관리가 용이하다.

- 코드간의 관계를 이용해서 적은 노력으로 쉽게 코드를 변경할 수 있다.

### 3. 신뢰성이 높은 프로그래밍을 가능하게 한다.

- 제어자와 메서드를 이용해서 데이터를 보호하고 올바른 값을 유지하도록 하며, 코드의 중복을 제거하여 코드의 불일치로 인한 오동작을 방지할 수 있다.

객체지향언어의 가장 큰 장점은 '코드의 재사용성이 높고 유지보수가 용이하다.'는 것이다. 이러한 객체지향언어의 장점은 프로그램의 개발과 유지보수에 드는 시간과 비용을 획기적으로 개선하였다.

앞으로 상속, 다형성과 같은 객체지향개념을 학습할 때 재사용성과 유지보수 그리고 코드의 중복 제거, 이 세가지 관점에서 보면 보다 쉽게 이해할 수 있을 것이다.

객체지향 프로그래밍은 프로그래머에게 거시적 관점에서 설계할 수 있는 능력을 요구하기 때문에 객체지향개념을 이해했다 하더라도 자바의 객체지향적 장점들을 충분히 활용한 프로그램을 작성하기란 쉽지 않을 것이다.

너무 객체지향개념에 얽매어서 고민하기 보다는 일단 프로그램을 기능적으로 완성한 다음 어떻게 하면 보다 객체지향적으로 코드를 개선할 수 있는가에 대해서 고민하여 점차 개선해 나가

는 것이 좋다.

이러한 경험들이 축적되어야 프로그램을 객체지향적으로 설계할 수 있는 능력이 길러지는 것이  
이지 처음부터 이론을 많이 안다고 해서 좋은 설계를 할 수 있는 것은 아니다.

특히 프로그래밍에 익숙하지 않다면 객체지향개념 보다는 연산자와 조건문과 같은 프로그래밍  
언어의 기본적인 요소들에 먼저 익숙해질 수 있도록 간단한 예제들을 많이 작성해 보는 것이  
좋다.

## 2. 클래스와 객체

### 2.1 클래스와 객체의 정의와 용도

클래스란 '객체를 정의해놓은 것.' 또는 클래스는 '객체의 설계도 또는 틀'이라고 정의할 수 있  
다. 클래스는 객체를 생성하는데 사용되며, 객체는 클래스에 정의된 대로 생성된다.

클래스의 정의 - 클래스란 객체를 정의해 놓은것이다.

클래스의 용도 - 클래스는 객체를 생성하는데 사용된다.

객체의 사전적인 정의는, '실제로 존재하는 것'이다. 우리가 주변에서 볼 수 있는 책상, 의자,  
자동차와 같은 사물들이 곧 객체이다. 객체지향이론에서는 사물과 같은 유형적인 것뿐만 아니  
라, 개념이나 논리와 같은 무형적인 것들도 객체로 간주한다.

프로그래밍에서의 객체는 클래스에 정의된 내용대로 메모리에 생성된 것을 뜻한다.

객체의 정의 - 실제로 존재하는것. 사물 또는 개념

객체의 용도 - 객체가 가지고 있는 기능과 속성에 따라 다름

유형의 객체 - 책상, 의자, 자동차, TV와 같은 사물

클래스와 객체의 관계를 우리가 살고 있는 실생활에서 예를 들면, 제품 설계도와 제품과의 관계라고 할 수 있다. 예를 들면, TV설계도(클래스)는 TV라는 제품(객체)을 정의한 것이며, TV(객체)를 만드는데 사용된다.

또한 클래스는 단지 객체를 생성하는데 사용될 뿐이지 객체 그 자체는 아니다. 우리가 원하는 기능의 객체를 사용하기 위해서는 먼저 클래스로부터 객체를 생성하는 과정이 선행되어야 한다.

우리가 TV를 보기 위해서는, TV(객체)가 필요한 것이지 TV설계도(클래스)가 필요한 것은 아니며, TV설계도(클래스)는 단지 TV라는 제품(객체)을 만드는데만 사용될 뿐이다. 그리고 TV설계도를 통해 TV가 만들어진 후에야 사용할 수 있는 것이다.

프로그래밍에서는 먼저 클래스를 작성한 다음, 클래스로부터 객체를 생성하여 사용한다.

[참고]객체를 사용한다는 것은 객체가 가지고 있는 속성과 기능을 사용한다는 뜻이다.

클래스	객체
제품 설계도 TV 설계도	제품 TV
붕어빵 기계	붕어빵

[표6-1]클래스와 객체의 예

클래스를 정의하고 클래스를 통해 객체를 생성하는 이유는 설계도를 통해서 제품을 만드는 이유와 같다. 하나의 설계도만 잘 만들어 놓으면 제품을 만드는 일이 쉬워진다. 제품을 만들 때마다 매번 고민할 필요없이 설계도 대로만 만들면 되기 때문이다.

설계도 없이 제품을 만든다고 생각해보라. 복잡한 제품일 수록 설계도 없이 제품을 만든다는 것은 상상할 수도 없을 것이다.

이와 마찬가지로 클래스를 한번만 잘 만들어 놓기만 하면, 매번 객체를 생성할 때마다 어떻게 객체를 만들어야 할지를 고민하지 않아도 된다. 그냥 클래스로부터 객체를 생성해서 사용하기만 하면 되는 것이다.

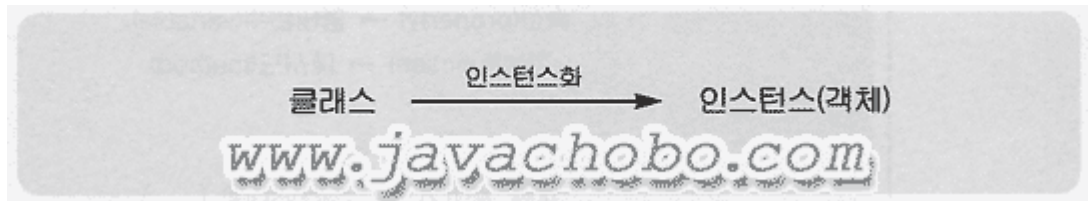
J2SDK(Java2 Standard Development Kit)에서는 프로그래밍을 위해 많은 수의 유용한 클래스(Java API)를 기본적으로 제공하고 있으며, 우리는 이 클래스들을 이용해서 원하는 기능의 프로그램을 보다 쉽게 작성할 수 있다.

## 2.2 객체와 인스턴스

클래스로부터 객체를 만드는 과정을 클래스의 인스턴스화(instantiate)라고 하며, 어떤 클래스로부터 만들어진 객체를 그 클래스의 인스턴스(instance)라고 한다.

예를 들면, Tv클래스로부터 만들어진 객체를 Tv클래스의 인스턴스라고 한다. 결국 인스턴스는 객체와 같은 의미이지만, 객체는 모든 인스턴스를 대표하는 포괄적인 의미를 갖고 있으며, 인스턴스는 어떤 클래스로부터 만들어진 것인지를 강조하는 보다 구체적인 의미를 갖고 있다. 예를 들면, '책상은 인스턴스다.'라고 하기 보다는 '책상은 객체다.'라는 쪽이, '책상은 책상 클래스의 객체이다.'라고 하기 보다는 '책상은 책상 클래스의 인스턴스다.'라고 하는 것이 더 자연스럽다.

인스턴스와 객체는 같은 의미이므로 두 용어의 사용을 엄격히 구분지을 필요는 없지만, 위의 예에서 본 것과 같이 문맥에 따라 구별하여 사용하는 것이 좋다.



## 2.3 객체의 구성요소 - 속성과 기능

객체는 속성과 기능, 두 종류의 구성요소로 이루어져 있으며, 일반적으로 객체는 다수의 속성과 다수의 기능을 갖는다. 즉, 객체는 속성과 기능의 집합이라고 할 수 있다. 그리고, 객체가 가지고 있는 속성과 기능을 그 객체의 멤버(구성원, member)라 한다.

x 알아두면 좋아요. [www.javachobo.com](http://www.javachobo.com)

속성과 기능은 아래와 같이 같은 뜻의 여러 가지 용어가 있으며, 앞으로 이 중에서도 '속성' 보다는 '멤버변수'를, '기능' 보다는 '메서드'를 주로 사용할 것이다.

속성(property) - 멤버변수(member variable), 특성(attribute), 필드(field), 상태(state)  
 기능(function) - 메서드(method), 행위(behavior), 함수(function)

클래스란 객체를 정의한 것이므로 클래스에는 객체의 모든 속성과 기능이 정의되어있다. 클래스로부터 객체를 생성하면, 클래스에 정의된 속성과 기능을 가진 객체가 만들어지는 것이다.

보다 쉽게 이해할 수 있도록 TV를 예로 들어보자. TV의 속성으로는 전원상태, 크기, 길이, 높이, 색상, 볼륨, 채널과 같은 것들이 있으며, 기능으로는 켜기, 끄기, 볼륨 높이기, 채널 변경하기 등이 있다.



속 성	크기, 길이, 높이, 색상, 볼륨, 채널 등
기 능	켜기, 끄기, 볼륨 높이기, 볼륨 낮추기, 채널 변경하기 등

[그림6-1]TV의 속성과 기능

객체지향 프로그래밍에서는 속성과 기능을 각각 변수와 함수로 표현한다.

속성(property) → 멤버변수(variable)

기능(function) → 메서드(method)

채널 → int channel

채널 높이기 → channelUp() { ... }

위에서 분석한 내용을 토대로 Tv클래스를 만들어 보면 다음과 같다.

```
class Tv {  
    // Tv의 속성(멤버변수)  
    String color;        // 색상  
    boolean power;      // 전원상태(on/off)  
    int channel;         // 채널  
  
    // Tv의 기능(메서드)  
    void power() {    power = !power; }    /* TV를 켜거나 끄는 기능을 하는 메서드 */  
    void channelUp() {    ++channel; }    /* TV의 채널을 높이는 기능을 하는 메서드 */  
    void channelDown() {    --channel; }    /* TV의 채널을 낮추는 기능을 하는 메서드 */  
}
```

[참고] 멤버변수와 메서드를 선언하는데 있어서 순서는 관계없지만, 일반적으로 메서드보다는 멤버변수를 먼저 선언하고 멤버변수는 멤버변수끼리 메서드는 메서드끼리 모아 놓는 것이 일반적이다.

실제 TV가 갖는 기능과 속성은 이 외에도 더 있지만, 프로그래밍에 필요한 속성과 기능만을 선택하여 클래스를 작성하면 된다.

각 변수의 자료형은, 속성의 값에 알맞은 것을 선택해야한다. 전원상태(power)의 경우, on과 off 두 가지 값을 가질 수 있으므로 boolean형으로 선언했다.

#### [알아두면 좋아요]

power() 의 power = !power;문은 power의 값이 true면 false로, false면 true로 변경하는 일을 한다. power의 값에 관계없이 항상 반대의 값으로 변경해주면 되므로 굳이 if문을 사용할 필요가 없다. 참고로 if문을 사용하여 코드를 작성하면 다음과 같다.

```
if (power) {  
    power = false;
```

```

    } else {
        power = true;
    }
}

```

## 2.4 인스턴스의 생성과 사용

Tv클래스를 완성했으니, 이제는 Tv클래스의 인스턴스를 만들어서 사용해 보도록 하자. 클래스로부터 인스턴스를 생성하는 방법은 여러 가지가 있지만 일반적으로는 다음과 같이 한다.

```

클래스명 변수명;    // 클래스의 객체를 참조하기 위한 참조변수를 선언한다.
변수명 = new 클래스명(); // 클래스의 객체를 생성 후, 객체의 주소를 참조변수에 저장한다.

Tv t;                // Tv클래스 타입의 참조변수 t를 선언
t = new Tv();        // Tv인스턴스를 생성한 후, 생성된 Tv인스턴스의 주소를 t에 저장한다.

```

### [예제6-1] TvTest.java

```

class Tv {
    // Tv의 속성(멤버변수)
    String color;        // 색상
    boolean power;       // 전원상태(on/off)
    int channel;         // 채널

    // Tv의 기능(메서드)
    void power() {    power = !power; }    /* TV를 켜거나 끄는 기능을 하는 메서드 */
    void channelUp() { ++channel; }        /* TV의 채널을 높이는 기능을 하는 메서드 */
    void channelDown() { --channel; }      /* TV의 채널을 낮추는 기능을 하는 메서드

```



```

*/
}

class TvTest {
    public static void main(String args[]) {
        Tv t;           // Tv인스턴스를 참조하기 위한 변수 t를 선언
        t = new Tv();    // Tv인스턴스를 생성한다.
        t.channel = 7;   // Tv인스턴스의 멤버변수 channel의 값을 7로 한다.
        t.channelDown(); // Tv인스턴스의 메서드 channelDown()을 호출한다.
        System.out.println("현재 채널은 " + t.channel + " 입니다.");
    }
}

```

#### [실행결과]

현재 채널은 6 입니다.

위의 예제는 Tv클래스로부터 인스턴스를 생성하고, 인스턴스의 속성(channel)과 함수(channelDown())를 사용하는 방법을 보여 주는 것이다. 예제를 그림과 함께 한 줄 씩 살펴보도록 하자.

#### 1. Tv t;

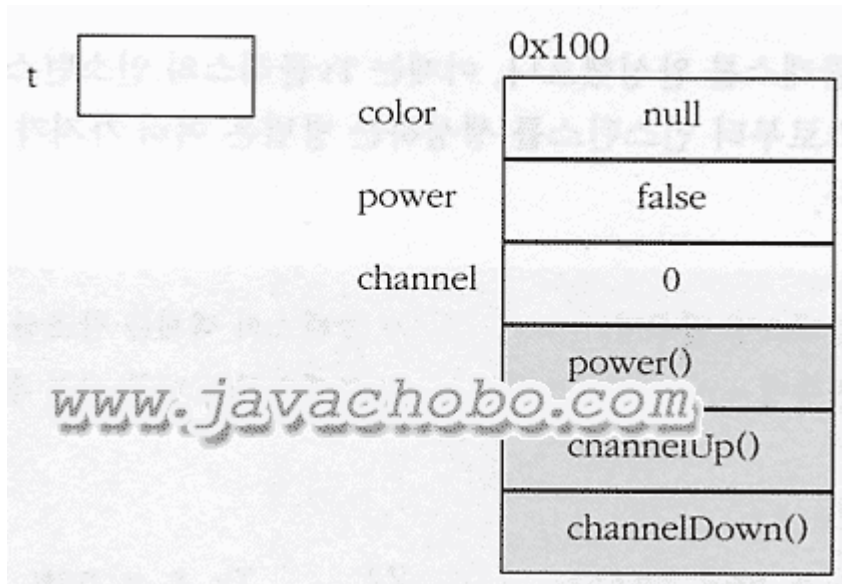
Tv클래스 타입의 참조변수 t를 선언한다. 메모리에 참조변수 t를 위한 공간이 마련된다.



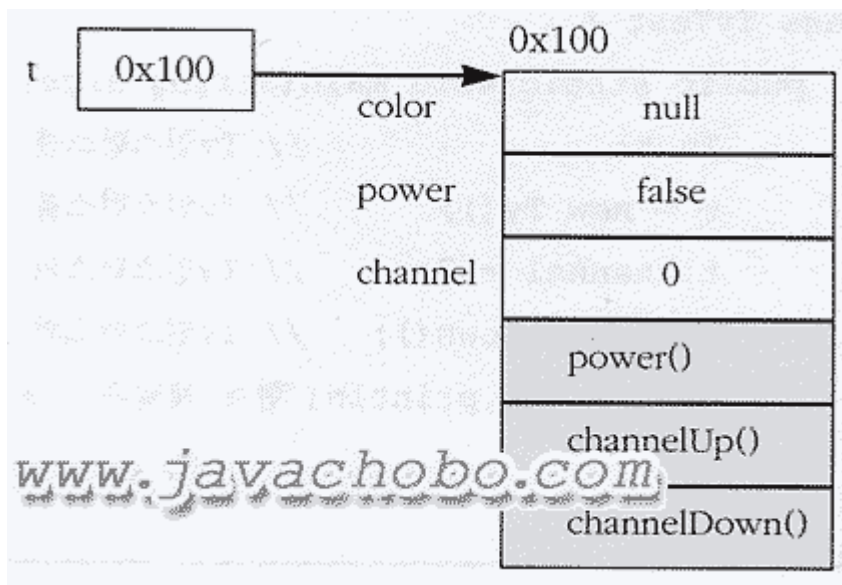
#### 2. t = new Tv();

연산자 new에 의해 Tv클래스의 인스턴스가 메모리의 빈 공간에 생성된다. 주소가 0x100인 곳에 생성되었다고 가정하자. 이 때, 멤버변수는 각 자료형에 해당하는 기본값으로 초기화 된다.

color는 참조형이므로 null로, power는 boolean이므로 false로, 그리고 channel은 int이므로 0으로 초기화 된다.



그 다음에는 대입연산자(=)에 의해서 생성된 객체의 주소값이 참조변수 `t`에 저장된다. 이제는 참조변수 `t`를 통해 Tv인스턴스에 접근할 수 있다. 인스턴스를 다루기 위해서는 참조변수가 반드시 필요하다.

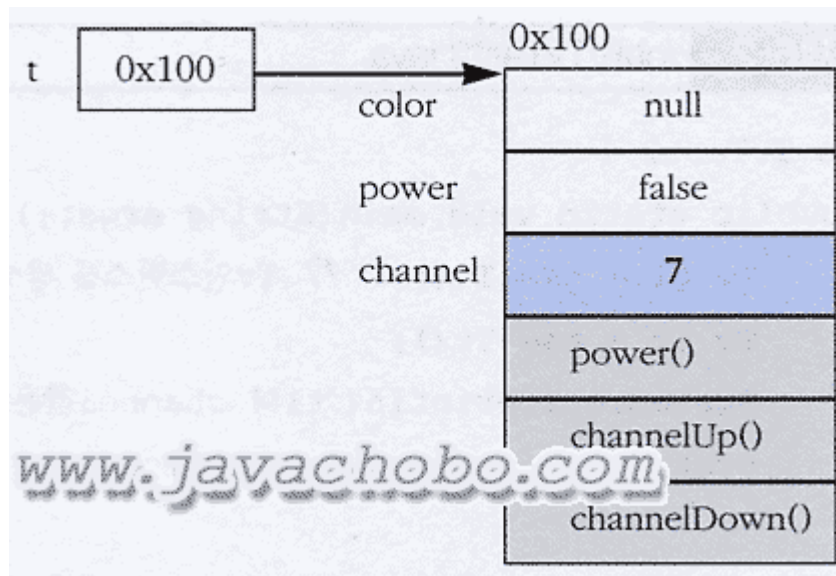


[참고] 위 그림에서의 화살표는 참조변수 `t`가 Tv인스턴스를 참조하고 있다는 것을 알기 쉽게 하기 위해 추가한 상징적인 것이다. 이 때, 참조변수 `t`가 Tv인스턴스를 '가리키고 있다' 또는 '참조하고 있다'라고 한다.

3. `t.channel = 7 ;`

참조변수 `t`에 저장된 주소에 있는 인스턴스의 멤버변수 `channel`에 7을 저장한다. 여기서 알

수 있는 것처럼, 인스턴스의 멤버변수(속성)를 사용하려면 '참조변수.멤버변수'와 같이 하면 된다.

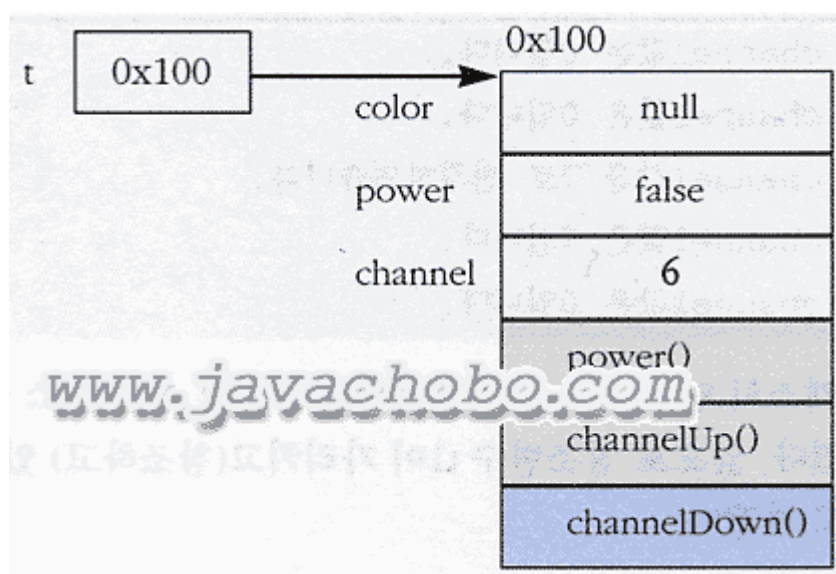


#### 4. `t.channelDown();`

참조변수 `t`가 참조하고 있는 `Tv`인스턴스의 `channelDown`메서드를 호출한다. `channelDown` 메서드는 멤버변수 `channel`에 저장되어 있는 값을 1감소시킨다.

```
void channelDown() {    --channel;    }
```

`channelDown()`에 의해서 `channel`의 값은 7에서 6이 된다.



#### 5. `System.out.println("현재 채널은 " + t.channel + " 입니다.");`

참조변수 t가 참조하고 있는 Tv인스턴스의 멤버변수 channel에 저장되어 있는 값을 출력한다. 현재 channel의 값은 6이므로 '현재 채널은 6 입니다.'가 화면에 출력된다.

인스턴스와 참조변수의 관계는 마치 우리가 일상생활에서 사용하는 TV와 TV리모콘의 관계와 같다. TV리모콘(참조변수)을 사용하여 TV(인스턴스)를 다루기 때문이다. 다른 점이라면, 인스턴스는 오직 참조변수를 통해서만 다룰 수 있다는 것이다.

그리고, TV를 사용하려면 TV 리모콘을 사용해야하고, 에어컨을 사용하려면, 에어컨 리모콘을 사용해야하는 것처럼, Tv인스턴스를 사용하려면, Tv클래스 타입의 참조변수가 필요한 것이다.

#### [예제6-2] TvTest2.java

```
class TvTest2 {
    public static void main(String args[]) {
        Tv t1 = new Tv();
        Tv t2 = new Tv();
        System.out.println("t1의 channel값은 " + t1.channel + "입니다.");
        System.out.println("t2의 channel값은 " + t2.channel + "입니다.");

        t1.channel = 7;    // channel 값을 7으로 한다.
        System.out.println("t1의 channel값을 7로 변경하였습니다.");

        System.out.println("t1의 channel값은 " + t1.channel + "입니다.");
        System.out.println("t2의 channel값은 " + t2.channel + "입니다.");
    }
}
```

#### [실행결과]

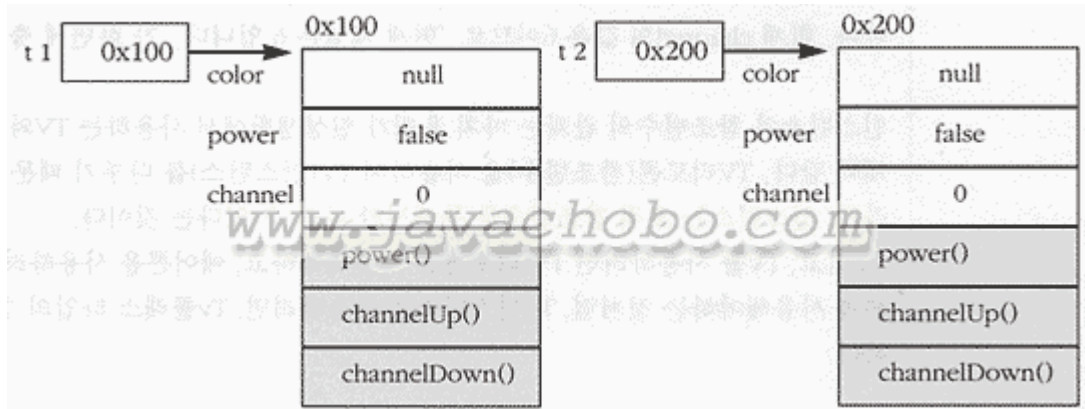
```
t1의 channel값은 0입니다.
t2의 channel값은 0입니다.
t1의 channel값을 7로 변경하였습니다.
t1의 channel값은 7입니다.
t2의 channel값은 0입니다.
```

위의 예제는 Tv클래스의 인스턴스 t1과 t2를 생성한 후에, 인스턴스 t1의 멤버변수인 channel의 값을 변경하였다.

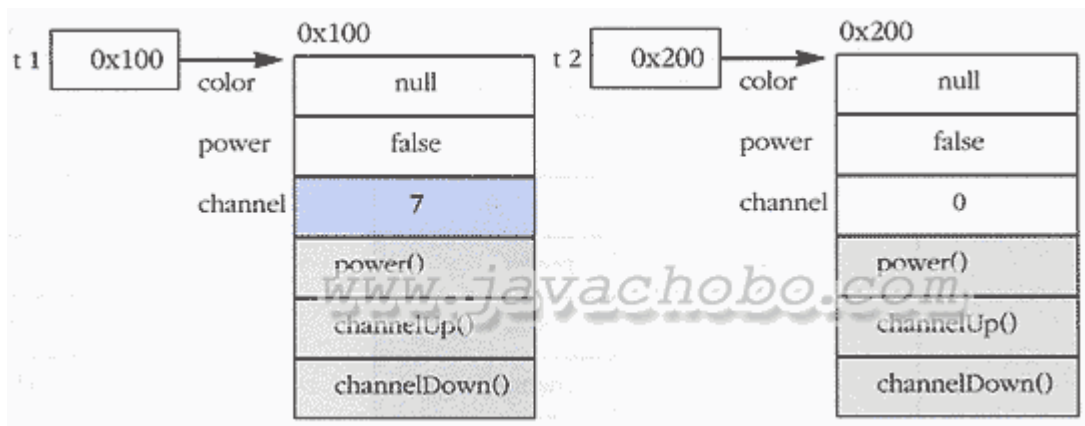
[참고] 참조변수 t1이 가리키고(참조하고) 있는 인스턴스를 간단히 인스턴스 t1이라고 했다.

1. Tv t1 = new Tv();

Tv t2 = new Tv();



2. t1.channel = 7; // 참조변수 t1이 가리키고 있는 인스턴스의 멤버변수 channel의 값을 7로 한다.



같은 클래스로부터 생성되었을지라도 각 인스턴스들의 속성(멤버변수)은 서로 다른 값을 유지할 수 있으며, 메서드의 내용은 모든 인스턴스에 대해 동일하다.

#### [예제6-3] TvTest3.java

```
class TvTest3 {  
    public static void main(String args[]) {  
        Tv t1 = new Tv();  
    }  
}
```

```

Tv t2 = new Tv();

System.out.println("t1의 channel값은 " + t1.channel + "입니다.");
System.out.println("t2의 channel값은 " + t2.channel + "입니다.");

t2 = t1;      // t1이 저장하고 있는 값(주소)을 t2에 저장한다.
t1.channel = 7;  // channel 값을 7로 한다.
System.out.println("t1의 channel값을 7로 변경하였습니다.");

System.out.println("t1의 channel값은 " + t1.channel + "입니다.");
System.out.println("t2의 channel값은 " + t2.channel + "입니다.");
}
}

```

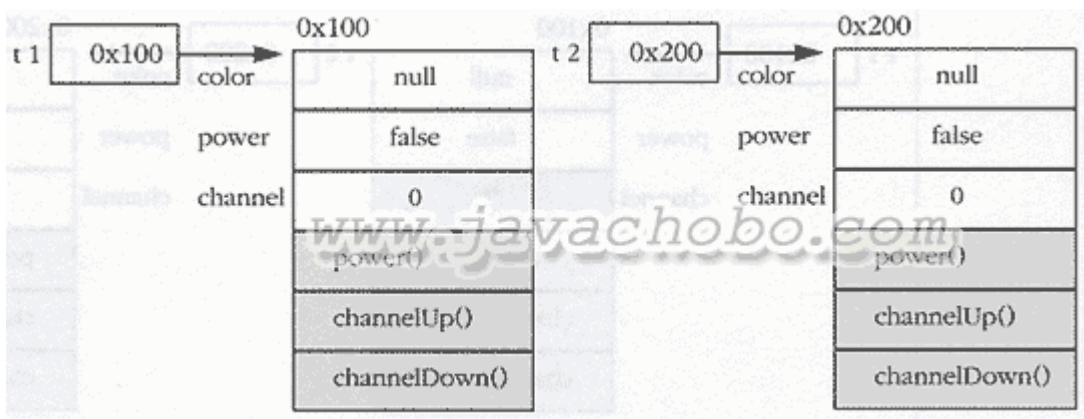
#### [실행결과]

t1의 channel값은 0입니다.  
t2의 channel값은 0입니다.  
t1의 channel값을 7로 변경하였습니다.  
t1의 channel값은 7입니다.  
t2의 channel값은 7입니다.

```

1. Tv t1 = new Tv();
   Tv t2 = new Tv();

```



```

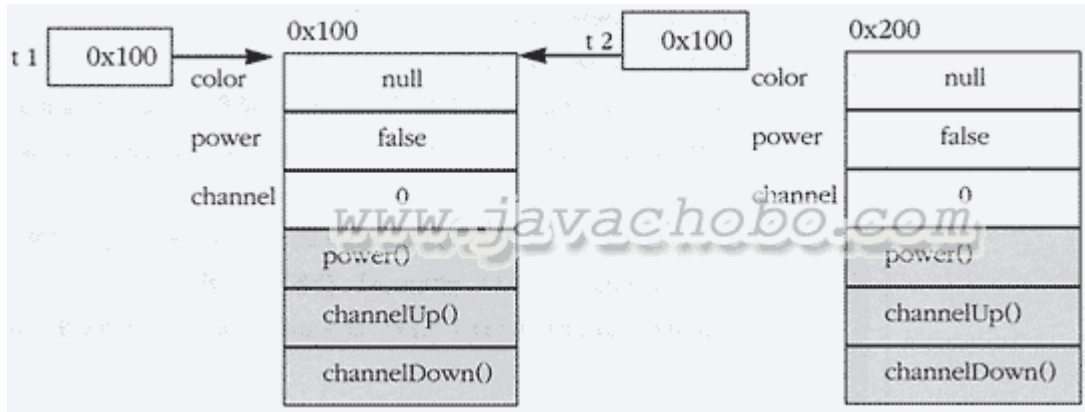
2. t2 = t1;      // t1이 저장하고 있는 값(주소)을 t2에 저장한다.

```

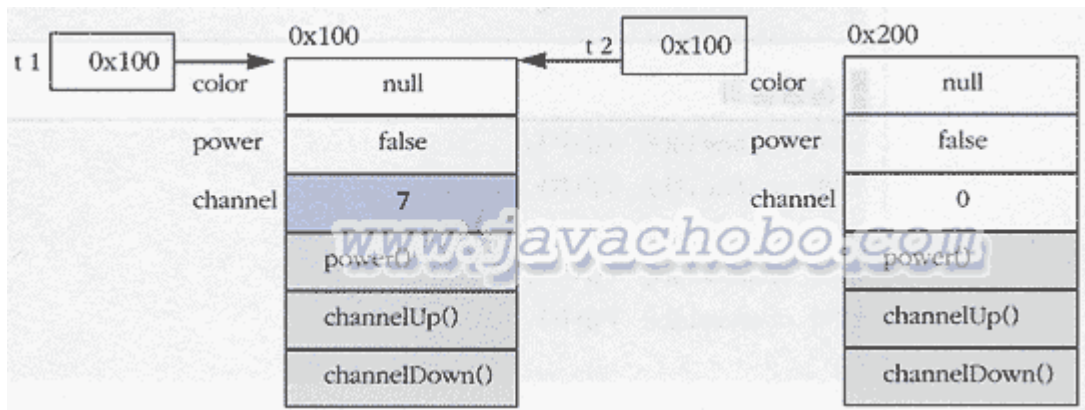
t1은 참조변수이므로, 인스턴스의 주소를 저장하고 있다. 이 문장이 수행되면, t2가 가지고 있

던 값은 잃어버리게 되고, t1에 저장되어 있던 값이 t2에 저장되게 된다. 그렇게 되면, t2 역시 t1이 참조하고 있던 인스턴스를 같이 참조하게 되고, t2가 원래 참조하고 있던 인스턴스는 더 이상 사용할 수 없게 된다.

[참고]자신을 참조하고 있는 참조변수가 하나도 없는 인스턴스는 더 이상 사용되어질 수 없으므로 가비지 컬렉터(Garbage Collector)에 의해서 자동적으로 메모리에서 제거된다.



3. t1.channel = 7; // channel 값을 7로 한다.



4. System.out.println("t1의 channel값은 " + t1.channel + "입니다.");

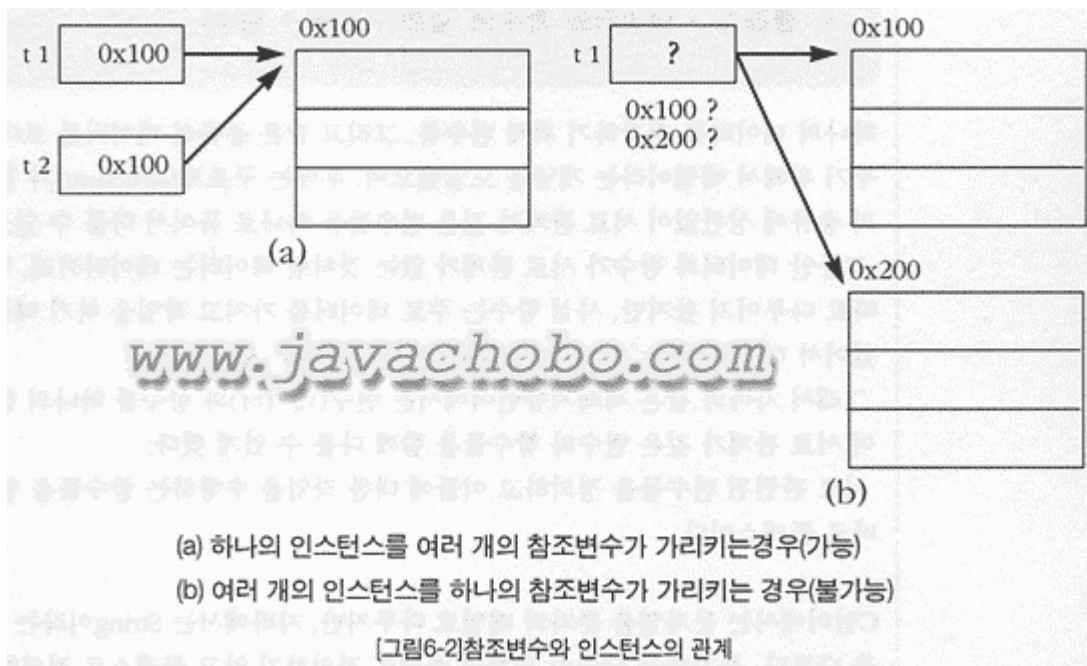
System.out.println("t2의 channel값은 " + t2.channel + "입니다.");

이제 t1, t2 모두 같은 Tv클래스의 인스턴스를 가리키고 있기 때문에, t1.channel의 값과 t2.channel의 값은 7이며 다음과 같은 결과가 화면에 출력된다.

t1의 channel값은 7입니다.

t2의 channel값은 7입니다.

이 예제에서 알 수 있듯이, 참조변수에는 하나의 값(주소)만이 저장될 수 있으므로 둘 이상의 참조변수가 하나의 인스턴스를 가리키는(참조하는) 것은 가능하지만 하나의 참조변수로 여러 개의 인스턴스를 가리키는 것은 가능하지 않다.



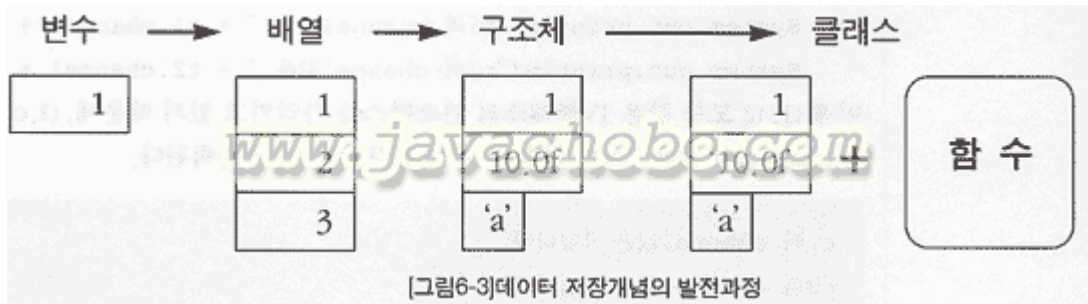
## 2.5 클래스의 또 다른 정의

클래스는 '객체를 생성하기 위한 틀'이며 '클래스는 속성과 기능으로 정의되어있다.'고 했다. 이것은 객체지향이론의 관점에서 내린 정의이고, 이번엔 프로그래밍적인 관점에서 클래스의 정의와 의미를 살펴보도록 하자.

### 1. 클래스 - 데이터와 함수의 결합

프로그래밍언어에서 데이터 처리를 위한 데이터 저장형태의 발전과정은 다음과 같다.





1. 변수 - 하나의 데이터를 저장할 수 있는 공간
2. 배열 - 같은 종류의 여러 데이터를 하나의 집합으로 저장할 수 있는 공간
3. 구조체 - 서로 관련된 여러 데이터를 종류에 관계없이 하나의 집합으로 저장할 수 있는 공간
4. 클래스 - 데이터와 함수의 결합(구조체 + 함수)

하나의 데이터를 저장하기 위해 변수를, 그리고 같은 종류의 데이터를 보다 효율적으로 다루기 위해서 배열이라는 개념을 도입했으며, 후에는 구조체(structure)가 등장하여 자료형의 종류에 상관없이 서로 관계가 깊은 변수들을 하나로 묶어서 다룰 수 있도록 했다.

그동안 데이터와 함수가 서로 관계가 없는 것처럼 데이터는 데이터끼리, 함수는 함수끼리 따로 다루어져 왔지만, 사실 함수는 주로 데이터를 가지고 작업을 하기 때문에 많은 경우에 있어서 데이터와 함수는 관계가 깊다.

그래서 자바와 같은 객체지향언어에서는 변수(데이터)와 함수를 하나의 클래스에 정의하여 서로 관계가 깊은 변수와 함수들을 함께 다룰 수 있게 했다.

서로 관련된 변수들을 정의하고 이들에 대한 작업을 수행하는 함수들을 함께 정의한 것이 바로 클래스이다.

C 언어에서는 문자열을 문자의 배열로 다루지만, 자바에서는 String이라는 클래스로 문자열을 다룬다. 문자열을 단순히 문자의 배열로 정의하지 않고 클래스로 정의한 이유는 문자열과 문자열을 다루는데 필요한 함수들을 함께 묶기 위해서이다.

```
public final class String implements java.io.Serializable, Comparable {
    private char[] value;    // 문자열을 저장하기 위한 공간
```

```

public String replace(char oldChar, char newChar) {
    ...
    char[] val = value;    // 같은 클래스 내의 변수를 사용해서 작업을 한다.
    ...
}
    ...
}

```

이 코드는 String클래스의 실제 소스의 일부이다. 클래스 내부에 value라는 문자형 배열이 선언되어 있고, 문자열을 다루는 데 필요한 함수들을 함께 정의해 놓았다. 문자열의 일부를 뽑아 낸다던가 문자열의 길이를 알아내는 함수들은 항상 문자열을 작업대상으로 필요로 하기 때문에 문자열과 깊은 관계에 있으므로 함께 정의하였다.

이렇게 함으로써 변수와 함수가 서로 유기적으로 연결되어 작업이 간단하고 명료해 진다.

## 2. 클래스 - 사용자정의 타입(User-defined Type)

프로그래밍언어에서 제공하는 자료형(Primitive type)외에 프로그래머가 서로 관련된 변수들을 묶어서 하나의 타입으로 새로 추가하는 것을 사용자정의 타입(User-defined type)이라고 한다.

많은 프로그래밍언어에서 사용자정의 타입을 정의할 수 있는 방법을 제공하고 있으며, 자바와 같은 객체지향언어에서는 클래스가 곧 사용자 정의 타입이다. 기본형의 개수는 8개로 정해져 있지만 참조형의 개수가 정해져 있지 않은 이유는 이처럼 프로그래머가 새로운 타입을 추가할 수 있기 때문이다.

시간을 표현하기 위해서 다음과 같이 3개의 변수를 선언할 수 있다.

```

int hour;        // 시간을 표현하기 위한 변수
int minute;      // 분을 표현하기 위한 변수
float second;    // 초를 표현하기 위한 변수, 백분의 일초까지 표현하기 위해 float로 했다.

```

만일 3개의 시간을 다뤄야 한다면 다음과 같이 해야 할 것이다.

```
int hour1, hour2, hour3;
int minute1, minute2, minute3;
float second1, second2, second3;
```

이처럼, 다뤄야 하는 시간의 개수가 늘어날 때마다 시, 분, 초를 위한 변수를 추가해줘야 하는데 다뤄야 하는 데이터의 개수가 많으면 이런 식으로는 곤란하다.

```
int[] hour = new int[3];
int[] minute; = new int[3];
float[] second; = new float[3];
```

배 열로 처리하면 다뤄야 하는 시간 데이터의 개수가 늘어나더라도 배열의 크기만 변경해주면 되므로, 변수를 매번 새로 선언해줘야 하는 불편함과 복잡함은 없어졌다. 그러나 하나의 시간을 구성하는 시, 분, 초가 서로 분리되어 있기 때문에 프로그램 수행과정에서 시, 분, 초가 따로 뒤섞여서 올바르지 않은 데이터가 될 가능성이 있다. 이런 경우 시, 분, 초를 하나로 묶는 사용자정의 타입, 즉 클래스를 정의하여 사용해야한다.

```
class Time {
    int hour;
    int minute;
    float second;
}
```

시, 분, 초를 저장하기 위한 세 변수를 멤버변수로 갖는 Time클래스를 정의하였다. 새로 작성된 사용자정의 타입인 Time클래스를 사용해서 코드를 변경하면 다음과 같다.

비객체지향적 코드	객체지향적 코드
<pre>int hour1, hour2, hour3; int minutel, minute2, minute3; float second1, second2, second3;</pre>	<pre>Time t1 = new Time(); Time t2 = new Time(); Time t3 = new Time();</pre>
<pre>int[] hour = new int[3]; int[] minute = new int[3]; float[] second = new float[3];</pre>	<pre>Time[] t= new Time[3]; t[0] = new Time(); t[1] = new Time(); t[2] = new Time();</pre>

[표6-2]비객체지향적 코드와 객체지향적 코드의 비교

이제 시, 분, 초가 하나의 단위로 묶어서 다루어지기 때문에 다른 시간 데이터와 섞이는 일은 없겠지만, 시간 데이터에는 다음과 같은 추가적인 제약조건이 있다.

1. 시, 분, 초는 모두 0보다 커야한다.
2. 시의 범위는 0~23, 분과 초의 범위는 0~59이다.

이러한 조건들이 모두 프로그램 코드에 반영될 때, 보다 정확한 데이터를 유지할 수 있을 것이다. 객체지향언어가 아닌 언어에서는 이러한 추가적인 조건들을 반영하기가 어렵다. 그러나 객체지향언어에서는 제어자와 메서드를 이용해서 이러한 조건들을 코드에 쉽게 반영할 수 있다. 아직 제어자에 대해서 배우지는 않았지만 위의 조건들을 반영하여 Time클래스를 작성해 보았다. 가볍게 참고만 하기 바란다.

```
public class Time {
    private int hour;
    private int minute;
    private int second;

    public int getHour() {
        return hour;
    }
}
```

```

    public void setHour(int h) {
        if (h < 0 || h > 23)
            return;
        hour=h;
    }

    public int getMinute() {
        return minute;
    }

    public void setMinute(int m) {
        if (m < 0 || m > 59)
            return;
        minute=m;
    }

    public int getSecond() {
        return second;
    }

    public void setSecond(int s) {
        if (s < 0 || s > 59)
            return;
        second=s;
    }

}

```

제어자를 이용해서 변수의 값을 직접 변경하지 못하도록 하고 대신 메서드를 통해서 값변경하도록 작성하였다. 값을 변경할 때 지정된 값의 유효성을 조건문으로 점검한 다음에 유효한 값일 경우에만 변경한다.

이 외에도 시간의 차를 구하는 함수와 같이 시간과 관련된 함수를 추가로 정의하여 Time클래스를 향상시켜 보는 것도 좋은 프로그래밍 공부거리가 될 것이다.

# [Java의 정석]제6장 객체지향개념 1 - 3. 변수와 메서드

자바의정석

2012/12/22 17:07

<http://blog.naver.com/gphic/50157739859>

## 3. 변수와 메서드

### 3.1 선언위치에 따른 변수의 종류

변수는 클래스변수, 인스턴스변수, 지역변수 모두 세 종류가 있다. 변수의 종류를 결정짓는 중요한 요소는 '변수의 선언된 위치'이므로 변수의 종류를 파악하기 위해서는 변수가 어느 영역에 선언되었는지를 확인하는 것이 중요하다. 멤버변수를 제외한 나머지 변수들은 모두 지역변수이며, 멤버변수 중 static이 붙은 것은 클래스변수, 붙지 않은 것은 인스턴스변수이다.

아래의 그림에는 모두 3개의 int형 변수가 선언되어 있는데, iv와 cv는 클래스 영역에 선언되어 있으므로 멤버변수이다. 그 중 cv는 키워드 static과 함께 선언되어 있으므로 클래스 변수이며, iv는 인스턴스변수이다. 그리고, lv는 메서드인 method() 내부에 선언되어 있으므로 지역 변수이다.

<pre> class Variables {     int iv;           // 인스턴스 변수     static int cv;    // 클래스변수      void method() {         int lv=0;     // 지역변수     } } </pre>		
		클래스 영역
		메서드 영역
www.javachobo.com		
변수의 종류	선언위치	생성시기
클래스변수 (class variable)	클래스 영역	클래스가 메모리에 올라갈 때
인스턴스 변수 (instance variable)		인스턴스가 생성되었을 때
지역변수 (local variable)	클래스 영역 이외의 영역 (메서드, 생성자, 초기화 블록 내)	변수 선언문이 수행되었을 때

[표6-3]변수의 종류와 특징

### 1. 인스턴스변수(instance variable)

클래스 영역에 선언되며, 클래스의 인스턴스를 생성할 때 만들어진다. 그렇기 때문에 인스턴스 변수의 값을 읽어 오거나 저장하기 위해서는 먼저 인스턴스를 생성해야한다.

인스턴스는 독립적인 저장공간을 가지므로 서로 다른 값을 가질 수 있다. 인스턴스마다 고유한 상태를 유지해야하는 속성의 경우, 인스턴스변수로 선언한다.

### 2. 클래스변수(class variable)

클래스 변수를 선언하는 방법은 인스턴스변수 앞에 static을 덧붙이기만 하면 된다. 인스턴스마다 독립적인 저장공간을 갖는 인스턴스변수와는 달리, 클래스변수는 모든 인스턴스가 공통된 저장공간(변수)을 공유하게 된다. 그래서 클래스 변수를 공유 변수(shared variable)라고도 한다.

한 클래스의 모든 인스턴스들이 공통적인 값을 유지해야하는 속성의 경우, 클래스변수로 선언해야 한다.

인스턴스변수는 인스턴스를 생성한 후에야 사용가능하지만, 클래스 변수는 인스턴스를 생성하지 않고도 언제라도 바로 사용할 수 있다는 특징이 있으며, '클래스이름.클래스변수'와 같은 형

식으로 사용한다.

[참고]위의 예제에서 Variables클래스의 클래스변수 cv를 사용하려면 Variables.cv와 같이 하면 된다.

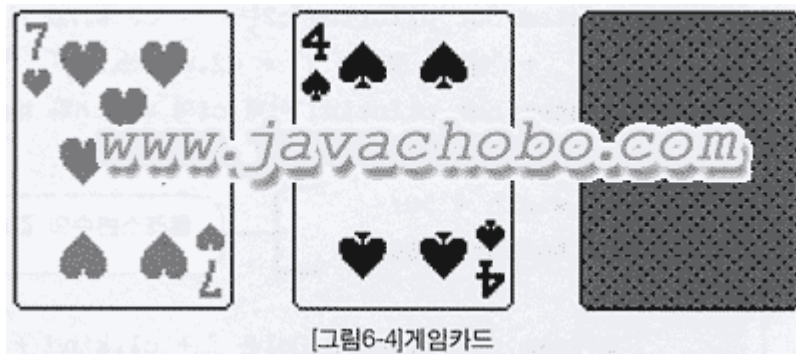
### 3. 지역변수(local variable)

메서드 내에 선언되어 메서드 내에서만 사용 가능하며, 메서드가 종료되면 소멸되어 사용할 수 없게 된다. for문 또는 while문의 블록 내에 선언된 지역변수는, 지역변수가 선언된 블록 내에서만 사용 가능하며, 블록을 벗어나면 소멸되어 사용할 수 없게 된다.

[참고]여기서의 메서드는 생성자와 초기화 블록을 포함한 개념이다. 앞으로 배우게 될 생성자와 초기화 블록은 내부적으로 메서드로 취급된다.

## 3.2 클래스변수와 인스턴스변수

클래스변수와 인스턴스변수의 차이를 이해하기 위한 예로 카드 게임에 사용되는 카드를 클래스로 정의해보자.



카드 클래스를 작성하기 위해서는 먼저 카드를 분석해서 속성과 기능을 알아 내야한다. 속성으로는 카드의 무늬, 숫자, 폭, 높이 정도를 생각할 수 있을 것이다.

이 중에서 어떤 속성을 클래스 변수로 선언할 것이며, 또 어떤 속성들을 인스턴스 변수로 선언할 것인지 생각해보자.

```
class Card {  
    String kind ;           // 카드의 무늬 - 인스턴스 변수  
    int number;            // 카드의 숫자 - 인스턴스 변수
```



```

static int width = 100 ;      // 카드의 폭 - 클래스 변수
static int height = 250 ;    // 카드의 높이 - 클래스 변수
}

```

각 Card인스턴스는 자신만의 무늬(kind)와 숫자(number)를 유지하고 있어야 하므로 이들을 인스턴스변수로 선언하였고, 각 카드들의 폭(width)과 높이(height)는 모든 인스턴스가 공통적으로 같은 값을 유지해야하므로 클래스변수로 선언하였다.

만일 카드의 폭을 변경해야할 필요가 있을 때는 모든 카드의 width값을 변경하지 않고, 한 카드의 width값만 변경해도 모든 카드의 width값이 변경되는 셈이다.

#### [예제6-4] CardTest.java

```

class CardTest{
    public static void main(String args[]) {
        // 클래스변수(static 변수)는 객체생성없이 '클래스이름.클래스변수'로 직접 사용
        가능하다.

        System.out.println("Card.width = " + Card.width);
        System.out.println("Card.height = " + Card.height);

        Card c1 = new Card();
        c1.kind = "Heart";
        c1.number = 7;

        Card c2 = new Card();
        c2.kind = "Spade";
        c2.number = 4;

        System.out.println("c1은 " + c1.kind + ", " + c1.number + "이며, 크기는 (" +
        c1.width + ", " + c1.height + ")");

        System.out.println("c2는 " + c2.kind + ", " + c2.number + "이며, 크기는 (" +
        c2.width + ", " + c2.height + ")");      System.out.println("이제 c1의 width와
        height를 각각 50, 80으로 변경합니다.");

        c1.width = 50;

```

```

        c1.height = 80;

        System.out.println("c1은 " + c1.kind + ", " + c1.number + "이며, 크기는 (" +
c1.width + ", " + c1.height + ")");

        System.out.println("c2는 " + c2.kind + ", " + c2.number + "이며, 크기는 (" +
c2.width + ", " + c2.height + ")");
    }
}

class Card {
    String kind ;           // 카드의 무늬 - 인스턴스 변수
    int number;             // 카드의 숫자 - 인스턴스 변수
    static int width = 100;  // 카드의 폭 - 클래스 변수
    static int height = 250; // 카드의 높이 - 클래스 변수
}

```

#### [실행결과]

```

Card.width = 100
Card.height = 250
c1은 Heart, 7이며, 크기는 (100, 250)
c2는 Spade, 4이며, 크기는 (100, 250)
이제 c1의 width와 height를 각각 50, 80으로 변경합니다.
c1은 Heart, 7이며, 크기는 (50, 80)
c2는 Spade, 4이며, 크기는 (50, 80)

```

Card클래스의 클래스변수(static변수)인 width, height는 Card클래스의 인스턴스를 생성하  
지 않고도 '클래스이름.클래스변수'와 같은 방식으로 사용할 수 있다.

Card인스턴스인 c1과 c2는 클래스 변수인 width와 height를 공유하기 때문에, c1의 width  
와 height를 변경하면 c2의 width와 height값도 바뀐 것과 같은 결과를 얻는다.

Card.width, c1.width, c2.width는 모두 같은 저장공간을 참조하므로 항상 같은 값을 갖게  
된다.

인스턴스 변수는 인스턴스가 생성될 때 마다 생성되므로 인스턴스마다 각기 다른 값을 유지할  
수 있지만, 클래스 변수는 모든 인스턴스가 하나의 저장공간을 공유하므로, 항상 공통된 값을  
갖는다.

[플래시동영상]자료실의 플래시동영상 [MemberVar.swf](#)을 꼭 보도록하자.

### 3.3 메서드

메서드는 어떤 작업을 수행하기 위한 명령문의 집합이다. 주로 어떤 값을 입력받아서 처리하고 그 결과를 되돌려 준다. 경우에 따라서는 입력받는 값이 없을 수도 있고 결과를 반환하지 않을 수도 있다.

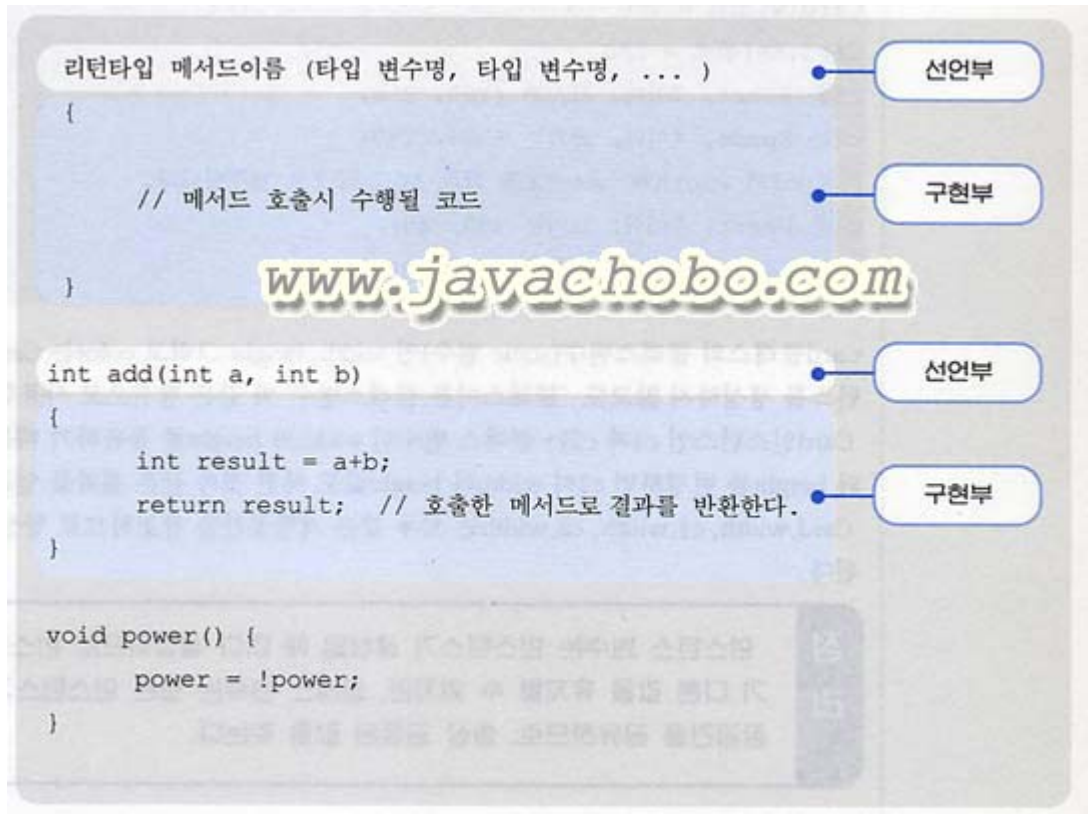
메서드를 작성하는 가장 큰 이유는 반복적으로 사용되는 코드를 줄이기 위해서이다. 자주 사용되는 내용의 코드를 메서드로 작성해 놓고 필요한 곳에서 호출만 하면 된다. 코드의 양도 줄일 수 있고 코드를 한 곳에서만 관리하면 되므로 유지보수가 편리하다.

- 하나의 메서드는 한 가지 기능만 수행하도록 작성하는 것이 좋다.
- 반복적으로 수행되어야 하는 여러 문장을 하나의 메서드로 정의해 놓으면 좋다.
- 관련된 여러 문장을 하나의 메서드로 만들어 놓는 것이 좋다.

메서드의 내부 코드를 몰라도 메서드를 호출할 때 어떤 값을 필요로 하고 어떤 결과를 반환한다는 것만 알아도 프로그램을 작성할 수 있다.

실제 프로젝트에서는 고급개발자들이 프로젝트에 사용될 주요 메서드를 미리 작성해 놓고, 초중급개발자들이 이 들을 사용해서 개발하는 방식으로 프로젝트를 진행한다.

이제 메서드를 작성하는 방법에 대해서 알아보도록 하자.



메서드는 크게 선언부와 구현부(몸통, body), 두 부분으로 나누어져 있다. 메서드의 선언부에는 리턴타입, 메서드이름, 그리고 괄호()에 매개변수를 선언하고, 구현부에는 메서드가 호출되었을 때 수행되어야 할 코드를 넣어 주면 된다.

메서드는 호출될 때 이 매개변수를 통해서 호출하는 메서드로부터 작업수행에 필요한 값들을 제공 받는다. 매개변수의 수는 메서드에 따라 없을 수도 있고, 여러 개일 수도 있다. 매개변수가 여러 개인 경우 쉼표(,)로 구분하여 나열한다.

메서드의 괄호()에 선언된 매개변수는 지역변수로 간주되어 메서드 내에서만 사용될 수 있으며, 메서드가 종료되는 순간 메모리에서 제거되어 더 이상 사용할 수 없게 된다.

리턴타입(return type)은 메서드의 수행결과를 어떤 타입(자료형)으로 반환할 것인지를 알려주는 것이다. 메서드가 결과값을 반환하지 않는 경우에는 리턴타입 대신 void를 사용하며, 메서드가 결과값을 반환하는 경우에는 메서드 내에 반드시 return문을 사용해서 리턴타입에 맞는 결과값을 호출한 메서드에게 반환하도록 해야 한다.

위의 add메서드는 두 개의 정수형(int)값을 입력 받아서, 덧셈 연산을 한 후 그 결과를 호출한 메서드에게 정수형(int) 결과값을 돌려주는 일을 한다.

이렇게 정의된 add메서드를 호출할 때는 메서드의 괄호()에 선언된 자료형 또는 타입에 맞는 값을 입력해주어야 한다. 그리고, 메서드의 결과를 저장하기 위해서는 메서드에 선언된 리턴타

입과 같은 타입의 변수를 준비해야한다.

### 3.4 return문

메서드가 정상적으로 종료되는 경우는 다음과 같이 두 가지가 있다.

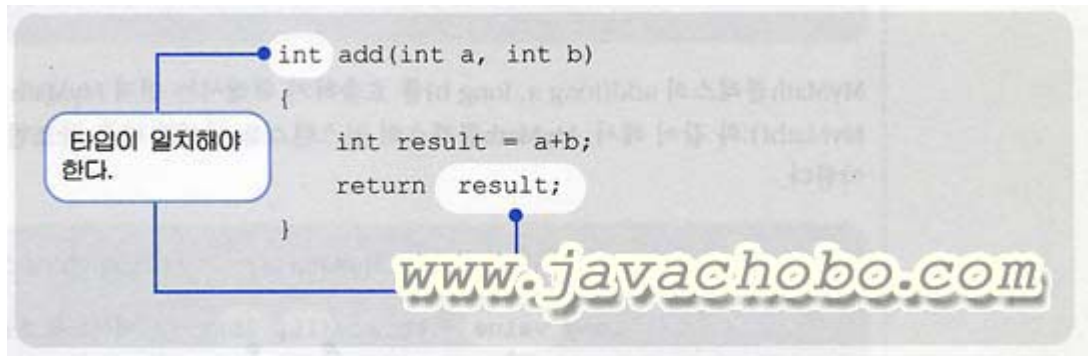
- 메서드의 블록{}내에 있는 모든 문장들을 수행했을 때
- 메서드의 블록{}내에 있는 문장을 수행중 return문을 만났을 때

return문은 현재 실행 중인 메서드를 종료하고 호출한 메서드로 되돌아가게 한다.

1. 반환값이 없는 경우 - return문만 써주면 된다.  
`return;`
2. 반환값이 있는 경우 - return문 뒤에 반환값을 지정해 주어야 한다.  
`return 반환값;`

반환값이 있는 경우에는 메서드의 선언부에 정의된 반환타입과 반환값의 타입이 일치하거나 반환값의 타입이 반환타입으로 자동형변환이 가능한 것이어야 한다.

[참고] 원칙적으로는 모든 메서드의 마지막에는 return문이 있어야 하지만, 반환값이 없는 메서드의 경우 return문을 생략한다고 볼 수 있다.



위의 그림에서 알 수 있는 것처럼, add메서드의 리턴타입이 int로 선언되었으므로, return문에서 반환해주는 값의 타입이 int 또는 int로 자동형변환이 가능한 byte, short, char 중의 하나이어야 한다.

### 3.5 메서드의 호출

메서드를 작성하는 방법에 대해서 알아보았으니 이제는 작성한 메서드를 사용하는 방법에 대해서 알아보도록 하자.

메서드를 호출하기 위해서는 다음과 같이 한다.

```
참조변수.메서드이름();           // 메서드에 선언된 매개변수가 없는 경우
참조변수.메서드이름(값1, 값2, ...); // 메서드에 선언된 매개변수가 있는 경우
```

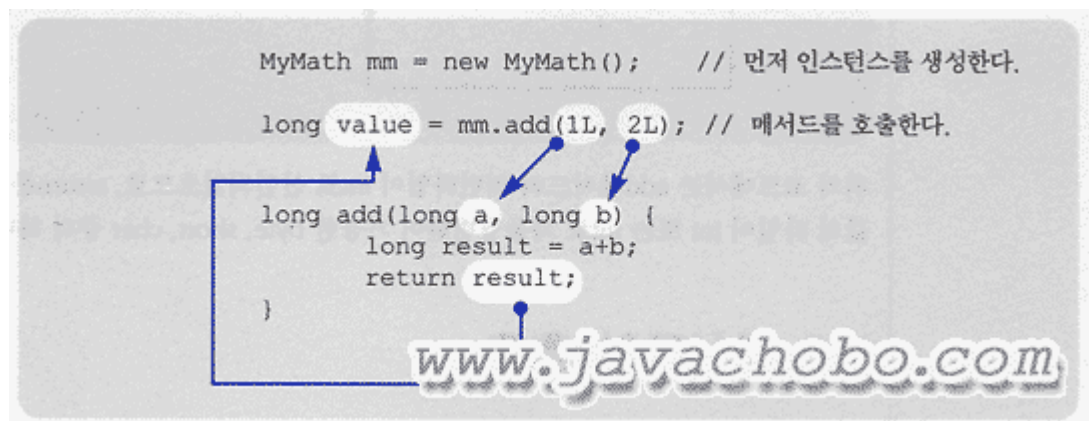
같은 클래스 내의 메서드끼리는 참조변수를 사용하지 않고도 서로 호출이 가능하지만 static메서드는 같은 클래스 내의 인스턴스 메서드를 호출할 수 없다.

다음은 두 개의 값을 매개변수로 받아서 사칙연산을 수행하는 4개의 메서드를 가진 MyMath 클래스를 정의한 것이다.

```
class MyMath {
    long add(long a, long b) {
        long result = a+b;
        return result;
    }
}
```

```
//    return a + b;    // 위의 두 줄을 이와 같이 한 줄로 간단히 할 수 있다.
}
long subtract(long a, long b) {    return a - b;    }
long multiply(long a, long b) {    return a * b;    }
double divide(double a, double b) {    return a / b;    }
}
```

MyMath클래스의 add(long a, long b)를 호출하기 위해서는 먼저 MyMath mm = new MyMath();와 같이 해서, MyMath클래스의 인스턴스를 생성한 다음 참조변수 mm을 통해야 한다.



add메서드의 매개변수의 타입이 long이므로 long 또는 long으로 자동형변환이 가능한 값을 지정해야 한다. 호출시 매개변수로 지정된 값은 메서드의 매개변수로 복사된다. 위의 코드에서는 1L과 2L의 값이 long타입의 매개변수 a와 b에 각각 복사된다.

메서드는 호출시 넘겨받은 값으로 연산을 수행하고 그 결과를 반환하면서 종료된다. 반환된 값은 대입연산자에 의해서 변수 value에 저장된다. 메서드의 결과를 저장하기 위한 변수 value역시 반환값과 같은 타입이거나 반환값이 자동형변환되어 저장될 수 있는 타입이어야 한다.

#### [예제6-5] MyMathTest.java

```
class MyMathTest {
    public static void main(String args[]) {
        MyMath mm = new MyMath();
        long result1 = mm.add(5L, 3L);
        long result2 = mm.subtract(5L, 3L);
    }
}
```

```

        long result3 = mm.multiply(5L, 3L);
        double result4 = mm.divide(5L, 3L);        // double대신 long값을 입력했다.
        System.out.println("add(5L, 3L) = " + result1);
        System.out.println("subtract(5L, 3L) = " + result2);
        System.out.println("multiply(5L, 3L) = " + result3);
        System.out.println("divide(5L, 3L) = " + result4);
    }
}

class MyMath {
    long add(long a, long b) {
        long result = a+b;
        return result;
        //    return a + b;    // 위의 두 줄을 이와 같이 한 줄로 간단히 할 수 있다.
    }

    long subtract(long a, long b) {
        return a - b;
    }

    long multiply(long a, long b) {
        return a * b;
    }

    double divide(double a, double b) {
        return a / b;
    }
}

```

#### [실행결과]

```

add(5L, 3L) = 8
subtract(5L, 3L) = 2
multiply(5L, 3L) = 15
divide(5L, 3L) = 1.6666666666666667

```

사칙연산을 위한 4개의 메서드가 정의되어 있는 MyMath클래스를 이용한 예제이다. 이 예제



를 통해서 클래스에 선언된 메서드를 어떻게 호출하는지 알 수 있을 것이다.

여 기서 눈여겨봐야 할 곳은 `divide(double a, double b)`를 호출하는 부분이다. `divide`메서드에 선언된 매개변수 타입은 `double`형인데, 이와 다른 `long`형의 값인 `5L`과 `3L`을 사용해서 호출하는 것이 가능하였다.

```
double result4 = mm.divide( 5L , 3L );

double divide(double a, double b) {
    return a / b;
}
```

호출 시에 입력된 값은 메서드의 매개변수에 대입되는 값이므로, `long`형의 값을 `double`형 변수에 저장하는 것과 같아서 `double a = 5L;`을 수행 했을 때와 같이 `long`형의 값인 `5L`은 `double`형 값인 `5.0`으로 자동형변환되어 `divide`의 매개변수 `a`에 저장된다.

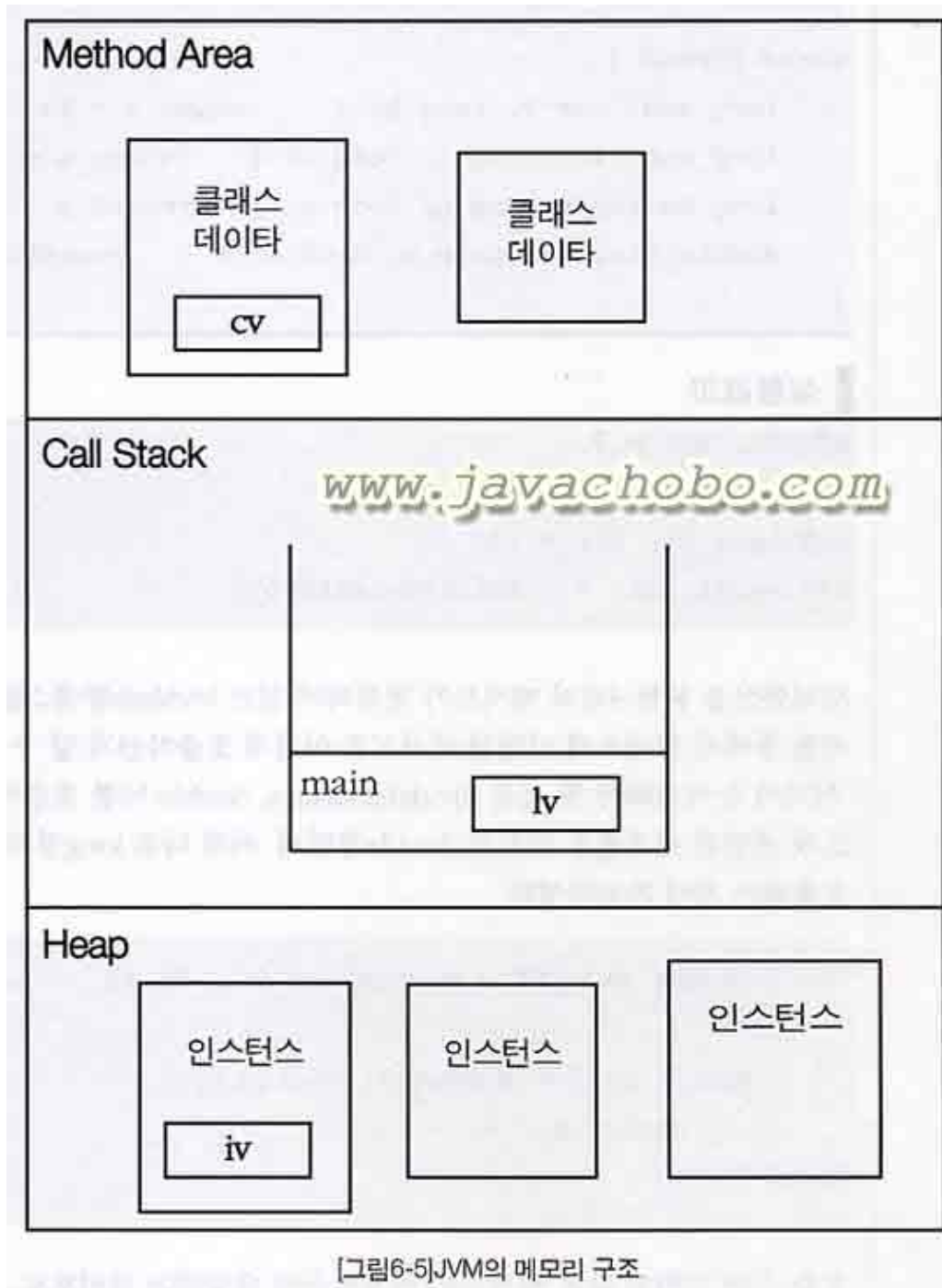
그래서, `divide`메서드에 두 개의 정수값(`5L`, `3L`)을 입력하여 호출하였음에도 불구하고 연산결과가 `double`형의 값이 된다.

이와 마찬가지로 `add(long a, long b)`메서드에도 매개변수 `a`, `b`에 `int`형의 값을 넣어 `add(5,3)`과 같이 호출하는 것이 가능하다.

### 3.6 JVM의 메모리구조

응용프로그램이 실행되면, JVM은 시스템으로부터 프로그램을 수행하는데 필요한 메모리를 할당받고 JVM은 이 메모리를 용도에 따라 여러 영역으로 나누어 관리한다.

그 중 3가지 주요영역(Method Area, 호출스택, Heap)에 대해서 알아보도록 하자.



[참고] cv는 클래스변수, lv는 지역변수, iv는 인스턴스변수를 뜻한다.

#### 1. 메소드영역(Method Area)

- 프로그램 실행 중 어떤 클래스가 사용되면, JVM은 해당 클래스의 클래스파일(\*.class)을 읽어서 분석하여 클래스에 대한 정보(클래스 데이터)를 Method Area에 저장한다.  
이 때, 그 클래스의 클래스변수(class variable)도 이 영역에 함께 생성된다.

#### 2. 힙(Heap)

- 인스턴스가 생성되는 공간. 프로그램 실행 중 생성되는 인스턴스는 모두 이 곳에 생성된다.  
즉, 인스턴스변수(instance variable)들이 생성되는 공간이다.

### 3. 호출스택(Call Stack 또는 Execution Stack)

호출스택은 메서드의 작업에 필요한 메모리 공간을 제공한다. 메서드가 호출되면, 호출스택에 호출된 메서드를 위한 메모리가 할당되며, 이 메모리는 메서드가 작업을 수행하는 동안 지역변수(매개변수 포함)들과 연산의 중간결과 등을 저장하는데 사용된다. 그리고, 메서드가 작업을 마치게 되면, 할당되었던 메모리공간은 반환되어 비워진다.

각 메서드를 위한 메모리상의 작업공간은 서로 구별되며, 첫 번째로 호출된 메서드를 위한 작업공간이 호출스택의 맨 밑에 마련되고, 첫 번째 메서드 수행중에 다른 메서드를 호출하게 되면, 첫 번째 메서드의 바로 위에 두 번째로 호출된 메서드를 위한 공간이 마련된다.

이 때 첫 번째 메서드는 수행을 멈추고, 두 번째 메서드가 수행되기 시작한다. 두 번째로 호출된 메서드가 수행을 마치게 되면, 두 번째 메서드를 위해 제공되었던 호출스택의 메모리공간이 반환되며, 첫 번째 메서드는 다시 수행을 계속하게 된다. 첫 번째 메서드가 수행을 마치면, 역시 제공되었던 메모리 공간이 호출스택에서 제거되며 호출스택은 완전히 비워지게 된다.

호출스택의 제일 상위에 위치하는 메서드가 현재 실행 중인 메서드이며, 나머지는 대기상태에 있게 된다.

따라서, 호출스택을 조사해 보면 메서드 간의 호출관계와 현재 수행중인 메서드가 어느 것인지 알 수 있다.

호출스택의 특징을 요약해보면 다음과 같다.

- 언제나 호출스택의 제일 위에 있는 메서드가 현재 실행 중인 메서드이다.
- 아래에 있는 메서드가 바로 위의 메서드를 호출한 메서드이다.

반환타입(return type)이 있는 메서드는 종료되면서 결과값을 자신을 호출한 메서드(caller)에게 반환한다. 대기상태에 있던 호출한 메서드(caller)는 넘겨받은 반환값으로 수행을 계속 진행하게 된다.

#### [예제6-6] CallStackTest.java

```
class CallStackTest {  
    public static void main(String[] args) {  
        firstMethod();  
    }  
}
```

```

    }

    static void firstMethod() {
        secondMethod();
    }

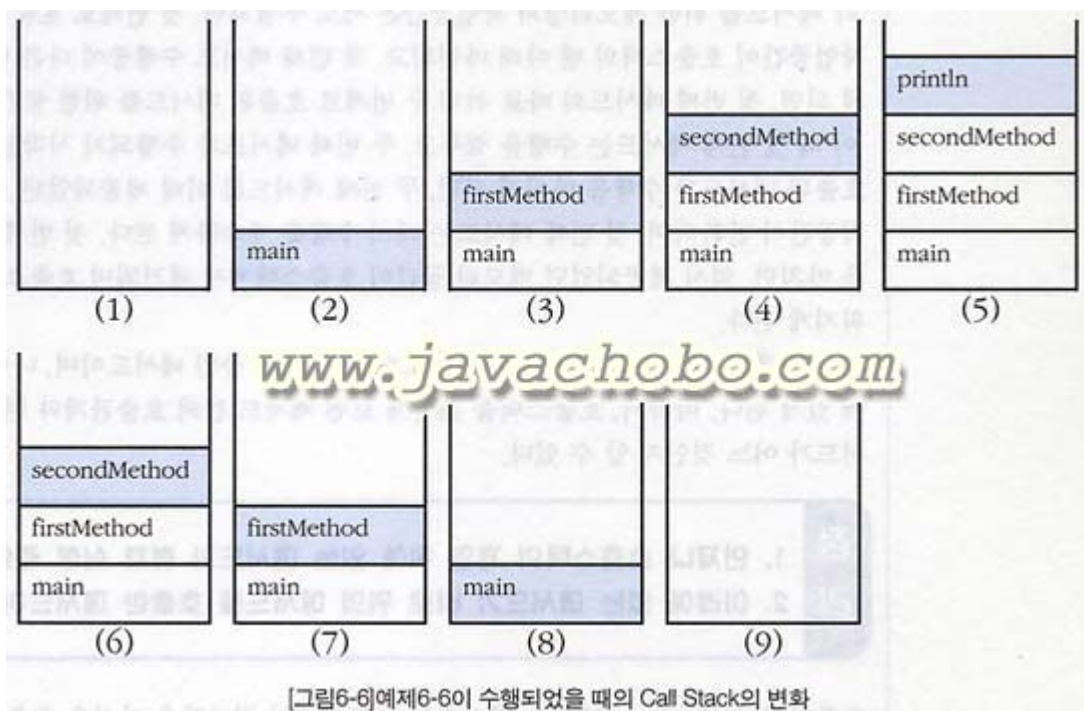
    static void secondMethod() {
        System.out.println("secondMethod()");
    }
}

```

#### [실행결과]

secondMethod()

위의 예제를 실행시켰을 때, 프로그램이 수행되는 동안 호출스택의 변화를 그림과 함께 살펴보도록 하자



(1)~(2) 위의 예제를 컴파일한 후 실행시키면, JVM에 의해서 main메서드가 호출됨으로써 프로그램이 시작된다. 이때, 호출스택에는 main메서드를 위한 메모리공간이 할당되고 main메서드의 코드가 수행되기 시작한다.

(3) main메서드에서 firstMethod()를 호출한 상태이다. 아직 main메서드가 끝난 것은 아니므로 main메서드는 호출스택에 대기상태로 남아있고 firstMethod()의 수행이 시작된다.

- (4) firstMethod()에서 다시 secondMethod()를 호출했다. firstMethod()는 secondMethod()가 수행을 마칠 때까지 대기상태에 있게 된다. secondMethod()가 수행을 마쳐야 firstMethod()의 나머지 문장들을 수행할 수 있기 때문이다.
- (5) secondMethod()에서 println메서드를 호출했다. 이때, println메서드에 의해서 화면에 "secondMethod()"가 출력된다.
- (6) println메서드의 수행이 완료되어 호출스택에서 사라지고 자신을 호출한 secondMethod()로 되돌아간다. 대기 중이던 secondMethod()는 println메서드를 호출한 이후부터 수행을 재개한다.
- (7) secondMethod()에 더 이상 수행할 코드가 없으므로 종료되고, 자신을 호출한 firstMethod()로 돌아간다.
- (8) firstMethod()에도 더 이상 수행할 코드가 없으므로 종료되고, 자신을 호출한 main메서드로 돌아간다.
- (9) main메서드에도 더 이상 수행할 코드가 없으므로 종료되어, 호출스택은 완전히 비워지게 되고 프로그램은 종료된다.

**[예제6-7] CallStackTest2.java**

```
class CallStackTest2 {  
    public static void main(String[] args) {  
        System.out.println("main(String[] args)이 시작되었음.");  
        firstMethod();  
        System.out.println("main(String[] args)이 끝났음.");  
    }  
    static void firstMethod() {  
        System.out.println("firstMethod()이 시작되었음.");  
        secondMethod();  
        System.out.println("firstMethod()이 끝났음.");  
    }  
    static void secondMethod() {  
        System.out.println("secondMethod()이 시작되었음.");  
        System.out.println("secondMethod()이 끝났음.");  
    }  
}
```

#### [실행결과]

```
main(String[] args)이 시작되었음.  
firstMethod()이 시작되었음.  
secondMethod()이 시작되었음.  
secondMethod()이 끝났음.  
firstMethod()이 끝났음.  
main(String[] args)이 끝났음.
```

### 3.7 기본형 매개변수와 참조형 매개변수

자 바에서는 메서드를 호출할 때 매개변수로 지정한 값을 메서드의 매개변수에 복사해서 넘겨 준다. 매개변수의 타입이 기본형(Primitive type)일 때는 기본형 값이 복사되겠지만, 참조형 (Reference type)이면 인스턴스의 주소가 복사된다.

메서드의 매개변수를 기본형으로 선언하면 단순히 저장된 값만 얻지만, 참조형으로 선언하면 값이 저장된 곳의 주소를 알 수 있기 때문에 값을 읽어 오는 것은 물론 값을 변경하는 것도 가능하다.

기본형 매개변수 - 변수의 값을 읽기만 할 수 있다.(read only)

참조형 매개변수 - 변수의 값을 읽고 변경할 수 있다.(read & write)

#### [예제6-8] ParameterTest.java

```
class Data { int x; }  
class ParameterTest {  
    public static void main(String[] args) {  
  
        Data d = new Data();  
        d.x = 10;  
        System.out.println("main() : x = " + d.x);  
    }  
}
```

```

        change(d.x);
        System.out.println("After change(d.x)");
        System.out.println("main() : x = " + d.x);
    }

    static void change(int x) {
        x = 1000;
        System.out.println("change() : x = " + x);
    }
}

```

#### [실행결과]

```

main() : x = 10
change() : x = 1000
After change(d.x)
main() : x = 10

```

[플래시동영상] 플래시동영상 자료실에서 기본형 매개변수 vs. 참조형 매개변수 ([PrimitiveParam.swf](#) vs. [ReferenceParam.swf](#))를 보면 실행과정과 함께 자세히 설명하고 있으니 여기서는 자세한 설명은 생략하겠다.

#### [예제6-9] ParameterTest2.java

```

class Data { int x; }
class ParameterTest2 {
    public static void main(String[] args) {

        Data d = new Data();
        d.x = 10;
        System.out.println("main() : x = " + d.x);

        change(d);
        System.out.println("After change(d)");
        System.out.println("main() : x = " + d.x);
    }
}

```

```

    }
    static void change(Data d) {
        d.x = 1000;
        System.out.println("change() : x = " + d.x);
    }
}

```

#### [실행결과]

```

main() : x = 10
change() : x = 1000
After change(d)
main() : x = 1000

```

### 3.8 재귀호출(Recursive Call)

메서드의 내부에서 메서드 자기자신을 다시 호출하는 것을 재귀호출(Recursive Call)이라 한다. 반복적인 작업을 해야 하는 메서드에 재귀호출을 이용하면, 메서드를 훨씬 간단하게 할 수 있는 경우가 있다. 하지만, 재귀호출은 다소 효율이 떨어진다는 단점이 있다.

대표적인 재귀호출의 예는 팩토리얼(factorial)을 구하는 것이다. 팩토리얼은 한 숫자를 1이 될 때까지 1씩 감소시켜가면서 계속해서 곱해 나가는 것인데,  $n!$  ( $n$ 은 양의 정수)와 같이 표현한다. 예를 들면,  $5! = 5 * 4 * 3 * 2 * 1 = 120$ 이다.

그리고, 팩토리얼을 수학적 함수로 표현하면 아래와 같이 표현할 수 있다.

$$f(n) = n * f(n-1), \text{ 단 } f(1) = 1.$$

다음 예제는 위의 함수를 자바로 구현한 것이다.

#### [예제6-10] FactorialTest.java



```

class FactorialTest {
    public static void main(String args[]) {
        System.out.println(factorial(4));
    }

    static long factorial(int n) {
        long result=0;
        if ( n == 1) {
            result = 1;
        } else {
            result = n * factorial(n-1);    // 다시 메서드 자신을 호출한다.
        }

        return result;
    }
}

```

#### [실행결과]

24

[참고] 삼항연산자 `? :` 를 이용하면 factorial메서드를 아래와 같이 더욱 간략하게 표현할 수 있다.

```

static long factorial(int n) {
    return (n == 1) ? 1 : n * factorial(n-1);
}

```

위 예제는 팩토리얼을 계산하는 메서드를 구현하고 테스트하는 것이다. factorial메서드가 main메서드와 같은 클래스내의 static메서드이므로 인스턴스를 생성하지 않고 직접 호출할 수 있다.

메서드가 자기자신을 다시 호출을 함으로써 반복문을 사용하지 않고도 반복적인 작업을 수행하도록 할 수 있다. 재귀호출은 반복문이 무한반복에 빠지는 것 처럼 무한호출이 되기 쉬우므

로 주의해야한다.

#### [예제6-11] MainTest.java

```
class MainTest {  
    public static void main(String args[]) {  
        main(null);        // 자기자신을 다시 호출한다.  
    }  
}
```

#### [실행결과]

```
java.lang.StackOverflowError  
    at MainTest.main(MainTest.java:3)  
    at MainTest.main(MainTest.java:3)  
    ...  
    at MainTest.main(MainTest.java:3)  
    at MainTest.main(MainTest.java:3)
```

main메서드 역시 자기자신을 호출하는 것이 가능한데, 아무런 조건도 없이 계속해서 자기자신을 다시 호출하기 때문에 무한호출에 빠지게 된다.

main메서드가 종료되지 않고 호출스택에 계속해서 쌓이게 되므로 결국 호출스택의 메모리 한계를 넘게 되고 StackOverflowError가 발생하여 프로그램은 비정상적으로 종료된다.

#### [예제6-12] PowerTest.java

```
class PowerTest  
{  
    public static void main(String[] args) {  
        int x = 2;  
        int n = 5;  
        long result = 0;  
  
        for(int i=1; i<=n; i++) {  
            result += power(x, i);  
        }  
    }  
}
```

```

    }

    System.out.println(result);
}

static long power(int x, int n) {
    if(n==1) return x;
    return x * power(x, n-1);
}
}

```

[실행결과]

62

x의 1제곱부터 x의 n제곱까지의 합을 구하는 예제이다. 재귀호출을 사용하여 x의 n제곱을 구하는 power메서드를 작성하였다. x는 2, n은 5로 계산했기 때문에  $2^1 + 2^2 + 2^3 + 2^4 + 2^5$ 의 결과인 62가 출력되었다.

### 3.9 클래스메서드(static메서드)와 인스턴스메서드

변수에서 그랬던 것과 같이, 메서드 앞에 static이 붙어 있으면 클래스메서드이고 붙어 있지 않으면 인스턴스메서드이다.

클래스 메서드는 호출방법 역시 클래스변수처럼, 객체를 생성하지 않고도 '클래스이름.메서드이름(매개변수)'와 같은 식으로 호출이 가능하다.

그렇다면 어느 경우에 static을 사용해서 클래스메서드로 정의해야하는 것일까?

클래스는 '데이터(변수)와 데이터에 관련된 메서드의 집합'이라고 할 수 있다. 같은 클래스 내에 있는 메서드와 멤버변수는 아주 밀접한 관계가 있다. 인스턴스메서드는 인스턴스변수와 관련된 작업을 하는, 즉 메서드의 작업을 수행하는데 인스턴스변수를 필요로 하는 메서드이다. 그래서 인스턴스변수와 관계없거나(메서드 내에서 인스턴스변수를 사용하지 않거나), 클래스

변수만을 사용하는 메서드들은 클래스메서드로 정의한다.

물론 인스턴스변수를 사용하지 않는다고 해서 반드시 클래스 메서드로 정의해야하는 것은 아니지만, 그렇게 하는 것이 일반적이다.

참 고로 Math클래스의 모든 메서드는 클래스메서드임을 알 수 있다. Math클래스에는 인스턴스변수가 하나도 없거니와 Math클래스의 함수들은 작업을 수행하는데 필요한 값들을 모두 매개변수로 받아서 처리 하기 때문이다. 이처럼, 단순히 함수들만의 집합인 경우에는 클래스메서드로 선언한다.

[참고]인스턴스 변수 뿐만 아니라 인스턴스 메서드를 호출하는 경우에도 인스턴스 메서드로 선언되어야 한다. 인스턴스 메서드를 호출하는 것 역시 인스턴스 변수를 간접적으로 사용하는 것이기 때문이다.

#### [예제6-12] MyMathTest2.java

```
class MyMath2 {
    long a, b;

    // 인스턴스변수 a, b를 이용한 작업을 하므로 매개변수가 필요없다.
    long add() {    return a + b; }
    long subtract() {    return a - b; }
    long multiply() {    return a * b; }
    double divide() {    return a / b; }

    // 인스턴스변수와 관계없이 매개변수만으로 작업이 가능하다.
    static long add(long a, long b) {    return a + b; }
    static long subtract(long a, long b) {    return a - b; }
    static long multiply(long a, long b) {    return a * b; }
    static double divide(double a, double b) {    return a / b; }
}

class MyMathTest2 {
    public static void main(String args[]) {
        // 클래스메서드 호출
        System.out.println(MyMath2.add(200L, 100L));
        System.out.println(MyMath2.subtract(200L, 100L));
    }
}
```

```

        System.out.println(MyMath2.multiply(200L, 100L));
        System.out.println(MyMath2.divide(200.0, 100.0));

        MyMath2 mm = new MyMath2();
        mm.a = 200L;
        mm.b = 100L;
        // 인스턴스메서드는 객체생성 후에만 호출이 가능함.
        System.out.println(mm.add());
        System.out.println(mm.subtract());
        System.out.println(mm.multiply());
        System.out.println(mm.divide());
    }

```

#### [실행결과]

```

300
100
20000
2.0
300
100
20000
2.0

```

인스턴스메서드인 `add()`, `subtract()`, `multiply()`, `divide()`는 인스턴스변수인 `a`와 `b`만으로도 충분히 원하는 작업이 가능하기 때문에, 매개변수를 필요로 하지 않으므로 괄호()에 매개변수를 선언하지 않았다.

반면에 `add(long a, long b)`, `subtract(long a, long b)` 등은 인스턴스변수 없이 매개변수만으로 작업을 수행하기 때문에 `static`을 붙여서 클래스메서드로 선언하였다.

`MyMathTest2`의 `main`메서드에서 보면, 클래스메서드는 객체생성없이 바로 호출이 가능했고, 인스턴스메서드는 `MyMath2`클래스의 인스턴스를 생성한 후에야 호출이 가능했다.

이 예제를 통해서 어떤 경우 인스턴스메서드로, 또는 클래스메서드로 선언해야하는지, 그리고 그 차이를 이해하는 것은 매우 중요하다.

### 3.10 클래스멤버와 인스턴스멤버간의 참조와 호출.

같은 클래스에 속한 멤버들간에는 별도의 인스턴스를 생성하지 않고도 서로 참조 또는 호출이 가능하다. 단, 클래스멤버가 인스턴스멤버를 참조 또는 호출하고자 하는 경우에는 인스턴스를 생성해야 한다.

그 이유는 인스턴스멤버가 존재하는 시점에 클래스멤버는 항상 존재하지만, 클래스멤버가 존재하는 시점에 인스턴스멤버가 항상 존재한다는 것을 보장할 수 없기 때문이다.

#### [예제6-13] MemberCall.java

```
class MemberCall {
    int iv = 10;
    static int cv = 20;

    int iv2 = cv;
// static int cv2 = iv;           에러. 클래스변수는 인스턴스 변수를 사용할 수 없음.
    static int cv2 = new MemberCall().iv; // 굳이 사용하려면 이처럼 객체를 생성해야 함.

    static void classMethod1() {
        System.out.println(cv);
// System.out.println(iv);   에러. 클래스메서드에서 인스턴스변수를 바로 사용할 수 없음.
        MemberCall c = new MemberCall();
        System.out.println(c.iv); // 객체를 생성한 후에야 인스턴스변수의 참조가 가능 함.
    }

    void instanceMethod1() {
        System.out.println(cv);
        System.out.println(iv); // 인스턴스메서드에서는 인스턴스변수를 바로 사용가능.
    }

    static void classMethod2() {
        classMethod1();
    }
}
```

```
//      instanceMethod1(); 에러. 클래스메서드에서는 인스턴스메서드를 바로 호출할 수
없음.

      MemberCall c = new MemberCall();
      c.instanceMethod1(); // 인스턴스를 생성한 후에야 인스턴스메서드를 호출할 수
있음.
    }

    void instanceMethod2() { // 인스턴스메서드에서는 인스턴스메서드와 클래스메서드
      classMethod1();      // 모두 인스턴스생성없이 바로 호출이 가능하다.
      instanceMethod1();
    }
}
```

클래스멤버(클래스변수와 클래스메서드)는 언제나 참조 또는 호출이 가능하다.

그렇기 때문에 인스턴스멤버가 클래스멤버를 참조, 호출하는 것은 아무런 문제가 안 된다. 클래스멤버간의 참조 또는 호출 역시 아무런 문제가 없다.

그러나, 인스턴스멤버(인스턴스변수와 인스턴스메서드)는 반드시 객체를 생성한 후에만 참조 또는 호출이 가능하기 때문에 클래스멤버가 인스턴스멤버를 참조, 호출하기 위해서는 객체를 생성하여야 한다.

하지만, 인스턴스멤버간의 호출에는 아무런 문제가 없다. 하나의 인스턴스멤버가 존재한다는 것은 인스턴스가 이미 생성되어있다는 것을 의미하며, 즉 다른 인스턴스멤버들도 모두 존재하기 때문이다.

실제로는 같은 클래스 내에서 클래스멤버가 인스턴스멤버를 참조 또는 호출해야하는 경우는 드물다. 만일 그런 경우가 발생한다면, 인스턴스메서드로 작성해야할 메서드를 클래스메서드로 한 것은 아닌지 한번 더 생각해봐야 한다.

#### [알아두면 좋아요]

수학에서의 대입법처럼, `c = new MemberCall()`이므로 `c.instanceMethod1()`에서 `c`대신 `new MemberCall()`을 대입하여 사용할 수 있다.

```
MemberCall c = new MemberCall();
```

```
int result = c.instanceMethod1();
```

위의 두 줄을 다음과 같이 한 줄로 할 수 있다.

```
int result = new MemberCall().instanceMethod1();
```

대신 참조변수를 사용하지 않았기 때문에 생성된 MemberCall인스턴스는 더 이상 사용할 수 없다.



# [Java의 정석]제6장 객체지향개념 1 - 4. 메서드 오버로딩 (Overloading)

자바의정석

2012/12/22 17:07

<http://blog.naver.com/gphic/50157739891>

## 4. 메서드 오버로딩(Method Overloading)

### 4.1 메서드 오버로딩이란?

메서드는 변수와 마찬가지로 같은 클래스 내에서 서로 구별될 수 있어야 하기 때문에 각기 다른 이름을 가져야 한다.

하지만, 자바에서는 한 클래스 내에 이미 사용하려는 이름과 같은 이름을 가진 메서드가 있더라도 매개변수의 개수 또는 타입이 다르면, 같은 이름을 사용해서 메서드를 정의할 수 있도록 했다.

이처럼, 한 클래스 내에 같은 이름의 메서드를 여러 개 정의하는 것을 메서드 오버로딩 (Method Overloading) 또는 간단히 오버로딩(Overloading)이라 한다.

오버로딩(Overloading)의 사전적 의미는 '과적하다.' 즉, 많이 싣는 것을 뜻한다. 보통 하나의 메서드 이름에 하나의 기능만을 구현해야하는데, 하나의 메서드 이름으로 여러 기능을 구현하기 때문에 붙여진 이름이라 생각할 수 있다. 앞으로는 메서드 오버로딩을 간단히 오버로딩이라고 하겠다.

### 4.2 오버로딩의 조건

같은 이름의 메서드를 정의한다고 해서 무조건 오버로딩인 것은 아니다. 오버로딩이 성립하기 위해서는 다음과 같은 조건을 만족해야한다.

- 메서드 이름이 같아야 한다.
- 매개변수의 개수 또는 타입이 달라야 한다.

- 매개변수는 같고 리턴타입이 다른 경우는 오버로딩이 성립되지 않는다.  
(리턴타입은 오버로딩을 구현하는데 아무런 영향을 주지 못한다.)

비록 메서드의 이름이 같아 하더라도 매개변수가 다르다면 서로 구별될 수 있기 때문에 오버로딩이 가능한 것이다. 위의 조건을 만족시키지 못하는 메서드는 중복정의 된 것으로 간주되어 컴파일시에 에러가 발생한다.

[참고] 오버로딩된 메서드들은 매개변수에 의해서만 구별될 수 있다.

#### 4.3 오버로딩의 예

오버로딩의 예로 가장 대표적인 것은 `println` 메서드이다. 지금까지 여러분은 `println` 메서드에 괄호 안에 값만 지정해주면 화면에 출력하는데 아무런 어려움이 없었다.

하지만, 실제로는 `println` 메서드를 호출할 때 매개변수로 지정하는 값의 타입에 따라서 호출되는 `println` 메서드가 달라진다.

`PrintStream` 클래스에는 어떤 종류의 매개변수를 지정해도 출력할 수 있도록 아래와 같이 10개의 오버로딩된 `println` 메서드를 정의해놓고 있다.

```
void println()  
void println(boolean x)  
void println(char x)  
void println(char[] x)  
void println(double x)  
void println(float x)  
void println(int x)  
void println(long x)  
void println(Object x)  
void println(String x)
```

`println` 메서드를 호출할 때 매개변수로 넘겨주는 값의 타입에 따라서 위의 오버로딩된 메서드들 중의 하나가 선택되어 실행되는 것이다.

오버로딩에 관련된 몇 가지 예를 들어 자세히 살펴보도록 하자.

[보기1]

```
int add(int a, int b) { return a+b; }  
int add(int x, int y) { return x+y; }
```

위의 두 메서드는 매개변수의 이름만 다를 뿐 매개변수의 타입이 같기 때문에 오버로딩이 성립하지 않는다. 매개변수의 이름이 다르면 메서드 내에서 사용되는 변수의 이름이 달라질 뿐, 아무런 의미가 없다. 그래서, 이 두 메서드는 정확히 같은 것이다. 마치 수학에서,  $f(x) = x + 1$  과  $f(a) = a + 1$  이 같은 표현인 것과 같다.

컴파일하면, 'add(int,int) is already defined(이미 같은 메서드가 정의되었다).'라는 메시지가 나타날 것이다.

[보기2]

```
int add(int a, int b) { return a+b; }  
long add(int a, int b) { return (long)(a + b); }
```

이번 경우는 리턴타입만 다른 경우이다. 매개변수의 타입과 개수가 일치하기 때문에, add(3,3)과 같이 호출하였을 때 어떤 메서드가 호출된 것인지 결정할 수 없기 때문에 오버로딩으로 간주되지 않는다.

이 경우 역시 컴파일하면, 'add(int,int) is already defined(이미 같은 메서드가 정의되었다).'라는 메시지가 나타날 것이다..

[보기3]

```
long add(int a, long b) { return a+b; }  
long add(long a, int b) { return a+b; }
```

두 메서드 모두 int형과 long형 매개변수가 하나씩 선언되어 있지만, 서로 순서가 다른 경우이

다. 이 경우에는 호출 시 매개변수의 값에 의해 호출될 메서드가 구분될 수 있으므로 중복된 메서드 정의가 아닌, 오버로딩으로 간주한다.

이처럼 단지 매개변수의 순서만을 다르게 하여 오버로딩을 구현하면, 사용자가 매개변수의 순서를 외우지 않아도 되는 장점이 있지만, 오히려 단점이 될 수도 있기 때문에 주의해야한다.

예를 들어 `add(3,3L)`과 같이 호출되면 첫번째 메서드가, `add(3L, 3)`과 같이 호출되면 두 번째 메서드가 호출된다. 단, 이 경우에는 `add(3,3)`과 같이 호출할 수 없다. 이와 같이 호출할 경우, 두 메서드 중 어느 메서드가 호출된 것인지 알 수 없기 때문에 메서드를 호출하는 쪽에서 컴파일 에러가 발생한다.

#### [보기4]

```
int add(int a, int b) { return a+b; }
long add(long a, long b) { return a+b; }
int add(int[] a) {
    for(int i=0,result=0; i < a.length; i++) {
        result += a[i];
    }
    return result;
}
```

위 메서드들은 모두 바르게 오버로딩되어있다. 정의된 매개변수가 서로 다르긴 해도, 세 메서드 모두 매개변수로 넘겨받은 값을 더해서 그 결과를 돌려주는 일을 한다.

같은 일을 하지만 매개변수를 달리해야하는 경우에, 이와 같이 이름은 같고 매개변수를 다르게 하여 오버로딩을 구현한다.

[참고]세 번째 메서드는 정수형 배열을 매개변수로 넘겨주면, 배열의 모든 원소들의 값을 더해서 결과를 반환하는 작업을 한다.

## 4.4 오버로딩의 장점

지금까지 오버로딩의 정의와 성립하기 위한 조건을 알아보았다. 그렇다면 오버로딩을 구현함으로써 얻는 이득은 무엇인가에 대해서 생각해볼도록 하자.

만일 메서드도 변수처럼 단지 이름만으로 구별된다면, 한 클래스내의 모든 메서드들은 이름이 달라야한다. 그렇다면, 이전에 예로 들었던 10가지의 println메서드들은 각기 다른 이름을 가져야 한다.

예를 들면, 아래와 같은 방식으로 메서드 이름이 변경되어야 할 것이다.

```
void println()
void printlnBoolean(boolean x)
void printlnChar(char x)
void printlnDouble(double x)
void printlnString(String x)
```

모두 근본적으로는 같은 기능을 하는 메서드들이지만, 서로 다른 이름을 가져야 하기 때문에 메서드를 작성하는 쪽에서는 이름을 짓기도 어렵고, 메서드를 사용하는 쪽에서는 이름을 일일이 구분해서 기억해야하기 때문에 서로 부담이 된다.

하지만 오버로딩을 통해, 여러 메서드들이 println이라는 하나의 이름으로 정의될 수 있다면, println이라는 이름만 기억하면 되므로, 기억하기도 쉽고, 이름도 짧게 할 수 있어서 오류의 가능성을 많이 줄일 수 있다. 그리고, 메서드의 이름만 보고도 '이 메서드들은 이름이 같으니, 같은 기능을 하겠구나.'라고 쉽게 예측할 수 있게 된다.

또 하나의 장점은 메서드의 이름을 절약할 수 있다는 것이다. 하나의 이름으로 여러 개의 메서드를 정의할 수 있으니, 메서드의 이름을 짓는데 고민을 덜 수 있는 동시에 사용되었어야 할 메서드 이름을 다른 메서드의 이름으로 사용할 수 있기 때문이다.

#### [예제6-15] OverloadingTest.java

```
class OverloadingTest {
    public static void main(String args[]) {
        MyMath2 mm2 = new MyMath2();
        System.out.println("mm2.add(3, 3) 결과:" + mm2.add(3,3));
        System.out.println("mm2.add(3L, 3) 결과: " + mm2.add(3L,3));
        System.out.println("mm2.add(3, 3L) 결과: " + mm2.add(3,3L));
        System.out.println("mm2.add(3L, 3L) 결과: " + mm2.add(3L,3L));

        int[] a = {100, 200, 300};
```

```

        System.out.println("mm2.add(a) 결과: " + mm2.add(a));
    }
}

class MyMath2 {
    int add(int a, int b) {
        System.out.print("int add(int a, int b) - ");
        return a+b;
    }

    long add(int a, long b) {
        System.out.print("long add(int a, long b) - ");
        return a+b;
    }

    long add(long a, int b) {
        System.out.print("long add(long a, int b) - ");
        return a+b;
    }

    long add(long a, long b) {
        System.out.print("long add(long a, long b) - ");
        return a+b;
    }

    int add(int[] a) {          // 배열의 모든 요소의 합을 결과로 돌려준다.
        System.out.print("int add(int[] a) - ");
        int result = 0;
        for(int i=0; i < a.length;i++) {
            result += a[i];
        }
        return result;
    }
} // end of class

```

**[실행결과]**

```
int add(int a, int b) - mm2.add(3, 3) 결과:6  
long add(long a, int b) - mm2.add(3L, 3) 결과: 6  
long add(int a, long b) - mm2.add(3, 3L) 결과: 6  
long add(long a, long b) - mm2.add(3L, 3L) 결과: 6  
int add(int[] a) - mm2.add(a) 결과: 600
```

[참고]add(3L, 3), add(3, 3L), add(3L, 3L)의 결과는 모두 6L이지만,  
System.out.println(6L);을 수행하면 6이 출력된다.

# [Java의 정석]제6장 객체지향개념 1 - 5. 생성자(Constructor)

자바의정석

2012/12/22 17:07

<http://blog.naver.com/gphic/50157739930>

## 5. 생성자(Constructor)

### 5.1 생성자란

생성자는 인스턴스가 생성될 때 호출되는 '인스턴스 초기화 메서드'이다. 따라서 인스턴스변수의 초기화 작업에 주로 사용되며, 인스턴스 생성 시에 실행되어야 할 작업을 위해서도 사용된다.

[참고]인스턴스 초기화란, 인스턴스변수들을 초기화하는 것을 뜻한다.

생성자 역시 메서드처럼 클래스 내에 선언되며, 구조도 메서드와 유사하지만 리턴값이 없다는 점이 다르다. 그렇다고 해서 생성자 앞에 리턴값이 없음을 뜻하는 키워드 void를 사용하지는 않고, 단지 아무 것도 적지 않는다. 생성자의 조건은 다음과 같다.

1. 생성자의 이름은 클래스의 이름과 같아야 한다.
2. 생성자는 리턴 값이 없다.

[참고]생성자도 오버로딩이 가능하므로 하나의 클래스에 여러 개의 생성자가 있을 수 있다.

생성자는 다음과 같이 정의한다.

```
클래스이름(타입 변수명, 타입 변수명, ... ) {  
    /* 인스턴스 생성 시 수행될 코드, 주로 인스턴스멤버의 초기화 코드를 적는다. */  
}
```

예)

```
class Card {  
    Card() { // 매개변수가 없는 생성자.
```



```

        //...
    }

    Card(String k, int num) {    // 매개변수가 있는 생성자.
        //...
    }
    //...
}

```

사실 연산자 `new`가 인스턴스를 생성하는 것이지 생성자가 인스턴스를 생성하는 것은 아니다. 생성자라는 용어 때문에 오해하기 쉬운데, 생성자는 단순히 인스턴스변수들의 초기화에 사용되는 조금 특별한 메서드일 뿐이다. 생성자가 갖는 몇 가지 특징만 제외하면 메서드와 다르지 않다.

`Card`클래스의 인스턴스를 생성하는 코드를 예로 들어, 수행되는 과정을 단계별로 나누어 보았다.

```
Card c = new Card();
```

1. 연산자 `new`에 의해서 메모리(heap)에 `Card`클래스의 인스턴스가 생성된다.
2. 생성자 `Card()`가 호출되어 수행된다.
3. 연산자 `new`의 결과로, 생성된 `Card`인스턴스의 주소가 반환되어 참조변수 `c`에 저장된다.

지금까지 인스턴스를 생성하기위해 사용해왔던 '클래스이름()'이 바로 생성자였던 것이다. 인스턴스를 생성할 때는 반드시 클래스 내에 정의된 생성자 중의 하나를 선택하여 지정해주어야 한다.

## 5.2 기본 생성자(default constructor)

지금까지는 생성자를 모르고도 프로그래밍을해 왔지만, 사실 모든 클래스에는 반드시 하나 이

상의 생성자가 정의되어 있어야 한다.

그러나 지금까지 클래스에 생성자를 정의하지 않고도 인스턴스를 생성할 수 있었던 이유는 컴파일러가 제공하는 '기본 생성자(default constructor)' 덕분이었다.

컴파일 할 때, 소스파일(\*.java)의 클래스에 생성자가 하나도 정의되지 않은 경우 컴파일러는 자동적으로 아래와 같은 내용의 기본 생성자를 추가하여 컴파일 한다.

```
클래스이름() {}
```

```
Card() {}
```

컴파일러가 자동적으로 추가해주는 기본 생성자는 이와 같이 매개변수도 없고 아무런 내용도 없는 아주 간단한 것이다.

그 동안 우리는 인스턴스를 생성할 때 컴파일러가 제공한 기본 생성자를 사용해왔던 것이다.

특히 인스턴스 초기화 작업이 요구되어지지 않는다면 생성자를 정의하지 않고 컴파일러가 제공하는 기본 생성자를 사용하는 것도 좋다.

[참고]클래스의 접근제어자(Access Modifier)가 public인 경우에는 기본생성자로 'public 클래스이름() {}'이 추가된다.

#### [예제6-16] ConstructorTest.java

```
class Data1 {  
    int value;  
}
```

```
class Data2 {  
    int value;  
    Data2(int x) {    // 매개변수가 있는 생성자.  
        value = x;  
    }  
}
```

```
class ConstructorTest {
```

```

    public static void main(String[] args) {
        Data1 d1 = new Data1();
        Data2 d2 = new Data2();          // compile error발생
    }
}

```

#### [컴파일결과]

```

ConstructorTest.java:15: cannot resolve symbol
symbol : constructor Data2 ()
location: class Data2
        Data2 d2 = new Data2();          // compile error발생
        ^
1 error

```

이것은 Data2에서 Data2()라는 생성자를 찾을 수 없다는 내용의 에러메시지인데, Data2에 Data2()생성자가 정의되어 있지 않기 때문에 에러가 발생한 것이다.

Data1의 인스턴스를 생성하는 코드에는 에러가 없는데, Data2의 인스턴스를 생성하는 코드에서 에러가 발생하는 이유는 무엇일까?

그 이유는 Data1에는 정의되어 있는 생성자가 하나도 없으므로 컴파일러가 기본생성자를 추가해주었지만, Data2에는 이미 생성자 Data2(int x)가 정의되어 있으므로 기본생성자가 추가되지 않았기 때문이다.

컴파일러가 자동적으로 기본생성자를 추가해주는 경우는 '클래스 내에 생성자가 하나도 없을 때'뿐이라는 것을 명심해야한다.

[참고]이 예제에서 컴파일 에러가 발생하지 않도록 하기 위해서는 Data2의 인스턴스를 생성할 때 생성자 Data2(int x)를 사용하던가, Data2에 생성자 Data2()를 추가로 정의해주면 된다.

### 5.3 매개변수가 있는 생성자.

생성자도 메서드처럼 매개변수를 선언하여, 호출 시 값을 넘겨받아서 인스턴스의 초기화 작업에 사용할 수 있다. 인스턴스마다 각기 다른 값으로 초기화되어야하는 경우가 많기 때문에 매개변수를 사용한 초기화는 매우 유용하다.

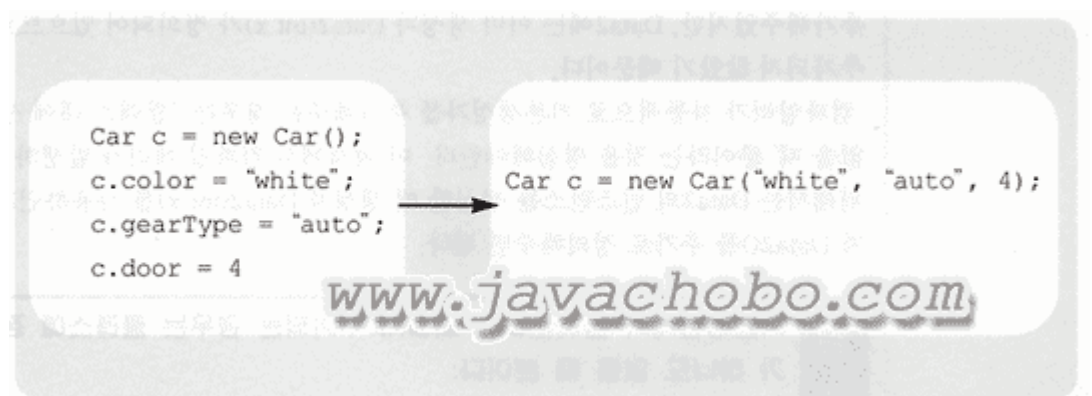
아래의 코드는 자동차를 클래스로 정의한 것인데, 단순히 color, gearType, door 세 개의 인

스턴스변수와 두 개의 생성자만을 가지고 있다.

```
class Car {  
    String color;        // 색상  
    String gearType;     // 변속기 종류 - auto(자동), manual(수동)  
    int door;            // 문의 갯수  
  
    Car() {}  
    Car(String c, String g, int d) {  
        color = c;  
        gearType = g;  
        door = d;  
    }  
}
```

Car인스턴스를 생성할 때, 생성자 Car()를 사용한다면, 인스턴스를 생성한 다음에 인스턴스 변수들을 따로 초기화해주어야 하지만, 매개변수가 있는 생성자 Car(String color, String gearType, int door)를 사용한다면 인스턴스를 생성하는 동시에 원하는 값으로 초기화를 할 수 있게 된다.

인스턴스를 생성한 다음에 인스턴스변수의 값을 변경하는 것보다 매개변수를 갖는 생성자를 활용하는 것이 코드를 보다 간결하고 직관적으로 만든다.



위의 양쪽 코드 모두 같은 내용이지만, 오른쪽의 코드가 더 간결하고 직관적이다. 이처럼 클래스를 작성할 때 다양한 생성자를 제공함으로써 인스턴스 생성 후에 별도로 초기화를 하지 않아

도 되도록 하는 것이 바람직하다.

[예제6-17] CarTest.java

```
class Car {
    String color;        // 색상
    String gearType;     // 변속기 종류 - auto(자동), manual(수동)
    int door;            // 문의 개수

    Car() {}
    Car(String c, String g, int d) {
        color = c;
        gearType = g;
        door = d;
    }
}

class CarTest {
    public static void main(String[] args) {
        Car c1 = new Car();
        c1.color = "white";
        c1.gearType = "auto";
        c1.door = 4;

        Car c2 = new Car("white", "auto", 4);
        System.out.println("c1의 color=" + c1.color + ", gearType=" + c1.gearType +
            ", door="+c1.door);
        System.out.println("c2의 color=" + c2.color + ", gearType=" + c2.gearType +
            ", door="+c2.door);
    }
}
```

[실행결과]

c1의 color=white, gearType=auto, door=4

c2의 color=white, gearType=auto, door=4

### 3.4 생성자에서 다른 생성자 호출하기 - this

같은 클래스의 멤버들간에 서로 호출할 수 있는 것처럼 생성자간에도 서로 호출이 가능하다. 단, 다음의 두 조건을 만족시켜야 한다.

- 생성자의 이름으로 클래스이름 대신 this를 사용한다.
- 한 생성자에서 다른 생성자를 호출할 때는 반드시 첫 줄에서만 호출이 가능하다.

#### [예제6-18] CarTest2.java

```
class Car {  
    String color;        // 색상  
    String gearType;     // 변속기 종류 - auto(자동), manual(수동)  
    int door;            // 문의 개수  
  
    Car() {  
        this("white", "auto", 4); // Car(String color, String gearType, int door)를 호  
출  
    }  
  
    Car(String color) {  
        this(color, "auto", 4);  
    }  
  
    Car(String color, String gearType, int door) {  
        this.color = color;  
        this.gearType = gearType;  
        this.door = door;  
    }  
}
```

```

}

class CarTest2 {
    public static void main(String[] args) {
        Car c1 = new Car();
        Car c2 = new Car("blue");

        System.out.println("c1의 color=" + c1.color + ", gearType=" + c1.gearType+
            ", door="+c1.door);

        System.out.println("c2의 color=" + c2.color + ", gearType=" + c2.gearType+
            ", door="+c2.door);
    }
}

```

#### [실행결과]

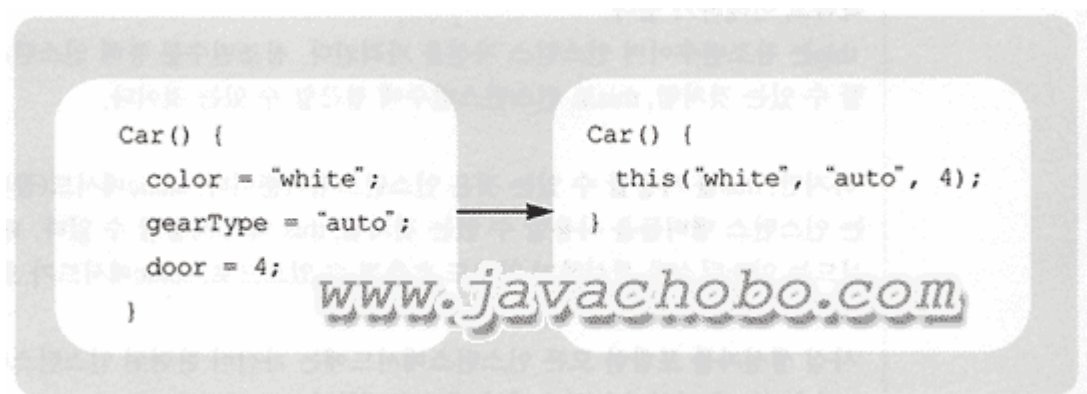
```

c1의 color=white, gearType=auto, door=4
c2의 color=blue, gearType=auto, door=4

```

생성자 Car()에서 또 다른 생성자 Car(String color, String gearType, int door)를 호출하였다. 이처럼 생성자간의 호출에는 생성자의 이름 대신 this를 사용해야만 하므로 'Car' 대신 'this'를 사용했다. 그리고 생성자 Car()의 첫째 줄에서 호출하였다는 점을 눈여겨보기 바란다.

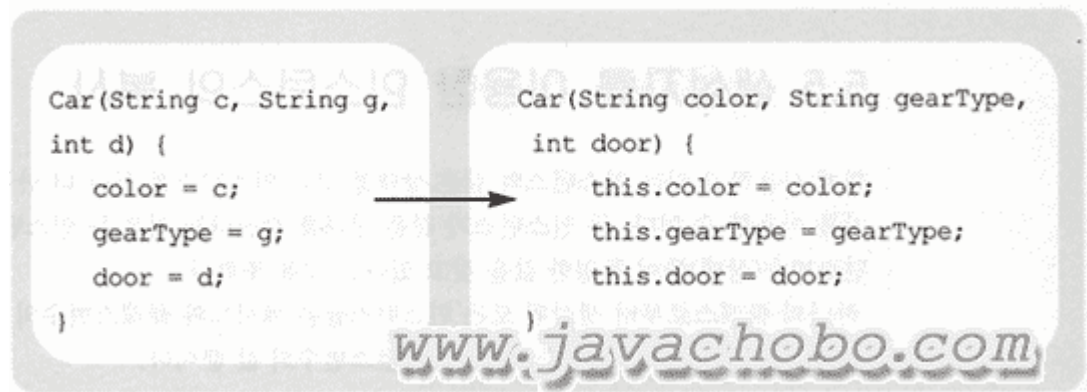
[참고] 생성자에서 다른 생성자를 첫 줄에서만 호출이 가능하도록 한 이유 중 하나는 생성자 내에서 초기화 작업도중에 다른 생성자 호출하게 되면, 호출된 다른 생성자 내에서도 멤버변수들의 값을 초기화를 할 것이므로 다른 생성자를 호출하기 이전의 초기화 작업이 무의미해질 수 있기 때문이다.



위 코드는 양쪽 모두 같은 일을 하지만 오른쪽의 코드는 생성자 Car(String color, String gearType, int door)를 활용해서 더 간략히 한 것이다. Car c1 = new Car();와 같이 생성자 Car()를 사용해서 Car인스턴스를 생성한 경우에, 인스턴스변수 color는 "white", gearType은 "auto", door는 4로 초기화 되도록 하였다.

이것은 마치 실생활에서 자동차(Car인스턴스)를 생산할 때, 아무런 옵션도 주지 않으면, 기본적으로 흰색(white)에 자동변속기어(auto) 그리고 문의 개수가 4개인 자동차가 생산되도록 하는 것에 비유할 수 있다.

같은 클래스 내의 생성자들은 일반적으로 서로 관계가 깊은 경우가 많아서 이처럼 서로 호출하도록 하여 유기적으로 연결해주면 더 좋은 코드를 얻을 수 있다. 그리고, 수정이 필요한 경우에도 보다 적은 코드만을 변경하면 되므로 유지보수가 쉬워진다.



왼쪽코드의 `color = c;`는 생성자의 매개변수로 선언된 지역변수 `c`의 값을 인스턴스변수 `color`에 저장한다. 이 때 변수 `color`와 `c`는 이름만으로도 서로 구별되므로 아무런 문제가 없다.

하지만, 오른쪽코드에서처럼 생성자의 매개변수로 선언된 변수의 이름이 `color`로 인스턴스변수 `color`와 같을 경우에는 이름만으로는 두 변수가 서로 구별이 안 된다. 이런 경우에는 인스턴스변수 앞에 'this'를 사용하면 된다.

그렇게 하면, `this.color`는 인스턴스변수이고, `color`는 생성자의 매개변수로 정의된 지역변수로 서로 구별이 가능하다. 만일 오른쪽코드에서 `this.color = color`대신 `color = color`와 같이 하면 둘 다 지역변수로 간주된다.

이처럼 생성자의 매개변수로 인스턴스변수들의 초기값을 제공받는 경우가 많기 때문에 매개변수와 인스턴스변수의 이름이 일치하는 경우가 자주 있다. 이 때는 왼쪽코드와 같이 매개변수 이름을 다르게 하는 것 보다 `this`를 사용해서 구별되도록 하는 것이 의미가 더 명확하고 이해하



기 쉽다.

this는 참조변수로 인스턴스 자신을 가리킨다. 참조변수를 통해 인스턴스의 멤버에 접근할 수 있는 것처럼, this로 인스턴스변수에 접근할 수 있는 것이다.

하지만, this를 사용할 수 있는 것은 인스턴스멤버뿐이다. static메서드(클래스메서드)에서는 인스턴스 멤버들을 사용할 수 없는 것처럼, this 역시 사용할 수 없다. 왜냐하면, static메서드는 인스턴스를 생성하지 않고도 호출될 수 있으므로, static메서드가 호출된 시점에 인스턴스가 존재하지 않을 수도 있기 때문이다.

사실 생성자를 포함한 모든 인스턴스메서드에는 자신이 관련된 인스턴스를 가리키는 참조변수 this가 지역변수로 숨겨진 채로 존재한다.

일 반적으로 인스턴스메서드는 특정 인스턴스와 관련된 작업을 하기 때문에 자신과 관련된 인스턴스의 정보가 필요하지만, static메서드는 인스턴스와 관련없는 작업을 하기 때문에 인스턴스에 대한 정보가 필요없기 때문이다. 인스턴스메서드와 static메서드의 차이를 다시 한번 되새겨 보도록 하자.

#### [정리]

this - 인스턴스 자신을 가리키는 참조변수, 인스턴스의 주소가 저장되어 있다.  
this(), this(매개변수) - 생성자, 같은 클래스의 다른 생성자를 호출할 때 사용한다.

### 5.5 생성자를 이용한 인스턴스의 복사

현 재 사용하고 있는 인스턴스와 같은 상태를 갖는 인스턴스를 하나 더 만들고자 할 때 생성자를 이용할 수 있다. 두 인스턴스가 같은 상태를 갖는다는 것은 두 인스턴스의 모든 인스턴스변수(상태)들이 동일한 값을 갖고 있다는 것을 뜻한다.

하나의 클래스로부터 생성된 모든 인스턴스들은 메서드와 클래스변수의 값이 서로 동일하며, 인스턴스들간의 차이는 오직 인스턴스변수의 값 뿐이다.

```
Car(Car c) {
```

```

        color = c.color;
        gearType = c.gearType;
        door = c.door;
    }

```

위의 코드는 Car클래스의 참조변수를 매개변수로 선언한 생성자이다. 매개변수로 넘겨진 참조 변수가 가리키는 Car인스턴스의 인스턴스변수인 color, gearType, door의 값을 인스턴스 자신으로 복사하는 것이다.

어떤 인스턴스의 상태를 자세히 몰라도 똑같은 인스턴스를 추가로 생성할 수 있다. Java API의 많은 클래스들이 인스턴스의 복사를 위한 생성자를 정의해놓고 있으니 참고하기 바란다.

지금까지 생성자에 대해서 모르고도 자바프로그래밍이 가능했던 것을 생각한다면, 생성자는 그리 중요하지 않은 것으로 생각될지도 모른다.

하지만, 지금까지 보아온 것과 같이 생성자를 적절히 활용하면 보다 간결하고 직관적인, 객체 지향적인 코드를 작성할 수 있을 것이다.

#### [예제6-19] CarTest3.java

```

class Car {
    String color;        // 색상
    String gearType;      // 변속기 종류 - auto(자동), manual(수동)
    int door;            // 문의 개수

    Car() {
        this("white", "auto", 4);
    }

    Car(Car c) {        // 인스턴스의 복사를 위한 생성자.
        color = c.color;
        gearType = c.gearType;
        door = c.door;
    }

    Car(String color, String gearType, int door) {

```

```

        this.color = color;
        this.gearType = gearType;
        this.door = door;
    }
}

class CarTest3 {
    public static void main(String[] args) {
        Car c1 = new Car();
        Car c2 = new Car(c1);    // c1의 복사본 c2를 생성한다.
        System.out.println("c1의 color=" + c1.color + ", gearType=" + c1.gearType +
            ", door=" + c1.door);
        System.out.println("c2의 color=" + c2.color + ", gearType=" + c2.gearType +
            ", door=" + c2.door);
        c1.door=100;    // c1의 인스턴스변수 door의 값을 변경한다.
        System.out.println("c1.door=100; 수행 후");
        System.out.println("c1의 color=" + c1.color + ", gearType=" + c1.gearType +
            ", door=" + c1.door);
        System.out.println("c2의 color=" + c2.color + ", gearType=" + c2.gearType +
            ", door=" + c2.door);
    }
}

```

#### [실행결과]

```

c1의 color=white, gearType=auto, door=4
c2의 color=white, gearType=auto, door=4
c1.door=100; 수행 후
c1의 color=white, gearType=auto, door=100
c2의 color=white, gearType=auto, door=4

```

인스턴스 c2는 c1을 복사하여 생성된 것이므로 서로 같은 상태를 갖지만, 서로 독립적으로 메모리공간에 존재하는 별도의 인스턴스이므로 c1의 값들이 변경되어도 c2는 영향을 받지 않는다.

#### [정리]

인스턴스를 생성할 때는 다음의 2가지 사항을 결정해야한다.

1. 클래스 - 어떤 클래스의 인스턴스를 생성할 것인가?
2. 생성자 - 선택한 클래스의 어떤 생성자로 인스턴스를 생성할 것인가?

# [Java의 정석]제6장 객체지향개념 1 - 6. 변수의 초기화(초기화블럭)

자바의정석

2012/12/22 17:09

<http://blog.naver.com/gphic/50157740030>

## 6. 변수의 초기화

### 6.1 변수의 초기화

변수를 선언하고 처음으로 값을 저장하는 것을 "변수의 초기화"라고 한다. 변수의 초기화는 경우에 따라서 필수적이기도 하고 선택적이기도 하지만, 가능하면 선언과 동시에 적절한 값으로 초기화 하는 것이 바람직하다.

멤버변수는 초기화를 하지 않아도 자동적으로 변수의 자료형에 맞는 기본값으로 초기화가 이루어지므로 초기화를 하지 않고 사용해도 되지만 지역변수는 사용하기 전에 반드시 초기화를 해야 한다.

```
class InitTest {  
    int x;  
    int y = x;  
  
    void method1() {  
        int i;  
        int j = i; // 컴파일 에러발생 : 지역변수를 초기화하지 않고 사용했음.  
    }  
}
```

위의 코드에서 x, y는 멤버변수이고, i, j는 지역변수이다. 그 중 x와 i는 선언만 하고 초기화를 하지 않았다. 그리고 y를 초기화 하는데 x를 사용하였고, j를 초기화 하는데 i를 사용하였다. 멤버변수 x는 초기화를 해주지 않아도 자동적으로 int형의 기본값인 0으로 초기화되므로, int y = x;와 같이 할 수 있다. x의 값이 0이므로 y역시 0이 저장된다.

하지만, method1의 지역변수 i는 자동적으로 초기화되지 않으므로, 초기화 되지 않은 상태에서 변수 j를 초기화 하는데 사용될 수 없기 때문에 컴파일시에 에러가 발생한다.

[참고]각 자료형의 기본값은 아래와 같다.

자료형	기본값
boolean	false
char	'\u0000'
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
참조형 변수	null

변수의 초기화에 대한 예를 몇 가지 더 들어보자.

선언 방법	선언예
<pre>int i=10; int j=10;</pre>	int형 변수 i를 선언하고 10으로 초기화 한다. int형 변수 j를 선언하고 10으로 초기화 한다.
<pre>int i=10, j=10;</pre>	자료형이 같은 경우 쉼표(.)를 사용해서 여러변수를 함께 선언할 수 있다.
<pre>int i=10, long j=10;</pre>	자료형이 다른 경우에는 이와 같이 함께 선언할 수 없다.
<pre>int i=10; int j=i;</pre>	i에 저장된 값으로 j를 초기화 한다. j에는 10이 저장된다.
<pre>int j=i int i=10;</pre>	i가 선언되기 전에 i를 사용할 수 없다.

멤버변수의 초기화에는 생성자 이외에도 명시적 초기화와 초기화블록을 이용한 방법이 있다.

이처럼 멤버변수의 초기화는 지역변수와 달리 여러가지 방법이 있다. 앞으로 멤버변수의 초기화에 대한 모든 방법에 대해 비교, 정리할 것이다.

#### 멤버변수의 초기화 방법

1. 명시적 초기화(Explicit initialization)
2. 생성자(Constructor)
3. 초기화 블록(Initialization block)
  - 인스턴스 초기화 블록 : 인스턴스변수를 초기화 하는데 사용.
  - 클래스 초기화 블록 : 클래스변수를 초기화 하는데 사용.

### 6.2 명시적 초기화(explicit initialization)

변수를 선언과 동시에 초기화하는 것을 명시적 초기화라고 한다. 가장 기본적이면서도 간단한 초기화 방법이므로 여러 초기화 방법 중에서 가장 우선적으로 고려되어야 한다.

```
class Car {  
    int door = 4;           // 기본형(primitive type) 변수의 초기화  
    Engine e = new Engine(); // 참조형(reference type) 변수의 초기화  
  
    //...  
}
```

명시적 초기화가 간단하고 명료하긴 하지만, 보다 복잡한 초기화 작업이 필요할 때는 "초기화 블록(initialization block)" 또는 생성자를 사용해야한다.

### 6.3 초기화 블록(initialization block)

초기화 블록에는 "클래스 초기화 블록"과 "인스턴스 초기화 블록" 두 가지 종류가 있다. 클래스 초기화 블록은 클래스변수의 초기화에 사용되고, 인스턴스 초기화 블록은 인스턴스변수의 초기화에 사용된다.

- \* 클래스 초기화 블록 - 클래스변수들의 초기화에 사용된다.
- \* 인스턴스 초기화 블록 - 인스턴스변수들의 초기화에 사용된다.

초기화 블록을 작성하려면, 인스턴스 초기화 블록은 단순히 클래스 내에 블록{}만들고 그 안에 코드를 작성하기만 하면 된다. 그리고, 클래스 초기화 블록은 인스턴스 초기화 블록 앞에 단순히 static을 덧붙이기만 하면 된다.

초기화 블록 내에는 메서드 내에서와 같이 조건문, 반복문, 예외처리구문 등을 자유롭게 사용할 수 있으므로, 초기화 작업이 복잡하여 명시적 초기화 만으로는 부족한 경우 초기화 블록을 사용한다.

```
class InitBlock {  
    static { /* 클래스 초기화블록 입니다. */ }  
  
    { /* 인스턴스 초기화블록 입니다. */ }  
  
    // ...  
}
```

클래스 초기화 블록은 클래스가 메모리에 처음 로딩될 때 한번만 수행되며, 인스턴스 초기화 블록은 생성자와 같이 인스턴스를 생성할 때 마다 수행된다. 그리고, 생성자보다는 인스턴스 초기화 블록이 먼저 수행된다는 사실도 기억해두자.

[참고]클래스가 처음 로딩될 때 클래스변수들이 메모리에 만들어지고, 바로 클래스 초기화블록이 클래스변수들을 초기화하게 되는 것이다.



인스턴스변수의 초기화는 주로 생성자를 사용하기 때문에, 인스턴스 초기화 블록은 잘 사용되지 않는다. 대신 클래스의 모든 생성자에서 공통적으로 수행되어야 하는 코드가 있는 경우 생성자에 넣지 않고 인스턴스 초기화 블록에 넣어 두면 코드의 중복을 줄일 수 있어서 좋다.

```
Car() {  
    System.out.println("Car인스턴스가 생성되었습니다.");    // 생성자마다 수행되어야 하는 코드  
    color="White";  
    gearType = "Auto";  
}  
  
Car(String color, String gearType) {  
    System.out.println("Car인스턴스가 생성되었습니다.");    // 생성자마다 수행되어야 하는 코드  
    this.color=color;  
    this.gearType = gearType;  
}
```

예를 들면, 위와 같이 클래스의 모든 생성자에 공통적으로 수행되어야 하는 문장들이 있을 때, 이 문장들을 각 생성자 마다 써주기 보다는 아래와 같이 인스턴스 블록을 이용하면 코드가 보다 간결해진다.

```
{ System.out.println("Car인스턴스가 생성되었습니다."); } // 인스턴스 블록  
  
Car() {  
    color="White";  
    gearType = "Auto";  
}  
  
Car(String color, String gearType) {  
    this.color=color;  
    this.gearType = gearType;
```

```
}
```

이처럼 코드의 중복을 제거하는 것은 코드의 신뢰성을 높여 주고, 오류의 발생가능성을 줄여 준다는 장점이 있다. 즉, 재사용성을 높이고 중복을 제거하는 것, 이 것이 바로 객체지향프로그래밍이 추구하는 궁극적인 목표이다.

프로그래머는 이와 같은 객체지향언어의 요소들을 잘 이해하고 활용하여 코드의 중복을 최대한 제거하기 위해서 노력해야한다.

#### [예제6-20] BlockTest.java

```
class BlockTest {

    // 클래스 초기화 블록
    static {
        System.out.println("static { }");
    }

    // 인스턴스 초기화 블록
    {
        System.out.println("{}");
    }

    // 생성자
    public BlockTest() {
        System.out.println("생성자");
    }

    public static void main(String args[]) {
        System.out.println("BlockTest bt = new BlockTest(); ");
        BlockTest bt = new BlockTest();

        System.out.println("BlockTest bt2 = new BlockTest(); ");
        BlockTest bt2 = new BlockTest();
    }
}
```

```

    }
}

```

#### [실행결과]

```

static {}
BlockTest bt = new BlockTest();
{}
생성자
BlockTest bt2 = new BlockTest();
{}
생성자

```

BlockTest 가 실행되면서 메모리에 로딩될 때, 클래스 초기화 블록이 가장 먼저 수행되어 "static {}"이 화면에 출력된다. 그 다음에 main메서드가 수행되어 BlockTest인스턴스가 생성되면서 인스턴스 초기화 블록이 먼저 수행되고, 그 다음에 생성자가 수행된다. 위의 실행결과에서도 알 수 있듯이 클래스 초기화 블록은 처음 메모리에 로딩될 때 한번만 수행되었지만, 인스턴스 초기화 블록은 인스턴스가 생성될 때 마다 수행되었다.

#### [예제6-21] StaticBlockTest.java

```

class StaticBlockTest {
    static int[] arr = new int[10];

    static {
        for(int i=0; i < arr.length; i++) {
            // 1과 10사이의 임의의 값을 얻어서 배열 arr에 저장한다.
            arr[i] = (int)(Math.random()*10) + 1;
        }
    } // 클래스 초기화블록의 끝.

    public static void main(String args[]) {
        for(int i=0; i< arr.length; i++)
            System.out.println("arr["+i+"] : " + arr[i]);
    }
}

```

```
}
```

#### [실행결과]

```
arr[0] :4  
arr[1] :8  
arr[2] :7  
arr[3] :2  
arr[4] :2  
arr[5] :10  
arr[6] :7  
arr[7] :10  
arr[8] :1  
arr[9] :7
```

명시적 초기화를 통해 배열 arr을 생성하고, 클래스 초기화 블록을 이용해서 배열의 각 요소들을 Math.random()을 사용해서 임의의 값으로 채우도록 했다.

이처럼 배열이나 예외처리가 필요한 초기화에서는 명시적 초기화 만으로는 복잡한 초기화 작업을 할 수 없다. 이런 경우에 추가적으로 클래스 초기화 블록을 사용하도록 한다.

[참고]인스턴스변수의 복잡한 초기화는 생성자 또는 인스턴스 초기화 블록을 사용한다.

## 6.4 멤버변수의 초기화 시기와 순서

지금까지 멤버변수를 초기화하는 방법에 대해서 알아보았다. 이제는 초기화가 수행되는 시기와 순서에 대해서 정리해보도록 하자.

- \* 클래스변수의 초기화시점 : 클래스가 처음 로딩될 때 단 한번 초기화 된다.
- \* 인스턴스변수의 초기화시점 : 인스턴스가 생성될 때마다 각 인스턴스별로 초기화가 이루어진다.
- \* 클래스변수의 초기화순서 : 기본값 -> 명시적초기화 -> 클래스 초기화 블록

\* 인스턴스변수의 초기화순서 : 기본값 -> 명시적초기화 -> 인스턴스 초기화 블록 -> 생성자

프로그램 실행도중 클래스에 대한 정보가 요구되어질 때, 클래스는 메모리에 로딩된다. 예를 들면, 클래스 멤버를 사용했을 때, 인스턴스를 생성한 경우 등이 이에 해당한다. 하지만, 해당 클래스가 이미 메모리에 로딩되어 있다면, 또다시 로딩하지 않는다. 물론 초기화도 다시 수행되지 않는다.

[참고]클래스의 로딩시기는 JVM의 종류에 따라 좀 다를 수 있는데, 클래스가 필요할 때 바로 메모리에 로딩하도록 설계가 되어있는 것도 있고, 실행효율을 높이기 위해서 사용될 클래스들을 프로그램이 시작될 때 미리 로딩하도록 되어있는 것도 있다.

```
class InitTest {  
    // 명시적 초기화  
    static int cv = 1;  
    int iv = 1;  
  
    // 클래스 초기화 블록  
    static {    cv = 2;    }  
  
    // 인스턴스 초기화 블록  
    {    iv = 2;    }  
  
    // 생성자  
    InitTest () {  
        iv = 3;  
    }  
}
```

[플래시동영상]자료실의 [Initailization.swf](#)에 보다 자세한 설명이 있으니 꼭 보도록 하자.

위의 InitTest클래스는 클래스변수(cv)와 인스턴스변수(iv)를 각각 하나씩 가지고 있다. new InitTest();와 같이 하여 인스턴스를 생성했을 때, cv와 iv가 초기화되어가는 과정을 단계별로 자세히 살펴보도록 하자.

클래스초기화			인스턴스초기화			
기본값	명시적 초기화	클래스 초기화블럭	기본값	명시적 초기화	인스턴스 초기화블럭	생성자
cv 0	cv 1	cv 2	cv 2 iv 0	cv 2 iv 1	cv 2 iv 2	cv 2 iv 3
1	2	3	4	5	6	7

- \* 클래스변수 초기화 (1~3) : 클래스가 처음 메모리에 로딩될 때 차례대로 수행됨.
- \* 인스턴스변수 초기화(4~7) : 인스턴스를 생성할 때 차례대로 수행됨

[중요] 클래스변수는 항상 인스턴스변수보다 항상 먼저 생성되고 초기화 된다.

1. cv가 메모리(Method Area)에 생성되고, cv는 int형 변수이므로 int형의 기본값인 0이 cv에 저장된다.
2. 그 다음에는 명시적 초기화(int cv=1)에 의해서 cv에 1이 저장된다.
3. 마지막으로 클래스 초기화 블럭(cv=2)이 수행되어 cv에는 2가 저장된다.
4. InitTest클래스의 인스턴스가 생성되면서 iv가 메모리(heap)에 존재하게 된다. iv 역시 int형 변수이므로 기본값 0이 저장된다.
5. 명시적 초기화에 의해서 iv에 1이 저장되고
6. 인스턴스 초기화 블럭이 수행되어 iv에 2가 저장된다.
7. 마지막으로 생성자가 수행되어 iv에는 3이 저장된다.

#### [예제6-22] ProductTest.java

```
class Product {
    static int count = 0;    // 생성된 인스턴스의 수를 저장하기 위한 변수
    int serialNo;           // 인스턴스 고유의 번호

    {
        // Product인스턴스가 생성될 때마다 count의 값을 1씩 증가시켜서 serialNo에 저장한다.
    }
}
```

```

        ++count;
        serialNo = count;
    }

    public Product() {}
}

class ProductTest {
    public static void main(String args[]) {
        Product p1 = new Product();
        Product p2 = new Product();
        Product p3 = new Product();

        System.out.println("p1의 제품번호(serial no)는 " + p1.serialNo);
        System.out.println("p2의 제품번호(serial no)는 " + p2.serialNo);
        System.out.println("p3의 제품번호(serial no)는 " + p3.serialNo);
        System.out.println("생산된 제품의 수는 모두 " +Product.count +"개 입니다.");
    }
}

```

#### [실행결과]

```

p1의 제품번호(serial no)는 1
p2의 제품번호(serial no)는 2
p3의 제품번호(serial no)는 3
생산된 제품의 수는 모두 3개 입니다.

```

공장에서 제품을 생산할 때 제품마다 생산일련번호(serial no)를 부여하는 것과 같이 Product 클래스의 인스턴스가 고유의 일련번호(serialNo)를 갖도록 하였다.

Product클래스의 인스턴스를 생성할 때마다 인스턴스 블록이 수행되어, 클래스변수 count의 값을 1증가시킨 다음, 그 값을 인스턴스변수 serialNo에 저장한다.

이렇게 함으로써 새로 생성되는 인스턴스는 이전에 생성된 인스턴스보다 1이 증가된 serialNo 값을 갖게 된다.

생성자가 하나 밖에 없기 때문에 인스턴스 블록 대신, Product클래스의 생성자를 사용해도 결과는 같지만, 코드의 의미상 모든 생성자에서 공통적으로 수행되어야하는 내용이기 때문에 인스턴스 블록을 사용하였다.

[참고]만일 count를 인스턴스변수로 선언했다면, 인스턴스가 생성될 때마다 0으로 초기화 될 것이므로 모든 Product인스턴스의 serialNo값은 항상 1이 될 것이다.

#### [예제6-23] DocumentTest.java

```
class Document {
    static int count = 0;
    String name;        // 문서명(Document name)

    public Document() { // 문서 생성 시 문서명을 지정하지 않았을 때
        this("제목없음" + ++count);
    }

    public Document(String name) {
        this.name = name;
        System.out.println("문서 " + this.name + "가 생성되었습니다.");
    }
}

class DocumentTest {
    public static void main(String args[]) {
        Document d1 = new Document();
        Document d2 = new Document("자바.txt");
        Document d3 = new Document();
        Document d4 = new Document();
    }
}
```

#### [실행결과]

문서 제목없음1가 생성되었습니다.  
문서 자바.txt가 생성되었습니다.  
문서 제목없음2가 생성되었습니다.  
문서 제목없음3가 생성되었습니다.

바로 이전의 일련번호 예제를 응용한 것으로, 워드프로세서나 문서편집기에 이와 유사한 코드



가 사용된다. 문서(Document)를 생성할 때, 문서의 이름을 지정하면 그 이름의 문서가 생성되지만, 문서의 이름을 지정하지 않으면 프로그램에서 일정한 규칙에 의해서 자동적으로 이름을 결정한다.

예를 들면, "제목없음1", "제목없음2", "제목없음3"... 과 같은 식으로 문서의 이름이 결정된다. 문서의 이름은 서로 구별될 수 있어야 하기 때문이다.

# [Java의 정석]제7장 객체지향개념 2 - 1. 상속(Inheritance)

자바의정석

2012/12/22 17:09

<http://blog.naver.com/gphic/50157740067>

## 1. 상속(Inheritance)

### 1.1 상속의 정의와 장점

상속이란, 기존의 클래스를 재사용하여 새로운 클래스를 작성하는 것이다. 상속을 통해서 클래스를 작성하면, 보다 적은 양의 코드로 새로운 클래스를 작성할 수 있고 코드를 공통적으로 관리할 수 있기 때문에 코드의 추가 및 변경이 매우 용이하다.

이러한 특징은 코드의 재사용성을 높이고 코드의 중복을 제거하여 프로그램의 생산성과 유지보수에 크게 기여한다.

자바에서 상속을 구현하는 방법은 아주 간단하다. 새로 작성하고자 하는 클래스의 이름 뒤에 상속받고자 하는 클래스의 이름을 키워드 'extends'와 함께 써 주기만 하면 된다.

예를 들어 새로 작성하려는 클래스의 이름이 Child이고 상속받고자 하는 기존 클래스의 이름이 Parent라면 다음과 같이 하면 된다.

```
class Child extends Parent {  
    // ...  
}
```

이 두 클래스는 서로 상속 관계에 있다고 하며, 상속해주는 클래스를 '조상클래스'라 하며 상속받는 클래스를 '자손 클래스'라 한다.

[참고]서로 상속관계에 있는 두 클래스를 아래와 같은 용어를 사용해서 표현하기도 한다.

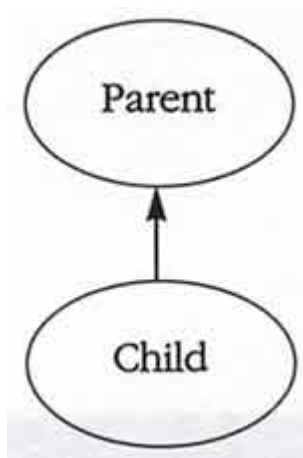
조상클래스

자손클래스

부모(parent) 클래스	자식(child) 클래스
상위(super) 클래스	하위(sub) 클래스
기반(base) 클래스	파생된(derived) 또는 확장된(extended) 클래스

다음과 같이 서로 상속관계에 있는 두 클래스를 그림으로 표현하면 다음과 같다.

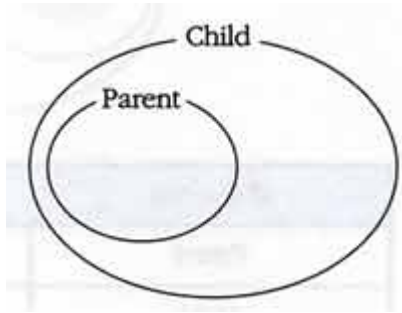
```
class Parent {}
class Child extends Parent {}
```



클래스는 타원으로 표현했고 클래스간의 상속 관계는 화살표로 표시했다. 이와 같이 클래스간의 상속관계를 그림으로 표현한 것을 상속계층도(Class Hierarchy)라고 한다.

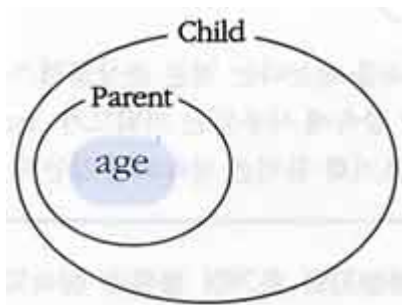
프로그램이 커질수록 클래스간의 관계가 복잡해지는데, 이 때 위와 같이 그림으로 표현하면, 클래스간의 관계를 보다 쉽게 이해할 수 있다.

자 손 클래스는 조상클래스의 모든 멤버를 상속받기 때문에, Child클래스는 Parent클래스의 멤버들을 포함한다고 할 수 있다. 클래스는 멤버들의 집합이므로 클래스 Parent와 Child의 관계를 다음과 같이 표현할 수도 있다.



만일 Parent클래스에 age라는 정수형 변수를 멤버변수로 추가하면, 자손 클래스는 조상의 멤버를 상속받기 때문에, Child클래스는 자동적으로 age라는 멤버변수가 추가된 것과 같은 효과를 얻는다.

```
class Parent {
    int age;
}
class Child extends Parent { }
```



클래스이름	클래스의 멤버
Parent	age
Child	age

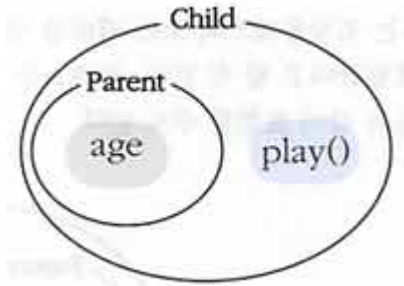
이번엔 반대로 자손인 Child클래스에 새로운 멤버로 play() 메서드를 추가해보자.

```
class Parent {
    int age;
}
class Child extends Parent {
```

```

void play() {
    System.out.println("놀자~");
}
}

```



클래스이름	클래스의 멤버
Parent	age
Child	age, play()

Child클래스에 새로운 코드가 추가되어도 조상인 Parent클래스는 아무런 영향도 받지 않는다. 여기서 알 수 있는 것처럼, 조상클래스가 변경되면 자손클래스는 자동적으로 영향을 받게 되지 만, 자손클래스가 변경되는 것은 조상클래스에 아무런 영향을 주지 못한다.

자손클래스는 조상클래스의 모든 멤버를 상속 받으므로 항상 조상클래스보다 같거나 많은 멤 버를 갖는다. 즉, 상속에 상속을 거듭할 수록 상속받는 클래스의 멤버 개수는 점점 늘어나게 된 다.

그래서 상속을 받는다는 것은 조상클래스를 확장(extend)한다는 의미로 해석할 수도 있으며 이 것이 상속에 사용되는 키워드가 'extends'인 이유이기도 하다.

[참고] 생성자나 초기화 블록은 상속되지 않는다. 오직 멤버변수와 메서드만 상속된다.

[참고] 접근제어자(Access Modifier)로 private 또는 default가 사용된 멤버들은 상속되지 않 는다기보다는 상속은 받지만 자손 클래스로부터의 접근이 제한되는 것으로 보는 것이 옳다.

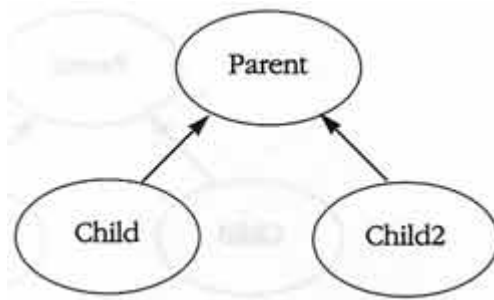
이번엔 Parent클래스로부터 상속받는 Child2클래스를 새로 작성해보자. Child2클래스를 포 함한 세 클래스간의 상속계층도는 다음과 같을 것이다.

```

class Parent {}
class Child extends Parent {}

```

```
class Child2 extends Parent { }
```



클래스 Child와 Child2가 모두 Parent클래스를 상속받고 있으므로 Parent클래스와 Child클래스, 그리고 Parent클래스와 Child2클래스는 서로 상속관계에 있지만 클래스 Child와 Child2간에는 서로 아무런 관계도 성립되지 않는다. 클래스간의 관계에서 형제 관계와 같은 것은 없다. 부모와 자식의 관계(상속관계)만이 존재할 뿐이다.

만일 Child클래스와 Child2클래스에 공통적으로 추가되어야 하는 멤버(멤버변수나 메서드)가 있다면, 이 두 클래스에 각각 따로 추가해주는 것보다는 이들의 공통조상인 Parent클래스에 추가하는 것이 좋다.

Parent 클래스의 자손인 Child클래스와 Child2클래스는 조상의 멤버를 상속받기 때문에, Parent클래스에 새로운 멤버를 추가해주는 것은 Child클래스와 Child2클래스에 새로운 멤버를 추가해주는 것과 같은 효과를 얻는다.

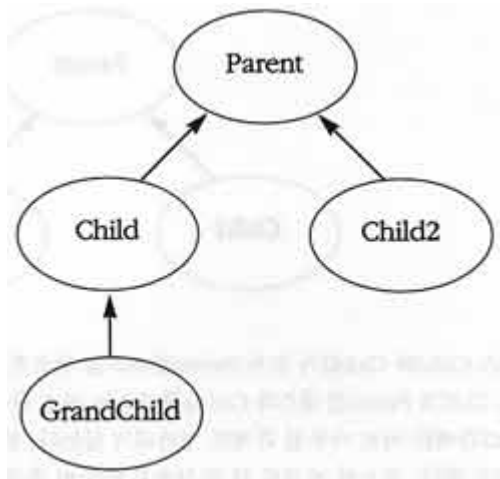
이제는 Parent클래스 하나만 변경하면 되므로 작업이 간단해진다. 이보다 더 중요한 사실은 같은 내용의 코드를 한 곳에서 관리함으로써 코드의 중복이 줄어든다는 것이다. 코드의 중복이 많아지면 유지보수가 어려워지고 일관성을 유지하기 어렵다.

이처럼 같은 내용의 코드를 하나 이상의 클래스에 중복적으로 추가해야 하는 경우에는 상속관계를 이용해서 코드의 중복을 최소화해야 한다. 프로그램이 어떤 때는 잘 동작하지만 어떤 때는 오동작을 하는 이유는 중복된 코드 중에서 바르게 변경되지 않은 곳이 있기 때문이다.

여기에 또다시 Child클래스로부터 상속받는 GrandChild라는 새로운 클래스를 추가한다면 상속계층도는 다음과 같을 것이다.

```
class Parent { }
class Child extends Parent { }
class Child2 extends Parent { }
```

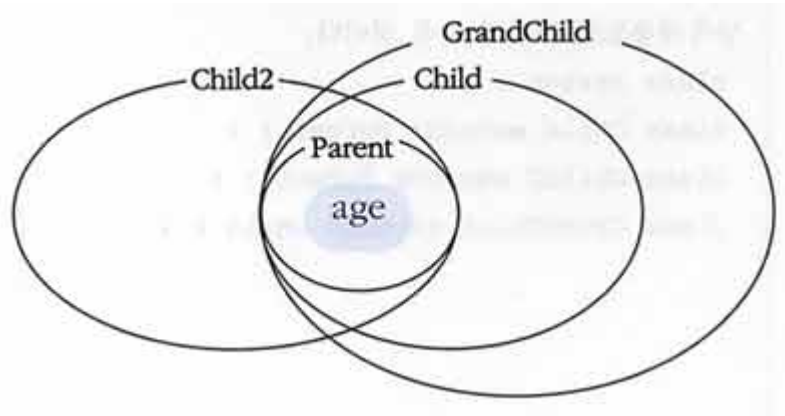
```
class GrandChild extends Child { }
```



자손클래스는 조상클래스의 모든 멤버를 물려받으므로 GrandChild클래스는 Child클래스의 모든 멤버, Child클래스의 조상인 Parent클래스로부터 상속받은 멤버까지, 상속받게 된다. 그 래서 GrandChild클래스는 Child클래스의 자손이면서 Parent클래스의 자손이기도 하다. 좀더 정확히 말하자면, Child클래스는 GrandChild클래스의 직접 조상이고, Parent클래스는 GrandChild클래스의 간접 조상이 된다. 그래서 GrandChild클래스는 Parent클래스와 간접적인 상속관계에 있다고 할 수 있다.

이제 Parent클래스에 전과 같이 정수형 변수인 age를 멤버변수로 추가해 보자.

```
class Parent {  
    int age;  
}  
class Child extends Parent { }  
class Child2 extends Parent { }  
class GrandChild extends Child { }
```



클래스이름	클래스의 멤버
Parent	age
Child	age
Child2	age
GrandChild	age

Parent 클래스는 클래스 Child, Child2, GrandChild의 조상이므로 Parent클래스에 추가된 멤버변수 age는 Parent클래스의 모든 자손에 추가된다. 반대로 Parent클래스에서 멤버변수 age를 제거 한다면, Parent의 자손클래스인 Child, Child2, GrandChild에서도 제거된다. 이처럼 조상클래스만 변경해도 모든 자손클래스에, 자손의 자손 클래스에까지 영향을 미치게 된다.

클래스간의 상속관계를 맺어 주면 자손클래스들의 공통적인 부분은 조상클래스에서 관리하고 자손 클래스는 자신에 정의된 멤버들만 관리하면 되므로 각 클래스의 코드가 적어져서 관리가 쉬워진다.

전체 프로그램을 구성하는 클래스들을 면밀히 설계 분석하여, 클래스간의 상속관계를 적절히 맺어 주는 것이 객체지향 프로그래밍에서 가장 중요한 부분이다.

#### [예제 7-1] CaptionTvTest.java

```
class Tv {
    boolean power;    // 전원상태(on/off)
    int channel;      // 채널
```



```

void power() {    power = !power; }
void channelUp() {    ++channel; }
void channelDown() {    --channel;    }
}

class CaptionTv extends Tv {
    boolean caption;        // 캡션상태(on/off)
    void displayCaption(String text) {
        if (caption) {    // 캡션 상태가 on(true)일 때만 text를 보여 준다.
            System.out.println(text);
        }
    }
}

class CaptionTvTest {
    public static void main(String args[]) {
        CaptionTv ctv = new CaptionTv();
        ctv.channel = 10;        // 조상클래스로부터 상속받은 멤버
        ctv.channelUp();        // 조상클래스로부터 상속받은 멤버
        System.out.println(ctv.channel);
        ctv.displayCaption("Hello, World");
        ctv.caption = true;        // 캡션기능을 켜다.
        ctv.displayCaption("Hello, World");    // 캡션을 화면에 보여 준다.
    }
}

```

#### [실행결과]

11

Hello, World

Tv클래스로부터 상속받고 기능을 추가하여 CaptionTv클래스를 작성하였다. 멤버변수 caption은 캡션기능의 상태를 저장하기 위한 boolean형 변수이고, displayCaption(String text)는 매개변수로 넘겨받은 문자열 text를 캡션이 켜져 있는 경우(caption의 값이 true인 경우)에만 화면에 출력한다.

자손클래스의 인스턴스를 생성하면 조상클래스의 멤버도 함께 생성되기 때문에 따로 조상클래스의 인스턴스를 생성하지 않고도 조상클래스의 멤버들을 사용할 수 있다.

## 1.2 클래스간의 관계 - 포함관계

지금까지 상속을 통해 클래스간에 관계를 맺어 주고 클래스를 재사용하는 방법에 대해서 알아보았다. 상 속이외에도 클래스를 재사용하는 또 다른 방법이 있는데, 그 것은 클래스간에 '포함 (Composite)' 관계를 맺어 주는 것이다. 클래스간의 포함관계를 맺어 주는 것은 한 클래스의 멤버변수로 선언하여 다른 클래스를 포함시키는 것을 뜻한다.

원(Circle)을 표현하기 위한 Circle이라는 클래스를 다음과 같이 작성하였다고 가정하자.

```
class Circle {  
    int x;        // 원점의 x좌표  
    int y;        // 원점의 y좌표  
    int r;        // 반지름(radius)  
}
```

그리고 좌표상의 한 점을 다루기 위해 Point클래스가 다음과 같이 작성되어 있다고 가정하자.

```
class Point {  
    int x;        // x좌표  
    int y;        // y좌표  
}
```

Point클래스를 재사용해서 Circle클래스를 작성한다면 다음과 같이 할 수 있을 것이다.

```
class Circle {  
    Point c = new Point();    // 원점
```

```
int r;
}
```

이 와 같이 한 클래스를 작성하는 데 다른 클래스를 멤버변수로 정의하여 포함시키는 것은 좋은 생각이다. 하나의 거대한 클래스를 작성하는 것보다 단위별로 여러 개의 클래스를 작성한 다음, 이 단위 클래스들을 포함관계로 재사용하면 보다 간결하고 손쉽게 클래스를 작성할 수 있을 것이다. 또한 작성된 단위 클래스들은 다른 클래스를 작성하는데 재사용될 수 있을 것이다.

```
class Car {
    Engine e = new Engine();    // 엔진
    Door[] d = new Door[4];     // 문, 문의 개수를 넷으로 가정하고 배열로 처리했다.
    //...
}
```

위 와 같은 Car클래스를 작성할 때, Car클래스의 단위구성요소인 Engine, Door와 같은 클래스를 미리 작성해 놓고 이 들을 Car클래스의 멤버변수로 선언하여 포함관계를 맺어 주면, 클래스를 작성하는 것도 쉽고 코드도 간결해서 이해하기도 쉽다. 그리고 단위클래스별로 코드를 작게 나누어서 작성되어 있기 때문에 코드를 관리하는데도 수월하다.

### 1.3 클래스간의 관계 결정하기

클래스를 작성하는데 있어서 상속관계를 맺어 줄 것인지 포함관계를 맺어 줄 것인지 결정하는 것은 때때로 혼동스러울 수 있다.

전에 예를 든 Circle클래스의 경우, Point클래스를 포함시키는 대신 상속관계를 맺어 주었다면 다음과 같을 것이다.

```
class Circle extends Point {
```

```
int r;  
}
```

두 경우를 비교해 보면 Circle클래스를 작성하는데 있어서 Point클래스를 포함시키거나 상속 받도록 하는 것은 결과적으로 별 차이가 없어 보인다.  
그럴 때는 '~은 ~이다(is-a)'와 '~은 ~을 가지고 있다(has-a)'를 넣어서 문장을 만들어 보면 클래스들간의 관계가 보다 명확해 진다.

원(Circle)은 점(Point)이다. - Circle is a Point.  
원(Circle)은 점(Point)를 가지고 있다. - Circle has a Point.

원은 원점(Point)와 반지름으로 구성되므로 위의 두 문장을 비교해 보면 첫 번째 문장보다 두 번째 문장이 더 옳다는 것을 알 수 있을 것이다.  
이 처럼 클래스를 가지고 문장을 만들었을 때 '~은 ~이다.'라는 문장이 성립한다면, 서로 상속 관계를 맺어 주고, '~은 ~을 가지고 있다.'는 문장이 성립한다면 포함관계를 맺어 주면 된다.  
그래서 Circle클래스와 Point클래스간의 관계는 상속관계 보다는 포함관계를 맺어 주는 것이 더 옳다.

몇 가지 더 예를 들면, Car클래스와 SportsCar클래스는 'SportsCar는 Car이다.'와 같이 문장을 만드는 것이 더 옳기 때문에 이 두 클래스는 Car클래스를 조상으로 하는 상속관계를 맺어 주어야 한다.

Card클래스와 Deck클래스는 'Deck는 Card를 가지고 있다.'와 같이 문장을 만드는 것이 더 옳기 때문에 Deck클래스에 Card클래스를 포함시켜야 한다.

[참고]Deck은 카드 한 벌을 뜻한다.

상속관계 - '~은 ~이다.(is-a)'  
포함관계 - '~은 ~을 가지고 있다.(has-a)'

[참고]프로그램에 사용되는 모든 클래스들을 분석하여 가능한 많은 관계를 맺어 주도록 노력하

여 코드의 재사용성을 높여야 한다.

#### [예제7-2] DrawShape.java

```
import java.awt.Frame;
import java.awt.Graphics;

class DrawShape extends Frame {
    public static void main(String[] args)
    {
        DrawShape win = new DrawShape("도형그리기");
    }

    public void paint(Graphics g) {
        Point[] p = { new Point(100, 100), new Point(140, 50), new Point(200,
100)};
        Triangle t = new Triangle(p);
        Circle c = new Circle(new Point(150, 150), 50);

        // 원을 그린다.
        g.drawOval(c.center.x, c.center.y, c.r, c.r);

        // 직선 3개로 삼각형을 그린다.
        g.drawLine(t.p[0].x, t.p[0].y, t.p[1].x, t.p[1].y);
        g.drawLine(t.p[1].x, t.p[1].y, t.p[2].x, t.p[2].y);
        g.drawLine(t.p[2].x, t.p[2].y, t.p[0].x, t.p[0].y);
    }

    DrawShape(String title) {
        super(title);
        setSize(300, 300);
        setVisible(true);
    }
}
```

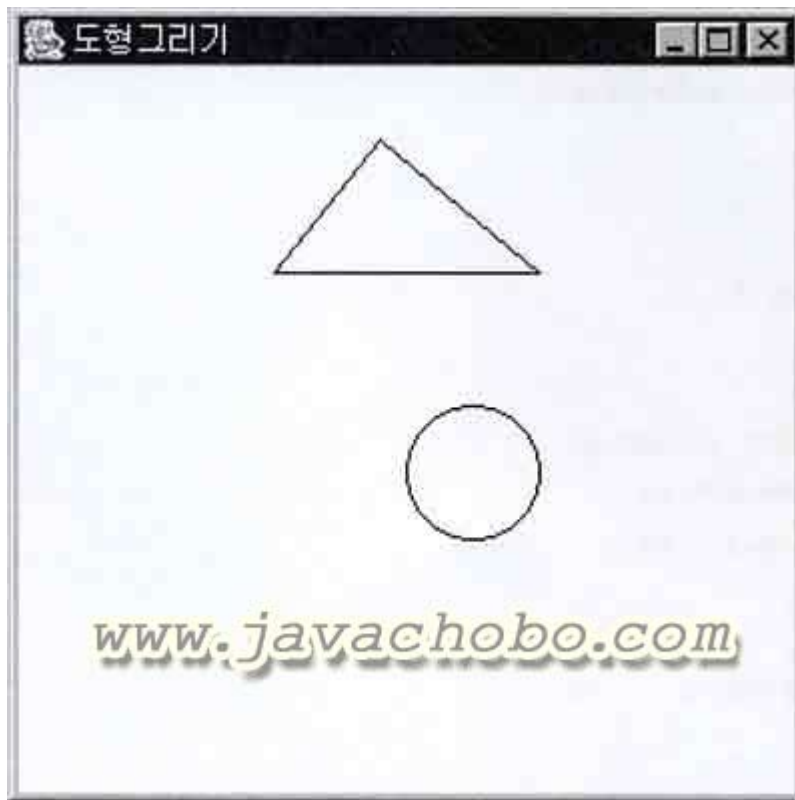
```

class Point {
    int x;
    int y;
    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    Point() {
        this(0,0);
    }
}

class Circle {
    Point center;    // 원의 원점좌표
    int r;           // 반지름
    Circle() {
        this(new Point(0, 0), 100);
    }
    Circle(Point center, int r) {
        this.center = center;
        this.r = r;
    }
}

class Triangle {
    Point[] p = new Point[3]; // 3개의 Point인스턴스를 담을 배열을 생성한다.
    Triangle(Point[] p) {
        this.p = p;
    }
    Triangle(Point p1, Point p2, Point p3) {
        p[0] = p1;
        p[1] = p2;
        p[2] = p3;
    }
}

```



[그림7-1]예제7-2의 실행결과

Circle클래스와 Triangle클래스를 이용하여 원과 삼각형을 그려보았다. 자바의 AWT를 이용해서 새로운 윈도우를 만들고 여기에 도형을 그렸다. DrawShape클래스의 인스턴스 win을 생성하면 자동으로 paint(Graphics g)메서드가 호출되어 화면에 도형을 그린다.

아직 AWT를 이용한 윈도우프로그래밍을 배우지 않았으니, Circle클래스와 Triangle클래스를 작성하고 사용하는 부분만 이해해도 좋다.

#### [예제7-3] DeckTest.java

```
class DeckTest {
    public static void main(String args[]) {
        Deck d = new Deck();    // 카드 한 벌(Deck)을 만든다.
        Card c = d.pick(0);    // 섞기 전에 제일 위의 카드를 뽑는다.
        System.out.println(c);
        d.shuffle();            // 카드를 섞는다.
        c = d.pick(0);          // 섞은 후에 제일 위의 카드를 뽑는다.
        System.out.println(c);
    }
}
```

```

}

// Deck클래스
class Deck {
    final int CARD_NUM = 52;    // 카드의 개수
    Card c[] = new Card[CARD_NUM];

    Deck () {    // Deck의 카드를 초기화한다.
        int i=0;

        for(int k=Card.KIND_MAX; k > 0; k--) {
            for(int n=1; n < Card.NUM_MAX + 1 ; n++) {
                c[i++] = new Card(k, n);
            }
        }
    }

    Card pick(int index) {    // 지정된 위치(index)에 있는 카드 하나를 선택한다.
        return c[index%CARD_NUM];
    }

    Card pick() {    // Deck에서 카드 하나를 선택한다.
        int index = (int)(Math.random() * CARD_NUM);
        return pick(index);
    }

    void shuffle() {    // 카드의 순서를 섞는다.
        for(int n=0; n < 1000; n++) {
            int i = (int)(Math.random() * CARD_NUM);
            Card temp = c[0];    // 첫 번째 카드와 임의로 선택된 카드를 서로 바꾼다.
            c[0] = c[i];
            c[i] = temp;
        }
    }
}

```



```

// Card클래스
class Card {
    static final int KIND_MAX = 4;        // 카드 무늬의 수
    static final int NUM_MAX = 13;       // 무늬별 카드 수

    static final int SPADE = 4;
    static final int DIAMOND = 3;
    static final int HEART = 2;
    static final int CLOVER = 1;

    int kind;
    int number;

    Card() {
        this(SPADE, 1);
    }

    Card(int kind, int number) {
        this.kind = kind;
        this.number = number;
    }

    public String toString() {
        String kind="";
        String number="";

        switch(this.kind) {
            case 4 :
                kind = "SPADE";
                break;
            case 3 :
                kind = "DIAMOND";
                break;
            case 2 :

```

```

        kind = "HEART";
        break;
    case 1 :
        kind = "CLOVER";
        break;
    default :
    }

    switch(this.number) {
        case 13 :
            number = "K";
            break;
        case 12 :
            number = "Q";
            break;
        case 11 :
            number = "J";
            break;
        default :
            number = this.number + "";
    }

    return "kind : " + kind + ", number : " + number;
}
}

```

#### [실행결과]

kind : SPADE, number : 1

kind : HEART, number : 7

Deck클래스를 작성하는데 Card클래스를 재사용하여 포함관계로 작성하였다. 카드 한 벌 (Deck)는 모두 52장의 카드로 이루어져 있으므로 Card클래스를 크기가 52인 배열로 처리하였다.

shuffle()은 카드 한 벌의 첫 번째 장과 임의로 선택한 위치에 있는 카드의 위치를 서로 바꾸는 방식으로 카드를 섞는다. random()을 사용했기 때문에 매 실행 시 마다 결과가 다르게 나타날 것이다.

## 1.4 단일상속(Single Inheritance)

C++에서는 여러 클래스로부터 상속받는 다중상속(Multiple Inheritance)을 허용하지만, 자바에서는 단일 상속만을 허용하기 때문에 하나 이상의 클래스로부터 상속을 받을 수 없다. 예를 들면, TV클래스와 VCR클래스가 있을 때, 이 두 클래스로부터 상속을 받는 TVCR클래스를 작성할 수 없다. 그래서 TVCR클래스는 조상클래스로 TV클래스와 VCR클래스 중 하나만 선택해야한다.

```
class TVCR extends TV, VCR {    // 이와 같은 표현을 허용하지 않는다.  
    //...  
}
```

다 중상속을 허용하면 여러 클래스로부터 상속받을 수 있기 때문에 복합적인 기능을 가진 클래스를 쉽게 작성할 수 있다는 장점이 있지만, 클래스간의 관계가 매우 복잡해진다는 것과 서로 다른 클래스로부터 상속받은 멤버들간의 이름이 같은 경우 구별할 수 있는 방법이 없다는 단점을 가지고 있다.

만일 다중상속을 허용해서 TVCR클래스가 TV클래스와 VCR클래스를 모두 조상으로 하여 두 클래스의 멤버들을 상속받는다고 가정해 보자.

TV클래스에도 power()라는 메서드가 있고, VCR클래스에도 power()라는 메서드가 있을 때 자손인 TVCR클래스는 어느 조상클래스의 power()메서드를 상속받게 되는 것일까?

둘 다 상속받게 된다면, TVCR클래스 내에서 선언부(이름과 매개변수)만 같고 서로 다른 내용의 두 메서드를 어떻게 구별할 것인가?

static메서드라면, 메서드 이름 앞에 클래스의 이름을 붙여서 구별할 수 있다지만, 인스턴스메서드의 경우 선언부가 같은 두 메서드를 구별할 수 있는 방법은 없다.

이것을 해결하는 방법은 조상클래스의 메서드의 이름인 매개변수를 바꾸는 방법 밖에 없다. 이렇게 하면 그 조상클래스의 power()메서드를 사용하던 모든 클래스들도 변경을 해야 하므로 그리 간단한 문제가 아니다.

자바에서는 다중상속의 이러한 문제점을 해결하기 위해 다중상속의 장점을 포기하고 단일상속만을 허용한다. 대신 앞으로 배우게 될 인터페이스(interface)를 이용해서 보완된 형태의 다중상속을 구현할 수 있도록 하고 있다.

단일 상속이 하나의 조상클래스만을 가질 수 있기 때문에 다중상속에 비해 불편한 점도 있겠지만, 클래스간의 관계가 보다 명확해지고 코드를 더욱 신뢰성있게 만들어 준다는 점에서는 다중상속보다 유리하다.

#### [예제7-4] TVCR.java

```
class Tv {
    boolean power;    // 전원상태(on/off)
    int channel;      // 채널

    void power() {    power = !power; }
    void channelUp() { ++channel; }
    void channelDown() { --channel; }
}

class VCR {
    boolean power;    // 전원상태(on/off)
    int counter = 0;

    void power() {    power = !power; }
    void play() { /* 내용생략*/ }
    void stop() { /* 내용생략*/ }
    void rew() { /* 내용생략*/ }
    void ff() { /* 내용생략*/ }
}

class TVCR extends Tv {
    VCR vcr = new VCR();
    int counter = vcr.counter;

    void play() {
        vcr.play();
    }
    void stop() {
        vcr.stop();
    }
}
```

```

void rew() {
    vcr.rew();
}
void ff() {
    vcr.ff();
}
}

```

자 바는 다중상속을 허용하지 않으므로 Tv클래스를 조상으로 하고, VCR클래스는 TVCR클래스에 포함시켰다. 그리고 TVCR클래스에 VCR클래스의 메서드와 일치하는 선언부를 가진 메서드를 선언하고 내용은 VCR클래스의 것을 호출해서 사용하도록 했다. 외부적으로는 TVCR클래스의 인스턴스를 사용하는 것처럼 보이지만 내부적으로는 VCR클래스의 인스턴스를 생성해서 사용하는 것이다.

이렇게 함으로써 VCR클래스의 메서드의 내용이 변경되더라도 TVCR클래스의 메서드들 역시 변경된 내용이 적용되는 결과를 얻을 수 있을 것이다.

#### 1.4 Object클래스 - 모든 클래스의 조상

Object클래스는 모든 클래스 상속계층도의 제일 위에 위치하는 조상클래스이다. 다른 클래스로부터 상속 받지 않는 모든 클래스들은 자동적으로 Object클래스로부터 상속받게 함으로써 이 것을 가능하게 한다.

만일 다음과 같이 다른 클래스로부터 상속을 받지 않는 Tv클래스를 정의하였다고 하자.

```

class Tv {
    // ...
}

```

위의 코드를 컴파일 하면 컴파일러는 위의 코드를 다음과 같이 자동적으로 'extends Object'를 추가하여 Tv클래스가 Object클래스로부터 상속받도록 한다.

```
class Tv extends Object {
    // ...
}
```

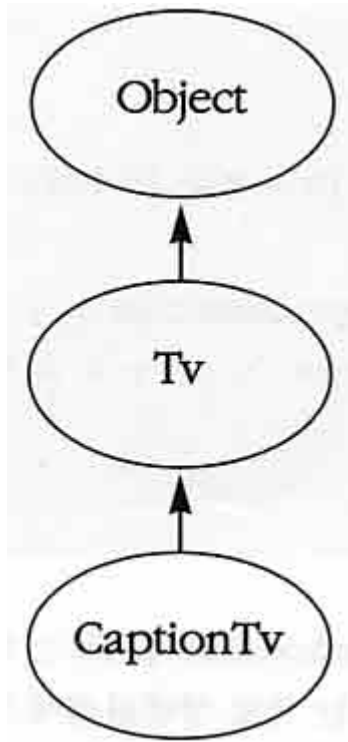
이렇게 함으로써 Object클래스가 모든 클래스의 조상이 되도록 한다. 만일 다른 클래스로부터 상속을 받는다고 하더라도 상속계층도를 따라 조상클래스, 조상클래스의 조상클래스를 찾아 올라가다 보면 결국 마지막 최상위 조상은 Object클래스일 것이다.

[참고]이미 어떤 클래스로부터 상속받도록 작성된 클래스에 대해서는 컴파일러가 'extends Object'를 추가하지 않는다.

```
class Tv {
    //...
}

class CaptionTv extends Tv {
    // ...
}
```

위와 같이 Tv클래스가 있고, Tv클래스를 상속받는 CaptionTv가 있을 때 상속계층도는 다음과 같다.



[참고]상속계층도를 단순화하기 위해서 Object클래스를 생략하는 경우가 많다.

이처럼 모든 상속계층도의 최상위에는 Object클래스가 위치한다. 그래서 자바의 모든 클래스들은 Object클래스의 멤버들을 상속 받기 때문에 Object클래스에 정의된 멤버들을 사용할 수 있다.

그동안 toString()이나 equals(Object o)와 같은 메서드를 따로 정의하지 않고도 사용할 수 있었던 이유는 이 메서드들이 Object클래스에 정의된 것들이기 때문이다.

Object클래스에는 toString(), equals()와 같은 모든 인스턴스가 가져야 할 기본적인 8개의 메서드를 정의해 놓고 있으며 이에 대해서는 후에 자세히 학습하게 될 것이다.

# [Java의 정석]제7장 객체지향개념 2 - 2. 오버라이딩(Overriding)

자바의정석

2012/12/22 17:10

<http://blog.naver.com/gphic/50157740090>

## 2. 오버라이딩(Overriding)

### 2.1 오버라이딩이란?

조상클래스로부터 상속받은 메서드의 내용을 변경하는 것을 오버라이딩이라고 한다. 상속받은 메서드를 그대로 사용하기도 하지만, 자손클래스 자신에 맞게 변경해야하는 경우가 많다. 이럴 때 조상의 메서드를 오버라이딩한다.

[참고]override의 사전적 의미는 '~위에 덮어쓰다(overwrite).' 또는 '~에 우선하다.'이다.

2차원 좌표계의 한 점을 표현하기 위한 Point클래스가 있을 때, 이를 조상으로 하는 Point3D 클래스, 3차원 좌표계의 한 점을 표현하기 위한 클래스를 다음과 같이 새로 작성하였다고 하자.

```
class Point {
    int x;
    int y;

    String getLocation() {
        return "x : " + x + ", y : " + y;
    }
}

class Point3D extends Point {
    int z;

    String getLocation() {    // 오버라이딩
        return "x : " + x + ", y : " + y + ", z : " + z;
    }
}
```



```
}  
}
```

Point 클래스의 getLocation메서드는 한 점의 x, y 좌표를 문자열로 반환하도록 작성되었다. 이 두 클래스는 서로 상속관계에 있으므로 Point3D클래스는 Point클래스로부터 getLocation 메서드를 상속받지만, Point3D클래스는 3차원 좌표계의 한 점을 표현하기 위한 것이므로 조상인 Point클래스로부터 상속받은 getLocation메서드는 Point3D클래스에 맞지 않는다. 그래서 이 메서드를 Point3D클래스 자신에 맞게 z축의 좌표값도 포함하여 반환하도록 오버라이딩 하였다.

Point 클래스를 사용하던 사람들은 새로 작성된 Point3D클래스가 Point클래스의 자손이므로 Point3D클래스의 인스턴스에 대해서 getLocation메서드를 호출하면 Point클래스의 getLocation메서드가 그랬듯이 점의 좌표를 문자열로 얻을 수 있을 것이라고 기대할 것이다. 그렇기 때문에 새로운 메서드를 제공하는 것보다 오버라이딩을 하는 것이 바른 선택이다.

## 2.2 오버라이딩의 조건

오버라이딩은 메서드의 내용만을 새로 작성하는 것이므로 메서드의 선언부는 조상의 것과 완전히 일치해야한다.

그래서 오버라이딩이 성립하기 위해서는 다음과 같은 조건을 만족해야한다.

- 자손클래스에서 오버라이딩하는 메서드는 조상클래스의 메서드와
  - 이름이 같아야 한다.
  - 매개변수가 같아야 한다.
  - 리턴타입이 같아야 한다.

한마디로 요약하면 선언부가 서로 일치해야한다는 것이다. 다만 접근제어자(Access Modifier)와 예외(Exception)는 제한된 조건 하에서 다르게 변경할 수 있다.

1. 접근제어자는 조상클래스의 메서드보다 좁은 범위로 변경 할 수 없다.

- 만일 조상클래스에 정의된 메서드의 접근제어자가 protected라면, 이를 오버라이딩하는 자손클래스의 메서드는 접근제어자가 protected나 public이어야 한다.

[참고]대부분의 경우 같은 범위의 접근제어자를 사용한다. 접근제어자의 접근범위를 넓은 것으로 나열하면 public, protected, default, private이다.

2.조상클래스의 메서드보다 많은 수의 예외를 선언할 수 없다.

- 아래의 코드를 보면 Child클래스의 parentMethod에 선언된 예외의 개수가 조상인 Parent 클래스의 parentMethod에 선언된 예외의 개수보다 적으므로 바르게 오버라이딩 되었다.

```
Class Parent {  
    void parentMethod() throws IOException, SQLException {  
        //..  
    }  
}  
  
Class Child extends Parent {  
    void parentMethod() throws IOException {  
        //..  
    }  
    //..  
}
```

여기서 주의해야할 점은 단순히 선언된 예외의 개수의 문제가 아니라는 것이다.

```
Class Child extends Parent {  
    void parentMethod() throws Exception {  
        //..  
    }  
    //..  
}
```

만일 위와 같이 오버라이딩을 하였다면, 분명히 조상클래스에 정의된 메서드보다 적은 개수의 예외를 선언한 것처럼 보이지만 Exception은 모든 예외의 최고 조상이므로 가장 많은 개수의 예외를 던질 수 있도록 선언한 것이다.

그래서 예외의 개수는 적거나 같아야 한다는 조건을 만족시키지 못하는 잘못된 오버라이딩인 것이다.

### 2.3 오버로딩 vs. 오버라이딩

오버로딩과 오버라이딩은 서로 혼동하기 쉽지만 사실 그 차이는 명백하다. 오버로딩은 기존에 없는 새로운 메서드를 추가하는 것이고, 오버라이딩은 조상으로부터 상속받은 메서드의 내용을 변경하는 것이다.

오버로딩(Overloading) - 기존에 없는 새로운 메서드를 정의하는 것(new)  
오버라이딩(Overriding) - 상속받은 메서드의 내용을 변경하는 것(change, modify)

아래의 코드를 보고 오버로딩과 오버라이딩을 구별할 수 있어야 한다.

```
class Parent {  
    void parentMethod() {}  
}  
  
class Child extends Parent {  
    void parentMethod() {} // 오버라이딩  
    void parentMethod(int i) {} // 오버로딩  
  
    void childMethod() {}  
    void childMethod(int i) {} // 오버로딩  
    void childMethod() {}    // 에러!!! 중복정의 되었음.(already defined in Child)
```

```
}
```

## 2.4 super

super는 자손클래스에서 조상클래스로부터 상속받은 멤버를 참조하는데 사용되는 참조변수이다. 멤버변수와 지역변수의 이름이 같을 때 this를 사용해서 구별했듯이 상속받은 멤버와 자신의 클래스에 정의된 멤버의 이름이 같을 때는 super를 사용해서 구별할 수 있다.

조상클래스로부터 상속받은 멤버도 자손클래스 자신의 멤버이므로 super대신 this를 사용할 수 있다. 그래도 조상클래스의 멤버와 자손클래스의 멤버가 중복 정의되어 서로 구별해야하는 경우에만 super를 사용하는 것이 좋다.

조상의 멤버와 자신의 멤버를 구별하는데 사용된다는 점을 제외하고는 super와 this는 근본적으로 같다. 모든 인스턴스메서드에는 자신이 속한 인스턴스의 주소가 지역변수로 저장되는데, 이것이 참조변수인 this와 super의 값이 된다.

static메서드(클래스메서드)는 인스턴스와 관련이 없다. 그래서 this와 마찬가지로 super역시 static메서드에서는 사용할 수 없고 인스턴스메서드에서만 사용할 수 있다.

### [예제 7-5] SuperTest.java

```
class SuperTest {
    public static void main(String args[]) {
        Child c = new Child();
        c.method();
    }
}

class Parent {
    int x=10;
}

class Child extends Parent {
    void method() {
        System.out.println("x=" + x);
    }
}
```

```

        System.out.println("this.x=" + this.x);
        System.out.println("super.x="+ super.x);
    }
}

```

#### [실행결과]

```

x=10
this.x=10
super.x=10

```

이 경우 x, this.x, super.x 모두 같은 변수를 의미하므로 모두 같은 값이 출력되었다.

#### [예제 7-6] SuperTest2.java

```

class SuperTest2 {
    public static void main(String args[]) {
        Child c = new Child();
        c.method();
    }
}

class Parent {
    int x=10;
}

class Child extends Parent {
    int x=20;
    void method() {
        System.out.println("x=" + x);
        System.out.println("this.x=" + this.x);
        System.out.println("super.x="+ super.x);
    }
}

```

#### [실행결과]

```
x=20
this.x=20
super.x=10
```

이전 예제와는 달리 같은 이름의 멤버변수가 조상클래스인 Parent에도 있고 자손클래스인 Child클래스에도 있을 때는 super.x와 this.x는 서로 다른 값을 참조하게 된다. super.x는 조상클래스로부터 상속받은 멤버변수 x를 뜻하며, this.x는 자손클래스에 선언된 멤버변수를 뜻한다.

이처럼 조상클래스에 선언된 멤버변수와 같은 이름의 멤버변수를 자손클래스에서 중복해서 정의하는 것이 가능하며 참조변수 super를 이용해서 서로 구별할 수 있다.

변수만이 아니라 메서드 역시 super를 써서 호출할 수 있다. 특히 조상클래스의 메서드를 자손클래스에서 오버라이딩한 경우에 super를 사용한다.

```
class Point {
    int x;
    int y;

    String getLocation() {
        return "x : " + x + ", y : " + y;
    }
}

class Point3D extends Point {
    int z;
    String getLocation() {    // 오버라이딩
        // return "x : " + x + ", y : " + y + ", z : " + z;
        return super.getLocation() + ", z : " + z;
    }
}
```

getLocation 메서드를 오버라이딩할 때 조상클래스의 getLocation메서드를 호출해서 포함시켰다. 조상클래스의 메서드의 내용에 추가적으로 작업을 덧붙이는 경우라면 이처럼 super를

사용해서 조상클래스의 메서드를 포함시키는 것이 좋다. 후에 조상클래스의 메서드가 변경되더라도 변경된 내용이 자손클래스의 메서드에 자동적으로 반영될 것이기 때문이다.

## 2.5 super() - 생성자

this()와 마찬가지로 super() 역시 생성자이다. this()는 같은 클래스의 다른 생성자를 호출하는 데 사용되지만, super()는 조상클래스의 생성자를 호출하는데 사용된다.

자손클래스의 인스턴스를 생성하면, 자손의 멤버와 조상의 멤버가 모두 합쳐진 하나의 인스턴스가 생성된다. 그래서 자손클래스의 인스턴스가 조상클래스의 멤버들을 사용할 수 있는 것이다.

이 때 조상클래스 멤버의 생성과 초기화 작업이 수행되어야 하기 때문에 자손클래스의 생성자에서 조상클래스의 생성자가 호출되어야 한다.

생성자의 첫 줄에서 조상클래스의 생성자를 호출해야하는 이유는 자손클래스의 멤버가 조상클래스의 멤버를 사용할 수도 있으므로 조상의 멤버들이 먼저 초기화되어 있어야 하기 때문이다.

인스턴스를 생성할 때는 다음의 2가지를 선택해야한다.

1. 클래스 - 어떤 클래스의 인스턴스를 생성할 것인가?
2. 생성자 - 선택한 클래스의 어떤 생성자를 이용해서 인스턴스를 생성할 것인가?

이처럼 인스턴스를 생성할 때는 클래스를 선택하는 것만큼 생성자를 선택하는 것도 중요한 일이다.

이와 같은 조상클래스 생성자의 호출은 클래스의 상속관계를 거슬러 올라가면서 계속 반복된다. 마지막으로 모든 클래스의 최고조상인 Object클래스의 생성자인 Object()까지 가서야 끝이 난다.

그래서 Object클래스를 제외한 모든 클래스의 생성자는 첫 줄에 반드시 자신의 다른 생성자 또는 조상의 생성자를 호출해야한다. 그렇지 않으면 컴파일러는 생성자의 첫 줄에 super();를 자동적으로 추가할 것이다.

[예제 7-7] PointTest.java

```

class PointTest {
    public static void main(String args[]) {
        Point3D p3 = new Point3D(1,2,3);
    }
}

class Point {
    int x;
    int y;

    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    String getLocation() {
        return "x :" + x + ", y :" + y;
    }
}

class Point3D extends Point {
    int z;
    Point3D(int x, int y, int z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }
    String getLocation() {    // 오버라이딩
        return "x :" + x + ", y :" + y + ", z :" + z;
    }
}

```

#### [컴파일결과]

```

C:\wj2sdk1.4.1\work>javac PointTest.java
PointTest.java:22: cannot resolve symbol
symbol : constructor Point ()

```



```
location: class Point
Point3D(int x, int y, int z) {
^
1 error
```

이 예제를 컴파일하면 위와 같은 컴파일에러가 발생할 것이다. Point3D클래스의 생성자에서 조상클래스의 생성자인 Point()를 찾을 수 없다는 내용이다.

Point3D클래스의 생성자의 첫 줄이 생성자(조상의 것이든 자신의 것이든)를 호출하는 문장이 아니기 때문에 컴파일러는 다음과 같이 자동적으로 'super();'를 Point3D클래스의 생성자의 첫 줄에 넣어 준다.

```
Point3D(int x, int y, int z) {
    super();
    this.x = x;
    this.y = y;
    this.z = z;
}
```

그래서 Point3D클래스의 인스턴스를 생성하면, 생성자 Point3D(int x, int y, int x)가 호출되면서 첫 문장인 super();를 수행하게 된다. super()는 Point3D클래스의 조상인 Point클래스의 기본 생성자인 Point()를 뜻하므로 Point()가 호출된다.

그러나 Point클래스에 생성자 Point()가 정의되어 있지 않기 때문에 위와 같은 컴파일 에러가 발생한 것이다. 이 에러를 수정하려면, Point클래스에 생성자 Point()를 추가해주던가, 생성자 Point3D(int x, int y, int z)의 첫 줄에서 Point(int x, int y)를 호출하도록 변경하면 된다.

[참고]생성자가 정의되어 있는 클래스에는 컴파일러가 기본 생성자를 자동적으로 추가하지 않는다.

```
Point3D(int x, int y, int z) {
    super(x, y);    // 조상클래스의 생성자 Point(int x, int y)를 호출한다.
    this.z = z;
}
```

위와 같이 변경하면 된다. 문제없이 컴파일 될 것이다. 조상클래스의 멤버변수는 이처럼 조상의 생성자를 이용해서 초기화 하도록 해야 하는 것이다.

[예제7-8] PointTest2.java

```
class PointTest2 {
    public static void main(String argsp[]) {
        Point3D p3 = new Point3D();
        System.out.println("p3.x=" + p3.x);
        System.out.println("p3.y=" + p3.y);
        System.out.println("p3.z=" + p3.z);
    }
}

class Point {
    int x=10;
    int y=20;

    Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

class Point3D extends Point {
    int z=30;

    Point3D() {
        this(100, 200, 300);    // Point3D(int x, int y, int z)를 호출한다.
    }

    Point3D(int x, int y, int z) {
        super(x, y);            // Point(int x, int y)를 호출한다.
        this.z = z;
    }
}
```

```
}  
}
```

#### [실행결과]

```
p3.x=100
```

```
p3.y=200
```

```
p3.z=300
```

Point3D클래스의 인스턴스를 생성하면, 조상인 Point클래스의 인스턴스도 생성되므로 Point클래스의 생성자도 호출되고, Point클래스의 조상인 Object클래스의 생성자까지 호출된다.

이처럼 어떤 클래스의 인스턴스를 생성하면, 클래스의 상속관계를 최고조상인 Object클래스까지 거슬러 올라가면서 조상클래스의 인스턴스를 생성한다.

컴파일러는 Point클래스의 생성자인 Point(int x, int y)의 첫 줄에 super();를 자동적으로 삽입할 것이다. 이 super()는 Object클래스의 생성자인 Object()를 뜻한다.

```
Point(int x, int y) {  
    super();  
    this.x = x;  
    this.y = y;  
}
```

생성자의 호출은 계속 이어져서 결국 Object클래스의 생성자인 Object()까지 호출되어야 끝나는 것이다.

# [Java의 정석]제7장 객체지향개념 2 - 3. package와 import

자바의정석

2012/12/22 17:10

<http://blog.naver.com/gphic/50157740114>

## 3. package와 import

### 3.1 패키지(package)

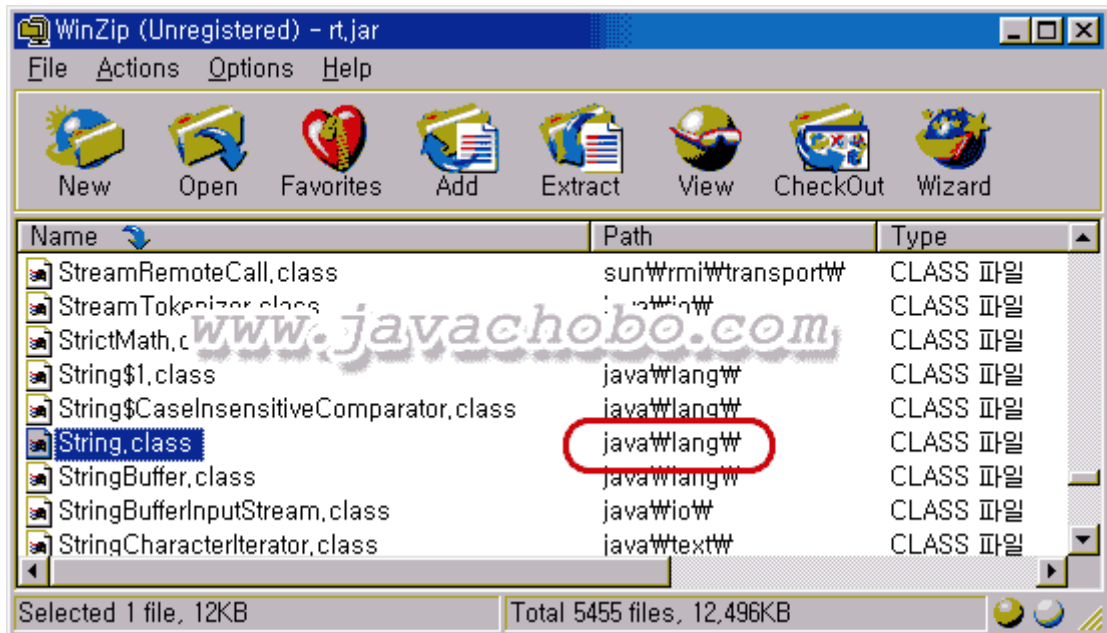
패키지란, 클래스의 묶음이다. 패키지에는 클래스 또는 인터페이스를 포함 시킬 수 있으며, 서로 관련된 클래스들끼리 그룹 단위로 나누어 놓음으로써 클래스를 효율적으로 관리할 수 있다.

또한 같은 이름의 클래스 일지라도 서로 다른 패키지에 존재하는 것이 가능하므로, 자신만의 패키지 체계를 유지함으로써 다른 개발자가 개발한 클래스 라이브러리의 클래스와 이름이 충돌하는 것을 피할 수 있다.

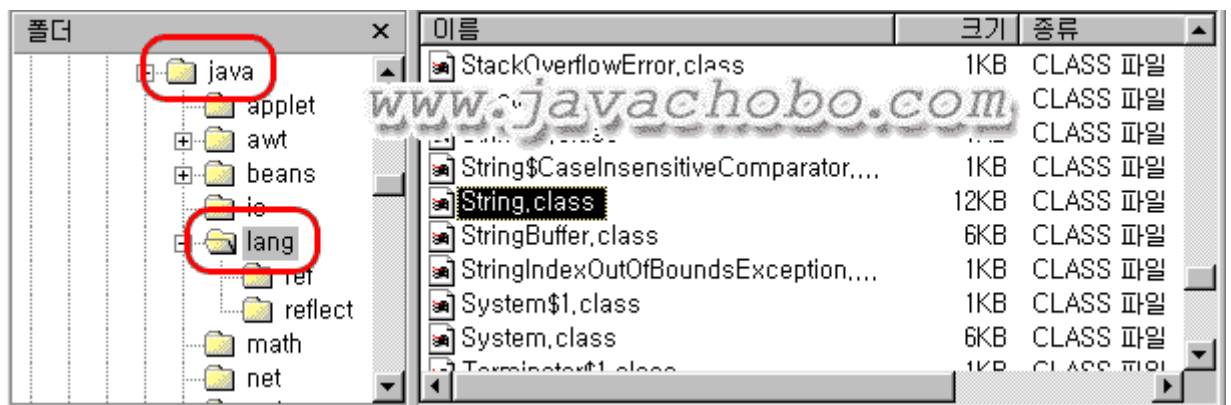
지금까지는 단순히 클래스명으로만 클래스를 구분 했지만 클래스의 실제 이름(full name)은 패키지명을 포함한 것이다. 예를 들면, String클래스의 패키지명을 포함한 이름은 java.lang.String이다. 즉, java.lang패키지에 속한 String클래스라는 의미이다. 그래서 같은 이름의 클래스일 지라도 서로 다른 패키지에 속하면 패키지명으로 구별이 가능하다.

클래스가 물리적으로 하나의 클래스파일(.class)인 것과 같이 패키지는 물리적으로 하나의 디렉토리이다. 그래서 어떤 패키지에 속한 클래스는 해당 디렉토리에 존재하는 클래스파일(.class)이어야 한다.

예를 들어, java.lang.String클래스는 물리적으로 디렉토리 java의 서브디렉토리인 lang에 속한 String.class파일이다. 그리고 우리가 자주 사용하는 System클래스 역시 java.lang패키지에 속하므로 lang디렉토리에 포함되어 있다.



String 클래스는 rt.jar파일에 압축되어 있으며 아래의 그림은 압축을 풀기 전과 후의 그림이다. 클래스와 관련 파일들을 압축한 것이 jar파일(\*.jar)이며, jar파일은 jar.exe이외에 알집이나 winzip으로 압축을 풀 수 있다.



디렉토리가 하위디렉토리를 가질 수 있는 것처럼, 패키지도 다른 패키지를 포함할 수 있으며 점(.)으로 구분한다. 예를 들면 java.lang패키지에서 lang패키지는 java패키지의 하위패키지이다.

- 하나의 소스파일에는 첫 번째 문장으로 단 한번의 패키지 선언을 허용한다.
- 모든 클래스는 반드시 하나의 패키지에 속해야한다.
- 패키지는 점(.)을 구분자로 하여 계층구조로 구성할 수 있다.
- 패키지는 물리적으로 클래스 파일(.class)을 포함하는 하나의 디렉토리이다.

### 3.2 패키지의 선언

패키지를 선언하는 것은 아주 간단하다. 클래스나 인터페이스의 소스파일(.java)에서 다음과 같이 한 줄만 적어주면 된다.

```
package 패키지명;
```

위와 같은 패키지 선언문은 반드시 소스파일에서 주석과 공백을 제외한 첫 번째 문장이어야 하며, 하나의 소스파일에 단 한번만 선언될 수 있다. 해당 소스파일에 포함된 모든 클래스나 인터페이스는 선언된 패키지에 속하게 된다.

패키지명은 대소문자를 모두 허용하지만, 클래스명과 쉽게 구분하기 하기위해서 소문자로 하는 것을 원칙으로 하고 있다.

모든 클래스는 반드시 하나의 패키지에 포함되어야 한다고 했다. 그럼에도 불구하고 지금까지 소스파일을 작성할 때 패키지를 선언하지 않고도 아무런 문제가 없었던 이유는 자바에서 기본적으로 제공하는 '이름 없는 패키지(unnamed package)' 때문이다.

소스파일에 자신이 속할 패키지를 지정하지 않은 클래스는 자동적으로 '이름 없는 패키지'에 속하게 된다. 결국 패키지를 지정하지 않는 모든 클래스들은 같은 패키지에 속하는 셈이 된다.

간단한 프로그램이나 애플릿은 패키지를 지정하지 않아도 별 문제 없지만, 큰 프로젝트나 Java API와 같은 클래스 라이브러리를 작성하는 경우에는 미리 패키지를 구성하여 적용하도록 한다.

#### [예제 7-9] PackageTest.java

```
package com.javachobo.book;

class PackageTest
{
```

```

public static void main(String[] args)
{
    System.out.println("Hello World!");
}
}

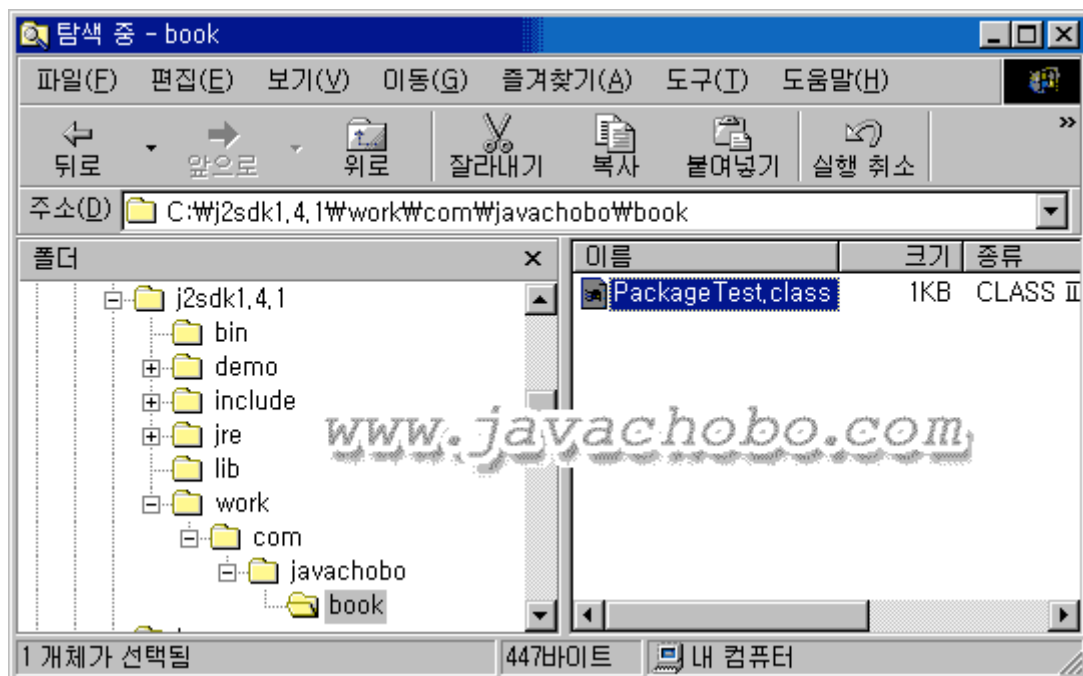
```

위의 예제를 작성한 뒤 다음과 같이 '-d' 옵션을 추가하여 컴파일을 한다.

```
C:\wj2sdk1.4.1\work>javac -d . PackageTest.java
```

'-d' 옵션은 소스파일에 지정된 경로를 통해 패키지의 위치를 찾아서 클래스파일을 생성한다. 만일 지정된 패키지와 일치하는 디렉토리가 존재하지 않는다면 자동적으로 생성한다.

'-d' 옵션 뒤에는 해당 패키지의 루트(root) 디렉토리의 경로를 적어준다. 여기서는 현재 디렉토리(.) 즉, 'C:\wj2sdk1.4.1\work' 로 지정했기 때문에 컴파일을 수행하고 나면 다음과 같은 구조로 디렉토리가 생성된다.



기존에 디렉토리가 존재하지 않았으므로 컴파일러가 패키지의 계층구조에 맞게 새로 디렉토리를 생성하고 컴파일된 클래스파일(PackageTest.class)을 book 디렉토리에 놓았다.

[참고] 만일 '-d' 옵션을 사용하지 않으면, 프로그래머가 직접 패키지의 계층구조에 맞게 디렉토리를 생성해야한다.

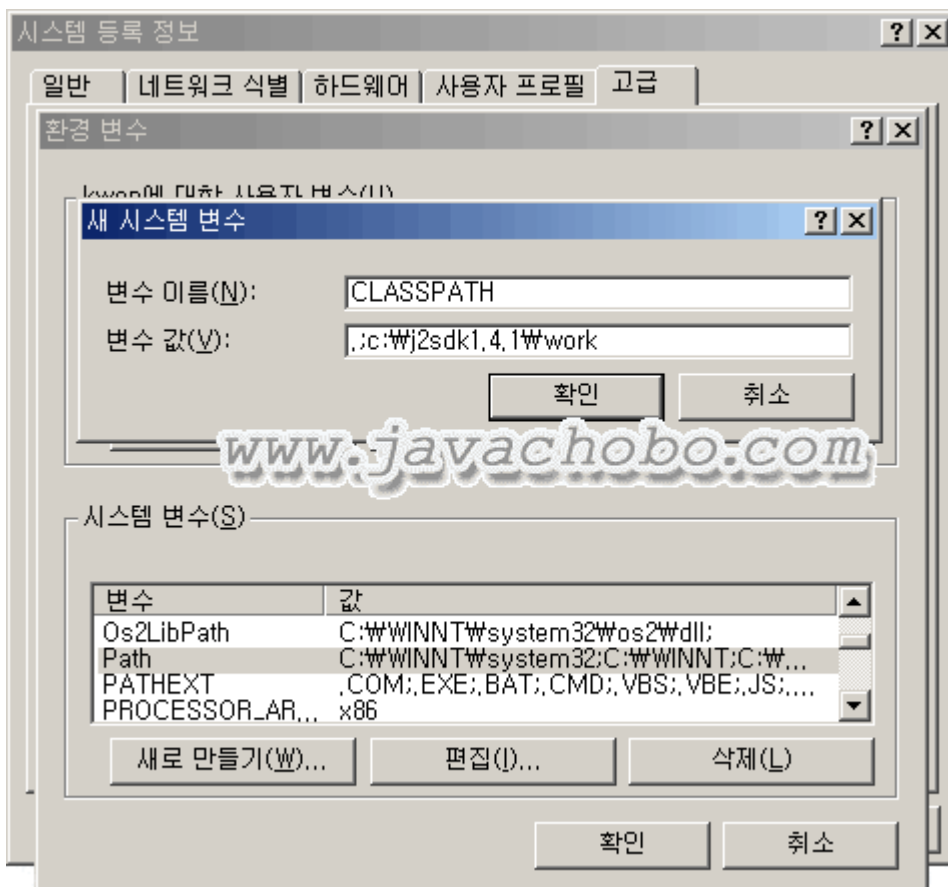
이 제는 패키지의 루트 디렉토리를 클래스패스(classpath)에 포함시켜야 한다. com.javachobo.book 패키지의 루트 디렉토리는 디렉토리 'com'의 상위 디렉토리인 'C:\wj2sdk1.4.1\work'이다. 이 디렉토리를 클래스패스에 포함시켜야만 실행 시 JVM이 PackageTest클래스를 찾을 수 있다.

[참고]클래스패스는 컴파일러(javac.exe)나 JVM 등이 클래스의 위치를 찾는데 사용되는 경로이다.

Windows 98이하에서는 autoexec.bat파일에 다음과 같은 명령을 한 줄 추가하면 된다.

```
SET CLASSPATH=.;C:\wj2sdk1.4.1\work;
```

Windows 2000에서는 '시스템등록정보-고급-환경변수-새로만들기'를 통해 아래의 그림과 같이 설정해 준다.



';'를 구분자로 하여 여러 개의 경로를 클래스패스에 지정할 수 있으며, 맨 앞에 '.'를 추가한 이유는 현재



디렉토리(.)를 클래스패스에 포함시키기 위해서이다.

클래스패스를 지정해 주지 않으면 기본적으로 현재 디렉토리(.)가 클래스패스로 지정되지만, 이처럼 클래스패스를 따로 지정해주는 경우에는 더 이상 현재 디렉토리가 자동적으로 클래스패스로 지정되지 않기 때문에 이처럼 별도로 추가를 해주어야 한다.

jar파일을 클래스패스에 추가하기 위해서는 경로와 파일명을 적어주어야 한다. 예를 들어 C:\wj2sdk1.4.1\work\util.jar파일을 클래스패스에 포함시키려면 다음과 같이 한다.

```
SET CLASSPATH=.;C:\wj2sdk1.4.1\work;C:\wj2sdk1.4.1\work\util.jar;
```

이제 클래스패스가 바르게 설정되었는지 확인하기 위해 다음과 같은 명령어를 입력해보자.

```
C:\WINDOWS>echo %classpath%  
.;C:\wj2sdk1.4.1\work;
```

현재 디렉토리를 의미하는 '.'와 'C:\wj2sdk1.4.1\work'가 클래스패스로 잘 지정되었음을 알 수 있다. 자, 이제 PackageTest예제를 실행시켜보자.

#### [실행결과]

```
C:\WINDOWS>java com.javachobo.book.PackageTest  
Hello World!
```

실행 시에는 이와 같이 PackageTest클래스의 패키지명을 모두 적어주어야 한다.

J2SDK에 기본적으로 설정되어 있는 클래스패스를 이용하면 위의 예제에서와 같이 클래스패스를 따로 설정하지 않아도 된다. 새로 추가하고자 하는 클래스를 'J2SDK설치디렉토리\jre\classes' 디렉토리에, jar파일인 경우에는 'J2SDK설치디렉토리\jre\lib\ext' 디렉토리에 넣기만 하면 된다.

[참고] jre디렉토리 아래의 classes디렉토리는 J2SDK설치 시에 자동으로 생성되지 않으므로 사용자가 직접 생성해야한다.

또는 실행 시에 '-cp' 옵션을 이용해서 일시적으로 클래스패스를 지정해 줄 수도 있다.

```
C:\WINDOWS>java -cp c:\wj2sdk1.4.1\work com.javachobo.book.PackageTest
```

### 3.3 import문

소스코드를 작성할 때 다른 패키지의 클래스를 사용할 때는 패키지명이 포함된 이름을 사용해야한다. 하지만, 매번 패키지명을 붙여서 작성하기란 여간 불편한 일이 아닐 것이다.

클래스의 코드를 작성하기 전에 import문으로 사용하고자 하는 클래스의 패키지를 미리 명시해주면 소스코드에 사용되는 클래스이름에서 패키지명은 생략할 수 있다.

import문의 역할은 컴파일러에게 소스파일에 사용된 클래스의 패키지에 대한 정보를 제공하는 것이다. 컴파일 시에 컴파일러는 import문을 통해 소스파일에 사용된 클래스들의 패키지를 알아 낸 다음, 모든 클래스이름 앞에 패키지명을 붙여 준다.

[참고] import문은 프로그램의 성능에 전혀 영향을 미치지 않는다. import문을 많이 사용하면 컴파일 시간이 아주 조금 더 걸릴 뿐이다.

### 3.4 import문의 선언

모든 소스파일(.java)에서 import문은 package문 다음에, 그리고 클래스 선언 문 이전에 위치해야한다. 그리고 import문은 package문과는 달리 한 소스파일에 여러 번 선언할 수 있다.

일반적인 소스파일의 구성은 'package문-import문-클래스선언'의 순서로 되어 있다.

import문을 선언하는 방법은 다음과 같다.

```
import 패키지명.클래스명;  
또는  
import 패키지명.*;
```

키워드import와 패키지명을 생략하고자 하는 클래스의 이름을 패키지명과 함께 써주면 된다. 같은 패키지에서 여러 개의 클래스가 사용될 때, import문을 여러 번 사용하는 대신 '패키지명.\*'을 이용해서 지정된 패키지에 속하는 모든 클래스를 패키지명 없이 사용할 수 있다.

[참고] 클래스이름을 지정해주는 대신 '\*'을 사용하면, 컴파일러는 해당 패키지에서 일치하는 클래스이름을 찾아야 하는 수고를 더 해야 할 것이다. 단지 그 뿐이다. 다른 차이는 없다.

```
import java.util.Calendar;
import java.util.Date;
import java.util.ArrayList;
```

이처럼 import문을 여러 번 사용하는 대신

```
import java.util.*;
```

위와 같이 한 문장으로 처리할 수 있다. 한 패키지에서 여러 클래스를 사용하는 경우 클래스의 이름을 일일이 지정해주는 것보다 '패키지명.\*'과 같이 하는 것이 편리하다.

하지만, import하는 패키지의 수가 많을 때는 어느 클래스가 어느 패키지에 속하는지 구별하기 어렵다는 단점이 있다.

한가지 더 알아두어야 할 것은 import문에서 클래스의 이름 대신 '\*'을 사용하는 것이 하위 패키지의 클래스까지 포함하는 것은 아니라는 것이다.

```
import java.util.*;
import java.text.*;
```

그래서, 위의 두 문장 대신 다음과 같이 할 수는 없다.

```
import java.*;
```

#### [예제7-10] ImportTest.java

```
import java.text.SimpleDateFormat;
import java.util.Date;

class ImportTest
{
    public static void main(String[] args)
    {
        Date today = new Date();

        SimpleDateFormat date = new SimpleDateFormat("yyyy/MM/dd");
        SimpleDateFormat time = new SimpleDateFormat("hh:mm:ss a");

        System.out.println("오늘 날짜는 " + date.format(today));
        System.out.println("현재 시간은 " + time.format(today));
    }
}
```

#### [실행결과]

오늘 날짜는 2003/01/07

현재 시간은 01:59:53 오후

현재 날짜와 시간을 지정된 형식에 맞춰 출력하는 예제이다. SimpleDateFormat과 Date클래스는 다른 패키지에 속한 클래스이므로 import문으로 어느 패키지에 속하는 클래스인지 명시해 주었다. 그래서 소스에서 클래스이름 앞에 패키지명을 생략할 수 있었다.

만일 import문을 지정하지 않았다면 다음과 같이 클래스이름에 패키지명도 적어줘야 했을 것이다.

```
java.util.Date today = new java.util.Date();
```

```
java.text.SimpleDateFormat date = new java.text.SimpleDateFormat("yyyy/MM/
dd");
java.text.SimpleDateFormat time = new java.text.SimpleDateFormat("hh:mm:ss
a");
```

import문으로 패키지를 지정하지 않으면 위와 같이 '이름없는 패키지'에 속한 클래스를 제외한 모든 클래스이름 앞에 패키지명을 반드시 붙여야 한다.

지금까지 System과 String 같은 java.lang패키지의 클래스들을 패키지명 없이 사용할 수 있었던 이유는 모든 소스파일에는 묵시적으로 다음과 같은 import문이 선언되어 있기 때문이다.

```
import java.lang.*;
```

java.lang패키지에는 매우 빈번히 사용되는 중요한 클래스들이 속한 패키지이기 때문에 따로 import문으로 지정하지 않아도 되도록 한 것이다.

# [Java의 정석]제7장 객체지향개념 2 - 4. 제어자(modifier)

자바의정석

2012/12/22 17:10

<http://blog.naver.com/gphic/50157740138>

## 4. 제어자(Modifier)

### 4.1 제어자란?

제어자(Modifier)는 클래스, 변수 또는 메서드의 선언부에 함께 사용되어 부가적인 의미를 부여한다.

그리고 제어자의 종류는 크게 접근제어자와 그 외의 제어자로 나눌 수 있다.

접근제어자 - public, protected, default, private

그 외 - static, final, abstract, native, transient, synchronized, volatile, strictfp

제어자는 클래스나 멤버변수와 메서드에 주로 사용되며, 하나의 대상에 대해서 여러 제어자를 조합하여 사용하는 것이 가능하다.

단, 접근제어자는 한번에 네 가지 중 하나만 선택해서 사용할 수 있다. 즉, 하나의 대상에 대해서 public과 private을 함께 사용할 수 없다는 것이다.

[참고]제어자들 간의 순서는 관계없지만 주로 접근제어자를 제일 왼쪽에 놓는 경향이 있다.

### 4.2 static - 클래스의, 공통적인

static은 '클래스의' 또는 '공통적인'의 의미를 가지고 있다. 인스턴스변수는 하나의 클래스로부터 생성되었다 하더라도 각각 다른 값을 유지하지만, 클래스변수(static멤버변수)는 인스턴스에 관계없이 같은 값을 갖는다. 그 이유는 단 하나의 변수를 모든 인스턴스가 공유하기 때문이다.

그리고, static이 붙은 멤버변수와 메서드, 그리고 초기화블럭은 인스턴스가 아닌 클래스에 관계된 것이기 때문에 인스턴스를 생성하지 않고도 사용할 수 있다.

인스턴스메서드와 static메서드의 근본적인 차이는 메서드 내에서 인스턴스 멤버를 사용하는가의 여부에 있다.

static이 사용될 수 있는 곳 - 멤버변수, 메서드, 초기화블럭

제어자	대상	의 미
static	멤버변수	<ul style="list-style-type: none"> <li>- 모든 인스턴스에 공통적으로 사용되는 클래스변수(공유변수)가 된다.</li> <li>- 클래스변수는 인스턴스를 생성하지 않고도 사용 가능하다.</li> <li>- 클래스가 메모리에 로드될 때 생성된다.</li> </ul>
	메서드	<ul style="list-style-type: none"> <li>- 인스턴스를 생성하지 않고도 호출이 가능한 static메서드가 된다.</li> <li>- static메서드 내에서는 인스턴스멤버들을 사용할 수 없다.</li> </ul>

인스턴스 멤버를 사용하지 않는 메서드는 static을 붙여서 static메서드로 선언하는 것을 고려해보도록 하자. 가능하다면 static메서드로 하는 것이 인스턴스를 생성하지 않고도 호출이 가능해서 더 편리하고 속도도 더 빠르다.

[참고]static초기화블럭은 클래스가 메모리에 로드될 때 단 한번만 수행되며, 주로 클래스변수(static멤버변수)를 초기화하는데 주로 사용된다.

#### 4.3 final - 마지막의, 변경될 수 없는

final은 '마지막의' 또는 '변경될 수 없는'의 의미를 가지고 있으며 거의 모든 대상에 사용될 수 있다.

변수에 사용되면 값을 변경할 수 없는 상수가 되며, 메서드에 사용되면 오버라이딩을 할 수 없게 되고 클래스에 사용되면 자신을 확장하는 자손클래스를 정의하지 못하게 된다.

final이 사용될 수 있는 곳 - 클래스, 메서드, 멤버변수, 지역변수

제어자	대상	의 미
final	클래스	변경될 수 없는 클래스, 확장될 수 없는 클래스가 된다. 그래서, final로 지정된 클래스는 다른 클래스의 조상이 될 수 없다.
	메서드	변경될 수 없는 메서드, final로 지정된 메서드는 오버라이딩을 통해 재정의 될 수 없다.
	멤버변수	변수 앞에 final이 붙으면, 값을 변경할 수 없는 상수가 된다.
	지역변수	

[참고]대표적인 final클래스로는 String과 Math가 있다.

#### 4.4 생성자를 이용한 final 멤버변수 초기화.

final이 붙은 멤버 변수는 상수이므로 일반적으로 선언과 초기화를 동시에 하지만, 멤버변수의 경우 생성자에서 초기화 되도록 할 수 있다.

클래스 내에 매개변수를 갖는 생성자를 선언하여, 인스턴스를 생성할 때 final이 붙은 멤버변수를 초기화하는데 필요한 값을 생성자의 매개변수로부터 제공받는 것이다.

이 기능을 활용하면 각 인스턴스마다 final이 붙은 멤버변수가 다른 값을 갖도록 하는 것이 가능하다.

만일 이것이 불가능하다면, 클래스에 선언된 final이 붙은 멤버변수는 모든 인스턴스에서 같은 값을 가져야만 할 것이다.

예를 들어 카드의 경우, 각 카드마다 다른 종류와 숫자를 갖지만, 일단 카드가 생성되면 카드의 값이 변경되어서는 안 된다. 52장의 카드 중에서 하나만 잘못 바뀌어도 같은 카드가 2장이 되는 일이 생기기 때문이다. 그래서 카드의 값을 바꾸기 보다는 카드의 순서를 바꾸는 쪽이 더 안전한 방법이다.

#### [예제 7-11] FinalCardTest.java

```
class Card {
    final int NUMBER;        // 상수지만 선언과 함께 초기화 하지 않고
    final String KIND;       // 생성자에서 단 한번만 초기화할 수 있다.
```



```

static int width = 100;
static int height = 250;

    Card(String kind, int num) {    // 매개변수로 넘겨받은 값으로 KIND와 NUMBER를
초기화한다.
        KIND = kind;
        NUMBER = num;
    }

    Card() {
        this("HEART", 1);
    }

    public String toString() {
        return "" + KIND + " " + NUMBER;
    }
}

class FinalCardTest {
    public static void main(String args[]) {
        Card c = new Card("HEART", 10);
//    c.NUMBER = 5;    에러발생! cannot assign a value to final variable NUMBER
        System.out.println(c.KIND);
        System.out.println(c.NUMBER);
    }
}

```

#### [실행결과]

HEART

10

#### 4.5 abstract - 추상의, 미완성의

abstract는 '미완성'의 의미를 가지고 있다. 메서드의 선언부만 작성하고 실제 수행내용은 구현하지 않은 추상메서드를 선언하는데 사용된다.

그리고, 클래스에 사용되어 클래스 내에 추상메서드가 존재한다는 것을 쉽게 알 수 있게 한다.

abstract가 사용될 수 있는 곳 - 클래스, 메서드

제어자	대상	의 미
abstract	클래스	클래스 내에 추상메서드가 선언되어 있음을 의미한다.
	메서드	선언부만 작성하고 구현부는 작성하지 않은 추상메서드임을 알린다.

[참고] 추상메서드가 없는 클래스도 abstract를 붙여서 추상클래스로 선언하는 것이 가능하기  
는 하지만 그렇게 해야 할 이유는 없다.

#### 4.6 접근제어자(Access Modifiers)

접근제어자는 멤버 또는 클래스에 사용되어, 해당하는 멤버 또는 클래스를 외부에서 접근하지  
못하도록 제한하는 역할을 한다.

접근제어자가 default임을 알리기 위해 실제로 default를 붙이지는 않는다. 클래스나 멤버변  
수, 메서드, 생성자에 접근제어자가 지정되어 있지 않다면, 접근제어자가 default임을 뜻한다.

접근제어자가 사용될 수 있는 곳 - 클래스, 멤버변수, 메서드, 생성자

private - 같은 클래스 내에서만 접근이 가능하다.

default - 같은 패키지 내에서만 접근이 가능하다.

protected - 같은 패키지 내에서, 그리고 다른 패키지의 자손클래스에서 접근이 가능하다.

public - 접근 제한이 전혀 없다.

제어자	같은 클래스	같은패키지	자손클래스	전 체
public				
protected				
default				
private				

접근 범위 순으로 나열 하면 다음과 같다.

private < default < protected < public

public은 접근 제한이 전혀 없는 것이고, private은 같은 클래스 내에서만 사용하도록 제한하는 가장 높은 제한이다. 그리고 default는 같은 패키지내의 클래스에서만 접근이 가능하도록 하는 것이다.

마지막으로, protected는 패키지에 관계없이 상속관계에 있는 자손클래스에서 접근할 수 있도록 하는 것이 제한목적이지만, 같은 패키지 내에서도 접근이 가능하다. 그래서 protected가 default보다 접근범위가 더 넓다.

대 상	사용가능한 접근제어자
클래스	public, (default)
메서드	public protected, (default), private
멤버변수	
지역변수	없 음

## 6.1 접근제어자를 이용한 캡슐화

클래스나 멤버, 주로 멤버에 접근제어자를 사용하는 이유는 클래스의 내부에 선언된 데이터를 보호하기 위해서이다.

데이터가 유효한 값을 유지하도록, 또는 비밀번호와 같은 데이터를 외부에서 함부로 변경하지

못하도록 하기 위해서는 외부로부터의 접근을 제한하는 것이 필요하다.

이 것을 데이터 감추기(Data Hiding)이라고 하며, 객체지향개념의 캡슐화(Encapsulation)에 해당하는 내용이다.

또 다른 이유는 클래스 내에서만 사용되는, 내부 작업을 위해 임시로 사용되는 멤버변수나 부분작업을 처리하기 위한 메서드 등의 멤버들을 클래스 내부에 감추기 위해서이다.

이러한 외부에서 접근할 필요가 없는 멤버들을 private으로 지정하여 외부에 노출시키지 않음으로써 복잡성을 줄일 수 있다. 이 것 역시 캡슐화에 해당한다.

접근 제어자를 사용하는 이유

- 외부로부터 데이터를 보호하기 위해서
- 외부에는 불필요한, 내부적으로만 사용되는, 부분을 감추기 위해서

이제 보다 구체적인 예제를 통해 자세히 알아보도록 하자. 시간을 표시하기 위한 클래스 Time을 다음과 같이 정의했다고 하자.

```
public class Time {  
    public int hour;  
    public int minute;  
    public int second;  
}
```

이 클래스의 인스턴스를 생성한 다음, 멤버변수에 직접 접근하여 값을 변경할 수 있을 것이다.

```
Time t = new Time();  
t.hour=25;
```

멤버변수 hour는 0보다는 같거나 크고 24보다는 작은 범위의 값을 가져야 하지만 위의 코드

에서처럼 잘못된 값을 지정한다고 해도 이 것을 막을 방법은 없다.

이런 경우 멤버변수는 private으로 제한하고 멤버변수의 값을 읽고 변경할 수 있는 public메서드를 제공함으로써 간접적으로 멤버변수의 값을 다룰 수 있도록 하는 것이 바람직하다.

```
public class Time {  
    private int hour;  
    private int minute;  
    private int second;  
  
    public int getHour() {    return hour; }  
    public void setHour(int hour) {  
        if (hour < 0 || hour > 24) return;  
        this.hour = hour;  
    }  
    public int getMinute() {    return minute; }  
    public void setMinute(int minute) {  
        if (minute < 0 || minute > 60) return;  
        this.minute = minute;  
    }  
    public int getSecond() {    return second; }  
    public void setSecond(int second) {  
        if (second < 0 || second > 60) return;  
        this.second = second;  
    }  
}
```

get으로 시작하는 메서드는 단순히 멤버변수의 값을 반환하는 일을 하고, set으로 시작하는 메서드는 매개변수에 지정된 값을 검사하여 조건에 맞는 값일 때만 멤버변수의 값을 변경하도록 작성되어 있다.

만일 상속을 통해 확장될 것이 예상되는 클래스라면 멤버에 접근 제한을 주되 자손클래스에서 접근하는 것이 가능하도록 하기 위해 private대신 protected를 사용한다.

[참고] 보통 멤버변수의 값을 읽는 메서드의 이름을 'get멤버변수이름'으로 하고, 멤버변수의

값을 변경하는 메서드의 이름을 'set멤버변수이름'으로 하지만 반드시 그렇게 해야 하는 것은 아니다. 그리고, get으로 시작하는 메서드를 getter, set으로 시작하는 메서드를 setter라고 부른다.

#### [예제7-12] Time.java

```
public class Time {
    private int hour;
    private int minute;
    private int second;

    public Time(int hour, int minute, int second) {
        setHour(hour);
        setMinute(minute);
        setSecond(second);
    }

    public int getHour() {    return hour; }
    public void setHour(int hour) {
        if (hour < 0 || hour >23) return;
        this.hour = hour;
    }
    public int getMinute() {    return minute; }
    public void setMinute(int minute) {
        if (minute < 0 || minute > 59) return;
        this.minute = minute;
    }
    public int getSecond() {    return second; }
    public void setSecond(int second) {
        if (second < 0 || second > 59) return;
        this.second = second;
    }
    public String toString() {
        return hour + ":" + minute + ":" + second;
    }
}
```

```

}

class TimeTest {
    public static void main(String[] args)
    {
        Time t = new Time(12, 35, 30);
        System.out.println(t);
//      t.hour=13;    에러발생! hour has private access in Time
        t.setHour(t.getHour()+1); // 현재시간보다 1시간 후로 변경한다.
        System.out.println(t);
    }
}

```

#### [실행결과]

12:35:30

13:35:30

Time 클래스의 모든 멤버변수의 접근제어자를 private으로 하고, 이 들을 다루기 위한 public 메서드를 추가했다. 그래서 t.hour=13;과 같이 멤버변수로의 직접적인 접근은 허가되지 않는다. 메서드를 통한 접근만이 허용될 뿐이다.

[참고] 하나의 소스파일(\*.java)에는 하나 이상의 public클래스가 존재할 수 없으며, 소스파일의 이름은 반드시 public클래스의 이름과 같아야 한다.

[참고] 위의 예제에서 set메서드의 조건을 강화하여, second(초)가 60이 되면, second의 값은 0으로 하고 minute(분)의 값을 증가시키도록 변경해보는 것도 좋은 연습이 될 것이다.

## 4.8 생성자의 접근제어자

생성자에 접근제어자를 사용함으로써 인스턴스의 생성을 제한할 수 있다. 보통 생성자의 접근 제어자는 클래스의 접근제어자와 같지만, 다르게 지정할 수도 있다.

생성자의 접근제어자를 private으로 지정하면, 외부에서 생성자에 접근할 수 없으므로 인스턴스를 생성할 수 없게 된다. 그래도 클래스 내부에서는 인스턴스의 생성이 가능하다.

```
class Singleton {
    private Singleton() {
        //...
    }
    //...
}
```

대신 인스턴스를 생성해서 반환해주는 public메서드를 제공함으로써 외부에서 이 클래스의 인스턴스를 사용하도록 할 수 있다. 이 메서드는 public인 동시에 static이어야 한다.

```
class Singleton {
    // getInstance()에서 사용될 수 있도록 인스턴스가 미리 생성되어야 하므로 static이어야 한다.
    private static Singleton s = new Singleton();

    private Singleton() {
        //...
    }

    // 인스턴스를 생성하지 않고도 호출할 수 있어야 하므로 static이어야 한다.
    public static Singleton getInstance() {
        return s ;
    }

    //...
}
```

이처럼 생성자를 통해 직접 인스턴스를 생성하지 못하게 하고 public메서드를 통해 인스턴스에 접근하게 함으로써 사용할 수 있는 인스턴스의 개수를 제한할 수 있다.

또 한가지, 생성자가 private인 클래스는 다른 클래스의 조상이 될 수 없다. 왜냐하면, 자손클래스의 인스턴스를 생성할 때 조상클래스의 생성자를 호출해야만 하는데, 생성자의 접근제어



자가 private이므로 자손클래스에서 호출하는 것이 불가능하기 때문이다.

그래서, 클래스 앞에 final을 더 추가하여 상속할 수 없는 클래스라는 것을 알리는 것이 좋다.

[참고] Math클래스는 몇 개의 상수와 static메서드만으로 구성되어 있기 때문에 인스턴스를 생성할 필요가 없다. 그래서 외부로부터의 불필요한 접근을 막기 위해 다음과 같이 생성자의 접근제어자를 private으로 지정하였다.

```
public final Math {  
    private Math() {}  
    //...  
}
```

#### [예제 7-13] SingletonTest.java

```
class Singleton {  
    private static Singleton s = new Singleton();  
  
    private Singleton() {  
        //...  
    }  
  
    public static Singleton getInstance() {  
        return s;  
    }  
  
    //...  
}  
  
class SingletonTest {  
    public static void main(String args[]) {  
        // Singleton s = new Singleton();    // 에러!!! Singleton() has private  
        // access in Singleton  
        Singleton s1 = Singleton.getInstance();  
    }  
}
```

```
}
```

#### 4.9 제어자(Modifier)의 조합

지금까지 접근제어자와 static, final, abstract에 대해서 학습했다. 이 외에도 더 많은 제어자들이 있으나 관련 내용이 현재 학습범위를 넘어선다고 판단되어 생략하였다. 이들은 앞으로 자리를 더 깊게 공부하게 되면서 자연스럽게 학습하게 될 것이다.

제어자가 사용될 수 있는 대상을 중심으로 제어자를 정리해보았다. 제어자의 기본적인 의미와 그 대상에 따른 의미 변화를 다시 한번 되새겨 보도록 하자.

대 상	사용가능한 제어자
클래스	public, (default), final, abstract
메서드	모든 접근제어자, final, abstract, static
멤버변수	모든 접근제어자, final, static
지역변수	final <a href="http://www.javachobo.com">www.javachobo.com</a>

마지막으로 제어자를 조합해서 사용할 때 주의해야 할 사항에 대해 정리해 보았다.

1. 메서드에 static과 abstract를 함께 사용할 수 없다.

- static메서드는 몸통이 있는 메서드에만 사용할 수 있기 때문이다.

2. 클래스에 abstract와 final을 동시에 사용할 수 없다.

- 클래스에 사용되는 final은 클래스를 확장할 수 없다는 의미이고 abstract는 상속을 통해서 완성되어야 한다는 의미이므로 서로 모순되기 때문이다.

3. abstract메서드의 접근제어자가 private일 수 없다.

- abstract메서드는 자손클래스에서 구현해주어야 하는데 접근제어자가 private이면, 자손클래스에서 접근할 수 없기 때문이다.

4. 메서드에 private과 final을 같이 사용할 필요는 없다.

- 접근제어자가 private인 메서드는 오버라이딩될 수 없기 때문이다. 이 둘 중 하나만 사용해도 의미가 충분하다.

# [Java의 정석]제7장 객체지향개념 2 - 5. 다형성(Polymorphism)

자바의정석

2012/12/22 17:11

<http://blog.naver.com/gphic/50157740169>

## 5. 다형성(Polymorphism)

### 5.1 다형성이란?

상속과 함께 객체지향개념의 중요한 특징중의 하나인 다형성에 대해서 배워 보도록 하자. 다형성은 상속과 깊은 관계가 있으므로 학습하기에 앞서 상속에 대한 충분히 알고 있어야 한다.

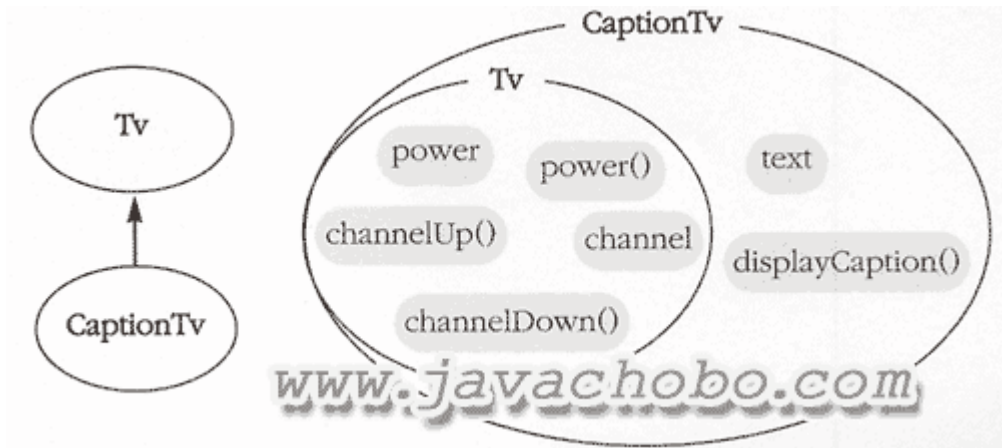
객체지향개념에서의 다형성이란 '여러 가지 형태를 가질 수 있는 능력'을 의미하며, 자바에서는 한 타입의 참조변수로 여러 타입의 객체를 참조할 수 있도록 함으로써 다형성을 프로그램적으로 구현하였다.

이를 좀더 구체적으로 말하자면, 조상클래스 타입의 참조변수로 자손클래스의 인스턴스를 참조할 수 있도록 하였다는 것이다. 예제를 통해서 보다 자세히 알아보도록 하자.

```
class Tv {  
    boolean power;    // 전원상태(on/off)  
    int channel;    // 채널  
  
    void power() {    power = !power; }  
    void channelUp() {    ++channel; }  
    void channelDown() {    --channel;    }  
}  
  
class CaptionTv extends Tv {  
    String text;    // 캡션을 보여 주기 위한 문자열  
    void caption() { /* 내용생략 */ }
```

```
}
```

Tv클래스와 CaptionTv클래스가 이와 같이 정의되어 있을 때, 두 클래스간의 관계를 그림으로 나타내면 아래와 같다.



클래스 Tv와 CaptionTv는 서로 상속관계에 있으며, 이 두 클래스의 인스턴스를 생성하고 사용하기 위해서는 다음과 같이 할 수 있다.

```
Tv t = new Tv();
CaptionTv c = new CaptionTv();
```

지금까지 우리는 생성된 인스턴스를 다루기 위해서, 인스턴스의 타입과 일치하는 타입의 참조 변수만을 사용했다. 즉, Tv인스턴스를 다루기 위해서는 Tv타입의 참조변수를 사용하고, CaptionTv인스턴스를 다루기 위해서는 CaptionTv타입의 참조변수를 사용했다. 이처럼 인스턴스의 타입과 참조변수의 타입이 일치하는 것이 보통이지만, Tv와 CaptionTv 클래스가 상속관계에 있을 경우, 다음과 같이 조상클래스 타입의 참조변수로 자손클래스 타입의 객체를 참조하도록 하는 것도 가능하다.

```
Tv t = new CaptionTv();
```

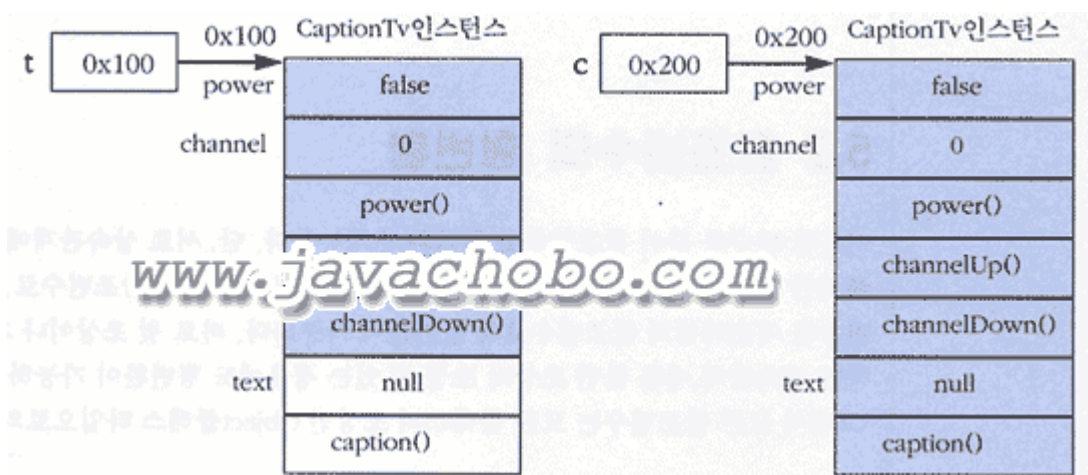
그러면 이제 인스턴스를 같은 타입의 참조변수로 참조하는 것과 조상타입의 참조변수로 참조하는 것은 어떤 차이가 있는지에 대해서 알아보도록 하자.

```
CaptionTv c = new CaptionTv();
```

```
Tv t = new CaptionTv();
```

위 의 코드에서 CaptionTv인스턴스 2개를 생성하고, 참조변수 c, t가 생성된 인스턴스를 하나씩 참조하도록 하였다. 이 경우 실제 인스턴스가 CaptionTv타입이라 할지라도, 참조변수 t로는 CaptionTv인스턴스의 모든 멤버를 사용할 수 없다.

Tv타입의 참조변수로는 CaptionTv인스턴스 중에서 Tv클래스의 멤버들(상속받은 멤버포함)만 사용할 수 있다. 따라서, 생성된 CaptionTv인스턴스의 멤버 중에서 Tv클래스에 정의 되지 않은 멤버, text와 caption()은 참조변수 t로 사용이 불가능하다. 즉, t.text 또는 t.caption()와 같이 할 수 없다는 것이다.



[참고] 실제로는 모든 클래스의 최고조상인 Object클래스로부터 상속받은 부분도 포함되어야 하지만 간단히 하기위해 생략했다.

반대로 아래와 같이 자손타입의 참조변수로 조상타입의 인스턴스를 참조하는 것은 가능할까?

```
CaptionTv c = new Tv();    // 컴파일 에러 발생
```

그렇지 않다. 위의 코드를 컴파일 하면 에러가 발생한다. 그 이유는 실제 인스턴스인 Tv의 멤버 개수보다 참조변수 c가 사용할 수 있는 멤버 개수가 더 많기 때문이다. 그래서 이를 허용하지 않는다.

CaptionTv클래스에는 text와 caption()이 정의되어 있으므로 참조변수 c로는 c.text, c.caption()과 같은 방식으로 c가 참조하고 있는 인스턴스에서 text와 caption()을 사용하려 할 수 있다.

하지만, c가 참조하고 있는 인스턴스는 Tv타입이고, Tv타입의 인스턴스에는 text와 caption()이 존재하지 않기 때문에 이들을 사용하려 하면 문제가 발생한다.

그래서, 자손타입의 참조변수로 조상타입의 인스턴스를 참조하는 것은 존재하지 않는 멤버를 사용하고자 할 가능성이 있으므로 허용하지 않는다. 참조변수가 사용할 수 있는 멤버의 개수는 인스턴스의 멤버 개수보다 같거나 작아야 하는 것이다.

[참고] 클래스는 상속을 통해서 확장될 수는 있어도 축소될 수는 없어서, 조상인스턴스의 멤버 개수는 자손인스턴스의 멤버 개수보다 항상 작거나 같다.

참조변수의 타입이 참조변수가 참조하고 있는 인스턴스에서 사용할 수 있는 멤버의 개수를 결정한다는 사실을 이해하는 것은 매우 중요하다.

그렇다면, 인스턴스의 타입과 일치하는 참조변수를 사용하면 인스턴스의 멤버들을 모두 사용할 수 있을 텐데 왜 조상타입의 참조변수를 사용해서 인스턴스의 일부 멤버만을 사용하도록 할까?

이에 대한 답은 앞으로 배우게 될 것이며, 지금은 조상타입의 참조변수로도 자손클래스의 인스턴스를 참조할 수 있다는 것과 그 차이에 대해서만 이해하면 된다.

## 5.2 참조변수의 형변환

기본형 변수와 같이 참조변수도 형변환이 가능하다. 단, 서로 상속관계에 있는 클래스사이에서만 가능하기 때문에 자손타입의 참조변수를 조상타입의 참조변수로, 조상타입의 참조변수를 자손타입의 참조변수로의 형변환이 가능하다.

[참고] 바로 윗 조상이나 자손이 아닌 간접적인 상속관계, 예를 들면 조상의 조상,에 있는 경우에도 형변환이 가능하다. 따라서 모든 참조변수는 모든 클래스의 조상인 Object클래스 타입으로의 형변환이 가능하다.

기본형 변수의 형변환에서 작은 자료형에서 큰 자료형의 형변환은 생략이 가능하듯이, 참조형 변수의 형변환에서는 자손타입의 참조변수를 조상타입으로 형변환하는 경우에는 형변환을 생략할 수 있다.

자손타입 -> 조상타입 (Up-casting) : 형변환 생략가능

자손타입 <- 조상타입 (Down-casting) : 형변환 생략불가

조상타입의 참조변수를 자손타입의 참조변수로 변환하는 것을 다운캐스팅(Down-casting)이라고 하며, 자손타입의 참조변수를 조상타입의 참조변수로 변환하는 것을 업캐스팅(Up-casting)이라고 한다.

참조변수간의 형변환 역시 캐스트연산자를 사용하며, ()안에 변환하고자 하는 타입의 이름(클래스명)을 적어주면 된다.

```
class Car {  
    String color;  
    int door;  
    void drive() {        // 운전하는 기능  
        System.out.println("drive, Brrrr~");  
    }  
}
```



```

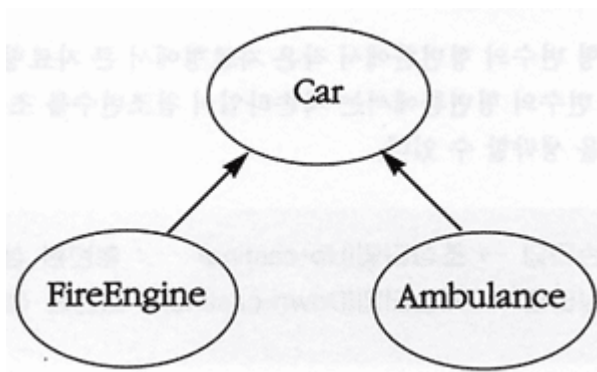
void stop() {           // 멈추는 기능
    System.out.println("stop!!!");
}

class FireEngine extends Car {    // 소방차
    void water() {               // 물 뿌리는 기능
        System.out.println("water!!!");
    }
}

class Ambulance extends Car {     // 앰불런스
    void siren() {                // 사이렌을 울리는 기능
        System.out.println("siren~~~");
    }
}

```

이와 같이 세 클래스, Car, FireEngine, Ambulance가 정의되어 있을 때, 이 세 클래스간의 관계를 그림으로 표현하면 아래와 같다.



[참고] 이처럼 클래스들간의 상속관계를 그림으로 나타내 보면, 형변환의 가능여부를 쉽게 확인할 수 있다.

Car 클래스는 FireEngine클래스와 Ambulance클래스의 조상이다. 그렇다고 해서 FireEngine클래스와 Ambulance클래스가 형제관계는 아니다. 자바에서는 조상과 자식관계만 존재하기 때문에 FireEngine클래스와 Ambulance클래스는 서로 아무런 관계가 없는 것으

로 간주된다.

따라서, Car타입의 참조변수와 FireEngine타입의 참조변수 그리고 Car타입의 참조변수와 Ambulance타입의 참조변수간에는 서로 형변환이 가능하지만, FireEngine타입의 참조변수와 Ambulance타입의 참조변수간에는 서로 형변환이 가능하지 않다.

```
FireEngine f;  
Ambulance a;  
a = (Ambulance)f;    // 컴파일 에러!!!  
f = (FireEngine)a;    // 컴파일 에러!!!
```

먼저 Car타입의 참조변수와 FireEngine타입의 참조변수간의 형변환을 예로 들어보자.

```
Car car = null;  
FireEngine fe = new FireEngine();  
FireEngine fe2 = null;  
  
car = fe;                // car = (Car)fe;에서 형변환이 생략된 형태이다.  
fe2 = (FireEngine)car;    // 형변환을 생략할 수 없다.
```

참조변수 car와 fe의 타입이 서로 다르기 때문에, 대입연산(=)이 수행되기 전에 형변환을 수행하여 두 변수간의 타입을 맞춰 주어야 한다.

그러나, 자손타입의 참조변수를 조상타입의 참조변수에 할당할 경우 형변환을 생략할 수 있어서 car = fe;와 같이 하였다. 원칙적으로는 car = (Car)fe;와 같이 해야 한다.

반대로 조상타입의 참조변수를 자손타입의 참조변수에 할당할 경우 형변환을 생략할 수 없으므로, fe2 = (FireEngine)car; 와 같이 명시적으로 형변환을 해주어야 한다.

참고로 형변환을 생략할 수 있는 경우와 생략할 수 없는 경우에 대한 이유를 설명하자면 다음과 같다.

Car타입의 참조변수 c가 있다고 가정하자. 참조변수 c가 참조하고 있는 인스턴스는 아마도

Car인스턴스이거나 자손인 FireEngine인스턴스일 것이다.

Car 타입의 참조변수 c를 Car타입의 조상인 Object타입의 참조변수로 형변환 하는 것은 참조변수가 다룰 수 있는 멤버의 개수가 실제 인스턴스가 갖고 있는 멤버의 개수보다 적을 것이 분명하므로 문제가 되지 않는다. 그래서 형변환을 생략할 수 있도록 한 것이다.

하지만, Car타입의 참조변수 c를 자손인 FireEngine타입으로 변환하는 것은 참조변수가 다룰 수 있는 멤버의 개수를 늘리는 것이므로, 실제 인스턴스의 멤버 개수보다 참조변수가 사용할 수 있는 멤버의 개수가 더 많아질 수 있으므로 문제가 발생할 가능성이 있다.

그래서 자손타입으로의 형변환은 생략할 수 없으며, 형변환을 수행하기 전에 instanceof연산자를 사용해서 참조변수가 참조하고 있는 실제 인스턴스의 타입을 확인하는 것이 안전하다.

형변환은 참조변수의 타입을 변환하는 것이지 인스턴스를 변환하는 것은 아니기 때문에 참조변수의 형변환은 인스턴스에 아무런 영향을 미치지 않는다.

단지 참조변수의 형변환을 통해서, 참조하고 있는 인스턴스에서 사용할 수 있는 멤버의 범위(개수)를 조절하는 것 뿐이다.

[참고] 전에 예로 든 Tv t = new CaptionTv();도 Tv t = (Tv)new CaptionTv();의 생략된 형태이다.

만일 이해가 잘 안 간다면, Tv t = (Tv)new CaptionTv();는 아래의 두 줄을 간략히 한 것이라고 생각하면 이해가 될 것이다.

```
CaptionTv c = new CaptionTv();  
Tv t = (Tv)c;
```

#### [예제7-14] CastingTest1.java

```
class CastingTest1 {  
    public static void main(String args[]) {  
        Car car = null;  
        FireEngine fe = new FireEngine();  
        FireEngine fe2 = null;  
  
        fe.water();  
    }  
}
```

```

        car = fe;        // car =(Car)fe;에서 형변환이 생략된 형태다.
//    car.water();      컴파일 에러!!! Car타입의 참조변수로는 water()를 호출할 수 없
다.

        fe2 = (FireEngine)car;    // 자손타입 <- 조상타입
        fe2.water();
    }
}

class Car {
    String color;
    int door;
    void drive() {        // 운전하는 기능
        System.out.println("drive, Brrrr~");
    }
    void stop() {         // 멈추는 기능
        System.out.println("stop!!!");
    }
}

class FireEngine extends Car {    // 소방차
    void water() {               // 물을 뿌리는 기능
        System.out.println("water!!!");
    }
}

```

#### [실행결과]

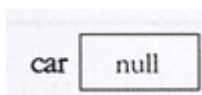
water!!!

water!!!

위 예제의 주요실행과정을 그림과 함께 자세히 살펴보자.

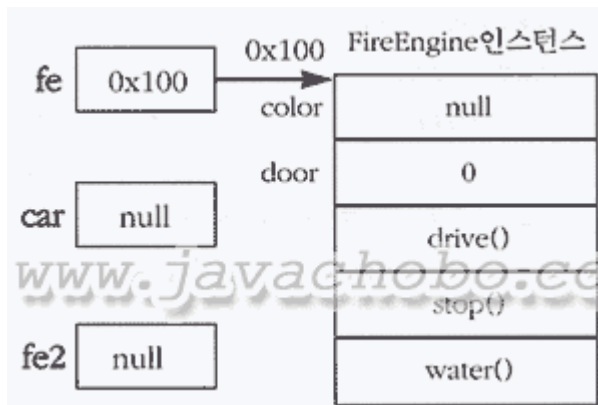
#### 1. Car car = null;

Car타입의 참조변수 car을 선언하고 null로 초기화한다.



2. `FireEngine fe = new FireEngine();`

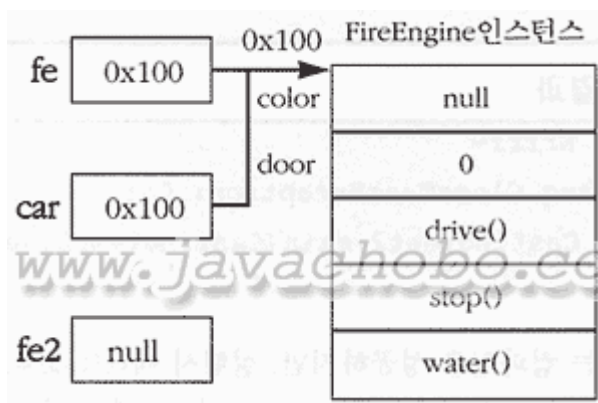
FireEngine인스턴스를 생성하고 FireEngine타입의 참조변수로 참조하도록 한다.



3. `car = fe;`

참조변수 `fe`가 참조하고 있는 인스턴스를 참조변수 `car`가 참조하도록 한다. `fe`의 값(`fe`가 참조하고 있는 인스턴스의 주소)이 `car`에 저장된다. 이때 두 참조변수의 타입이 다르므로 참조변수 `fe`가 형변환되어야 하지만 생략되었다.

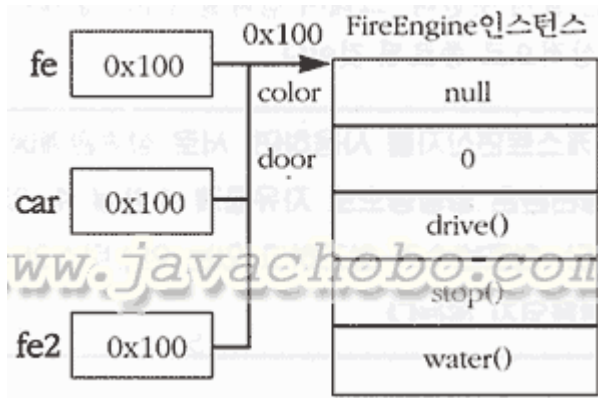
이제는 참조변수 `car`를 통해서도 `FireEngine`인스턴스를 사용할 수 있지만, `fe`와는 달리, `car`는 `Car`타입이므로 `Car`클래스의 멤버가 아닌 `water()`는 사용할 수 없다.



4. `fe2 = (FireEngine)car;`

참조변수 `car`가 참조하고 있는 인스턴스를 참조변수 `fe2`가 참조하도록 한다. 이때 두 참조변수의 타입이 다르므로 참조변수 `car`를 형변환하였다. `car`에는 `FireEngine`인스턴스의 주소가 저장되어 있으므로 `fe2`에도 `FireEngine`인스턴스의 주소가 저장된다.

이제는 참조변수 `fe2`를 통해서도 `FireEngine`인스턴스를 사용할 수 있지만, `car`와는 달리, `fe2`는 `FireEngine`타입이므로 `FireEngine`인스턴스의 모든 멤버들을 사용할 수 있다.



#### [예제7-15] CastingTest2.java

```
class CastingTest2 {
    public static void main(String args[]) {
        Car car = new Car();
        Car car2 = null;
        FireEngine fe = null;

        car.drive();
        fe = (FireEngine)car;    // 실행 시 에러가 발생한다.
        fe.drive();
        car2 = fe;
        car2.drive();
    }
}
```

#### [실행결과]

```
drive, Brrrr~
java.lang.ClassCastException: Car
    at CastingTest2.main(CastingTest2.java:8)
```

이 예제는 컴파일은 성공하지만, 실행시 예외(ClassCastException)가 발생한다. 예외가 발생한 곳은 문장은 CastingTest2.java의 8번째 라인인 `fe = (FireEngine)car;`이며, 발생이유는 형변환에 오류가 있기 때문이다. 캐스트 연산자를 이용해서 조상타입의 참조변수를 자손타입의 참조변수로 형변환한 것이기 때문에 문제가 없어 보이지만, 문제는 참조변수 `car`가 참조하고 있는 인스턴스가 Car타입의 인스턴스라는데 있다. 전에 배운 것처럼 조상타입의 인스턴스를 자손타입의 참조변수로 참조하는 것은 허용되지 않기 때문이다.

위의 예제에서 `Car car = new Car();`를 `Car car = new FireEngine();`와 같이 변경하면, 컴파일시 뿐 만 아니라 실행 시에도 에러가 발생하지 않을 것이다.

컴파일시에는 참조변수간의 타입만 체크하기 때문에 실행 시 생성될 인스턴스의 타입에 대해서는 알지 못한다. 그래서 컴파일시에는 문제가 없었지만, 실행 시에는 에러가 발생하여 실행이 비정상적으로 종료된 것이다.

캐스트연산자를 사용하면, 서로 상속관계에 있는 클래스 타입의 참조변수간의 형변환은 양방향으로 자유롭게 수행될 수 있다. 단, 참조변수가 참조하고 있는 인스턴스의 타입보다 자손타입으로의 형변환은 허용되지 않는다.

### 5.3 instanceof연산자

참조변수가 참조하고 있는 인스턴스의 실제 타입을 알아보기 위해 instanceof연산자를 사용한다.

주로 조건문에 사용되며, instanceof의 왼쪽에는 참조변수를 오른쪽에는 타입(클래스명)이 피연산자로 위치한다. 그리고 연산의 결과로 boolean값인 true, false 중의 하나를 반환한다. instanceof를 이용한 연산결과로 true값을 얻었다는 것은 참조변수가 검사한 타입으로 형변환이 가능하다는 것을 뜻한다.

```
if (c instanceof FireEngine) {    // c는 Car타입의 참조변수
    FireEngine fe = (FireEngine)c;
    fe.water();
    //...
}
```

위의 코드는 instanceof연산자로 Car타입의 참조변수 c가 FireEngine타입의 인스턴스를 참조하고 있는지를 검사하고, 그 결과가 true이면, 형변환을 통해 FireEngine타입의 참조변수가 참조하도록 하여, FireEngine인스턴스의 멤버인 water()를 사용할 수 있도록 한 것이다.

조상타입의 참조변수로 자손타입의 인스턴스를 참조할 수 있기 때문에, 참조변수의 타입과 인스턴스의 타입이 항상 일치하지는 않는다는 것을 배웠다. 이 경우, 실제 인스턴스의 멤버들을 모두 사용할 수 없기 때문에, 참조변수의 형변환을 통해서 실제 인스턴스의 모든 멤버들을 사용할 수 있도록 할 수 있다.

#### [예제7-16] InstanceofTest.java

```
class InstanceofTest {
    public static void main(String args[]) {
        FireEngine fe = new FireEngine();

        if(fe instanceof FireEngine) {
            System.out.println("This is a FireEngine instance.");
        }

        if(fe instanceof Car) {
            System.out.println("This is a Car instance.");
        }

        if(fe instanceof Object) {
            System.out.println("This is an Object instance.");
        }
    }
}
```

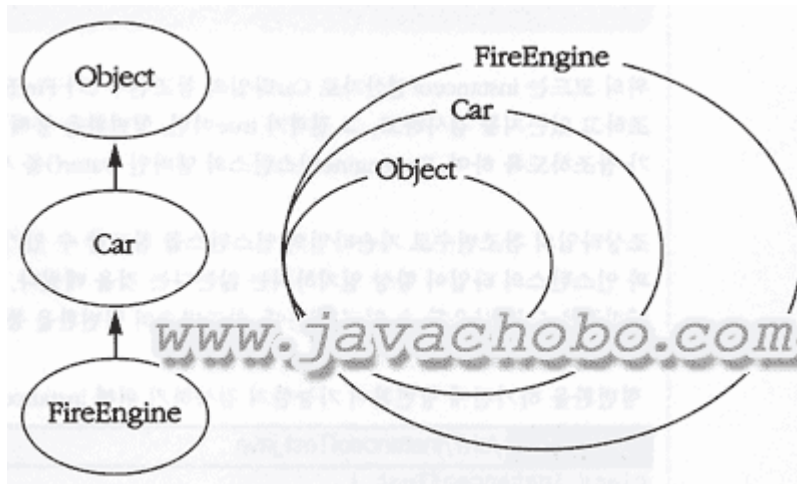
#### [실행결과]

```
This is a FireEngine instance.
This is a Car instance.
This is an Object instance.
```

비록 생성된 인스턴스는 FireEngine타입일지라도, Object타입과 Car타입의 instanceof연산에서도 true를 결과로 얻었다. 그 이유는 FireEngine클래스는 Object클래스와 Car클래스의 자손클래스이므로 조상의 멤버들을 상속받았기 때문에, FireEngine인스턴스는 Object인스턴스와 Car인스턴스를 포함하고 있는 셈이기 때문이다.



요약하면, 실 제 인스턴스와 같은 타입의 instanceof연산 이외에 조상타입의 instanceof연산에도 true를 결과로 얻으며, instanceof연산의 결과가 true라는 것은 검사한 타입으로의 형변환을 해도 아무런 문제가 없다는 뜻이다.



#### 5.4 참조변수와 인스턴스의 연결.

조상타입의 참조변수와 자손타입의 참조변수의 차이점이 사용할 수 있는 멤버의 개수에 있다고 배웠다. 여기서 한가지 더 알아두어야 할 내용이 있다.

조상클래스에 선언된 멤버변수와 같은 이름의 멤버변수를 자손클래스에 중복으로 정의했을 때, 조상타입의 참조변수로 자손 인스턴스를 참조하는 경우와 자손타입의 참조변수로 자손인스턴스를 참조하는 경우 다른 결과를 얻는다.

메서드의 경우 조상클래스의 메서드를 자손의 클래스에서 오버라이딩한 경우에도 참조변수의 타입에 관계없이 항상 실제 인스턴스의 메서드(오버라이딩된 메서드)가 호출되지만, 멤버변수의 경우 참조변수의 타입에 따라 다르게 사용된다.

[참고] static메서드는 멤버변수처럼 참조변수의 타입에 영향을 받는다. 참조변수의 타입에 영향을 받지 않는 것은 인스턴스메서드 뿐이다.

결 론부터 말하자면, 멤버변수가 조상클래스와 자손클래스에 중복으로 정의된 경우, 조상타입의 참조변수를 사용했을 때는 조상클래스에 선언된 멤버변수가 사용되고, 자손타입의 참조변수를 사용했을 때는 자손클래스에 선언된 멤버변수가 사용된다.

하지만, 중복 정의되지 않은 경우, 조상타입의 참조변수를 사용했을 때와 자손타입의 참조변수를 사용했을 때의 차이는 없다. 중복된 경우는 참조변수의 타입에 따라 달라지지만, 중복되지 않은 경우 선택의 여지가 없기 때문이다.

**[예제 7-17] BindingTest.java**

```
class BindingTest {
    public static void main(String[] args) {
        Parent p = new Child();
        Child c = new Child();

        System.out.println("p.x = " + p.x);
        p.method();

        System.out.println("c.x = " + c.x);
        c.method();
    }
}

class Parent {
    int x = 100;
    void method() {
        System.out.println("Parent Method");
    }
}

class Child extends Parent {
    int x = 200;
    void method() {
        System.out.println("Child Method");
    }
}
```

**[실행결과]**

```
p.x = 100
Child Method
c.x = 200
Child Method
```

타입은 다르지만, 참조변수 p, c 모두 Child 인스턴스를 참조하고 있다. 그리고, Parent 클래스와 Child 클래스는 서로 같은 멤버들을 정의하고 있다.

이 때 조상타입의 참조변수 p로 Child 인스턴스의 멤버들을 사용하는 것과 자손타입의 참조변수 c로 Child 인스턴스의 멤버들을 사용하는 것의 차이를 알 수 있다.

메서드인 method()의 경우 참조변수의 타입에 관계없이 항상 실제 인스턴스의 타입인 Child 클래스에 정의된 메서드가 호출되지만, 멤버변수인 x는 참조변수의 타입에 따라서 달라진다.

#### [예제 7-18] BindingTest2.java

```
class BindingTest2 {
    public static void main(String[] args) {
        Parent p = new Child();
        Child c = new Child();

        System.out.println("p.x = " + p.x);
        p.method();

        System.out.println("c.x = " + c.x);
        c.method();
    }
}

class Parent {
    int x = 100;
    void method() {
        System.out.println("Parent Method");
    }
}

class Child extends Parent {}
```

#### [실행결과]

```
p.x = 100
Parent Method
c.x = 100
Parent Method
```

이전의 예제와는 달리 Child클래스에는 아무런 멤버도 정의되어 있지 않고 단순히 조상으로부터 멤버들을 상속받는다. 그렇기 때문에 참조변수의 타입에 관계없이 조상의 멤버들을 사용하게 된다.

이처럼 자손클래스에서 조상클래스의 멤버를 중복으로 정의하지 않았을 때는 참조변수의 타입에 따른 변화는 없다. 어느 클래스의 멤버가 호출되어야 할지, 즉 조상의 멤버가 호출되어야 할지, 자손의 멤버가 호출되어야 할지에 대해 선택의 여지가 없기 때문이다.

참조변수의 타입에 따라 결과가 달라지는 경우는 조상클래스의 멤버변수와 같은 이름의 멤버변수를 자손클래스에 중복해서 정의한 경우뿐이다.

#### [예제7-19] BindingTest3.java

```
class BindingTest3 {
    public static void main(String[] args) {
        Parent p = new Child();
        Child c = new Child();

        System.out.println("p.x = " + p.x);
        p.method();

        System.out.println("c.x = " + c.x);
        c.method();
    }
}

class Parent {
    int x = 100;
    void method() {
        System.out.println("Parent Method");
    }
}
```

```

    }
}

class Child extends Parent {
    int x = 200;
    void method() {
        System.out.println("x=" + x);    // this.x와 같다.
        System.out.println("super.x=" + super.x);
        System.out.println("this.x=" + this.x);
    }
}

```

#### [실행결과]

```

p.x = 100
x=200
super.x=100
this.x=200
c.x = 200
x=200
super.x=100
this.x=200

```

자손클래스 Child에 선언된 멤버변수 x와 조상클래스 Parent로부터 상속받은 멤버변수 x를 구분하는데 참조변수 super와 this가 사용된다.

자손인 Child클래스에서의 super.x는 조상클래스인 Parent에 선언된 멤버변수 x를 뜻하며, this.x 또는 x는 Child클래스의 멤버변수 x를 뜻한다. 그래서 위 결과에서 x와 this.x의 값이 같다.

전 에 배운 것과 같이 멤버변수들은 주로 private으로 접근이 제어되고, 메서드를 통해서 멤버변수에 접근하도록 하지, 이번 예제에서처럼 다른 외부 클래스에서 참조변수를 통해 가능하면 직접적으로 멤버변수에 접근할 수 있도록 하지는 않는다.

이 예제에서 알 수 있듯이 멤버변수에 직접 접근하면, 참조변수의 타입에 따라 사용되는 멤버변수가 달라질 수 있으므로 주의해야한다.

## 5.5 매개변수의 다형성

참조변수의 다형적인 특징은 메서드의 매개변수에도 적용된다. 아래와 같이 Product, Tv, Computer, Buyer클래스가 정의되어 있다고 가정하자.

```
class Product {  
    int price;           // 제품의 가격  
    int bonusPoint;      // 제품구매 시 제공하는 보너스점수  
}  
class Tv extends Product {}  
class Computer extends Product {}  
class Audio extends Product {}  
  
class Buyer {           // 고객, 물건을 사는 사람  
    int money = 1000;    // 소유금액  
    int bonusPoint = 0;  // 보너스점수  
}
```

Product클래스는 Tv와 Computer클래스의 조상이며, Buyer클래스는 제품(Product)를 구입하는 사람을 클래스로 표현한 것이다.

Buyer클래스에 물건을 구입하는 기능의 메서드를 추가해보자. 구입할 대상이 필요하므로 매개변수로 구입할 제품을 넘겨받아야 한다. Tv를 살수 있도록 매개변수를 Tv타입으로 하였다.

```
void buy(Tv t) {  
    // Buyer가 가진 돈(money)에서 제품의 가격(t.price)만큼 뺀다.  
    money = money - t.price;  
    // Buyer의 보너스점수(bonusPoint)에 제품의 보너스점수(t.bonusPoint)를 더한다.  
    bonusPoint = bonusPoint + t.bonusPoint;  
}
```

buy(Tv t)는 제품을 구입하면 제품을 구입한 사람이 가진 돈에서 제품의 가격을 빼고, 보너스

점수는 추가하는 작업을 하도록 작성되었다. 그런데 의미론적으로는 buy(Tv t)는 Tv밖에 살 수 없기 때문에 아래와 같이 다른 제품들도 구입할 수 있는 메서드가 추가로 필요하다.

```
void buy(Computer c) {
    money = money - c.price;
    bonusPoint = bonusPoint + c.bonusPoint;
}

void buy(Audio a) {
    money = money - a.price;
    bonusPoint = bonusPoint + a.bonusPoint;
}
```

이렇게 되면, 제품의 종류가 늘어날 때마다 Buyer클래스에는 새로운 buy메서드를 추가해주어야 할 것이다.

그러나 메서드의 매개변수에 참조변수의 다형성을 이용하면 아래와 같이 하나의 메서드로 간단히 처리할 수 있다.

```
void buy(Product p) {
    money = money - p.price;
    bonusPoint = bonusPoint + p.bonusPoint;
}
```

매개변수가 Product타입의 참조변수라는 것은, 메서드의 매개변수로 Product클래스의 자손 타입의 참조변수면 어느 것이나 매개변수로 받아들일 수 있다는 뜻이다.

그리고, Product클래스에 price와 bonusPoint가 선언되어 있기 때문에 참조변수 p로 인스턴스의 price와 bonusPoint를 사용할 수 있기에 이와 같이 할 수 있다.

앞으로 다른 제품 클래스를 추가할때 Product클래스를 상속받도록 하기만 하면,

buy(Product p)메서드의 매개변수로 받아들여질 수 있다.

```
Buyer b = new Buyer();
Tv t = new Tv();
Computer c = new Computer();
b.buy(t);
b.buy(c);
```

Tv클래스와 Computer클래스는 Product클래스의 자손이므로 위의 코드에서처럼, buy(Product p)메서드에 매개변수로 Tv인스턴스와 Computer인스턴스를 제공하는 것이 가능하다.

#### [예제7-20] PolyArgumentTest.java

```
class Product {
    int price;          // 제품의 가격
    int bonusPoint;     // 제품구매 시 제공하는 보너스점수
    Product(int price) {
        this.price = price;
        bonusPoint =(int)(price/10.0);    // 보너스점수는 제품가격의 10%
    }
}

class Tv extends Product {
    Tv() {
        // 조상클래스의 생성자 Product(int price)를 호출한다.
        super(100);          // Tv의 가격을 100만원으로 한다.
    }

    public String toString() {    // Object클래스의 toString()을 오버라이딩한다.
        return "Tv";
    }
}
```



```

class Computer extends Product {
    Computer() {
        super(200);
    }

    public String toString() {
        return "Computer";
    }
}

class Buyer {           // 고객, 물건을 사는 사람
    int money = 1000;    // 소유금액
    int bonusPoint = 0;  // 보너스점수

    void buy(Product p) {
        if(money < p.price) {
            System.out.println("잔액이 부족하여 물건을 살수 없습니다.");
            return;
        }
        money -= p.price;    // 가진 돈에서 구입한 제품의 가격을 뺀다.
        bonusPoint += p.bonusPoint;  // 제품의 보너스 점수를 추가한다.
        System.out.println(p + "을/를 구입하셨습니다.");
    }
}

class PolyArgumentTest {
    public static void main(String args[]) {
        Buyer b = new Buyer();
        Tv tv = new Tv();
        Computer com = new Computer();

        b.buy(tv);
        b.buy(com);
    }
}

```

```

        System.out.println("현재 남은 돈은 " + b.money + "만원입니다.");
        System.out.println("현재 보너스점수는 " + b.bonusPoint + "점입니다.");
    }
}

```

#### [실행결과]

Tv을/를 구입하셨습니다.

Computer을/를 구입하셨습니다.

현재 남은 돈은 700만원입니다.

현재 보너스점수는 30점입니다.

고객(Buyer)이 buy(Product p)메서드를 이용해서 제품, Tv와 Computer를 구입하고, 고객의 잔고와 보너스점수를 출력하는 예제이다.

한 가지 예를 더 들어 PrintStream클래스에 정의되어있는 print(Object o)메서드를 살펴보자. 매개변수로 Object타입의 변수를 선언하였다. Object클래스는 모든 클래스의 조상이므로 이 메서드의 매개변수로 어떤 타입의 인스턴스도 가능하므로, 이 하나의 메서드로 모든 타입의 인스턴스를 처리할 수 있는 것이다.

이 메서드는 o.toString()을 호출하여 문자열을 얻은 다음 문자열을 출력하는 일을 한다. 실제 코드는 아래와 같다.

```

public void print(Object obj) {
    write(String.valueOf(obj));
}

public static String valueOf(Object obj) {
    return (obj == null) ? "null" : obj.toString();
}

```

## 5.7 여러 종류의 객체를 하나의 배열로 다루기

조상타입의 참조변수로 자손타입의 객체를 참조하는 것이 가능하므로, Product클래스가 Tv, Computer, Audio클래스의 조상일 때, 다음과 같이 할 수 있는 것을 이미 배웠다.

```
Product p1 = new Tv();
Product p2 = new Computer();
Product p3 = new Audio();
```

위의 코드를 Product타입의 참조변수 배열로 하면 아래와 같다.

```
Product p[] = new Product[3];
p[0] = new Tv();
p[1] = new Computer();
p[2] = new Audio();
```

이처럼 조상타입의 참조변수 배열을 사용하면, 공통의 조상을 가진 서로 다른 종류의 객체를 배열로 묶어서 다룰 수 있다.

또는 묶어서 다루기를 원하는 객체들의 상속관계를 따져서 가장 가까운 공통조상 클래스 타입의 참조변수 배열을 생성해서 객체들을 저장하면 된다.

이러한 특징을 이용해서 예제 PolyArgumentTest.java의 Buyer클래스에 구입한 제품을 저장하기 위한 Product배열을 추가해보도록 하자.

```
class Buyer {
    int money = 1000;
    int bonusPoint = 0;
    Product item[] = new Product[10];    // 구입한 제품을 저장하기 위한 배열
    int i = 0;                          // Product배열 item에 사용될 index
```

```

void buy(Product p) {
    if(money < p.price) {
        System.out.println("잔액이 부족하여 물건을 살수 없습니다.");
        return;
    }
    money -= p.price;
    bonusPoint += p.bonusPoint;
    item[i++] = p;          // 구입한 제품을 Product배열인 item에 저장한다.
    System.out.println(p + "을/를 구입하셨습니다.");
}
}

```

구입한 제품을 담기 위해 Buyer클래스에 Product배열인 item을 추가해주었다. 그리고 buy메서드에 item[i++] = p;문장을 추가함으로써 물건을 구입하면, 배열 item에 저장되도록 했다. 이렇게 함으로써, 모든 제품클래스의 조상인 Product클래스 타입의 배열을 사용함으로써 구입한 제품을 하나의 배열로 간단하게 다룰 수 있게 된다.

#### [예제7-21] PolyArgumentTest2.java

```

class Product {
    int price;          // 제품의 가격
    int bonusPoint;     // 제품구매 시 제공하는 보너스점수
    Product(int price) {
        this.price = price;
        bonusPoint =(int)(price/10.0);
    }

    Product() {
        price = 0;
        bonusPoint = 0;
    }
}

class Tv extends Product {

```

```

    Tv() {
        // 조상클래스의 생성자 Product(int price)를 호출한다.
        super(100);
    }

    public String toString() {
        return "Tv";
    }
}

class Computer extends Product {
    Computer() {
        super(200);
    }

    public String toString() {
        return "Computer";
    }
}

class Audio extends Product {
    Audio() {
        super(50);
    }

    public String toString() {
        return "Audio";
    }
}

class Buyer {
    // 고객, 물건을 사는 사람
    int money = 1000;    // 소유금액
    int bonusPoint = 0;  // 보너스점수
    Product item[] = new Product[10];    // 구입한 제품을 저장하기 위한 배열
    int i=0;            // Product배열에 사용될 카운터
}

```

```

void buy(Product p) {
    if(money < p.price) {
        System.out.println("잔액이 부족하여 물건을 살수 없습니다.");
        return;
    }

    money -= p.price;    // 가진 돈에서 구입한 제품의 가격을 뺀다.
    bonusPoint += p.bonusPoint;    // 제품의 보너스 점수를 추가한다.
    item[i++] = p;        // 구입한 제품을 Product배열인 item에 저장한다.
    System.out.println(p + "을/를 구입하셨습니다.");
}

void summary() {        // 구매한 물품에 대한 정보를 요약해서 보여 준다.
    int sum = 0;        // 구입한 물품의 가격합계
    String itemList = ""; // 구입한 물품목록
    // 반복문을 이용해서 구입한 물품의 총 가격과 목록을 만든다.

    for(int i=0; i < item.length; i++) {
        if(item[i]==null) break;
        sum += item[i].price;
        itemList += item[i] + ", ";
    }

    System.out.println("구입하신 물품의 총금액은 " + sum + "만원입니다.");
    System.out.println("구입하신 제품은 " + itemList + "입니다.");
}
}

class PolyArgumentTest2 {
    public static void main(String args[]) {
        Buyer b = new Buyer();
        Tv tv = new Tv();
        Computer com = new Computer();
        Audio audio = new Audio();

        b.buy(tv);
    }
}

```

```

        b.buy(com);
        b.buy(audio);
        b.summary();
    }
}

```

#### [실행결과]

Tv을/를 구입하셨습니다.  
 Computer을/를 구입하셨습니다.  
 Audio을/를 구입하셨습니다.  
 구입하신 물품의 총금액은 350만원입니다.  
 구입하신 제품은 Tv, Computer, Audio, 입니다.

[참고] 구입한 제품목록의 마지막에 출력되는 콤마(,)가 눈에 거슬린다면, `itemList += item[i]` + ", ";를 `itemList += (i==0) ? "" + item[i] : ", " + item[i];`과 같이 변경하자. 보다 깔끔한 결과를 얻을 수 있을 것이다.

위 예제에서 Product배열로 구입한 제품들을 저장할 수 있도록 하고 있지만, 배열의 크기를 10으로 했기 때문에 10개 이상의 제품을 구입할 수 없는 것이 문제다. 그렇다고 해서 배열의 크기를 무조건 크게 설정할 수만은 없는 일이다.

이런 경우, Vector클래스를 사용하면 된다. Vector 클래스는 내부적으로 Object타입의 배열을 가지고 있어서, 이 배열에 객체를 추가하거나 제거할 수 있게 작성되어 있다. 그리고, 배열의 크기를 동적으로 관리해주기 때문에 저장할 인스턴스의 개수에 신경 쓰지 않아도 된다.

```

public class Vector extends AbstractList implements List, Cloneable,
    java.io.Serializable {
    protected Object elementData[];
    ...
}

```

[참고] Vector클래스는 이름 때문에 클래스의 기능을 오해할 수 있는데, 단지 동적으로 크기가 관리되는 객체배열이라고 생각하면 된다.

Vector클래스의 주요 메서드는 다음과 같다.

메서드 / 생성자	설 명
Vector()	10개의 객체를 저장할 수 있는 Vector인스턴스를 생성한다. 10개 이상의 인스턴스가 저장되면, 자동적으로 크기가 증가된다.
boolean add(Object o)	Vector에 객체를 추가한다. 추가에 성공하면 결과값으로 true를, 실패하면 false를 반환한다.
boolean remove(Object o)	Vector에 저장되어 있는 객체를 제거한다. 제거에 성공하면 true를, 실패하면 false를 반환한다.
boolean isEmpty()	Vector가 비어있는지 검사한다. 비어있으면 true, 비어있지 않으면 false를 반환한다.
Object get(int index)	지정된 위치(index)의 객체를 반환한다. 반환타입이 Object 타입이므로 적절한 타입으로의 형변환이 필요하다.
int size()	Vector에 저장된 객체의 개수를 반환한다.

[표7-1] Vector클래스의 주요메서드

#### [예제7-22] PolyArgumentTest3.java

```
import java.util.*;           // Vector클래스를 사용하기 위해서 추가해주었다.

class Tv extends Product {
    Tv() { super(100); }
    public String toString() { return "Tv"; }
}

class Computer extends Product {
    Computer() { super(200); }
    public String toString() { return "Computer"; }
}

class Audio extends Product {
    Audio() { super(50); }
    public String toString() { return "Audio"; }
}

class Buyer {                // 고객, 물건을 사는 사람
```



```

int money = 1000;    // 소유금액
int bonusPoint = 0;  // 보너스점수
Vector item = new Vector();    // 구입한 제품을 저장하는데 사용될 Vector객체

void buy(Product p) {
    if(money < p.price) {
        System.out.println("잔액이 부족하여 물건을 살수 없습니다.");
        return;
    }
    money -= p.price;        // 가진 돈에서 구입한 제품의 가격을 뺀다.
    bonusPoint += p.bonusPoint;    // 제품의 보너스 점수를 추가한다.
    item.add(p);            // 구입한 제품을 Vector에 저장한다.
    System.out.println(p + "을/를 구입하셨습니다.");
}

void refund(Product p) {    // 구입한 제품을 환불한다.
    if(item.remove(p)) {    // 제품을 Vector에서 제거한다.
        money += p.price;
        bonusPoint -= p.bonusPoint;
        System.out.println(p + "을/를 반품하셨습니다.");
    } else {                // 제거에 실패한 경우
        System.out.println("구입하신 제품 중 해당 제품이 없습니다.");
    }
}

void summary() {            // 구매한 물품에 대한 정보를 요약해서 보여준다.
    int sum = 0;            // 구입한 물품의 가격합계
    String itemList = "";   // 구입한 물품목록
    // 반복문을 이용해서 구입한 물품의 총 가격과 목록을 만든다.

    if(item.isEmpty()) {    // Vector가 비어있는지 확인한다.
        System.out.println("구입하신 제품이 없습니다.");
        return;
    }
}

```

```

        for(int i=0; i < item.size();i++) {
            Product p = (Product)item.get(i); // Vector의 i번째에 있는 객체를 얻어 온
다.

            sum += p.price;
            itemList += (i==0) ? "" + p : ", " + p;
        }
        System.out.println("구입하신 물품의 총금액은 " + sum + "만원입니다.");
        System.out.println("구입하신 제품은 " + itemList + "입니다.");
    }
}

class PolyArgumentTest3 {
    public static void main(String args[]) {
        Buyer b = new Buyer();
        Tv tv = new Tv();
        Computer com = new Computer();
        Audio audio = new Audio();

        b.buy(tv);
        b.buy(com);
        b.buy(audio);
        b.summary();
        System.out.println();
        b.refund(com);
        b.summary();
    }
}

```

#### [실행결과]

Tv을/를 구입하셨습니다.  
 Computer을/를 구입하셨습니다.  
 Audio을/를 구입하셨습니다.  
 구입하신 물품의 총금액은 350만원입니다.  
 구입하신 제품은 Tv, Computer, Audio입니다.

Computer을/를 반품하셨습니다.

구입하신 물품의 총금액은 150만원입니다.

구입하신 제품은 Tv, Audio입니다.

Vector클래스를 사용하기 위해서 예제의 첫째 줄에 `import java.util.*;`를 추가해주었다.

그리고, 구입한 물건을 다시 반환할 수 있도록 `refund(Product p)`를 추가하였다. 이 메서드가 호출되면, 구입물품이 저장되어 있는 `item`에서 해당제품을 제거한다.

# [Java의 정석]제7장 객체지향개념 2 - 6. 추상클래스(Abstract class)

자바의정석

2012/12/22 17:11

<http://blog.naver.com/gphic/50157740199>

## 6. 추상클래스(Abstract class)

### 6.1 추상클래스란?

클래스를 설계도에 비유한다면, 추상클래스는 미완성 설계도에 비유할 수 있다. 미완성 설계도란, 단어의 뜻 그대로 완성되지 못한 채로 남겨진 설계도를 말한다.

클래스가 미완성이라는 것은 멤버의 개수에 관계된 것이 아니라, 단지 미완성 메서드(추상메서드)를 포함하고 있다는 의미이다.

미완성 설계도로 완성된 제품을 만들 수 없듯이 추상클래스로는 인스턴스는 생성할 수 없다.

추상클래스는 상속을 통해서 자손클래스에 의해서 완성될 수 있다.

추상클래스 자체로는 클래스로서의 역할을 다 못하지만, 새로운 클래스를 작성하는데 있어서 바탕이 되는 조상클래스로서 중요한 의미를 갖는다.

추상클래스는 키워드 'abstract'를 붙이기만 하면 된다. 이렇게 함으로써 이 클래스를 사용할 때, 클래스 선언부의 abstract를 보고 이 클래스에는 추상메서드가 있으니 상속을 통해서 구현 해주어야 한다는 것을 쉽게 알 수 있을 것이다.

```
abstract class 클래스이름 {  
    // ...  
}
```

추상클래스는 추상메서드를 포함하고 있다는 것을 제외하고는 일반클래스와 전혀 다르지 않다. 추상클래스에도 생성자가 있으며, 멤버변수와 일반 메서드도 가질 수 있다.

[참고]추상메서드를 포함하고 있지 않은 클래스에도 키워드 'abstract'를 붙여서 추상클래스로 지정할 수도 있다. 추상메서드가 없는 완성된 클래스라 할지라도 추상클래스로 지정되면 클래스의 인스턴스를 생성할 수 없다.

## 6.2 추상메서드(abstract 메서드)

메서드는 선언부와 구현부(몸통)로 구성되어 있다고 했다. 선언부만 작성하고 구현부는 작성하지 않은 체로 남겨 둔 것이 추상메서드이다. 즉, 설계만 해 놓고 실제 수행될 내용은 작성하지 않기 때문에 미완성 메서드인 것이다.

메서드를 이와 같이 미완성 상태로 남겨 놓는 이유는 메서드의 내용이 상속받는 클래스에 따라 달라질 수 있기 때문에 조상클래스에서는 선언부만을 작성하고, 주석을 덧붙여 어떤 기능을 수행할 목적으로 작성되었는지 알려 주고, 실제 내용은 상속받는 클래스에서 구현하도록 비워두는 것이다.

그래서, 추상클래스를 상속받는 자손클래스는 조상의 추상메서드를 상황에 맞게 적절히 구현해주어야 한다.

추상메서드 역시 키워드 'abstract'를 앞에 붙여 주고, 추상메서드는 구현부가 없으므로 괄호{} 대신 문장의 끝을 알리는 ';'을 적어준다.

```
/* 주석을 통해 어떤 기능을 수행할 목적으로 작성하였는지 설명한다. */  
abstract 리턴타입 메서드이름();
```

추상클래스로부터 상속받는 자손클래스는 오버라이딩을 통해 조상인 추상클래스의 추상메서드를 모두 구현해주어야 한다. 만일 조상으로부터 상속받은 추상메서드 중 하나라도 구현하지 않는다면, 자손클래스 역시 추상클래스로 지정해 주어야 한다.

실제 작업내용인 구현부가 없는 메서드가 무슨 의미가 있을까 싶기도 하겠지만, 메서드를 작성할 때 실제 작업내용인 구현부 보다 더 중요한 부분이 선언부이다.

메서드의 이름과 메서드의 작업에 필요한 매개변수, 그리고 작업의 결과로 어떤 타입의 값을 반환할 것인가를 결정하는 것은 쉽지 않은 일이다. 선언부만 작성해도 메서드의 절반 이상이 완성된 것이라 해도 과언이 아니다.

메서드를 사용하는 쪽에서는 메서드가 실제로 어떻게 구현되어있는지 보다는 메서드의 이름과 매개변수, 리턴타입, 즉 선언부만 알고 있으면 되므로 내용이 없을 지라도 추상메서드를 사용하는 코드를 작성하는 것이 가능하며, 실제로는 자손클래스에 구현된 완성된 메서드가 호출

되도록 할 수 있다.

### 6.3 추상클래스의 작성

여러 클래스에 공통적으로 사용될 수 있는 클래스를 바로 작성하기도 하고, 기존의 클래스의 공통적인 부분을 뽑아서 추상클래스로 만들어 상속하도록 하는 경우도 있다.

참고로 추상의 사전적 정의는 다음과 같다.

추상[抽象] - 낱말의 구체적 표상(表象)이나 개념에서 공통된 성질을 뽑아 이를 일반적인 개념으로 파악하는 정신 작용

상속이 자손클래스를 만드는데 조상클래스를 사용하는 것이라면, 추상화는 기존의 클래스의 공통부분을 뽑아 내서 조상클래스를 만드는 것이라고 할 수 있다.

추상화를 구체화와 반대되는 의미로 이해하면 보다 쉽게 이해할 수 있을 것이다. 상속계층도를 따라 내려갈 수록 클래스는 점점 기능이 추가되어 구체화의 정도가 심해지며, 상속계층도를 따라 올라갈 수록 클래스는 추상화의 정도가 심해진다고 할 수 있다.

즉, 상속계층도를 따라 내려 갈수록 세분화되며, 올라갈수록 공통요소만 남게 된다.

추상화 - 클래스간의 공통점을 찾아내서 공통의 조상을 만드는 작업  
구체화 - 상속을 통해 클래스를 구현, 확장하는 작업

여러 클래스에 널리 사용될 수 있는 Player라는 추상클래스를 작성해 보았다. 이 클래스는 VCR이나 Audio와 같은 재생 가능한(Player) 기기를 클래스로 작성할 때, 이 둘의 조상클래스로 사용될 수 있을 것이다.

```

abstract class Player {
    boolean pause;        // 일시정지상태를 저장하기 위한 변수
    int currentPos;       // 현재 Play되고 있는 위치를 저장하기 위한 변수

    Player() {            // 추상클래스도 생성자가 있다.
        pause = false;
        currentPos = 0;
    }

    /** 지정된 위치(pos)에서 재생을 시작하는 기능이 수행하도록 작성되어야 한다. */
    abstract void play(int pos);    // 추상메서드

    /** 재생을 즉시 멈추는 기능을 수행하도록 작성되어야 한다. */
    abstract void stop();          // 추상메서드

    void play() {
        play(currentPos);          // 추상메서드를 사용할 수 있다.
    }

    void pause() {
        if(pause) {               // pause가 true일 때(정지상태)에서 pause가 호출되면,
            pause = false;         // pause의 상태를 false로 바꾸고,
            play(currentPos);       // 현재의 위치에서부터 play를 한다.
        } else {                  // pause가 false일 때(play상태)에서 pause가 호출되면,
            pause = true;          // pause의 상태를 true로 바꾸고
            stop();                // play를 멈춘다.
        }
    }
}

```

[참고] /\*\* \*/ 주석은 javadoc.exe를 사용해서 Java API와 같은 클래스의 멤버에 대한 설명을 담은 문서를 만드는데 사용된다.

이제 Player클래스를 조상으로 하는 CDPlayer 클래스를 만들어 보자.

```

class CDPlayer extends Player {

```

```

// 조상 클래스의 추상메서드를 구현한다.
void play(int currentPos) {
    /* 실제구현 내용 생략 */
}

void stop() {
    /* 실제구현 내용 생략 */
}

// CDPlayer클래스에 추가로 정의된 멤버
int currentTrack; // 현재 재생 중인 트랙

void nextTrack() {
    currentTrack++;
    //...
}

void preTrack() {
    if(currentTrack > 0) {
        currentTrack--;
    }
    //...
}
}

```

조상클래스의 추상메서드를 CDPlayer클래스의 기능에 맞게 완성해주고, CDPlayer만의 새로운 기능들을 추가하였다.

사실 Player클래스의 play(int pos)와 stop()을 추상메서드로 하는 대신, 아무 내용도 없는 메서드로 작성할 수도 있다. 아무런 내용도 없이 단지 괄호{}만 있을지라도, 추상메서드가 아닌 일반메서드로 간주되기 때문이다.

```

class Player {

```



```

...
void play(int pos) {}
void stop() {}
...
}

```

어차피 자손클래스에서 오버라이딩하여 자신의 클래스에 맞게 구현할 테니 추상메서드로 선언하는 것과 내용없는 빈 몸통만 만들어 놓는 것이나 별 차이가 없어 보인다.

그래도 굳이 abstract를 붙여서 추상메서드로 선언하는 이유는 자손클래스에서 추상메서드를 반드시 구현하도록 강요하기 위해서이다.

만일 추상메서드로 정의되어 있지 않고 빈 몸통만 가지도록 정의되어 있다면, 상속받는 자손클래스에서는 이 메서드들이 온전히 구현된 것으로 인식하고 오버라이딩을 통해 자신의 클래스에 맞도록 구현하지 않을 수도 있기 때문이다.

하지만 abstract를 사용해서 추상메서드로 정의해놓으면, 자손클래스를 작성할 때 이들이 추상메서드이므로 내용을 새로 구현해주어야 한다는 사실을 인식하고 자신의 클래스에 알맞게 구현할 것이다.

이번엔 기존의 클래스로부터 공통된 부분을 뽑아 내어 추상클래스를 만들어 보도록 하자.

```

class Marine {           // 보병
    int x, y;             // 현재 위치
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void stop() {         /* 현재 위치에 정지 */ }
    void stimPack() {     /* 스팀팩을 사용한다. */ }
}

class Tank {             // 탱크
    int x, y;             // 현재 위치
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void stop() {         /* 현재 위치에 정지 */ }
    void changeMode() {   /* 공격모드를 변환한다. */ }
}

```

```

}

class Dropship {           // 수송선
    int x, y;               // 현재 위치
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void stop() {          /* 현재 위치에 정지 */ }
    void load() { /* 선택된 대상을 태운다. */ }
    void unload() { /* 선택된 대상을 태운다. */ }
}

```

유명한 컴퓨터게임인 Starcraft에 나오는 유닛들을 클래스로 간단히 정의해보았다. 이 유닛들은 각자 나름대로의 기능을 가지고 있지만 공통부분을 뽑아 내어 하나의 클래스로 만들고, 이 클래스로부터 상속받도록 변경해보자.

```

abstract class Unit {
    int x, y;
    abstract void move(int x, int y);
    void stop() {          /* 현재 위치에 정지 */ }
}

class Marine extends Unit {    // 보병
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void stimPack() {        /* 스팀팩을 사용한다. */ }
}

class Tank extends Unit { // 탱크
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
    void changeMode() {      /* 공격모드를 변환한다. */ }
}

class Dropship extends Unit { // 수송선
    void move(int x, int y) { /* 지정된 위치로 이동 */ }
}

```

```

void load() { /* 선택된 대상을 태운다.*/ }
void unload() { /* 선택된 대상을 내린다.*/ }
}

```

각 클래스의 공통부분을 뽑아 내서 Unit클래스를 정의하고 이로부터 상속받도록 하였다. 이 Unit클래스는 다른 유닛을 위한 클래스를 작성하는데 재활용될 수 있을 것이다. 이들 클래스에 대해서 stop메서드는 선언부와 구현부 모두 공통적이지만, Marine, Tank는 지상유닛이고 Dropship는 공중유닛이기 때문에 이동하는 방법이 서로 달라서 move메서드의 실제 구현 내용이 다를 것이다.

그 래도 move메서드의 선언부는 같기 때문에 추상메서드로 정의할 수 있다. 최대한의 공통부분을 뽑아 내기 위한 것이기도 하지만, 모든 유닛은 이동할 수 있어야 하기 때문에 Unit클래스에는 move메서드가 반드시 필요한 것이기 때문이다. move메서드가 추상메서드로 선언된 것에는, 앞으로 Unit클래스를 상속받아서 작성되는 클래스는 move메서드를 자신의 클래스에 알맞게 반드시 구현해야한다는 의미가 담겨 있는 것이기도 하다.

```

Unit[] group = new Unit[5];
group[0] = new Marine();
group[1] = new Tank();
group[2] = new Marine();
group[3] = new Dropship();
group[4] = new Marine();

for(int i=0;i< group.length;i++) {
    // Unit배열의 모든 유닛을 좌표(100, 200)의 위치로 이동한다.
    group[i].move(100, 200);
}

```

위의 코드는 공통조상인 Unit클래스 타입의 참조변수 배열을 통해서 서로 다른 종류의 인스턴스를 하나의 묶음으로 다룰 수 있다는 것을 보여 주기 위한 것이다. 다형성에서 배웠듯이 조상클래스타입의 참조변수로 자손클래스의 인스턴스를 참조하는 것이

가능하기 때문에 이처럼 조상클래스타입의 배열에 자손클래스의 인스턴스를 담을 수 있는 것이다.

만 일 이들 클래스간의 공통조상이 없었다면 이처럼 하나의 배열로 다룰 수 없었을 것이다. Unit클래스에 move메서드가 비록 추상메서드로 정의되어 있다 하더라도 이처럼 Unit클래스 타입의 참조변수로 move메서드를 호출하는 것이 가능하다. 메서드는 참조변수의 타입에 관계 없이 실제 인스턴스에 구현된 것이 호출되기 때문이다.

group[i].move(100, 200)과 같이 호출하는 것이 Unit클래스의 추상메서드인 move를 호출하는 것 같이 보이지만 실제로는 이 추상메서드가 구현된 Marine, Tank, Dropship인스턴스의 메서드가 호출되는 것이다.

모든 클래스의 조상인 Object클래스 타입의 배열로도 서로 다른 종류의 인스턴스를 하나의 묶음으로 다룰 수 있지만, Object클래스에는 move메서드가 정의되어 있지 않기 때문에 move 메서드를 호출하는 부분에서 에러가 발생한다.

```
Object[] group = new Object[5];
group[0] = new Marine();
group[1] = new Tank();
group[2] = new Marine();
group[3] = new Dropship();
group[4] = new Marine();

for(int i=0;i < group.length;i++) {
    // 에러!!! Object클래스에는 move메서드가 정의되어 있지 않다.
    group[i].move(100, 200);
}
```

# [Java의 정석]제7장 객체지향개념 2 - 7. 인터페이스(Interface)

자바의정석

2012/12/22 17:12

<http://blog.naver.com/gphic/50157740226>

## 7. 인터페이스(interface)

### 7.1 인터페이스란?

인터페이스는 일종의 추상클래스이다. 인터페이스는 추상클래스처럼 추상메서드를 갖지만 추상클래스보다 추상화 정도가 높아서 추상클래스와 달리 몸통을 갖춘 일반 메서드 또는 멤버변수를 구성원으로 가질 수 없다.

오직 추상메서드와 상수만을 멤버로 가질 수 있으며, 그 외의 다른 어떠한 요소도 허용하지 않는다.

추상클래스를 부분적으로만 완성된 '미완성 설계도'라고 한다면, 인터페이스는 구현된 것은 아무 것도 없고 밑그림만 그려져 있는 '기본 설계도'라 할 수 있다.

추상클래스처럼 인터페이스도 완성되지 않은 불완전한 것이기 때문에 그 자체만으로 사용되기 보다는 다른 클래스를 작성하는데 도움 줄 목적으로 작성된다.

### 7.2 인터페이스의 작성

인터페이스를 작성하는 것은 클래스를 작성하는 것과 같다. 다만 키워드로 class 대신 interface를 사용한다는 것만 다르다. 그리고 interface에도 클래스와 같이 접근제어자로 public 또는 default를 사용할 수 있다.

```
interface 인터페이스이름 {  
    public static final 타입 상수이름 = 값;  
    public abstract 메서드이름(매개변수목록);
```

```
}
```

일반적인 클래스의 멤버들과 달리 인터페이스의 멤버들은 다음과 같은 제약사항을 가지고 있다.

- 모든 멤버변수는 public static final 이어야 하며, 이를 생략할 수 있다.
- 모든 메서드는 public abstract 이어야 하며, 이를 생략할 수 있다.

[참고]인터페이스에 정의된 모든 멤버에 예외없이 적용되는 사항이기 때문에 제어자를 생략할 수 있는 것이며, 편의상 생략하는 경우가 많다. 생략된 제어자는 컴파일시에 컴파일러가 자동적으로 추가해준다.

```
interface PlayingCard {  
    public static final int SPADE = 4;  
    final int DIAMOND = 3;        // public static final int DIAMOND=3;  
    static int HEART = 2;         // public static final int HEART=2;  
    int CLOVER = 1;               // public static final int CLOVER=1;  
  
    public abstract String getCardNumber();  
    String getCardKind();        // public abstract String getCardKind();로 간주된다.  
}
```

### 7.3 인터페이스의 상속

interface는 클래스가 아닌 인터페이스로부터만 상속받을 수 있으며, 클래스와는 달리 다중상속, 즉 여러 개의 인터페이스로부터 상속을 받는 것이 가능하다.

[참고]클래스와는 달리 Object클래스와 같은 모든 인터페이스의 최고조상은 없다.

```
interface Movable {  
    /** 지정된 위치(x, y)로 이동하는 기능의 메서드 */  
    void move(int x, int y);  
}  
  
interface Attackable {  
    /** 지정된 대상(u)을 공격하는 기능의 메서드 */  
    void attack(Unit u);  
}  
  
interface Fightable extends Movable, Attackable { }
```

클래스에서의 상속과 마찬가지로 자손인터페이스(Fightable)은 조상인터페이스(Movable, Attackable)에 정의된 멤버를 모두 상속받는다.

그래서 Fightable자체에는 정의된 멤버가 하나도 없지만 조상인터페이스로 부터 상속받은 두 개의 추상메서드, move(int x, int y)와 attack(Unit u)를 멤버로 갖게 된다.

#### 7.4 인터페이스의 구현

추상클래스에서와 같이 인터페이스의 추상메서드를 몸통을 구현하는 자손클래스를 작성해야 한다. 이때는 인터페이스를 '구현(implement)한다'고 하며 키워드 implements를 사용한다.

```
class 클래스이름 implements 인터페이스이름 {  
    // 인터페이스에 정의된 추상메서드를 구현해야한다.  
}
```

```
class Fighter implements Fightable {
    public void move() { /* 내용 생략*/ }
    public void attack() { /* 내용 생략*/ }
}
```

[참고]이 때 'Fighter클래스는 Fightable인터페이스를 구현한다.'라고 한다.

만일 구현하는 인터페이스의 메서드 중 일부만 구현한다면, 추상클래스로 선언되어야 한다.

```
abstract class Fighter implements Fightable {
    public void move() { /* 내용 생략*/ }
}
```

그리고 다음과 같이 상속과 구현을 동시에 할 수도 있다.

```
class Fighter extends Unit implements Fightable {
    public void move(int x, int y) { /* 내용 생략 */ }
    public void attack(Unit u) { /* 내용 생략 */ }
}
```

[참고]인 터페이스의 이름에는 주로 Fightable과 같이 '~를 할 수 있는'의 의미인 'able'로 끝나는 것들이 많은데, 그 이유는 어떠한 기능 또는 행위를 하는데 필요한 메서드를 제공한다는 의미를 강조하기 위해서이다. 또한 그 인터페이스를 구현한 클래스는 '~를 할 수 있는' 능력을 갖추었다는 의미이기도 하다.

이름이 'able'로 끝나는 것은 인터페이스라고 쉽게 추측할 수 있지만, 모든 인터페이스의 이름이 반드시 'able'로 끝나야 하는 것은 아니다.

#### [예제 7-23] FighterTest.java

```
class FighterTest
```



```

{
    public static void main(String[] args)
    {
        Fighter f = new Fighter();

        if (f instanceof Unit)    {
            System.out.println("f는 Unit클래스의 자손입니다.");
        }
        if (f instanceof Fightable) {
            System.out.println("f는 Fightable인터페이스를 구현했습니다.");
        }
        if (f instanceof Movable) {
            System.out.println("f는 Movable인터페이스를 구현했습니다.");
        }
        if (f instanceof Attackable) {
            System.out.println("f는 Attackable인터페이스를 구현했습니다.");
        }
        if (f instanceof Object) {
            System.out.println("f는 Object클래스의 자손입니다.");
        }
    }
}

class Fighter extends Unit implements Fightable {
    public void move(int x, int y) { /* 내용 생략 */}
    public void attack(Unit u) { /* 내용 생략 */}
}

class Unit {
    int currentHP;        // 유닛의 체력
    int x;                // 유닛의 위치(x좌표)
    int y;                // 유닛의 위치(y좌표)
}

interface Fightable extends Movable, Attackable {}
interface Movable {    void move(int x, int y);    }

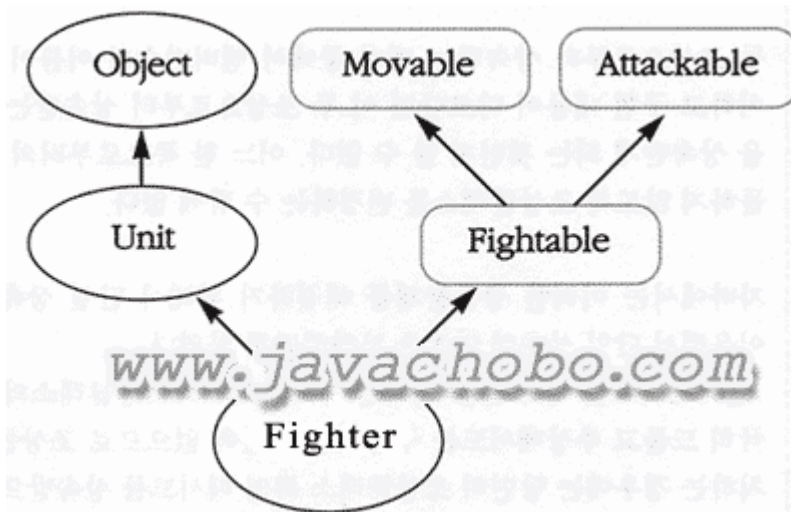
```

```
interface Attackable {    void attack(Unit u); }
```

#### [실행결과]

f는 Unit클래스의 자손입니다.  
f는 Fightable인터페이스를 구현했습니다.  
f는 Movable인터페이스를 구현했습니다.  
f는 Attackable인터페이스를 구현했습니다.  
f는 Object클래스의 자손입니다.

예제에 사용된 클래스와 인터페이스간의 관계를 그려보면 다음과 같다.



실 제로 Fighter클래스는 Unit클래스로부터 상속받고 Fightable인터페이스만을 구현했지만, Unit클래스는 Object클래스의 자손이고, Fightable인터페이스는 Movable과 Attackable인터페이스의 자손이므로 Fighter클래스는 이 모든 클래스와 인터페이스의 자손이 되는 셈이다. 인터페이스는 상속 대신 구현이라는 용어를 사용하지만, 인터페이스로부터 상속받은 추상메서드를 구현하는 것이기 때문에 인터페이스도 조금은 다른 의미의 조상이라고 할 수 있다.

여기서 주의 깊게 봐두어야 할 것은 Movable인터페이스에 정의된 void move(int x, int y)를 Fighter클래스에서 구현할 때 접근제어자를 public으로 했다는 것이다.

```
interface Movable {  
    void move(int x, int y);  
}
```

```
class Fighter extends Unit implements Fightable {
    public void move(int x, int y) { /* 실제구현내용 생략 */}
    public void attack(Unit u) { /* 실제구현내용 생략 */}
}
```

오버라이딩을 할 때는 조상의 메서드보다 넓은 범위의 접근제어자를 지정해야한다는 것을 기억할 것이다. Movable인터페이스에 void move(int x, int y)와 같이 정의되어 있지만 사실 public abstract가 생략된 것이기 때문에 실제로 public abstract void move(int x, int y)이다.

따라서 이를 구현하는 Fighter클래스에서는 void move(int x, int y)의 접근제어자를 반드시 public으로 해야 하는 것이다.

## 7.5 인터페이스를 이용한 다중상속

두 조상으로부터 상속받는 멤버 중에서 멤버변수의 이름이 같거나 메서드의 선언부가 일치하고 구현 내용이 다르다면 이 두 조상으로부터 상속받는 자손클래스는 어느 조상의 것을 상속받게 되는 것인지 알 수 없다. 어느 한 쪽으로부터의 상속을 포기하던가, 이름이 충돌하지 않도록 조상클래스를 변경하는 수 밖에는 없다.

자바에서는 이러한 충돌문제를 해결하기 위해서 단일 상속만을 허용하고, 인터페이스를 이용해서 단일 상속의 단점을 보완하도록 하였다.

인터페이스는 상수만을 정의할 수 있으므로 조상클래스의 멤버변수와 충돌하는 경우는 극히 드물고 추상메서드는 구현내용이 전혀 없으므로 조상클래스의 메서드와 선언부가 일치하는 경우에는 당연히 조상클래스 쪽의 메서드를 상속받으면 되므로 문제되지 않는다. 이렇게 해서 상속받는 멤버의 충돌은 피할 수 있지만, 다중상속의 장점을 잃게 된다는 것이 아쉽다.

만 일 두 개의 클래스로부터 상속을 받아야 할 상황이라면, 두 조상클래스 중에서 비중이 높은 쪽을 선택하고 다른 한쪽은 클래스 내부에 멤버로 포함시키는 방식으로 처리하거나 어느 한쪽

을 필요한 부분을 뽑아서 인터페이스로 만든 다음 구현하도록 한다.

다음과 같이 Tv클래스와 VCR클래스가 있을 때, TVCR클래스를 작성하기 위해 두 클래스로부터 상속을 받을 수만 있으면 좋겠지만 다중상속을 허용하지 않으므로, 한 쪽만 선택하여 상속 받고 나머지 한 쪽은 클래스 내에 포함시켜서 내부적으로 인스턴스를 생성해서 사용하도록 한다.

```
public class Tv {
    protected boolean power;
    protected int channel;
    protected int volume;
    public void power() { power = ! power;}
    public void channelUp() { channel++;}
    public void channelDown() { channel--;}
    public void volumeUp() { volume++;}
    public void volumeDown() { volume--;}
}

public class VCR {
    protected int counter;          // VCR의 카운터
    public void play() {
        // Tape을 재생한다.
    }
    public void stop() {
        // 재생을 멈춘다.
    }
    public void reset() {
        counter =0;
    }
    public int getCounter() {
        return counter;
    }
    public void setCounter(int c) {
        counter = c;
    }
}
```

```

    }
}

```

VCR클래스에 정의된 메서드와 일치하는 추상메서드를 갖는 인터페이스를 작성한다.

```

public interface IVCR {
    public void play();
    public void stop();
    public void reset();
    public int getCounter();
    public void setCounter(int c);
}

```

이제 IVCR 인터페이스를 구현하고 Tv클래스로부터 상속받는 TVCR클래스를 작성한다. 이때 VCR클래스 타입의 참조변수를 멤버변수로 선언하여 IVCR인터페이스의 추상메서드를 구현하는데 사용한다.

```

public class TVCR extends Tv implements IVCR {
    VCR vcr = new VCR();
    public void play() {
        vcr.play(); // 코드를 작성하는 대신 VCR인스턴스의 메서드를 호출하면 된다.
    }
    public void stop() {
        vcr.stop();
    }
    public void reset() {
        vcr.reset();
    }
    public int getCounter() {
        return vcr.getCounter();
    }
}

```

```

    }
    public void setCounter(int c) {
        vcr.setCounter(c);
    }
}

```

IVCR인터페이스를 구현하기 위해서는 새로 메서드를 작성해야하는 부담이 있지만 이처럼 VCR클래스의 인스턴스를 사용하면 손쉽게 다중상속을 구현할 수 있다.

또한 VCR클래스의 내용이 변경되어도 변경된 내용이 TVCR클래스에도 자동적으로 반영되는 효과도 얻을 수 있다.

사실 인터페이스를 새로 작성하지 않고도 VCR클래스를 TVCR클래스에 포함시키는 것만으로도 충분하지만, 인터페이스를 이용하면 다형적 특성을 이용할 수 있다는 장점이 있다.

## 7.6 인터페이스를 이용한 다형성

다형성에 대해 학습할 때 자손클래스의 인스턴스를 조상타입의 참조변수로 참조하는 것이 가능하다는 것을 배웠다.

인터페이스 역시 이를 구현한 클래스의 조상이라 할 수 있으므로 해당 인터페이스 타입의 참조변수로 이를 구현한 클래스의 인스턴스를 참조할 수 있으며, 인터페이스 타입으로의 형변환도 가능하다.

인터페이스 Fightable을 클래스 Fighter가 구현했을 때, 다음과 같이 Fighter인스턴스를 Fightable타입의 참조변수로 참조하는 것이 가능하다.

```

Fightable f = (Fightable)new Fighter();
또는
Fightable f = new Fighter();

```

[참고]물론 Fightable타입의 참조변수로는 인터페이스 Fightable에 정의된 멤버들만 호출이 가능하다.

따라서 인터페이스는 다음과 같이 메서드의 매개변수의 타입으로 사용될 수 있다.

```
void attack(Fightable f) {  
    //...  
}
```

인터페이스 타입의 매개변수가 갖는 의미는 메서드 호출 시 해당 인터페이스를 구현한 클래스의 인스턴스를 매개변수로 제공해야한다는 것이다.

그래서 attack메서드를 호출할 때는 매개변수로 Fightable인터페이스를 구현한 클래스의 인스턴스를 넘겨주어야 한다.

```
class Fighter extends Unit implements Fightable {  
    public void move(int x, int y) { /* 내용 생략 */}  
    public void attack(Fightable f) { /* 내용 생략 */}  
}
```

위와 같이 Fightable인터페이스를 구현한 Fighter클래스가 있을 때, attack메서드의 매개변수로 Fighter인스턴스를 넘겨 줄 수 있다. 즉, attack(new Fighter())와 같이 할 수 있다는 것이다.

그리고 다음과 같이 메서드의 리턴타입으로 인터페이스의 타입을 지정하는 것 역시 가능하다.

```
Fightable method() {  
    // ...  
    return new Fighter();  
}
```

리턴타입이 인터페이스라는 것은 메서드가 해당 인터페이스를 구현한 클래스의 인스턴스를 반

환한다는 것을 의미한다.

위의 코드에서는 method()의 리턴타입이 Fightable인터페이스기 때문에 메서드의 return문에서Fightable인터페이스를 구현한 Fighter클래스의 인스턴스를 반환한다.

#### [예제7-24] ParserTest.java

```
class ParserTest {
    public static void main(String args[]) {
        Parseable parser = ParserManager.getParser("XML");
        parser.parse("document.xml");
        parser = ParserManager.getParser("HTML");
        parser.parse("document2.html");
    }
}

interface Parseable {
    public abstract void parse(String fileName);    // 구문 분석작업을 수행한다.
}

class ParserManager {
    // 리턴타입이 Parseable인터페이스이다.
    public static Parseable getParser(String type) {
        if(type.equals("XML")) {
            return new XMLParser();
        } else {
            Parseable p = new HTMLParser();
            return p;
            // return new HTMLParser();    위의 두 줄을 간단히 할 수 있다.
        }
    }
}

class XMLParser implements Parseable {
    public void parse(String fileName) {
        /* 구문 분석작업을 수행하는 코드를 적는다. */
    }
}
```



```

        System.out.println(fileName + " - XML parsing completed.");
    }
}

class HTMLParser implements Parseable {
    public void parse(String fileName) {
        /* 구문 분석작업을 수행하는 코드를 적는다. */
        System.out.println(fileName + " - HTML parsing completed.");
    }
}

```

#### [실행결과]

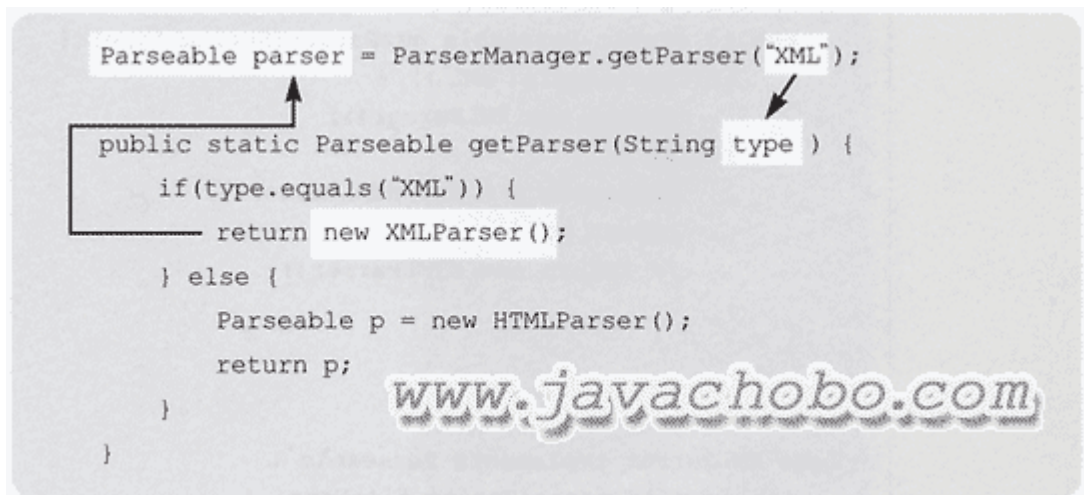
```

document.xml - XML parsing completed.
document2.html - HTML parsing completed.

```

Parseable인터페이스는 구문분석(parsing)을 수행하는 기능을 구현할 목적으로 추상메서드 `parse(String fileName)`를 정의했다. 그리고 XMLParser클래스와 HTMLParser클래스는 Parseable인터페이스를 구현하였다.

ParserManager클래스의 `getParser`메서드는 매개변수로 넘겨받는 `type`의 값에 따라 XMLParser인스턴스 또는 HTMLParser인스턴스를 반환한다.



```

Parseable parser = ParserManager.getParser("XML");

public static Parseable getParser(String type) {
    if(type.equals("XML")) {
        return new XMLParser();
    } else {
        Parseable p = new HTMLParser();
        return p;
    }
}

```

www.javachobo.com

`getParser`메서드의 수행결과로 참조변수 `parser`는 XMLParser인스턴스의 주소값을 갖게 된다. 마치 `Parseable parser = new XMLParser();`이 수행된 것과 같다.

```
parser.parse("document.xml");
```

참조변수 `parser`를 통해 `parse()`를 호출하면, `parser`가 참조하고 있는 `XMLParser`인스턴스의 `parse`메서드가 호출된다.

만일 나중에 새로운 종류의 XML구문분석기 `NewXMLParser`클래스가 나와도 `ParserTest`클래스는 변경할 필요없이 `ParserManager`클래스의 `getParser`메서드에서 `return new XMLParser();` 대신 `return new NewXMLParser();`로 변경하기만 하면 된다.

이러한 장점은 특히 분산환경 프로그래밍에서 그 위력을 발휘한다. 사용자 컴퓨터에 설치된 프로그램을 변경하지 않고, 서버측의 변경만으로도 사용자가 새로 개정된 프로그램을 사용하는 것이 가능하다.

## 7.7 인터페이스의 장점

인터페이스를 사용하는 이유와 그 장점을 정리해 보면 다음과 같다.

- 개발시간을 단축시킬 수 있다.
- 표준화가 가능하다.
- 서로 관계없는 클래스들에게 관계를 맺어 줄 수 있다.
- 독립적인 프로그래밍이 가능하다.

### 1. 개발시간을 단축시킬 수 있다.

일단 인터페이스가 작성되면, 이를 사용해서 프로그램을 작성하는 것이 가능하다. 메서드를 호출하는 쪽에서는 메서드의 내용에 관계없이 선언부만 알면 되기 때문이다.

그리고 동시에 다른 한 쪽에서는 인터페이스를 구현하는 클래스를 작성하도록 하여, 인터페이스를 구현하는 클래스가 작성될 때까지 기다리지 않고도 양쪽에서 동시에 개발을 진행할 수 있

다.

## 2. 표준화가 가능하다.

프로젝트에 사용되는 기본 틀을 인터페이스로 작성한 다음 개발자들에게 인터페이스를 구현하여 프로그램을 작성하도록 함으로써 보다 일관되고 정형화된 프로그램의 개발이 가능하다.

## 3. 서로 관계없는 클래스들에게 관계를 맺어 줄 수 있다.

서로 상속관계에 있지도 않고, 같은 조상클래스를 가지고 있지 않은 서로 아무런 관계도 없는 클래스들에게 하나의 인터페이스를 공통적으로 구현하도록 함으로써 관계를 맺어 줄 수 있다.

## 4. 독립적인 프로그래밍이 가능하다.

인터페이스를 이용하면 클래스의 선언과 구현을 분리 시킬 수 있기 때문에 실제구현에 독립적인 프로그램을 작성하는 것이 가능하다. 클래스와 클래스간의 직접적인 관계를 인터페이스를 이용해서 간접적인 관계로 변경하면, 한 클래스의 변경이 관련된 다른 클래스에 영향을 미치지 않는 독립적인 프로그래밍이 가능하다.

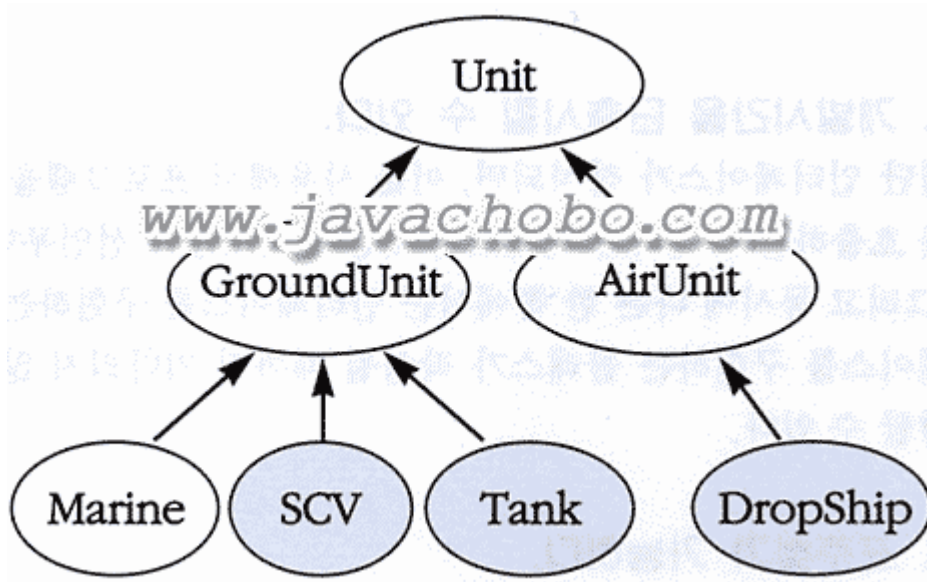
예를 들어 한 데이터베이스 회사가 제공하는 특정 데이터베이스를 사용하는데 필요한 클래스를 사용해서 프로그램을 작성했다면 이 프로그램은 다른 종류의 데이터베이스를 사용하기 위해서는 전체 프로그램 중에서 데이터베이스 관련된 부분은 모두 변경해야 할 것이다.

그러나 데이터베이스 관련 인터페이스를 정의하고 이를 이용해서 프로그램을 작성하면, 데이터베이스의 종류가 변경되더라도 프로그램을 변경하지 않도록 할 수 있다.

단, 데이터베이스 회사에서 제공하는 클래스도 인터페이스를 구현하도록 요구해야 한다. 데이터베이스를 이용한 응용프로그램을 작성하는 쪽에서는 인터페이스를 이용해서 프로그램을 작성하고, 데이터베이스 회사에서는 인터페이스를 구현한 클래스를 작성해서 제공해야 한다.

실제로 자바에는 다수의 데이터베이스 관련 인터페이스를 제공하고 있으며, 프로그래머는 이 인터페이스를 이용해서 프로그래밍을 하면 특정 데이터베이스에 종속되지 않는 프로그램을 작성할 수 있다.

게임 스타크래프트에 나오는 유닛을 클래스로 표현하고 이들간의 관계를 상속계층도로 표현해 보았다.



게임에 나오는 모든 유닛들의 최고 조상은 Unit클래스이고 유닛의 종류는 지상유닛 (GoundUnit)과 공중유닛(AirUnit)으로 나뉘어진다.

그리고 지상유닛에는 Marine, SCV, Tank가 있고, 공중유닛으로는 DropShip이 있다. SCV에게 Tank와 DropShip과 같은 기계화 유닛을 수리할 수 있는 기능을 제공하기 위해 repair메서드를 정의한다면 다음과 같을 것이다.

```

void repair(Tank t) {
    // Tank를 수리한다.
}

void repair(DropShip d) {
    // DropShip을 수리한다.
}
  
```

이런 식으로 수리가 가능한 유닛의 개수 만큼 다른 버전의 오버로딩된 메서드를 정의해야할 것이다.

이것을 피하기 위해 매개변수의 타입을 이 들의 공통 조상으로 하면 좋겠지만 DropShip은 공통조상이 다르기 때문에 공통조상의 타입으로 메서드를 정의한다고 해도 최소한 2개의 메서드가 필요할 것이다.

```

void repair(GroundUnit gu) {
    // 매개변수로 넘겨진 지상유닛(GroundUnit)을 수리한다.
}

void repair(AirUnit au) {
    // 매개변수로 넘겨진 공중유닛(AirUnit)을 수리한다.
}

```

그리고 GroundUnit의 자손 중에는 Marine과 같이 기계화 유닛이 아닌 클래스도 포함될 수 있기 때문에 repair메서드의 매개변수 타입으로 GroundUnit은 부적합하다. 현재의 상속관계에서는 이들의 공통점은 없다. 이 때 인터페이스를 이용하면 기존의 상속체계를 유지하면서 이들 기계화 유닛에 공통점을 부여할 수 있다. 다음과 같이 Repairable이라는 인터페이스를 정의하고 수리가 가능한 기계화 유닛에게 이 인터페이스를 구현하도록 하면 된다.

```

interface Repairable {}

class SCV extends GroundUnit implements Repairable {
    //...
}

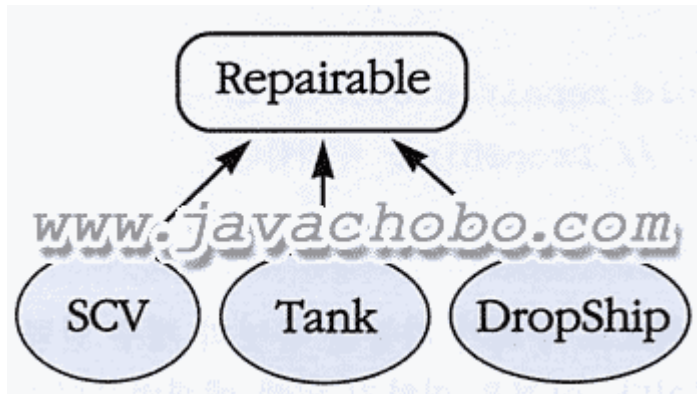
class Tank extends GroundUnit implements Repairable {
    //...
}

class DropShip extends AirUnit implements Repairable {
    //...
}

```

이제 이 세 개의 클래스에는 같은 인터페이스를 구현했다는 공통점이 생겼다. 인터페이스 Repairable에 정의된 것은 아무것도 없고, 단지 인스턴스의 타입체크에 사용된다.

Repairable인터페이스를 중심으로 상속계층도를 그려보면 다음과 같다.



그리고, repair메서드의 매개변수의 타입을 Repairable로 선언하면, 이 메서드의 매개변수로 Repairable인터페이스를 구현한 클래스의 인스턴스만 받아들여질 것이다.

```
void repair(Repairable r) {  
    // 매개변수로 넘겨받은 유닛을 수리한다.  
}
```

앞으로 새로운 클래스가 추가될 때, SCV의 repair메서드에 의해서 수리가 가능하도록 하려면 Repairable인터페이스를 구현하도록 하면 될 것이다.

#### [예제7-25] RepairableTest.java

```
class RepairableTest  
{  
    public static void main(String[] args)  
    {  
        Tank t = new Tank();  
        DropShip d = new DropShip();  
  
        Marine m = new Marine();  
        SCV s = new SCV();
```

```

        s.repair(t);    // SCV가 Tank를 수리하도록 한다.
        s.repair(d);
//        s.repair(m);    에러발생 : repair(Repairable) in SCV cannot be applied to
//        (Marine)
    }
}

interface Repairable {}

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

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

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

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

```

```

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

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

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

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

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

    void repair(Repairable r) {
        if (r instanceof Unit) {

```



```

        Unit u = (Unit)r;
        while(u.hitPoint!=u.MAX_HP) {
            /* Unit의 HP를 증가시킨다. */
            u.hitPoint++;
        }
        System.out.println( u.toString() + "의 수리가 끝났습니다.");
    }
}
//...
}

```

#### [실행결과]

Tank의 수리가 끝났습니다.

DropShip의 수리가 끝났습니다.

repair메서드의 매개변수 r은 Repairable타입이기 때문에 인터페이스 Repairable에 정의된 멤버만 사용할 수 있다.

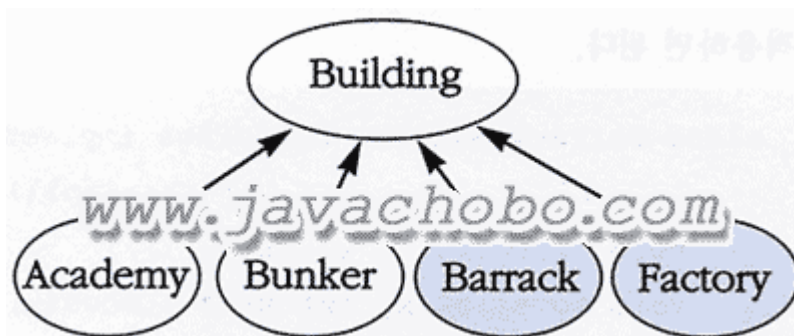
그러나 Repairable에는 정의된 멤버가 없으므로 이 타입의 참조변수로는 할 수 있는 일은 아무 것도 없다.

그래서 instanceof연산자로 타입을 체크한 뒤 캐스팅하여 Unit클래스에 정의된 hitPoint와 MAX\_HP를 사용할 수 있도록 하였다.

그 다음엔 유닛의 현재체력(hitPoint)이 유닛이 가질 수 있는 최고체력(MAX\_HP)이 될 때까지 체력을 증가시키는 작업을 수행한다.

Marine은 Repairable인터페이스를 구현하지 않았으므로 SCV클래스의 repair메서드의 매개변수로 Marine을 사용하면 컴파일 시에 에러가 발생한다.

이와 유사한 예를 한가지 더 들어보자. 게임 스타크래프트에 나오는 건물들을 클래스로 표현하고 이들의 관계를 상속계층도로 표현하였다.



건물을 표현하는 클래스 Academy, Bunker, Barrack, Factory가 있고 이들의 조상인 Building클래스가 있다고 하자. 이 때 Barrack클래스와 Factory클래스에 다음과 같은 내용의, 건물을 이동시킬수 있는, 새로운 메서드를 추가하고자 한다면 어떻게 해야할까?

```
void liftOff() { /* 내용생략 */}
void move(int x, int y) { /* 내용생략 */}
void stop() { /* 내용생략 */}
void land() { /* 내용생략 */}
```

Barrack 클래스와 Factory클래스 모두 위의 코드를 적어주면 되긴 하지만, 코드가 중복된다는 단점이 있다. 그렇다고 해서 조상클래스인 Building클래스에 코드를 추가해주면, Building클래스의 다른 자손인 Academy와 Bunker클래스도 추가된 코드를 상속받으므로 안된다. 이런 경우에도 인터페이스를 이용해서 해결할 수가 있다. 우선 새로 추가하고자하는 메서드를 정의하는 인터페이스를 정의하고 이를 구현하는 클래스를 작성한다.

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

class LiftableImpl implements Liftable {
    void liftOff() { /* 내용 생략 */}
    void move(int x, int y) { /* 내용 생략 */}
    void stop() { /* 내용 생략 */}
    void land() { /* 내용 생략 */}
```

```
}
```

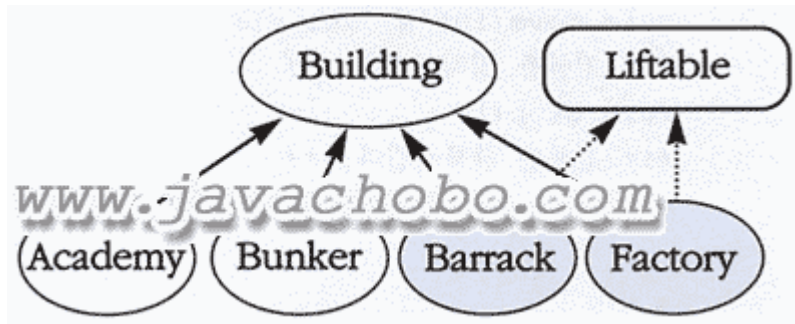
마지막으로 새로 작성된 인터페이스와 이를 구현한 클래스를 Barrack과 Factory클래스에 적용하면 된다.

```
class Barrack extends Buildings implements Lifiable {  
    LifiableImpl l = new LifiableImpl();  
    void liftOff() { l.liftOff();}  
    void move(int x, int y) { l.move(x, y);}  
    void stop() { l.stop();}  
    void land() { l.land();}  
    void trainMarine() { /* 내용 생략 */}  
    void trainMedic() { /* 내용 생략 */}  
    // ...  
}
```

```
class Factory extends Buildings implements Lifiable {  
    LifiableImpl l = new LifiableImpl();  
    void liftOff() { l.liftOff();}  
    void move(int x, int y) { l.move(x, y);}  
    void stop() { l.stop();}  
    void land() { l.land();}  
    void trainMarine() { /* 내용 생략 */}  
    void trainMedic() { /* 내용 생략 */}  
    // ...  
}
```

Barrack클래스가 Lifiable인터페이스를 구현하도록 하고, 인터페이스를 구현한 LifiableImpl 클래스를 Barrack클래스에 포함시켜서 내부적으로 호출해서 사용하도록 한다.

이렇게 함으로써 같은 내용의 코드를 Barrack클래스와 Factory클래스에서 각각 작성하지 않고 LifiableImpl클래스 한 곳에서 관리할 수 있다. 그리고 작성된 Lifiable인터페이스와 이를 구현한 LifiableImpl클래스는 후에 다시 재사용될 수 있을 것이다.



## 7.8 인터페이스의 이해

지금까지 인터페이스의 특징과 구현하는 방법, 장점 등 인터페이스에 대한 일반적인 사항들에 대해서 모두 살펴보았다. 하지만 '인터페이스란 도대체 무엇인가?'라는 의문은 여전히 남아있을 것이다. 이번 절에서는 인터페이스의 규칙이나 활용이 아닌, 본질적인 측면에 대해 살펴보자.

먼저 인터페이스를 이해하기 위해서는 다음의 두가지 사항을 반드시 염두에 두고 있어야 한다.

- 클래스를 사용하는 쪽(User)과 클래스를 제공하는 쪽(Provider)이 있다.
- 메서드를 사용(호출)하는 쪽(User)에서는 사용하려는 메서드(Provider)의 선언부만 알면 된다.(내용은 몰라도 된다.)

다음과 같이 클래스 A와 클래스 B가 있다고 하자.

```

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

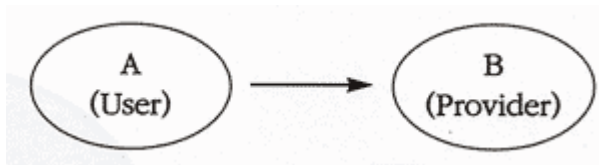
```

    }

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

```

클래스 A(User)는 클래스 B(Provider)의 인스턴스를 생성하고 메서드를 호출한다. 이것을 간단히 'A-B'라고 표현하자. 이 두 클래스는 서로 직접적인 관계에 있다.



이 경우 클래스 A를 작성하기 위해서는 클래스 B가 이미 작성되어 있어야 한다. 그리고 클래스 B의 methodB()의 선언부가 변경되면, 이를 사용하는 클래스 A도 변경되어야 한다.

이와 같이 직접적인 관계의 두 클래스는 한 쪽(Provider)가 변경되면 다른 한 쪽(User)도 변경되어야 한다는 단점이 있다.

그 러나 클래스 A가 클래스 B를 직접 호출하지 않고 인터페이스를 매개체로 해서 클래스 A가 인터페이스를 통해서 클래스 B의 메서드에 접근하도록 하면, 클래스 B에 변경사항이 생기거나 클래스 B와 같은 기능의 다른 클래스로 대체 되어도 클래스 A는 전혀 영향을 받지 않도록 하는 것이 가능하다.

두 클래스간의 관계를 간접적으로 변경하기 위해서는 먼저 인터페이스를 이용해서 클래스 B(Provider)의 선언과 구현을 분리해야한다.

먼저 다음과 같이 클래스 B에 정의된 메서드를 추상메서드로 정의하는 인터페이스 I를 정의한다.

```

interface I {

```

```
public abstract void methodB();
}
```

그 다음에는 클래스 B가 인터페이스 I를 구현하도록 한다.

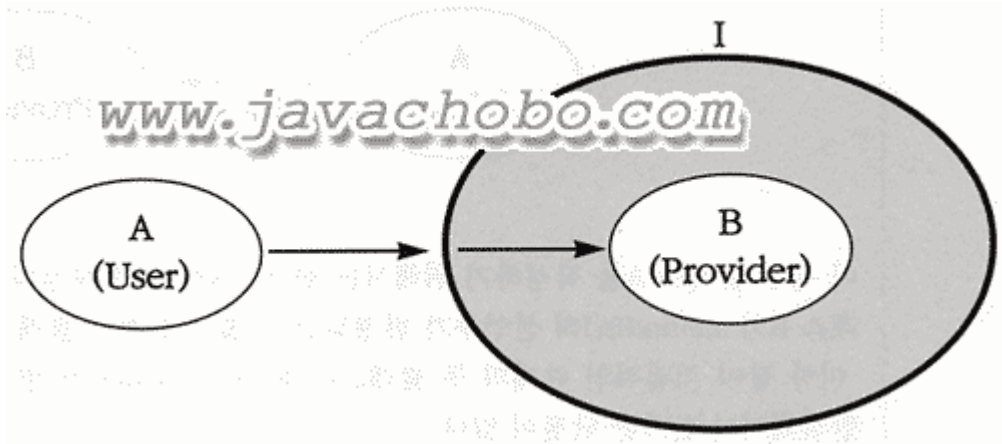
```
class B implements I {
    public void methodB() {
        System.out.println("methodB in B class");
    }
}
```

이제 클래스 A는 클래스 B 대신 인터페이스 I를 사용해서 작성할 수 있다.

```
class A {
    public void methodA(I i) {
        i.methodB();
    }
}
```

[참고]methodA가 호출될 때 인터페이스 I를 구현한 클래스의 인스턴스(클래스 B의 인스턴스)를 제공받아야 한다.

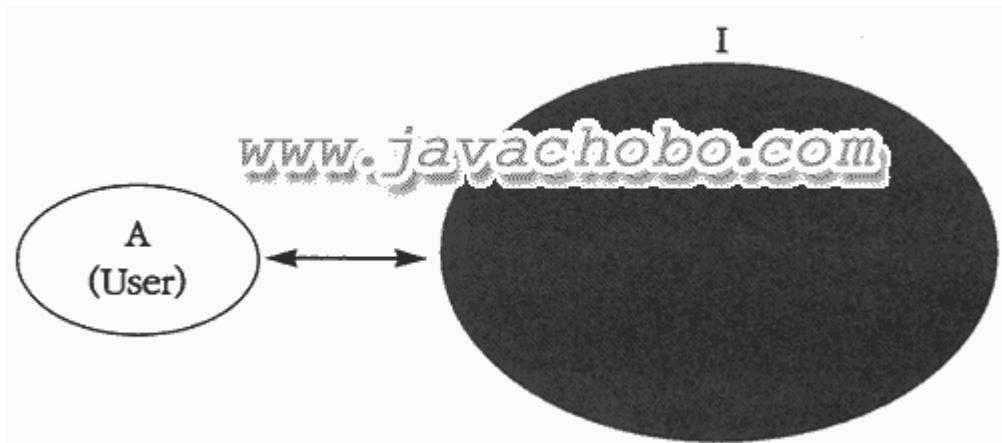
클래스 A를 작성하는데 있어서 클래스 B가 사용되지 않았다는 점에 주목하자. 이제 클래스 A와 클래스 B는 'A-B'의 직접적인 관계에서 'A-I-B'의 간접적인 관계로 바뀐 것이다.



결국 클래스 A는 여전히 클래스 B의 메서드를 호출하지만, 클래스 A는 인터페이스 I하고만 직접적인 관계에 있기 때문에 클래스 B의 변경에 영향을 받지 않도록 하는 것이 가능하다.

클래스 A는 인터페이스를 통해 실제로 사용하는 클래스의 이름을 몰라도 되고 심지어는 실제로 구현된 클래스가 존재하지 않아도 문제되지 않는다.

클래스 A는 직접적인 관계에 있는 인터페이스 I의 영향만을 받는다.



인터페이스 I는 실제구현 내용(클래스 B)을 감싸고 있는 껍데기이며, 클래스 A는 껍데기 안에 어떤 알맹이(클래스)가 들어 있는지 몰라도 된다.

[예제 7-26] InterfaceTest.java

```

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

```

```

interface I {
    public abstract void play();
}

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

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

class InterfaceTest {
    public static void main(String[] args) {
        A a = new A();
        a.autoPlay(new B());
        a.autoPlay(new C());
    }
}

```

#### [실행결과]

```

play in B class
play in C class

```

[참고]클래스 A를 작성하는데 클래스 B가 관련되지 않았다는 사실에 주목한다.

클래스 A가 인터페이스 I를 사용해서 작성되긴 하였지만, 이처럼 매개변수를 통해서 인터페이스 I를 구현한 클래스의 인스턴스를 동적으로 제공받아야 한다.

클래스 Thread의 생성자인 Thread(Runnable target)와 AWT컴포넌트의 addActionListener(ActionListener l)가 이런 방식으로 되어 있다.

[참고]Runnable과 ActionListener는 인터페이스이다.



이처럼 매개변수를 통해 동적으로 제공받을 수 도 있지만 다음과 같이 다른 제 3의 클래스를 통해서 제공받을 수도 있다. JDBC의 DriverManager클래스가 이런 방식으로 되어 있다.

[예제7-27] InterfaceTest2.java

```
class InterfaceTest2 {
    public static void main(String[] args) {
        A a = new A();
        a.methodA();
    }
}

class A {
    void methodA() {
        I i = InstanceManager.getInstance();
        i.methodB();
    }
}

interface I {
    public abstract void methodB();
}

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

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

[실행결과]

methodB in B class

# [Java의 정석]제8장 예외처리

자바의정석

2012/12/22 17:12

<http://blog.naver.com/gphic/50157740254>

## 8. 예외처리(Exception Handling)

### 1.1 프로그램 오류

프로그램이 실행 중 어떤 원인에 의해서 오작동을 하거나 비정상적으로 종료되는 경우가 있다. 이러한 결과를 초래하는 원인을 프로그램 에러 또는 오류라고 한다.

이를 발생시점에 따라 '컴파일 에러(compile-time error)'와 '런타임 에러(runtime error)'로 나눌 수 있는데, 글자 그대로 '컴파일 에러'는 컴파일 할 때 발생하는 에러이고 프로그램의 실행도중에 발생하는 에러를 '런타임 에러'라고 한다.

컴파일 할 때(compile-time)는 컴파일러가 소스코드(\*.java)에 대해 오타나 잘못된 구문, 자료형 체크 등의 기본적인 검사를 수행하여 오류가 있는지를 알려 준다. 컴파일러가 알려 준 에러들을 모두 수정해서 컴파일을 성공적으로 마치고 나면, 클래스 파일(\*.class)이 생성되고, 생성된 클래스 파일을 실행할 수 있게 되는 것이다.

하지만, 컴파일을 에러없이 성공적으로 마쳤다고 해서 프로그램이 실행 시에도 에러가 발생하지 않는 것은 아니다.

컴파일러가 소스코드의 기본적인 사항은 컴파일시에 모두 걸러 줄 수는 있지만, 실행도중에 발생할 수 있는 잠재적인 오류에 대해서까지 검사할 수 없기 때문에 컴파일은 잘되었어도 실행 중에 에러에 의해서 잘못된 결과를 얻거나 프로그램이 비정상적으로 종료될 수 있다.

여러분들은 이미 실행도중에 발생하는 런타임 에러를 여러 번 경험했을 것이다. 예를 들면 프로그램이 실행 중 동작을 멈춘 상태로 오랜 시간 지속되거나, 갑자기 프로그램이 실행을 멈추고 종료되는 경우 등이 이에 해당한다.

런타임 에러를 방지하기 위해서는 프로그램의 실행도중 발생할 수 있는 모든 경우의 수를 고려하여 이에 대한 대비를 하는 것이 필요하다.

자바에서는 실행 시(runtime) 발생할 수 있는 프로그램 오류를 '에러(Error)'와 '예외(Exception)', 두 가지로 구분하였다.

에러는 메모리 부족(OutOfMemoryError)이나 스택오버플로우(StackOverflowError)와 같이 일단 발생하면 복구할 수 없는 심각한 오류이고, 예외는 발생하더라도 수습될 수 있는 비교적 덜 심각한 것이다.

에러가 발생하면, 프로그램의 비정상적인 종료를 막을 길이 없지만, 예외는 발생하더라도 프로그래머가 이에 대한 적절한 코드를 미리 작성해 놓음으로써 프로그램의 비정상적인 종료를 막을 수 있다.

에러(Error) - 프로그램 코드에 의해서 수습될 수 없는 심각한 오류

예외(Exception) - 프로그램 코드에 의해서 수습될 수 있는 다소 미약한 오류

## 1.2 예외처리(Exception Handling)의 정의와 목적.

프로그램의 실행도중에 발생하는 에러는 어쩔 수 없지만, 예외는 프로그래머가 이에 대한 처리를 미리 해주어야 한다.

예외처리(Exception Handling)란, 프로그램 실행 시 발생할 수 있는 예기치 못한 예외의 발생에 대비한 코드를 작성하는 것이며, 예외처리의 목적은 예외의 발생으로 인한 실행 중인 프로그램의 갑작스런 비정상 종료를 막고, 정상적인 실행상태를 유지할 수 있도록 하는 것이다.

예외처리(Exception Handling)의

정의 - 프로그램 실행 시 발생할 수 있는 예외의 발생에 대비한 코드를 작성하는 것

목적 - 프로그램의 비정상 종료를 막고, 정상적인 실행상태를 유지하는 것

[참고]에러와 예외는 모두 실행 시(runtime) 발생하는 오류이다.

## 1.3 예외처리구문 - try-catch

예외를 처리하기 위해서는 try-catch문을 사용하며, 그 구조는 다음과 같다.

```
try {
    // 예외가 발생할 가능성이 있는 문장들을 넣는다.
} catch (Exception1 e1) {
    // Exception1이 발생했을 경우, Exception1을 처리하기 위한 문장을 적는다.
} catch (Exception2 e2) {
    // Exception2가 발생했을 경우, Exception2를 처리하기 위한 문장을 적는다.
...
} catch (ExceptionN eN) {
    // ExceptionN이 발생했을 경우, ExceptionN을 처리하기 위한 문장을 적는다.
}
```

하나의 try블럭 다음에는 여러 종류의 예외를 처리할 수 있도록 하나 이상의 catch블럭이 올 수 있으며, 이 중 발생한 예외의 종류와 일치하는 단 한 개의 catch블럭 만이 수행된다.

발생한 예외의 종류와 일치하는 catch블럭이 없으면, 예외는 처리되지 않는다.

[참고]try블럭이나 catch블럭 내에 포함된 문장이 하나라고 해서 if문에서처럼 중괄호{}를 생략할 수는 없다.

#### [예제8-1] ExceptionEx1.java

```
class ExceptionEx1 {
    public static void main(String[] args)
    {
        try {
            try { } catch (Exception e) {}
        } catch (Exception e) {
            try { } catch (Exception e) {} // 컴파일 에러 발생 !!!
        }

        try {
```

```

        } catch (Exception e)    {

        }

    }    // main메서드의 끝
}

```

위의 예제는 아무 일도 하지 않는다. 단순히 try-catch문의 사용 예를 보여 주기 위해서 작성한 코드이다. 이처럼, 하나의 메서드 내에 여러 개의 try-catch문이 사용될 수 있으며, try블럭 또는 catch블럭에 또 다른 try-catch문이 포함될 수 있다.

catch블럭의 괄호 내에 선언된 변수는 catch블럭 내에서만 유효하기 때문에, 위의 모든 catch블럭에 참조변수 'e' 하나만을 사용해도 된다.

하지만, catch블럭 내에 또 하나의 try-catch문이 포함된 경우, 같은 이름의 참조변수를 사용해서는 안 된다. 각 catch블럭에 선언된 두 참조변수의 영역이 서로 겹치기 때문에 다른 이름을 사용해서 구별해야하기 때문이다.

따라서 위의 예제에서 catch블럭 내의 try-catch문에 선언되어 있는 참조변수이름을 'e'가 아닌 다른 것으로 바꿔야 한다.

#### [예제8-2] ExceptionEx2.java

```

class ExceptionEx2 {
    public static void main(String args[]) {
        int number = 100;
        int result = 0;

        for(int i=0; i < 10; i++) {
            result = number / (int)(Math.random() * 10);
            System.out.println(result);
        }
    }
}

```

#### [실행결과]

```
20
100
java.lang.ArithmeticException: / by zero
    at ExceptionEx2.main(ExceptionEx2.java:7)
```

위의 예제는 변수 number에 저장되어 있는 값 100을 0~9사이의 임의의 정수로 나눈 결과를 출력하는 일을 10번 반복한다.

random메서드를 사용했기 때문에 매번 실행할 때마다 결과가 다르겠지만, 대부분의 경우 10번이 출력되기 이전에 예외가 발생하여 프로그램이 비정상적으로 종료될 것이다.

결과에 나타난 메시지를 보면 Exception의 발생원인과 위치를 알 수 있으며, 이 예제의 결과에 나타난 메시지를 분석해 보면, 0으로 나누려 했기 때문에 ArithmeticException이 발생했고, 발생위치는 ExceptionEx2클래스의 main메서드(ExceptionEx2.java의 7번째 라인)라는 것을 알 수 있다.

[참고]자 바에서는 정수값을 0으로 나누면, ArithmeticException(산술연산에 오류가 있을 때 발생하는 예외)이 발생하도록 되어 있다. 하지만, 실수값을 0으로 나눌 때는 ArithmeticException이 발생하지 않는다. 마찬가지로 정수값을 0.0(실수값)으로 나누는 것 역시 ArithmeticException이 발생하지 않는다.

이제 어디서 왜 Exception이 발생하는지 알았으니, 예외처리구문을 추가 해서 실행도중 예외가 발생하더라도 프로그램이 실행 도중에 비정상적으로 종료되지 않도록 수정해 보자.

#### [예제8-3] ExceptionEx3.java

```
class ExceptionEx3 {
    public static void main(String args[]) {
        int number = 100;
        int result = 0;

        for(int i=0; i < 10; i++) {
            try {
                result = number / (int)(Math.random() * 10);
                System.out.println(result);
            } catch (ArithmeticException e) {
                // ArithmeticException이 발생하면 수행된다.
            }
        }
    }
}
```

```

        System.out.println("0");
    } // try-catch의 마지막
} // for의 마지막
}
}

```

#### [실행결과]

```

16
20
11
0
25
100
25
33
14
12

```

[참고] 실행할 때마다 결과가 달라지므로, 위의 결과는 여러분들이 실행한 결과와 다를 수 있다.

위의 예제는 ExceptionEx2.java에 단순히 try-catch문을 추가한 것이다.

ArithmeticException이 발생했을 경우에는 0을 화면에 출력하도록 했다. 위의 결과에서 보면, 4번째에 0이 출력되었는데 그 이유는 for문의 4번째 반복에서 ArithmeticException이 발생했기 때문이다. 그래서, ArithmeticException에 해당하는 catch블럭을 찾아서 그 catch블럭 내의 문장들을 실행한 다음 try-catch문을 벗어나 for문의 다음 반복을 계속 수행하여 작업을 모두 마치고 정상적으로 종료되었다.

만일 예외처리를 하지 않았다면, 세 번째 줄까지만 출력되고 예외가 발생해서 프로그램이 비정상적으로 종료되었을 것이다.

#### 1.4 try-catch문에서의 흐름

try-catch문에서, 예외가 발생한 경우와 발생하지 않았을 때의 흐름(문장의 실행순서)가 달라지는데, 아래에는 이 두 가지 경우에 따른 문장 실행순서를 정리하였다.



1. try블럭 내에서 예외가 발생한 경우.

곧바로 try블럭을 벗어나 제일 첫 번째 위치한 catch블럭부터 차례대로 내려오면서, 발생한 예외와 일치하는 catch블럭이 있는지 확인한다.

일치하는 catch블럭을 찾게 되면, 그 catch블럭 내의 문장들을 수행하고 전체 try-catch문을 빠져나가서 그 다음 문장을 계속해서 수행한다.

2. try블럭 내에서 예외가 발생하지 않은 경우.

try블럭 내의 마지막 문장까지 다 수행할 때까지도 예외가 발생하지 않으면, 어떠한 catch블럭도 거치지 않고 전체 try-catch문을 빠져나가서 그 다음 문장을 계속해서 수행한다.

#### [예제8-4] ExceptionEx4.java

```
class ExceptionEx4 {  
    public static void main(String args[]) {  
        System.out.println(1);  
        System.out.println(2);  
        try {  
            System.out.println(3);  
            System.out.println(4);  
        } catch (Exception e) {  
            System.out.println(5);  
        } // try-catch의 마지막  
        System.out.println(6);  
    } // main메서드의 마지막  
}
```

#### [실행결과]

```
1  
2  
3  
4  
6
```

위의 예제에서는 예외가 발생하지 않았으므로 catch블럭의 문장이 실행되지 않았다. 아래의

예제는 위의 예제를 변경해서, try블럭에서 예외가 발생하도록 하였다.

#### [예제8-5] ExceptionEx5.java

```
class ExceptionEx5 {
    public static void main(String args[]) {
        System.out.println(1);
        System.out.println(2);
        try {
            System.out.println(3);
            System.out.println(0/0);    // ArithmeticException을 발생시킨다.
            System.out.println(4);    // 실행되지 않는다.
        } catch (ArithmeticException ae) {
            System.out.println(5);
        }    // try-catch의 마지막
        System.out.println(6);
    }    // main메서드의 마지막
}
```

#### [실행결과]

```
1
2
3
5
6
```

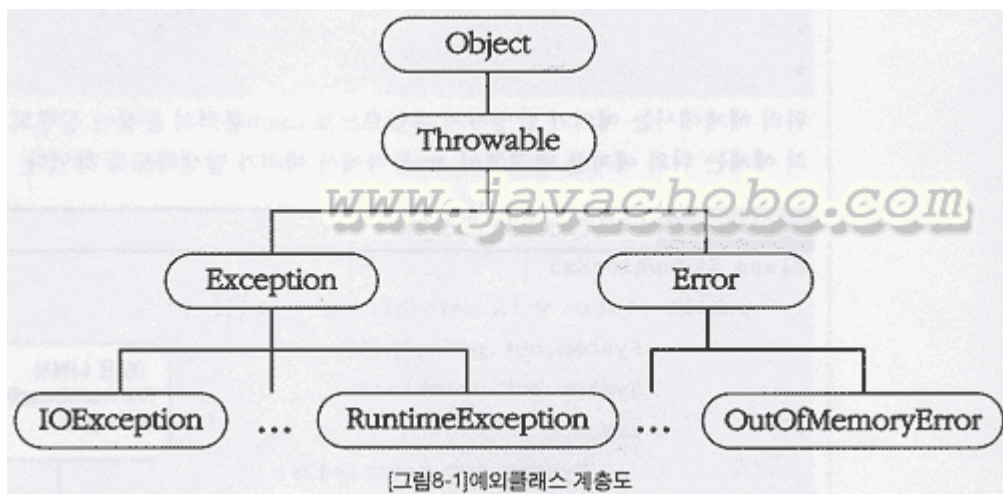
위 의 예제의 결과를 보면, 1, 2, 3을 출력한 다음 try블럭에서 예외가 발생했기 때문에 try블럭을 바로 벗어나서 System.out.println(4)은 실행되지 않는다. 그리고는 발생한 예외에 해당하는 catch블럭으로 이동하여 문장들을 수행한다. 그리고는 전체 try-catch문을 벗어나서 그 다음 문장을 실행하여 6을 화면에 출력한다.

try블럭에서 예외가 발생하면, Exception이 발생한 이후에 있는 try블럭의 문장들은 수행되지 않으므로, try블럭에 포함시킬 범위를 잘 선택해야한다.

## 1.5 예외 클래스의 계층구조

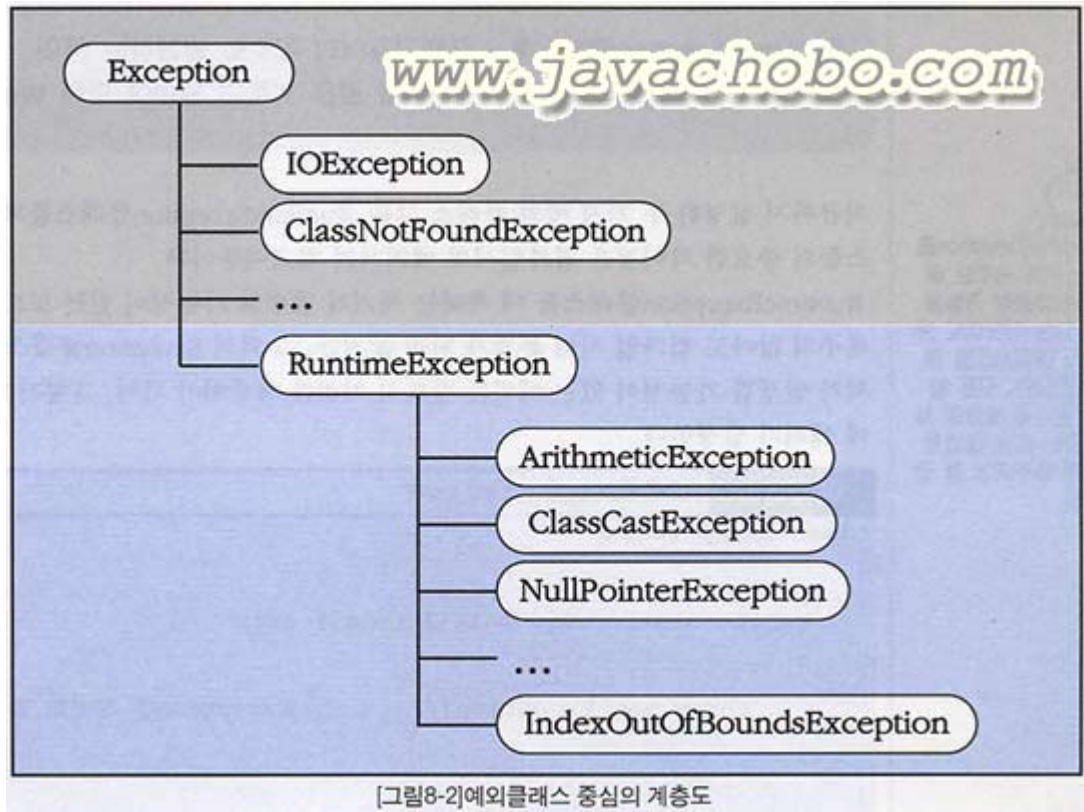
자 바에서는 실행 시 발생할 수 있는 오류(Exception과 Error)를 클래스로 정의하였다. 이미 배웠던 것과 같이 모든 클래스의 조상은 Object클래스이므로 Exception과 Error클래스 역시 Object클래스의 자손들이다.

모든 예외의 최고 조상은 Exception클래스이며, 상속계층도를 Exception클래스부터 도식화 하면 다음과 같다.



[참고]위의 그림은 전체의 Exception클래스들 중에서 몇 개의 주요 클래스들만을 나열한 것이다.

예외 클래스들은 다음과 같이 두 개의 그룹으로 나뉘질 수 있다.



- RuntimeException클래스와 그 자손클래스들(위 그림에서의 아래 부분), 그리고
- 그 외의 Exception클래스와 그 자손클래스들(위 부분)이다.

앞으로 RuntimeException클래스와 그 자손 클래스들을 'RuntimeException클래스들'이라 하고, RuntimeException클래스들을 제외한 나머지 클래스들을 '그 외의 Exception클래스들'이라 하겠다.

RuntimeException클래스들은 주로 프로그래머의 실수에 의해서 발생할 수 있는 예외들로서 자바의 프로그래밍 요소들과 관계가 깊다. 예를 들면, 배열의 범위를 벗어난다던가 (IndexOutOfBoundsException), 값이 null인 참조변수의 멤버를 호출하려 했다던가 (NullPointerException), 클래스간의 형변환을 잘못했다던가(ClassCastException), 전의 예제에서 본 것처럼 정수를 0으로 나누려 했다던가(Arithmetic-Exception)하는 경우에 발생하는 예외들이다.

위의 예제에서는 RuntimeException클래스들 중의 하나인 ArithmeticException을 try-catch문으로 처리하였지만, 사실 try-catch문을 사용하기보다는 0으로 나누지 않도록 프로그램을 변경하는 것이 바른 처리방법이다.

이처럼 RuntimeException예외들이 발생할 가능성이 있는 코드들은 try-catch문을 사용하기 보다는 프로그래머들이 보다 주의 깊게 작성하여 예외가 발생하지 않도록 해야 할 것이다.

그 외의 Exception클래스들은 주로 외부의 영향으로 발생할 수 있는 것들로서, 프로그램의 사용자들의 동작에 의해서 발생하는 경우가 많다. 예를 들면, 존재하지 않는 파일의 이름을 입력했다던가(FileNotFoundException), 실수로 클래스의 이름을 잘못 적었다던가(ClassNotFoundException), 입력한 데이터의 형식이 잘못되었다던가(DataFormatException) 하는 경우에 발생하는 예외들이다. 이런 종류의 예외들은 반드시 처리를 해주어야 한다.

RuntimeException클래스들 - 프로그래머의 실수로 발생하는 예외  
그 외의 클래스들 - 사용자의 실수와 같은 외적인 요인에 의해 발생하는 예외

지금까지 설명한 두 가지 그룹, RuntimeException클래스들과 그 외의 Exception클래스들의 중요한 차이점은 컴파일시의 예외처리 체크여부이다.

RuntimeException 클래스들 그룹에 속하는 예외가 발생할 가능성이 있는 코드에는 예외처리를 해주지 않아도 컴파일 시에 문제가 되지 않지만, 그 외의 Exception클래스들 그룹에 속하는 예외가 발생할 가능성이 있는 예외는 반드시 처리를 해주어야 하며, 그렇지 않으면 컴파일시에 에러가 발생한다.

[참고]RuntimeException클래스들에 속하는 예외가 발생할 가능성이 있는 코드에도 예외처리를 해야 한다면, 모든 참조 변수와 배열이 사용되는 곳에 예외처리를 해주어야 할 것이다.

#### [예제8-6] ExceptionEx6.java

```
class ExceptionEx6
{
    public static void main(String[] args)
    {
        throw new Exception();        // Exception을 강제로 발생시킨다.
    }
}
```

위의 예제를 작성한 후에 컴파일 하면, 아래와 같은 에러가 발생하며 컴파일이 완료되지 않을 것이다.

```
ExceptionEx6.java:5: unreported exception java.lang.Exception; must be caught
or declared to be thrown
    throw new Exception();
    ^
1 error
```

예외처리가 되어야 할 부분에 예외처리가 되어 있지 않다는 에러이다. 위의 결과에서 알 수 있는 것처럼, 위에서 분류한 '그 외의 Exception클래스들'이 발생할 가능성이 있는 문장들에 대해 예외처리를 해주지 않으면 컴파일 조차 되지 않는다.

따라서 위의 예제를 아래와 같이 try-catch문으로 처리해주어야 컴파일이 성공적으로 이루어질 것이다.

#### [예제8-7] ExceptionEx7.java

```
class ExceptionEx7 {
    public static void main(String[] args)
    {
        try    {
            throw new Exception();
        } catch (Exception e)    {
            System.out.println("Exception이 발생했습니다.");
        }
    }    // main메서드의 끝
}
```

#### [실행결과]

Exception이 발생했습니다.

#### [예제8-8] ExceptionEx8.java

```
class ExceptionEx8
{
    public static void main(String[] args)
    {
        throw new RuntimeException();    // RuntimeException을 강제로 발생시킨
다.
    }
}
```

위의 예제를 컴파일 하면, 예외처리를 하지 않았음에도 불구하고 이전의 예제와는 달리 성공적으로 컴파일될 것이다. 그러나 실행하면 RuntimeException이 발생하여 비정상적으로 종료될 것이다.

이 예제가 명백히 RuntimeException을 발생시키는 코드를 가지고 있고, 이에 대한 예외처리를 하지 않았음에도 불구하고 성공적으로 컴파일 되었다.

이와 같이 RuntimeException클래스들은 예외처리를 해주지 않아도 컴파일러가 문제삼지 않는 것을 알아야 한다.

#### 1.6 예외 발생시키기

키워드 throw를 사용해서 프로그래머가 고의로 예외를 발생시킬 수 있으며, 방법은 아래의 순서를 따르면 된다.

1. 먼저, 키워드 new를 이용해서 발생시키려는 예외 클래스의 객체를 만든 다음  
Exception e = new Exception("고의로 발생시켰음");
2. 키워드 throw를 이용해서 예외를 발생시킨다.  
throw e;

#### [예제8-9] ExceptionEx9.java

```
class ExceptionEx9 {  
    public static void main(String args[]) {  
        try {  
            Exception e = new Exception("고의로 발생시켰음.");  
            throw e;                // 예외를 발생시킴  
// throw new Exception("고의로 발생시켰음."); 위의 두 줄을 한 줄로 줄여 쓸 수 있다.  
        } catch (Exception e) {  
            System.out.println("에러 메시지 : " + e.getMessage());  
            e.printStackTrace();  
        }  
        System.out.println("프로그램이 정상 종료되었음.");  
    }  
}
```

#### [실행결과]

```
에러 메시지 : 고의로 발생시켰음.  
java.lang.Exception: 고의로 발생시켰음.  
    at ExceptionEx9.main(ExceptionEx9.java:4)  
프로그램이 정상 종료되었음.
```

Exception인스턴스를 생성할 때, 생성자에 String을 넣어 주면, 이 String이 Exception인스턴스에 메시지로 저장된다. 이 저장된 메시지는 getMessage()를 이용해서 얻을 수 있다.

### 1.7. 예외의 발생과 catch블럭

catch블럭은 괄호()와 블럭{} 두 부분으로 나뉘져 있는데, 괄호()내에는 처리하고자 하는 예외와 같은 타입의 참조변수 하나를 선언해야한다.

예외가 발생하면, 발생한 예외에 해당하는 클래스의 인스턴스가 만들어 진다. 예제8-5에서는 ArithmeticException이 발생했으므로, ArithmeticException클래스의 인스턴스가 생성되었다.



예외가 발생한 문장이 try-catch문의 try블럭에 포함되어 있다면, 이 예외를 처리할 수 있는 catch블럭이 있는지를 찾게 된다.

첫 번째 catch블럭부터 차례로 내려가면서, catch블럭의 괄호()내에 선언된 참조변수의 종류와 생성된 예외클래스의 인스턴스에 instanceof연산자를 이용해서 검사하게 되는데, 검사결과가 true인 catch블럭을 만날 때까지 검사는 계속된다.

검사결과가 true인 catch블럭을 찾게 되면, 블럭에 있는 문장들을 모두 수행한 후에 try-catch문을 빠져나가고 예외는 처리되지만, 검사결과가 true인 catch블럭이 하나도 없으면 예외는 처리되지 않는다.

모든 예외 클래스들은 Exception클래스의 자손이므로, catch블럭의 괄호()에 Exception클래스 타입의 참조변수를 선언해 놓으면 어떤 종류의 예외가 발생하더라도 이 catch블럭에 의해서 처리된다.

#### [예제8-10] ExceptionEx10.java

```
class ExceptionEx10 {
    public static void main(String args[]) {
        System.out.println(1);
        System.out.println(2);
        try {
            System.out.println(3);
            System.out.println(0/0);    // ArithmeticException을 발생시킨다.
            System.out.println(4);    // 실행되지 않는다.
        } catch (Exception e)    { // ArithmeticException대신 Exception을 사용.
            System.out.println(5);
        }    // try-catch의 마지막
        System.out.println(6);
    }    // main메서드의 마지막
}
```

#### [실행결과]

```
1
2
3
```

5  
6

이 예제는 예제8-5를 변경한 것인데, 결과는 같다. catch블럭의 괄호()에 ArithmeticException클래스의 참조변수 대신에 Exception클래스의 참조변수를 선언하였다.

ArithmeticException 클래스는 Exception클래스의 자손클래스이므로, ArithmeticException클래스의 인스턴스와 Exception클래스와의 instanceof연산결과가 true가 되어 Exception클래스의 참조변수를 선언한 catch블럭의 문장들이 수행되고 예외는 처리되는 것이다.

#### [예제8-11] ExceptionEx11.java

```
class ExceptionEx11 {  
    public static void main(String args[]) {  
        System.out.println(1);  
        System.out.println(2);  
        try {  
            System.out.println(3);  
            System.out.println(0/0);    // ArithmeticException을 발생시킨다.  
            System.out.println(4);    // 실행되지 않는다.  
        } catch (ArithmeticException ae)    {  
            if (ae instanceof ArithmeticException)  
                System.out.println("true");  
            System.out.println("ArithmeticException");  
        } catch (Exception e)    {  
            System.out.println("Exception");  
        }    // try-catch의 마지막  
        System.out.println(6);  
    }    // main메서드의 마지막  
}
```

#### [실행결과]

1  
2

```
3
true
ArithmeticException
6
```

위의 예제에서는 두 개의 catch블럭, ArithmeticException클래스의 참조변수를 선언한 것과 Exception클래스의 참조변수를 선언한 것을 사용하였다.

try블럭에서 ArithmeticException이 발생하였으므로, catch블럭을 하나씩 차례로 검사하게 되는데, 첫 번째 검사에서 일치하는 catch블럭을 찾았기 때문에, 두 번째 catch블럭은 검사하지 않게 된다.

만일 try블럭 내에서 ArithmeticException이 아닌 다른 종류의 예외가 발생한 경우에는 두 번째 catch블럭인 Exception클래스의 참조변수를 선언한 곳에서 처리되었을 것이다.

이처럼, try-catch문의 마지막에 Exception클래스 타입의 참조변수를 선언한 catch블럭을 사용하면, 어떤 종류의 예외가 발생하더라도 이 catch블럭에 의해 처리되도록 할 수 있다.

예외가 발생했을 때 생성되는 예외클래스의 인스턴스에는 발생한 예외에 대한 정보가 담겨져 있으며, getMessage()와 printStackTrace()를 통해서 이 정보들을 얻을 수 있다.

catch블럭의 괄호()에 선언된 참조변수를 통해 이 인스턴스에 접근할 수 있다. 이 참조변수는 선언된 catch블럭 내에서만 사용 가능하며, 주로 사용되는 메서드는 다음과 같다.

**printStackTrace()** - 예외 발생 당시의 호출스택(Call Stack)에 있었던 메서드의 정보와 예외 메시지를 화면에 출력한다.

**getMessage()** - 발생한 예외클래스의 인스턴스에 저장된 예외메시지를 얻을 수 있다.

#### [예제8-12] ExceptionEx12.java

```
class ExceptionEx12 {
    public static void main(String args[]) {
        System.out.println(1);
        System.out.println(2);
        try {
```

```

        System.out.println(3);
        System.out.println(0/0); // 0으로 나눠서 ArithmeticException을 발생시킨
다.

        System.out.println(4);    // 실행되지 않는다.
    } catch (ArithmeticException ae)    {
        ae.printStackTrace();    // 참조변수 ae를 통해 생성된 인스턴스에 접근할 수
있다.

        System.out.println("예외메시지 : " + ae.getMessage());
    }    // try-catch의 마지막
    System.out.println(6);
}    // main메서드의 마지막
}

```

#### [실행결과]

```

1
2
3
java.lang.ArithmeticException: / by zero
    at ExceptionEx12.main(ExceptionEx12.java:7)
예외메시지 : / by zero
6

```

위 예제의 결과는 예외가 발생해서 비정상적으로 종료되었을 때의 결과와 비슷하지만 예외는 처리되었으며 프로그램은 정상적으로 종료되었다.

대신 ArithmeticException인스턴스의 printStackTrace()를 사용해서, 호출스택(Call Stack)에 대한 정보와 예외메시지를 출력하였다.

이처럼 try-catch문으로 예외처리를 하여 예외가 발생해도 비정상적으로 종료하지 않도록 해 주는 동시에, printStackTrace() 또는 getMessage()와 같은 메서드를 통해서 예외의 발생원인을 알 수 있다.

그리고 printStackTrace(PrintStream s) 또는 printStackTrace(PrintWriter s)를 사용하면, 발생한 예외에 대한 정보를 파일에 저장할 수도 있다.

#### [예제8-13] ExceptionEx13.java

```

import java.io.*;

class ExceptionEx13 {
    public static void main(String args[]) {

        PrintStream ps = null;
        FileOutputStream fos=null;

        try {
            fos = new FileOutputStream("error.log");
            ps = new PrintStream(fos);

            System.out.println(1);
            System.out.println(2);

            System.out.println(3);
            System.out.println(0/0);    // 0으로 나눠서 ArithmeticException을 발생
시킨다.
            System.out.println(4);    // 실행되지 않는다.
        } catch (Exception ae)    {
            ae.printStackTrace(ps);
            ps.println("예외메시지 : " + ae.getMessage());    // 화면대신 error.log파
일에 출력한다.
        }    // try-catch의 마지막
        System.out.println(6);
    }    // main메서드의 마지막
}

```

#### [실행결과]

```

C:\wj2sdk1.4.1\work>java ExceptionEx13
1
2
3
6
C:\wj2sdk1.4.1\work>type error.log

```

```
java.lang.ArithmeticException: / by zero
    at ExceptionEx13.main(ExceptionEx13.java:17)
예외메시지 : / by zero
```

try블럭 내에 선언된 변수는, try블럭 밖에서 사용할 수 없기 때문에 변수 ps와 fos를 try블럭 외부에 선언하였다.

위의 예제는 printStackTrace() 대신 printStackTrace(PrintStream s)를 사용해서, 호출스택의 내용을 화면에 출력하는 대신 error.log 파일에 저장하는 일을 한다.

그 래서 이 전 예제에서는 printStackTrace()에 의해서 화면에 출력되었던 내용이 error.log파일에 저장되었다. 파일 error.log는 생성 시에 경로없이 파일의 이름만 지정하였으므로 현재 디렉토리에 생성될 것이다. 생성된 error.log파일의 내용은 다음과 같다.

```
java.lang.ArithmeticException: / by zero
    at ExceptionEx13.main(ExceptionEx13.java:17)
예외 메시지 : / by zero
```

위 예제를 잘 보면 PrintStream클래스의 참조변수인 ps가 main메서드 내에 선언되어 있기 때문에, main메서드에서만 참조변수 ps가 사용 가능하다. 즉, main메서드에서만 error.log파일에 접근할 수 있으므로, main메서드가 아닌 다른 메서드에서 발생한 예외에 대한 내용을 기록할 수 없다.

이럴 때는 System.err을 이용하면 된다. 다음은 System.err을 이용하여 위의 예제를 변경한 것이다.

#### [예제8-14] ExceptionEx14.java

```
import java.io.*;
import java.util.*;

class ExceptionEx14 {
    public static void main(String args[]) {

        PrintStream ps = null;
```

```

        FileOutputStream fos=null;
        try {
            fos = new FileOutputStream("error.log");    // error.log파일에 출력할 준
            비를 한다.
            ps=new PrintStream(fos);
            System.setErr(ps);        // err의 출력을 화면이 아닌, error.log파일로 변경
            한다.
            System.out.println(1);
            System.out.println(2);
            System.out.println(3);
            System.out.println(0/0);    // 0으로 나뉘서 ArithmeticException을 발생
            시킨다.
            System.out.println(4);    // 실행되지 않는다.
        } catch (Exception ae)    {
            System.err.println("-----");
            System.err.println("예외발생시간 : " + new Date()); // 현재시간을 기록한
            다.
            ae.printStackTrace(System.err);
            System.err.println("예외메시지 : " + ae.getMessage());
            System.err.println("-----");
        }    // try-catch의 마지막
        System.out.println(6);
    }    // main메서드의 마지막
}

```

#### [실행결과]

```
c:\wj2sdk1.4.1\work>java ExceptionEx14
```

```
1
```

```
2
```

```
3
```

```
6
```

```
c:\wj2sdk1.4.1\work>type error.log
```

```
-----
```

```
예외발생시간 : Thu Jan 30 12:36:44 KST 2003
```

```
java.lang.ArithmeticException: / by zero
```

```
at ExceptionEx13.main(ExceptionEx14.java:17)
```

예외메시지 : / by zero

-----

System.out이나 System.err은 System클래스의 static멤버로서 프로그램의 어디에서나 사용할 수 있다. System.err은 System.out과 마찬가지로 기본적으로 출력방향이 화면으로 되어 있어서, setErr메서드를 이용해서 출력방향을 바꾸지 않는 한 err에 출력하는 내용은 모두 화면에 보여지도록 되어 있다.

위의 예제에서는 setErr메서드를 이용해서 System.err의 출력방향을 error.log라는 이름의 파일로 변경한다. 출력방향이 변경되었기 때문에, System.err의 println메서드나 print메서드를 이용해서 출력하는 내용은 error.log파일에 저장된다.

[참고]err은 System클래스의 static멤버로서 PrintStream클래스 타입의 참조변수이며, System.err은 System.out처럼 출력방향이 화면으로 설정되어 있다.

파일입출력에 관한 내용이 나와서 이번 예제가 다소 어렵게 느껴질 수도 있겠지만, 일단은 프로그램에서 발생한 예외에 대한 기록을 파일로 남기는 방법이 있다는 정도만 이해하면 된다.

## 1.8 finally블럭

finally 블럭은 try-catch문과 함께, 예외의 발생여부에 상관없이 실행되어야 할 코드를 포함시킬 목적으로 사용된다. try-catch문의 끝에 선택적으로 덧붙여 사용할 수 있으며, try-catch-finally의 순서로 구성된다.

```
try {  
    // 예외가 발생할 가능성이 있는 문장들을 넣는다.  
} catch (Exception1 e1) {  
    // 예외처리를 위한 문장을 적는다.  
} finally {  
    // 예외의 발생여부에 관계없이 항상 수행되어야 하는 문장들을 넣는다.  
    // finally블럭은 try-catch문의 맨 마지막에 위치해야 한다.  
}
```



예외가 발생한 경우에는 try -> catch -> finally의 순으로 실행되고, 예외가 발생하지 않은 경우에는 try -> finally의 순으로 실행된다.

#### [예제8-15] FinallyTest.java

```
class FinallyTest {
    public static void main(String args[]) {
        try {
            startInstall();      // 프로그램 설치에 필요한 준비를 한다.
            copyFiles();         // 파일들을 복사한다.
            deleteTempFiles();   // 프로그램 설치에 사용된 임시파일들을 삭제한다.
        } catch (Exception e) {
            e.printStackTrace();
            deleteTempFiles();
        } // try의 끝
    } // main의 끝

    static void startInstall() { /* 프로그램 설치에 필요한 준비를 하는 코드를 적는다. */ }
    static void copyFiles() { /* 파일들을 복사하는 코드를 적는다. */ }
    static void deleteTempFiles() { /* 임시파일들을 삭제하는 코드를 적는다. */ }
}
```

[참고] startInstall(), copyFiles(), deleteTempFiles()에 주석문 이외에는 아무런 문장이 없지만, 각 메서드의 의미에 해당하는 작업을 수행하는 코드들이 작성되어 있다고 가정하자.

이 예제가 하는 일은 프로그램설치를 위한 준비를 하고 파일들을 복사하고 설치가 완료되면, 프로그램을 설치하는데 사용된 임시파일들을 삭제하는 순서로 진행된다. 프로그램의 설치과정 중에 예외가 발생하더라도, 설치에 사용된 임시파일들이 삭제되도록 catch블럭에 deleteTempFiles()메서드를 넣었다.

결국 try블럭의 문장을 수행하는 동안에(프로그램을 설치하는 과정에), 예외의 발생여부에 관계없이 deleteTempFiles()메서드는 실행되어야 하는 것이다.

이럴 때 finally블럭을 사용하면 좋다. 아래의 코드는 위의 예제를 finally블럭을 사용해서 변경한 것이며, 두 예제의 기능은 동일하다.

#### [예제8-16] FinallyTest2.java

```

class FinallyTest2 {
    public static void main(String args[]) {
        try {
            startInstall();      // 프로그램 설치에 필요한 준비를 한다.
            copyFiles();         // 파일들을 복사한다.
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            deleteTempFiles();   // 프로그램 설치에 사용된 임시파일들을 삭제한다.
        } // try의 끝
    } // main의 끝

    static void startInstall() { /* 프로그램 설치에 필요한 준비를 하는 코드를 적는다. */ }
    static void copyFiles() { /* 파일들을 복사하는 코드를 적는다. */ }
    static void deleteTempFiles() { /* 임시파일들을 삭제하는 코드를 적는다. */ }
}

```

#### [예제8-17] FinallyTest3.java

```

class FinallyTest3 {
    public static void main(String args[]) {
        // method1()은 static메서드이므로 아래와 같이 인스턴스 생성없이 직접 호출이 가능하다.
        FinallyTest3.method1();
        System.out.println("method1()의 수행을 마치고 main메서드로 돌아왔습니다.");
    } // main메서드의 끝입니다.

    static void method1() {
        try {
            System.out.println("method1()이 호출되었습니다.");
            return ; // 현재 실행 중인 메서드를 종료한다.
        } catch (Exception e) {
            e.printStackTrace();
        } finally {

```

```

        System.out.println("method1()의 finally블럭이 실행되었습니다.");
    }
} // method1메서드의 끝입니다.
}

```

#### [실행결과]

method1()이 호출되었습니다.  
 method1()의 finally블럭이 실행되었습니다.  
 method1()의 수행을 마치고 main메서드로 돌아왔습니다.

위의 결과에서 알 수 있듯이, try블럭에서 return문이 실행되는 경우에도 finally블럭의 문장들이 먼저 실행된 후에, 현재 실행 중인 메서드를 종료한다.

이와 마찬가지로 catch블럭의 문장 수행중에 return문을 만나도 finally블럭의 문장들은 수행된다.

### 1.9 메서드에 예외 선언하기

예외를 처리하는 방법에는 지금까지 배워 온 try-catch문을 사용하는 것 이외에, 예외를 메서드에 선언하는 방법이 있다.

메서드에 예외를 선언하려면, 메서드의 선언부에 키워드 throws를 사용해서 메서드 내에서 발생할 수 있는 예외를 적어주기만 하면 된다. 그리고, 예외가 여러 개일 경우에는 쉼표(,)로 구분한다.

```

void method() throws Exception1, Exception2, ... ExceptionN {
    // 메서드의 내용
}

```

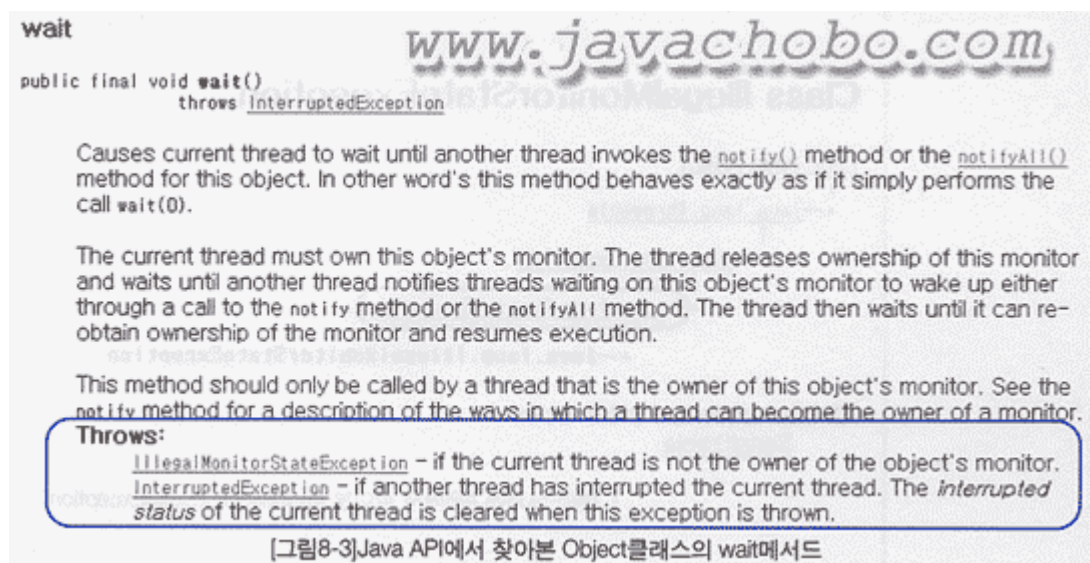
[참고]예외를 발생시키는 키워드 throw와 예외를 메서드에 선언할 때 쓰이는 throws를 잘 구별하도록 한다.

이렇게 메서드의 선언부에 예외를 선언함으로써, 메서드를 사용하려는 사람이 메서드의 선언

부를 보았을 때, 이 메서드를 사용하기 위해서는 어떠한 예외들이 처리되어야 하는지 쉽게 알 수 있다.

기존의 많은 언어들에서는 메서드에 예외선언을 하지 않기 때문에, 경험 많은 프로그래머가 아니고서는 어떤 상황에 어떤 종류의 예외가 발생할 가능성이 있는지 충분히 예측하기 힘들기 때문에 그에 대한 대비를 하는 것이 어려웠다.

그러나 자 바에서는 메서드를 작성할 때 메서드 내에서 발생할 가능성이 있는 예외를 메서드의 선언부에 명시하여, 이 메서드를 사용하는 쪽에서는 이에 대한 처리를 하도록 강요하기 때문에, 프로그래머들의 짐을 덜어 주는 것은 물론이고 보다 견고한 프로그램 코드를 작성할 수 있도록 도와준다.



위의 그림은 Java API문서에서 찾아본 java.lang.Object클래스의 wait메서드에 대한 설명이다.

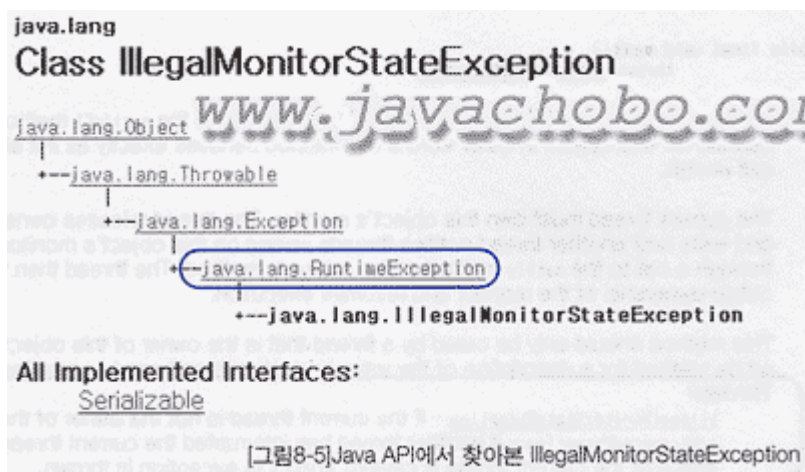
메서드의 선언부에 InterruptedException이 키워드 throws와 함께 적혀 있는 것을 볼 수 있다. 이 것이 의미하는 바는 이 메서드에서는 InterruptedException이 발생할 수 있으니, 이 메서드를 호출하고자 하는 메서드에서는 InterruptedException을 처리 해주어야 한다는 것이다.

InterruptedException에 밑줄이 있는 것으로 보아 링크가 걸려 있음을 알 수 있을 것이다. 이 링크를 클릭하면, InterruptedException에 대한 설명을 볼 수 있다.



위의 그림에서 볼 수 있는 것처럼, InterruptedException은 Exception클래스의 자손임을 알 수 있다. 따라서 InterruptedException은 반드시 처리해주어야 하는 예외임을 알 수 있다. 그래서 wait메서드의 선언부에 키워드 throws와 함께 선언되어져 있는 것이다. Java API의 wait메서드 설명의 아래쪽에 있는 'Throws:'를 보면, wait메서드에서 발생할 수 있는 예외의 리스트와 언제 발생하는가에 대한 설명이 덧붙여져 있다.

여기에는 두 개의 예외가 적혀 있는데 메서드에 선언되어 있는 InterruptedException외에 또 하나의 예외(IllegalMonitorStateException)가 있다. IllegalMonitorStateException 역시 링크가 걸려 있으므로 클릭하면, IllegalMonitorStateException에 대한 정보를 얻을 수 있다.



이 그림에서도 볼 수 있듯이 IllegalMonitorStateException은 RuntimeException클래스의 자손이므로 IllegalMonitorStateException은 예외처리를 해주지 않아도 된다. 그래서 wait메서드의 선언부에 IllegalMonitorStateException을 적지 않은 것이다.

지금까지 알아본 것처럼, 메서드에 예외를 선언할 때 일반적으로 RuntimeException클래스 들은 적지 않는다. 이 들을 메서드 선언부의 throws에 선언한다고 해서 문제가 되지는 않지만,

보통 반드시 처리해주어야 하는 예외들만 선언한다.

이처럼 Java API문서를 통해 사용하고자 하는 메서드의 선언부와 'Throws:'를 보고, 이 메서드에서는 어떤 예외가 발생할 수 있으며 반드시 처리해주어야 하는 예외는 어떤 것들이 있는지 확인하는 것이 좋다.

[참고] 반드시 처리해주어야 하는 예외는 RuntimeException클래스들을 제외한 나머지 클래스들이다.

사실 예외를 메서드의 throws에 명시하는 것은 예외를 처리하는 것이 아니라, 자신(예외가 발생할 가능성이 있는 메서드)을 호출한 메서드에게 예외를 전달하여 예외처리를 떠맡기는 것이다.

예외를 전달받은 메서드가 또다시 자신을 호출한 메서드에게 전달할 수 있으며, 이런 식으로 계속 호출스택에 있는 메서드들을 따라 전달되다가 제일 마지막에 있는 main메서드에서도 예외가 처리되지 않으면, main메서드 마져 종료되어 프로그램이 전체가 종료된다.

#### [예제8-18] ExceptionEx18.java

```
class ExceptionEx18 {
    public static void main(String[] args) throws Exception
    {
        method1(); // 같은 클래스내의 static멤버이므로 객체생성없이 직접 호출가능.
    }    // main메서드의 끝

    static void method1() throws Exception {
        method2();
    }    // method1의 끝

    static void method2() throws Exception {
        throw new Exception();
    }    // method2의 끝
}
```

#### [실행결과]

```
java.lang.Exception
    at ExceptionEx18.method2(ExceptionEx18.java:12)
```

```
at ExceptionEx18.method1(ExceptionEx18.java:8)
at ExceptionEx18.main(ExceptionEx18.java:4)
```

위의 실행결과를 보면, 프로그램의 실행도중 java.lang.Exception이 발생하여 비정상적으로 종료했다는 것과 예외가 발생했을 때 호출스택(Call Stack)의 내용을 알 수 있다

위의 결과로부터 다음과 같은 사실을 알 수 있다.

- 예외가 발생했을 때, 모두 3개의 메서드(main, method1, method2)가 호출스택에 있었으며,
- 예외가 발생한 곳은 제일 윗 줄에 있는 method2()라는 것과
- main메서드가 method1()를, 그리고 method1()은 method2()를 호출했다는 것

위의 예제를 보면, method2()에서 throw new Exception(); 문장에 의해 예외가 강제적으로 발생했으나 try-catch문으로 예외처리를 해주지 않았으므로, method2()는 종료되면서 예외를 자신을 호출한method1()에게 넘겨준다. method1()에서도 역시 예외처리를 해주지 않았으므로 종료되면서 main메서드에게 예외를 넘겨준다.

그러나 main메서드에서 조차 예외처리를 해주지 않았으므로 main메서드가 종료되어 프로그램이 예외로 인해 비정상적으로 종료하게 되는 것이다.

이처럼 예외가 발생한 메서드에서 예외처리를 하지 않고 자신을 호출한 메서드에게 예외를 넘겨줄 수는 있지만, 이것으로 예외가 처리된 것은 아니고, 예외를 단순히 전달만 하는 것이다. 결국 어디서든 간에 반드시 try-catch문을 사용해서 예외처리를 해주어야 한다.

#### [예제8-19] ExceptionEx19.java

```
class ExceptionEx19 {
    public static void main(String[] args)
    {
        method1();        // 같은 클래스내의 static멤버이므로 객체생성없이 직접 호출
        가능.
    }    // main메서드의 끝
```

```

static void method1() {
    try {
        throw new Exception();
    } catch (Exception e) {
        System.out.println("method1메서드에서 예외가 처리되었습니다.");
        e.printStackTrace();
    }
} // method1의 끝
}

```

#### [실행결과]

method1메서드에서 예외가 처리되었습니다.

java.lang.Exception

at ExceptionEx19.method1(ExceptionEx19.java:9)

at ExceptionEx19.main(ExceptionEx19.java:4)

예외는 처리되었지만, printStackTrace()를 통해 예외에 대한 정보를 화면에 출력하였다. 예외가 method1에서 발생했으며, main메서드가 method1을 호출했음을 알 수 있다.

#### [예제8-20] ExceptionEx20.java

```

class ExceptionEx20 {
    public static void main(String[] args)
    {
        try {
            method1();
        } catch (Exception e) {
            System.out.println("main메서드에서 예외가 처리되었습니다.");
            e.printStackTrace();
        }
    } // main메서드의 끝

    static void method1() throws Exception {
        throw new Exception();
    }
}

```



```

    }    // method1메서드의 끝
}

```

#### [실행결과]

main메서드에서 예외가 처리되었습니다.

```
java.lang.Exception
```

```
    at ExceptionEx20.method1 (ExceptionEx20.java:13)
```

```
    at ExceptionEx20.main (ExceptionEx20.java:5)
```

두 예제 모두 main메서드가 method1()을 호출하며, method1()에서 예외가 발생한다. 차이점은 예외처리 방법에 있다.

첫 번째 예제는 method1()에서 예외처리를 했고, 두 번째 예제는 method1()에서 예외를 선언하여 자신을 호출하는 메서드(main메서드)에 예외를 전달 했으며, 호출한 메서드(main메서드)에서는 try-catch문으로 예외처리를 했다.

첫 번째 예제처럼 예외가 발생한 메서드(method1) 내에서 처리되어지면, 호출한 메서드(main)에서는 예외가 발생했다는 사실조차 모르게 된다.

두 번째 예제처럼 예외가 발생한 메서드에서 예외를 처리하지 않고 호출한 메서드로 넘겨주면, 호출한 메서드에서는 method1()을 호출한 라인에서 예외가 발생한 것으로 간주되어 이에 대한 처리를 하게 된다.

이처럼 예외가 발생한 메서드(method1())에서 예외를 처리할 수도 있고, 예외가 발생한 메서드를 호출한 메서드(main메서드)에서 처리할 수도 있다. 또는 두 메서드가 예외처리를 분담할 수도 있다.

이번엔 보다 그럴듯한 예를 들어보도록 하겠다.

#### [예제8-21] ExceptionEx21.java

```
import java.io.*;
```

```
class ExceptionEx21 {
```

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

```
    {
```

```
        // command line에서 입력받은 값을 이름으로 갖는 파일을 생성한다.
```

```
        File f = createFile(args[0]);
```

```
        System.out.println( f.getName() + " 파일이 성공적으로 생성되었습니다.");
```

```

} // main메서드의 끝

static File createFile(String fileName) {
    try {
        if (fileName==null || fileName.equals(""))
            throw new Exception("파일이름이 유효하지 않습니다.");
    } catch (Exception e) {
        // fileName이 부적절한경우, 파일 이름을 '제목없음.txt'로 한다.
        fileName = "제목없음.txt";
    } finally {
        File f = new File(fileName);    // File클래스의 객체를 만든다.
        createNewFile(f);                // 생성된 객체를 이용해서 파일을 생성한다.
        return f;
    }
} // createFile메서드의 끝

static void createNewFile(File f) {
    try {
        f.createNewFile();    // 파일을 생성한다.
    } catch (Exception e){ }
} // createNewFile메서드의 끝
} // 클래스의 끝

```

#### [실행결과]

C:\wj2sdk1.4.1\work>java ExceptionEx21 "test.txt"

test.txt 파일이 성공적으로 생성되었습니다.

C:\wj2sdk1.4.1\work>java ExceptionEx21 ""

제목없음.txt 파일이 성공적으로 생성되었습니다.

C:\wj2sdk1.4.1\work>dir \*.txt

드라이브 C에 레이블이 없습니다

볼륨 일련 번호 251C-08DD

디렉터리 C:\wj2sdk1.4\work

제목없음 TXT 0 03-01-24 0:47 제목없음.txt

TEST TXT 0 03-01-24 0:47 test.txt

1개 파일 0 바이트

0개 디렉터리 848.14 MB 사용 가능

[참고]실행 시 커맨드라인에 파일이름을 입력하지 않으면, args[0]이 유효하지 않으므로 7번째 줄(File f = createFile(args[0]);)에서 ArrayIndexOutOfBoundsException이 발생한다.

이 예제는 예외가 발생한 메서드에서 직접 예외를 처리하도록 되어 있다. createFile메서드를 호출한 main메서드에서는 예외가 발생한 사실을 알지 못한다. 적절하지 못한 파일이름(fileName)이 입력되면, 예외를 발생시키고, catch블럭에서, 파일이름을 '제목없음.txt'로 설정해서 파일을 생성한다. 그리고, finally블럭에서는 예외의 발생여부에 관계없이 파일을 생성하는 일을 한다.

[참고]File클래스의 createNewFile()은 예외가 선언된 메서드 이므로 finally블럭 내에 또다시 try-catch문으로 처리해야하므로 좀 복잡해 진다. 여러분들의 이해를 돕기 위해 예제의 기본 흐름을 되도록 간단히 하려고 내부적으로 예외처리를 한 createNewFile(File f)메서드를 만들어서 사용했다.

#### [예제8-22] ExceptionEx22.java

```
import java.io.*;

class ExceptionEx22 {
    public static void main(String[] args)
    {
        try {
            File f = createFile(args[0]);
            System.out.println( f.getName() + "파일이 성공적으로 생성되었습니다.");
        } catch (Exception e) {
            System.out.println(e.getMessage() +" 다시 입력해 주시기 바랍니다.");
        }
    }
    // main메서드의 끝
}
```

```

static File createFile(String fileName) throws Exception {
    if (fileName==null || fileName.equals(""))
        throw new Exception("파일 이름이 유효하지 않습니다.");
    File f = new File(fileName);           // File클래스의 객체를 만든다.
    f.createNewFile();    // File객체의 createNewFile메서드를 이용해서 실제파일
을 생성한다.
    return f;           // 생성된 객체의 참조를 반환한다.
}    // createFile메서드의 끝
}    // 클래스의 끝

```

#### [실행결과]

C:\wj2sdk1.4.1\work>java ExceptionEx22 test2.txt

test2.txt파일이 성공적으로 생성되었습니다.

C:\wj2sdk1.4.1\work>java ExceptionEx22 ""

파일 이름이 유효하지 않습니다. 다시 입력해 주시기 바랍니다.

이 두 번째 예제에서는 예외가 발생한 createFile메서드에서 잘못 입력된 파일 이름을 임의로 처리하지 않고, 호출한 main메서드에게 예외가 발생했음을 알려서 파일 이름을 다시 입력 받도록 하였다.

첫 번째 예제와는 달리 createFile메서드에 예외를 선언했기 때문에, File클래스의 createNewFile()에 대한 예외처리를 하지 않아도 되므로 createNewFile(File f)메서드를 굳이 따로 만들지 않았다.

두 예제 모두 커맨드라인으로부터 파일 이름을 입력 받아서 파일을 생성하는 일을 하며, 파일 이름을 잘못 입력했을 때(null 또는 빈 문자열""일 때) 예외가 발생하도록 되어 있다.

두 예제의 차이점의 예외의 처리방법에 있다. 첫 번째 예제는 예외가 발생한 createFile메서드 자체 내에서 처리를 하며, 두 번째 예제는 createFile메서드를 호출한 메서드(main메서드)에서 처리한다.

이처럼 예외가 발생한 메서드 내에서 자체적으로 처리해도 되는 것은 메서드 내에서 try-catch문을 사용해서 처리하고, 두 번째 예제처럼 메서드에 호출 시 넘겨받아야 할 값(fileName)을 다시 받아야 하는 경우(메서드에서 자체적으로 해결이 안 되는 경우)에는 예외를 메서드에 선언해서, 호출한 메서드에서 처리하면 된다.

### 1.10 예외 되던지기(Exception Re-throwing)

한 메서드에서 발생할 수 있는 예외가 여럿인 경우, 몇 개는 try-catch문을 통해서 메서드 내에서 자체적으로 처리하고, 그 나머지는 선언부에 지정하여 호출한 메서드에서 처리하도록 함으로써, 양쪽에서 나눠서 처리되도록 할 수 있다.

그리고 심지어는 단 하나의 예외에 대해서도 예외가 발생한 메서드와 호출한 메서드, 양쪽에서 처리하도록 할 수 있다.

이 것은 예외를 처리한 후에 인위적으로 다시 발생시키는 방법을 통해서 가능한데, 이 것을 '예외 되던지기(Exception Re-throwing)'이라고 한다.

먼저 예외가 발생할 가능성이 있는 메서드에서 try-catch문을 사용해서 예외를 처리해주고, catch문에서 필요한 작업을 행한 후에 throw문을 사용해서 예외를 다시 발생시킨다.

다시 발생한 예외는 이 메서드를 호출한 메서드에게 전달되고, 호출한 메서드의 try-catch문에서 예외를 또다시 처리하면 된다.

이 방법은 하나의 예외에 대해서 예외가 발생한 메서드와 이를 호출한 메서드 양쪽 모두에서 처리해줘야 할 작업이 있을 때 사용된다.

이 때 주의해야할 점은 예외가 발생할 메서드에서는 try-catch문을 사용해서 예외처리를 해주고 동시에 메서드의 선언부에 발생할 예외를 throws에 지정해줘야 한다는 것이다.

#### [예제8-23] ExceptionEx23.java

```
class ExceptionEx23 {
    public static void main(String[] args)
    {
        try {
            method1();
        } catch (Exception e) {
            System.out.println("main메서드에서 예외가 처리되었습니다.");
        }
    } // main메서드의 끝

    static void method1() throws Exception {
```

```

    try {
        throw new Exception();
    } catch (Exception e) {
        System.out.println("method1메서드에서 예외가 처리되었습니다.");
        throw e;           // 다시 예외를 발생시킨다.
    }
} // method1메서드의 끝
}

```

#### [실행결과]

method1메서드에서 예외가 처리되었습니다.

main메서드에서 예외가 처리되었습니다.

결과에서 알 수 있듯이 method1()와 main메서드 양쪽의 catch블럭이 모두 수행되었음을 알 수 있다. method1()의 catch블럭에서 예외를 처리하고도 throw문을 통해 다시 예외를 발생시켰다. 그리고 이 예외를 main메서드 한번 더 처리하였다.

#### 1.11 사용자정의 예외 만들기

기존의 정의된 예외 클래스 외에 필요에 따라 프로그래머가 새로운 예외 클래스를 정의하여 사용할 수 있다. 보통 Exception클래스로부터 상속받는 클래스를 만들지만, 필요에 따라서 알맞은 예외 클래스를 선택할 수 있다.

```

class MyException extends Exception {
    MyException(String msg) {    // 생성자.
        super(msg);             // 조상클래스인 Exception클래스의 생성자를 호출한다.
    }
}

```

Exception 클래스로부터 상속받아서 MyException클래스를 만들었다. 필요하다면, 멤버변수

나 메서드를 추가할 수 있다. Exception클래스는 생성 시에 String값을 받아서 메시지로 저장할 수 있다. 여러분이 만든 사용자정의 예외 클래스도 메시지를 저장할 수 있으려면, 위에서 보는 것과 같이 String을 매개변수로 받는 생성자를 추가해주어야 한다.

```
class MyException extends Exception {
    private final int errorCode = 100; // 에러 코드 값을 저장하기 위한 필드를 추가 했다.
    MyException(String msg) {    // 생성자.
        super(msg);
    }
    public int getCode()    {    // 에러 코드를 얻을수 있는 메서드도 추가했다.
        return errorCode;    // 이 메서드는 주로 getMessage()와 함께 사용될 것이다.
    }
}
```

이전의 코드를 좀더 개선하여 메시지 뿐만 아니라 에러코드 값도 저장할 수 있도록 errorCode와 getCode()를 MyException클래스의 멤버로 추가했다.

이렇게 함으로써 MyException이 발생했을 때, catch블럭에서 getMessage메서드와 getCode메서드를 사용해서, 에러코드와 메시지를 모두 얻을 수 있을 것이다.

#### [예제8-24] NewExceptionTest.java

```
class NewExceptionTest {
    public static void main(String args[]) {
        try {
            startInstall();    // 프로그램 설치에 필요한 준비를 한다.
            copyFiles();    // 파일들을 복사한다.
        } catch (SpaceException e)    {
            System.out.println("에러 메세지 : " + e.getMessage());
            e.printStackTrace();
            System.out.println("공간을 확보한 후에 다시 설치하시기 바랍니다.");
        } catch (MemoryException me)    {
            System.out.println("에러 메세지 : " + me.getMessage());
            me.printStackTrace();
            System.gc();    // Garbage Collection을 수행하여 메모리를 늘려준다.
        }
    }
}
```

```

        System.out.println("다시 설치를 시도하세요.");
    } finally {
        deleteTempFiles();    // 프로그램 설치에 사용된 임시파일들을 삭제한다.
    } // try의 끝
}    // main의 끝

static void startInstall() throws SpaceException, MemoryException {
    if(!enoughSpace())        // 충분한 설치 공간이 없으면...
        throw new SpaceException("설치할 공간이 부족합니다.");
    if (!enoughMemory())      // 충분한 메모리가 없으면...
        throw new MemoryException("메모리가 부족합니다.");
} // startInstall메서드의 끝

static void copyFiles() { /* 파일들을 복사하는 코드를 적는다. */ }
static void deleteTempFiles() { /* 임시파일들을 삭제하는 코드를 적는다. */ }

static boolean enoughSpace() {
    // 설치하는데 필요한 공간이 있는지 확인하는 코드를 적는다.
    return false;
}

static boolean enoughMemory() {
    // 설치하는데 필요한 메모리공간이 있는지 확인하는 코드를 적는다.
    return true;
}
} // ExceptionTest클래스의 끝

class SpaceException extends Exception {
    SpaceException(String msg) {
        super(msg);
    }
}

class MemoryException extends Exception {
    MemoryException(String msg) {
        super(msg);
    }
}

```



```
}  
}
```

#### [실행결과]

에러 메시지 : 설치할 공간이 부족합니다.

SpaceException: 설치할 공간이 부족합니다.

at NewExceptionTest.startInstall(NewExceptionTest.java:22)

at NewExceptionTest.main(NewExceptionTest.java:4)

공간을 확보한 후에 다시 설치하시기 바랍니다.

이 예제를 실제 설치 프로그램과 비슷하게 보이려고 하다보니 좀 복잡해 졌다.

MemoryException과 SpaceException, 이 두 개의 사용자정의 예외 클래스를 새로 만들어서 사용했다. SpaceException은 프로그램을 설치하려는 곳에 충분한 공간이 없을 경우에 발생하도록 했으며, MemoryException은 설치작업을 수행하는데 메모리 충분히 확보되지 않았을 경우에 발생하도록 하였다.

이 두 개의 예외는 startInstall()메서드를 수행하는 동안에 발생할 수 있으며, enoughSpace()와 enoughMemory()의 실행결과에 따라서 발생하는 예외의 종류가 달라지도록 했다.

이번 예제에서 enoughSpace메서드와 enoughMemory메서드는 단순히 false와 true를 각각 return하도록 되어 있지만, 설치공간과 사용 가능한 메모리를 확인하는 기능을 한다고 가정하였다.

# [Java의 정석]제9장 java.lang패키지 - 1.Object클래스

자바의정석

2012/12/22 17:13

<http://blog.naver.com/gphic/50157740357>

java.lang패키지는 자바프로그래밍에 가장 기본이 되는 클래스들을 포함하고 있다. 그렇기 때문에 java.lang패키지의 클래스들은 import문을 사용하지 않고도 사용할 수 있도록 되어 있다. 그 동안 String클래스나 System클래스를 import문을 사용하지 않고도 사용할 수 있었던 이유가 바로 java.lang패키지에 속한 클래스들이기 때문이었던 것이다.

우선 java.lang패키지의 여러 클래스들 중에서도 자주 사용되는 클래스 몇 가지만을 골라서 학습해보도록 하자.

## 1. Object클래스

Object클래스에 대해서는 클래스의 상속을 학습할 때 배웠지만, 여기서는 보다 자세히 알아보도록 하자. Object클래스는 모든 클래스의 최고 조상이기 때문에 Object클래스의 멤버들은 모든 클래스에서 바로 사용 가능하다.

Object클래스는 멤버변수는 없고 8개의 메서드만 가지고 있다. 이 메서드들은 모든 인스턴스가 가져야 할 기본적인 것들이며, 우선 이 중에서 중요한 몇 가지에 대해서 알아보도록 하자.

### 1.1 equals메서드

매개변수로 객체의 참조변수를 받아서 비교하여 그 결과를 boolean값으로 알려 주는 역할을 한다. 아래의 코드는 Object클래스에 정의되어 있는 equals메서드의 실제 내용이다.

```
public boolean equals(Object obj) {  
    return (this==obj);  
}
```

위의 코드에서 알 수 있듯이 두 객체의 같고 다름을 참조변수의 값으로 판단한다. 그렇기 때문에 서로 다른 두 객체를 equals메서드로 비교하면 항상 false를 결과로 얻게 된다.

[참고]객체를 생성할 때, 메모리의 비어있는 공간을 찾아 생성하므로 서로 다른 두 개의 객체가 같은 주소를 갖는 일은 있을 수 없다. 하지만, 두 개 이상의 참조변수가 같은 주소값을 갖는 것 (한 객체를 참조하는 것)은 가능하다.

#### [예제9-1] EqualsEx1.java

```
class Value {
    int value;

    Value(int value) {
        this.value = value;
    }
}

class EqualsEx1
{
    public static void main(String[] args)
    {
        Value v1 = new Value(10);
        Value v2 = new Value(10);
        if (v1.equals(v2)) {
            System.out.println("v1과 v2는 같습니다.");
        } else {
            System.out.println("v1과 v2는 다릅니다.");
        }

        v2 = v1;

        if (v1.equals(v2)) {
            System.out.println("v1과 v2는 같습니다.");
        } else {
            System.out.println("v1과 v2는 다릅니다.");
        }
    }
}
```

```

    }
}
}

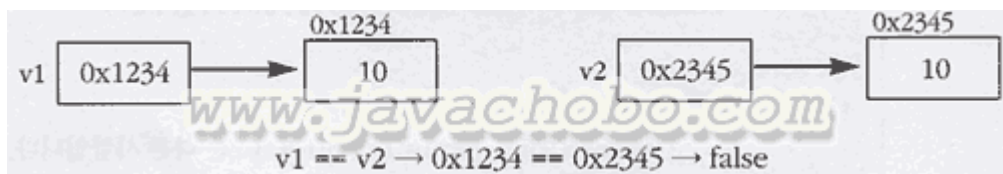
```

#### [실행결과]

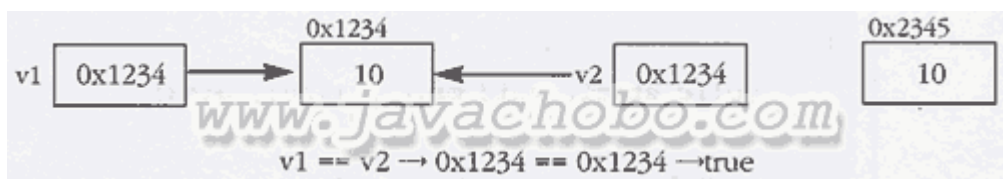
v1과 v2는 다릅니다.

v1과 v2는 같습니다.

value라는 멤버변수를 갖는 Value클래스를 정의하고, 두 개의 Value클래스의 인스턴스 생성한 다음 equals메서드를 이용해서 두 인스턴스를 비교하도록 했다. equals메서드는 주소값으로 비교를 하기 때문에, 두 Value인스턴스의 멤버변수 value의 값이 10으로 서로 같을지라도 equals메서드로 비교한 결과는 false일 수 밖에 없는 것이다.



하지만, v2 = v1;을 수행한 후에는 참조변수 v2는 v1이 참조하고 있는 인스턴스의 주소값이 저장되므로 v2도 v1과 같은 주소값이 저장된다. 그래서 이번에는 v1.equals(v2)의 결과가 true가 되는 것이다.



Object클래스로부터 상속받은 equals메서드는 결국 두 개의 참조변수가 같은 객체를 참조하고 있는지, 즉 두 참조변수에 저장된 값(주소값)이 같은지를 판단하는 기능 밖에 할 수 없다는 것을 알 수 있다. equals메서드 사용해서 Value인스턴스가 가지고 있는 value값을 비교하도록 할 수는 없을까? equals메서드를 Value클래스에서 오버라이딩하여 내용을 변경하면 된다.

#### [예제9-2] EqualsEx2.java

```

class Value {

```

```

int value;
public boolean equals(Object obj) {
    if ( obj instanceof Value) {
        return value == ((Value)obj).value;
    } else {
        return false;
    }
}
Value(int value) {
    this.value = value;
}
}

class EqualsEx2
{
    public static void main(String[] args)
    {
        Value v1 = new Value(10);
        Value v2 = new Value(10);
        if (v1.equals(v2)) {
            System.out.println("v1과 v2는 같습니다.");
        } else {
            System.out.println("v1과 v2는 다릅니다.");
        }

        v2 = v1;

        if (v1.equals(v2)) {
            System.out.println("v1과 v2는 같습니다.");
        } else {
            System.out.println("v1과 v2는 다릅니다.");
        }
    }
}

```

**[실행결과]**

v1과 v2는 같습니다.

v1과 v2는 같습니다.

equals메서드가 Value인스턴스가 갖고 있는 멤버변수 value의 값을 비교하도록 하기 위해 equals메서드를 다음과 같이 오버라이딩했다.

```
public boolean equals(Object obj) {  
    if (obj != null && obj instanceof Value) {  
        return value == ((Value)obj).value;  
    } else {  
        return false;  
    }  
}
```

이렇게 함으로써 Value클래스의 equals메서드는 기존의 Object클래스에 정의된 equals메서드와는 다르게 주소값이 아닌 두 인스턴스의 value값을 비교하고 그 결과를 돌려주도록 되어 있다. equals메서드의 매개변수 타입이 Object타입이기 때문에 Value타입으로 캐스팅(casting) 해주어야 멤버변수인 value의 값을 참조할 수 있다.

우리가 자주 사용하는 String클래스 역시 Object클래스의 equals메서드를 그대로 사용하는 것이 아니라 이처럼 오버라이딩을 통해서 String인스턴스가 갖는 문자열 값을 비교하도록 되어 있다.

그렇기 때문에 같은 내용의 문자열을 갖는 두 String인스턴스에 equals메서드를 사용하면 항상 true값을 얻는 것이다.

## 1.2 hashCode메서드

이 메서드는 각 인스턴스의 같고다름을 비교하기 위한 인스턴스 구별 값인 해쉬코드(hashcode)를 반환한다. 해쉬코드는 인스턴스의 주소와 관련된 정수값(int)으로 서로 다른 인스턴스는 서로 다른 해쉬코드값을 가질 것을 보장한다. 그래서 서로 다른 두 인스턴스가 같은 해쉬코드값을 갖는 경우는 없다.

하지만, hashCode메서드를 오버라이딩해서 원래의 해쉬코드와는 다른 값을 반환하도록 할 수는 있다.

생성된 인스턴스의 해쉬코드는 프로그램이 실행될 때마다 할당받는 메모리주소가 다를 것이므로 매번 실행할 때마다 다른 값을 가지게 되지만, 적어도 프로그램이 시작된 후부터 종료될 때까지는 같은 값을 유지한다.

Hashtable클래스나 Vector클래스와 같은 인스턴스를 Object타입의 배열에 담아서 작업하는 클래스에서는 인스턴스의 같고 다름을 판단하는데 해쉬코드값을 이용한다.

서로 다른 종류의 인스턴스를 구분할 공통기준이 해쉬코드(주소값)외에는 없기 때문이기도 하지만, 인스턴스의 멤버변수의 값들을 비교하는 것보다는 4byte의 해쉬코드값을 비교하는 것이 더 빠르기 때문이다.

### 1.3 toString메서드

toString()은 인스턴스에 대한 정보를 문자열(String)로 제공할 목적으로 정의한 것이다. Object클래스에 정의된 toString()은 아래와 같다.

```
public String toString() {  
    return getClass().getName() + "@" + Integer.toHexString(hashCode());  
}
```

클래스를 작성할 때 toString()을 오버라이딩하지 않는다면, 위와 같은 내용이 그대로 사용될 것이다. 즉, toString()을 호출하면 클래스이름에 16진수의 해쉬코드를 얻게 될 것이다. 해쉬코드는 인스턴스의 주소와 관련된 값으로, 서로 다른 인스턴스는 서로 다른 해쉬코드값을 가질 것을 보장한다.

[참고]getClass메서드와 hashCode메서드 역시 Object클래스의 인스턴스메서드이므로 인스턴스 생성없이 바로 호출할 수 있었다.

#### [예제9-3] CardToString.java

```

class Card {
    String kind;
    int number;
    Card() {
        this("SPADE", 1);
    }
    Card(String kind, int number) {
        this.kind = kind;
        this.number = number;
    }
}

class CardToString
{
    public static void main(String[] args)
    {
        Card c1 = new Card();
        Card c2 = new Card();

        System.out.println(c1.toString());
        System.out.println(c2.toString());
    }
}

```

#### [실행결과]

Card@47e553

Card@20c10f

Card인스턴스 두 개를 생성한 다음, 각 인스턴스에 toString()을 호출한 결과를 출력했다. Card클래스에서 Object클래스로부터 상속받은 toString()을 오버라이딩 하지 않았기 때문에, Card인스턴스에 toString()을 호출하면, Object클래스의 toString()이 호출된다.

그래서 위의 결과에 클래스이름과 해쉬코드가 출력되었다. 서로 다른 인스턴스에 대해서 toString()을 호출하였으므로 클래스의 이름은 같아도 해쉬코드값이 다르다는 것을 확인할 수



있다.

#### [예제9-4] ToStringTest.java

```
class ToStringTest {  
    public static void main(String args[]) {  
        String str = new String("KOREA");  
        java.util.Date today = new java.util.Date();  
  
        System.out.println(str);  
        System.out.println(str.toString());  
        System.out.println(today);  
        System.out.println(today.toString());  
    }  
}
```

#### [실행결과]

```
KOREA  
KOREA  
Fri May 17 23:26:13 KST 2002  
Fri May 17 23:26:13 KST 2002
```

위의 결과에서 알수 있듯이 String클래스와 Date클래스의 toString()을 호출하였더니 클래스 이름과 해쉬코드 대신 다른 결과가 출력되었다.

String클래스의 toString()은 String인스턴스가 갖고 있는 문자열을 반환하도록 오버라이딩되어 있고, Date클래스의 경우, Date인스턴스가 갖고 있는 날짜와 시간을 문자열로 하여 반환하도록 오버라이딩되어 있다.

이처럼 toString()은 보통 인스턴스나 클래스에 대한 정보 또는 인스턴스 변수들의 값을 문자열로 변환하여 반환하도록 오버라이딩된다.

이제 Card클래스에서도 toString()을 오버라이딩해서 보다 쓸모있는 정보를 제공할 수 있도록 바꿔 보자.

#### [예제9-5] CardToString2.java

```

class Card {
    String kind;
    int number;
    Card() {
        this("SPADE", 1);
    }
    Card(String kind, int number) {
        this.kind = kind;
        this.number = number;
    }
    public String toString() {
        // Card인스턴스의 kind와 number를 문자열로 반환한다.
        return "kind : " + kind + ", number : " + number;
    }
}

class CardToString2
{
    public static void main(String[] args)
    {
        Card c = new Card("HEART", 10);
        System.out.println(c.toString());
    }
}

```

#### [실행결과]

```
kind : HEART, number : 10
```

Card클래스의 조상인 Object클래스의 toString()을 오버라이딩해서, Card인스턴스의 toString()을 호출하면, 인스턴스가 갖고 있는 인스턴스변수 kind와 number의 값을 문자열로 변환하여 반환하도록 했다.

Object클래스에 정의된 toString()의 접근제어자가 public이므로 Card클래스에서 오버라이딩할 때도 public으로 했다는 것을 눈 여겨 보도록 하자.

[참고] 조상클래스에 정의된 메서드를 자손클래스에서 오버라이딩할 때는 조상클래스에서 정의된 접근범위보다 같거나 더 넓어야 한다. Object클래스에서 toString()이 public으로 선언되어 있기 때문에, 이것을 오버라이딩하는 Card클래스에서는 toString()의 접근제어자를 public으로 할 수 밖에 없다.

#### 1.4 clone메서드

이 메서드는 자신을 복제하여 새로운 인스턴스를 생성하는 일을 한다. 어떤 인스턴스에 대해 작업을 할 때, 원래의 인스턴스는 보존하고 clone메서드를 이용해서 새로운 인스턴스를 생성하여 작업을 하면 작업이전의 값이 보존되므로 작업에 실패해서 원래의 상태로 되돌리거나 변경되기전의 값을 참고하는데 도움이 될 것이다.

Object클래스에 정의된 clone메서드는 단순히 멤버변수의 값을 복사하기 때문에 배열이나 인스턴스가 멤버로 정의되어 있는 클래스의 인스턴스는 완전한 복제가 이루어지지 않는다.

예를 들어 배열의 경우 복제된 인스턴스도 같은 배열의 주소를 갖기 때문에 복제된 인스턴스의 작업이 원래의 인스턴스에 영향을 미치게 된다.

이런 경우 clone메서드를 오버라이딩해서 새로운 배열을 생성하고 배열의 내용을 복사하도록 해야한다.

##### [예제9-6] CloneTest.java

```
// Cloneable인터페이스를 구현한 클래스에서만 clone메서드를 호출할 수 있다.
```

```
// 이 인터페이스를 구현하지 않고 clone메서드를 호출하려면
```

```
CloneNotSupportedException이 발생한다.
```

```
class Point implements Cloneable {
```

```
    int x;
```

```
    int y;
```

```
    Point(int x, int y) {
```

```
        this.x = x;
```

```
        this.y = y;
```

```
    }
```

```
    public String toString() {
```

```

        return "x="+x +", y="+y;
    }

    public Point copy() {
        Object obj=null;
        try {
            // clone메서드에서는 CloneNotSupportedException이 선언되어 있으므로
            // 이 메서드를 호출할 때는 try-catch문을 사용해야한다.
            obj = clone();
        } catch(CloneNotSupportedException e) {    }
        return (Point)obj;
    }
}

class CloneTest {
    public static void main(String[] args){
        Point original = new Point(3, 5);
        Point copy = original.copy();
        System.out.println(original);
        System.out.println(copy);
    }
}

```

#### [실행결과]

x=3, y=5

x=3, y=5

clone메서드는 접근제어자가 protected이므로 접근제어자가 public인 새로운 copy메서드를 선언하고 그 내부에서 clone메서드를 통해 인스턴스 복제를 하도록 처리했다.

Cloneable 인터페이스를 구현한 클래스의 인스턴스만이 clone메서드를 통한 복제가 가능하다. 인스턴스 복제는 데이터를 복사하는 것이기 때문에 데이터를 보호하기 위해서, 클래스 작성자가 복제를 허용하는 경우, Cloneable인터페이스를 구현한 경우,에만 복제가 가능하도록 하기 위해서이다.

# [Java의 정석]제9장 java.lang패키지 - 2.String클래스

자바의정석

2012/12/22 17:14

<http://blog.naver.com/gphic/50157740383>

## 2. String클래스

기존의 다른 언어에서는 문자열을 char형의 배열로 다루었으나 자바에서는 문자열을 위한 클래스를 제공한다. 그 것이 바로 String클래스인데, String클래스는 문자열을 저장하고 이를 다루는데 필요한 메서드를 제공한다.

지금까지는 String클래스의 기본적인 몇 가지 기능만 사용해 왔지만, String클래스에는 문자열을 다루는데 유용한 메서드들이 많이 있다. 이제 String클래스에 대해서 자세히 알아보도록 하자.

### 2.1 String클래스의 특징

String 클래스에는 문자열을 저장하기 위해서 문자형 배열 변수(char[]) value를 인스턴스 변수로 정의해놓고 있다. 인스턴스 생성 시 생성자의 매개변수로 입력받는 문자열은 이 인스턴스변수(value)에 문자형 배열(char[])로 저장되는 것이다.

```
public final class String implements java.io.Serializable, Comparable {  
    /** The value is used for character storage. */  
    private char[] value;  
    ...  
}
```

한번 생성된 String인스턴스가 갖고 있는 문자열은 읽어 올 수만 있고, 변경할 수는 없다.

예를 들어 "a" + "b"와 같이 '+'연산자를 이용해서 문자열을 결합하는 경우 인스턴스내의 문자열이 바뀌는 것이 아니라 새로운 문자열("ab")이 담긴 String인스턴스가 생성되는 것이다.

```
String a = "a";  
String b = "b";  
String ab = a + b;
```

이처럼 덧셈연산자(+)를 사용해서 문자열을 결합하는 것은 매 연산 시마다 새로운 문자열을 가진 String 인스턴스가 생성되어 메모리공간을 차지하게 되므로 가능한 한 결합횟수를 줄이는 것이 좋다. 문자열간의 결합이나 추출 등 문자열을 다루는 작업이 많이 필요한 경우에는 String 클래스 대신 StringBuffer 클래스를 사용하는 것을 고려해보도록 한다. String 인스턴스와는 달리 StringBuffer 인스턴스에 저장된 문자열은 변경이 가능하므로 하나의 StringBuffer 인스턴스만으로도 문자열을 다루는 것이 가능하다.

#### [예제9-9] StringEx1.java

```
class StringEx1  
{  
    public static void main(String[] args)  
    {  
        String str1 = "abc";  
        String str2 = "abc";  
  
        System.out.println(" String str1 = ₩"abc₩";");  
        System.out.println(" String str2 = ₩"abc₩";");  
  
        if(str1 == str2) {  
            System.out.println(" str1 == str2 ? true");  
        } else {  
            System.out.println(" str1 == str2 ? false");  
        }  
  
        if(str1.equals(str2)) {  
            System.out.println(" str1.equals(str2) ? true");  
        } else {  
            System.out.println(" str1.equals(str2) ? false");  
        }  
    }  
}
```

```

    }

    System.out.println();

    String str3 = new String("₩"abc₩");
    String str4 = new String("₩"abc₩");

    System.out.println(" String str3 = new String(₩"abc₩");");
    System.out.println(" String str4 = new String(₩"abc₩");");

    if(str3 == str4) {
        System.out.println(" str3 == str4 ? true");
    } else {
        System.out.println(" str3 == str4 ? false");
    }

    if(str3.equals(str4)) {
        System.out.println(" str3.equals(str4) ? true");
    } else {
        System.out.println(" str3.equals(str4) ? false");
    }
}
}
}

```

#### [실행결과]

```

String str1 = "abc";
String str2 = "abc";
str1 == str2 ? true
str1.equals(str2) ? true

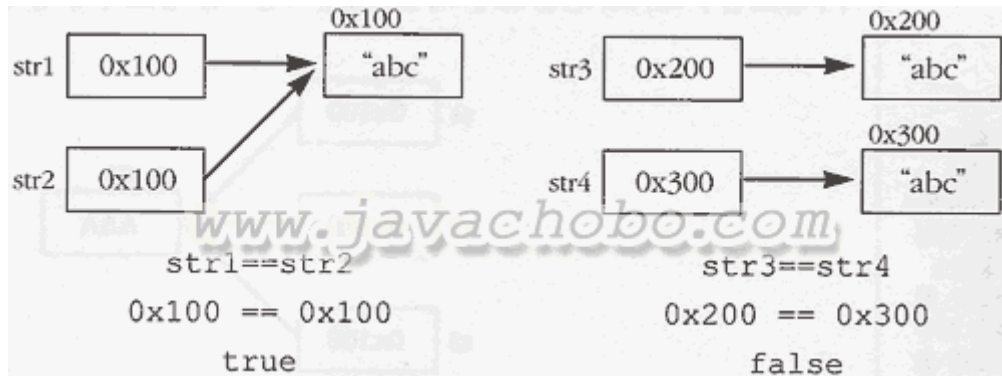
String str3 = new String("abc");
String str4 = new String("abc");
str3 == str4 ? false
str3.equals(str4) ? true

```

문자열을 만들 때는 두 가지 방법, 문자열 리터럴을 지정하는 방법과 String클래스의 생성자를 사용해서 만드는 방법이 있다. 이 예제는 문자열을 리터럴로 생성하는 것과 String클래스의 생성자를 사

용해서 생성하는 것과의 차이를 보여 주기 위한 것이다.

`equals(String s)`를 사용했을 때는 두 문자열의 내용("abc")을 비교하기 때문에 두 경우 모두 `true`를 결과로 얻는다. 하지만, 각 `String`인스턴스의 주소값을 등가비교연산자(`==`)로 비교했을 때는 결과가 다르다. 리터럴로 문자열을 생성했을 경우, 같은 내용의 문자열들은 모두 하나의 `String`인스턴스를 참조하도록 되어 있다. 어차피 `String`인스턴스가 저장하고 있는 문자열은 변경할 수 없기 때문에 아무런 문제가 없다.



그러나, `String`클래스의 생성자를 이용한 `String`인스턴스의 경우에는 `new`연산자에 의해서 메모리할당이 이루어지기 때문에 항상 새로운 `String`인스턴스가 생성된다.

아래의 왼쪽 그림은 리터럴로 문자열을 생성했을 때의 상황이고, 오른쪽 그림은 생성자를 이용해서 문자열을 생성했을 때의 상황을 그림으로 나타낸 것이다.

#### [예제9-8] StringEx2.java

```
class StringEx2
{
    public static void main(String args[]) {
        String s1 = "AAA";
        String s2 = "AAA";
        String s3 = "AAA";
        String s4 = "BBB";
    }
}
```

위의 예제를 컴파일 하면 `StringEx2.class`파일이 생성된다. 이 파일의 내용을 16진 코드에디터로 보면 아래의 그림과 같다.



```

00000000h: CA FE BA BE 00 03 00 2D 00 13 0A 00 05 00 0E 08 ; 靠...-.....
00000010h: 00 0F 08 00 10 07 00 11 07 00 12 01 00 06 3C 69 ; .....<i
00000020h: 6E 69 74 3E 01 00 03 28 29 56 01 00 04 43 6F 64 ; nit>...{}V...Cod
00000030h: 65 01 00 0F 4C 69 6E 65 4E 75 6D 62 65 72 54 61 ; e...LineNumberTa
00000040h: 62 6C 65 01 00 04 6D 61 69 6E 01 00 16 28 5B 4C ; ble...main...([L
00000050h: 6A 61 76 61 2F 6C 61 6E 67 2F 53 74 72 69 6E 67 ; java/lang/String
00000060h: 3B 29 56 01 00 0A 53 6F 75 72 63 65 46 69 6C 65 ; ;)V...SourceFile
00000070h: 01 00 14 53 74 72 69 6E 67 45 78 61 6D 70 6C 65 ; ...StringExample
00000080h: 31 35 2E 6A 61 76 61 0C 00 06 00 07 01 00 03 41 ; 15.java.....A
00000090h: 41 41 01 00 03 42 42 42 01 00 0F 53 74 72 69 6E ; AA...BBB...Strin
000000a0h: 67 45 78 61 6D 70 6C 65 31 35 01 00 10 6A 61 76 ; gExample15...jav
000000b0h: 61 2F 6C 61 6E 67 2F 4F 62 6A 65 63 74 00 20 00 ; a/lang/Object. .
000000c0h: 04 00 05 00 00 00 00 00 02 00 00 00 06 00 07 00 ; .....
000000d0h: 01 00 08 00 00 00 1D 00 01 00 01 00 00 00 05 2A ; .....*
000000e0h: F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; 0.0 .....
000000f0h: c 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; .....
00000100h: 00 3b 00 01 00 00 00 00 0e 12 02 4c 12 02 4d ; .6.....L..M
00000110h: 12 02 4E 12 03 3A 04 B1 00 00 00 01 00 09 00 00 ; ..N... ..
00000120h: 00 16 00 05 00 00 00 04 00 03 00 05 00 06 00 06 ; .....
00000130h: 00 09 00 07 00 0D 00 08 00 01 00 0C 00 00 00 02 ; .....

```

[그림9-1]StringEx2.class파일의 내용

[참고] 일반 문서 편집기로도 StringEx2.class파일의 내용을 볼 수 있다.

우측 부분을 보면 알아볼 수 있는 글자들이 눈에 띈 것이다. 그 중에서도 "AAA"와 "BBB"가 있는 것을 발견할 수 있을 것이다. 이와 같이 String리터럴들은 컴파일 시에 클래스파일에 저장된다. 위의 예제를 실행하게 되면 "AAA"라는 문자열을 담고 있는 String인스턴스가 하나 생성된 후, 참조 변수 s1, s2, s3는 모두 이 String인스턴스를 참조하게 된다.

모든 클래스 파일(\*.class)에는 constant pool이라는 상수값 목록이 있어서, 여기에 클래스 내에서 사용되는 문자열 리터럴과 상수값들이 저장되어 있다.

#### [예제9-9] StringEx3.java

```

class StringEx3
{
    public static void main(String[] args)
    {
        String s1 = "AAA";
        String s2 = new String("AAA");

        if (s1==s2) {

```

```

        System.out.println("s1==s2 ? true");
    } else {
        System.out.println("s1==s2 ? false");
    }

    s2 = s2.intern();
    System.out.println("s2에 intern()을 호출한 후");

    if (s1==s2) {
        System.out.println("s1==s2 ? true");
    } else {
        System.out.println("s1==s2 ? false");
    }

}
}

```

#### [실행결과]

```

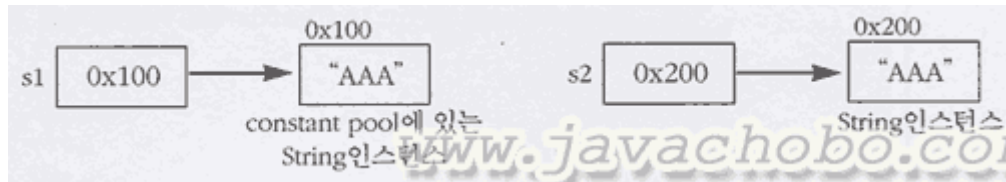
s1==s2 ? false
s2에 intern()을 호출한 후
s1==s2 ? true

```

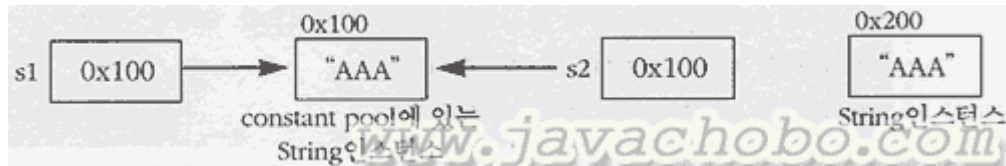
String 클래스의 intern메서드는 String인스턴스의 문자열을 constant pool에 등록하는 일을 한다. 등록하고자 하는 문자열이 constant pool에 이미 존재하는 경우에는 그 문자열의 주소값을 반환한다.(그리 중요한 메서드는 아니지만, 많은 책들이 어렵게 설명하고 있기 때문에 참고로 추가하였다.)

위의 예제에서는 참조변수 s2가 가리키는 문자열이 이미 constant pool에 등록되어 있기 때문에, s2.intern()의 결과로 등록되어 있는 String인스턴스(문자열을 담고 있는)의 주소값을 얻게 된다. 그래서 참조변수 s1과 s2는 같은 String인스턴스를 가리키게 되어 s1==s2의 결과가 true가 되는 것이다.

1. String s1 = "AAA";  
String s2 = new String("AAA");



2. `s2 = s2.intern();`



#### [예제9-10] StringEx4.java

```
class StringEx4
{
    public static void main(String[] args)
    {
        String[] words = { new String("aaa"), new String("bbb"), new String("ccc")
};

        for(int i=0; i < words.length; i++) {
            if(words[i].equals("ccc")) {
                System.out.println("words에서 equals메서드로 찾았습니다.");
            }
            if(words[i] == "ccc") {
                System.out.println("words에서 ==연산자로 찾았습니다.");
            }
        }

        for(int i=0; i < words.length; i++) {
            words[i] = words[i].intern();
        }

        System.out.println("<< String배열 words의 문자열에 intern메서드를 수행한 후
```

```

>>");
    for(int i=0; i < words.length; i++) {
        if(words[i].equals("ccc")) {
            System.out.println("words에서 equals메서드로 찾았습니다.");
        }
        if(words[i] == "ccc") {
            System.out.println("words에서 ==연산자로 찾았습니다.");
        }
    }
} // end of main
} // end of class

```

#### [실행결과]

```

words에서 equals메서드로 찾았습니다.
<< String배열 words의 문자열에 intern메서드를 수행한 후 >>
words에서 equals메서드로 찾았습니다.
words에서 ==연산자로 찾았습니다.

```

intern메서드가 수행된 String인스턴스는 equals메서드 뿐만 아니라 등가비교연산자(==)를 가지고  
도 문자열을 비교할 수 있다는 것을 보여 주는 예제이다.

일반적으로 문자열들을 비교하기 위해서 equals메서드를 사용하지만, equals메서드로 문자열의 내  
용을 비교하는 것보다는 등가비교연산자(==)를 이용해서 주소(4 byte)를 비교하는 것이 더 빠르다.

그래서 비교해야할 문자열의 개수가 많은 경우에는 보다 빠른 문자열 검색을 위해서 intern메서드와  
등가비교연산자(==)를 사용하기도 한다.

#### [예제9-11] StringEx5.java

```

class StringEx5
{
    static String s;
    static String s2 = "";
    public static void main(String[] args)
    {
        for(int i=1; i < 10; i++) {

```

```
        s += i;        // s = s + i;

        s2 += i;
    }

    System.out.println(s);
    System.out.println(s2);
}
}
```

**[실행결과]**

```
null123456789
123456789
```

이 예제는 1부터 9까지의 숫자를 문자열로 만들어서 덧붙이는 일을 하는데, 결과의 첫 줄을 보면 맨 앞에 null이라고 출력되어있는 것을 알 수 있다. 참조변수 s는 멤버변수이기 때문에 따로 초기화 해주지 않으면, 자신의 타입에 해당하는 기본값인 null로 초기화 된다. 덧셈연산자(+)는 값이 null인 참조변수를 "null"로 변환한 후 연산을 한다. 그렇기 때문에 "null123456789"와 같은 결과가 나온 것이다.

s2의 경우 빈 문자열("", empty string)로 초기화 했기 때문에 "123456789"와 같은 결과를 얻었다. 이처럼, String형 참조변수를 초기화 하지 않으면 문자열간의 결합에 있어서 문제가 될 수 있으므로, String s = "";와 같이 변수의 선언과 함께 빈 문자열로 초기화 해주는 것이 좋다.

**[참고]** 프로그램을 테스트할 때 에러를 쉽게 발견하기 위한 목적으로 일부러 빈 문자열 대신 null로 초기화 해주기도 한다.

## 2.2 빈 문자열(empty string)

크기가 0인 배열이 존재할 수 있을까? 답은 '존재할 수 있다.'이다. 빈 문자열이 바로 크기가 0인 char형 배열이다. 전에도 학습했던 것과 같이 문자열은 내부적으로 char배열로 저장된다.

String s = ""과 같이 했을 때, s가 크기가 0인 char형 배열이라고 해서, char c = ''와 같은 것은 아니다. char형의 변수에는 반드시 하나의 문자를 지정해야한다.

초기화하지 않은 char형 멤버변수 c의 경우, '\u0000'로 자동초기화가 이루어진다. '\u0000'은 유니코드의 첫 번째 문자로써 아무런 문자도 지정되지 않은 빈 문자이다. 만일 아무 내용도 없는 빈 문자를 char형 변수에 저장하고자 한다면, char c = '\u0000'과 같이 해야지 char c=""과 같은 표현은 허용되지 않는다.

이처럼 char형 변수의 경우, 변수에 문자를 반드시 지정해 주어야 하지만, char형 배열은 char[] cArray = new char[0];과 같은 표현이 가능하다. 크기가 0이기 때문에 아무런 문자도 저장할 수 없는 배열이라 무의미하게 느껴지겠지만 어쨌든 이러한 표현이 가능하다.

String s=""과 같은 문장이 있을 때 참조변수 s가 참조하고 있는 String인스턴스의 내부에는 new char[0]과 같이 크기가 0인 char형 배열을 내부적으로 갖고 있는 것이다.

[참고]char형 배열뿐만 아니라 모든 배열이 크기가 0인 배열 생성이 가능하다.

#### [예제9-12] StringEx6.java

```
class StringEx6
{
    static char[] c = new char[0];    // 크기가 0인 char배열 생성
    // 생성자 String(char[] c) 사용
    static String s = new String(c);    // static String s = new String("");와 같다.
    public static void main(String[] args)
    {
        System.out.println(s);
        System.out.println(c.length);
    }
}
```

#### [실행결과]

0

크기가 0인 배열을 생성해서 char형 배열 참조변수 c를 초기화 해주었다. 크기가 0이긴 해도 배열이 생성되며, 생성된 배열의 주소값이 참조변수 c에 저장된다.

[참고]위 예제결과의 첫 줄에는 빈 문자열("")이 출력된 것이다.

## 2.2 String클래스의 생성자와 메서드

메서드 / 설명	예 제	결 과
<b>String(String s)</b> 주어진 문자열(s)을 갖는 String인스턴스를 생성한다.	<pre>String s = new String("Hello");</pre>	s = "Hello"
<b>String(char[] value)</b> 주어진 문자열(value)을 갖는 String인스턴스를 생성한다.	<pre>char[] c = {'H', 'e', 'l', 'l', 'o'}; String s = new String(c);</pre>	s = "Hello"
<b>String(StringBuffer buf)</b> StringBuffer인스턴스가 갖고 있는 문자열과 같은 값을 갖는 String인스턴스를 생성한다.	<pre>StringBuffer sb = new StringBuffer("Hello"); String s = new String(sb);</pre>	s = "Hello"
<b>char charAt(int index)</b> index의 위치에 있는 문자를 알려준다.(index는 0부터 시작)	<pre>String s = "Hello"; String n = "0123456"; char c = s.charAt(1); char c2 = n.charAt(1);</pre>	c = 'e' c2 = '1'

[www.javachobo.com](http://www.javachobo.com)



메서드 / 설명	예 제	결 과
<b>String concat(String str)</b> 문자열(str)을 뒤에 덧붙인다.	String s = "Hello"; String s2 = s.concat(" World");	s2 = "Hello World"
<b>boolean endsWith(String suffix)</b> 지정된 문자열(suffix)로 끝나는지 검사한다.	String file = "Hello.txt"; boolean b = file.endsWith(".txt");	b = true
<b>boolean equals(Object obj)</b> 매개변수로 받은 문자열(obj)과 String 인스턴스의 문자열을 비교한다. obj가 String이 아니거나, 문자열이 다르면 false를 반환한다. (대소문자를 구분한다)	String s = "Hello"; boolean b = s.equals("Hello"); boolean b2 = s.equals("hello");	b = true b2 = false
<b>boolean equalsIgnoreCase(String str)</b> 문자열과 String 인스턴스의 문자열을 대소문자 구분없이 비교한다.	String s = "Hello"; boolean b = s.equalsIgnoreCase("HELLO"); boolean b2 = s.equalsIgnoreCase("heLLo");	b = true b2 = true
<b>int indexOf(int ch)</b> 주어진 문자(ch) 또는 문자코드값의 문자가 문자열에 존재하는지 확인하여 위치(index)를 알려준다. 없으면 -1값을 반환한다. 그리고, index는 0부터 시작한다.	String s = "Hello"; int idx1 = s.indexOf('o'); int idx2 = s.indexOf(111); int idx3 = s.indexOf('k');	idx1 = 4 idx2 = 4 idx3 = -1
<b>int indexOf(String str)</b> 주어진 문자열이 존재하는지 확인하여 위치(index)를 알려준다. 없으면 -1값을 반환한다. 그리고, index는 0부터 시작한다.	String s = "ABCDEFGH"; int idx1 = s.indexOf("CD");	idx1 = 2
<b>String intern()</b> 문자열을 constant pool에 등록한다. 이미 constant pool에 같은 내용의 문자열이 있을 경우 그 문자열(String 인스턴스)의 주소값을 반환한다.	String s = new String("abc"); String s2 = new String("abc"); boolean b = (s==s2); boolean b2 = s.equals(s2); boolean b3 = (s.intern()==s2.intern());	b = false b2 = true b3 = true
<b>int lastIndexOf(int ch)</b> 지정된 문자 또는 문자코드값 문자열의 끝에서부터 찾아서 위치(index)를 알려준다. 없으면 -1을 반환한다.	String s = "java.lang.Object"; int idx1 = s.lastIndexOf('.'); int idx2 = s.indexOf('.');	idx1 = 9 idx2 = 4
<b>int lastIndexOf(String str)</b> 지정된 문자열을 인스턴스의 문자열 끝에서부터 찾아서 위치(index)를 알려준다. 없으면 -1을 반환한다.	String s = "java.java.java"; int idx1 = s.lastIndexOf("java"); int idx2 = s.indexOf("java");	idx1 = 10 idx2 = 0
<b>int length()</b> 문자열의 길이를 알려준다.	String s = "Hello"; int length = s.length();	length = 5
<b>String replace(char old, char nw)</b> 문자열중의 문자(old)를 새로운 문자(nw)로 바꾼 문자열을 반환한다.	String s = "Hello"; String s1 = s.replace('H', 'C');	s1 = "Cello"

www.javachobo.com

www.javachobo.com

메서드 / 설명	예 제	결 과
<b>String replaceAll( String regex, String replacement)</b> 문자열중에서 지정된 문자열(regex)과 일 치하 는 것 을 새 로운 문자열 (replacement)로 모두 변경한다.	String ab = "AABBAABB"; String r = ab.replaceAll("BB", "bb");	r = "AAbbAAbb"
<b>String replaceFirst( String regex, String replacement)</b> 문자열중에서 지정된 문자열(regex)과 일 치하 는 것 중, 첫째 것만 새로운 문자열 (replacement)로 변경한다.	String ab = "AABBAABB"; String r = ab.replaceFirst("BB", "bb");	r = "AAbbAABB"
<b>String[] split(String regex)</b> 문자열을 지정된 분리자(regex)로 나누 어 문자열배열에 담아 반환한다.	String animals = "dog,cat,bear"; String[] arr = animals.split(",");	arr[0] = "dog" arr[1] = "cat" arr[2] = "bear"
<b>String[] split(String regex, int limit)</b> 문자열을 지정된 분리자(regex)로 나누 되 지정된 개수만큼만 나누어 문자열 배열로 반환한다.	String animals = "dog,cat,bear"; String[] arr = animals.split(",", 2);	arr[0] = "dog" arr[1] = "cat,bear"
<b>boolean startsWith( String prefix)</b> 주어진 문자열(prefix)로 시작하는지 검 사한다.	String s = "java.lang.Object"; boolean b = s.startsWith("java"); boolean b2 = s.startsWith("lang");	b = true b2 = false
<b>String substring(int begin)</b> <b>String substring(int begin, int end)</b> 주어진 시작위치(begin)부터 끝위치 (end) 범위에 포함된 문자열을 얻는다. 이때, 시작위치의 문자는 범위에 포함 되지만, 끝위치의 문자는 포함되지 않 는다.(begin ≤ x < end)	String s = "java.lang.Object"; String c = s.substring(10); String p = s.substring(5, 9);	c = "Object" p = "lang"
<b>String toLowerCase()</b> String인스턴스에 저장되어있는 문자 열을 소문자로 변환한 결과를 반환한 다.	String s = "Hello"; String s1 = s.toLowerCase();	s1 = "hello"
<b>String toString()</b> String인스턴스에 저장되어있는 문자 열을 반환한다.	String s = "Hello"; String s1 = s.toString();	s1 = "Hello"
<b>String toUpperCase()</b> String인스턴스에 저장되어있는 문자 열을 대문자로 변환한 결과를 반환한 다.	String s = "Hello"; String s1 = s.toUpperCase();	s1 = "HELLO"

www.javachobo.com

www.javachobo.com

메서드 / 설명	예 제	결 과
<b>String trim()</b> 문자열의 앞과 뒤에 있는 공백을 없앤 결과를 반환한다. 이때 문자열 중간에 있는 공백은 제거되지 않는다.	<pre>String s = "  Hello World  "; String s1 = s.trim();</pre>	<pre>s1 = "Hello World"</pre>
<b>static String valueOf( boolean b)</b> <b>static String valueOf( char c)</b> <b>static String valueOf( int i)</b> <b>static String valueOf( long l)</b> <b>static String valueOf( float f)</b> <b>static String valueOf( double d)</b> <b>static String valueOf( Object o)</b>	<pre>String b = String.valueOf(true); String c = String.valueOf('a'); String i = String.valueOf(100); String l = String.valueOf(100L); String f = String.valueOf(10.0f); String d = String.valueOf(10.0d); java.util.Date d =     new java.util.Date(); String date = String.valueOf(d);</pre>	<pre>b = "true" c = "a" i = "100" l = "100" f = "10.0" d = "10.0" date = "Sun May 19 09:44:52 KST 2002"</pre>
매개변수로 넘겨 받은 값을 문자열로 변환하여 반환한다. 참조변수에는 toString()을 호출한 결과를 반환한다.		

[참고] 문자 'o'의 코드는 10진수로 111이다.

[참고] replaceFirst, replaceAll, split메서드는 JSDK1.4에서 새로 추가된 것들이다.

#### [예제9-13] StringEx7.java

```
class StringEx7
{
    public static void main(String[] args)
    {
        int value = 100;
        String strValue = String.valueOf(value);    // int를 String으로 변환한다.

        int value2 = 100;
        String strValue2 = value2 + "";    // int를 String으로 변환하는 또 다른 방법

        System.out.println(strValue);
        System.out.println(strValue2);
    }
}
```

```

    }
}

```

**[실행결과]**

```

100
100

```

이 예제는 정수형(int)값을 String으로 변환하는 두 가지 방법을 보여 주고 있다. String클래스의 `valueOf`메서드는 매개변수로 기본형 변수와 객체를 지정할 수 있으며, 그 결과로 String을 얻을 수 있다.

변수의 값을 String으로 변환하는 또 다른 방법은 덧셈연산자(+)를 사용하는 것이다. 덧셈연산자는 두 개의 피연산자 중 어느 한 쪽이라도 String이면 연산결과는 String이 된다.

그래서 변수에 빈 문자열을 더하면, 그 결과로 String을 얻을 수 있다.

[참고] 참조변수에 String을 더하면, 참조변수가 가리키고 있는 인스턴스의 `toString()`을 호출하여 String을 얻은 다음 결합한다.

연산식	결 과
<code>String a = 7 + " ";</code>	<code>a = "7 "</code>
<code>String b = " " + 7;</code>	<code>b = " 7"</code>
<code>String c = 7 + "";</code>	<code>c = "7"</code>
<code>String d = "" + 7;</code>	<code>d = "7"</code>
<code>String e = "" + "";</code>	<code>e = ""</code>
<code>String f = 7 + 7 + "";</code>	<code>f = "14"</code>
<code>String g = "" + 7 + 7;</code>	<code>g = "77"</code>

[참고] 덧셈연산자(+)는 왼쪽에서 오른쪽으로 연산을 진행해 나가기 때문에 `7 + 7 + ""`에서는 `7 + 7`을 먼저 수행한 다음에 그 결과인 14와 ""를 더해서 "14"가 된다.

지금까지 기본형 값을 문자열로 바꾸는 방법에 대해서 알아보았으니, 이제는 문자열을 기본형 값으로 바꾸는 방법에 대해서 정리해 보자.



기본형 → 문자열	문자열 → 기본형
String valueOf(boolean b)	boolean Boolean.getBoolean(String s)
String valueOf(char c)	char Character.valueOf(String s)
String valueOf(int i)	byte Byte.parseByte(String s)
String valueOf(long l)	short Short.parseShort(String s)
String valueOf(float f)	int Integer.parseInt(String s)
String valueOf(double d)	long Long.parseLong(String s)
	float Float.parseFloat(String s)
	double Double.parseDouble(String s)

[참고] byte, short값을 문자열로 변경할 때는 String valueOf(int i)를 사용하면 된다.

위의 표에 있는 메서드만 알고 있으면, 문자열과 기본형 값의 변환에는 아무런 문제가 없을 것이다.  
이 변환은 프로그래밍에서 반드시 알고 있어야 하는 아주 중요한 내용이다.

#### [예제9-14] StringEx8.java

```
class StringEx8
{
    public static void main(String[] args)
    {
        String[] numbers = { "1", "2", "3", "4", "5" };
        String result1 = "";
        int result2 = 0;

        for(int i=0; i < numbers.length; i++) {
            result1 += numbers[i];
            result2 += Integer.parseInt(numbers[i]);
        }

        System.out.println("result1 : " + result1);
        System.out.println("result2 : " + result2);
    }
}
```

#### [실행결과]

result1 : 12345

result2 : 15

이 예제는 문자열을 정수형(int) 값으로 변환하는 예를 보여 준 것이다. 문자열에 공백 또는 문자가 포함되어 있는 경우 변환 시 예외(NumberFormatException)가 발생할 수 있으므로 주의해야 한다. 그러나 소수점을 의미하는 '.' 이나 float형 값을 뜻하는 f와 같은 자료형 접미사는 허용된다. 단, 자료형에 알맞은 변환을 하는 경우에만 허용된다.

만일 "1.0f"를 int형 변환 메서드인 Integer.parseInt(String s)을 사용해서 변환하려하면 예외가 발생된다. 이때는 Float.parseFloat(String s)를 사용해야 한다.

이처럼 문자열을 숫자로 변환하는 과정에서는 예외가 발생하기 쉽기 때문에 주의를 기울여야 한다.

#### [알아두면 좋아요.]

Integer클래스의 static int parseInt(String s, int radix) 를 사용하면 16진수 값으로 표현된 문자열도 변환할 수 있기 때문에 대소문자 구별없이 a, b, c, d, e, f도 사용할 수 있다.

int result = Integer.parseInt("a", 16);의 경우 result에는 정수값 10이 저장된다.(16진수 a는 10진수로는 10을 뜻한다.)

#### [예제9-15] StringEx9.java

```
class StringEx9
{
    public static void main(String[] args)
    {
        String fullName = "Hello.java";

        // fullName에서 '.'의 위치를 찾는다.
        int index = fullName.indexOf('.');

        // fullName의 첫번째 글자부터 '.'이 있는 곳까지 문자열을 추출한다.
        String fileName = fullName.substring(0, index);

        // '.'의 다음 문자 부터 시작해서 문자열의 끝까지 추출한다.
        // fullName.substring(index+1, fullName.length());의 결과와 같다.
```

```

String ext = fullName.substring(index+1);

System.out.println(fullName + "의 확장자를 제외한 이름은 " + fileName);
System.out.println(fullName + "의 확장자는 " + ext);
}
}

```

#### [실행결과]

Hello.java의 확장자를 제외한 이름은 Hello  
Hello.java의 확장자는 java

위 예제는 substring메서드를 이용하여 한 문자열에서 내용의 일부를 추출하는 예를 보인 것이다. substring(int start, int end)를 사용할 때 주의해야할 점은 매개변수로 사용되는 문자열에서 각 문자의 위치를 뜻하는 index가 0부터 시작한다는 것과 start부터 end의 범위 중 end위치에 있는 문자는 결과에 포함되지 않는다는 것이다.

[참고] end에서 start값을 빼면 substring에 의해 추출될 글자의 수가 된다.

index	0	1	2	3	4	5	6	7	8	9
char	H	e	l	l	o	.	j	a	v	a

www.javachobo.com

연 산 식	결 과
String str = "Hello.java";	index = 5
int index = str.indexOf('.');	index2 = 5
int index2 = str.lastIndexOf('.');	a = "Hello"
String a = str.substring(0, 5);	b = "java"
String b = str.substring(6, 10);	

[참고] substring의 철자에 주의하도록 한다. subString이 아니다.

#### [예제9-16] StringCount.java

```

public class StringCount
{
    private int count;

```

```

private String source = "";

public StringCount(String source) {
    this.source = source;
}

public int stringCount(String s) {
    return stringCount(s, 0);
}

public int stringCount(String s, int pos) {
    int index = 0;
    if (s == null || s.length() == 0)
        return 0;
    if ( (index = source.indexOf(s, pos)) != -1 ) {
        count++;
        stringCount(s, index + s.length());
    }
    return count;
}

public static void main(String[] args)
{
    String str = "aabbccAABBCCaa";
    System.out.println(str);
    StringCount sc = new StringCount(str);
    System.out.println("aa를 " + sc.stringCount("aa") + "개 찾았습니다.");
}
}

```

#### [실행결과]

```

aabbccAABBCCaa
aa를 2개 찾았습니다.

```

하나의 긴 문자열(source) 중에서 특정 문자열과 일치하는 문자열의 개수를 구하는 예제이다.



stringCount메서드는 재귀호출을 이용해서 반복적으로 일치하는 문자열의 개수를 구한다.  
이 예제를 응용해서 문자열을 치환하는 예제를 한번 작성해보도록 하자.

# [Java의 정석]제 10 장 내부 클래스(inner class)

자바의정석

2012/12/22 17:14

<http://blog.naver.com/gphic/50157740412>

## Chapter 10. 내부클래스(inner class)

### 1. 내부클래스(inner class)란?

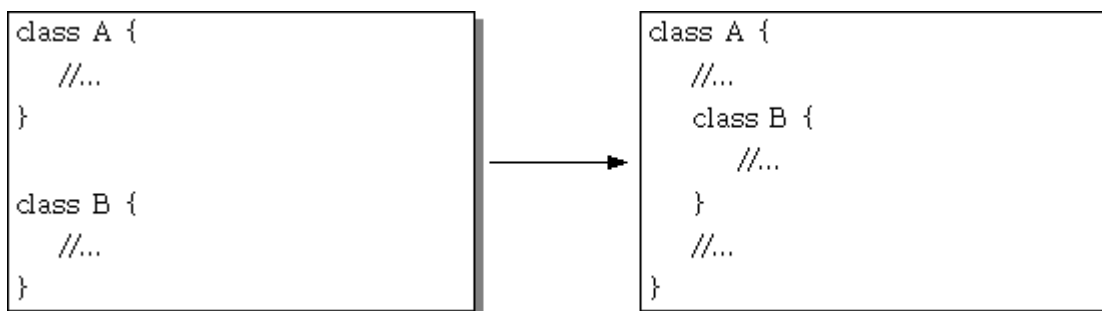
내부클래스란, 클래스 내에 선언된 클래스이다. 클래스에 다른 클래스 선언하는 이유는 간단하다. 두 클래스가 서로 긴밀한 관계에 있기 때문이다. .

한 클래스를 다른 클래스의 내부클래스로 선언하면 두 클래스의 멤버들간에 서로 쉽게 접근할 수 있다는 것과 외부에는 불필요한 클래스를 감춤으로써 코드의 복잡성을 줄일 수 있다는 장점을 얻을 수 있다.

#### .내부클래스의 장점

- 내부클래스에서 외부클래스의 멤버들을 쉽게 접근할 수 있다.
- 코드의 복잡성을 줄일 수 있다.(캡슐화)

[참고]내부 클래스는 JDK1.1버전 이후에 추가된 개념이다.



왼 쪽의 A와 B 두 개의 독립적인 클래스를 오른쪽과 같이 바꾸면 B는 A의 내부클래스(inner class)가 되고 A는 B를 감싸고 있는 외부클래스(outer class)가 된다. 이 때 내부클래스인 B는 외부클래스인 A를 제외하고는 다른 클래스에서 사용되지 않아야한다.

내부클래스는 주로 AWT나 Swing과 같은 GUI어플리케이션의 이벤트처리 외에는 잘 사용하지 않을 정도로 사용빈도가 높지 않으므로 내부클래스의 기본 원리와 특징을 이해하는 정도까지만 학습해도 충분하다. 실제로는 발생하지 않을 경우 까지 이론적으로 만들어 내서 고민하지말자.

내부클래스는 클래스 내에 선언된다는 점을 제외하고는 일반적인 클래스와 다르지 않다. 다만 앞으로 배우게 될 내부클래스의 몇 가지 특징만 잘 이해하면 실제로 활용하는데 어려움이 없을 것이다.

2. 내부클래스의 종류와 특징

내 부클래스의 종류는 변수의 선언위치에 따른 종류와 같다. 내부클래스는 마치 변수를 선언하는 것과 같은 위치에 선언할 수 있으며, 변수의 선언위치에 따라 인스턴스변수, 클래스변수(static변수), 지역변수로 구분되는 것과 같이 내부클래스도 선언위치에 따라 다음과 같이 구분되어 진다. 내부클래스의 유효범위와 성질이 변수와 유사하므로 서로 비교해보면 이해하는데 많은 도움이 된다.

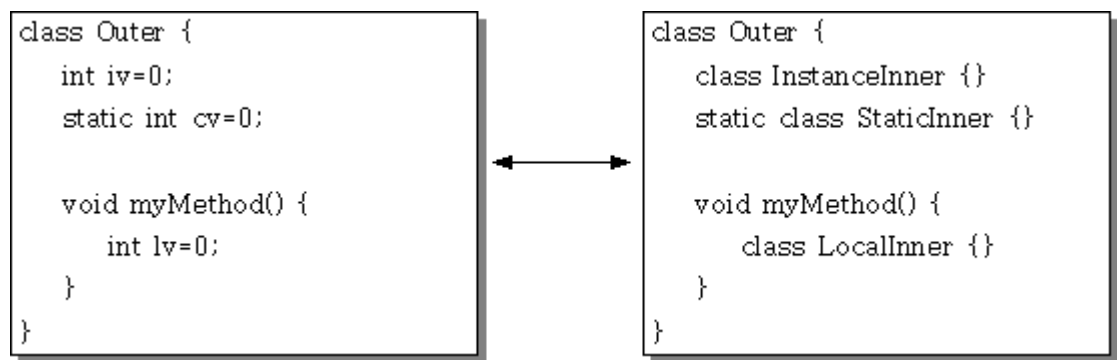
인스턴스클래스 (instance class)	외부클래스의 멤버변수 선언위치에 선언하며, 외부클래스의 인스턴스멤버처럼 다루어진다. 주로 외부클래스의 인스턴스멤버들과 관련된 작업에 사용될 목적으로 선언된다.
스태틱클래스 (static class)	외부클래스의 멤버변수 선언위치에 선언하며, 외부클래스의 static 멤버처럼 다루어진다. 주로 외부클래스의 static멤버, 특히 static메서드에서 사용될 목적으로 선언된다.
지역클래스 (local class)	외부클래스의 메서드나 초기화블럭 안에 선언하며, 선언된 영역 내부에서만 사용될 수 있다.
익명클래스 (anonymous class)	클래스의 선언과 객체의 생성을 동시에 하는 이름없는 클래스(일회용)

[표10-1]내부클래스의 종류와 특징  
[참고]초기화블럭 관련내용은 1권 p.167를 참고

3. 내부클래스의 선언

아래의 오른쪽 코드에는 외부클래스(Outer)에 3개의 서로 다른 종류의 내부클래스를 선언했다. 양쪽의 코드를 비교해 보면 내부클래스의 선언위치가 변수의 선언위치와 동일함을 알 수 있다.

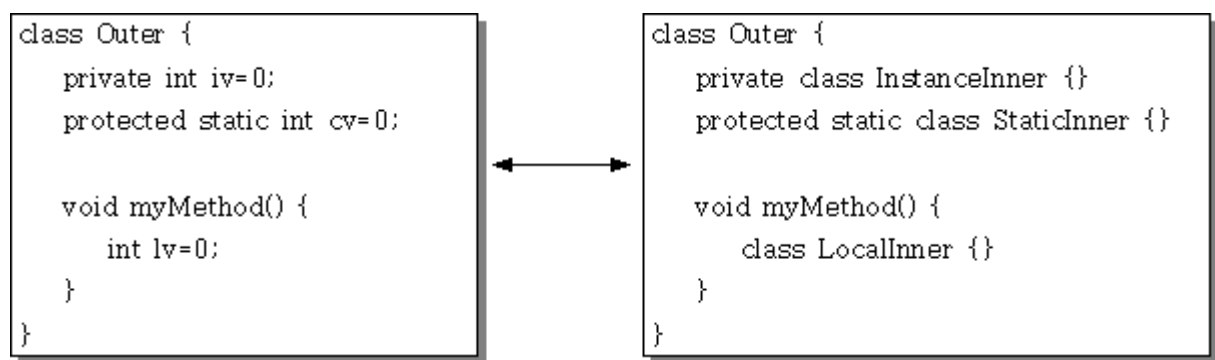
변 수가 선언된 위치에 따라 인스턴스변수, 스테틱변수(클래스변수), 지역변수로 나뉘듯이 내부클래스도 이와 마찬가지로 선언된 위치에 따라 나뉜다. 그리고, 각 내부클래스의 선언위치에 따라 같은 선언위치의 변수와 동일한 유효범위(scope)와 접근성(accessibility)을 갖는다.



#### 4. 내부클래스의 제어자와 접근성

아래코드에서 인스턴스클래스(InstanceInner)와 스테틱클래스(StaticInner)는 외부클래스(Outer)의 멤버변수(인스턴스 변수와 클래스변수)와 같은 위치에 선언되며, 또한 멤버변수와 같은 성질을 갖는다.

따라서 내부클래스가 외부클래스의 멤버와 같이 간주되고, 인스턴스멤버와 static멤버간의 규칙이 내부클래스에도 똑같이 적용된다.



내부클래스도 클래스이기 때문에 abstract나 final과 같은 제어자를 사용할 수 있을 뿐만 아니라, 멤버변수들처럼 private, protected과 접근제어자도 사용이 가능하다.

[예제10-1] InnerEx1.java

```

class InnerEx1 {
    class InstanceInner {
        int iv=100;
//        static int cv=100;           // 에러! static변수를 선언할 수 없다.
        final static int CONST = 100;    // static final은 상수이므로 허용한다.

    }

    static class StaticInner {
        int iv=200;
        static int cv=200;    //    static클래스만 static멤버를 정의할 수 있다.
    }

    void myMethod() {
        class LocalInner {
            int iv=300;
//            static int cv=300;           // 에러! static변수를 선언할 수 없다.
            final static int CONST = 300;    // static final은 상수이므로 허용한다.
        }
    }

    public static void main(String args[]) {
        System.out.println(InstanceInner.CONST);
        System.out.println(StaticInner.cv);
    }
}

```

#### [실행결과]

100

200

[참고]final이 붙은 변수는 상수(constant)이기 때문에 어떤 경우라도 static을 붙이는 것이 가능하다.

내부클래스 중에서 스테틱클래스(StaticInner)만 static멤버를 가질 수 있다. 드문 경우지만 내부클래스에 static변수를 선언해야한다면 스테틱클래스로 선언해야한다.

다만 final과 static이 동시에 붙은 변수는 상수이므로 모든 내부클래스에서 정의가 가능하다.

#### [예제 10-2] InnerEx2.java

```
class InnerEx2 {  
    class InstanceInner {}  
    static class StaticInner {}  
  
    InstanceInner iv = new InstanceInner(); // 인스턴스 멤버간에는 서로 직접 접근이 가능하다.  
    static StaticInner cv = new StaticInner(); // static 멤버간에는 서로 직접 접근이 가능하다.  
  
    static void staticMethod() {  
//      static멤버는 인스턴스 멤버에 직접 접근할 수 없다.  
//      InstanceInner obj1 = new InstanceInner();  
        StaticInner obj2 = new StaticInner();  
  
//      굳이 접근하려면 아래와 같이 객체를 생성해야한다.  
//      인스턴스 내부클래스는 외부클래스를 먼저 생성해야만 생성할 수 있다.  
        InnerEx2 outer = new InnerEx2();  
        InstanceInner obj1 = outer.new InstanceInner();  
    }  
  
    void instanceMethod() {  
// 인스턴스 메서드에서는 인스턴스멤버와 static멤버 모두 접근 가능하다.  
        InstanceInner obj1 = new InstanceInner();  
        StaticInner obj2 = new StaticInner();  
//      메서드 내에 지역적으로 선언된 내부클래스는 외부에서 접근할 수 없다.  
//      LocalInner lv = new LocalInner();  
    }  
  
    void myMethod() {  
        class LocalInner {}  
        LocalInner lv = new LocalInner();  
    }  
}
```

```

    }
}

```

인스턴스멤버는 같은 클래스에 있는 인스턴스멤버와 static멤버 모두 직접 호출이 가능하지만, static멤버는 인스턴스멤버를 직접 호출할 수 없는 것처럼, 인스턴스클래스는 외부클래스의 인스턴스멤버를 객체 생성없이 바로 사용할 수 있지만, 스택틱클래스는 외부클래스의 인스턴스멤버를 객체 생성없이 사용할 수 없다.

마찬가지로 인스턴스클래스는 스택틱클래스의 멤버들을 객체생성없이 사용할 수 있지만, 스택틱클래스에서는 인스턴스클래스의 멤버들을 객체생성없이 사용할 수 없다.

### [예제10-3] InnerEx3.java

```

class InnerEx3 {
    private int outerlv = 0;
    static int outerCv = 0;

    class InstanceInner {
        int iiv = outerlv;        // 외부클래스의 private멤버도 접근가능하다.
        int iiv2 = outerCv;
    }

    static class StaticInner {
// static클래스는 외부클래스의 인스턴스 멤버에 접근할 수 없다.
//         int siv = outerlv;
        static int scv = outerCv;
    }

    void myMethod() {
        int lv = 0;
        final int LV = 0;

        class LocalInner {
            int liv = outerlv;
            int liv2 = outerCv;
//     외부클래스의 지역변수는 final이 붙은 변수(상수)만 접근가능하다.

```

```
//      int liv3 = lv;
      int liv4 = LV;
    }
  }
}
```

내부클래스에서 외부클래스의 변수들에 대한 접근성을 보여 주는 예제이다. 내부클래스의 전체 내용 중에서 제일 중요한 부분이므로 잘봐두도록 하자.

인스턴스클래스(InstanceInner)는 외부클래스(InnerEx3)의 인스턴스멤버이기 때문에 인스턴스변수 outerlv와 static변수 outerCv를 모두 사용할 수 있다. 심지어는 outerlv의 접근제어자가 private일지라도 사용가능하다.

스태틱클래스(StaticInner)는 외부클래스(InnerEx3)의 static멤버이기 때문에 외부클래스의 인스턴스멤버인 outerlv와 InstanceInner를 사용할 수 없다. 단지 static멤버인 outerCv만을 사용할 수 있다.

지역클래스(LocalInner)는 외부클래스의 인스턴스멤버와 static멤버를 모두 사용할 수 있으며, 지역클래스가 포함된 메서드에 정의된 지역변수도 사용할 수 있다.

단, final이 붙은 지역변수만 접근가능한데 그 이유는 메서드가 수행을 마쳐서 지역변수가 소멸된 시점에도, 지역클래스의 인스턴스가 소멸된 지역변수를 참조하려는 경우가 발생할 수 있기 때문이다.

#### [예제10-4] InnerEx4.java

```
class Outer {
    class InstanceInner {
        int iv=100;
    }
    static class StaticInner {
        int iv=200;
        static int cv=300;
    }

    void myMethod() {
        class LocalInner {
            int iv=400;
        }
    }
}
```



```

    }
}

class InnerEx4 {
    public static void main(String args[]) {
        // 인스턴스 내부클래스의 인스턴스를 생성하려면
        // 외부클래스의 인스턴스를 먼저 생성해야한다.
        Outer oc = new Outer();
        Outer.InstanceInner ii = oc.new InstanceInner();

        System.out.println("ii.iv : "+ ii.iv);
        System.out.println("Outer.StaticInner.cv : "+ Outer.StaticInner.cv);

        // Static내부클래스의 인스턴스는 외부클래스를 생성하지 않아도 된다.
        Outer.StaticInner si = new Outer.StaticInner();

        System.out.println("si.iv : "+ si.iv);
    }
}

```

#### [실행결과]

```

ii.iv : 100
Outer.StaticInner.cv : 300
si.iv : 200

```

외 부클래스가 아닌 다른 클래스에서 내부클래스를 생성하고 내부클래스의 멤버에 접근하는 예제이다. 실제로 이런 경우가 발생했다는 것은 내부클래스로 선언해서는 안되는 클래스를 내부클래스로 선언했다는 의미이다. 참고로만 봐두고 가볍게 넘어가도록 하자.

위 예제를 컴파일했을 때 생성되는 클래스 파일은 다음과 같다.

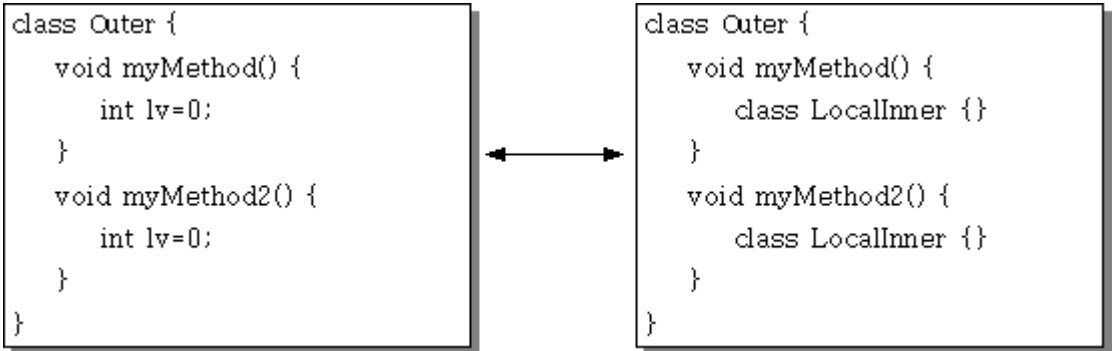
```

InnerEx4.class
Outer.class
Outer$InstanceInner.class
Outer$StaticInner.class

```

Outer\$1LocallInner.class

컴파일 했을 때 생성되는 파일명은 '외부클래스명\$내부클래스명.class'형식으로 되어 있다. 다만 서로 다른 메서드 내에서는 같은 이름의 지역변수를 사용하는 것이 가능한 것처럼, 지역내부클래스는 다른 메서드에 같은 이름의 내부클래스가 존재할 수 있기 때문에 내부클래스명 앞에 숫자가 붙는다.



만일 오른쪽의 코드를 컴파일 한다면 다음과 같은 클래스파일이 생성될 것이다.

Outer.class  
Outer\$1LocallInner.class  
Outer\$2LocallInner.class

#### [예제10-5] InnerEx5.java

```
class Outer {  
    int value=10;    // Outer.this.value  
  
    class Inner {  
        int value=20;    // this.value  
        void method1() {  
            int value=30;  
            System.out.println(" value :" + value);  
            System.out.println(" this.value :" + this.value);  
            System.out.println("Outer.this.value :" + Outer.this.value);  
        }  
    }  
}
```

```

    } // Inner클래스의 끝
} // Outer클래스의 끝

class InnerEx5 {
    public static void main(String args[]) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner();
        inner.method1();
    }
} // InnerEx5 끝

```

#### [실행결과]

```

value :30
this.value :20
Outer.this.value :10

```

위의 예제는 내부클래스와 외부클래스에 선언된 변수의 이름이 같을 때 변수 앞에 'this' 또는 '외부클래스명.this'를 붙여서 서로 구별할 수 있다는 것을 보여준다.

### 5. 익명클래스(anonymous class)

이 제 마지막으로 익명 클래스에 대해서 알아보도록 하자. 익명클래스는 특이하게도 다른 내부클래스들과는 달리 이름이 없다. 클래스의 선언과 객체의 생성을 동시에 하기 때문에 한 단번만 사용될 수 있고 오직 하나의 객체만을 생성할 수 있는 일회용 클래스이다.

```

new 조상클래스이름() {
    // 멤버들 선언
}

또는

new 구현인터페이스이름() {
    // 멤버들 선언
}

```

이름이 없기 때문에 생성자도 가질 수 없으며, 조상클래스의 이름이나 구현하고자 하는 인터페이스의 이름을 사용해서 정의하기 때문에 하나의 클래스로 상속받는 동시에 인터페이스를 구현하거나 하나 이상의 인터페이스를 구현할 수 없다. 오로지 단 하나의 클래스를 상속받거나 단 하나의 인터페이스만을 구현할 수 있다.

익명클래스는 구문이 다소 생소하지만, 인스턴스클래스를 익명클래스로 바꾸는 연습을 몇 번만 해 보면 금새 익숙해 질 것이다.

#### [예제10-6] InnerEx6.java

```
class InnerEx6{
    Object iv = new Object(){ void method(){} };      // 익명클래스
    static Object cv = new Object(){ void method(){} }; // 익명클래스

    void myMethod() {
        Object lv = new Object(){ void method(){} };  // 익명클래스
    }
}
```

위의 예제는 단순히 익명클래스의 사용 예를 보여 준 것이다. 이 예제를 컴파일 하면 다음과 같이 4개의 클래스파일이 생성된다.

```
InnerEx6.class
InnerEx6$1.class ← 익명클래스
InnerEx6$2.class ← 익명클래스
InnerEx6$3.class ← 익명클래스
```

익명클래스는 이름이 없기 때문에 '외부클래스명\$숫자.class'의 형식으로 클래스파일명이 결정된다.

#### [예제10-7] InnerEx7.java

```

import java.awt.*;
import java.awt.event.*;

class InnerEx7
{
    public static void main(String[] args)
    {
        Button b = new Button("Start");
        b.addActionListener(new EventHandler());
    }
}

class EventHandler implements ActionListener
{
    public void actionPerformed(ActionEvent e) {
        System.out.println("Action Event occurred!!!");
    }
}

```

#### [예제 10-8] InnerEx8.java

```

import java.awt.*;
import java.awt.event.*;

class InnerEx8
{
    public static void main(String[] args)
    {
        Button b = new Button("Start");
        b.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                System.out.println("Action Event occurred!!!");
            }
        } // 익명 클래스의 끝
    );
}

```

```
    }    // main메서드의 끝  
}    // InnerEx8클래스의 끝
```

예제 InnerEx7.java를 익명클래스를 이용해서 변경한 것이 예제 InnerEx8.java이다. 먼저 두 개의 독립된 클래스를 작성한 다음에, 다시 익명클래스를 이용하여 변경하면 보다 쉽게 코드를 작성할 수 있다.

# [Java의 정석 2판] 11.4 제네릭스(Generics)

자바의정석

2012/12/22 17:15

<http://blog.naver.com/gphic/50157740440>

## 4. 제네릭스(Generics)

제네릭스는, JDK1.5에서의 가장 큰 변화 중의 하나로, 다양한 타입의 객체들을 다루는 메서드나 컬렉션 클래스에 컴파일 시의 타입 체크(compile-time type check)를 해주는 기능이다. 객체의 타입을 컴파일 시에 체크하기 때문에 객체의 타입 안정성을 높이고 형변환의 번거로움이 줄어든다.

ArrayList와 같은 컬렉션 클래스는 다양한 종류의 객체를 담을 수 있긴 하지만 보통 한 종류의 객체를 담는 경우가 더 많다. 그런데도 꺼낼 때 마다 타입체크를 하고 형변환을 하는 것은 아무래도 불편할 수밖에 없다.

### ▶ 제네릭스의 장점

```
<?xml:namespace prefix="o" ns="urn:schemas-microsoft-com:office:office" />
```

1. 타입 안정성을 제공한다.

2. 타입체크와 형변환을 생략할 수 있으므로 코드가 간결해진다.

[참고] 타입 안정성을 높인다는 것은 의도하지 않은 타입의 객체를 저장하는 것을 막고, 저장된 객체를 꺼낼 때 원래의 타입과 다른 타입으로 형변환되어 발생할 수 있는 오류를 줄여준다는 뜻이다.

간단히 얘기하면 다른 객체의 타입을 미리 명시해줌으로써 형변환을 하지 않아도 되게 하는 것이다. 그게 전부이다. 너무 어렵게 생각하지 않길 바란다.

제네릭스에서는 참조형 타입(reference type), 간단히 말해서 '타입(type)'을 의미하는 기호로 'T'를 사용한다. 'T'는 영단어 'type'의 첫 글자로, 어떠한 참조형 타입도 가능하다는 것을 의미한다.

'T'뿐 만 아니라 때때로 요소(element)를 의미하는 'E', 키(key)를 의미하는 'K', 값(value)을 의미하는 'V'도 사용된다. 이들은 기호의 종류만 다를 뿐 '임의의 참조형 타입'을 의미한다는 것은 모두 같다. 마치 수학적 식  $f(x, y) = x + y$ 가  $f(k, v) = k + v$ 와 다르지 않은 것처럼 말이다.

기존에는 다양한 종류의 타입을 다루는 메서드의 매개변수나 리턴타입으로 Object타입의 참조변수를 많이 사용했고, 그로 인해 형변환이 불가피했지만, 이젠 Object타입 대신 원하는 타입을 지정하기만 하면 되는 것이다.

[참고] 타입을 지정하지 않으면 Object타입으로 간주된다.

제 네릭스는 자바의 부가적인 기능 중에 하나일 뿐이다. 사용하면 편리하긴 하지만 사용하지 않아도 그만이다. 저자의 개인적인 생각으로는 프로그래밍을 처음 배우는 사람에게는 학습부담이 될까 다소 우려스럽다. 다형성이나 형변환, 타입체크 등의 중요한 기본원칙에 대해서 완전히 이해하지 못한 상태에서 원칙에서 벗어나는 기능인 제네릭을 배운다면 혼란스러울 수 있기 때문이다.

일단 편안하게 읽어보고 이해가 잘 안 간다면, 부담없이 건너뛰고 다음 진도로 넘어가길 바란다. 자바가 좀 더 익숙해 진 후에 다시 보면, 보다 쉽게 이해가 갈 수 있는 부분이기 때문이다.

#### 4.1 ArrayList<E>

대표적인 컬렉션 클래스인 ArrayList를 통해 제네릭스를 사용하는 방법을 설명해나갈 것이다. 대부분의 컬렉션 클래스들에 대해서도 동일한 방식으로 사용하면 된다.

```
public class ArrayList extends AbstractList
    implements List, RandomAccess, Cloneable, java.io.Serializable
{
    private transient Object[] elementData;
    public boolean add(Object o) { /* 내용생략 */ }
    public Object get(int index) { /* 내용생략 */ }
    ...
}
```

위의 코드는 제네릭스가 도입되기 이전의 ArrayList의 소스이고 아래의 소스는 제네릭스가 도입된 이후의 소스이다. Object타입 대신 임의의 타입 'E'가 사용된 것을 알 수 있다.

```
public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    private transient E[] elementData;
    public boolean add(E o) { /* 내용생략 */ }
    public E get(int index) { /* 내용생략 */ }
    ...
}
```



```
}
```

위의 코드에 사용된 'E'는 요소를 뜻하는 'Element'의 약자로, 'E' 대신 어떤 다른 문자를 사용해도 상관없지만 소스코드 내에서 같은 문자를 일관되게 사용해야한다.

이는 마치 메서드의 매개변수이름을 다르게 해도 메서드 내에서만 같은 이름을 사용하면 문제없는 것과 같다. 만일 타입을 지정하지 않으면 'E'는 Object타입으로 간주된다.

임의의 타입 'E'는 ArrayList타입의 참조변수를 선언하거나 ArrayList를 생성할 때 지정할 수 있으며, 만일 ArrayList에 Tv타입의 객체만을 저장하기 위해 'ArrayList<Tv> tvList = new ArrayList<Tv>();'와 같이한다면 'E'는 'Tv'가 되는 것이다.

```
public class ArrayList<Tv> extends AbstractList<Tv>
    implements List<Tv>, RandomAccess, Cloneable,
        java.io.Serializable
{
    private transient Tv[] elementData;
    public boolean add(Tv o) { /* 내용생략 */ }
    public Tv get(int index) { /* 내용생략 */ }
    ...
}
```

아래와 같이 컬렉션 클래스 이름 바로 뒤에 저장할 객체의 타입을 적어주면, 컬렉션에 저장할 수 있는 객체는 지정한 타입의 객체뿐이다.

```
컬렉션클래스<저장할 객체의 타입> 변수명 = new 컬렉션클래스<저장할 객체의 타입>();
ArrayList<Tv> tvList = new ArrayList<Tv>();
```

아래의 코드는 ArrayList에 Tv객체만 저장할 수 있도록 작성한 것이다. Tv타입이 아닌 객체를 저장하려하면 컴파일 시에 에러가 발생한다.

```
// Tv객체만 저장할 수 있는 ArrayList를 생성
ArrayList<Tv> tvList = new ArrayList<Tv>();
tvList.add(new Tv());
tvList.add(new Audio()); // 컴파일 에러 발생!!!
```

저장된 객체를 꺼낼 때는 형변환할 필요가 없다. 이미 어떤 타입의 객체들이 저장되어 있는지 알고 있기 때문이다. 제네릭스를 적용한 코드(오른쪽)와 그렇지 않은 코드(왼쪽)를 잘 비교해보자.

<pre>ArrayList tvList = new ArrayList(); tvList.add(new Tv()); Tv t = (Tv)tvList.get(0);</pre>	<pre>ArrayList&lt;Tv&gt; tvList = new ArrayList&lt;Tv&gt;(); tvList.add(new Tv()); Tv t = tvList.get(0);</pre>
--	--

만일 다형성을 사용해야하는 경우에는 조상타입을 지정함으로써 여러 종류의 객체를 저장할 수 있다.

```
class Product{}
class Tv extends Product{}
class Audio extends Product{}

// Product클래스의 자손객체들을 저장할 수 있는 ArrayList를 생성
ArrayList<Product> list = new ArrayList<Product>();
list.add(new Product());
list.add(new Tv()); // 컴파일 에러가 발생하지 않는다.
list.add(new Audio()); // 컴파일 에러가 발생하지 않는다.

Product p = list.get(0); // 형변환이 필요없다.
Tv t = (Tv)list.get(1); // 형변환을 필요로 한다.
```

ArrayList가 Product타입의 객체를 저장하도록 지정하면, 이들의 자손인 Tv와 Audio타입의 객체도 저장할 수 있다. 다만 꺼내올 때 원래의 타입으로 형변환해야 한다. 제네릭스에서도 다형성을 적용해서 아래와 같이 할 수 있다.

```
List<Tv> tvList = new ArrayList<Tv>(); // 허용
```

그러나 Product클래스가 Tv클래스의 조상이라 할지라도 아래와 같이 할 수는 없다.

```
ArrayList<Product> list = new ArrayList<Tv>(); // 허용 안함~!!!
```

그 래서 아래와 같이 메서드의 매개변수 타입이 ArrayList<Product>로 선언된 경우, 이 메서드의 매개변수로는 ArrayList<Product>타입의 객체만 사용할 수 있다. 그렇지 않으면 컴파일 에러가 발생한다.

```
public static void printAll(ArrayList<Product> list) {  
    for(Unit u : list) { // 항상된 for문, 부록 참고  
        System.out.println(u);  
    }  
}  
  
public static void main(String[] args) {  
    ArrayList<Product> productList = new ArrayList<Product>();  
    ArrayList<Tv> tvList = new ArrayList<Tv>();  
  
    printAll(productList);  
    printAll(tvList); // 컴파일 에러 발생~!!!  
}
```

컬렉션에 저장될 객체에도 다형성이 필요할 때도 있을 것 같은데 다형성을 사용할 수는 없을까?

물 론 방법이 있다. 와일드 카드 '?'를 사용하면 된다. 보통 제네릭에서는 단 하나의 타입을 지정하지만, 와일드 카드는 하나 이상의 타입을 지정하는 것을 가능하게 해준다. 아래와 같이 어떤 타입('?')이 있고 그 타입이 Product의 자손이라고 선언하면, Tv객체를 저장하는 'ArrayList<Tv>' 또는 Audio객체를 저장하는 'ArrayList<Audio>'를 매개변수로 넘겨줄 수 있다. Tv와 Audio 모두 Product의 자손이기 때문이다.

```
// Product 또는 그 자손들이 담긴 ArrayList를 매개변수로 받는 메서드  
public static void printAll(ArrayList<? extends Product> list) {  
    for(Unit u : list) {  
        System.out.println(u);  
    }  
}
```

만일 아래와 같은 코드가 있다면 타입을 별도로 선언함으로써 코드를 간략히 할 수 있다.

```

public static void printAll(ArrayList<? extends Product> list,
    ArrayList<? extends Product> list2) {

    for(Unit u : list) {
        System.out.println(u);
    }
}

```

```

public static <T extends Product> void printAll(ArrayList<T> list,
    ArrayList<T> list2) {

    for(Unit u : list) {
        System.out.println(u);
    }
}

```

두 번째 코드는 'T'라는 타입이 Product의 자손타입이라는 것을 미리 정의해 놓고 사용한 것이다. 위의 두 코드는 서로 같은 의미의 코드이므로 잘 비교해보자.

[주의] 여기서 만일 Product가 클래스가 아닌 인터페이스라 할지라도 키워드로 'implements'를 사용하지 않고 클래스와 동일하게 'extends'를 사용한다는 것에 주의하자.

[예제11-80]/ch11/GenericsEx1.java

```

import java.util.*;

class Product {}
class Tv extends Product{}
class Audio extends Product{}

class GenericsEx1 {
    public static void main(String[] args) {
        ArrayList<Product> productList = new ArrayList<Product>();
        ArrayList<Tv> tvList = new ArrayList<Tv>();

        productList.add(new Tv());
        productList.add(new Audio());

        tvList.add(new Tv());
        tvList.add(new Tv());

        printAll(productList);
        // printAll(tvList);    // 컴파일 에러가 발생한다.

        printAll2(productList); // ArrayList<Product>
        printAll2(tvList);    // ArrayList<Tv>
    }

    public static void printAll(ArrayList<Product> list) {
        for(Product p : list) {
            System.out.println(p);
        }
    }

    // public static void printAll2(ArrayList<? extends Product> list) {
    public static <T extends Product> void printAll2(ArrayList<T> list) {
        for(Product p : list) {
            System.out.println(p);
        }
    }
}

```

```
}  
}
```

#### [실행결과]

```
Tv@9cab16  
Audio@1a46e30  
Tv@9cab16  
Audio@1a46e30  
Tv@3e25a5  
Tv@19821f
```

이 예제의 결과는 별 의미가 없다. 예제를 변경해가면서 전에 설명한 내용을 직접 테스트 해보자.

한 가지 설명하고 넘어갈 것은 예제에 사용된 for문의 형태인데, 이 새로운 구문의 for문은 ‘향상된 for문’이라고 한다. list에 담긴 모든 요소를 반복할 때마다 하나씩 가져다 Product타입의 참조변수 p에 저장한다. 아래의 두 코드는 같은 것이니 잘 비교해 보자.

[참고] 향상된 for문에 대한 보다 자세한 내용은 책의 마지막에 있는 ‘부록’을 참고하라.

```
Iterator it = list.iterator();
```

```
for(;it.hasNext();) {  
    Product p = (Product)it.next();  
    System.out.println(p);  
}
```

```
for(Product p : list) {  
    System.out.println(p);  
}
```

## 4.2 Iterator<E>

다음은 Iterator의 실제 소스인데, 컬렉션 클래스 뿐 만아니라 Iterator에도 제네릭스가 적용되어 있는 것을 알 수 있다.

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove();  
}
```

아래의 예제는 Iterator에 제네릭스를 적용한 예이다.

[예제11-81]/ch11/GenericsEx2.java

```

import java.util.*;

class GenericsEx2
{
    public static void main(String[] args)
    {
        ArrayList<Student> list = new ArrayList<Student>();
        list.add(new Student("자바왕",1,1));
        list.add(new Student("자바짱",1,2));
        list.add(new Student("홍길동",2,1));
        list.add(new Student("전우치",2,2));

        Iterator<Student> it = list.iterator();

        while(it.hasNext()) {
            Student s = it.next();
            System.out.println(s.name);
        }
    } // main
}

class Student {
    String name = "";
    int ban;
    int no;

    Student(String name, int ban, int no) {
        this.name = name;
        this.ban = ban;
        this.no = no;
    }
}

```

[실행결과]



자바왕  
자바짱  
홍길동  
전우치

ArrayList에 Student객체를 저장할 것이라고 지정을 했어도 Iterator타입의 참조변수를 선언할 때 저장된 객체의 타입을 지정해주지 않으면, Iterator의 next()를 호출할 때 형변환을 해야 한다.

```
Iterator it = list.iterator();
```

```
while(it.hasNext()) {  
    Student s = (Student)it.next();  
    System.out.println(s.name);  
}
```

```
Iterator<Student> it = list.iterator();
```

```
while(it.hasNext()) {  
    Student s = it.next();  
    System.out.println(s.name);  
}
```

#### 4.3 Comparable<T>과 Collections.sort()

클래스의 기본 정렬기준을 구현하는 Comparable인터페이스에도 제네릭스가 적용된다. 먼저 예제를 살펴보자. 이 예제는 Comparable를 사용해서 학생들의 총점을 기준으로 내림차순 정렬하여 출력한다.

[예제11-82]/ch11/GenericsEx3.java

```

import java.util.*;

class GenericsEx3
{
    public static void main(String[] args)
    {
        ArrayList<Student> list = new ArrayList<Student>();
        list.add(new Student("자바왕",1,1,100,100,100));
        list.add(new Student("자바짱",1,2,90,80,70));
        list.add(new Student("홍길동",2,1,70,70,70));
        list.add(new Student("전우치",2,2,90,90,90));

        Collections.sort(list); // list를 정렬한다.

        Iterator<Student> it = list.iterator();

        while(it.hasNext()) {
            Student s = it.next();
            System.out.println(s);
        }
    }
}

class Student implements Comparable<Student> {
    String name = "";
    int ban = 0;
    int no = 0;
    int koreanScore = 0;
    int mathScore = 0;
    int englishScore = 0;

    int total = 0;

    Student(String name, int ban, int no, int koreanScore, int mathScore, int
englishScore) {

```

```

        this.name = name;
        this.ban = ban;
        this.no = no;
        this.koreanScore = koreanScore;
        this.mathScore = mathScore;
        this.englishScore = englishScore;

        total = koreanScore + mathScore + englishScore;
    }

    public String toString() {
        return name + "₩t"
            + ban + "₩t"
            + no + "₩t"
            + koreanScore + "₩t"
            + mathScore + "₩t"
            + englishScore + "₩t"
            + total + "₩t";
    }

    public int compareTo(Student o) {
        return o.total - this.total;
    }
} // end of class Student

```

#### [실행결과]

자바왕	1	1	100	100	100	300
전우치	2	2	90	90	90	270
자바짱	1	2	90	80	70	240

홍길동 2 1 70 70 70 210

학생들의 정보를 담은 Student인스턴스를 ArrayList에 담은 다음, Collection.sort()를 이용해서 Student클래스에 정의된 기본정렬(총점별로 내림차순)로 정렬해서 출력하는 간단한 예제이다.

먼저 제네릭스가 적용된 Comparable의 실제 소스를 보면 아래와 같다.

```
public interface Comparable<T> {  
    public int compareTo(T o); // 지정한 타입 T를 매개변수로 한다.  
}
```

여기서 타입 'T'대신 타입 'Student'를 지정했기 때문에 아래와 같이 된다.

```
public interface Comparable<Student> {  
    public int compareTo(Student o); // 지정한 타입 T를 매개변수로 한다.  
}
```

만일 이 예제에서 제네릭스를 사용하지 않았다면, 왼쪽의 코드 대신 오른쪽과 같은 코드를 사용해야 했을 것이다.

```
public int compareTo(Student o) {  
    return o.total - this.total;  
}
```

```
public int compareTo(Object o) {  
    int result = -1;  
  
    if(o instanceof Student) {  
        Student tmp = (Student)o;  
        result = tmp.total - this.total;  
    }  
    return result;  
}
```

한 눈에 봐도 왼쪽의 코드가 더 간결하다는 것을 알 수 있다. 타입이 미리 체크되어 있기 때문에 instanceof로 타입을 체크하거나 형변환할 필요가 없기 때문이다.

여기서 Collections.sort()에 적용된 제네릭스에 대해서 좀 더 자세히 살펴볼 필요가 있다.

다음은 Collections.sort()의 선언부인데 지금까지 보던 것과는 달리 좀 복잡하다.

```
public static <T extends Comparable<? super T>> void sort(List<T> list)
```

②

①

① ArrayList와 같이 List인터페이스를 구현한 컬렉션을 매개변수의 타입으로 정의하고 있다. 그리고 그 컬렉션에는 'T'라는 타입의 객체를 저장하도록 선언되어 있다.

② 'T'는 Comparable인터페이스를 구현한 클래스의 타입이어야 하며(<T extends Comparable>), 'T'또는 그 조상의 타입을 비교하는 Comparable이어야 한다는 것(Comparable<? super T>)을 의미한다.

만일 Student클래스가 Person클래스의 자손이라면, <? super T>는 Student타입이나 Person타입이 가능하다.(물론 Object타입도 가능)

<? extends T> - T 또는 T의 자손 타입을 의미한다.

<? super T> - T 또는 T의 조상 타입을 의미한다.

앞서 배운 <? extends T>와 반대로 <? super T>는 T 또는 T의 조상 타입을 의미한다.

[예제11-83]/ch11/GenericsEx4.java

```

import java.util.*;

class GenericsEx4
{
    public static void main(String[] args)
    {
        ArrayList<Student> list = new ArrayList<Student>();
        list.add(new Student("자바왕",1,1,100,100,100));
        list.add(new Student("자바짱",1,2,90,80,70));
        list.add(new Student("홍길동",2,1,70,70,70));
        list.add(new Student("전우치",2,2,90,90,90));

        Collections.sort(list); // list를 정렬한다.

        Iterator<Student> it = list.iterator();

        while(it.hasNext()) {
            Student s = it.next();
            System.out.println(s);
        }
    } // main
}

// <T extends Comparable<? super T>>에서 'T'가 Student타입이므로
// <Student extends Comparable<Student>>와
// <Student extends Comparable<Person>>이 가능하다.
class Student extends Person implements Comparable<Person> {
    String name = "";
    int ban = 0;
    int no = 0;
    int koreanScore = 0;
    int mathScore = 0;
    int englishScore = 0;

    int total = 0;

```

```

Student(String name, int ban, int no, int koreanScore, int mathScore, int
englishScore) {
    super(ban+"-"+no, name);
    this.name = name;
    this.ban = ban;
    this.no = no;
    this.koreanScore = koreanScore;
    this.mathScore = mathScore;
    this.englishScore = englishScore;

    total = koreanScore + mathScore + englishScore;
}

```

```

public String toString() {
    return name + "\t"
        + ban + "\t"
        + no + "\t"
        + koreanScore + "\t"
        + mathScore + "\t"
        + englishScore + "\t"
        + total + "\t";
}

```

// Comparable<Person>이므로 Person타입의 매개변수를 선언.

```

public int compareTo(Person o) {
    return id.compareTo(o.id); // String클래스의 compareTo()를 호출
}

```

```

} // end of class Student

```

```

class Person {
    String id;
    String name;
}

```

```
Person(String id, String name) {  
    this.id = id;  
    this.name = name;  
}  
}
```



#### [실행결과]

|     |   |   |     |     |     |     |
|-----|---|---|-----|-----|-----|-----|
| 자바왕 | 1 | 1 | 100 | 100 | 100 | 300 |
| 자바짱 | 1 | 2 | 90  | 80  | 70  | 240 |
| 홍길동 | 2 | 1 | 70  | 70  | 70  | 210 |
| 전우치 | 2 | 2 | 90  | 90  | 90  | 270 |

이전 예제의 Student클래스를 변경하여 Person클래스의 자손이 되도록 하고, Student의 조상인 Person을 Comparable의 타입으로 지정하였다.

Collections.sort() 의 매개변수가 '<T extends Comparable<? super T>>'이기 때문에 예제에서는 'T'가 Student타입이므로 '<Student extends Comparable<Person>>' 또는 '<Student extends Comparable<Student>>'가 가능하며, 그 중에서 '<Student extends Comparable<Person>>'의 경우를 보여주고 있다.

```
// <T extends Comparable<? super T>>에서 'T'가 Student타입이므로
// <Student extends Comparable<Student>>와
// <Student extends Comparable<Person>>이 가능하다.
class Student extends Person implements Comparable<Person> {
    ...
}
```

Person클래스에 정의된 멤버변수 id는 Student클래스의 멤버변수 ban과 no를 문자열로 붙여서 만들었기 때문에, 실행 결과를 보면 전과 달리 반과 번호를 기준으로 오름차순 정렬되어 있는 것을 알 수 있다.

#### 4.4 HashMap<K,V>

HashMap 처럼 데이터를 키(key)와 값(value)의 형태로 저장하는 컬렉션 클래스는 지정해 줘야할 타입이 두 개이다. 그래서 'K,V'와 같이 두 개의 타입을 콤마','로 구분해서 적어줘야 한다. 여기서 'K'와 'V'는 각각 'Key'의 'Value'의 첫 글자에서 따온 것일 뿐, 'T'나 'E'와 마찬가지로 임의의 참조형 타입(reference type)을 의미한다.

다음은 HashMap의 실제 소스이다.

```
public class HashMap<K,V> extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable
{
    ...
    public V get(Object key) { /* 내용 생략 */ }
    public V put(K key, V value) { /* 내용 생략 */ }
    public V remove(Object key) { /* 내용 생략 */ }
    ...
}
```

만일 키의 타입이 String이고 저장할 값의 타입이 Student인 HashMap을 생성하려면 다음과 같이 한다.

```
HashMap<String, Student> map = new HashMap<String, Student>(); // 생성
map.put("자바왕", new Student("자바왕", 1, 1, 100, 100, 100)); // 데이터 저장
```

위와 같이 HashMap을 생성하였다면, HashMap의 실제 소스는 'K'대신 String이, 'V'대신 Student가 사용되어 아래와 같이 바뀌는 셈이 된다.

```
public class HashMap<String, Student>
    extends AbstractMap<String, Student>
    implements Map<String, Student>, Cloneable, Serializable
{
    ...
    public Student get(Object key) { /* 내용 생략 */ }
```

```
public Student put(String key, Student value) { /* 내용 생략 */ }  
public Student remove(Object key) { /* 내용 생략 */ }  
...  
}
```

여기서도 타입을 지정하지 않으면 Object타입으로 간주된다.

[예제11-84]/ch11/GenericsEx5.java

```

import java.util.*;

class GenericsEx5
{
    public static void main(String[] args)
    {
        HashMap<String,Student> map = new HashMap<String,Student>();
        map.put("1-1", new Student("자바왕",1,1,100,100,100));
        map.put("1-2", new Student("자바짱",1,2,90,80,70));
        map.put("2-1", new Student("홍길동",2,1,70,70,70));
        map.put("2-2", new Student("전우치",2,2,90,90,90));

        Student s1 = map.get("1-1");
        System.out.println("1-1 : " + s1.name);

        Iterator<String> itKey = map.keySet().iterator();

        while(itKey.hasNext()) {
            System.out.println(itKey.next());
        }

        Iterator<Student> itValue = map.values().iterator();
        int totalSum = 0;

        while(itValue.hasNext()) {
            Student s = itValue.next();
            totalSum += s.total;
        }

        System.out.println("전체 총점:"+totalSum);
    } // main
}

```

```

444 class Student extends Person implements Comparable<Person> {
    String name = "";

```

```
int ban = 0;
int no = 0;
int koreanScore = 0;
int mathScore = 0;
int englishScore = 0;
```

```
int total = 0;
```

```
Student(String name, int ban, int no, int koreanScore, int mathScore, int
englishScore) {
    super(ban+"-"+no, name);
    this.name = name;
    this.ban = ban;
    this.no = no;
    this.koreanScore = koreanScore;
    this.mathScore = mathScore;
    this.englishScore = englishScore;

    total = koreanScore + mathScore + englishScore;
}
```

```
public String toString() {
    return name + "\n"
        + ban + "\n"
        + no + "\n"
        + koreanScore + "\n"
        + mathScore + "\n"
        + englishScore + "\n"
        + total + "\n";
}
```

// Comparable<Person>이므로 Person타입의 매개변수를 선언.

```
public int compareTo(Person o) {
```

```
445 return id.compareTo(o.id); // String클래스의 compareTo()를 호출
}
```

```
} // end of class Student
```

```
class Person {  
    String id;  
    String name;  
  
    Person(String id, String name) {  
        this.id = id;  
        this.name = name;  
    }  
}
```

### [실행결과]

1-1 :자바왕

2-2

1-2

1-1

2-1

전체 총점:1020

이 전 예제를 ArrayList대신 HashMap을 사용하도록 변경한 것일 뿐 별로 특별한 것은 없다. HashMap에서 값을 꺼내오는 `get(Object key)`를 사용할 때, 그리고 저장된 키와 값들을 꺼내오는 `keySet()`과 `values()`를 사용할 때 형변환을 하지 않아도 된다.

```
Student s1 = (Student)map.get("1-1");
```

```
Student s1 = map.get("1-1");
```