

자바개요

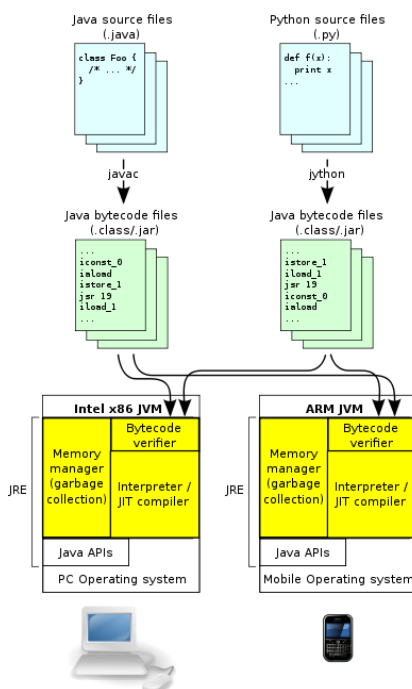
자바는 썬 마이크로 시스템즈의 제임스 고슬링(James Gosling)과 다른 연구원들이 개발한 객체 지향적 프로그래밍 언어로써 썬 마이크로 시스템즈에서 무료로 제공하고 있다. 1991년 그린 프로젝트(Green Project)라는 이름으로 시작하여 1995년에 발표되었으며 가전제품 내에 탑재하여 동작하는 프로그램을 위해 개발했지만 현재에는 웹 어플리케이션 개발에 가장 많이 사용하는 언어 중 하나이며, 모바일 기기용 소프트웨어 개발에도 널리 사용되고 있다.

자바의 가장 큰 특징은 컴파일 된 코드가 플랫폼 독립적이라는 점이다. 자바 컴파일러는 자바 언어로 작성된 소스 프로그램을 바이트코드라는 특수한 바이너리 형태로 변환하며, 변환된 바이트코드는 자바 가상 머신(JVM)에 의해 실행된다.

자바 가상 머신(JVM)은 자바 바이트코드를 어느 플랫폼에서나 동일한 형태로 실행시킨다. 때문에 자바로 개발된 프로그램은 CPU나 운영 체제의 종류에 관계없이 JVM이 설치된 시스템에서는 실행될 수 있다.

- 자바는 JVM에 종속적이다.
- JVM은 플랫폼에 종속적이다.

자바 소스코드는 바이트코드로 컴파일되며, JVM에 의해 코드 검증 및 해석되어 실행된다.



1. 자바 개발 환경 구축하기

1.1 Java SDK 설치

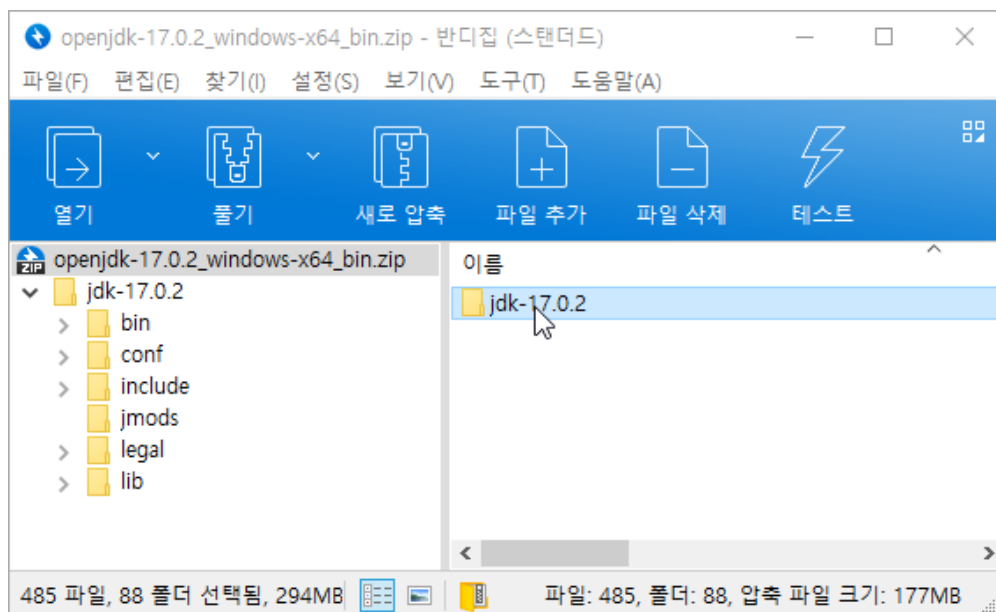
아래의 홈페이지에 접속하여 Java SE Development Kit를 다운로드하기 위해 아래의 URL에 접속한다.

<https://jdk.java.net/archive/>

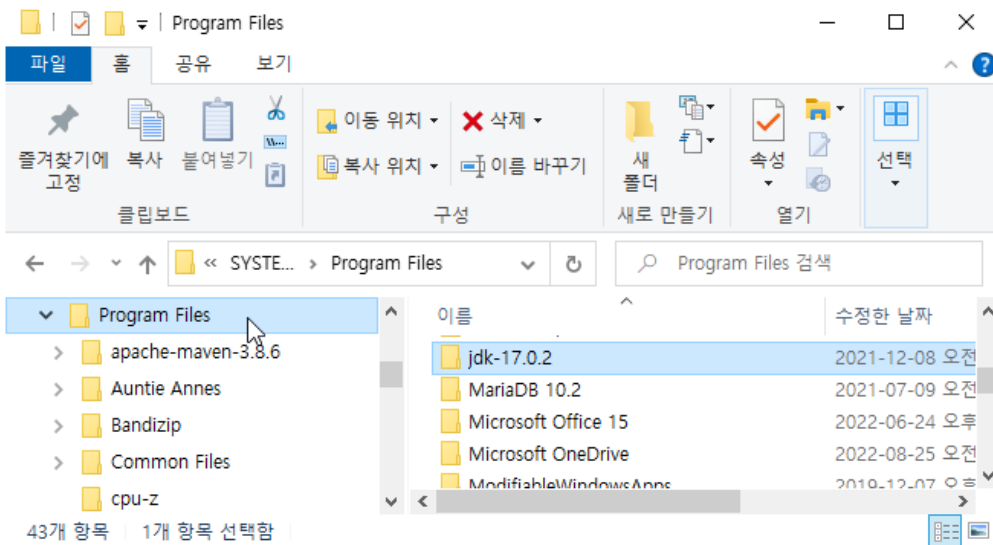
사용중인 플랫폼에 맞는 JDK를 다운로드 한다.



압축파일의 내용은 아래와 같다. 압축 파일내의 jdk-17.0.2 폴더를 설치 폴더에 드래그 앤 드롭하여 압축을 해제한다.



본 강의에서는 "C:\Program Files"에 설치하였다.



JDK의 설치가 완료되었다면 자바 어플리케이션 실행에 필요한 환경변수를 등록한다. 등록하는 환경 변수와 값은 아래와 같다.

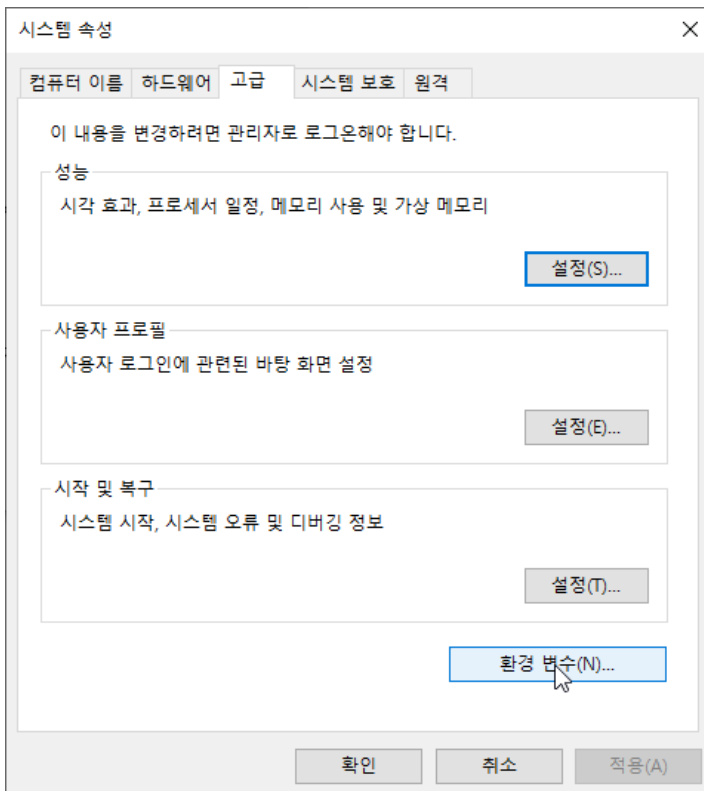
환경 변수 JAVA_HOME을 추가

환경 변수 PATH에 자바관련 실행파일 경로 추가

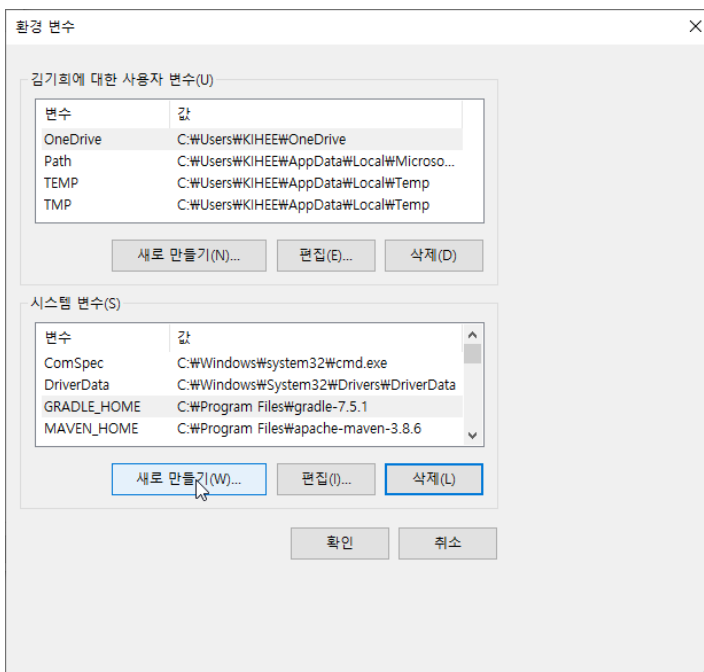
환경 변수 추가를 위해 Windows Key + x, y를 누른 후 설정 창 우측의 "고급 시스템 설정"을 클릭한다.



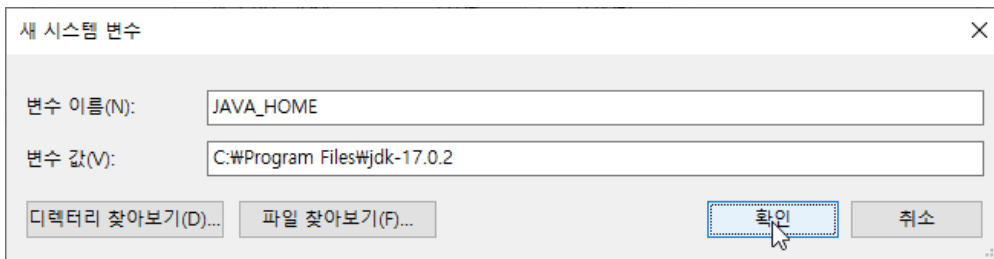
시스템 속성창의 고급 탭을 선택한 후 하단의 "환경 변수(N)..." 버튼을 클릭한다.



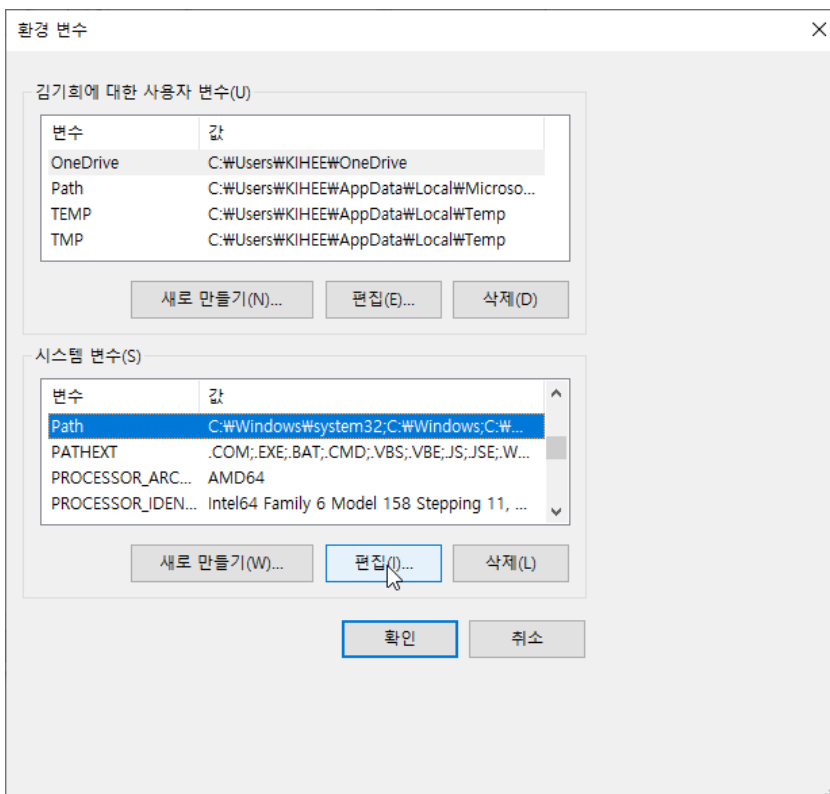
시스템 변수 그룹 하단의 "새로 만들기(w)..." 버튼을 클릭한다.



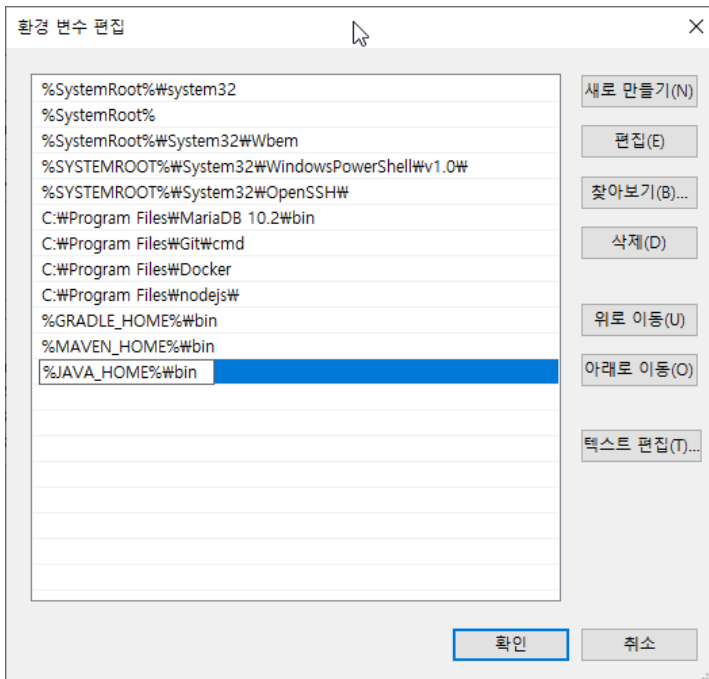
아래와 같이 입력한 후 "확인" 버튼을 클릭하여 JAVA_HOME 환경변수를 등록한다.



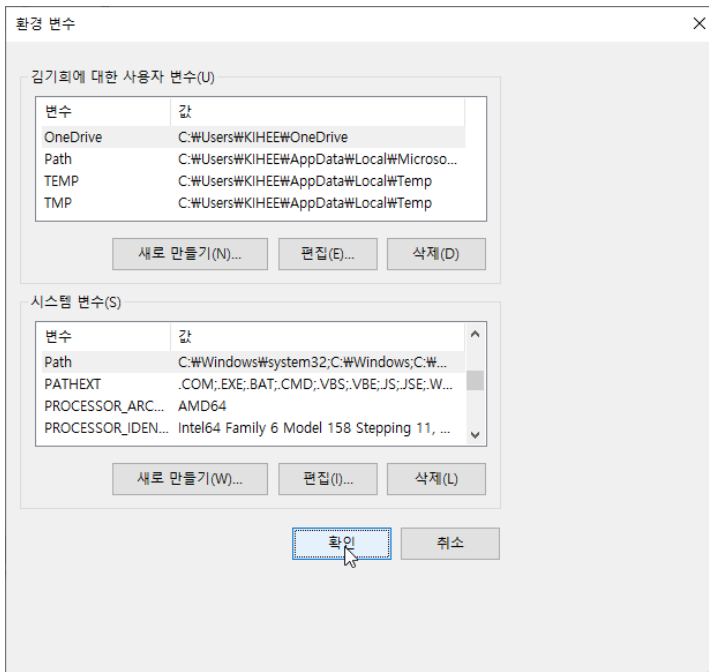
PATH 환경변수에 자바관련 실행파일 경로 등록을 위해 시스템 그룹의 "편집(E)..." 버튼을 클릭한다.



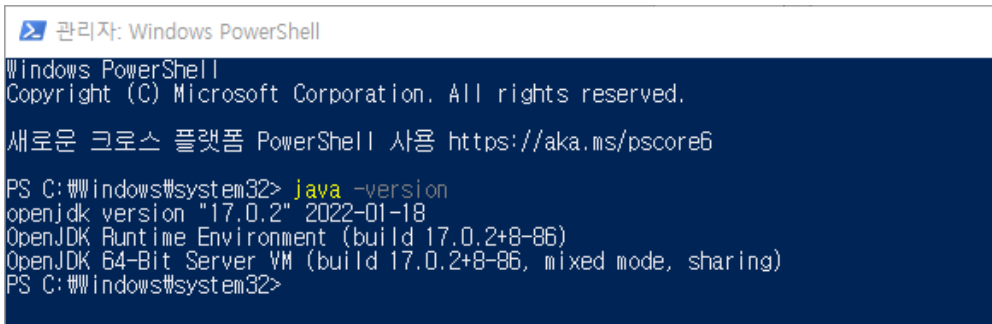
우측의 "새로 만들기(N)" 버튼을 클릭한 후 아래와 같이 입력한 후 "확인" 버튼을 클릭하여 PATH 환경변수의 편집을 완료한다.



환경변수 등록 및 편집이 완료되었다면 반드시 환경 변수 창의 "확인" 버튼을 클릭하여 환경 변수 창을 닫는다.



환경변수 설정 확인을 위해 Windows Key + x, a를 눌러 Windows PowerShell 프롬프트 창을 띄운 후 "java -version" 명령을 이용하여 Java Runtime 정보의 출력을 확인한다.



```
관리자: Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

새로운 크로스 플랫폼 PowerShell 사용 https://aka.ms/pscore6

PS C:\Windows\system32> java -version
openjdk version "17.0.2" 2022-01-18
OpenJDK Runtime Environment (build 17.0.2+8-86)
OpenJDK 64-Bit Server VM (build 17.0.2+8-86, mixed mode, sharing)
PS C:\Windows\system32>
```

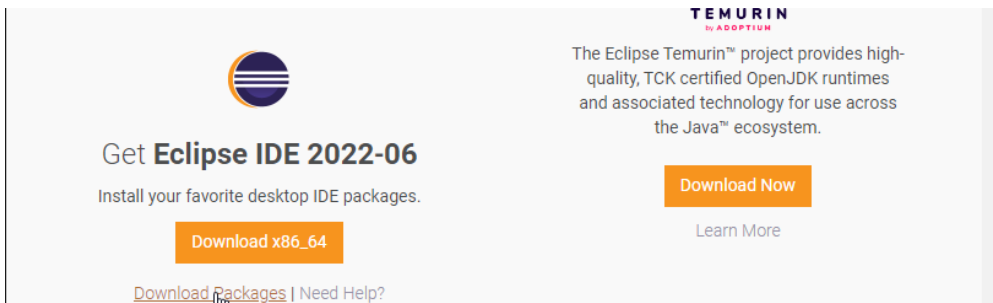
1.2 자바 개발 툴(이클립스) 설치하기

이클립스는 다양한 언어를 지원하는 프로그래밍 통합 개발 환경을 제공하는 툴이다. 이클립스는 자바로 작성되어 있으며, 자유소프트웨어 이면서도 막강한 기능을 제공하며, 자바만이 아닌 C/C++ 등 다양한 언어를 지원하는 프로그래밍 개발 툴이다.

이클립스는 아래의 주소에서 다운로드 한다.

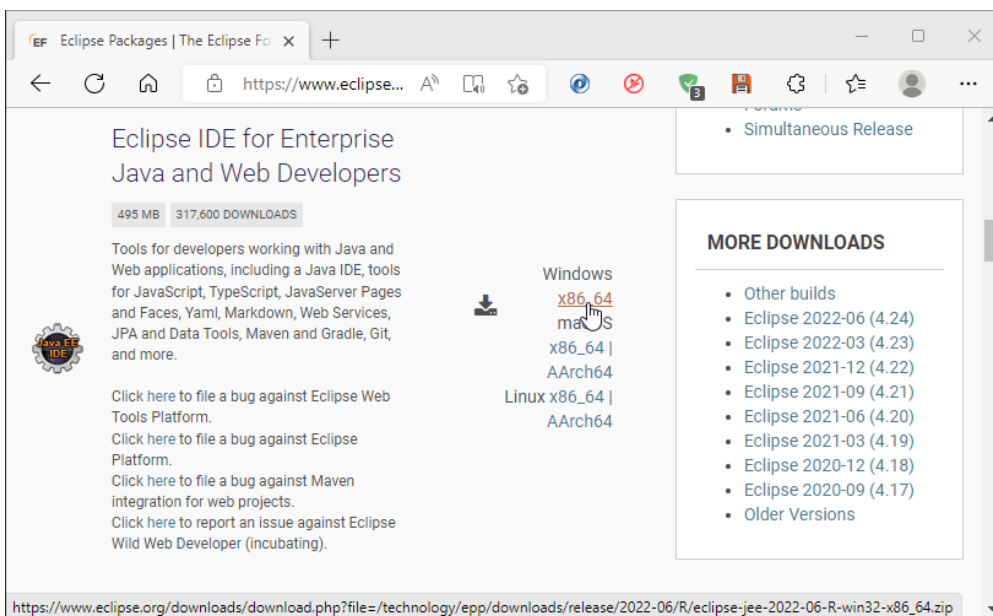
<http://www.eclipse.org/downloads/>

"Download Packages" 링크를 클릭한다.

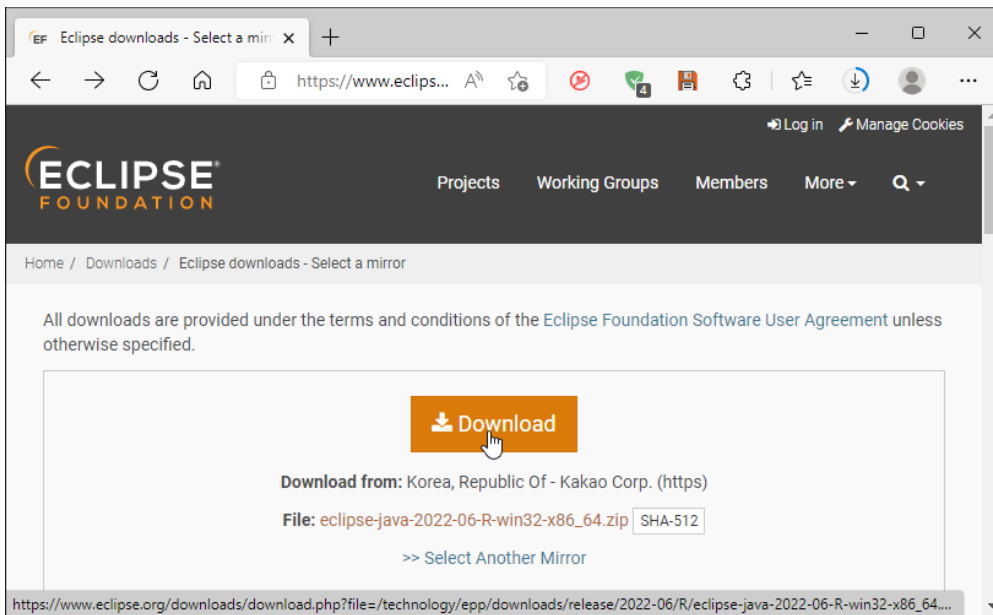


Java Application 개발을 위한 패키지는 "Eclipse IDE for Java Developers" 이지만 이후 웹 프로젝트 기반의 Spring MVC 수업도 진행될 것이므로 "Eclipse IDE for Enterprise Java and Web Developers" 버전을 다운로드 한다.

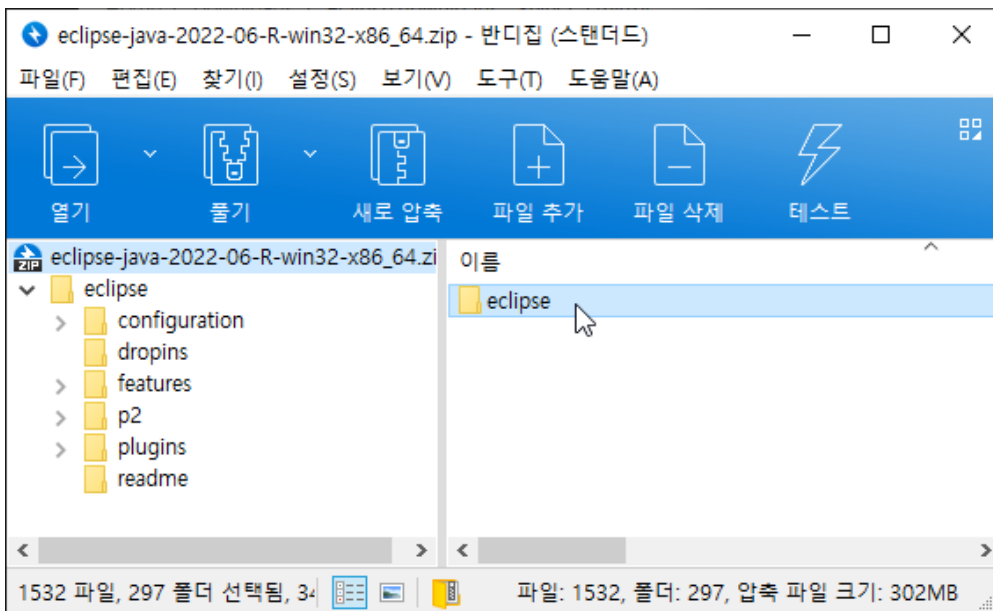
현재 사용중인 플랫폼에 맞는 링크를 클릭한다.



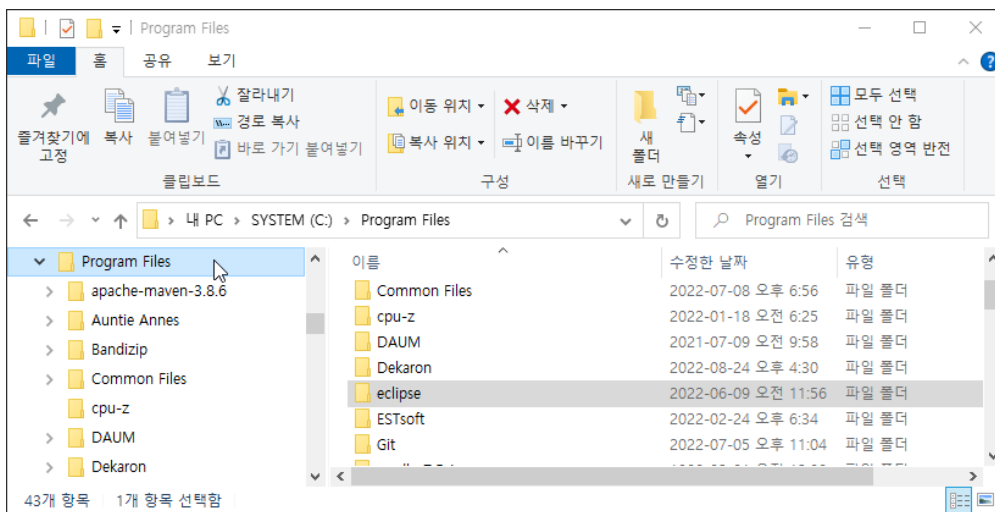
"Download" 버튼을 클릭하여 이클립스 IDE를 다운로드 한다.



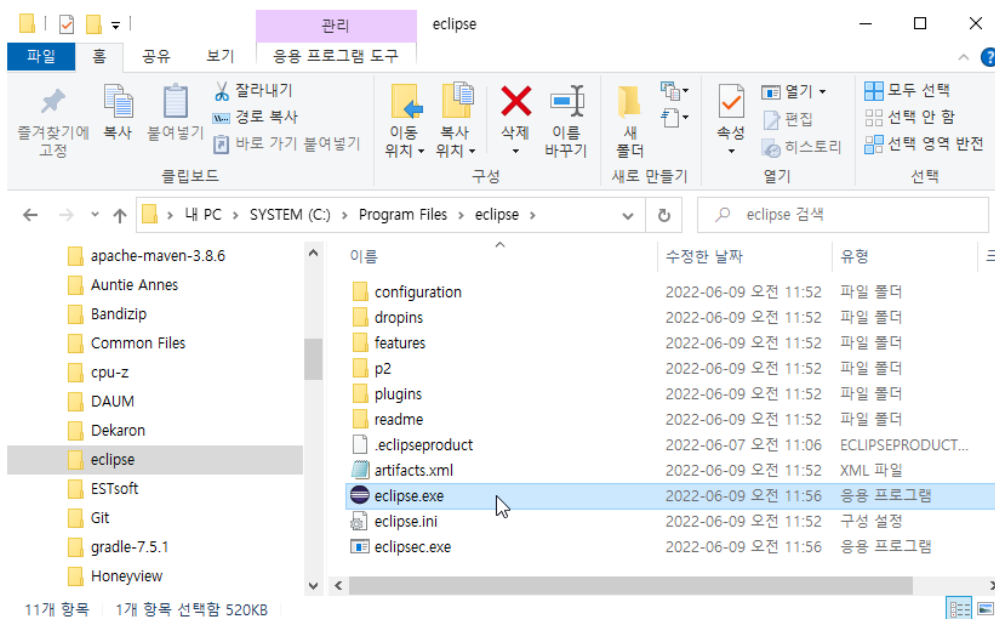
다운로드 파일을 압축 프로그램을 이용하여 열면 아래와 같이 eclipse 폴더를 확인할 수 있다.



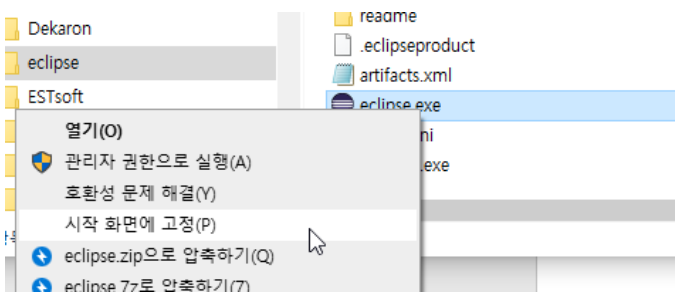
eclipse 폴더를 "C:\Program Files" 폴더에 드래그 앤 드롭하여 압축을 해제한다.



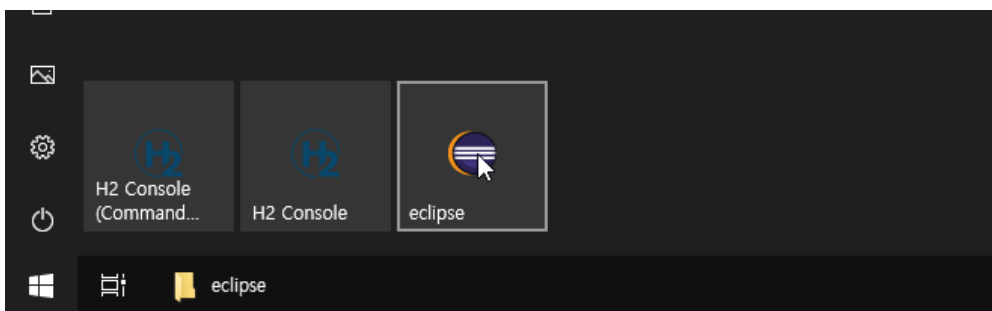
압축이 해제된 폴더 내에 eclipse.exe 파일이 이클립스 IDE 실행을 위한 파일이다.



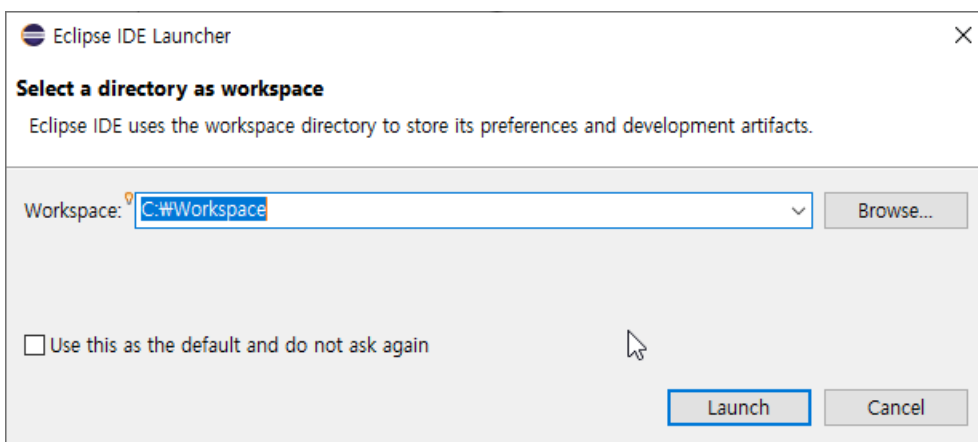
필수 작업은 아니지만 자주 사용될 것이므로 eclipse.exe 파일을 마우스 오른쪽 버튼으로 클릭하여 "시작 화면에 고정"을 선택하여 시작 화면에 바로 가기를 생성해 두자.



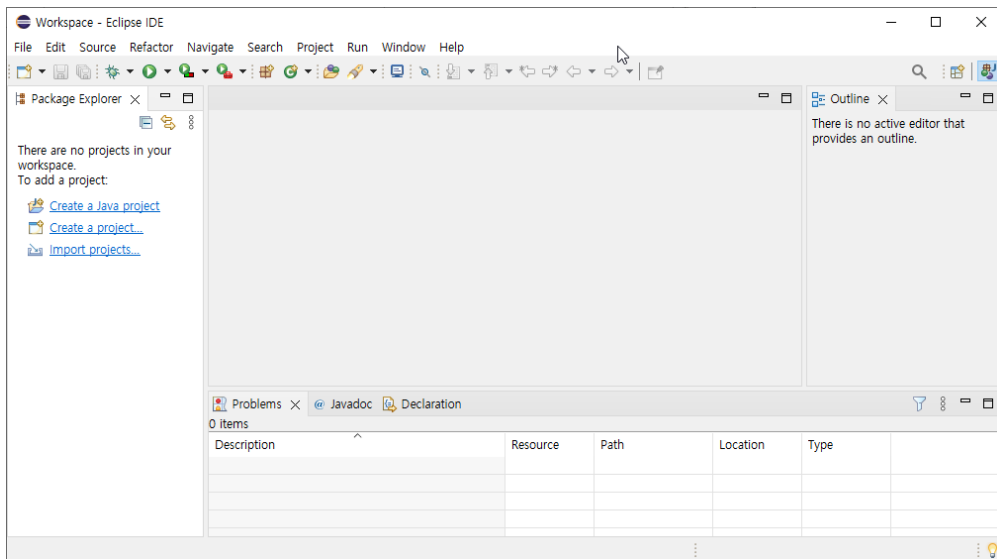
이후 시작 메뉴를 통해 이클립스를 실행할 수 있다.



이클립스 실행 아이콘을 클릭하여 이클립스를 실행한다. 이클립스 실행 시 작업 폴더 지정을 묻는 창에는 아래와 같이 작업 폴더로 사용할 경로를 입력하거나 "Browse..." 버튼을 통해 선택한 후 "Launch" 버튼을 클릭한다.



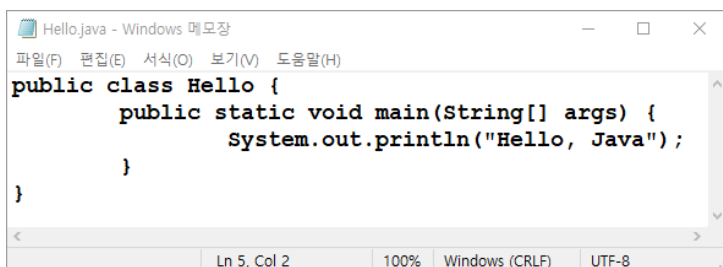
아래는 이클립스 IDE의 화면이다.



2. 자바 프로그램의 작동 원리

2.1 Hello Java 프로그램 작성하기

메모장을 이용하여 아래와 같은 자바 프로그램 코드를 작성한 후, C:\Workspace 폴더에 "Hello.java"라는 이름으로 저장한다.

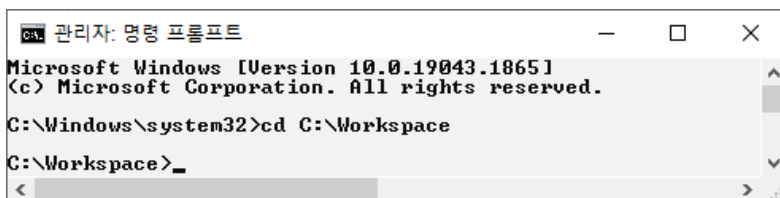


```
public class Hello {  
    public static void main(String[] args) {  
        System.out.println("Hello, Java");  
    }  
}
```

위의 프로그램 코드는 자바 언어로 작성된 코드로서 시스템이 실행하기에는 부적절 하다. 위의 코드를 시스템이 실행 가능하도록 하기 위해서는 시스템이 해석 가능한 코드로 변환을 하여야 하는데 이러한 과정을 컴파일이라고 한다.

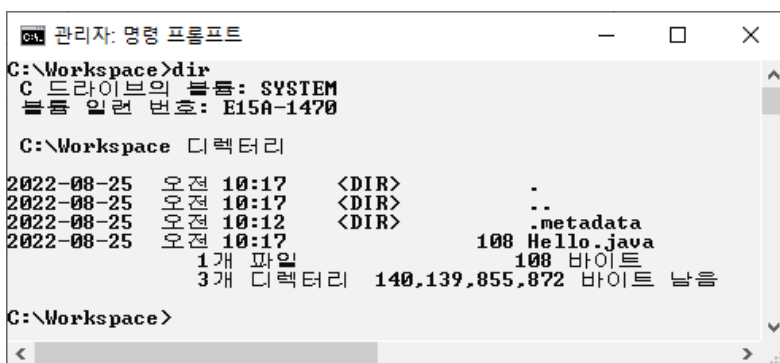
2.2 java 컴파일

컴파일을 위해 prompt 창을 띄운 후 Hello.java 파일이 위치한 C:\Workspace 디렉토리로 이동한다.



```
Microsoft Windows [Version 10.0.19043.1865]  
<C> Microsoft Corporation. All rights reserved.  
C:\Windows\system32>cd C:\Workspace  
C:\Workspace>
```

아래와 같이 dir 명령을 이용하여 현재 디렉토리 내의 파일 목록을 확인한다.



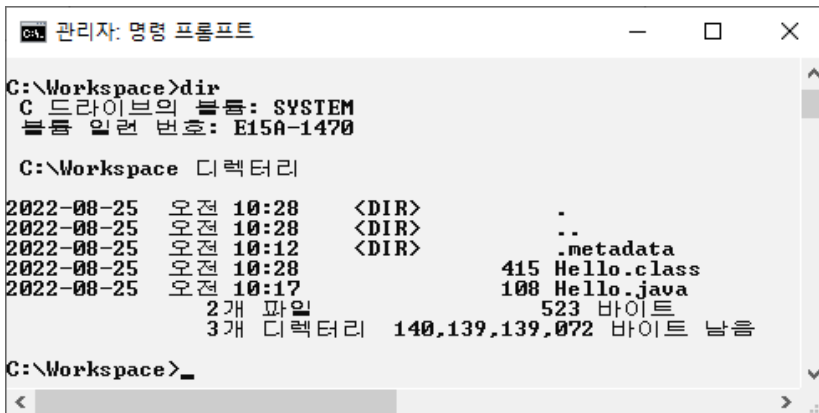
```
C:\Workspace>dir  
C 드라이브의 볼륨: SYSTEM  
볼륨 일련 번호: E15A-1470  
  
C:\Workspace 디렉터리  
2022-08-25 오전 10:17 <DIR> .  
2022-08-25 오전 10:17 <DIR> ..  
2022-08-25 오전 10:12 <DIR> .metadata  
2022-08-25 오전 10:17 108 Hello.java  
1개 파일 108 바이트  
3개 디렉터리 140,139,855,872 바이트 남음  
  
C:\Workspace>
```

아래의 명령을 이용하여 Hello.java 파일을 컴파일한다.



```
C:\Workspace>javac Hello.java
C:\Workspace>
```

아무런 메시지가 출력되지 않는다면 정상적으로 컴파일 된 것이다. 또한 아래의 명령을 이용해 확장자가 ".class"인 파일이 생성되었음을 확인할 수 있다. 이렇게 파일의 확장자가 ".class"인 파일을 가리켜 클래스 파일이라고 한다.



```
C:\Workspace>dir
C 드라이브의 볼륨: SYSTEM
볼륨 일련 번호: E15A-1470

C:\Workspace 디렉터리
2022-08-25 오전 10:28 <DIR> .
2022-08-25 오전 10:28 <DIR> ..
2022-08-25 오전 10:12 <DIR> .metadata
                415 Hello.class
                108 Hello.java
                2개 파일              523 바이트
                3개 디렉터리 140,139,139,072 바이트 남음

C:\Workspace>
```

2.3 자바 프로그램의 실행

생성된 클래스 파일은 자바 실행기를 이용하여 실행이 가능하다. 클래스 파일이 위치한 디렉토리로 이동한 후, 아래와 같은 명령을 이용하여 클래스 파일을 실행한다.



```
C:\Workspace>java Hello
Hello, Java
C:\Workspace>
```

프로그램이 정상적으로 실행되었다면 "Hello, Java"라는 메시지가 출력될 것이다.

최신 버전의 JDK는 자바 소스파일을 이용하여 실행이 가능하다. 하지만 이것은 Java Runtime이 소스파일을 컴파일하여 자바 바이트코드를 실행해 줄 뿐, 컴파일이 필요 없음을 의미하지는 않는다.



```
C:\Workspace>java Hello.java
Hello, Java
C:\Workspace>
```

2.4 자바 프로그램의 작동원리

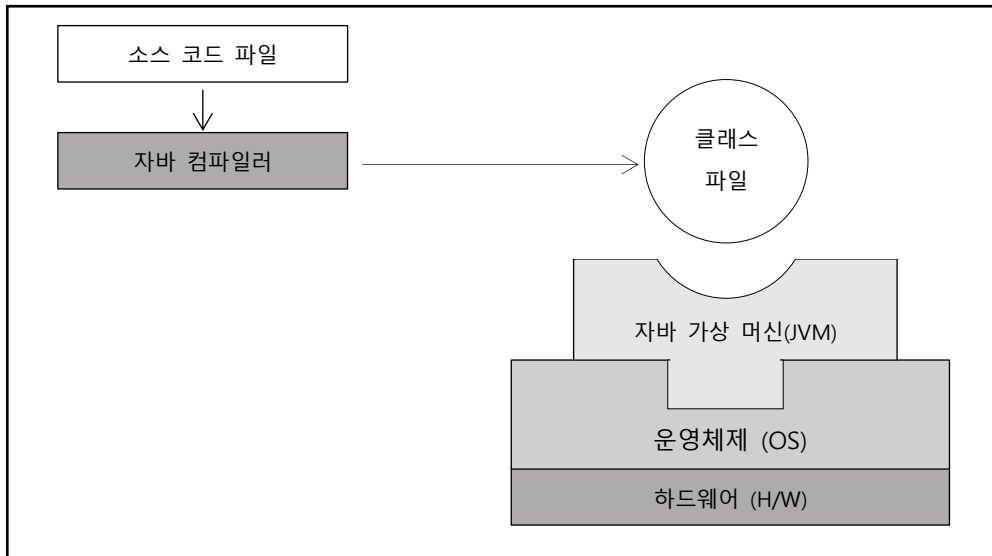
자바는 C나 C++ 또는 다른 프로그래밍 언어와는 달리 컴파일러에 의해서 컴파일 된 파일을 직접적으로 실행하는 것이 아니라 자바실행기(java.exe)를 이용하여 클래스 파일을 실행하게 된다. 이것은 C나 C++언어로 작성된 소스코드를 컴파일 하면 컴파일러는 해당 시스템에서 실행할 수 있는 기계어 코드를 만들어내지만 자바 컴파일러(javac.exe)는 자바 가상 머신(JVM)이 실행할 수 있는 기계어로 된 파일을 만들어내기 때문이다.

자바 가상 머신(JVM)은 JRE에 포함된 소프트웨어로서 자바 클래스 파일을 실제 시스템이 실행할 수 있는 기계어로 번역하는 역할을 한다. 즉 자바 클래스파일은 실제 시스템이 아닌 가상 머신에 의해 구동되도록 하여야 하는데 이때 사용되는 것이 자바실행기이며, 이 자바 실행기를 가동하는 실행파일이 "java.exe" 이다.

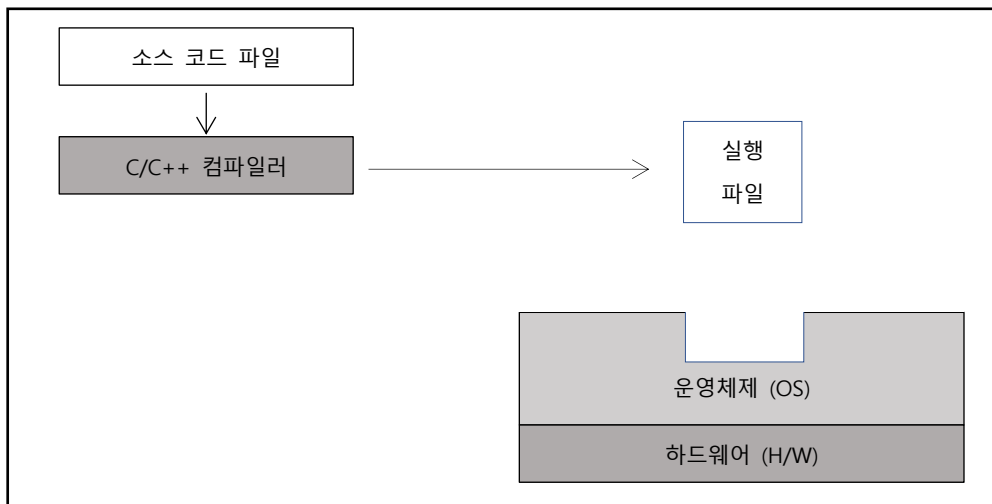
2.5 JVM의 이점

자바 언어는 JVM이라는 자바 가상 머신을 통해 클래스 파일을 실행하기 때문에 JVM이 설치되어 있는 시스템이라면 운영체제나 하드웨어에 상관없이 구동이 가능하게 된다. 즉 자바언어는 자바 가상 머신에 종속적이긴 하지만 실제 시스템의 운영체제나 하드웨어로 부터는 독립되어 있으므로 이식성이 매우 좋다.

반면 자바 클래스 파일을 실행하기 위해서는 실제 시스템에 반드시 자바 가상 머신이 설치되어 있어야 하므로 JVM은 시스템에 종속적이다.



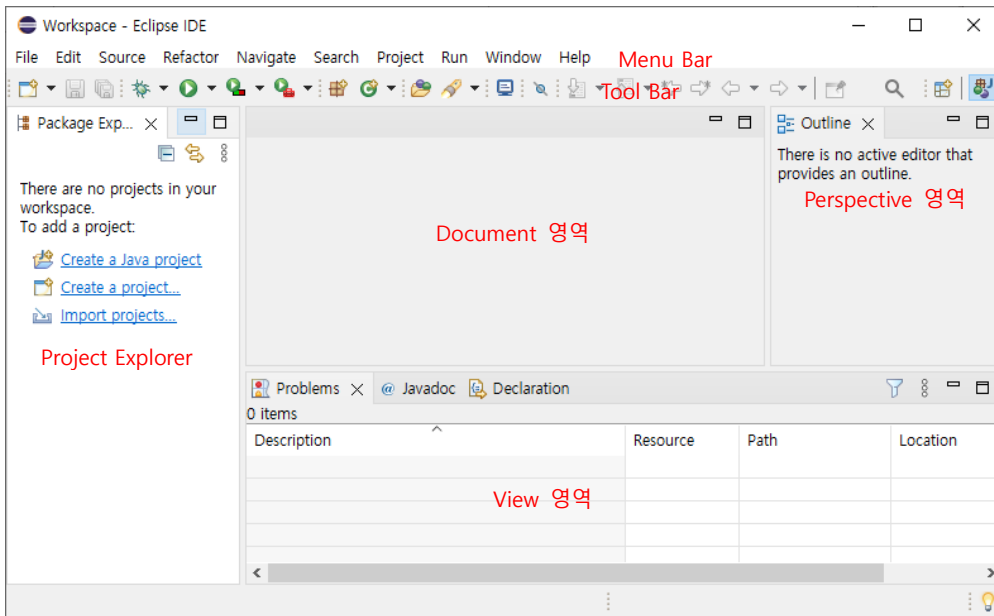
자바 프로그램의 컴파일과 실행과정



C/C++ 프로그램의 컴파일과 실행과정

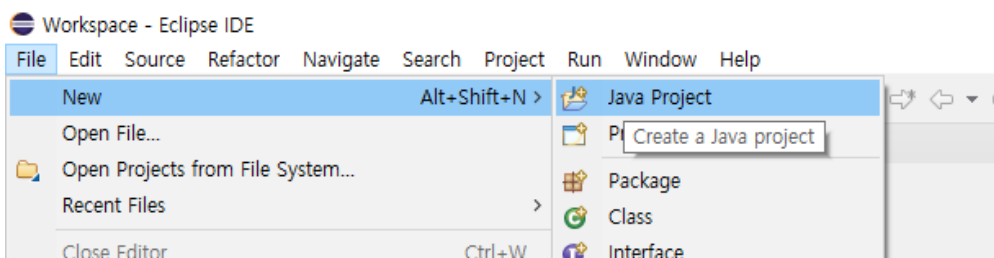
3. 이클립스 사용법

3.1 이클립스의 화면구성



3.2 프로젝트 추가하기

이클립스를 이용하여 자바 프로그램을 작성하기 위해서 가장 먼저 해야 할 것은 프로젝트의 구성이다. 프로젝트는 메뉴바의 File > New > Java Project 를 선택하거나, Project Explorer 영역에서 마우스 오른쪽쪽을 클릭한 후, 나타나는 팝업메뉴를 이용하여 추가할 수 있다.



우선 Project name 란에 프로젝트 이름을 입력한다. 프로젝트의 이름은 보통 작은 프로젝트 명.도메인 명을 반대로 기술하여 작성한다. 예를 들어 powerlinux 라는 회사의 도메인이 powerlinux.co.kr 이며, 이 회사의 hello라는 프로젝트라고 한다면 아래와 같이 프로젝트 이름을 지정하는 것이 관례이다.

New Java Project

Create a Java Project
Create a Java project in the workspace or in an external location.

Project name:

☒ Use default location
Location: [Browse...](#)

JRE

☒ Use an execution environment JRE: [Configure JREs...](#)

☐ Use a project specific JRE:

☐ Use default JRE 'jre' and workspace compiler preferences

Project layout

☐ Use project folder as root for sources and class files

☒ Create separate folders for sources and class files [Configure default...](#)

Working sets

☐ Add project to working sets [New...](#)

Working sets: [Select...](#)

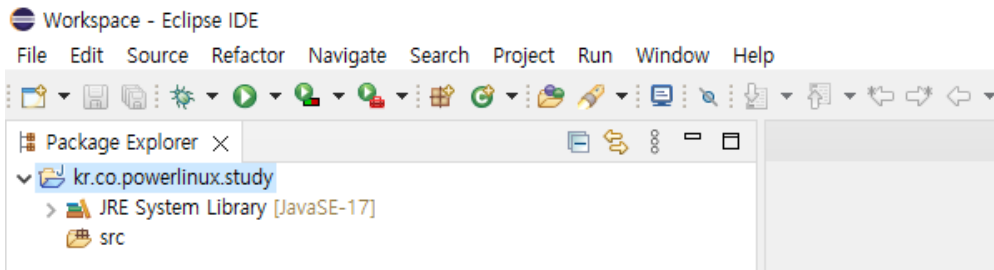
Module

☐ Create module-info.java file

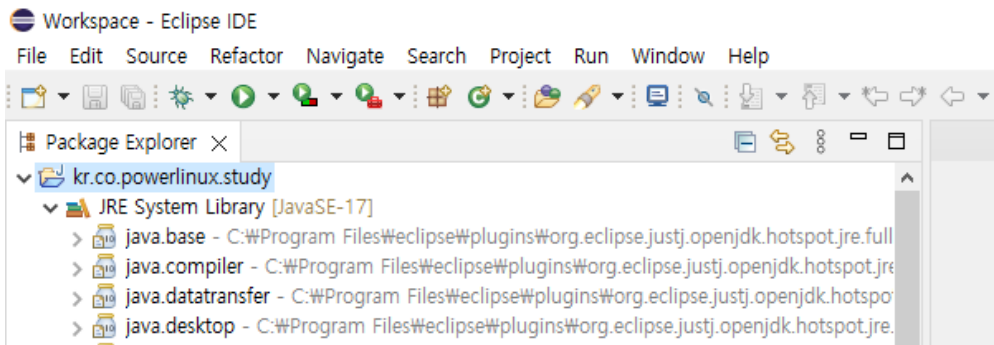
[?](#) [< Back](#) [Next >](#) [Finish](#) [Cancel](#)

프로젝트 명을 위와 같이 기술하는 이유는 거대한 하나의 프로젝트를 여러 개의 작은 프로젝트로 분할하여 팀별 작업을 한 후, 나중에 통합하였을 때, 이름 충돌과 같은 문제를 방지하기 위함이다. 이후의 예제에서는 간단한 프로젝트 명을 사용하도록 할 것이다. 프로젝트 명을 지정하였다면 "Finish" 버튼을 클릭하여 프로젝트 생성을 완료한다.

프로젝트가 생성이 되었다면 아래의 이미지와 같이 Package Explorer에 프로젝트가 등록되어 있을 것이다.



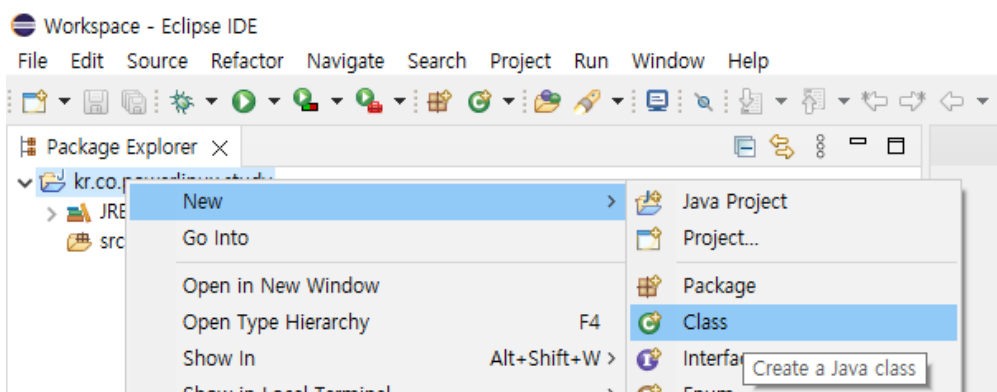
Project Explorer에 등록된 프로젝트 명을 더블 클릭하면 아래와 같이 "src"라는 디렉토리와 "JRE System Library" 항목이 나타나며 "JRE System Library" 항목을 더블 클릭하면 현재의 프로젝트에서 참조가 가능한 라이브러리 목록이 나타나게 된다.



3.3 시작프로그램 작성하기

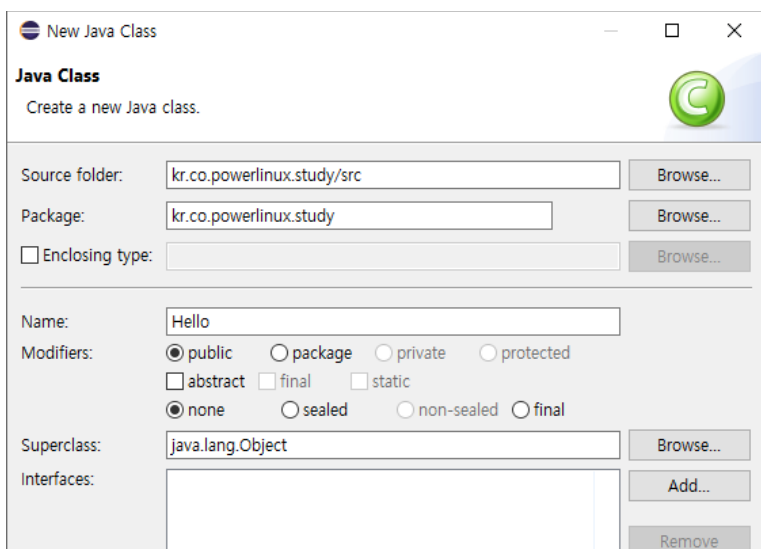
프로그래밍 언어에서의 라이브러리란? 소프트웨어를 만들 때 쓰이는 클래스나 서브루틴들의 집합을 말한다. 자바에서의 라이브러리란 클래스들의 집합이라고 할 수 있다. 이것은 이미 누군가가 자바 프로그래밍을 작성하고 컴파일 하여 다른 개발자가 참조(사용)할 수 있도록 제공하였음을 의미한다. 위의 라이브러리들은 이전에 설치한 JDK에 포함된 기본 라이브러리로 자바를 개발하여 제공하는 sun사에서 제공하는 라이브러리들이며, 개발자는 이 라이브러리를 참조하여 프로그램을 개발하게 된다.

새로운 클래스 파일을 작성하기 위해 메뉴바의 "File > New > Class"를 선택한다.

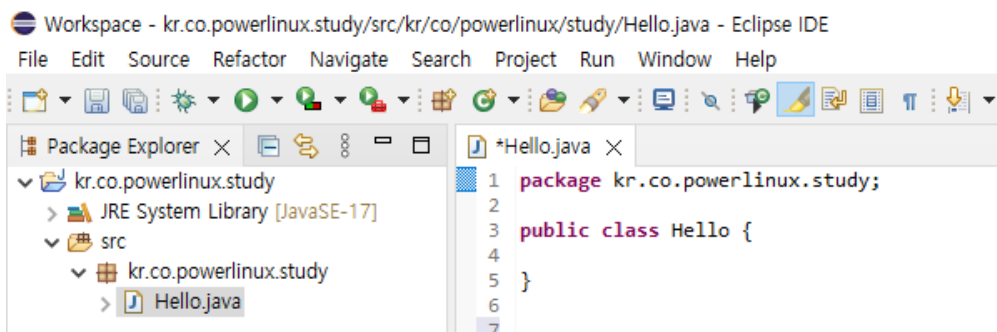


새로운 클래스파일을 추가할 때에는 반드시 Project Explorer에서 클래스를 추가할 프로젝트를 선택한 후, 추가하도록 한다. 물론 Project Explorer에서 클래스를 추가하고자 하는 프로젝트 항목을 마우스 오른쪽으로 클릭하여 추가할 수 도 있다.

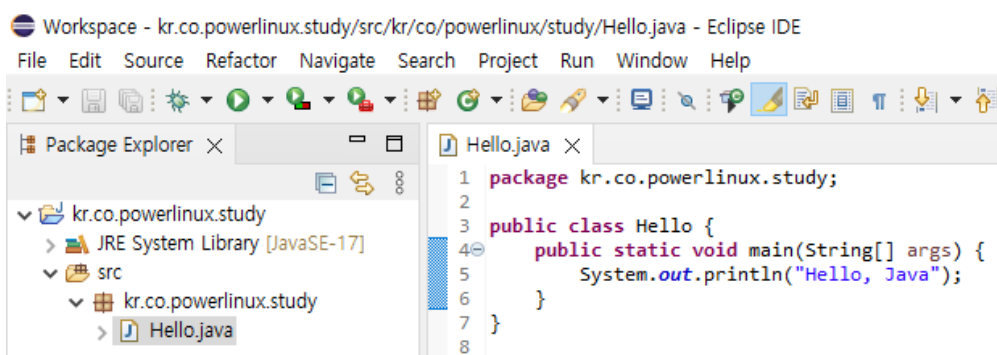
클래스 등록 창이 뜨면 아래와 같이 클래스 이름을 입력한 후, "Finish" 버튼을 클릭한다. 이때 등록하는 클래스이름의 첫 문자는 반드시 영문자이어야 하며, 대문자이어야 한다.



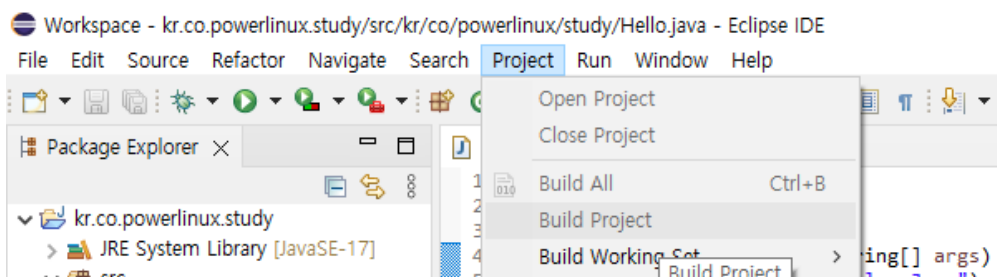
클래스 파일이 추가되면 아래와 같이 Document 영역에 새로 추가된 클래스 파일의 내용이 열린다. 그리고 이 Document 영역에서 코딩을 작성하게 된다.



Document 영역에 아래와 같이 간단한 프로그램 코드를 작성한 후, 파일을 저장한다.



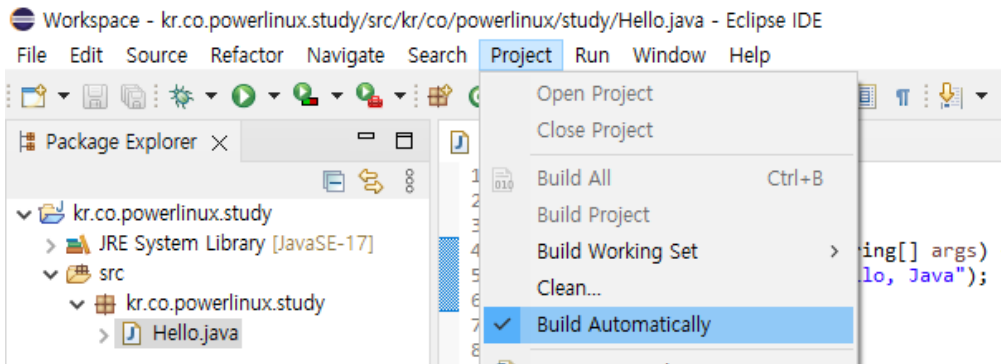
이클립스를 이용하여 프로그램을 개발할 경우, 프로그램 소스파일 하나하나를 개별적으로 컴파일 하는 것이 아니라 프로젝트내의 모든 자바 소스파일을 일괄적으로 컴파일 하게 된다. 이 과정을 가리켜 빌드(Build)라고 한다. 프로젝트의 빌드는 메뉴바의 "Project > Build ..." 메뉴를 이용하여 빌드를 하게 된다.



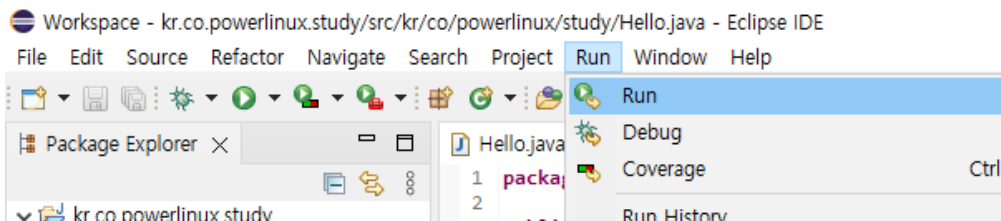
지금은 Build ... 메뉴가 활성화가 되어 있지 않은데, 이것은 이미 파일을 저장함과 동시에 자동적으로 빌드가 되었기 때문이다.

Build Automatically 메뉴가 체크되어 있는 경우, 파일을 저장할 때마다 빌드를 하므로 느린 시스템에서는 작업이 원활하지 않을 수 있으므로 되도록이면 자동 빌드 기능을 꺼 두고, 실행하기 전에 빌

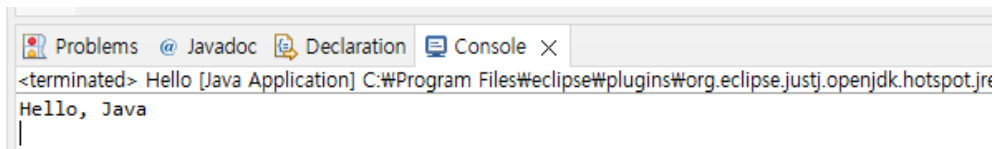
드를 하는 것이 좋다.



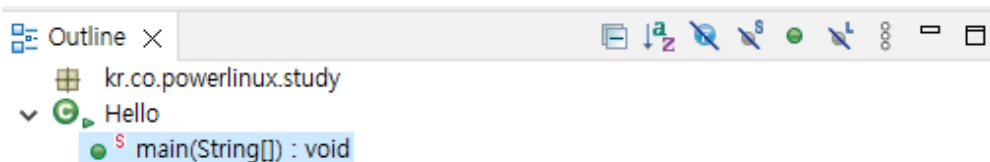
빌드가 완료되었다면 메뉴바의 "Run > Run" 메뉴를 선택하여 자바 프로그램을 실행할 수 있다.



프로젝트의 실행결과와는 아래와 같이 Console 뷰를 통해 확인할 수 있다.

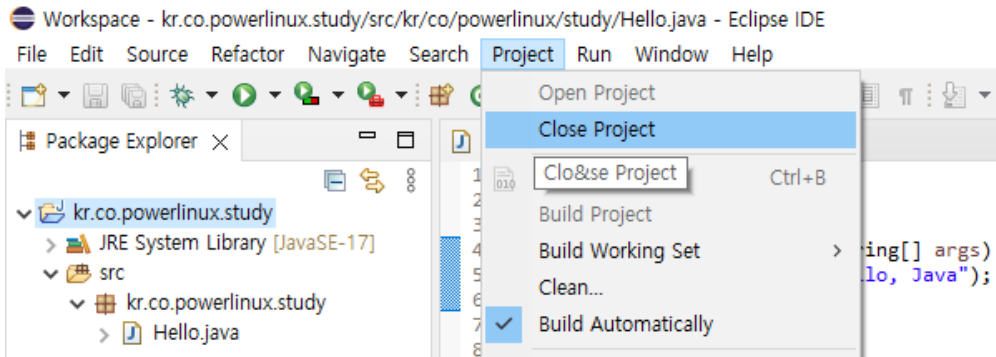


우측의 Outline Perspective에서는 현재 작업중인 프로젝트내의 클래스와 클래스멤버 등의 목록을 확인할 수 있으며, 그리고 이 목록에서 선택하여 빠른 이동이 가능하다.



3.4. 새로운 프로젝트 추가하기

새로운 프로젝트를 추가하는 방법은 위에서 알아본 새로운 프로젝트 작성하기 방법과 동일하다. 이때에는 기존의 프로젝트는 닫아 두는 것이 좋다. 위에서 설명한 Build Automatically 기능에 의해 파일을 저장할 때 마다 열려져 있는 프로젝트 또한 모두 빌드하기 때문이다.



4. 자바 프로그램 작성의 기초

4.1 자바 프로그램의 작성 규칙

- 자바의 클래스명은 반드시 대문자로 시작하여야 한다.
- 자바 소스 파일명은 파일에 포함된 클래스명과 동일한 이름이어야 한다.
- 자바 프로그램 코드는 대소문자를 구분한다.
- 자바 프로그램은 키워드와 식별자 그리고 `statement` 로 구성된다.
- `statement` 는 선언문, 대입문, 제어문이 있으며 제어문은 조건문, 반복문, 분기문 등으로 구성된다.

주석이란? 컴파일 시 제외되는 부분으로 실행에는 영향을 미치지 않는 구문을 말한다. 자바에서 주석을 작성하는 방법은 두 가지 방법이 있다.

// 단일라인 주석	// 이후의 라인의 끝까지의 문장을 주석으로 처리
/* ... */	/* 와 */ 사이의 모든 문장을 주석으로 처리

4.2 상수

상수란? 수식에서 변하지 않는 값을 말한다. 이와는 반대로 변하는 값을 가리켜 변수라고 한다. 자바 프로그래밍에서 사용 가능한 상수는 다음과 같은 종류가 있다.

구분	종류	설명	예
정수형	10진수		15
	8진수	접두어로 0을 사용	017
	16진수	접두어로 0x로 사용	0x0F
	long형	접미어로 L또는 l을 사용	15L
실수형	float	접미어로 F또는 f를 사용	3.14F
	double	부동소수점형	3.14
		지수형	0.314E+1
문자형	char	단일 따옴표로 감싸서 표현	'a'
		Unicode를 나타내는 제어문자	'\u0041'
문자열	String	이중 따옴표로 감싸서 표현	"Hello"

※ 자바에서 정수형 상수의 기본자료형은 int형이며, 실수형 상수의 기본자료형은 double 형이다.

자바에서의 문자열은 이중 따옴표(")를 감싸서 표현하며, 문자는 단일 따옴표(')를 감싸서 표현한다.

예) 문자열

```
String name = "KIHEE";
```

예) 문자

```
char c = 'A';
```

문자를 표현하는데 있어서 아래와 같이 ASCII 코드를 이용할 수 도 있다.

```
char c = 65;
```

또한 아래와 같이 8진수 표기법도 사용이 가능하다.

```
char c = '\101';
```

위의 '\8진수' 표기법은 0부터 255까지의 Unicode만을 표기할 수 있으며 다른 문자는 표기할 수 없

다. 하지만 아래와 같이 \u (소문자 u)를 쓰고 그 뒤에 4자리 16진수를 이용하여 255값 이상의 Unicode를 표기할 수 있다.

```
char c = '\u0041';
```

o 이스케이프 시퀀스(Escape Sequence)

이스케이프 시퀀스란? 콘솔에 문자를 출력하는데 있어서 그 표현이 곤란한 문자들을 표현하는 문자를 말한다. 예를 들어 라인 넘김 또는 커서의 위치를 이동시키거나 문자열 내에서 큰따옴표를 나타내는 등의 용도로 사용된다.

시퀀스	ASCII Code	의미
\0	0	null(널)
\b	8	백스페이스 (Backspace)
\t	9	탭 (tab)
\n	10	개행 (new line)
\r	13	캐리지 리턴 (carriage return)
\"	22	큰따옴표(")
\'	27	작은따옴표(')
\\	92	백슬래쉬(\)

4.3 변수

변수는 상수 값을 저장하는 공간이며, 이 공간을 가리키는 이름을 변수명이라고 한다. 즉 데이터가 저장되는 메모리 공간에 접근하는 방법을 제공하는 것이 변수이다. 상수 값을 저장하기 위해서는 우선 저장할 공간을 확보하여야 하는데 이 과정을 변수의 선언이라고 하며, 이때 해당 공간에 어떠한 상수 값이 저장되는지, 즉 변수에 저장되는 데이터의 자료형을 지정하여야만 한다.

변수는 로컬변수와 멤버변수가 있다. 로컬변수는 특정 메소드 내에서만 유효한 변수를 말하며, 멤버 변수는 클래스 구성요소로써 멤버필드라고도 하며 클래스 내의 모든 영역에서 참조 가능한 변수를 말한다.

멤버변수에 관한 내용은 차후 언급하므로 여기서는 지역변수에 대한 내용만을 다루도록 하겠다.

```
-----
public class SampleVariable {
    int var; // SampleVariable의 멤버변수

    public static void main(String[] ar) {
        int num = 3; // 로컬변수

        num = 100 + 200;
        System.out.println("num : " + num);
    }
}
-----
```

로컬변수는 사용하기 전에 반드시 선언하고 초기화 하여야 하며 접근제한자를 지정할 수 없다. 로컬 변수 선언의 형식은 다음과 같다.

형식) 자료형 식별자;
 자료형 식별자 = 초기값;

예) int age;
 int age = 19;

동일한 자료형의 변수는 콤마연산자(,)를 이용하여 한꺼번에 선언할 수 있다.

형식) 자료형 식별자1, 식별자2;

예) String last = "KIH", first = "KIM";

4.3.1 변수명 명명규칙

- 첫 글자는 '_', '\$', 영문 대소문자 (한글가능)
- 글자수에 제한 없음
- 공백문자 및 특수문자 사용불가
- 숫자는 첫 글자가 아닐 때 사용가능
- 예약어(키워드)는 변수명으로 사용불가

4.4 자료형

자료형이란? 데이터의 종류를 말한다. 자바의 자료형으로는 크게 기본자료형(Primitive Type)과 객체형(Non Primitive Type)이 있다. 객체형은 나중에 설명이 나오므로 지금은 기본자료형에 대해서만 설명하도록 하겠다.

○ 기본자료형 (Primitive Type)

구분	자료형	크기	표현범위
정수형	byte	1	-128 ~ 127
	short	2	-32768 ~ 32767
	int	4	-2147483648 ~ 2147483647
	long	8	-9223372036854775808 ~ 9223372036854775807
실수형	float	4	1.4E(-45승) ~ 3.402923E(38승)
	double	8	4.9E(-324승) ~ 1.8E(308승)
논리형	boolean	1	true 또는 false
문자형	char	2	0 ~ 65535 (Unicode 코드 값)

○ 객체참조형 (Non Primitive Type)

구분	크기	저장 값
배열형	4	배열의 시작주소
클래스형	4	객체의 주소

4.4.1 Promotion과 Casting

프로모션(Promotion)과 캐스팅(Casting)은 자료형의 변환을 말한다.

◦ 프로모션 (Promotion)

작은 자료형의 값이 보다 큰 자료형으로 변환되는 것 으로서 자료의 손실이 발생하지 않으며 컴파일러에 의해 자동으로 형변환이 이뤄진다.

◦ 캐스팅 (Casting)

큰 자료형의 값이 보다 작은 자료형의 값으로 변환되는 것으로써 자료의 손실이 발생할 수 있으므로 컴파일러에 의해 자동으로 형변환이 이뤄지지 않는다. 즉 데이터의 손실을 고려하여 캐스팅 연산자를 이용하여 명시적으로 형변환이 이뤄지는 것을 말한다. 이러한 데이터의 손실은 자료형의 변환 시 오버플로우(Overflow)나 언더플로우(Underflow) 또는 반올림, 소수점 잘림 등의 현상으로 인해 발생된다.

※ 자료의 형변환이 발생하는 시점

- 1) 서로 다른 자료형 간의 연산 시 발생
- 2) int형보다 작은 정수형 간의 연산 시 발생 (int형으로 형 변환)

아래의 예제에서는 int형간의 연산이므로 형변환이 이뤄지지 않는다. 그러므로 결과 역시 int형이 된다.

```
-----
public class CastingTest {
    public static void main(String[] ar) {
        int i = 10;
        int f = 3;

        System.out.println(i / f);
    }
}
-----
```

하지만 아래의 예제는 int형과 float형간의 연산이므로 int형이 float형으로 형변환이 이뤄진 후, 연산이 수행이 되므로 결과 또한 float형이 된다.

```
-----
public class CastingTest {
    public static void main(String[] ar) {
        int i = 10;
        float f = 3.0f;

        System.out.println(i / f);
    }
}
-----
```

5. System 클래스와 기본 Console 출력

5.1 System 클래스와 기본 출력 처리

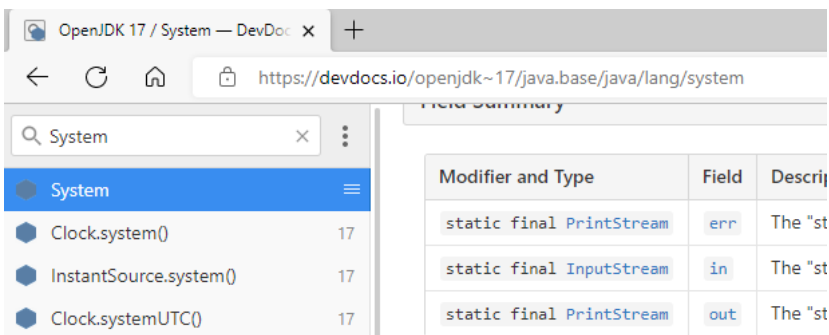
◦ System class

System 클래스는 실행 환경과 관련된 속성과 메서드를 제공하는 클래스로써 모든 멤버가 static 이므로 객체의 생성 없이 사용된다. System 클래스의 클래스 변수 in과 out은 입출력(java.io) 패키지의 InputStream 클래스와 PrintStream 클래스의 인스턴스이다. 그러므로 System.in은 InputStream 클래스의 메서드를, 그리고 System.out은 PrintStream 클래스의 메서드를 사용할 수 있다.

아래의 이미지는 System 클래스에 대한 API Document이며 아래의 URL을 통해 열람이 가능하다.

<https://devdocs.io/openjdk~17>

아래의 문서 내용에서 System 클래스는 err, in, out 세 가지의 static 필드가 존재하는 것을 알 수 있으며 각각의 필드에 대한 타입 또한 알 수 있다.



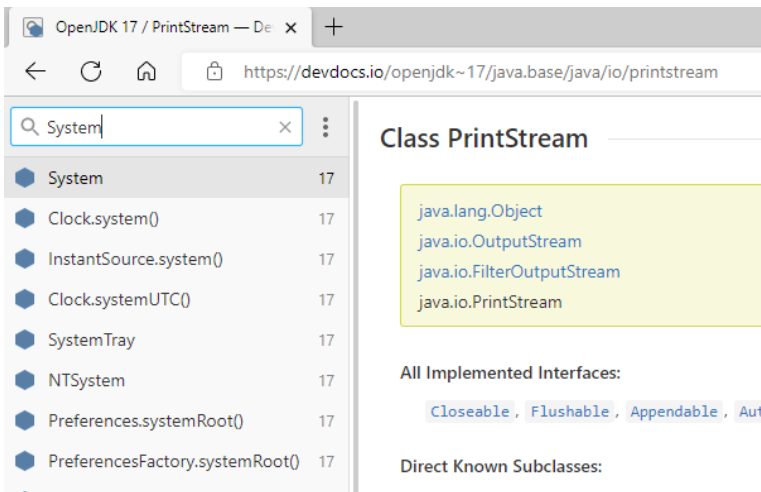
위의 이미지에서 err과 out 필드는 PrintStream 클래스 타입임을 알 수 있다.

PrintStream 클래스가 어느 패키지에 포함되어 있는지 알고 싶다면 PrintStream 링크 위에 마우스 커서를 올려보자. 아래와 같이 "class in java.io"라는 내용의 풍선도움말이 나타날 것이다.

PrintStream은 java.io 라는 패키지에 포함되어 있음을 알 수 있다.

Modifier and Type	Field	Description
static final PrintStream	err	The "standard" error output stream.
static final InputStream	in	The "standard" input stream.
static final PrintStream	out	The "standard" output stream.

만약 `PrintStream`에 대한 API Document를 참조하고자 한다면 `PrintStream` 링크를 클릭한다.



`PrintStream` 클래스의 API Document를 보면 Method Summary 항목에 많은 메서드가 존재함을 확인할 수 있다.

Method Summary

Modifier and Type	Method	Description
<code>PrintStream</code>	<code>append(char c)</code>	Appends the specified character to this stream.
<code>PrintStream</code>	<code>append(CharSequence csq)</code>	Appends the specified character sequence to this stream.
<code>PrintStream</code>	<code>append(CharSequence csq, int start, int end)</code>	Appends a subsequence of the specified character sequence to this stream.

이중에서 이번 시간에 알아볼 것은 아래의 메서드들이다.

- `write()`
- `print()`
- `println()`
- `printf()`

○ `write()` 메서드

`write()` 메서드는 주어진 값에 해당하는 ASCII 문자를 출력한다. 즉 `write()` 메서드는 1바이트 단위로 출력하는 메서드이다. `write()` 메서드는 auto flush 기능을 지원하지 않는다.

`flush`란? 버퍼상에 저장된 데이터를 프로그램으로 입력 받거나, 콘솔(파일)로 출력하는 기능을 말한다. 프로그램에서의 데이터의 흐름은 크게 입력과 출력이 있다. 이때 데이터의 흐름이 콘솔이나 파

일로 부터 프로그램으로 흐르는 것을 입력이라 하며, 프로그램으로 부터 콘솔이나 파일로 흐르는 것을 출력이라 한다.

```

프로그램 <----- 입력 -----> 콘솔/파일
프로그램 -----> 출력 -----> 콘솔/파일

```

이때 입력과 출력의 흐름을 스트림(Stream)이라고 하여 입력 스트림, 출력 스트림이라고 부른다. 이러한 입력 및 출력 스트림은 프로그램과 콘솔 및 파일간의 데이터 흐름을 나타내는데 이때 프로그램과 콘솔 및 파일 사이에 입출력 데이터를 저장해두는 메모리가 존재하며 이를 버퍼라고 부른다. 버퍼가 존재하는 이유는 보다 효율적인 데이터 입출력을 위해서 이다. 예를 들어 숫자나 영문자만을 입출력 한다면 1바이트 단위로 입출력이 가능하지만 한글이나 한자와 같은 유니코드(Unicode)로 나타내는 문자는 하나의 문자를 입력하거나 출력하기 위해서는 2바이트 단위의 입출력이 이뤄져야 한다.

버퍼는 입력과 출력 시 입력되는 데이터를 미리 저장해 두었다가 어느 시점에서 필요한 만큼의 데이터를 가져올 수 있도록 해준다. 또한 프로그램에서의 입출력은 정확히는 프로그램과 콘솔 및 파일간의 입출력이 아닌 프로그램과 입출력 버퍼 간의 입출력이라고 봐야 한다.

write()메서드는 인수로 주어진 ASCII 코드 값에 해당하는 문자를 출력하는데 이때 출력데이터는 콘솔이 아닌 출력 버퍼에 저장되어진다. 그러므로 출력 버퍼의 내용을 밀어 내야만이 콘솔에 데이터가 출력되는데 이처럼 버퍼의 데이터를 밀어내는 것을 가리켜 flush라고 한다.

```

-----
import java.io.IOException;

public class IOStream {
    public static void main(String[] ar) throws IOException {
        byte[] name = new byte[]{'H', 'e', 'l', 'l', 'o'};
        System.out.write(name);
        System.out.flush();
    }
}
-----

```

◦ print(), println() 메서드

print()메서드와 println()메서드는 인수로 주어진 데이터를 출력한다. 두 메서드의 차이점은 print() 메서드는 인수로 주어진 데이터를 출력한 후 개행을 하지 않지만 println() 메서드는 주어진 인수 뒤에 개행 문자를 추가하여 출력하게 된다.

print() 메서드와 println() 메서드는 write() 메서드와는 달리 auto flush 기능이 지원되므로 출력을 위해 별도로 flush() 메서드를 이용하지 않아도 된다.

◦ printf() 메서드

printf() 메서드는 C언어의 printf() 함수와 동일한 형식의 출력 메서드이다. printf() 메서드의 사용방법은 첫 번째 인수로 출력형식을 지정하며 두 번째 인수부터는 출력형식에서 사용된 서식문자에 대응하는 데이터를 지정한다.

```
System.out.printf("출력형식", 데이터1, 데이터2, ...);
```

출력형식은 서식문자를 포함하는데 형식문자는 많은 종류가 있으나 아래의 서식문자 만큼은 외워 두도록 하자.

%c	문자출력
%d	10진수 출력
%f	실수 출력
%s	문자열 출력

6. 자바의 연산자

6.1 연산자의 개념

피연산자(상수나 상수를 저장하고 있는 변수)들 사이의 계산방식을 특정한 기호로 표시한 것을 말한다.

연산자의 우선순위는 다음과 같다.

- 1) 최우선 연산자
- 2) 단항 연산자
- 3) 산술 연산자
- 4) 쉬프트 연산자
- 5) 비트 연산자
- 6) 논리 연산자
- 7) 삼항 연산자
- 8) 배정/대입 연산자
- 9) 순차 연산자

○ 자바에서 제공하는 연산자

연산기호	결합방향	우선순위
[], ., (expr)	→	1
expr++, expr--	←	2
++expr, --expr, +expr, -expr, ~, !, (type)	←	3
*, /, %	→	4
+, -	→	5
<<, >>, >>>	→	6
<, >, <=, >=, instanceof	→	7
==, !=	→	8
&	→	9
^	→	10
	→	11
&&	→	12
	→	13
? expr : expr	←	14
=, +=, -=, *=, /=, %=, ^=, =, <<=, >>=, >>>=	←	15

6.2 최우선 연산자

[], ., ()

◦ . (객체참조 연산자)

객체참조 연산자는 객체에 포함된 멤버를 참조하기 위해 사용되어지는 연산자이다. 아래의 예제는 System 클래스내의 out 멤버의 println() 메서드를 호출함을 나타낸다.

```
System.out.println("Test");
```

즉 위의 구문은 System 클래스내의 out 멤버의 println() 메서드를 "Test"라는 인자를 전달하여 호출한다. 연산자를 사용함에 있어서 연산의 우선순위가 중요하지만 연산의 우선순위만큼 중요한 것이 결합방향이다. 예를 들어 동일한 연산자가 하나의 수식에 나열되었을 때 연산 되는 순서는 결합방향으로 수행되기 때문이다.

◦ [] (배열 참조 연산자)

배열 참조 연산자는 배열의 원소를 참조하는데 사용되는 연산자로 [] 내에는 배열원소의 위치 값을 지정하게 되는데 이를 첨자라고 한다. 아래는 int형 데이터 10개를 저장할 수 있는 배열을 선언하는 예이다.

```
int[] arr = new int[10];
```

위와 같은 배열을 참조하는데 있어서 첫 번째 원소의 참조는 아래와 같이 참조한다.

```
arr[0] = 10;
```

배열 원소의 위치 값(첨자)은 0부터 시작한다.

◦ () (괄호 연산자)

괄호 연산자는 연산의 우선순위를 조절할 수 있는 연산자이다. 괄호로 감싸여진 연산 식은 그 어느 연산 식보다 우선적으로 연산이 이뤄진다. 예를 들어 $2 * 5 + 10$ 이라는 연산 식에서 * 연산자가 + 연산자보다 우선순위가 높으므로 연산의 결과값이 20이지만 () 연산자를 이용하여 연산의 우선순위를 바꿈으로써 다른 결과가 된다.

```
2 * 5 + 10    : 20    ► 10 + 10
2 * (5 + 10) : 30    ► 2 * 15
```

6.3 단항 연산자

논리부정	: !
비트부정	: ~
부호	: +, -
증감(전위)	: ++, --
형변환	: (자료형)

◦ ++, -- (증감 연산자)

++ 나 -- 연산자는 전위연산자와 후위연산자가 있다. 전위연산자란 피연산자의 앞쪽에 연산자가 나타나며, 후위연산자는 피연산자의 뒤에 연산자가 나타난다.

++expr, --expr	: 전위 증감 연산자
expr++, expr--	: 후위 증감 연산자

전위 연산자와 후위 연산자는 피연산자의 값을 1씩 증가시키거나 1씩 감소시킨다. 이때 전위 연산자는 피연산자의 값을 먼저 1 증감시킨 후 사용되어지며, 후위 연산자는 피연산자의 값을 우선 사용한 후 1 증감시키게 된다.

```
-----
public class Operator {
    public static void main(String[] ar) {
        int a = 1;
        int b = 1;
        int c = ++a + ++b;

        System.out.printf("a, b, c : %d, %d, %d", a, b, c);
    }
}
-----
```


◦ +, - (부호)

부호 연산자는 피연산자가 양수인가, 음수인가를 나타내는 연산자이다. 부호연산자를 사용하지 않는 경우 피연산자는 양수이다.

◦ ~ (비트부정)

비트부정은 피연산자의 데이터 값을 비트단위로 부정을 한다.

정수 10의 비트	: 00000000 00000000 00000000 00001010
정수 10의 비트부정	: 11111111 11111111 11111111 11110101

 정수 10의 비트부정 값은 몇일까요?

◦ ! (논리부정)

! 연산자는 논리값을 부정하는 연산자로서 피연산자의 논리값이 true 인 경우 결과는 false가 되며, 피연산자의 논리값이 false 인 경우 true가 된다.

◦ (자료형) (형변환 연산자)

형 변환 연산자는 Casting 연산자라고 하며 피 연산자의 자료형을 변환하는 연산자이다. 컴퓨터가 연산을 수행함에 있어서 정수형 데이터 간의 연산은 결과 역시 정수형 결과가 얻어진다. 아래의 예제에서 변수 op1과 op2는 int형 변수이므로 연산식의 결과 또한 int형이 된다. 그러므로 변수 result의 결과값은 3.0이 된다.

```
-----  
public class Operator {  
    public static void main(String[] ar) {  
        int op1 = 10;  
        int op2 = 3;  
        float result = op1 / op2;  
  
        System.out.println("result : " + result);  
    }  
}
```

만약 보다 정확한 데이터 연산이 필요하다면 아래와 같이 Casting 연산자를 이용하여 피연산자의 자료형을 바꾸어 주어야 한다. 아래의 예제에서는 op2에 대하여 Casting 연산자를 이용하여 float 형으로 형변환을 함으로써 int형과 float형간의 연산이 되어 결과는 float형이 된다.

서로 다른 두 자료형의 피연산자들 사이의 연산 식에서, 피연산자들은 동일한 자료형으로 일치시킨 후 연산이 실행된다. 즉 int형 / float형의 연산 식에서는 int형이 float형으로 형변환이 이루어진 후 연산이 수행되게 된다.

```
-----  
public class Operator {  
    public static void main(String[] ar) {  
        int op1 = 10;  
        int op2 = 3;  
        float result = op1 / (float)op2;  
  
        System.out.println("result : " + result);  
    }  
}
```

6.4 산술 연산자와 쉬프트 연산자

◦ *, /, %, +, - (산술 연산자)

산술 연산자를 사용함에 있어서 산술 연산자 사이에도 우선순위가 존재한다.

높은 순위 : *, /, %

낮은 순위 : +, -

즉 산술연산자중에서도 *, /, % 연산자가 우선적으로 수행되고 난 후, +, - 연산식이 수행된다. *, /, % 연산자는 우선순위가 동일하다. 이 때에는 위의 표에 명시된 연산식의 결합방향을 참조하도록 하자.

/ 연산을 수행함에 있어 정수형 피연산자 사이의 연산의 결과값은 정수형이 된다. 예를 들어 $10 / 3$ 의 결과값은 10에서 3을 나눈 몫과 나머지 값 중에서 몫 값만을 취하여 3이 된다.

% 연산은 모듈러스 연산자라고 하며 좌측 피연산자의 값을 우측 피연산자로 나눈 몫과 나머지 값 중, 나머지 값만을 취하게 된다.

◦ <<, >>, >>> 쉬프트 연산자

쉬프트 연산자는 좌측 피연산자의 데이터 값을 우측 피연산자의 값만큼 비트단위로 이동을 시킨다.

<< (left shift) 연산자는 좌측 피연산자의 데이터 값을 우측 피연산자의 값만큼 좌측으로 비트 이동을 시킨다. 이때 밀려 들어오는 우측 비트의 값은 0으로 채워지며, 밀려나가는 좌측 비트의 값은 소멸된다.

```
int a = 3;
int result = a << 2;    ▶ result의 값은  $3 * 2^2$ 
```

>> (right shift) 연산자는 좌측 피연산자의 데이터 값을 우측 피연산자의 값만큼 우측으로 비트 이동을 시킨다. 이때 밀려 들어오는 좌측 비트의 값은 부호비트(MSB)로 채워지며 밀려나가는 우측 비트는 소멸된다.

```
int a = 24;
int result = a >> 2;    ▶ result의 값은  $24 / 2^2$ 
```

>>> (unsigned right shift) 연산자는 >> 연산자와 같으나 >> 연산자의 차이점은 >> 연산자는 우측에 채워지는 비트가 첫 번째 비트 값(부호값)이지만 >>> 연산자에서는 무조건 0값으로 채워진다.

>>> 연산자는 데이터 중 특정 위치의 값을 추출하고자 할 때 주로 사용되어진다. 예를 들어 int형 변수의 데이터 값 중 앞쪽 2바이트와 뒤쪽 2바이트 값을 추출하고자 하는 경우, 즉 앞쪽의 데이터 값이 무조건 0이 되어야 하는 경우에 사용되어진다.

아래의 예제는 변수 a의 데이터 중 후미 2바이트 값만을 취하는 코드이다. 이때 >>> 연산자가 아닌

>> 연산자를 사용하면 F0C9가 아닌 FFFF0C9의 결과값이 출력되게 된다. 하지만 >>> 연산자를 이용하면 채워지는 비트를 0으로 채워 0000F0C9값을 출력하게 된다.

```
-----  
public class Operator {  
    public static void main(String[] ar) {  
        int a = 0xEAE3F0C9;  
  
        System.out.printf("%x \n", a);  
        System.out.printf("%x \n", a<<16);  
        System.out.printf("%x \n", a>>16);  
        System.out.printf("%x \n", a>>>16);  
    }  
}
```

6.5 관계, 비트, 논리 연산자

○ 비교 연산자

비교관계 : >, <, <=, >=, instanceof
 항등관계 : ==, !=

○ 비트단위 논리 연산자

& : 논리곱 (and)
 | : 논리합 (or)
 ^ : 배타적 논리합 (exclusive or)

○ 논리 연산자

&& : 논리곱
 || : 논리합

○ >, <, >=, <=, instanceof, ==, != (비교 연산자)

비교연산자는 두 피연산자의 값이 큰가, 작은가 또는 같은가 등의 연산을 수행하며, 그 결과값은 부울 값이 된다. 비교연산자의 우선 순위는 비교관계 연산자의 우선순위가 항등관계 연산자의 우선순위보다 높으며 연산의 결과값은 true 또는 false를 나타내는 boolean형값이 된다.

우선순위 높음 : >, <, >=, <=, instanceof
 우선순위 낮음 : ==, !=

○ instanceof (객체형 비교)

instanceof 연산자는 피연산자는 단항 연산자는 아니다. instanceof 연산자의 좌측에는 객체의 인스턴스가 명시되며, 우측에는 자료형이 명시된다. 즉 instanceof 연산자는 좌측에 명시된 객체가 우측에 명시된 클래스형인가 비교하며 그 결과는 논리형(부울형)이 된다.

아래의 예제에서는 변수 name이 Object 클래스의 인스턴스인가를 비교한다. 만약 name이 Object 클래스의 인스턴스 라면 "Object"라는 문자열을 출력할 것이고, 그렇지 않다면 String 클래스의 인스턴스인가를 비교하게 된다.

```
-----
public class Operator {
    public static void main(String[] ar) {
        String name = "KIHEE KIM";

        if (name instanceof Object) {
            System.out.println("Object");
        }
        else if (name instanceof String) {
            System.out.println("String");
        }
    }
}
```

```

    else {
        System.out.println("Unknown");
    }
}
}

```

◦ &, ^, | (비트단위 논리 연산자)

비트단위 논리 연산자는 두 피연산자의 값을 비트단위로 논리연산을 수행한다. 이때 & 연산자를 비트단위 논리곱이라고 하는 이유는 두 피연산자의 값을 곱하였을 때, 그 결과값이 0이 아니면 1, 0이면 0의 결과가 되기 때문이다. 마찬가지로 | 연산자는 비트단위 논리합이라고 하여 두 피연산자의 값을 더하였을 때, 그 결과값이 0이 아니면 1, 0이면 0의 결과가 된다.

^ 연산자는 배타적 논리합이라고 하여 두 피연산자를 더하였을 때 반드시 그 결과값이 1일 때만 1이 되며 그 이외의 경우에는 0이 된다.

◦ & (비트단위 논리곱)

A & B : A * B의 값이 0이 아니면 1, 0이면 0

◦ ^ (비트단위 배타적 논리합)

A ^ B : A + B의 값이 1일때만 1, 그렇지 않은 경우 0

◦ | (비트단위 논리합)

A | B : A + B의 값이 0이 아니면 1, 0이면 0

아래는 비트단위 논리곱 연산자를 이용하여 데이터를 비트로 표현하는 코드이다.

```

public class Operator {
    public static void main(String[] ar) {

        int a = 10;
        int b = 1;

        b <<= Integer.SIZE - 1;
        do {
            if ((a & b) > 0) System.out.print(1);
            else System.out.print(0);
            b >>= 1;
        } while (b != 0);
        System.out.println();
    }
}

```

※ 비트단위 논리연산자는 정수형 데이터만을 대상으로 연산이 가능하다.

◦ &&, || (논리 연산자)

&& : 논리곱 (두 피연산자의 값이 모두 참일 때만 true)

|| : 논리합 (두 피연산자의 값이 모두 거짓일 때만 false)

비교연산 식에서 피연산자의 자료형은 부울형이 되어야 한다. 이때 좌측의 피연산자의 값에 의해 연산식의 결과가 결정되는 경우, 우측의 피연산식은 수행되지 않는다. 예를 들어 아래와 같은 연산식이 있을 때 함수 `func_a()`의 수행결과가 `false`이면 `func_b()`는 수행되지 않는다.

```
-----
public class Operator {

    public static boolean func_a() {
        System.out.println("Call func_a()");
        return false;
    }

    public static boolean func_b() {
        System.out.println("Call func_b()");
        return false;
    }

    public static void main(String[] ar) {

        if (func_a() && func_b())
            System.out.println(true);
        else
            System.out.println(false);
    }
}
-----
```

위의 연산 식에서 `func_a()`의 결과가 `false`라면 우측 피연산자의 결과를 볼 필요가 없으므로 `func_b()`는 호출되지 않는다. 반대로 아래의 연산 식에서는 `func_a()`의 수행결과가 `true`이면 `func_b()`는 호출되지 않는다.

```
-----
public class Operator {

    public static boolean func_a() {
        System.out.println("Call func_a()");
        return true;
    }

    public static boolean func_b() {
        System.out.println("Call func_b()");
        return false;
    }

    public static void main(String[] ar) {

        if (func_a() || func_b())
            System.out.println(true);
        else
            System.out.println(false);
    }
}
-----
```

위의 연산식 또한 `func_a()`의 호출결과가 `true`라면 `func_b()` 함수의 호출결과에 상관없이 논리연산의 결과값이 `true`가 되므로 `func_b()` 함수는 호출하지 않게 된다.

논리 연산식을 실행하는 데 있어서 때로는 좌측 피연산자의 결과에 상관없이 우측 피연산자를 평가

해야 하는 경우가 있다. 이 경우에는 비트단위 논리 연산자를 이용할 수 있다.

아래의 예제는 비트단위 논리 연산자를 이용함으로써 연산 식에서 사용된 두 피연산자를 모두 평가하고 있다. & 연산식의 피연산자로 사용된 func_a() 함수의 호출과 func_b() 함수의 호출이 && 연산자가 아닌 & 연산자를 이용함으로써 & 연산 식은 피 연산자로 논리형이 아닌 데이터를 필요로 하게 되고, func_a() 와 func_b()의 호출 결과를 정수형으로 변환한 후, 비트단위 논리 연산을 수행함으로써 좌측 피연산자의 결과와 상관없이 우측 피연산자에 명시된 함수도 호출하게 된다 .

```
-----
public class Operator {

    public static boolean func_a() {
        System.out.println("Call func_a()");
        return false;
    }

    public static boolean func_b() {
        System.out.println("Call func_b()");
        return true;
    }

    public static void main(String[] ar) {

        if (func_a() & func_b())
            System.out.println(true);
        else
            System.out.println(false);
    }
}
-----
```

6.6 삼항, 배정, 대입, 순차 연산자

◦ `expr ? value1 : value2` (삼항 연산자)

삼항 연산자는 수식의 값이 참이면 `value1`을 거짓이면 `value2`를 취한다. 이때 `value1`과 `value2`은 상수나 변수 또는 수식이 될 수 있으며 두 값 또는 두 수식의 결과값은 동일한 자료형으로 형변환이 이뤄진다.

◦ `=, +=, /=, %=, +=, -=, <<=, >>=, >>>=` (배정 대입 연산자)

대입 및 배정 대입연산자는 연산자 우측의 피연산자(`Rv`)의 값을 좌측의 피연산자(`Lv`)에 저장한다. 이때 `Lv`에는 상수나 변수, 수식이 올 수 있으나 `Rv`에는 반드시 변수가 와야 한다.

대입 연산자는 연산의 수행 시, 두 피연산자를 동일한 자료형으로 변환하는 형변환이 발생하며 이때 `Rv`가 `Lv`형에 맞추어 형변환이 이뤄진다.

`*=` 연산자는 `Rv`의 값에 `Lv`값을 곱한 값을 다시 `Rv`에 저장한다. 아래의 두 수식의 결과는 동일하다.

```
a /= 3;
a = a / 3;
```

하지만 위의 두 연산 식은 약간의 차이가 있다. 연산식을 사용함에 있어서 자료형 변환이 발생할 수 있다고 배웠다. 예를 들어 아래의 연산 식은 정상적으로 컴파일 되지가 않는다. 변수 `a`는 `int`형이며 `3.0`은 `double`형이므로 나누기 연산 식은 `int`형을 `double`형으로 형변환(프로모션)을 한 후, 연산을 수행하게 되며 그 결과 역시 `double`형이 된다. 이때 `double`형은 `int`형보다 큰 자료형 이므로 변수 `a`에 저장할 수 없기 때문이다.

```
int a = 10;
a = 10 / 3.0;
```

위의 코드가 정상적으로 컴파일 되기 위해서는 아래와 같이 결과의 값을 자료형 변환(캐스팅)하여야만 한다.

```
int a = 10;
a = (int)(10 / 3.0);
```

반면 배정 대입 연산자를 이용하는 경우 자동으로 형변환이 이뤄지므로 별도의 자료형 변환이 필요치 않다.

```
int a = 10;
a /= 3.0;
```

아래의 또 다른 예를 보도록 하자. 아래의 예제는 두 `byte`형 변수의 더한 값을 다시 `byte`형 변수에 저장하는 예이다. 아래의 코드 역시 컴파일 시 에러가 발생한다. 그 이유는 산술연산을 수행함에 있어서 정수형 보다 작은 자료형은 모두 정수형으로 변환된 후, 연산식이 실행되기 때문이다. 즉 아래의 `a + b`의 결과값은 `int`형이 되므로 `byte`형 변수에 저장할 수 없다.

```
byte a = 10;
byte b = 48;
```

```
a = a + b;
```

위의 코드 역시 아래와 같이 배정 대입 연산자를 이용하면 자동으로 자료형 변환이 이뤄진다.

```
byte a = 10;  
byte b = 48;  
a += b;
```

◦ , (순차실행연산)

, 연산자는 순차실행 연산자라고 하며, 하나의 라인에서 여러 연산식을 순차적으로 처리하도록 해 준다.

```
int a = 10, b = 20;
```

7. 제어문

제어문이란? 프로그램 실행의 흐름을 변경하거나 제어하는 구문을 말한다. 이러한 제어문의 종류는 아래와 같다.

- 선택문 : if문
- 반복문 : while문, do ~ while문, for문
- 분기문 : switch문, break문, continue문

위의 제어문 이 외에도 throw, try ~ catch문 역시 제어문에 포함된다.

7.1 선택문

○ 형식 1

주어진 조건이 참인 경우, 문장을 수행한다.

```
if (조건) {  
    문장;  
}
```

○ 형식 2

주어진 조건이 참인 경우, 문장 1을 수행하며, 거짓인 경우 문장 2를 수행한다.

```
if (조건) {  
    문장1;  
} else {  
    문장2;  
}
```

○ 형식 3

주어진 조건1이 참인 경우, 문장 1을 수행한다. 만약 조건 1이 거짓이면 조건2를 평가하며 조건2가 참이면 문장2를 수행한다. 마지막 else절은 있을 수도 있고 없을 수도 있으며, else 절이 없는 경우 모든 조건이 참이 아닌 경우 그 어느 문장도 수행되지 않을 수도 있으나 else절이 있다면 위의 모든 조건이 일치하지 않더라도 else절 만큼은 실행된다.

```
if (조건1) {  
    문장1;  
} else if (조건2) {  
    문장2;  
} else if (조건n) {  
    문장n;  
} else {  
    문장m;  
}
```

7.2 반복문

◦ for문

for문은 반복문으로써 동일하거나 유사한 문장을 수치에 의해 제어할 수 있습니다.

```
for (초기화 영역 ; 조건부 영역 ; 증감부 영역) {
    문장;
}
```

아래의 예제는 '*' 문자를 5번 반복하여 출력하게 되는데 실행의 흐름은 아래와 같다.

- 1) i의 값을 0으로 초기화 한다.
- 2) i의 값이 5보다 작으냐를 검사한다.
- 3) 조건이 일치하면 {...} 부분을 실행하고, 일치하지 않으면 for 문을 탈출한다.
- 4) i의 값을 하나 증가시킨다. (2 ~ 4 까지 반복적으로 실행)

```
-----public class
StatementTest {
    public static void main(String[] ar) {
        for (int i=0 ; i<5 ; i++) {
            System.out.print("*");
        }
        System.out.println();
    }
}
-----
```

◦ 배열을 지원하는 향상된 for문

아래 형식의 for문은 배열의 항목 수 만큼 반복 수행된다. 매번 반복할 때마다 배열항목의 값을 변수에 자동으로 대입하고 문장을 수행하며, 더 이상 대입할 원소가 없는 경우 반복문을 탈출한다.

```
for (자료형 변수명 : 배열명) {
    문장;
}
```

아래는 배열을 지원하는 for문을 이용하여 배열 원소의 값을 출력하는 예제이다.

```
-----public class
LectureTest {
    public static void main(String[] args) {
        int[] arr = new int[]{10, 20, 30, 40, 50};

        for (int i : arr) {
            System.out.println(i);
        }
    }
}
-----
```

◦ 다중 for문

다중 for문은 for문이 또 다른 for문을 포함하는 형식을 말한다.


```

for (초기화 영역 ; 조건부 영역 ; 증감부 영역) {
    for (초기화 영역 ; 조건부 영역 ; 증감부 영역) {
        문장;
    }
}

```

아래의 예제는 다중 for문을 이용하여 배열을 정렬하는 예제이다.

```

-----public class
StatementTest {

    public static void main(String[] ar) {
        int[] arr = new int[]{50, 20, 90, 10, 30, 40, 80, 60, 70, 100};

        for (int i=0 ; i<arr.length-1 ; i++) {
            for (int j=i+1 ; j<arr.length ; j++) {
                if (arr[i] > arr[j]) {
                    int tmp = arr[i];
                    arr[i] = arr[j];
                    arr[j] = tmp;
                }
            }
        }

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

```

◦ while문

동일하거나 유사한 문장을 행위에 의해 제어하는 문장으로 주어진 조건이 만족하는 동안, 즉 조건이 true인동안 반복적으로 문장을 수행한다.

```

while (논리조건문 or true) {
    문장;
}

```

◦ do ~ while문

do ~ while문은 while문과 동일하지만 while문이 선 비교 후 실행인 반면 do ~ while 문은 선 실행 후 비교 구문이다. while문은 주어진 논리적 조건에 따라 한번도 수행되지 않을 수 있지만 do ~ while문은 적어도 한번은 실행된다.

```

do {
    문장;
} while (논리조건문 or true);

```

7.3 분기문

◦ switch문

switch문은 다중분기문으로써 switch 이후 주어진 수식의 값에 일치하는 case문으로 분기한다. 이때 condition으로는 수식이나 변수, 함수 등이 올 수 있으나 그 결과값이 정수이어야 하며, case값 역시 정수형 이어야만 한다.

switch문은 condition의 값이 1이라면 문장1을, n이라면 문장n을 수행하며, 그 어느 case 값과 일치 하지 않는 경우에는 default절을 수행하게 된다. 이때 default절은 있을 수도 있고 없을 수 도 있으나 default절이 없는 경우 어느 case도 실행되지 않을 수 도 있다. 또한 case절에 break문이 있을 수도 있고, 없을 수 도 있다. 만약 break문이 없다면 일치하는 case절 부터 그 이하의 모든 case절이 실행된다.

```
switch (condition) {  
    case 1 :  
        문장1;  
        break;  
  
    .....  
  
    case n :  
        문장n;  
        break;  
    default :  
        문장m;  
}
```

◦ break문

break문은 반복문이나 switch문을 수행하다 break문을 만나면 그 즉시 수행하던 조건 문이나 switch문을 빠져나와 다음 코드를 실행하게 된다. for문이나 while문 내의 break문은 break문을 포함하고 있는 반복문의 블럭만을 탈출하게 된다.

◦ continue문

for문이나 while문 또는 do ~ while문의 루프 내에서 사용되어지며 continue문을 만나면 for문은 증감부 영역으로, while문은 조건부로, do ~ while문은 반복문의 시작부분으로 실행을 이동시킨다.

continue문은 반복문 내에서 어떤 특정 조건의 처리를 생략하고자 할 때 사용하게 된다.

- LABEL (레이블)

레이블이란 중첩된 반복문에 사용되어지며 break문에서는 이 레이블을 이용하여 탈출할 반복문을 지정할 수 있다. 아래의 예제에서 return aaa 구문은 레이블을 이용함으로써 break문이 포함된 안쪽의 while문이 아닌 바깥쪽의 while문을 탈출하게 된다.

```
-----public class
StatementTest {

    public static void main(String[] ar) {
        int a = 0;
        int b = 0;

        aaa: while (a < 10) {

            b = 0;
            bbb: while (b < 10) {
                if (b == 5) break aaa;
                System.out.print("*");
                b++;
            }
            System.out.println();
            a++;
        }
    }
}
-----
```

8. 메서드

8.1 메서드의 정의 및 용법

메서드는 자주 반복하여 사용하는 내용에 대하여 특정 이름으로 정의한 프로그래밍 코드의 집합이다. 메서드는 클래스를 구성하는 요소로서 클래스의 기능에 대한 부분을 담당하게 된다. 클래스에 대한 내용은 차후 자세하게 살펴볼 예정이니 여기서는 메서드에 대한 내용만을 다루도록 할 것이다.

아래의 예제는 구구단을 출력하는 예제이다.

```
-----
public class MethodTest {
    public static void main(String[] ar) {

        for (int i=2 ; i<10 ; i++) {
            System.out.println("====" + i + "단 ===");
            for (int j=2 ; j<10 ; j++) {
                System.out.println(i + " x " + j + " = " + j * j);
            }
            System.out.println();
        }
    }
}
-----
```

위의 구구단 출력 프로그램은 코드는 프로그램 자체로는 문제가 없이 잘 실행된다. 하지만 고객의 요구사항이 변경되어 특정 단수만 출력하고자 한다면 프로그램 전체를 고쳐야 할 것이다. 하지만 위의 프로그램 코드 중 반복적으로 실행되는 부분을 별도의 메소드로 묶음으로써 보다 간결한 프로그래밍이 가능해진다.

아래의 예제코드에서는 구구단 출력의 반복적인 부분을 별도의 메소드로 작성하고 이것을 호출함으로써 특정 단수만 출력할 수 있도록 변경하였다.

```
-----
public class MethodTest {
    public static void main(String[] ar) {

        printDan(2);
        printDan(4);
        printDan(6);
    }

    public static void printDan(int dan) {
        System.out.println("====" + dan + "단 ===");
        for (int i=2 ; i<10 ; i++) {
            System.out.println(dan + " x " + i + " = " + dan * i);
        }
        System.out.println();
    }
}
-----
```

이처럼 메서드는 반복적으로 사용되는 부분을 특정 이름(메서드명)으로 정의하고 묶어 줌으로써 코드의 재사용성을 높여준다.

8.2 메서드의 정의 방법

메서드는 다음과 같은 형식으로 정의한다.

```
[접근제한자] [지정예약어] 리턴자료형 메서드명([형식인수목록]) [throws 예외클래스] {  
    메서드 내용 정의 부;  
}
```

메서드를 정의하는데 있어서 사용되는 접근제한자나 지정예약어 등은 이후 클래스 파트에서 자세하게 다뤄질 것이므로 지금은 `public`이라는 접근제한자와 `static`이라는 지정예약어를 사용하여야 한다는 것만 알아 두도록 하자.

8.3 메서드의 종류

메서드는 정의되는 방법에 따라 아래와 같이 세가지 종류가 있다.

- Call By Name
- Call By Value
- Call By Reference

○ Call By Name

메서드 이름에 의해 호출되는 메서드로 매개변수가 존재하지 않는 메서드이다. 아래의 예제를 보면 printStar() 함수를 호출함에 있어서 아무런 매개변수도 넘어가지 않는 것을 볼 수 있다. 이러한 메서드 호출방식을 가리켜 Call By Name 이라고 한다.

```
-----
public class MethodTest {
    public static void main(String[] ar) {
        printStar();
    }

    public static void printStar() {
        for (int i=0 ; i<20 ; i++)
            System.out.print("*");
        System.out.println();
    }
}
-----
```

○ Call By Value

메서드 호출 시 전달되는 매개변수 값을 기초로 하여 실행되는 메서드로써 매개변수 값의 복사가 발생하는 메서드이다. 아래의 예제를 보면 printDan() 메서드를 호출함에 있어서 int형 변수 dan을 매개변수로 넘기고 있으며, 이 매개변수를 받은 printDan() 메서드 내에서는 전달받은 dan의 값을 2 증가시키고 있다. 하지만 프로그램을 실행하여보면 main() 메서드 내의 변수 dan의 값은 증가되지 않는 것을 확인할 수 있다.

이것은 호출 시 넘겨주는 매개변수 dan과 정의 시 지정된 형식인수 dan이 서로 다른 주소상의 변수이며 메서드 호출 시 주어진 변수 dan의 값을 복사하여 사용하기 때문이다. 이처럼 매개변수의 값을 복사하여 사용하는 형식을 Call By Value 라고 한다.

```
-----
public class MethodTest {
    public static void main(String[] ar) {
        int dan = 2;
        printDan(dan);
        printDan(dan);
        printDan(dan);
    }

    public static void printDan(int dan) {
        System.out.println("====" + dan + "단 ===");
        for (int i=2 ; i<10 ; i++) {

```

```

        System.out.println(dan + " x " + i + " = " + dan * i);
    }
    System.out.println();
    dan = dan + 2;
}
}

```

◦ Call By Reference

C언어나 C++언어와는 달리 자바에는 Call By Reference 형식의 메서드는 없다. 이것은 C언어나 C++언어와는 달리 자바는 포인터 개념이 없기 때문이다. 그대로 Call By Reference 형식의 메서드를 설명한다면 이것은 메서드 호출 시 전달되는 매개변수가 기본 자료형이 아닌 객체형을 인수로 하는 경우이다.

아래의 예제코드에서는 swap() 메서드를 호출함에 있어서 Integer라는 Wrapper 클래스의 인스턴스를 매개변수로 넘기고 있다. 그리고 swap() 메서드에서는 두 개의 Integer인스턴스를 x와 y라는 이름으로 받아 이 두 객체의 값을 바꾸는 예제이다.

```

public class MethodTest {
    public static void main(String[] ar) {
        Integer i = new Integer(10);
        Integer j = new Integer(20);

        System.out.printf("%d : %d\n", i, j);
        swap(i, j);
        System.out.printf("%d : %d\n", i, j);
    }

    public static void swap(Integer x, Integer y) {
        int temp = x.intValue();
        x = y.intValue();
        y = temp;
    }
}

```

하지만 위의 예제를 작성한 후, 실행하여 보면 두 객체의 값이 변경되지 않았음을 확인할 수 있다. 그렇다면 객체를 매개변수로 넘겼음에도 두 값이 변경되지 않은 이유는 무엇일까? 위의 예제코드의 swap() 메서드는 분명히 Call By Reference이다. 그럼에도 두 값이 변경이 발생되지 않는 이유는 다음과 같다.

우선 함수 호출 시 주어진 두 객체 i와 j의 값을 swap() 메서드에서 x와 y라는 이름으로 받는다. 이때 i와 x, 그리고 j와 y는 같은 객체를 참조하고 있을 뿐 i와 x 그리고 j와 y는 다른 주소상의 공간을 할당 받은 변수이다. 즉 i와 j는 main() 메서드에서 할당된 변수이며 이 변수에는 new 예약어에 의해 동적으로 할당된 객체의 주소를 가지고 있다고 봐야 한다. 마찬가지로 x와 y는 swap() 메소드가 호출 시 할당되고 그 값으로 각각 i와 j가 가리키는 주소 값이 저장되어 있다. 여기서 주의할 점은 i와 x가 같은 객체를 참조하고 있을 뿐, i와 x 자체는 서로 다른 공간을 할당 받은 변수라는 것이다.

swap() 메서드를 보면 두 객체의 값을 바꾸는 것처럼 보이나 이것은 두 객체의 값을 바꾸는 것이 아니라 x와 y에 새로운 인스턴스를 할당하는 것이다.

아래의 코드는 x 객체에 저장된 정수 값을 가져와 temp에 저장한다.

```
int temp = x.intValue();
```

객체 x에 y.intValue()의 값을 할당한다. 하지만 이때 객체 x에 저장된 정수 값을 y에 저장된 정수 값으로 바꾸는 것이 아니라 새로운 Integer 클래스의 인스턴스를 만들고 그 인스턴스를 가리키는 객체가 되어버린다.

```
x = y.intValue();
```

그러므로 변수 x는 매개변수로 넘겨받은 i가 가리키는 인스턴스가 아닌 새로 생성된 Integer 클래스의 인스턴스이다. 변수 y 역시 x와 마찬가지로 y객체에 저장된 값을 변수 temp에 저장된 값으로 바꾸는 것이 아니라 변수 temp의 값으로 Integer 클래스의 인스턴스를 생성하고 그 인스턴스를 가리키게 된다. 그리고 형식인수인 x와 y 두 변수는 메서드 내에서만 유효한 지역변수로서 메소드의 탈출과 동시에 소멸되어 버린다.

즉 자바에서는 메소드 호출 시 인자로 넘겨진 객체(레퍼런스) 자체를 바꿀 수 없다. 하지만 인자로 넘겨진 객체의 상태 값을 변경할 수는 있다.

```
-----
class MyInteger {
    int value;

    public static void swap(MyInteger a, MyInteger b) {
        int temp = a.value;
        a.value = b.value;
        b.value = temp;
    }
}

public class OperatorTest {
    public static void main(String[] ar) {
        MyInteger t1 = new MyInteger();
        MyInteger t2 = new MyInteger();

        t1.value = 20;
        t2.value = 30;
        System.out.println(t1.value + ":" + t2.value);
        MyInteger.swap(t1, t2);
        System.out.println(t1.value + ":" + t2.value);
    }
}
-----
```


8.4 리턴 결과형

리턴 결과형은 메서드를 실행한 후 결과로 되돌려 주는 값의 자료형을 명시한다. 메서드에서 결과값의 반환은 `return` 이라는 예약어를 이용하여 반환하게 된다.

```
return 식;
```

만약 반환 값이 없는 메서드인 경우에는 `return` 구문이 생략될 수 있으나 아래와 같이 `return` 구문만 사용할 수 도 있다.

```
return;
```

`return` 구문에 의해 반환할 수 있는 값은 오직 하나뿐이며, `return` 구문과 동시에 함수를 탈출하게 된다.

메서드에서 반환할 수 있는 결과값의 자료형은 아래와 같다.

- `void` : 실행 후 돌려 줄 결과가 없을 때
- 기본자료형
- 객체형 : 배열, 클래스, 인터페이스형

9. 배열

배열이란? 동일한 자료형으로 선언된 메모리상의 연속적인 공간을 말한다. 배열을 이용함으로써 효율적인 데이터 관리가 가능해진다.

9.1 단일 차원 배열

◦ 단일차원 배열의 선언

```
자료형[] 배열명;
자료형 배열명[];
```

예)

```
int[] arr;
int arr[];
```

배열은 선언하여 사용하기 전에 반드시 초기화를 하여야만 한다.

◦ 단일차원 배열의 초기화

```
배열명 = new 자료형[배열원소개수];
배열명 = new 자료형[]{배열원소목록};
```

예)

```
arr = new int[3];
arr = new int[]{10, 20, 30};
```

배열을 초기화 할 때, 초기화 값을 지정하지 않는 경우에는 기본값으로 초기화가 이뤄진다. 아래의 예제에서 int형 배열 arr에 초기값을 지정하지 않았음에도 자동으로 0으로 초기화 되었음을 확인할 수 있다.

```
-----
public class ArrayTest {
    public static void main(String[] ar) {
        int[] arr;
        arr = new int[3];

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

기본 초기값은 자료형에 따라 달라지는데 아래와 같다.

자료형	초기화 값
boolean	false

byte, short, int	0
char	'\0'
long	0L
float	0.0f
double	0.0
String	null

○ 선언과 동시에 초기화

```
자료형[] 배열명 = new 자료형[배열원소개수];
자료형[] 배열명 = new 자료형[]{배열원소목록};
자료형[] 배열명 = {배열원소목록};
```

예)

```
int[] arr = new int[3];
int[] arr = new int[]{10, 20, 30};
int[] arr = {10, 20, 30};
```

배열을 선언하는데 있어서 선언과 동시에 초기화가 가능하다. 위의 예에서 세 번째 방식은 C언어에서 사용하던 방식으로 일반적으로 자바에서는 많이 사용하지는 않는다.

아래의 예제는 1차원 배열을 이용하여 국어, 영어, 수학점수를 입력 받고 각각의 점수와 총점, 평균을 출력하는 예제이다.

```
-----
public class ArrayTest {
    public static void main(String[] ar) throws IOException {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

        int[] subjumsu = new int[3];
        String[] subname = new String[]{"국어", "영어", "수학"};
        int tot = 0;
        double avg = 0.0;

        for (int i=0 ; i<subname.length ; i++) {
            System.out.print(subname[i] + "의 점수 입력 : ");
            subjumsu[i] = Integer.parseInt(in.readLine());
            tot += subjumsu[i];
        }
        avg = tot/3.0;

        for (int i=0 ; i<subname.length ; i++) {
            System.out.println(subname[i] + " : " + subjumsu[i]);
        }
        System.out.println("총점 : " + tot);
        System.out.println("평균 : " + avg);
    }
}
-----
```

9.2 배열의 관리

- 배열명은 Reference 이다.
- 배열명은 4Byte 객체이다.
- 배열의 길이는 length 속성을 이용할 수 있다.
- 배열의 실제 데이터는 Heap 영역에 저장된다.
- Garbage Collection에 의해 소멸된다.
- 각 공간은 0으로 부터 시작하는 첨자로 구분된다.
- 공간의 값은 자동으로 Default 초기화가 된다.
- 동적으로 공간할당이 가능하다.

배열을 선언하고 초기화 하는데 있어서 new라는 예약어를 이용한다. new 예약어에는 동적 할당 예약어로서 클래스의 인스턴스(객체)를 생성하거나 배열의 공간을 초기화(동적 할당)할 때 사용된다. new 예약어에 의해 동적으로 할당되는 객체는 그 객체의 데이터를 저장하기 위해 Garbage Collection Heap영역에 공간을 할당하고 할당된 메모리공간의 시작주소를 Runtime Stack영역에 4바이트를 할당하고 저장하게 된다.

```
int[] arr = new int[3];
```

위와 같이 배열을 선언하고 초기화 하는 경우 아래와 같이 메모리가 할당된다.

Runtime Stack				Garbage Collection Heap			
			주소 값	0	0	0	
			4Byte	4Byte	4Byte	4Byte	



자바에서의 배열은 c언어와는 달리 동적으로 할당이 가능하다. 여기서 말하는 동적이라는 의미는 Garbage Collection Heap에 할당함을 의미하는 것이 아니라 고정된 크기가 아닌 조건이나 상황에 따라 배열의 개수가 달라짐을 의미한다. c언어에서는 배열의 동적 할당이 불가능 하므로 배열의 크기를 바꾸기 위해서는 소스코드에서 배열의 크기를 변경한 후, 다시 컴파일 하여야만 하지만 자바에서는 배열의 동적 할당이 가능함으로 다시 컴파일 하는 일 없이 새로운 크기의 배열의 할당이 가능하게 된다.

아래의 예제는 처리할 과목의 수를 입력 받고, 입력 받은 과목수 만큼의 배열공간을 할당한다.

```
-----
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class ArrayTest {
    public static void main(String[] ar) throws IOException {
        int[] sub;
        int count = getSubCount();

        sub = new int[count];
        System.out.println(sub.length + "개의 배열이 생성되었습니다.");
    }
}
```

```
public static int getSubCount() throws IOException {  
    BufferedReader in = new BufferedReader(  
        new InputStreamReader(  
            System.in));  
    System.out.print("처리할 과목수 : ");  
    return Integer.parseInt(in.readLine());  
}  
}
```

9.3 main() 메소드의 매개변수

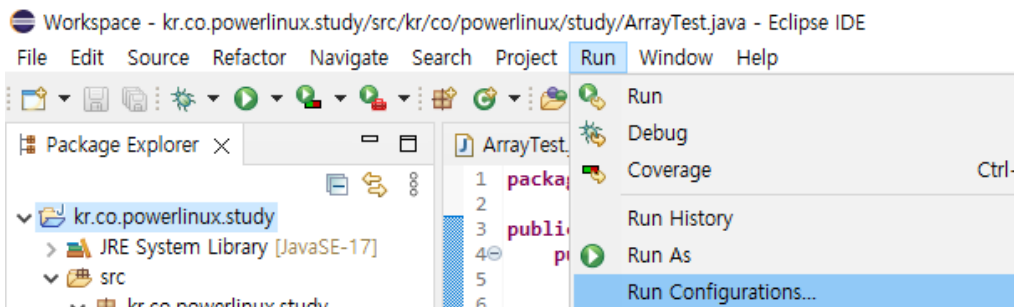
main() 메소드의 매개변수는 String 클래스 타입의 배열이다.

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

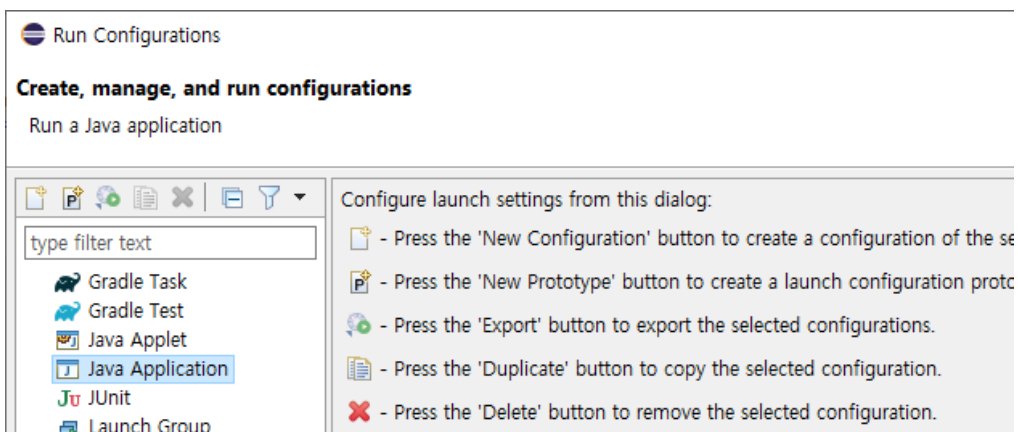
main() 메소드의 매개변수는 프로그램 시작 시 초기값을 전달받기 위한 용도로 사용된다. 아래와 같이 자바 소스코드를 작성한 후, ArrayTest.java 라는 이름으로 저장을 한다.

```
public class ArrayTest {
    public static void main(String[] args) {
        System.out.println("입력된 총 인수의 수 : " + args.length);
        for (String arg : args) {
            System.out.println(arg);
        }
    }
}
```

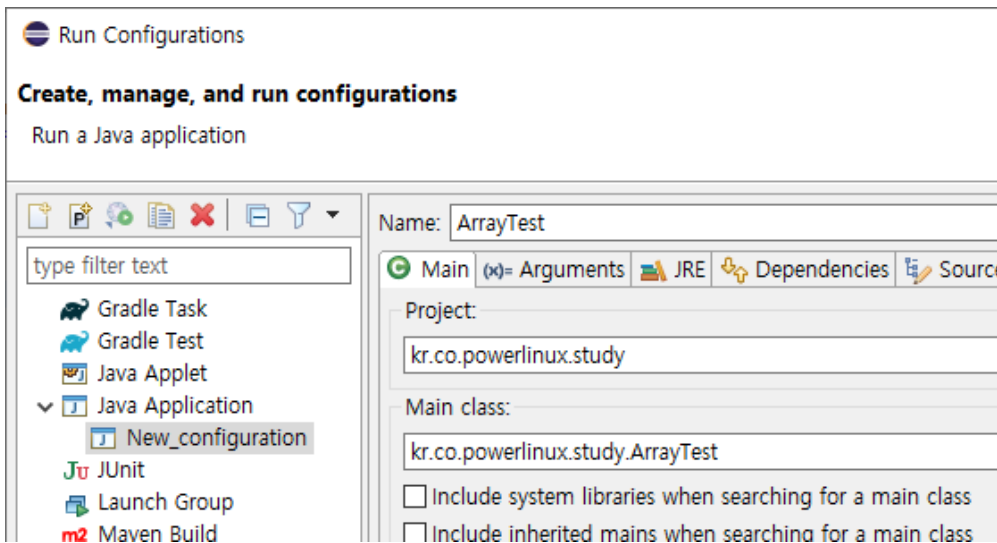
이클립스에서 어플리케이션 실행 시 전달할 파라미터 값을 지정하는 방법은 다음과 같다. 우선 Run > Run Configurations... 메뉴를 선택한다.



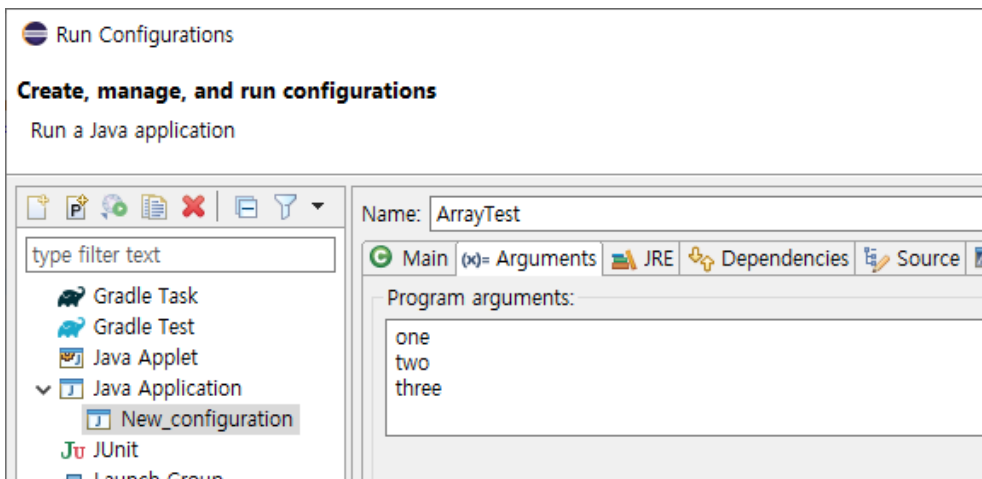
Java Application 타입을 더블 클릭하여 새로운 Java Application Configuration을 생성한다.



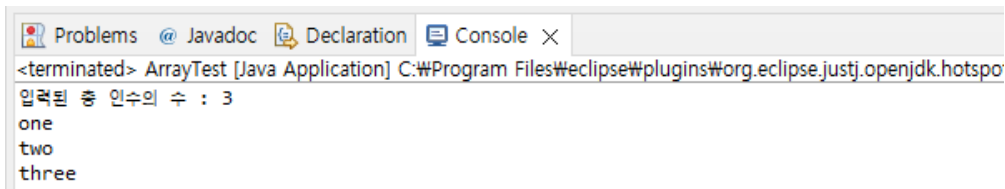
Main 탭의 Run Configuration Name과 Main class를 입력한다.



Arguments 탭을 선택한 후 Program arguments에 main() 메서드에 전달할 인수를 입력한후 "Run" 버튼을 클릭하여 실행한다.



실행결과는 Console View를 통해 확인이 가능하다.



위에서 처럼 main() 메서드의 매개변수는 String 클래스형의 배열로써 자바 프로그램을 실행 시 주어진 파라미터를 입력 받는 용도로 사용된다.

아래의 예제는 처리할 과목수를 프로그램 실행 시, 파라미터 값으로 넘겨받는 예제이다.
아래의 코드를 이클립스에서 작성한 후 실행하면 아래와 같은 메시지가 출력되며 정상적으로 실행이 되지 않는다.

```
-----  
public class ArrayTest {  
    public static void main(String[] ar) throws Exception {  
        int[] sub;  
  
        if (ar.length < 1) {  
            System.out.println("프로그램을 실행하기 위해서는 처리할 과목수를 입력하여야 합니다.");  
            System.out.println("Usage : java ArrayTest count");  
            System.exit(0);  
        }  
  
        sub = new int[Integer.parseInt(ar[0])];  
        System.out.println(sub.length + "개의 배열을 할당하였습니다.");  
    }  
}
```

결과

```
-----  
프로그램을 실행하기 위해서는 처리할 과목수를 입력하여야 합니다.  
Usage : java ArrayTest count  
-----
```

이클립스에서 클래스를 실행함에 있어서 파라미터를 지정하는 방법은 아래와 같다. 우선 Project Explorer 에서 프로젝트를 선택한 후, 메뉴바의 Run > Run Configurations... 를 선택한다.

9.4 다차원 배열

다차원 배열은 배열의 첨자가 2개 이상인 배열을 말한다.

○ 2차원 배열의 선언

```
자료형[][] 배열명;
자료형[] 배열명[];
자료형 배열명[][];
```

예)

```
int[][] arr;
int[] arr[];
int arr[][];
```

○ 2차원 배열의 초기화

```
배열명 = new 자료형[개수][개수];
배열명 = new 자료형[][]{{초기화 리스트}, {초기화 리스트}};
배열명 = new 자료형[개수][];    // 동적배열
```

예)

```
arr = new int[3][2];
arr = new int[][]{{1, 2, 3}, {4, 5, 6}};

// 동적배열의 초기화
arr = new int[2][];
arr[0] = new int[5];
arr[1] = new int[]{1, 2, 3};
```

아래의 예제는 2차원 배열을 선언하고 초기화 하는 예제이다. 배열 arr은 2차원 배열이며 arr[0]은 배열 원소의 개수가 5개인 1차원 배열이며, arr[1]은 배열원소의 개수가 3개인 1차원배열이다. 즉 2차원 배열은 1차원 배열을 원소로 하는 배열임을 알 수 있다.

```
-----
public class ArrayTest {
    public static void main(String[] ar) {
        int[][] arr = new int[2][];

        arr[0] = new int[5];
        arr[1] = new int[]{10, 20, 30};

        for (int i=0 ; i<arr.length ; i++) {
            for (int j=0 ; j<arr[i].length ; j++) {
                System.out.printf("arr[%d][%d] : %d\t", i, j, arr[i][j]);
            }
            System.out.println();
        }
    }
}
-----
```

9.5 다차원 배열의 관리

- 다차원 배열에서 각 영역은 모두 Reference 이고, 마지막 배열공간만이 실제 데이터 공간이다.
- 다차원 배열은 단일차원 배열로 구성된다.
- 다차원 배열은 다중 for문과 함께 사용한다.
- 다차원 배열은 동적배열이 가능하다.

아래 예제의 2차원 배열 arr의 원소의 개수를 출력해보면 배열 arr의 원소의 개수는 2개임을 알 수가 있다. 또한 arr[0]과 arr[1]은 원소의 개수가 3개임을 나타낸다. 즉 자바에서의 다차원 배열은 배열로 이루어진 배열이다. 아래의 2차원 배열에서 arr은 arr[0]과 arr[1]을 원소로 하며, arr[0]은 arr[0][0], arr[0][1], arr[0][2]을 원소로 한다. 그러므로 arr과 arr[0], arr[1]은 레퍼런스이며 arr[0][1] ... arr[1][2] 가 실제 데이터를 저장하는 원소이다.

```
-----
public class ArrayTest {
    public static void main(String[] ar) {

        int arr[][] = new int[2][3];
        System.out.println("배열 arr의 원소의 개수 : " + arr.length);

        for (int i=0 ; i<arr.length ; i++) {
            System.out.println("배열 arr[" + i + "]의 원소의 개수 : " + arr[i].length);
        }
    }
}
-----
```

10. 클래스

클래스에 대한 개념은 바라보는 시각에 따라 다양한 표현이 가능하다. 분석 및 설계 관점에서는 추상적 개념으로 표현할 수 있고, 프로그래머 관점에서는 구체적인 구현으로 표현할 수 도 있다. 이러한 클래스를 객체 지향 개념 관점으로 표현하면 여러 유사 객체들이 공통적으로 가지는 속성이나 행위를 기술하는 명세장치이다.

예를 들어 인터넷 쇼핑물의 경우를 살펴보자. 쇼핑물의 고객은 수십 혹은 수천명에 이를 것이다. 그러나 이들 고객들이 회원가입을 할 때 회원 가입에 나타나는 가입양식은 동일하다. 즉, 각각의 개별 회원들이 회원 가입을 위해 적는 내용은 다르지만 모든 회원들은 아이디(id)나 비밀번호(password), 전화번호(phone) 등 공통적인 속성을 가지며 쇼핑몰에서 물건을 구입(buy)할 수 있다. 이처럼 아이디나 비밀번호, 전화번호와 같은 속성과 구입이라는 행위를 회원이라는 틀로 묶을 수 있는데 이 것을 클래스라고 하며, 이 하나의 클래스를 이용하여 서로 다른 속성값을 가지는 회원이라는 객체가 생성된다.

클래스는 다음과 같은 특성을 가지고 있다.

- 클래스는 고유한 이름을 가진다.
- 클래스는 속성을 지닌다.
- 클래스는 잘 정의된 행위를 가진다.

- **클래스는 고유한 이름을 가진다.**

객체의 일반적인 특징 중 하나는 다른 객체와 구별을 위한 유일한 식별자를 가진다 이와 마찬가지로 특정 도메인 내에서 클래스는 중복될 수 없으므로 클래스 역시 다른 클래스와 구분되기 위한 고유한 식별자(클래스명)를 가지며 이 식별자는 중복이 되어서는 안 된다.

- **클래스는 속성을 지닌다.**

클래스는 객체가 공통적으로 가지는 속성이나 행위를 기술하는 명세장치라고 하였다. 클래스는 의미 있는 정보를 저장하기 위한 저장소 역할을 하기 위해 속성을 내포한다. 이 속성을 (Property 또는 Attribute)을 구현 관점에서는 상태 변수, 멤버 변수 혹은 **멤버 필드** 등으로 표현한다.

- **클래스는 잘 정의된 행위를 가진다.**

클래스는 잘 정의된 행위를 지녀야 한다. 행위를 구현 관점에서 표현하면 멤버 메서드, 멤버 함수라고 한다. 예를 들어 회원 클래스 내에 아이디나 비밀번호, 전화번호와 같은 속성이 있다면, 같은 클래스 내에 '회원가입'이나 '회원정보수정', '회원탈퇴'와 같은 행위(기능)이 있다.

10.1 클래스의 기본 구성

○ 클래스의 형식

```
[접근제한자] [지정예약어] class 클래스명
    [extends 상위클래스] [implements 인터페이스] {
        클래스 멤버 ...
    }
```

○ 클래스의 접근제한자

public	클래스를 공개함으로써 어디서든 접근이 가능
package	동일한 패키지(디렉토리)내에서만 접근이 가능

○ 클래스의 지정예약어

final	더 이상 상속이 불가능한 클래스
abstract	객체의 생성이 불가능 한 추상 클래스

○ 클래스 멤버

Nested Class	중첩클래스
Field	데이터 저장공간으로 멤버필드, 멤버변수라고 함
Construct	생성자 메서드
Method	특정 행위의 기술영역으로 멤버 메서드라고 함

10.2 클래스 사용법

객체지향 개념을 공부하는데 있어서 흔히 혼동하는 것 중 하나가 클래스와 객체이다. 그러므로 우선 클래스와 객체의 관계를 알아보자.

클래스와 객체의 관계는 다음과 같다.

- 클래스는 틀이고 객체는 실 사례이다.
- 클래스는 상태가 없고 객체는 상태가 있다.
- 클래스와 객체는 동일한 수의 속성과 메소드 수를 갖는다.

○ 클래스는 틀이고 객체는 실 사례이다.

객체 지향 언어에서 프로그램의 주체는 클래스가 아닌 객체이다. 즉 객체를 생성하고 이 객체가 가지는 상태 값(데이터)을 객체가 가지는 행위를 통해 변경하고 처리하는 것이 객체지향 프로그래밍이다. 그러므로 프로그램의 주체는 클래스가 아닌 객체이다. 이 객체를 생성하기 위한 틀이 클래스이며 이 클래스를 new 연산자를 이용하여 객체화 한 것을 가리켜 객체 또는 인스턴스라고 하며, 객체화 하는 것을 가리켜 "객체를 생성한다" 또는 "인스턴스화 한다"라고 말한다.

생성된 객체는 클래스에 명시된 속성과 행위를 가지며, 속성에 대한 상태 값을 저장하기 위한 메모리 공간을 할당 받게 된다. 그러므로 실제 데이터를 가지며 행위를 수행하는 주체는 객체가 된다.

○ 클래스는 상태를 갖지 않지만 객체는 상태를 갖는다.

클래스와 객체는 모두 내부적으로 속성을 내포하고 있다. 그러나 클래스는 속성에 대한 선언만을 내포하고 있는 반면 객체는 선언된 각 속성들에 대한 값을 지니게 되는데 이러한 속성들에 대한 값을 상태라고 한다.

예외적으로 클래스가 상태 값을 갖는 경우가 존재할 수 도 있는데 클래스 멤버라 불리는 static 멤버가 그 예이다.

○ 클래스와 객체는 동일한 속성 및 메소드 수를 갖는다.

한 클래스로 부터 생성되는 객체들의 속성이나 메소드 수는 동일하다. 예를 들어 고객이라는 클래스로 부터 생성된 객체들은 그 고객이 남자이던 여자인지 모두 동일한 속성수와 메소드 수를 가지게 된다.

○ 객체의 생성 방법

클래스로부터 객체를 생성하는 방법은 아래와 같다.

형식) 클래스 객체명 = new 클래스();

아래의 예제는 Person 이라는 클래스로부터 객체를 생성하는 예제이다.

```
-----
class Person {
    String name;
    int age;
```

```
String phone;
}

public class ClassTest {
    public static void main(String[] ar) {

        Person man = new Person();
        man.name = "김기희";
        man.age = 30;
        man.phone = "017-233-8409";

        System.out.println("이름 : " + man.name);
        System.out.println("연령 : " + man.age);
        System.out.println("전화 : " + man.phone);
    }
}
```

클래스로부터 객체를 생성할 때에는 new 예약어를 이용한다. new 예약어는 동적할당 구문으로써, 클래스로부터 생성된 객체의 데이터의 저장공간은 Garbage Collection Heap 영역에 할당된다.

클래스로부터 생성된 객체는 클래스와 동일한 수의 속성과 행위를 가진다고 하였다. 위의 예제에서는 행위(메서드)는 없지만 세 개의 속성이 선언 되었으므로 Person 클래스로부터 생성된 man 객체는 세 개의 속성을 가지며 이 속성에 대한 상태 값을 참조하거나 변경할 수 있게 되는데 이때 사용되는 연산자가 "."(객체참조) 연산자이다.

우리는 하나의 클래스로부터 여러 개의 객체가 생성될 수 있다고 배웠다. 또한 클래스와 마찬가지로 객체 역시 다른 객체와 구별을 위한 식별자(객체명)를 가진다고 하였다. 아래와 같이 Person 클래스로부터 kim과 lee라는 두 객체가 생성되었을 때, kim과 lee 두 객체는 Person 클래스에 선언된 속성을 동일하게 가지게 된다.

```
Person kim = new Person();
Person lee = new Person();
```

하지만 이때 객체 kim과 lee는 서로 다른 객체이므로 속성명은 동일하나 서로 다른 저장공간을 가지고 있을 것이다. 그리고 이 저장공간에 접근하기 위해서는 어느 객체의 속성인가를 나타내야 하는데 이때 사용되어지는 연산자가 "." 연산자이다.

```
kim.name = "KIHEE KIM";    <- kim 객체의 name 속성값을 변경
lee.name = "WOOCHANG LEE"; <- lee 객체의 name 속성값을 변경
```

10.3 생성자 메서드

생성자 메서드는 클래스를 이용하여 객체를 생성할 때 호출되어지는 메서드로써 생성되는 객체의 상태 값을 초기화 하기 위해 사용되어진다. 아래는 생성자 메서드의 특징이다.

- 클래스명과 동일한 식별자를 가진다.
- 리턴 자료형을 명시하지도 않으며, 리턴 하지도 않는다.
- 객체 생성 시 반드시 하나의 생성자가 호출된다.
- 속성(멤버 필드)의 값을 초기화 하기 위해 사용된다.
- 생성자 함수가 없을 때, 자동으로 default 생성자가 제공되고, 호출된다.
- 오버로딩이 가능하다.

자바에서는 아래와 같이 생성자 메서드가 없는 경우에는 JVM에 의해서 자동으로 default() 생성자가 제공되고 호출된다. 아래의 예제에서 Person 클래스는 생성자 함수가 없으므로 (JVM에 의해 default 생성자가 제공되지만 실행코드가 없음) Person 클래스로 부터 생성되는 객체의 멤버 name, age, phone 의 값을 초기화 할 수가 없다.

정확히는 name, age, phone 멤버의 값은 default 값으로 초기화 될 뿐, 객체를 생성시 특정 값으로 초기화 할 수 없음을 의미한다.

```
-----
class Person {
    String name;
    int age;
    String phone;
}

public class ClassTest {
    public static void main(String[] ar) {

        Person man = new Person();
    }
}
-----
```

이제 위의 코드에 생성자 메서드를 추가하여 Person 클래스로부터 객체가 생성될 때, 객체의 상태 값(멤버 필드 값)을 특정 값으로 초기화가 가능하도록 해보자.

아래의 예제에서는 세 개의 인수를 전달받는 생성자 함수를 선언하였다. 위에서도 설명하였듯이 생성자 함수는 클래스명과 동일한 식별자를 가진다. 또한 반환 값이 없으므로 리턴 결과의 자료형을 명시하지 않을 뿐만 아니라 return에 의한 반환도 하지 않는다.

아래와 같이 생성자 함수를 명시하는 경우, JVM은 default 생성자를 제공하지 않으므로 반드시 생성자 함수를 호출 할 때, 선언 시 주어진 매개변수 만큼의 인자를 전달하여야만 한다.

```
-----
class Person {
    String name;
    int age;
    String phone;
}
-----
```

```

    public Person(String _name, int _age, String _phone) {
        name = _name;
        age = _age;
        phone = _phone;
    }
}

public class ClassTest {
    public static void main(String[] ar) {

        Person man = new Person("김기희", 30, "017-233-8409");
        System.out.println("이름 : " + man.name);
        System.out.println("나이 : " + man.age);
        System.out.println("전화 : " + man.phone);
    }
}

```

때로는 객체를 생성할 때, 모든 데이터를 초기화 하는 것이 아니라 특정 멤버 변수만을 초기화 하고 싶다면 아래와 같이 정의할 수 있다. 아래에서는 동일한 식별자를 가지는 생성자 메서드가 여러 개 정의되었다. 이때 생성자 메서드의 호출은 주어지는 인자 값과 일치하는 매개변수를 가지는 생성자 메서드가 호출되어지는데 이것을 가리켜 오버로딩(Overloading)이라고 한다. 오버로딩에 대해서는 나중에 자세하게 다룰 예정이니 지금은 오버로딩이 무엇이다 라는 것만 알아 두자. 또한 아래의 예제에서는 생성자 메서드 이외에 print() 라는 멤버 메서드를 하나 추가하였다. 우리는 이전 시간에 메서드는 반복적으로 실행되는 부분을 별도의 메서드로 정의하고, 호출하여 사용함으로써 코드의 재사용성을 높일 수 있다고 배웠다. 즉 Person 클래스에 print() 라는 행위를 추가함으로써 이 클래스로 부터 파생된 모든 객체들은 print()하는 기능을 가지게 된다.

```

class Person {
    String name;
    int age;
    String phone;

    public Person() {
    }

    public Person(String _name) {
        name = _name;
    }

    public Person(String _name, int _age) {
        name = _name;
        age = _age;
    }

    public Person(String _name, int _age, String _phone) {
        name = _name;
        age = _age;
        phone = _phone;
    }

    public void print() {
        System.out.println("===== " + name + " =====");
    }
}

```



```

        System.out.println("나이 : " + age);
        System.out.println("전화 : " + phone);
    }
}

public class ClassTest {
    public static void main(String[] ar) {

        Person kim = new Person();
        Person lee = new Person("이순신");
        Person sim = new Person("심청이", 19);
        Person lim = new Person("임꺽정", 40, "02-3432-6554");

        kim.print();
        lee.print();
        sim.print();
        lim.print();
    }
}

```

간혹 생성자 함수나 메서드의 매개변수명을 사용함에 있어서 멤버 필드와 동일한 이름을 사용하는 경우가 있다.

아래의 예제에서는 생성자 메서드의 매개변수명이 클래스의 속성명과 동일하다. 이러한 경우 객체의 멤버 필드는 메서드의 매개변수에 의해 가려지게 되어 주어진 인자 값으로 초기화가 이뤄지지 않는다.

```

class Person {
    String name;
    int age;
    String phone;

    public Person(String name, int age, String phone) {
        name = name;
        age = age;
        phone = phone;
    }

    public void print() {

        System.out.println("===== " + name + " =====");
        System.out.println("나이 : " + age);
        System.out.println("전화 : " + phone);
    }
}

public class ClassTest {
    public static void main(String[] ar) {

        Person lim = new Person("임꺽정", 40, "02-3432-6554");
        lim.print();
    }
}

```

위와 같이 멤버 필드명과 매개변수명이 동일한 경우, 멤버 필드는 매개변수에 의해 가려지게 된다.

아래의 대입 연산식의 Lv와 Rv 모두 같은 매개변수 name 인 것이다.

```
name = name;
```

이와 같은 경우에는 멤버 필드 name과 매개변수 name을 구분하기 위해 this라는 예약어를 이용한다.

```
-----
class Person {
    String name;
    int age;
    String phone;

    public Person(String name, int age, String phone) {
        this.name = name;
        this.age = age;
        this.phone = phone;
    }

    public void print() {
        System.out.println("===== " + name + " =====");
        System.out.println("나이 : " + age);
        System.out.println("전화 : " + phone);
    }
}

public class ClassTest {
    public static void main(String[] ar) {

        Person lim = new Person("임격정", 40, "02-3432-6554");
        lim.print();
    }
}
-----
```

this 예약어는 클래스 내에서만 사용되어지며, 자신이 포함된 인스턴스를 가리킨다. 즉 객체 lim에서의 this는 lim객체가 되는 것이다. 그러므로 this.name = name; 구문은 객체의 멤버필드 name에 매개변수 name의 값을 대입하라는 의미가 된다.

this가 자신이 포함된 클래스의 인스턴스를 가리킨다면 this()는 클래스내의 생성자 함수의 호출을 의미한다. 아래의 예제를 보면 this(); 구문에 의해 default 생성자 메서드인 Person() 메서드가 호출됨을 확인할 수 있다.

```
-----
class Person {
    String name;
    int age;
    String phone;

    public Person() {
        System.out.println("default 생성자 함수를 호출하였습니다.");
    }

    public Person(String name, int age, String phone) {
        this();
    }
}
-----
```

```
        this.name = name;
        this.age = age;
        this.phone = phone;
    }

    public void print() {
        System.out.println("===== " + name + " =====");
        System.out.println("나이 : " + age);
        System.out.println("전화 : " + phone);
    }
}

public class ClassTest {
    public static void main(String[] ar) {

        Person lim = new Person("임꺽정", 40, "02-3432-6554");
        lim.print();
    }
}
```

this() 구문의 사용은 아래와 같은 제약이 따른다.

- this() 구문은 생성자 메서드에서만 사용이 가능하다.
- this() 구문에 의한 생성자 메서드의 호출은 한 메서드 내에서 한번만 가능하다.
- this() 구문의 사용은 메서드 내에서 제일 먼저 사용하여야 한다.
- this()와 super()는 둘 중 하나만 사용이 가능하다.

11. 캡슐화와 정보은닉

11.1 캡슐화의 개념

클래스 혹은 객체라는 개념을 접할 때, 가장 먼저 생각나는 용어 가운데 하나가 바로 캡슐화이다. 이는 클래스 혹은 객체라는 개념과 캡슐화가 늘 항상 함께 따라다니는 실과 바늘과 같은 관계이기 때문이다.

캡슐화는 한마디로 다음과 같이 정의할 수 있다.

"캡슐화는 관련된 데이터와 함수를 묶는 장치이다."

즉 캡슐화란 관련된 데이터와 관련된 함수들을 클래스라는 하나의 캡슐 속으로 그룹화 혹은 묶어주는 장치 또는 메커니즘이라고 할 수 있다. 캡슐화를 다른 말로 표현할 때 "번들링(Bundling)"이라고도 한다.

캡슐화는 관련된 데이터와 함수를 하나의 장치 속으로 묶어주기 때문에 어떤 개념을 추상화 시키는 데 있어서 큰 도움이 되며, 추상화가 잘 될 수록 프로그램의 모듈화를 향상시키는 결과를 초래한다.

○ 캡슐화의 개념

- 1) 클래스를 통해 캡슐을 정의한다.
- 2) 한 캡슐 내에는 서로 관련된 데이터와 함수가 묶여 있어야 한다.
- 3) 캡슐은 추상화를 표현하는 장치이다.

자바는 `class` 를 이용하여 데이터와 함수를 하나로 묶을 수 있다. 여기서의 데이터란 클래스의 속성 또는 객체의 멤버 필드를 의미하며, 함수는 클래스나 객체의 메서드를 의미한다.

클래스의 속성은 객체의 멤버 필드, 클래스의 행위는 객체의 메소드이다. 이때 객체의 멤버필드에 대하여 직접적으로 접근할 수 도 있지만 객체의 멤버필드에 대해 직접적인 접근보다는 메서드를 통한 접근을 이용하는 것이 바람직하다.

아래의 예에서는 객체의 멤버변수에 대해 직접적으로 값을 변경하고 있다.

```
-----
class Person {
    String name;
    int age;
    String phone;

    public void print() {

        System.out.println("===== " + name + " =====");
        System.out.println("나이 : " + age);
        System.out.println("전화 : " + phone);
    }
}

public class ClassTest {
    public static void main(String[] ar) {
```

```

    Person kim = new Person();
    kim.name = "김기희";
    kim.age = 20;
    kim.phone = "017-233-8409";
    kim.print();
}
}

```

위의 코드를 빌드 한 후, 실행하여 보면 이상 없이 실행됨을 확인할 수 있다. 그렇다면 위의 코드에는 어떤 문제가 있을까? 위의 클래스를 사용함에 있어서 다음과 같은 조건이 따른다고 가정해보자.

- 이름은 2자 이상 4자 이하의 문자열을 입력하여야 한다
- 나이 값은 0보다 크고 100보다 작아야 한다.

위의 클래스를 이용하여 객체를 생성하고 사용하는데 있어서 객체의 멤버필드에 대한 직접적인 접근의 허용은 위와 같은 조건의 검사에 대한 기회를 잃어버리게 된다. 위의 예제코드를 조금 수정하여 멤버필드의 데이터를 지정할 수 있는 메서드를 추가해보자.

```

class Person {
    String name;
    int age;
    String phone;

    public void setName(String name) {
        if (name.length() < 2 || name.length() > 4) {
            System.err.println("이름은 2자이상 4자이하하여야 합니다.");
            return;
        }

        this.name = name;
    }

    public void setAge(int age) {
        if (age < 0 || age > 100) {
            System.err.println("연령은 0보다 크고 100보다 작아야 합니다.");
            return;
        }

        this.age = age;
    }

    public void setPhone(String phone) {
        this.phone = phone;
    }

    public void print() {
        System.out.println("===== " + name + " =====");
        System.out.println("나이 : " + age);
        System.out.println("전화 : " + phone);
    }
}

public class ClassTest {
    public static void main(String[] ar) {

```

```
Person kim = new Person();
kim.setName("한");
kim.setAge(200);
kim.setPhone("017-233-8409");

kim.print();
}
```

위의 예제에서는 setName(), setAge(), setPhone()이라는 메서드를 제공하여 이 메소드들로 하여금 객체의 멤버의 값을 지정할 수 있도록 하였다. 이때 setName() 메서드에서는 입력된 이름이 2자 이상 4자 이하인가를 검사하며, setAge() 메서드에서는 1 부터 99까지의 값인가를 확인하고 조건에 일치할 경우에만 멤버 필드의 값을 지정하도록 하였다.

이때 name, age, phone이라는 속성(멤버 필드)은 Person 이라는 형태의 객체의 상태를 나타내는 속성이며, setName(), setAge, setPhone()은 Person 이라는 형태의 객체의 상태 값을 바꿀 수 있는 행위이므로 서로 연관되어 있으며, 이들 속성과 행위는 Person 이라는 클래스 하나로 묶여져 있는데 이것을 캡슐화라고 한다.

11.2 캡슐화의 특성

캡슐화의 특성은 다음과 같다.

- 1) 추상화의 단위가 된다.
- 2) 재사용의 단위가 된다.
- 3) 정보은닉을 실현하는 장치이다.

추상화란? 복잡한 문제를 다루기 위해서 불필요한 부분들은 숨기고 중요한 부분만을 표현하는 것을 의미한다. 예를 들어 위의 예제에서 이름값을 지정하는데 있어서 2자 이상 또는 4자 이하라는 조건을 어떻게 검사할 것인가? 또는 연령을 입력하는데 있어서 1 부터 100까지만의 값을 어떻게 입력 받을 것인가에 대한 부분은 Person 클래스를 사용하는 사용자 입장에서 알 필요가 없는 부분이다. Person 클래스를 사용하는 사용자 입장에서 알아야 할 것은 위의 클래스는 어떠한 기능이 있으며 이 기능을 어떻게 사용하면 된다 라는 것만 알면 되는 것이다.

객체지향에서 캡슐화의 단위는 클래스 단위로 정의되고, 실제 프로그램 실행 시에는 객체 단위로 생성되어 움직인다. 그러므로 자바와 같은 객체 지향 프로그램에서의 재사용의 단위는 클래스 혹은 객체 단위로 사용되므로 함수 단위로 재사용하는 것에 비해 훨씬 재사용성이 높아지게 된다.

뿐만 아니라 캡슐화를 통해 행위 데이터와 함수를 하나로 묶고 숨김으로써 정보은닉을 실현할 수 있다.

11.3 캡슐화와 정보은닉

객체 지향 개념 가운데 클래스와 객체와 더불어 많이 혼동하는 개념이 캡슐화와 정보은닉이다. 이 둘을 같은 의미로 혼용하여 사용하기도 하나 다음과 같이 구분할 수 있다.

- 1) 캡슐화는 관련된 요소들을 묶어 줌으로써, 캡슐의 내부와 외부를 구별 짓는 장치이다.
- 2) 정보은닉은 캡슐 내부의 요소들에 대한 세부 구현 사항을 외부로 부터 숨기는 장치이다.

캡슐화는 관련된 요소들을 묶음으로써 캡슐 내부와 외부를 구별 짓는 장치이다. 우리는 지금까지 `class`를 이용하여 서로 관련된 데이터와 함수를 하나로 묶는 캡슐화 방법에 대해 배웠다. 이러한 캡슐화는 캡슐의 내부와 외부의 구분을 가능하게 해준다. 그리고 캡슐 내부에 있는 요소들을 외부로부터 숨길 수 있는데 이것을 정보의 은닉이라고 한다.

정보의 은닉은 캡슐화 되어있는 데이터와 함수들에 대해서 외부에서 내부가 어떤 구조인지, 어떤 데이터와 함수를 가지고 있으며, 함수가 어떻게 구현되어 있는지에 대한 세부 사항을 숨기는 것이다. 즉 캡슐화 되어 있다고 해서 정보은닉이 되어 있는 것은 아니다.

11.4 정보은닉과 접근제한자

캡슐화가 관련된 데이터를 묶어주는 장치라면 정보은닉은 캡슐내부의 데이터와 함수를 외부에 노출시키지 않고 숨기는 장치를 말한다. 그러므로 정보은닉은 다음과 같이 정의할 수 있다.

◦ 정보은닉의 개념

- 1) 데이터와 함수를 객체 내부에 숨기는 장치이다.
- 2) 블랙박스이다.
- 3) 외부 객체와의 통신을 위한 공개 인터페이스를 지닌다.

정보은닉을 위해서는 캡슐화가 되어 있어야 한다. 이것은 캡슐의 내부와 외부로 구분할 수 있어야 하기 때문이다. 그리고 캡슐 내부의 데이터나 함수 일부를 외부로 부터 숨길 수 있는데 이를 정보은닉이라고 하였다. 하지만 내부의 데이터나 함수를 모두 숨겨버리면 해당 객체는 외부와 메시지를 주고 받을 수 없게 된다. 그러므로 모든 데이터와 함수를 숨기는 것이 아니라 외부에 노출 시킬 필요가 없는 데이터와 함수들만 은폐를 시켜야 한다. 뿐만 아니라 은폐를 시킨 데이터나 함수에 접근할 수 있는 방법, 즉 외부 객체들과 데이터를 주고 받을 수 있는 공개 인터페이스를 반드시 하나 이상 정의하여야 한다.

위의 예제에서 Person 클래스는 name, age, phone 이라는 속성을 가지고 있으며, 이 속성에 값을 지정할 수 있는 인터페이스 setName(), setAge(), setPhone() 이라는 메서드를 제공하고 있다. 하지만 name, age, phone 이라는 멤버가 은폐되어 있지 않으므로 사용자는 인터페이스가 아닌 멤버 필드에 대한 직접적인 접근이 가능하다. 즉 캡슐화는 되어 있지만 정보은닉은 구현되지 않은 상태이다.

아래의 예제는 위의 Person 클래스를 조금 수정하여, 멤버 필드에 대한 직접적인 접근을 불가능 하도록 하였다.

```
-----  
class Person {  
    private String name;  
    private int age;  
    private String phone;  
  
    public void setName(String name) {  
        if (name.length() < 2 || name.length() > 4) {  
            System.err.println("이름은 2자이상 4자이하여야 합니다.");  
            return;  
        }  
  
        this.name = name;  
    }  
  
    public void setAge(int age) {  
        if (age < 0 || age > 100) {  
            System.err.println("연령은 0보다 크고 100보다 작아야 합니다.");  
            return;  
        }  
  
        this.age = age;  
    }  
}
```

```

    public void setPhone(String phone) {
        this.phone = phone;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }

    public String getPhone() {
        return phone;
    }

    public void print() {
        System.out.println("===== " + name + " =====");
        System.out.println("나이 : " + age);
        System.out.println("전화 : " + phone);
    }
}

public class ClassTest {
    public static void main(String[] ar) {

        Person kim = new Person();
        kim.setName("김기희");
        kim.setAge(20);
        kim.setPhone("017-233-8409");

        kim.print();
    }
}

```

위의 예제를 보면 멤버필드 name, age, phone에 대하여 private 라는 접근제한자를 지정함으로써 멤버필드에 대한 접근을 오직 같은 클래스 내에서만 접근이 가능하도록 허용하였다. 클래스 외부 즉 캡슐 외부에서는 접근이 불가능 할 뿐만 아니라 객체 내부에 멤버 필드가 있는지조차 알 수가 없게 된다. 이때 private 멤버에 대해 접근할 수 있는 인터페이스를 제공하여야 하는데 이 인터페이스가 setXXX()와 getXXX()라는 메서드이며, 이러한 메서드를 Setter 그리고 Getter라고 부른다.

특히 private 멤버필드에 대한 접근방법을 제공하는 메서드만이 인터페이스라고 불리는 것은 아니다. 인터페이스는 내부에 숨겨진 특정작업을 수행하는 함수에 접근할 수 있는 기능을 제공할 수 도 있다.

◦ 정보은닉의 이점

- 1) 추상화를 향상시켜준다.
- 2) 내부 데이터나 알고리즘의 변경이 용이하다.
- 3) 모듈의 독립성을 높여준다.
- 4) 확장성을 높여준다.

객체의 추상화는 캡슐화와 정보의 은닉을 통해 이루어진다고 할 수 있다. 추상화란 복잡한 현실세계

를 간결하게 표현하는 것이라고 하였다. 즉 클래스를 이용하여 캡슐화를 하고, 클래스의 멤버에 대하여 접근제한자를 이용하여 정보의 은닉을 구현하고 간단한 인터페이스를 제공함으로써 복잡한 내부를 간결하게 만들 수 있는데 이것을 추상화라고 한다.

즉 정보의 은닉이 높으면 높을수록 추상화 정도는 높아지게 된다. 추상화 정도가 높다는 것은 내부의 복잡한 구조를 많이 숨겼으며, 이를 이용할 수 있는 간단한 인터페이스만을 제공함을 의미한다. 그러므로 인터페이스의 규격만 맞추면 캡슐내부의 데이터나 알고리즘을 변경하는데 있어서 상당히 용이하다.

모듈이란 프로그램의 재사용의 단위이다. 그러므로 c언어에서는 함수를 모듈이라고 할 수 있다. 객체지향 언어에서는 객체가 재사용의 단위이므로 객체를 모듈이라고 할 수 있다. 정확히는 객체를 생성해내기 위한 클래스가 모듈이다. 정보의 은닉은 이러한 모듈내의 데이터와 함수를 숨기고, 간단한 인터페이스만을 제공하므로 인터페이스가 변경되지 않는다면 얼마든지 데이터나 함수를 추가하거나 삭제할 수 있다. 이것은 곧 모듈의 독립성과 확장성을 높여 줌을 의미한다.

11.5 접근제한자

접근제한자란? 클래스내의 멤버에 대한 접근을 제한하기 위한 예약어이다.

◦ 접근제한자의 종류

private	동일한 클래스 내에서만 접근 가능
package	동일파일과 동일한 패키지(폴더)에서만 접근 가능
protected	동일파일과 동일 패키지(폴더), 상속관계에 있는 클래스에서 접근 가능
public	모든 영역에서 접근 가능

◦ private 접근제한자

private 접근제한자는 동일한 클래스 내에서만 접근이 가능하도록 해준다. 즉 클래스 외부에서는 멤버에 대한 접근이 완전히 차단된다. 여기서 혼동하지 말아야 할 것은 객체 외부에서의 접근을 무조건 차단하는 것은 아니다.

아래의 예제는 private 접근제한자의 특성을 보여주는 예제이다. Person 클래스의 name, age, phone 이라는 속성은 private 접근제한자가 지정되었으므로 클래스 외부에서의 접근은 불가능 하다. 뿐만 아니라 객체를 통한 직접적인 접근 또한 불가능하다. 그러나 동일한 클래스 내에서는 직접적인 접근이 가능하게 된다. 이처럼 동일한 클래스 내에서의 접근을 허용하는 것이 private 접근제한자이다.

```
-----
class Person {
    private String name;
    private int age;
    private String phone;

    public Person() {

    }

    public Person(Person p) {
        name = p.name;
        age = p.age;
        phone = p.phone;
    }

    public void setName(String name) {
        if (name.length() < 2 || name.length() > 4) {
            System.err.println("이름은 2자이상 4자이하여야 합니다.");
            return;
        }

        this.name = name;
    }

    public void setAge(int age) {
        if (age < 0 || age > 100) {
            System.err.println("연령은 0보다 크고 100보다 작아야 합니다.");
            return;
        }
    }
}
```

```

    }

    this.age = age;
}

public void setPhone(String phone) {
    this.phone = phone;
}

public String getName() {
    return name;
}

public int getAge() {
    return age;
}

public String getPhone() {
    return phone;
}

public void print() {
    System.out.println("===== " + name + " =====");
    System.out.println("나이 : " + age);
    System.out.println("전화 : " + phone);
}
}

public class ClassTest {
    public static void main(String[] ar) {

        Person kim = new Person();
        kim.setName("김기희");
        kim.setAge(20);
        kim.setPhone("017-233-8409");

        Person kim2 = new Person(kim);

        kim.print();
        kim2.print();
    }
}

```

◦ package 접근제한자.

package 접근제한자는 동일한 파일이나 동일한 폴더에서의 접근을 허용한다. 여기서 동일한 폴더라 함은 동일한 패키지를 의미한다. package 접근제한자는 별도로 접근제한자를 명시하지 않는다.

아래 예제의 Person 클래스의 멤버들은 별도로 접근제한자를 명시하지 않았다. 이처럼 별도의 접근제한자를 명시하지 않았을 때, package 접근제한자가 적용된다. package 접근제한자는 동일한 파일과 동일한 폴더에서 접근이 가능하므로 main() 메서드에서도 직접적인 접근이 가능하게 된다.

```

class Person {
    String name;
    int age;
    String phone;
}

```

```

public Person() {

}

public void print() {

    System.out.println("===== " + name + " =====");
    System.out.println("나이 : " + age);
    System.out.println("전화 : " + phone);
}

}

public class ClassTest {
    public static void main(String[] ar) {

        Person kim = new Person();
        kim.name = "김기희";
        kim.age = 30;
        kim.phone = "017-233-8409";

        kim.print();
    }
}

```

◦ protected 접근제한자

protected 접근제한자는 동일한 파일과 동일한 폴더 그리고 상속관계에 있는 자식 클래스에서 접근이 가능한 접근제한자이다. 상속에 대해서는 이후 자세하게 다룰 예정이므로 여기서는 간단하게 설명하도록 하겠다.

아래의 예제에서 Student 클래스는 Person 클래스를 상속하였다. 상속은 extends 라는 예약어를 이용하여 상속받을 클래스를 지정한다. 이때 Person 클래스를 상위 클래스 또는 부모 클래스, 슈퍼 클래스라고 하며 Student 클래스를 하위 클래스, 자식 클래스 또는 파생 클래스라고 한다.

상속관계는 Is-A 관계이다. 즉 "자식 클래스는 부모 클래스이다" 라는 공식이 성립된다. 그러므로 아래의 예제에서는 "Student는 Person이다"라는 공식이 성립된다. 그러므로 Student는 Person이가 지고 있는 모든 속성을 마치 자신의 속성처럼 사용할 수 있게 된다. 아래의 예제에서 Student 클래스의 생성자 메서드를 보면 Student 클래스에서 Person 클래스의 속성에 대해 직접적인 접근을 하고 있음을 볼 수 있다. 이처럼 동일한 파일과 동일한 폴더(패키지) 그리고 상속관계에 있는 클래스로부터의 접근을 허용하는 접근제한자가 protected 접근제한자이다.

```

class Person {
    protected String name;
    protected int age;
    protected String phone;

    public void print() {

        System.out.println("===== " + name + " =====");
        System.out.println("나이 : " + age);
        System.out.println("전화 : " + phone);
    }
}

```

```
class Student extends Person {
    private String part;

    public Student(String name, int age, String phone, String part) {
        this.name = name;
        this.age = age;
        this.phone = phone;
        this.part = part;
    }

    public void print() {
        super.print();
        System.out.println("과목 : " + part);
    }
}

public class ClassTest {
    public static void main(String[] ar) {

        Student kim = new Student("김기희", 30, "017-233-8409", "JAVA");
        kim.print();
    }
}
```

◦ public 접근제한자

public 접근제한자는 모든 영역에서 접근을 허용한다. 모든 메서드에 대해 public 접근제한자를 지정하는 것은 아니지만 인터페이스 역할을 담당하는 메서드에 대해서는 public 접근제한자를 지정하는 것이 관례이다.

11.6 멤버 필드와 멤버 메서드

◦ 멤버 필드

멤버 필드는 클래스의 구성요소로서 속성에 해당하며 객체화 되었을 때, 상태 값을 저장하기 위한 변수이다. 클래스의 멤버 필드는 다음과 같은 형식으로 선언된다.

```
[접근제한자] [지정예약어] 자료형 필드명[ = 값];
```

멤버 필드 값의 초기화는 생성자 메서드에서도 할 수 있다. 하지만 자바에서는 아래와 같이 멤버 필드의 선언과 동시에 값을 초기화 할 수 있다.

```
private int age = 20;
```

◦ 멤버 메서드

멤버 메서드는 멤버 필드와 함께 클래스의 구성요소로서 행위(기능)에 해당하며, 객체화 되었을 때 해당 객체와 정보를 교환할 수 있는 인터페이스 역할을 하게 된다. 멤버 메서드의 정의 형식은 다음과 같다.

```
[접근제한자] [지정예약어] 리턴결과형 메서드명(매개변수목록)
                                [throws 예외클래스목록] {
    메서드 내용부...
}
```


12. 객체

12.1 객체의 개념 및 특성

○ 객체의 개념

객체에 대한 개념은 보는 관점에 따라 달리 표현된다. 분석 및 설계단계에서는 물리적 사물 또는 논리적 개념이라고 하며, 구현 관점에서는 객체를 모듈이라고 한다. 추상화 단위로서의 객체는 캡슐화와 정보은닉의 도구이다.

- 1) 객체는 물리적 사물 또는 논리적 개념이다.
- 2) 객체는 모듈이다.
- 3) 객체는 캡슐화와 정보은닉의 도구이다.

객체 지향 프로그래밍에 있어서 프로그램의 주체는 클래스가 아닌 객체이다. 물론 클래스를 틀로 하여 객체를 생성하지만 클래스는 상태를 가지지 않는 반면, 객체는 상태 값을 가지기 때문이다. 또한 하나의 클래스로부터 여러 개의 객체를 생성할 수 있으므로 객체는 재사용의 단위가 된다. 그러므로 객체는 모듈이라고 할 수 있다.

이러한 객체는 데이터와 관련된 함수를 포함한 하나의 모듈이므로 캡슐화 장치이자 정보은닉의 도구라고 할 수 있다.

객체 지향 개념	분석 및 설계	객체 지향 언어
객체	객체	객체
속성(Attribute)	속성(Property)	멤버 필드, 멤버 변수
행위(Behavior)	메소드(Method)	멤버 메소드, 오퍼레이션

○ 객체의 특성

객체는 다음과 같은 특성을 갖는다.

- 1) 객체는 유일한 식별자를 가져야 한다.
- 2) 객체는 상태를 가져야 한다.
- 3) 객체는 잘 정의된 함수를 가진다.

객체는 클래스로부터 생성되며, 하나의 클래스로부터 여러 개의 객체가 생성될 수 있음을 배웠다. 하나의 클래스로부터 생성되는 여러 개의 객체는 각각의 객체를 구별하기 위한 식별자(객체명)가 필요하며 이 식별자는 고유한 이름이어야 한다.

클래스로부터 생성된 객체는 클래스에 선언된 속성에 대한 값을 가진다. 이것은 객체의 생성은 클래스에 선언된 속성값을 저장하기 위한 메모리 공간이 할당됨을 의미한다. 그리고 객체는 이 속성에 대한 값을 가지게 된다.

객체는 잘 정의된 함수를 가진다. 자바에서는 이 함수를 메소드라고 부르는데 이 메소드는 외부와

정보를 주고 받을 수 있는 인터페이스 역할을 한다.

- 객체의 생명주기

객체는 아래와 같이 3단계의 생명주기를 가진다.

객체의 생성 -> 객체의 사용 -> 객체의 소멸

객체의 생성이란 클래스를 이용하여 메모리의 특정 주소에 공간을 확보하는 단계를 말한다. 이후 메모리를 이용하여 메모리 공간에 데이터를 기록하거나 변경하고, 기록된 데이터를 전송하게 되는데 이를 메시지를 송수신한다 라고도 한다.

자바에서는 C나 C++과는 달리 소멸자를 호출 할 필요가 없다. 더 이상 참조되지 않는 객체는 Garbage Collector에 의해 자동으로 소멸된다.

12.2 객체의 생성

객체는 클래스내의 객체 생성 함수 또는 생성자 메서드에 의해 생성된다. 아래의 예는 Person 클래스의 생성자 함수를 호출함으로써 Person 클래스의 객체를 생성하는 예이다.

```
Person p = new Person();
```

객체의 생성은 다음과 같은 3단계에 의해 이뤄진다.

- 1) 객체의 선언 : Person p
- 2) 객체의 인스턴스화 : 메모리 공간의 할당
- 3) 객체의 상태 초기화 : Person() 생성자 메서드의 호출

12.3 자기 자신의 객체를 참조하는 this

this는 클래스 내부에서 사용되어지며 객체 자신을 가리키는 객체이다. 아래의 예제를 보면 Student 클래스를 이용하여 kim와 lee라는 두 개의 객체가 생성되었으며 Student 클래스의 생성자 메서드에서는 this라는 키워드를 이용하고 있다.

이 this는 kim 이라는 객체 내에서의 this는 kim객체를 가리키며, lee 라는 객체 내에서의 this는 lee객체를 가리키는 객체이다.

```
-----
class Person {
    protected String name;
    protected int age;
    protected String phone;

    public void print() {

        System.out.println("===== " + name + " =====");
        System.out.println("나이 : " + age);
        System.out.println("전화 : " + phone);
    }
}

class Student extends Person {
    private String part;

    public Student(String name, int age, String phone, String part) {
        this.name = name;
        this.age = age;
        this.phone = phone;
        this.part = part;
    }

    public void print() {
        super.print();
        System.out.println("과목 : " + part);
    }
}

public class ClassTest {
    public static void main(String[] ar) {

        Student kim = new Student("김기희", 30, "017-233-8409", "JAVA");
        Student lee = new Student("이우창", 45, "000-0000-0020", "Maya");

        kim.print();
        lee.print();
    }
}
-----
```

12.4 static 예약어

`static`은 정적이라는 의미로써 객체의 생성 없이 바로 사용 가능한 멤버를 선언할 때 사용되는 예약어이다.

◦ static 필드

클래스로부터 파생된 객체가 데이터 값을 공유하기 위해 선언하는 공간으로써 클래스 변수라고도 한다. `static` 필드는 이름에서도 알 수 있듯이 동적이 아닌 정적이라는 의미를 가진다. 즉 멤버 필드는 인스턴스를 생성(객체화)하여야만이 사용할 수 있다. 우리는 객체를 생성할 때 `new`라는 예약어를 이용하였는데 `new`예약어는 동적 할당을 하는 예약어이다. 반면 `static` 필드는 객체를 생성하지 않고도 사용할 수 있다.

그렇다면 어떻게 객체를 생성하지 않고도 사용할 수 있을까? 그 이유는 `static` 필드는 자바 프로그램이 구동될 때 메모리 공간을 할당 받기 때문이다.

아래 예제의 `count` 는 `static` 멤버 필드이다. `static` 멤버 필드는 클래스 변수라고 하여 "클래스명.`static` 필드명" 형식으로 객체를 생성하지 않고도 사용이 가능하다. 물론 객체를 통한 접근도 가능하지만 `static` 멤버 필드이므로 클래스를 통한 접근이 원칙이다.

```
-----
class Person {
    public static int count;

    protected String name;
    protected int age;
    protected String phone;

    static {
        count = 0;
    }

    public Person(String name, int age, String phone) {
        Person.count++;
        this.name = name;
        this.age = age;
        this.phone = phone;
    }

    public void print() {
        System.out.println("===== " + name + " =====");
        System.out.println("나이 : " + age);
        System.out.println("전화 : " + phone);
    }
}

public class ClassTest {
    public static void main(String[] ar) {

        System.out.println("현재 생성된 객체의 수 : " + Person.count);
        Person kim = new Person("김기희", 30, "017-233-8409");
        Person lee = new Person("이우창", 45, "031-3432-3321");
        kim.print();
        lee.print();
    }
}
```

```

        System.out.println("현재 생성된 객체의 수 : " + Person.count);
    }
}

```

static 필드는 static 초기화 영역에서 초기화를 하지만 멤버 필드와 마찬가지로 선언과 동시에 초기화가 가능하다.

```

class Person {
    public static int count = 0;

    protected String name;
    protected int age;
    protected String phone;

    public Person(String name, int age, String phone) {
        Person.count++;
        this.name = name;
        this.age = age;
        this.phone = phone;
    }

    public void print() {
        System.out.println("===== " + name + " =====");
        System.out.println("나이 : " + age);
        System.out.println("전화 : " + phone);
    }
}

public class ClassTest {
    public static void main(String[] ar) {

        System.out.println("현재 생성된 객체의 수 : " + Person.count);
        Person kim = new Person("김기희", 30, "017-233-8409");
        Person lee = new Person("이우창", 45, "031-3432-3321");
        kim.print();
        lee.print();
        System.out.println("현재 생성된 객체의 수 : " + Person.count);
    }
}

```

o static 메서드

static 메서드는 static 필드에 접근할 수 있는 인터페이스를 제공하는 것이 목적이다. 위에서 static 필드는 객체를 생성하지 않고도 사용할 수 있다고 하였다. 이것은 객체를 생성하기 전에 필드의 공간이 할당되었음을 의미한다. 그러므로 객체를 생성하지 않고도 static 필드에 접근할 수 있는 방법을 제공하여야 하는데 이것이 static 메소드이다.

물론 멤버 메서드에서도 static 필드에 대한 접근이 가능하다. 이것은 멤버 메서드를 포함한 객체가 생성되기 이전에 static 필드에 대한 공간의 할당과 초기화가 이뤄지기 때문이다. 반면 static 메서드는 객체의 멤버 필드나 멤버 메서드에 대한 참조가 불가능 하다. 그 이유 역시 멤버 필드나 멤버 메서드의 사용은 객체를 생성하여야만이 사용할 수 있기 때문이다. 뿐만 아니라 static 메서드 내에서는

this 객체 역시 참조가 불가능 하다. this는 이미 생성된 객체 자신을 가리키는 객체이기 때문이다.

아래의 예제는 private 접근제한자가 지정된 static 필드에 값을 참조할 수 있는 인터페이스로써 getCount() 라는 static 메소드를 제공하였다.

```
-----
class Person {
    private static int count = 0;

    protected String name;
    protected int age;
    protected String phone;

    public static int getCount() {
        return Person.count;
    }

    public Person(String name, int age, String phone) {
        Person.count++;
        this.name = name;
        this.age = age;
        this.phone = phone;
    }

    public void print() {
        System.out.println("===== " + name + " =====");
        System.out.println("나이 : " + age);
        System.out.println("전화 : " + phone);
    }
}

public class ClassTest {
    public static void main(String[] ar) {

        System.out.println("현재 생성된 객체의 수 : " + Person.getCount());
        Person kim = new Person("김기희", 30, "017-233-8409");
        Person lee = new Person("이우창", 45, "031-3432-3321");
        kim.print();
        lee.print();
        System.out.println("현재 생성된 객체의 수 : " + Person.getCount());
    }
}
-----
```

12.5 final 예약어

final 지정예약어는 class와 멤버 필드, 멤버 메소드 모두에 지정할 수 있으며, 어느 요소에 지정하는가에 따라 그 용도가 달라지게 된다.

◦ final 클래스

클래스를 선언할 때 final 지정예약어를 사용할 수 있다. 이때 final로 지정된 클래스는 상속이 불가능 하게 된다. 아래는 이전에 배웠던 System 클래스의 API Document 이다. 이 Document에서는 System 클래스가 final 클래스임을 나타낸다. 즉 System 클래스는 상속이 불가능 하다.

◦ final 멤버 필드

멤버 필드를 final로 선언하는 경우 해당 필드의 값은 더 이상 바꿀 수 없게 된다. 즉 멤버 필드를 상수화 시킨다. 그러므로 final로 지정된 멤버필드는 반드시 선언과 동시에 초기값을 지정하여야만 한다. 통상적으로 final 멤버 필드의 이름은 모두 대문자로 선언한다.

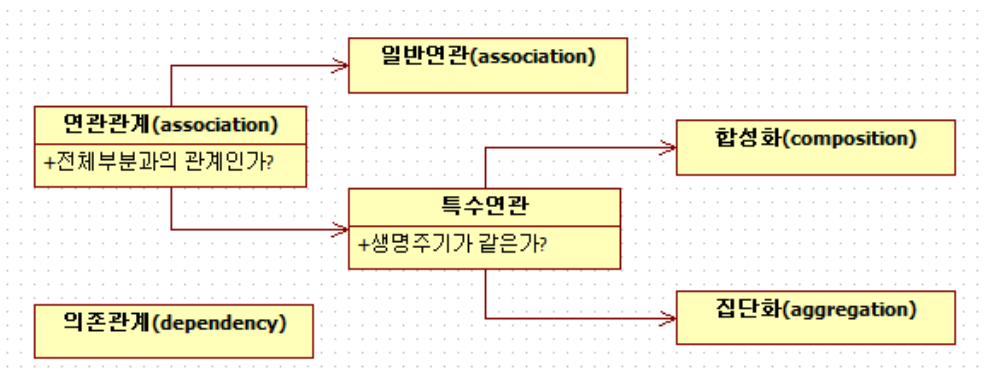
예) `public static final float PI = 3.14f;`

◦ final 메소드

멤버 메소드를 final로 선언하면 이 메서드는 오버라이딩이 불가능 하게 된다. 오버라이딩은 상속에 대한 내용을 배워야 이해를 할 수 있는 부분이므로 이 내용은 상속에 대한 강의 시 진행하도록 하자.

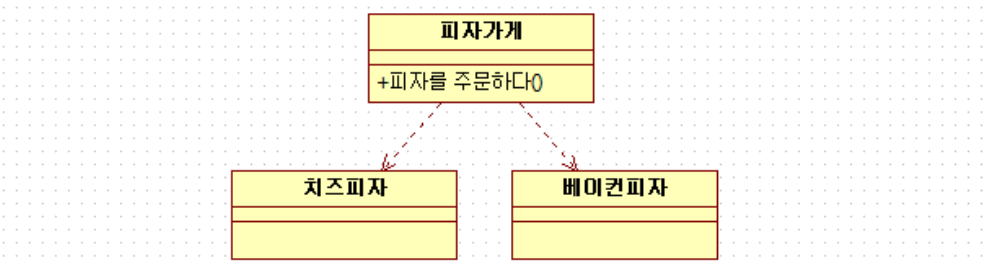
12.6 연관관계

객체지향 프로그램에는 다음과 같은 연관관계가 있다.



12.6.1 의존관계 (dependency)

의존관계와 연관관계의 구분은 사용되어지는 클래스가 반드시 사용되는 클래스에 필요한가의 차이이다. 예를 들어 피자 가게에서 판매하는 피자가 치즈피자와 베이컨피자가 있다면 치즈피자와 베이컨피자는 피자 가게에 없어서는 안 되는 객체이다. 이 경우 피자 가게는 치즈피자와 베이컨 피자에 의존적이라고 하며 아래와 같이 표기한다.



위의 UML에서 치즈피자와 베이컨피자는 피자 가게에 있어서 없어서는 안 되는 객체이다. 즉 새로운 피자가 추가된다면 피자 가게의 코드가 변경되어야 한다. 물론 기존의 피자가 없어질 경우에도 피자 가게의 코드는 변경이 되어야 만 한다. 위와 같은 관계를 의존관계라고 하며 피자 가게는 여러 피자에 대해 의존적인 1:다의 의존관계가 성립된다.

위와 같은 1:다의 의존관계는 객체지향 프로그램에 있어서 그리 바람직하지 못하다. 위에서도 설명하였지만 새로운 피자 제품이 출시되거나 기존의 피자 제품 판매가 종료되면 그에 맞추어 피자 가게의 프로그램코드 또한 바뀌어야 하기 때문이다.

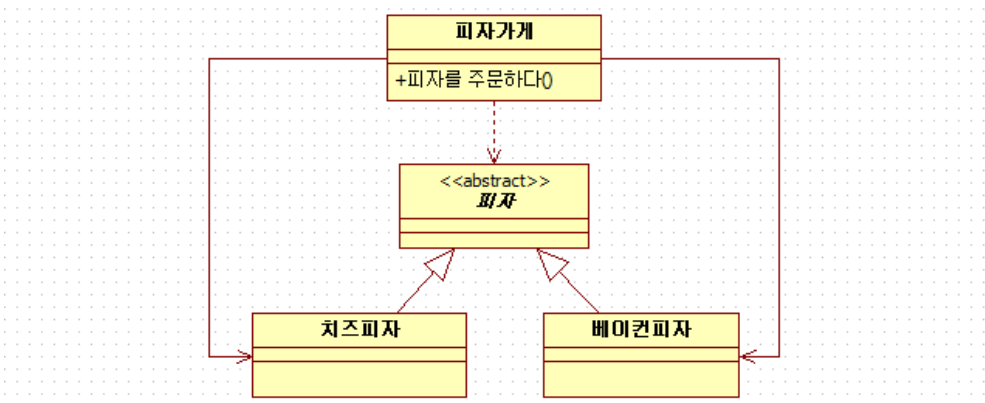
의존관계는 다음과 같은 상황에서 발생할 수 있다.

- 한 클래스의 메서드가 다른 클래스의 객체를 인자로 받아 그 객체의 메서드를 호출한다.

- 한 클래스의 메서드가 다른 클래스의 객체를 반환한다.
- 메서드 내에서 다른 클래스의 메서드가 반환하는 객체의 메서드를 호출한다.

12.6.2 연관관계

연관관계는 의존관계와는 달리 어느 한 클래스다 다른 클래스와 연관이 있음을 의미한다. 위의 피자 가게의 경우 여러 피자들에 대해 의존적이다. 의존적이라는 의미는 피자가게에 있어서 치즈피자나 베이컨피자가 없어서는 안 된다는 것을 의미한다. 위의 문제는 아래와 같이 일반화(상속)을 이용하여 해결할 수 있다.



위의 UML에서 치즈피자와 베이컨피자를 일반화한 피자라는 추상메서드를 만들고, 이 추상메서드를 치즈피자와 베이컨피자가 상속하고 있다. 일반화(상속)의 특성에 의해 하위 클래스의 인스턴스는 상위 클래스형으로 upcasting이 가능해진다. 즉 아래와 같은 내용이 성립된다.

- 치즈피자는 피자이다.
- 베이컨피자는 피자이다.

그러므로 피자가게에서는 치즈피자의 인스턴스와 베이컨피자의 인스턴스를 피자라는 타입으로 취급할 수 있게 된다. 여기서 피자가게의 치즈피자와 베이컨피자에 대한 1:다의 의존관계가 피자가게와 피자(추상클래스 또는 인터페이스)간의 1:1의 의존관계로 변경된 것을 확인할 수 있다. 즉 피자가게 입장에서 실제 피자가 어떤 피자인가 알 필요도 없으며, 새로운 피자가 추가되거나 삭제되는 것에 영향을 받지 않게 된다. 이때 피자가게와 피자는 의존관계이지만 피자가게에서 실제 취급하는 인스턴스는 치즈피자나 베이컨피자 이므로 피자가게와 치즈피자, 피자가게와 베이컨피자는 연관(association)이다. 이것은 피자라는 추상 클래스의 메서드를 오버라이딩(overriding)을 통해 치즈피자나 베이컨피자의 메서드를 호출하기 때문이다.

연관관계는 다음과 같이 분류된다.

- 단 방향 연관관계
- 양방향 연관관계

단 방향 연관관계인가, 양방향연관관계인가의 구분은 연결된 클래스가 서로의 존재를 알고 있는가? 로써 구분된다.

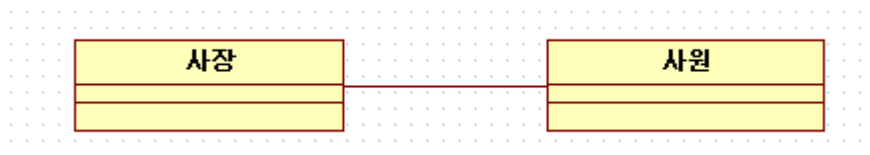
○ 단 방향 연관관계

아래는 단 방향 연관관계를 나타낸다. 전자제품을 취급하는 대리점이 있다면 대리점은 제품을 알고 있어야 하지만 제품은 어느 대리점인지 알 필요가 없다.



○ 양방향 연관관계

아래는 양방향 연관관계를 나타낸다. 사장과 사원이 있다면 사장은 업무를 지시하기 위해서 자기 사원을 알고 있어야 하며, 사원 역시 업무보고를 하기 위해서 자기 사장을 알고 있어야만 한다.



12.7 복합객체

12.7.1 복합객체의 개념

객체지향에서 객체란 단순히 데이터와 오퍼레이션만을 가지는 작은 단위 뿐만 아니라 데이터 부분에 다른 객체를 가질 수 도 있다. 이와 같은 관계에 있을 때 이를 복합화(Composition) 또는 복합(포함) 관계라고 한다. 객체지향에서 복합화란 한 객체가 다른 객체를 포함하는 것을 말한다.

객체가 객체를 포함할 경우, 다른 객체를 포함하는 객체를 전체객체(Whole Object)라고 하고, 포함되는 객체를 부분객체(Part Object)라고 한다. 복합객체는 부분객체들이 존재하여야 만 객체로서 의미를 가지며, 복합객체에 포함된 부분객체는 복합객체에 종속적이다.

- 한 객체가 다른 객체를 포함하는 것을 말한다.
- 복합객체는 포함하는 부분객체들이 없이는 의미 있는 객체가 되지 못한다.
- 복합객체에 포함되는 부분객체는 복합객체에 종속적이다.

12.7.2 복합객체의 특성

객체지향에서 객체들의 관계는 여러 가지 형태로 존재할 수 있다. 대표적인 관계로서 연관(Association)관계, 포함(Composition, Aggregation)관계, 상속(Inheritance)관계 등을 들 수 있다. 여기서 말하는 복합관계는 다음과 같은 상황들이 속한다.

- 복합객체는 객체가 선언되는 시점에서 발생한다.
- 합성화는 클래스의 객체 생성 함수 내에서 발생한다.
- 집단화는 실제 해당 객체를 사용하기 직전에 발생한다.

1) 객체가 선언되는 시점에서 발생한다.

객체가 선언되는 시점이란 전체객체 내의 데이터 부분에 부분객체가 정의되는 것을 말한다.

2) 클래스의 객체 생성 함수 내에서 발생한다.

객체들 간의 복합관계는 복합객체를 생성하는 객체 생성 함수 내에서 부분객체가 함께 생성될 수 있다. 이 경우 복합관계에서 전체객체와 부분객체 사이의 의존도가 가장 강한 관계이다. 이 경우는 전체객체와 부분객체의 생명주기가 동일하기 때문에 의존도가 더 강하다라고 표현하고 있다. 전체객체와 부분객체의 생명주기가 동일하다는 것은 전체객체의 생성과 더불어 부분객체도 함께 생성이 되고, 전체객체가 소멸이 되면 부분객체도 함께 소멸된다.

아래의 예제에서 전체객체인 Knight와 부분객체인 weapon 은 함께 생성되고 소멸된다.

```
public class Knight {
    private Weapon weapon;

    public Knight(){
```

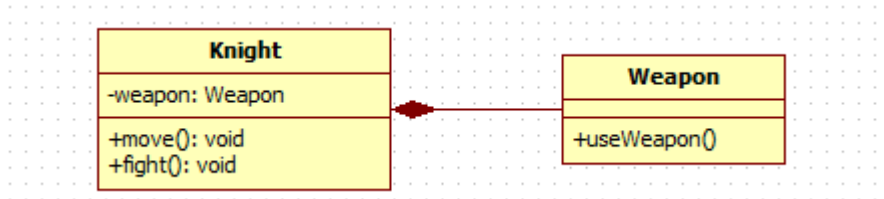
```

    weapon = new Sword();
}

public void move(){ ... }
public void fight(){ ... }
}

```

위의 예에서 처럼 부분객체를 생성자에서 생성하면 전체객체를 생성함과 동시에 부분객체가 생성된다. 위와 같은 관계를 **합성화(Composition)**하고 하며 UML에서는 아래와 같이 표기한다.



3) 실제 해당 객체를 사용하기 직전에 발생한다.

객체들 간의 복합관계는 복합관계가 필요한 시점에 부분객체를 포함 시킬 수 도 있다. 이는 복합객체 내의 특정 메서드들 가운데 특정 메서드를 수행할 때 해당 메서드 내에서 부분객체를 생성하여 사용하는 경우이다.

아래의 예제에서는 부분객체의 생성을 `fight()` 메서드 호출 시 생성을 한다. 아래의 복합객체는 전체객체를 생성할 때 부분객체가 함께 생성되지 않지만 전체객체와 부분객체는 함께 소멸된다.

```

public class Knight {
    private Weapon weapon;

    public Knight(){
        ...
    }

    public void move(){ ... }
    public void fight(){
        weapon = new Sword();
    }
}

```

때로는 부분객체를 클래스 내에서 생성하지 않고 외부에서 만들어진 객체를 인수로 넘겨받아 사용할 수 도 있다. 아래의 예에서 `setWeapon()` 메서드는 호출 시 넘겨받은 인수로 부분객체를 생성한다. 아래의 경우는 외부로부터 생성된 객체를 인수로 넘겨받아 부분객체를 생성하므로 전체객체와 함께 부분객체가 생성되지도 않으며, 전체객체가 소멸될 때 부분객체가 함께 소멸되지도 않는다.

```

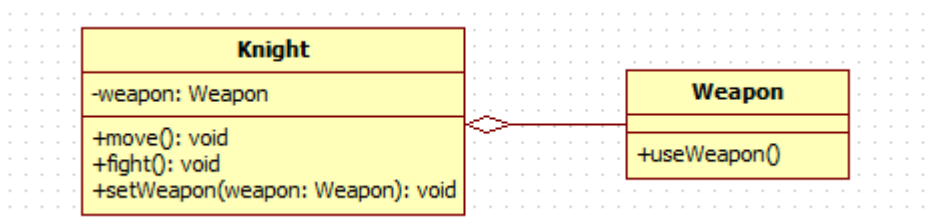
public class Knight {
    private Weapon weapon;

    public Knight(){

```

```
    ...  
}  
  
public void setWeapon(Weapon weapon){  
    this.weapon = weapon;  
}  
  
public void move(){ ... }  
public void fight(){ ... }  
}
```

위와 같은 복합객체를 **집단화(Aggregation)**로 정의하고 UML에서는 아래와 같이 표기한다.



13. 중첩 클래스 (Inner Class)

중첩 클래스란? 클래스 내에 또 다른 클래스가 정의 된 것을 말한다. 중첩클래스는 Inner Class 또는 Nested Class 라고도 불리며 종류는 아래와 같다.

- 중첩 클래스 (Inner Class)
- 정적 중첩 클래스 (Static Inner Class)
- 지역 중첩 클래스 (Local Inner Class)
- 익명 중첩 클래스 (Anonymous Inner Class)

13.1 중첩 클래스 (Inner Class)

중첩 클래스는 클래스 내에 또 다른 클래스를 정의 함으로써 클래스 관리의 효율을 높인 것이다. 중첩 클래스는 아래와 같은 형식으로 정의되며 멤버로써 static 멤버를 포함할 수 없다.

```
class Outer {
    class Inner {
        ....
    }
    ....
}
```

위의 형식에서 클래스 내부에 정의 된 Inner 클래스는 보통 Outer 클래스 내부에서만 사용되는 것이 보통이나 아래의 형식으로 외부에서도 객체를 생성하여 사용할 수 있다.

```
Outer.Inner obj = new Outer().new Inner();
```

Outer 클래스 내에 정의 된 Inner 클래스는 Outer 클래스의 멤버이지만 이 역시 클래스이므로 객체를 생성하여야만 사용이 가능하며, Inner 클래스의 객체 생성을 위해서는 Outer의 객체가 필요하다. 그러므로 우선 Outer 객체를 생성한 후, 생성된 Outer 객체를 통해 Inner 클래스의 객체를 생성하게 된다.

중첩 클래스는 이미 생성된 Outer 클래스의 객체를 통해서도 생성이 가능하다. 아래의 구문은 Outer 클래스의 객체를 생성하고, 이 객체를 이용하여 Inner 클래스의 객체를 생성하는 구문이다.

```
Outer o = new Outer();
Outer.Inner i1 = o.Inner();
Outer.Inner i2 = o.Inner();
```

위에서 i1과 i2는 Outer 클래스내의 Inner 클래스형 이므로 Outer.Inner 라는 클래스형 객체가 된다. Inner 클래스의 객체는 Outer 클래스의 객체로 부터 생성이 되므로 Inner 클래스 내에서는 Outer 클래스의 멤버를 사용할 수 있게 된다. 위의 경우 i1과 i2는 Outer 클래스의 객체 o 로 부터 생성이 되었으므로 객체 i1과 i2는 객체 o의 멤버를 공유하여 사용하게 된다.

아래 예제의 Person 클래스는 Job 클래스를 포함하고 있으며, Person 클래스에서는 Job 클래스의 객체를 생성하여 사용하고 있다.

```
-----
class Person {
    private String name;
    private int age;
    private Job job;

    class Job {
        private String jobname;

        public Job(String jobname) {
            this.jobname = jobname;
        }

        public void setJobName(String jobname) {
```



```

        this.jobname = jobname;
    }

    public String getJobName() {
        return jobname;
    }

    public void showJobInfo() {
        System.out.println(name + "님의 직업은 " + jobname + "입니다.");
    }
}

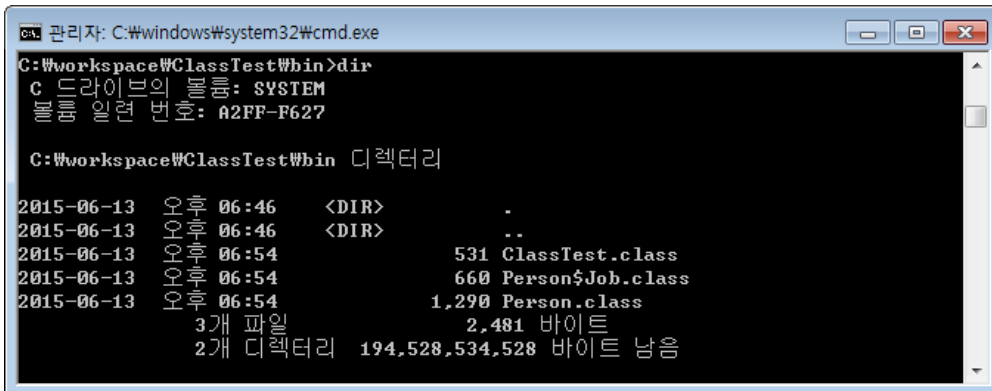
public Person(String name, int age, String jobname) {
    this.name = name;
    this.age = age;
    this.job = new Job(jobname);
}

public void showJobInfo() {
    job.showJobInfo();
}
}

public class ClassTest {
    public static void main(String[] ar) {
        Person kim = new Person("김기희", 30, "강사");
        kim.showJobInfo();
    }
}

```

위의 예제파일을 컴파일 한 후, 실행하면 아래와 같은 클래스 파일이 생성된다.



```

C:\workspace\ClassTest\bin>dir
C 드라이브의 볼륨: SYSTEM
볼륨 일련 번호: A2FF-F627

C:\workspace\ClassTest\bin 디렉터리

2015-06-13 오후 06:46 <DIR>      .
2015-06-13 오후 06:46 <DIR>      ..
2015-06-13 오후 06:54             531 ClassTest.class
2015-06-13 오후 06:54             660 Person$Job.class
2015-06-13 오후 06:54             1,290 Person.class
                3개 파일              2,481 바이트
                2개 디렉터리 194,528,534,528 바이트 남음

```

중첩파일은 컴파일 하면 외부클래스명\$내부클래스명.class 라는 파일로 클래스파일이 생성된다.

클래스 내부에 선언된 중첩 클래스는 외부 클래스의 static 멤버를 포함한 모든 멤버를 참조할 수는 있지만 자신은 static 멤버를 가질 수 없다. 그 이유는 static 멤버는 클래스의 객체를 생성하지 않고도 클래스명을 이용하여 직접적인 접근을 허용한다고 하였다. 즉 내부에 정의된 중첩 클래스 자체가 static 클래스가 아니므로 객체의 생성 없이는 중첩 클래스에 접근이 불가능 하며 static 멤버의 포함을 허용하지 않는다.

13.2 정적 중첩 클래스 (Static Inner Class)

정적 중첩 클래스는 내부의 클래스를 정의하는데 있어서 static 지정예약어를 지정하여 선언한다. 정적 중첩 클래스는 static 멤버를 포함한 모든 멤버를 가질 수 있으나 외부(Outer) 클래스의 non-static 멤버는 참조할 수 없다.

```
class Outer {
    static class Inner {
        ....
    }
    ....
}
```

정적 중첩 클래스는 클래스가 static 으로 선언되었기 때문에 외부 클래스를 객체화 하지 않고도 클래스를 통해 직접적인 객체의 생성이 가능하다.

```
Outer.Inner obj = new Outer.Inner();
```

아래의 예제코드는 정적 중첩 클래스의 사용 예이다. Person 클래스 내에 선언된 Job 클래스는 static 클래스 이므로 Person 클래스의 non-static 멤버는 참조할 수 가 없다.

```
-----
class Person {
    private String name;
    private int age;
    private Job job;

    static class Job {
        private String jobname;

        public Job(String jobname) {
            this.jobname = jobname;
        }

        public void setJobName(String jobname) {
            this.jobname = jobname;
        }

        public String getJobName() {
            return jobname;
        }
    }

    public Person(String name, int age, Job job) {
        this.name = name;
        this.age = age;
        this.job = job;
    }

    public void showJobInfo() {
        System.out.println(name + "님의 직업은 " + job.getJobName() + "입니다.");
    }
}

public class ClassTest {
    public static void main(String[] ar) {
```

```

    Person.Job job = new Person.Job("강사");
    Person kim = new Person("김기희", 30, job);
    kim.showJobInfo();
}
}
}

```

그렇다면 일반 정첩 클래스와 정적 중첩 클래스 중 어느 형식을 사용하여야 할 것인가? 일반 중첩 클래스는 외부 클래스의 멤버를 사용할 수 있다고 하였다. 내부 클래스에서 외부 클래스의 멤버를 참조하게 되면 내부 클래스는 외부클래스에 종속되어버린다. 즉 외부 클래스의 객체 없이 내부 클래스의 객체만으로는 정상적인 프로그램의 구현이 힘들게 된다. 만약 내부 클래스의 객체가 외부 클래스의 객체와 함께 사용되어야 하는 경우라면 일반 중첩 클래스를 사용하여도 문제는 없을 것이다. 하지만 클래스 내부에 선언된 중첩 클래스만을 객체화 하여 독립적으로 사용하고자 한다면 이 경우에는 static 중첩 클래스를 사용하여야 할 것이다.

정적 중첩 클래스 역시 "외부클래스명\$내부클래스명.class" 로 파일이 생성된다.

```

관리자: C:\windows\system32\cmd.exe
C:\workspace\ClassTest\Wbin>dir
C 드라이브의 볼륨: SYSTEM
볼륨 일련 번호: A2FF-F627

C:\workspace\ClassTest\Wbin 디렉터리

2015-06-13 오후 06:46 <DIR>      .
2015-06-13 오후 06:46 <DIR>      ..
2015-06-13 오후 07:15              662 ClassTest.class
2015-06-13 오후 07:15              585 Person$Job.class
2015-06-13 오후 07:15             1,054 Person.class
                3개 파일              2,301 바이트
                2개 디렉터리 194,523,860,992 바이트 남음

```

13.3 지역 중첩 클래스 (Local Inner Class)

지역 중첩 클래스는 많이 사용되지 않는 형식이지만 다음에 설명할 익명 중첩 클래스(Anonymous Inner class)의 기본 형식이므로 알아보도록 하자.

지역 중첩 클래스는 메서드 내에서 선언된 클래스를 말한다.

```
class Outer {
    method() {
        class Inner {
            .....
        }
    }
}
```

지역 중첩 클래스는 메소드 내부에서 정의하였으므로 해당 메소드 이 외의 영역에서는 객체화가 불가능하다. 즉 클래스가 정의된 메소드 내에서 객체화하여 사용되고 소멸됨을 의미한다.

아래 예제의 Person 클래스의 doEat() 메서드에는 Food 라는 클래스가 정의되어 있다. 이렇게 메서드 내에서 정의 된 클래스를 지역 중첩 클래스라고 하며 클래스가 선언된 doEat() 메서드 내에서만 사용이 가능하다. 그렇다면 외부에서 사용할 수 도 없는 지역 중첩 클래스를 사용하는 이유는 무엇일까? 아래의 예제를 보면 Person 클래스의 doEat() 메서드를 호출하면 단순히 음식을 먹는 것에 그치지 않고 음식을 만들고, 먹고, 설거지 까지 진행하게 된다. 이 처럼 메서드 내의 알고리즘이나 로직을 객체 단위로 실행하기 위해 사용하는 것이 지역 중첩 클래스이다.

```
-----
class Person {

    public void doEat(String food) {
        class Food {
            String food;

            public Food(String food) {
                this.food = food;
            }

            public void makeFood() {
                System.out.println(food + "요리를 합니다.");
            }

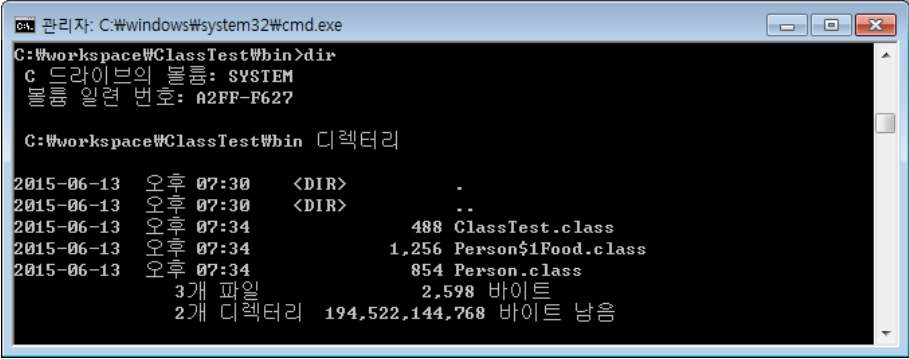
            public void doDishes() {
                System.out.println("설거지를 합니다.");
            }

            public void doEat() {
                makeFood();
                System.out.println(food + "를 점심으로 먹었습니다.");
                doDishes();
            }
        }

        if (food == null || food.trim().equals("")) {
            System.out.println("재료를 주시기 바랍니다.");
            return;
        }
    }
}
```

```
        Food f = new Food(food);  
        f.doEat();  
    }  
}  
  
public class ClassTest {  
    public static void main(String[] ar) {  
        Person kim = new Person();  
        kim.doEat("라면");  
    }  
}
```

지역 중첩 클래스는 "외부클래스명\$일련번호내부클래스명.class" 라는 이름으로 클래스 파일이 생성된다.



```
C:\workspace\ClassTest\bin>dir  
C 드라이브의 볼륨: SYSTEM  
볼륨 일련 번호: A2FF-F627  
  
C:\workspace\ClassTest\bin 디렉터리  
  
2015-06-13 오후 07:30 <DIR> .  
2015-06-13 오후 07:30 <DIR> ..  
2015-06-13 오후 07:34          488 ClassTest.class  
2015-06-13 오후 07:34       1,256 Person$1Food.class  
2015-06-13 오후 07:34          854 Person.class  
                3개 파일                2,598 바이트  
                2개 디렉터리 194,522,144,768 바이트 남음
```

13.4 익명 중첩 클래스 (Anonymous Inner class)

익명 중첩 클래스는 외부에서 정의된 클래스의 특정 메서드를 재정의(오버라이딩)하여 사용하는 방식으로 지역 중첩 클래스와 마찬가지로 메서드 내에서 정의되고 사용되어진다.

```
class Outer {
    method() {
        new Inner() {
            override_method() {
                .....
            }
        };
    }
}
```

익명 중첩 클래스는 메소드 내에서 외부 클래스의 객체를 생성할 때, 외부 클래스가 가지는 일부 메서드를 재정의하여 사용할 수 있다.

아래의 예제는 위의 지역 중첩 클래스를 익명 중첩 클래스의 형식으로 바꾸었다. Food 클래스가 외부에 정의되어 있으며 Person 클래스의 doEat() 메서드에서는 Food 클래스의 객체를 생성하여 사용하고 있다. 이때 Food 클래스의 객체를 생성함과 동시에 doEat() 메서드를 재정의 함으로써 Food 클래스의 doEat() 메서드의 실행 내용을 바꾸었다.

```
-----
class Food {
    String food;

    public Food(String food) {
        this.food = food;
    }

    public void makeFood() {
        System.out.println(food + "요리를 합니다.");
    }

    public void doDishes() {
        System.out.println("설거지를 합니다.");
    }

    public void doEat() {
        makeFood();
        System.out.println(food + "를 점심으로 먹었습니다.");
        doDishes();
    }
}

class Person {

    public void doEat(String food) {

        if (food == null || food.trim().equals("")) {
            System.out.println("재료를 주시기 바랍니다.");
            return;
        }

        Food f = new Food(food) {
```

```

        public void doEat() {
            System.out.println(food + "를 함께 먹었습니다.");
        }
    };

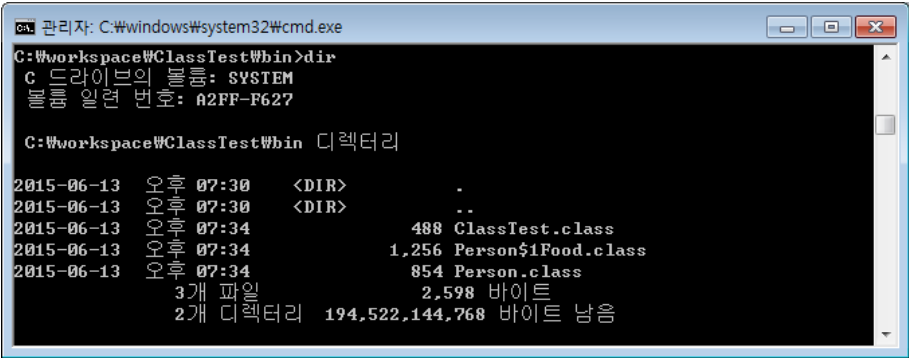
    f.makeFood();
    f.doEat();
    f.doDishes();
}

public class ClassTest {
    public static void main(String[] ar) {
        Person kim = new Person();
        kim.doEat("라면");
    }
}

```

익명 중첩 클래스를 사용하는데 있어서 주의할 사항이 몇 가지 있다. 첫 번째로 외부 클래스에 정의된 메서드는 접근이 허용되는 한 모두 사용이 가능하다. 반면 외부에 정의된 클래스에 정의되지 않은 메서드를 새로 추가할 수 있는데 이렇게 새로 추가된 메서드는 재정의된 메서드나 새로 추가된 메서드 내에서는 사용이 가능하나 그 이외의 영역에서는 사용이 불가능하다.

익명 중첩 클래스는 지역 중첩 클래스와 동일하게 "외부클래스명\$일련번호내부클래스명.class" 라는 이름으로 클래스 파일이 생성된다.



```

C:\workspace\ClassTest\bin>dir
C 드라이브의 볼륨: SYSTEM
볼륨 일련 번호: A2FF-F627

C:\workspace\ClassTest\bin 디렉터리

2015-06-13 오후 07:30 <DIR>      .
2015-06-13 오후 07:30 <DIR>      ..
2015-06-13 오후 07:34          488 ClassTest.class
2015-06-13 오후 07:34       1,256 Person$1Food.class
2015-06-13 오후 07:34          854 Person.class
                3개 파일
                2,598 바이트
                2개 디렉터리 194,522,144,768 바이트 남음

```

14. 상속과 다형성

객체 지향 프로그래밍에 있어서 가장 큰 특징 가운데 하나가 재사용성의 향상이며, 이것이 객체 지향 프로그래밍이 탄생하게 된 이유이기도 하다. 우리는 앞서 클래스를 이용하여 캡슐화를 하고, 캡슐화를 통해 정보은닉을 구현하여 추상화를 높이며, 높은 추상화는 모듈의 독립성과 재사용성을 높인다는 것을 알아보았다.

객체 지향 프로그래밍에 있어서 재사용성을 향상시키기 위한 개념이 또 하나 있는데 이것이 상속이다.

14.1 상속의 개념

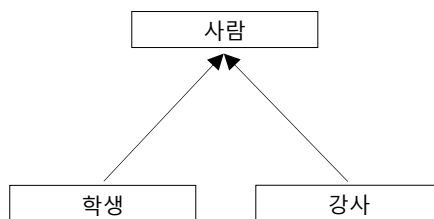
상속의 개념은 다음과 같다.

- 1) 상속은 데이터나 함수의 재사용을 위한 장치이다.
- 2) 상속은 Is-A 관계이다.

상속은 동일한 데이터나 함수의 중복 정의를 피하고 공통된 데이터나 함수를 한 곳에서 정의하여 재사용하자는 개념이다. 상속은 그 방법에 따라 일반화(Generalization)와 특수화(Specialization)으로 구분할 수 있다.

일반화란 현존하는 클래스들로부터 공통된 속성과 함수들을 추출하여 이들을 가지는 새로운 클래스를 정의하는 방법이며, 특수화는 특정 클래스로부터 세부적인 여러 형태의 클래스를 분류하여 표현하는 방법을 말한다.

상속관계에 있어서 공통된 데이터나 함수가 정의된 클래스를 슈퍼 클래스(베이스 클래스, 부모 클래스)라고 하고, 이러한 공통된 내용들을 그대로 재사용하면서 각자에 한정적인 데이터나 함수를 가지는 클래스를 서브 클래스(파생 클래스, 자식 클래스)라고 한다.



상속은 Is-A 관계이다. 여기서 Is-A 관계란 서브 클래스 Is-A 슈퍼 클래스를 의미한다. 위 도식은 Character이 슈퍼 클래스이며, Archer과 Knight는 서브 클래스이다. 그러므로 "Archer은 Character이다", "Knight는 Character이다"라는 Is-A관계가 성립된다. 이러한 성립관계는 Archer과 Knight는 Character이라는 클래스형으로 표현할 수 있음을 의미하며, 객체 지향 프로그래밍에서는 이것을 Upcasting 메커니즘 이라고 한다. Upcasting 메커니즘을 통해 상속에서 제공하는 재사용성 이외에도 "대체성(Substitutability)"이라는 특징을 제공한다.

대체성이란 상속관계에서 슈퍼 클래스의 객체 대신에 서브 클래스의 객체가 대신 대체되어 사용될 수 있다는 것이다.

아래의 예제는 String 클래스에 대한 API Document 이다. 아래의 문서에서는 String 클래스는 java.lang 패키지에 포함되어 있으며, java.lang 패키지의 Object 클래스를 상속하고 있음을 보여준다.

String

DocFlavor.STRING

17

String.String()

17

String.String()

17

String.String()

17

String.String()

17

String.String()

17

Class String

java.lang.Object
 java.lang.String

All Implemented Interfaces:

Serializable, CharSequence, Comparable

public final class String extends Object

아래의 예제는 A 클래스는 그 어떤 클래스도 상속하지 않았다. 그럼에도 getClass() 라는 메소드를 사용하고 있음을 볼 수 있다. 자바에서의 모든 클래스는 명시하지 않아도 java.lang.Object 클래스를 상속하게 된다. 즉 getClass() 메서드는 A 클래스에는 정의되지 않았지만 java.lang.Object 클래스에 정의되어 있으므로 상속에 의해 자신의 멤버인 것처럼 사용할 수 있는 것이다.

```

-----
class A {
    public void print() {
        System.out.println("A 클래스");
    }
}

public class InheritTest {
    public static void main(String[] ar) {
        A obj = new A();
        obj.print();
        System.out.println("클래스 : " + obj.getClass());
    }
}
-----

```

자바에서의 상속은 extends 구문을 사용한다.

```

class 서브클래스 extends 슈퍼클래스 {
    .....
}

```

아래의 예제는 Knight 클래스와 Archer 클래스가 Character 클래스를 상속하는 예제이다. 지난 강의에서 배웠던 this()가 자신 클래스의 생성자의 호출을 의미한다면 super()는 부모 클래스의 생성자 함수의 호출을 나타낸다. 아래의 예제에서는 Character 클래스에 default 생성자가 없으며, 매개변수가 있는 생성자가 있으므로 JVM에 의해 default 생성자가 제공되지 않으므로 자식 클래스의 생성자에서 명시적으로 부모 클래스의 생성자를 호출하였다.

```

-----
class Character {
    String name;
}

```

```

    public Character(String name) {
    }
}

class Knight extends Character {
    public Knight(String name) {
        super(name);
    }
}

class Archer extends Character {
    public Archer(String name) {
        super(name);
    }
}

public class InheritTest {
    public static void main(String[] ar) {
        Knight knight = new Knight("기사");
        Archer archer = new Archer("궁수");
    }
}

```

클래스가 상속관계에 있을 때, 자식 클래스의 인스턴스를 생성시, 자식 클래스의 생성자 메서드에서는 부모클래스의 디폴트 생성자를 자동으로 호출한다. 아래의 예제를 작성하여 실행하여 보면 Knight 클래스와 Archer 클래스의 인스턴스를 생성할 때, 부모 클래스인 Character 클래스의 디폴트 생성자를 호출 하는 것을 볼 수 있다. 그러므로 부모 클래스에 매개변수가 없는 default 생성자가 없다면 반드시 명시적으로 부모 클래스의 생성자 함수를 호출하여야만 한다.

```

class Character {
    String name;

    public Character() {
        System.out.println("Character 클래스의 생성자 호출");
    }
}

class Knight extends Character {
    public Knight(String _name) {
        name = _name;
        System.out.println("Knight 클래스의 생성자 호출");
    }
}

class Archer extends Character {
    public Archer(String _name) {
        name = _name;
        System.out.println("Archer 클래스의 생성자 호출");
    }
}

public class InheritTest {
    public static void main(String[] ar) {
        Knight knight = new Knight("기사");
    }
}

```

```

        Archer archer = new Archer("궁수");
    }
}

```

this()나 super()를 이용한 생성자 함수의 호출은 생성자 함수에서만 가능하며, 생성자 함수 내에서도 무조건 제일 먼저 호출되어야 한다. 또한 this()와 super()는 하나의 생성자 함수 내에서 오직 둘 중 하나만, 한번만 호출할 수 있다.

this()가 자신 클래스의 생성자 함수의 호출이며, super()가 부모 클래스의 생성자 함수의 호출이라면 this는 자신의 객체를 의미하며 super는 부모 클래스의 인스턴스를 나타낸다.

아래의 예제에서 Archer 클래스는 Character 클래스를 상속 받았음에도 부모 클래스에 있는 멤버 name의 이름으로 멤버를 선언 함으로써 부모 클래스의 name이라는 멤버를 가리게 된다. 즉 Archer 클래스 내부에서의 name은 this가 생략된 this.name 이므로 부모 클래스의 name이 아닌 자신의 멤버 name값을 설정하게 된다. 반면 Knight 클래스의 name은 Knight 클래스에 name 멤버가 없으므로 상속 받은 클래스의 name 멤버를 의미한다. 이 처럼 상속관계에 있어서 부모 클래스와 자식클래스에 동일한 이름의 멤버가 존재할 경우, 어떤 클래스의 멤버인가를 구분해주는 예약어가 this와 super 이다.

```

-----
class Character {
    String name;

    public Character() {
        System.out.println("Character 클래스의 생성자 호출");
    }

    public void printName() {
        System.out.println("이름 : " + name);
    }
}

class Knight extends Character {
    public Knight(String _name) {
        name = _name;
        System.out.println("Knight 클래스의 생성자 호출");
    }
}

class Archer extends Character {
    String name;

    public Archer(String _name) {
        name = _name;
        System.out.println("Archer 클래스의 생성자 호출");
    }
}

public class InheritTest {
    public static void main(String[] ar) {
        Knight knight = new Knight("기사");
        knight.printName();
        Archer archer = new Archer("궁수");
        archer.printName();
    }
}

```

}
}

14.2 상속의 특징

상속은 다음과 같이 여러 가지 특성을 제공한다.

- 1) 데이터와 함수의 중복성을 제거한다.
- 2) 데이터나 함수의 추가가 용이하다.
- 3) 새로운 클래스의 추가가 용이하다.

클래스들 간의 관계가 상속관계에 놓이게 되면 부모 클래스와 자식 클래스들 사이에는 중복된 데이터가 존재하지 않게 된다. 이것은 자식클래스에서 부모클래스의 데이터나 함수를 자신의 멤버인 것처럼 사용할 수 있기 때문이다. 물론 부모 클래스의 데이터나 함수를 상속받지 않고 재정의(오버라이딩)하여 사용할 수 도 있다. 그러나 일반적으로는 부모 클래스에 정의된 데이터나 함수를 자식 클래스에서 정의하지 않아도 그대로 사용할 수 있으므로 기존의 절차적 프로그래밍 기법에 비해 중복성이 많이 감소된다.

상속관계에 있어서 자식 클래스는 부모 클래스의 데이터나 함수를 상속받기 때문에 자식 클래스는 부모 클래스에 의존적이다. 하지만 자식 클래스는 독립적인 단위이므로 클래스에 데이터나 함수를 새롭게 추가할 수 있다. 이것은 곧 자식클래스에 추가되는 새로운 데이터나 함수는 부모 클래스에 영향을 미치지 않는다는 것을 의미한다.

상속관계는 계층 구조이다. 그러므로 새로운 하위 클래스의 추가가 용이하며 추가되는 클래스는 자신의 슈퍼 클래스에 정의된 데이터나 함수를 상속 받는다. 즉 동일한 클래스를 부모 클래스로 하는 여러 개의 자식 클래스가 존재할 수 있음을 의미한다.

아래의 간단한 상속의 예제이다. Knight와 Archer 클래스는 Character 클래스를 상속하였으므로 Character 클래스가 가지는 모든 멤버를 재정의 없이 그대로 사용이 가능하다.

```
-----
class Character {
    protected String weapon;

    public Character() {
        weapon = "맨손";
    }

    public void attack() {
        System.out.println(weapon + "(으)로 공격합니다.");
    }

    public void setWeapon(String weapon) {
        this.weapon = weapon;
    }
}

class Knight extends Character {
}

class Archer extends Character {
```

```

}

public class InheritTest {
    public static void main(String[] ar) {
        Knight knight = new Knight();
        knight.setWeapon("검");
        knight.attack();

        Archer archer = new Archer();
        archer.setWeapon("활");
        archer.attack();
    }
}

```

상속관계에서 자식 클래스는 부모 클래스의 멤버를 자신의 것처럼 상속하여 사용할 수 있다고 하였다. 이때 부모 클래스의 멤버에 대하여 `private` 접근제한자를 지정하는 경우, 자식 클래스에서는 접근이 불가능하게 된다. 그렇다면 `public` 접근제한자를 지정하는 경우 외부에서도 접근이 가능하므로 정보은닉을 구현할 수 없게 되며, 클래스의 추상화는 낮아지고 모듈의 독립성 또한 낮아지게 된다.

상속관계에서 부모 클래스의 멤버를 자식 클래스에서만 접근이 가능하도록 제한 할 수 있는데 이때 사용되는 접근제한자가 `protected` 접근제한자이다. 아래 예제의 `Character` 클래스의 `name` 멤버는 `private` 접근제한자가 지정되었으므로 상속관계에 있는 자식 클래스 `Knight`와 `Archer`에서는 접근할 수가 없다.

```

class Character {
    private String name;

    public void printName() {
        System.out.println("이름 : " + name);
    }
}

class Knight extends Character {
    public Knight(String _name) {
        name = _name;
    }
}

class Archer extends Character {
    public Archer(String _name) {
        name = _name;
    }
}

public class InheritTest {
    public static void main(String[] ar) {
        Knight knight = new Knight("기사");
        knight.printName();
        Archer archer = new Archer("궁수");
        archer.printName();
    }
}

```

이처럼 상속관계에서 자식 클래스에 대하여 접근을 허용하도록 하는 접근제한자가 `protected` 접근

제한자이다.

```
-----
class Character {
    protected String name;

    public void printName() {
        System.out.println("이름 : " + name);
    }
}

class Knight extends Character {
    public Knight(String _name) {
        name = _name;
    }
}

class Archer extends Character {
    public Archer(String _name) {
        name = _name;
    }
}

public class InheritTest {
    public static void main(String[] ar) {
        Knight knight = new Knight("기사");
        knight.printName();
        Archer archer = new Archer("궁수");
        archer.printName();
    }
}
-----
```

protected 접근제한자는 멤버 필드만이 아니라 멤버 메소드에도 지정할 수 있는데 이 경우, 클래스 내부나 상속관계에 있는 자식 클래스에서만 접근이 가능하다. 아래 예제에서 Character 클래스의 멤버 메소드인 printName() 메서드는 protected 접근제한자가 지정되었으므로 외부에서는 사용이 불가능하게 된다. 물론 아래의 예제는 동일한 파일 내에서 printName() 메서드를 호출 함으로 정상적으로 실행이 가능하지만 다른 패키지에서는 printName() 메서드를 호출 할 수가 없다.

```
-----
class Character {
    protected String name;

    protected void printName() {
        System.out.println("이름 : " + name);
    }
}

class Knight extends Character {
    public Knight(String _name) {
        name = _name;
    }
}

class Archer extends Character {
```



```

    public Archer(String _name) {
        name = _name;
    }
}

public class InheritTest {
    public static void main(String[] ar) {
        Knight knight = new Knight("기사");
        knight.printName();
        Archer archer = new Archer("궁수");
        archer.printName();
    }
}

```

다른 패키지에서 printName() 메서드를 호출하고자 한다면 Character 클래스의 printName() 메서드가 public 접근제한자로 지정되거나 아래와 같이 하위 클래스에서 public 접근제한자로 오버라이딩되어야만 한다.

```

class Character {
    protected String name;

    protected void printName() {
        System.out.println("이름 : " + name);
    }
}

class Knight extends Character {
    public Knight(String _name) {
        name = _name;
    }

    public void printName() {
        super.printName();
    }
}

class Archer extends Character {
    public Archer(String _name) {
        name = _name;
    }

    public void printName() {
        super.printName();
    }
}

public class InheritTest {
    public static void main(String[] ar) {
        Knight knight = new Knight("기사");
        knight.printName();
        Archer archer = new Archer("궁수");
        archer.printName();
    }
}

```

14.3 다형성 (Polymorphism)

다형성이란? "여러 개의 형태를 가진다"는 의미의 그리스어에서 유래된 말로서 특정한 심벌이나 연산자에 대해 상황이 다르면 그 의미도 다르게 부여할 수 있는 특성을 말한다.

객체 지향 언어에서의 다형성은 다음과 같이 정의된다.

"다형성은 특정 심벌이나 연산자에 대해 상황 별로 의미를 달리 부여하는 장치이다."

객체 지향 프로그래밍에서는 위와 같은 다형성으로 오버로딩(Overloading)과 오버라이딩(Overriding)을 가진다.

◦ 오버로딩 (Overloading 또는 중첩)

오버로딩은 연산자 오버로딩과 함수 오버로딩이 있다. C++과는 달리 자바에서는 연산자 오버로딩을 문법적으로 지원하지 않으므로 함수 오버로딩에 대해서만 설명하도록 하겠다.

함수 오버로딩이란? 동일한 이름의 함수가 상황에 따라 달리 반응하는 현상을 말한다. 함수 오버로딩은 상황에 따라 두 종류의 형태로 정의될 수 있다.

- 1) 동일한 클래스 내에서의 함수 오버로딩
- 2) 상속관계에서의 함수 오버로딩

동일한 클래스 내에서의 함수 오버로딩은 하나의 클래스 내에 동일한 이름의 함수가 여러 개 존재하는 것을 말한다. 이때 몇 가지 제약사항이 있는데 다음과 같다.

- 1) 함수명은 동일하여야 한다.
- 2) 함수의 매개변수의 개수나, 매개변수의 타입이 달라야 한다.
- 3) 예외 전가 구문이 있는 경우, 예외 전가 구문도 일치하여야 한다.

아래의 예제는 동일한 클래스 내에서의 함수 오버로딩에 대한 예제이다. Character 클래스의 move() 메서드가 두 개가 정의되어 있지만 메서드 호출 시 주어지는 인자에 따라 다른 메서드가 호출된다.

```
-----
class Character {
    protected String weapon;

    public Character() {
        weapon = "맨손";
    }

    public void attack() {
        System.out.println(weapon + "(으)로 공격합니다.");
    }

    public void setWeapon(String weapon) {
        this.weapon = weapon;
    }
}
```

```

    public void move() {
        System.out.println("걸어서 이동합니다.");
    }

    public void move(String vehicle) {
        System.out.println(vehicle + "(을)를 타고 이동합니다.");
    }
}

class Knight extends Character {
}

class Archer extends Character {
}

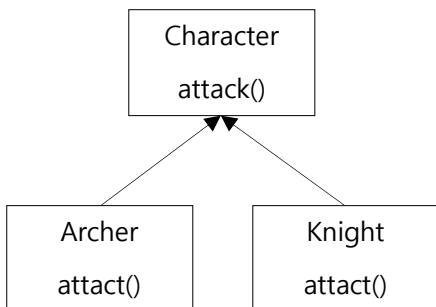
public class InheritTest {
    public static void main(String[] ar) {
        Knight knight = new Knight();
        knight.setWeapon("검");
        knight.attack();
        knight.move();

        Archer archer = new Archer();
        archer.setWeapon("활");
        archer.attack();
        knight.move("말");
    }
}

```

상속관계에서의 함수 오버로딩은 두 개 이상의 클래스들이 동일한 메시지에 대하여 각각 자신의 방식으로 반응하는 것을 말한다.

예를 들어 아래와 같은 상속구조에서 자식 클래스인 Archer와 Knight은 부모 클래스인 Character의 attack이라는 행위를 상속받게 되므로 attack()라는 메서드를 그대로 물려받아 사용할 수 있다. 하지만 이때 Character이 구현 클래스가 아닌 추상 클래스이며, attack이라는 행위가 추상 메서드라면 Archer과 Knight는 반드시 attack이라는 행위를 구현하여야만 한다. 이것은 Archer와 Knight은 하는 공격하는 방법이 다르기 때문이다. 즉 Archer은 활로 공격할 것이며, Knight는 검으로 공격할 것이므로 attack이라는 행위에 대한 구현은 자식 클래스에서 정의하도록 한 것이다.



Archer 클래스와 Knight 클래스는 Character 클래스를 상속하였으므로 Character 클래스 타입으

로 Upcasting이 가능하다. "Archer은 Character이다", "Knight는 Character이다" 라는 관계가 성립하기 때문이다. 그러므로 아래와 같이 Archer 클래스의 객체와 Knight 클래스의 객체는 Character 클래스 타입으로 다룰 수 있게 된다.

```
Character knight = new Knight();
Character archer = new Archer();
```

Character 클래스에는 attack()메서드가 있으므로 knight객체와 archer객체를 통해 attack() 메소드를 호출할 수 있다.

```
knight.attack();
archer.attack();
```

하지만 이때 attach() 메소드 호출로 인해 실행되는 메소드는 Character 클래스의 attack() 메소드가 아닌 Archer 클래스의 attack()메소드와 Knight 클래스의 attack() 메소드가 실행된다. 즉 해당 객체는 Character 클래스형의 객체이지만 어떤 클래스의 인스턴스 인가에 의해 호출할 메서드가 실행타임에 결정되게 되는데 이것을 동적 바인딩(Dynamic Binding) 혹은 Late Binding이라고 한다.

상속관계에 있어서의 함수 오버로딩은 한 클래스내의 오버로딩과는 달리 함수명과 매개변수의 개수, 매개변수의 타입과 예외 전가구문이 완전히 일치하여야 한다.

아래의 예제는 상속관계에서의 오버로딩의 예제이다. Character 클래스가 추상 클래스이며, attack()라는 추상메서드가 선언되어 있으므로 Knight 클래스와 Archer 클래스에서는 반드시 attack() 메서드를 구현하여야 만 한다.

main() 메서드를 보면 knight 객체와 archer 객체는 각각 Knight 클래스와 Archer 클래스의 인스턴스이지만 객체형은 Character 클래스형으로 선언되었다. 그러므로 knight.attack() 메서드와 archer.attack() 메서드는 Character 클래스의 attack() 메서드를 호출 하는 것이지만 다형성에 의해 각각 Knight 클래스와 Archer 클래스의 attack() 메서드가 호출되게 된다.

컴파일 타임에는 Character 클래스의 attack() 메서드가 존재하는가 만을 테스트 하며, 실행 시에는 객체가 어떤 클래스의 인스턴스 인가에 따라 해당 클래스의 attack() 메소드를 호출하는 것이다.

```
-----
abstract class Character {
    public abstract void attack();
}

class Knight extends Character {

    private void readySword() {
        System.out.println("검을 뽑았습니다.");
    }

    public void attack() {
        readySword();
        System.out.println("공격을 합니다.");
    }
}

class Archer extends Character {

    private void readyBow() {
```

```

        System.out.println("활을 조준합니다.");
    }

    public void attack() {
        readyBow();
        System.out.println("활을 쏩니다.");
    }
}

public class InheritTest {
    public static void main(String[] ar) {
        Character knight = new Knight();
        knight.attack();

        Character archer = new Archer();
        archer.attack();
    }
}

```

◦ 오버라이딩 (Overriding 또는 재정의)

함수 오버라이딩은 상속관계에서만 발생하며 사용방법 또한 상속관계에서의 함수 오버로딩과 같다. 다만 차이점이 있다면 함수 오버로딩은 부모 클래스가 추상 클래스이고, 추상함수에 대해 발생하는 반면, 함수 오버라이딩은 상속구조에서 부모 클래스가 추상 클래스이든 일반 클래스이든 상관없이 없으며, 부모 클래스 내에 구현된 함수에 대하여 함수명만을 상속받고 함수의 구현은 자식 클래스에서 각각 재정의 하는 것을 말한다.

상속관계에서의 오버로딩은 자식 클래스들의 특정 메소드의 수행이 서로 다를 때, 각각의 클래스에 알 맞는 메소드를 수행하기 위한 것인 반면, 오버라이딩은 부모 클래스의 메소드 내용을 다른 내용으로 변경하는 것을 목적으로 함을 의미한다.

아래의 예제는 오버라이딩의 예제이다. Knight 클래스와 Archer 클래스는 Character 클래스를 상속하였으므로 Character 클래스의 멤버 메소드인 attack() 메소드를 그대로 사용할 수 있다. 하지만 Knight 클래스의 경우 부모 클래스의 attack() 메소드를 그대로 사용하여도 괜찮지만 Archer의 경우 공격하는 방식이 틀리므로 Archer 클래스 내에서 attack() 라는 메서드를 재정의 하였다. 이처럼 상속관계에 오버로딩이 동적 바인딩에 의한 메소드의 수행이라면, 오버라이딩은 상속관계에 있어 부모 클래스의 메소드의 내용을 다른 내용으로 변경하는 것을 목적으로 한다.

```

class Character {
    public void attack() {
        System.out.println("검으로 공격합니다.");
    }
}

class Knight extends Character {
}

class Archer extends Character {
}

```

```
private void readyBow() {
    System.out.println("활을 조준합니다.");
}

public void attack() {
    readyBow();
    System.out.println("활을 쏩니다.");
}

}

public class InheritTest {
    public static void main(String[] ar) {
        Character knight = new Knight();
        knight.attack();

        Character archer = new Archer();
        archer.attack();
    }
}
```

15. 추상클래스

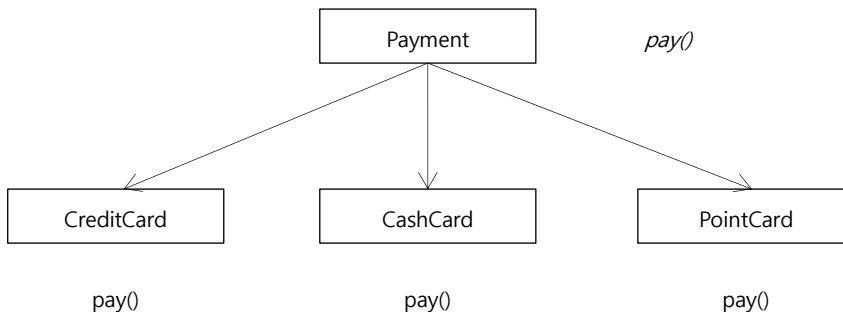
추상클래스는 서로 다른 유형의 클래스들 간에 공통된 행위를 추출하여 정의한 가상의 클래스이다. 즉 추상클래스는 여러 클래스들 간의 공통된 행위들에 있어 중복성을 배제하기 위해 일반화 시킨 클래스이며, 객체를 생성할 수 없다. 따라서 상속구조에서 객체를 생성할 수 있는 슈퍼클래스와 구별하기 위해 추상클래스를 별도로 구분하여 정의한다.

15.1 추상클래스의 개념

- 1) 추상클래스는 서로 다른 유형의 클래스들의 공통된 행위를 추출하여 정의한 것이다.
- 2) 추상클래스는 단일 인터페이스를 제공한다.
- 3) 추상클래스는 적어도 하나 이상의 가상함수를 가져야 한다.

추상클래스는 상속관계에 있어서 객체를 생성할 수 있는 슈퍼클래스와 구별하기 위해 별도로 구분하여 정의한다고 하였다. 이것은 추상클래스는 객체화가 될 수 없음을 의미하며, 반드시 상속을 통해서만이 객체화가 가능함을 의미한다. 이때 추상클래스를 상속받고 실행부가 없는 추상클래스의 메서드를 재정의 하는 자식 클래스를 가리켜 구현 클래스 (Concrete Class)라고 한다.

예를 들어 결제수단이 있다고 할 때, 결제를 하는데 있어서 신용카드를 이용한 결제, 현금카드를 이용한 결제, 포인트 카드를 이용한 결제가 있을 수 있다. 어떤 카드를 이용하던 공통적으로 지불이라는 행위를 가지게 된다. 이때 지불이라는 행위가 여러 카드들이 가지는 공통적인 행위인 것이다.



위에서 Payment 클래스는 여러 함수를 가질 수 있지만 pay() 함수는 Payment 클래스를 상속받는 여러 클래스의 공통적인 행위이다. 즉 CreditCard, CashCard, PointCard의 공통적인 기능인 지불이라는 행위를 Payment 클래스로 정의하였다.

자바에서의 추상클래스는 abstract 키워드를 이용하여 클래스를 정의한다. 아래의 Payment 클래스는 abstract 키워드에 의해 추상클래스로 정의되었으므로 객체화가 불가능 하며, 상속을 통해 Payment 클래스를 상속받은 자식 클래스(구현 클래스)를 통해 사용되어진다.

```

-----
abstract class Payment {
    public void pay() {
        System.out.println("결제금액을 지불합니다.");
    }
}

class CreditCard extends Payment {
}

class CashCard extends Payment {
}

class PointCard extends Payment {
}
  
```



```

}

public class AbstractTest {
    public static void main(String[] ar) {
        Payment credit = new CreditCard();
        Payment cash = new CashCard();
        Payment point = new PointCard();

        credit.pay();
        cash.pay();
        point.pay();
    }
}

```

추상클래스는 두 가지 목적을 지닌다. 첫 번째 목적은 서로 다른 클래스들이 공통적으로 구현된 함수들을 추출하기 위함이다. 두 번째 목적은 추상 클래스로부터 파생된 모든 클래스들에 대해 단일의 공통된 인터페이스를 생성하기 위함이다.

아래 예제의 Payment 클래스는 추상 클래스로써 pay() 라는 추상 메서드를 가지고 있다. 추상메서드는 메서드의 선언만 있을 뿐, 메서드의 실행 부는 정의되어 있지 않다. 그리고 Payment라는 추상클래스를 상속받은 모든 클래스는 Payment 클래스내에 선언된 모든 추상 메서드를 반드시 오버라이딩 하여야만 한다. 이것은 CreditCard, CashCard, PointCard가 공통적으로 가지는 행위이지만 클래스(카드의 종류)에 따라 지불하는 방식이 틀리기 때문이다.

```

abstract class Payment {
    public abstract void pay();
}

class CreditCard extends Payment {
    public void pay() {
        System.out.println("신용카드로 결제합니다.");
    }
}

class CashCard extends Payment {
    public void pay() {
        System.out.println("현금카드로 결제합니다.");
    }
}

class PointCard extends Payment {
    public void pay() {
        System.out.println("포인트카드로 결제합니다.");
    }
}

public class AbstractTest {
    public static void main(String[] ar) {
        Payment credit = new CreditCard();
        Payment cash = new CashCard();
        Payment point = new PointCard();
    }
}

```

```
        credit.pay();
        cash.pay();
        point.pay();
    }
}
```

위에서 보았듯이 추상클래스는 자식클래스에서 서로 다르게 구현된 공통의 행위들에 대한 템플릿만을 제공하며, 서로 다르게 구현된 하위 클래스들의 함수가 `pay()`라는 공통 인터페이스를 통해 외부에 서비스를 제공할 수 있도록 하는데 이것이 이전 강의에서 언급했던 동적 바인딩 메커니즘이다.

자바에서 추상클래스는 `abstract` 키워드를 이용하여 클래스를 정의하며 일반 클래스의 모든 멤버를 포함할 수 있다. 추상클래스는 일반적인 클래스와는 달리 객체화가 불가능하며 일반적인 클래스와는 달리 추상함수를 가질 수 있다. 반대로 일반함수는 추상클래스를 포함할 수 없다.

추상클래스는 다음과 같은 특징을 가진다.

15.2 추상클래스의 특징

- 1) 재사용성을 높여준다.
- 2) 자식 클래스 객체들을 추상 클래스 형으로 대체할 수 있다.
- 3) 서로 다른 종류의 객체들을 수집할 수 있다.
- 4) 유지보수성을 향상시켜준다.
- 5) 확장성을 높여준다.

추상클래스는 상속구조를 기반으로 하여 자식 클래스들이 가지는 공통된 속성이나 함수를 추출함으로써 재사용성을 높여준다. 뿐만 아니라 Upcasting 메커니즘에 의해 추상클래스가 객체화 될 수는 없지만 추상클래스를 상속받은 자식클래스(구현클래스)의 인스턴스는 추상클래스 형으로 대체가 가능하다.

아래의 예제는 추상클래스 Payment를 상속받은 CreditCard, CashCard, PointCard의 객체를 생성하는데 있어서 이들이 공통적으로 상속받은 클래스인 Payment 클래스 형으로 Upcasting하여 사용할 수 있음을 보여준다. 이것은 서로 다른 각각의 클래스를 Payment 클래스 형으로 수집하여 관리할 수 있음을 의미한다.

```
-----
abstract class Payment {
    public abstract void pay();
}

class CreditCard extends Payment {
    public void pay() {
        System.out.println("신용카드로 결제합니다.");
    }
}

class CashCard extends Payment {
    public void pay() {
        System.out.println("현금카드로 결제합니다.");
    }
}

class PointCard extends Payment {
    public void pay() {
        System.out.println("포인트카드로 결제합니다.");
    }
}

public class AbstractTest {
    public static void main(String[] ar) {
        Payment[] pays = new Payment[]{new CreditCard(),
            new CashCard(),
            new PointCard()};

        for (int i=0 ; i<pays.length ; i++) {
            pays[i].pay();
        }
    }
}
-----
```

추상클래스의 가상함수는 함수의 선언만 있으며 구현부가 없으므로 가상함수에 대한 알고리즘은 자식 클래스에서 구현하도록 하고 있다. 이것은 자식 클래스에서 얼마든지 알고리즘의 변경이나 로직의 변경이 가능함을 의미한다.

15.3 추상클래스의 구성멤버

추상클래스는 객체화가 될 수 없는 클래스로써 클래스의 모든 멤버와 더불어 abstract 메서드를 포함할 수 있다. abstract 메서드란? 메서드의 스펙 만을 정의할 뿐, 구현부가 없다. 아래의 클래스 Card에는 pay() 메서드가 정의되어 있으나 구현부가 없다. 즉 Card 클래스를 상속받는 클래스들은 반드시 abstract 메서드인 pay() 메서드를 재정의 하여야 한다.

```
-----
abstract class Card {
    public abstract void pay();
}

class CreditCard extends Card {

    public void pay() {
        System.out.println("지불하다.");
    }
}

public class AbstractTest {
    public static void main(String[] args) {
        Card mycard = new CreditCard();
        mycard.pay();
    }
}
-----
```

여러분의 이해를 돕기 위해 또 하나의 예제를 보도록 하자. 아래는 피자가게에서 피자를 주문 받아 판매하는 프로그램이다. 피자가게에서는 손님으로 부터 주문을 받은 후, 해당 주문에 따라 피자를 만들게 된다. 그러므로 이 프로그램에서는 다음의 객체가 필요하다.

```
피자 가게      : PizzaStore
피자           : Pizza
```

우선 판매제품인 피자 클래스는 아래와 같이 정의하였다. 피자클래스에는 다음과 같은 메서드가 정의되어 있다.

```
prepare()      : 피자를 만드는 메서드
cut()          : 피자를 자르는 메서드
boxing()       : 피자를 포장하는 메서드
```

```
-----
public class Pizza {
    private String name;

    public Pizza(){
        name = "베이직 피자";
    }

    public void prepare(){
        System.out.println("피자를 만듭니다.");
    }
}
-----
```

```

    }

    public void cut(){
        System.out.println("피자를 자릅니다.");
    }

    public void boxing(){
        System.out.println("피자를 포장합니다.");
    }

    public String toString(){
        return name;
    }
}

```

아래는 피자 가게 클래스로서 아래의 메서드가 정의되어 있다.

orderPizza() : 피자를 주문 받고 주문 받은 피자를 반환하는 메서드

```

public class PizzaStore {
    private String name;

    public PizzaStore(){
        name = "기희네 피자집";
    }

    public Pizza orderPizza(){
        Pizza pizza = new Pizza();
        pizza.prepare();
        pizza.cut();
        pizza.boxing();

        return pizza;
    }

    public String toString(){
        return name;
    }
}

```

손님역할은 별도의 클래스를 정의하지 않고 main() 가 역할을 담당하도록 하였다.

```

public class AppStart {
    public static void main(String[] args){
        PizzaStore store = new PizzaStore();
        Pizza pizza = store.orderPizza();
        System.out.println(pizza.toString());
    }
}

```

위의 예제를 실행하면 PizzaStore 클래스의 인스턴스인 store 객체의 orderPizza() 메서드 호출에

의해 Pizza 객체를 생성하고 반환하는 것을 확인 할 수 있다.

위의 예제에서는 오직 한가지 피자만을 주문 받을 수 있다. 만약 여러 가지의 피자를 주문 받고자 한다면 어떻게 하여야 할까? 피자 종류에 따른 클래스를 정의하여야 할 것이다. 다양한 종류의 피자를 판매하기 위해 아래와 같이 CheesePizza와 BaconPizza 클래스를 정의하였다.

```
-----
public class CheesePizza {
    private String name;

    public CheesePizza(){
        name = "치즈피자";
    }

    public void prepare(){
        System.out.println("치즈를 이용하여 피자를 만듭니다.");
    }

    public void cut(){
        System.out.println("피자를 자릅니다.");
    }

    public void boxing(){
        System.out.println("피자를 포장합니다.");
    }

    public String toString(){
        return name;
    }
}

public class BaconPizza {
    private String name;

    public BaconPizza(){
        name = "베이컨피자";
    }

    public void prepare(){
        System.out.println("베이컨을 이용하여 피자를 만듭니다.");
    }

    public void cut(){
        System.out.println("피자를 자릅니다.");
    }

    public void boxing(){
        System.out.println("피자를 포장합니다.");
    }

    public String toString(){
        return name;
    }
}
```

```
    }
}
```

피자가게 또한 다양한 피자가 존재하므로 어떤 피자인가 구분하여 주문을 받아야만 한다. 그리고 주문에 따라 치즈피자를 주문하였다면 CheesePizza의 인스턴스를 반환하여야 하며, 베이컨피자를 주문하였다면 BaconPizza의 인스턴스를 반환하여야만 한다. 물론 기본피자를 주문하였다면 Pizza의 인스턴스를 반환하여야 할 것이다. 그러므로 PizzaStore 클래스에는 피자 종류에 따라 각각의 피자를 주문 받아 피자를 만드는 메서드가 필요하게 된다.

```
public class PizzaStore {
    private String name;

    public PizzaStore(){
        name = "기희네 피자집";
    }

    public Pizza orderPizza(){
        Pizza pizza = new Pizza();
        pizza.prepare();
        pizza.cut();
        pizza.boxing();
        return pizza;
    }

    public CheesePizza orderCheesePizza(){
        CheesePizza pizza = new CheesePizza();
        pizza.prepare();
        pizza.cut();
        pizza.boxing();
        return pizza;
    }

    public BaconPizza orderBaconPizza(){
        BaconPizza pizza = new BaconPizza();
        pizza.prepare();
        pizza.cut();
        pizza.boxing();
        return pizza;
    }

    public String toString(){
        return name;
    }
}
```

피자를 주문할 때 또한 피자 종류에 따라 각각 다른 메서드를 호출하여야만 한다.

```
public class AppStart {
    public static void main(String[] args){
        PizzaStore store = new PizzaStore();
        Pizza pizza = store.orderPizza();
        CheesePizza cheesePizza = store.orderCheesePizza();
    }
}
```



```

        BaconPizza baconPizza = store.orderBaconPizza();

        System.out.println(pizza.toString());
        System.out.println(cheesePizza.toString());
        System.out.println(baconPizza.toString());
    }
}

```

만약 새로운 피자가 추가된다면 어떠 한가? 10여가지의 피자 종류가 추가가 된다면? 피자 종류의 추가에 따른 새로운 피자 클래스의 추가는 어쩔 수 없지만 PizzaStore 에는 각각의 피자를 주문 받기 위한 메서드가 무수히 정의되어야 할 것이다.

위의 예제에서 피자를 주문 받기 위한 메서드가 각각의 피자 종류만큼 정의한 이유는 무엇인가? 그것은 주문에 따라 만들어야 하는 피자가 다르기 때문이다. 만약 종류와 상관없이 그것이 치즈피자가 되었던 베이컨피자가 되었던 피자라는 일관된 유형으로 관리를 할 수 있다면 위와 같이 피자종류에 따른 주문 메서드 역시 하나의 메서드로 처리가 가능할 것이다.

위에서 추상클래스에 대하여 설명하였다. 추상클래스는 서로 다른 유형의 클래스들 간의 공통된 행위를 추출하여 정의한다고 하였다. 위의 예제를 보면 CheesePizza가 되었던 BaconPizza가 되었던 공통적으로 피자를 만들고, 자르고, 포장한다. 그러므로 Pizza클래스를 아래와 같은 추상클래스로 정의한다.

일반클래스는 추상메서드를 포함할 수 없으므로 Pizza 클래스는 abstract 클래스가 될 것이다. 또한 일반클래스가 추상메서드를 포함할 수 는 없지만, 추상클래스는 생성자를 포함하여 일반 메서드를 포함할 수 있다. 어떤 종류의 피자가 되었던 만드는 방법만 틀릴 뿐, 자르고 포장하는 방법은 동일하므로 cut() 메서드와 boxing() 메서드는 추상클래스에서 그 행위를 정의하였다. 반면 피자를 만드는 방법은 피자의 종류에 따라 달라지므로 반드시 오버라이딩 하여야 하는 abstract 메서드로 정의하였다.

```

public abstract class Pizza {
    protected String name;

    public abstract void prepare();

    public void cut(){
        System.out.println("피자를 자릅니다.");
    }

    public void boxing(){
        System.out.println("피자를 포장합니다.");
    }

    public String toString(){
        return name;
    }
}

```

위의 Pizza 클래스는 추상 클래스이므로 객체화가 불가능 하다. 위에서도 설명하였지만 추상클래스는 상속에 의해 자식클래스(구현클래스)에 의해 객체화가 가능하다고 하였으므로 위의 추상클래스를 상속하는 구현클래스를 정의하여야 한다. 이전에 사용하였던 CheesePizza 클래스와 BaconPizza 클래스를 아래와 같이 수정하도록 한다.

우선 CheesePizza 클래스와 BaconPizza 클래스는 Pizza 클래스를 상속받는다. 이로 인해 "CheesePizza 는 Pizza" 이다. "BaconPizza 는 Pizza" 이다 라는 Is-A 공식이 성립되므로 CheesePizza 클래스와 BaconPizza 클래스에서는 부모클래스의 cut(), boxing() 메서드를 사용할 수 있게 된다.

하지만 피자를 만드는 메서드인 prepare() 메서드가 abstract 로 정의된 추상 메서드이므로 하위 클래스인 CheesePizza 클래스와 BaconPizza 클래스에서는 반드시 재정의의 하여야만 하며, 각각의 피자 종류에 맞도록 피자를 만들도록 행위를 정의하면 될 것이다.

```
-----
public class CheesePizza extends Pizza {
    public CheesePizza(){
        name = "치즈피자";
    }

    public void prepare(){
        System.out.println("치즈를 이용하여 피자를 만듭니다.");
    }
}

public class BaconPizza extends Pizza {
    public BaconPizza(){
        name = "베이컨피자";
    }

    public void prepare(){
        System.out.println("베이컨을 이용하여 피자를 만듭니다.");
    }
}
-----
```

CheesePizza 클래스와 BaconPizza 클래스가 Pizza 클래스를 상속하므로 CheesePizza 클래스의 인스턴스와 BaconPizza 클래스의 인스턴스는 Pizza 클래스 타입이 될 수 있다. 그러므로 피자종류에 상관없이 Pizza 클래스 형으로 다룰 수 있게 된다. 하지만 어떤 피자를 주문 받았는가를 구분하여야 하므로 피자의 종류는 메서드의 매개변수 값을 통해 구분하도록 하였다.

아래의 예제의 orderPizza() 메서드는 주문 받은 피자의 종류를 type 이라는 이름의 변수로 넘겨받고 인수의 값에 따라 CheesePizza 또는 BaconPizza 클래스의 객체를 생성하고 반환하게 된다.

※ orderPizza() 메서드에서 인수로 넘겨받는 값을 비교하여 해당 피자의 인스턴스를 만들게 되는데, 취급하지 않는 값을 입력하게 되면 피자객체의 인스턴스를 생성하지 않고 null을 반환하게 된다. 만약 피자객체를 생성하지 않는다면 pizza 변수의 값은 null 이므로 prepare(), cut(), boxing() 메서드를 호출 할 수 없게 된다. 정확히는 호출 하는 순간 NullPointerException 이라는 오류가 발

생하게 될 것이다. 우리는 아직 예외처리에 대해 배우지 않았으므로 pizza 객체가 null 인가를 확인하여 처리하였다.

```
-----
public class PizzaStore {
    private String name;

    public PizzaStore(){
        name = "기희네 피자집";
    }

    public Pizza orderPizza(String type){
        Pizza pizza = null;

        if (type.toLowerCase().equals("bacon")){
            pizza = new BaconPizza();
        }
        else if (type.toLowerCase().equals("cheese")){
            pizza = new CheesePizza();
        }

        if (pizza != null){
            pizza.prepare();
            pizza.cut();
            pizza.boxing();
        }

        return pizza;
    }

    public String toString(){
        return name;
    }
}
-----
```

피자를 주문할 때 역시 어떤 피자인가를 인수로 넘겨야만 한다.

```
-----
public class AppStart {
    public static void main(String[] args){
        PizzaStore store = new PizzaStore();
        Pizza pizza = store.orderPizza("cheese");

        System.out.println(pizza.toString());
    }
}
-----
```

이제 새로운 피자를 추가하고자 한다면 Pizza 클래스를 상속받는 피자클래스를 정의하고 PizzaStore 클래스의 orderPizza() 메서드에서 그 피자를 생산하는 코드만을 추가하기만 하면 된다. 그러므로 추상클래스는 다양한 클래스의 공통된 행위를 추출하여 정의함으로써 공통된 인터페이스를 제공하여, 코드의 재사용성을 높여주며, 확장성을 높여준다.

※ 생각해볼 문제

위의 예제에서는 피자가게가 하나라는 가정하에 프로그래밍 하였다. 만약 피자가게가 여러 개가 생긴다면? 그리고 각각의 피자가게에서 취급하는 피자의 종류가 다르다면?

힌트 : 클래스로부터 달라져야 하는 부분(행위)를 찾아서 분리하라.

16. 인터페이스

인터페이스의 사전적 정의는 사물 간 또는 사물과 인간 사이의 의사소통이 가능하도록 일시적 혹은 영구적 접근을 목적으로 만들어진 물리적, 가상적 매체를 의미한다.

16.1 인터페이스의 개념

- 1) 인터페이스는 의사소통을 위한 물리적, 가상적 매체이다.
- 2) 인터페이스는 특정 기능을 수행하기 위해 선언된 함수들의 집합이다.
- 3) 인터페이스는 소프트웨어 서비스에 대한 명세이다.

예를 들어 TV의 경우, 우리가 TV를 켜거나 채널을 변경하거나 소리를 높이거나 낮추기 위해 리모컨을 이용하게 된다. 즉 리모컨은 사람과 TV와의 상호작용을 위한 물리적인 매체이자 인터페이스인 것이다. 이와 마찬가지로 자바에서의 인터페이스는 특정 기능을 수행하기 위해 선언된 함수들의 집합이다. 인터페이스는 특정한 기능 또는 특정한 서비스를 수행하는데 필요한 함수들만을 선언하고 있으며, 인터페이스에서 선언된 함수에 대한 구현은 클래스나 컴포넌트를 통해서 하게 된다. 이것은 명세와 구현을 분리할 수 있게 해주어 소프트웨어에 대한 재사용성을 높여준다.

예를 들어 컴퓨터의 마우스나 키보드가 고장이 났다고 해서 컴퓨터 전부를 교체하지는 않는다. 마우스나 컴퓨터만 새로 교체하면 된다. 이때 마우스가 어느 회사 제품이건, 키보드가 몇 개의 키를 가지는 키보드이건 포트만 동일하면 얼마든지 다른 회사의 마우스나 키보드로 교체가 가능하다. 이때 여기서 말하는 포트의 개념이 인터페이스이다. 이러한 의미에서의 인터페이스는 소프트웨어의 독립적 서비스에 대한 명세라고 할 수 있다.

16.2 인터페이스의 특성

- 1) 함수의 선언만을 포함한다.
- 2) 객체를 생성할 수 없다.
- 3) 클래스와 상속관계를 맺을 수 없다.
- 4) 변수를 포함할 수 없다.
- 5) 명세와 구현을 분리할 수 있다.

◦ 인터페이스 작성방법

```
interface 인터페이스명 [extends 상위인터페이스] {
    인터페이스 구성멤버;
}
```

클래스가 인터페이스를 구현할 때에는 implements 구문을 이용하여 구현할 인터페이스를 명시한다. 반면 인터페이스가 다른 인터페이스를 상속할 때에는 extends 구문을 이용한다. 인터페이스는 기능에 대한 명세를 제공하는 장치이므로 인터페이스명은 Serializable이나 Cloneable과 같이 ~able 로 끝나는 것이 보통이다.

자바는 기본적으로 다중상속을 허용하지 않는다. 즉 extends를 이용하여 상속할 수 있는 클래스는 오직 하나뿐이다. 자바에서는 implements 라는 구문을 이용하여 한 개 이상의 인터페이스를 상속할 수 있으며 인터페이스의 implements를 가리켜 "구현한다"라고 부른다. 이것은 자바의 인터페이스가 일반적인 클래스나 추상클래스와는 달리 멤버 필드나 구현 메소드를 포함하고 할 수 없으며 반드시 오버라이딩 되어야 하는 추상메서드 만을 포함하고 있기 때문이다.

자바의 인터페이스는 추상클래스와 마찬가지로 메서드들에 대한 구현부분이 존재하지 않으므로 인스턴스의 생성이 불가능하다. 하지만 Upcasting 메커니즘에 의해 인터페이스를 구현한 클래스들은 인터페이스 형으로 casting이 가능하다.

아래 예제에서 CreditCard, CashCard, PointCard는 Payment라는 인터페이스를 구현하고 있다. Payment 인터페이스는 인스턴스의 생성은 불가능 하지만 Payment 인터페이스를 구현한 클래스들은 Payment 형으로 casting이 가능하다.

```
-----
interface Payment {
    public abstract void pay();
}

class CreditCard implements Payment {
    public void pay() {
        System.out.println("신용카드로 결제합니다.");
    }
}

class CashCard implements Payment {
    public void pay() {
        System.out.println("현금카드로 결제합니다.");
    }
}
```

```
    }  
}  
  
class PointCard implements Payment {  
    public void pay() {  
        System.out.println("포인트카드로 결제합니다.");  
    }  
}  
  
public class AbstractTest {  
    public static void main(String[] ar) {  
        Payment[] pays = new Payment[]{new CreditCard(),  
            new CashCard(),  
            new PointCard()};  
  
        for (int i=0 ; i<pays.length ; i++) {  
            pays[i].pay();  
        }  
    }  
}  
-----
```

16.3 인터페이스의 구성 멤버

자바의 인터페이스는 사용함에 있어서 몇 가지 제약사항이 따른다. 자바의 인터페이스는 클래스와 상속관계를 맺을 수 없다. 클래스가 다른 클래스를 상속할 수 있는 반면 인터페이스는 클래스를 상속할 수 없을 뿐만 아니라 클래스가 인터페이스를 상속할 수 없다. 위에서도 설명하였듯이 인터페이스는 "상속한다"가 아닌 "구현한다"라고 하였다. 또한 추상 클래스는 내부에 멤버 필드를 가질 수 있지만 인터페이스는 멤버 필드를 가질 수 없으며 오직 상수만을 가질 수 있다.

다음은 인터페이스가 가질 수 있는 멤버이다.

- public static final 멤버
- public abstract 멤버 메서드 선언
- public static inner 클래스
- default 메서드

인터페이스는 소프트웨어에서 말하는 관심 분리(Separate of Concern)의 원칙을 반영한 장치이다. 인터페이스는 명세를 표현한 장치이고, 이를 구현하는 것은 클래스 또는 컴포넌트가 된다. 이것은 인터페이스에 의해 명세화 된 함수들은 클래스나 컴포넌트에 의해 구현되며 구현부를 외부에 노출시키지 않고 은폐 시킬 수 있는 방법을 제공해준다. 즉 사용자 관점에서는 인터페이스만 보이며 클래스나 컴포넌트의 내부를 알 필요가 없어지며, 개발자의 입장에서는 인터페이스 규격만을 지키면 클래스나 컴포넌트의 내부 로직이나 알고리즘은 얼마든지 변경이 가능하게 해준다.

○ public static final 멤버

자바는 다중상속에서 발생할 수 있는 문제점의 해결 방법으로 다중상속을 지원하지 않는다. 대신 인터페이스의 구현을 통해 다중상속의 이점과 다형성을 제공한다.

다중상속의 문제점은 상속은 오직 부모 클래스로부터 상속된다는 것에서 생기는 문제점이다. 만약 아래와 같이 다중상속이 허용된다면 어떤 문제가 발생할까?

```
class GrandFather {
    protected int skill;
}

class Father extends GrandFather {
}

class Mother extends GrandFather {
}

class Me extends Father, Mother {
}
```

Father와 Mother는 GrandFather로부터 상속받은 멤버 skill을 가지게 된다. 그렇다면 Me는 어떤

멤버를 상속받는가? 문론 Father의 skill과 Method의 skill을 상속받을 것이다. 문제는 skill은 GrantFather로부터 시작하여 Father나 Mother를 통해 간접적으로 물려받은 멤버이지만 Me의 입장에서는 GrantFather가 아닌 Father나 Mother로 부터 물려받은 것이므로 Father로 부터 물려받은 skill과 Mother로부터 물려받은 skill을 구분해야 한다.

위와 같은 문제를 해결하는 방법으로 interface는 static final 멤버필드 만이 가질 수 있다.

◦ public abstract 멤버 메서드 선언

인터페이스는 의사소통을 위한 매체라는 의미를 가진다. 자바의 인터페이스 또한 의미 그대로 의사소통을 위한 매체로서의 기능만을 수행한다. 이것은 인터페이스가 오직 기능을 수행하기 위한 스펙(specification)만을 제공함을 의미한다. 문론 예외적으로 default 메서드를 통해 기능을 제공할 수 있기는 하지만 default 메서드 이 외의 모든 메서드는 구현부를 가지지 않는 추상 메서드가 된다.

◦ public static inner 클래스

인터페이스는 추상 클래스와 마찬가지로 new 연산에 의한 객체의 생성이 불가능하다. 문론 인터페이스의 구현체를 인터페이스 타입으로 UpCasting하여 사용하는 것은 가능하지만 인터페이스 자체는 직접적으로 new 연산을 통한 객체화가 불가능하다. 이것은 인터페이스가 inner 클래스를 포함한다면 오직 static inner 클래스만 포함할 수 있음을 의미한다. non-static inner 클래스를 이용한 인스턴스의 생성은 생성하고자 하는 inner 클래스의 outer 클래스 객체가 필요한데 인터페이스는 new에 의한 객체의 생성이 불가능 하기 때문이다.

◦ default 메서드

default 메서드는 JDK 1.8(Java 8) 버전부터 제공되는 기능으로 인터페이스 내에 구현 메서드를 포함할 수 있다. default 메서드는 메서드 선언 시 반드시 default 지정자를 이용하여 선언하여야 한다.

```
interface Fightable {
    default void fight(String weapon) {
        System.out.println(weapon + "을 이용하여 싸웁니다.");
    }
}
```

default Method는 상속이 가능하다. 예를 들어 아래와 Fightable 인터페이스를 상속한 Useable 인터페이스는 Fightable의 fight() 메서드를 상속받는다.

```
interface Useable extends Fightable {
}
```

자바는 다중상속이 가지는 문제점으로 인하여 다중상속을 허용하지 않는다고 하였다. 그렇다면 만약 아래와 같이 동일한 default 메서드를 가지는 두 개의 인터페이스를 구현한 구현체는 인터페이스의 default 메서드를 어떻게 호출해야 할까?

```
interface Fightable {
    default void fight(String weapon) {
        System.out.println(weapon + "을 이용하여 싸웁니다.");
    }
}

interface Useable {
    default void fight(String weapon) {
        System.out.println(weapon + "을 사용합니다.");
    }
}
```

동일한 default 메서드를 가지는 복수의 인터페이스를 구현한 클래스는 default 메서드 오버라이딩을 통해 어느 인터페이스의 default 메서드를 호출할 것인가를 결정해야만 한다. 그리고 구현체는 메서드 오버라이딩을 통해 인터페이스를 선택하여 default 메서드를 호출한다.

```
class Person implements Fightable, Useable {

    @Override
    public void fight(String weapon) {
        Fightable.super.fight(weapon);
    }
}

public class AppStart {
    public static void main(String[] args) {
        Person person = new Person();
        person.fight("활");
    }
}
```

문론 default 메서드의 호출은 오버라이딩 된 메서드에서만 가능한 것은 아니며 아래와 같이 다른 메서드에서도 default 메서드를 선택적으로 호출이 가능하다. 그러나 아래와 같이 다른 멤버 메서드에서 default 메서드를 선택적으로 호출이 가능하다 하더라도 implements 한 인터페이스들 내에 동일한 default 메서드가 존재할 경우 반드시 메서드 오버라이딩을 통해 어떤 인터페이스의 default 메서드를 호출 할 것인가를 결정해야 한다.

```
class Person implements Fightable, Useable {

    @Override
    public void fight(String weapon) {
        Fightable.super.fight(weapon);
    }

    public void use(String weapon) {
        Useable.super.fight(weapon);
    }
}
```

```
}  
  
public class AppStart {  
    public static void main(String[] args) {  
        Person person = new Person();  
        person.fight("활");  
        person.use("검");  
    }  
}
```

17. 람다식과 Functional Interface

람다식은 함수적 프로그래밍 기법으로써 익명함수를 지칭하는 용어이다. 람다의 근간은 수학과 기초 컴퓨터 과학 분야에서의 람다 대수이다. 람다 대수는 수학에서 사용하는 함수를 보다 단순하게 표현하는 방법이다.

자바에서의 람다식은 Java 8부터 지원하며 Anonymous Inner Class의 함수적 표현이라고 할 수 있다. 정확하게는 익명 클래스의 재정의 메서드를 하나의 함수로 표현한 것으로 익명함수라고 한다. 그리고 익명함수라는 것은 일급객체 (First Class Citizen)이라는 특징을 갖는다.

일급객체라는 것은 함수 자체를 변수에 대입할 수 있음을 의미한다. 그리고 이것은 함수 호출 시 인수로 때로는 함수의 반환 값으로 사용될 수 있음을 의미하기도 한다.

람다식 사용시의 장단점은 다음과 같다.

◦ 장점

- 1) 코드의 간결성
- 2) 지연연산의 수행
- 3) 병렬처리가 가능하다.

◦ 단점

- 1) 람다식 호출이 까다롭다.
- 2) Stream 사용 시 단순 for 문 혹은 while 문 사용시 성능이 떨어진다.
- 3) 가독성이 낮다.

17.1 람다식의 형식

(타입 매개변수, ...) -> { 실행문; ... }

자바의 람다식은 익명 중첩 클래스(Anonymous Inner Class)에서 재정의 되는 메서드의 함수적 표현이라고 하였다. 그러므로 람다식을 사용하기 위해서는 람다식에 사용될 인터페이스가 필요하다.

아래의 Runnable 인터페이스에는 @FunctionalInterface 어노테이션이 부여된 함수적 인터페이스이다. 함수적 인터페이스(Functional Interface)는 하나의 메서드 선언(default 메서드는 제외)만을 가지는 인터페이스를 말한다.

Interface Runnable

All Known Subinterfaces:

RunnableFuture<V>, RunnableScheduledFuture<V>

All Known Implementing Classes:

AsyncBoxView.ChildState, ForkJoinWorkerThread, FutureTask, RenderableImageProducer, S

Functional Interface:

This is a functional interface and can therefore be used as the assignment target for a lambda expr

```
@FunctionalInterface public interface Runnable
```

Method Summary

Modifier and Type	Method	Description
void	run()	When an object implementing interface Runnable is used to create

위의 인터페이스를 이용하여 익명 중첩 클래스 객체를 선언하는 경우 아래와 같이 객체 생성 시 인터페이스에 선언된 추상 메서드를 오버라이딩 하여야 한다.

```
Runnable r = new Runnable() {
    @Override
    public void run() {
        for (int i=1 ; i<=10 ; i++) {
            System.out.println(i + "번째 작업 실행");
            try {
                Thread.sleep(500);
            } catch (Exception e) {}
        }
    }
};
new Thread(r).start();
```

위의 익명 중첩 클래스 객체를 람다식으로 표현하면 아래와 같다.

```
Runnable r = () -> {
    for (int i=1 ; i<=10 ; i++) {
        System.out.println(i + "번째 작업 실행");
        try {
            Thread.sleep(500);
        } catch (Exception e) {}
    }
};
new Thread(r).start();
```

익명 중첩 클래스 객체는 원형 타입이 가지는 추상 메서드를 반드시 오버라이딩 하여야 한다. 그 이유는 익명 중첩 클래스는 원형이 클래스라면 원형 클래스를 상속하는 클래스이며 원형이 인터페이스라면 원형이 되는 인터페이스의 구현체이기 때문이다.

반면 람다식은 함수이므로 하나의 람다식은 원형이 되는 인터페이스에 선언된 하나의 메서드만을 오버라이딩 할 수 있다. 그러므로 람다식의 원형이 되는 인터페이스는 반드시 하나의 추상 메서드만을 가져야 하며 이러한 인터페이스를 함수적 인터페이스라고 한다.

익명 중첩 클래스는 클래스나 인터페이스 모두가 원형이 될 수 있다. 심지어 객체화가 가능한 구현 클래스라 하더라도 익명 중첩 클래스 객체의 원형이 될 수 있다. 반면 람다식에 사용되는 인터페이스는 반드시 함수적 인터페이스이어야 하며 람다식의 원형이 되는 인터페이스를 가리켜 타겟 타입(target type) 이라고 한다.

함수적 인터페이스라는 것은 반드시 @FunctionalInterface 어노테이션이 부여된 인터페이스를 말하는 것은 아니다. @FunctionalInterface는 컴파일러에게 해당 인터페이스가 오직 하나의 추상 메서드만을 가지는 인터페이스임을 전달함으로써 컴파일 시 체크를 활성화 하는 역할만 할 뿐이다.

함수적 인터페이스는 오직 하나의 추상 메서드(default 메서드는 제외)만을 가지는 인터페이스를 의미한다.

아래 예제의 Computable 인터페이스는 하나의 추상 메서드만을 가지고 있으므로 함수적 인터페이스이며 람다식의 타겟 타입이 될 수 있다.

```
interface Computable {
    int compute(int a, int b);
}
```

람다식은 익명 중첩 클래스에서 재정의 되는 메서드의 함수적 표현이라고 하였다. 그러므로 람다식은 함수 객체이며 이름이 없으므로 익명함수 객체이다. 이때 변수 computer에 저장된 객체는 람다식의 타겟 타입인 Computable 인터페이스 타입의 객체가 된다. 그리고 computer 객체의 compute() 메서드 호출을 통해 람다식을 수행한다.

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

```
    Computable computer = (int a, int b)->{  
        return a + b;  
    };  
  
    int result = computer.compute(10, 20);  
    System.out.println(result);  
}  
}
```

17.2 람다식 작성 규칙

람다식의 형식은 아래와 같다.

`([형식인수[, 형식인수...]]) -> { ... };`

1) 인수가 없는 경우 인수 괄호를 생략할 수 없다.

`() -> { ... }`

2) 인수가 하나인 경우 인수 괄호를 생략할 수 있다.

`a -> { ... }`

3) 인수가 두 개 이상인 경우 인수 괄호를 생략할 수 없다.

`(a, b) -> { ... }`

4) 인수에 대한 자료형은 생략이 가능하다. 인수에 대한 자료형을 생략하는 경우 원형으로 부터 인수에 대한 타입을 추론한다.

`(int a, int b) -> { ... }` OK

`(a, b) -> { ... }` OK

5) 실행부가 단일행인 경우 중괄호를 생략할 수 있으며 중괄호를 생략하는 경우 해당 라인의 연산 결과값을 return 한다.

`(a, b) -> a + b;` `<- a + b` 연산의 결과를 return 함

`(a, b) -> { return a + b; }` `<-` 값을 반환해야 하는 경우 명시적으로 return

17.3 람다식에서의 클래스 멤버 변수와 지역 변수의 참조

◦ 람다식에서의 클래스 멤버 변수 참조

람다식은 익명 중첩 클래스에서 재정의 되는 메서드의 함수 표현식이라고 하였다. 하지만 익명 중첩 클래스와 람다식은 차이점이 있다.

아래의 예제는 Computable 인터페이스의 익명 중첩 클래스 객체를 생성한 후 익명 클래스 객체에서 오버라이딩 된 메서드를 통해 연산을 수행하는 예제이다. 익명 중첩 클래스의 경우 이름이 존재하지 않으므로 명시적으로 생성자를 기술할 수 없을 뿐 일반적인 클래스와 동일하게 new 연산에 의해 객체를 생성한다. 이것은 익명 중첩 클래스 또한 일반 클래스와 마찬가지로 캡슐화가 가능하며 자체적으로 멤버 필드나 멤버 메서드를 가질 수 있음을 의미한다.

익명 중첩 클래스가 캡슐화가 가능하다는 것은 자신만의 객체 범위를 가진다는 것을 의미하며 이것은 바깥쪽 클래스의 멤버가 익명 중첩 클래스의 멤버에 의해 가려질 수 있음을 의미한다. 그러므로 익명 중첩 클래스의 멤버가 바깥쪽 클래스의 멤버를 가릴 경우 범위를 구분하여 참조하여야 한다.

```
interface Computable {
    int compute(int a, int b);
}

public class AppStart {
    int a = 100;
    int b = 10;

    public void func() {
        Computable computer = new Computable() {
            @Override
            public int compute(int a, int b) {
                return (AppStart.this.a * a) + (AppStart.this.b * b);
            }
        };
        int result = computer.compute(3, 2);
        System.out.println(result);
    }

    public static void main(String[] args) {
        new AppStart().func();
    }
}
```

람다식 또한 익명 중첩 클래스와 마찬가지로 인수에 의해 멤버 필드가 가려질 수 있으며 이 경우 익명 중첩 클래스와 동일하게 클래스 멤버와 인수를 구분하여 사용하여야 한다.

```
interface Computable {
    int compute(int a, int b);
}

public class AppStart {
    int a = 100;
    int b = 10;
```

```

public void func() {
    Computable computer = (a, b) ->
        (AppStart.this.a * a) + (AppStart.this.b * b);
    int result = computer.compute(3, 2);
    System.out.println(result);
}

public static void main(String[] args) {
    new AppStart().func();
}
}

```

◦ 람다식에서 지역변수의 사용

익명 중첩 클래스 내에서는 오직 final 지역변수만을 참조할 수 있다. 예를 들어 아래 예제의 익명 중첩 클래스의 메서드에서는 func() 메서드의 지역변수 a와 b를 참조하고 있다. 그러므로 지역변수 a와 b는 final 지정자를 사용하지 않았어도 final 변수가 된다.

```

interface Computable {
    int compute(int a, int b);
}

public class AppStart {

    public void func() {
        int a = 100;
        int b = 10;

        Computable computer = new Computable() {
            @Override
            public int compute(int x, int y) {
                return (a * x) + (b * y);
            }
        };
        int result = computer.compute(3, 2);
        System.out.println(result);
    }

    public static void main(String[] args) {
        new AppStart().func();
    }
}

```

람다식 또한 익명 중첩 클래스의 함수 형태이므로 람다식에서 참조할 수 있는 지역변수는 final 변수이어야 한다. 그러므로 func() 메서드 내의 지역변수 a와 b는 final 지정자가 사용되지 않았어도 final 변수가 된다.

```

interface Computable {
    int compute(int a, int b);
}

public class AppStart {

```

```
public void func() {  
    int a = 100;  
    int b = 10;  
  
    Comutable computer = (int x, int y) -> {  
        return (a * x) + (b * y);  
    };  
  
    int result = computer.compute(3, 2);  
    System.out.println(result);  
}  
  
public static void main(String[] args) {  
    new AppStart().func();  
}  
}
```

18. Collection 프레임워크

프레임워크(FrameWork)는 잘 정의 된, 약속된 구조와 골격을 말한다. 자바의 컬렉션 프레임워크는 객체(인스턴스)를 저장하고 참조하기 위해 잘 정의된 클래스들의 구조를 말한다.

즉 자바의 컬렉션 프레임워크는 별도의 구현이나 이해 없이 자료구조와 알고리즘을 적용할 수 있도록 설계된 클래스들의 집합이다.

18.1 Collection 이란?

Collection이란 공통적인 특성을 지닌 자료들의 집합을 말한다. 물론 이와 같은 특성을 지닌 저장소로 배열이라는 것도 있다. 배열과 컬렉션은 다음과 같은 차이가 있다.

	배열	컬렉션
기본자료형	저장가능	저장불가
저장소의 크기	변경불가	변경가능

9.2 컬렉션의 종류

자바 컬렉션은 java.util 패키지에 포함되어 있으며 아래와 같은 종류가 있다.

종류	정렬	중복허용	구현클래스
Set	정렬안됨	불가	HashSet, TreeSet
Map	정렬안됨	키는 불가, 값은 허용	HashMap, TreeMap, Hashtable, Properties
List	정렬됨	허용	ArrayList, LinkedList, Stack, Vector

18.3 Set

Set은 인터페이스로써 Collection인터페이스와 Iterable인터페이스를 상속하였으며, Set인터페이스를 구현한 구현클래스로써 대표적인 것은 HashSet클래스가 있다. Set 컬렉션의 특징은 다음과 같다.

- 내부적으로 정렬이 이뤄지지 않는다.
- 데이터의 중복이 불가능하다.

Interface Set<E>

Type Parameters:

E - the type of elements maintained by this set

All Superinterfaces:

`Collection<E>`, `Iterable<E>`

All Known Subinterfaces:

`EventSet`, `NavigableSet<E>`, `SortedSet<E>`

All Known Implementing Classes:

`AbstractSet`, `ConcurrentHashMap.KeySetView`, `ConcurrentSkipListSet`, `CopyOnWriteArraySet`, `TreeSet`

아래의 예제는 HashSet 객체를 생성하여 객체를 저장하고 추출하는 방법을 보여준다. HashSet 객체에 데이터를 저장할 때에는 add() 메서드를 이용한다. 하지만 데이터를 추출할 때에는 iterator()메서드를 이용하여 추출하여야 한다.

```
-----
import java.util.Set;
import java.util.HashSet;
import java.util.Iterator;

class A {}
class B {}

public class SetTest {
    public static void main(String[] ar) {
        A a = new A();
        B b = new B();
        String name = new String("KIHEE");

        HashSet hs = new HashSet();
        hs.add(a);    hs.add(b);    hs.add(name);

        Iterator it = hs.iterator();
        while (it.hasNext()) {
            Object o = it.next();
            System.out.println(o);
        }
    }
}
```

```

    }
}
}

```

Iterator는 인터페이스로써 Enumeration과 동일한 기능을 하는 열거자(반복자)로써 주요 메서드는 다음과 같다.

메서드	설명
hasNext()	데이터가 있는지 확인한다.
next()	데이터를 얻어내고 다음 데이터로 이동한다.

HashSet을 이용하는데 있어서 HashSet 컬렉션에 저장할 수 있는 객체의 타입을 지정할 수 있다. 아래의 예제에서는 클래스 S타입의 객체를 저장할 수 있는 HashSet 객체를 생성하였다. 아래와 같이 저장되는 객체를 지정한 경우 iterator()메서드에 의해 반환되는 열거자 또한 타입을 명시하여야만 한다.

아래 예제의 hs는 클래스 S타입의 객체를 저장할 수 있는 HashSet 객체이다. 그러므로 S, A, B 클래스의 인스턴스는 저장이 가능하다. 반면 name은 String클래스형이므로 hs에 담을 수 없다. 또한 hs에 담긴 객체들이 클래스 S타입이므로 iterator()메서드에 의해 반환되는 열거자 또한 클래스 S타입이 된다. 그러므로 iterator()메서드에 의해 반환되는 값을 저장하기 위한 Iterator의 타입은 Iterator<S> 타입이어야 한다.

```

import java.util.HashSet;
import java.util.Iterator;

class S {}
class A extends S {}
class B extends S {}

public class SetTest {
    public static void main(String[] ar) {
        S s = new S();
        A a = new A();
        B b = new B();
        String name = new String("KIHEE");

        HashSet<S> hs = new HashSet<S>();
        hs.add(s);    hs.add(a);    hs.add(b);

        Iterator<S> it = hs.iterator();
        while (it.hasNext()) {
            S o = it.next();
            System.out.println(o);
        }
    }
}

```

18.4 Map

Map 인터페이스를 구현한 대표적인 구현클래스로는 HashMap 클래스가 있다. Map은 키와 값을 쌍으로 하는 EntrySet으로 구성되어 있으며 내부적인 정렬은 발생되지 않는다. 즉 데이터를 꺼낼 때 순서가 보장되지 않는다. 하지만 키값을 이용하여 저장하기 때문에 키값에 의해 원하는 원소의 추출이 가능하다. Map은 데이터는 중복 가능하지만 키값은 중복이 불가능 하다.

Interface Map<K,V>

Type Parameters:

K - the type of keys maintained by this map

V - the type of mapped values

All Known Subinterfaces:

Bindings, ConcurrentMap<K,V>, ConcurrentNavigableMap<K,V>, NavigableMap<K,V>, SortedMap

All Known Implementing Classes:

AbstractMap, Attributes, AuthProvider, ConcurrentHashMap, ConcurrentSkipListMap, LinkedHashMap, PrinterStateReasons, Properties, Provider, RenderingHints, Simple

다음 예제는 HashMap 객체에 객체를 저장하고 추출하는 예이다. HashMap은 데이터를 저장할 때 key와 value를 쌍으로 하여 입력하며, 추출 시에는 key를 이용하여 추출할 수 있다.

```
-----
import java.util.HashMap;

public class MapTest {
    public static void main(String[] ar) {
        HashMap<String, String> hm = new HashMap<String, String>();
        String kim = "Kihee KIM";
        String lee = "Sanghn LEE";
        String choi = "Junghun CHOI";

        hm.put("kim", kim);
        hm.put("lee", lee);
        hm.put("choi", choi);

        System.out.println(hm.get("kim"));
        System.out.println(hm.get("lee"));
        System.out.println(hm.get("choi"));
    }
}
-----
```

아래는 HashMap의 메서드이다.

반환값	메서드	설명
-----	-----	----

void	clear()	객체들을 삭제한다.
boolean	containsKey(key)	일치하는 key값이 있는지 확인한다.
Set	entrySet()	key와 value값을 Entry타입의 객체로 저장된 Set으로 반환한다.
boolean	equals(HashMap)	동일한 HashMap인지를 비교한다.
Entry	get(key)	key에 해당하는 value를 반환한다.
int	hashCode()	해시코드를 반환한다.
boolean	isEmpty()	비었는가를 확인한다.
Set	keySet()	모든 key값을 Set형태로 반환한다.
void	put(key, value)	key값으로 value를 저장한다.
void	putAll(HashMap)	인수로 주어진 HashMap내의 모든 key와 value를 저장한다.
void	remove(key)	key에 해당하는 객체를 삭제한다.
int	size()	현재 저장된 객체의 수를 반환한다.
Collection	values()	저장된 모든 객체를 Collection타입으로 반환한다.

위의 예제에서는 키값을 이용하여 데이터를 추출하고 있다. 하지만 경우에 따라서는 어떤 키가 있는지 알 수 없을 때도 있으므로 아래와 같이 키값을 우선 추출하고 추출된 키값을 이용하여 HashMap으로부터 객체를 추출할 수 있다.

```

import java.util.HashMap;
import java.util.Iterator;
import java.util.Set;

public class MapTest {
    public static void main(String[] ar) {
        HashMap<String, String> hm = new HashMap<String, String>();
        String kim = "Kihee KIM";
        String lee = "Sanghn LEE";
        String choi = "Junghun CHOI";

        hm.put("kim", kim);
        hm.put("lee", lee);
        hm.put("choi", choi);

        Set<String> keys = hm.keySet();
        Iterator<String> it = keys.iterator();
        while (it.hasNext()) {
            String name = hm.get(it.next());
            System.out.println(name);
        }
    }
}

```

Map인터페이스를 구현한 구현클래스로 Hashtable 클래스도 있다. Hashtable의 사용법은 HashMap과 비슷하지만 저장된 요소들의 키를 얻어내는데 있어서 Iterator가 아닌 Enumeration타입도 사용할 수 있다.


```

import java.util.Enumeration;
import java.util.Hashtable;

public class MapTest {
    public static void main(String[] ar) {
        Hashtable<String, String> ht = new Hashtable<String, String>();
        String kim = "Kihee KIM";
        String lee = "Sanghn LEE";
        String choi = "Junghun CHOI";

        ht.put("kim", kim);
        ht.put("lee", lee);
        ht.put("choi", choi);

        Enumeration<String> keys = ht.keys();
        while (keys.hasMoreElements()) {
            String key = keys.nextElement();
            String name = ht.get(key);
            System.out.println(name);
        }
    }
}

```

Enumeration은 인터페이스로써 Iterator와 마찬가지로 열거자이며 다음과 같은 메소드가 있다.

반환값	메소드	설명
boolean	hasMoreElements()	다음 요소가 있는지 검사한다.
Element	nextElement()	하나의 요소를 꺼내고 다음 요소로 이동한다.

18.5 List

List는 Set과 Map과는 달리 내부적으로 정렬이 이뤄진다. 즉 데이터를 저장하고 추출하는데 있어서 그 순서가 지켜진다. 뿐만 아니라 데이터 중복이 불가능한 Set이나 키값의 중복이 불가능한 Map과는 달리 데이터의 중복이 가능하다.

List 인터페이스를 구현한 구현클래스로 대표적인 것은 ArrayList와 Vector가 있다.

Interface List<E>

Type Parameters:

E - the type of elements in this list

All Superinterfaces:

Collection<E>, Iterable<E>

All Known Implementing Classes:

AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArray, Vector

아래의 예제는 ArrayList를 이용하여 객체를 저장하고 추출하는 예제이다. List타입의 컬렉션은 내부적으로 정렬이 되므로 아래와 같이 index값을 이용하여 저장된 데이터의 추출이 가능하다.

```
-----
import java.util.ArrayList;

abstract class Person {
    protected String name = null;

    abstract public void disp();
}

class Student extends Person {
    public Student(String name) {
        this.name = name;
    }

    public void disp() {
        System.out.println("학생 : " + name);
    }
}

class Teacher extends Person {
    public Teacher(String name) {
        this.name = name;
    }

    public void disp() {
        System.out.println("강사 : " + name);
    }
}

public class MapTest {
    public static void main(String[] ar) {
```

```

ArrayList<Person> al = new ArrayList<Person>();
Person t = new Teacher("김기희");
Person s = new Student("김재희");
al.add(t);    al.add(s);

Object[] objs = al.toArray();
for (int i=0 ; i<objs.length ; i++) {
    ((Person)objs[i]).disp();
}
}
}

```

ArrayList의 중요 메서드는 아래와 같다.

반환값	메소드	설명
boolean	add(E)	맨 뒤에 요소추가
void	add(int, E)	지정된 인덱스에 요소삽입
boolean	addAll(Collection)	컬렉션의 모든 요소를 뒤에 추가
void	clear()	모든 요소를 삭제
boolean	contains(Object)	객체를 포함하는지를 확인
E	elementAt(int)	지정된 인덱스의 요소를 반환
E	get(int)	지정된 인덱스의 요소를 반환
int	indexOf(Object)	지정된 요소의 위치 값을 반환
boolean	isEmpty()	비었는가를 확인
E	remove(index)	인덱스에 해당하는 요소를 반환하고 삭제
boolean	remove(Object)	동일한 객체를 삭제
int	size()	저장된 요소의 수를 반환
Object[]	toArray()	모든 요소를 배열로 반환

18.6 Stack

Stack 클래스는 List 컬렉션 클래스의 Vector 클래스를 상속받아 전형적인 스택 메모리 구조의 클래스를 제공한다.

Class Stack<E>

```
java.lang.Object
java.util.AbstractCollection<E>
java.util.AbstractList<E>
java.util.Vector<E>
java.util.Stack<E>
```

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

```
public class Stack<E> extends Vector<E>
```

Stack은 선형 메모리 공간에 데이터를 저장하며 LIFO(후입선출)의 시멘틱을 따르는 자료구조이다. 그러므로 나중에 저장된 데이터가 가장 먼저 인출되는 구조이다.

```
public class Stack<E> extends Vector<E> {
    ...
}
```

Stack 클래스의 제공 메서드는 아래와 같다.

메서드	설명
boolean empty()	스택이 비어있는가를 검사
E peek()	스택으로부터 마지막 요소를 취득한다.
E pop()	스택으로부터 마지막 요소를 취득한다. 취득된 요소는 스택에서 제거된다.
E push(E e)	스택에 요소를 담는다.
int search(Object o)	스택에서 객체나 나오는 순서 값을 반환한다. 존재하지 않을 경우 -1을 반환

18.7 Queue

Queue는 Stack과는 달리 인터페이스로 제공되며 FIFO(선입선출) 시멘틱을 따르는 자료구조이다.

Interface Queue<E>

Type Parameters:

E - the type of elements held in this queue

All Superinterfaces:

Collection<E>, Iterable<E>

All Known Subinterfaces:

BlockingDeque<E>, BlockingQueue<E>, Deque<E>, TransferQueue<E>

All Known Implementing Classes:

AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedDeque, ConcurrentL
LinkedBlockingQueue, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, Prio

Queue 인터페이스의 대표적인 구현체로는 LinkedList 클래스가 있다.

```
public class LinkedList<E> extends AbstractSequentialList<E> implements
List<E>, Deque<E>, Cloneable, java.io.Serializable {
    ...
}
```

Queue 인터페이스의 메서드는 아래와 같다.

메서드	설명
boolean isEmpty()	큐가 비어있는가를 검사
boolean add(E e)	큐의 마지막에 요소를 삽입한다.
E element()	큐의 맨 앞의 요소를 반환한다. 큐에 요소가 없으면 NoSuchElementException 예외 발생
E poll()	큐의 맨 앞의 요소를 반환한다. 반환된 요소는 큐에서 제거된다. 큐에 요소가 없으면 null을 반환한다.
E peek()	큐의 맨 앞의 요소를 반환한다. 큐에 요소가 없으면 null을 반환한다.
E remove()	큐의 맨 앞의 요소를 제거한다.

19. java.lang 패키지

java.lang 패키지는 자바 프로그래밍에 필요한 기본 클래스들을 포함하고 있으며 import 문이 없어도 자동으로 포함한다.

19.1 Object 클래스

Object 클래스는 모든 클래스의 최상위 클래스로써 extends 구문에 의해 상속을 하지 않더라도 자동으로 상속이 된다. Object 클래스의 주요 메서드 중 아래의 메서드 만큼은 외워 두도록 하자.

반환형	메서드	설명
Object	clone()	객체의 복사본을 반환한다.
boolean	equals(Object)	동일한 객체인지 비교한다.
Class	getClass()	객체의 클래스타입을 반환한다.
String	toString()	객체를 대표하는 값을 반환한다.

◦ toString() 메서드

toString() 메서드는 객체를 대표하는 값을 반환한다. 예를 들어 printf() 메서드를 호출할 때 실인수로 객체를 전달하면 printf() 메서드는 인수로 전달받은 객체의 식별자를 출력하게 된다. 이때 식별자의 출력은 toString() 메서드의 호출을 통해 얻어진 값이다.

아래 예제의 Person 클래스는 extends 문이 없어도 자동으로 Object 클래스를 상속한다. 그러므로 Person 클래스는 Object 클래스형으로 캐스팅이 가능하며 (p instanceof Object) 라는 수식의 조건이 참이 되어 toString() 메서드를 호출한다. toString() 메서드는 Object 클래스의 메서드이지만 다형성에 의해 Person 클래스의 toString() 메서드를 호출하게 된다.

```
-----
class Person {
    String name;
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String toString() {
        String str = "이름 : " + name;
        str += "\n";
        str += "나이 : " + age;
        return str;
    }
}
```

```
public class LangPackage {
    public static void main(String[] args) {
        Object p = new Person("김기희", 20);

        if (p instanceof Object)
            System.out.println(p.toString());
    }
}
```

◦ equals(Object o) 메서드

equals(Object o) 메서드는 동일한 객체인가를 비교하는 메서드이다. 우리가 객체를 비교함에 있어 등가연산자 (==)와 equals() 메서드를 이용할 수 있다. 이때 등가연산자를 이용한 비교는 동일한 레퍼런스인가를 비교하며 equals() 메서드는 객체가 가지고 있는 멤버가 동일한가를 비교하게 된다.

이때 동일한 레퍼런스라 함은 실제로 동일한 인스턴스를 가리키는가를 의미한다. 반면 equals() 메서드는 서로 다른 인스턴스를 가리키는 객체라 할 지라도 객체 내의 멤버의 속성값이 동일하다면 true 값을 반환한다.

아래의 예제는 동일한 객체인가를 비교하는 예제이다. equals() 메서드는 객체를 비교하는 메서드로써 두 객체가 동일한 객체인가를 비교한다. 동일한 객체라는 것은 동일한 클래스형을 의미하는 것이 아니라 두 객체가 동일한 인스턴스를 참조함을 의미한다.

```
class Person {
    String name;
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String toString() {
        String str = "이름 : " + name;
        str += "\n";
        str += "나이 : " + age;
        return str;
    }
}

public class LangPackage {
    public static void main(String[] args) {
        Person p1 = new Person("김기희", 20);
        Person p2 = p1;

        if (p1.equals(p2))
            System.out.println("두 객체는 동일한 객체입니다.");
        else
```

```

        System.out.println("두 객체는 동일한 객체가 아닙니다.");
    }
}
-----

```

◦ clone() 메서드

clone() 메서드는 객체의 복사본을 생성하고 반환하는 메서드이다. 복제 기능을 사용하기 위해서는 Cloneable 인터페이스를 구현하여야만 한다. 하지만 API를 보면 Cloneable 인터페이스에는 그 어떤 메서드도 포함하고 있지 않다. 즉 Cloneable 인터페이스를 구현(implements)하는 것으로 복제 기능이 제공되는가를 판단하게 되며 객체의 복제는 Object 클래스의 clone() 메서드의 오버라이딩에 의해 이뤄진다.

아래의 예제는 clone() 메서드에 의해 객체를 복제하는 예제이다. clone() 메서드는 protected 메서드로서 복제를 위해서는 반드시 재정의의 하여야 한다. 이때 복제된 객체는 동일한 속성과 동일한 데이터를 가지고 있지만 원본과는 다른 위치의 메모리를 참조하는 인스턴스이다.

```

-----
class Person implements Cloneable {
    String name;
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String toString() {
        String str = "이름 : " + name;
        str += "\n";
        str += "나이 : " + age;
        return str;
    }

    public Person clone() throws CloneNotSupportedException {
        Person p = (Person) super.clone();
        p.name = this.name;
        p.age = this.age;
        return p;
    }
}

public class AppStart {
    public static void main(String[] args) {
        Person p1 = new Person("김기희", 20);
        Person p2 = null;

        try {
            p2 = p1.clone();
        } catch (CloneNotSupportedException e) {

```



```

        System.out.println("복제가 지원되지 않는 객체입니다.");
    }

    if (p1.equals(p2))
        System.out.println("두 객체는 동일한 객체입니다.");
    else
        System.out.println("두 객체는 동일한 객체가 아닙니다.");
    }
}

```

아래의 API Document에서 볼 수 있듯이 clone() 메서드는 protected 메서드로써 외부에서는 호출이 불가능하므로 재정의의 하여야만 한다. 또한 CloneNotSupportedException 예외를 throws 하므로 재정의되는 메서드는 반드시 CloneNotSupportedException 예외를 throws 하는 메서드이어야 한다.

clone

```
protected Object clone() throws CloneNotSupportedException
```

Creates and returns a copy of this object. The precise meaning of "copy" may depend on the class expression:

```
x.clone() != x
```

will be true, and that the expression:

위의 예제에서 p2객체는 p1객체의 clone() 메서드의 호출을 통해 얻어진 복제된 인스턴스이다. 그러므로 p2.equals(p1) 또는 p1.equals(p2)는 true를 반환하여야만 한다. 하지만 위의 예제를 실행해 보면 두 객체가 동일하지 않다는 결과가 출력된다.

p1.equals(p2)로 사용하던 p2.equals(p1)으로 사용하던 p1과 p2는 Person 클래스의 인스턴스이다. 그러므로 p1.equals(p2) 또는 p2.equals(p1)은 Person 클래스의 equals() 메서드를 호출하는 것이지만 Person 클래스에는 equals() 메서드가 없으므로 Person 클래스의 상위 클래스인 Object 클래스의 equals() 메서드가 수행되며 그 결과로서 false 값이 반환되고 있는 것이다.

Person 클래스의 인스턴스에 대하여 equals() 메서드를 사용하고자 한다면 Person 클래스에 equals() 메서드를 재정의 하여야 할 것이며 재정의 되는 equals() 메서드에서는 Person 클래스의 모든 멤버들의 값의 동일 여부를 판단하고 그 결과를 반환하여야 할 것이다.

아래 예제의 Person 클래스에는 인스턴스를 비교하기 위한 equals() 메서드가 재정의 되어 있다.

```
class Person implements Cloneable {
```

```

String name;
int age;

public Person(String name, int age) {
    this.name = name;
    this.age = age;
}

public String toString() {
    String str = "이름 : " + name;
    str += "\n";
    str += "나이 : " + age;
    return str;
}

public Person clone() throws CloneNotSupportedException {
    Person p = (Person) super.clone();
    p.name = this.name;
    p.age = this.age;
    return p;
}

public boolean equals(Object o) {
    boolean result = true;
    Person p = (Person) o;
    if (!name.equals(p.name)) {
        result = false;
    } else if (age != p.age) {
        result = false;
    }
    return result;
}
}

```

◦ getClass() 메서드

getClass() 메서드는 인스턴스가 어떤 클래스 타입인가를 Class 타입으로 반환한다.

```

class Person implements Cloneable {
    String name;
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String toString() {
        String str = "이름 : " + name;
        str += "\n";
        str += "나이 : " + age;
        return str;
    }
}

```

```
    public Person clone() throws CloneNotSupportedException {
        Person p = new Person(null, 0);
        p.name = this.name;
        p.age = this.age;
        return p;
    }
}

public class LangPackage {
    public static void main(String[] args) {
        Person p = new Person("김기희", 20);
        Class c = p.getClass();

        System.out.println("객체 p는 " + c.getName() + " 클래스 형입니다.");
    }
}
-----
```

19.2 String 클래스와 StringBuffer 클래스

String 클래스와 StringBuffer 클래스는 문자열을 위한 클래스이다. 두 클래스의 차이점은 다음과 같다.

◦ String

> 기본형처럼 사용 가능하며 저장된 데이터를 편집할 수 없다.

◦ StringBuffer

> 자동으로 크기가 조절되는 버퍼를 제공하여 저장된 문자열을 편집할 수 있으며 동기화가 지원된다.

20.2.1 String 클래스

String 클래스는 char형 배열로서 기본형처럼 사용이 가능하다. 아래는 name이라는 String객체를 생성하는 구문이다. 이때 문자열을 가리키는 객체인 name은 스택에 할당된다.

```
String name = "김기희";
```

위의 구문은 아래와 같이 사용할 수 도 있다. 하지만 아래의 구문은 new 연산자에 의한 객체의 생성이므로 문자열을 저장하기 위한 공간은 힙 영역에 할당된다.

```
String name = new String("김기희");
```

아래는 "==" 연산자를 이용한 문자열 비교의 예이다. 아래 예제의 결과는 str1과 str2가 동일한 문자열임을 나타낸다. 하지만 이것은 동일한 문자열인가를 비교하는 것이 아니라 str1과 str2가 참조하는 위치가 동일한가를 비교하는 것이다.

```
-----
class StringTest {
    public static void main(String[] args) {
        String str1 = "김기희";
        String str2 = "김기희";

        if (str1 == str2)
            System.out.println("동일한 문자열입니다.");
        else
            System.out.println("동일한 문자열이 아닙니다.");
    }
}
-----
```

아래 예제의 str1과 str2는 동일한 문자열임에도 다른 문자열로 간주된다. 이것은 "==" 연산자가 str1과 str2에 저장된 문자열을 비교하는 것이 아님을 보여준다.

```

class StringTest {

    public static void main(String[] args) {
        String str1 = "김기희";
        String str2 = new String("김기희");

        if (str1 == str2)
            System.out.println("동일한 문자열입니다.");
        else
            System.out.println("동일한 문자열이 아닙니다.");
    }
}

```

그러므로 문자열의 비교는 equals()메서드를 이용하도록 하여야 한다. equals() 메서드는 문자열 비교가 아닌 객체를 비교하는 메서드이다. 하지만 equals() 메서드를 이용하는 경우 두 비교대상의 toString()메서드 호출에 의해 반환되는 값을 이용하여 비교하므로 결국 문자열을 비교하게 된다.

```

class StringTest {

    public static void main(String[] args) {
        String str1 = "김기희";
        String str2 = new String("김기희");

        if (str1.equals(str2))
            System.out.println("동일한 문자열입니다.");
        else
            System.out.println("동일한 문자열이 아닙니다.");
    }
}

```

○ String 관련 메소드

반환값	메서드	설명
int	indexOf(String)	문자열의 위치를 찾는다.
int	indexOf(String, int)	문자열의 n번째 위치를 찾는다.
int	lastIndexOf(String)	문자열의 위치를 뒤에서 부터 찾는다.
int	lastIndexOf(String, int)	문자열의 n번째 위치를 뒤에서 부터 찾는다.
char	charAt(int)	n번째 위치의 문자를 반환한다.
String	substring(int)	n번째부터의 문자열을 반환한다.
String	substring(int, int)	n번째부터 m번째까지의 문자열을 반환한다.
String	replace(char, char)	문자열을 치환한다.

String	trim()	문자열 앞 뒤의 공백문자를 제거한다.
String	toUpperCase()	문자열을 대문자로 반환한다.
String	toLowerCase	문자열을 소문자로 반환한다.
String	valueOf(Object)	인수로 주어진 객체를 문자열로 반환한다.

19.2.2 StringBuffer 클래스

StringBuffer 클래스는 자동으로 크기가 조절되는 문자열을 저장하기 위한 버퍼를 제공하며, 버퍼에 저장된 문자열의 편집이 가능하다. StringBuffer의 생성자는 다음과 같다.

생성자	설명
StringBuffer()	디폴트 생성자
StringBuffer(int)	버퍼크기를 지정하는 생성자
StringBuffer(String)	문자열의 길이만큼 버퍼를 지정하고 초기화하는 생성자

StringBuffer 클래스의 주요 메서드는 다음과 같다.

반환값	메서드	설명
int	capacity()	현재 할당된 버퍼의 크기를 반환한다.
int	length()	버퍼에 저장된 문자열의 길이를 반환한다.
StringBuffer	append(String)	버퍼에 저장된 문자열에 문자를 추가한다.

```

-----
class StringTest {

public static void main(String[] args) {
StringBuffer sb = new StringBuffer("김");
sb = sb.append("기희");

System.out.println("확보된 공간 : " + sb.capacity());
System.out.println("저장된 문자수 : " + sb.length());
}
}
-----

```

19.3 Wrapper 클래스

Wrapper 클래스는 기본형 데이터를 객체형으로 다루기 위해 사용되는 클래스이다. Wrapper 클래스는 기본형과 대응되는 클래스들을 통칭한다.

아래는 기본형에 대한 wrapper 클래스를 나타낸다.

기본형	Wrapper 클래스
boolean	Boolean
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Wrapper 클래스는 기본형을 조작하기 위한 메서드를 제공하며 기본형과 String형 간의 형변환 메서드를 제공한다.

XXX형을 String형으로 변환하는 방법

```
String str = XXX.toString(값);
String str = String.valueOf(값);
```

```
-----
public class WrapperTest {
    public static void main(String[] ar) {
        float pi = 3.14F;
        String str = Float.toString(pi);

        System.out.println(str);
    }
}
-----
```

String형을 XXX형으로 변환하는 방법

```
XXX value = XXX.parseXXX(String);
```

```
-----
public class WrapperTest {
    public static void main(String[] ar) {
        String str = new String("3.14");
        Double d = Double.parseDouble(str);

        System.out.println(d);
    }
}
-----
```

※ Boxing, UnBoxing, Auto-Boxing

Boxing 이란? Wrapper 클래스를 사용하는데 있어서 기본형이 자동으로 Wrapper 클래스형으로 변환되는 것을 말한다. 반대로 Wrapper 클래스형이 기본형으로 변환이 가능하며 이를 UnBoxing 이라고 한다.

아래의 예제에서 al은 Integer 클래스 타입만을 담을 수 있도록 Generic된 ArrayList 이다. 그러므로 al에 정수를 저장하기 위해서는 al.add(new Integer(100))과 같이 Integer 클래스의 인스턴스를 생성하여 저장을 하여야 한다. 하지만 al.add(200)과 같이 Integer 클래스의 인스턴스가 아닌 기본형을 지정할 수 있는데 이때 기본형을 이용한 Integer 클래스형 객체가 자동으로 생성되어 ArrayList에 저장되는데 이것을 Boxing이라고 하며, 자동으로 Boxing이 이뤄지므로 Auto-Boxing이라고 한다.

```
import java.util.ArrayList;

public class WrapperTest {
    public static void main(String[] ar) {
        ArrayList<Integer> al = new ArrayList<Integer>();
        al.add(new Integer(100));
        al.add(200);

        for (int i = 0; i < al.size(); i++) {
            Integer tmp = al.get(i);
            System.out.println(tmp);
        }
    }
}
```
