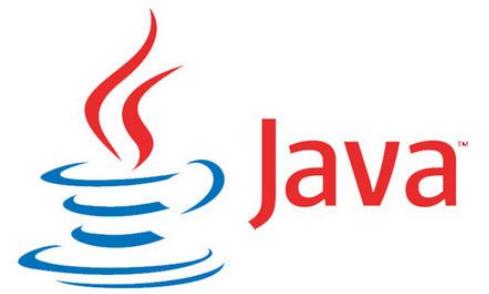


자바 프로그래밍 기초

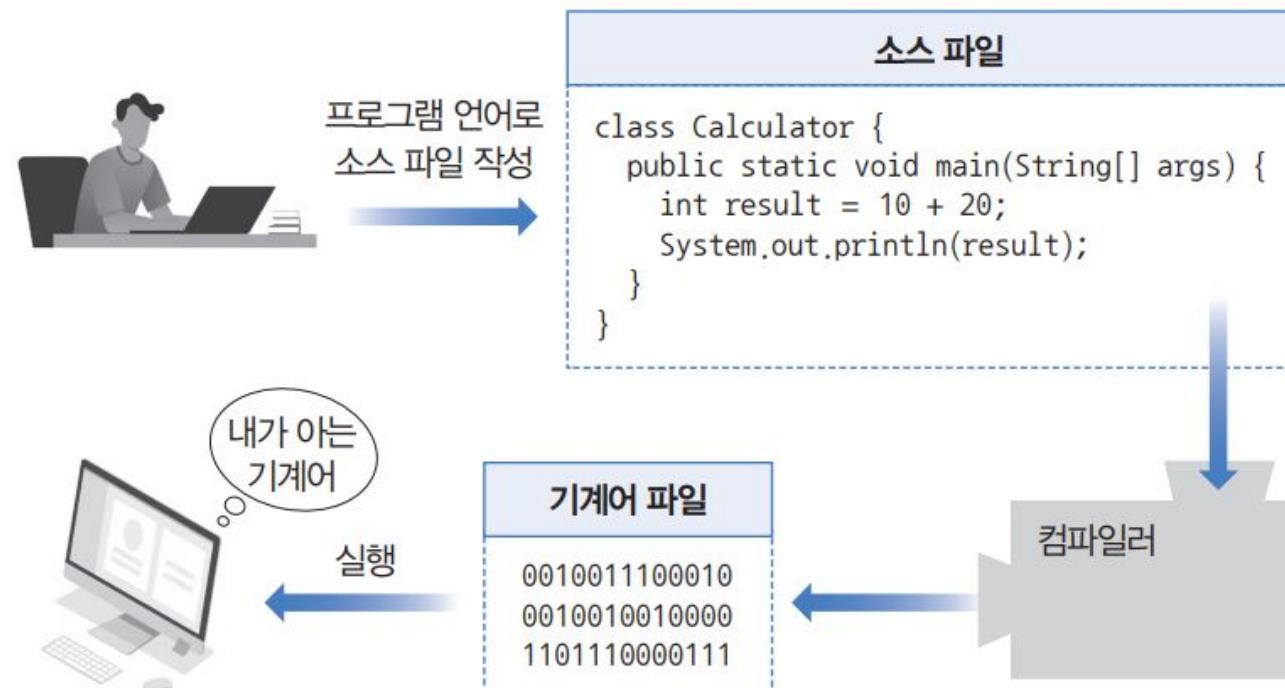
임성국 (eventia@gmail.com)



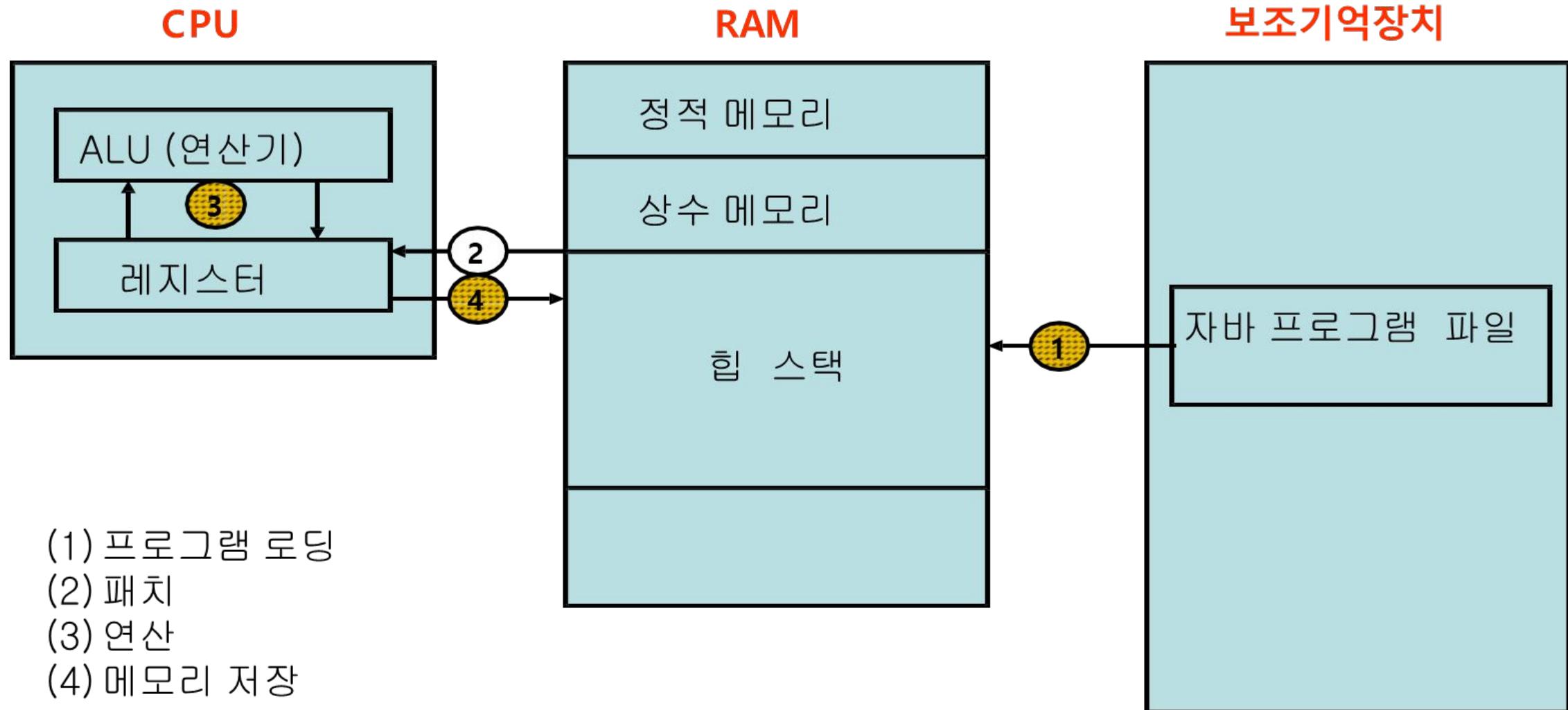
Chapter 01 자바 시작하기

프로그래밍 언어의 역할

- 사람과 컴퓨터의 대화 도움
- 사람의 언어와 기계어 사이에서 다리 역할 → 컴파일
- 고급(인간 중심) 언어와 저급(기계 중심) 언어로 구분

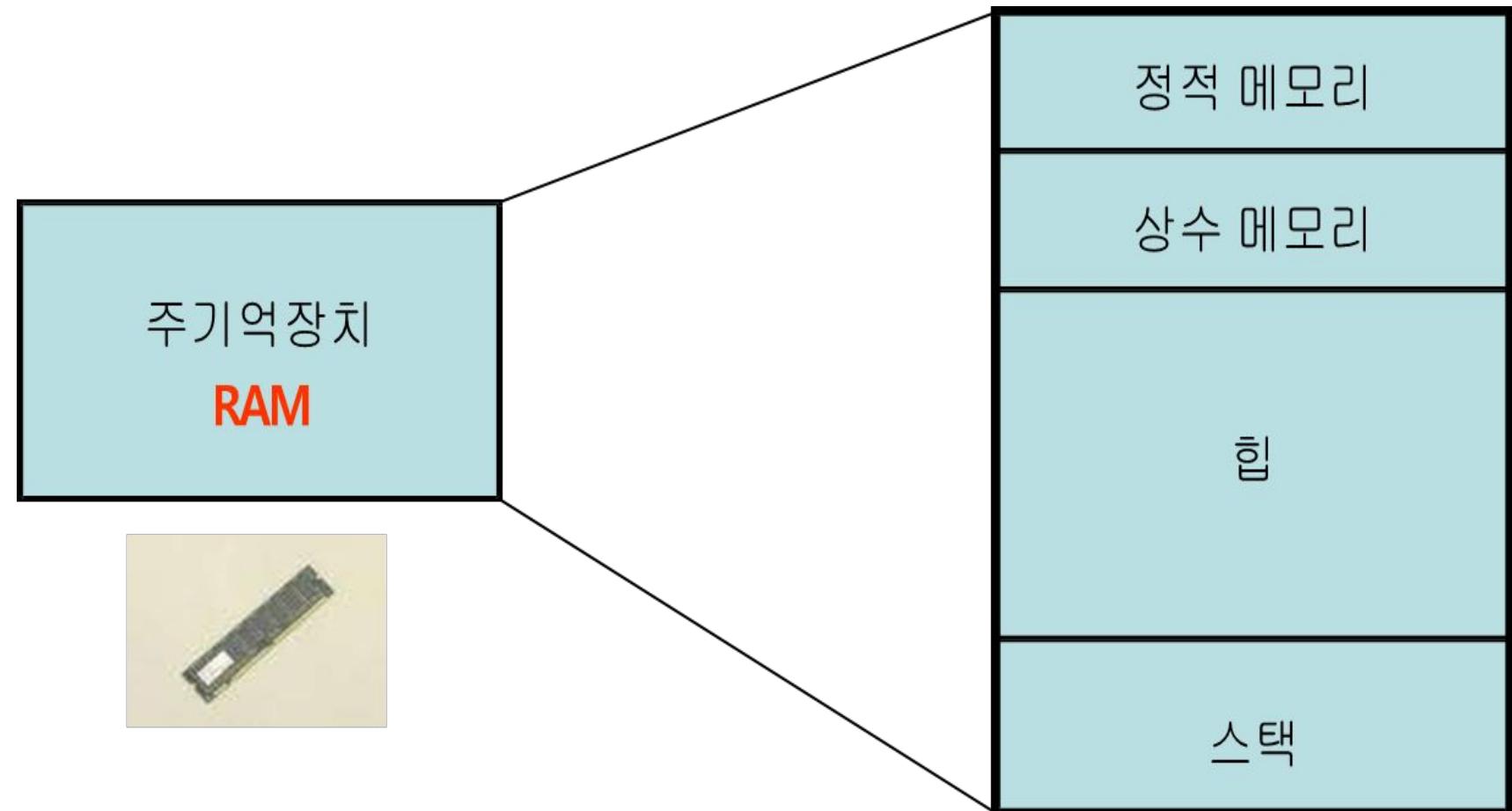


프로그램 실행 과정



메모리 종류

- 주 기억장치는 물리적으로 정적메모리, 상수메모리, 힙, 스택을 구별하지 않는다.

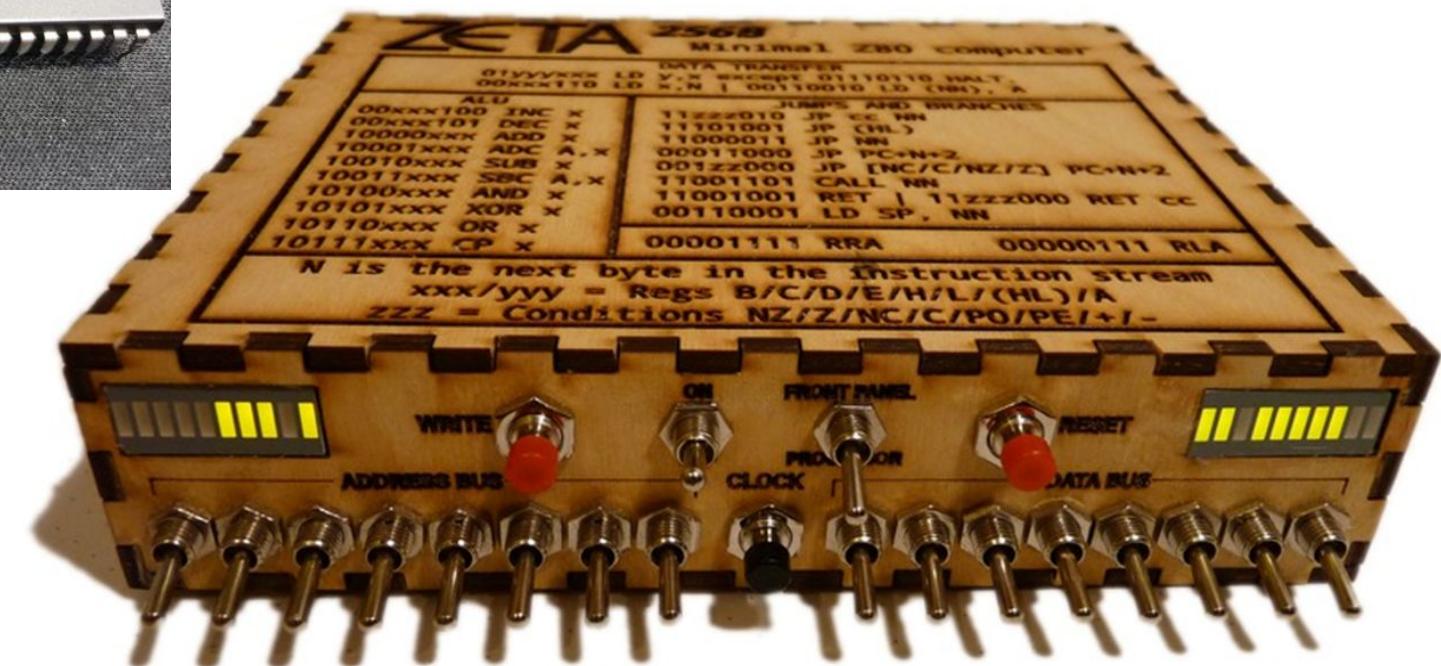


메모리 종류

❑ 속도에 따른 메모리 종류

- ❑ 레지스터 (register) - CPU 내부의 고속 기억장치
- ❑ 내부 기억장치 - RAM과 같이 CPU 버스와 직접 연결된 메인 메모리
 - ❑ 스택 (stack) - 메소드와 변수를 위한 내부 기억장치 (RAM)
 - ❑ 힙 (heap) - 객체 할당을 위한 내부 기억장치 (RAM)
 - ❑ 정적 메모리 (static memory) - 공유 메소드나 공유 변수 저장
 - ❑ 상수 메모리 (constants pool) - 변경되지 않는 값을 저장
- ❑ 외부 기억장치 - 하드디스크, 플로피 디스크, 플래시 메모리와 같은 보조 기억장치

초기 컴퓨터 - Z80 : 인텔 8080 이후, 8비트,





자바 소개

- 안드로이드 및 데스크톱 애플리케이션이나 웹사이트를 개발하는 핵심 언어
- 1995년 썬마이크로시스템즈(Sun Microsystems)에서 처음 발표
- 2010년 오라클에서 썬 인수, 자바 개발 도구(JDK) 배포해 기술 지원



자바 특징

- 윈도우, 맥OS, 리눅스 등 모든 운영체제에서 실행 가능
- 먼저 객체(부품)를 만들고, 객체들을 서로 연결해서 더 큰 프로그램을 완성시키는 객체 지향 프로그래밍(OOP)에 최적화된 언어
- 메모리(RAM)를 자동 정리해 메모리 관리에 용이
- 무료로 다운로드해서 사용할 수 있는 오픈 소스 라이브러리(Open Source Library)가 풍부

자바 개발 도구(JDK) 설치

- JDK에는 Open JDK와 Oracle JDK 두 가지 있음

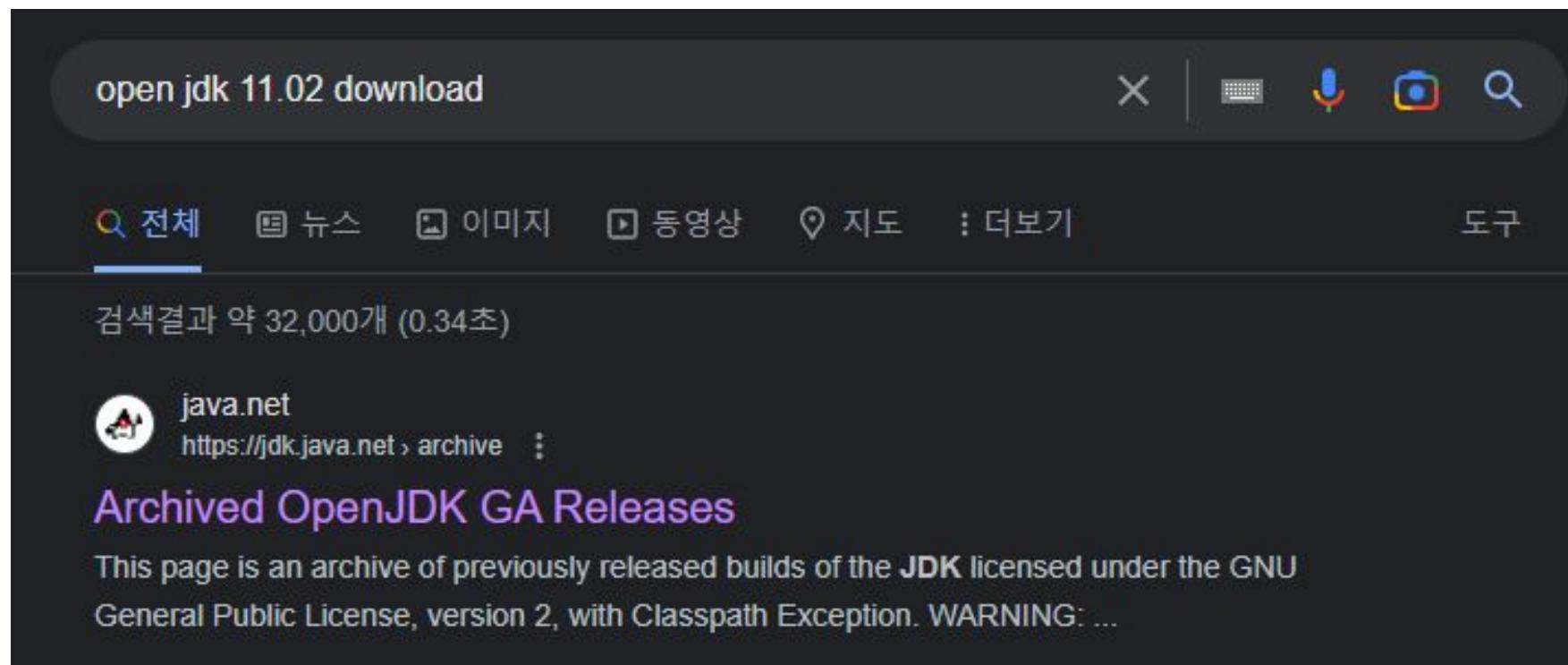
구분	Open JDK	Oracle JDK
라이선스	GNU GPL version 2	Oracle Technology Network License
사용료	무료	개발 및 학습용: 무료, 상업용: 유료
개발 소스 공개 의무	없음	없음

- Open JDK 다운로드: <https://jdk.java.net> 및 <https://adoptium.net>
- Oracle JDK 다운로드: <https://www.oracle.com/java/technologies/downloads>
- JDK LTS 버전 다운로드: <https://adoptium.net>

- 하위호환성이 좋다.
- OracleJDK(유료) 와 OpenJDK(무료)가 있다.
- OracleJDK에는 LTS버전이 있다.
- 전자정부표준프레임워크에 적용되는 버전은 자바8(3.7이상), 자바11(4.0)이다.
 - **자바8(LTS)** - 전자정부표준프레임워크 3.7 이상에서 사용
 - 자바9, 자바10
 - **자바11(LTS)** - 전자정부표준프레임워크 4.0 에서 최초 도입
 - 자바12, 자바13, 자바14, 자바15, 자바16
 - **자바17(LTS)** - 2021.09.15 출시
 - 자바18, 자바19, 자바20, 자바21(2023.09.20), 자바22(2024.03.19 예정), etc

자바 환경 설정

- 윈도우버전 jdk 11 버전 다운로드 :
- 검색 keyword :
 - jdk 11.0.22 download
 - JDK 8u202 download --- (1.8.202 이후 라이센스 변경으로 유료화)



압축 푼 후 디렉토리를 다음과 같이 만들어 둘 것

- jdk11은 압축 풀고 복사, 1.8은 설치시 경로 지정

C:\Java\jdk11
 \bin
 \conf
 \include
 \jmods
 \legal
 \lib

~~C:\Java\jdk1.8.0_202~~ \bin
 \include
 \jre
 \lib

~~C:\Java\jre1.8.0_202~~ \bin
 \lib

자바환경변수 설정하기 (2024년 3월 상태를 기준으로 11.0.22 사용)

1. JAVA_HOME

C:\Java\jdk11

2. path

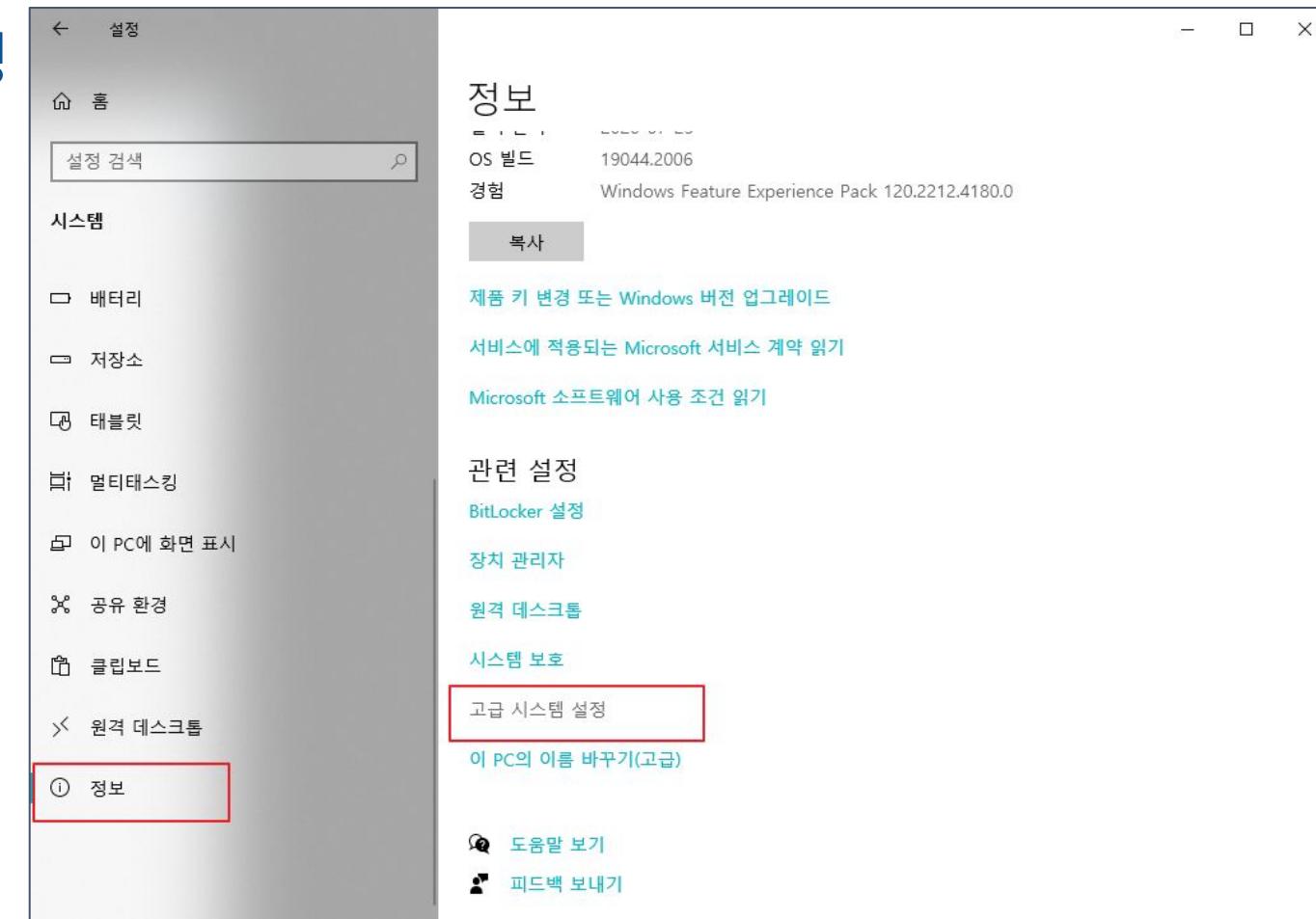
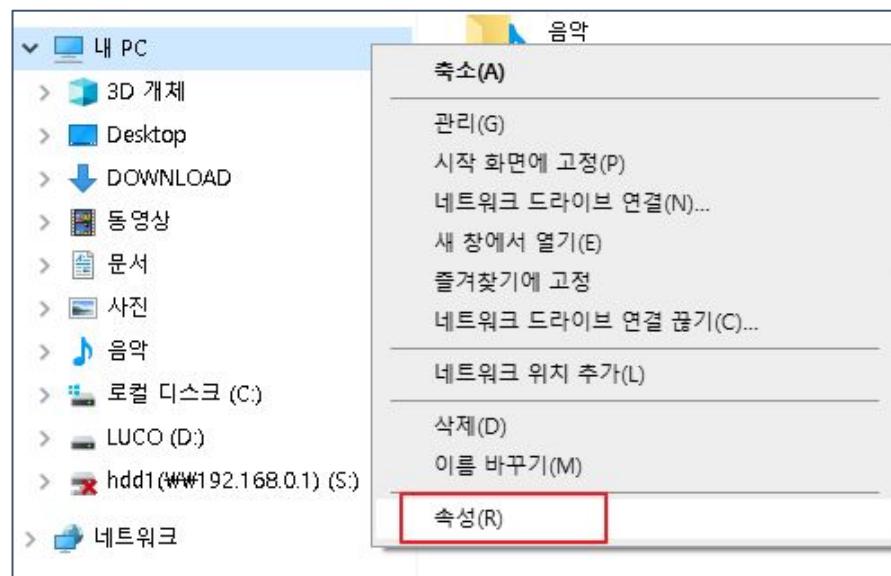
%JAVA_HOME%\bin

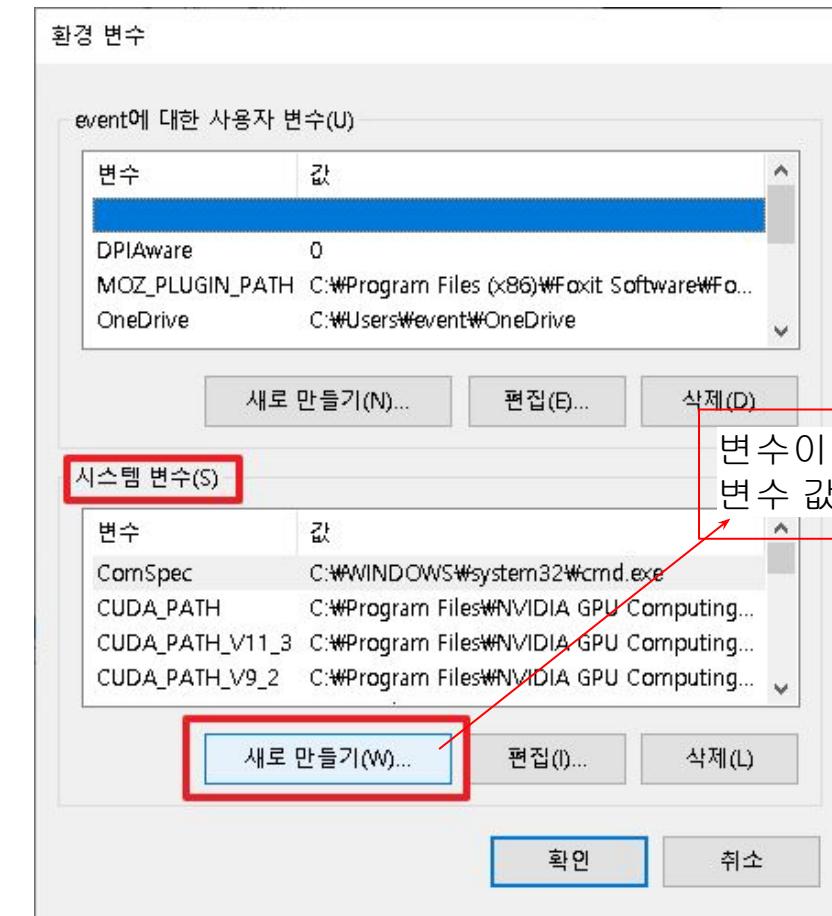
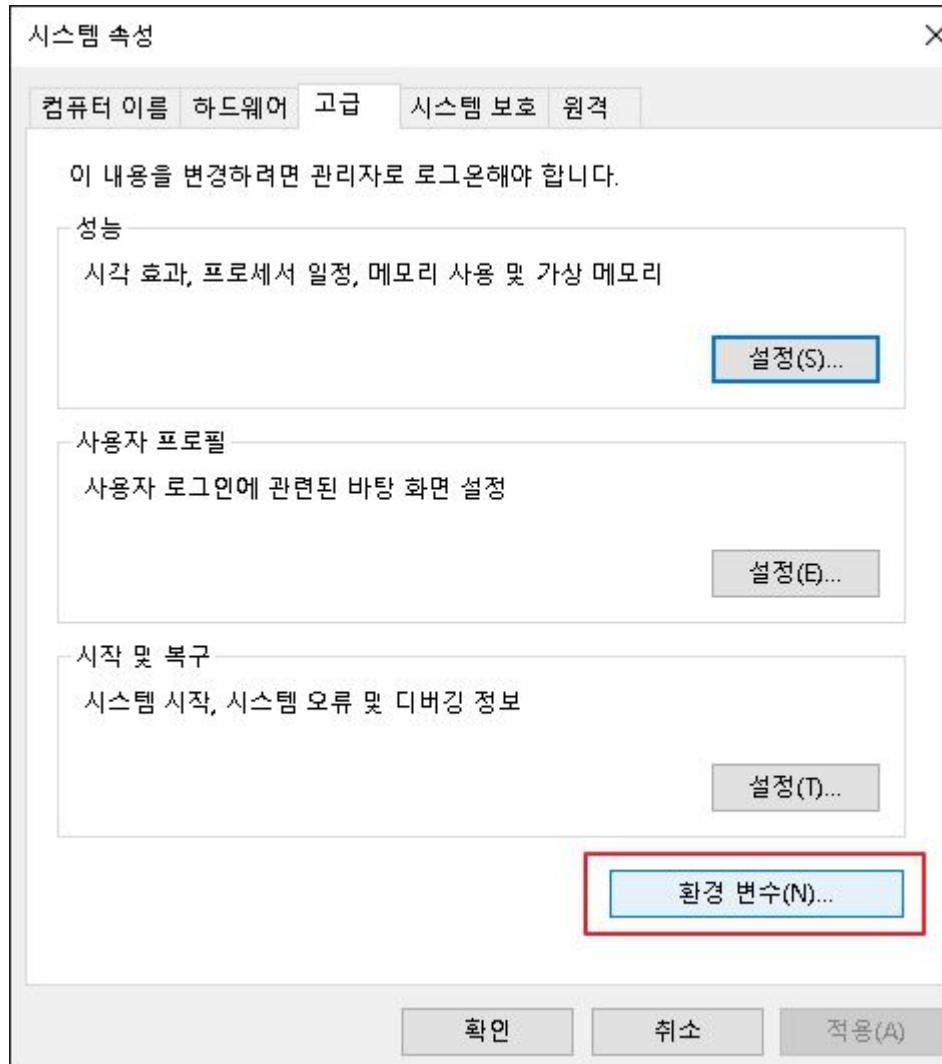
3. classpath (필요없음)

%JAVA_HOME%\lib

1. JAVA_HOME C:\Program Files\Java\jdk11
2. path %JAVA_HOME%\bin
3. classpath %JAVA_HOME%\lib

내PC -> 속성 -> 정보 -> 고급시스템설정

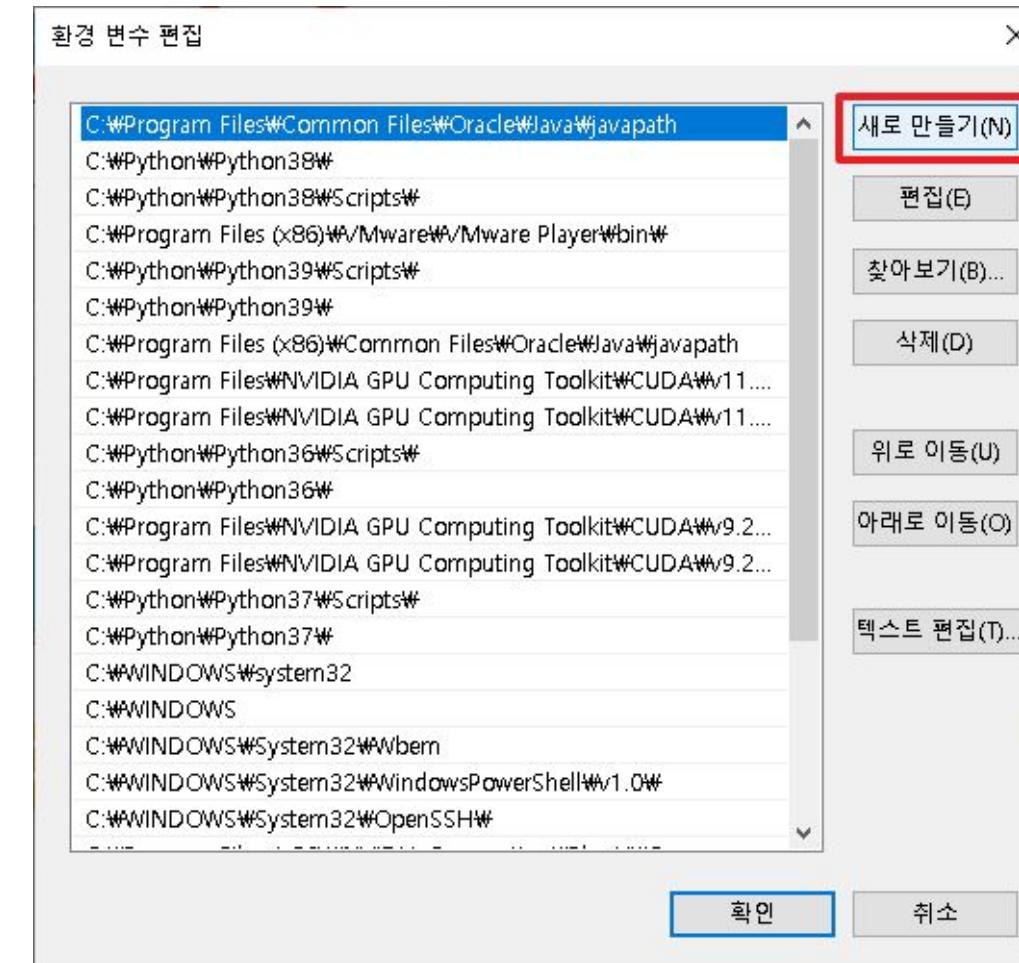
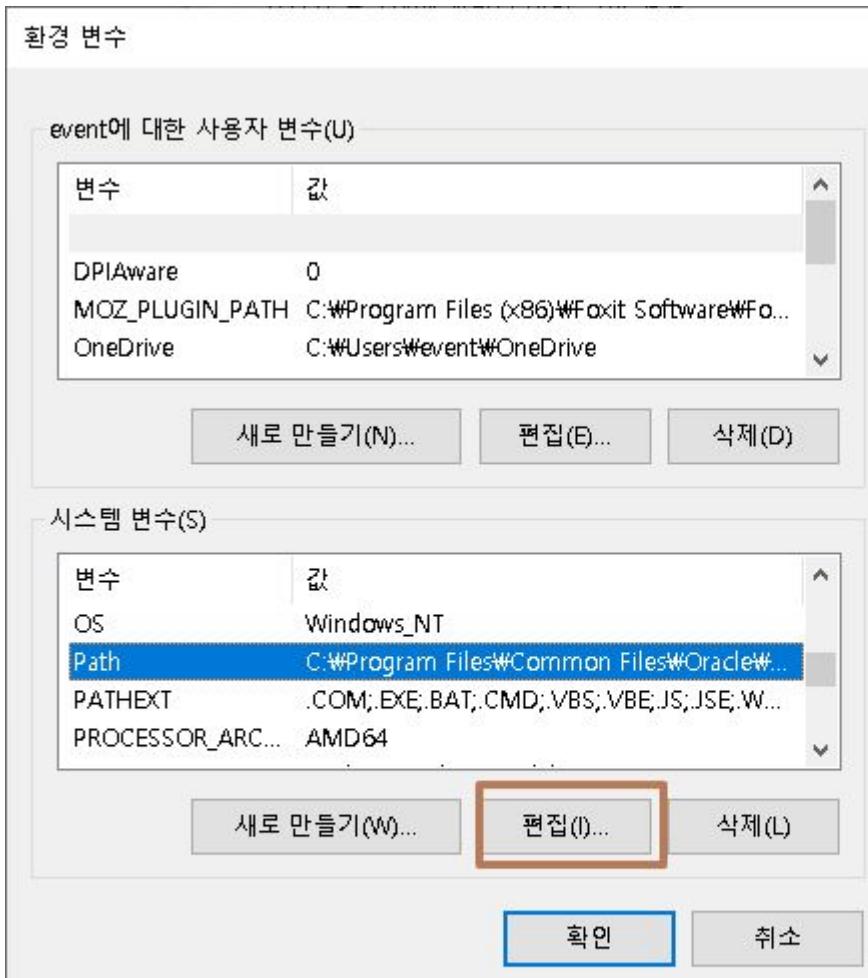




JAVA_HOME 이 없는 경우
-> 새로 만들기

자바 시작하기

환경변수 설정

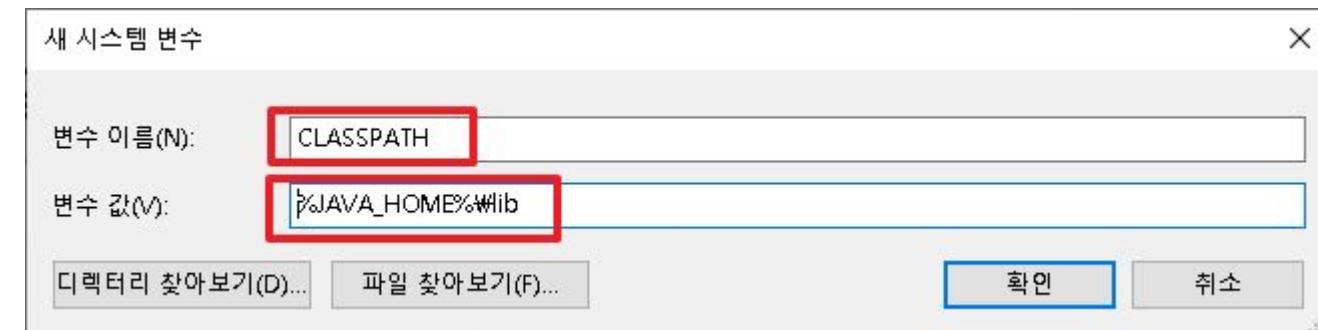


%JAVA_HOME%\bin



CLASSPATH

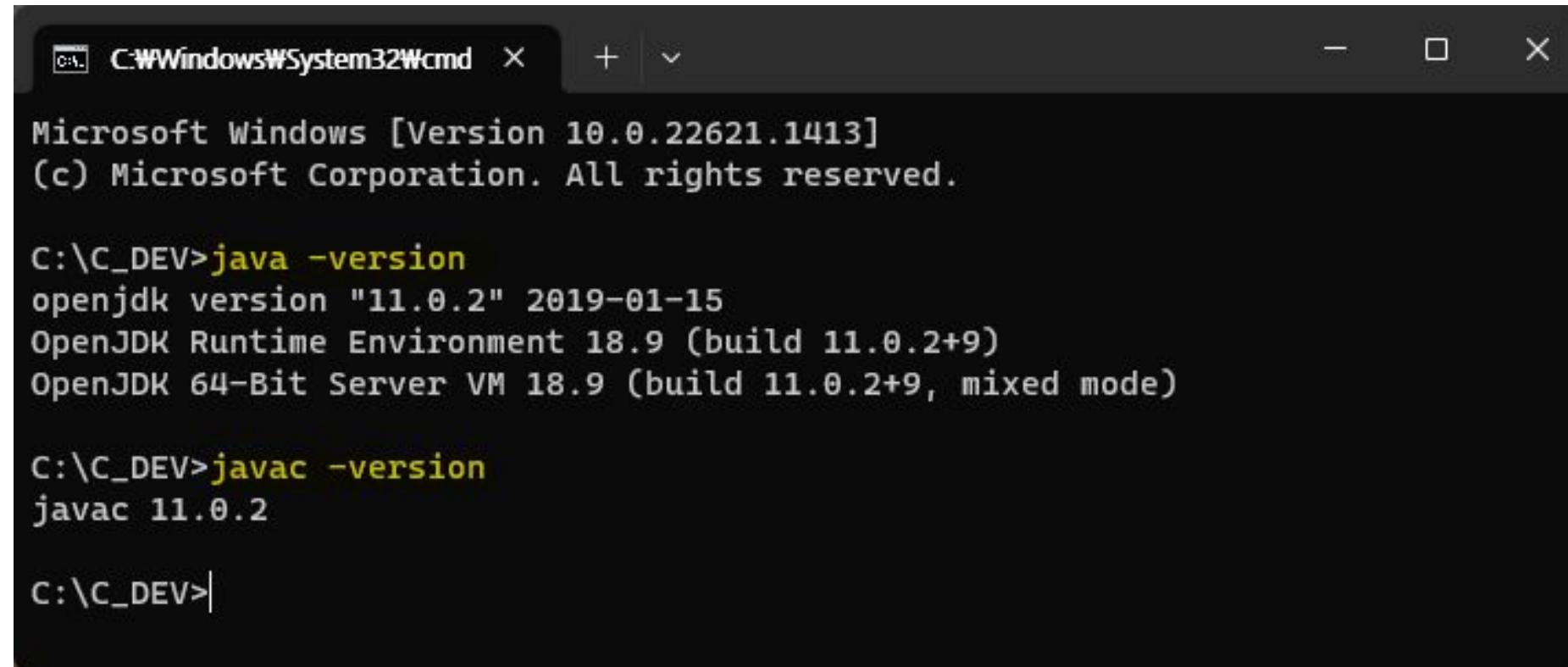
%JAVA_HOME%\lib



확인을 눌러 설치를 완료

명령프롬프트에서 java -version 입력하고 엔터 -> 제대로 설치되었는지 확인

명령프롬프트에서 javac -version 입력하고 엔터 -> 제대로 설치되었는지 확인



The screenshot shows a Windows Command Prompt window titled 'Windows System32 cmd'. The window displays the following text:

```
Microsoft Windows [Version 10.0.22621.1413]
(c) Microsoft Corporation. All rights reserved.

C:\C_DEV>java -version
openjdk version "11.0.2" 2019-01-15
OpenJDK Runtime Environment 18.9 (build 11.0.2+9)
OpenJDK 64-Bit Server VM 18.9 (build 11.0.2+9, mixed mode)

C:\C_DEV>javac -version
javac 11.0.2

C:\C_DEV>
```

이클립스

STS

<https://www.eclipse.org/downloads/>에서 “2022-12-R 선택” 다운로드 후 설치 =>

Eclipse Downloads | The Eclipse

The Eclipse Installer
2022-09 R now includes a JRE for macOS, Windows and Linux.

Get Eclipse IDE 2022-09

Install your favorite desktop IDE packages.

Download x86_64

Download Packages | Need Help?

eclipseinstaller by Oomph

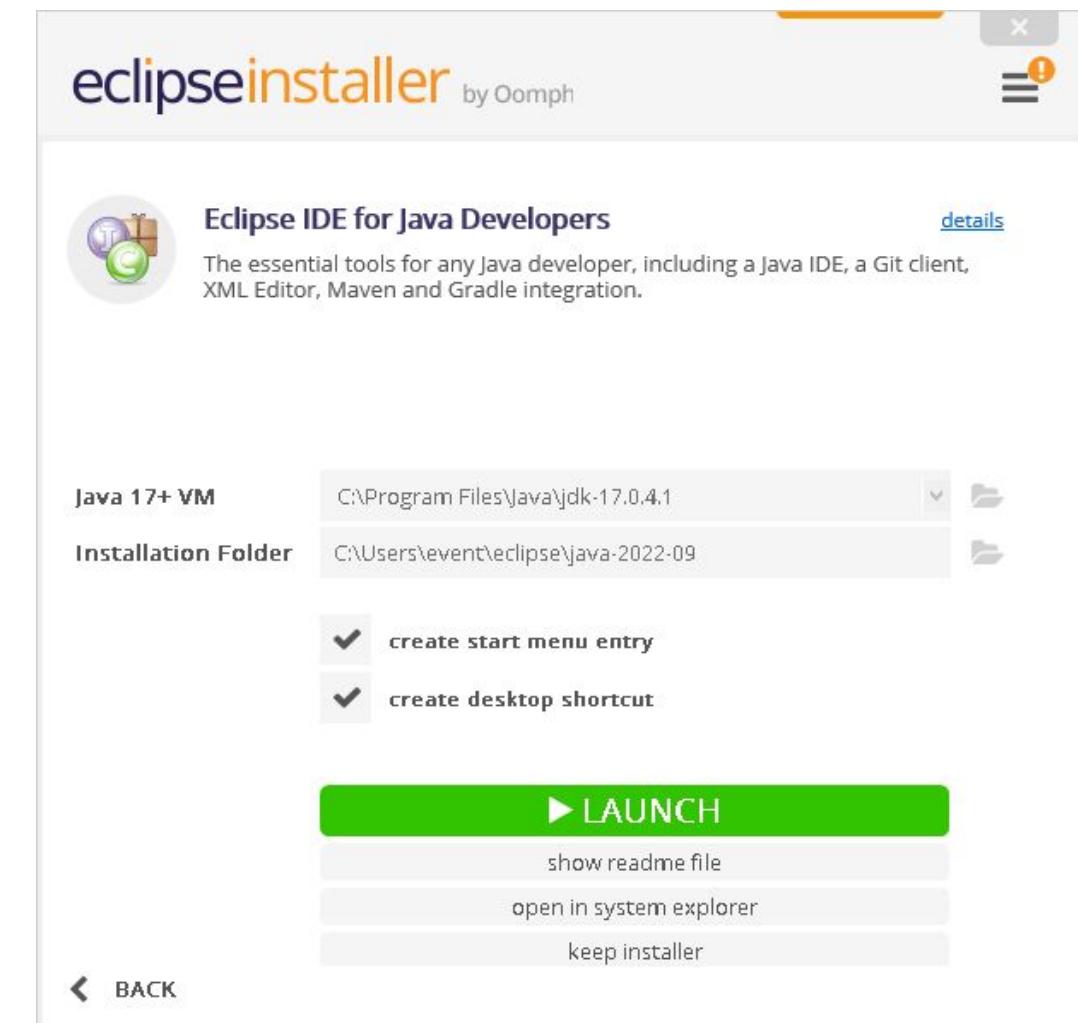
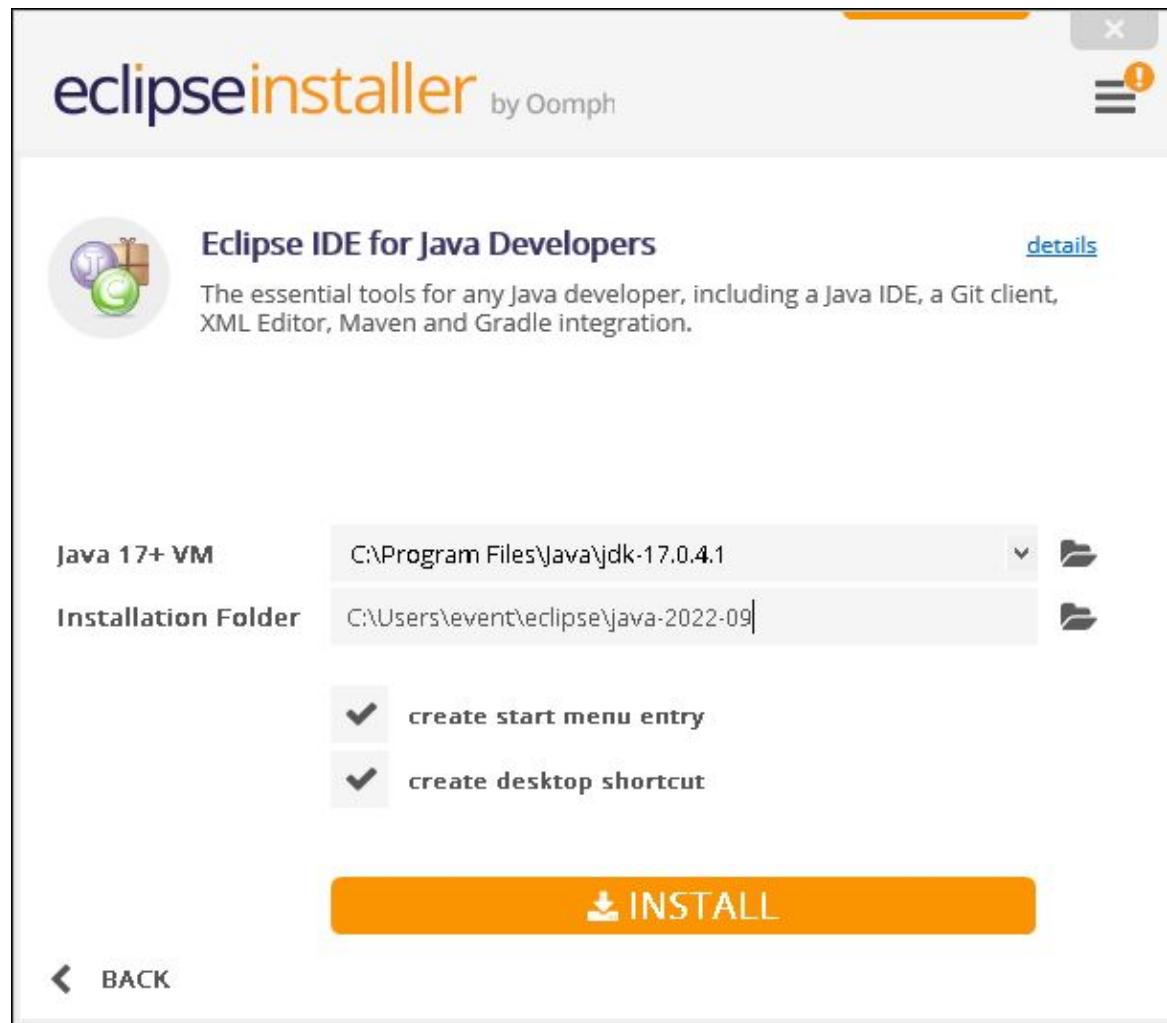
type filter text

 Eclipse IDE for Java Developers
The essential tools for any Java developer, including a Java IDE, a Git client, XML Editor, Maven and Gradle integration

 Eclipse IDE for Enterprise Java and Web Developers
Tools for developers working with Java and Web applications, including a Java IDE, tools for JavaScript, TypeScript, JavaServer Pages and Faces, Yaml,...

 Eclipse IDE for C/C++ Developers
An IDE for C/C++ developers.

 Eclipse IDE for Embedded C/C++ Developers
An IDE for Embedded C/C++ developers. It includes managed cross build plug-ins (Arm and RISC-V) and debug plug-ins (SEGGER J-Link, OpenOCD, pyocd, and...)



자바가 설치된 디렉토리와 이클립스를 설치할 디렉토리를 지정 -> 수정없이 진행

“sts download”

- ❑ Spring Tool Suite 3.9.17
- ❑ Eclipse 4.19

The screenshot shows a news article from the Spring website. The title is "Spring Tools 4.11.0 released". The article is dated June 21, 2021, and has 7 comments. It announces the release of Spring Tools 4.11.0 for Eclipse, Visual Studio Code, and Theia. The article lists major changes to the Eclipse distribution, including updates to Eclipse 2021-06 and support for Java 16. It also mentions early-access builds for Apple Silicon (ARM M1). A reminder section notes that the Eclipse-based distribution requires JDK11 or newer. The right sidebar features a navigation menu with a red box highlighting the "Spring Tools 4" link under the "DEVELOPMENT TOOLS" section.

Spring Tools 4.11.0 released

RELEASERS | MARTIN LIPPERT | JUNE 21, 2021 | 7 COMMENTS

Dear Spring Community,

I am happy to announce the 4.11.0 release of the Spring Tools 4 for Eclipse, Visual Studio Code, and Theia.

major changes to the Spring Tools 4 for Eclipse distribution

- updated to Eclipse 2021-06 release (including support for Java 16) ([new](#) and noteworthy)
- early-access builds for Apple Silicon platform (ARM M1) available

reminder

- the Eclipse-based distribution of the Spring Tools 4 requires a JDK11 (or newer) to run on
- the Eclipse-based distribution ships with an embedded JDK16 runtime, no need to install or configure a specific IDE on anymore

additional highlights

Overview
Spring Boot
Spring Framework
Spring Cloud
Spring Cloud Data Flow
Spring Data
Spring Integration
Spring Batch
Spring Security
View all projects

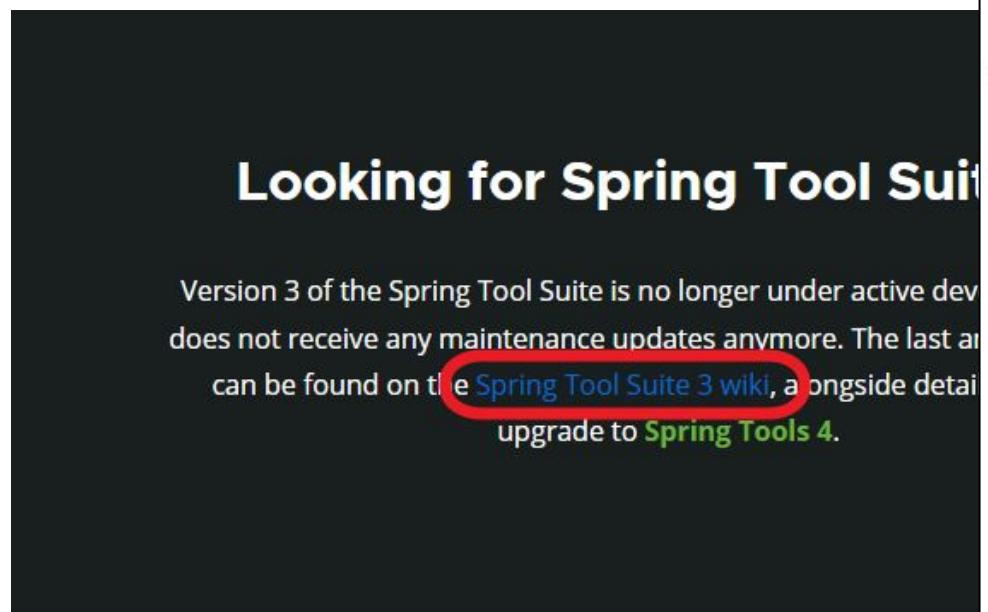
DEVELOPMENT TOOLS

Spring Tools 4

Spring Initializr

“sts download”

- ❑ Spring Tool Suite **3.9.17**
- ❑ Eclipse **4.19**



Previous STS3 Versions

Spring Tool Suite 3.9.17 (New and Noteworthy)

full distribution on Eclipse 4.20

- 3.9.17: https://download.springsource.com/release/STS/3.9.17.RELEASE/dist/e4.20/spring-tool-suite-3.9.17.RELEASE-e4.20.0-win32-x86_64.zip
- https://download.springsource.com/release/STS/3.9.17.RELEASE/dist/e4.20/spring-tool-suite-3.9.17.RELEASE-e4.20.0-macosx-cocoa-x86_64.dmg
 - https://download.springsource.com/release/STS/3.9.17.RELEASE/dist/e4.20/spring-tool-suite-3.9.17.RELEASE-e4.20.0-linux-gtk-x86_64.tar.gz

full distribution on Eclipse 4.19

- https://download.springsource.com/release/STS/3.9.17.RELEASE/dist/e4.19/spring-tool-suite-3.9.17.RELEASE-e4.19.0-win32-x86_64.zip
- https://download.springsource.com/release/STS/3.9.17.RELEASE/dist/e4.19/spring-tool-suite-3.9.17.RELEASE-e4.19.0-macosx-cocoa-x86_64.dmg
- https://download.springsource.com/release/STS/3.9.17.RELEASE/dist/e4.19/spring-tool-suite-3.9.17.RELEASE-e4.19.0-linux-gtk-x86_64.tar.gz

full distribution on Eclipse 4.18

- https://download.springsource.com/release/STS/3.9.17.RELEASE/dist/e4.18/spring-tool-suite-3.9.17.RELEASE-e4.18.0-win32-x86_64.zip
- https://download.springsource.com/release/STS/3.9.17.RELEASE/dist/e4.18/spring-tool-suite-3.9.17.RELEASE-e4.18.0-macosx-cocoa-x86_64.dmg
- https://download.springsource.com/release/STS/3.9.17.RELEASE/dist/e4.18/spring-tool-suite-3.9.17.RELEASE-e4.18.0-linux-gtk-x86_64.tar.gz

■ 설치

C:\Java\sts-3.9.17.RELEASE\STS.exe 로 실행

설치할 항목

1. open jdk 11.02 => C:\Java\jdk-11.0.2
2. JDK8u202 => C:\Java\jdk1.8.0_202
+----- JRE => C:\Java\jre1.8.0_202

3. 환경변수설정

JAVA_HOME --- C:\Java\jdk-11.0.2

PATH %JAVA_PATH%\bin

classpath %JAVA_PATH%\lib

4 STS 설치 : sts download 검색

Spring Tool Suite 3.9.17 + Eclipse 4.19

=> C:\Java\sts-3.9.17.RELEASE

실행 C:\Java\sts-3.9.17.RELEASE\STS.exe

====> 설치해 보세요.

workspace

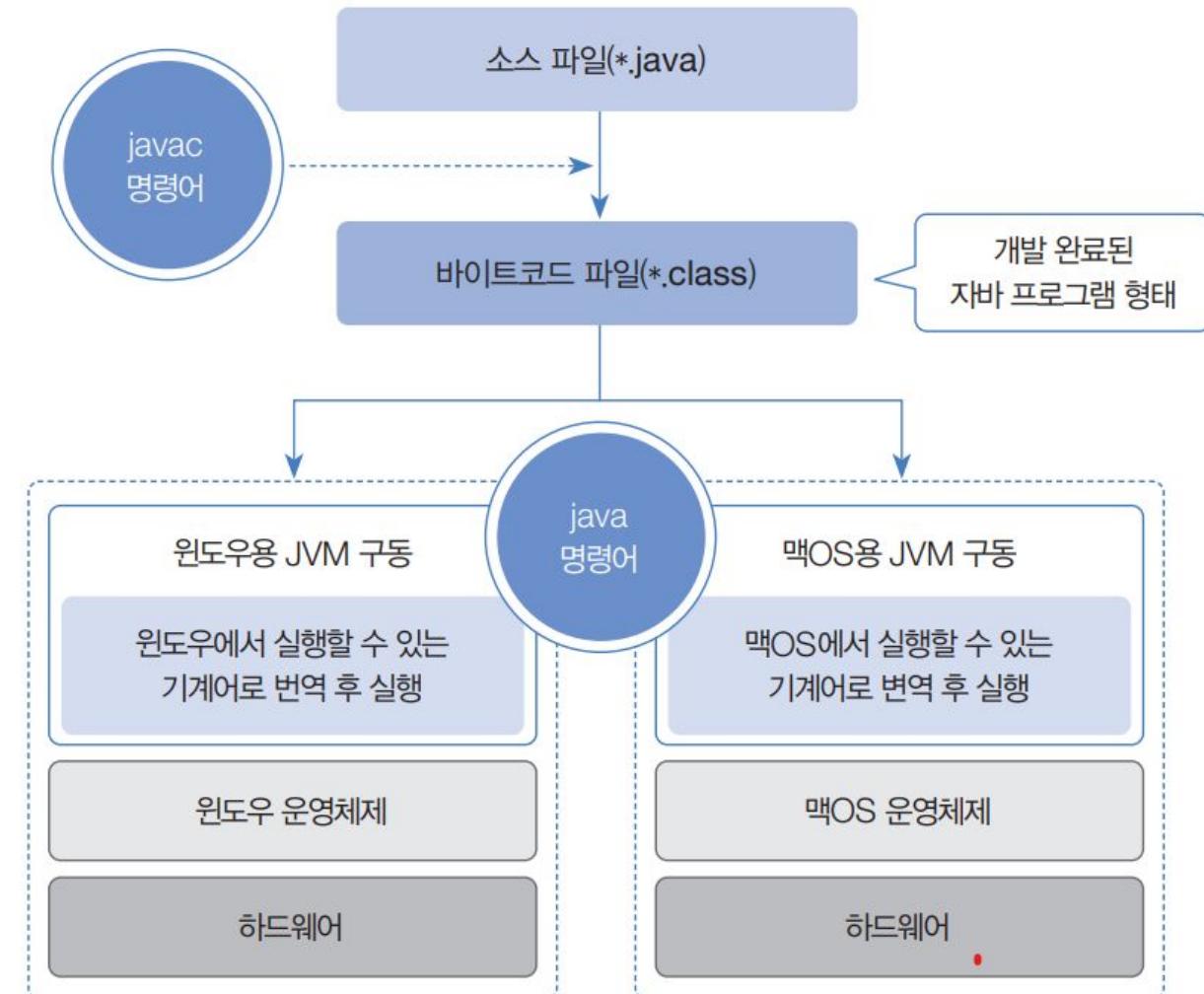
C:\dev\java

바이트코드 파일

- 소스 파일(.java)을 작성한 후 바이트코드 파일(.class)로 컴파일

자바 가상 머신

- java 명령어로 자바 가상 머신(JVM)을 구동시켜 바이트코드 파일(.class)을 기계어로 번역 및 실행
- 자바 가상 머신은 운영체제별로 다르게 설치됨



소스 파일 생성

- temp 디렉토리를 다음 구조로 생성



- Hello.java 소스 파일을 생성하고
다음과 같은 형식으로 코드가 작성됨

»»> Hello.java

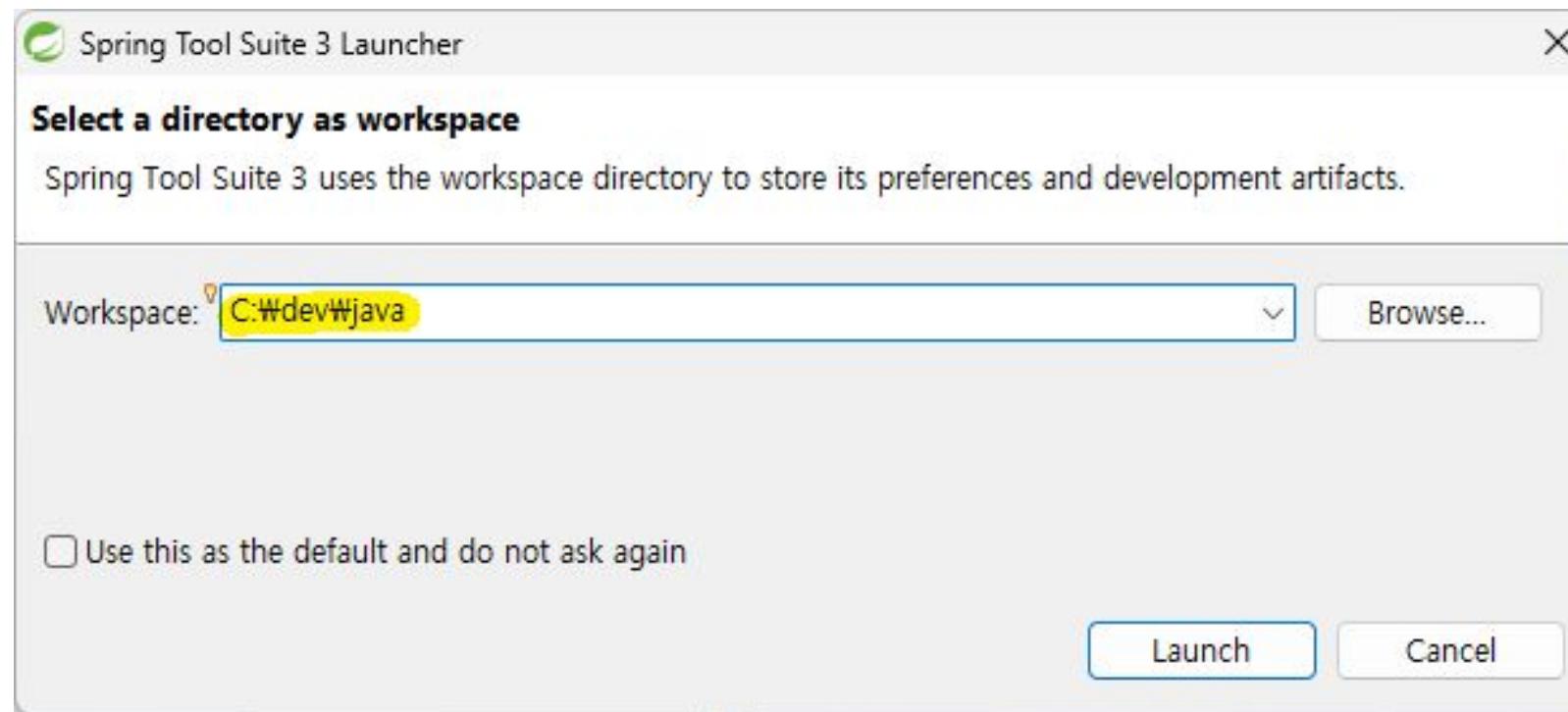
```

1  package ch01.sec06;                                //바이트코드 파일이 위치할 패키지 선언
2
3  public class Hello {                               //Hello 클래스 선언
4      public static void main(String[] args) { //main() 메소드 선언
5          System.out.println("Hello, Java");    //콘솔에 출력하는 코드
6      }
7  }

```

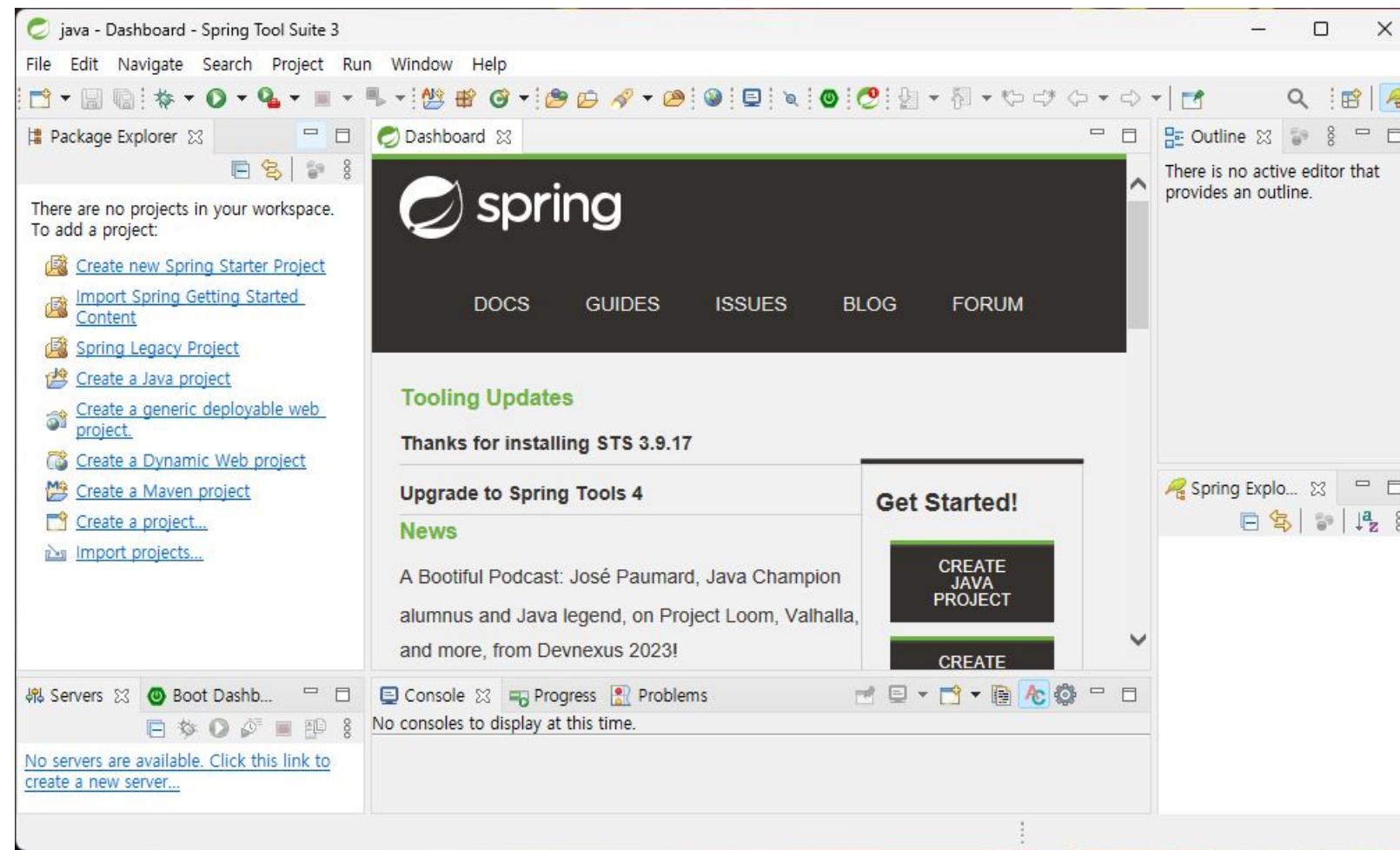
이클립스 최초 실행

- 앞으로 작성할 프로젝트의 소스를 저장할 폴더 위치(workspace)를 지정

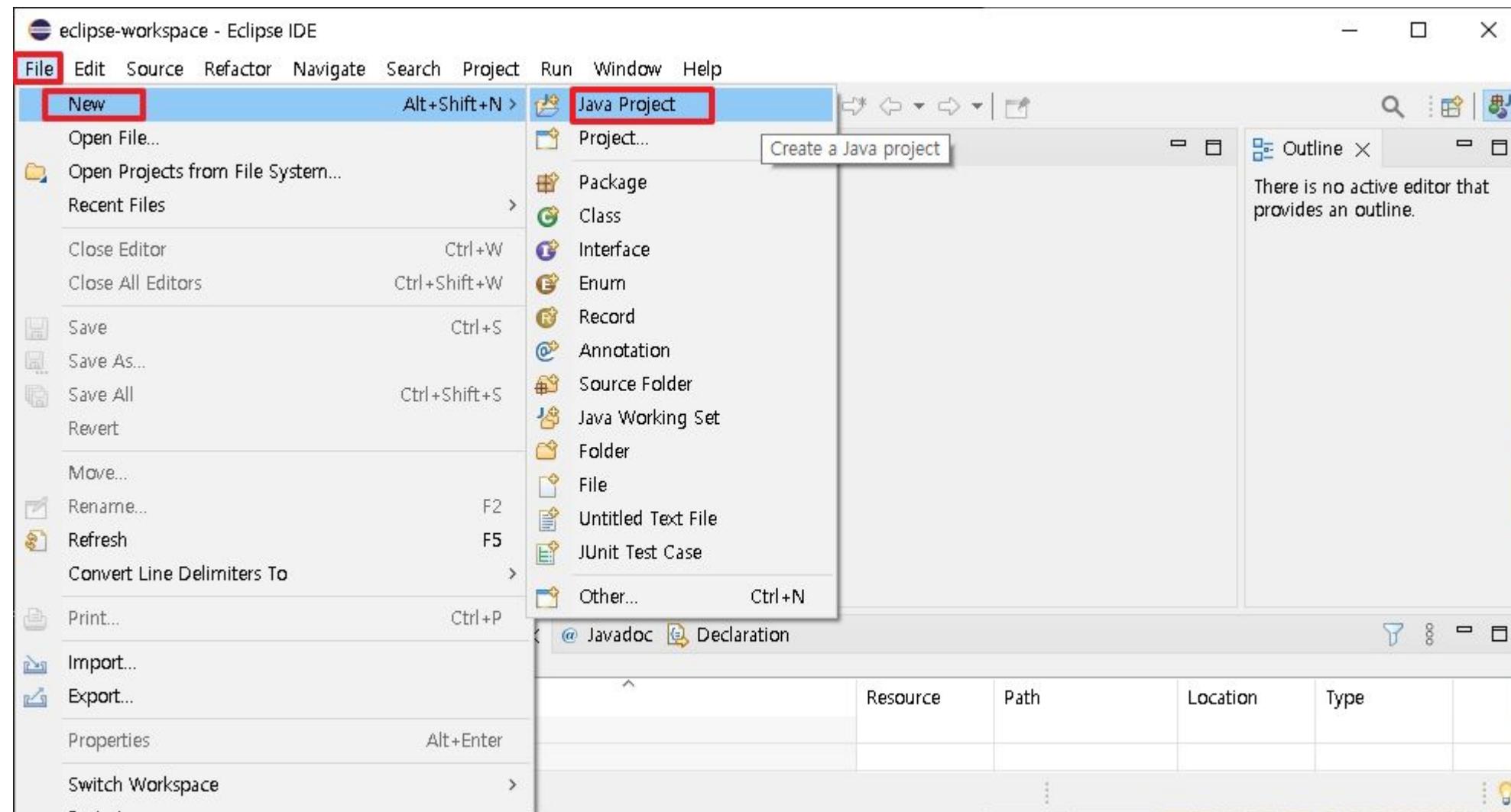


- workspace 를 **C:\dev\java** 로 지정 (option)

Welcome 창 닫기

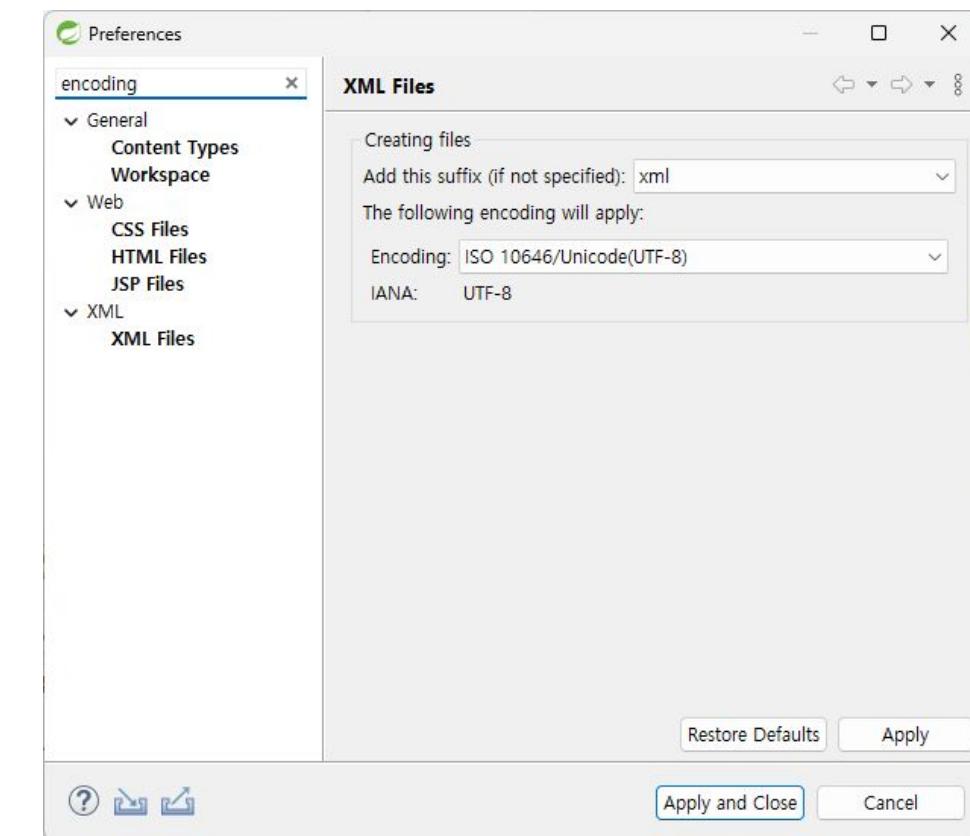
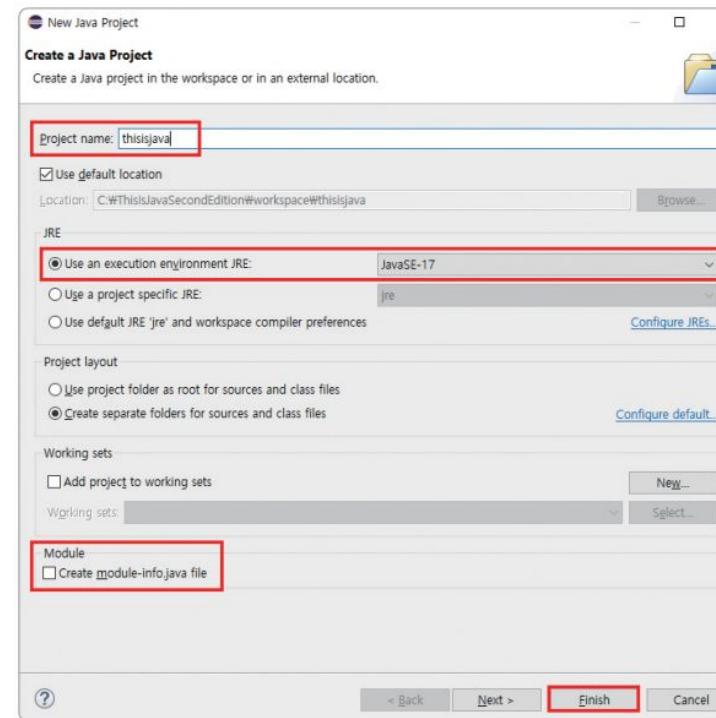


File > New > Java Project 실행



이클립스 프로젝트 생성

- Java Project 먼저 생성하고 JRE는 ‘JavaSE-11’ 선택
- [Window]-[Preferences]에서 [General]-[Workspace]-<Text file encoding>을 ‘UTF-8’ 선택
- [Window]-[Preferences]에서 “encoding” 검색 수정점



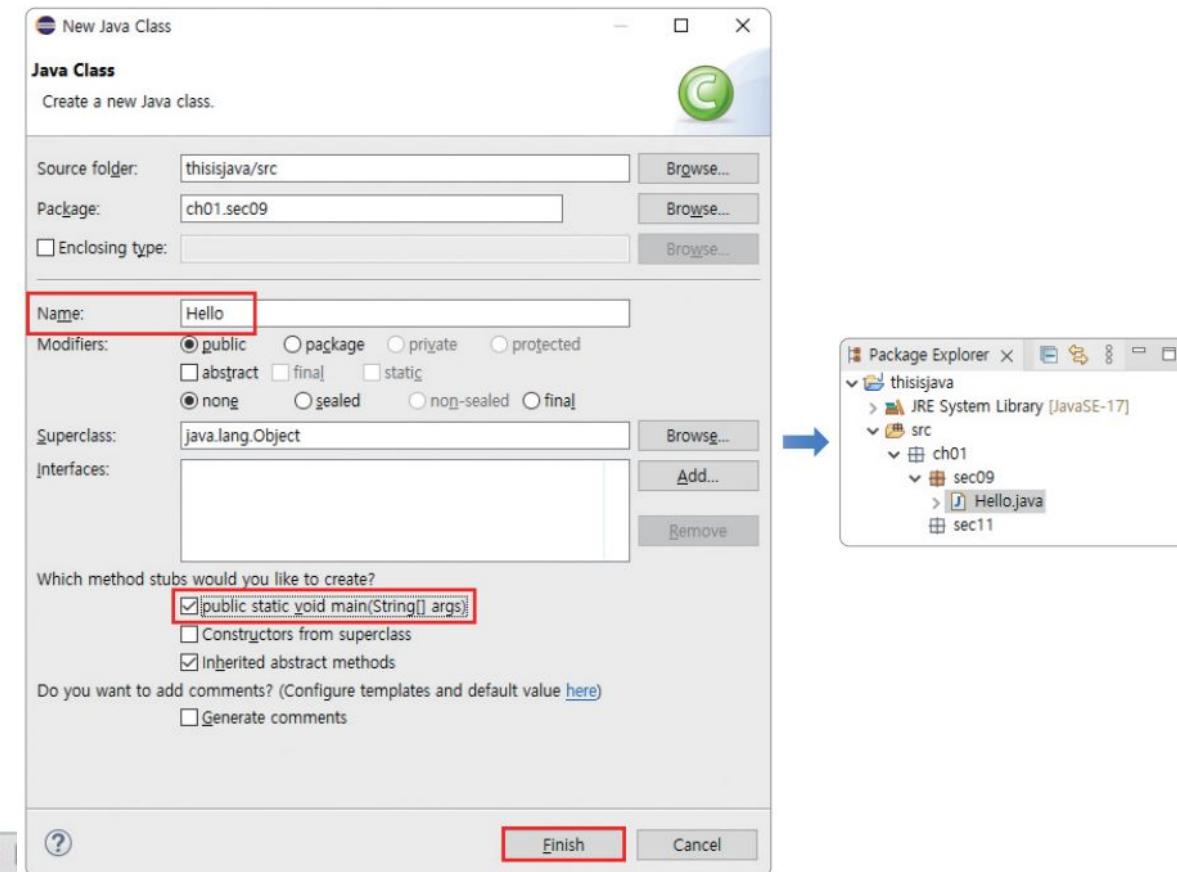
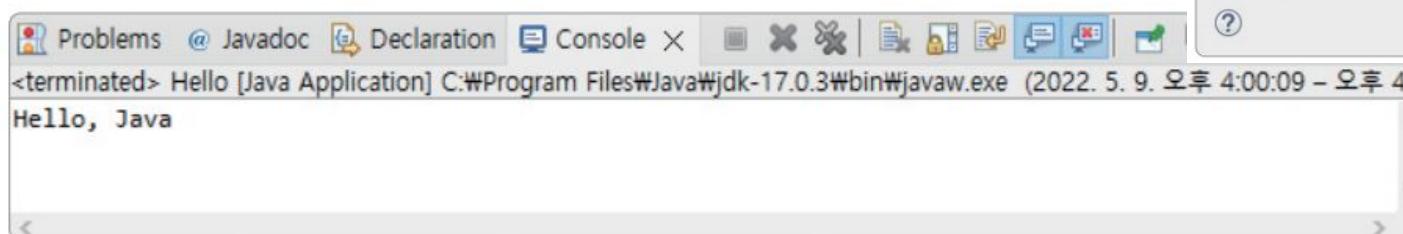
이클립스 소스 작성 및 실행

- 소스 파일, 바이트코드 관리 위한 패키지 생성
- src에 Class를 추가해 Hello.java 소스 파일 작성



```
1 package ch01.sec09;
2
3 public class Hello {
4     public static void main(String[] args) {
5         System.out.println("Hello, Java");
6     }
7 }
```

- 소스 파일을 실행하고 Console 뷰에서 결과 확인



이클립스 소스 작성 및 실행

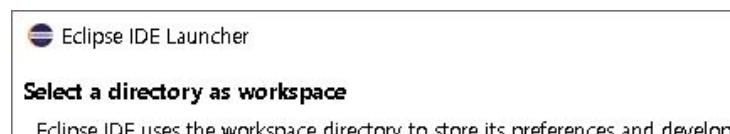
- 패키지 : kr.ac.wsq.itedu.ch01.sec01

사이트 : itedu.wsu.ac.kr

하위 디렉토리 : ./ch01/sec01

- 프로젝트명 : Hello

- workspace : C:\dev\java



- [File]-[New]-[Java Project]
- []Create module-info.java file

체크 해제

- 텍스트입력

```

1 package kr.ac.wsq.itedu.ch01.sec01;
2
3 public class Hello {
4     public static void main(String[] args) {
5         // TODO Auto-generated method stub
6         System.out.println("Hello World");
7     }
8 }
9

```

- run

```

<terminated> Hello (1) [Java Application] C:\Program Files\Java\jdk-17.0.4.1\bin\javaw.exe
Hello World

```

```
C:\dev\java
    \Hello
        \bin
        \src
            \kr
                \ac
                    \wsu
                        \itedu
                            \cd01
                                \sec01
```

코딩용 폰트 - 글자의 가로폭이 일정할 것

- 123AaliLIOo한글 - 나눔고딕
12345678901234 - (가로폭이 다름)
- 01AaIiLl0o한글 - 나눔고딕코딩
12345678901234 - (가로폭 일정)
- 01AaIiLl0o한글 - Consolas
12345678901234 - (가로폭 일정)

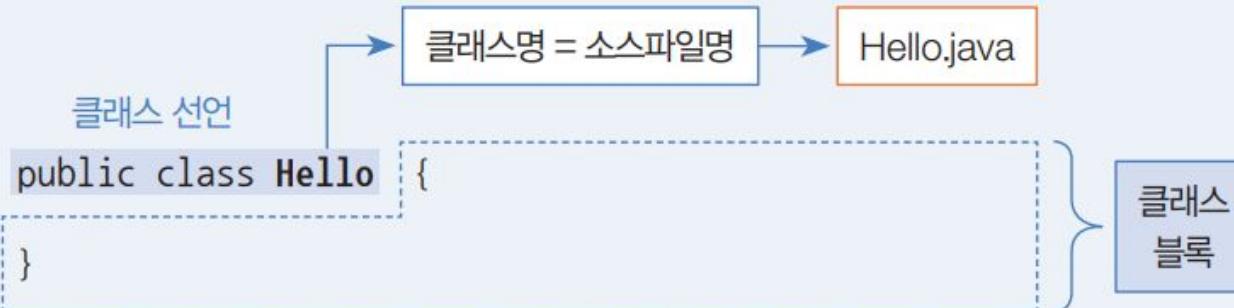
추천 폰트

- **Consolas** - 윈도우 내장 기본 폰트
- **D2coding** - 네이버, 한글디자인 우수
- Source Code Pro
- Ubuntu Monospace

기타 개인의 취향대로

- 1. 가독성 2. 고정폭
- 1|| - 숫자1, L의 소문자, I의 대문자 구별

클래스 선언



메소드



주석 기호와 설명

- 프로그램 실행과 상관없이 코드에 설명 붙임

구분	주석 기호	설명
행 주석	// ...	//부터 행 끝까지 주석으로 처리한다.
범위 주석	/* ... */	/*와 */ 사이에 있는 내용은 모두 주석으로 처리한다.
도큐먼트 주석	/** ... */	/**와 */ 사이에 있는 내용은 모두 주석으로 처리한다. javadoc 명령어로 API 도큐먼트를 생성하는 데 사용한다.

- 주석 기호는 문자열(“ ”) 내부에 작성하면 안 됨

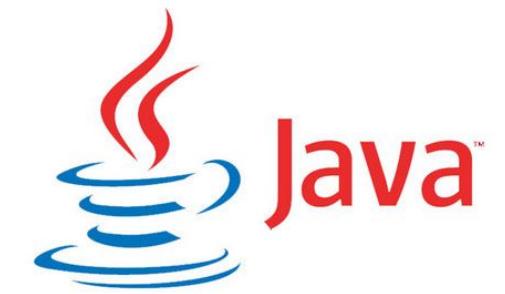
```
System.out.println("Hello, /*주석이 될 수 없음*/ welcome to the java world!");
```

실행문

- 실행문은 변수 선언, 변수값 저장, 메소드 호출에 해당하는 코드
- 실행문 끝에는 반드시 세미콜론(:)을 붙여 실행문의 끝 표시

```
int x;           //변수 x 선언
x = 1;          //변수 x에 1 값을 저장
int y = 2;       //변수 y를 선언하고 2 값을 저장
int result = x + y; //변수 result를 선언하고 변수 x와 y를 더한 값을 저장
System.out.println(result); //콘솔에 변수의 값을 출력하는 println() 메소드 호출
```

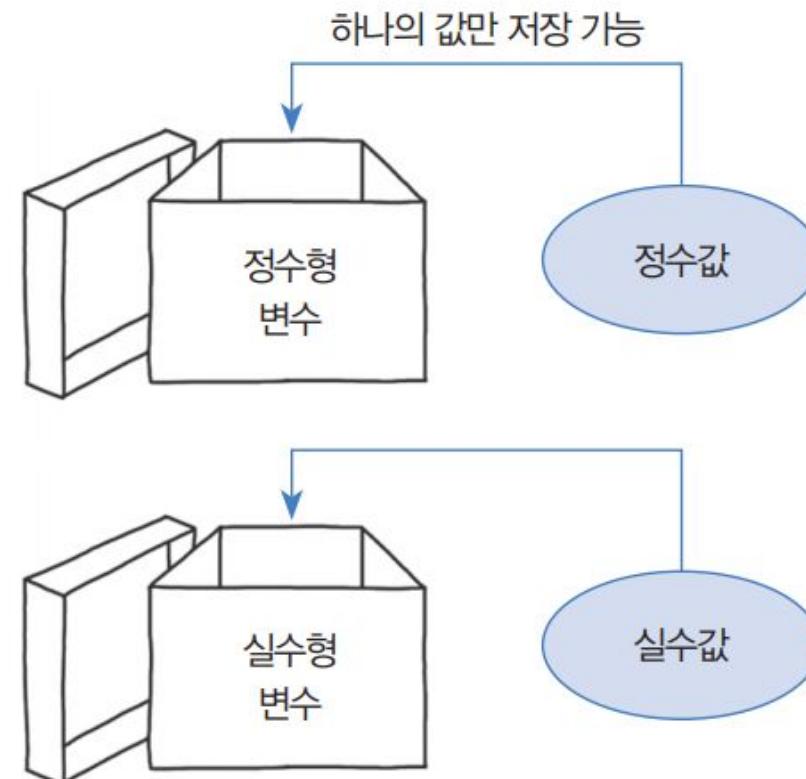




Chapter 02 변수와 타입

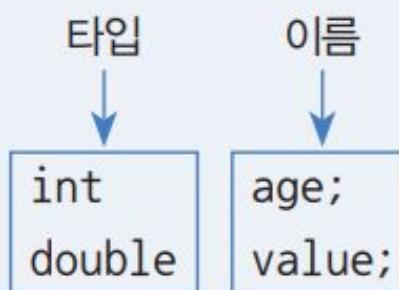
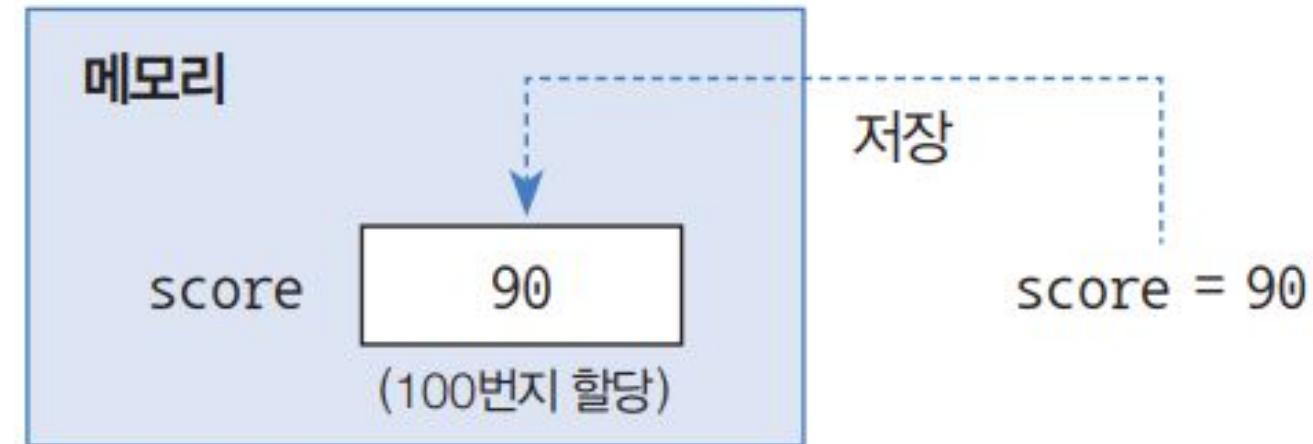
변수

- 변수(variable)란 하나의 값을 저장할 수 있는 메모리 번지에 붙여진 이름
- 자바의 변수는 다양한 타입(정수형, 실수형 등)의 값을 저장할 수 없다.



변수 선언

- 변수 변수를 사용하려면 변수 선언이 필요. 변수 선언은 어떤 타입의 데이터를 저장할 것인지 그리고 변수 이름이 무엇인지를 결정하는 것
- 변수에 최초로 값이 대입될 때 메모리에 할당 되고, 해당 메모리에 값이 저장



//정수(int) 값을 저장할 수 있는 age 변수 선언
 //실수(double) 값을 저장할 수 있는 value 변수 선언

byte, short, char, int, long 타입

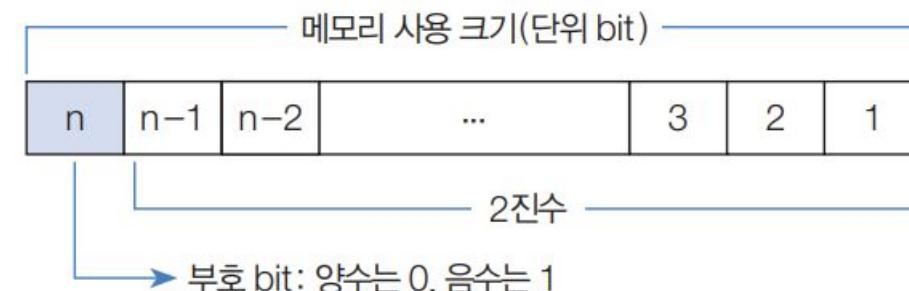
- 변수는 선언될 때의 타입에 따라 저장할 수 있는 값의 종류와 허용 범위가 달라짐
- 정수 타입은 5개로 메모리 할당 크기와 저장되는 값의 범위가 다름

타입	메모리 크기		저장되는 값의 허용 범위	
byte	1byte*	8bit	$-2^7 \sim (2^7-1)$	-128 ~ 127
short	2byte	16bit	$-2^{15} \sim (2^{15}-1)$	-32,768 ~ 32,767
char	2byte	16bit	$0 \sim (2^{16}-1)$	0 ~ 65535 (유니코드)
int	4byte	32bit	$-2^{31} \sim (2^{31}-1)$	-2,147,483,648 ~ 2,147,483,647
long	8byte	64bit	$-2^{63} \sim (2^{63}-1)$	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807

* 1byte = 8bit, bit는 0과 1이 저장되는 단위

- 메모리 크기를 n이라고 했을 때 정수 타입은 동일한 구조의 2진수로 저장

- `int i = 10;`
- `int oct = 010;`
- `int hex = 0x10;`
- `long bigNum = 7123456789012L;`



문자 리터럴과 char 타입

- 문자 리터럴: 하나의 문자를 작은 따옴표(')로 감싼 것
- 문자 리터럴을 유니코드로 저장할 수 있도록 char 타입 제공

```
char var1 = 'A';      // 'A' 문자와 매핑되는 숫자: 65로 대입  
char var3 = '가';      // '가' 문자와 매핑되는 숫자: 44032로 대입
```

- char 타입도 정수 타입에 속함

```
char c = 65;          // 10진수 65와 매핑되는 문자: 'A'  
char c = 0x0041;      // 16진수 0x0041과 매핑되는 문자: 'A'
```

float과 double 타입

- 실수 타입에는 float과 double이 있음

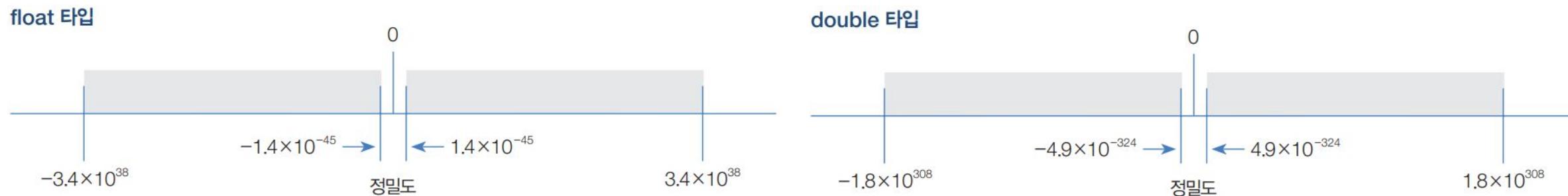
타입	메모리 크기		저장되는 값의 허용 범위(양수 기준)	유효 소수 이하 자리
float	4 byte	32 bit	$1.4 \times 10^{-45} \sim 3.4 \times 10^{38}$	7자리
double	8 byte	64 bit	$4.9 \times 10^{-324} \sim 1.8 \times 10^{308}$	15자리

- double & float 선언

```
double dnum = 10.1d;
```

```
float fnum = 10.1f;
```

- double 타입이 float 타입보다 큰 실수를 저장할 수 있고 정밀도도 높음



boolean 타입 변수에 대입되는 논리 타입

- 참과 거짓을 의미하는 true와 false로 구성되며 boolean 타입 변수에 대입할 수 있음

```
boolean stop = true;  
boolean stop = false;
```

- 주로 두 가지 상태값을 저장하는 경우에 사용. 조건문과 제어문의 실행 흐름을 변경하는 데 사용

int x = 10;	연산식	
boolean result =	(x == 20);	//변수 x의 값이 20인가?
boolean result =	(x != 20);	//변수 x의 값이 20이 아닌가?
boolean result =	(x > 20);	//변수 x의 값이 20보다 큰가?
boolean result =	(0 < x && x < 20);	//변수 x의 값이 0보다 크고, 20보다 적은가?
boolean result =	(x < 0 x > 200);	//변수 x의 값이 0보다 적거나 200보다 큰가?

1. 클래스 이름의 첫 글자는 항상 대문자로 한다.
 - a. 변수와 메서드 이름의 첫 글자는 항상 소문자로 한다.
2. 여러 단어 이름은 단어의 첫 글자를 대문자로 한다. (Camel Type)
 - a. lastIndexOf, StringBuffer
3. 상수의 이름은 대문자로 한다. 단어는 ‘_’로 구분한다.
 - a. PI, MAX_NUMBER

문자열과 String 타입

- 문자열: 큰따옴표("")로 감싼 문자들
- 문자열을 변수에 저장하려면 String 타입을 사용

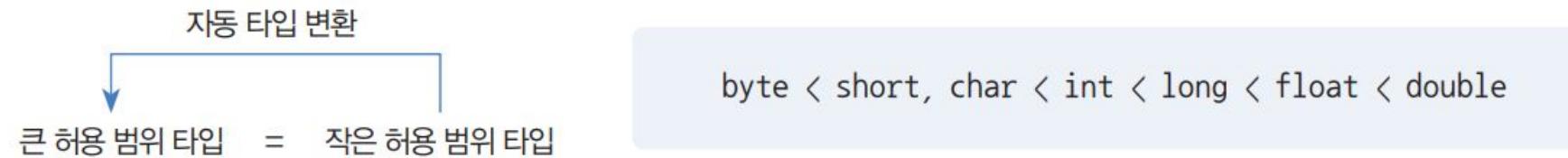
```
String var1 = "A";  
String var2 = "홍길동";
```

- 이스케이프 문자: 문자열 내부에 역슬래쉬(\)가 붙은 문자

이스케이프 문자	
\"	" 문자 포함
'	' 문자 포함
\\	\ 문자 포함
\u16진수	16진수 유니코드에 해당하는 문자 포함
\t	출력 시 탭만큼 띄움
\n	출력 시 줄바꿈(라인피드)
\r	출력 시 캐리지 리턴

자동 타입 변환

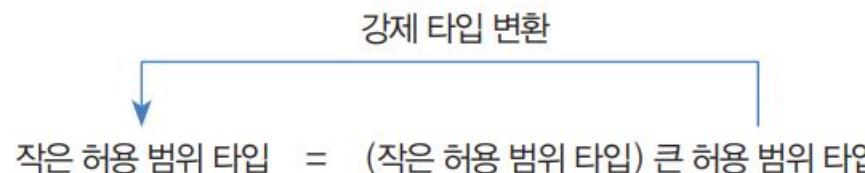
- 데이터 타입을 다른 타입으로 변환하는 것
- 값의 허용 범위가 작은 타입이 허용 범위가 큰 타입으로 대입될 때 발생



- 정수 타입이 실수 타입으로 대입되면 무조건 자동 타입 변환이 됨
- 예외: char 타입보다 허용 범위가 작은 byte 타입은 char 타입으로 자동 변환될 수 없음

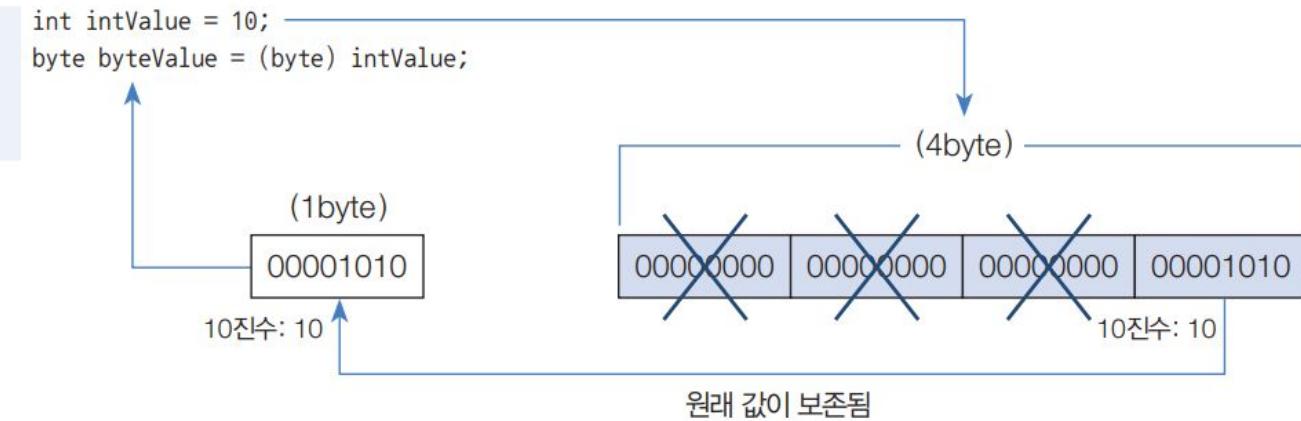
캐스팅 연산자로 강제 타입 변환하기

- 큰 허용 범위 타입을 작은 허용 범위 타입으로 쪼개어서 저장하는 것
- 캐스팅 연산자로 괄호()를 사용하며, 괄호 안에 들어가는 타입은 쪼개는 단위



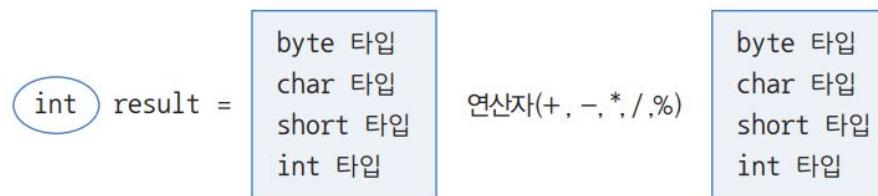
- 예: int → byte 강제 타입 변환

```
int intValue = 10;
byte byteValue = (byte) intValue; //강제 타입 변환
```



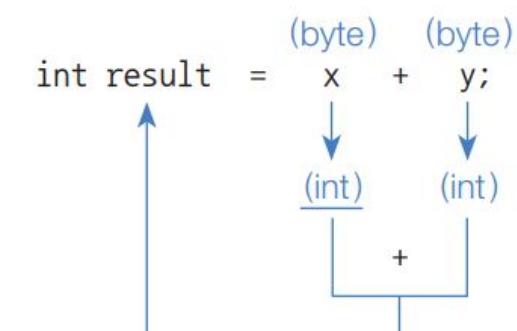
연산식에서 int 타입의 자동 변환

- 정수 타입 변수가 산술 연산식에서 피연산자로 사용되면 int 타입보다 작은 byte, short 타입 변수는 int 타입으로 자동 변환되어 연산 수행



byte 타입 변수가 피연산자로 사용된 경우	int 타입 변수가 피연산자로 사용된 경우
<pre>byte x = 10; byte y = 20; byte result = x + y; //컴파일 에러 int result = x + y;</pre>	<pre>int x = 10; int y = 20; int result = x + y;</pre>

- byte 변수가 피연산자로 사용되면 변수값은 int 값으로 연산되며, 결과값 역시 byte 변수가 아닌 int 변수에 저장해야 함



String 타입 변환하기

변환 타입	사용 예
String → byte	<code>String str = "10"; byte value = Byte.parseByte(str);</code>
String → short	<code>String str = "200"; short value = Short.parseShort(str);</code>
String → int	<code>String str = "300000"; int value = Integer.parseInt(str);</code>
String → long	<code>String str = "40000000000"; long value = Long.parseLong(str);</code>
String → float	<code>String str = "12.345"; float value = Float.parseFloat(str);</code>
String → double	<code>String str = "12.345"; double value = Double.parseDouble(str);</code>
String → boolean	<code>String str = "true"; boolean value = Boolean.parseBoolean(str);</code>

- 기본 타입의 값을 문자열로 변경할 때는 `String.valueOf()` 메소드 이용

변수 범위를 나타내는 중괄호 {} 블록

- 조건문과 반복문의 중괄호 {} 블록 내에 선언된 변수는 해당 중괄호 {} 블록 내에서만 사용 가능

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

```
    int var1; —————— 메소드 블록에서 선언
```

```
    if(...) {  
        int var2; —————— if 블록에서 선언  
        //var1과 var2 사용 가능  
    }
```

if
블록

```
    for(...) {  
        int var3; —————— for 블록에서 선언  
        //var1과 var3 사용 가능  
        //var2는 사용 못함  
    }
```

for
블록

메소드
블록

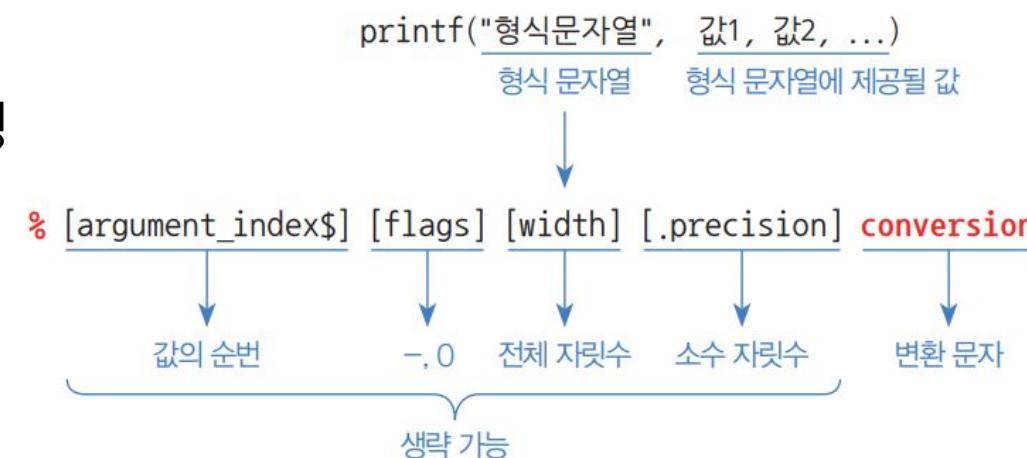
```
    //var1 사용 가능  
    //var2와 var3는 사용 못함  
}
```

println() 메소드로 변수값 출력하기

- 모니터에 값을 출력하기 위해 System.out.println() 이용
- 출력 방법에 따라 println() 이외에도 다음과 같이 print(), printf()를 사용할 수 있음

메소드	의미
println(내용);	괄호 안의 내용을 출력하고 행을 바꿔라.
print(내용);	괄호 안의 내용을 출력하고 행은 바꾸지 말아라.
printf("형식문자열", 값1, 값2, ...);	형식 문자열에 맞추어 뒤의 값을 출력해라.

- printf()의 형식 문자열에서는 %와 conversation(변환 문자)를 필수로 작성하고 나머지는 생략 가능



기타

Conversion Type Characters ::

Formatting String	Output ::
 System.out.printf("%d", 10);	10
System.out.printf("%f", 10.1);	10.100000
System.out.printf("%c", 'a');	a
System.out.printf("%C", 'a');	A
System.out.printf("%s", "hello");	hello
System.out.printf("%S", "hello");	HELLO
System.out.printf("%b", 5 < 4);	false
System.out.printf("%B", 5 < 4);	FALSE
System.out.printf("%b", null);	false
System.out.printf("%b", "cow");	

- %o - 8진수 정수 형식
- %x 또는 %X - 16진수 정수
- %e 또는 %E - 지수 표현식

Output:

Java int printf chart
(number: 123457890)

PATTERN	RESULT
%d	123457890
%,d	123,457,890
%,15d	123,457,890
%+,15d	+123,457,890
%-+,15d	+123,457,890
%0,15d	0000123,457,890

Java double printf chart
(number: 12345.12345)

PATTERN	RESULT
%f	12345.123450
%15f	12345.123450
%-15f	12345.123450
%-15.3f	12345.123
%015.3f	00000012345.123

```
System.out.printf("%f",12345.12345);
System.out.printf("%15f",12345.12345);
```

Scanner 타입 변수 활용하기

- Scanner 타입 변수를 선언하고 대입 연산자 =를 사용해서 new 연산자로 생성한 Scanner 객체를 변수에 대입

생성된 Scanner를 변수에 대입

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

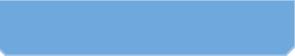
scanner 변수 선언 Scanner 객체 생성

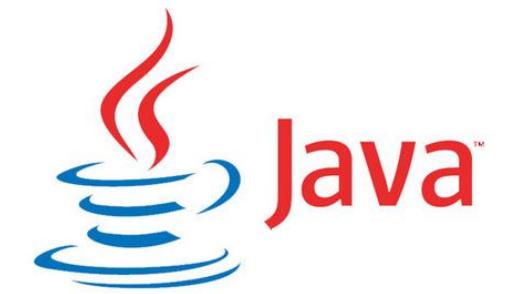
- scanner.nextLine()을 실행하면 키보드로 입력된 내용을 문자열로 읽고 좌측 String 변수에 저장

읽은 문자열을 String 변수에 저장

```
String inputData = scanner.nextLine();
```

String 변수 선언 [Enter] 키를 누르면 입력된 문자열을 읽음





Chapter 03 연산자

부호 연산자

- 부호 연산자는 변수의 부호를 유지하거나 변경

연산식	설명	
+	피연산자	피연산자의 부호 유지
-	피연산자	피연산자의 부호 변경

증감 연산자

- 증감 연산자는 변수의 값을 1 증가시키거나 1 감소시킴

연산식	설명	
++	피연산자	피연산자의 값을 1 증가시킴
-	피연산자	피연산자의 값을 1 감소시킴
피연산자	++	다른 연산을 수행한 후에 피연산자의 값을 1 증가시킴
피연산자	-	다른 연산을 수행한 후에 피연산자의 값을 1 감소시킴

산술 연산자

- 더하기(+), 빼기(-), 곱하기(*), 나누기(/), 나머지(%)로 총 5개

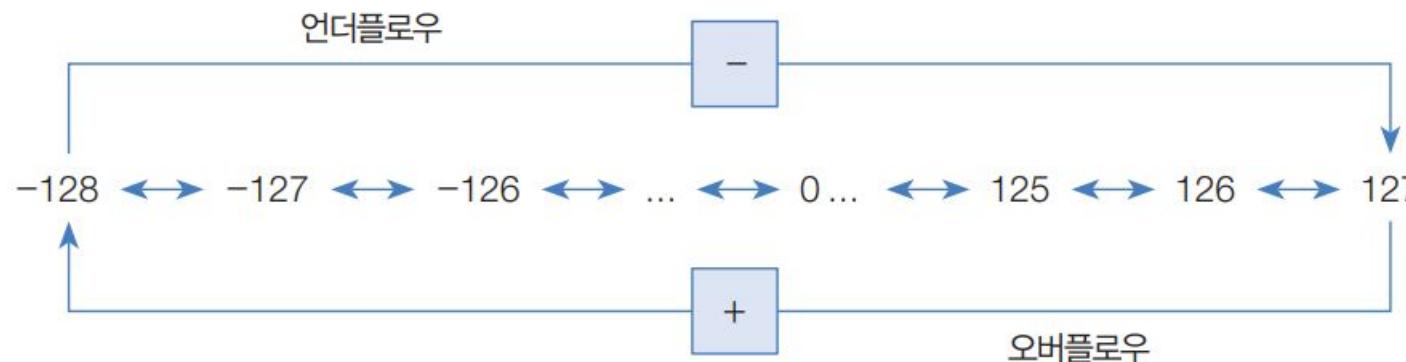
연산식		설명	
피연산자	+	피연산자	덧셈 연산
피연산자	-	피연산자	뺄셈 연산
피연산자	*	피연산자	곱셈 연산
피연산자	/	피연산자	나눗셈 연산
피연산자	%	피연산자	나눗셈의 나머지를 산출하는 연산

오버플로우

- 타입이 허용하는 최대값을 벗어나는 것

언더플로우

- 타입이 허용하는 최소값을 벗어나는 것



정수 연산

- 산술 연산을 정확하게 계산하려면 실수 타입을 사용하지 않는 것이 좋음

» AccuracyExample1.java

```
1 package ch03.sec04;
2
3 public class AccuracyExample1 {
4     public static void main(String[] args) {
5         int apple = 1;
6         double pieceUnit = 0.1;
7         int number = 7;
8
9         double result = apple - number*pieceUnit;
10        System.out.println("사과 1개에서 남은 양: " + result);
11    }
12 }
```

실행 결과

사과 1개에서 남은 양: 0.2999999999999993

- 정확한 계산이 필요하면 정수 연산으로 변경

나눗셈 연산에서 예외 방지하기

- 나눗셈(/) 또는 나머지(%) 연산에서 좌측 피연산자가 정수이고 우측 피연산자가 0일 경우 ArithmeticException 발생
- 좌측 피연산자가 실수이거나 우측 피연산자가 0.0 또는 0.0f이면 예외가 발생하지 않고 연산의 결과는 Infinity(무한대) 또는 NaN(Not a Number)이 됨

```
5 / 0.0 → Infinity
```

```
5 % 0.0 → NaN
```

- Infinity 또는 NaN 상태에서 계속해서 연산을 수행하면 안 됨
- Double.isInfinite()와 DoubleisNaN()를 사용해 /와 % 연산의 결과가 Infinity 또는 NaN인지 먼저 확인하고 다음 연산을 수행하는 것이 좋음

비교 연산자

- 비교 연산자는 동등(==, !=) 또는 크기(<, <=, >, >=)를 평가해서 boolean 타입인 true/false를 산출
- 흐름 제어문인 조건문(if), 반복문(for, while)에서 실행 흐름을 제어할 때 주로 사용

구분	연산식			설명
동등 비교	피연산자1	==	피연산자2	두 피연산자의 값이 같은지를 검사
	피연산자1	!=	피연산자2	두 피연산자의 값이 다른지를 검사
크기 비교	피연산자1	>	피연산자2	피연산자1이 큰지를 검사
	피연산자1	>=	피연산자2	피연산자1이 크거나 같은지를 검사
	피연산자1	<	피연산자2	피연산자1이 작은지를 검사
	피연산자1	<=	피연산자2	피연산자1이 작거나 같은지를 검사

■ 둘 ()를 사용

논리 연산자

- 논리곱(&&), 논리합(||), 배타적 논리합(^) 그리고 논리 부정(!) 연산을 수행
- 흐름 제어문인 조건문(if), 반복문(for, while) 등에서 주로 이용

구분	연산식			결과	설명
AND (논리곱)	true	&& 또는 &	true	true	피연산자 모두가 true일 경우에만 연산 결과가 true
	true		false	false	
	false		true	false	
	false		false	false	
OR (논리합)	true	 또는 	true	true	피연산자 중 하나만 true이면 연산 결과는 true
	true		false	true	
	false		true	true	
	false		false	false	
XOR (배타적 논리합)	true	^	true	false	피연산자가 하나는 true이고 다른 하나가 false일 경우에만 연산 결과가 true
	true		false	true	
	false		true	true	
	false		false	false	
NOT (논리 부정)		!	true	false	피연산자의 논리값을 바꿈
			false	true	

비트 논리 연산자

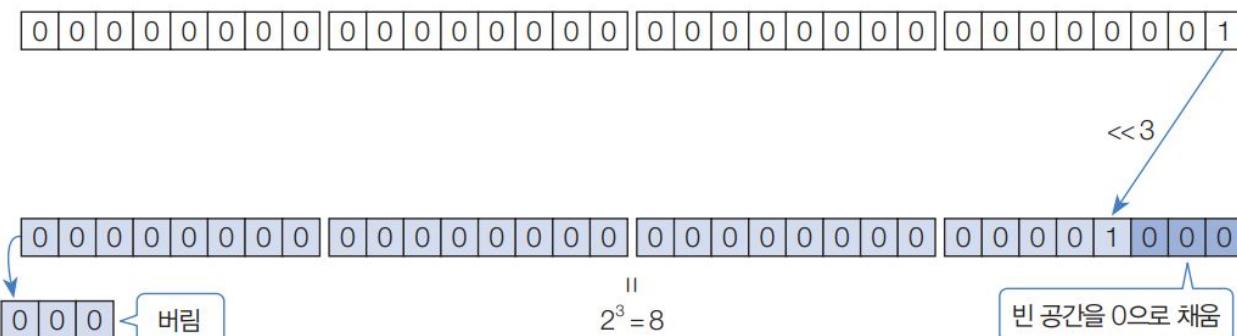
- bit 단위로 논리 연산을 수행. 0과 1이 피연산자가 됨
- byte, short, int, long만 피연산자가 될 수 있고, float, double은 피연산자가 될 수 없음

구분	연산식			결과	설명
AND (논리곱)	1	&	1	1	두 비트 모두 1일 경우에만 연산 결과가 1
	1		0	0	
	0		1	0	
	0		0	0	
OR (논리합)	1		1	1	두 비트 중 하나만 1이면 연산 결과는 1
	1		0	1	
	0		1	1	
	0		0	0	
XOR (배타적 논리합)	1	^	1	0	두 비트 중 하나는 1이고 다른 하나가 0일 경우 연산 결과는 1
	1		0	1	
	0		1	1	
	0		0	0	
NOT (논리 부정)		~	1	0	보수
			0	1	

비트 이동 연산자

- 비트를 좌측 또는 우측으로 밀어서 이동시키는 연산을 수행

구분	연산식			설명
이동 (shift)	a	«	b	정수 a의 각 비트를 b만큼 왼쪽으로 이동 오른쪽 빈자리는 0으로 채움 $a \times 2^b$ 와 동일한 결과가 됨
	a	»	b	정수 a의 각 비트를 b만큼 오른쪽으로 이동 왼쪽 빈자리는 최상위 부호 비트와 같은 값으로 채움 $a / 2^b$ 와 동일한 결과가 됨
	a	»»	b	정수 a의 각 비트를 b만큼 오른쪽으로 이동 왼쪽 빈자리는 0으로 채움



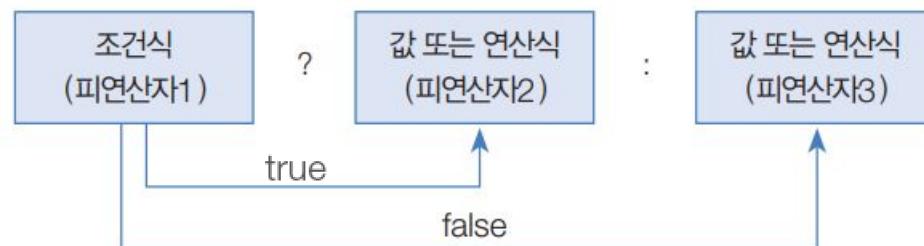
대입 연산자

- 우측 피연산자의 값을 좌측 피연산자인 변수에 대입. 우측 피연산자에는 리터럴 및 변수, 다른 연산식이 올 수 있음
- 단순히 값을 대입하는 단순 대입 연산자와 정해진 연산을 수행한 후 결과를 대입하는 복합 대입 연산자가 있음

구분	연산식			설명	
복합 대입 연산자	단순 대입 연산자	변수	=	피연산자	우측의 피연산자의 값을 변수에 저장
	변수	+=	피연산자	우측의 피연산자의 값을 변수의 값과 더한 후에 다시 변수에 저장 (변수 = 변수 + 피연산자)	
	변수	-=	피연산자	우측의 피연산자의 값을 변수의 값에서 뺀 후에 다시 변수에 저장 (변수 = 변수 - 피연산자)	
	변수	*=	피연산자	우측의 피연산자의 값을 변수의 값과 곱한 후에 다시 변수에 저장 (변수 = 변수 * 피연산자)	
	변수	/=	피연산자	우측의 피연산자의 값으로 변수의 값을 나눈 후에 다시 변수에 저장 (변수 = 변수 / 피연산자)	
	변수	%=	피연산자	우측의 피연산자의 값으로 변수의 값을 나눈 후에 나머지를 변수에 저장 (변수 = 변수 % 피연산자)	
	변수	&=	피연산자	우측의 피연산자의 값과 변수의 값을 & 연산 후 결과를 변수에 저장 (변수 = 변수 & 피연산자)	
	변수	=	피연산자	우측의 피연산자의 값과 변수의 값을 연산 후 결과를 변수에 저장 (변수 = 변수 피연산자)	
	변수	^=	피연산자	우측의 피연산자의 값과 변수의 값을 ^ 연산 후 결과를 변수에 저장 (변수 = 변수 ^ 피연산자)	
	변수	<<=	피연산자	우측의 피연산자의 값과 변수의 값을 << 연산 후 결과를 변수에 저장 (변수 = 변수 << 피연산자)	
	변수	>>=	피연산자	우측의 피연산자의 값과 변수의 값을 >> 연산 후 결과를 변수에 저장 (변수 = 변수 >> 피연산자)	
	변수	>>>=	피연산자	우측의 피연산자의 값과 변수의 값을 >>> 연산 후 결과를 변수에 저장 (변수 = 변수 >>> 피연산자)	

삼항 연산자

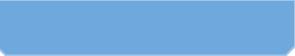
- 총 3개의 피연산자를 가짐
- ? 앞의 피연산자는 boolean 변수 또는 조건식. 이 값이 true이면 콜론(:) 앞의 피연산자가 선택되고, false이면 콜론 뒤의 피연산자가 선택됨

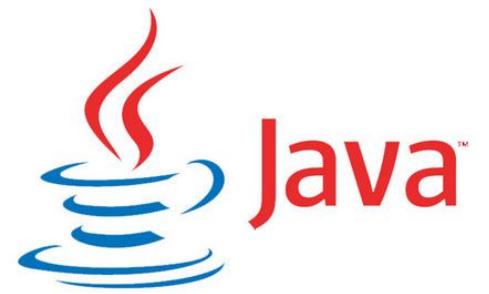


연산이 수행되는 순서

- 덧셈(+), 뺄셈(-) 연산자보다 곱셈(*), 나눗셈(/) 연산자가. &&보다는 >, < 우선순위가 높음
- 우선순위가 같은 연산자의 경우 대부분 왼쪽에서부터 오른쪽으로(→) 연산을 수행

연산자	연산 방향	우선순위
증감(++, --), 부호(+, -), 비트(~), 논리(!)	←	
산술(*, /, %)	→	높음
산술(+, -)	→	
수프트(<<, >>, >>>)	→	
비교(<, >, <=, >=, instanceof)	→	
비교(==, !=)	→	
논리(&)	→	
논리(^)	→	
논리()	→	
논리(&&)	→	
논리()	→	
조건(?:)	→	
대입(=, +=, -=, *=, /=, %=, &=, ^=, =, <=, >=, >>=)	←	낮음

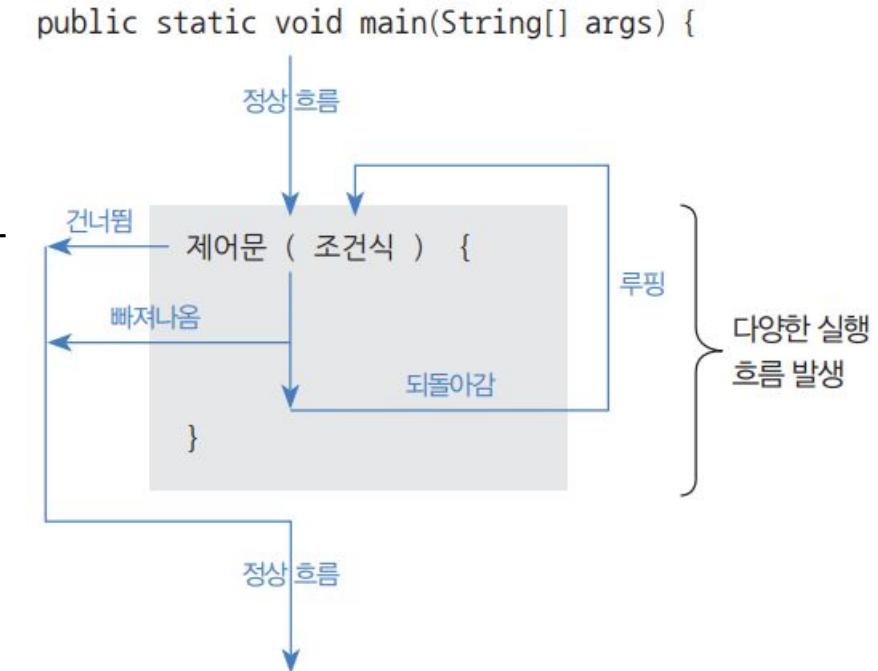




Chapter 04 조건문과 반복문

코드가 실행되는 흐름 제어하기

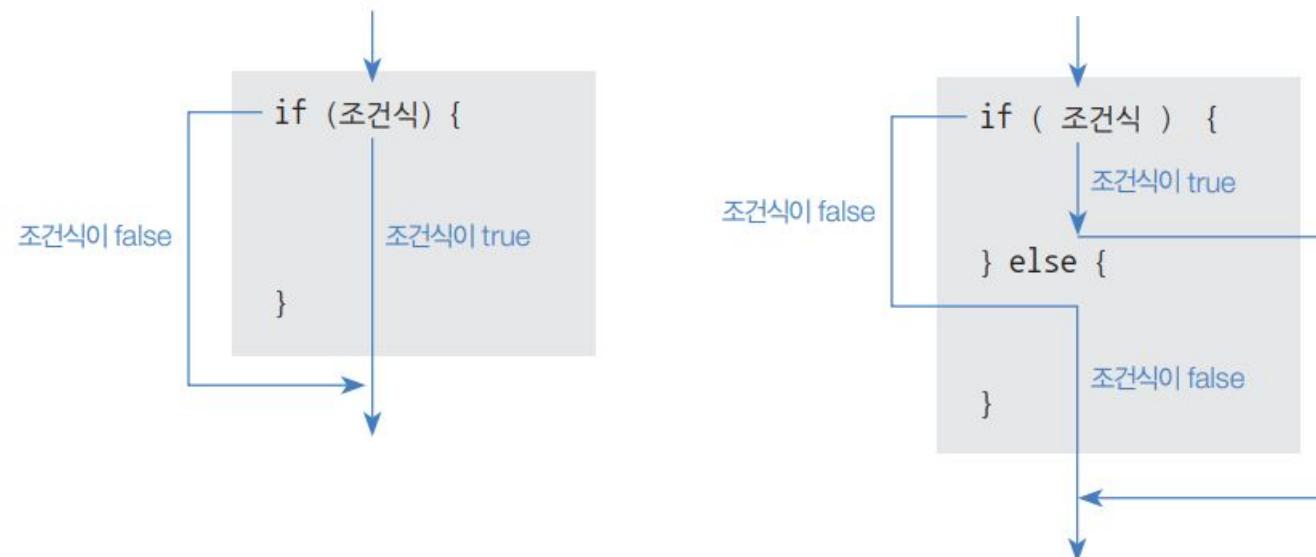
- 실행 흐름: main() 메소드의 시작 중괄호({})에서 끝 중괄호(})까지 위부터 아래로 실행되는 흐름
- 흐름 제어문: 실행 흐름을 개발자가 원하는 방향으로 바꿀 수 있도록 해주는 것
- 루핑: 반복문이 실행 완료된 경우 제어문 처음으로 다시 되돌아가 반복 실행되는 것



조건문	반복문
if 문, switch 문	for 문, while 문, do-while 문

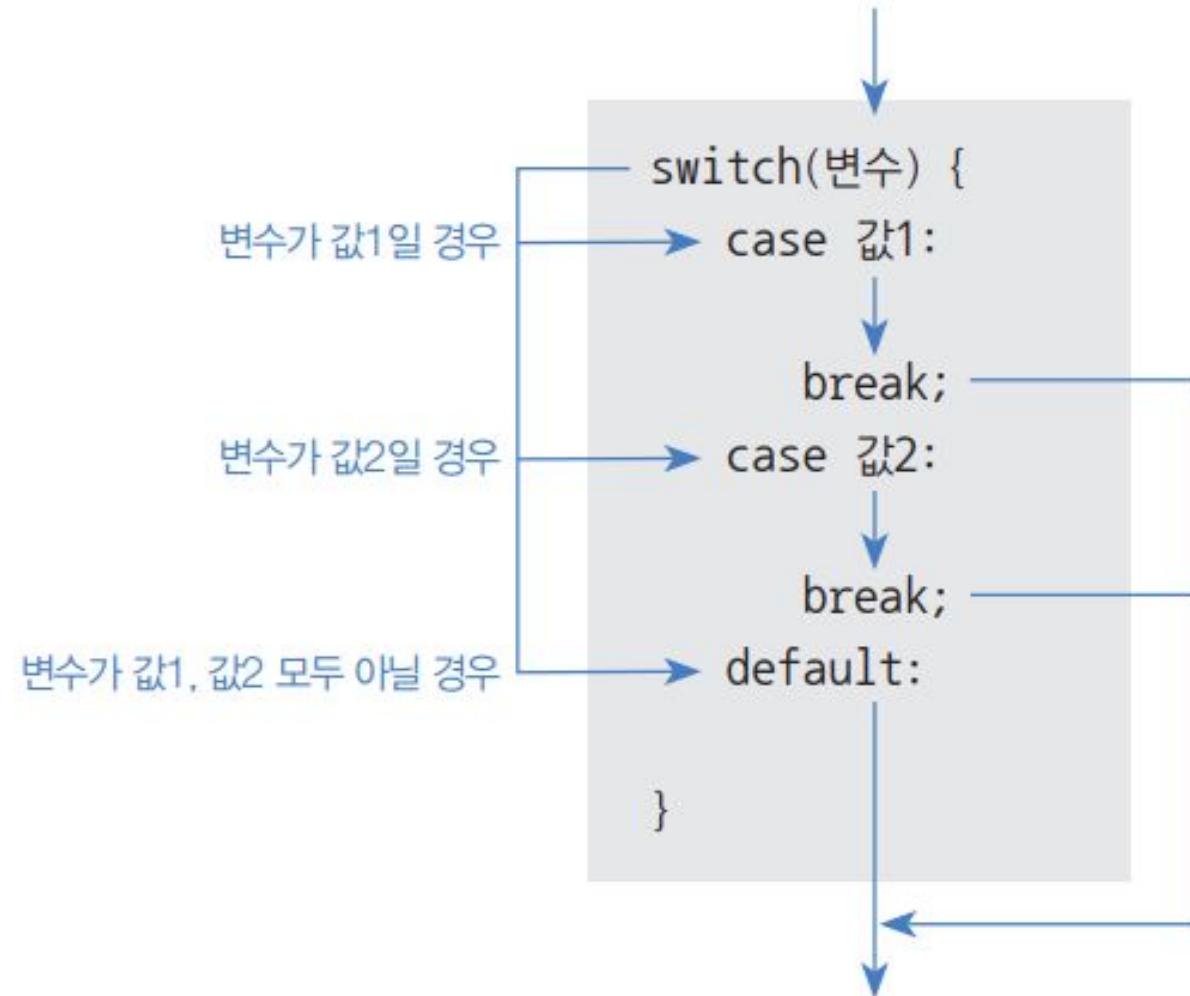
조건에 따라 실행되는 if 문

- If 문: 조건식의 결과에 따라 블록 실행 여부가 결정
- 조건식에는 true 또는 false 값을 산출할 수 있는 연산식이나 boolean 변수가 올 수 있음
- 조건식이 true이면 블록을 실행하고 false이면 블록을 실행하지 않음
- if-else 문: 조건식이 true이면 if 문 블록이 실행되고, false이면 else 블록이 실행



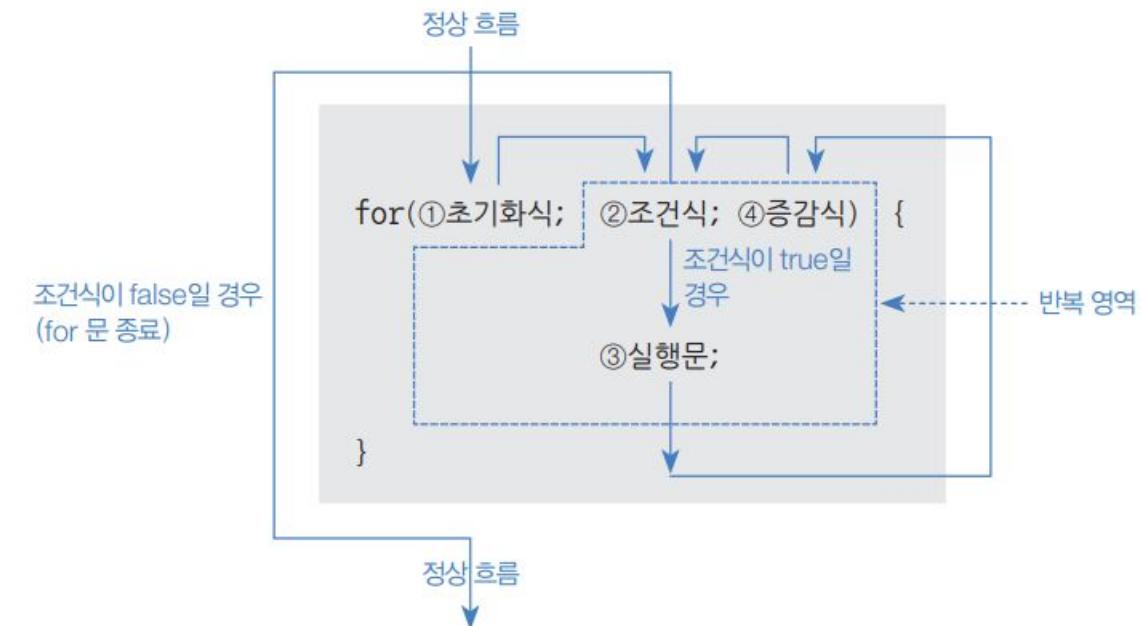
변수값에 따라 case를 실행하는 switch 문

- switch 문: 괄호 안의 변수값에 따라 해당 case로 가서 실행문을 실행.
- 변수값과 동일한 값을 갖는 case가 없으면 default로 가서 실행문을 실행하며, default 생략 가능
- break는 다음 case를 실행하지 않고 switch 문을 빠져나갈 때 사용.
break가 없다면 다음 case가 연달아 실행



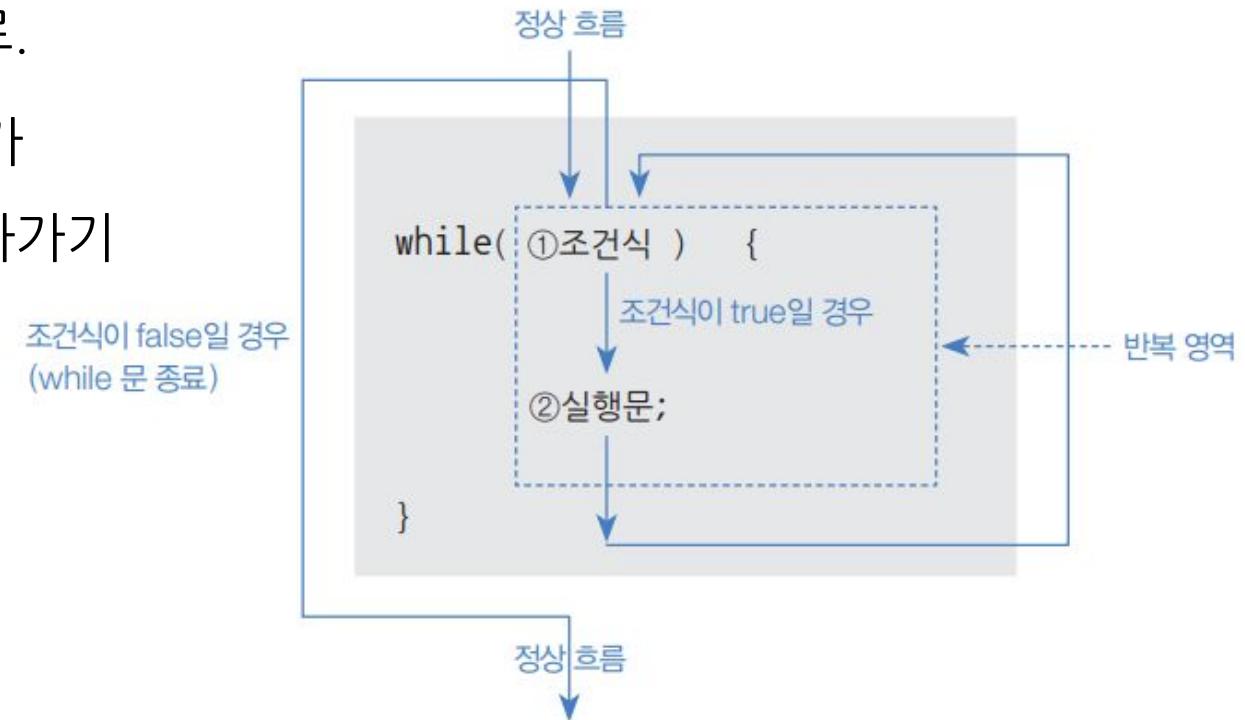
실행문을 반복하는 for 문

- for 문은 실행문을 여러 번 반복 실행해주기 때문에 코드를 간결하게 만들어줌
- ①초기화식이 제일 먼저 실행 ②조건식을 평가해서 true이면 ③실행문을 실행시키고, false이면 for 문을 종료하고 블록을 건너뜀.
- ②조건식이 true가 되어 ③실행문을 모두 실행하면 ④증감식이 실행.
- 다시 ②조건식을 평가. 평가 결과가 다시 true이면 ③ → ④ → ②로 다시 진행하고, false이면 for 문이 끝남
- 초기화식에서 부동 소수점을 쓰는 float 타입을 사용하지 않도록 주의



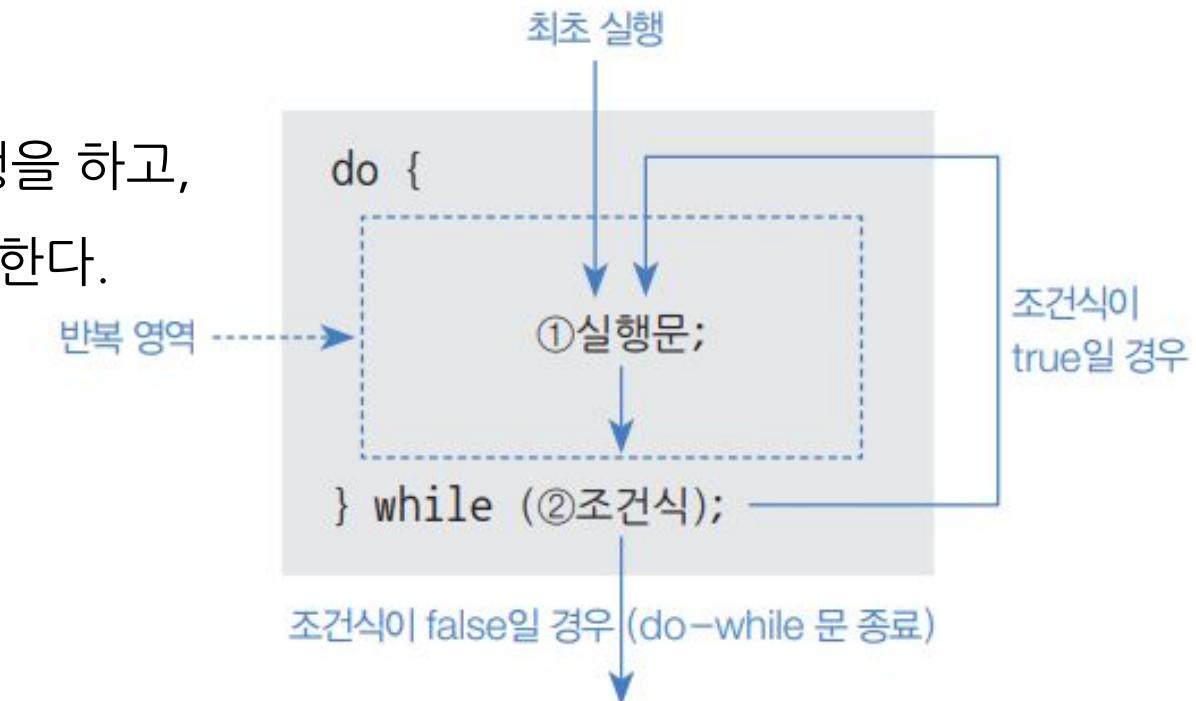
조건식에 따라 실행문을 반복하는 while 문

- 조건식이 true일 경우에 계속해서 반복하고, false가 되면 반복을 멈추고 while 문을 종료
- while 문이 처음 실행될 때 ①조건식을 평가. 평가 결과가 true이면 ②실행문을 실행한다.
- ②실행문이 모두 실행되면 조건식으로 되돌아가서 ①조건식을 다시 평가. 다시 조건식이 true라면 ② → ①로 진행하고, false라면 while 문을 종료.
- 조건식에 true를 사용하면 while(true) {...}가 되어서 무한 반복. 이 경우 while 문을 빠져나가기 위한 코드 필요



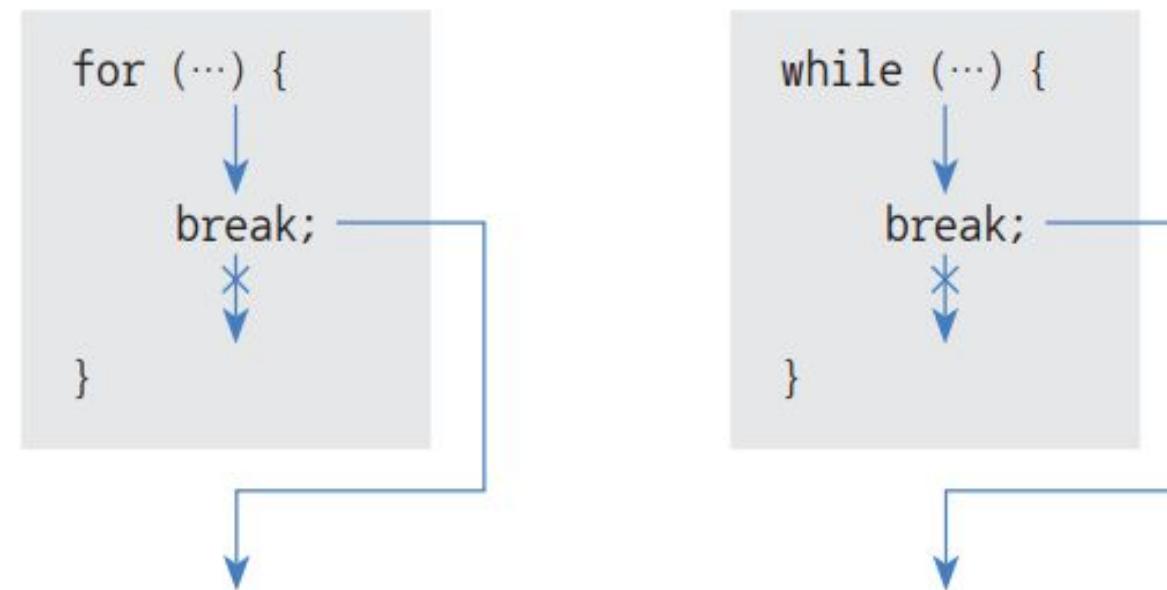
실행 결과에 따라 실행문을 반복하는 do-while 문

- 블록 내부를 먼저 실행시키고 실행 결과에 따라서 반복 실행을 계속할지 결정
- 작성 시 while() 뒤에 반드시 세미콜론(;)을 붙여야 하는데 주의
- do-while 문이 처음 실행될 때 ① 실행문을 우선 실행한다. ① 실행문이 모두 실행되면 ② 조건식을 평가
- 평가 결과가 true이면 ① → ②와 같이 반복 실행을 하고, 조건식의 결과가 false이면 do-while 문을 종료한다.



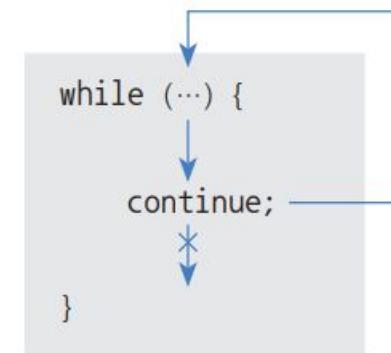
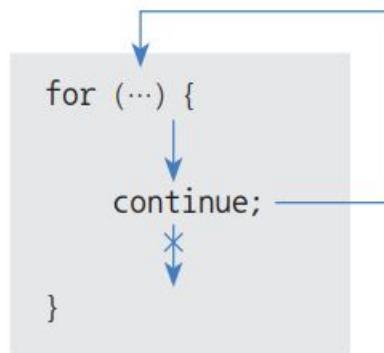
제어문을 종료하는 break 문

- 반복문인 for 문, while 문, do-while 문을 실행 중지하거나 조건문인 switch 문을 종료할 때 사용
- break 문은 대개 if 문과 같이 사용되어 조건식에 따라 for 문과 while 문을 종료



조건식으로 이동하는 continue 문

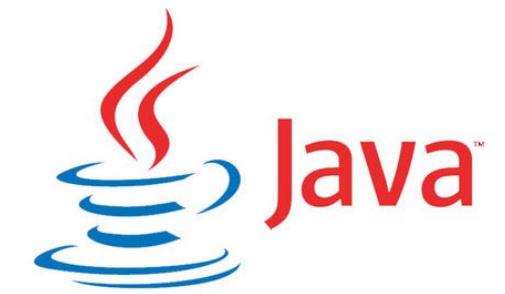
- 반복문인 for 문, while 문, do-while 문에서만 사용
- 블록 내부에서 continue 문이 실행되면 for 문의 증감식 또는 while 문, do-while 문의 조건식으로 바로 이동
- break 문과 달리 반복문을 종료하지 않고 계속 반복을 수행
- 대개 if 문과 같이 사용되며, 특정 조건을 만족하는 경우에 continue 문을 실행해서 그 이후의 문장을 실행하지 않고 다음 반복으로 넘어감



- 반복문 앞에 이름을 붙이고, 그이름을 break, continue와 같이 사용함으로써 둘 이상의 반복문을 벗어나거나 반복을 건너뛰는 것이 가능하다.

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

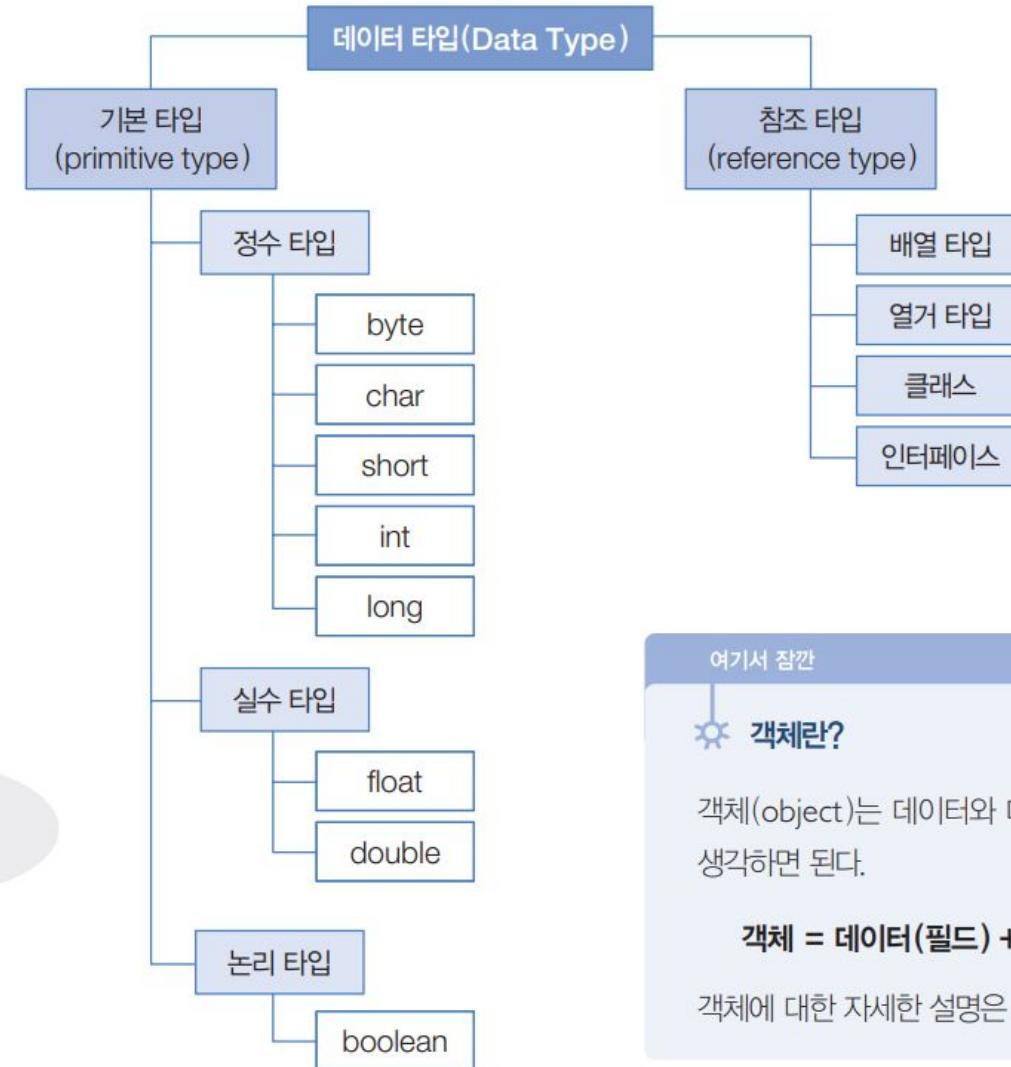




Chapter 05 참조 타입

참조 타입

- 객체의 번지를 참조하는 타입.
- 배열, 열거, 클래스, 인터페이스 타입
- 기본 타입으로 선언된 변수는 값 자체를 저장하지만, 참조 타입으로 선언된 변수는 객체가 생성된 메모리 번지를 저장



여기서 잠깐

★ 객체란?

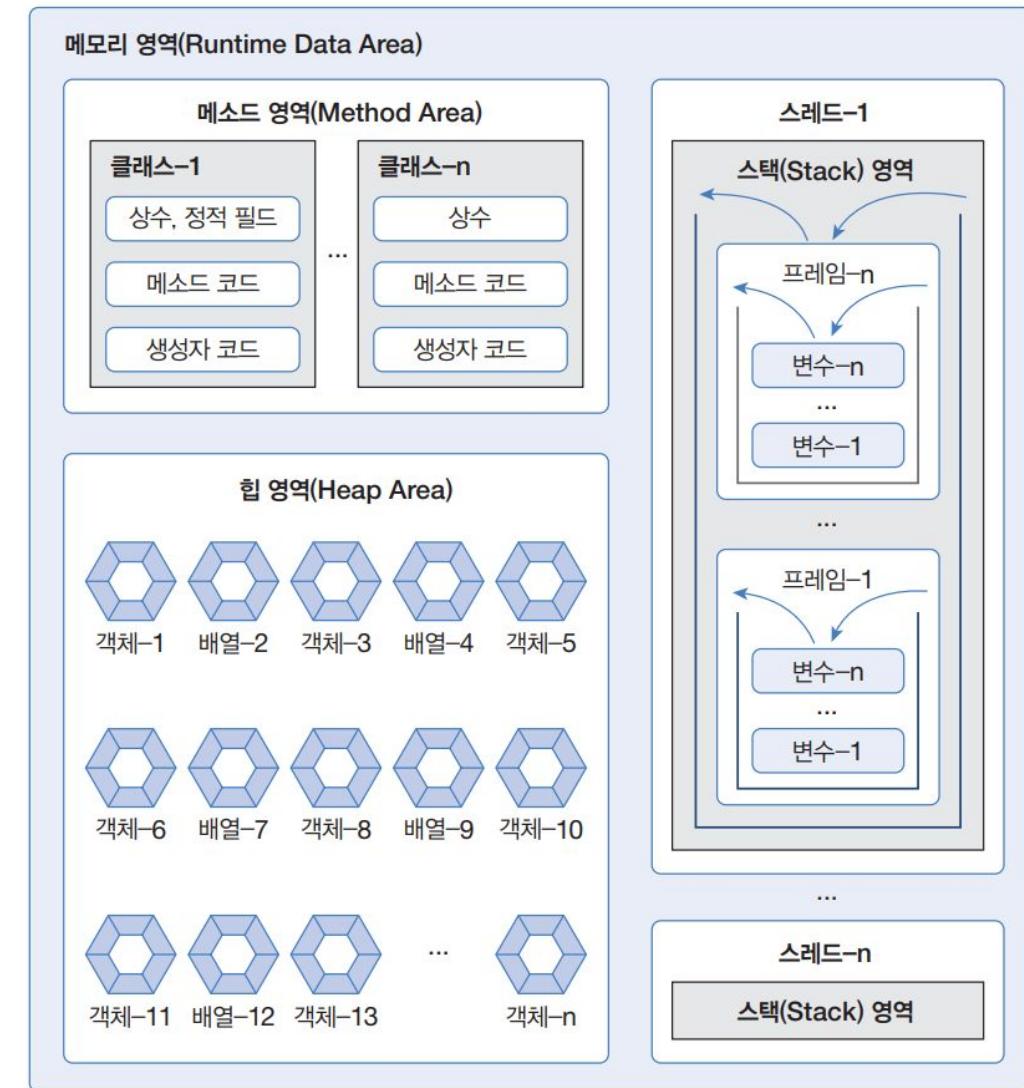
객체(object)는 데이터와 메소드로 구성된 덩어리라고 생각하면 된다.

객체 = 데이터(필드) + 메소드

객체에 대한 자세한 설명은 6장에서 살펴본다.

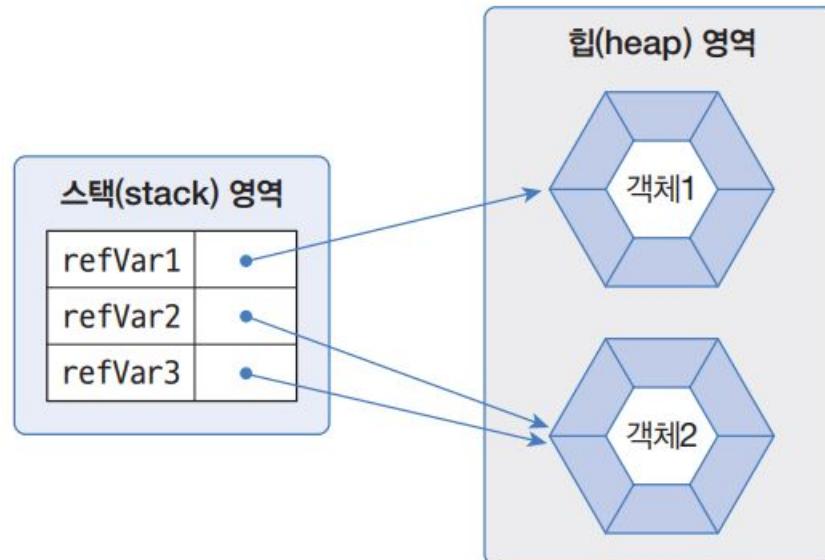
메소드, 힙, 스택 영역

- JVM은 운영체제에서 할당받은 메모리 영역을 메소드 영역, 힙 영역, 스택 영역으로 구분해서 사용
- 메소드 영역: 바이트코드 파일을 읽은 내용이 저장되는 영역
- 힙 영역: 객체가 생성되는 영역. 객체의 번지는 메소드 영역과 스택 영역의 상수와 변수에서 참조
- 스택 영역: 메소드를 호출할 때마다 생성되는 프레임이 저장되는 영역



==, != 연산자

- ==, != 연산자는 객체의 번지를 비교해 변수의 값이 같은지, 아닌지를 조사
- 번지가 같다면 동일한 객체를 참조하는 것이고, 다르다면 다른 객체를 참조하는 것

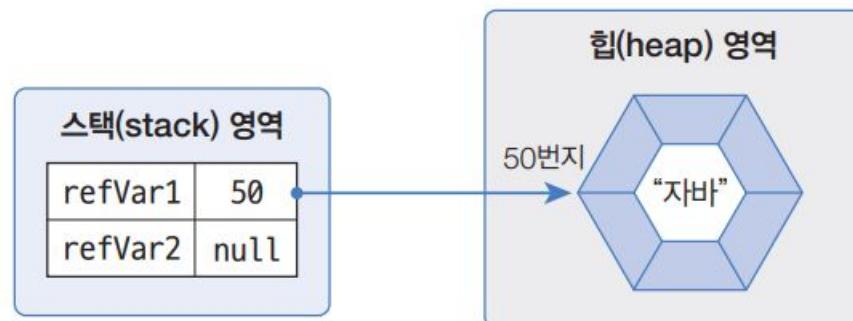


null 값

- null(널) 값: 참조 타입 변수는 아직 번지를 저장하고 있지 않다는 뜻
- null도 초기값으로 사용할 수 있기 때문에 null로 초기화된 참조 변수는 스택 영역에 생성

NullPointerException

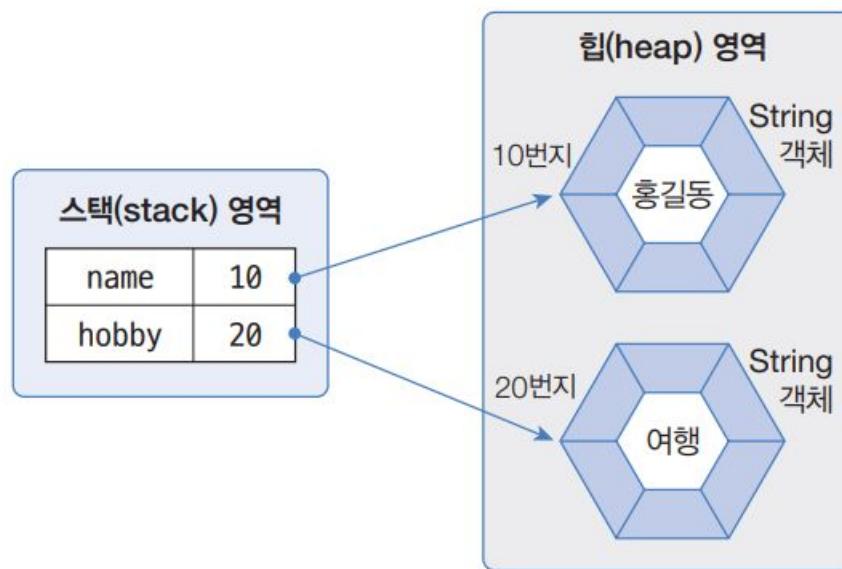
- 변수가 null인 상태에서 객체의 데이터나 메소드를 사용하려 할 때 발생하는 예외
- 참조 변수가 객체를 정확히 참조하도록 번지를 대입해야 해결됨



String 타입

- 문자열은 String 객체로 생성

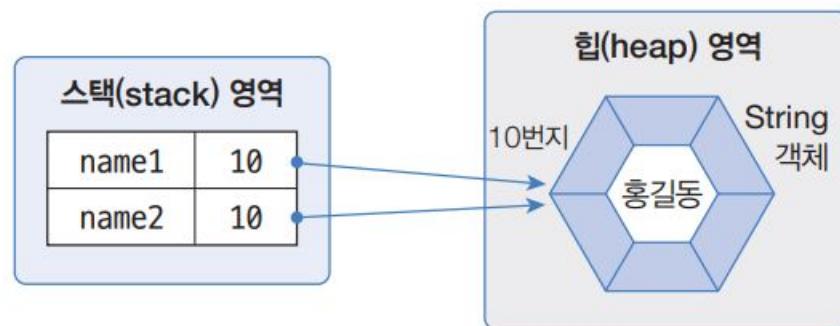
```
String name;           //String 타입 변수 name 선언  
name = "홍길동";     //name 변수에 문자열 대입  
String hobby = "여행"; //String 타입 변수 hobby를 선언하고 문자열 대입
```



문자열 비교

- 문자열 리터럴이 동일하다면 String 객체를 공유

```
String name1 = "홍길동";  
String name2 = "홍길동";
```



- new 연산자(객체 생성 연산자)로 직접 String 객체를 생성/대입 가능

문자열 추출

- charAt() 메소드로 문자열에서 매개값으로 주어진 인덱스의 문자를 리턴해 특정 위치의 문자를 얻을 수 있음

자	바		프	로	그	래	밍
0	1	2	3	4	5	6	7

문자열 길이

- 문자열에서 문자의 개수를 얻고 싶다면 length() 메소드를 사용

총 8문자

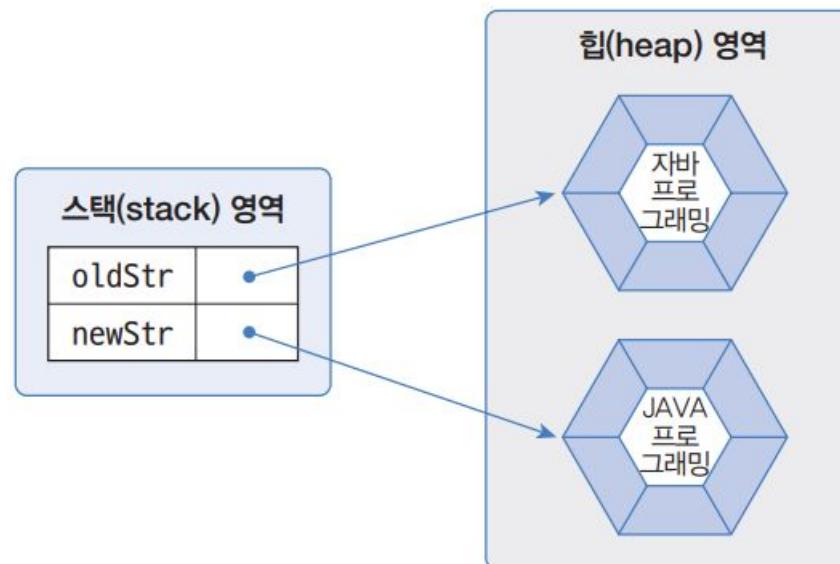
A diagram illustrating the length of a string. It shows a horizontal array of eight boxes, each containing a character from the Korean word "자바프로그래밍". Above the array, a blue bracket spans the entire width of the boxes, labeled "총 8문자" (Total 8 characters). Below the array, numerical indices from 0 to 7 are aligned under each character box.

자	바		프	로	그	래	밍
0	1	2	3	4	5	6	7

문자열 대체

- replace() 메소드는 기존 문자열은 그대로 두고, 대체한 새로운 문자열을 리턴

```
String oldStr = "자바 프로그래밍";
String newStr = oldStr.replace("자바", "JAVA");
```



문자열 잘라내기

- 문자열에서 특정 위치의 문자열을 잘라내어 가져오고 싶다면 `substring()` 메소드를 사용

메소드	설명
<code>substring(int beginIndex)</code>	<code>beginIndex</code> 에서 끝까지 잘라내기
<code>substring(int beginIndex, int endIndex)</code>	<code>beginIndex</code> 에서 <code>endIndex</code> 앞까지 잘라내기

문자열 찾기

- 문자열에서 특정 문자열의 위치를 찾고자 할 때에는 `indexOf()` 메소드를 사용

```
String subject = "자바 프로그래밍";
int index = subject.indexOf("프로그래밍");
```



문자열 분리

- 구분자가 있는 여러 개의 문자열을 분리할 때 `split()` 메소드를 사용

```
String board = "번호,제목,내용,글쓴이";
String[] arr = board.split(",");
```

arr[0] arr[1] arr[2] arr[3]

"번호"	"제목"	"내용"	"성명"
------	------	------	------

String[] args 매개변수의 필요성

- 자바 프로그램을 실행하기 위해 main() 메소드를 작성하면서 문자열 배열 형태인 String[] args 매개변수가 필요
- 프로그램 실행 시 입력값이 부족하면 길이가 0인 String 배열 참조

```
{ "10", "20" };  
main() 메소드 호출 시 전달  
public static void main(String[] args) { ... }
```

한정된 값으로 이루어진 Enum 타입

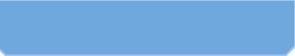
- 요일, 계절처럼 한정된 값을 갖는 타입
- 먼저 열거 타입 이름으로 소스 파일(.java)을 생성하고 한정된 값을 코드로 정의
- 열거 타입 이름은 첫 문자를 대문자로 하고 캐멀 스타일로 지어주는 것이 관례

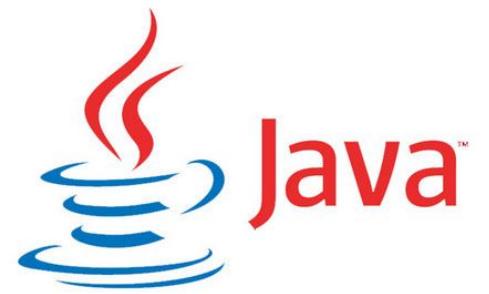
»» Week.java

```
1 package ch05.sec12;
2
3 public enum Week {
4     MONDAY,
5     TUESDAY,
6     WEDNESDAY,
7     THURSDAY,
8     FRIDAY,
9     SATURDAY,
10    SUNDAY
11 }
```

→ 열거 타입 이름

} 열거 상수 목록(한정된 값 목록)

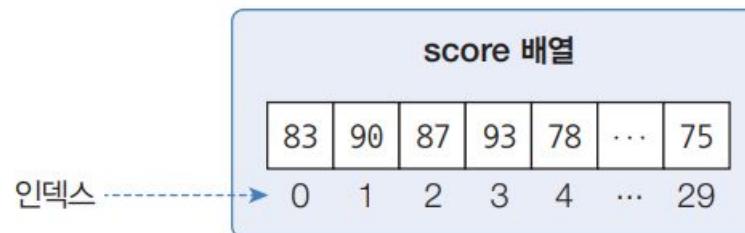




Chapter 05.A 배열

배열

- 연속된 공간에 값을 나열시키고, 각 값에 인덱스를 부여해 놓은 자료구조
- 인덱스는 대괄호 []와 함께 사용하여 각 항목의 값을 읽거나 저장하는데 사용



배열 변수 선언

- 두 가지 형태로 작성. 첫 번째가 관례적인 표기

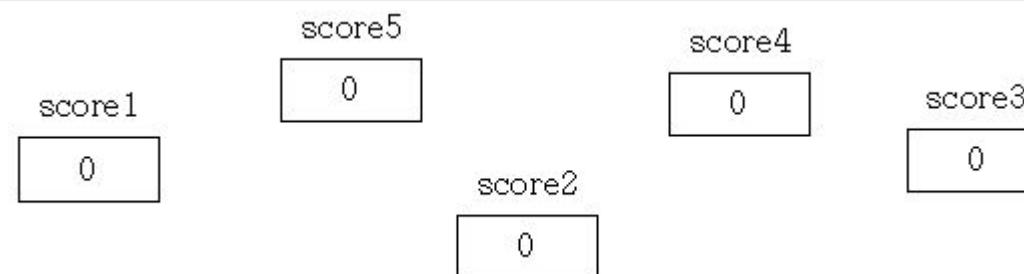
타입[] 변수;

타입 변수[];

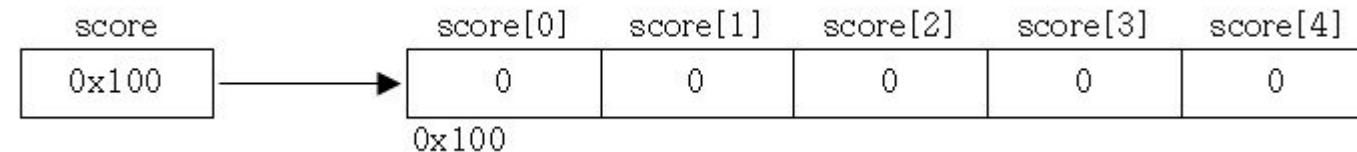
- 배열은 힙 영역에 생성되고 배열 변수는 힙 영역의 배열 주소를 저장
- 참조할 배열이 없다면 배열 변수도 null로 초기화할 수 있다

- 같은 타입의 여러 변수를 하나의 묶음으로 다룸
- 많은 양의 값(데이터)을 다룰 때 유용
- 배열의 각 요소는 서로 연속적

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



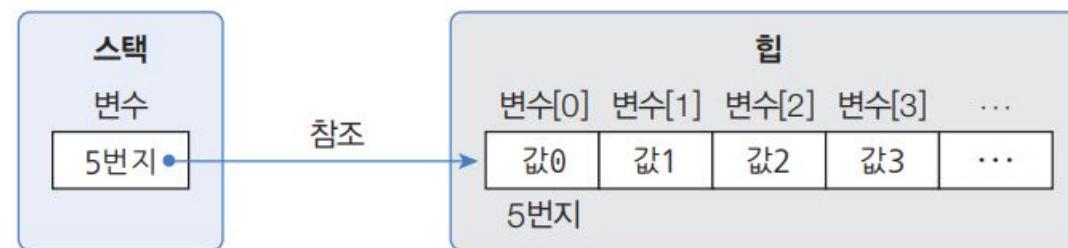
```
int[] score = new int[5]; // 5개의 int 값을 저장할 수 있는 배열을 생성한다.
```



값 목록으로 배열 생성

- 배열에 저장될 값의 목록이 있다면, 다음과 같이 간단하게 배열을 생성할 수 있음

```
타입[] 변수 = { 값0, 값1, 값2, 값3, ... };
```



- 배열 변수를 선언한 시점과 값 목록이 대입되는 시점이 다르다면 `new` 타입[]을 중괄호 앞에 붙여줌.
타입은 배열 변수를 선언할 때 사용한 타입과 동일하게 지정

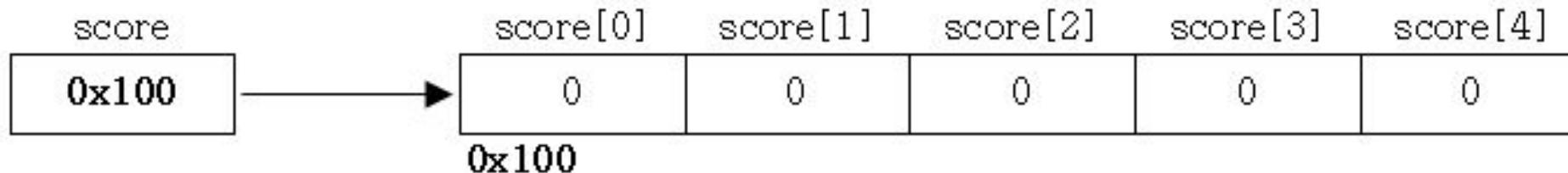
```
변수 = new 타입[] { 값0, 값1, 값2, 값3, ... };
```

값 목록으로 배열 생성

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

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

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



new 연산자로 배열 생성

- new 연산자로 값의 목록은 없지만 향후 값을 저장할 목적으로 배열을 미리 생성

```
타입[] 변수 = new 타입[길이];
```

- new 연산자로 배열을 처음 생성하면 배열 항목은 기본값으로 초기화된다.

데이터 타입	초기값
기본 타입	byte[] 0
	char[] '\u0000'
	short[] 0
	int[] 0
	long[] 0L
	float[] 0.0F
참조 타입	double[] 0.0
	boolean[] false
	클래스[] null
	인터페이스[] null

배열 길이

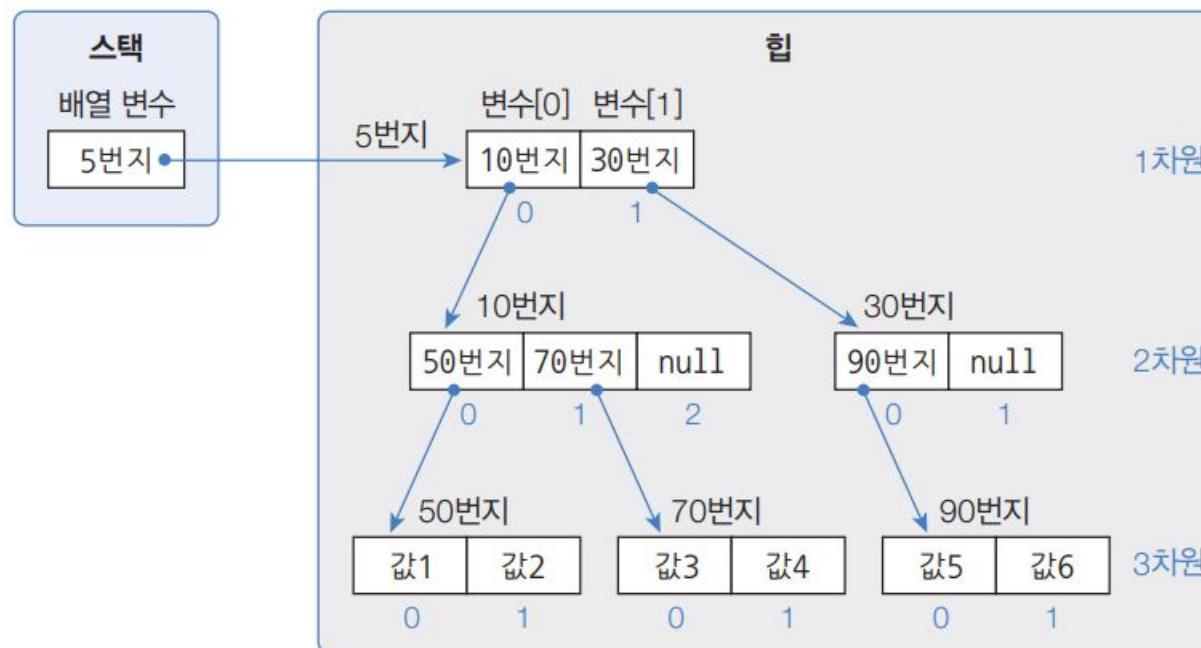
- 배열의 길이란 배열에 저장할 수 있는 항목 수
- 코드에서 배열의 길이를 얻으려면 도트(.) 연산자를 사용해서 참조하는 배열의 length 필드를 읽음

```
배열변수.length;
```

- 배열의 length 필드는 읽기만 가능하므로 값을 변경할 수는 없음
- 배열 길이는 for 문을 사용해서 전체 배열 항목을 반복할 때 많이 사용

다차원 배열

- 배열 항목에는 또 다른 배열이 대입된 배열

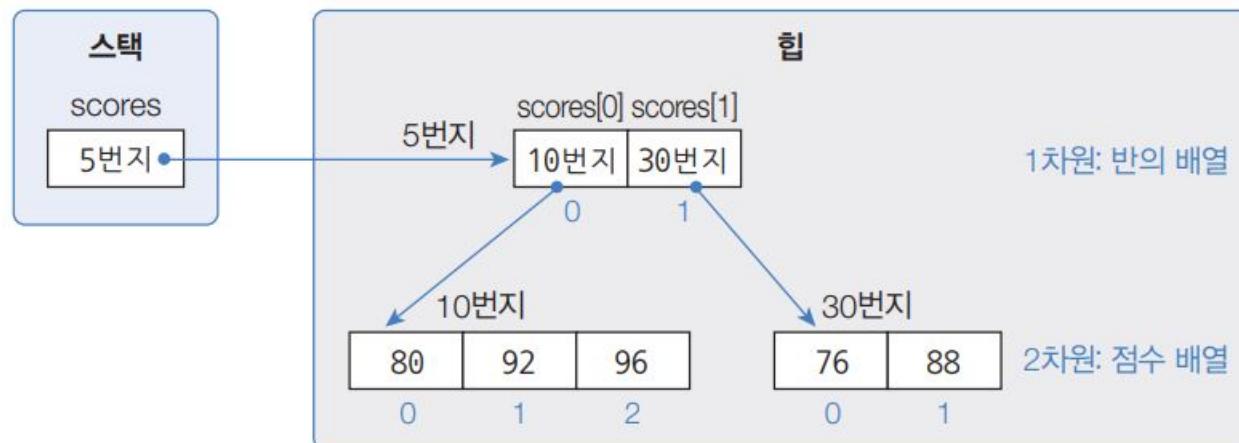


변수[1차원인덱스][2차원인덱스]...[N차원인덱스]

값 목록으로 다차원 배열 생성

- 값 목록으로 다차원 배열을 생성 시 배열 변수 선언 시 타입 뒤에 대괄호 []를 차원의 수만큼 붙이고, 값 목록도 마찬가지로 차원의 수만큼 중괄호를 중첩

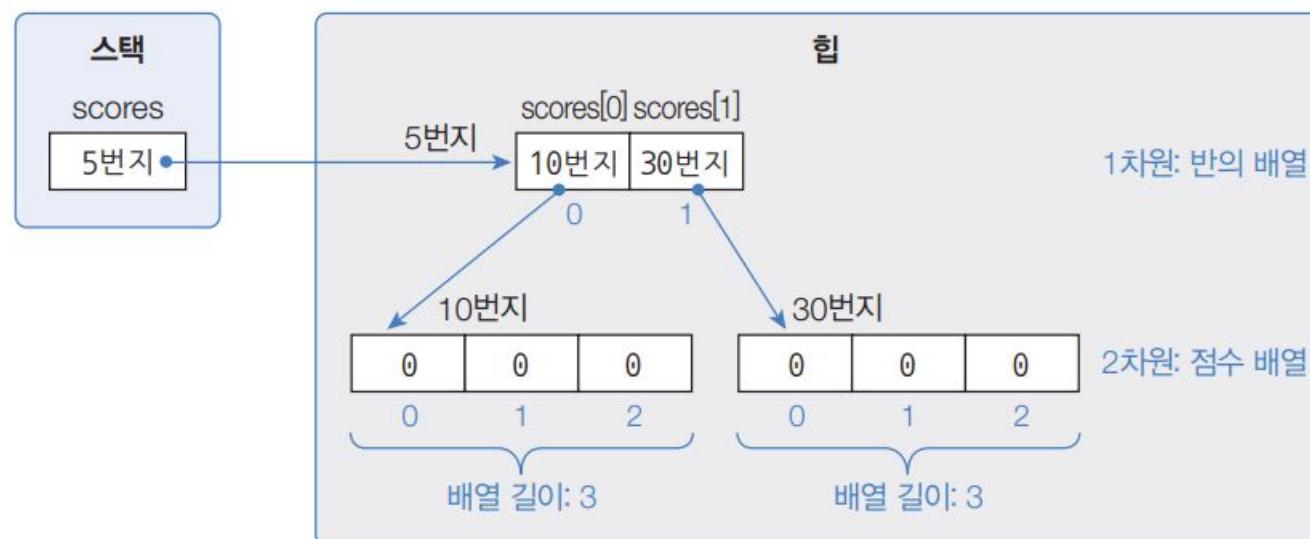
```
타입[][] 변수 = {
    {값1, 값2, ...},           1차원 배열의 0 인덱스
    {값3, 값4, ...},           1차원 배열의 1 인덱스
    ...
};
```



new 연산자로 다차원 배열 생성

- new 연산자로 다차원 배열을 생성하려면 배열 변수 선언 시 타입 뒤에 대괄호 []를 차원의 수만큼 붙이고, new 타입 뒤에도 차원의 수만큼 대괄호 []를 작성

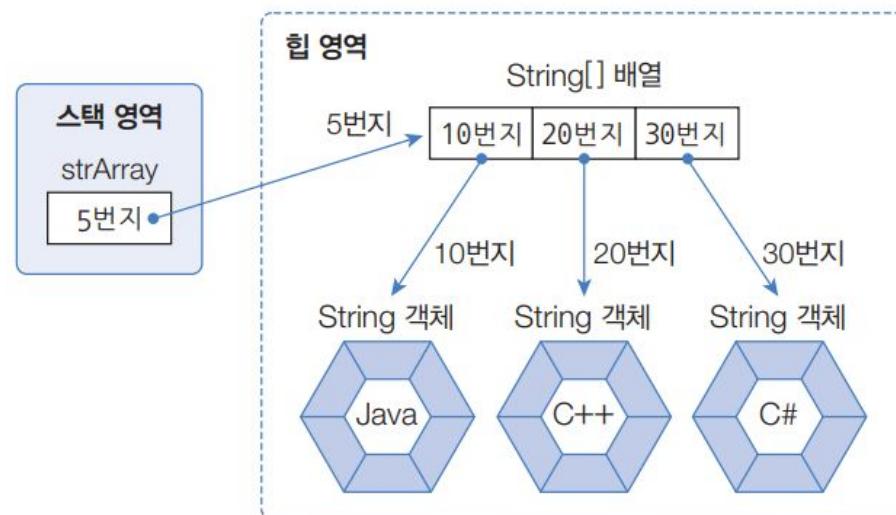
타입[][] 변수 = new 타입[1차원수][2차원수];



배열에서 객체 참조하기

- 기본 타입(byte, char, short, int, long, float, double, boolean) 배열은 각 항목에 값을 직접 저장
- 참조 타입(클래스, 인터페이스) 배열은 각 항목에 객체의 번지를 저장

```
String[] strArray = new String[3];
strArray[0] = "Java";
strArray[1] = "C++";
strArray[2] = "C#";
```



배열 복사하기

- 배열은 한 번 생성하면 길이를 변경할 수 없음. 더 많은 저장 공간이 필요하다면 더 큰 길이의 배열을 새로 만들고 이전 배열로부터 항목들을 복사해야 함.



- System의 arraycopy() 메소드를 이용해 배열 복사 가능

```
System.arraycopy(Object src, int srcPos, Object dest, int destPos, int length);
```

↑
원본 배열
복사
시작 인덱스

↑
원본 배열
복사
시작 인덱스

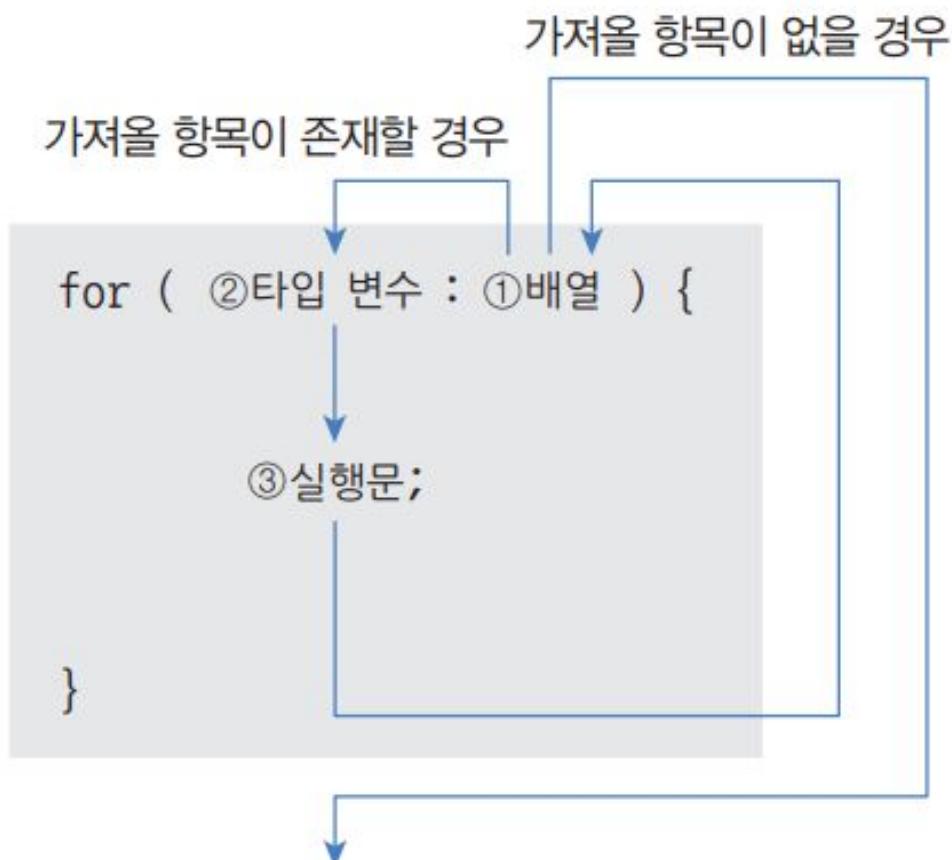
↑
새 배열
붙여넣기
시작 인덱스

↑
새 배열
붙여넣기
시작 인덱스

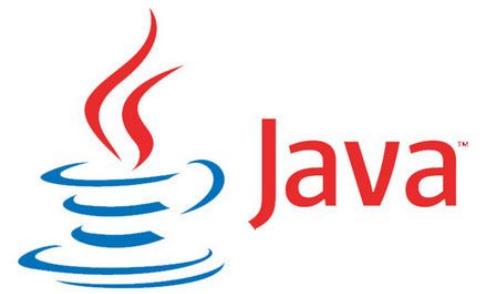
↑
복사 항목 수

배열 및 컬렉션 처리에 용이한 for 문

- 카운터 변수와 증감식을 사용하지 않고, 항목의 개수만큼 반복한 후 자동으로 for 문을 빠져나감
- for 문이 실행되면 ①배열에서 가져올 항목이 있을 경우
②변수에 항목을 저장, ③실행문을 실행
- 다시 반복해서 ①배열에서 가져올 다음 항목이 존재하면
② → ③ → ①로 진행하고, 가져올 다음 항목이 없으면 for 문을 종료



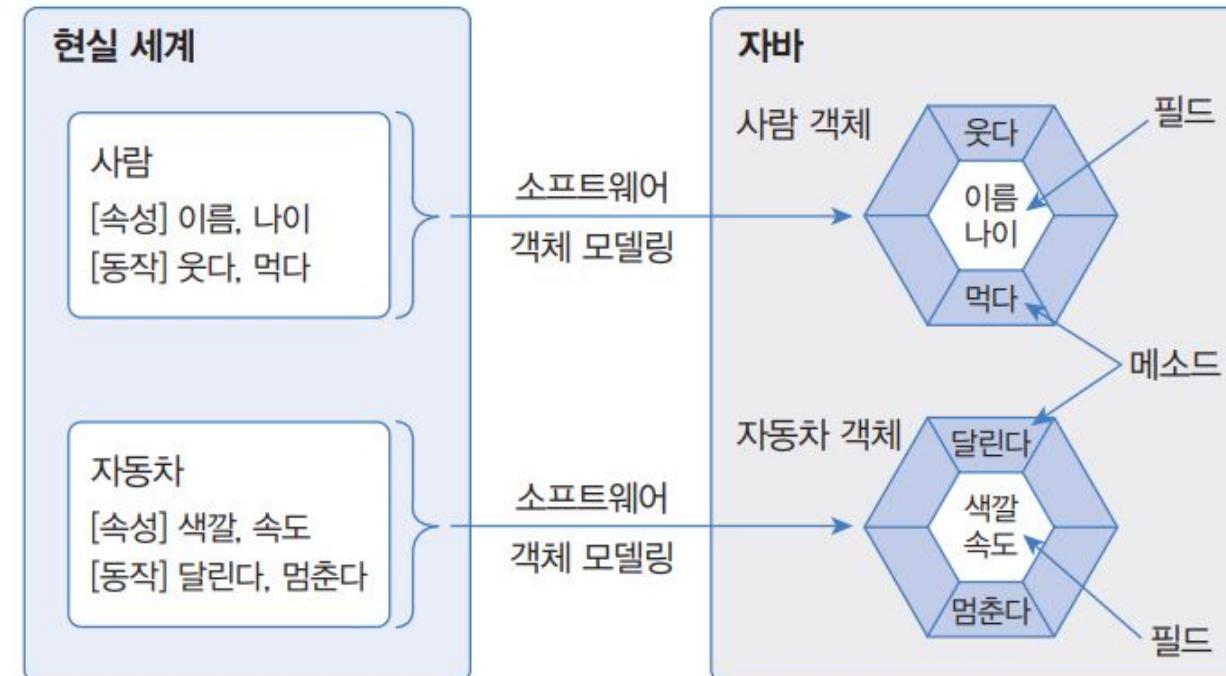




Chapter 06 클래스

객체

- 객체(object)란 물리적으로 존재하거나 개념적인 것 중에서 다른 것과 식별 가능한 것
- 객체는 속성과 동작으로 구성. 자바는 이러한 속성과 동작을 각각 필드와 메소드라고 부름

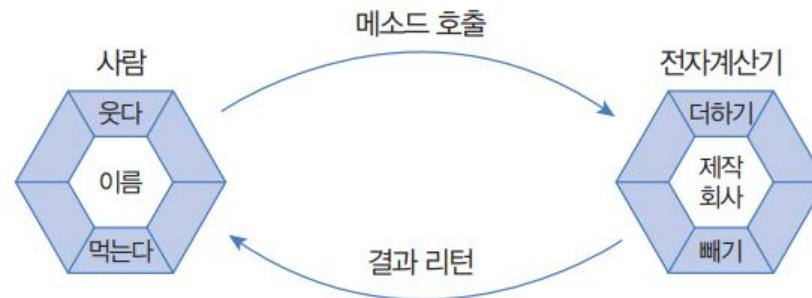


객체 지향 프로그래밍(OOP)

- 객체 객체들을 먼저 만들고, 이 객체들을 하나씩 조립해서 완성된 프로그램을 만드는 기법

객체의 상호작용

- 객체 지향 프로그램에서도 객체들은 다른 객체와 서로 상호작용하면서 동작
- 객체가 다른 객체의 기능을 이용할 때 이 메소드를 호출해 데이터를 주고받음

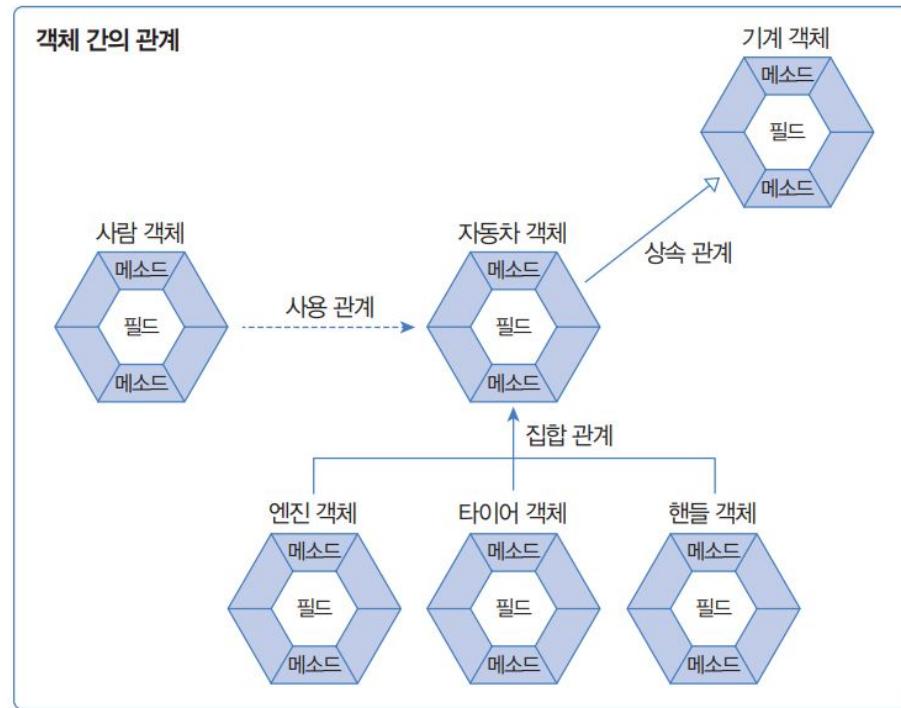


- 매개값: 객체가 전달하고자 하는 데이터이며, 메소드 이름과 함께 괄호() 안에 기술
- 리턴값: 메소드의 실행의 결과이며, 호출한 곳으로 돌려주는 값

메소드(매개값1, 매개값2, …);

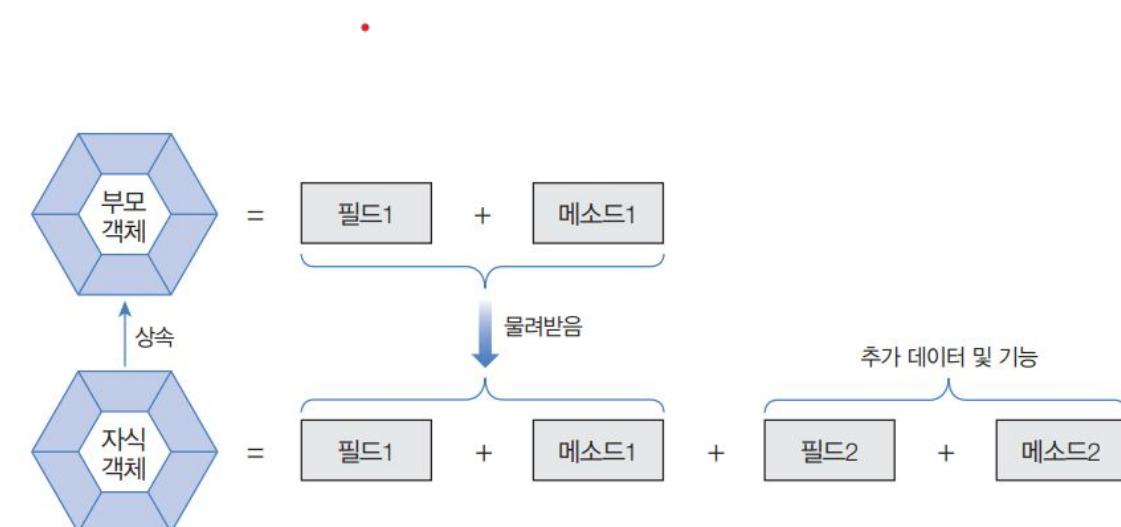
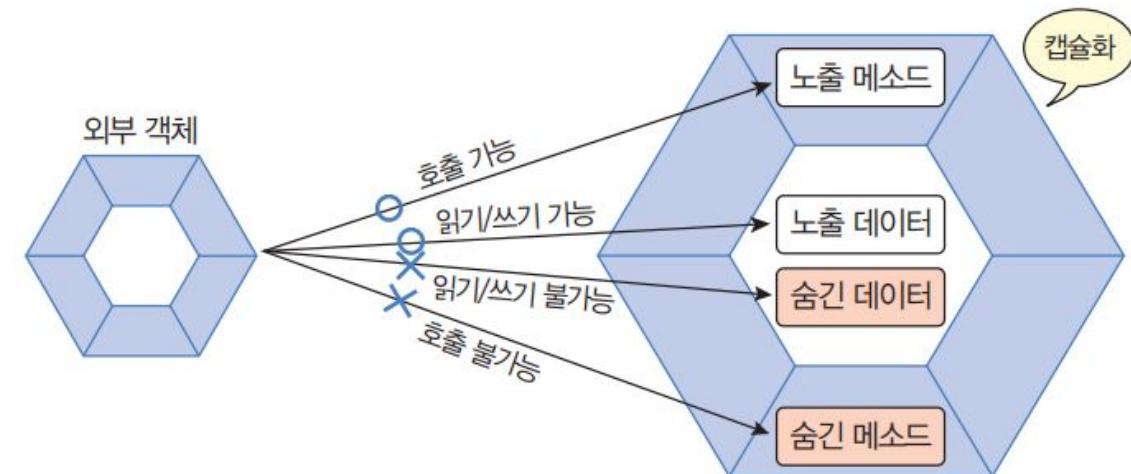
객체 간의 관계

- 집합 관계: 완성품과 부품의 관계
- 사용 관계: 다른 객체의 필드를 읽고 변경하거나 메소드를 호출하는 관계
- 상속 관계: 부모와 자식 관계. 필드, 메소드를 물려받음



객체 지향 프로그래밍의 특징

- 캡슐화: 객체의 데이터(필드), 동작(메소드)을 하나로 묶고 실제 구현 내용을 외부에 감추는 것
- 상속: 부모 객체가 자기 필드와 메소드를 자식 객체에게 물려줘 자식 객체가 사용할 수 있게 함
→ 코드 재사용성 높이고 유지 보수 시간 최소화
- 다형성: 사용 방법은 동일하지만 실행 결과가 다양함



클래스와 인스턴스

- 객체 지향 프로그래밍에서도 객체를 생성하려면 설계도에 해당하는 클래스가 필요
- 클래스로부터 생성된 객체를 해당 클래스의 인스턴스라고 부름
- 클래스로부터 객체를 만드는 과정을 인스턴스화라고 함
- 동일한 클래스로부터 여러 개의 인스턴스를 만들 수 있음



클래스 선언

- 객체를 생성(생성자)하고, 객체가 가져야 할 데이터(필드)가 무엇이고, 객체의 동작(메소드)은 무엇인지를 정의
- 클래스 선언은 소스 파일명과 동일하게 작성

[**클래스명.java**]

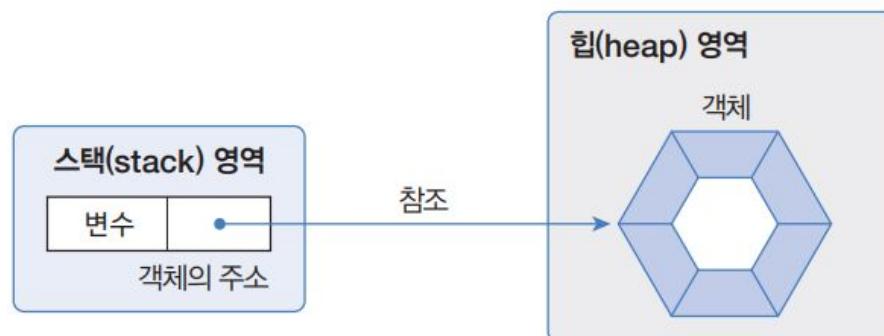
```
//클래스 선언  
public class 클래스명 {  
}
```

- 클래스명은 첫 문자를 대문자로 하고 캐멀 스타일로 작성. 숫자를 포함해도 되지만 첫 문자가 될 수 없고, 특수 문자 중 \$, _를 포함할 수 있음
- 공개 클래스: 어느 위치에 있든지 패키지와 상관없이 사용할 수 있는 클래스

클래스 변수

- 클래스로부터 객체를 생성하려면 객체 생성 연산자인 new가 필요
- new 연산자는 객체를 생성시키고 객체의 주소를 리턴

```
클래스 변수 = new 클래스();
```



- 라이브러리 클래스: 실행할 수 없으며 다른 클래스에서 이용하는 클래스
- 실행 클래스: main() 메소드를 가지고 있는 실행 가능한 클래스

생성자, 필드, 메소드

- 필드: 객체의 데이터를 저장하는 역할. 선언 형태는 변수 선언과 비슷하지만 쓰임새는 다름
- 생성자: new 연산자로 객체를 생성할 때 객체의 초기화 역할. 선언 형태는 메소드와 비슷하지만, 리턴 타입이 없고 이름은 클래스 이름과 동일
- 메소드: 객체가 수행할 동작. 함수로도 불림

• 필드

 객체의 데이터가 저장되는 곳

• 생성자

 객체 생성 시 초기화 역할 담당

• 메소드

 객체의 동작으로 호출 시 실행하는 블록

```
public class ClassName {  
    //필드 선언  
    → int fieldName;  
  
    //생성자 선언  
    → ClassName() { ... }  
  
    //메소드 선언  
    → int methodName() { ... }  
}
```

필드 선언

- 필드는 클래스 블록에서 선언되어야 함

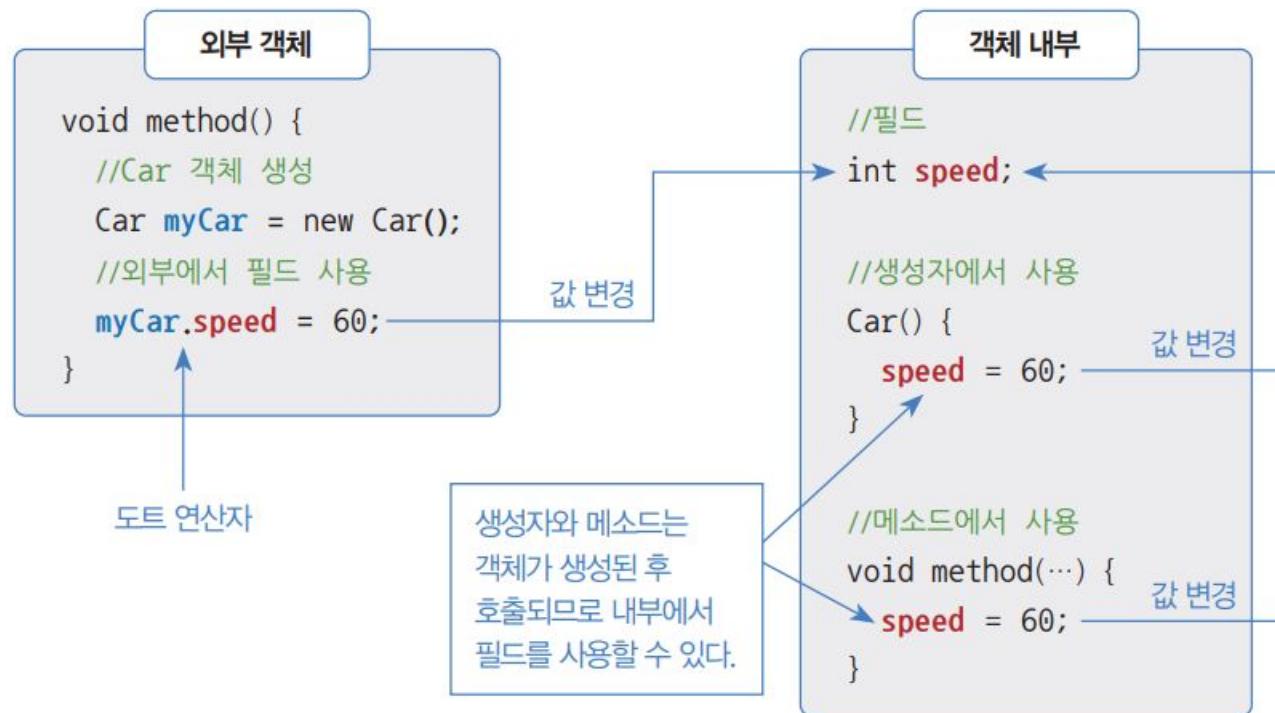
```
타입 필드명 [ = 초기값] ;
```

- 타입은 필드에 저장할 데이터의 종류를 결정. 기본 타입, 참조 타입 모두 가능
- 초기값을 제공하지 않을 경우 필드는 객체 생성 시 자동으로 기본값으로 초기화

분류		데이터 타입	기본값
기본 타입	정수 타입	byte	0
		char	\u0000 (빈 공백)
		short	0
		int	0
		long	0L
	실수 타입	float	0.0F
		double	0.0
	논리 타입	boolean	false
참조 타입		배열	null
		클래스(String 포함)	null
		인터페이스	null

필드 사용

- 필드값을 읽고 변경하는 것. 클래스로부터 객체가 생성된 후에 필드를 사용할 수 있음
- 필드는 객체 내부의 생성자와 메소드 내부에서 사용할 수 있고, 객체 외부에서도 접근해서 사용할 수 있음



기본 생성자

- 클래스에 생성자 선언이 없으면 컴파일러는 기본 생성자를 바이트코드 파일에 자동으로 추가

[public] 클래스() { }

생성자 선언

- 객체를 다양하게 초기화하기 위해 생성자를 직접 선언할 수 있음

```
클래스(매개변수, ...) {  
    // 객체의 초기화 코드  
}  
} 생성자 블록
```

- 생성자는 메소드와 비슷한 모양을 가지고 있으나, 리턴 타입이 없고 클래스 이름과 동일
 - 매개변수의 타입은 매개값의 종류에 맞게 작성

필드 초기화

- 객체마다 동일한 값을 갖고 있다면 필드 선언 시 초기값을 대입하는 것이 좋고, 객체마다 다른 값을 가져야 한다면 생성자에서 필드를 초기화하는 것이 좋음

```
public class Korean {  
    //필드 선언  
    String nation = "대한민국";  
    String name;           ← 초기화  
    String ssn;           ← 초기화  
  
    //생성자 선언  
    public Korean(String n, String s) {  
        name = n;           ←  
        ssn = s;            ←  
    }  
}
```

매개값으로 받은 이름과 주민등록번호를
필드 초기값으로 사용

생성자 오버로딩

- 매개변수를 달리하는 생성자를 여러 개 선언하는 것

```
public class Car {  
    Car() { … }  
    Car(String model) { … }  
    Car(String model, String color) { … }  
    Car(String model, String color, int maxSpeed) { … }  
}
```

[생성자 오버로딩]

매개변수의 타입, 개수, 순서가
다르게 여러 개의 생성자 선언

- 매개변수의 타입, 개수, 선언된 순서가 똑같을 경우 매개변수 이름만 바꾸는 것은 생성자 오버로딩이 아님
- 생성자가 오버로딩되어 있을 경우, new 연산자로 생성자를 호출할 때 제공되는 매개값의 타입과 수에 따라 실행될 생성자가 결정

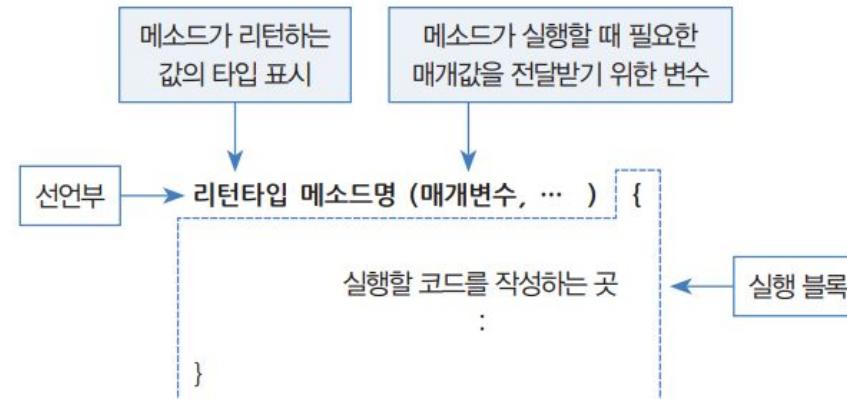
다른 생성자 호출

- 생성자 오버로딩이 많아질 경우 생성자 간의 중복된 코드가 발생할 수 있음
 - 이 경우 공통 코드를 한 생성자에만 집중적으로 작성하고, 나머지 생성자는 `this (...)`를 사용해 공통 코드를 가진 생성자를 호출

```
Car(String model) {  
    this(model, "은색", 250); }  
  
Car(String model, String color) {  
    this(model, color, 250); }  
  
Car(String model, String color, int maxSpeed) {  
    this.model = model;  
    this.color = color;  
    this.maxSpeed = maxSpeed; }  
}  
  
호출  
호출  
←  
공통 초기화 코드
```

메소드 선언

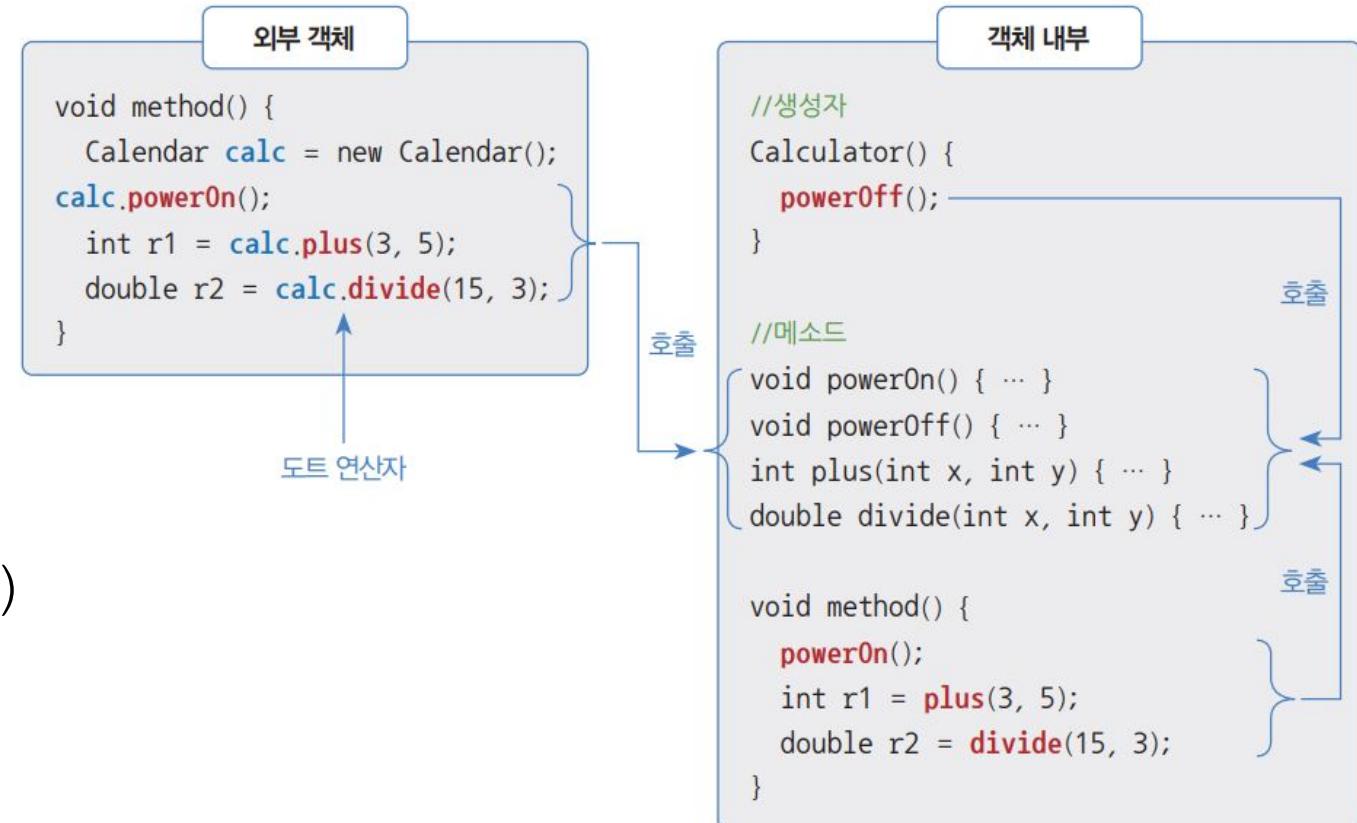
- 객체의 동작을 실행 블록으로 정의하는 것.



- 리턴 타입: 메소드 실행 후 호출한 곳으로 전달하는 결과값의 타입
- 메소드명: 메소드명은 첫 문자를 소문자로 시작하고, 캐멀 스타일로 작성
- 매개변수: 메소드를 호출할 때 전달한 매개값을 받기 위해 사용
- 실행 블록: 메소드 호출 시 실행되는 부분

메소드 호출

- 메소드 블록을 실제로 실행하는 것
- 클래스로부터 객체가 생성된 후에
메소드는 생성자와 다른 메소드
내부에서 호출될 수 있고, 객체
외부에서도 호출될 수 있음
- 외부 객체에서는 참조 변수와 도트(.)
연산자로 호출



가변길이 매개변수

- 메소드가 가변길이 매개변수를 가지고 있다면 매개변수의 개수와 상관없이 매개값을 줄 수 있음

```
int sum(int ... values) {  
}
```

- 메소드 호출 시 매개값을 쉼표로 구분해서 개수와 상관없이 제공할 수 있음
- 매개값들은 자동으로 배열 항목으로 변환되어 메소드에서 사용됨

```
int[] values = { 1, 2, 3 };  
int result = sum(values);
```

```
int result = sum(new int[] { 1, 2, 3 });
```

return 문

- 메소드의 실행을 강제 종료하고 호출한 곳으로 돌아간다는 의미
- 메소드 선언에 리턴 타입이 있을 경우에는 return 문 뒤에 리턴값을 추가로 지정해야 함

```
return [리턴값];
```

- return 문 이후에 실행문을 작성하면 ‘Unreachable code’라는 컴파일 에러가 발생

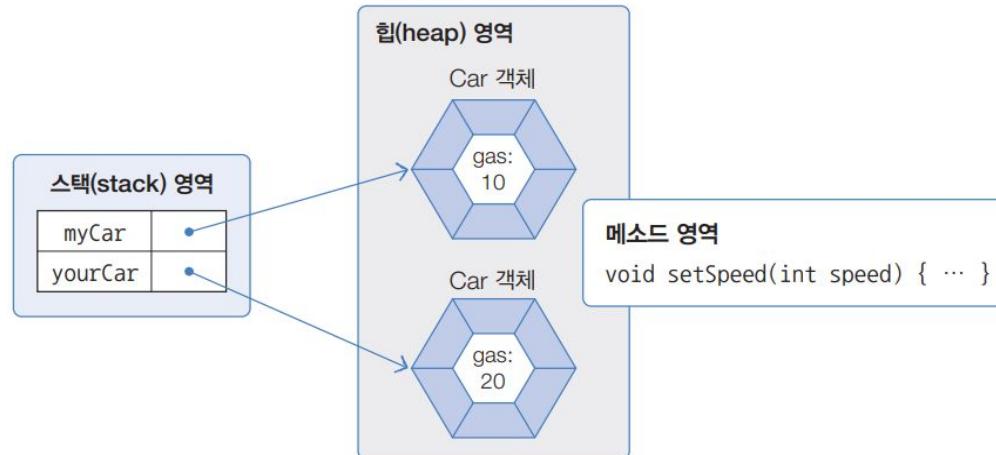
메소드 오버로딩

- 메소드 이름은 같되 매개변수의 타입, 개수, 순서가 다른 메소드를 여러 개 선언하는 것

```
class 클래스 {  
    리턴타입 메소드이름 ( 타입 변수, … ) { … }  
    ↑  
    무관  
    ↓  
    리턴타입 메소드이름 ( 타입 변수, … ) { … }  
}  
동일  
타입, 개수, 순서가 달라야 함
```

인스턴스 멤버 선언 및 사용

- 인스턴스 멤버: 필드와 메소드 등 객체에 소속된 멤버

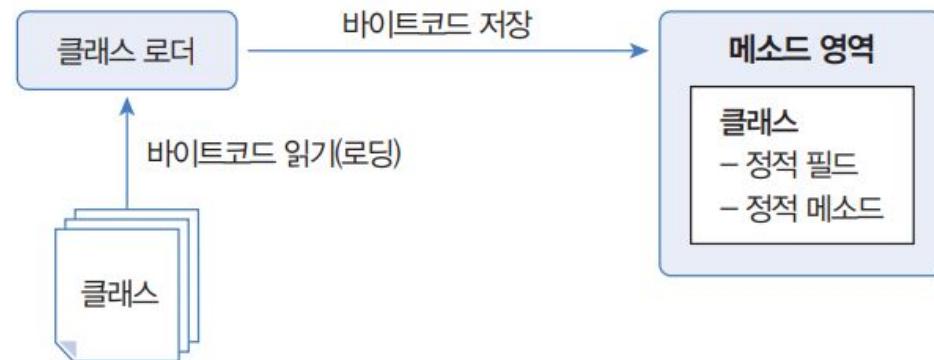


this 키워드

- 객체 내부에서는 인스턴스 멤버에 접근하기 위해 this를 사용. 객체는 자신을 ‘this’라고 지칭
- 생성자와 메소드의 매개변수명이 인스턴스 멤버인 필드명과 동일한 경우, 인스턴스 필드임을 강조하고자 할 때 this를 주로 사용

정적 멤버 선언

- 정적 멤버: 메소드 영역의 클래스에 고정적으로 위치하는 멤버



- static 키워드를 추가해 정적 필드와 정적 메소드로 선언

```
public class 클래스 {  
    //정적 필드 선언  
    static 타입 필드 [= 초기값];  
  
    //정적 메소드  
    static 리턴타입 메소드( 매개변수, … ) { … }  
}
```

정적 멤버 사용

- 클래스가 메모리로 로딩되면 정적 멤버를 바로 사용할 수 있음
- 클래스 이름과 함께 도트(.) 연산자로 접근

```
public class Calculator {  
    static double pi = 3.14159;  
    static int plus(int x, int y) { ... }  
    static int minus(int x, int y) { ... }  
}
```

```
double result1 = 10 * 10 * Calculator.pi;  
int result2 = Calculator.plus(10, 5);  
int result3 = Calculator.minus(10, 5);
```

- 정적 필드와 정적 메소드는 객체 참조 변수로도 접근

정적 블록

- 정적 필드를 선언할 때 복잡한 초기화 작업이 필요하다면 정적 블록을 이용

```
static {  
    ...  
}
```

- 정적 블록은 클래스가 메모리로 로딩될 때 자동으로 실행
- 정적 블록이 클래스 내부에 여러 개가 선언되어 있을 경우에는 선언된 순서대로 실행
- 정적 필드는 객체 생성 없이도 사용할 수 있기 때문에 생성자에서 초기화 작업을 하지 않음

인스턴스 멤버 사용 불가

- 정적 메소드와 정적 블록은 내부에 인스턴스 필드나 인스턴스 메소드를 사용할 수 없으며 this도 사용할 수 없음
- 정적 메소드와 정적 블록에서 인스턴스 멤버를 사용하고 싶다면 객체를 먼저 생성하고 참조 변수로 접근

```
static void Method3() {  
    //객체 생성  
    ClassName obj = new ClassName();  
    //인스턴스 멤버 사용  
    obj.field1 = 10;  
    obj.method1();  
}
```

final 필드 선언

- final 필드는 초기값이 저장되면 최종적인 값이 되어서 프로그램 실행 도중에 수정할 수 없게 됨
- final 필드에 초기값을 주려면 필드 선언 시에 초기값을 대입하거나 생성자에서 초기값을 대입

```
final 타입 필드 [=초기값];
```

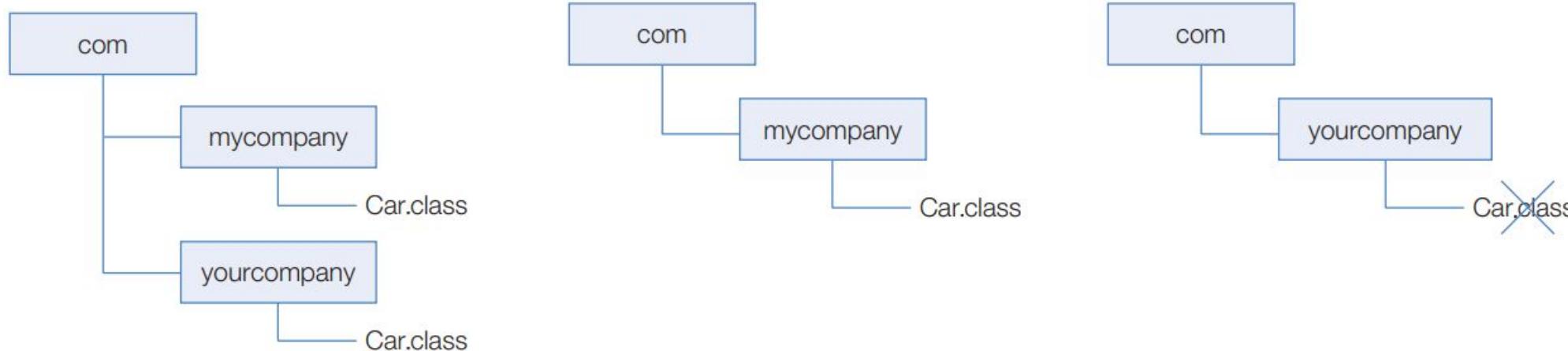
상수 선언

- 상수: 불변의 값을 저장하는 필드
- 상수는 객체마다 저장할 필요가 없고, 여러 개의 값을 가져도 안 되기 때문에 static이면서 final

```
static final 타입 상수 [= 초기값];
```

자바의 패키지

- 클래스의 일부분이며, 클래스를 식별하는 용도
- 패키지는 주로 개발 회사의 도메인 이름의 역순으로 만듦
- 상위 패키지와 하위 패키지를 도트(.)로 구분
- 패키지에 속한 바이트코드 파일(~.class)은 따로 떼어내어 다른 디렉토리로 이동할 수 없음



패키지 선언

- 패키지 선언은 package 키워드와 함께 패키지 이름을 기술한 것. 항상 소스 파일 최상단에 위치

```
package 상위패키지.하위패키지;
```

```
public class 클래스명 { … }
```

- 패키지 이름은 모두 소문자로 작성. 패키지 이름이 서로 중복되지 않도록 회사 도메인 이름의 역순으로 작성하고, 마지막에는 프로젝트 이름을 붙여줌

```
com.samsung.projectname  
com.lg.projectname  
org.apache.projectname
```

import문

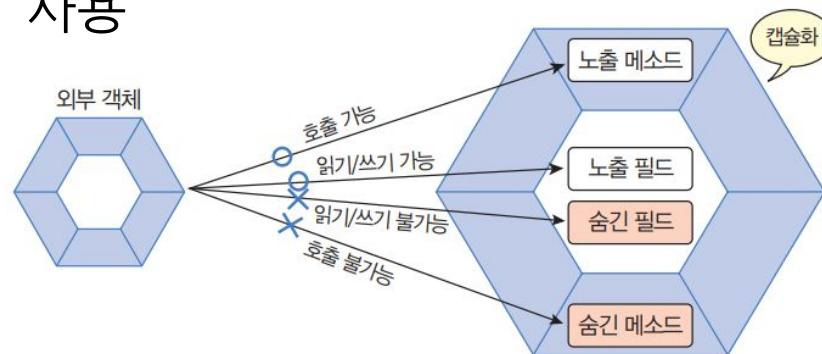
- 다른 패키지에 있는 클래스를 사용하려면 import 문을 이용해서 어떤 패키지의 클래스를 사용하는지 명시

```
package com.mycompany; •-----> Car 클래스의 패키지  
  
import com.hankook.Tire; •-----> Tire 클래스의 전체 이름  
  
public class Car {  
    //필드 선언 시 com.hankook.Tire 클래스를 사용  
    Tire tire = new Tire();  
}
```

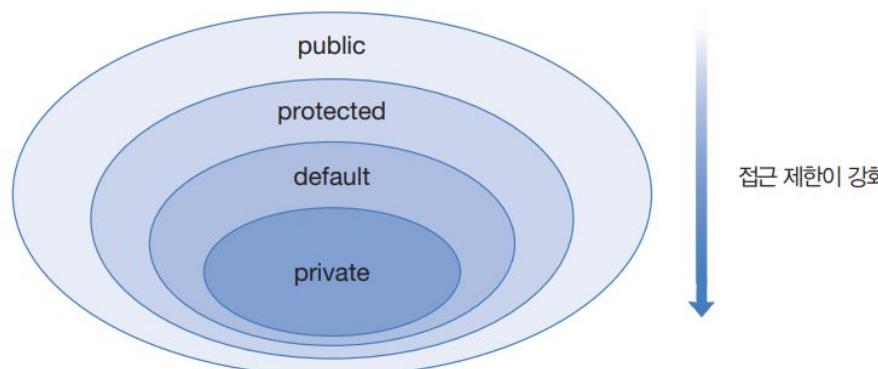
- import 문은 패키지 선언과 클래스 선언 사이에 작성. import 키워드 뒤에는 사용하고자 하는 클래스의 전체 이름을 기술

접근 제한자

- 중요한 필드와 메소드가 외부로 노출되지 않도록 해 객체의 무결성을 유지하기 위해서 접근 제한자 사용



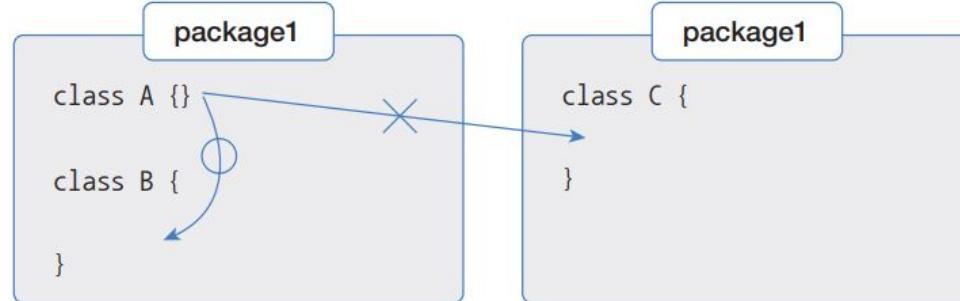
- 접근 제한자는 public, protected, private의 세 가지 종류



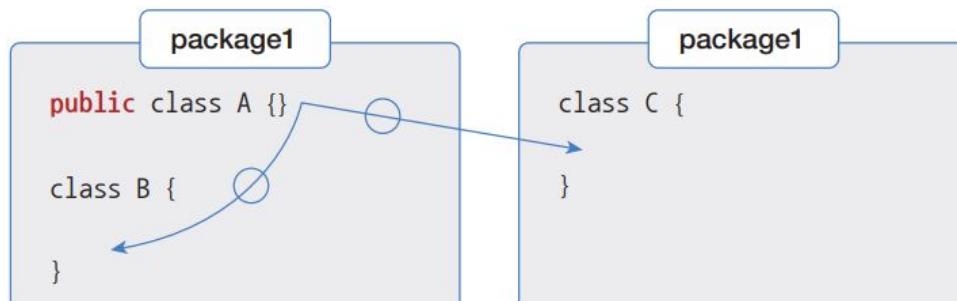
접근 제한자	제한 대상	제한 범위
public	클래스, 필드, 생성자, 메소드	없음
protected	필드, 생성자, 메소드	같은 패키지이거나, 자식 객체만 사용 가능 (7장 상속에서 자세히 설명)
(default)	클래스, 필드, 생성자, 메소드	같은 패키지
private	필드, 생성자, 메소드	객체 내부

클래스의 접근 제한

- 클래스를 선언할 때 public 접근 제한자를 생략하면 클래스는 다른 패키지에서 사용할 수 없음



- 클래스를 선언할 때 public 접근 제한자를 붙이면 클래스는 같은 패키지뿐만 아니라 다른 패키지에서도 사용할 수 있음



생성자의 접근 제한

- 생성자는 public, default, private 접근 제한을 가질 수 있음

```
public class ClassName {  
    //생성자 선언  
    [ public | private ] ClassName(…) { … }  
}
```

접근 제한자	생성자	설명
public	클래스(…)	모든 패키지에서 생성자를 호출할 수 있다. = 모든 패키지에서 객체를 생성할 수 있다.
	클래스(…)	같은 패키지에서만 생성자를 호출할 수 있다. = 같은 패키지에서만 객체를 생성할 수 있다.
private	클래스(…)	클래스 내부에서만 생성자를 호출할 수 있다. = 클래스 내부에서만 객체를 생성할 수 있다.

필드와 메소드의 접근 제한

- 필드와 메소드는 public, default, private 접근 제한을 가질 수 있음

//필드 선언

[public | private] 타입 필드;

//메소드 선언

[public | private] 리턴타입 메소드(…){ … }

접근 제한자	생성자	설명
public	필드 메소드(…)	모든 패키지에서 필드를 읽고 변경할 수 있다. 모든 패키지에서 메소드를 호출할 수 있다.
	필드 메소드(…)	같은 패키지에서만 필드를 읽고 변경할 수 있다. 같은 패키지에서만 메소드를 호출할 수 있다.
private	필드 메소드(…)	클래스 내부에서만 필드를 읽고 변경할 수 있다. 클래스 내부에서만 메소드를 호출할 수 있다.

Setter

- 데이터를 검증해서 유효한 값만 필드에 저장하는 메소드

Getter

- 필드값이 객체 외부에서 사용하기에 부적절한 경우, 적절한 값으로 변환해서 리턴할 수 있는 메소드

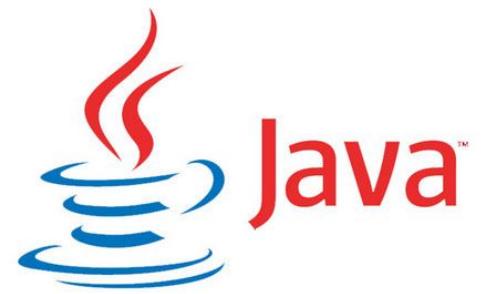
```
private 타입 fieldName; •----- 필드 접근 제한자: private  
  
//Getter  
public 타입 getFieldName() { •----- 접근 제한자: public  
    리턴 타입: 필드타입  
    메소드 이름: get + 필드이름(첫 글자 대문자)  
    리턴값: 필드값  
    }  
  
//Setter  
public void setFieldName(타입 fieldName) { •----- 접근 제한자: public  
    리턴 타입: void  
    메소드 이름: set + 필드이름(첫 글자 대문자)  
    매개변수 타입: 필드타입  
    }  
}
```

싱글톤 패턴

- 생성자를 private 접근 제한해서 외부에서 new 연산자로 생성자를 호출할 수 없도록 막아서 외부에서 마음대로 객체를 생성하지 못하게 함
- 대신 싱글톤 패턴이 제공하는 정적 메소드를 통해 간접적으로 객체를 얻을 수 있음

```
public class 클래스 {  
    //private 접근 권한을 갖는 정적 필드 선언과 초기화  
    private static 클래스 singleton = new 클래스(); •----- ①  
  
    //private 접근 권한을 갖는 생성자 선언  
    private 클래스() {}  
  
    //public 접근 권한을 갖는 정적 메소드 선언  
    public static 클래스 getInstance() { •----- ②  
        return singleton;  
    }  
}
```

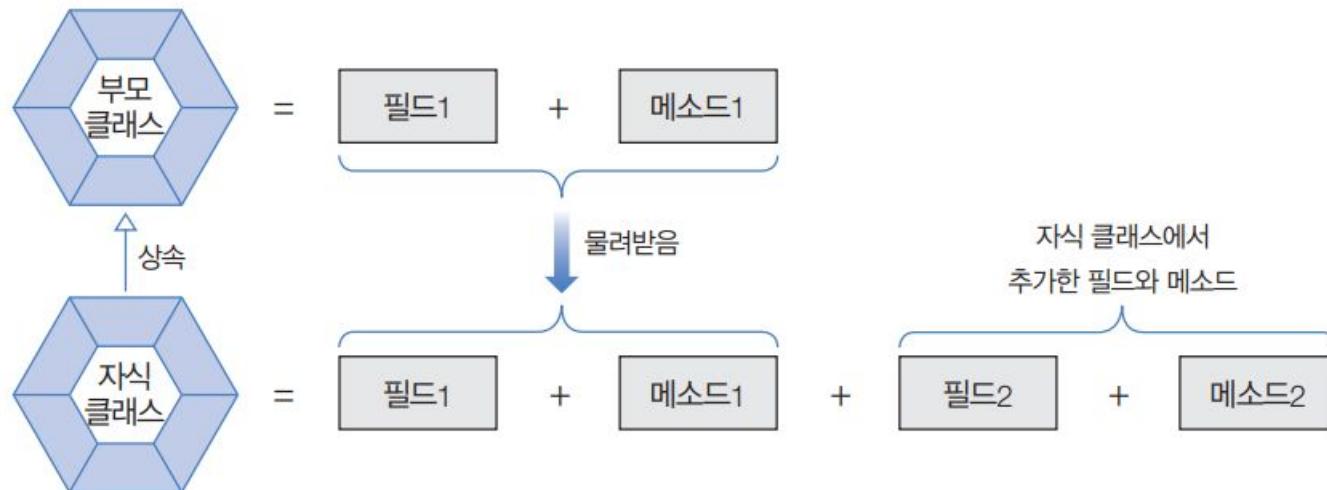




Chapter 07 상속

상속

- 부모 클래스의 필드와 메소드를 자식 클래스에게 물려줄 수 있음



상속의 이점

- 이미 개발된 클래스를 재사용하므로 중복 코드를 줄임
 - 클래스 수정을 최소화

클래스 상속

- 자식 클래스를 선언할 때 어떤 부모로부터 상속받을 것인지를 결정하고, 부모 클래스를 다음과 같이 extends 뒤에 기술

```
public class 자식클래스 extends 부모클래스 {  
}
```

- 다중 상속 허용하지 않음. extends 뒤에 하나의 부모 클래스만 상속

```
public class 자식클래스 extends 부모클래스1, 부모클래스2 {  
}
```

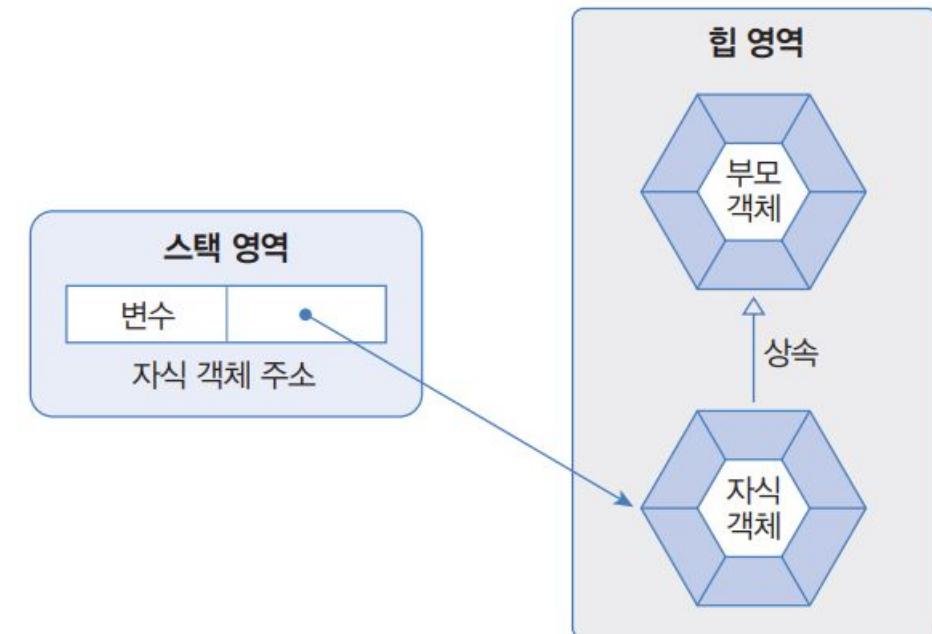
부모 생성자 호출

- 자식 객체를 생성하면 부모 객체가 먼저 생성된 다음에 자식 객체가 생성

```
자식클래스 변수 = new 자식클래스();
```

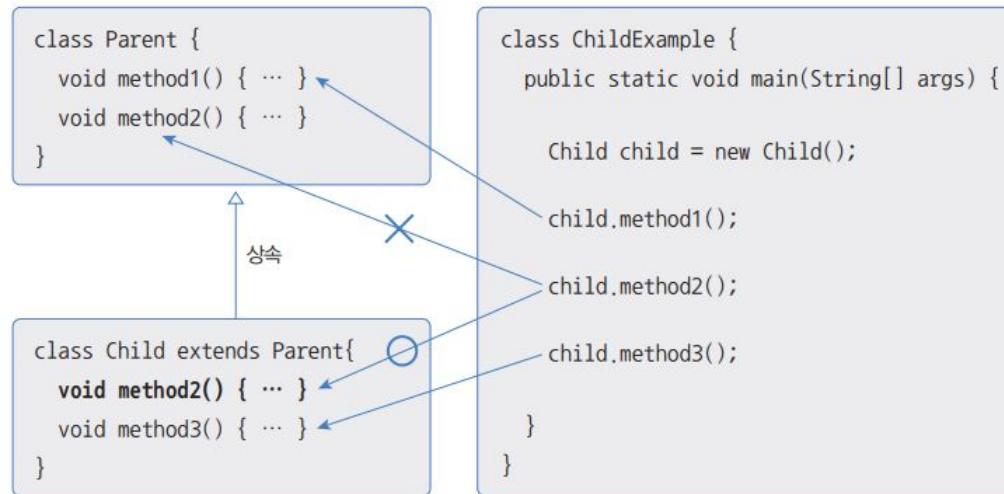
- 부모 생성자는 자식 생성자의 맨 첫 줄에 숨겨져 있는 `super()`에 의해 호출

```
//자식 생성자 선언  
public 자식클래스(...) {  
    super();  
    ...  
}
```



메소드 오버라이딩

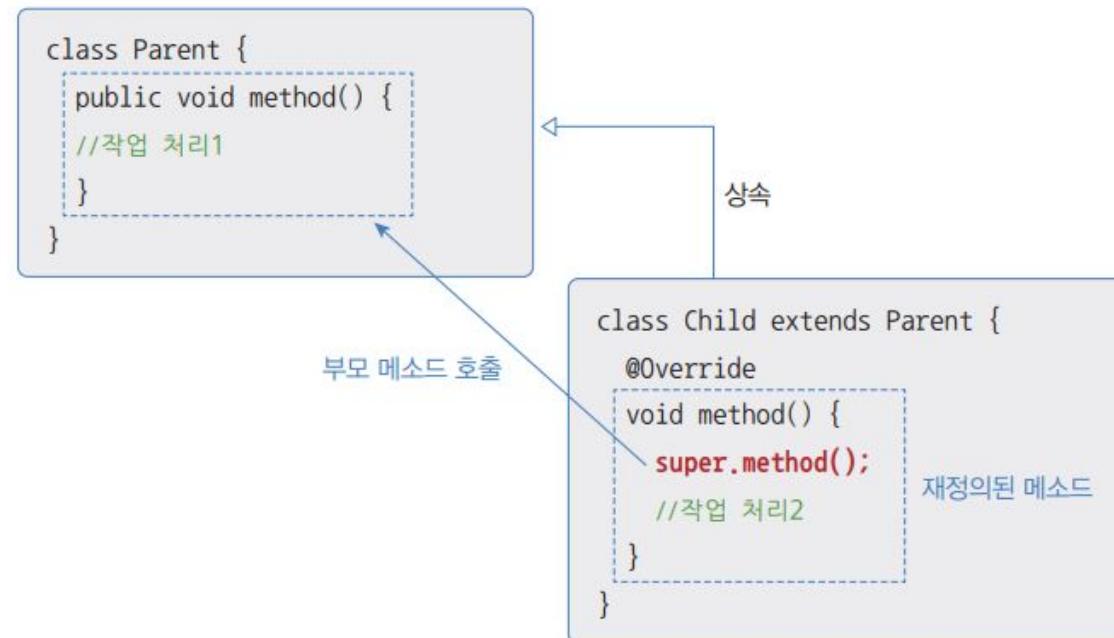
- 상속된 메소드를 자식 클래스에서 재정의하는 것. 해당 부모 메소드는 숨겨지고, 자식 메소드가 우선적으로 사용



- 부모 메소드의 선언부(리턴 타입, 메소드 이름, 매개변수)와 동일해야 함
- 접근 제한을 더 강하게 오버라이딩할 수 없음(public → private으로 변경 불가)
- 새로운 예외를 throws 할 수 없음

부모 메소드 호출

- 자식 메소드 내에서 super 키워드와 도트(.) 연산자를 사용하면 숨겨진 부모 메소드를 호출
- 부모 메소드를 재사용함으로써 자식 메소드의 중복 작업 내용을 없애는 효과



final 클래스

- final 클래스는 부모 클래스가 될 수 없어 자식 클래스를 만들 수 없음

```
public final class 클래스 { ... }
```

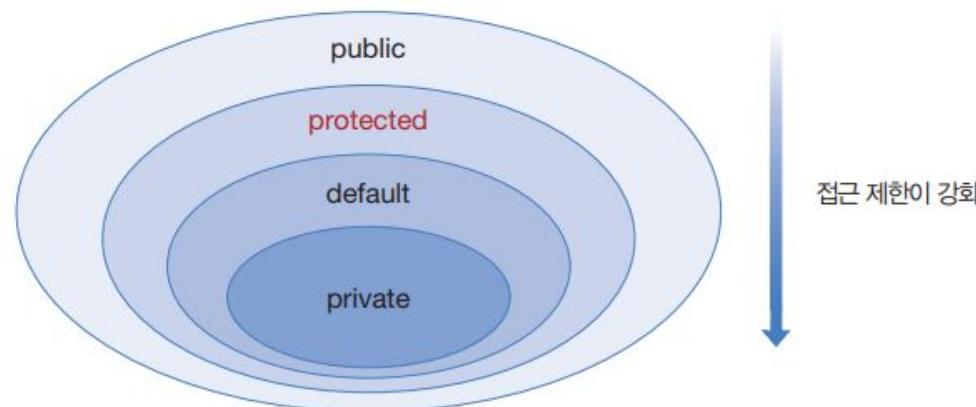
final 메소드

- 메소드를 선언할 때 final 키워드를 붙이면 오버라이딩할 수 없음
- 부모 클래스를 상속해서 자식 클래스를 선언할 때, 부모 클래스에 선언된 final 메소드는 자식 클래스에서 재정의할 수 없음

```
public final 리턴타입 메소드( 매개변수, ... ) { ... }
```

protected 접근 제한자

- `protected`는 상속과 관련이 있고, `public`과 `default`의 중간쯤에 해당하는 접근 제한
- `protected`는 같은 패키지에서는 `default`처럼 접근이 가능하나, 다른 패키지에서는 자식 클래스만 접근을 허용



NOTE ▶ `default`는 접근 제한자가 아니라 접근 제한자가 붙지 않은 상태를 말한다.

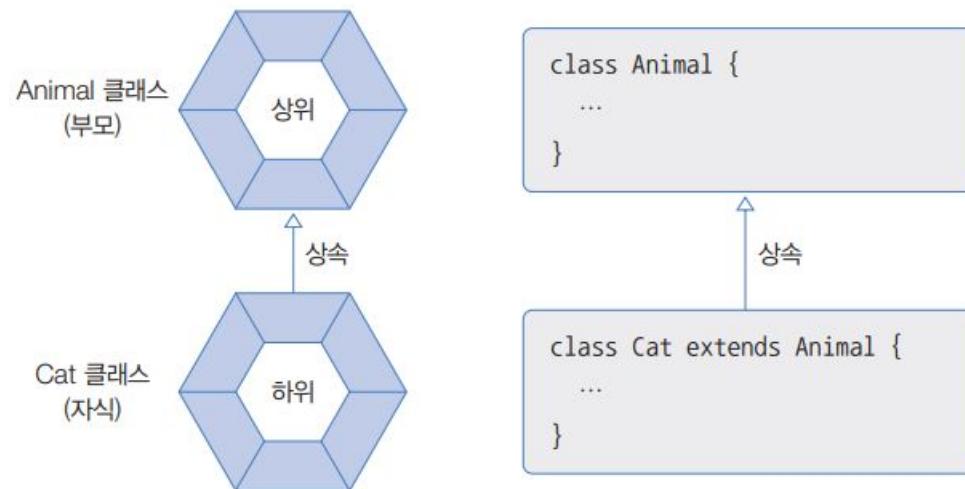
접근 제한자	제한 대상	제한 범위
<code>protected</code>	필드, 생성자, 메소드	같은 패키지이거나, 자식 객체만 사용 가능

자동 타입 변환

- 자동적으로 타입 변환이 일어나는 것

자동 타입 변환
부모타입 변수 = 자식타입객체;

- 자식은 부모의 특징과 기능을 상속받기 때문에 부모와 동일하게 취급



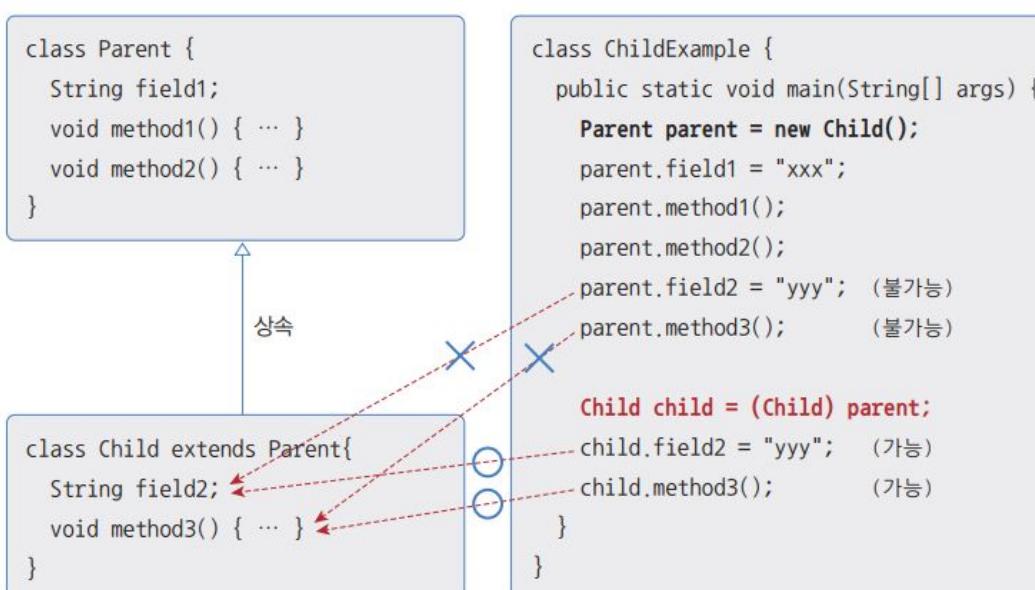
강제 타입 변환

- 부모 타입은 자식 타입으로 자동 변환되지 않음. 대신 캐스팅 연산자로 강제 타입 변환 가능

강제 타입 변환

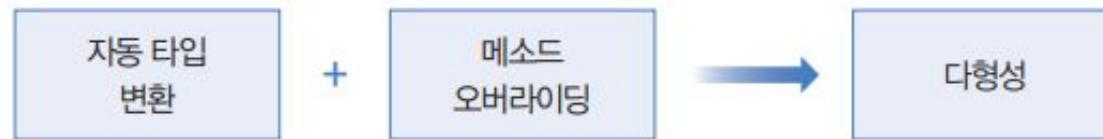
```
자식타입 변수 = (자식타입) 부모타입객체;
캐스팅 연산자
```

- 자식 객체가 부모 타입으로 자동 변환하면 부모 타입에 선언된 필드와 메소드만 사용 가능



다형성

- 사용 방법은 동일하지만 실행 결과가 다양하게 나오는 성질
- 다형성을 구현하기 위해서는 자동 타입 변환과 메소드 재정의가 필요



필드 다형성

- 필드 타입은 동일하지만, 대입되는 객체가 달라져서 실행 결과가 다양하게 나올 수 있는 것

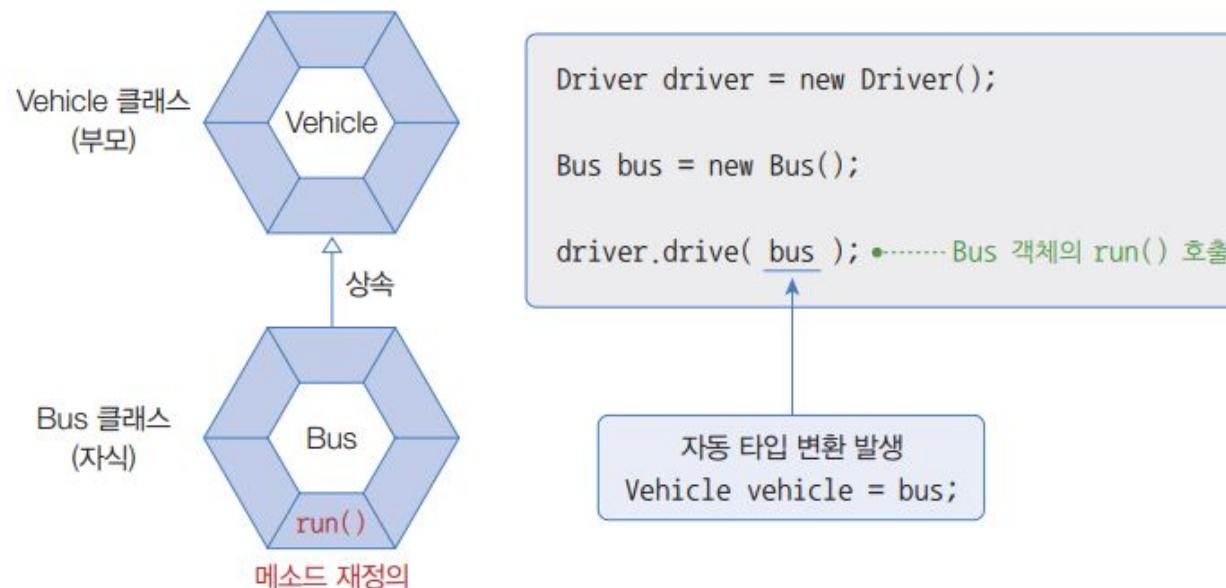
```

//Car 객체 생성
Car myCar = new Car();
//HankookTire 장착
myCar.tire = new HankookTire();
//KumhoTire 장착
myCar.tire = new KumhoTire();
  
```



매개변수 다형성

- 메소드가 클래스 타입의 매개변수를 가지고 있을 경우, 호출할 때 동일한 타입의 자식 객체를 제공할 수 있음
- 어떤 자식 객체가 제공되느냐에 따라서 메소드의 실행 결과가 달라짐



instanceof 연산자

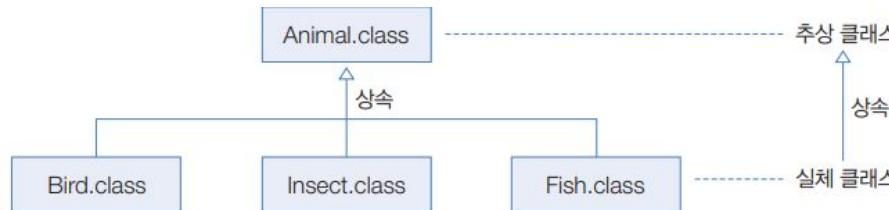
- 매개변수가 아니더라도 변수가 참조하는 객체의 타입을 확인할 때 instanceof 연산자를 사용
- instanceof 연산자에서 좌항의 객체가 우항의 타입이면 true를 산출하고 그렇지 않으면 false를 산출

```
boolean result = 객체 instanceof 타입;
```

- Java 12부터는 instanceof 연산의 결과가 true일 경우 우측 타입 변수를 사용할 수 있기 때문에 강제 타입 변환이 필요 없음

추상 클래스

- 객체를 생성할 수 있는 실제 클래스들의 공통적인 필드나 메소드를 추출해서 선언한 클래스
- 추상 클래스는 실제 클래스의 부모 역할. 공통적인 필드나 메소드를 물려받을 수 있음



추상 클래스 선언

- 클래스 선언에 `abstract` 키워드를 붙임
- `new` 연산자를 이용해서 객체를 직접 만들지 못하고 상속을 통해 자식 클래스만 만들 수 있다.

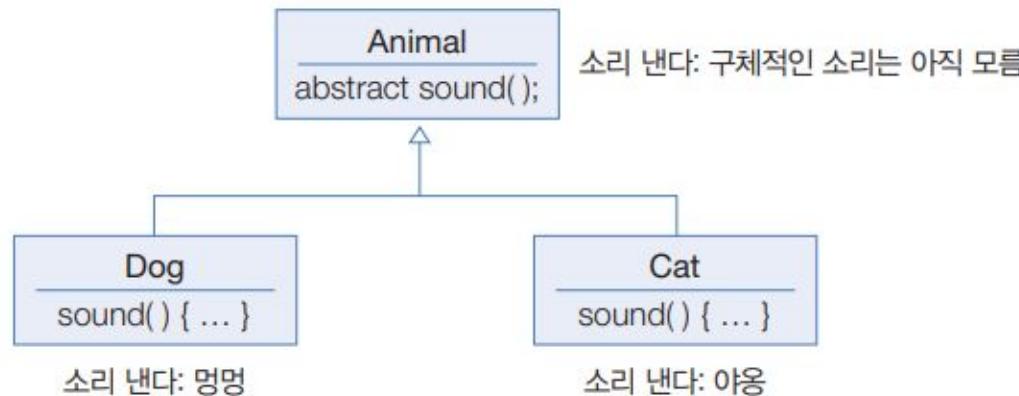
```

public abstract class 클래스명 {
    //필드
    //생성자
    //메소드
}
  
```

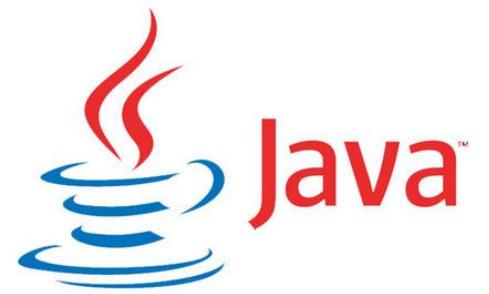
추상 메소드와 재정의

- 자식 클래스들이 가지고 있는 공통 메소드를 뽑아내어 추상 클래스로 작성할 때, 메소드 선언부만 동일하고 실행 내용은 자식 클래스마다 달라야 하는 경우 추상 메소드를 선언할 수 있음
- 일반 메소드 선언과의 차이점은 `abstract` 키워드가 붙고, 메소드 실행 내용인 중괄호 {}가 없다.

abstract 리턴타입 메소드명(매개변수, …);



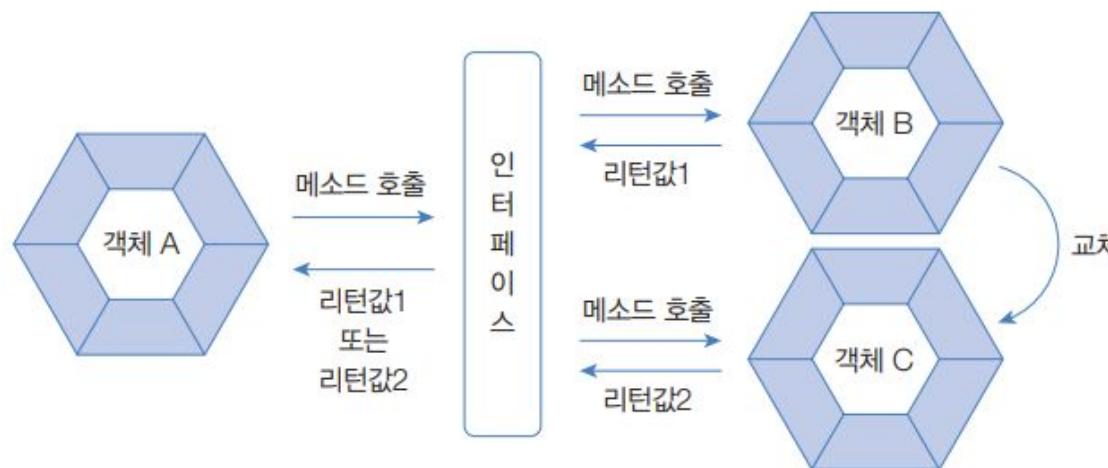
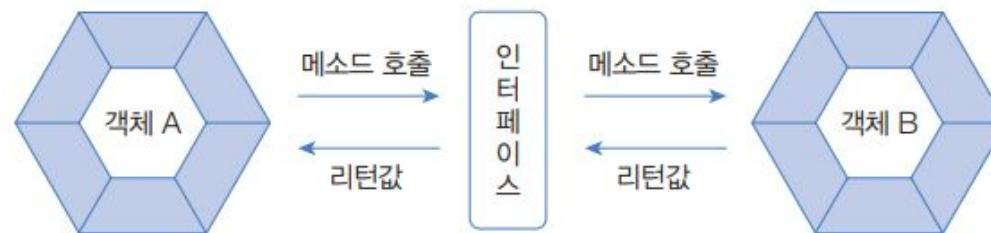




Chapter 08 인터페이스

인터페이스

- 두 객체를 연결하는 역할
- 다형성 구현에 주된 기술



인터페이스 선언

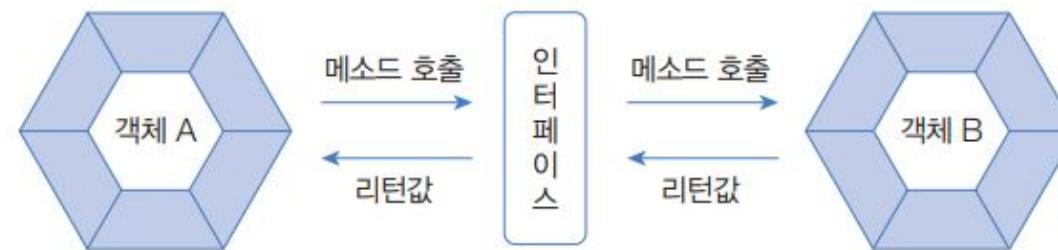
- 인터페이스 선언은 class 키워드 대신 interface 키워드를 사용
- 접근 제한자로는 클래스와 마찬가지로 같은 패키지 내에서만 사용 가능한 default, 패키지와 상관없이 사용하는 public을 붙일 수 있음

```
interface 인터페이스명 { … }           //default 접근 제한
public interface 인터페이스명 { … }    //public 접근 제한
```

```
public interface 인터페이스명 {
    //public 상수 필드
    //public 추상 메소드
    //public 디폴트 메소드
    //public 정적 메소드
    //private 메소드
    //private 정적 메소드
}
```

구현 클래스 선언

- 인터페이스에 정의된 추상 메소드에 대한 실행 내용이 구현



- implements 키워드는 해당 클래스가 인터페이스를 통해 사용할 수 있다는 표시이며, 인터페이스의 추상 메소드를 재정의한 메소드가 있다는 뜻

```
public class B implements 인터페이스명 { ... }
```

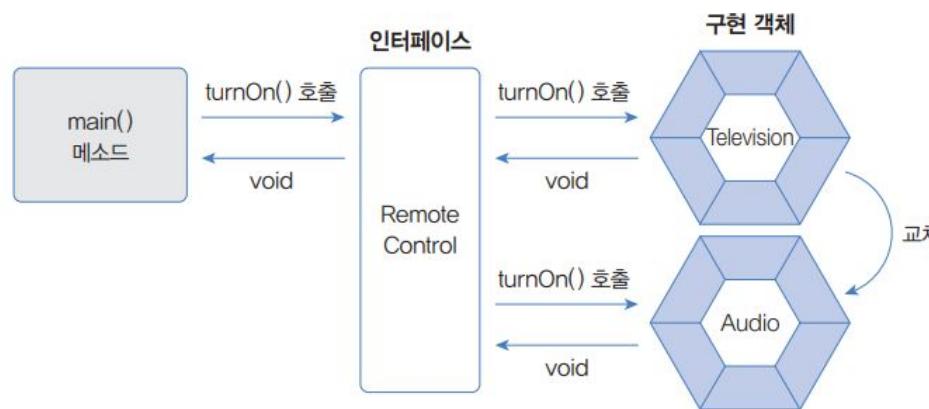
변수 선언과 구현 객체 대입

- 인터페이스는 참조 타입에 속하므로 인터페이스 변수에는 객체를 참조하고 있지 않다는 뜻으로 null을 대입할 수 있음

```
RemoteControl rc;  
RemoteControl rc = null;
```

- 인터페이스를 통해 구현 객체를 사용하려면, 인터페이스 변수에 구현 객체의 번지를 대입해야 함

```
rc = new Television();
```



상수 필드

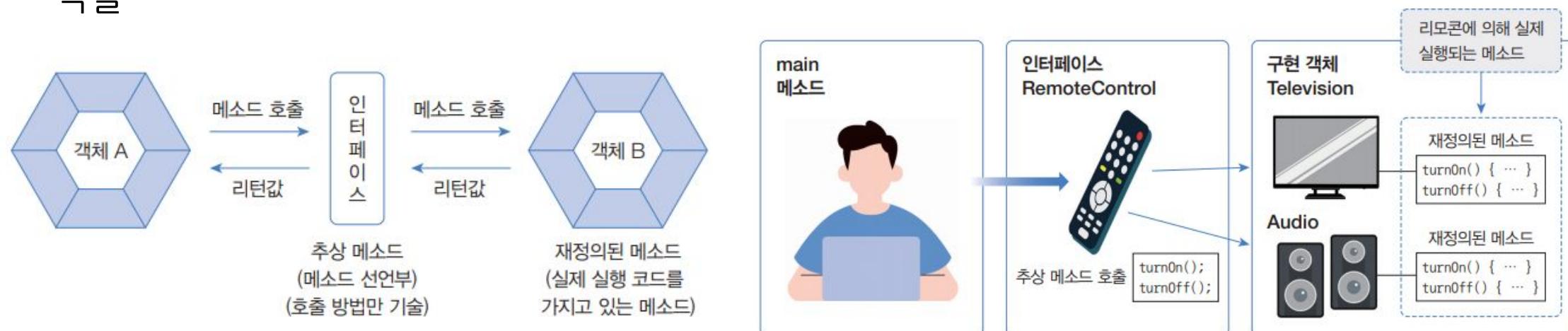
- 인터페이스는 public static final 특성을 갖는 불변의 상수 필드를 멤버로 가질 수 있음

```
[ public static final ] 타입 상수명 = 값;
```

- 인터페이스에 선언된 필드는 모두 public static final 특성을 갖기 때문에 public static final을 생략해도 자동으로 컴파일 과정에서 붙음
- 상수명은 대문자로 작성하되, 서로 다른 단어로 구성되어 있을 경우에는 언더바(_)로 연결

추상 메소드

- 리턴 타입, 메소드명, 매개변수만 기술되고 중괄호 {}를 붙이지 않는 메소드
- public abstract를 생략하더라도 컴파일 과정에서 자동으로 붙음
- 추상 메소드는 객체 A가 인터페이스를 통해 어떻게 메소드를 호출할 수 있는지 방법을 알려주는 역할



디폴트 메소드

- 인터페이스에는 완전한 실행 코드를 가진 디폴트 메소드를 선언할 수 있음
- 추상 메소드는 실행부(중괄호 {})가 없지만 디폴트 메소드는 실행부 있음. default 키워드가 리턴 타입 앞에 붙음

```
[public] default 리턴타입 메소드명(매개변수, ...) { ... }
```

- 디폴트 메소드의 실행부에는 상수 필드를 읽거나 추상 메소드를 호출하는 코드를 작성할 수 있음

정적 메소드

- 구현 객체가 없어도 인터페이스만으로 호출할 수 있음
- 선언 시 public을 생략하더라도 자동으로 컴파일 과정에서 붙음

```
[public | private] static 리턴타입 메소드명(매개변수, ...) { ... }
```

- 정적 실행부를 작성할 때 상수 필드를 제외한 추상 메소드, 디폴트 메소드, private 메소드 등을 호출할 수 없음

private 메소드

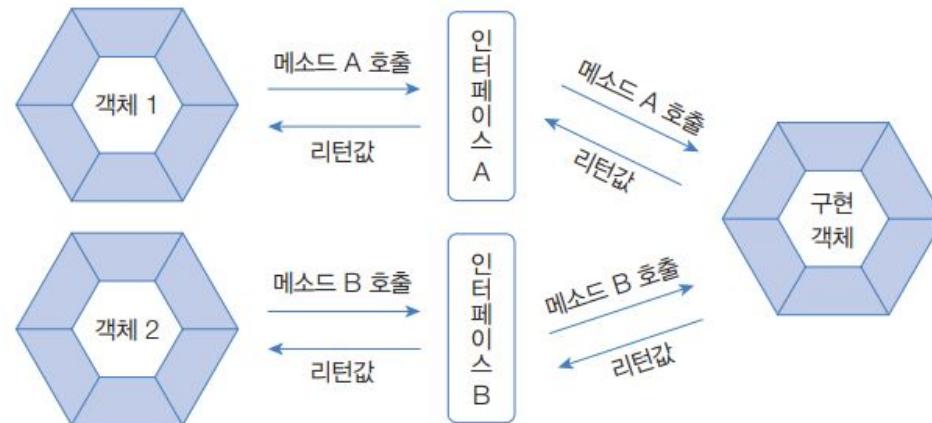
- 인터페이스의 상수 필드, 추상 메소드, 디폴트 메소드, 정적 메소드는 모두 public 접근 제한을 가짐
public을 생략하더라도 항상 외부에서 접근이 가능
- 인터페이스에 외부에서 접근할 수 없는 private 메소드 선언도 가능

구분	설명
private 메소드	구현 객체가 필요한 메소드
private 정적 메소드	구현 객체가 필요 없는 메소드

- private 메소드는 디폴트 메소드 안에서만 호출이 가능
- private 정적 메소드는 정적 메소드 안에서도 호출이 가능

다중 인터페이스

- 구현 객체는 여러 개의 인터페이스를 통해 구현 객체를 사용할 수 있음



- 구현 클래스는 인터페이스 A와 인터페이스 B를 implements 뒤에 쉼표로 구분해서 작성해, 모든 인터페이스가 가진 추상 메소드를 재정의

```
public class 구현클래스명 implements 인터페이스A, 인터페이스B {  
    //모든 추상 메소드 재정의  
}
```

인터페이스 상속

- 인터페이스도 다른 인터페이스를 상속할 수 있음. 다중 상속을 허용
- extends 키워드 뒤에 상속할 인터페이스들을 나열

```
public interface 자식인터페이스 extends 부모인터페이스1, 부모인터페이스2 { ... }
```

- 자식 인터페이스의 구현 클래스는 자식 인터페이스의 메소드뿐만 아니라 부모 인터페이스의 모든 추상 메소드를 재정의
- 구현 객체는 다음과 같이 자식 및 부모 인터페이스 변수에 대입될 수 있음

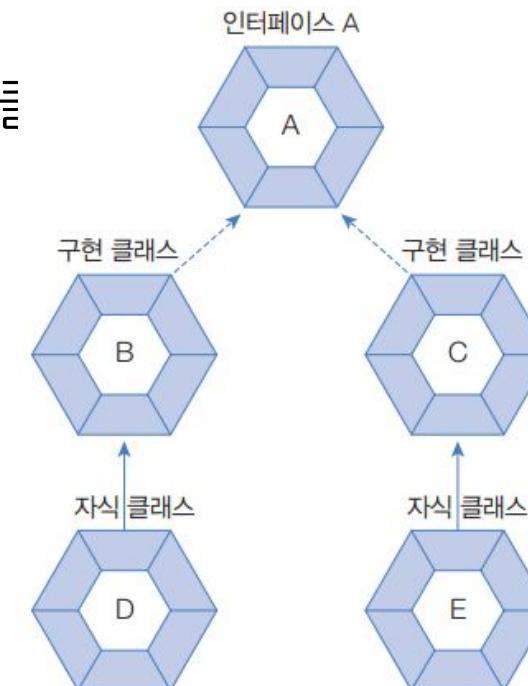
```
자식인터페이스 변수 = new 구현클래스(...);  
부모인터페이스1 변수 = new 구현클래스(...);  
부모인터페이스2 변수 = new 구현클래스(...);
```

자동 타입 변환

- 자동으로 타입 변환이 일어나는 것

자동 타입 변환
 ↓
 인터페이스 변수 = 구현객체;

- 부모 클래스가 인터페이스를 구현하고 있다면 자식 클래스가 그 인터페이스를 구현할 수 있음



변환될

```
B b = new B();
C c = new C();
D d = new D();
E e = new E();
```

↓

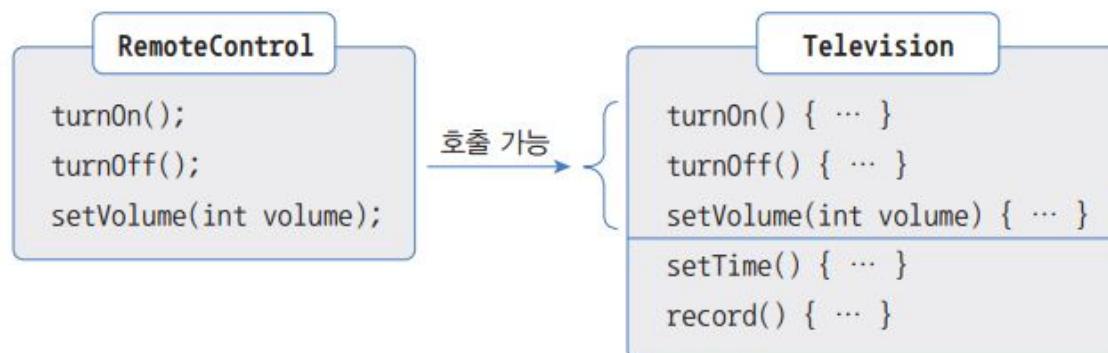
```
A a;
a = b; (가능)
a = c; (가능)
a = d; (가능)
a = e; (가능)
```

강제 타입 변환

- 캐스팅 기호를 사용해서 인터페이스 타입을 구현 클래스 타입으로 변환시키는 것

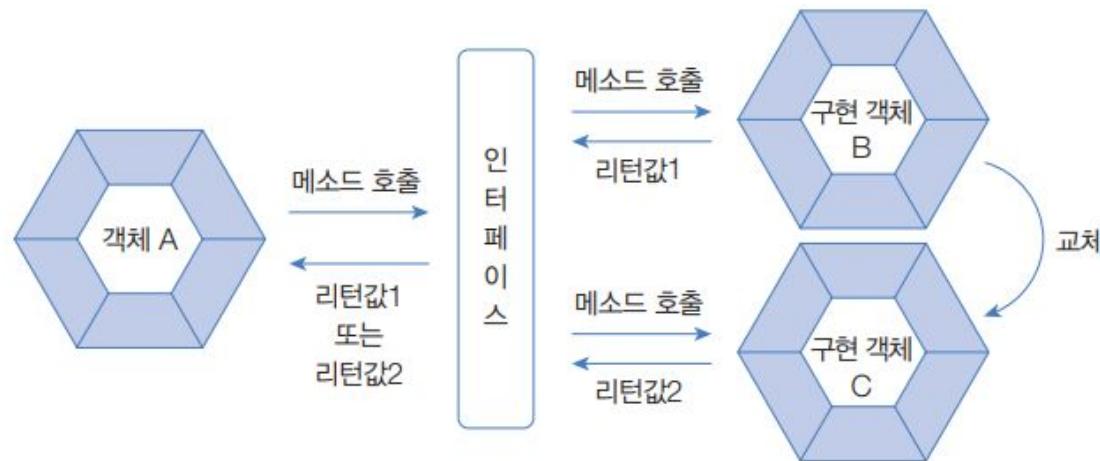
자동 타입 변환
구현클래스 변수 = (구현클래스) 인터페이스변수;

- 구현 객체가 인터페이스 타입으로 자동 변환되면, 인터페이스에 선언된 메소드만 사용 가능



다형성

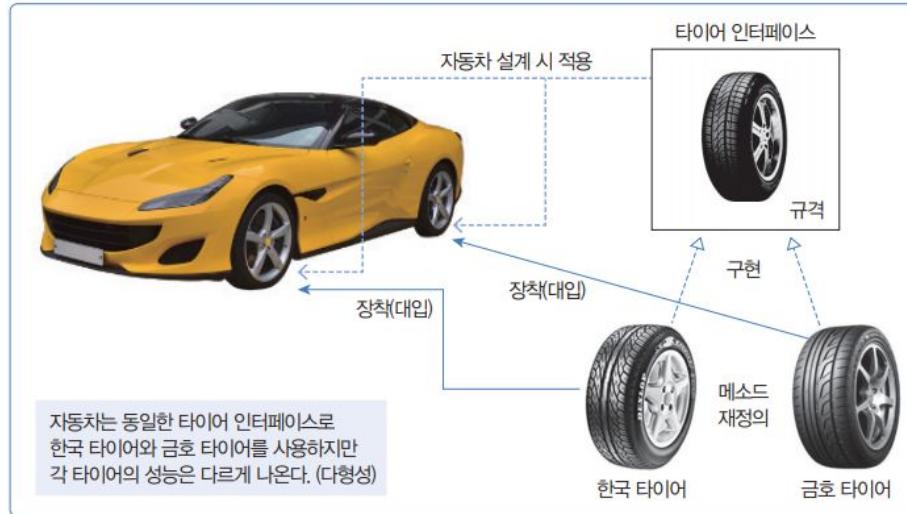
- 사용 방법은 동일하지만 다양한 결과가 나오는 성질



- 인터페이스 역시 다형성을 구현하기 위해 재정의와 자동 타입 변환 기능을 이용

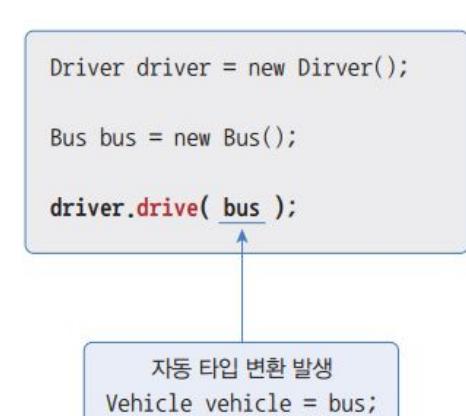
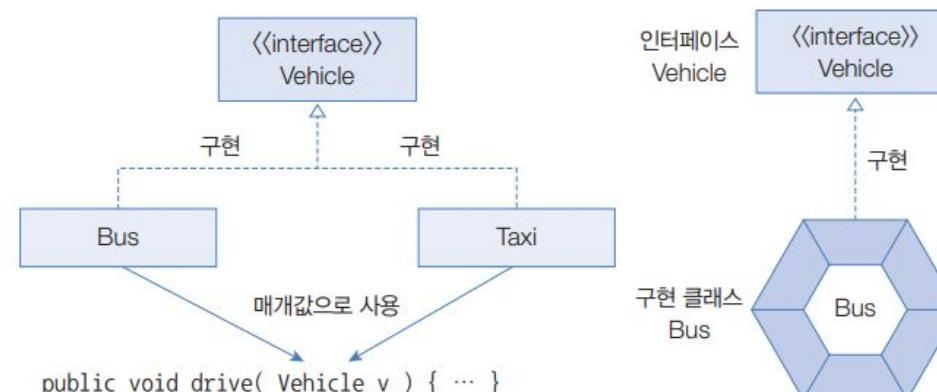


필드의 다형성



매개변수의 다형성

- 매개변수 타입을 인터페이스로 선언하면 메소드 호출 시 다양한 구현 객체를 대입할 수 있음



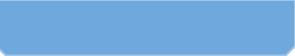
instanceof 연산자

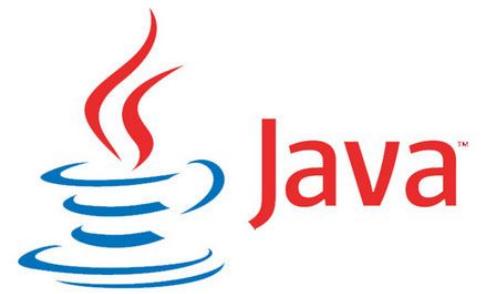
- 인터페이스에서도 객체 타입을 확인하기 위해 instanceof 연산자를 사용 가능

```
if( vehicle instanceof Bus ) {  
    //vehicle에 대입된 객체가 Bus일 경우 실행  
}
```

- Java 12부터는 instanceof 연산의 결과가 true일 경우, 우측 타입 변수를 사용할 수 있기 때문에 강제 타입 변환이 필요 없음

```
if(vehicle instanceof Bus bus) {  
    //bus 변수 사용  
}
```





Chapter 09 중첩 선언과 익명 객체

중첩 클래스

- 클래스 내부에 선언한 클래스. 클래스의 멤버를 쉽게 사용할 수 있고 외부에는 중첩 관계 클래스를 감춤으로써 코드의 복잡성을 줄일 수 있음
- 멤버 클래스: 클래스의 멤버로서 선언되는 중첩 클래스
- 로컬 클래스: 메소드 내부에서 선언되는 중첩 클래스

선언 위치에 따른 분류	선언 위치	객체 생성 조건
멤버 클래스	인스턴스 멤버 클래스	<pre>class A { class B { ... } }</pre> A 객체를 생성해야만 B 객체를 생성할 수 있음
	정적 멤버 클래스	<pre>class A { static class B { ... } }</pre> A 객체를 생성하지 않아도 B 객체를 생성할 수 있음
로컬 클래스	<pre>class A { void method() { class B { ... } } }</pre>	method가 실행할 때만 B 객체를 생성할 수 있음

인스턴스 멤버 클래스

- A 클래스의 멤버로 선언된 B 클래스

```
[public] class A {  
    [public | private] class B {  
    }  
}
```



구분	접근 범위
public class B {}	다른 패키지에서 B 클래스를 사용할 수 있다.
class B {}	같은 패키지에서만 B 클래스를 사용할 수 있다.
private class B {}	A 클래스 내부에서만 B 클래스를 사용할 수 있다.

- 인스턴스 멤버 클래스 B는 주로 A 클래스 내부에서 사용되므로 private 접근 제한을 갖는 것이 일반적

정적 멤버 클래스

- static 키워드와 함께 A 클래스의 멤버로 선언된 B 클래스

```
[public] class A {  
    [public | private] static class B {  
    }  
}
```



구분	접근 범위
public static class B {}	다른 패키지에서 B 클래스를 사용할 수 있다.
static class B {}	같은 패키지에서만 B 클래스를 사용할 수 있다.
private static class B {}	A 클래스 내부에서만 B 클래스를 사용할 수 있다.

- 정적 멤버 클래스는 주로 default 또는 public 접근 제한을 가진다.

로컬 클래스

- 생성자 또는 메소드 내부에서 다음과 같이 선언된 클래스
- 생성자와 메소드가 실행될 동안에만 객체를 생성할 수 있음

```
[public] class A {  
    //생성자  
    public A() {  
        class B { }  
    }  
  
    //메소드  
    public void method() {  
        class B { }  
    }  
}
```

The diagram illustrates two examples of local classes in Java. In the first example, a local class 'B' is defined within the constructor 'A()' of class 'A'. In the second example, a local class 'B' is defined within the method 'method()' of class 'A'. Both local class definitions are highlighted with dashed boxes. Arrows point from these boxes to a callout box labeled '로컬 클래스' (Local Class).

바깥 클래스의 멤버 접근 제한

- 정적 멤버 클래스 내부에서는 바깥 클래스의 필드와 메소드를 사용할 때 제한이 따름

구분	바깥 클래스의 사용 가능한 멤버
인스턴스 멤버 클래스	바깥 클래스의 모든 필드와 메소드
정적 멤버 클래스	바깥 클래스의 정적 필드와 정적 메소드

- 정적 멤버 클래스는 바깥 객체가 없어도 사용 가능해야 하므로 바깥 클래스의 인스턴스 필드와 인스턴스 메소드는 사용하지 못함

바깥 클래스의 객체 접근

- 중첩 클래스 내부에서 바깥 클래스의 객체를 얻으려면 바깥 클래스 이름에 this를 붙임

바깥클래스이름.this → 바깥객체

중첩 인터페이스

- 해당 클래스와 긴밀한 관계를 맺는 구현 객체를 만들기 위해 클래스의 멤버로 선언된 인터페이스

```
class A {  
    [public | private] [static] interface B {  
        //상수 필드  
        //추상 메소드  
        //디폴트 메소드  
        //정적 메소드  
    }  
}
```

중첩 인터페이스

- 안드로이드와 같은 UI 프로그램에서 이벤트를 처리할 목적으로 많이 활용

»> Button.java

```
1 package ch09.sec06.exam01;  
2  
3 public class Button {  
4     //정적 중첩 인터페이스  
5     public static interface ClickListener {  
6         //추상 메소드  
7         void onClick();  
8     }  
9 }
```

중첩 인터페이스 선언

익명 객체

- 이름이 없는 객체. 명시적으로 클래스를 선언하지 않기 때문에 쉽게 객체를 생성할 수 있음
- 필드값, 로컬 변수값, 매개변수값으로 주로 사용

익명 자식 객체

- 부모 클래스를 상속받아 생성되는 객체
- 부모 타입의 필드, 로컬 변수, 매개변수의 값으로 대입할 수 있음

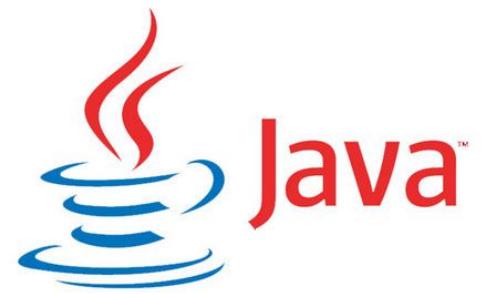
```
new 부모생성자(매개값, …) {  
    //필드  
    //메소드  
}
```

익명 구현 객체

- 인터페이스를 구현해서 생성되는 객체
- 인터페이스 타입의 필드, 로컬변수, 매개변수의 값으로 대입할 수 있음
- 안드로이드와 같은 UI 프로그램에서 이벤트를 처리하는 객체로 많이 사용

```
new 인터페이스() {  
    //필드  
    //메소드  
}
```

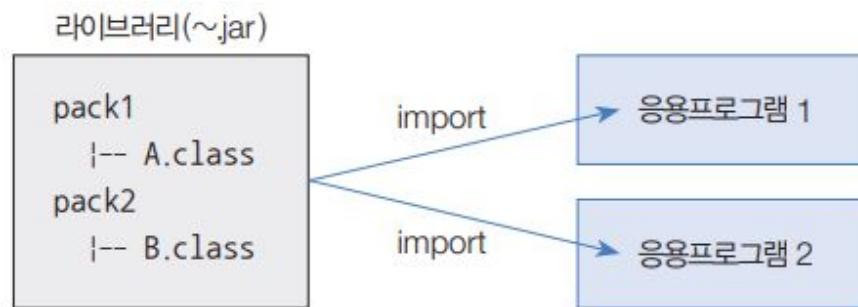




Chapter 10 라이브러리와 모듈

라이브러리 추가하기

- 프로그램 개발 시 활용할 수 있는 클래스와 인터페이스들을 모아놓은 것
- 일반적으로 JAR 압축 파일(~.jar) 형태. 클래스와 인터페이스의 바이트코드 파일(~.class)들이 압축



- 라이브러리 JAR 파일을 사용하려면 ClassPath(클래스를 찾기 위한 경로)에 추가
- 콘솔(명령 프롬프트 또는 터미널)에서 프로그램을 실행할 경우: java 명령어를 실행할 때 -classpath로 제공. 또는 CLASSPATH 환경 변수에 경로 추가
- 이클립스 프로젝트에서 실행할 경우: 프로젝트의 Build Path에 추가

모듈

- 패키지 관리 기능까지 포함된 라이브러리. Java 9부터 지원
- 모듈은 일부 패키지를 은닉하여 접근할 수 없게끔 할 수 있음

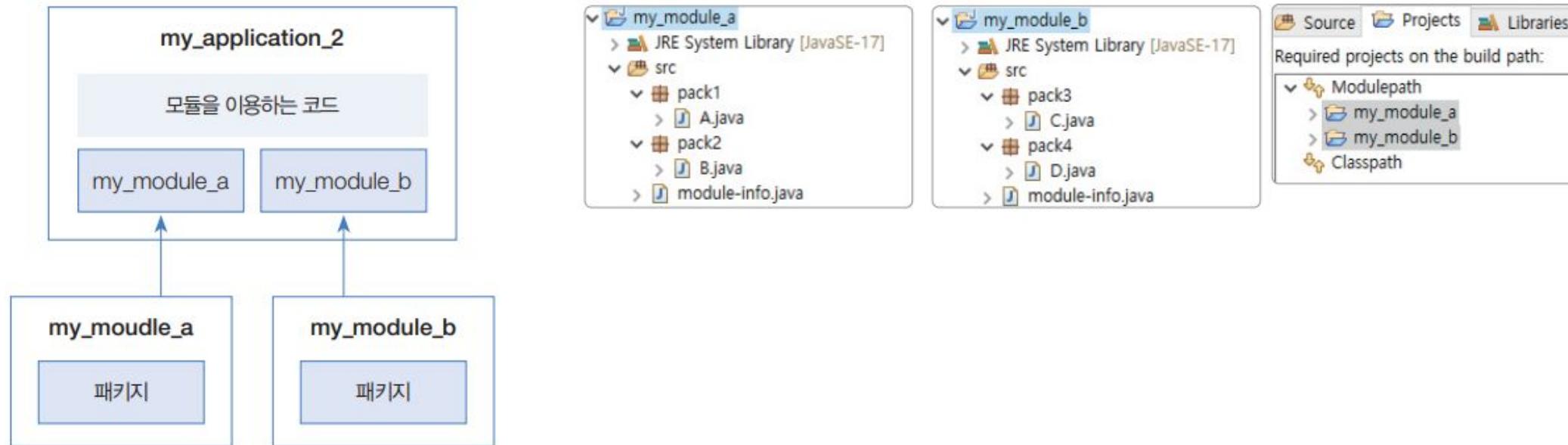


- 의존 모듈을 모듈 기술자(module-info.java)에 기술할 수 있어 모듈 간 의존 관계를 파악하기 쉬움
- 대규모 응용프로그램은 기능별로 모듈화해서 개발. 재사용성 및 유지보수에 유리



모듈화

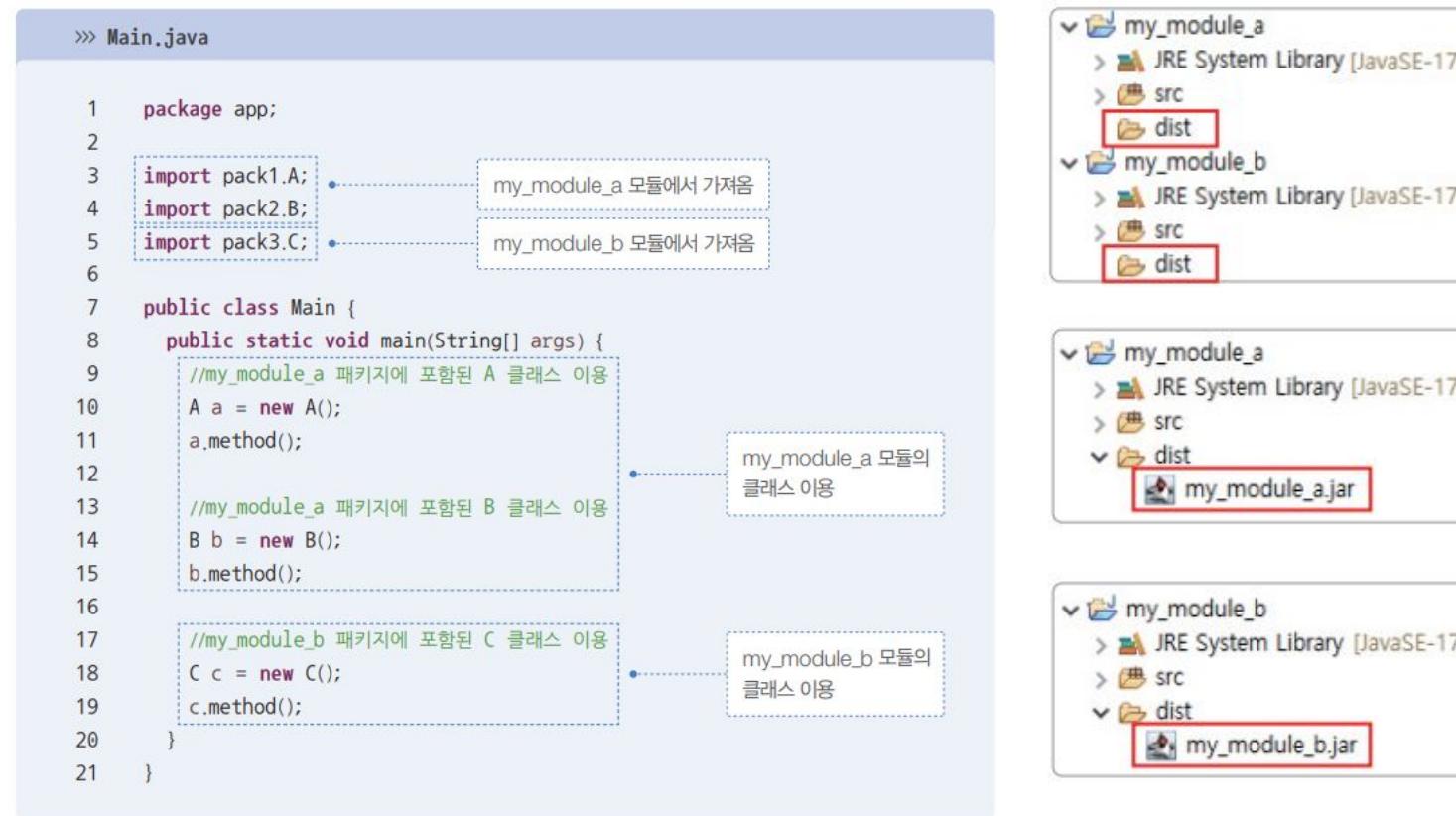
- 모듈화: 응용프로그램을 기능별로 서브 프로젝트(모듈)로 쪼갠 다음 조합해서 개발



- 응용프로그램의 규모가 클수록 협업과 유지보수 측면에서 모듈화 유리
- 다른 응용프로그램에서도 재사용 가능

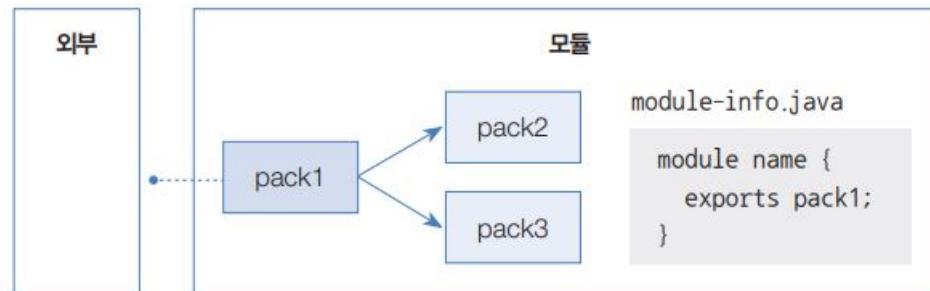
모듈 배포용 JAR 파일 생성

- 다른 모듈에서 쉽게 사용할 수 있게 바이트코드 파일(.class)로 구성된 배포용 JAR 파일을 모듈별로 따로 생성할 수 있음



패키지 은닉

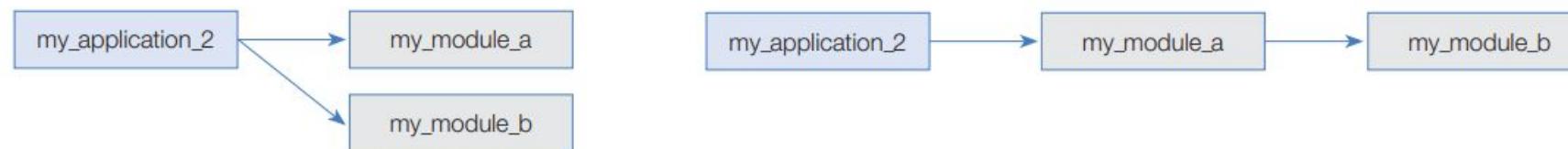
- 모듈은 모듈 기술자에서 exports 키워드를 사용해 내부 패키지 중 외부에서 사용할 패키지를 지정
- exports되지 않은 패키지는 자동적으로 은닉



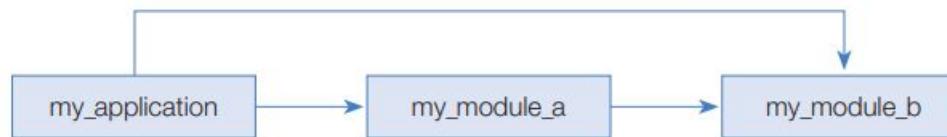
- 한 가지 패키지로 모듈 사용 방법 통일
- 다른 패키지를 수정하더라도 모듈 사용 방법이 바뀌지 않아 외부에 영향을 주지 않음

의존 설정 전이하기

- my_application_2 프로젝트는 직접적으로 두 모듈 my_module_a, my_module_b를 requires하고 있는 의존 관계
- my_application_2는 my_module_a에 의존하고, my_module_a는 my_module_b에 의존하는 관계로 변경하면 컴파일 오류 발생



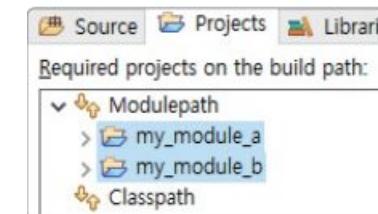
- 의존 설정 전이: my_module_a의 모듈 기술자에 transitive 키워드와 함께 my_module_b를 의존 설정하면 해결



집합 모듈

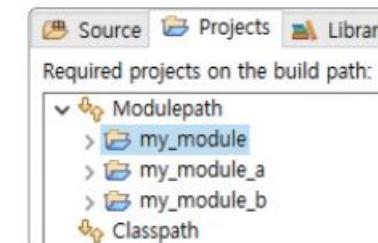
- 여러 모듈을 모아놓은 모듈. 자주 사용되는 모듈들을 일일이 requires하지 않아 편리.
- 집합 모듈은 자체적인 패키지를 가지지 않고, 모듈 기술자에 전이 의존 설정만 함

```
module my_module {  
    requires transitive my_module_a;  
    requires transitive my_module_b;  
}
```



>>> module-info.java

```
1 module my_application_2 {  
2     //requires my_module_a;  
3     //requires my_module_b;  
4     requires my_module; •----- my_module 모듈에만 의존  
5 }
```



리플렉션

- 실행 도중에 타입(클래스, 인터페이스 등)을 검사하고 구성 멤버를 조사하는 것
- 은닉된 패키지는 기본적으로 다른 모듈에 의해 리플렉션을 허용하지 않음
- 모듈은 모듈 기술자를 통해 모듈 전체 또는 지정된 패키지에 대해 리플렉션을 허용할 수 있고, 특정 외부 모듈에서만 리플렉션을 허용할 수도 있음

모듈 전체를 리플렉션 허용

```
open module 모듈명 {  
    ...  
}
```

지정된 패키지에 대해 리플렉션 허용

```
module 모듈명 {  
    ...  
    opens 패키지1;  
    opens 패키지2;  
}
```

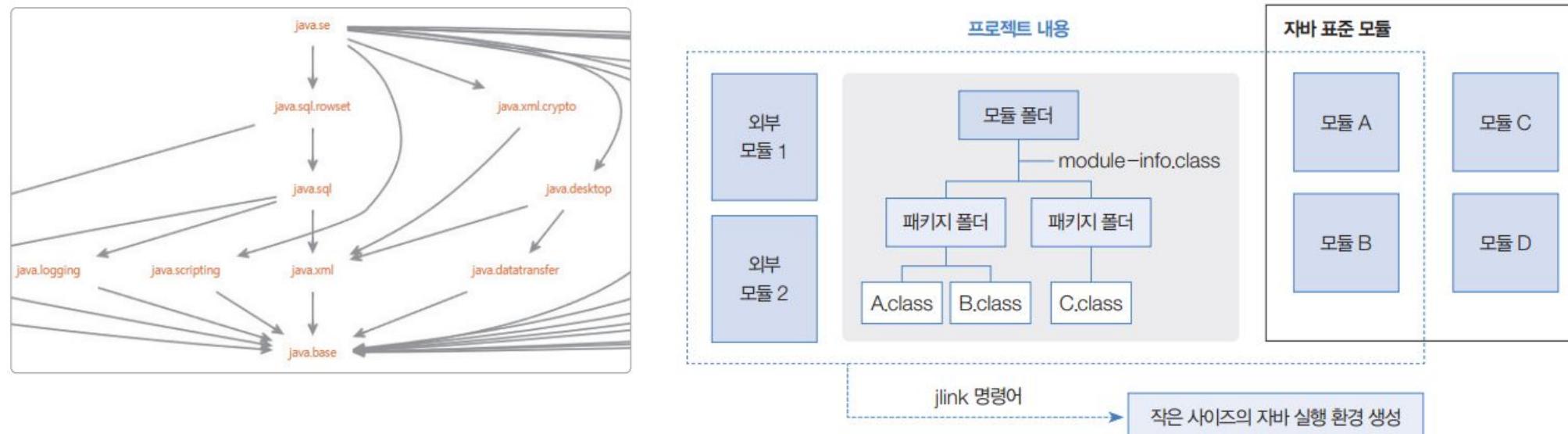
지정된 패키지에 대해 특정 외부 모듈에서만 리플렉션 허용

```
module 모듈명 {  
    ...  
    opens 패키지1 to 외부모듈명, 외부모듈명, ...;  
    opens 패키지2 to 외부모듈명;  
}
```

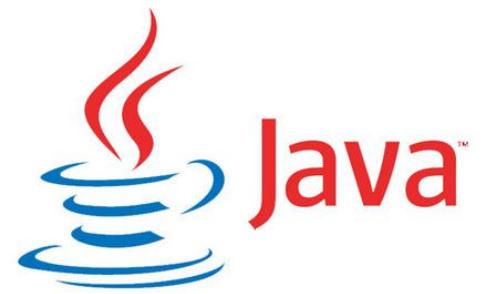
표준 라이브러리

- JDK가 제공하는 표준 라이브러리는 Java 9부터 모듈화됨
- 응용프로그램을 실행하는 데 필요한 모듈만으로 구성된 작은 사이즈의 자바 실행 환경(JRE)
- Java 17의 전체 모듈 그래프(화살표는 모듈간의 의존 관계를 표시)

<https://docs.oracle.com/en/java/javase/17/docs/api/java.se/module-summary.html>



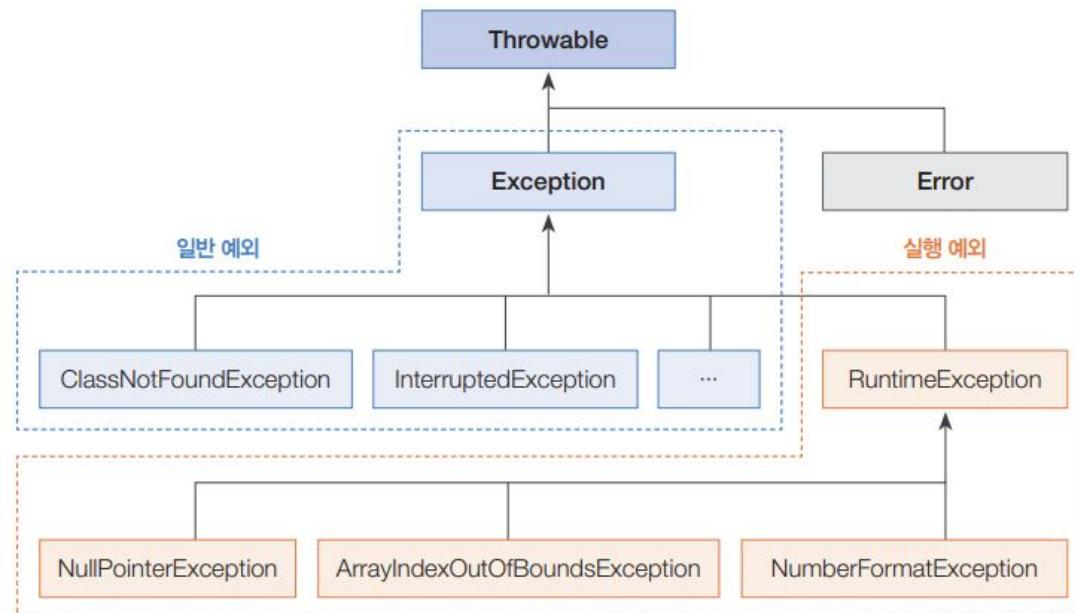




Chapter 11 예외 처리

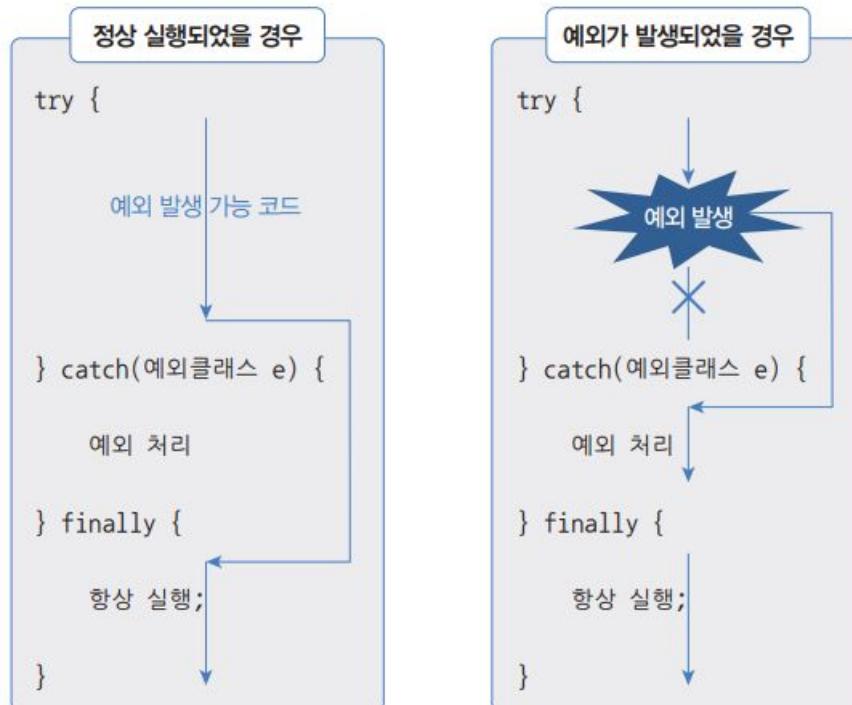
예외와 에러

- 예외: 잘못된 사용 또는 코딩으로 인한 오류
- 에러와 달리 예외 처리를 통해 계속 실행 상태를 유지할 수 있음
- 일반 예외(Exception): 컴파일러가 예외 처리 코드 여부를 검사하는 예외
- 실행 예외(Runtime Exception): 컴파일러가 예외 처리 코드 여부를 검사하지 않는 예외



예외 처리

- 예외 발생 시 프로그램의 갑작스러운 종료를 막고 정상 실행을 유지할 수 있게 처리하는 코드
- 예외 처리 코드는 try-catch-finally 블록으로 구성
- trycatch-finally 블록은 생성자 내부와 메소드 내부에서 작성



다중 catch로 예외 처리하기

- catch 블록의 예외 클래스는 try 블록에서 발생된 예외의 종류를 말함. 해당 타입의 예외가 발생하면 catch 블록이 선택되어 실행
- catch 블록이 여러 개라도 catch 블록은 단 하나만 실행됨
- 처리해야 할 예외 클래스들이 상속 관계에 있을 때는 하위 클래스 catch 블록을 먼저 작성하고 상위 클래스 catch 블록을 나중에 작성해야 함

```
try {  
    // ...  
    ArrayIndexOutOfBoundsException 발생  
    // ...  
    NumberFormatException 발생  
}  
catch(ArrayIndexOutOfBoundsException e) {  
    예외 처리1  
}  
catch(NumberFormatException e) {  
    예외 처리2  
}
```

```
try {  
    // ...  
    ArrayIndexOutOfBoundsException 발생  
    // ...  
    NumberFormatException 발생  
}  
catch(Exception e) {  
    예외 처리1  
}  
catch(ArrayIndexOutOfBoundsException e) {  
    예외 처리2  
}
```

리소스

- 데이터를 제공하는 객체
- 리소스는 사용하기 위해 열어야(open) 하며, 사용이 끝난 다음에는 닫아야(close) 함
- 리소스를 사용하다가 예외가 발생될 경우에도 안전하게 닫는 것이 중요
- try-with-resources 블록을 사용하면 예외 발생 여부와 상관없이 리소스를 자동으로 닫아줌

```
FileInputStream fis = null;
try {
    fis = new FileInputStream("file.txt");
    ...
} catch(IOException e) {
    ...
} finally {
    fis.close();
}
```

파일 열기

파일 닫기

```
try(FileInputStream fis = new FileInputStream("file.txt")){
    ...
} catch(IOException e) {
    ...
}
```

예외 떠넘기기

- 메소드 내부에서 예외 발생 시 throws 키워드 이용해 메소드를 호출한 곳으로 예외 떠넘기기
- throws는 메소드 선언부 끝에 작성. 떠넘길 예외 클래스를 쉼표로 구분해서 나열

```
리턴타입 메소드명(매개변수,⋯) throws 예외클래스1, 예외클래스2,⋯ {  
}
```

```
public void method1() {
    try {
        method2(); //method2() 호출
    } catch(ClassNotFoundException e) {
        System.out.println("예외 처리: " + e.getMessage());
    }
}

public void method2() throws ClassNotFoundException {
    Class.forName("java.lang.String2");
}
```

- 나열할 예외 클래스가 많으면 throws Exception 또는 throws Throwable 만으로 모든 예외

```
리턴타입 메소드명(매개변수,⋯) throws Exception {  
}
```

사용자 정의 예외

- 표준 라이브러리에 없어 직접 정의하는 클래스
- 일반 예외는 Exception의 자식 클래스로 선언. 실행 예외는 RuntimeException의 자식 클래스로 선언

```
public class XXXException extends [ Exception | RuntimeException ] {
    public XXXException() {
    }

    public XXXException(String message) {
        super(message);
    }
}
```

The diagram illustrates the code for a user-defined exception class named XXXException. It shows two constructor definitions. The first constructor, which takes no parameters, is labeled '기본 생성자' (Default Constructor). The second constructor, which takes a String parameter and calls the superclass's constructor, is labeled '예외 메시지를 입력받는 생성자' (Constructor that receives exception message).

예외 발생시키기

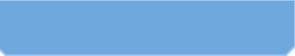
- throw 키워드와 함께 예외 객체를 제공해

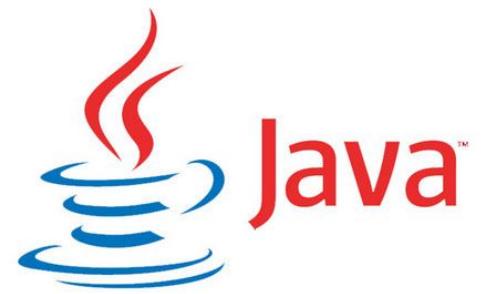
사용자 정의 예외를 직접 코드에서 발생

- 예외의 원인에 해당하는 메시지를
제공하려면 생성자 매개값으로 전달

```
throw new Exception()
throw new RuntimeException();
throw new InsufficientException();
```

```
throw new Exception("예외메시지")
throw new RuntimeException("예외메시지");
throw new InsufficientException("예외메시지");
```





Chapter 12 java.base 모듈

API 도큐먼트

- 자바 표준 모듈에서 제공하는 라이브러리를 쉽게 찾아서 사용할 수 있도록 도와주는 문서
- JDK 버전별로 사용할 수 있는 API 도큐먼트:

<https://docs.oracle.com/en/java/javase/index.html>

★ String 도큐먼트를 찾는 3가지 방법

방법1: 웹 사이트 메뉴 이용

- ① [All Modules] 탭에서 java.base 모듈을 클릭한다.
- ② java.base의 Packages 목록에서 java.lang 패키지를 클릭한다.
- ③ java.lang의 [All Classes and Interfaces] 탭에서 String 클래스를 클릭한다.

방법2: 웹 사이트 검색 이용

- ① 오른쪽 상단의 Search 검색란에 'String'을 입력한다.
- ② 드롭다운 목록에서 java.lang.String 항목을 선택한다.

방법3: 이클립스 이용

- ① 자바 코드에서 String 클래스를 마우스로 선택한 다음 **F1** 키를 누르면 Help 뷰가 나타난다.
- ② Help 뷰에서 Javadoc for 'java.lang.String' 링크를 클릭하면 String 도큐먼트로 이동한다.

java.base

- 모든 모듈이 의존하는 기본 모듈로, 모듈 중 유일하게 requires하지 않아도 사용할 수 있음

패키지	용도
java.lang	자바 언어의 기본 클래스를 제공
java.util	자료 구조와 관련된 컬렉션 클래스를 제공
java.text	날짜 및 숫자를 원하는 형태의 문자열로 만들어 주는 포맷 클래스를 제공
java.time	날짜 및 시간을 조작하거나 연산하는 클래스를 제공
java.io	입출력 스트림 클래스를 제공
java.net	네트워크 통신과 관련된 클래스를 제공
java.nio	데이터 저장을 위한 Buffer 및 새로운 입출력 클래스 제공

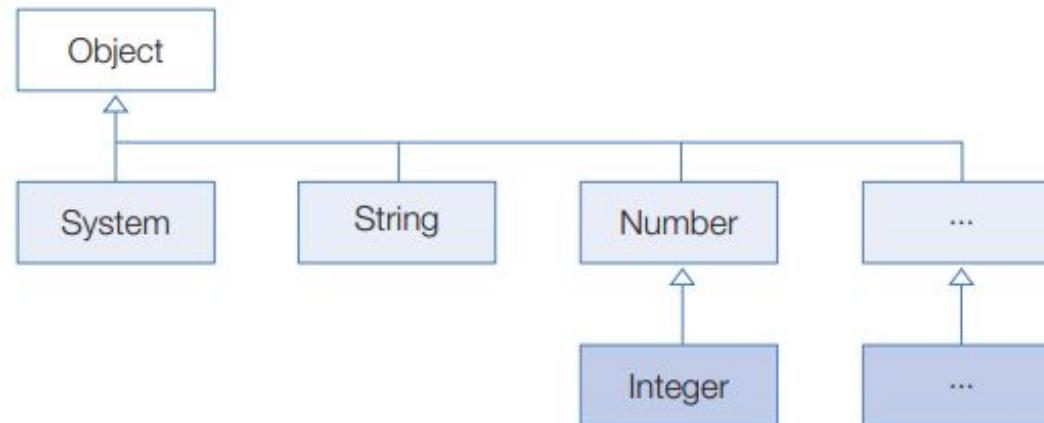
java.lang

- 자바 언어의 기본적인 클래스를 담고 있는 패키지. 이 패키지에 있는 클래스와 인터페이스는 import 없이 사용할 수 있음

클래스	용도	
Object	- 자바 클래스의 최상위 클래스로 사용	
System	- 키보드로부터 데이터를 입력받을 때 사용 - 모니터(콘솔)로 출력하기 위해 사용 - 프로세스를 종료시킬 때 사용 - 진행 시간을 읽을 때 사용 - 시스템 속성(프로퍼티)을 읽을 때 사용	
문자열 관련	String	- 문자열을 저장하고 조작할 때 사용
	StringBuilder	- 효율적인 문자열 조작 기능이 필요할 때 사용
	java.util.StringTokenizer	- 구분자로 연결된 문자열을 분리할 때 사용
포장 관련	Byte, Short, Character Integer, Float, Double Boolean	- 기본 타입의 값을 포장할 때 사용 - 문자열을 기본 타입으로 변환할 때 사용
	Math	- 수학 계산이 필요할 때 사용
Class	- 클래스의 메타 정보(이름, 구성 멤버) 등을 조사할 때 사용	

Object 클래스

- 클래스를 선언할 때 extends 키워드로 다른 클래스를 상속하지 않으면 암시적으로 java.lang.Object 클래스를 상속 → 자바의 모든 클래스는 Object의 자식이거나 자손 클래스



메소드	용도
boolean equals(Object obj)	객체의 번지를 비교하고 결과를 리턴
int hashCode()	객체의 해시코드를 리턴
String toString()	객체 문자 정보를 리턴

객체 동등 비교

- Object의 equals() 메소드는 객체의 번지를 비교하고 boolean 값을 리턴

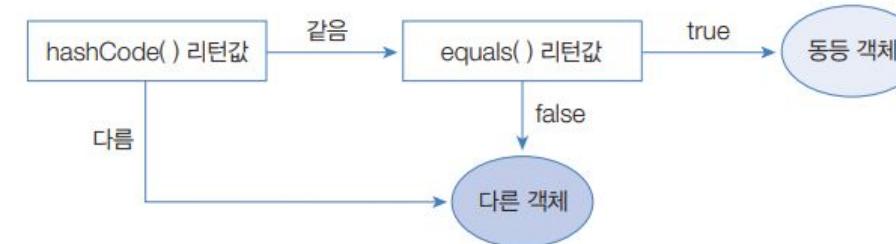
```
public boolean equals(Object obj)
```

객체 해시코드

- 객체를 식별하는 정수. Object의 hashCode() 메소드는 객체의 메모리 번지를 이용해서 해시코드를 생성하기 때문에 객체마다 다른 정수값을 리턴

```
public int hashCode()
```

- hashCode()가 리턴하는 정수값이 같은지 확인하고, equals() 메소드가 true를 리턴하는지 확인해서 동등 객체임을 판단



객체 문자 정보

- 객체를 문자열로 표현한 값. Object의 `toString()` 메소드는 객체의 문자 정보를 리턴
- 기본적으로 Object의 `toString()` 메소드는 ‘클래스명@16진수해시코드’로 구성된 문자열을 리턴

```
Object obj = new Object();
System.out.println(obj.toString());
```

java.lang.Object@de6ced

레코드 선언

- 데이터 전달을 위한 DTO 작성 시 반복적으로 사용되는 코드를 줄이기 위해 도입

```
public record Person(String name, int age) {  
}
```

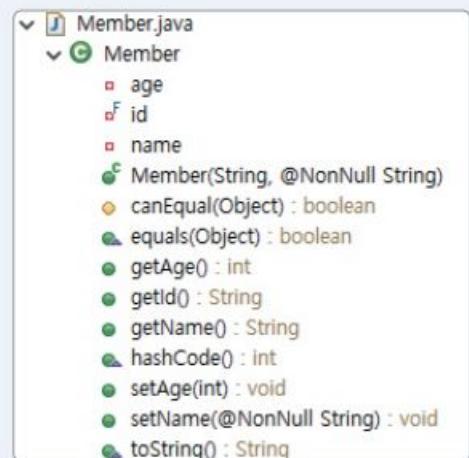
롬복 사용하기

- DTO 클래스를 작성할 때 Getter, Setter, hashCode(), equals (), toString() 메소드를 자동 생성
- 필드가 final이 아니며, 값을 읽는 Getter는 getXxx(또는 isXxx)로, 값을 변경하는 Setter는 setXxx로 생성됨

```
java -jar lombok.jar
```

>>> Member.java

```
1 package ch12.sec03.exam05;
2
3 import lombok.Data;
4 import lombok.NonNull;
5
6 @Data
7 public class Member {
8     private final String id;
9     @NonNull private String name;
10    private int age;
11 }
```



System 클래스

- System 클래스의 정적 static 필드와 메소드를 이용하면 프로그램 종료, 키보드 입력, 콘솔(모니터) 출력, 현재 시간 읽기, 시스템 프로퍼티 읽기 등이 가능

정적 멤버		용도	메소드			용도
필드	out	콘솔(모니터)에 문자 출력		long currentTimeMillis()		1/1000 초 단위로 진행된 시간을 리턴
	err	콘솔(모니터)에 에러 내용 출력			long nanoTime()	1/10 ⁹ 초 단위로 진행된 시간을 리턴
	in	키보드 입력				
메소드	exit(int status)	프로세스 종료		java.specification.version	자바 스펙 버전	17
	currentTimeMillis()	현재 시간을 밀리초 단위의 long 값으로 리턴		java.home	JDK 디렉토리 경로	C:\Program Files\Java\jdk-17.0.3
	nanoTime()	현재 시간을 나노초 단위의 long 값으로 리턴		os.name	운영체제	Windows 10
	getProperty()	운영체제와 사용자 정보 제공		user.name	사용자 이름	xxx
	getenv()	운영체제의 환경 변수 정보 제공		user.home	사용자 홈 디렉토리 경로	C:\Users\xxx
				user.dir	현재 디렉토리 경로	C:\ThisIsJavaSecondEdition\workspace>thisisjava

String 클래스

- String 클래스는 문자열을 저장하고 조작할 때 사용
- 문자열 리터럴은 자동으로 String 객체로 생성. String 클래스의 다양한 생성자를 이용해서 직접 객체를 생성할 수도 있음

```
//기본 문자셋으로 byte 배열을 디코딩해서 String 객체로 생성  
String str = new String(byte[] bytes);
```

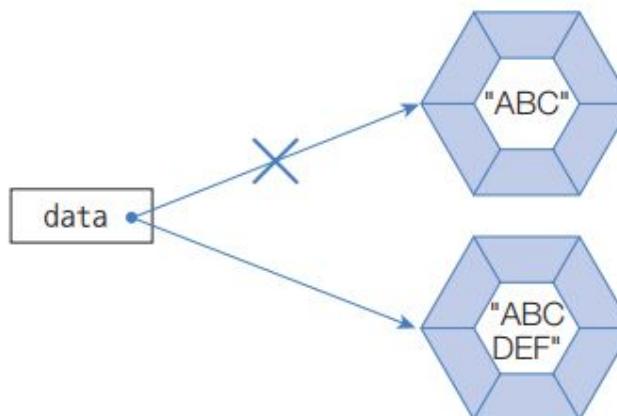
```
//특정 문자셋으로 byte 배열을 디코딩해서 String 객체로 생성  
String str = new String(byte[] bytes, String charsetName);
```

- 한글 1자를 UTF-8로 인코딩하면 3바이트가 되고, EUC-KR로 인코딩하면 2바이트가 됨

StringBuilder 클래스

- 잦은 문자열 변경 작업을 해야 한다면 String보다는 StringBuilder가 좋음
- StringBuilder는 내부 버퍼에 문자열을 저장해두고 그 안에서 추가, 수정, 삭제 작업을 하도록 설계

```
String data = "ABC";
data += "DEF";
```



리턴 타입	메소드(매개변수)	설명
StringBuilder	append(기본값 문자열)	문자열을 끝에 추가
StringBuilder	insert(위치, 기본값 문자열)	문자열을 지정 위치에 추가
StringBuilder	delete(시작 위치, 끝 위치)	문자열 일부를 삭제
StringBuilder	replace(시작 위치, 끝 위치, 문자열)	문자열 일부를 대체
String	toString()	완성된 문자열을 리턴

StringTokenizer 클래스

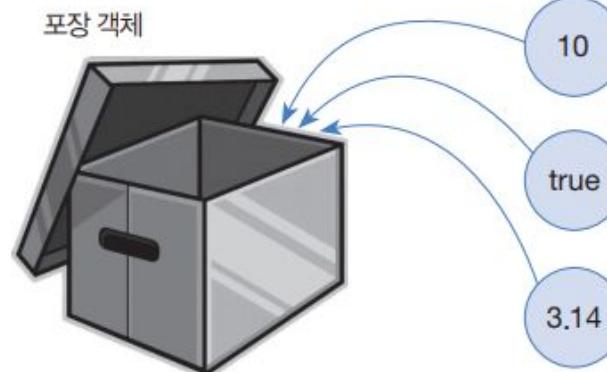
- 문자열에 여러 종류가 아닌 한 종류의 구분자만 있다면 StringTokenizer를 사용할 수도 있음
StringTokenizer 객체를 생성 시 첫 번째 매개값으로 전체 문자열을 주고, 두 번째 매개값으로 구분자를 줌. 구분자를 생략하면 공백이 기본 구분자가 됨

```
String data = "홍길동/이수홍/박연수";
StringTokenizer st = new StringTokenizer(data, "/");
```

리턴 타입	메소드(매개변수)	설명
int	countTokens()	분리할 수 있는 문자열의 총 수
boolean	hasMoreTokens()	남아 있는 문자열이 있는지 여부
String	nextToken()	문자열을 하나씩 가져옴

포장 객체

- 기본 타입(byte, char, short, int, long, float, double, boolean)의 값을 갖는 객체
- 포장하고 있는 기본 타입의 값을 변경할 수 없고, 단지 객체로 생성하는 목적



기본 타입	포장 클래스
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

박싱과 언박싱

- 박싱: 기본 타입 값을 포장 객체로 만드는 과정. 포장 클래스 변수에 기본 타입 값이 대입 시 발생
- 언박싱: 포장 객체에서 기본 타입 값을 얻어내는 과정. 기본 타입 변수에 포장 객체가 대입 시 발생

```
Integer obj = 100; //박싱  
int value = obj; //언박싱
```

```
int value = obj + 50; //언박싱 후 연산
```

문자열을 기본 타입 값으로 변환

- 포장 클래스는 문자열을 기본 타입 값으로 변환할 때도 사용.
- 대부분의 포장 클래스에는 ‘parse+기본타입’ 명으로 되어 있는 정적 메소드 있음

포장 값 비교

- 포장 객체는 번지를 비교하므로 내부 값을 비교하기 위해 ==와 != 연산자를 사용할 수 없음

```
Integer obj1 = 300;  
Integer obj2 = 300;  
System.out.println(obj1 == obj2);
```

- 다음 범위의 값을 갖는 포장 객체는 ==와 != 연산자로 비교할 수 있지만, 내부 값을 비교하는 것이 아니라 객체 번지를 비교하는 것

타입	값의 범위
boolean	true, false
char	\u0000 ~ \u007f
byte, short, int	-128 ~ 127

- 대신 포장 클래스의 equals() 메소드로 내부 값을 비교할 수 있음

Math 클래스

- 수학 계산에 사용할 수 있는 정적 메소드 제공

구분	코드	리턴값
절대값	int v1 = Math.abs(-5); double v2 = Math.abs(-3.14);	v1 = 5 v2 = 3.14
올림값	double v3 = Math.ceil(5.3); double v4 = Math.ceil(-5.3);	v3 = 6.0 v4 = -5.0
버림값	double v5 = Math.floor(5.3); double v6 = Math.floor(-5.3);	v5 = 5.0 v6 = -6.0
최대값	int v7 = Math.max(5, 9); double v8 = Math.max(5.3, 2.5);	v7 = 9 v8 = 5.3
최소값	int v9 = Math.min(5, 9); double v10 = Math.min(5.3, 2.5);	v9 = 5 v10 = 2.5
랜덤값	double v11 = Math.random();	0.0 ≤ v11 < 1.0
반올림값	long v14 = Math.round(5.3); long v15 = Math.round(5.7);	v14 = 5 v15 = 6

Date 클래스

- 날짜를 표현하는 클래스로 객체 간에 날짜 정보를 주고받을 때 사용
- Date() 생성자는 컴퓨터의 현재 날짜를 읽어 Date 객체로 만듦

```
Date now = new Date();
```

Calendar 클래스

- 달력을 표현하는 추상 클래스
- getInstance() 메소드로 컴퓨터에 설정된 시간대 기준으로 Calendar 하위 객체를 얻을 수 있음

```
Calendar now = Calendar.getInstance();
```

날짜와 시간 조작

- java.time 패키지의 LocalDateTime 클래스가 제공하는 메소드를 이용해 날짜와 시간을 조작 가능

날짜와 시간 비교

- LocalDateTime 클래스는 날짜와 시간을 비교할 수 있는 메소드 제공

메소드(매개변수)	설명
minusYears(long)	년 빼기
minusMonths(long)	월 빼기
minusDays(long)	일 빼기
minusWeeks(long)	주 빼기
plusYears(long)	년 더하기
plusMonths(long)	월 더하기
plusWeeks(long)	주 더하기
plusDays(long)	일 더하기
minusHours(long)	시간 빼기
minusMinutes(long)	분 빼기
minusSeconds(long)	초 빼기
minusNanos(long)	나노초 빼기
plusHours(long)	시간 더하기
plusMinutes(long)	분 더하기
plusSeconds(long)	초 더하기

리턴 타입	메소드(매개변수)	설명
boolean	isAfter(other)	이후 날짜인지?
	isBefore(other)	이전 날짜인지?
	isEqual(other)	동일 날짜인지?
long	until(other, unit)	주어진 단위(unit) 차이를 리턴

DecimalFormat

- 숫자를 형식화된 문자열로 변환

기호	의미	패턴 예	1234567.89 → 변환 결과
0	10진수(빈자리는 0으로 채움)	0 0.0 0000000000.00000	1234568 1234567.9 0001234567.89000
#	10진수(빈자리는 채우지 않음)	# .# #####,###,###	1234568 1234567.9 1234567.89
.	소수점	#.0	1234567.9
-	음수 기호	+#.0 -#.0	+1234567.9 -1234567.9
.	단위 구분	#,###.0	1,234,567.9
E	지수 문자	0.0E0	1.2E6
;	양수와 음수의 패턴을 모두 기술할 경우, 패턴 구분자	+#,### ; -#,###	+1,234,568(양수일 때) -1,234,568(음수일 때)
%	% 문자	#.%	123456789 %
\u00A4	통화 기호	\u00A4 #,###	₩1,234,568

SimpleDateFormat

- 날짜를 형식화된 문자열로 변환

패턴 문자	의미	패턴 문자	의미
y	년	H	시(0~23)
M	월	h	시(1~12)
d	일	K	시(0~11)
D	월 구분이 없는 일(1~365)	k	시(1~24)
E	요일	m	분
a	오전/오후	s	초
w	년의 몇 번째 주	S	밀리세컨드(1/1000초)
W	월의 몇 번째 주		

정규 표현식

- 문자 또는 숫자와 관련된 표현과 반복 기호가 결합된 문자열

Pattern 클래스로 검증

- java.util.regex 패키지의 Pattern 클래스는 정규 표현식으로 문자열을 검증하는 matches() 메소드 제공

표현 및 기호	설명		
[]	한 개의 문자	[abc]	a, b, c 중 하나의 문자
		[^abc]	a, b, c 이외의 하나의 문자
		[a-zA-Z]	a~z, A~Z 중 하나의 문자
\d	한 개의 숫자, [0~9]와 동일		
\s	공백		
\w	한 개의 알파벳 또는 한 개의 숫자, [a-zA-Z_0~9]와 동일		
\.	.		
.	모든 문자 중 한 개의 문자		
?	없음 또는 한 개		
*	없음 또는 한 개 이상		
+	한 개 이상		
{n}	정확히 n개		
{n,}	최소한 n개		
{n, m}	n개부터 m개까지		
a b	a 또는 b		
()	그룹핑		

```
boolean result = Pattern.matches("정규식", "검증할 문자열");
```

리플렉션

- Class 객체로 관리하는 클래스와 인터페이스의 메타 정보를 프로그램에서 읽고 수정하는 것
- 메타 정보: 패키지 정보, 타입 정보, 멤버(생성자, 필드, 메소드) 정보

```
① Class clazz = 클래스이름.class;           ————— 클래스로부터 얻는 방법  
② Class clazz = Class.forName("패키지…클래스이름");  
③ Class clazz = 객체참조변수.getClass();      ————— 객체로부터 얻는 방법
```

패키지와 타입 정보 얻기

- 패키지와 타입(클래스, 인터페이스) 이름 정보는 다음 메소드로 얻을 수 있음

메소드	용도
Package getPackage()	패키지 정보 읽기
String getSimpleName()	패키지를 제외한 타입 이름
String getName()	패키지를 포함한 전체 타입 이름

리소스 경로 얻기

- Class 객체는 클래스 파일(~.class)의 경로 정보를 기준으로 상대 경로에 있는 다른 리소스 파일 (이미지, XML, Property 파일)의 정보를 얻을 수 있음

메소드	용도
URL getResource(String name)	리소스 파일의 URL 리턴
InputStream getResourceAsStream(String name)	리소스 파일의 InputStream 리턴

```
C:\app\bin
| - Car.class
| - photo1.jpg
| - images
    | - photo2.jpg
```

어노테이션

- 코드에서 @으로 작성되는 요소. 클래스 또는 인터페이스를 컴파일하거나 실행할 때 어떻게 처리해야 할 것인지를 알려주는 설정 정보
- ① 컴파일 시 사용하는 정보 전달
- ② 빌드 툴이 코드를 자동으로 생성할 때 사용하는 정보 전달
- ③ 실행 시 특정 기능을 처리할 때 사용하는 정보 전달

어노테이션 타입 정의와 적용

- @interface 뒤에 사용할 어노테이션 이름 작성

```
public @interface AnnotationName {  
}
```

```
@AnnotationName
```

어노테이션 적용 대상

- 어노테이션을 적용할 수 있는 대상의 종류는 ElementType 열거 상수로 정의
- @Target 어노테이션으로 적용 대상 지정. @Target의 기본 속성 value 값은 ElementType 배열

ElementType 열거 상수	적용 요소
TYPE	클래스, 인터페이스, 열거 타입
ANNOTATION_TYPE	어노테이션
FIELD	필드
CONSTRUCTOR	생성자
METHOD	메소드
LOCAL_VARIABLE	로컬 변수
PACKAGE	패키지

```
@Target( { ElementType.TYPE, ElementType.FIELD, ElementType.METHOD } )
public @interface AnnotationName {
}
```

```
@AnnotationName
public class ClassName {
    @AnnotationName
    private String fieldName;

    //@AnnotationName
    public ClassName() { }

    @AnnotationName
    public void methodName() { }
}
```

TYPE(클래스)에 적용
필드에 적용
@Target에 CONSTRUCT가 없으므로 생성자에는 적용 못함
메소드에 적용

어노테이션 유지 정책

- 어노테이션 정의 시 @AnnotationName을 언제까지 유지할지 지정
- 어노테이션 유지 정책은 RetentionPolicy 열거 상수로 정의

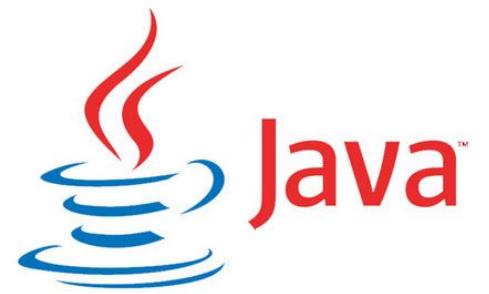
RetentionPolicy 열거 상수	어노테이션 적용 시점	어노테이션 제거 시점
SOURCE	컴파일할 때 적용	컴파일된 후에 제거됨
CLASS	메모리로 로딩할 때 적용	메모리로 로딩된 후에 제거됨
RUNTIME	실행할 때 적용	계속 유지됨

어노테이션 설정 정보 이용

- 애플리케이션은 리플렉션을 이용해 적용 대상에서 어노테이션의 정보를 다음 메소드로 얻어냄

리턴 타입	메소드명(매개변수)	설명
boolean	<code>isAnnotationPresent(AnnotationName.class)</code>	지정한 어노테이션이 적용되었는지 여부
Annotation	<code>getAnnotation(AnnotationName.class)</code>	지정한 어노테이션이 적용되어 있으면 어노테이션을 리턴하고, 그렇지 않다면 null을 리턴
Annotation[]	<code>getDeclaredAnnotations()</code>	적용된 모든 어노테이션을 리턴





Chapter 13 제네릭

제네릭

- 결정되지 않은 타입을 파라미터로 처리하고 실제 사용할 때 파라미터를 구체적인 타입으로 대체시키는 기능
- <T>는 T가 타입 파라미터임을 뜻하는 기호. 타입이 필요한 자리에 T를 사용할 수 있음을 알려줌

```
public class Box <T> {  
    public T content;  
}
```

Box<String> box = new Box<String>();  Box<String> box = new Box<>();

Box<Integer> box = new Box<Integer>();  Box<Integer> box = new Box<>();

제네릭 타입

- 결정되지 않은 타입을 파라미터로 가지는 클래스와 인터페이스
- 선언부에 ‘<’ > 부호가 붙고 그 사이에 타입 파라미터들이 위치

```
public class 클래스명<A, B, …> { ... }  
public interface 인터페이스명<A, B, …> { ... }
```

- 타입 파라미터는 일반적으로 대문자 알파벳 한 글자로 표현
- 외부에서 제네릭 타입을 사용하려면 타입 파라미터에 구체적인 타입을 지정. 지정하지 않으면 Object 타입이 암묵적으로 사용

제네릭 메소드

- 타입 피라미터를 가지고 있는 메소드. 타입 파라미터가 메소드 선언부에 정의
- 리턴 타입 앞에 <> 기호 추가하고 타입 파라미터 정의 후 리턴 타입과 매개변수 타입에서 사용

```
public <A, B, …> 리턴타입 메소드명(매개변수, ...) { ... }
```

↑
타입 파라미터 정의

- 타입 파라미터 T는 매개값의 타입에 따라 컴파일 과정에서 구체적인 타입으로 대체

```
public <T> Box<T> boxing(T t) { ... }
```

```
① Box<Integer> box1 = boxing(100);  
② Box<String> box2 = boxing("안녕하세요");
```

제한된 타입 파라미터

- 모든 타입으로 대체할 수 없고, 특정 타입과 자식 또는 구현 관계에 있는 타입만 대체할 수 있는 타입 파라미터

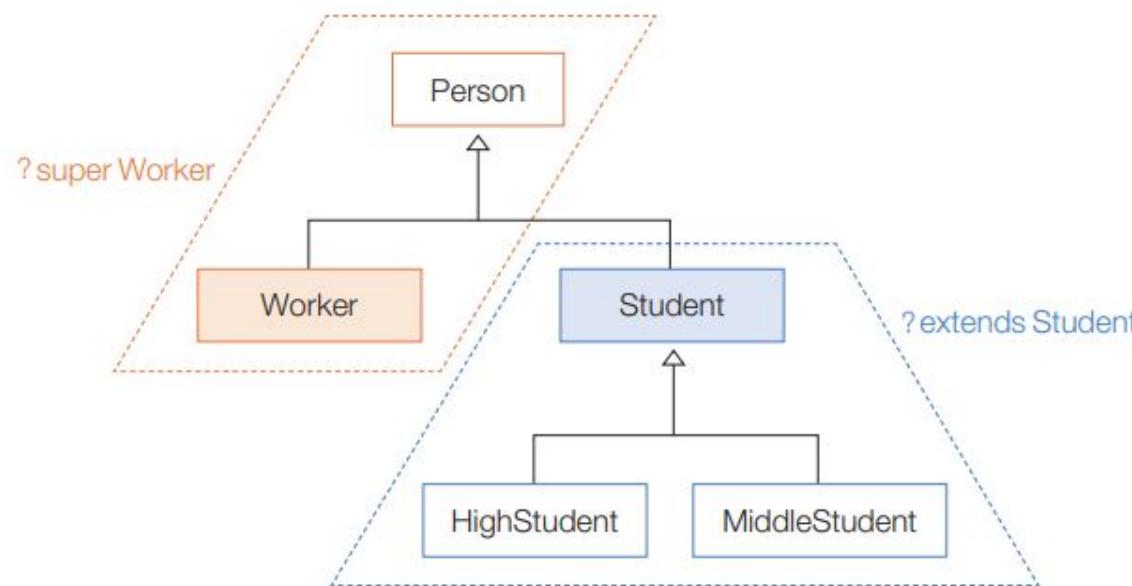
```
public <T extends 상위타입> 리턴타입 메소드(매개변수, ...) { ... }
```

- 상위 타입은 클래스뿐만 아니라 인터페이스도 가능

```
public <T extends Number> boolean compare(T t1, T t2) {  
    double v1 = t1.doubleValue(); //Number의 doubleValue() 메소드 사용  
    double v2 = t2.doubleValue(); //Number의 doubleValue() 메소드 사용  
    return (v1 == v2);  
}
```

와일드카드 타입 파라미터

- 제네릭 타입을 매개값이나 리턴 타입으로 사용할 때 범위에 있는 모든 타입으로 대체할 수 있는 타입 파라미터. ?로 표시

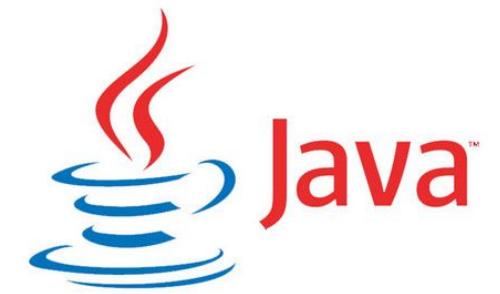


```
리턴타입 메소드명(제네릭타입<? extends Student> 변수) { … }
```

```
리턴타입 메소드명(제네릭타입<? super Worker> 변수) { … }
```

```
리턴타입 메소드명(제네릭타입<?> 변수) { … }
```

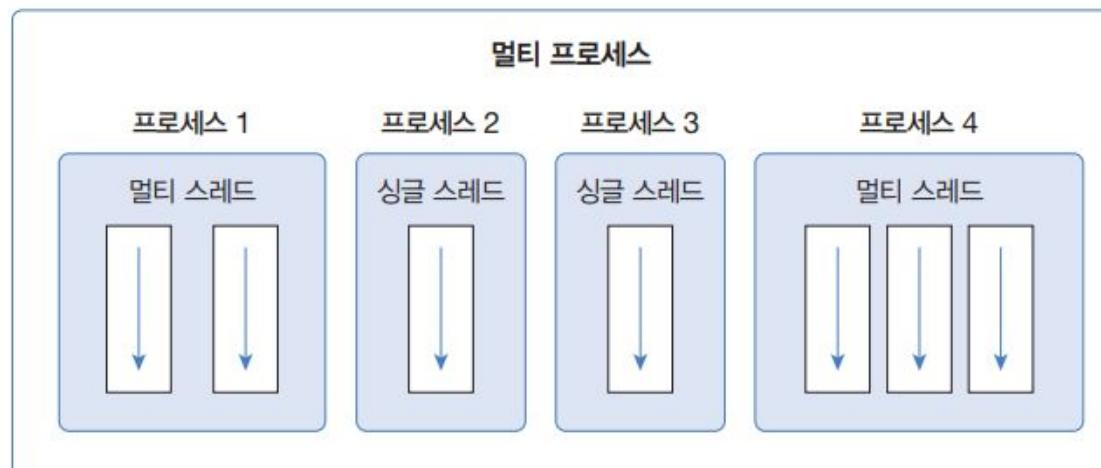




Chapter 14 멀티 스레드

멀티 프로세스와 멀티 스레드

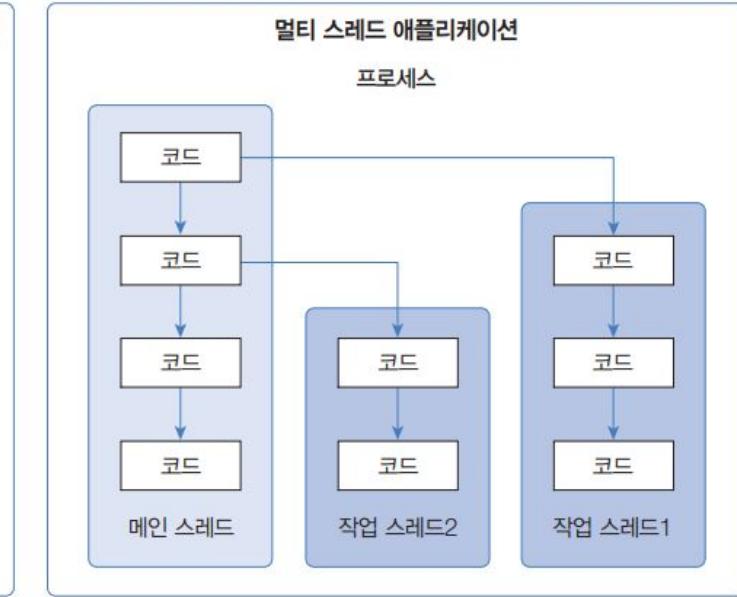
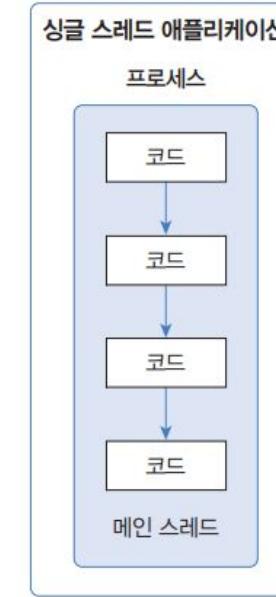
- **프로세스**: 운영체제는 실행 중인 프로그램을 관리
- **멀티 태스킹**: 두 가지 이상의 작업을 동시에 처리하는 것
- **스레드**: 코드의 실행 흐름
- **멀티 스레드**: 두 개의 코드 실행 흐름. 두 가지 이상의 작업을 처리
- **멀티 프로세스** = 프로그램 단위의 멀티 태스킹 / 멀티 스레드 = 프로그램 내부에서의 멀티 태스킹



메인 스레드

- 메인 스레드는 main() 메소드의 첫 코드부터 순차적으로 실행
- main() 메소드의 마지막 코드를 실행하거나 return 문을 만나면 실행을 종료
- 메인 스레드는 추가 작업 스레드들을 만들어서 실행시킬 수 있음
- 메인 스레드가 작업 스레드보다 먼저 종료되더라도 작업 스레드가 계속 실행 중이라면 프로세스는 종료되지 않음

```
public static void main(String[] args) {  
    String data = null;  
    if(...) {  
    }  
    while(...) {  
    }  
    System.out.println("...");  
}
```



작업 스레드

- 멀티 스레드 프로그램을 개발 시 먼저 몇 개의 작업을 병렬로 실행할지 결정하고 각 작업별로 스레드를 생성



Thread 클래스로 직접 생성

- java.lang 패키지에 있는 Thread 클래스로부터 작업 스레드 객체를 직접 생성하려면 Runnable 구현 객체를 매개값으로 갖는 생성자를 호출

```
Thread thread = new Thread(Runnable target);
```

Thread 자식 클래스로 생성

- Thread 클래스를 상속한 다음 run() 메소드를 재정의해서 스레드가 실행할 코드를 작성하고 객체를 생성
- 혹은 Thread 익명 자식 객체를 사용 가능

```
public class WorkerThread extends Thread {  
    @Override  
    public void run() {  
        //스레드가 실행할 코드  
    }  
}  
  
//스레드 객체 생성  
Thread thread = new WorkerThread();
```

```
Thread thread = new Thread() {  
    @Override  
    public void run() {  
        //스레드가 실행할 코드  
    }  
};  
thread.start();
```

작업 스레드의 이름

- 작업 스레드 이름을 Thread-n 대신 다른 이름으로 설정하려면 Thread 클래스의 setName() 메소드 사용

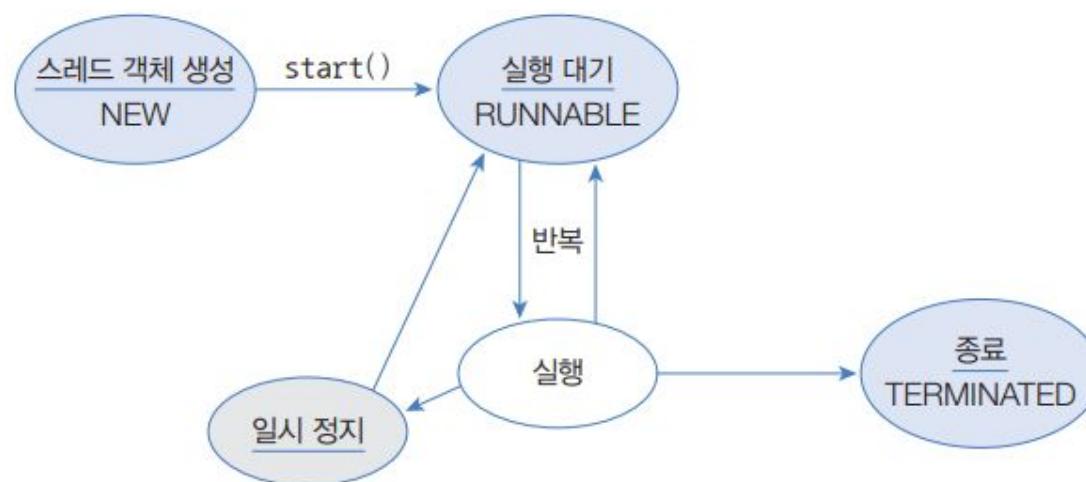
```
thread.setName("스레드 이름");
```

- 디버깅할 때 어떤 스레드가 작업을 하는지 조사하기 위해 주로 사용
- 어떤 스레드가 실행하고 있는지 확인하려면 정적 메소드인 currentThread()로 스레드 객체의 참조를 얻은 다음 getName() 메소드로 이름을 출력

```
Thread thread = Thread.currentThread();
System.out.println(thread.getName());
```

스레드 상태

- 실행 대기 상태: 실행을 기다리고 있는 상태
- 실행 상태: CPU 스케줄링에 따라 CPU를 점유하고 run() 메소드를 실행. 스케줄링에 의해 다시 실행 대기 상태로 돌아갔다가 다른 스레드가 실행 상태 반복
- 종료 상태: 실행 상태에서 run() 메소드가 종료되어 실행할 코드 없이 스레드의 실행을 멈춘 상태



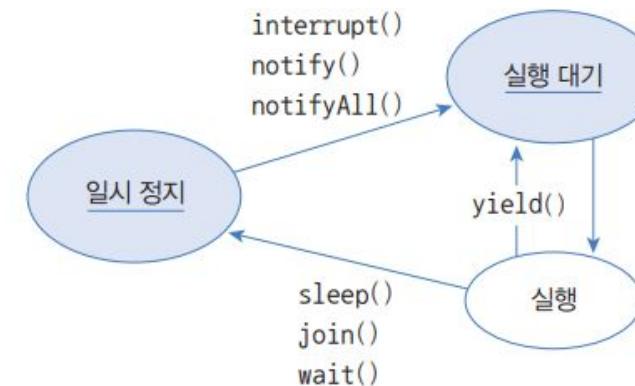
일시 정지 상태

- 스레드가 실행할 수 없는 상태
- 스레드가 다시 실행 상태로 가기 위해서는 일시 정지 상태에서 실행 대기 상태로 가야 함
- Thread 클래스의 sleep() 메소드: 실행 중인 스레드를 일정 시간 멈추게 함
- 매개값 단위는 밀리세컨드(1/1000)

```

try {
    Thread.sleep(1000);
} catch(InterruptedException e) {
    // interrupt() 메소드가 호출되면 실행
}

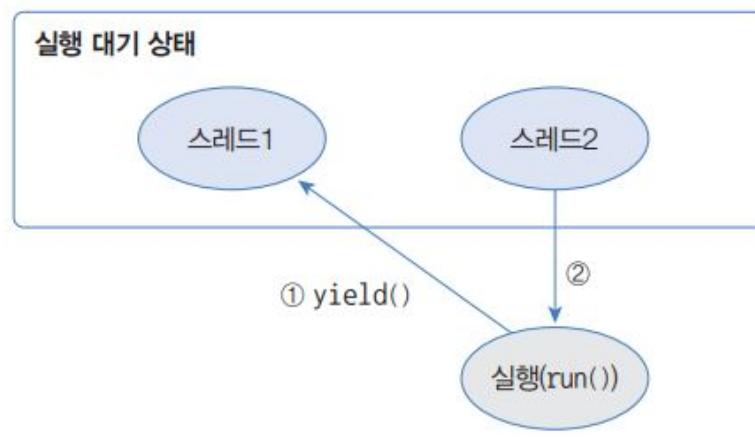
```



구분	메소드	설명
일시 정지로 보냄	sleep(long millis)	주어진 시간 동안 스레드를 일시 정지 상태로 만든다. 주어진 시간이 지나면 자동적으로 실행 대기 상태가 된다.
	join()	join() 메소드를 호출한 스레드는 일시 정지 상태가 된다. 실행 대기 상태가 되려면, join() 메소드를 가진 스레드가 종료되어야 한다.
	wait()	동기화 블록 내에서 스레드를 일시 정지 상태로 만든다.
일시 정지에서 벗어남	interrupt()	일시 정지 상태일 경우, InterruptedException을 발생시켜 실행 대기 상태 또는 종료 상태로 만든다.
	notify(), notifyAll()	wait() 메소드로 인해 일시 정지 상태인 스레드를 실행 대기 상태로 만든다.
실행 대기로 보냄	yield()	실행 상태에서 다른 스레드에게 실행을 양보하고 실행 대기 상태가 된다.

다른 스레드에게 실행 양보

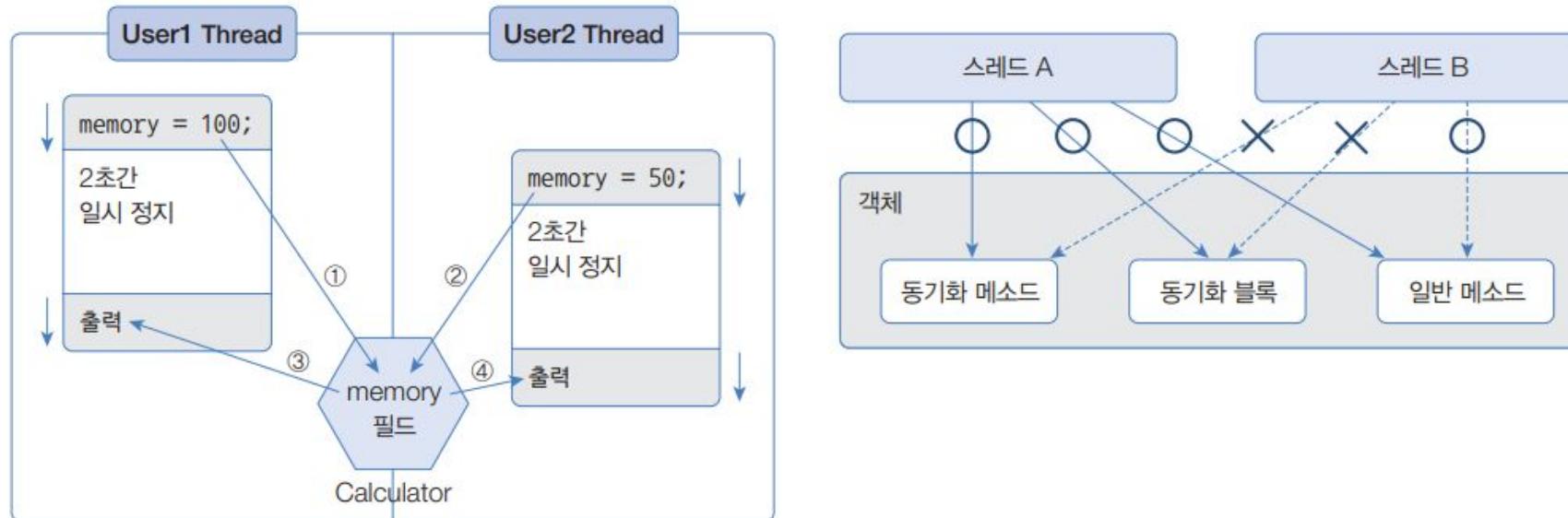
- `yield()` 메소드: 실행되는 스레드는 실행 대기 상태로 돌아가고, 다른 스레드가 실행되도록 양보
- 무의미한 반복을 막아 프로그램 성능 향상



```
public void run() {  
    while(true) {  
        if(work) {  
            System.out.println("ThreadA 작업 내용");  
        } else {  
            Thread.yield();  
        }  
    }  
}
```

동기화 메소드와 블록

- 스레드 작업이 끝날 때까지 객체에 잠금을 걸어 스레드가 사용 중인 객체를 다른 스레드가 변경할 수 없게 함



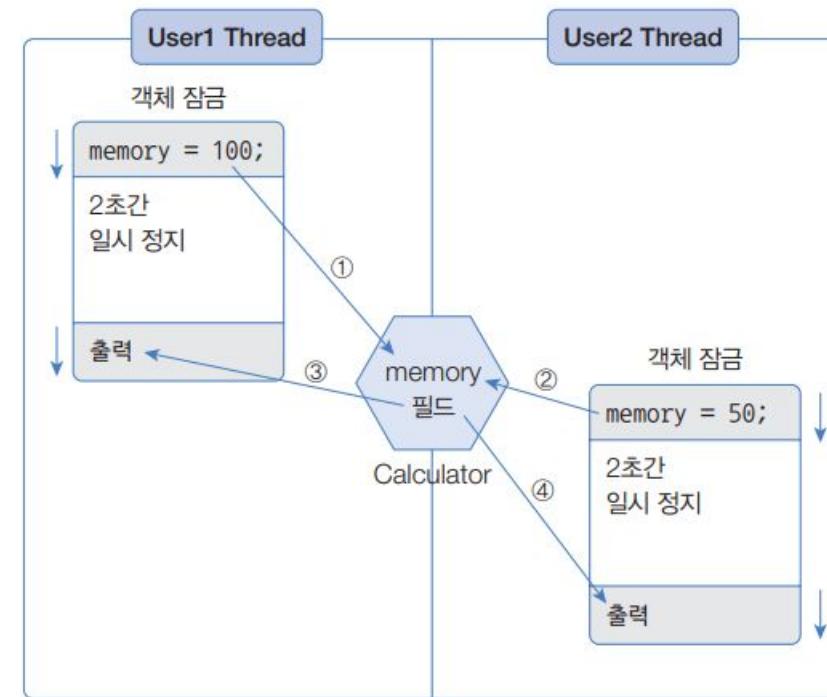
동기화 메소드 및 블록 선언

- 인스턴스와 정적 메소드에 synchronized 키워드 붙임
- 동기화 메소드를 실행 즉시 객체는 잠금이 일어나고, 메소드 실행이 끝나면 잠금 풀림
- 메소드 일부 영역 실행 시 객체 잠금을 걸고 싶다면 동기화 블록을 만듦

```
public synchronized void method() {
    //단 하나의 스레드만 실행하는 영역
}
```

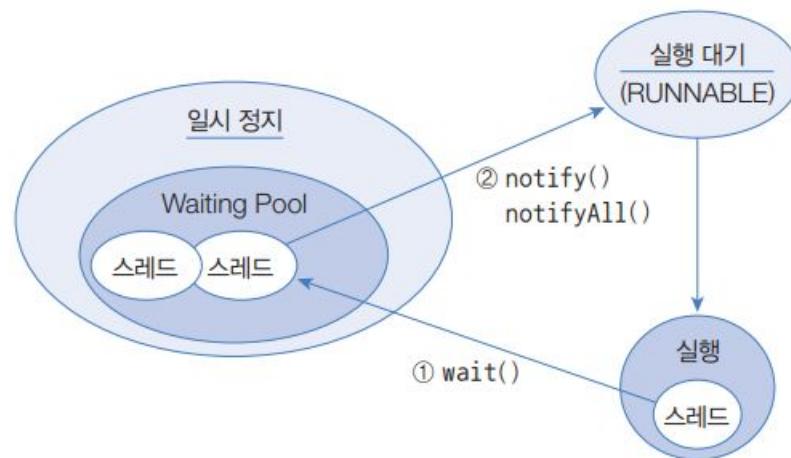
```
public void method () {
    //여러 스레드가 실행할 수 있는 영역

    synchronized(공유객체) {
```



wait()과 notify()를 이용한 스레드 제어

- 두 스레드 교대 실행 시 공유 객체는 두 스레드가 작업할 내용을 각각 동기화 메소드로 정함
- 한 스레드 작업 완료 시 notify() 메소드를 호출해 일시 정지 상태에 있는 다른 스레드를 실행 대기 상태로 만들고, wait() 메소드를 호출하여 자신은 일시 정지 상태로 만듦



- `notify()`: `wait()`에 의해 일시 정지된 스레드 중 한 개를 실행 대기 상태로 만듦
- `notifyAll()`: `wait()`에 의해 일시 정지된 모든 스레드를 실행 대기 상태로 만듦

안전하게 스레드 종료하기

- 스레드 강제 종료 stop() 메소드: deprecated(더 이상 사용하지 않음)
- 스레드를 안전하게 종료하려면 사용하던 리소스(파일, 네트워크 연결)를 정리하고 run() 메소드를 빨리 종료해야 함

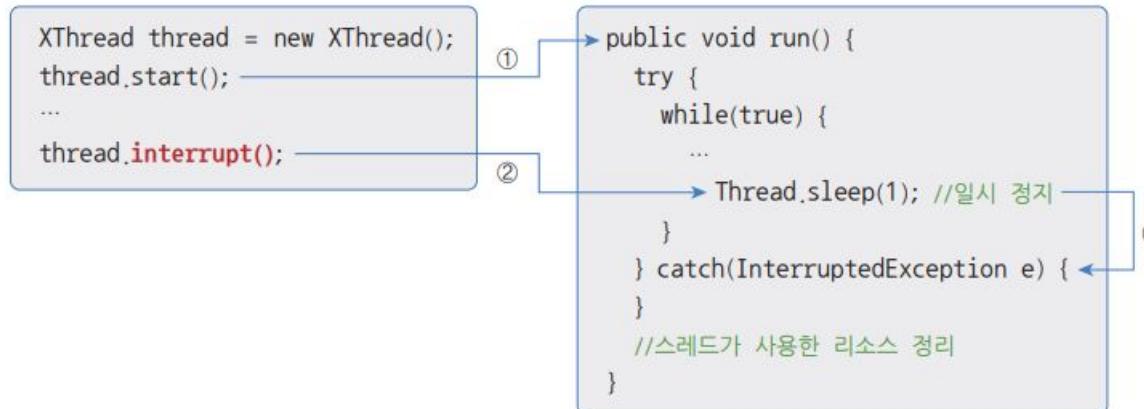
조건 이용

- while 문으로 반복 실행 시 조건을 이용해 run() 메소드 종료를 유도

```
public class XXXThread extends Thread {  
    private boolean stop; •----- stop이 필드 선언  
  
    public void run() {  
        while( !stop ) { •----- stop이 true가 되면 while 문을 빠져나감  
            //스레드가 반복 실행하는 코드;  
        }  
        //스레드가 사용한 리소스 정리 •----- 리소스 정리  
    } •----- 스레드 종료  
}
```

interrupt() 메소드 이용

- 스레드가 일시 정지 상태에 있을 때 InterruptedException 예외 발생
- 예외 처리를 통해 run() 메소드를 정상 종료



- `Thread`의 `interrupted()`와 `isInterrupted()` 메소드는 `interrupt()` 메소드 호출 여부를 리턴

```
boolean status = Thread.interrupted();
boolean status = objThread.isInterrupted();
```

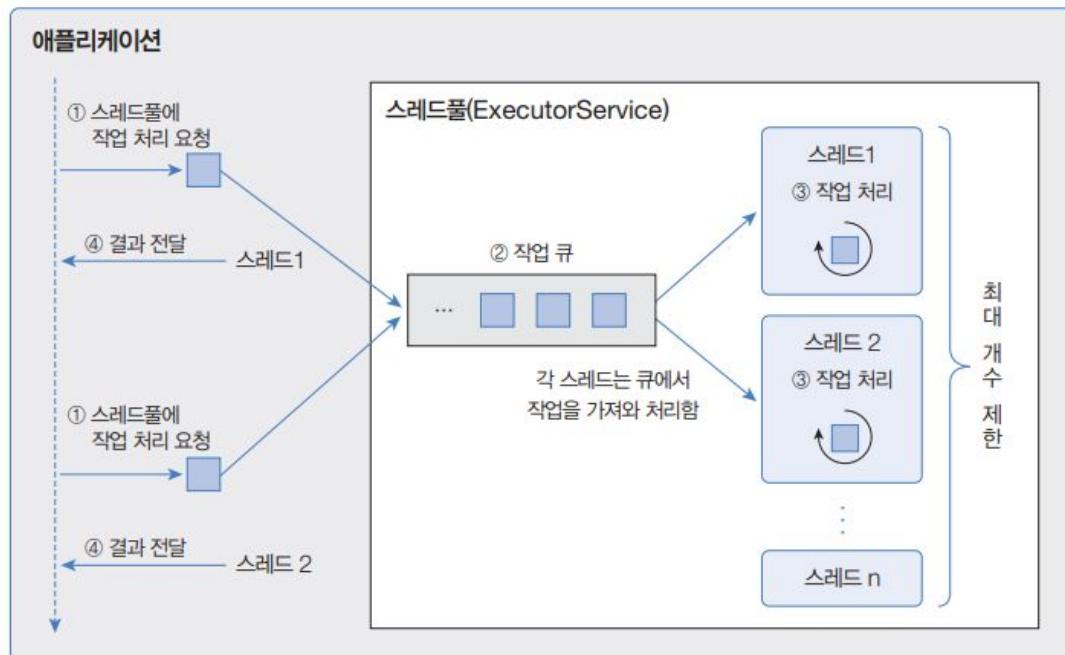
데몬 스레드

- 주 스레드의 작업을 돋는 보조적인 역할을 수행하는 스레드
- 주 스레드가 종료되면 데몬 스레드도 따라서 자동 종료
- 데몬 스레드를 적용 예: 워드프로세서의 자동 저장, 미디어플레이어의 동영상 및 음악 재생, 가비지 컬렉터
- 주 스레드가 데몬이 될 스레드의 `setDaemon(true)`를 호출

```
public static void main(String[] args) {  
    AutoSaveThread thread = new AutoSaveThread();  
    thread.setDaemon(true);  
    thread.start();  
    ...  
}
```

스레드풀로 작업 처리 제한하기

- 작업 처리에 사용되는 스레드 개수를 제한하고 작업 큐에 들어오는 작업들을 스레드가 하나씩 맡아 처리하는 방식
- 작업 처리가 끝난 스레드는 다시 작업 큐에서 새로운 작업을 가져와 처리
- 작업량이 증가해도 스레드의 개수가 늘어나지 않아 애플리케이션의 성능의 급격한 저하 방지



스레드풀 생성

- `java.util.concurrent` 패키지에서 `ExecutorService` 인터페이스와 `Executors` 클래스를 제공
- `Executors`의 다음 두 정적 메소드를 이용하면 스레드풀인 `ExecutorService` 구현 객체를 만들 수

메소드명(매개변수)	초기 수	코어 수	최대 수
<code>newCachedThreadPool()</code>	0	0	<code>Integer.MAX_VALUE</code>
<code>newFixedThreadPool(int nThreads)</code>	0	생성된 수	<code>nThreads</code>

- 초기 수: 스레드풀이 생성될 때 기본적으로 생성되는 스레드 수
- 코어 수: 스레드가 증가된 후 사용되지 않는 스레드를 제거할 때 최소한 풀에서 유지하는 스레드 수

```
ExecutorService executorService = Executors.newCachedThreadPool();
```

```
ExecutorService executorService = Executors.newFixedThreadPool(5);
```

스레드풀 종료

- 스레드풀의 스레드는 main 스레드가 종료되더라도 작업을 처리하기 위해 계속 실행 상태로 남음

리턴 타입	메소드명(매개변수)	설명
void	shutdown()	현재 처리 중인 작업뿐만 아니라 작업 큐에 대기하고 있는 모든 작업을 처리한 뒤에 스레드풀을 종료시킨다.
List<Runnable>	shutdownNow()	현재 작업 처리 중인 스레드를 interrupt해서 작업을 중지시키고 스레드풀을 종료시킨다. 리턴값은 작업 큐에 있는 미처리된 작업(Runnable)의 목록이다.

작업 생성과 처리 요청

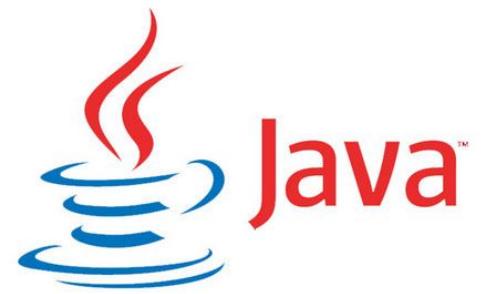
- 하나의 작업은 Runnable 또는 Callable 구현 클래스로 표현

Runnable 익명 구현 클래스	Callable 익명 구현 클래스
<pre>new Runnable() { @Override public void run() { //스레드가 처리할 작업 내용 } }</pre>	<pre>new Callable<T> { @Override public T call() throws Exception { //스레드가 처리할 작업 내용 return T; } }</pre>

- 작업 처리 요청: ExecutorService의 작업 큐에 Runnable 또는 Callable 객체를 넣는 행위

리턴 타입	메소드명(매개변수)	설명
void	execute(Runnable command)	<ul style="list-style-type: none"> – Runnable을 작업 큐에 저장 – 작업 처리 결과를 리턴하지 않음
Future<T>	submit(Callable<T> task)	<ul style="list-style-type: none"> – Callable을 작업 큐에 저장 – 작업 처리 결과를 얻을 수 있도록 Future를 리턴

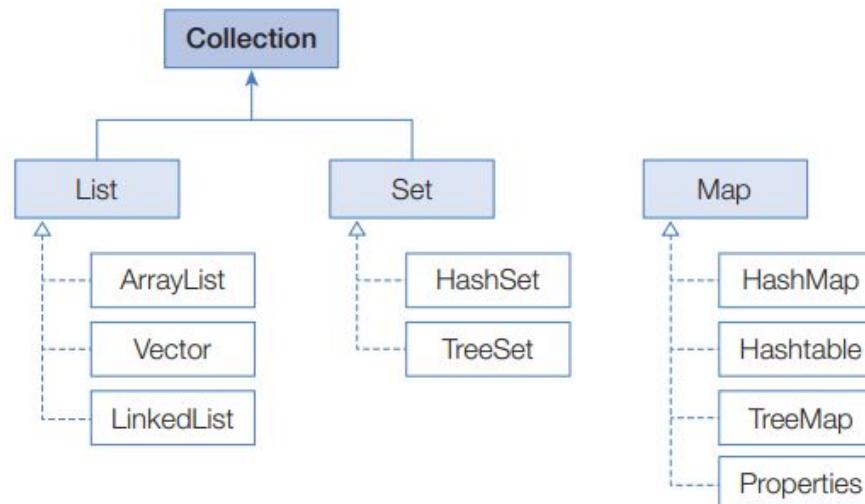




Chapter 15 컬렉션 자료구조

컬렉션 프레임워크

- 널리 알려진 자료구조를 바탕으로 객체들을 효율적으로 추가, 삭제, 검색할 수 있도록 관련 인터페이스와 클래스들을 포함시켜 놓은 `java.util` 패키지
- 주요 인터페이스: `List`, `Set`, `Map`



인터페이스 분류	특징	구현 클래스
Collection	List	- 순서를 유지하고 저장 - 중복 저장 가능
	Set	- 순서를 유지하지 않고 저장 - 중복 저장 안됨
Map	- 키와 값으로 구성된 엔트리 저장 - 키는 중복 저장 안됨	HashMap, Hashtable, TreeMap, Properties

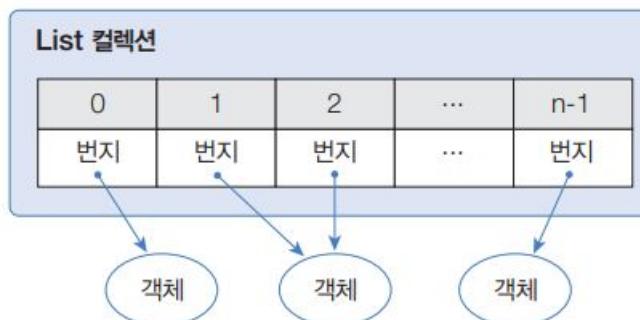
List 컬렉션

- 객체를 인덱스로 관리하기 때문에 객체를 저장하면 인덱스가 부여되고 인덱스로 객체를 검색, 삭제할 수 있는 기능을 제공

기능	메소드	설명
객체 추가	boolean add(E e)	주어진 객체를 맨 끝에 추가
	void add(int index, E element)	주어진 인덱스에 객체를 추가
	set(int index, E element)	주어진 인덱스의 객체를 새로운 객체로 바꿈
객체 검색	boolean contains(Object o)	주어진 객체가 저장되어 있는지 여부
	E get(int index)	주어진 인덱스에 저장된 객체를 리턴
	isEmpty()	컬렉션이 비어 있는지 조사
	int size()	저장되어 있는 전체 객체 수를 리턴
객체 삭제	void clear()	저장된 모든 객체를 삭제
	E remove(int index)	주어진 인덱스에 저장된 객체를 삭제
	boolean remove(Object o)	주어진 객체를 삭제

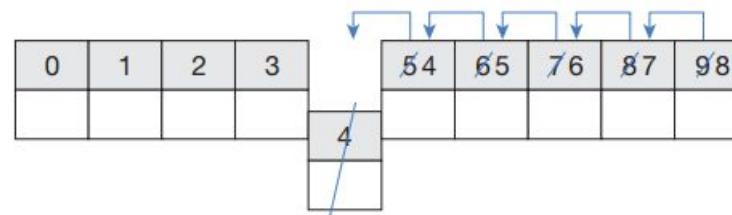
ArrayList

- ArrayList에 객체를 추가하면 내부 배열에 객체가 저장되고 제한 없이 객체를 추가할 수 있음
- 객체의 번지를 저장. 동일한 객체를 중복 저장 시 동일한 번지가 저장. null 저장 가능



```
List<E> list = new ArrayList<E>(); //E에 지정된 타입의 객체만 저장
List<E> list = new ArrayList<>(); //E에 지정된 타입의 객체만 저장
List list = new ArrayList(); //모든 타입의 객체를 저장
```

- ArrayList 컬렉션에 객체를 추가 시 인덱스 0번부터 차례대로 저장
- 특정 인덱스의 객체를 제거하거나 삽입하면 전체가 앞/뒤로 1씩 당겨지거나 밀림
- 빈번한 객체 삭제와 삽입이 일어나는 곳에선 바람직하지 않음



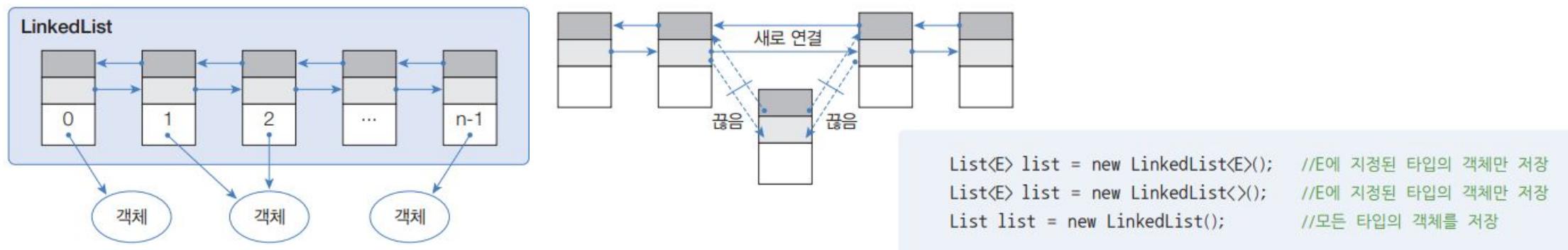
Vector

- 동기화된 메소드로 구성되어 있어 멀티 스레드가 동시에 Vector() 메소드를 실행할 수 없음
- 멀티 스레드 환경에서는 안전하게 객체를 추가 또는 삭제할 수 있음



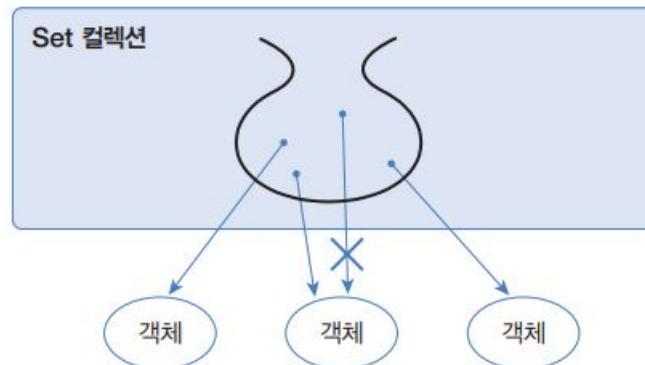
LinkedList

- 인접 객체를 체인처럼 연결해서 관리. 객체 삭제와 삽입이 빈번한 곳에서 ArrayList보다 유리



Set 컬렉션

- Set 컬렉션은 저장 순서가 유지되지 않음
- 객체를 중복해서 저장할 수 없고, 하나의 null만 저장할 수 있음(수학의 집합 개념)

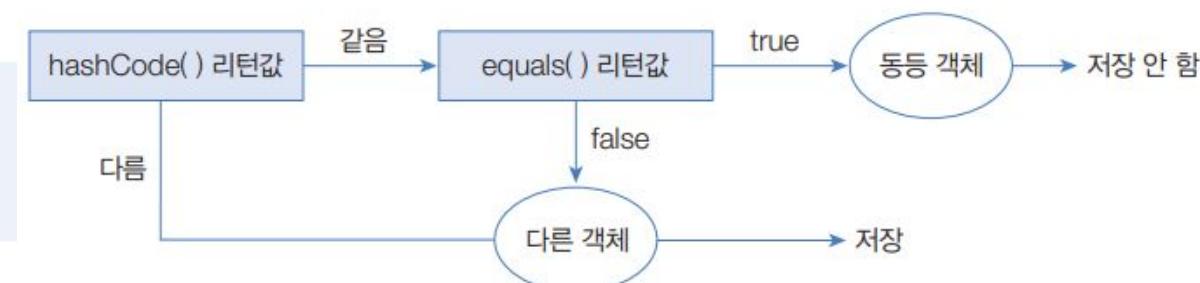


기능	메소드	설명
객체 추가	boolean add(E e)	주어진 객체를 성공적으로 저장하면 true를 리턴하고 중복 객체면 false를 리턴
객체 검색	boolean contains(Object o)	주어진 객체가 저장되어 있는지 여부
	isEmpty()	컬렉션이 비어 있는지 조사
	Iterator<E> iterator()	저장된 객체를 한 번씩 가져오는 반복자 리턴
	int size()	저장되어 있는 전체 객체 수 리턴
객체 삭제	void clear()	저장된 모든 객체를 삭제
	boolean remove(Object o)	주어진 객체를 삭제

HashSet

- 동등 객체를 중복 저장하지 않음
- 다른 객체라도 hashCode() 메소드의 리턴값이 같고, equals() 메소드가 true를 리턴하면 동일한 객체라고 판단하고 중복 저장하지 않음

```
Set<E> set = new HashSet<E>(); //E에 지정된 타입의 객체만 저장
Set<E> set = new HashSet<>(); //E에 지정된 타입의 객체만 저장
Set set = new HashSet(); //모든 타입의 객체를 저장
```



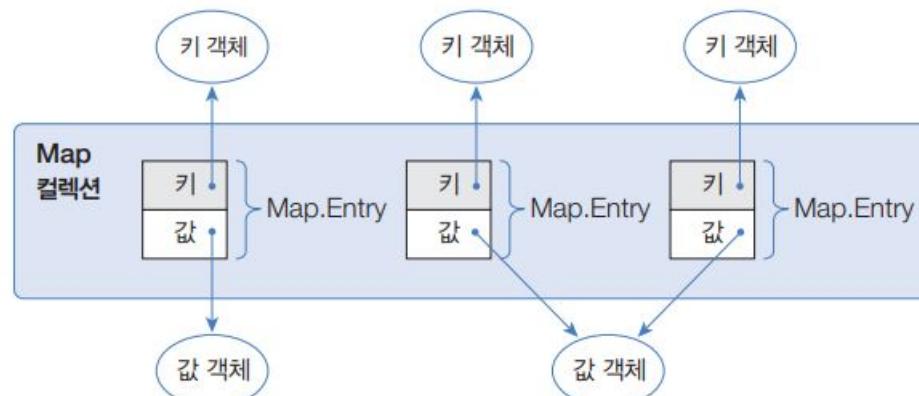
- iterator() 메소드: 반복자를 얻어 Set 컬렉션의 객체를 하나씩 가져옴

```
Set<E> set = new HashSet<>();
Iterator<E> iterator = set.iterator();
```

리턴 타입	메소드명	설명
boolean	hasNext()	가져올 객체가 있으면 true를 리턴하고 없으면 false를 리턴한다.
E	next()	컬렉션에서 하나의 객체를 가져온다.
void	remove()	next()로 가져온 객체를 Set 컬렉션에서 제거한다.

Map 컬렉션

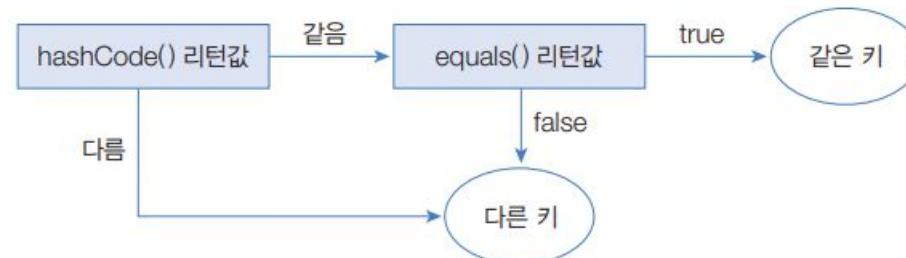
- 키와 값으로 구성된 엔트리 객체를 저장
- 키는 중복 저장할 수 없지만 값은 중복 저장할 수 있음. 기존에 저장된 키와 동일한 키로 값을 저장하면 새로운 값으로 대치



기능	메소드	설명
객체 추가	V put(K key, V value)	주어진 키와 값을 추가, 저장이 되면 값을 리턴
	boolean containsKey(Object key)	주어진 키가 있는지 여부
	boolean containsValue(Object value)	주어진 값이 있는지 여부
	Set<Map.Entry<K,V>> entrySet()	키와 값의 쌍으로 구성된 모든 Map.Entry 객체를 Set에 담아서 리턴
	V get(Object key)	주어진 키의 값을 리턴
	boolean isEmpty()	컬렉션이 비어있는지 여부
	Set<K> keySet()	모든 키를 Set 객체에 담아서 리턴
	int size()	저장된 키의 총 수를 리턴
객체 검색	Collection<V> values()	저장된 모든 값 Collection에 담아서 리턴
	void clear()	모든 Map.Entry(키와 값)를 삭제
객체 삭제	V remove(Object key)	주어진 키와 일치하는 Map.Entry 삭제, 삭제가 되면 값을 리턴

HashMap

- 키로 사용할 객체가 hashCode() 메소드의 리턴값이 같고 equals() 메소드가 true를 리턴할 경우 동일 키로 보고 중복 저장을 허용하지 않음



Hashtable

- 동기화된 메소드로 구성되어 있어 멀티 스레드가 동시에 Hashtable의 메소드들을 실행 불가
- 멀티 스레드 환경에서도 안전하게 객체를 추가, 삭제할 수 있다.



```

Map<String, Integer> map = new Hashtable<String, Integer>();
Map<String, Integer> map = new Hashtable<>();
  
```

This diagram shows two lines of Java code: `Map<String, Integer> map = new Hashtable<String, Integer>();` and `Map<String, Integer> map = new Hashtable<>();`. Blue arrows point from the type parameters `String` and `Integer` in the first line to the corresponding "키 타입" (key type) and "값 타입" (value type) boxes. Similar arrows point from the type parameters in the second line to their respective boxes.

Properties

- Properties는 Hashtable의 자식 클래스. 키와 값을 String 타입으로 제한한 컬렉션
- 주로 확장자가 .properties인 프로퍼티 파일을 읽을 때 사용
- 프로퍼티 파일은 키와 값이 = 기호로 연결된 텍스트 파일(ISO 8859-1 문자셋, 한글은 $\text{\u}+유니코드$)

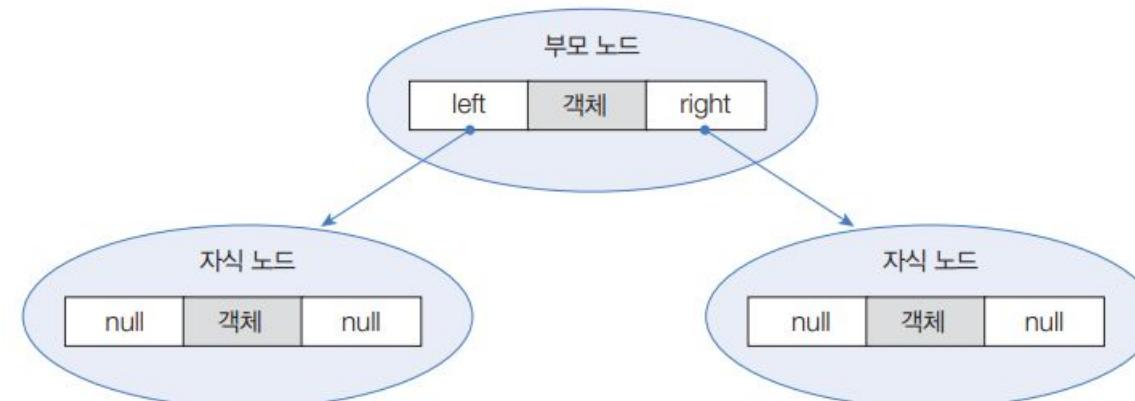
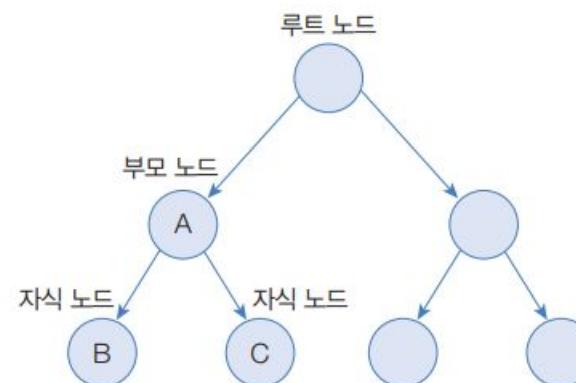
```
>>> database.properties
```

```
1 driver=oracle.jdbc.OracleDirver
2 url=jdbc:oracle:thin:@localhost:1521:orcl
3 username=scott
4 password=tiger
5 admin=\uD64D\uAE38\uB3D9
```

```
Properties properties = new Properties();
properties.load(Xxx.class.getResourceAsStream("database.properties"));
```

TreeSet

- 이진 트리를 기반으로 한 Set 컬렉션
- 여러 개의 노드가 트리 형태로 연결된 구조. 루트 노드에서 시작해 각 노드에 최대 2개의 노드를 연결할 수 있음
- TreeSet에 객체를 저장하면 부모 노드의 객체와 비교해서 낮은 것은 왼쪽 자식 노드에, 높은 것은 오른쪽 자식 노드에 저장



TreeSet 컬렉션을 생성하는 방법

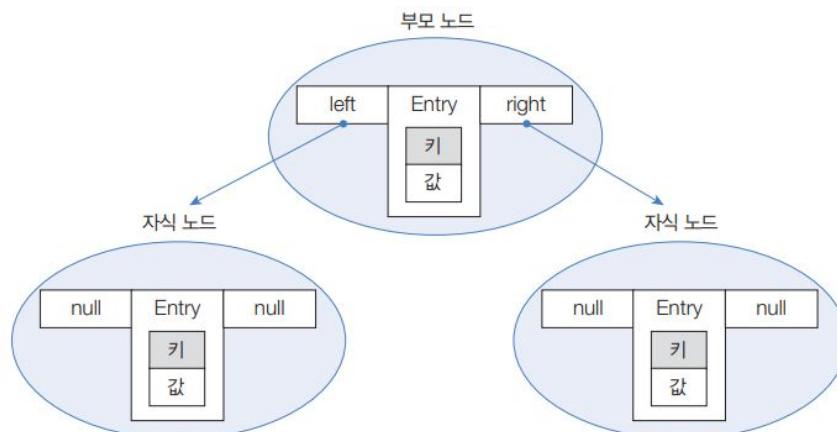
```
TreeSet<E> treeSet = new TreeSet<E>();
TreeSet<E> treeSet = new TreeSet<>();
```

- Set 타입 변수에 대입해도 되지만 TreeSet 타입으로 대입한 이유는 검색 관련 메소드가 TreeSet에만 정의되어 있기 때문

리턴 타입	메소드	설명
E	first()	제일 낮은 객체를 리턴
E	last()	제일 높은 객체를 리턴
E	lower(E e)	주어진 객체보다 바로 아래 객체를 리턴
E	higher(E e)	주어진 객체보다 바로 위 객체를 리턴
E	floor(E e)	주어진 객체와 동등한 객체가 있으면 리턴, 만약 없다면 주어진 객체의 바로 아래의 객체를 리턴
E	ceiling(E e)	주어진 객체와 동등한 객체가 있으면 리턴, 만약 없다면 주어진 객체의 바로 위의 객체를 리턴
E	pollFirst()	제일 낮은 객체를 꺼내오고 컬렉션에서 제거함
E	pollLast()	제일 높은 객체를 꺼내오고 컬렉션에서 제거함
Iterator<E>	descendingIterator()	내림차순으로 정렬된 Iterator를 리턴
NavigableSet<E>	descendingSet()	내림차순으로 정렬된 NavigableSet을 리턴
NavigableSet<E>	headSet(E toElement, boolean inclusive)	주어진 객체보다 낮은 객체들을 NavigableSet으로 리턴. 주어진 객체 포함 여부는 두 번째 매개값에 따라 달라짐
NavigableSet<E>	tailSet(E fromElement, boolean inclusive)	주어진 객체보다 높은 객체들을 NavigableSet으로 리턴. 주어진 객체 포함 여부는 두 번째 매개값에 따라 달라짐
NavigableSet<E>	subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)	시작과 끝으로 주어진 객체 사이의 객체들을 NavigableSet으로 리턴. 시작과 끝 객체의 포함 여부는 두 번째, 네 번째 매개값에 따라 달라짐

TreeMap

- 이진 트리를 기반으로 한 Map 컬렉션. 키와 값이 저장된 엔트리 저장
- 부모 키 값과 비교해서 낮은 것은 왼쪽, 높은 것은 오른쪽 자식 노드에 Entry 객체를 저장



```
TreeMap<K, V> treeMap = new TreeMap<K, V>();
TreeMap<K, V> treeMap = new TreeMap<>();
```

리턴 타입	메소드	설명
Map.Entry<K,V>	firstEntry()	제일 낮은 Map.Entry를 리턴
Map.Entry<K,V>	lastEntry()	제일 높은 Map.Entry를 리턴
Map.Entry<K,V>	lowerEntry(K key)	주어진 키보다 바로 아래 Map.Entry를 리턴
Map.Entry<K,V>	higherEntry(K key)	주어진 키보다 바로 위 Map.Entry를 리턴
Map.Entry<K,V>	floorEntry(K key)	주어진 키와 동등한 키가 있으면 해당 Map.Entry를 리턴, 없다면 주어진 키 바로 아래의 Map.Entry를 리턴
Map.Entry<K,V>	ceilingEntry(K key)	주어진 키와 동등한 키가 있으면 해당 Map.Entry를 리턴, 없다면 주어진 키 바로 위의 Map.Entry를 리턴
Map.Entry<K,V>	pollFirstEntry()	제일 낮은 Map.Entry를 꺼내오고 컬렉션에서 제거함
Map.Entry<K,V>	pollLastEntry()	제일 높은 Map.Entry를 꺼내오고 컬렉션에서 제거함
NavigableSet<K>	descendingKeySet()	내림차순으로 정렬된 키의 NavigableSet을 리턴
NavigableMap<K,V>	descendingMap()	내림차순으로 정렬된 Map.Entry의 NavigableMap을 리턴
NavigableMap<K,V>	headMap(K toKey, boolean inclusive)	주어진 키보다 낮은 Map.Entry들을 NavigableMap으로 리턴. 주어진 키의 Map.Entry 포함 여부는 두 번째 매개값에 따라 달라짐
NavigableMap<K,V>	tailMap(K fromKey, boolean inclusive)	주어진 객체보다 높은 Map.Entry들을 NavigableSet으로 리턴. 주어진 객체 포함 여부는 두 번째 매개값에 따라 달라짐
NavigableMap<K,V>	subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)	시작과 끝으로 주어진 키 사이의 Map.Entry들을 NavigableMap 컬렉션으로 반환. 시작과 끝 키의 Map.Entry 포함 여부는 두 번째, 네 번째 매개값에 따라 달라짐

Comparable과 Comparator

- TreeSet에 저장되는 객체와 TreeMap에 저장되는 키 객체를 정렬
- Comparable 인터페이스에는 compareTo() 메소드가 정의. 사용자 정의 클래스에서 이 메소드를 재정의해서 비교 결과를 정수 값으로 리턴

리턴 타입	메소드	설명
int	compareTo(T o)	주어진 객체와 같으면 0을 리턴 주어진 객체보다 적으면 음수를 리턴 주어진 객체보다 크면 양수를 리턴

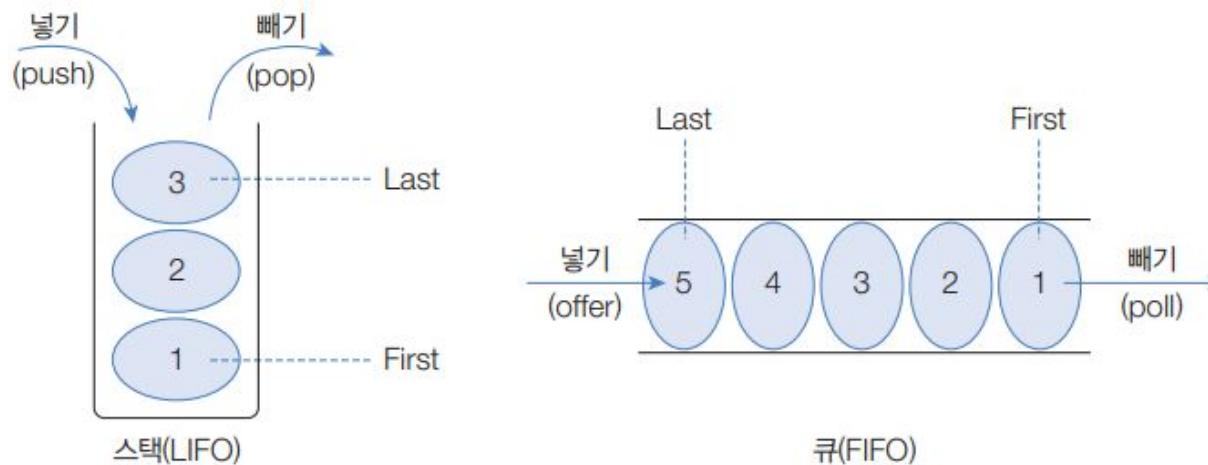
- 비교 기능이 없는 Comparable 비구현 객체를 저장하려면 비교자 Comparator를 제공
- 비교자는 compare() 메소드를 재정의해서 비교 결과를 정수 값으로 리턴

```
TreeSet<E> treeSet = new TreeSet<E>( new ComparatorImpl() );
    ↓
    비교자
    ↓
TreeMap<K,V> treeMap = new TreeMap<K,V>( new ComparatorImpl() );
```

리턴 타입	메소드	설명
int	compare(T o1, T o2)	o1과 o2가 동등하다면 0을 리턴 o1이 o2보다 앞에 오게 하려면 음수를 리턴 o1이 o2보다 뒤에 오게 하려면 양수를 리턴

후입선출과 선입선출

- 후입선출(LIFO): 나중에 넣은 객체가 먼저 빠져나가는 구조
- 선입선출(FIFO): 먼저 넣은 객체가 먼저 빠져나가는 구조
- 컬렉션 프레임워크는 LIFO 자료구조를 제공하는 스택 클래스와 FIFO 자료구조를 제공하는 큐 인터페이스를 제공



Stack

- Stack 클래스: LIFO 자료구조를 구현한 클래스

```
Stack<E> stack = new Stack<E>();
Stack<E> stack = new Stack<>();
```

리턴 타입	메소드	설명
E	push(E item)	주어진 객체를 스택에 넣는다.
E	pop()	스택의 맨 위 객체를 빼낸다.

Queue

- Queue 인터페이스: FIFO 자료구조에서 사용되는 메소드를 정의
- LinkedList: Queue 인터페이스를 구현한 대표적인 클래스

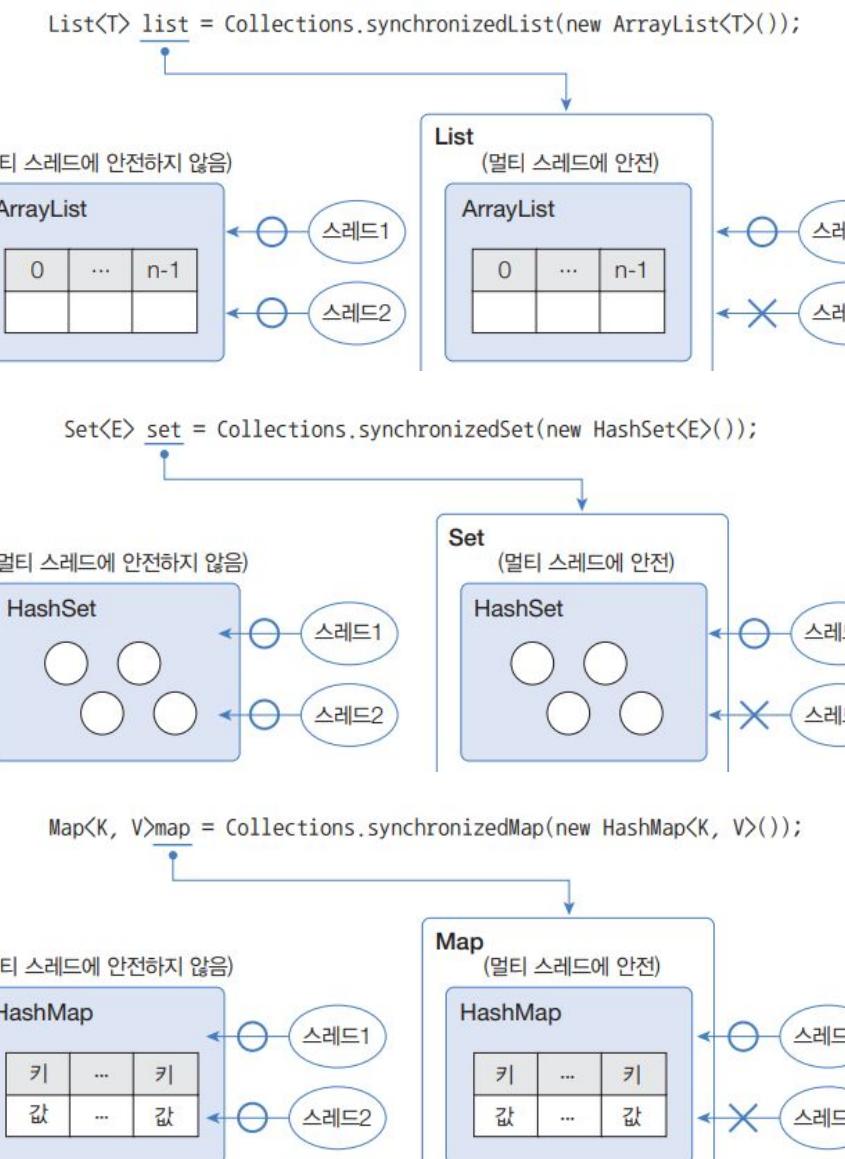
리턴 타입	메소드	설명
boolean	offer(E e)	주어진 객체를 큐에 넣는다.
E	poll()	큐에서 객체를 빼낸다.

```
Queue<E> queue = new LinkedList<E>();
Queue<E> queue = new LinkedList<>();
```

동기화된 컬렉션

- 동기화된 메소드로 구성된 Vector와 Hashtable은 멀티 스레드 환경에서 안전하게 요소를 처리
- Collections의 synchronizedXXX() 메소드: ArrayList, HashSet, HashMap 등 비동기화된 메소드를 동기화된 메소드로 래핑

리턴 타입	메소드(매개변수)	설명
List<T>	synchronizedList(List<T> list)	List를 동기화된 List로 리턴
Map<K,V>	synchronizedMap(Map<K,V> m)	Map을 동기화된 Map으로 리턴
Set<T>	synchronizedSet(Set<T> s)	Set을 동기화된 Set으로 리턴



수정할 수 없는 컬렉션

- 요소를 추가, 삭제할 수 없는 컬렉션. 컬렉션 생성 시 저장된 요소를 변경하고 싶지 않을 때 유용
- List, Set, Map 인터페이스의 정적

메소드인 `of()`로 생성

- List, Set, Map 인터페이스의 정적

메소드인 `copyOf()`을 이용해 기존

컬렉션을 복사

- 배열로부터 수정할 수 없는 List

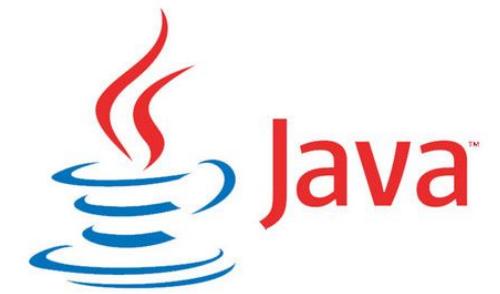
컬렉션을 만듦

```
List<E> immutableList = List.of(E... elements);
Set<E> immutableSet = Set.of(E... elements);
Map<K,V> immutableMap = Map.of( K k1, V v1, K k2, V v2, ... );
```

```
List<E> immutableList = List.copyOf(Collection<E> coll);
Set<E> immutableSet = Set.copyOf(Collection<E> coll);
Map<K,V> immutableMap = Map.copyOf(Map<K,V> map);
```

```
String[] arr = { "A", "B", "C" };
List<String> immutableList = Arrays.asList(arr);
```

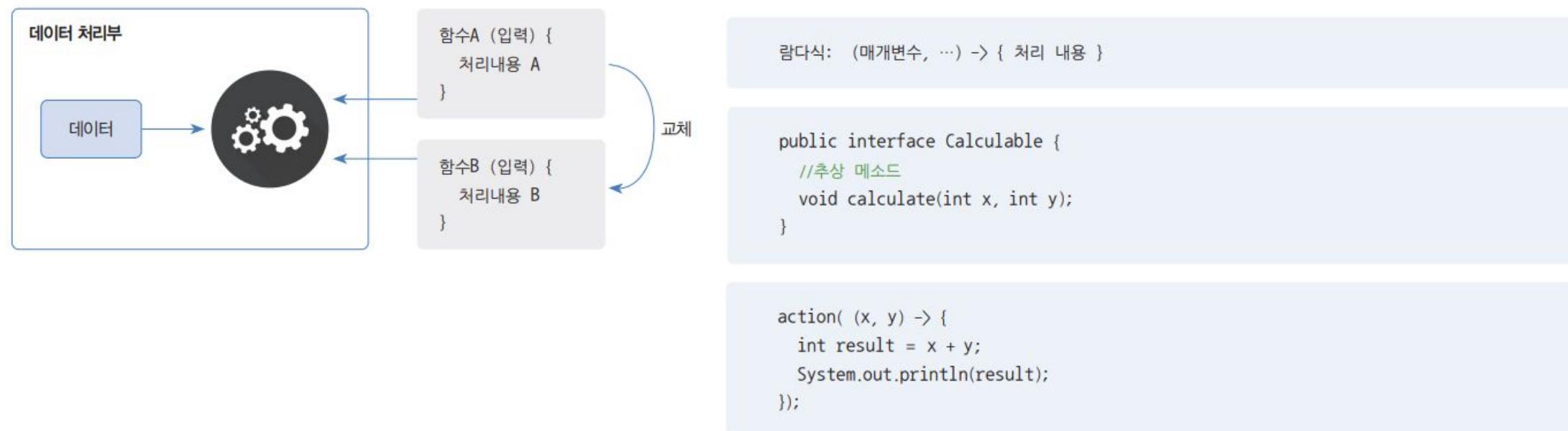




Chapter 16 람다식

람다식

- 함수형 프로그래밍: 함수를 정의하고 이 함수를 데이터 처리부로 보내 데이터를 처리하는 기법
- 데이터 처리부는 제공된 함수의 입력값으로 데이터를 넣고 함수에 정의된 처리 내용을 실행
- 람다식: 데이터 처리부에 제공되는 함수 역할을 하는 매개변수를 가진 중괄호 블록이다.
- 자바는 람다식을 익명 구현 객체로 변환



함수형 인터페이스

- 인터페이스가 단 하나의 추상 메소드를 가지는 것

인터페이스

```
public interface Runnable {  
    void run();  
}
```

람다식

```
( ) -> { ... }
```

인터페이스

```
@FunctionalInterface  
public interface Calculable {  
    void calculate(int x, int y);  
}
```

람다식

```
( x, y ) -> { ... }
```

- 인터페이스가 함수형 인터페이스임을 보장하기 위해서는 `@FunctionalInterface` 어노테이션을 붙임
- `@FunctionalInterface`: 컴파일 과정에서 추상 메소드가 하나인지 검사해 정확한 함수형 인터페이스를 작성할 수 있게 도와주는 역할

매개변수가 없는 람다식

- 함수형 인터페이스의 추상 메소드에
매개변수가 없을 경우 람다식 작성하기
- 실행문이 두 개 이상일 경우에는 중괄호를
생략할 수 없고, 하나일 경우에만 생략할 수
있음

```
( ) -> {
    실행문;
    실행문;
}
```

```
( ) -> 실행문
```

```

1 package ch16.sec02.exam02;
2
3 public class ButtonExample {
4     public static void main(String[] args) {
5         //Ok 버튼 객체 생성
6         Button btnOk = new Button();
7
8         //Ok 버튼 객체에 람다식(ClickListener 익명 구현 객체) 주입
9         btnOk.setOnClickListener(() -> {
10             System.out.println("Ok 버튼을 클릭했습니다.");
11         });
12
13         //Ok 버튼 클릭하기
14         btnOk.click();
15
16         //Cancel 버튼 객체 생성
17         Button btnCancel = new Button();
18
19         //Cancel 버튼 객체에 람다식(ClickListener 익명 구현 객체) 주입
20         btnCancel.setOnClickListener(() -> {
21             System.out.println("Cancel 버튼을 클릭했습니다.");
22         });
23
24         //Cancel 버튼 클릭하기
25         btnCancel.click();
26     }
27 }
```

매개값으로
람다식 대입

매개값으로
람다식 대입

매개변수가 있는 람다식

- 함수형 인터페이스의 추상 메소드에 매개변수가 있을 경우 람다식 작성하기
- 매개변수를 선언할 때 타입은 생략할 수 있고, 구체적인 타입 대신에 var를 사용할 수 있음

(타입 매개변수, ...) -> {
 실행문;
 실행문;
}

(var 매개변수, ...) -> {
 실행문;
 실행문;
}

(매개변수, ...) -> {
 실행문;
 실행문;
}

(타입 매개변수, ...) -> 실행문

(var 매개변수, ...) -> 실행문

(매개변수, ...) -> 실행문

- 매개변수가 하나일 경우에는 괄호를 생략 가능. 이때는 타입 또는 var를 붙일 수 없음

매개변수 -> {
 실행문;
 실행문;
}

매개변수 -> 실행문

리턴값이 있는 람다식

- 함수형 인터페이스의 추상 메소드에 리턴값이 있을 경우 람다식 작성하기
- return 문 하나만 있을 경우에는 중괄호와 함께 return 키워드를 생략 가능
- 리턴값은 연산식 또는 리턴값 있는 메소드 호출로 대체 가능

```
(매개변수, ...) -> {  
    실행문;  
    return 값;  
}
```

```
(매개변수, ...) -> return 값;  
(매개변수, ...) -> 값
```

메소드 참조

- 메소드를 참조해 매개변수의 정보 및 리턴 타입을 알아내 람다식에서 불필요한 매개변수를 제거

```
(left, right) -> Math.max(left, right);
```

정적 메소드와 인스턴스 메소드 참조

- 정적 메소드를 참조 시 클래스 이름 뒤에 :: 기호를 붙이고 정적 메소드 이름을 기술

```
클래스 :: 메소드
```

- 인스턴스 메소드일 경우에는 객체를 생성한 다음 참조 변수 뒤에 :: 기호를 붙이고 인스턴스 메소드 이름을 기술

```
참조변수 :: 메소드
```

매개변수의 메소드 참조

- 람다식에서 제공되는 a 매개변수의 메소드를 호출해서 b 매개변수를 매개값으로 사용

```
(a, b) -> { a.instanceMethod(b); }
```

- a의 클래스 이름 뒤에 :: 기호를 붙이고 메소드 이름을 기술

클래스 :: instanceMethod

```
1 package ch16.sec05.exam02;  
2  
3 public class MethodReferenceExample {  
4     public static void main(String[] args) {  
5         Person person = new Person();          (a, b) -> a.compareToIgnoreCase(b)  
6         person.ordering(String :: compareToIgnoreCase);  
7     }  
8 }
```

메소드 참조

생성자 참조

- 객체를 생성하는 것. 람다식이 단순히 객체를 생성하고 리턴하도록 구성되면 람다식을 생성자 참조로 대체 가능

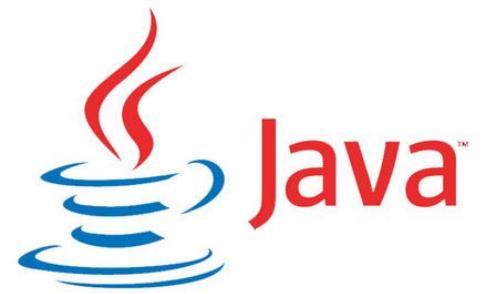
```
(a, b) -> { return new 클래스(a, b); }
```

- 클래스 이름 뒤에 :: 기호를 붙이고 new 연산자를 기술

```
클래스 :: new
```

- 생성자가 오버로딩되어 여러 개가 있을 경우, 컴파일러는 함수형 인터페이스의 추상 메소드와 동일한 매개변수 타입과 개수를 가지고 있는 생성자를 찾아 실행
- 해당 생성자가 존재하지 않으면 컴파일 오류 발생





Chapter 17 스트림 요소 처리

스트림

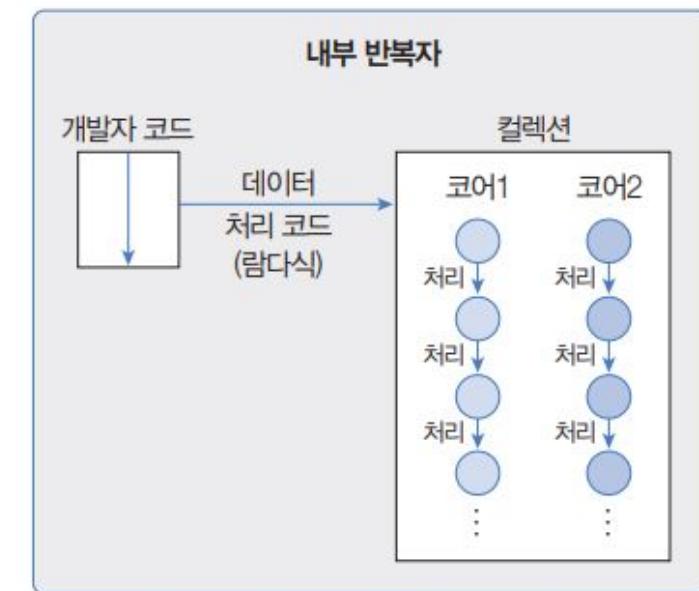
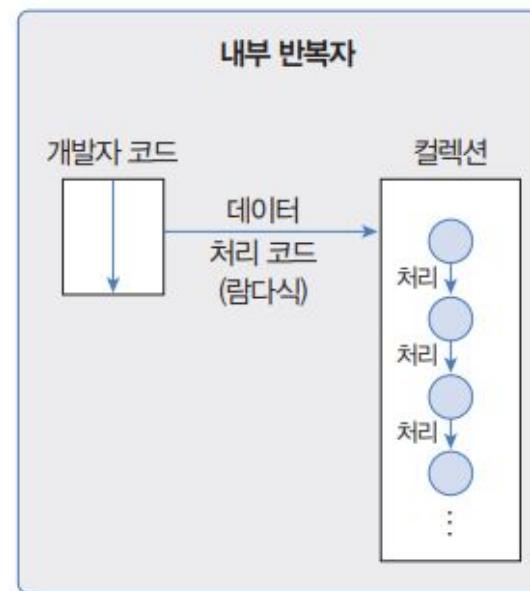
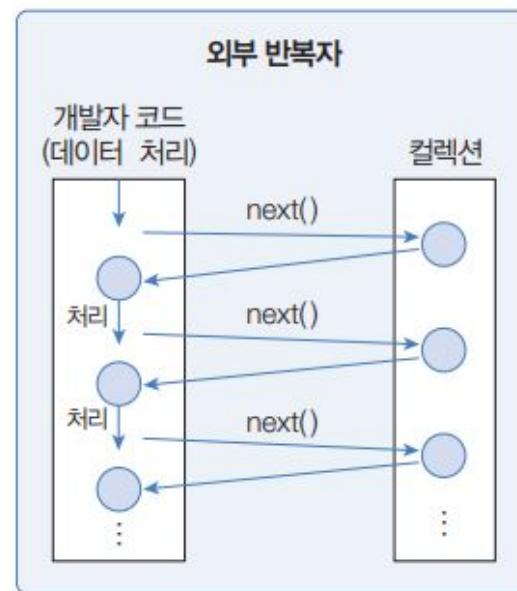
- Java 8부터 컬렉션 및 배열의 요소를 반복 처리하기 위해 스트림 사용
- 요소들이 하나씩 흘러가면서 처리된다는 의미

```
Stream<String> stream = list.stream();
stream.forEach( item -> //item 처리 );
```

- List 컬렉션의 stream() 메소드로 Stream 객체를 얻고, forEach() 메소드로 요소를 어떻게 처리할지를 람다식으로 제공
- 스트림과 Iterator 차이점
 - 1) 내부 반복자이므로 처리 속도가 빠르고 병렬 처리에 효율적
 - 2) 람다식으로 다양한 요소 처리를 정의
 - 3) 중간 처리와 최종 처리를 수행하도록 파이프 라인을 형성

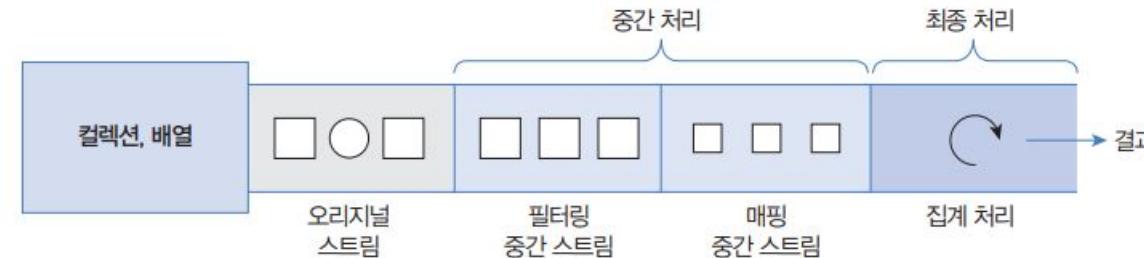
내부 반복자

- 요소 처리 방법을 컬렉션 내부로 주입시켜서 요소를 반복 처리
- 개발자 코드에서 제공한 데이터 처리 코드(람다식)를 가지고 컬렉션 내부에서 요소를 반복 처리
- 내부 반복자는 멀티 코어 CPU를 최대한 활용하기 위해 요소들을 분배시켜 병렬 작업 가능

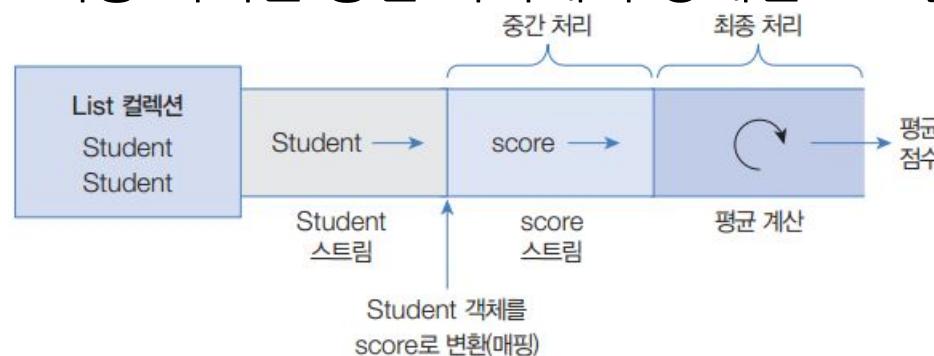


스트림 파이프라인

- 컬렉션의 오리지널 스트림 뒤에 필터링 중간 스트림이 연결될 수 있고, 그 뒤에 맵핑 중간 스트림이 연결될 수 있음

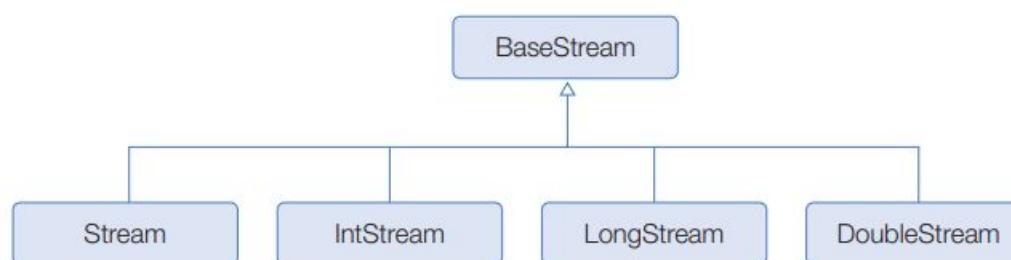


- 오리지널 스트림과 집계 처리 사이의 중간 스트림들은 최종 처리를 위해 요소를 걸러내거나(필터링), 요소를 변환시키거나(맵핑), 정렬하는 작업을 수행
- 최종 처리는 중간 처리에서 정제된 요소들을 반복하거나, 집계(카운팅, 총합, 평균) 작업을 수행



스트림 인터페이스

- java.util.stream 패키지에는 BaseStream 인터페이스를 부모로 한 자식 인터페이스들은 상속 관계
- BaseStream에는 모든 스트림에서 사용할 수 있는 공통 메소드들이 정의



리턴 타입	메소드(매개변수)	소스
<code>Stream<T></code>	<code>java.util.Collection.stream()</code> <code>java.util.Collection.parallelStream()</code>	List 컬렉션 Set 컬렉션
<code>Stream<T></code> <code>IntStream</code> <code>LongStream</code> <code>DoubleStream</code>	<code>Arrays.stream(T[])</code> , <code>Stream.of(T[])</code> <code>Arrays.stream(int[])</code> , <code>IntStream.of(int[])</code> <code>Arrays.stream(long[])</code> , <code>LongStream.of(long[])</code> <code>Arrays.stream(double[])</code> , <code>DoubleStream.of(double[])</code>	배열
<code>IntStream</code>	<code>IntStream.range(int, int)</code> <code>IntStream.rangeClosed(int, int)</code>	int 범위
<code>LongStream</code>	<code>LongStream.range(long, long)</code> <code>LongStream.rangeClosed(long, long)</code>	long 범위
<code>Stream<Path></code>	<code>Files.list(Path)</code>	디렉토리
<code>Stream<String></code>	<code>Files.lines(Path, Charset)</code>	텍스트 파일
<code>DoubleStream</code> <code>IntStream</code> <code>LongStream</code>	<code>Random.doubles(...)</code> <code>Random.ints()</code> <code>Random.longs()</code>	랜덤 수

컬렉션으로부터 스트림 얻기

- `java.util.Collection` 인터페이스는 스트림과 `parallelStream()` 메소드를 가지고 있어 자식 인터페이스인 `List`와 `Set` 인터페이스를 구현한 모든 컬렉션에서 객체 스트림을 얻을 수 있음

배열로부터 스트림 얻기

- `java.util.Arrays` 클래스로 다양한 종류의 배열로부터 스트림을 얻을 수 있음

숫자 범위로부터 스트림 얻기

- `IntStream` 또는 `LongStream`의 정적 메소드인 `range()`와 `rangeClosed()` 메소드로 특정 범위의 정수 스트림을 얻을 수 있음

파일로부터 스트림 얻기

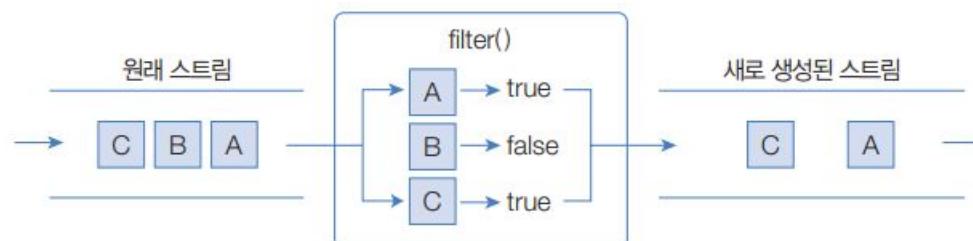
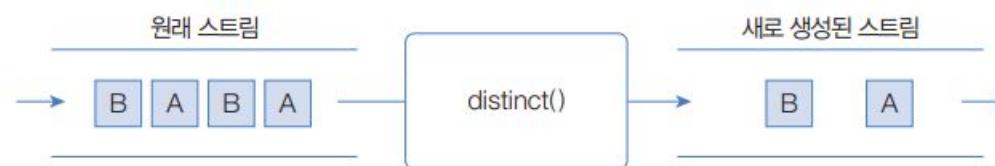
- `java.nio.file.Files`의 `lines()` 메소드로 텍스트 파일의 행 단위 스트림을 얻을 수 있음

필터링

- 필터링은 요소를 걸러내는 중간 처리 기능

리턴 타입	메소드(매개변수)	설명
Stream	distinct()	- 중복 제거
IntStream	filter(Predicate<T>)	- 조건 필터링
LongStream	filter(IntPredicate)	- 매개 타입은 요소 타입에 따른 함수형
DoubleStream	filter(LongPredicate)	인터페이스이므로 람다식으로 작성 가능
	filter(DoublePredicate)	

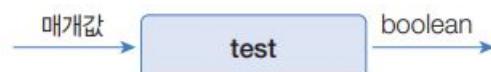
- filter() 메소드: 매개값으로 주어진 Predicate가 true를 리턴하는 요소만 필터링



- Predicate: 함수형 인터페이스

인터페이스	추상 메소드	설명
Predicate<T>	boolean test(T t)	객체 T를 조사
IntPredicate	boolean test(int value)	int 값을 조사
LongPredicate	boolean test(long value)	long 값을 조사
DoublePredicate	boolean test(double value)	double 값을 조사

- 모든 Predicate는 매개값을 조사한 후 boolean을 리턴하는 test() 메소드를 가지고 있다.



$T \rightarrow \{ \dots \text{return true} \}$

또는

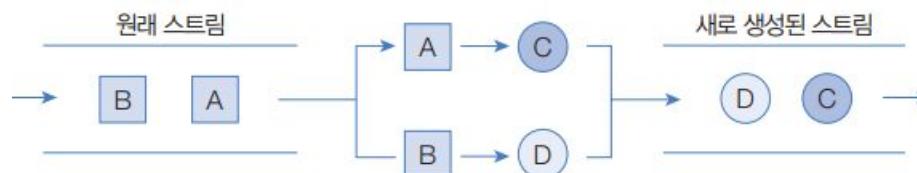
$T \rightarrow \text{true}; \quad //\text{return 문만 있을 경우 중괄호와 return 키워드 생략 가능}$

매팅

- 스트림의 요소를 다른 요소로 변환하는 중간 처리 기능
- 매팅 메소드: mapXxx(), asDoubleStream(), asLongStream(), boxed(), flatMapXxx() 등

요소를 다른 요소로 변환

- mapXxx() 메소드: 요소를 다른 요소로
변환한 새로운 스트림을 리턴

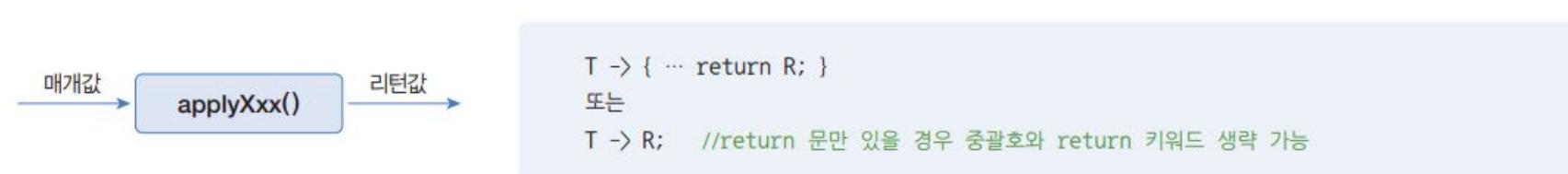


리턴 타입	메소드(매개변수)	요소 → 변환 요소
Stream<R>	map(Function<T, R>)	T → R
IntStream	mapToIntToIntFunction<T>()	T → int
LongStream	mapToLong(ToLongFunction<T>)	T → long
DoubleStream	mapToDouble(ToDoubleFunction<T>)	T → double
Stream<U>	mapToObj(IntFunction<U>)	int → U
	mapToObj(LongFunction<U>)	long → U
	mapToObj(DoubleFunction<U>)	double → U
DoubleStream	map.ToDouble(Int.ToDoubleFunction)	int → double
	map.ToDouble(Long.ToDoubleFunction)	long → double
	map.ToInt(Double.ToIntFunction)	double → int
	map.ToLong(Double.ToLongFunction)	double → long

- 매개타입인 Function은 함수형 인터페이스

인터페이스	추상 메소드	매개값 → 리턴값
Function<T,R>	R apply(T t)	T → R
IntFunction<R>	R apply(int value)	int → R
LongFunction<R>	R apply(long value)	long → R
DoubleFunction<R>	R apply(double value)	double → R
ToIntFunction<T>	int applyAsInt(T value)	T → int
ToLongFunction<T>	long applyAsLong(T value)	T → long
ToDoubleFunction<T>	double applyAsDouble(T value)	T → double
IntToLongFunction	long applyAsLong(int value)	int → long
InttoDoubleFunction	double applyAsDouble(int value)	int → double
LongToIntFunction	int applyAsInt(long value)	long → int
LongtoDoubleFunction	double applyAsDouble(long value)	long → double
DoubleToIntFunction	int applyAsInt(double value)	double → int
DoubleToLongFunction	long applyAsLong(double value)	double → long

- 모든 Function은 매개값을 리턴값으로 매핑(변환)하는 applyXxx() 메소드를 가짐

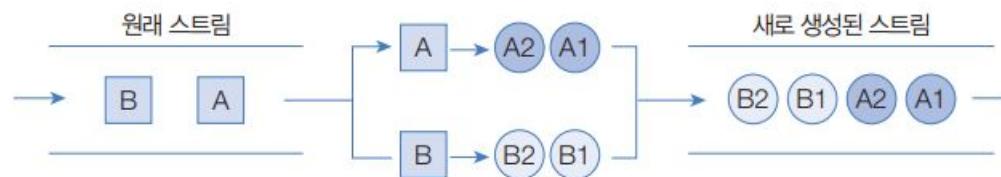


- 기본 타입 간의 변환이거나 기본 타입 요소를 래퍼(Wrapper) 객체 요소로 변환하려면 간편화 메소드를 사용할 수 있음

리턴 타입	메소드(매개변수)	설명
LongStream	asLongStream()	int → long
DoubleStream	asDoubleStream()	int → double long → double
Stream<Integer> Stream<Long> Stream<Double>	boxed()	int → Integer long → Long double → Double

요소를 복수 개의 요소로 변환

- flatMapXxx() 메소드: 하나의 요소를 복수 개의 요소들로 변환한 새로운 스트림을 리턴



리턴 타입	메소드(매개변수)	요소 → 변환 요소
Stream<R>	flatMap(Function<T, Stream<R>>)	T → Stream<R>
DoubleStream	flatMap(DoubleFunction<DoubleStream>)	double → DoubleStream
IntStream	flatMap(IntFunction<IntStream>)	int → IntStream
LongStream	flatMap(LongFunction<LongStream>)	long → LongStream
DoubleStream	flatMapToDouble(Function<T, DoubleStream>)	T → DoubleStream
IntStream	flatMapToInt(Function<T, IntStream>)	T → IntStream
LongStream	flatMapToLong(Function<T, LongStream>)	T → LongStream

정렬

- 요소를 오름차순 또는 내림차순으로 정렬하는 중간 처리 기능

리턴 타입	메소드(매개변수)	설명
Stream<T>	sorted()	Comparable 요소를 정렬한 새로운 스트림 생성
Stream<T>	sorted(Comparator<T>)	요소를 Comparator에 따라 정렬한 새 스트림 생성
DoubleStream	sorted()	double 요소를 올림차순으로 정렬
IntStream	sorted()	int 요소를 올림차순으로 정렬
LongStream	sorted()	long 요소를 올림차순으로 정렬

Comparable 구현 객체의 정렬

- 스트림의 요소가 객체일 경우 객체가 Comparable을 구현하고 있어야만 sorted() 메소드를 사용하여 정렬 가능. 그렇지 않으면 ClassCastException 발생

```
public Xxx implements Comparable {
    ...
}
```

```
List<Xxx> list = new ArrayList<>();
Stream<Xxx> stream = list.stream();
Stream<Xxx> orderedStream = stream.sorted();
```

Comparator를 이용한 정렬

- 요소 객체가 Comparable을 구현하고 있지 않다면, 비교자를 제공하면 요소를 정렬시킬 수 있음

```
sorted((o1, o2) -> { ... })
```

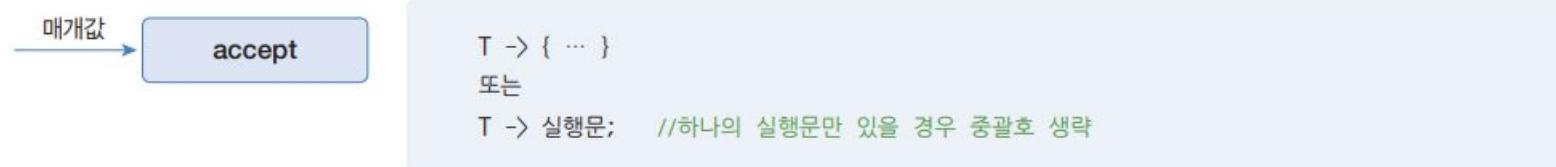
- 괄호 안에는 o1이 o2보다 작으면 음수, 같으면 0, 크면 양수를 리턴하도록 작성
- o1과 o2가 정수일 경우에는 Integer.compare(o1, o2)를, 실수일 경우에는 Double.compare(o1, o2)를 호출해서 리턴값을 리턴 가능

루핑

- 스트림에서 요소를 하나씩 반복해서 가져와 처리하는 것

리턴 타입	메소드(매개변수)	설명
Stream<T>	peek(Consumer<? super T>)	T 반복
	peek(IntConsumer action)	int 반복
	peek(DoubleConsumer action)	double 반복
void	forEach(Consumer<? super T> action)	T 반복
	forEach(IntConsumer action)	int 반복
	forEach(DoubleConsumer action)	double 반복

- 매개타입인 Consumer는 함수형 인터페이스. 모든 Consumer는 매개값을 처리(소비)하는 accept() 메소드를 가지고 있음



매칭

- 요소들이 특정 조건에 만족하는지 여부를 조사하는 최종 처리 기능
- allMatch(), anyMatch(), noneMatch() 메소드는 매개값으로 주어진 Predicate가 리턴하는 값에 따라 true 또는 false를 리턴

리턴 타입	메소드(매개변수)	조사 내용
boolean	allMatch(Predicate<T> predicate) allMatch(IntPredicate predicate) allMatch(LongPredicate predicate) allMatch(DoublePredicate predicate)	모든 요소가 만족하는지 여부
boolean	anyMatch(Predicate<T> predicate) anyMatch(IntPredicate predicate) anyMatch(LongPredicate predicate) anyMatch(DoublePredicate predicate)	최소한 하나의 요소가 만족하는지 여부
boolean	noneMatch(Predicate<T> predicate) noneMatch(IntPredicate predicate) noneMatch(LongPredicate predicate) noneMatch(DoublePredicate predicate)	모든 요소가 만족하지 않는지 여부

집계

- 최종 처리 기능으로 요소들을 처리해서 카운팅, 합계, 평균값, 최대값, 최소값 등 하나의 값으로 산출하는 것

스트림이 제공하는 기본 집계

- 스트림은 카운팅, 최대, 최소, 평균, 합계 등을 처리하는 다음과 같은 최종 처리 메소드를 제공

리턴 타입	메소드(매개변수)	설명
long	count()	요소 개수
OptionalXXX	findFirst()	첫 번째 요소
Optional<T> OptionalXXX	max(Comparator<T>) max()	최대 요소
Optional<T> OptionalXXX	min(Comparator<T>) min()	최소 요소
OptionalDouble	average()	요소 평균
int, long, double	sum()	요소 총합

Optional 클래스

- Optional, OptionalDouble, Optionallnt, OptionalLong 클래스는 단순히 집계값만 저장하는 것이 아니라, 집계값이 없으면 디폴트 값을 설정하거나 집계값을 처리하는 Consumer를 등록

리턴 타입	메소드(매개변수)	설명
boolean	isPresent()	집계값이 있는지 여부
T double int long	orElse(T) orElse(double) orElse(int) orElse(long)	집계값이 없을 경우 디폴트 값 설정
void	ifPresent(Consumer) ifPresent(DoubleConsumer) ifPresent(IntConsumer) ifPresent(LongConsumer)	집계값이 있을 경우 Consumer에서 처리

최종 처리에서 average 사용 시 요소 없는 경우를 대비하는 방법

- 1) isPresent() 메소드가 true를 리턴할 때만 집계값을 얻는다.
- 2) orElse() 메소드로 집계값이 없을 경우를 대비해서 디폴트 값을 정해놓는다.
- 3) ifPresent() 메소드로 집계값이 있을 경우에만 동작하는 Consumer 람다식을 제공한다.

스트림이 제공하는 메소드

- 스트림은 기본 집계 메소드인 `sum()`, `average()`, `count()`, `max()`, `min()`을 제공하지만, 다양한 집계 결과물을 만들 수 있도록 `reduce()` 메소드도 제공

인터페이스	리턴 타입	메소드(매개변수)
Stream	Optional<T>	reduce(BinaryOperator<T> accumulator)
	T	reduce(T identity, BinaryOperator<T> accumulator)
IntStream	OptionalInt	reduce(IntBinaryOperator op)
	int	reduce(int identity, IntBinaryOperator op)
LongStream	OptionalLong	reduce(LongBinaryOperator op)
	long	reduce(long identity, LongBinaryOperator op)
DoubleStream	OptionalDouble	reduce(DoubleBinaryOperator op)
	double	reduce(double identity, DoubleBinaryOperator op)

- `reduce()`는 스트림에 요소가 없을 경우 예외가 발생하지만, `identity` 매개값이 주어지면 이 값을 디폴트 값으로 리턴

필터링한 요소 수집

- Stream의 collect(Collector<T,A,R> collector) 메소드는 필터링 또는 맵핑된 요소들을 새로운 컬렉션에 수집하고, 이 컬렉션을 리턴
- 매개값인 Collector는 어떤 요소를 어떤 컬렉션에 수집할 것인지를 결정
- 타입 파라미터의 T는 요소, A는 누적기 accumulator, 그리고 R은 요소가 저장될 컬렉션

리턴 타입	메소드(매개변수)	인터페이스
R	collect(Collector<T,A,R> collector)	Stream

리턴 타입	메소드	설명
Collector<T, ?, List<T>>	toList()	T를 List에 저장
Collector<T, ?, Set<T>>	toSet()	T를 Set에 저장
Collector<T, ?, Map<K,U>>	toMap(Function<T,K> keyMapper, Function<T,U> valueMapper)	T를 K와 U로 맵핑하여 K를 키로, U를 값으로 Map에 저장

요소 그룹핑

- Collectors.groupingBy () 메소드에서 얻은 Collector를 collect() 메소드를 호출할 때 제공
- groupingBy()는 Function을 이용해서 T를 K로 맵핑하고, K를 키로 해 List<T>를 값으로 갖는 Map 컬렉션을 생성

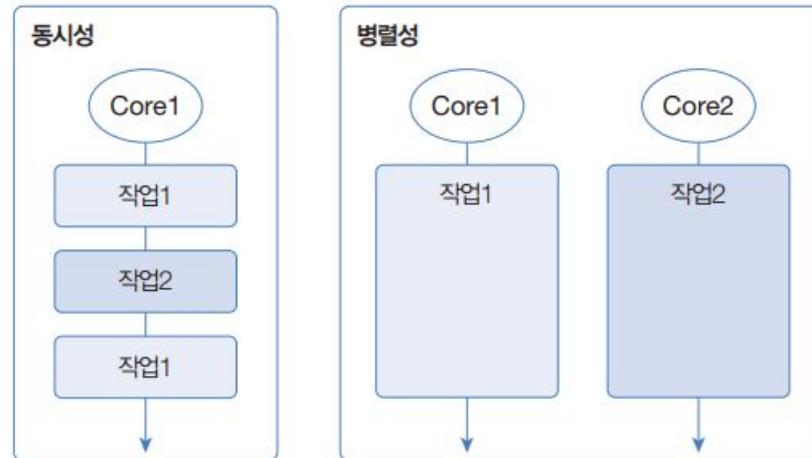
리턴 타입	메소드	K						
Collector<T,?,Map<K,List<T>>>	groupingBy(Function<T, K> classifier)	<table border="1"> <tr> <td>0</td> <td>...</td> <td>n</td> </tr> <tr> <td>T</td> <td>...</td> <td>T</td> </tr> </table>	0	...	n	T	...	T
0	...	n						
T	...	T						

- Collectors.groupingBy() 메소드는 그룹핑 후
맵핑 및 집계(평균, 카운팅, 연결, 최대,
최소, 합계)를 수행할 수 있도록 두 번째
매개값인 Collector를 가질 수 있음

리턴 타입	메소드(매개변수)	설명
Collector	mapping(Function, Collector)	맵핑
Collector	averagingDouble.ToDoubleFunction)	평균값
Collector	counting()	요소 수
Collector	maxBy(Comparator)	최대값
Collector	minBy(Comparator)	최소값
Collector	reducing(BinaryOperator<T>) reducing(T identity, BinaryOperator<T>)	커스텀 집계 값

동시성과 병렬성

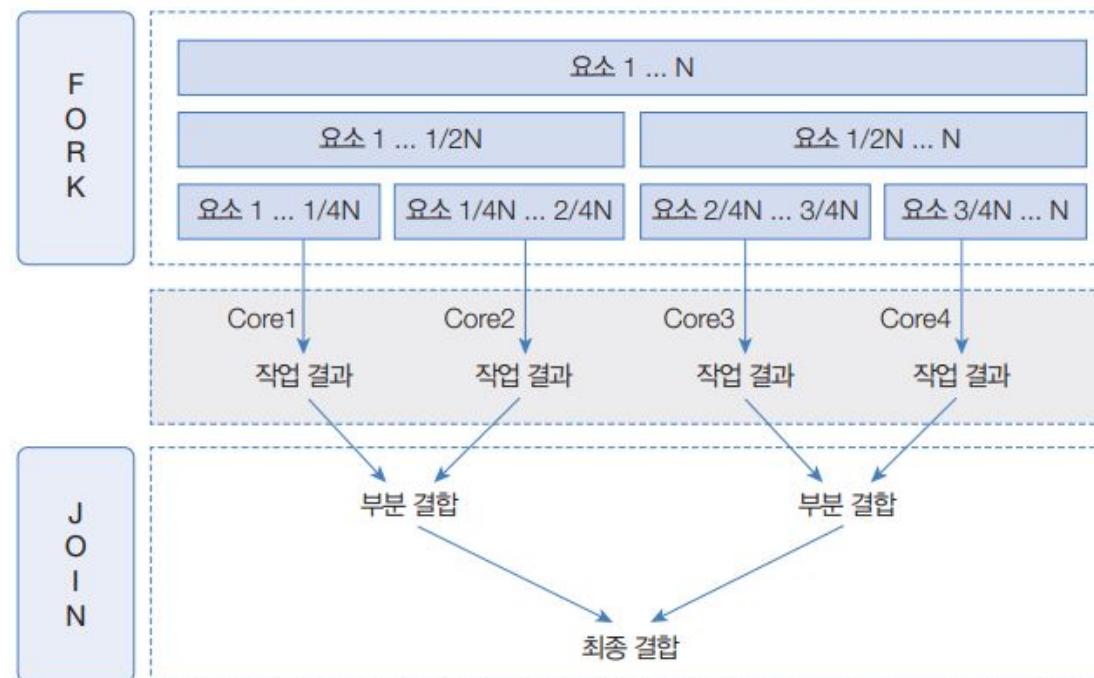
- 동시성: 멀티 작업을 위해 멀티 스레드가 하나의 코어에서 번갈아 가며 실행하는 것
- 병렬성: 멀티 작업을 위해 멀티 코어를 각각 이용해서 병렬로 실행하는 것



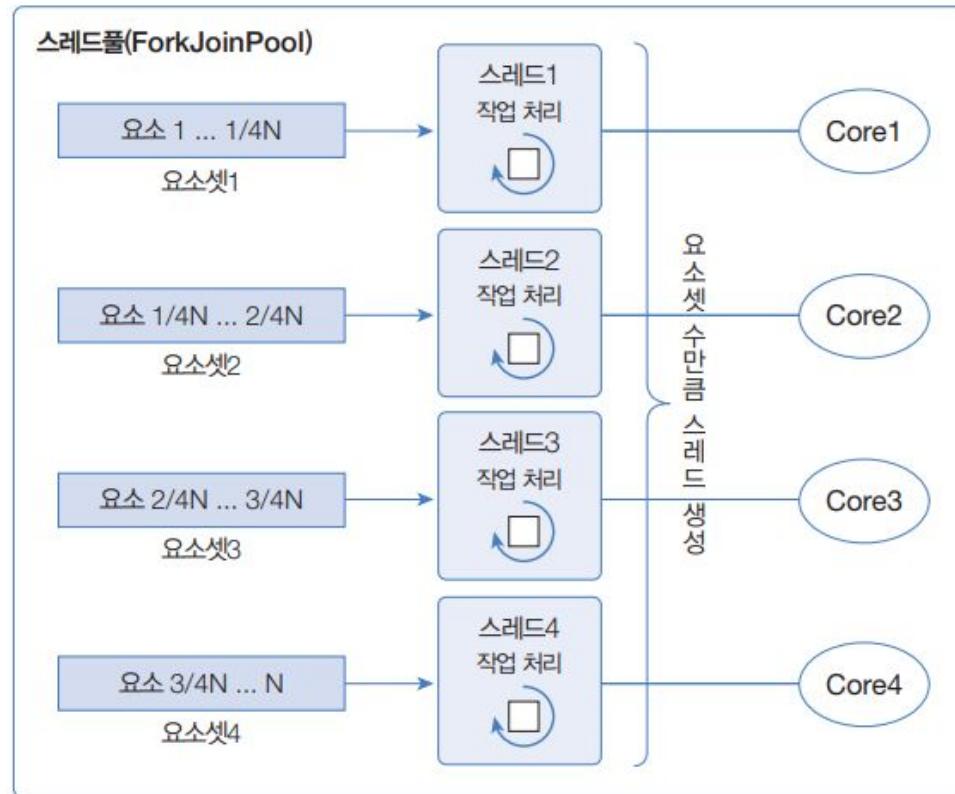
- 데이터 병렬성: 전체 데이터를 분할해서 서브 데이터셋으로 만들고 이 서브 데이터셋들을 병렬 처리해서 작업을 빨리 끝내는 것
- 작업 병렬성: 서로 다른 작업을 병렬 처리하는 것

포크조인 프레임워크

- 포크 단계: 전체 요소들을 서브 요소셋으로 분할하고, 각각의 서브 요소셋을 멀티 코어에서 병렬로 처리
- 조인 단계: 서브 결과를 결합해서 최종 결과를 만들어냄



- 포크조인 프레임워크는 ExecutorService의 구현 객체인 ForkJoinPool을 사용해서 작업 스레드를 관리



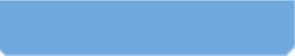
병렬 스트림 사용

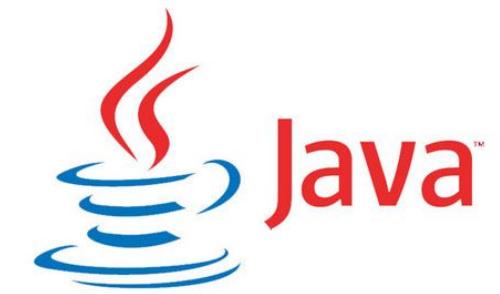
- 자바 병렬 스트림은 백그라운드에서 포크조인 프레임워크가 사용하므로 병렬 처리 용이
- parallelStream() 메소드는 컬렉션(List, Set)으로부터 병렬 스트림을 바로 리턴
- parallel() 메소드는 기존 스트림을 병렬 처리 스트림으로 변환

리턴 타입	메소드	제공 컬렉션 또는 스트림
Stream	parallelStream()	List 또는 Set 컬렉션
Stream	parallel()	java.util.Stream
IntStream		java.util.IntStream
LongStream		java.util.LongStream
DoubleStream		java.util.DoubleStream

병렬 처리 성능에 영향을 미치는 요인

- 요소의 수와 요소당 처리 시간
- 스트림 소스의 종류
- 코어(Core)의 수

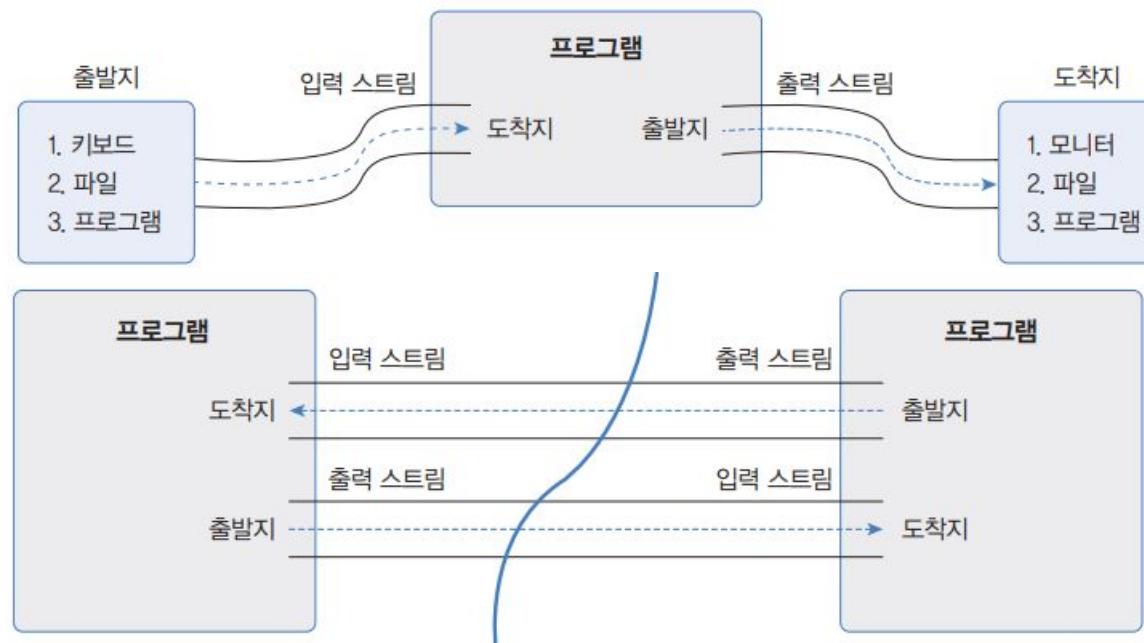




Chapter 18 데이터 입출력

입력 스트림과 출력 스트림

- 프로그램을 기준으로 데이터가 들어오면 입력 스트림, 데이터가 나가면 출력 스트림
- 프로그램이 다른 프로그램과 데이터를 교환하려면 양쪽 모두 입력 스트림과 출력 스트림이 필요



- 바이트 스트림: 그림, 멀티미디어, 문자 등 모든 종류의 데이터를 입출력할 때 사용
- 문자 스트림: 문자만 입출력할 때 사용

- 자바는 데이터 입출력과 관련된 라이브러리를 java.io 패키지에서 제공

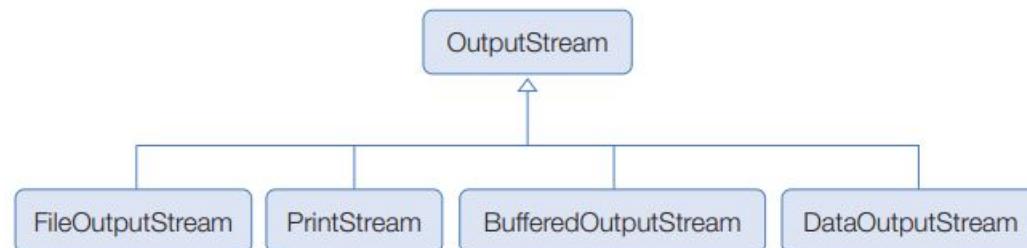
구분	바이트 스트림		문자 스트림	
	입력 스트림	출력 스트림	입력 스트림	출력 스트림
최상위 클래스	InputStream	OutputStream	Reader	Writer
하위 클래스 (예)	XXXInputStream (FileInputStream)	XXXOutputStream (FileOutputStream)	XXXReader (FileReader)	XXXWriter (FileWriter)

- 바이트 입출력 스트림의 최상위 클래스는 InputStream과 OutputStream
- 문자 입출력 스트림의 최상위 클래스는 Reader와 Writer



OutputStream

- OutputStream은 바이트 출력 스트림의 최상위 클래스로 추상 클래스
- 모든 바이트 출력 스트림 클래스는 이 OutputStream 클래스를 상속받아서 만들어짐

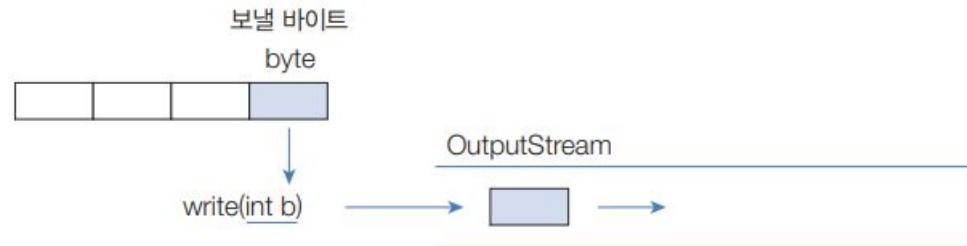


- OutputStream 클래스에는 모든 바이트 출력 스트림이 기본적으로 가져야 할 메소드가 정의됨

리턴 타입	메소드	설명
void	wrtie(int b)	1byte를 출력
void	write(byte[] b)	매개값으로 주어진 배열 b의 모든 바이트를 출력
void	write(byte[] b, int off, int len)	매개값으로 주어진 배열 b[off]부터 len개의 바이트를 출력
void	flush()	출력 버퍼에 잔류하는 모든 바이트를 출력
void	close()	출력 스트림을 닫고 사용 메모리 해제

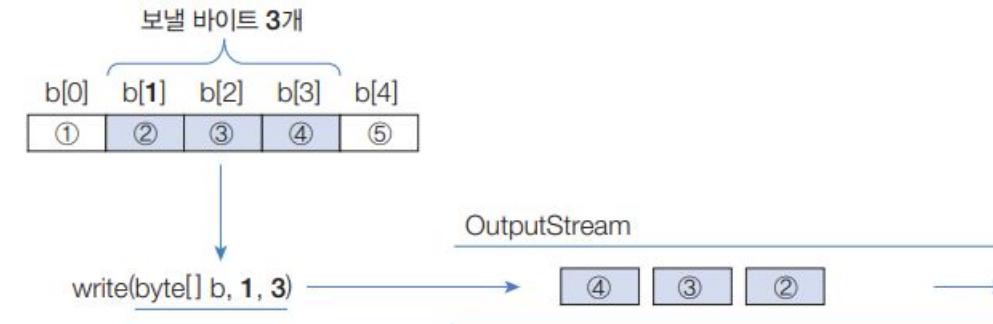
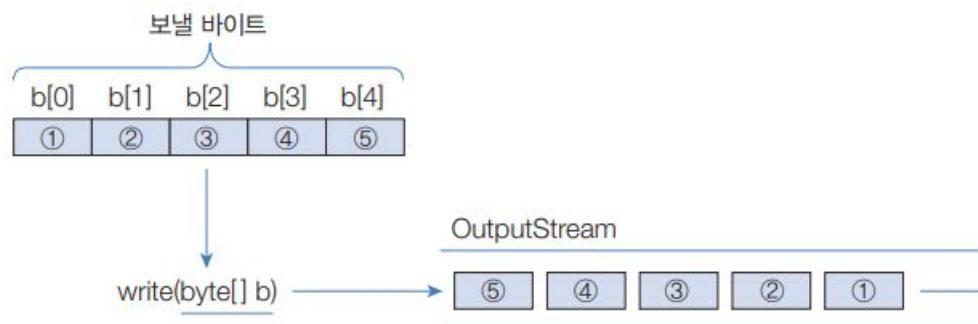
1 바이트 출력

- write(int b) 메소드: 매개값 int(4byte)에서 끝 1byte만 출력. 매개변수는 int 타입



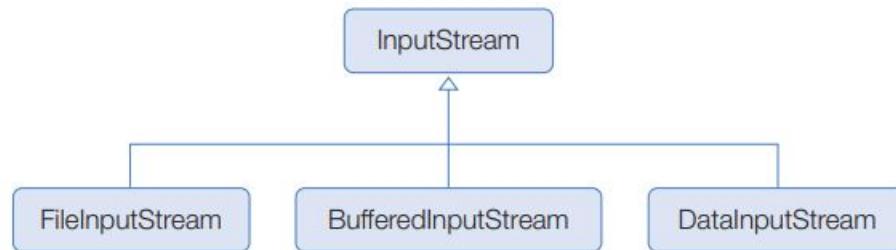
바이트 배열 출력

- write(byte[] b) 메소드: 매개값으로 주어진 배열의 모든 바이트를 출력
- 배열의 일부분을 출력하려면 write(byte[] b, int off, int len) 메소드를 사용



InputStream

- InputStream은 바이트 입력 스트림의 최상위 클래스로, 추상 클래스
- 모든 바이트 입력 스트림은 InputStream 클래스를 상속받아 만들어짐

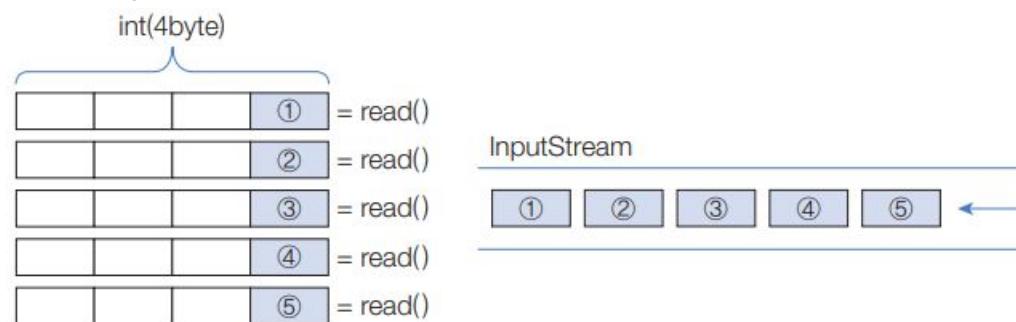


- InputStream 클래스에는 바이트 입력 스트림이 기본적으로 가져야 할 메소드가 정의됨

리턴 타입	메소드	설명
int	read()	1byte를 읽은 후 읽은 바이트를 리턴
int	read(byte[] b)	읽은 바이트를 매개값으로 주어진 배열에 저장 후 읽은 바이트 수를 리턴
void	close()	입력 스트림을 닫고 사용 메모리 해제

1 바이트 입력

- `read()` 메소드: 입력 스트림으로부터 1byte를 읽고 int(4byte) 타입으로 리턴. 리턴된 4byte 중 끝 1byte에만 데이터가 들어 있음



- 더 이상 입력 스트림으로부터 바이트를 읽을 수 없다면 `read()` 메소드는 -1을 리턴. 읽을 수 있는 마지막 바이트까지 반복해서 한 바이트씩 읽을 수 있음

```
InputStream is = ...;
while (true) {
    int data = is.read();      // 1 바이트를 읽고 리턴
    if (data == -1) break;    // -1을 리턴했을 경우 while 문 종료
}
```

바이트 배열로 읽기

- `read(byte[] b)` 메소드: 입력 스트림으로부터

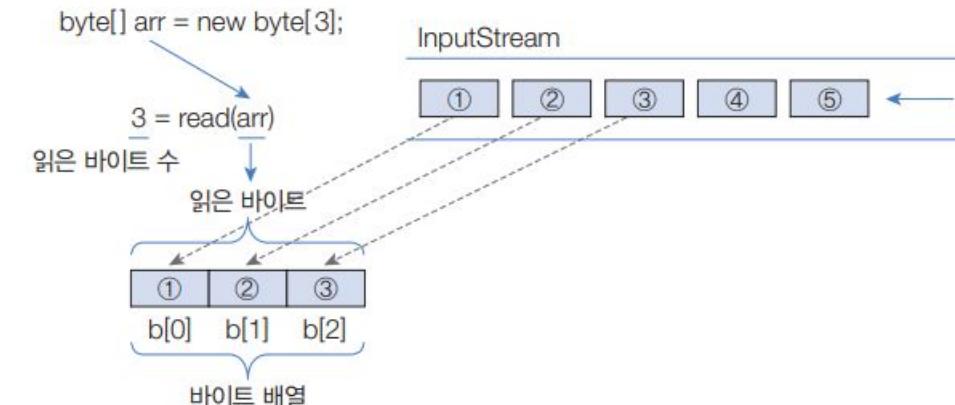
주어진 배열의 길이만큼 바이트를 읽고 배열에
저장한 다음 읽은 바이트 수를 리턴

- `read(byte[] b)`도 입력 스트림으로부터
바이트를 더 이상 읽을 수 없다면 -1을 리턴.

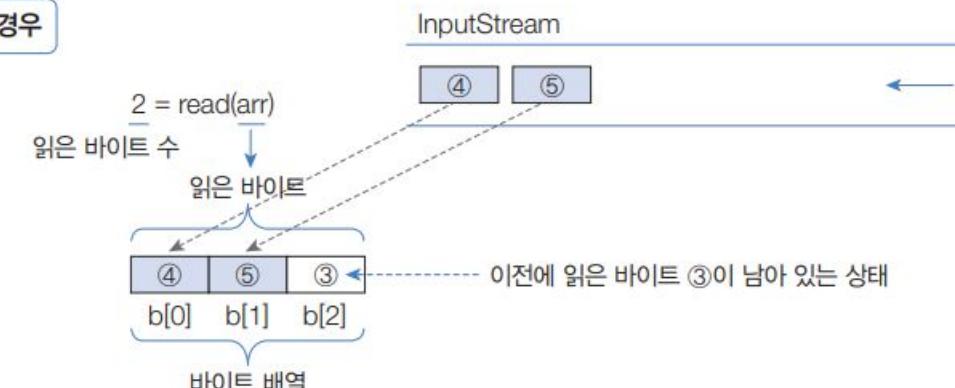
읽을 수 있는 마지막 바이트까지 반복해서
읽을 수 있음

```
InputStream is = ...;
byte[] data = new byte[100];
while (true) {
    int num = is.read(data); //최대 100byte를 읽고, 읽은 바이트는 배열 data 저장, 읽은
                           //수는 리턴
    if (num == -1) break;   // -1을 리턴하면 while 문 종료
}
```

첫 번째 읽을 경우

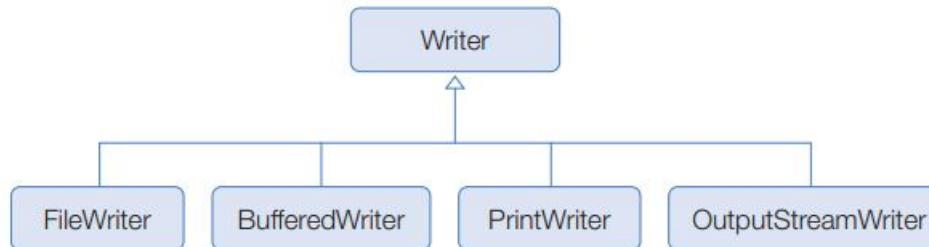


두 번째 읽을 경우



문자 출력

- Writer는 문자 출력 스트림의 최상위 클래스로, 추상 클래스. 모든 문자 출력 스트림 클래스는 Writer 클래스를 상속받아서 만들어짐

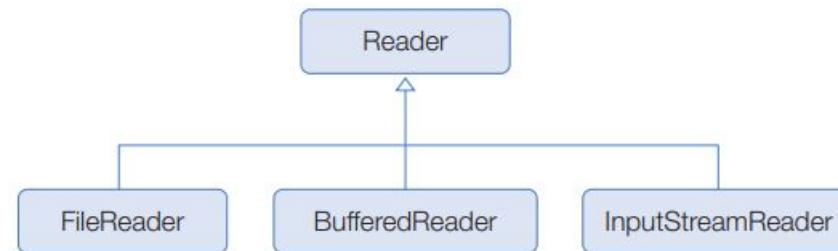


- Writer 클래스에는 모든 문자 출력 스트림이 기본적으로 가져야 할 메소드가 정의됨

리턴 타입	메소드	설명
void	wrtie(int c)	매개값으로 주어진 한 문자를 출력
void	write(char[] cbuf)	매개값으로 주어진 배열의 모든 문자를 출력
void	write(char[] cbuf, int off, int len)	매개값으로 주어진 배열에서 cbuf[off]부터 len개까지의 문자를 출력
void	write(String str)	매개값으로 주어진 문자열을 출력
void	write(String str, int off, int len)	매개값으로 주어진 문자열에서 off 순번부터 len개까지의 문자를 출력
void	flush()	버퍼에 잔류하는 모든 문자를 출력
void	close()	출력 스트림을 닫고 사용 메모리를 해제

Reader

- Reader는 문자 입력 스트림의 최상위 클래스로, 추상 클래스
- 모든 문자 입력 스트림 클래스는 Reader 클래스를 상속받아서 만들어짐

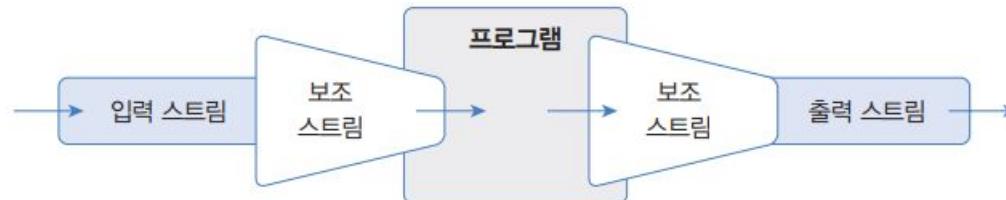


- Reader 클래스에는 문자 입력 스트림이 기본적으로 가져야 할 메소드가 정의됨

메소드		설명
int	read()	1개의 문자를 읽고 리턴
int	read(char[] cbuf)	읽은 문자들을 매개값으로 주어진 문자 배열에 저장하고 읽은 문자 수를 리턴
void	close()	입력 스트림을 닫고, 사용 메모리 해제

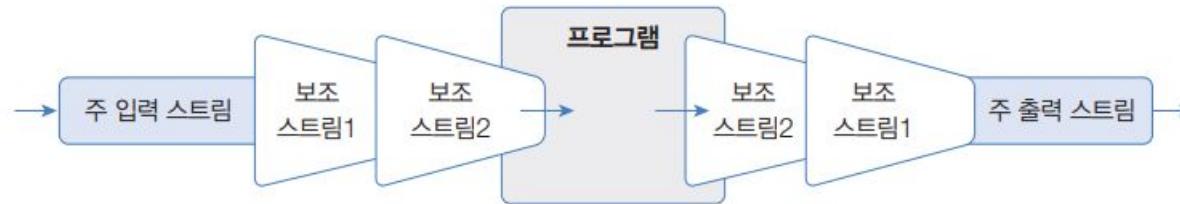
보조 스트림

- 다른 스트림과 연결되어 여러 편리한 기능을 제공해주는 스트림. 자체적으로 입출력을 수행할 수 없기 때문에 입출력 소스로부터 직접 생성된 입출력 스트림에 연결해서 사용



- 입출력 스트림에 보조 스트림을 연결하려면 보조 스트림을 생성할 때 생성자 매개값으로 입출력 스트림을 제공
- 보조스트림 변수 = new 보조스트림(입출력스트림);

- 보조 스트림은 또 다른 보조 스트림과 연결되어 스트림 체인으로 구성할 수 있음

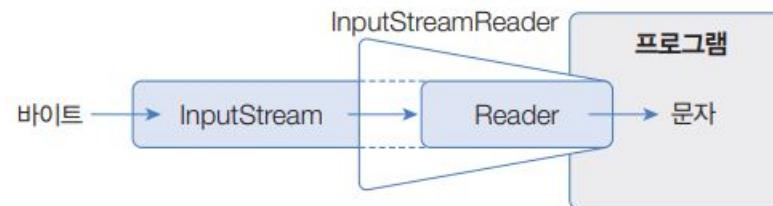


- 보조스트림2 변수 = new 보조스트림2(보조 스트림1);

보조 스트림	기능
InputStreamReader	바이트 스트림을 문자 스트림으로 변환
BufferedInputStream, BufferedOutputStream BufferedReader, BufferedWriter	입출력 성능 향상
DataInputStream, DataOutputStream	기본 타입 데이터 입출력
PrintStream, PrintWriter	줄바꿈 처리 및 형식화된 문자열 출력
ObjectInputStream, ObjectOutputStream	객체 입출력

InputStream을 Reader로 변환

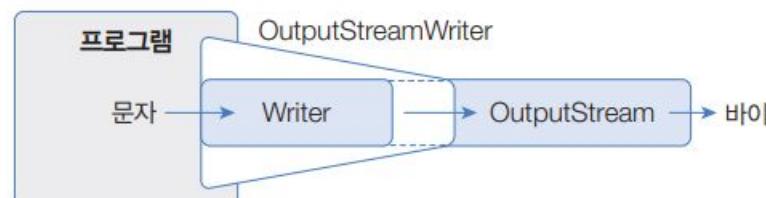
- InputStream을 Reader로 변환하려면 InputStreamReader 보조 스트림을 연결



```
InputStream is = new FileInputStream("C:/Temp/test.txt");
Reader reader = new InputStreamReader(is);
```

OutputStream을 Writer로 변환

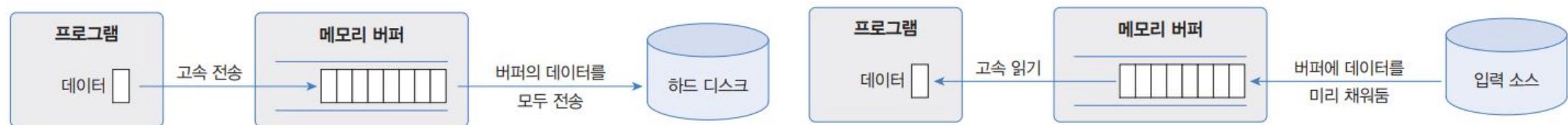
- OutputStream을 Writer로 변환하려면 OutputStreamWriter 보조 스트림을 연결



```
OutputStream os = new FileOutputStream("C:/Temp/test.txt");
Writer writer = new OutputStreamWriter(os);
```

메모리 버퍼로 실행 성능을 높이는 보조 스트림

- 프로그램이 중간에 메모리 버퍼buffer와 작업해서 실행 성능 향상 가능
- 출력 스트림의 경우 직접 하드 디스크에 데이터를 보내지 않고 메모리 버퍼에 데이터를 보냄으로써 출력 속도를 향상. 입력 스트림에서도 버퍼를 사용하면 읽기 성능 향상



- 바이트 스트림에는 BufferedInputStream, BufferedOutputStream이 있고 문자 스트림에는 BufferedReader, BufferedWriter가 있음

```
BufferedInputStream bis = new BufferedInputStream(바이트 입력 스트림);
BufferedOutputStream bos = new BufferedOutputStream(바이트 출력 스트림);
```

```
BufferedReader br = new BufferedReader(문자 입력 스트림);
BufferedWriter bw = new BufferedWriter(문자 출력 스트림);
```

기본 타입 스트림

- 바이트 스트림에 DataInputStream과 DataOutputStream 보조 스트림을 연결하면 기본 타입 (boolean, char, short, int, long, float, double) 값을 입출력할 수 있음



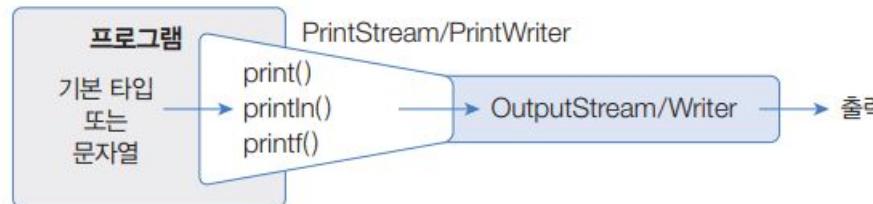
```

DataInputStream dis = new DataInputStream(바이트 입력 스트림);
DataOutputStream dos = new DataOutputStream(바이트 출력 스트림);
  
```

DataInputStream		DataOutputStream	
boolean	readBoolean()	void	writeBoolean(boolean v)
byte	readByte()	void	writeByte(int v)
char	readChar()	void	writeChar(int v)
double	readDouble()	void	writeDouble(double v)
float	readFloat()	void	writeFloat(float v)
int	readInt()	void	writeInt(int v)
long	readLong()	void	writeLong(long v)
short	readShort()	void	writeShort(int v)
String	readUTF()	void	writeUTF(String str)

PrintStream과 PrintWriter

- 프린터와 유사하게 출력하는 print(), println(), printf() 메소드를 가진 보조 스트림



- PrintStream은 바이트 출력 스트림과 연결되고, PrintWriter는 문자 출력 스트림과 연결

```

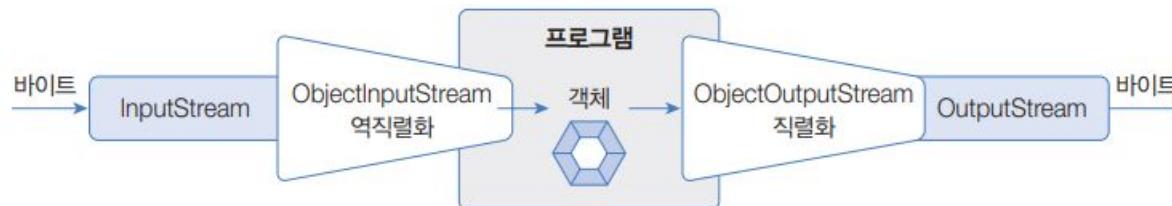
PrintStream ps = new PrintStream(바이트 출력 스트림);
PrintWriter pw = new PrintWriter(문자 출력 스트림);
  
```

PrintStream / PrintWriter

void	print(boolean b)	void	println(boolean b)
void	print(char c)	void	println(char c)
void	print(double d)	void	println(double d)
void	print(float f)	void	println(float f)
void	print(int i)	void	println(int i)
void	print(long l)	void	println(long l)
void	print(Object obj)	void	println(Object obj)
void	print(String s)	void	println(String s)
		void	println()

직렬화와 역직렬화

- 직렬화: 메모리에 생성된 객체를 파일 또는 네트워크로 출력하기 위해 필드값을 일렬로 늘어선 바이트로 변경하는 것
- 역직렬화: 직렬화된 바이트를 객체의 필드값으로 복원하는 것.
- ObjectOutputStream은 바이트 출력 스트림과 연결되어 객체를 직렬화하고,
ObjectInputStream은 바이트 입력 스트림과 연결되어 객체로 복원하는 역직렬화



```
ObjectInputStream ois = new ObjectInputStream(바이트 입력 스트림);
ObjectOutputStream oos = new ObjectOutputStream(바이트 출력 스트림);
```

Serializable 인터페이스

- 멤버가 없는 빈 인터페이스이지만, 객체를 직렬화할 수 있다고 표시하는 역할
- 인스턴스 필드값은 직렬화 대상. 정적 필드값과 transient로 선언된 필드값은 직렬화에서 제외되므로 출력되지 않음

```
public class XXX implements Serializable {
    public int field1;
    protected int field2;
    int field3;
    private int field4;
    public static int field5;
    transient int field6;
}
```

직렬화 → 일렬로 늘어선 바이트 데이터

field1 field2 field3 field4

//정적 필드는 직렬화에서 제외
//transient로 선언된 필드는 직렬화에서 제외

serialVersionUID 필드

- 직렬화할 때 사용된 클래스와 역직렬화할 때 사용된 클래스는 동일한 클래스여야 함
- 클래스 내용이 다르더라도 두 클래스가 동일한 serialVersionUID 상수값을 가지면 역직렬화 가능

```
public class Member implements
    Serializable {
    static final long
        serialVersionUID = 1;
    int field1;
    int field2;
}
```



```
public class Member implements
    Serializable {
    static final long
        serialVersionUID = 1;
    int field1;
    int field2;
    int field3;
}
```

File 클래스

- File 클래스로부터 File 객체를 생성

```
File file = new File("경로");
```

```
File file = new File("C:/Temp/file.txt");
File file = new File("C:\\Temp\\\\file.txt");
```

- exists() 메소드가 false를 리턴할 경우, 다음 메소드로 파일 또는 폴더를 생성

리턴 타입	메소드	설명
boolean	createNewFile()	새로운 파일을 생성
boolean	mkdir()	새로운 디렉토리를 생성
boolean	mkdirs()	경로상에 없는 모든 디렉토리를 생성

- exists() 메소드의 리턴값이 true라면 다음 메소드를 사용

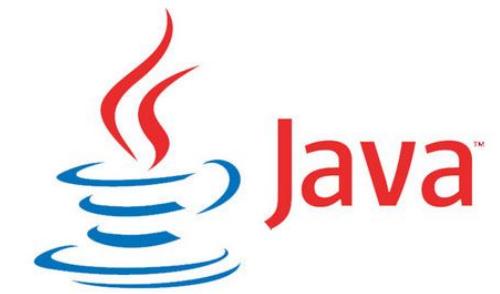
리턴 타입	메소드	설명
boolean	delete()	파일 또는 디렉토리 삭제
boolean	canExecute()	실행할 수 있는 파일인지 여부
boolean	canRead()	읽을 수 있는 파일인지 여부
boolean	canWrite()	수정 및 저장할 수 있는 파일인지 여부
String	getName()	파일의 이름을 리턴
String	getParent()	부모 디렉토리를 리턴
File	getParentFile()	부모 디렉토리를 File 객체로 생성 후 리턴
String	getPath()	전체 경로를 리턴
boolean	isDirectory()	디렉토리인지 여부
boolean	isFile()	파일인지 여부
boolean	isHidden()	숨김 파일인지 여부
long	lastModified()	마지막 수정 날짜 및 시간을 리턴
long	length()	파일의 크기 리턴
String[]	list()	디렉토리에 포함된 파일 및 서브 디렉토리 목록 전부를 String 배열로 리턴
String[]	list(FilenameFilter filter)	디렉토리에 포함된 파일 및 서브 디렉토리 목록 중에 FilenameFilter에 맞는 것만 String 배열로 리턴
File[]	listFiles()	디렉토리에 포함된 파일 및 서브 디렉토리 목록 전부를 File 배열로 리턴
File[]	listFiles(FilenameFilter filter)	디렉토리에 포함된 파일 및 서브 디렉토리 목록 중에 FilenameFilter에 맞는 것만 File 배열로 리턴

Files 클래스

- Files 클래스는 정적 메소드로 구성되어 있기 때문에 File 클래스처럼 객체로 만들 필요 없음
- Files의 정적 메소드는 운영체제의 파일 시스템에게 파일 작업을 수행하도록 위임

기능	관련 메소드
복사	copy
생성	createDirectories, createDirectory, createFile, createLink, createSymbolicLink, createTempDirectory, createTempFile
이동	move
삭제	delete, deleteIfExists
존재, 검색, 비교	exists, notExists, find, mismatch
속성	getLastModifiedTime, getOwner, getPosixFilePermissions, isDirectory, isExecutable, isHidden, isReadable, isSymbolicLink, isWritable, size, setAttribute, setLastModifiedTime, setOwner, setPosixFilePermissions, probeContentType
디렉토리 탐색	list, newDirectoryStream, walk
데이터 입출력	newInputStream, newOutputStream, newBufferedReader, newBufferedWriter, readAllBytes, lines, readAllLines, readString, readSymbolicLink, write, writeString

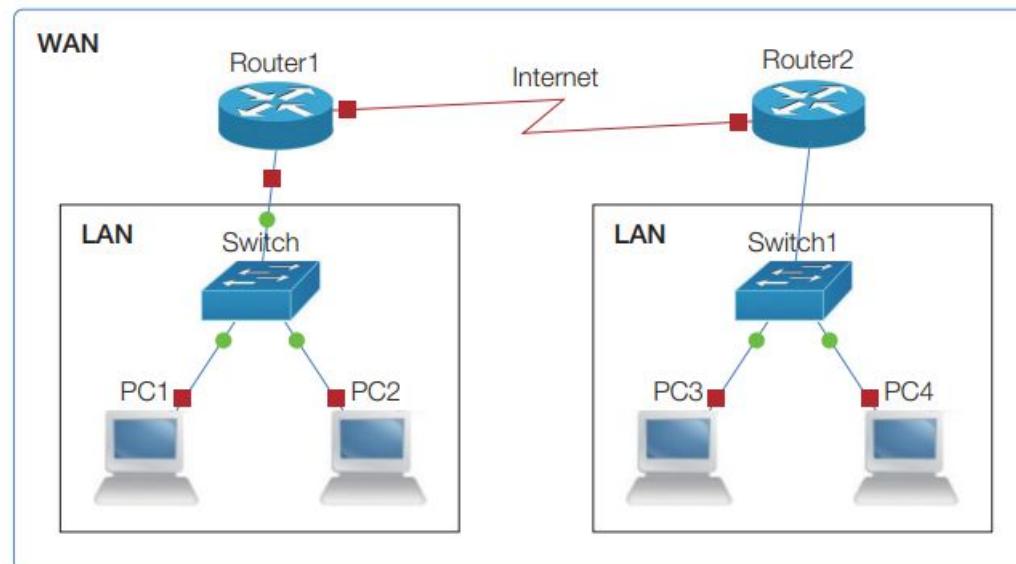




Chapter 19 네트워크 입출력

네트워크

- 네트워크: 여러 컴퓨터들을 통신 회선으로 연결한 것
- LAN: 가정, 회사, 건물, 특정 영역에 존재하는 컴퓨터를 연결한 것
- WAN: LAN을 연결한 것 = 인터넷



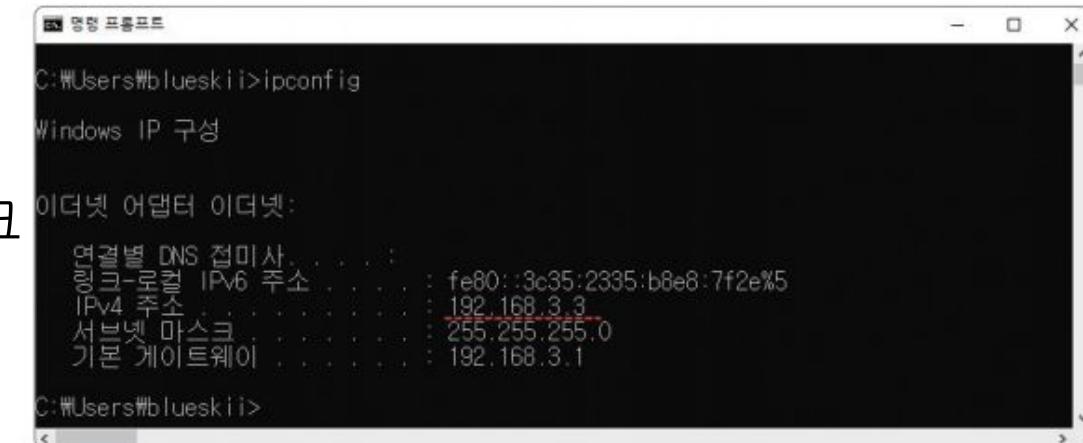
서버와 클라이언트

- 서버: 서비스를 제공하는 프로그램을
- 클라이언트: 서비스를 요청하는 프로그램
- 먼저 클라이언트가 서비스를 요청하고, 서버는 처리 결과를 응답으로 제공



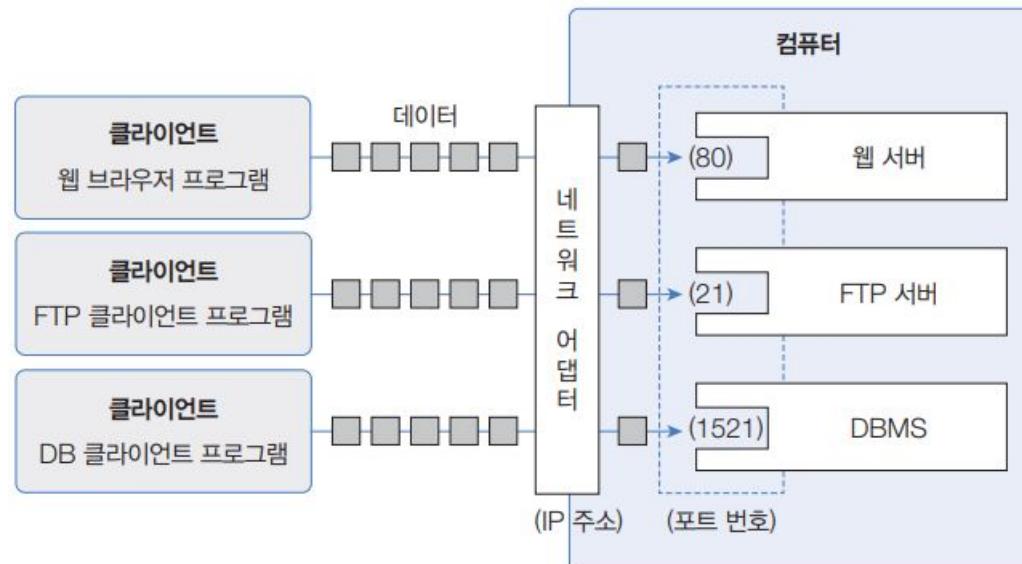
IP 주소

- IP 주소: 네트워크 어댑터(LAN 카드)마다 할당되는 컴퓨터의 고유한 주소
- ipconfig(윈도우), ifconfig(맥OS) 명령어로 네트워크 어댑터에 어떤 IP 주소가 부여되어 있는지 확인
- 프로그램은 DNS를 이용해서 컴퓨터의 IP 주소를 검색



Port 번호

- 운영체제가 관리하는 서버 프로그램의 연결 번호. 서버 시작 시 특정 Port 번호에 바인딩



구분명	범위	설명
Well Known Port Numbers	0~1023	국제인터넷주소관리기구(ICANN)가 특정 애플리케이션용으로 미리 예약한 Port
Registered Port Numbers	1024~49151	회사에서 등록해서 사용할 수 있는 Port
Dynamic Or Private Port Numbers	49152~65535	운영체제가 부여하는 동적 Port 또는 개인적인 목적으로 사용할 수 있는 Port

InetAddress

- 자바는 IP 주소를 java.net 패키지의 InetAddress로 표현
- 로컬 컴퓨터의 InetAddress를 얻으려면 InetAddress.getLocalHost() 메소드를 호출

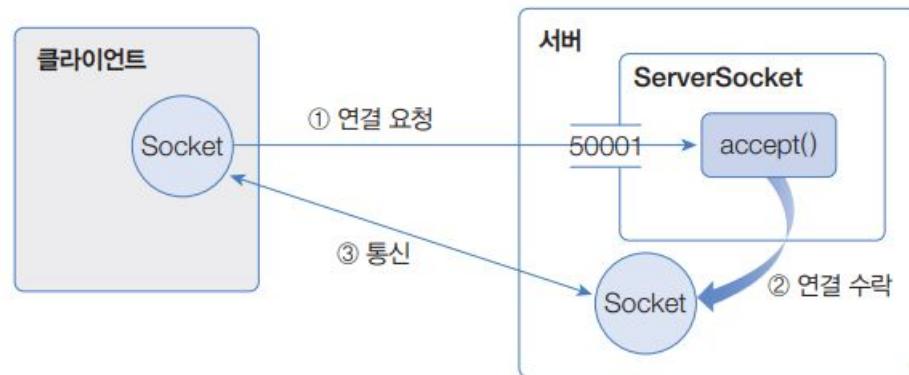
```
InetAddress ia = InetAddress.getLocalHost();
```

- getByName () 메소드는 DNS에서 도메인 이름으로 등록된 단 하나의 IP 주소를 가져오고, getAllByName() 메소드는 등록된 모든 IP 주소를 배열로 가져옴
- InetAddress 객체에서 IP 주소를 얻으려면 getHostAddress () 메소드를 호출

```
String ip = InetAddress.getHostAddress();
```

TCP

- TCP는 연결형 프로토콜로, 상대방이 연결된 상태에서 데이터를 주고 받는 전송용 프로토콜
- 클라이언트가 연결 요청을 하고 서버가 연결을 수락하면 통신 회선이 고정되고, 데이터는 고정 회선을 통해 전달. TCP는 보낸 데이터가 순서대로 전달되며 손실이 발생하지 않음
- ServerSocket은 클라이언트의 연결을 수락하는 서버 쪽 클래스이고, Socket은 클라이언트에서 연결 요청할 때와 클라이언트와 서버 양쪽에서 데이터를 주고 받을 때 사용되는 클래스



TCP 서버

- TCP 서버 프로그램을 개발하려면 우선 ServerSocket 객체를 생성

```
ServerSocket serverSocket = new ServerSocket(50001);
```

- 기본 생성자로 객체를 생성하고 Port 바인딩을 위해 bind() 메소드를 호출해도 ServerSocket 생성

```
ServerSocket serverSocket = new ServerSocket();  
serverSocket.bind(new InetSocketAddress(50001));
```

- 여러 개의 IP가 할당된 서버 컴퓨터에서 특정 IP에서만 서비스를 하려면 InetSocketAddress의 첫 번째 매개값으로 해당 IP를 줌

```
ServerSocket serverSocket = new ServerSocket();  
serverSocket.bind( new InetSocketAddress("xxx.xxx.xxx.xxx", 50001) );
```

- Port가 이미 다른 프로그램에서 사용 중이라면 BindException이 발생. 다른 Port로 바인딩하거나 Port를 사용 중인 프로그램을 종료하고 다시 실행해야 함

- ServerSocket이 생성되면 연결 요청을 수락을 위해 accept() 메소드를 실행
- accept()는 클라이언트가 연결 요청하기 전까지 블로킹(실행 멈춘 상태) 클라이언트의 연결 요청이 들어오면 블로킹이 해제되고 통신용 Socket을 리턴

```
Socket socket = serverSocket.accept();
```

- 리턴된 Socket을 통해 연결된 클라이언트의 IP 주소와 Port 번호를 얻으려면 getRemoteSocketAddress () 메소드를 호출해서 InetSocketAddress을 얻은 다음 getHostName()과 getPort() 메소드를 호출

```
InetSocketAddress isa = (InetSocketAddress) socket.getRemoteSocketAddress();
String clientIp = isa.getHostName();
String portNo = isa.getPort();
```

- ServerSocket의 close() 메소드를 호출해서 Port 번호를 언바이딩해야 서버 종료

```
serverSocket.close();
```

TCP 클라이언트

- 클라이언트가 서버에 연결 요청을 하려면 Socket 객체를 생성할 때 생성자 매개값으로 서버 IP 주소와 Port 번호를 제공
- 로컬 컴퓨터에서 실행하는 서버로 연결 요청을 할 경우에는 IP 주소 대신 localhost 사용 가능

```
Socket socket = new Socket( "IP", 50001 );
```

- 도메인 이름을 사용하려면 DNS에서 IP 주소를 검색하는 생성자 매개값으로 InetSocketAddress

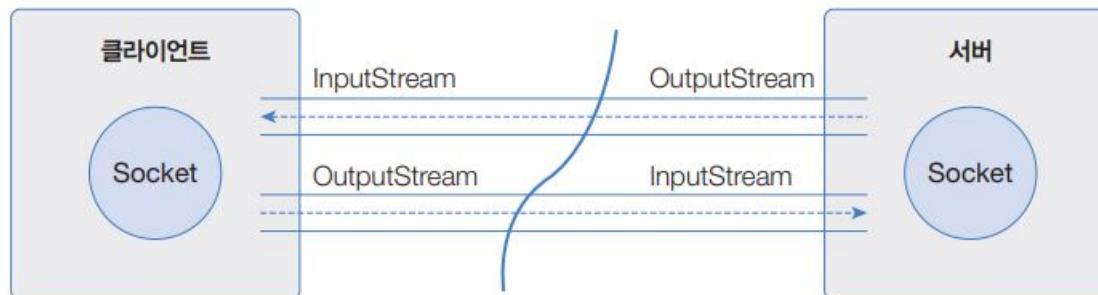
```
Socket socket = new Socket( new InetSocketAddress("domainName", 50001) );
```

- 기본 생성자로 Socket을 생성한 후 connect() 메소드로 연결 요청 가능

```
socket = new Socket();
socket.connect( new InetSocketAddress("domainName", 50001) );
```

입출력 스트림으로 데이터 주고 받기

- 클라이언트가 연결 요청(connect())을 하고 서버가 연결 수락(accept()) 했다면, 양쪽의 Socket 객체로부터 각각 InputStream과 OutputStream을 얻을 수 있다.



```
InputStream is = socket.getInputStream();
OutputStream os = socket.getOutputStream();
```

- 상대방에게 데이터를 보낼 때에는 보낼 데이터를 byte[] 배열로 생성하고, 이것을 매개값으로 해서 OutputStream의 write() 메소드를 호출
- 문자열을 좀 더 간편하게 보내고 싶다면 보조 스트림인 DataOutputStream을 연결해서 사용

```
String data = "보낼 데이터";
byte[ ] bytes = data.getBytes("UTF-8");
OutputStream os = socket.getOutputStream();
os.write(bytes);
os.flush();
```

```
String data = "보낼 데이터";
DataOutputStream dos = new DataOutputStream(socket.getOutputStream());
dos.writeUTF(data);
dos.flush();
```

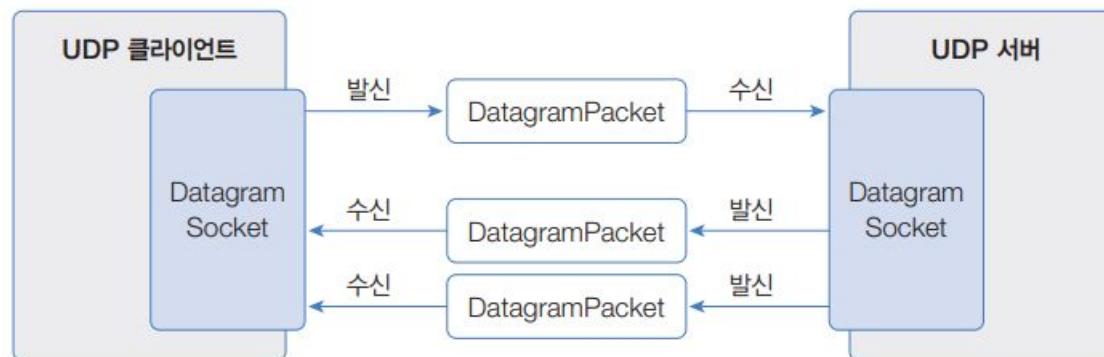
- 데이터를 받기 위해서는 받은 데이터를 저장할 byte[] 배열을 하나 생성하고, 이것을 매개값으로 해서 InputStream의 read() 메소드를 호출
- read() 메소드는 읽은 데이터를 byte[] 배열에 저장하고 읽은 바이트 수를 리턴
- 문자열을 좀 더 간편하게 받고 싶다면 보조 스트림인 DataInputStream을 연결해서 사용

```
byte[ ] bytes = new byte[1024];
InputStream is = socket.getInputStream();
int num = is.read(bytes);
String data = new String(bytes, 0, num, "UTF-8");
```

```
DataInputStream dis = new DataInputStream(socket.getInputStream());
String data = dis.readUTF();
```

UDP

- 발신자가 일방적으로 수신자에게 데이터를 보내는 방식. TCP처럼 연결 요청 및 수락 과정이 없기 때문에 TCP보다 데이터 전송 속도가 상대적으로 빠름
- 데이터 전달의 신뢰성보다 속도가 중요하다면 UDP를 사용하고, 데이터 전달의 신뢰성이 중요하다면 TCP를 사용
- DatagramSocket은 발신점과 수신점에 해당하고 DatagramPacket은 주고받는 데이터에 해당



UDP 서버

- DatagramSocket 객체를 생성할 때에는 다음과 같이 바인딩할 Port 번호를 생성자 매개값으로 제공

```
DatagramSocket datagramSocket = new DatagramSocket(50001);
```

- receive() 메소드는 데이터를 수신할 때까지 블로킹되고, 데이터가 수신되면 매개값으로 주어진 DatagramPacket에 저장

```
DatagramPacket receivePacket = new DatagramPacket(new byte[1024], 1024);
datagramSocket.receive(receivePacket);
```

- DatagramPacket 생성자의 첫 번째 매개값은 수신된 데이터를 저장할 배열이고 두 번째 매개값은 수신할 수 있는 최대 바이트 수

```
byte[] bytes = receivePacket.getData();
int num = receivePacket.getLength();
```

```
String data = new String(bytes, 0, num, "UTF-8");
```

- `getSocketAddress()` 메소드를 호출하면 정보가 담긴 `SocketAddress` 객체를 얻을 수 있음

```
SocketAddress socketAddress = receivePacket.getSocketAddress();
```

- `SocketAddress` 객체는 클라이언트로 보낼 `DatagramPacket`을 생성할 때 네 번째 매개값으로 사용

```
String data = "처리 내용";
byte[] bytes = data.getBytes("UTF-8");
DatagramPacket sendPacket = new DatagramPacket( bytes, 0, bytes.length,
socketAddress );
```

- `DatagramPacket`을 클라이언트로 보낼 때는 `DatagramSocket`의 `send()` 메소드를 이용

```
datagramSocket.send( sendPacket );
```

- UDP 서버를 종료하고 싶을 경우에는 `DatagramSocket`의 `close()` 메소드를 호출

```
datagramSocket.close();
```

UDP 클라이언트

- 서버에 요청 내용을 보내고 그 결과를 받는 역할
- UDP 클라이언트를 위한 DatagramSocket 객체는 기본 생성자로 생성. Port 번호는 자동 부여

```
String data = "요청 내용";
byte[ ] bytes = data.getBytes("UTF-8");
DatagramPacket sendPacket = new DatagramPacket(
    bytes, bytes.length, new InetSocketAddress("localhost", 50001)
);
```

- 생성된 DatagramPacket을 매개값으로해서 DatagramSocket의 send() 메소드를 호출하면 UDP 서버로 DatagramPacket이 전송

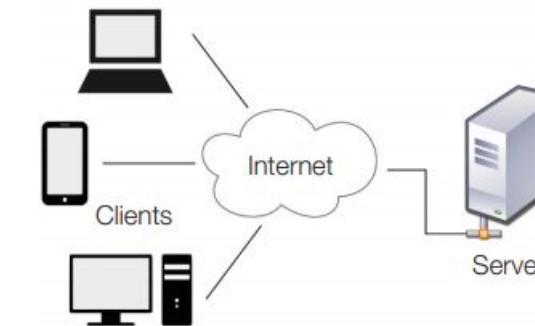
```
datagramSocket.send(sendPacket);
```

- DatagramSocket을 닫으려면 close() 메소드를 호출

```
datagramSocket.close();
```

서버의 동시 요청 처리

- 일반적으로 서버는 다수의 클라이언트와 통신. 서버는 클라이언트들로부터 동시에 요청을 받아서 처리하고, 처리 결과를 개별 클라이언트로 보내줌
- accept()와 receive()를 제외한 요청 처리 코드를 별도의 스레드에서 작업



TCP 서버

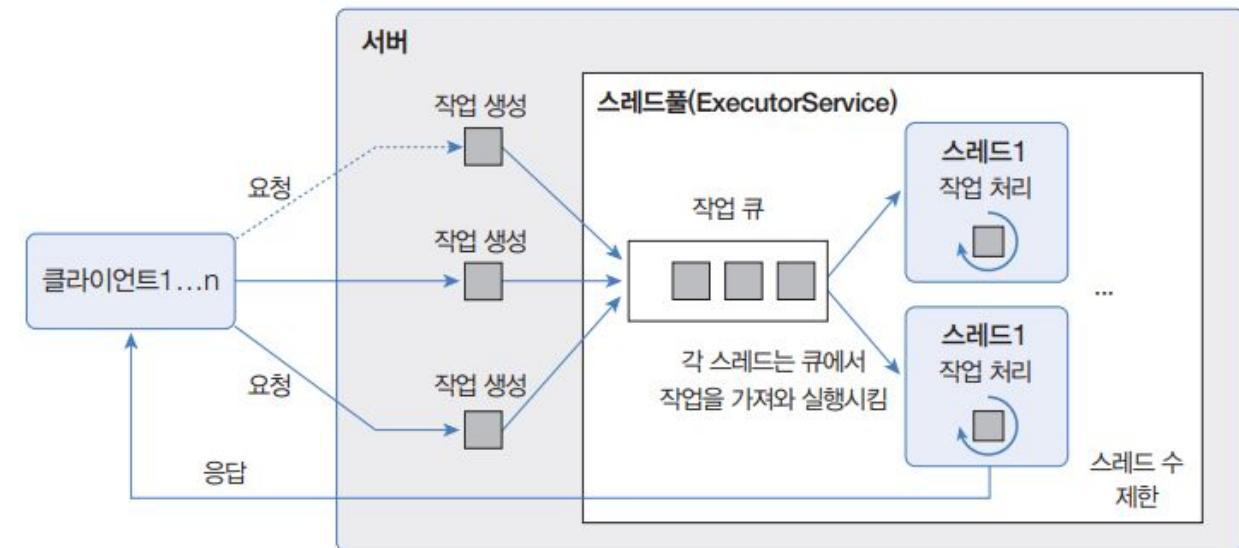
```
while(true) {
    Socket socket = serverSocket.
    accept();

    //데이터 받기
    ...
    //데이터 보내기
    ...
}
```

UDP 서버

```
while(true) {
    DatagramPacket receivePacket = ...
    datagramSocket.
    receive(receivePacket);
    ...
    스레드로 처리
    ...
    //10개의 뉴스를 클라이언트로 전송
    ...
}
```

- 스레드를 처리할 때 클라이언트의 폭증으로 인한 서버의 과도한 스레드 생성을 방지하기 위해 스레드풀을 사용하는 것이 바람직



TCP EchoServer 동시 요청 처리

- 스레드풀을 이용해서 클라이언트의 요청을 동시에 처리

```

12
13  public class EchoServer {
14      private static ServerSocket serverSocket = null;
15      private static ExecutorService executorService =
16          Executors.newFixedThreadPool(10);
17
18      public static void main(String[] args) {
19          System.out.println("-----");
20          System.out.println("서버를 종료하려면 q를 입력하고 Enter 키를 입력하세요.");
21          System.out.println("-----");

```

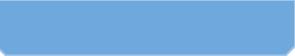
10개의 스레드로 요청을 처리하는 스레드풀 생성

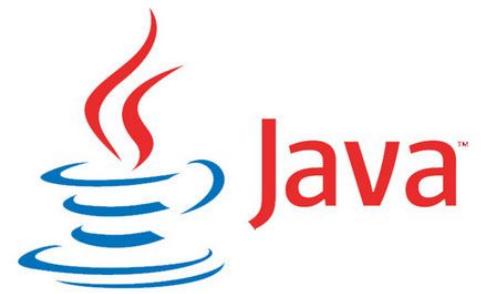
UDP NewsServer 동시 요청 처리

- 스레드풀을 이용해서 클라이언트의 요청을 동시에 처리

```
10  public class NewsServer extends Thread {  
11      private static DatagramSocket datagramSocket = null;  
12      private static ExecutorService executorService =  
13          Executors.newFixedThreadPool(10);  
14      public static void main(String[] args) throws Exception {  
15          System.out.println("-----");  
16          System.out.println("서버를 종료하려면 q를 입력하고 Enter 키를 입력하세요.");  
17          System.out.println("-----");
```

10개의 스레드로 요청을
처리하는 스레드풀 생성





Chapter 20 데이터베이스 입출력

1. jar 파일 다운로드

<https://projectlombok.org/download>

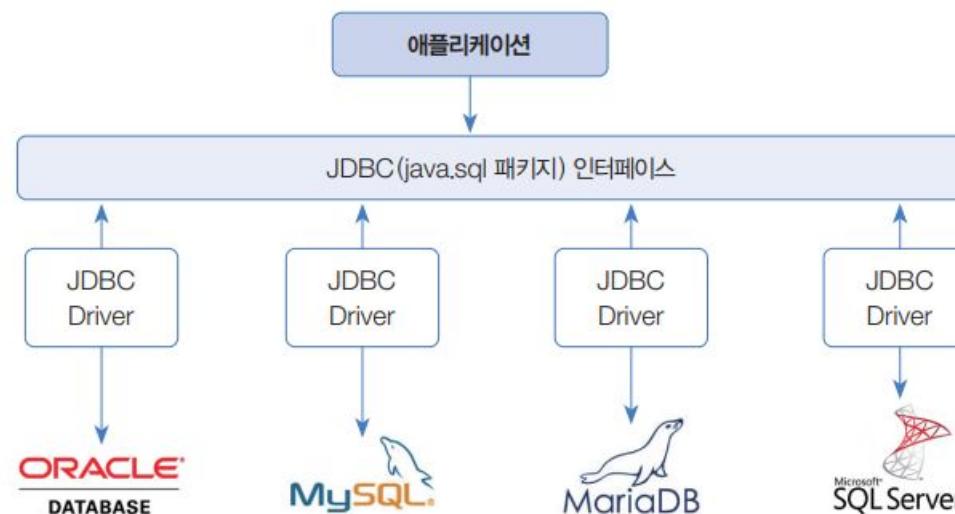
2. jar 파일을 STS/eclipse 실행파일이 있는 위치로 복사

3. cmd 명령 > java -jar lombok.jar

4. STS 설치된 경로 명 입력 후 진행하여 완료

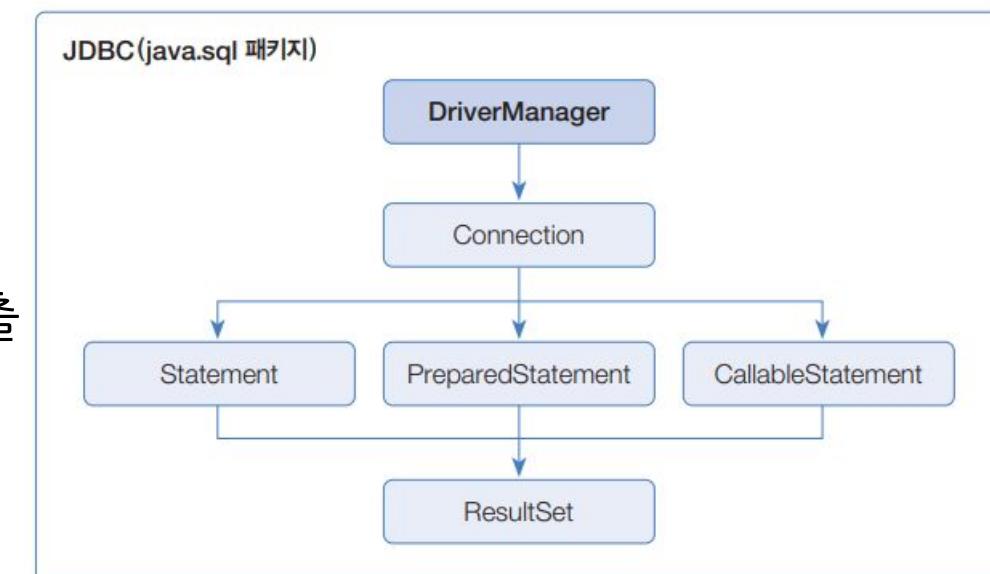
JDBC 라이브러리

- 자바는 데이터베이스(DB)와 연결해서 데이터 입출력 작업을 할 수 있도록 JDBC 라이브러리 (java.sql 패키지)를 제공
- JDBC는 데이터베이스 관리시스템(DBMS)의 종류와 상관없이 동일하게 사용할 수 있는 클래스와 인터페이스로 구성



JDBC Driver

- JDBC 인터페이스를 구현한 것으로, DBMS마다 별도로 다운로드 받아 사용
- DriverManager 클래스: JDBC Driver를 관리하며 DB와 연결해서 Connection 구현 객체를 생성
- Connection 인터페이스: Statement, PreparedStatement, CallableStatement 구현 객체를 생성하며, 트랜잭션 처리 및 DB 연결을 끊을 때 사용
- Statement 인터페이스: SQL의 DDL과 DML 실행 시 사용
- PreparedStatement: SQL의 DDL, DML 문 실행 시 사용.
매개변수화된 SQL 문을 써 편리성과 보안성 유리
- CallableStatement: DB에 저장된 프로시저와 함수를 호출
- ResultSet: DB에서 가져온 데이터를 읽음



Oracle 설치

- Enterprise Edition 설치 파일 다운로드:

<https://www.oracle.com/database/technologies/oracle-database-software-downloads.html#19c>

- C:\Oracle\WINDOWS.X64_193000_db_home

Oracle Base: C:\Oracle
비밀번호: oracle
컨테이너 데이터베이스로 생성: 체크 해제 (매우 중요)

- 프로그램에서 사용할 DB 계정을 생성하기 위해 SQL Plus에서 다음 SQL 문을 실행

```
SQL> create user java identified by oracle;
SQL> grant connect to java;
SQL> grant resource to java;
SQL> grant unlimited tablespace to java;
```

원격 연결

- 원격 연결 요청을 수락하기 위해 Net Configuration Assistant를 실행
- [시작] 메뉴 - [Oracle] - [OraDB19Home1] - [Net Configuration Assistant]

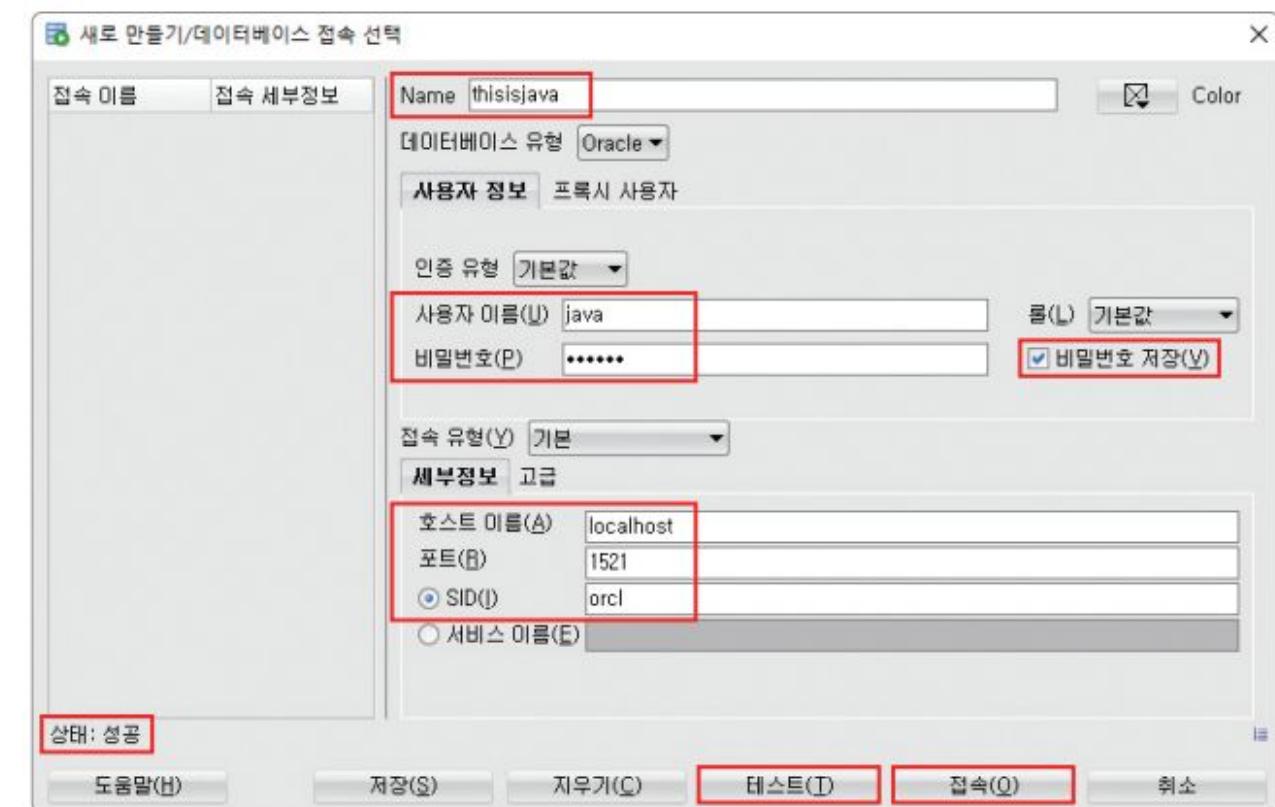
방화벽 해제

- Oracle을 설치한 운영체제의 방화벽 설정에서 1521 포트를 개방
- Windows Defender 방화벽 대화상자에서 [인바운드 규칙]을, 오른쪽 작업 창에서 [새 규칙]을 선택해 설정 변경



SQL Developer

- Oracle DB 모델링부터 DB 상태 확인, SQL 스크립트 및 PL/SQL 개발 등 용이한 무료 Client Tool
- 설치 파일 다운로드: <https://www.oracle.com/tools/downloads/sqldev-downloads.html>
- C:\Oracle\sqldeveloper
- 설치 후 [새로 만들기/데이터베이스 접속 선택] 대화상자에서 설정 후 테스트



데이터베이스 구성

- 테이블, 시퀀스, 프로시저, 함수를 생성하여 데이터베이스를 구성

The screenshot shows the Oracle SQL Developer interface. The main window displays a PL/SQL function named 'user_login'. The code implements a login logic by selecting a password from the 'users' table where the 'userid' matches the input 'a_userid'. It then compares the selected password with the input 'a_userpassword'. If they match, it returns 0; otherwise, it returns 1. If no data is found, it returns 2. The function uses the 'users' table and returns a PLS_INTEGER value.

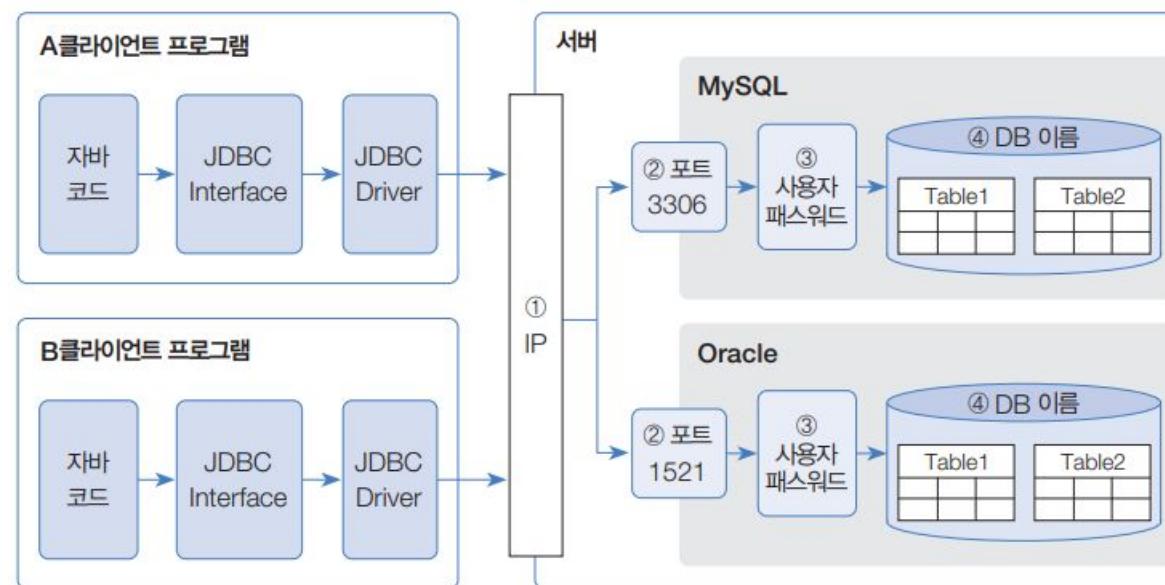
```
CREATE OR REPLACE FUNCTION user_login (
    a_userid      users.userid%TYPE,
    a_userpassword users.userpassword%TYPE
) RETURN PLS_INTEGER
IS
    v_userpassword users.userpassword%TYPE;
    v_result PLS_INTEGER;
BEGIN
    SELECT userpassword INTO v_userpassword
    FROM users
    WHERE userid = a_userid;

    IF v_userpassword = a_userpassword THEN
        RETURN 0;
    ELSE
        RETURN 1;
    END IF;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN 2;
END;
```

The status bar at the bottom indicates: '작업이 완료되었습니다.(0.016초)' (The operation completed successfully.(0.016 seconds)).

데이터베이스 연결

- 클라이언트 프로그램에서 DB와 연결하려면 해당 DBMS의 JDBC Driver가 필요
- ① DBMS가 설치된 컴퓨터의 IP 주소, ② DBMS가 허용하는 포트(Port) 번호, ③ 사용자(DB 계정) 및 비밀번호 ④ 사용하고자 하는 DB 이름 필요



JDBC Driver 설치

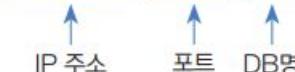
- 로컬 PC에 Oracle을 설치하면 JDBC Driver 파일 찾을 수 있음
(C:\Oracle\WINDOWS.X64_193000_db_home\jdbc\lib\ojdbc8.jar)
- 원격 PC에 Oracle을 설치하면 JDBC Driver만 별도로 다운로드
(<https://mvnrepository.com/artifact/com.oracle.database.jdbc/ojdbc8>)

DB 연결

- Class.forName() 메소드는 문자열로 주어진 JDBC Driver 클래스를 BuildPath에서 찾고, JDBC Driver를 메모리로 로딩

```
Class.forName("oracle.jdbc.OracleDriver");
Connection conn = DriverManager.getConnection("연결 문자열", "사용자", "비밀번호");
```

jdbc:oracle:thin:@localhost:1521/orcl



INSERT 문

- users 테이블에 새로운 사용자 정보를 저장하는 INSERT 문 실행
- INSERT 문을 String 타입 변수 sql에 문자열로 대입

```
INSERT INTO users (userid, username, userpassword, userage, useremail)
VALUES (?, ?, ?, ?, ?)
```

```
String sql = new StringBuilder()
.append("INSERT INTO users (userid, username, userpassword, userage, useremail) ")
.append("VALUES (?, ?, ?, ?, ?)")
.toString();
```

또는

```
String sql = "" +
    "INSERT INTO users (userid, username, userpassword, userage, useremail) " +
    "VALUES (?, ?, ?, ?, ?);"
```

- 매개변수화된 SQL 문을 실행하기 위해 Connection의 prepareStatement() 메소드로부터 PreparedStatement를 얻음

```
PreparedStatement pstmt = conn.prepareStatement(sql);
```

- 값을 지정한 후 executeUpdate() 메소드를 호출하면 SQL 문이 실행되면서 users 테이블에 1개의 행이 저장

```
pstmt.setString(1, "winter");
pstmt.setString(2, "한겨울");
pstmt.setString(3, "12345");
pstmt.setInt(4, 25);
pstmt.setString(5, "winter@mycompany.com");
```

```
int rows = pstmt.executeUpdate();
```

- close() 메소드를 호출하면 PreparedStatement가 사용했던 메모리 해제

```
pstmt.close();
```

UPDATE 문

- JDBC를 이용해서 UPDATE 문을 실행

```
UPDATE boards SET  
    btitle=?,  
    bcontent=?,  
    bfilename=?,  
    bfiledata=?  
WHERE bno=?
```

```
String sql = new StringBuilder()  
    .append("UPDATE boards SET ")  
    .append("btitle=?, ")  
    .append("bcontent=?, ")  
    .append("bfilename=?, ")  
    .append("bfiledata=? ")  
    .append("WHERE bno=?")  
    .toString();
```

- prepareStatement() 메소드로부터 PreparedStatement를 얻고, ?에 해당하는 값을 지정
- executeUpdate() 메소드를 호출. 수정된 행의 수가 리턴

```
PreparedStatement pstmt = conn.prepareStatement(sql);  
pstmt.setString(1, "눈사람");  
pstmt.setString(2, "눈으로 만든 사람");  
pstmt.setString(3, "snowman.jpg");  
pstmt.setBlob(4, new FileInputStream("src/ch20/oracle/sec07/snowman.jpg"));  
pstmt.setInt(5, 3);
```

```
int rows = pstmt.executeUpdate();
```

DELETE 문

- JDBC를 이용해서 DELETE 문 실행. 매개변수화된 DELETE 문을 String 타입 변수 sql에 대입

```
DELETE FROM boards WHERE bwriter=?
```

```
String sql = "DELETE FROM boards WHERE bwriter=?";
```

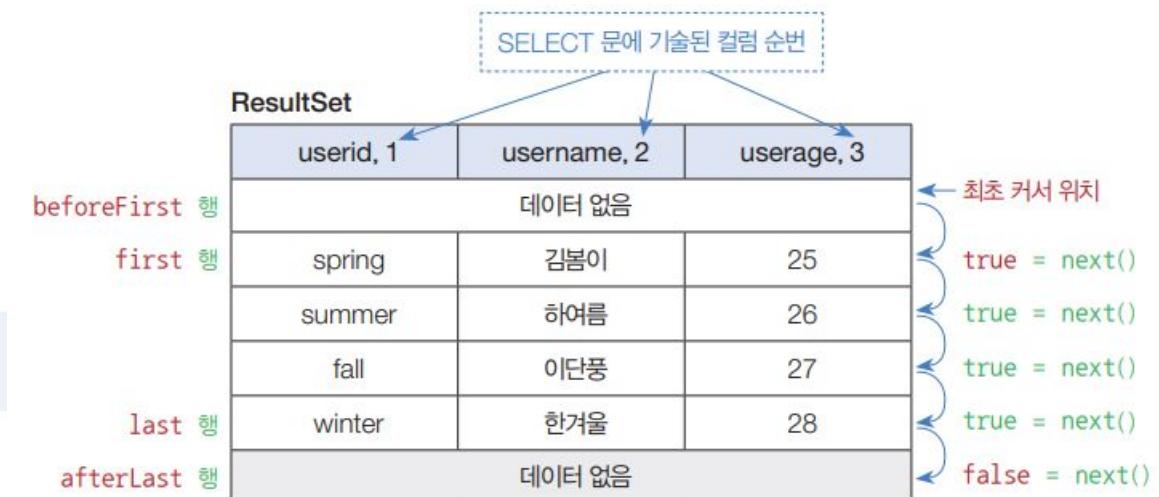
- prepareStatement() 메소드로부터 PreparedStatement를 얻고 ?에 값을 지정한 후, executeUpdate로 SQL 문을 실행

```
String sql = "DELETE FROM boards WHERE bwriter=?";
PreparedStatement pstmt = conn.prepareStatement(sql);
pstmt.setString(1, "winter");
int rows = pstmt.executeUpdate();
```

ResultSet 구조

- SELECT 문에 기술된 컬럼으로 구성된 행(row)의 집합

```
SELECT userid, username, usageage FROM users
```



- 커서cursor가 있는 행의 데이터만 읽을 수 있음
- first 행을 읽으려면 next() 메소드로 커서 이동

```
boolean result = rs.next();
```

1개의 데이터 행만 가져올 경우

```
ResultSet rs = pstmt.executeQuery();
if(rs.next()) {
    //첫 번째 데이터 행 처리
} else {
    //afterLast 행으로 이동했을 경우
}
```

n개의 데이터 행을 가져올 경우

```
ResultSet rs = pstmt.executeQuery();
while(rs.next()) {
    //last 행까지 이동하면서 데이터 행 처리
}
//afterLast 행으로 이동했을 경우
```

데이터 행 읽기

- 커서가 있는 데이터 행에서 각 컬럼의 값은 Getter 메소드로 읽음
- SELECT 문에 연산식이나 함수 호출이 포함되어 있다면 컬럼 이름 대신에 컬럼 순번으로 읽어야 함

컬럼 이름으로 읽기

```
String userId =  
    rs.getString("userid");  
String userName =  
    rs.getString("username");  
int userAge = rs.getInt("usage");
```

```
SELECT userid, usage - 1  
FROM users
```

컬럼 순번으로 읽기

```
String userId = rs.getString(1);  
String userName = rs.getString(2);  
int userAge = rs.getInt(3);
```

```
String userId =  
    rs.getString("userid");  
int userAge = rs.getInt(2);
```

사용자 정보 읽기

- 사용자 정보를 가져오는 SELECT 문.
prepareStatement() 메소드로부터
PreparedStatement를 얻고, ?에 값을 지정
- executeQuery() 메소드로 SELECT 문을
실행해서 ResultSet을 얻음.
- if 문을 이용해서 next() 메소드가 true를
리턴할 경우에는 데이터 행을 User 객체에
저장하고 출력

```
String sql = "" +
    "SELECT userid, username, userpassword, userage, useremail " +
    "FROM users " +
    "WHERE userid=?";
```

```
PreparedStatement pstmt = conn.prepareStatement(sql);
pstmt.setString(1, "winter");
```

```
ResultSet rs = pstmt.executeQuery();
if(rs.next()) {                                //1개의 데이터 행을 가져왔을 경우
    User user = new User();
    user.setUserId(rs.getString("userid"));
    user.setUserName(rs.getString("username"));
    user.setUserPassword(rs.getString("userpassword"));
    user.setUserAge(rs.getInt(4));           //컬럼 순번을 이용해서 컬럼 지정
    user.setUserEmail(rs.getString(5));      //컬럼 순번을 이용해서 컬럼 지정
    System.out.println(user);
} else {                                         //데이터 행을 가져오지 않았을 경우
    System.out.println("사용자 아이디가 존재하지 않음");
}
```

게시물 정보 읽기

- boards 테이블에서 bwriter가 winter인 게시물의 정보를 가져오기

- bwriter가 winter인 게시물 정보를 가져오는 SELECT 문. `prepareStatement()` 메소드로부터 `PreparedStatement`을 얻고, ?에 값을 지정

```
String sql = "" +  
    "SELECT bno, btitle, bcontent, bwriter, bdate, bfilename, bfiledata " +  
    "FROM boards " +  
    "WHERE bwriter=?";
```

```
PreparedStatement pstmt = conn.prepareStatement(sql);  
pstmt.setString(1, "winter");
```

- executeQuery() 메소드로 SELECT 문을 실행해서 ResultSet을 얻음
- while 문을 이용해서 next() 메소드가 false를 리턴할 때까지 반복해서 데이터 행을 Board 객체에 저장하고 출력한다
- Blob 객체에 저장된 바이너리 데이터를 얻기 위해서는 입력 스트림 또는 배열을 얻어냄
- Blob 객체에서 InputStream을 얻고, 읽은 바이트를 파일로 저장

```

ResultSet rs = pstmt.executeQuery();
while(rs.next()) {
    //데이터 행을 읽고 Board 객체에 저장
    Board board = new Board();
    board.setBno(rs.getInt("bno"));
    board.setBtitle(rs.getString("btitle"));
    board.setBcontent(rs.getString("bcontent"));
    board.setBwriter(rs.getString("bwriter"));
    board.setBdate(rs.getDate("bdate"));
    board.setBfilename(rs.getString("bfilename"));
    board.setBfiledata(rs.getBlob("bfiledata"));

    //콘솔에 출력
    System.out.println(board);
}

```

```

Blob blob = board.getBfiledata();
InputStream is =
    blob.getBinaryStream();

```

```

Blob blob = board.getBfiledata();
byte[] bytes = blob.getBytes(0,
    blob.length());

```

```

InputStream is = blob.getBinaryStream();
OutputStream os = new FileOutputStream("C:/Temp/" + board.getBfilename());
is.transferTo(os);
os.flush();
os.close();
is.close();

```

프로시저와 함수

- Oracle DB에 저장되는 PL/SQL 프로그램. 클라이언트 프로그램에서 매개값과 함께 프로시저 또는 함수를 호출하면 DB 내부에서 SQL 문을 실행하고, 실행 결과를 클라이언트 프로그램으로 돌려줌
- JDBC에서 프로시저와 함수를 호출 시 CallableStatement를 사용. 프로시저와 함수의 매개변수화된 호출문을 작성하고 Connection의 prepareCall() 메소드로부터 CallableStatement 객체를 얻음

```
//프로시저를 호출할 경우  
String sql = "{ call 프로시저명(?, ?, ...) }";  
CallableStatement cstmt = conn.prepareCall(sql);
```

```
//함수를 호출할 경우  
String sql = "{ ? = call 함수명(?, ?, ...) }"  
CallableStatement cstmt = conn.prepareCall(sql);
```

- 프로시저도 리턴값과 유사한 OUT 타입의 매개변수를 가질 수 있기 때문에 괄호 안의 ?중 일부는 OUT값(리턴값)일 수 있음

- prepareCall() 메소드로 CallableStatement을 얻으면 리턴값에 해당하는 ?는 registerOutParameter() 메소드로 지정하고, 그 이외의 ?는 호출 시 필요한 매개값으로 Setter

```
String sql = "{ call 프로시저명(?, ?, ?) }";
CallableStatement cstmt = conn.prepareCall(sql);
cstmt.setString(1, "값");           //프로시저의 첫 번째 매개값
cstmt.setString(2, "값");           //프로시저의 두 번째 매개값
cstmt.registerOutParameter(3, 리턴타입); //세 번째 ?는 OUT값(리턴값)임을 지정
```

```
String sql = "{? = call 함수명(?, ?)}";
CallableStatement cstmt = conn.prepareCall(sql);
cstmt.registerOutParameter(1, 리턴타입); //첫 번째 ?는 리턴값임을 지정
cstmt.setString(2, "값");             //함수의 첫 번째 매개값
cstmt.setString(3, "값");             //함수의 두 번째 매개값
```

- execute() 메소드로 프로시저 또는 함수 호출. Getter 메소드로 리턴값 얻음

```
cstmt.execute();
```

프로시저

```
int result = cstmt.getInt(3);
```

함수

```
int result = cstmt.getInt(1);
```

프로시저 호출

- IN 매개변수는 호출 시 필요한 매개값으로 사용되며, OUT 매개변수는 리턴값으로 사용
- 매개변수화된 호출문을 작성하고 CallableStatement를 얻음
- ?의 값을 지정하고 리턴 타입을 지정
- 프로시저를 실행하고 리턴값 얻음

```
CREATE OR REPLACE PROCEDURE user_create (
    a_userid          IN      users.userid%TYPE,
    a_username        IN      users.username%TYPE,
    a_userpassword   IN      users.userpassword%TYPE,
    a_usage           IN      users.usage%TYPE,
    a_useremail      IN      users.useremail%TYPE,
    a_rows            OUT     PLS_INTEGER
)
...
...
```

```
String sql = "{call user_create(?, ?, ?, ?, ?, ?)}";
CallableStatement cstmt = conn.prepareCall(sql);
```

```
cstmt.setString(1, "summer");
cstmt.setString(2, "한여름");
cstmt.setString(3, "12345");
cstmt.setInt(4, 26);
cstmt.setString(5, "summer@mycompany.com");
cstmt.registerOutParameter(6, Types.INTEGER);
```

```
cstmt.execute();
int rows = cstmt.getInt(6); //6번째 ? 값 얻기
```

함수 호출

- user_login()은 2개의 매개변수와 PLS_INTEGER 리턴 타입으로 구성
- 함수를 호출하기 위해 매개변수화된 호출문을 작성하고 CallableStatement를 얻음
- ?의 값을 지정하고 리턴 타입을 지정
- user_login() 함수는 userid와 userpassword가 일치하면 0을, userpassword가 틀리면 1을, userid가 존재하지 않으면 2를 리턴

```
CREATE OR REPLACE FUNCTION user_login (
    a_userid      users.userid%TYPE,
    a_userpassword users.userpassword%TYPE
) RETURN PLS_INTEGER
...

```

```
String sql = "{? = call user_login(?, ?)}";
CallableStatement cstmt = conn.prepareCall(sql);
```

```
cstmt.registerOutParameter(1, Types.INTEGER);
cstmt.setString(2, "winter");
cstmt.setString(3, "12345");
```

```
cstmt.execute();
int result = cstmt.getInt(1); //첫 번째 ? 값 얻기, 0|1|2 중 하나
```

트랜잭션

- 기능 처리의 최소 단위. 하나의 기능은 여러 소작업들로 구성
- 트랜잭션은 소작업들이 모두 성공하거나 실패해야 함

계좌 이체(트랜잭션)

//출금 작업

```
UPDATE accounts SET balance=balance-이체금액 WHERE ano=출금계좌번호
```

//입금 작업

```
UPDATE accounts SET balance=balance+이체금액 WHERE ano=입금계좌번호
```

- 커밋은 내부 작업을 모두 성공 처리하고, 롤백은 실행 전으로 돌아간다는 의미에서 모두 실패 처리
- JDBC에서 트랜잭션을 제어 시 Connection의 setAutoCommit() 메소드로 자동 커밋 기능을 꺼야

```
conn.setAutoCommit(false);
```

메인 메뉴

- main() 메소드는 BoardExample 객체를 생성하고 list() 메소드를 호출. list() 메소드는 게시물 목록을 출력하고 mainMenu() 메소드를 호출

메인 메뉴 선택 기능

- 키보드 입력을 받기 위해 Scanner 필드를 추가. mainMenu() 메소드에서 키보드 입력을 받기 위해 nextLine() 메소드를 호출. 메뉴 선택 번호에 따라 해당 메소드를 호출

Board 클래스 작성

- boards 테이블의 한 개의 행(게시물)을 저장할 Board 클래스를 작성.
- 컬럼 개수와 타입에 맞게 필드를 선언하고, 롬복 @Data 어노테이션을 이용해서 Getter, Setter, toString() 메소드를 자동 생성

게시물 목록 기능

- DB 연결이 필요하므로 Connection 필드를 추가하고, 생성자에서 DB 연결. boards 테이블에서 게시물 정보들을 가져와서 게시물 목록으로 출력하도록 list() 메소드를 수정

게시물 생성 기능

- 메인 메뉴에서 '1.Create'를 선택했을 때 호출되는 create() 메소드 수정

게시물 읽기 기능

- 메인 메뉴에서 '2.Read'를 선택했을 때 호출되는 read() 메소드 수정

게시물 수정 기능

- read() 메소드에서 보조 메뉴 '1.Update|2.Delete|3.List'를 추가하고, 보조 메뉴에서 '1.Update'를 선택하면 update() 메소드가, '2.Delete'를 선택하면 delete() 메소드가 호출
- update() 메소드는 매개값으로 받은 Board 객체를 수정해서 boards 테이블의 게시물 정보를 수정

게시물 삭제 기능

- 게시물 수정 기능을 구현할 때 보조 메뉴에서 '2.Delete'를 선택했을 때 delete() 메소드가 호출. delete() 메소드를 수정해 매개값으로 받은 Board 객체에서 bno를 얻어 boards 테이블에서 해당 게시물을 삭제

게시물 전체 삭제 기능

- 메인 메뉴에서 '3.Clear'를 선택했을 때 호출되는 clear() 메소드 수정

종료 기능

- 메인 메뉴에서 '4.Exit'를 선택했을 때 호출되는 exit() 메소드 수정

