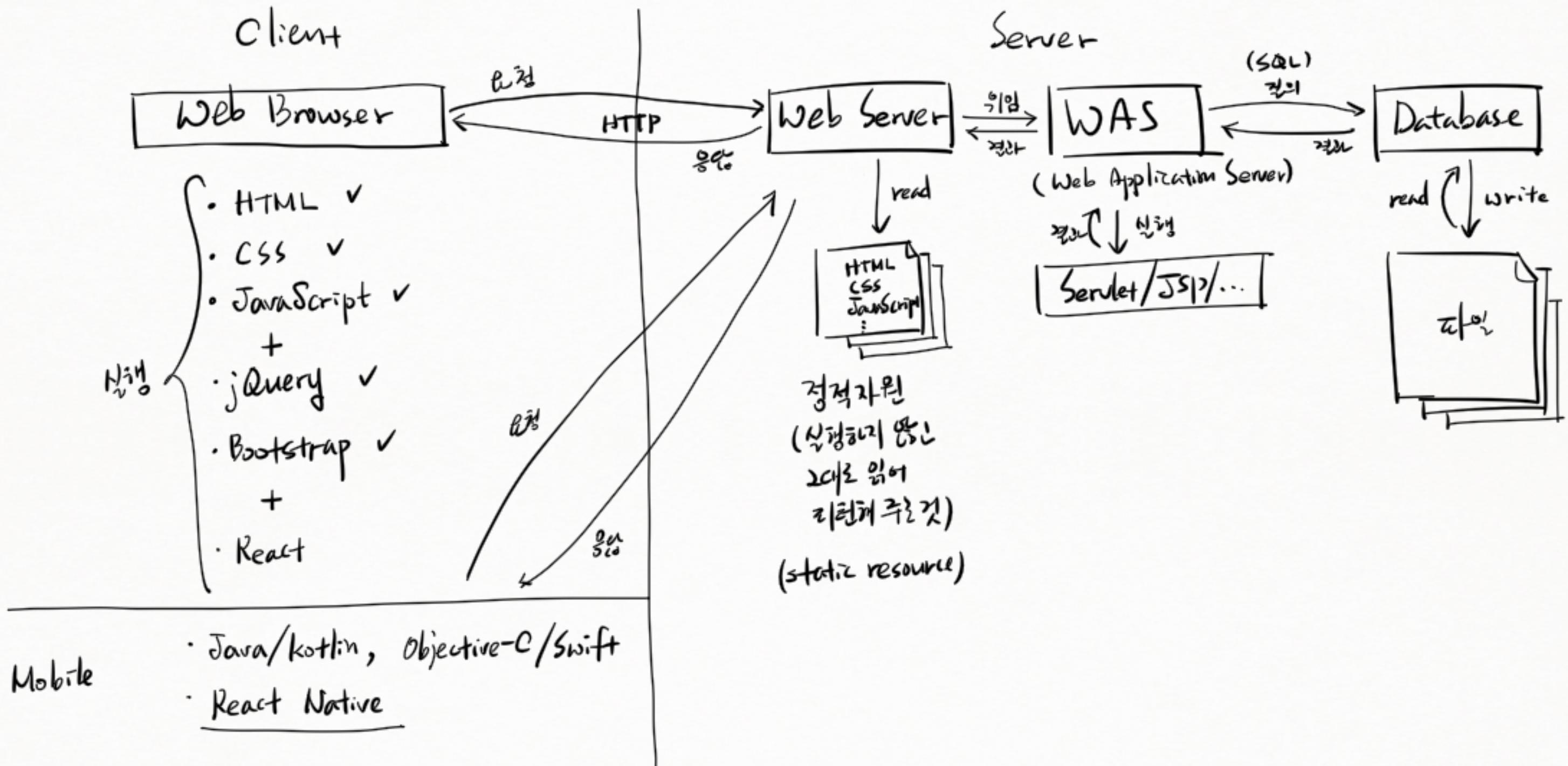
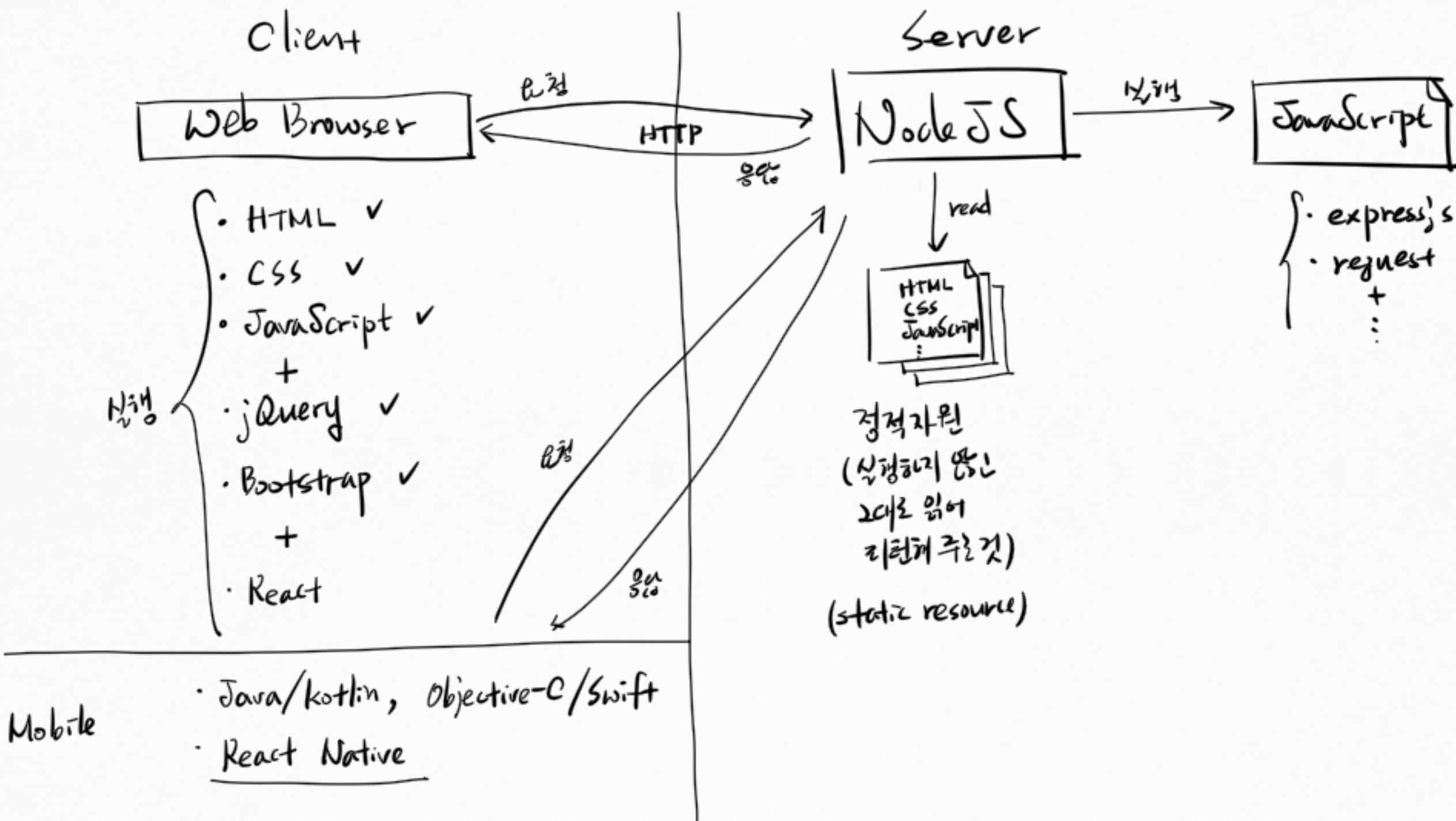


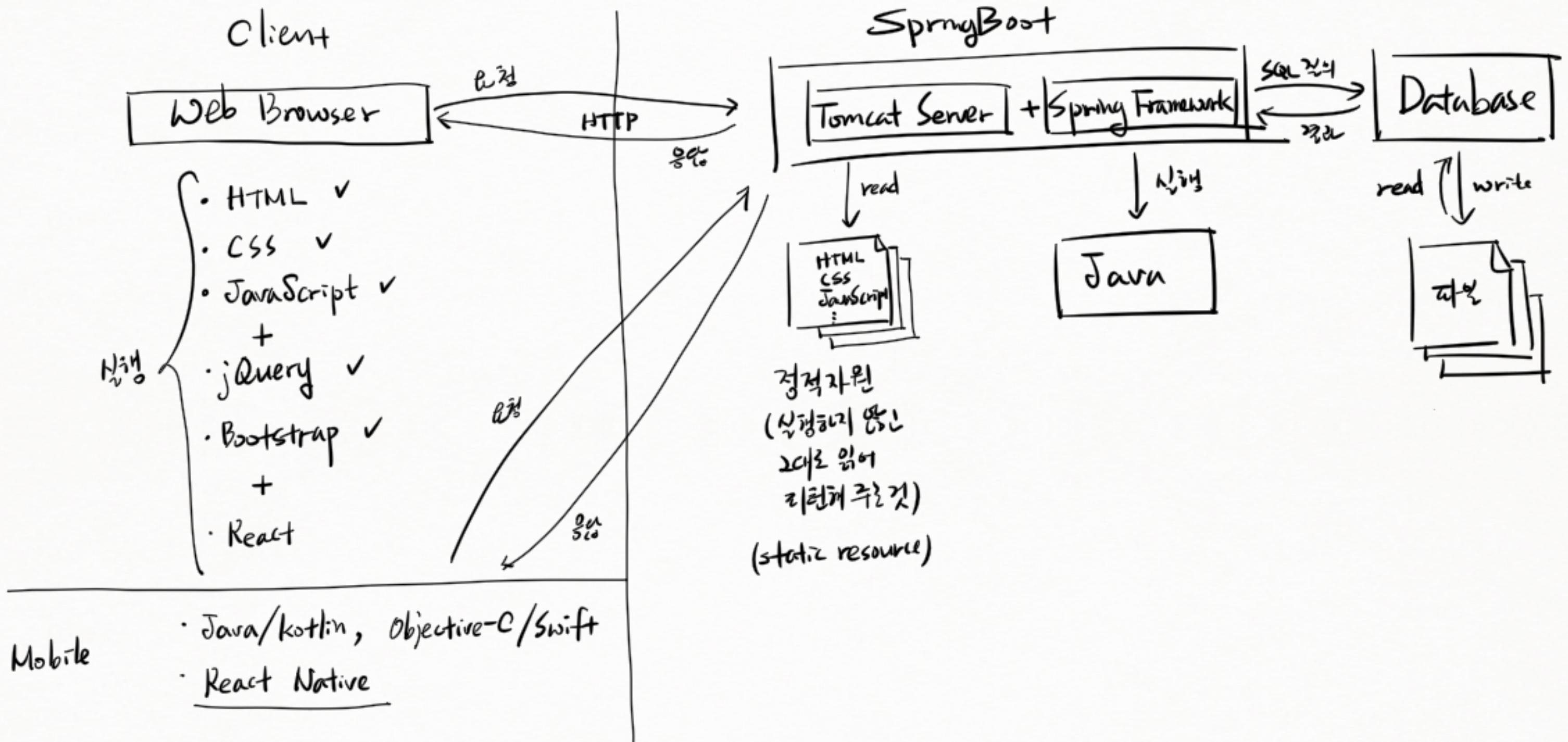
* Web Application Architecture by Java



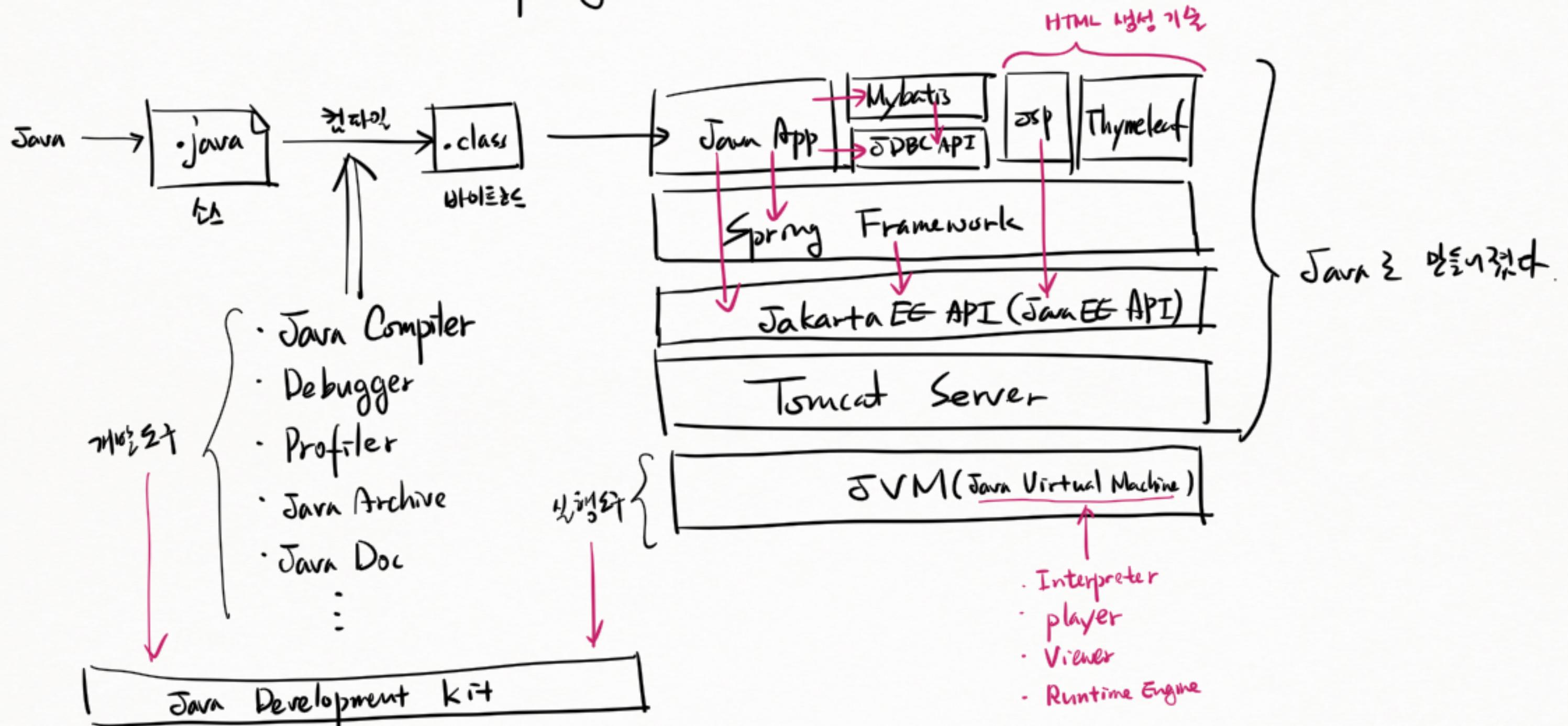
* Node.js Web Application Architecture



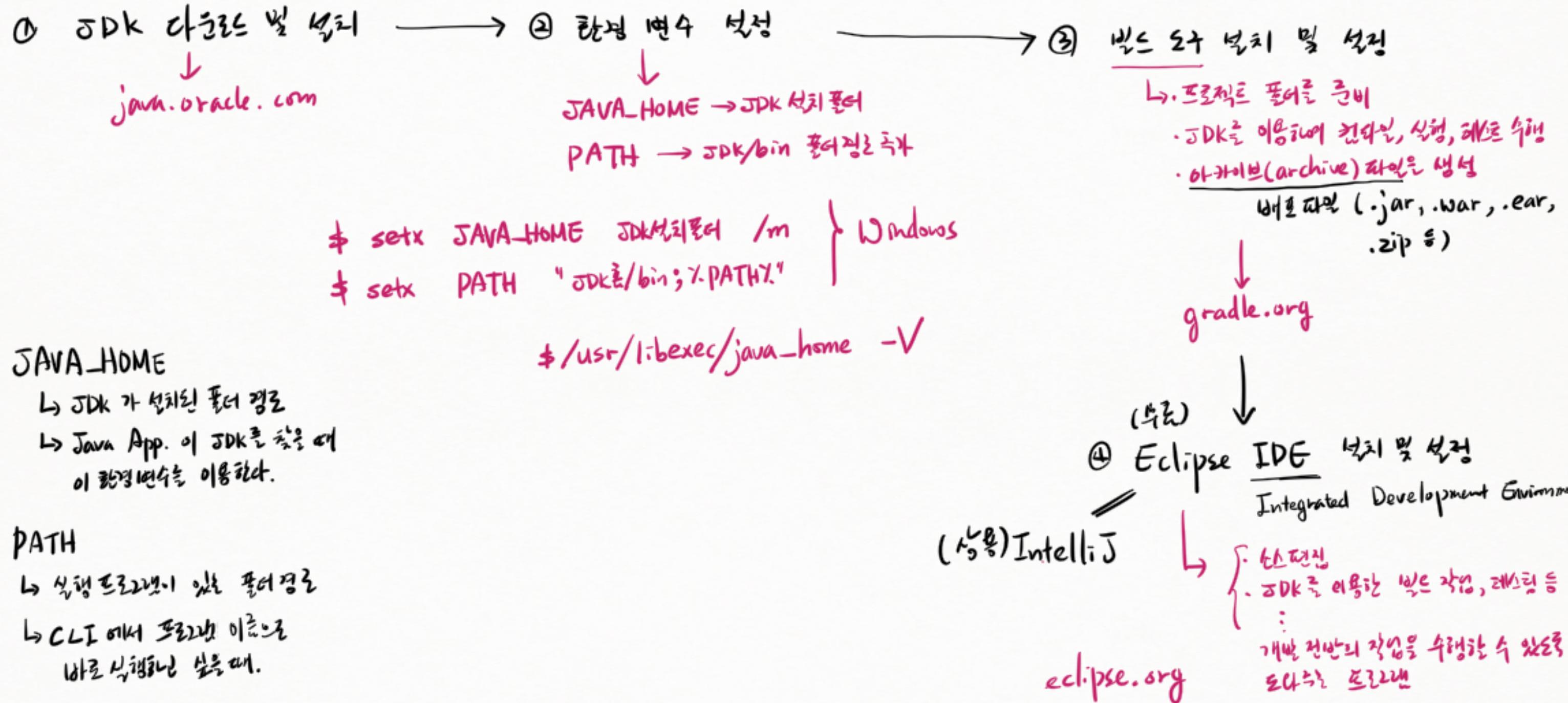
* SpringBoot Web Application Architecture



* SpringBoot 기술 스택

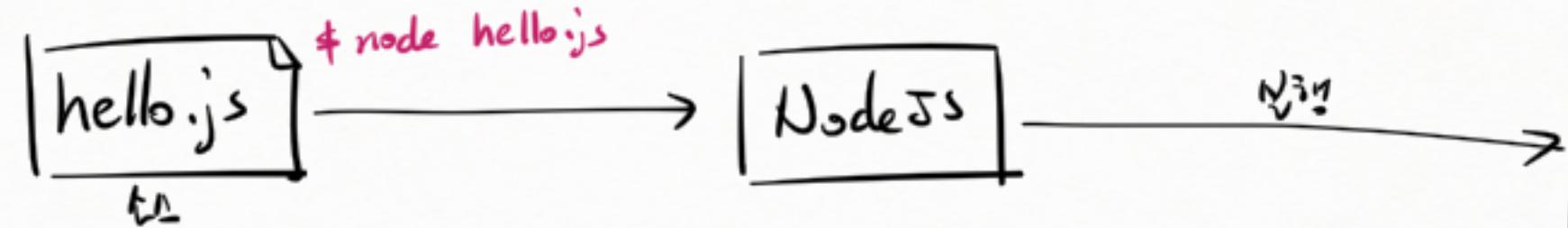


* Java 프로그래밍 준비

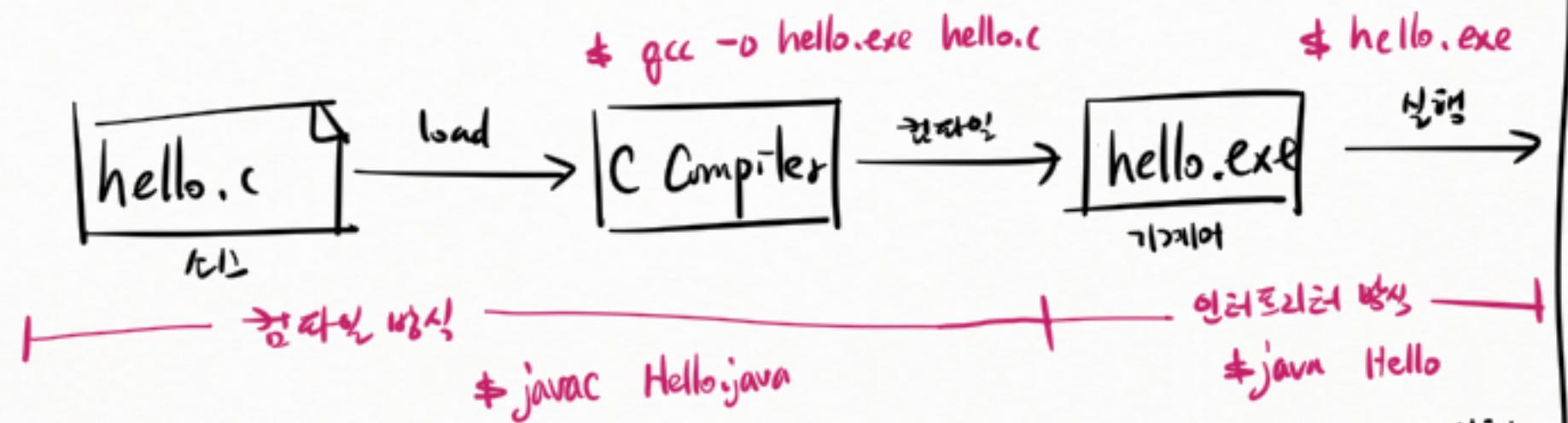


* App. 구현 방식 및 차이

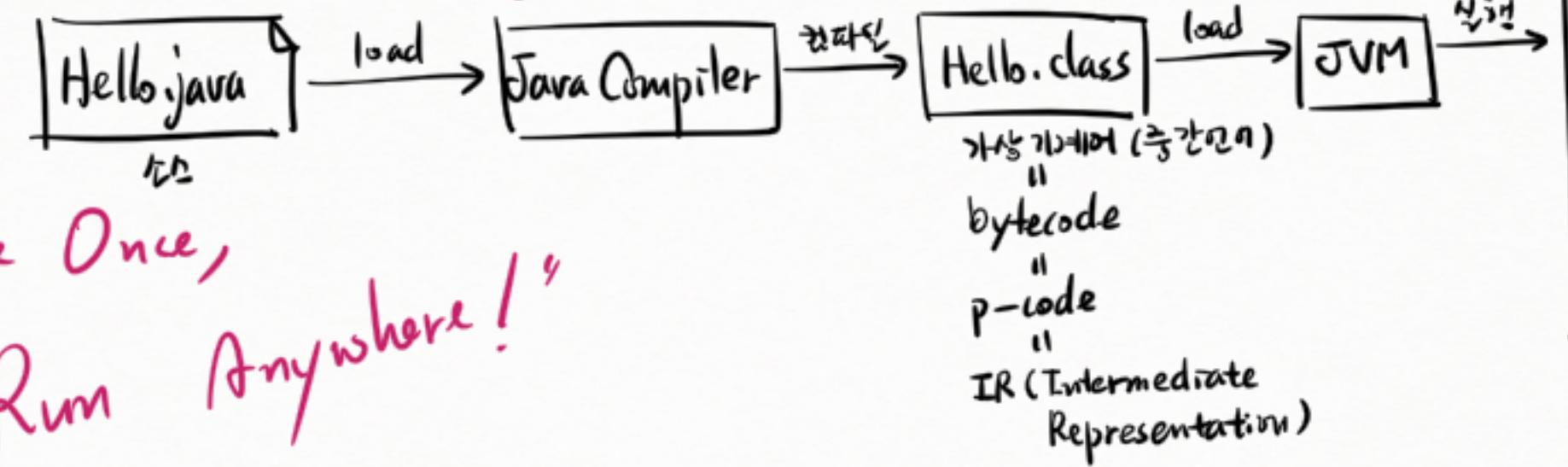
① 인터프리터 방식



② 컴파일 방식



③ 혼합형 방식
(hybrid)



"Write Once,
Run Anywhere!"

* Hybrid 방식을 도입한 이유

- 선수 간자적인 방식
보다 나은 이유
- ① OS마다 따로 컴파일 할 필요가 없다.
↳ 동일한 바이트코드 생성
 - ② OS용 JVM이 설치되어 있으면 실행할 수 있다.

소스



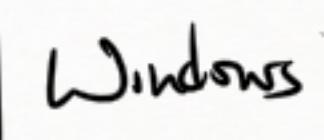
컴파일

바이트코드

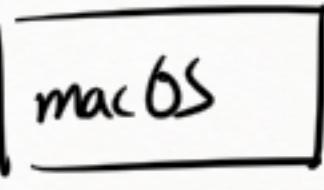
- 방법검사 수행 → 불법 오류를 모두 찾아낸다
- 최적화 → 실행 성능 향상



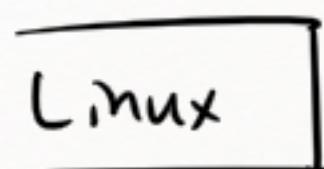
→



→



→



선수 인터프리터 방식
보다 나은 이유

- ③ 컴파일과정이 실행의 오류를 모두 찾아낸다
- ④ 일반 기계어는 하드웨어 기계어에 가까운 언어로
변환된 명령을 실행하니 대용량
소스에 작성된 명령을 실행하는 것보다
실행속도가 빠르다.

바이트코드 툴레이저

- 인터프리터
- 전파있 인진
- 베타일 머신

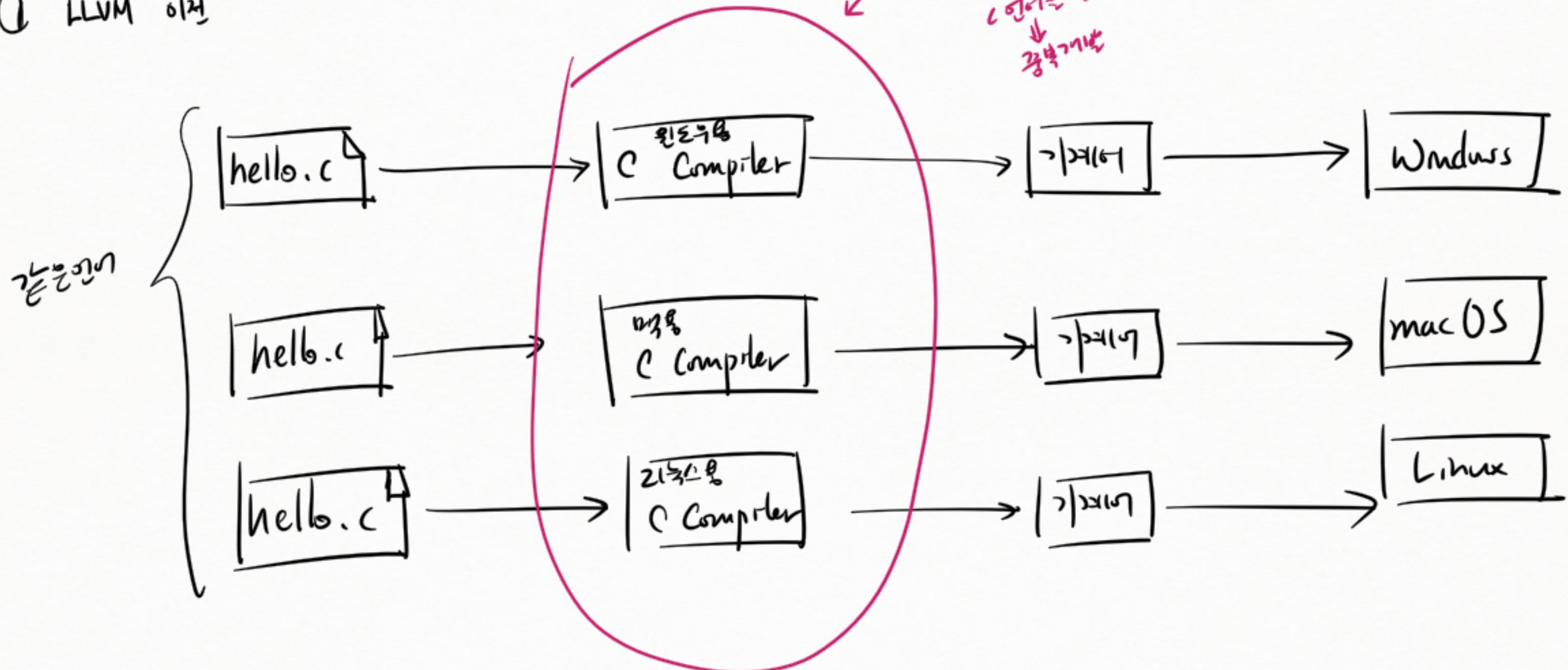
실행할 코드와
불법 오류검사를

하는
선수 인터프리터 방식보다
실행속도가 빠르다.

* LLVM 저작자

Low Level Virtual Machine

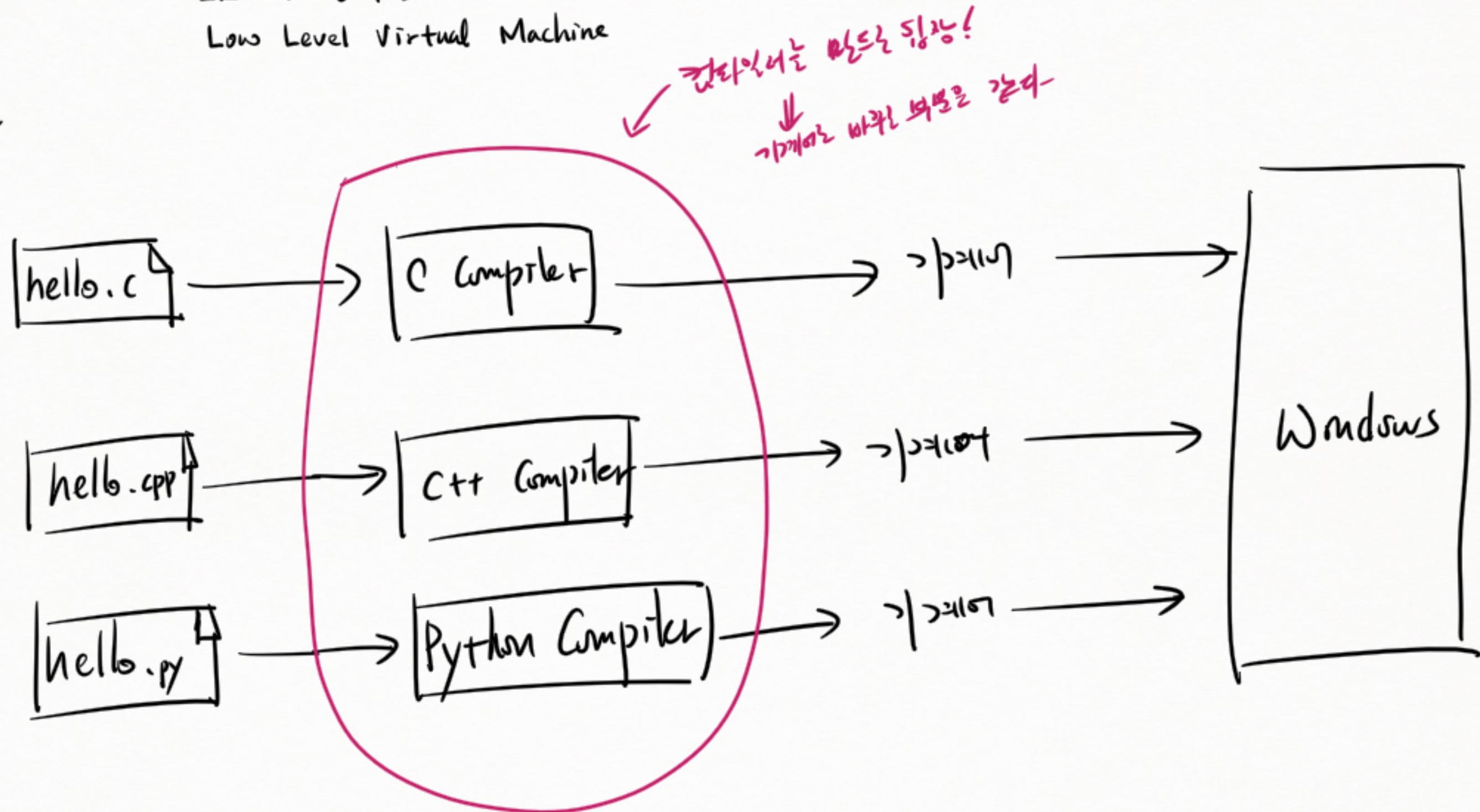
① LLVM 이전



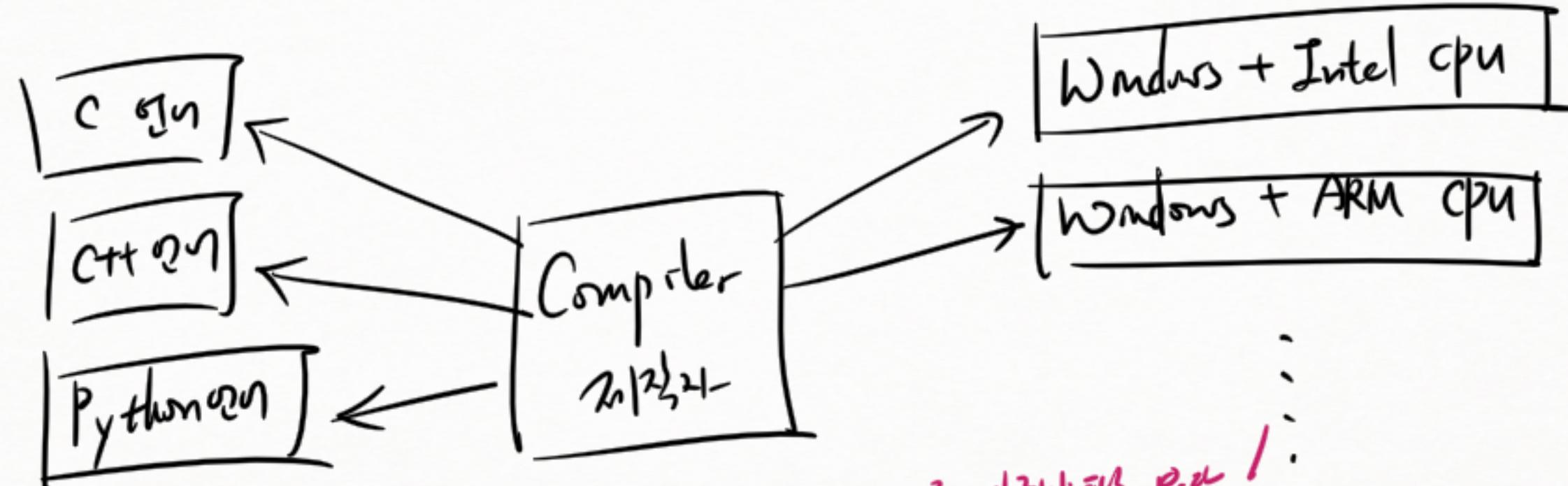
* LLVM 저작자

Low Level Virtual Machine

① LLVM 이전



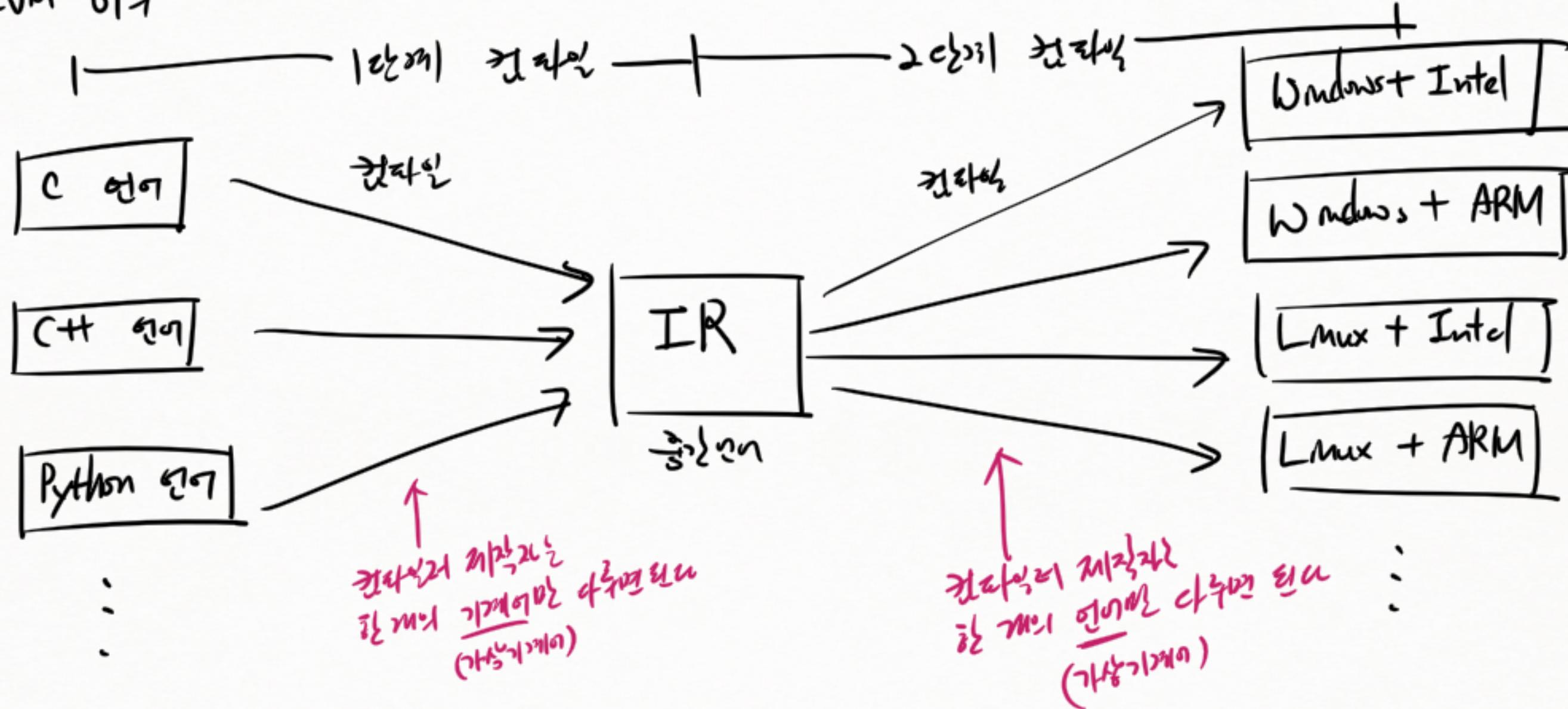
+ 가끔 틀렸을 때 예제



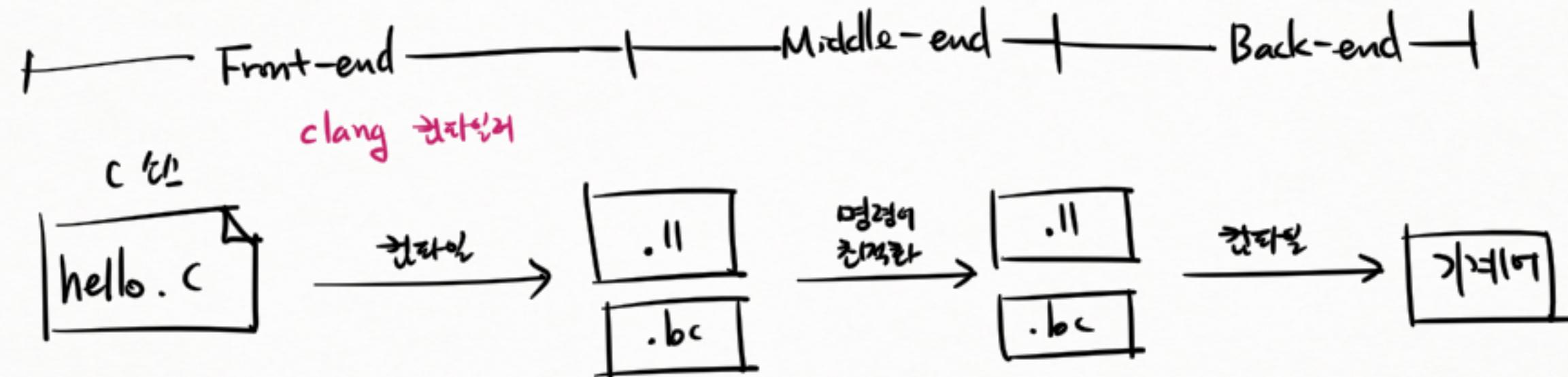
- 어떤 언어를 선택해도 되는지!

* LLVM 10장은 사용하는 기준
↳ 새 프로그램 언어와 컴파일러를 만들기 쉽다.

② LLVM 이후



* LLVM 컴파일 과정

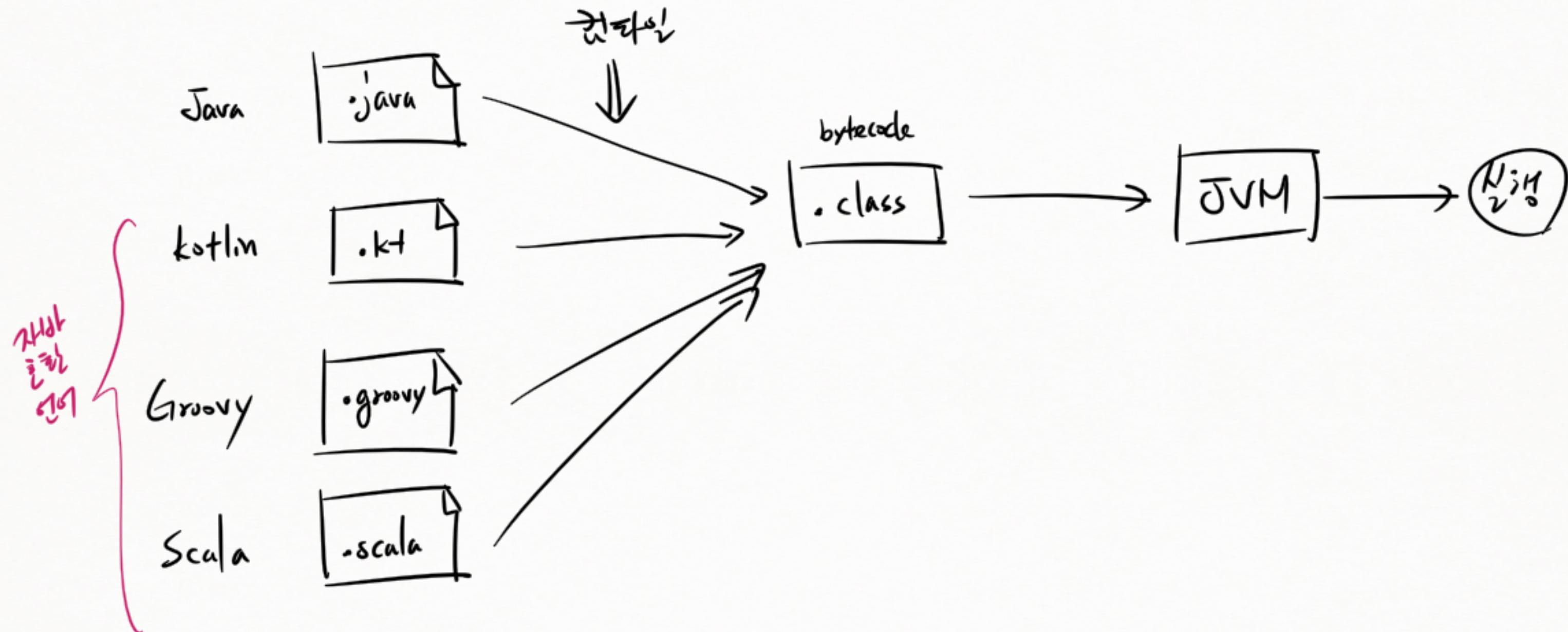


LLVM Assembly

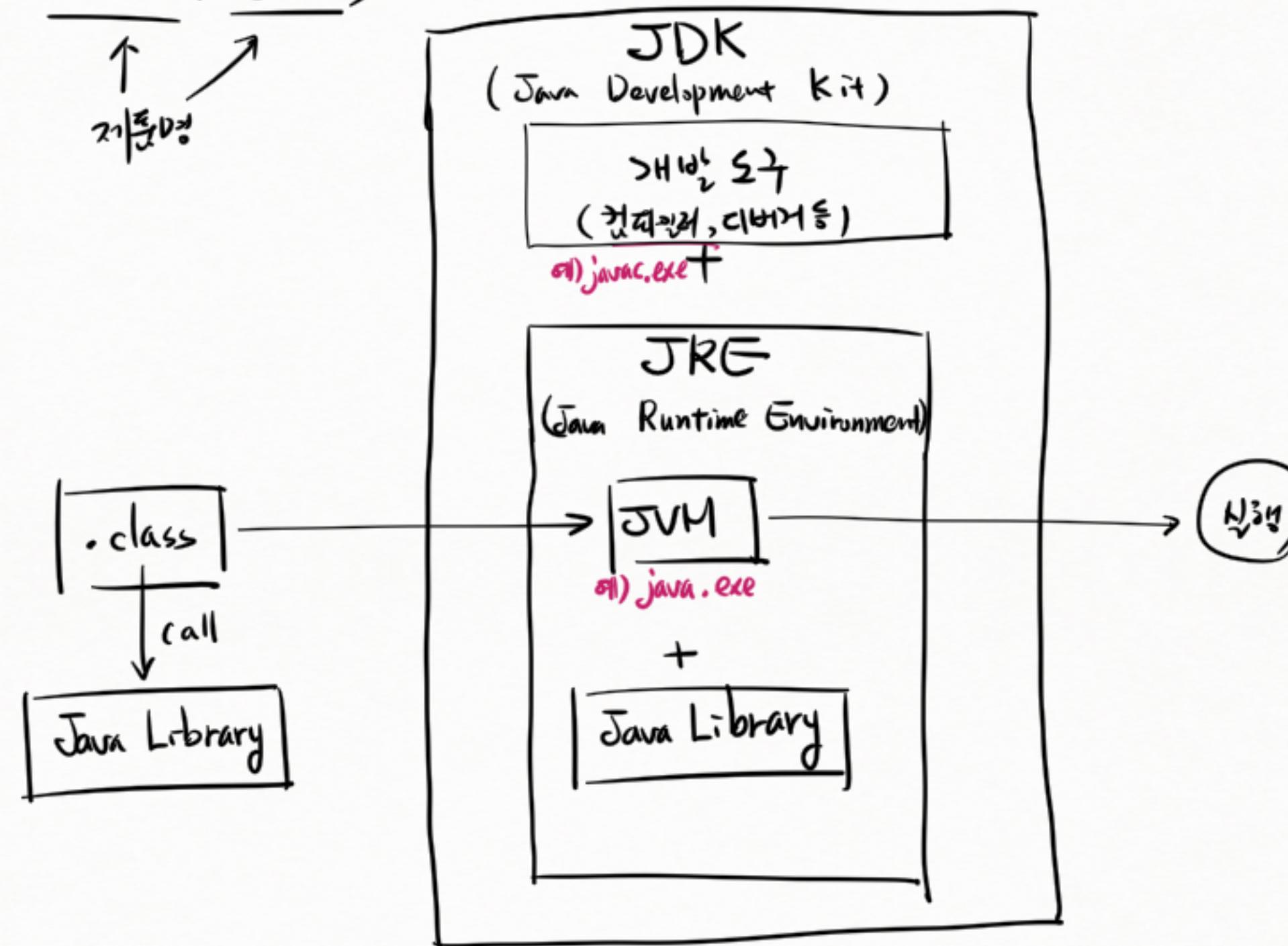
LLVM bitcode

LLVM bytecode

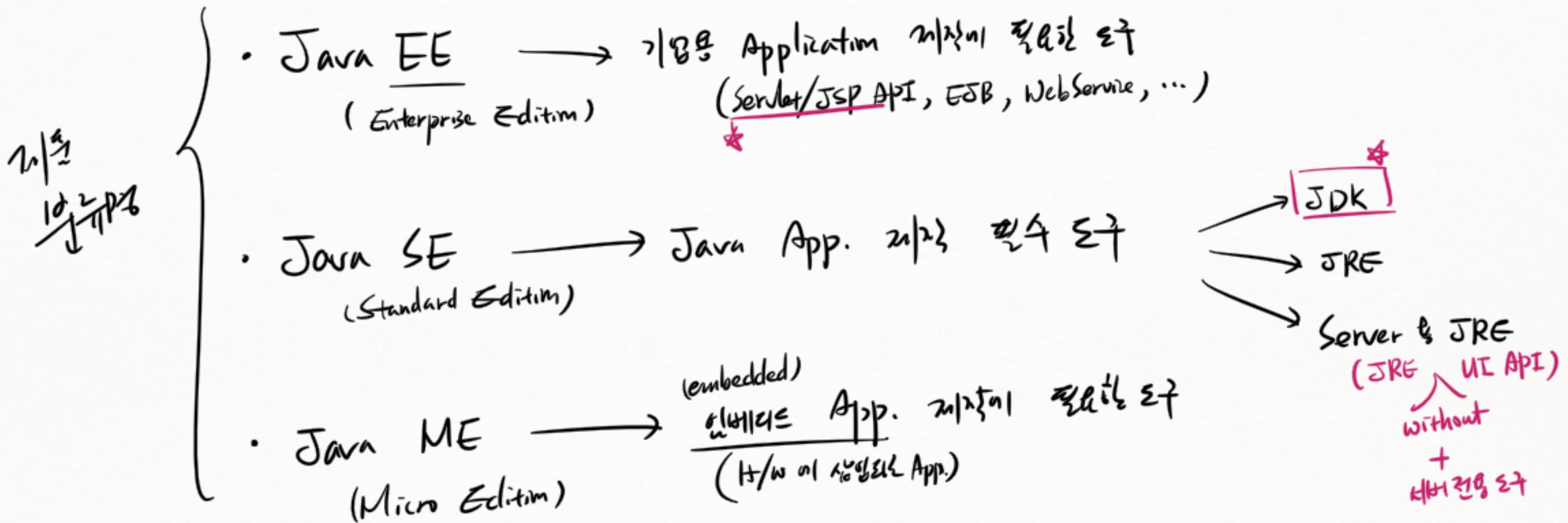
* Java et LLVM



* JDK, JRE, JVM



* Java EE, Java SE, Java ME



* Eclipse IDE



+ plug-in ⇒ 개별 도구 박스

- JDT
- CDT
- ...

* Java Project 폴더 구조

① 프로젝트 폴더 생성

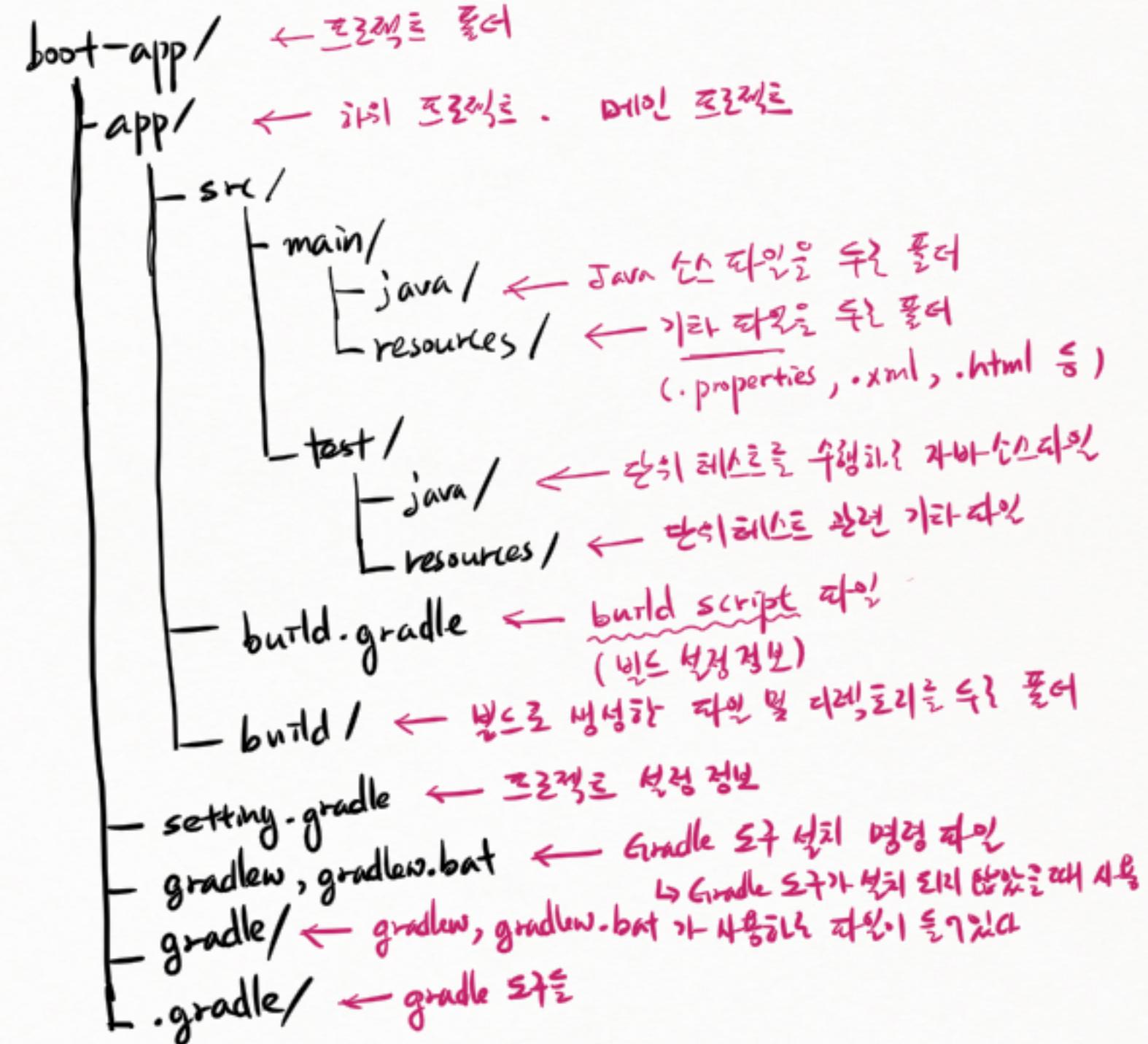
c:\Users\bitcamp\git\bitcamp-ncp\
User Home
↳ boot-project\

② 'boot-project\' 폴더는 Java 프로젝트 폴더로 초기화

\$ gradle init

③ 기본 이제 프로그램 실행

\$ gradle -q run



* SpringBoot 프로젝트 만들기

spring.io 사이트 접속 → Spring Boot 메뉴 → Spring Initializr 실행

- project: gradle-groovy

- Language: Java

- Spring Boot: 3.0.1

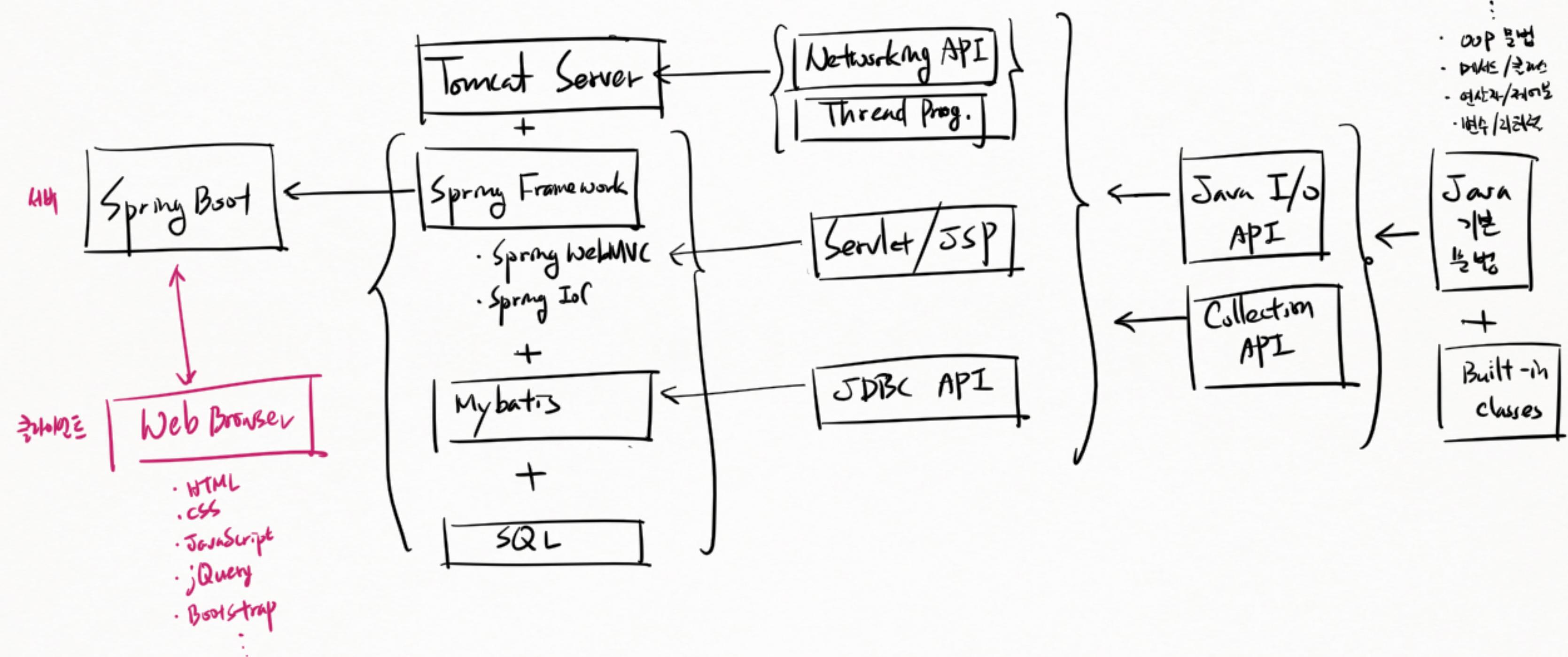
- Dependencies:

- ① Spring Boot Dev Tools ← 자동로딩

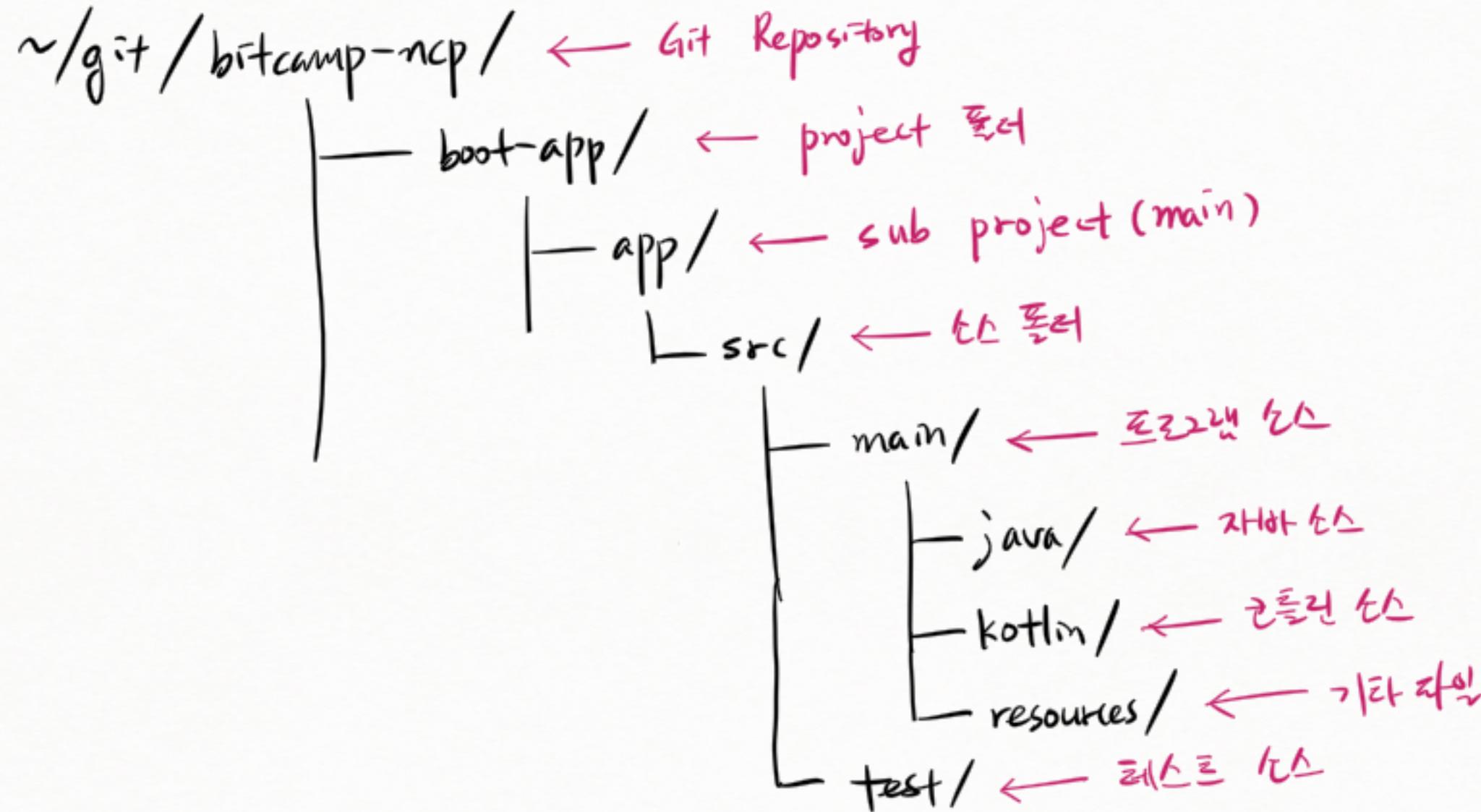
- ② Spring Configuration Processor ← properties 대체
자동로딩

- ③ Spring Web

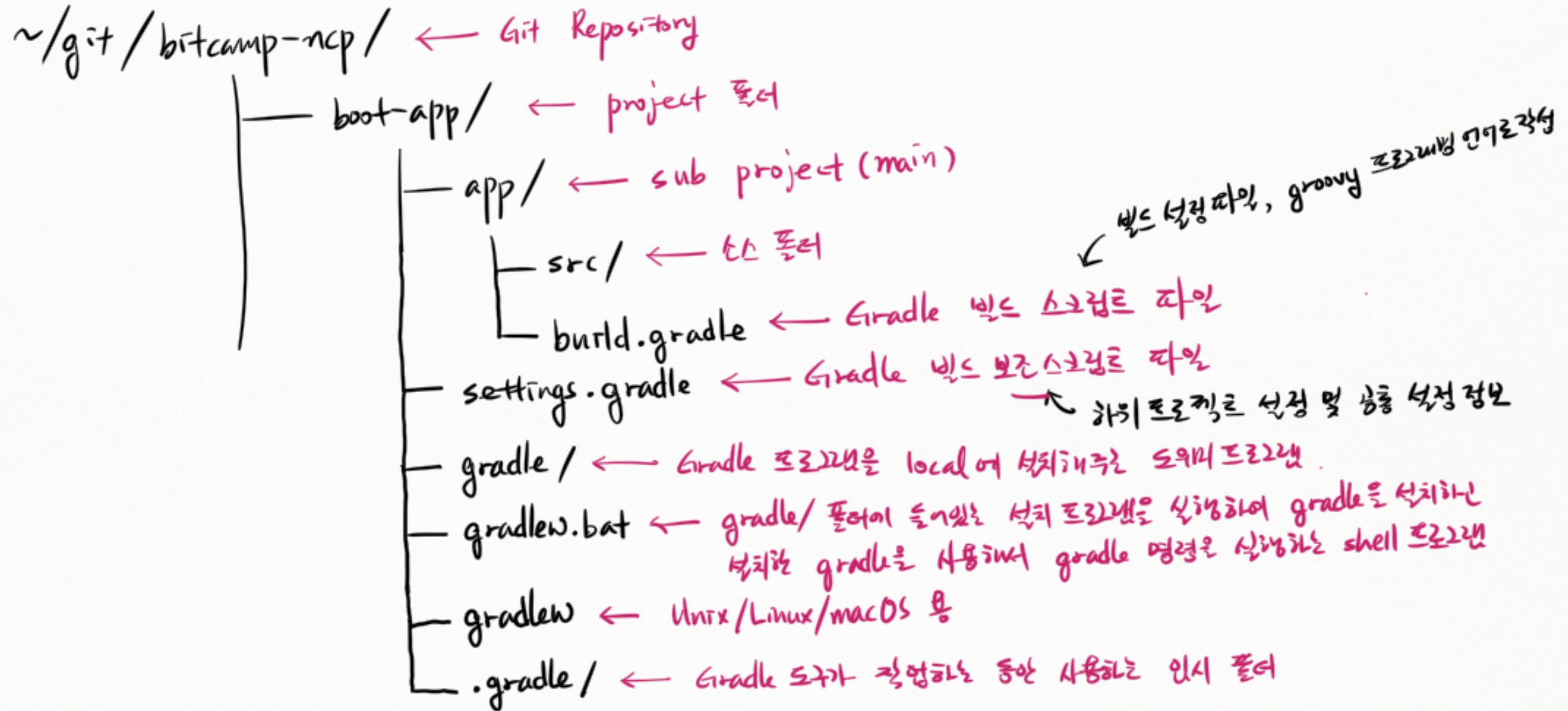
* Java 와서 Spring Boot 까지



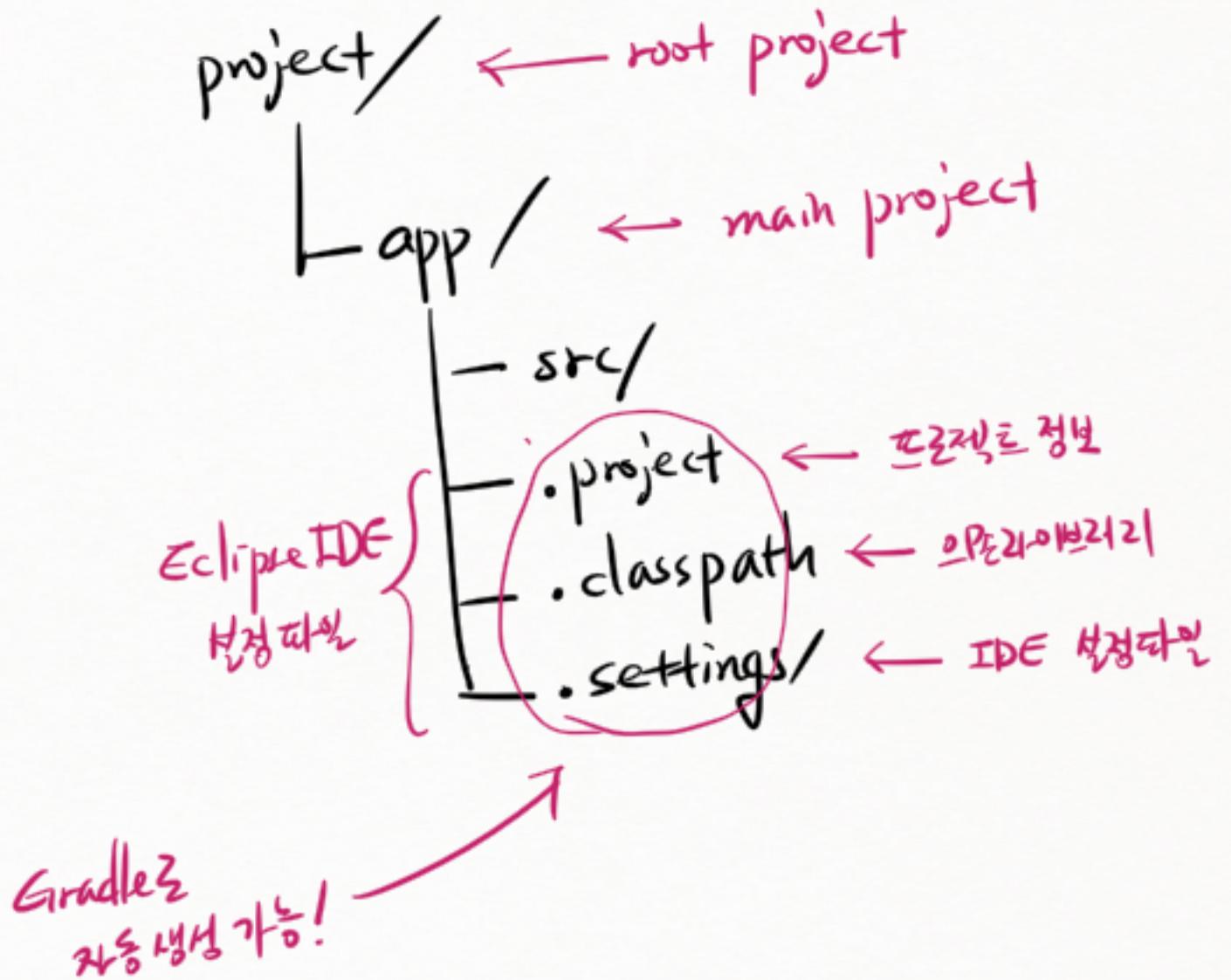
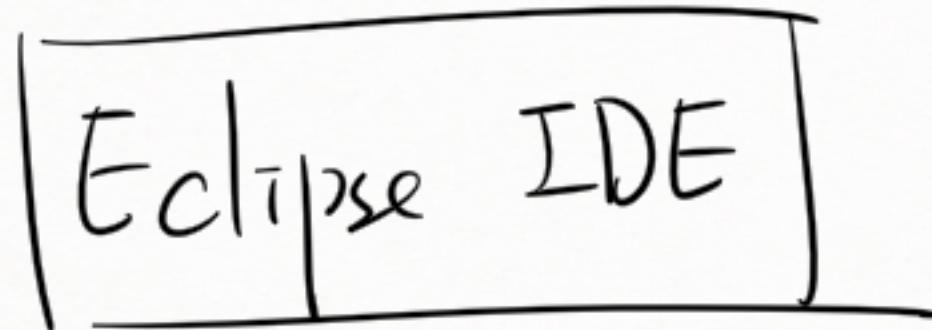
* Maven 워크스페이스의 좋은 프로젝트 디렉토리 구조



* Maven 워드넷의 풋을 프로젝트 디렉토리 구조



* Eclipse IDE 와 프로젝트

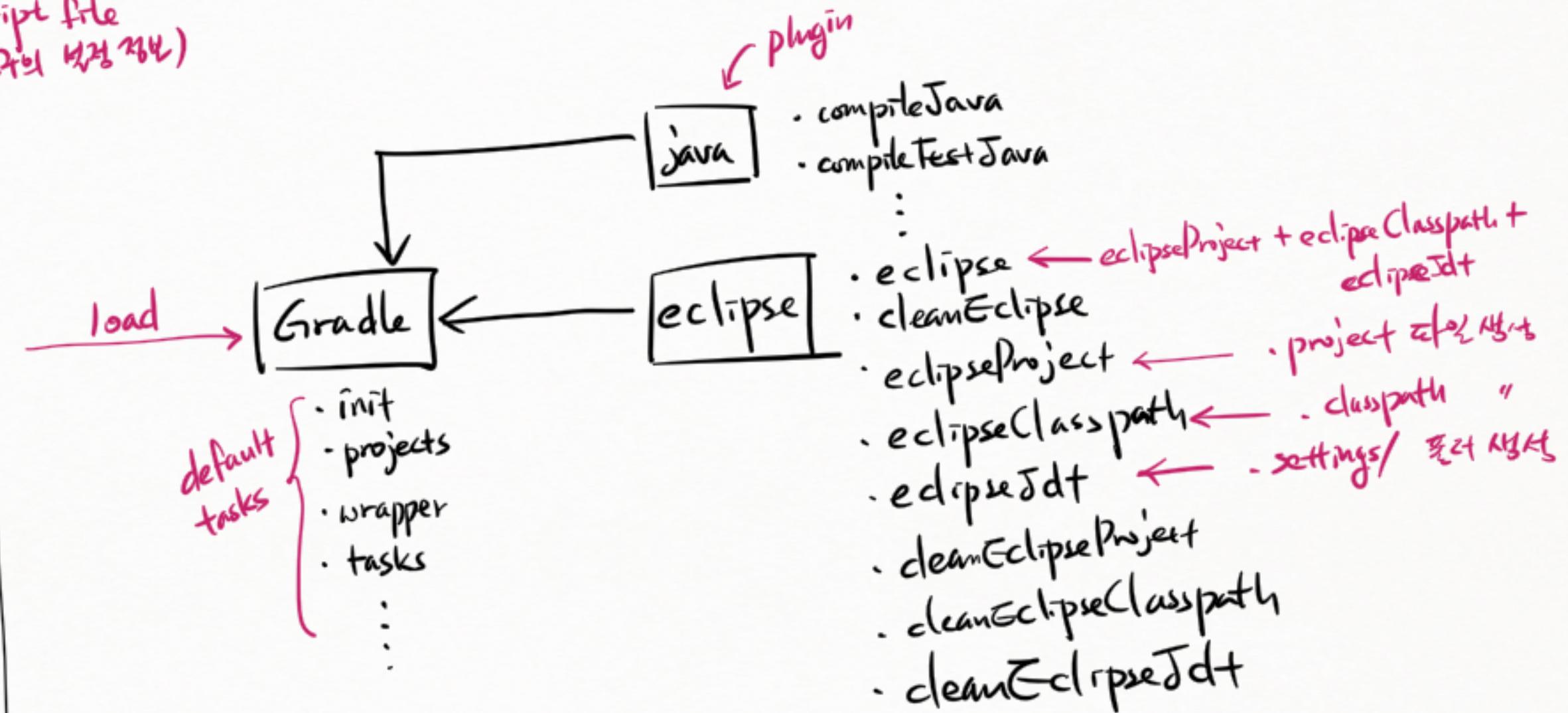


* Gradle 빌드 시

[build.gradle]

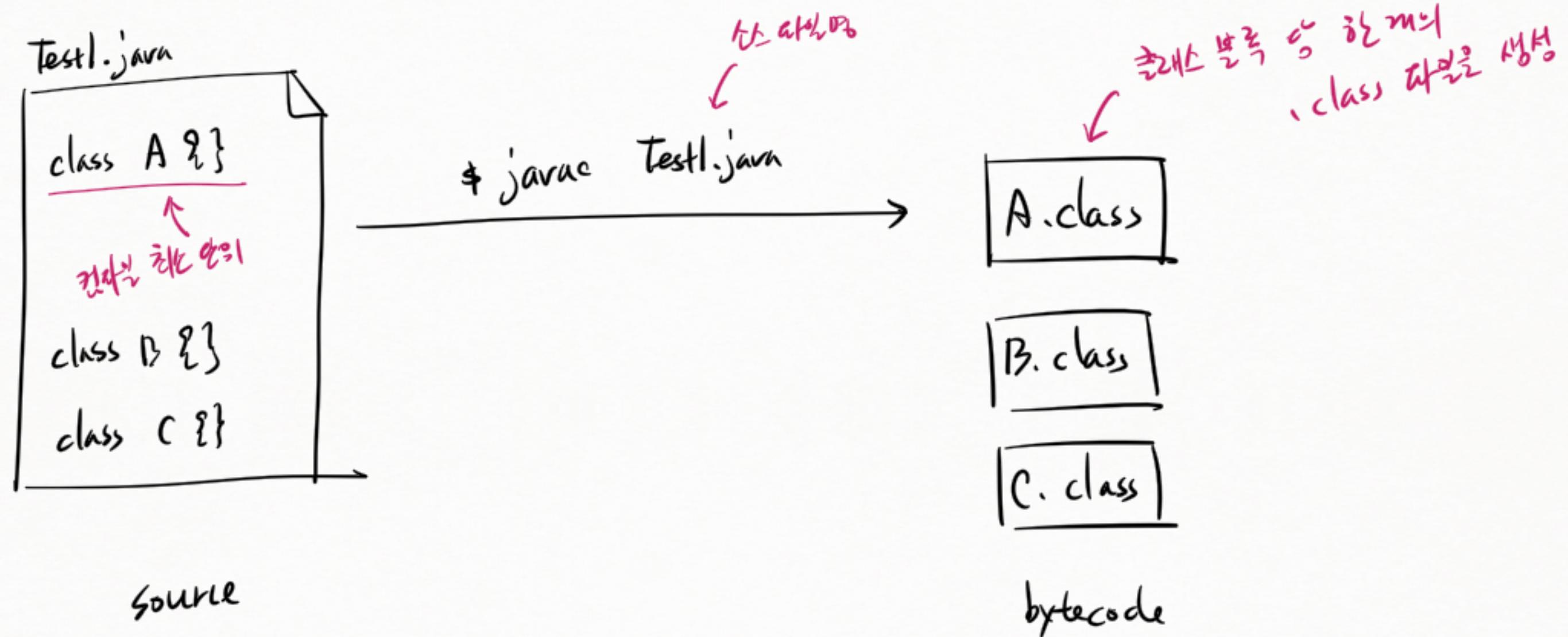
build script file
(Gradle 설정 파일)

```
plugins {  
    id '플러그인 ID'  
    :  
}
```

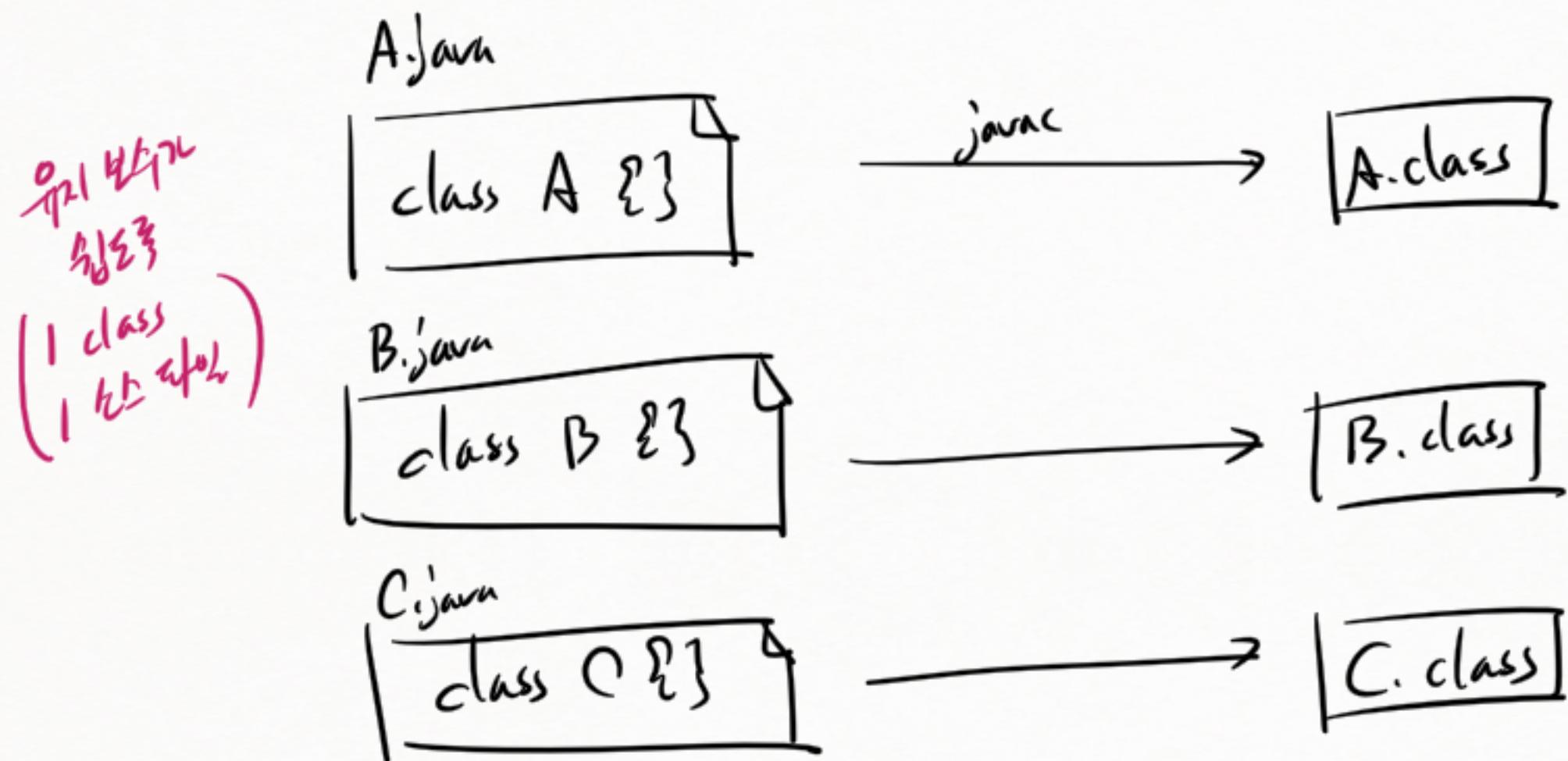


Java ဂျိတ်

* .java 와 .class , → 디자인



* class မှတ်လုပ်ခဲ့



* 한 번에 .class 파일 불러
↳ 바로 파일을 불러오는 듯 고리가 되도록 풀어

~~binary~~

src / A.java
B.java
C.java

myapp \$ javac -d bin src/A.java

↑
컴파일 결과물 위치

bin / A.class
B.class
C.class

* 소스 파일의 디렉토리

↳ 여러 명의 소스 파일을 관리하기 편리한 편이 좋다.

src / A.java
| p1 / B.java
| p2 / C.java

javac -d bin src/*.java ...

bin / A.class
B.class
C.class

→ 디렉토리는 .class 확장자 파일은
소스 파일의 경로와 동일하게
정해지게 된다.

* 키워드는 아이디어

↳ 코드 블록은 구조화된 실행 블록

src/A.java

```
class A {}
```

src/B.java

```
package p1;  
class B {}
```

src/C.java

```
package p2;  
class C {}
```



\$ javac -d bin src/*.java

bin/A.class

```
p1/B.class  
p2/C.class
```

* 터터 위는 아님

src/A.java

```
class A {}
```

src/p1/B.java

```
package p1;  
class B {}
```

src/p2/C.java

```
package p2;  
class C {}
```

터터 위는 찾기 쉽도록
터터 위에 package 이름
같은 헤더를 넣어.

java -d bin

src/*.java src/p1/*.java src/p2/*.java

bin/A.class

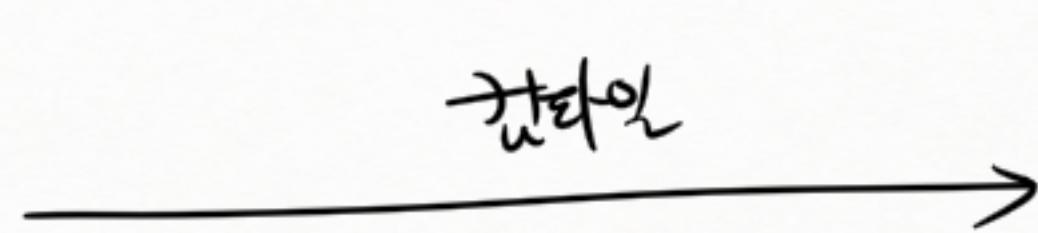
 + p1/B.class
 + p2/C.class

1

* 대상자와 대상화

```
package pl.px;  
class C {}
```

src/
|---pl/
|---px/
|---C.java



bin/
|---pl/
|---px/
|---C.class

* Maven 프로젝트의 풋한 c|24|호리 구조로 스스로 파일화

프로젝트

src/

└ main/

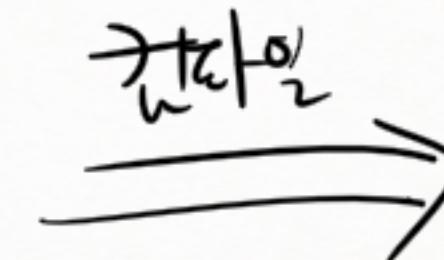
 └ java/

 └ pl/

 └ px/

 └ C.java

```
package pl.px;  
class C {}
```



bin/

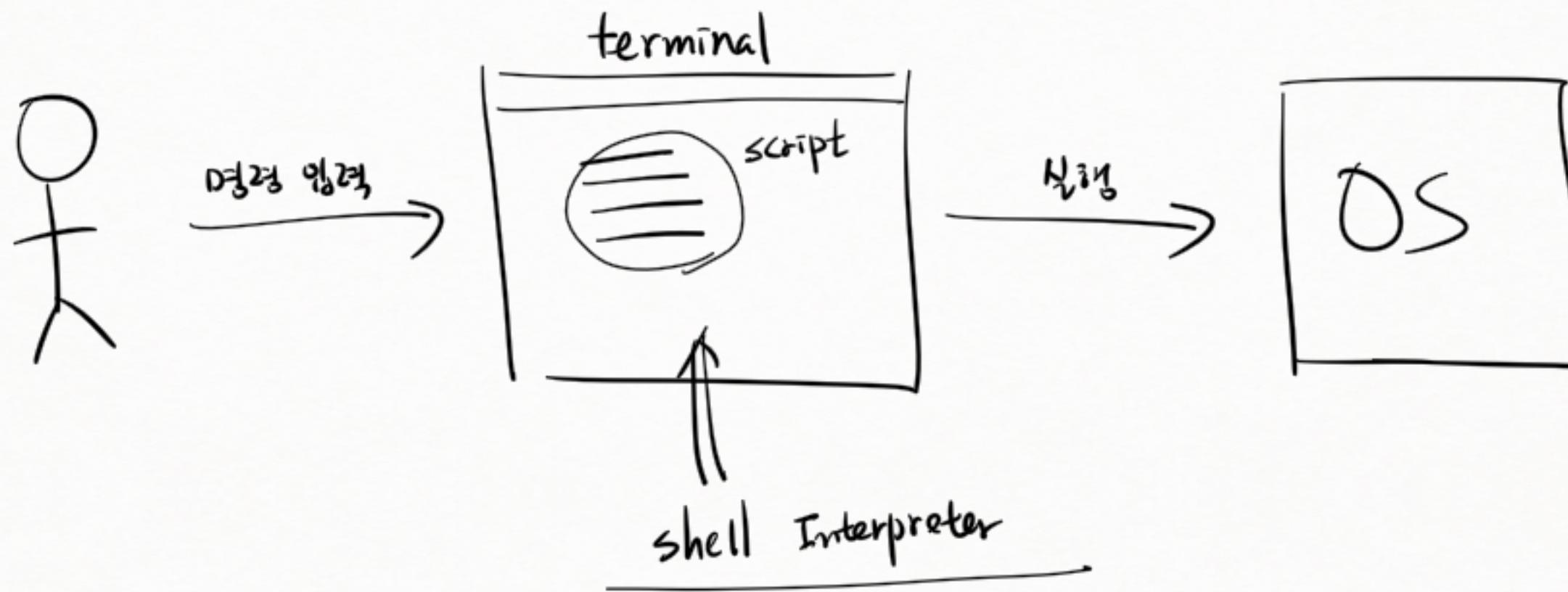
└ main/

 └ pl/

 └ px/

 └ C.class

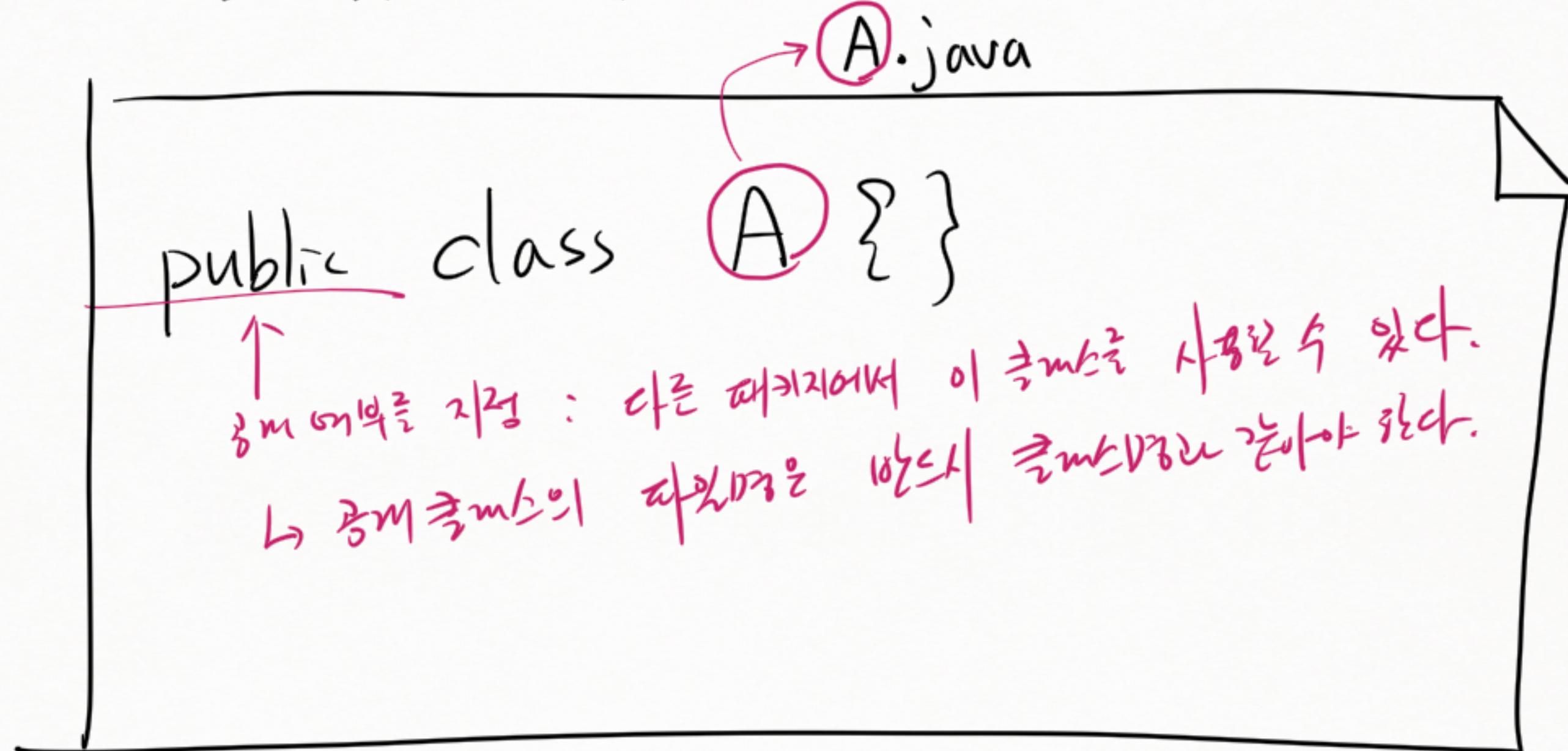
* Shell script



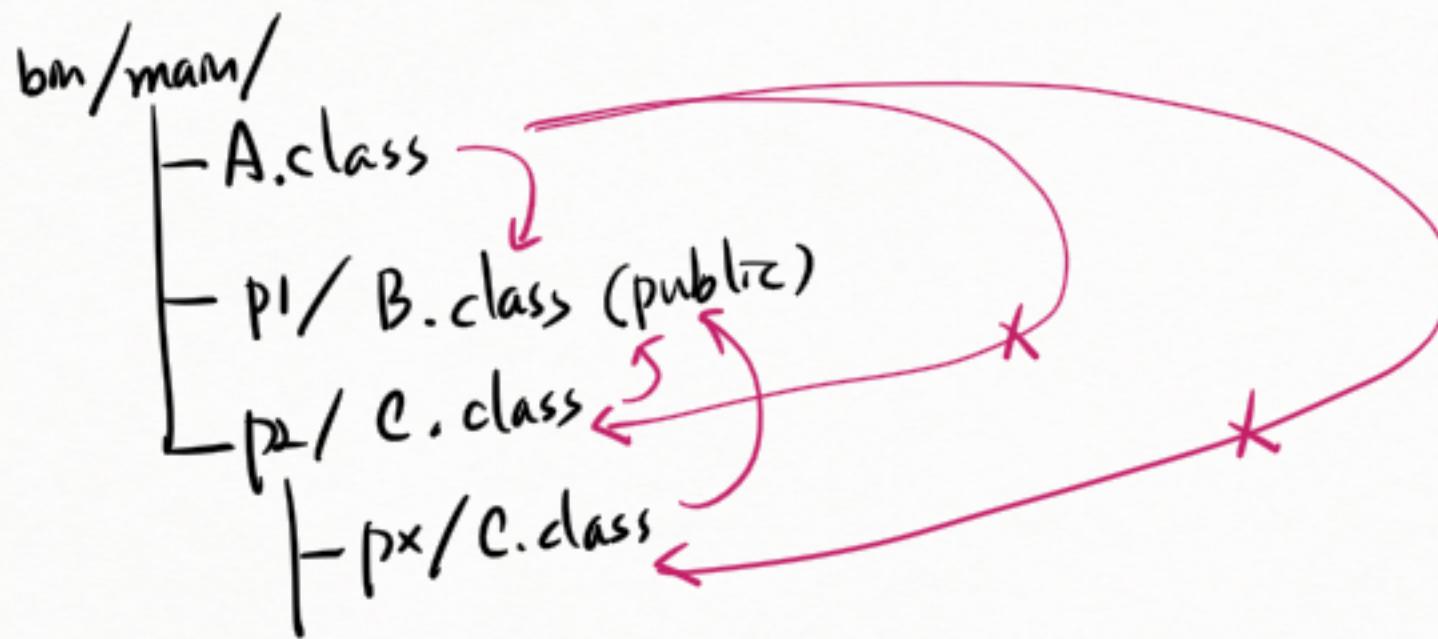
Windows : Batch Script , Powershell
.bat

Unix : zsh, bash, bsh, csh, ksh, sh
AppleScript , ...

* 클래스 복수의 경계 이후



* public vs non-public



public 키워드는
다는 데까지의 키워드이며
같은 가능.

* 다른 패키지 쿨스에 접근



★ 전파식

\$ javac -classpath b1/main A.java
 ||
 -cp

↑
패키지별 .class 파일들이 있는 폴더

파기지명 : 패키지에 속한 쿨스를 사용할 때는
 必不可少로 패키지를 지정해야 하는
 경우에만 가능하다.

* -classpath 사용법

java -classpath

bin/main

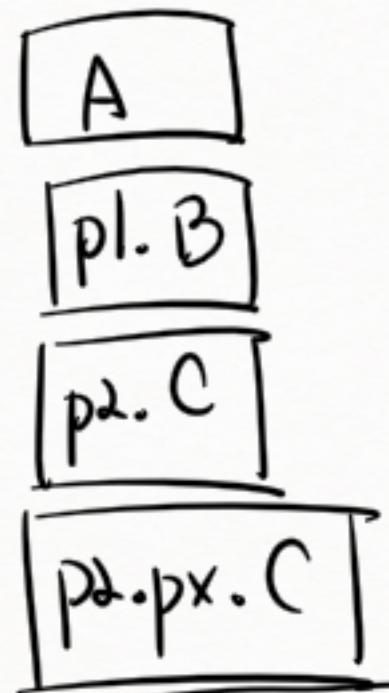


주로 대(대)자가 들어 있는 폴더이어야 한다.

~~bin/main/P~~

대(대)자 경로를 지정하지 않으면

* 다른 대상의 import는 import는



p2.px.aaaaa.bbbbbbb.ccccc.dddDD.D

A obj;
B obj;
C obj;
p2.px.C obj;
D obj;

A obj;
pl.B obj;
p2.C obj;
p2.px.C obj;
p2.px.aaaaa.bbbbbbb.ccccc.dddDD.D;

위에 있는 import는
import pl.B;
import p2.C;
~~import p2.px.C;~~ //이 줄은 제거할 것
import p2.px.aaaaa.bbbbbbb.ccccc.dddDD.D;

* -sourcepath

↳ 사용할 클래스의 소스 파일 경로
- classpath 를 경로를 지정해 주면 된다.

javac -d bin/main -sourcepath src/main/java src/main/java/A.java

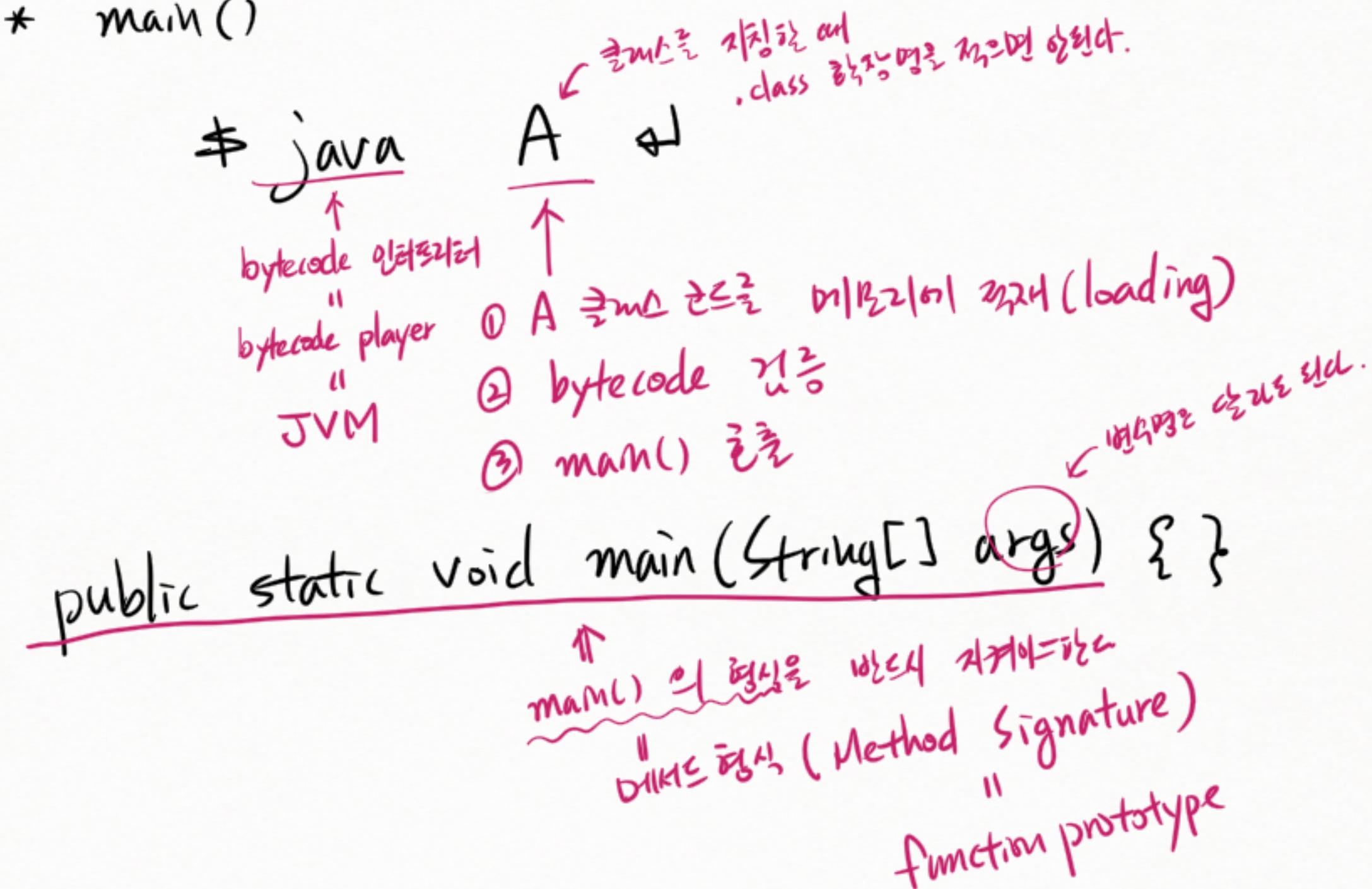
↑
A.java 를 컴파일 할 때,
A.java 가 사용하는
다른 클래스의 소스 파일 경로를 알려준다.

"
A.java 를 컴파일 할 때
A 를 실행할 때 필요한 소스
파일은 같이 컴파일 한다.

main()

Java App.의
entry point
(인입점)

* main()



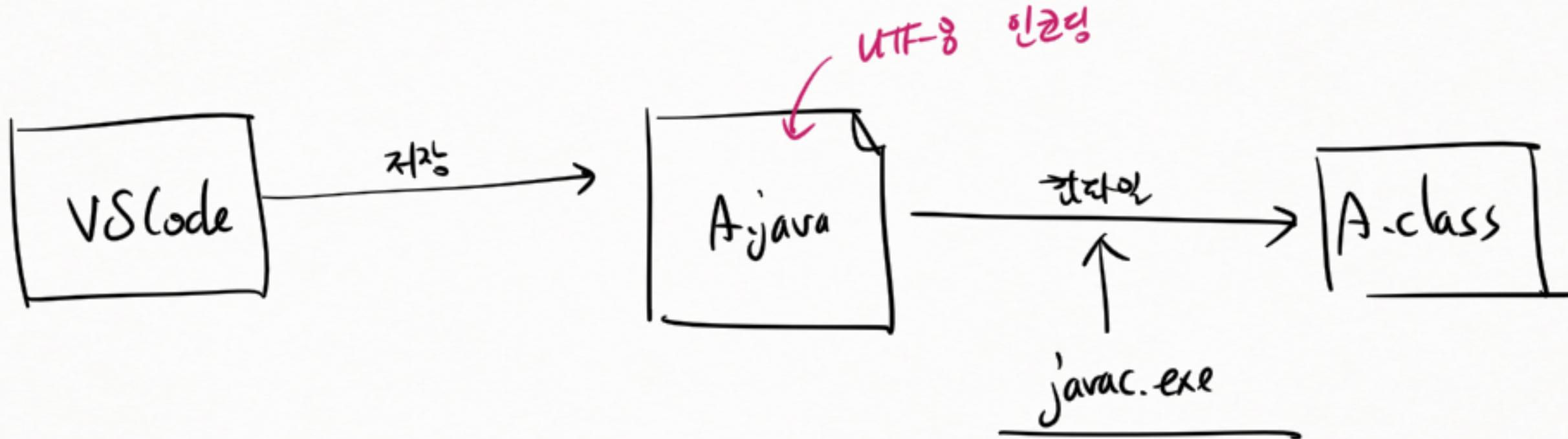
* -classpath

↳ 주로 디렉토리와 출마가 등장하는 대체로의 경로

↳ 자바 컴파일러와 JVM은 출마를 찾을 이 유연의 경로를 따라간다.

```
$ java -classpath bin/main A ↳  
          ""  
          -cp  
$ java -classpath bin/main pl.B ↳  
$ java -cp bin/main/B B ↳
```

* 소스파일 인코딩



Windows : MS949

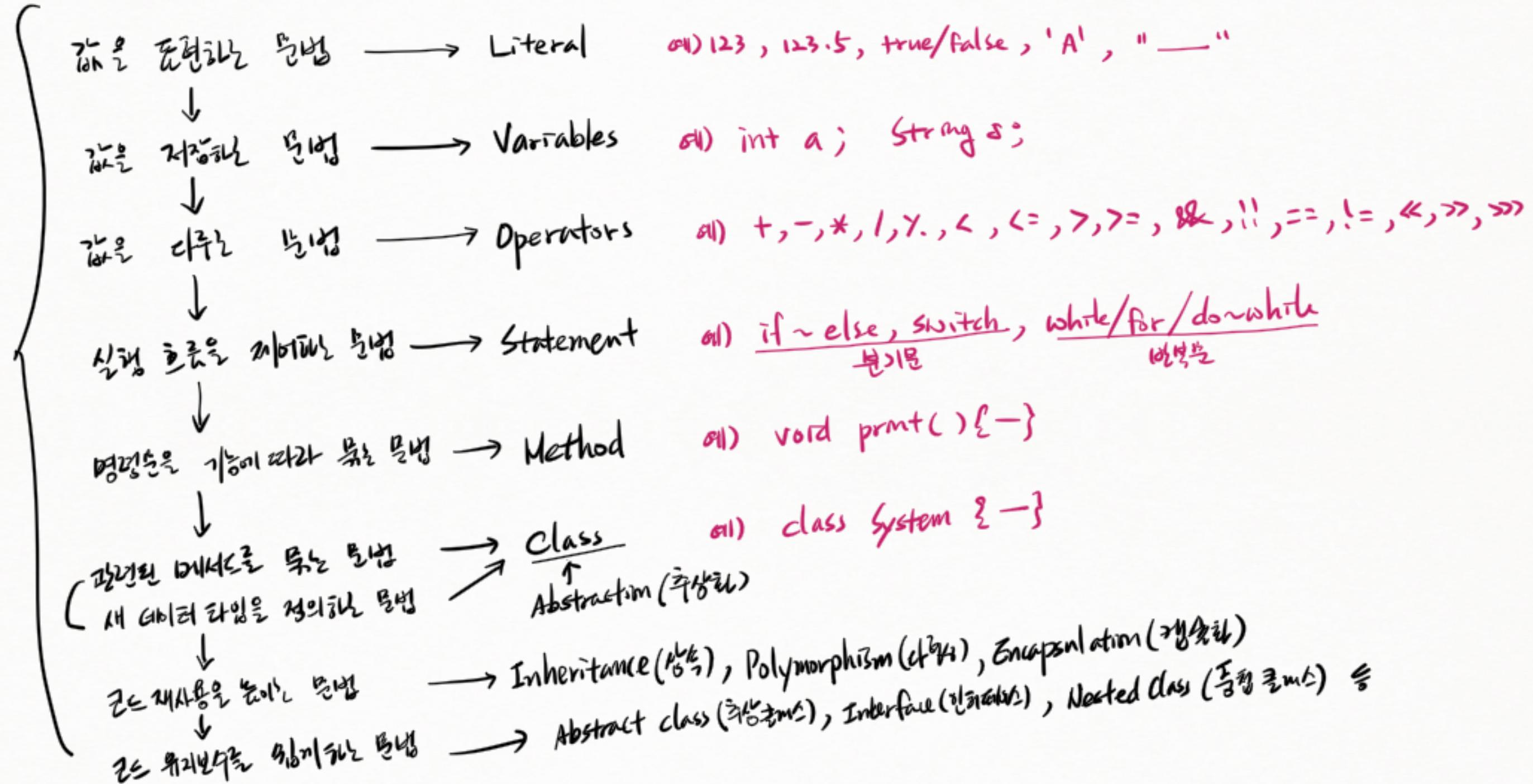
Unix
macOS }
Linux } UTF-8

· 컴파일러가 소스파일을 읽을 때,
소스파일의 인코딩 설정값이
OS의 기본 텍스트값과 같을 때
간주한다.

* -encoding

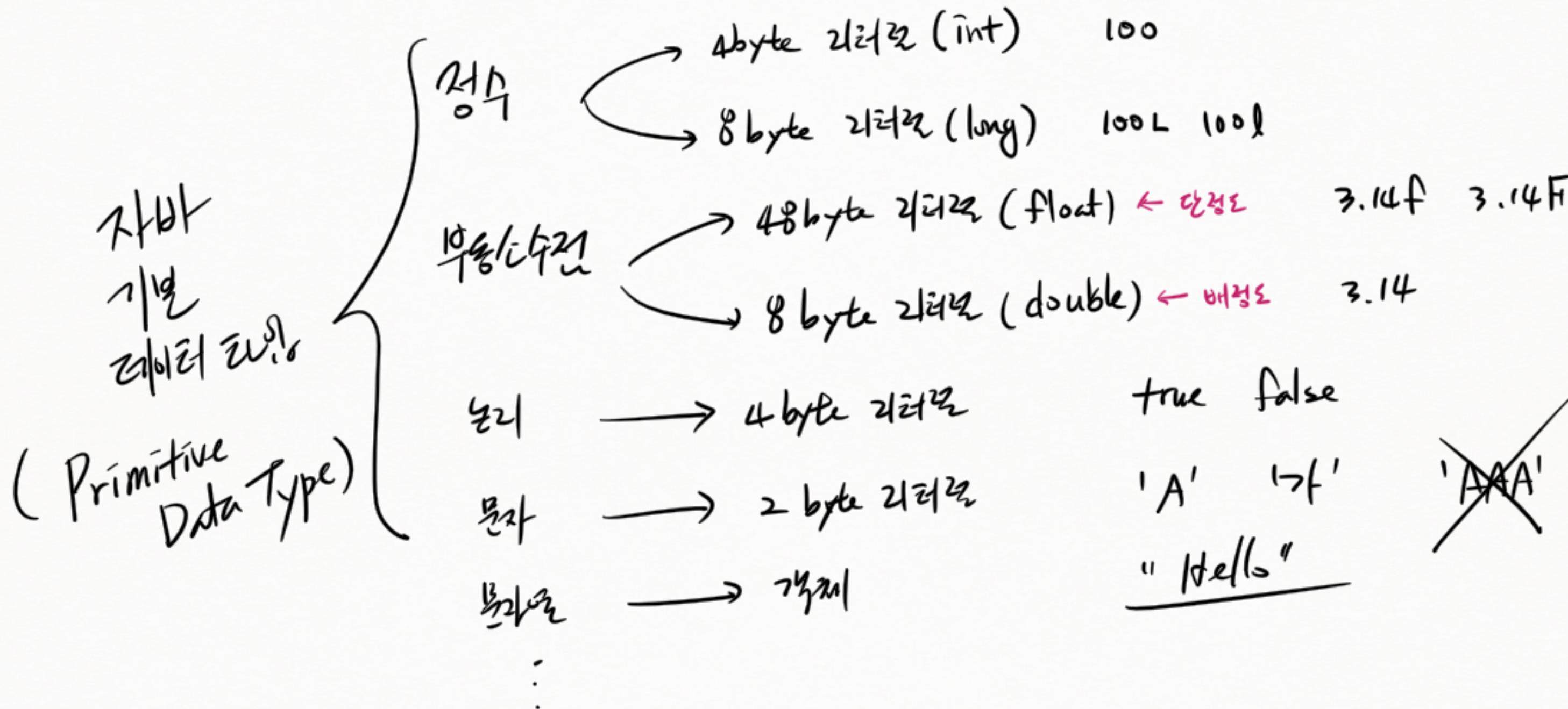
\$ javac -d bin/main -encoding UTF-8 src/*.java
↑
t는 파일의 인코딩을 알려준다.

DGP Language Hierarchy



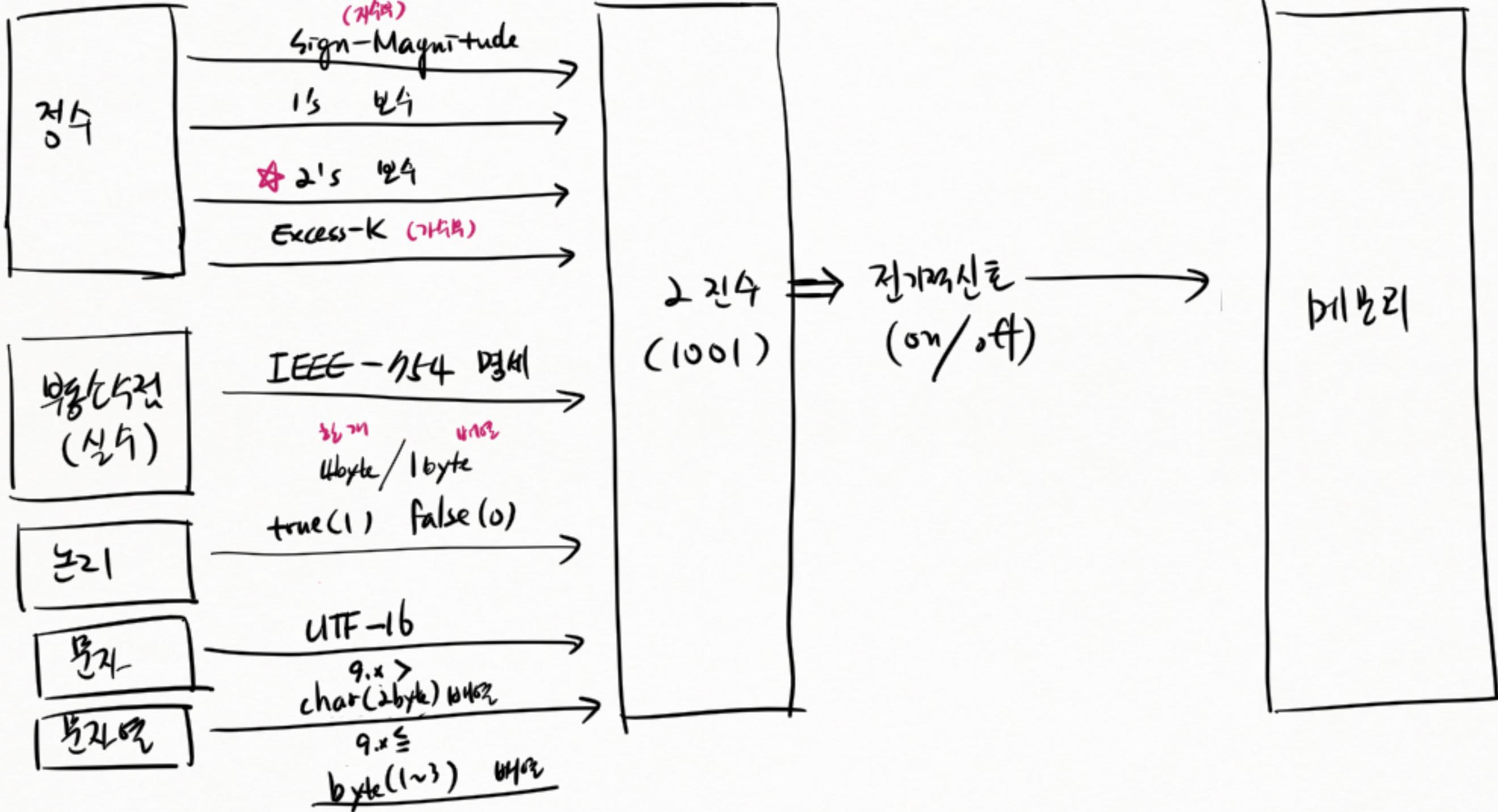
Literal

* 데이터 타입과 리터럴



* 2진수를 m/M2/I/I 2자리

encoding(변환)



* 정수 리터럴

int → LLByte

10진수: +100

8진수: -D¹⁴

2진수: 0B0101

0b0101

16진수: 0XAC00

0xAC00

8byte ← long
100L 100l

JavaScript et C++
정수 리터럴의 메모리 크기가 있다
(포함할 수 있는 범위가 있다)

100 (4byte 정수): 약 -2억 ~ +2억

100L } 8byte 정수: 약 -900억 ~ +900억
100l }

* 부동소수점 \rightarrow IEEE-754 명세

$$\begin{array}{r}
 + \underline{12.375} \\
 \hline
 1100 \quad \downarrow \\
 \times \frac{1.2}{0.750 \rightarrow 0} \\
 \hline
 \times \frac{1.2}{0.500 \rightarrow 1} \\
 \hline
 \times \frac{1.2}{1.0 \rightarrow 1} \\
 \hline
 \downarrow \\
 .011
 \end{array}$$

$$\begin{array}{r}
 12.375 \\
 \downarrow \\
 1100.011 \text{ (2진수)} \\
 = 1.100011 \times 2^3
 \end{array}$$

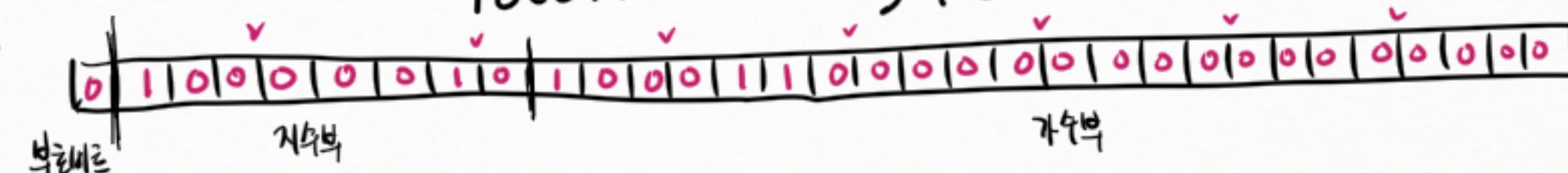
$$\boxed{2735.2705 \text{ (10진수)} \\ = 2.7352705 \times 10^3}$$

$$\begin{array}{r}
 \textcircled{1} \quad 6 \ 5 \ 4 \ 7 \ 2 \ 1 \ 0 \\
 14 \ 6 \ 3 \ 2 \ 1 \ 0 \ 4 \ 1 \ 1
 \end{array}$$

$$\begin{array}{r}
 \times \frac{1.00011}{\text{최종}} \\
 \oplus 100011 \\
 \text{(가수부)} \\
 \downarrow \text{Sign-magnitude} \\
 100011
 \end{array}$$

$$\begin{array}{r}
 \text{Excess-k} = 2^{(8-1)} - 1 = 128 - 1 = \textcircled{127} \\
 3 + \text{bias}_{\text{excess}} = 3 + 127 = \textcircled{130} = \underline{\underline{10000010}}
 \end{array}$$

$$\begin{array}{r}
 \textcircled{1} \quad 11000000 \\
 \textcircled{2} \quad 0x41460000 \quad \leftarrow 16\text{진수} \\
 \textcircled{3} \quad (\text{문자})
 \end{array}$$



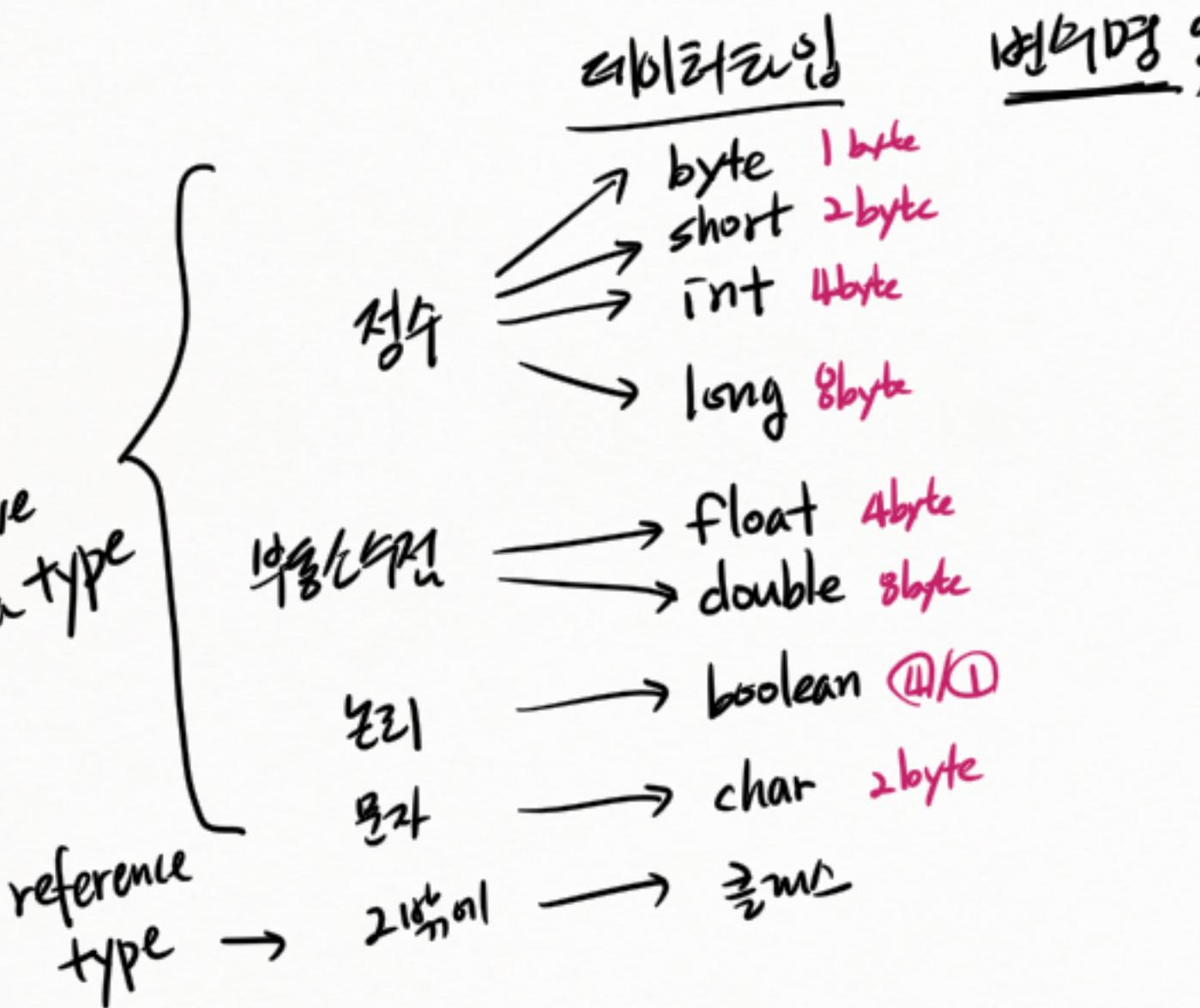
$$\begin{array}{r}
 \text{bias}_{\text{excess}} \\
 \text{Excess-k}
 \end{array}$$

Variables

* 변수 선언 (declaration)

↳ 같은 이름의 메모리를 ~~복수~~ 주어서 사용하는 방법

자바
원시
타입
primitive
data type

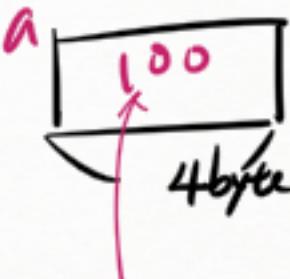


int a;

a = 100;

↑
assignment
operator
(할당연산자, 대입연산자)
대입연산자

storage location



* 정수 변수

byte $\approx 10^9$ m 3

↖ 4byte 짐이 처리해

$$\text{int } a = \underline{100}$$

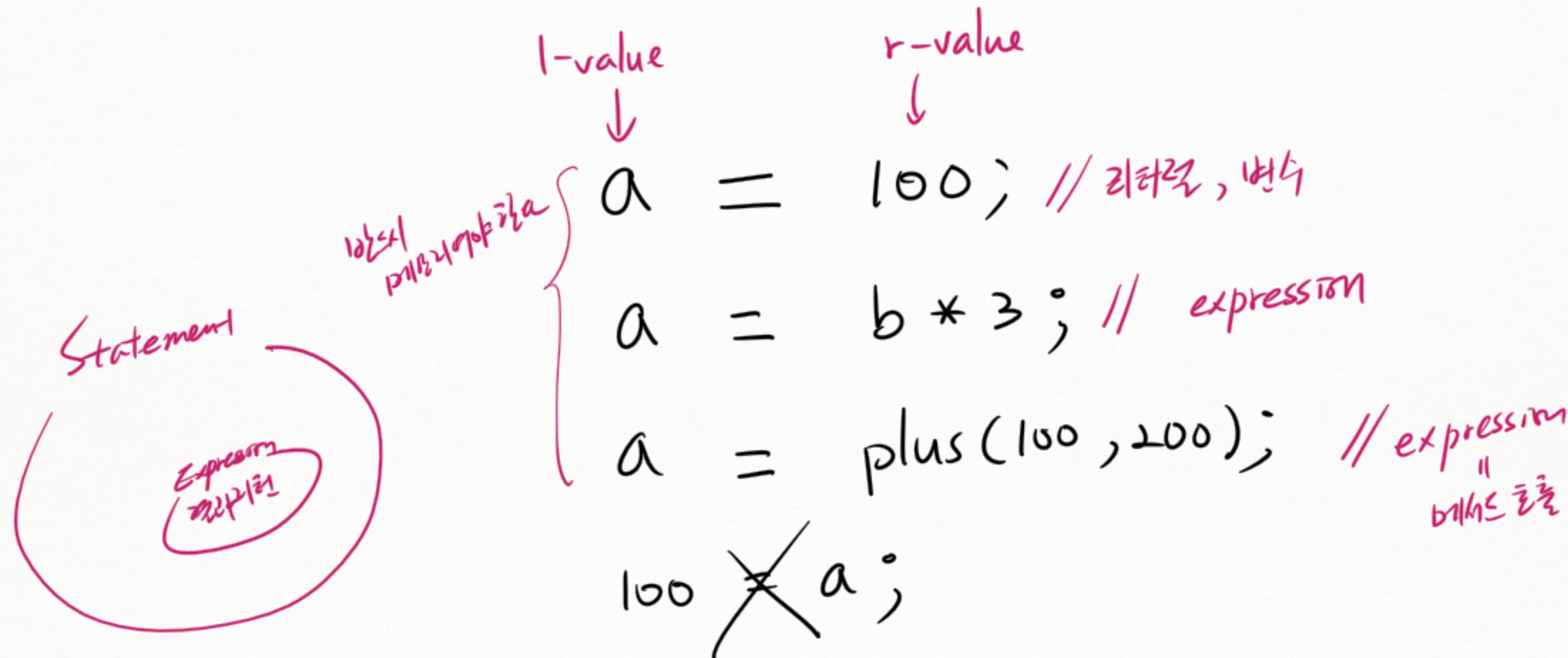
$$\text{long} \quad \frac{b}{\text{Byte}} = \frac{100}{4 \text{ byte}}$$

$$\text{short } \underline{\text{c}} = \frac{100}{4 \text{ byte}}$$

$$\text{byte} \frac{d}{1\text{byte}} = \frac{100}{4\text{byte}}$$

예외경우: 리터럴 값이 미리 나온 뒤에 그대로
만들 수 있으면 허용한다!

* l-value & r-value

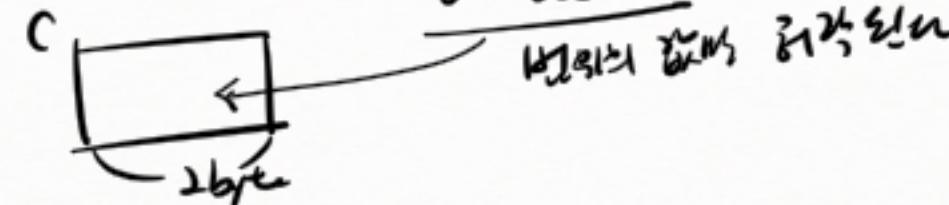


* char 변수

char c ; UTF-16

↑ 문자코드 (UCS2) 은 최장 2byte 대비

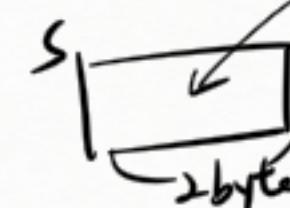
0 ~ 65535



↑ 문자코드는 1byte 정수값
c = 65 ;

System.out.println(c);
 ↓
 A

short s ; -32768 ~ +32767



s = 65 ;

System.out.println(s);
 ↓
 65

```
char c = 65;
```

```
System.out.println(c);
```

① 현재 프로그램에서 사용하는 폰트 파일을 찾는다
↳ D2Coding

② D2Coding 폰트 파일에서 65에 해당하는
문자를 찾는다

↳ A

③ 폰트 파일에서 읽은 A 문자 그림을 출력한다.

* 1 1 연습문제

'A'
↓ return
65 ← A 문자에 대응하는 $\begin{cases} \text{UCS2} \text{ 코드} \\ \text{UTF-16} \\ \text{Unicode2} \end{cases}$

char c = 'A';
 ↑
 자장
 ↓
 65

c2 = '한국'-1;

char c2 = 0xB625;
 ↑
 한국
 ↓
 c2 = '한국'
 ↓
 0xB625 = 4562P

W.H. O'Leary

* 배열 초기화

↳ 같은 종류의 배열을 여러가지 방법으로 초기화하는 법

↑
자료형 변수 = reference 변수

→ 데이터타입[] 변수명; ← 자료형

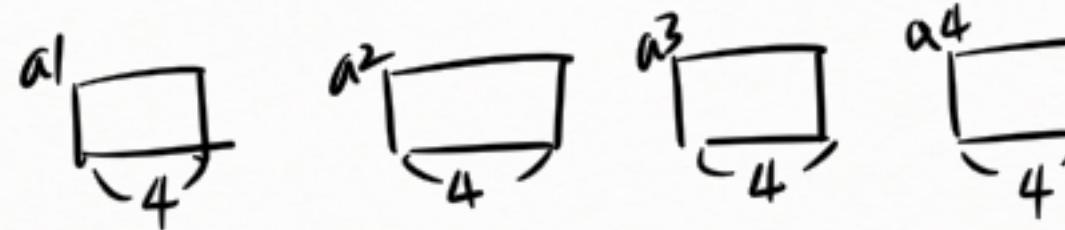
데이터타입 변수명[]; ← c style

변수명 = new 데이터타입[수량];

데이터타입[] 변수명 = new 데이터타입[수량];

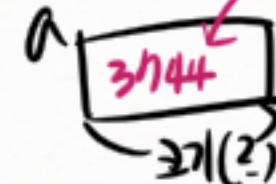
* 배열 접근

int a1, a2, a3, a4;



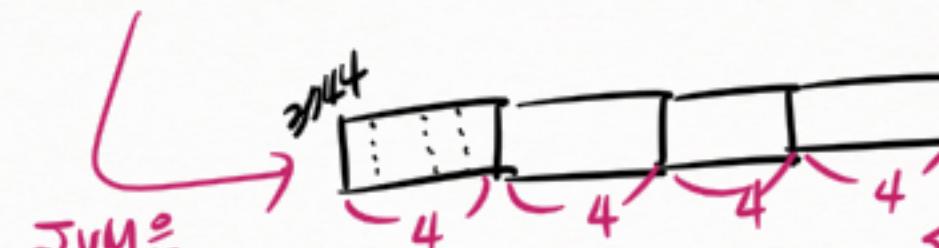
변수들이 개별적인 메모리가 연속된 메모리가 아님!

int[] a;

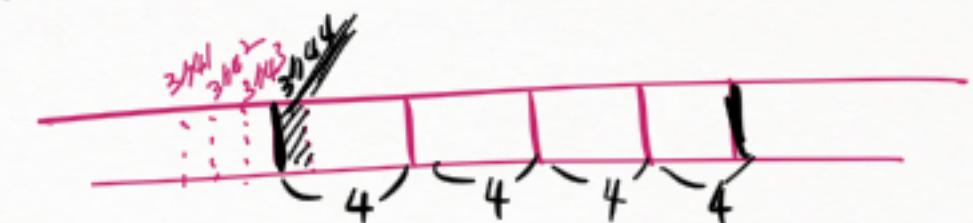


a는 메모리의 주소를 갖을 때
주소는 초기

a = new int[4];



JVM은
Heap 영역에
연속된 int 타입의 메모리를 준비한다
리턴값은 준비한 메모리의 시작 주소이다.

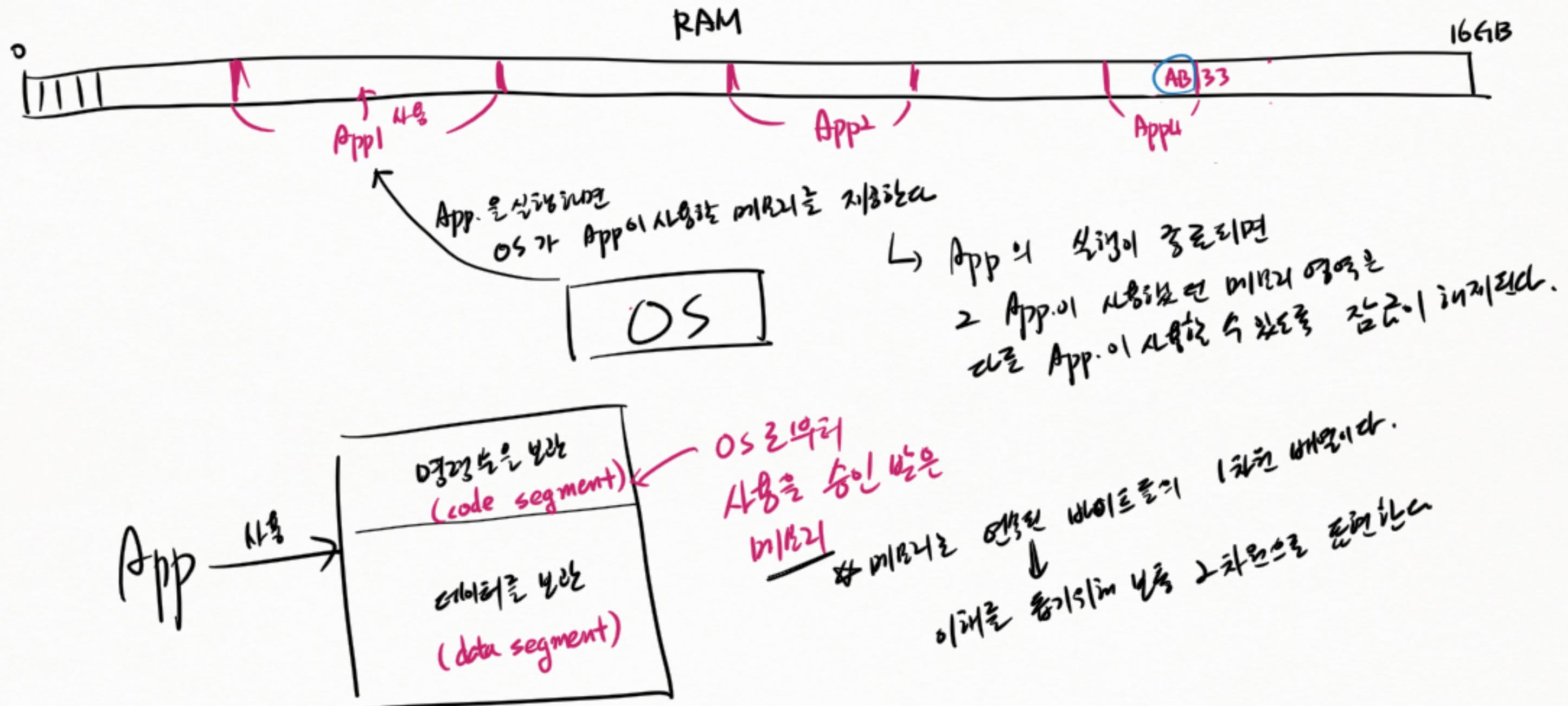


연속된 메모리라

new 명령을 통해 준비한 메모리
"인스턴스 (instance)"

(a) int 배열의 인스턴스

* RAM 은 미리 채워



* JVM 과 OS

