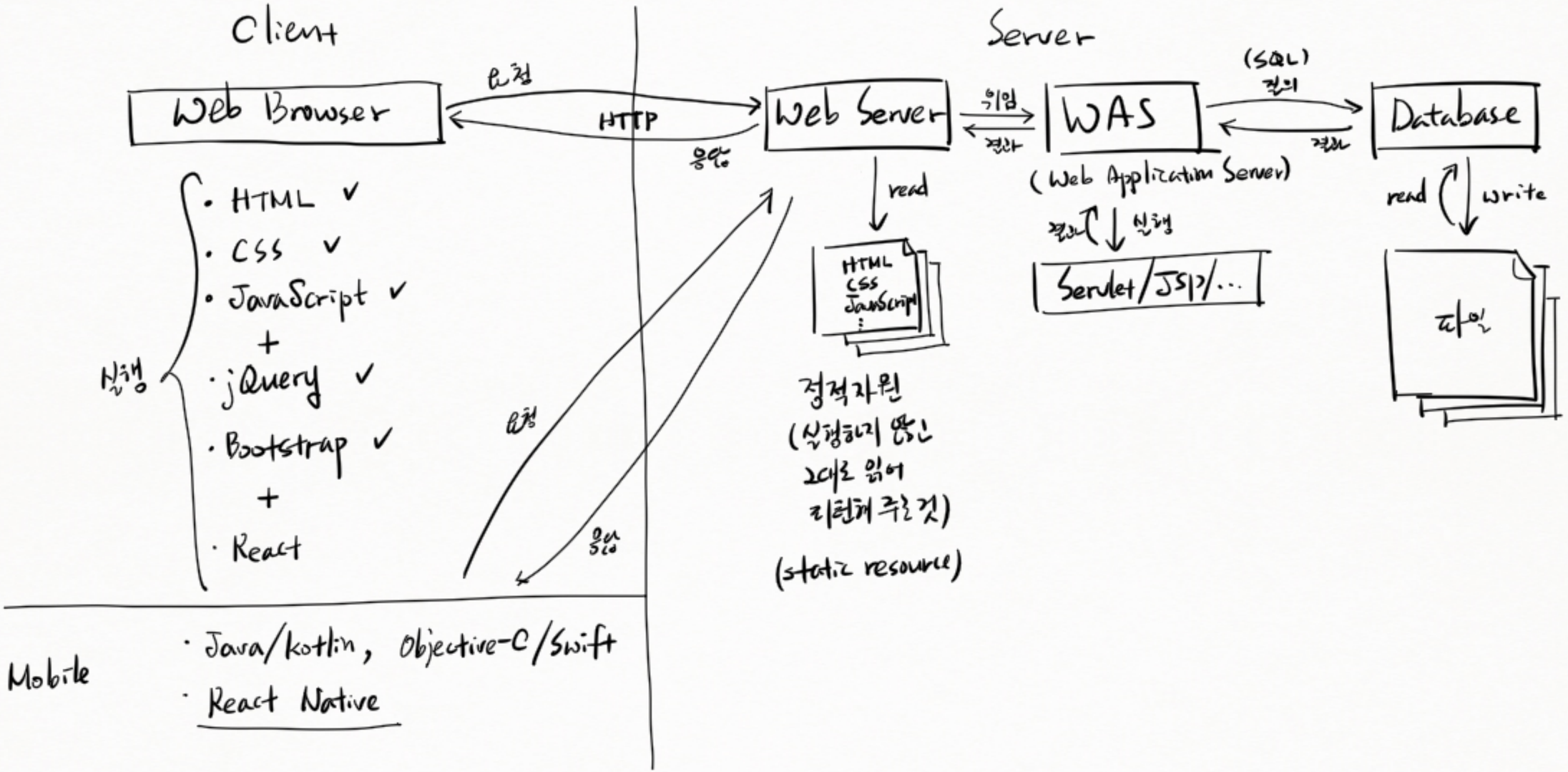
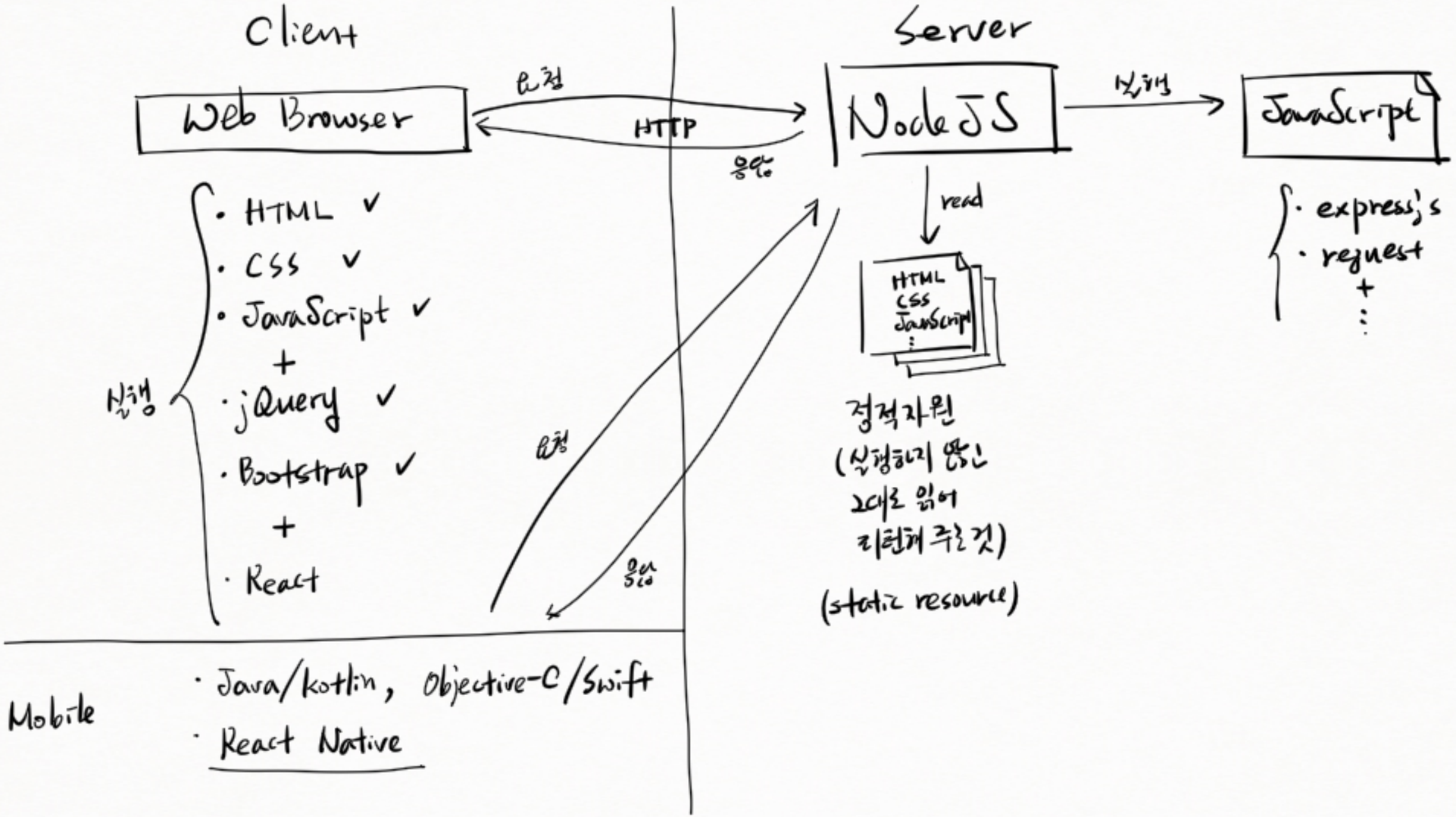
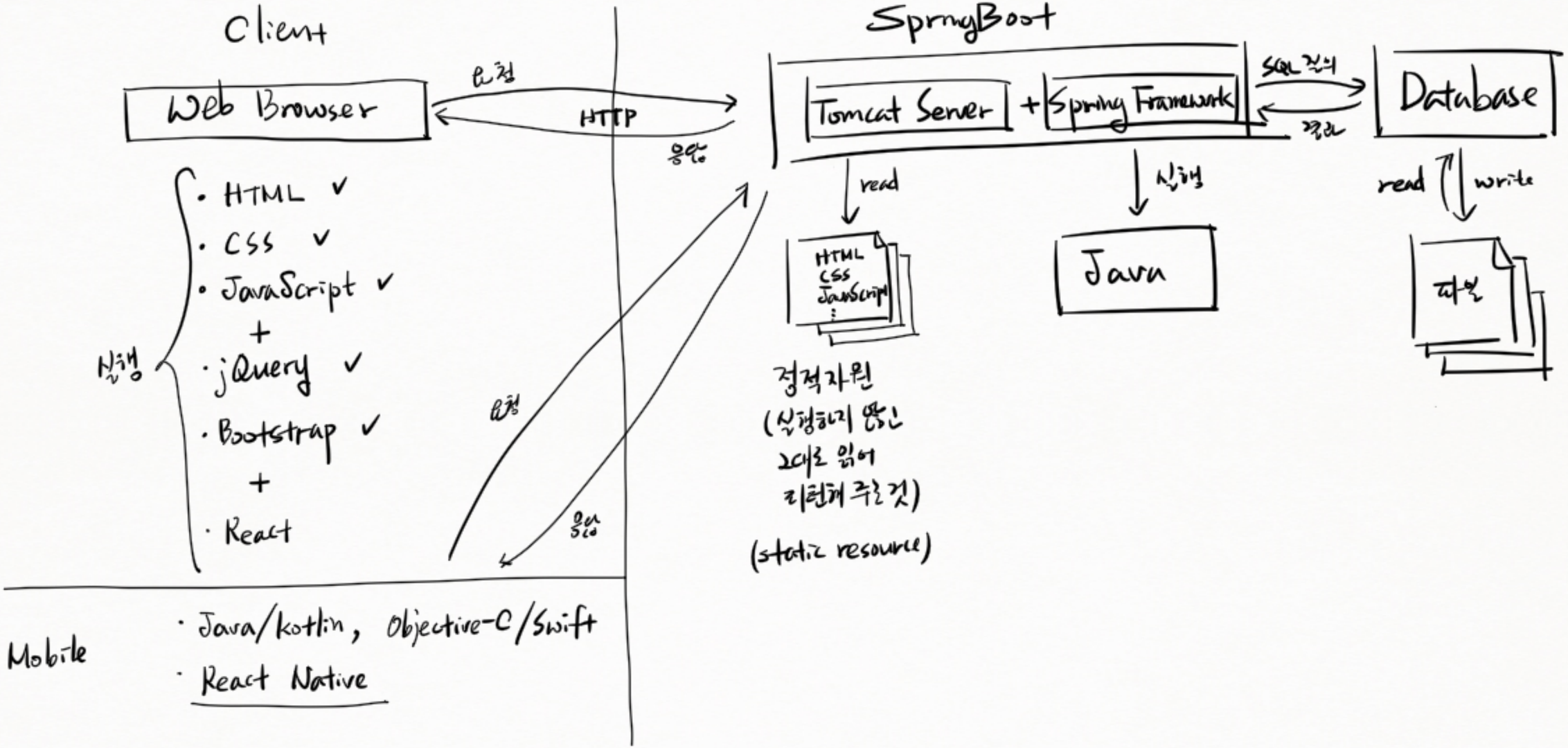
* Web Application Architecture by Java



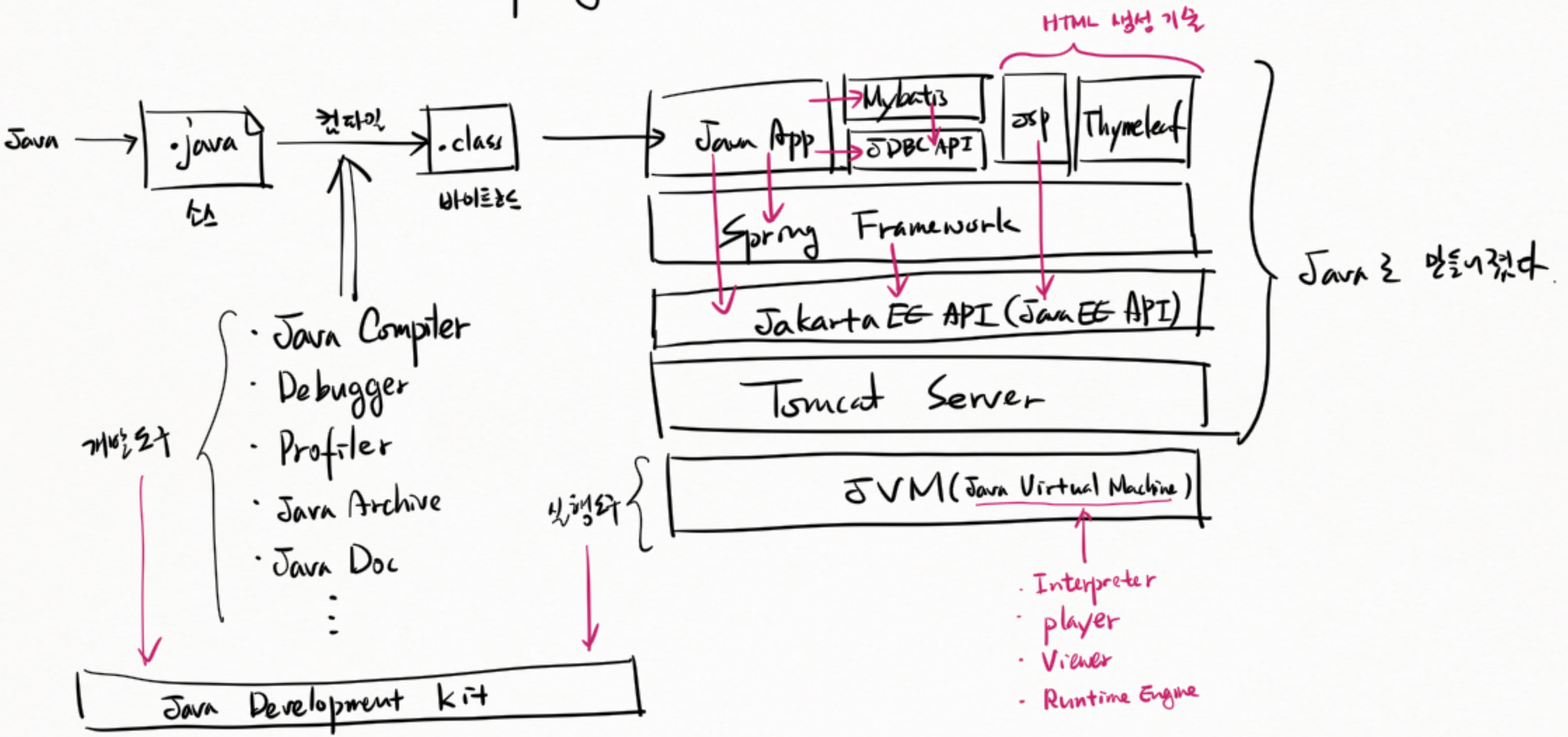
* Node.js Web Application Architecture



* SpringBoot Web Application Architecture



* SpringBoot 기술 스택



* Java 프로그램 준비

① JDK 다운로드 및 설치 → ② 환경 변수 설정 → ③ 빌드 도구 설치 및 설정

↓ java.oracle.com

↓ JAVA_HOME → JDK 설치 폴더
PATH → JDK/bin 폴더 경로 추가

$ setx JAVA_HOME JDK설치폴더 /m
$ setx PATH "JDK홈/bin;%PATH%"
} Windows

$ /usr/libexec/java_home -V

③ 빌드 도구 설치 및 설정
↳ · 프로젝트 폴더를 준비
· JDK를 이용하여 컴파일, 실행, 테스트 수행
· 아카이브(archive) 파일을 생성
배포 파일 (.jar, .war, .ear, .zip 등)

↓ gradle.org

↓

(추천)
④ Eclipse IDE 설치 및 설정
Integrated Development Environment

(상용) IntelliJ

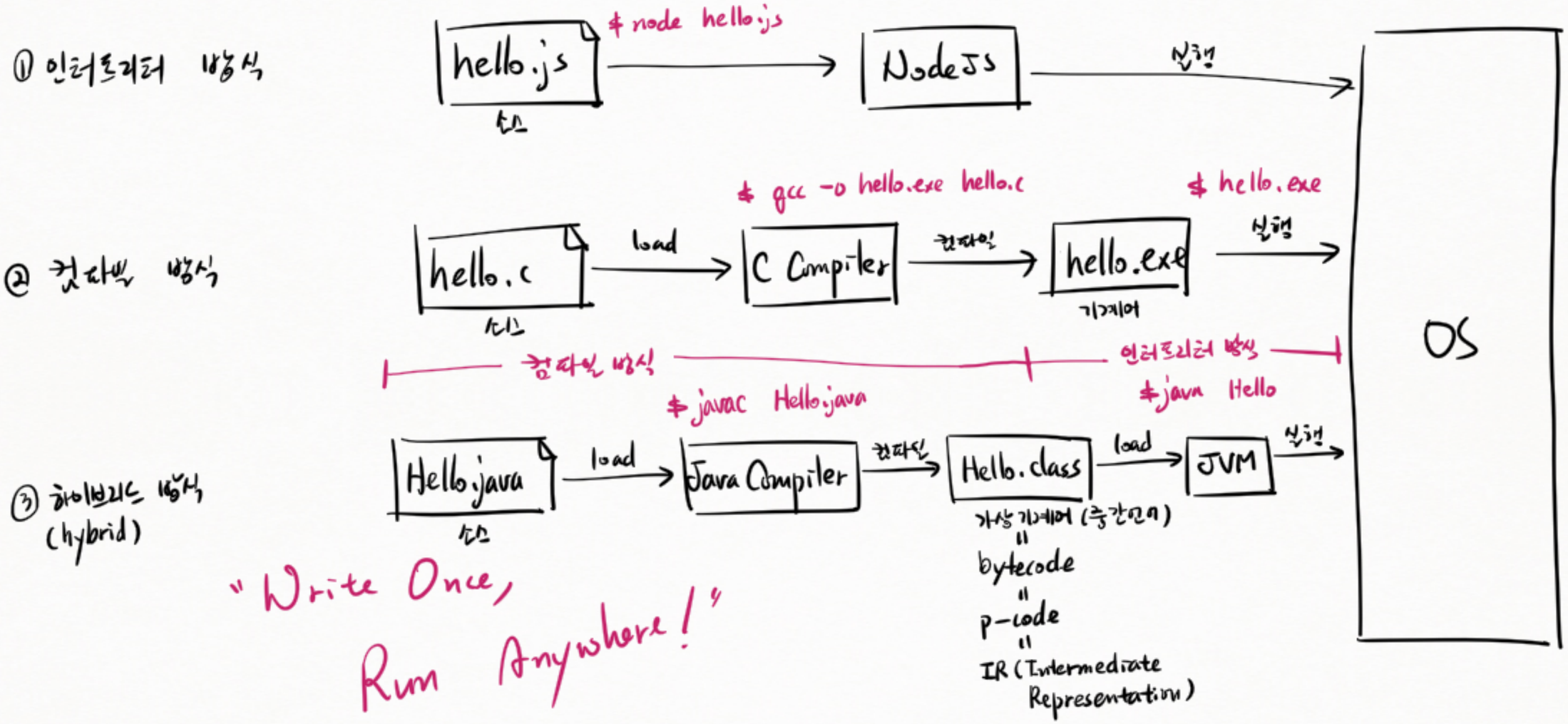
↳ · 소스 편집
· JDK를 이용한 빌드 작업, 테스팅 등
⋮
개발 전반의 작업을 수행할 수 있도록 도와주는 프로그램

eclipse.org

* JAVA_HOME
↳ JDK 가 설치된 폴더 경로
↳ Java App. 이 JDK를 찾을 때 이 환경변수를 이용한다.

* PATH
↳ 실행 프로그램이 있는 폴더 경로
↳ CLI 에서 프로그램 이름으로 바로 실행하고 싶을 때.

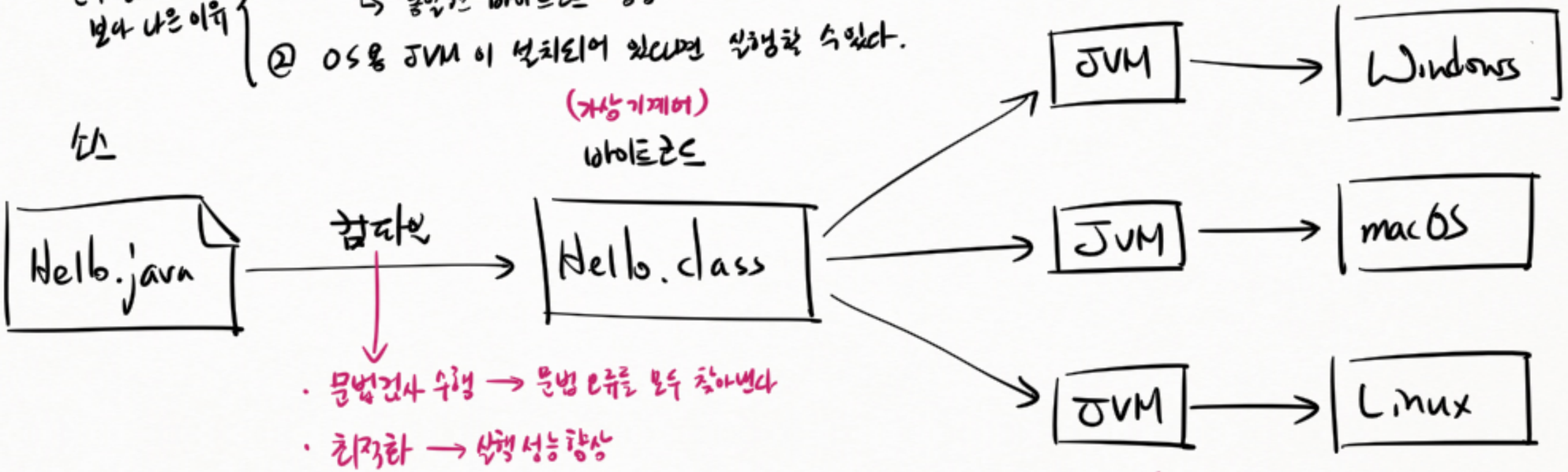
* App. 개발 및 실행



"Write Once, Run Anywhere!"

* Hybrid 방식을 도입한 이유

순수 컴파일 방식보다 나은 이유
① OS 마다 따로 컴파일 할 필요가 없다.
↳ 동일한 바이트코드 생성
② OS용 JVM 이 설치되어 있다면 실행할 수 있다.



· 문법검사 수행 → 문법 오류를 모두 찾아낸다
· 최적화 → 실행 성능 향상

바이트코드 플레이어
〃 인터프리터
〃 런타임 엔진
〃 바추얼 머신

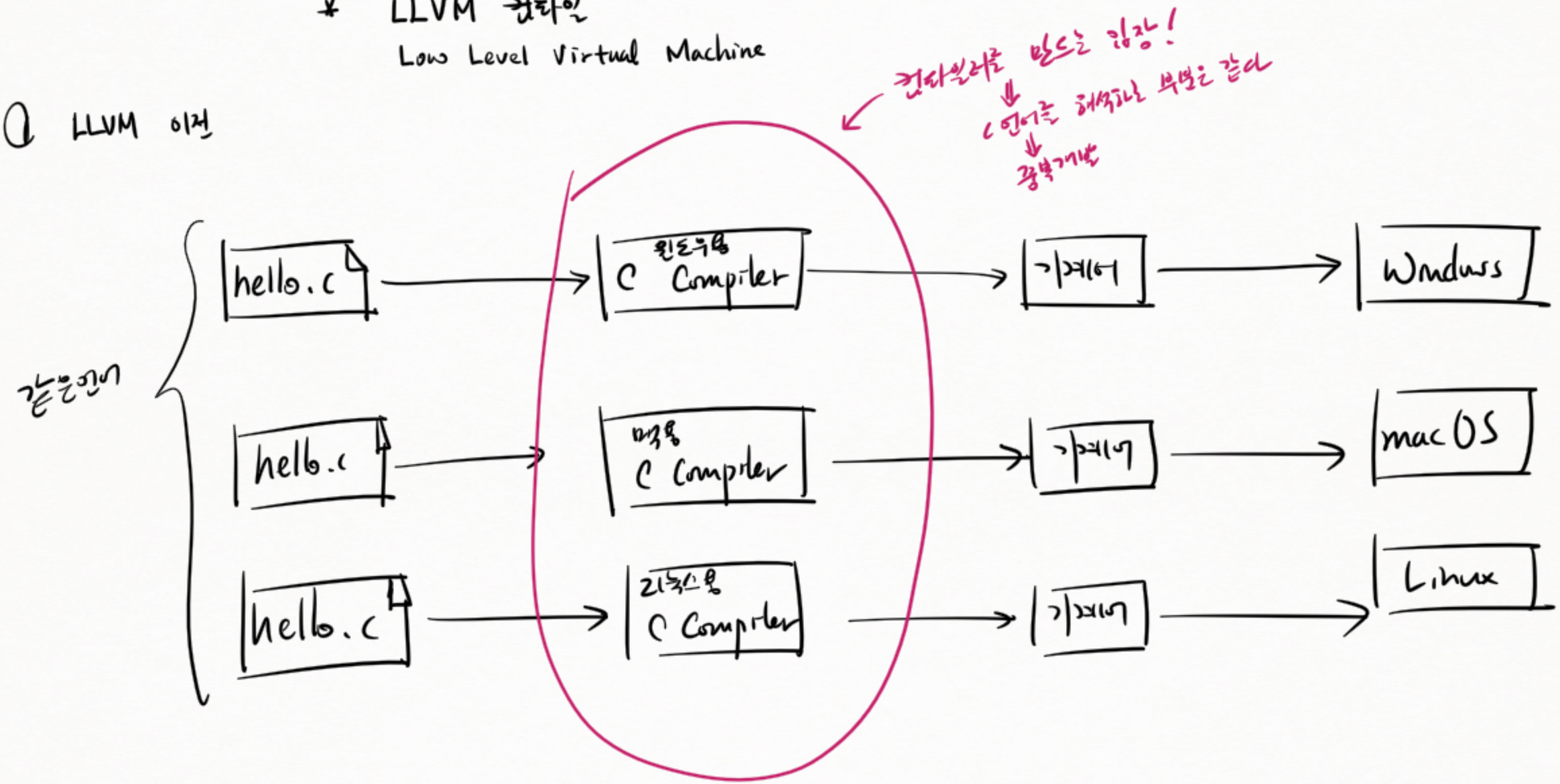
순수 인터프리터 방식보다 나은 이유
③ 컴파일 과정에 문법의 오류를 모두 찾아낸다
④ 완전 기계어가 아니라 기계어에 가까운 언어로 변환된 명령을 실행하기 때문에 소스에 작성된 명령을 실행하는 것보다 실행속도가 빠르다.

실행할 때마다 문법 오류 검사를 하는 순수 인터프리터 방식보다 실행속도가 빠르다.

* LLVM 컴파일

Low Level Virtual Machine

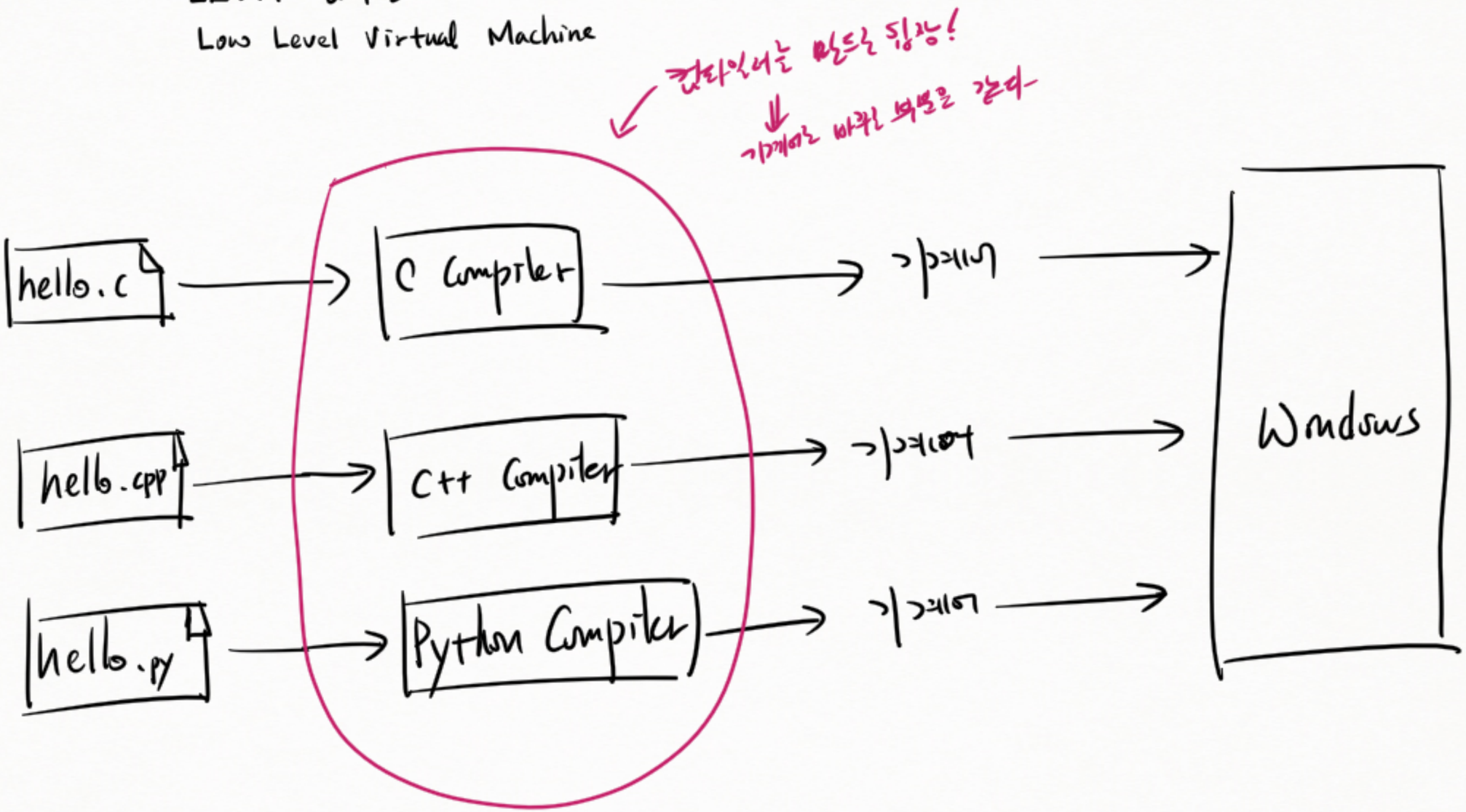
Q LLVM 이전



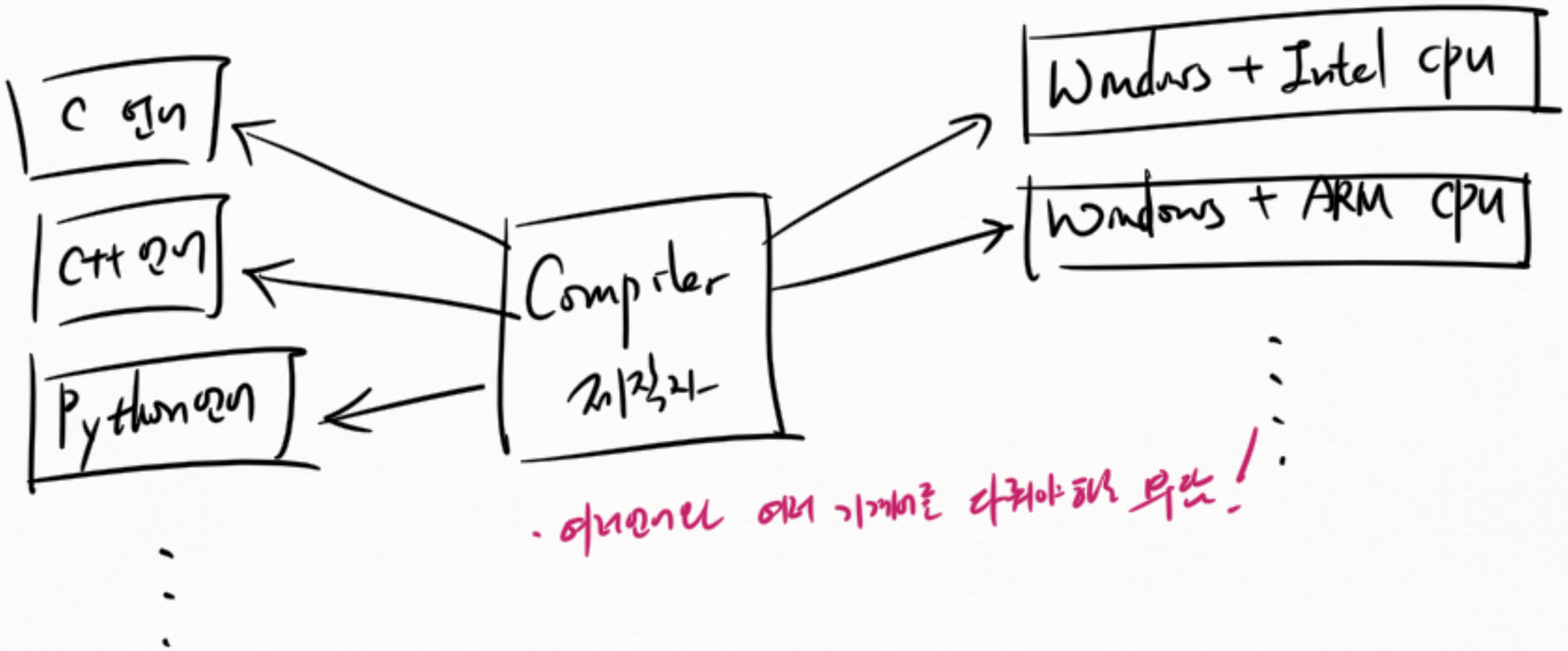
* LLVM 컴파일

Low Level Virtual Machine

① LLVM 이전



* 기존 방식의 문제점

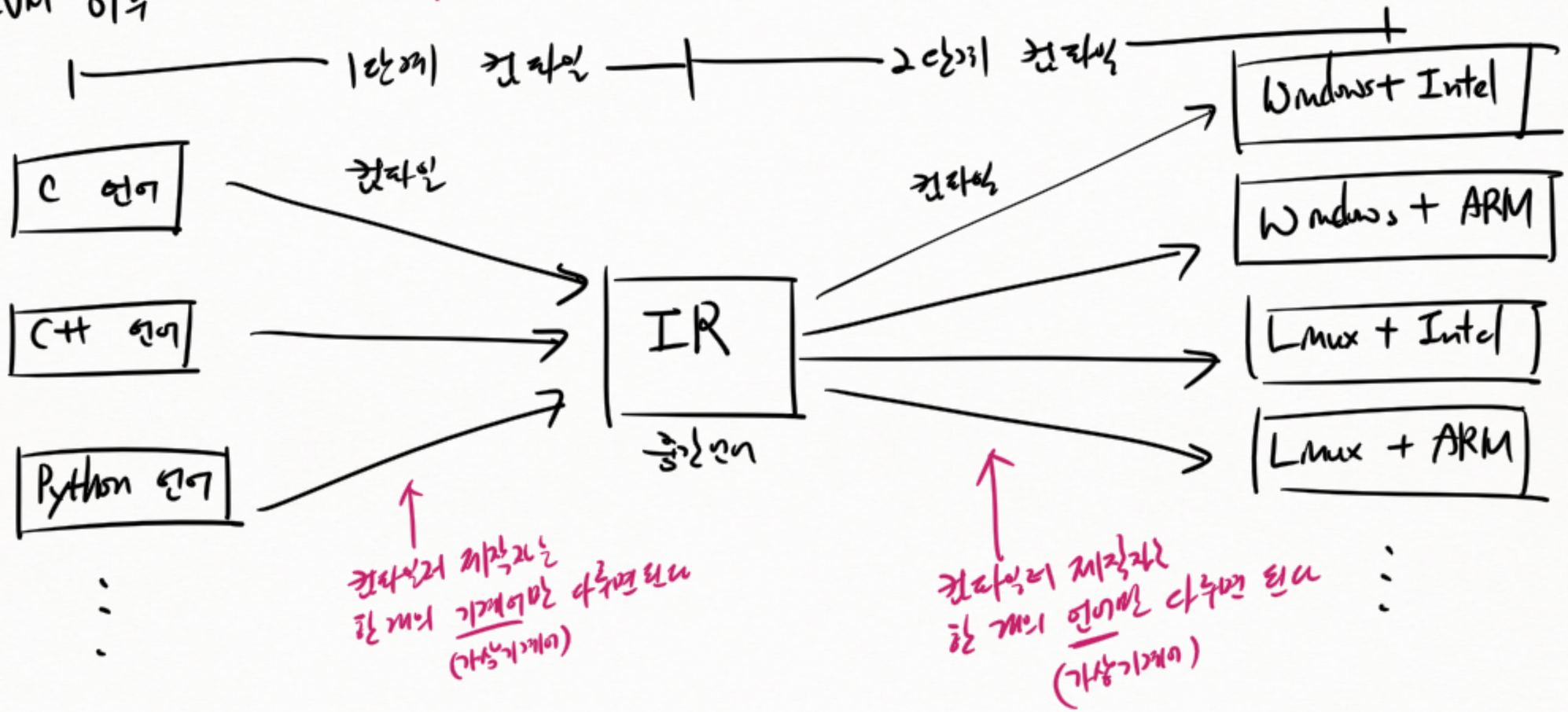


· 어떤언어냐 어떤 기계를 다뤄야 하는 문제!

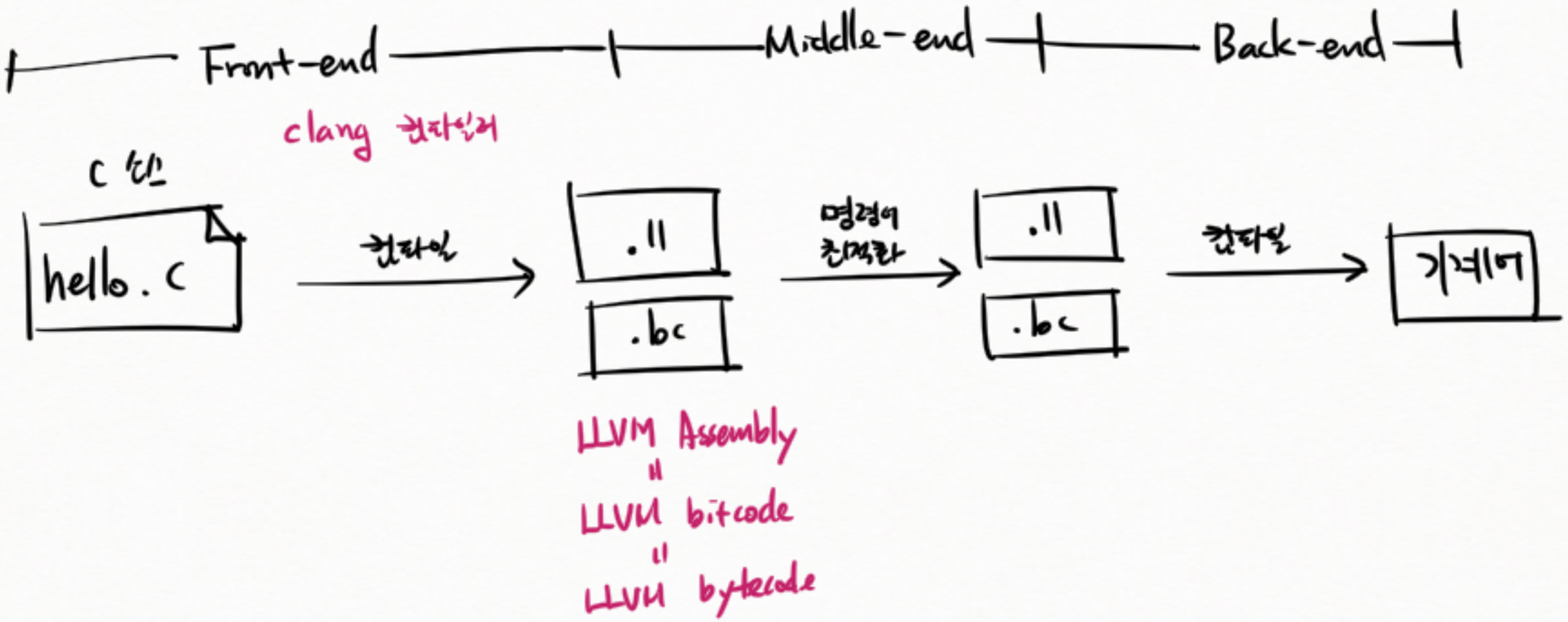
* LLVM 방식을 사용하는 이유

↳ 새 프로그래밍 언어나 컴파일러를 만들기 쉽다.

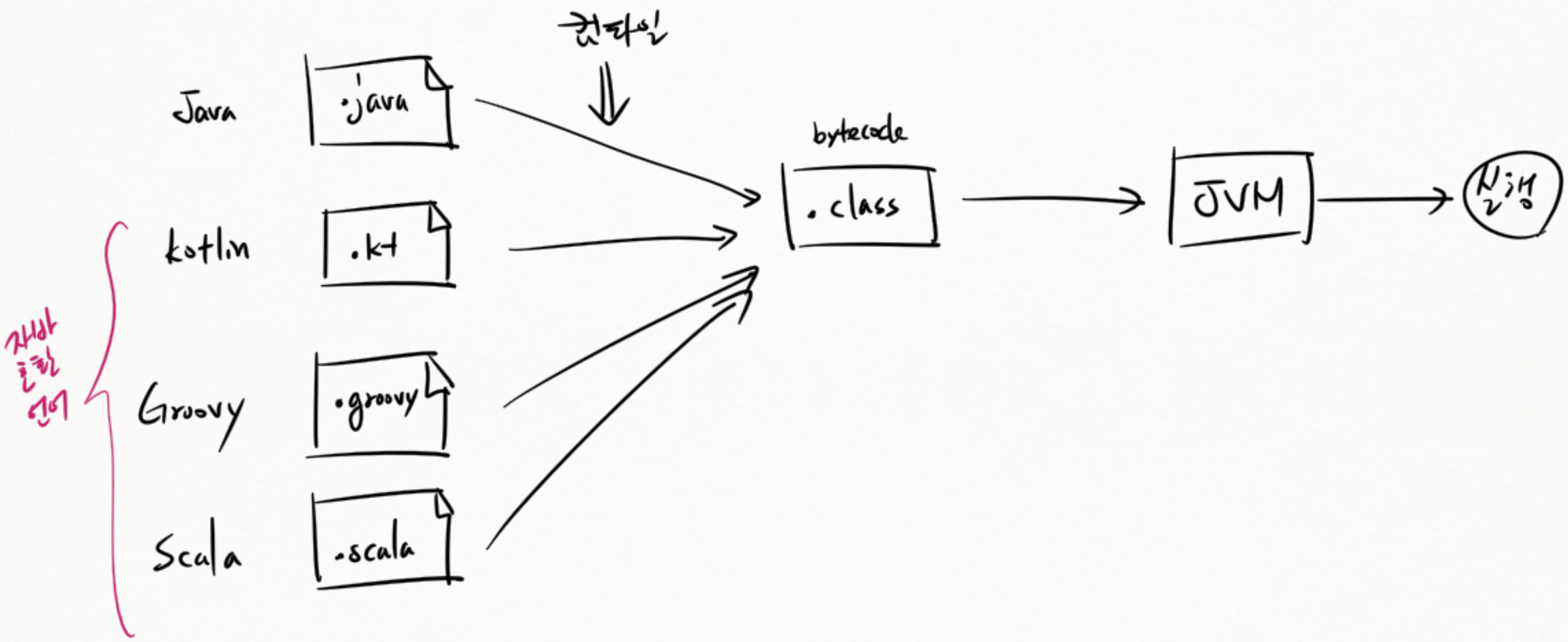
② LLVM 이후



* LLVM 컴파일 과정

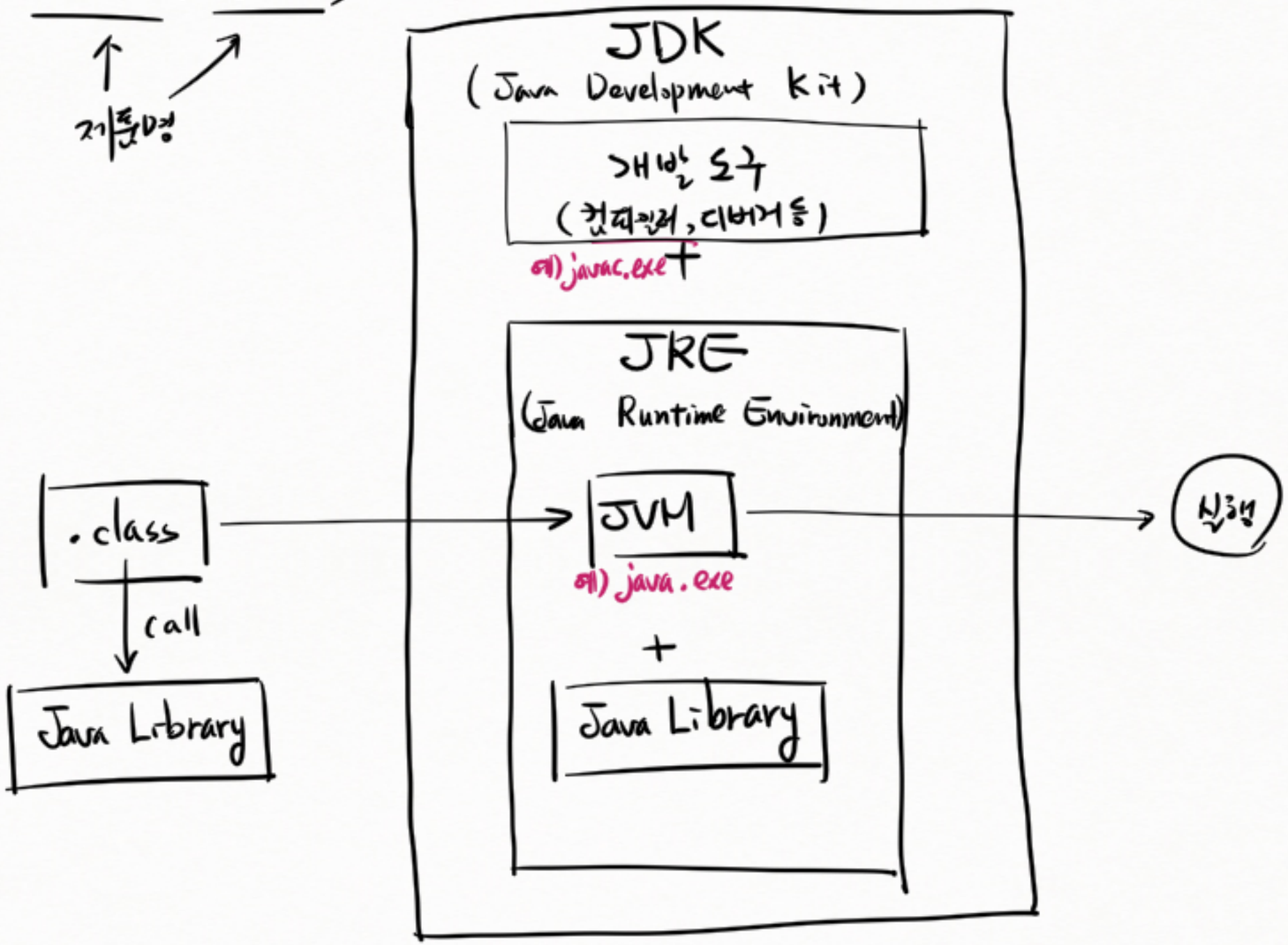

Front-end
Middle-end
Back-end
clang 컴파일러
C 언
hello.c
컴파일
.ll
.bc
LLVM Assembly
=
LLVM bitcode
=
LLVM bytecode
명령어 최적화
.ll
.bc
컴파일
기계어

* Java 와 LLVM


컴파일
Java
.java
kotlin
.kt
Groovy
.groovy
Scala
.scala
자바 플랫폼 언어
bytecode
.class
JVM
실행

* JDK, JRE, JVM

지칭명



* Java EE, Java SE, Java ME

과목 변경

- Java EE (Enterprise Edition) → 기업용 Application 제작에 필요한 도구 (Servlet/JSP API, EJB, WebService, …)
- Java SE (Standard Edition) → Java App. 제작 필수 도구
  - → JDK
  - → JRE
  - → Server & JRE (JRE ∧ UI API) without + 서버 전용 도구
- Java ME (Micro Edition) → (embedded) 임베디드 App. 제작에 필요한 도구 (H/W 에 실행되는 App.)

* Eclipse IDE

Eclipse IDE
개발 플랫폼

+ plug-in ⟹ 개발도구 확장

- JDT
- CDT
- ⋮

* Java Project 폴더 준비

① 프로젝트 폴더 생성

c:\Users\bitcamp\git\bitcamp-ncp\

User Home

↳ boot-project\

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

```
$ gradle init
```

③ 기본 예제 프로그램 실행

```
$ gradle -q run
```

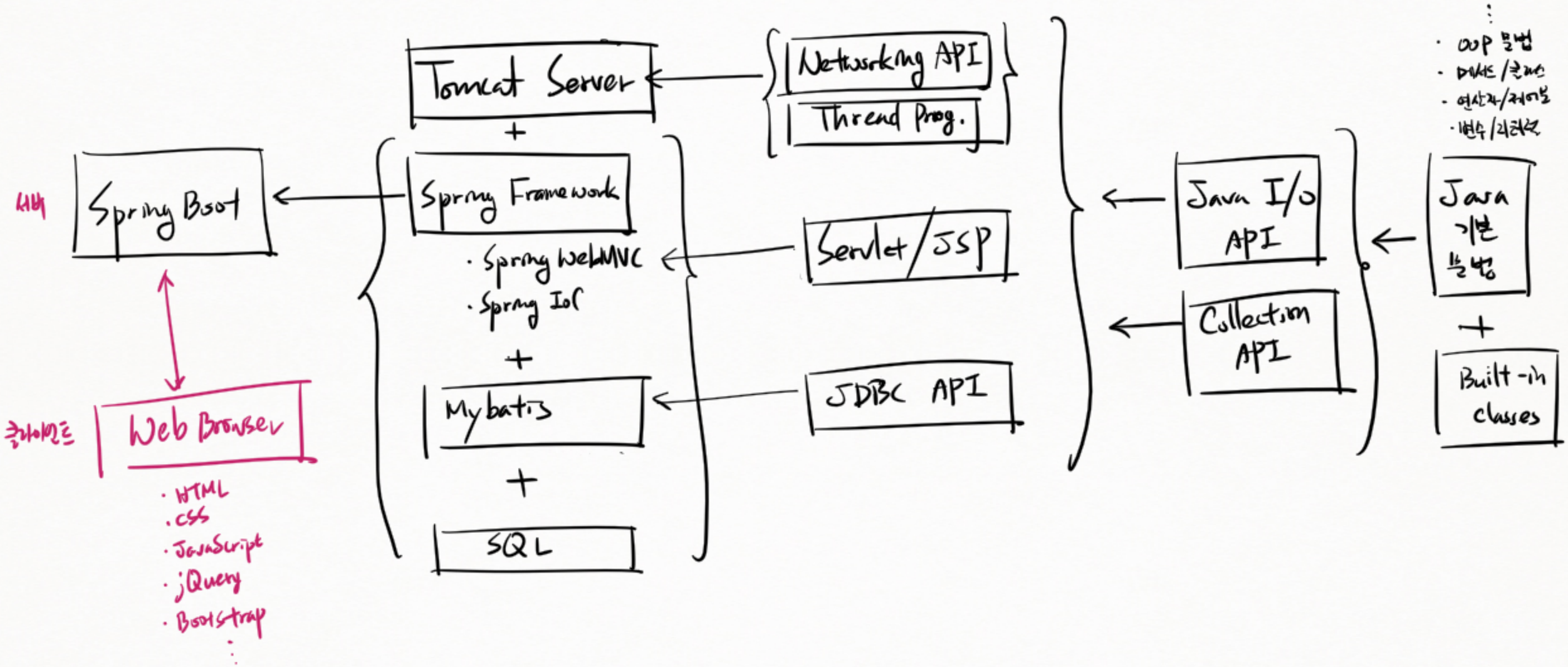
```
boot-app/                  ← 프로젝트 폴더
├ app/                     ← 하위 프로젝트. 메인 프로젝트
│  ├ src/
│  │  ├ main/
│  │  │  ├ java/           ← Java 소스 파일을 두는 폴더
│  │  │  └ resources/      ← 기타 파일을 두는 폴더
│  │  │                      (.properties, .xml, .html 등)
│  │  └ test/
│  │     ├ java/           ← 단위 테스트를 수행하는 자바 소스 파일
│  │     └ resources/      ← 단위테스트 관련 기타 파일
│  ├ build.gradle          ← build script 파일
│  │                         (빌드 설정 정보)
│  └ build/                ← 빌드로 생성한 파일 및 디렉토리를 두는 폴더
├ setting.gradle           ← 프로젝트 설정 정보
├ gradlew, gradlew.bat     ← Gradle 도구 설치 명령 파일
│                            ↳ Gradle 도구가 설치 되지 않았을 때 사용
├ gradle/                  ← gradlew, gradlew.bat 가 사용하는 파일이 들어있다
└ .gradle/                 ← gradle 도구들
```

* SpringBoot 프로젝트 만들기

spring.io 사이트 방문 → 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 파일 자동로딩
  3. Spring Web

* Java 에서 Spring Boot 까지



* Maven 빌드와의 표준 프로젝트 디렉토리 구조

```
~/git/bitcamp-ncp/   ← Git Repository
│
├── boot-app/   ← project 폴더
│   │
│   ├── app/   ← sub project (main)
│   │   │
│   │   └── src/   ← 소스 폴더
│   │       │
│   │       ├── main/   ← 프로그램 소스
│   │       │   │
│   │       │   ├── java/   ← 자바 소스
│   │       │   ├── kotlin/   ← 코틀린 소스
│   │       │   └── resources/   ← 기타 파일
│   │       │
│   │       └── test/   ← 테스트 소스
```

* Maven 빌드와의 표준 프로젝트 디렉토리 구조

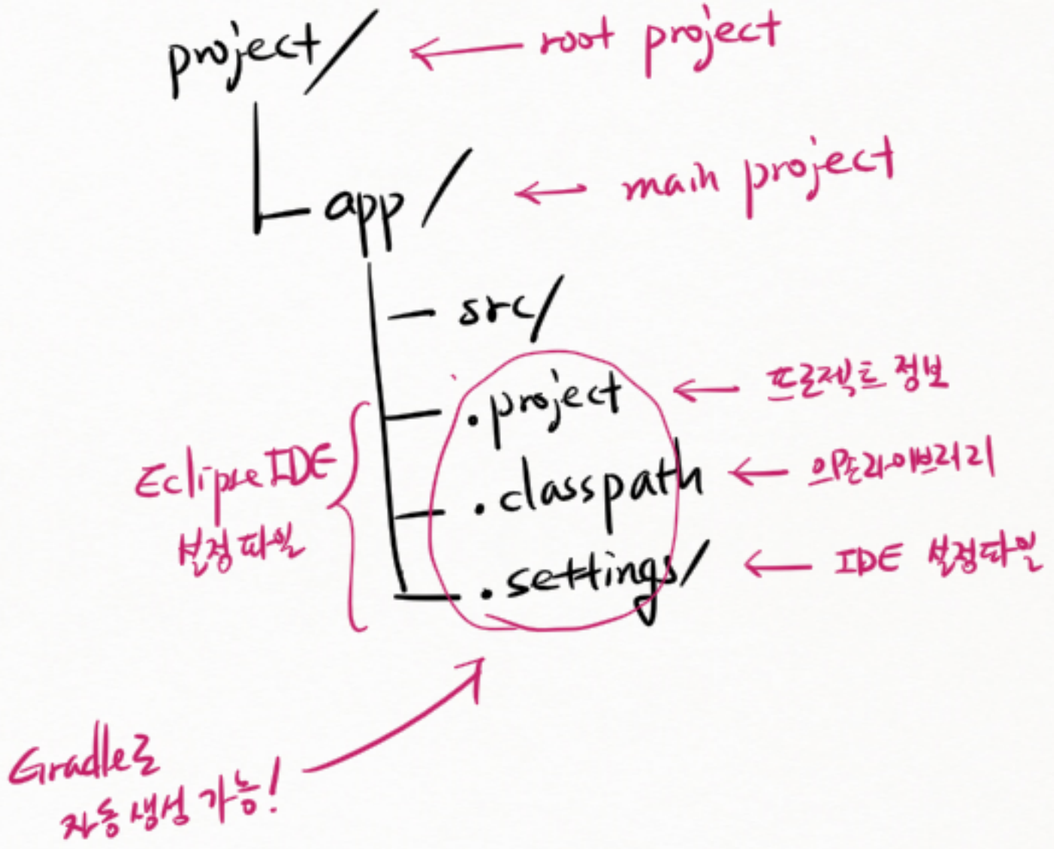
```
~/git/bitcamp-ncp/          ← Git Repository
    |— boot-app/            ← project 폴더
        |— app/             ← sub project (main)
            |— src/         ← 소스 폴더
            |— build.gradle ← Gradle 빌드 스크립트 파일
        |— settings.gradle  ← Gradle 빌드 보조 스크립트 파일
        |— gradle/          ← Gradle 프로그램을 local에 설치해주는 도우미 프로그램
        |— gradlew.bat      ← gradle/ 폴더에 들어있는 설치 프로그램을 실행하여 gradle을 설치하고
                              설치한 gradle을 사용해서 gradle 명령을 실행하는 shell 프로그램
        |— gradlew          ← Unix/Linux/macOS 용
        |— .gradle/         ← Gradle 도구가 작업하는 동안 사용하는 임시 폴더
```

빌드 설정 파일, groovy 프로그래밍 언어로 작성

하위 프로젝트 선정 및 공통 설정 정보

* Eclipse IDE 와 프로젝트

Eclipse IDE

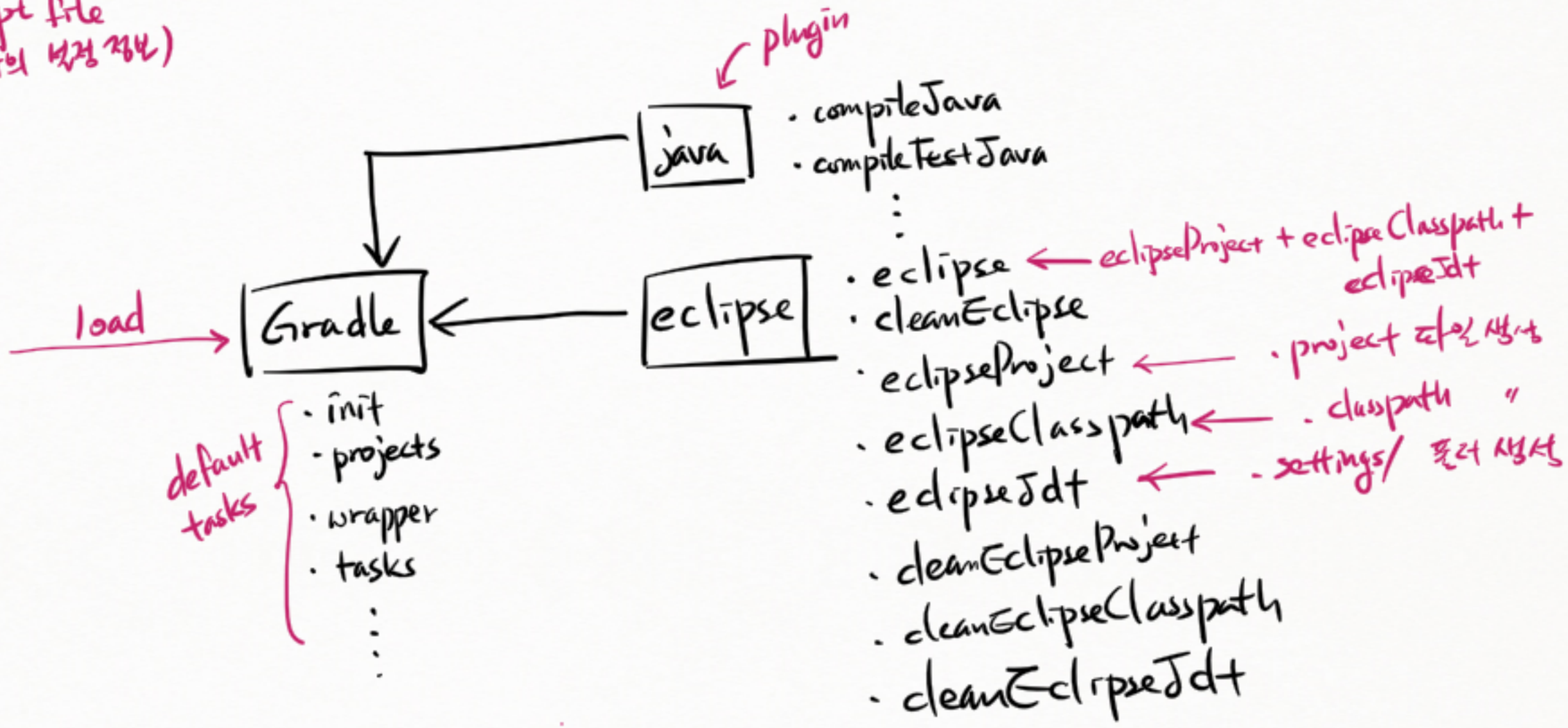


* Gradle 빌드 도구

[build.gradle]

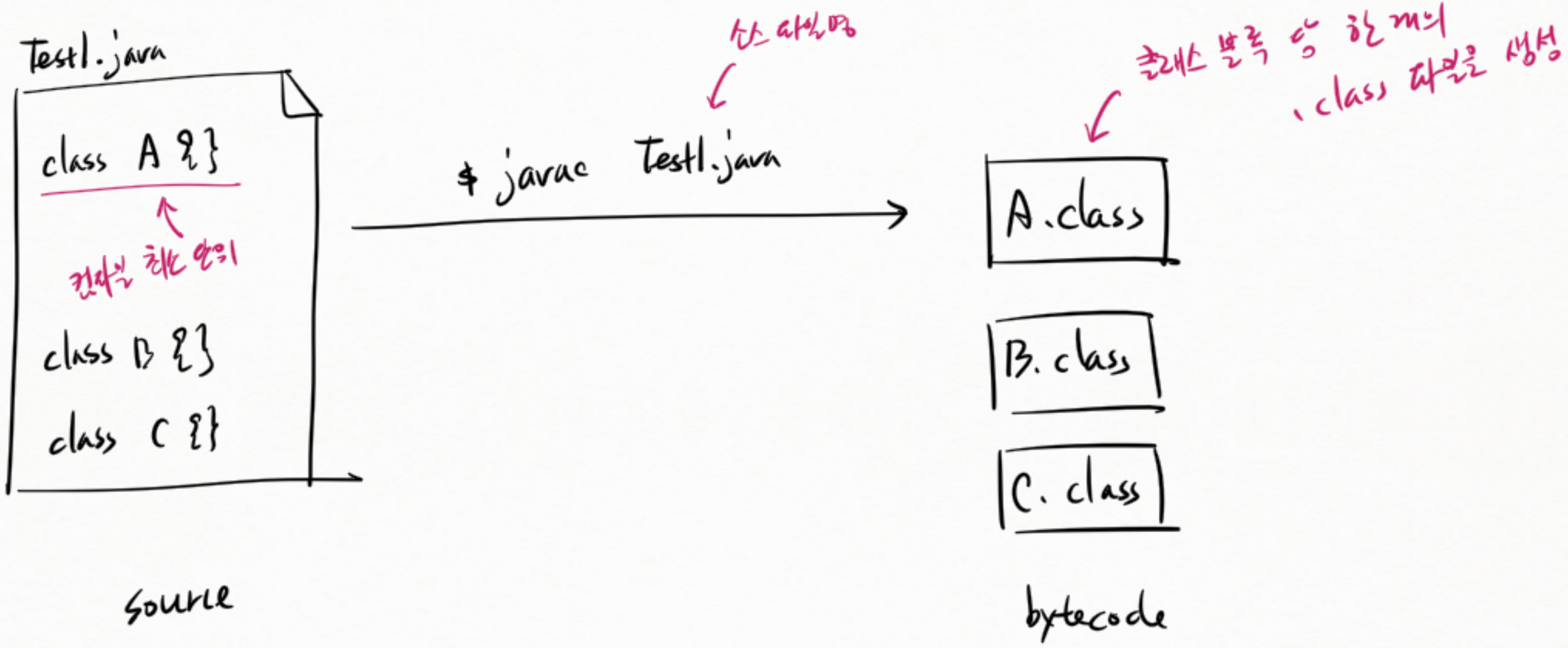
build script file
(빌드 도구의 설정 정보)

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



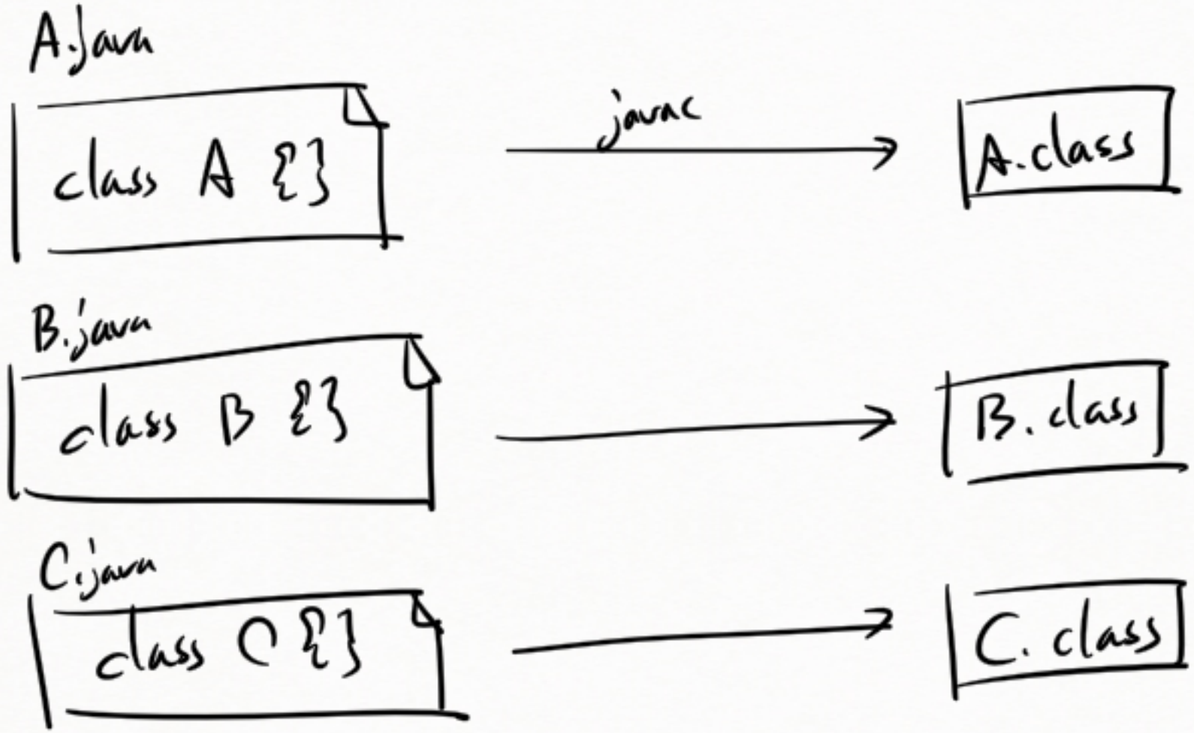
Java 문법

* .java 와 .class, 컴파일



* class 분리된 단스 파일

유지 보수가 쉽도록
(1 class
1 단스 파일)



* 순수 자바는 .class 파일 폴더

↳ 배포 파일을 추출하는 등 관리가 쉽도록 폴더

binary

src/ A.java
B.java
C.java

bin/ A.class
B.class
C.class

myapp$ javac -d bin src/A.java

컴파일결과물 두는 폴더

* 소스 파일과 디렉토리

↳ 여러 개의 소스 파일을 관리하기 쉽도록 하위 디렉토리에 분류한다.

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

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

→

```
bin/ A.class
     B.class
     C.class
```

컴파일러로 .class 클래스 파일은 소스 파일의 경로와 동일하게 분류되지 않는다.

* 소스 파일과 패키지

↳ 클래스 블록을 분류하기 위해 만든 문법

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

* 소스 파일 컴파일

소스 파일을 찾기 쉽도록
소스 파일 또한 package 이름과
같은 폴더에 둔다.

src/A.java

```
class A {}
```

src/p1/B.java

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

src/p2/C.java

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

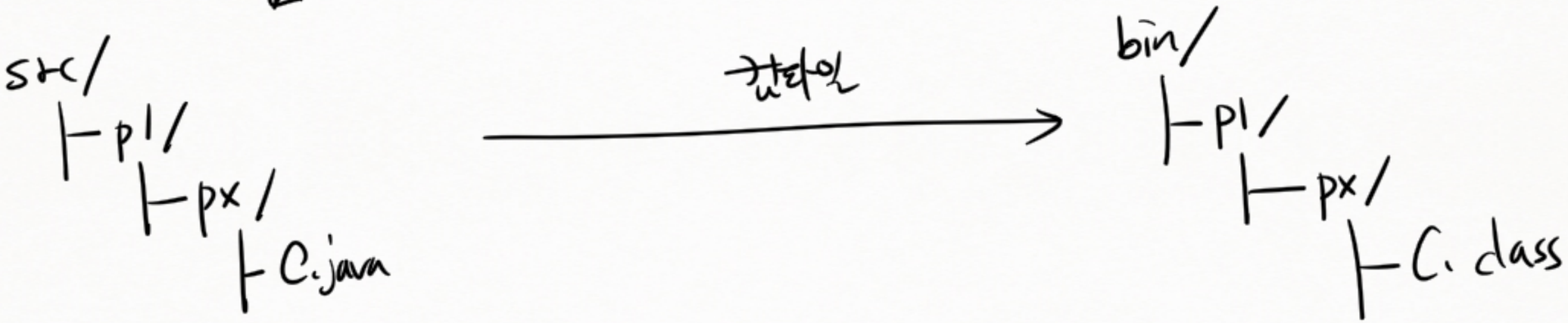
$ javac -d bin

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

bin/ A.class
- p1/ B.class
- p2/ C.class

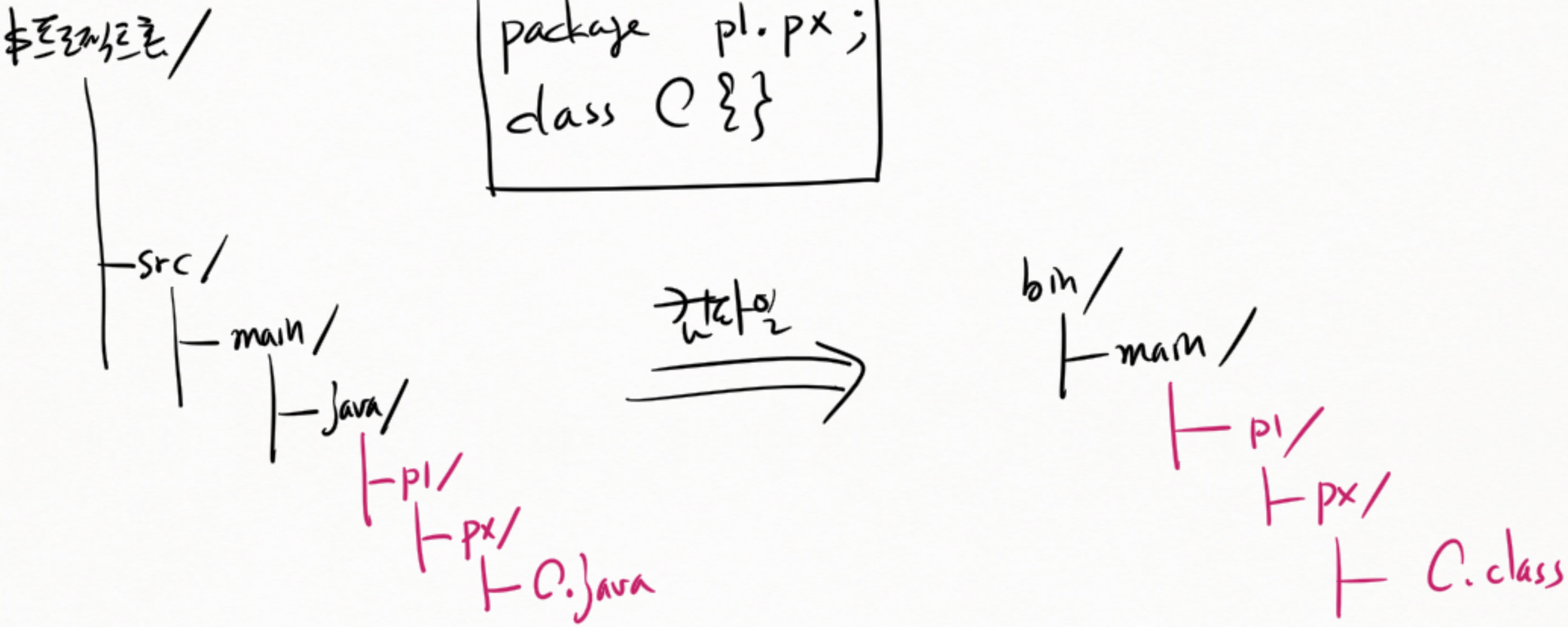
* 패키지와 디렉토리

```
package p1.px;
class C {}
```

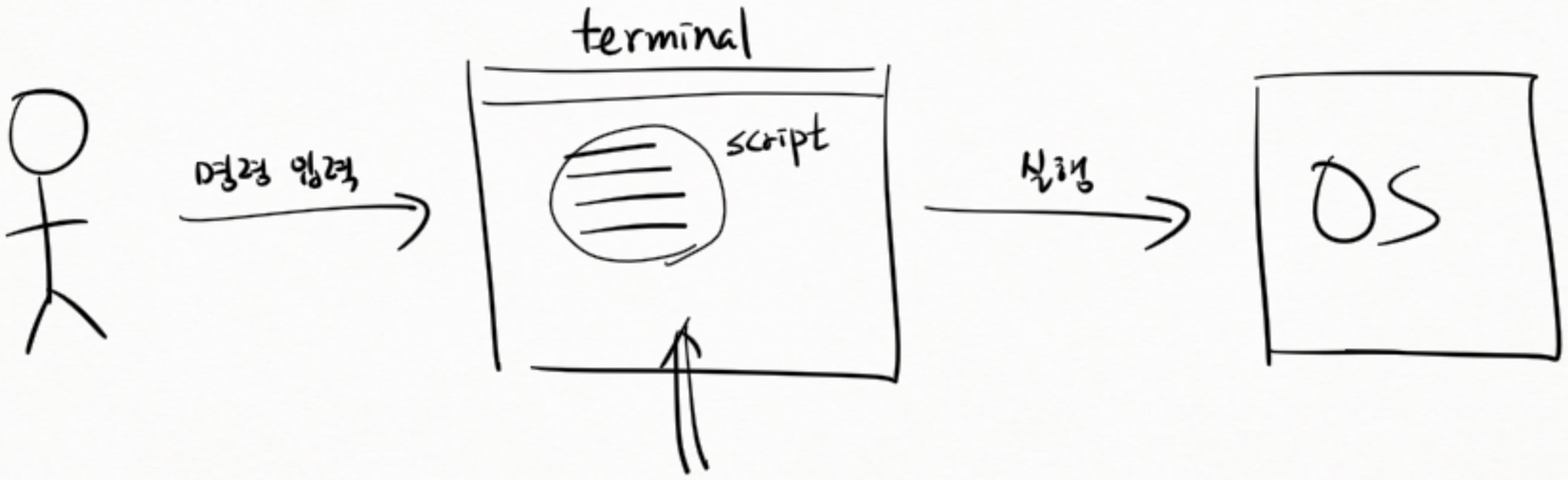


* Maven 프로젝트의 폴더 디렉토리 구조로 소스 파일 관리

```
package p1.px;
class C {}
```



* shell script



Windows : Batch script (.bat), PowerShell

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

* 클래스 블록의 공개 여부

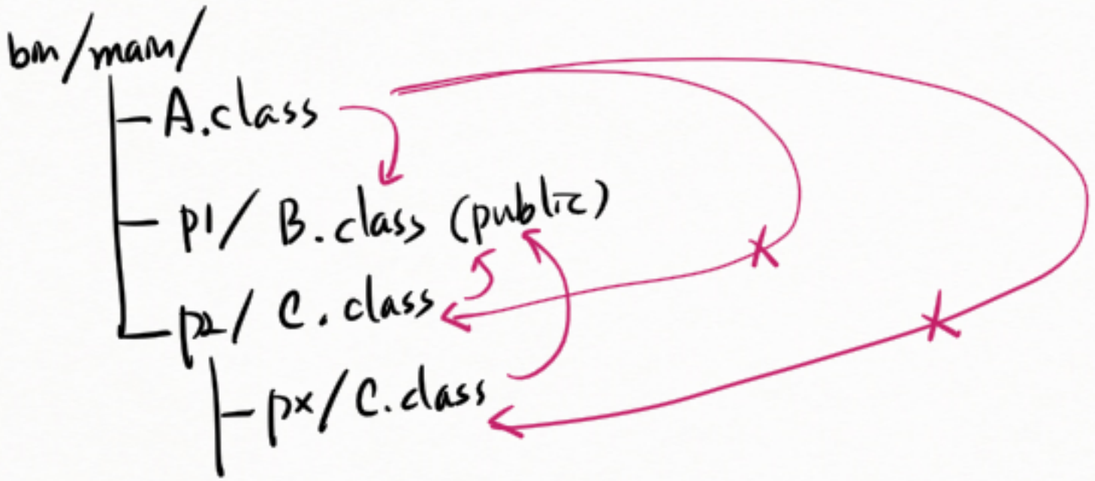
A.java

```
public class A { }
```

↑ 공개 여부를 지정 : 다른 패키지에서 이 클래스를 사용할 수 있다.

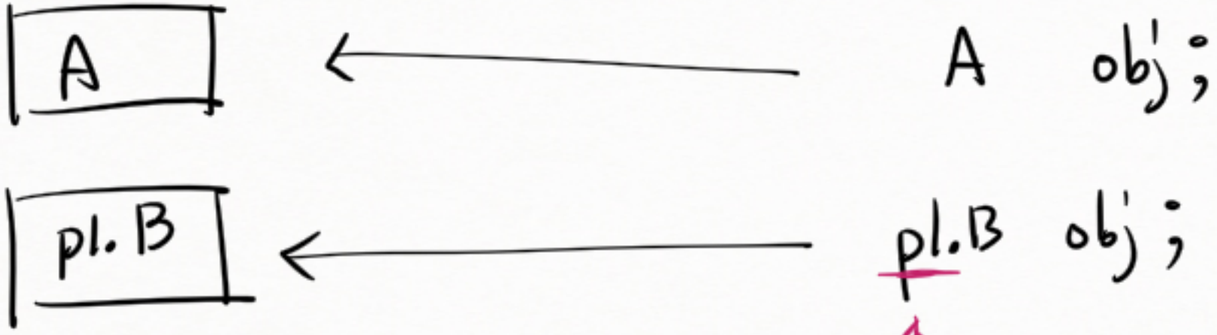
↳ 공개 클래스의 파일명은 반드시 클래스명과 같아야 한다.

* public 과 non-public



public 클래스를
다른 패키지의 클래스에서
접근 가능.

* 다른 패키지 클래스에 접근



패키지명 : 패키지에 소속된 클래스를 사용할 때는 반드시 패키지를 지정해야만 컴파일러가 찾을 수 있다.

☆ 컴파일

```
$ javac -classpath bin/main A.java
```

-classpath = -cp

패키지명.class 파일이 들어있는 폴더

* -classpath 사용법

javac -classpath bin/main

↑ 루트 패키지가 들어있는 폴더이어야 한다.

~~bin/main/p1~~

↑ 패키지 경로를 직접 지정할 수 없다

* 다른 패키지의 클래스 사용시 import 문

A

p1.B

p2.C

p2.px.C

p2.px.aaaaa.bbbbb.ccccc.ddddd.D

```
A obj;
p1.B obj2;
p2.C obj3;
p2.px.C obj4;
p2.px.aaaaa.bbbbb.ccccc.ddddd obj5;
```

반복을 방지하기 위해

```
import p1.B;
import p2.C;
import ~~p2.px.C;~~
import p2.px.aaaaa.bbbbb.ccccc.ddddd.D;
```

위의 C와 충돌함 즉 모호

```
A obj;
B obj2;
C obj3;
p2.px.C obj4;
D obj5;
```

* -sourcepath

↳ 사용하는 클래스의 소스 파일이 있으면
-classpath로 경로를 지정할 필요가 없다.

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

A.java를 컴파일 할 때,
A 클래스가 사용하는
다른 클래스의 소스 파일 경로를 알려준다.
||
A.java를 컴파일 할 때
A가 사용하는 클래스의 소스 파일도
같이 컴파일 된다.

Java App. 의

entry point

(진입점)

main()

* main()

클래스를 지칭할 때 .class 확장명을 적으면 안된다.

```
$ java A ↵
```

bytecode 인터프리터
=
bytecode player
=
JVM

① A 클래스 코드를 메모리에 적재(loading)
② bytecode 검증
③ main() 호출

변수명을 달리 해도 된다.

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

main() 의 형식을 반드시 지켜야 한다
=
메서드 형식 (Method Signature)
=
function prototype

* -classpath

↳ 루트 패키지 와 클래스가 들어있는 디렉토리 경로

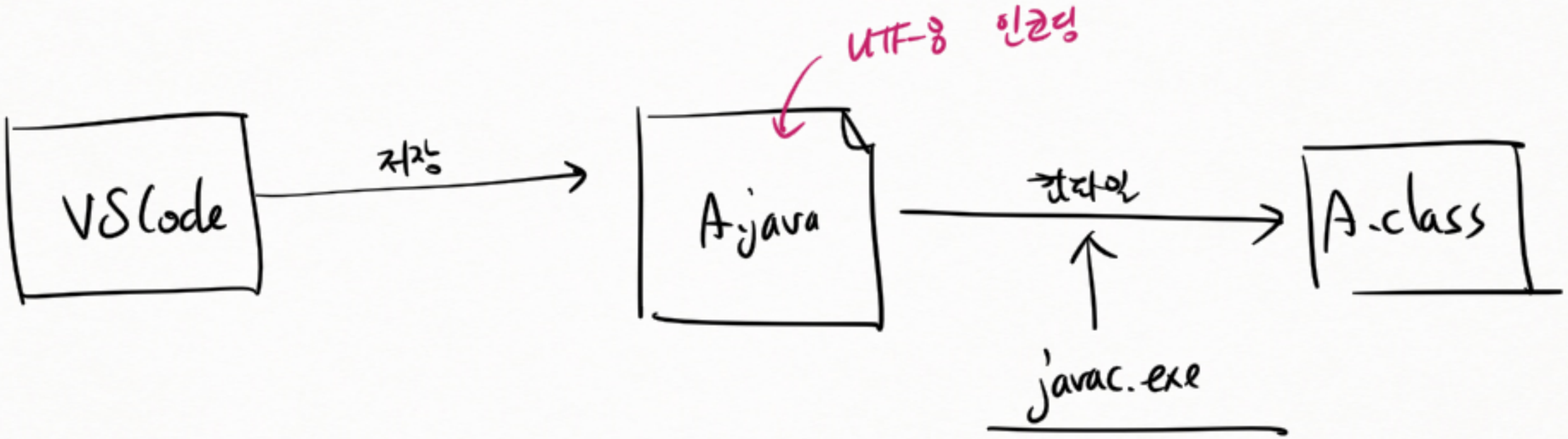
↳ 자바 컴파일러 와 JVM은 클래스를 찾을 이 옵션의 경로를 따라간다.

```
$ java -classpath bin/main A ↵
       ||
       -cp

$ java -classpath bin/main p1.B ↵

$ java -cp bin/main/p1 B ↵
```

* 소스파일 인코딩



Windows : MS949

Unix
macOS
Linux } UTF8

· 컴파일러가 소스파일을 읽을 때,
소스파일의 인코딩 문자집합이
OS의 기본 문자집합과 같을 것이라
간주한다.

* -encoding

* javac -d bin/main -encoding UTF-8 src/—/A.java

↑ 소스 파일의 인코딩을 알려준다.

OOP Language 기본 문법

- 값을 표현하는 문법 → Literal
  예) 123, 123.5, true/false, 'A', "___"
- ↓ 값을 저장하는 문법 → Variables
  예) int a; String s;
- ↓ 값을 다루는 문법 → Operators
  예) +, -, *, /, %, <, <=, >, >=, &&, !!, ==, !=, <<, >>, >>>
- ↓ 실행 흐름을 제어하는 문법 → Statement
  예) if ~ else, switch (분기문), while/for/do~while (반복문)
- ↓ 명령문을 기능에 따라 묶는 문법 → Method
  예) void print( ){—}
- ↓ 관련된 메서드를 묶는 문법 / 새 데이터 타입을 정의하는 문법 → Class (↑ Abstraction (추상화))
  예) class System {—}
- ↓ 코드 재사용을 높이는 문법 → Inheritance (상속), Polymorphism (다형성), Encapsulation (캡슐화)
- ↓ 코드 유지보수를 쉽게 하는 문법 → Abstract class (추상클래스), Interface (인터페이스), Nested class (중첩 클래스) 등

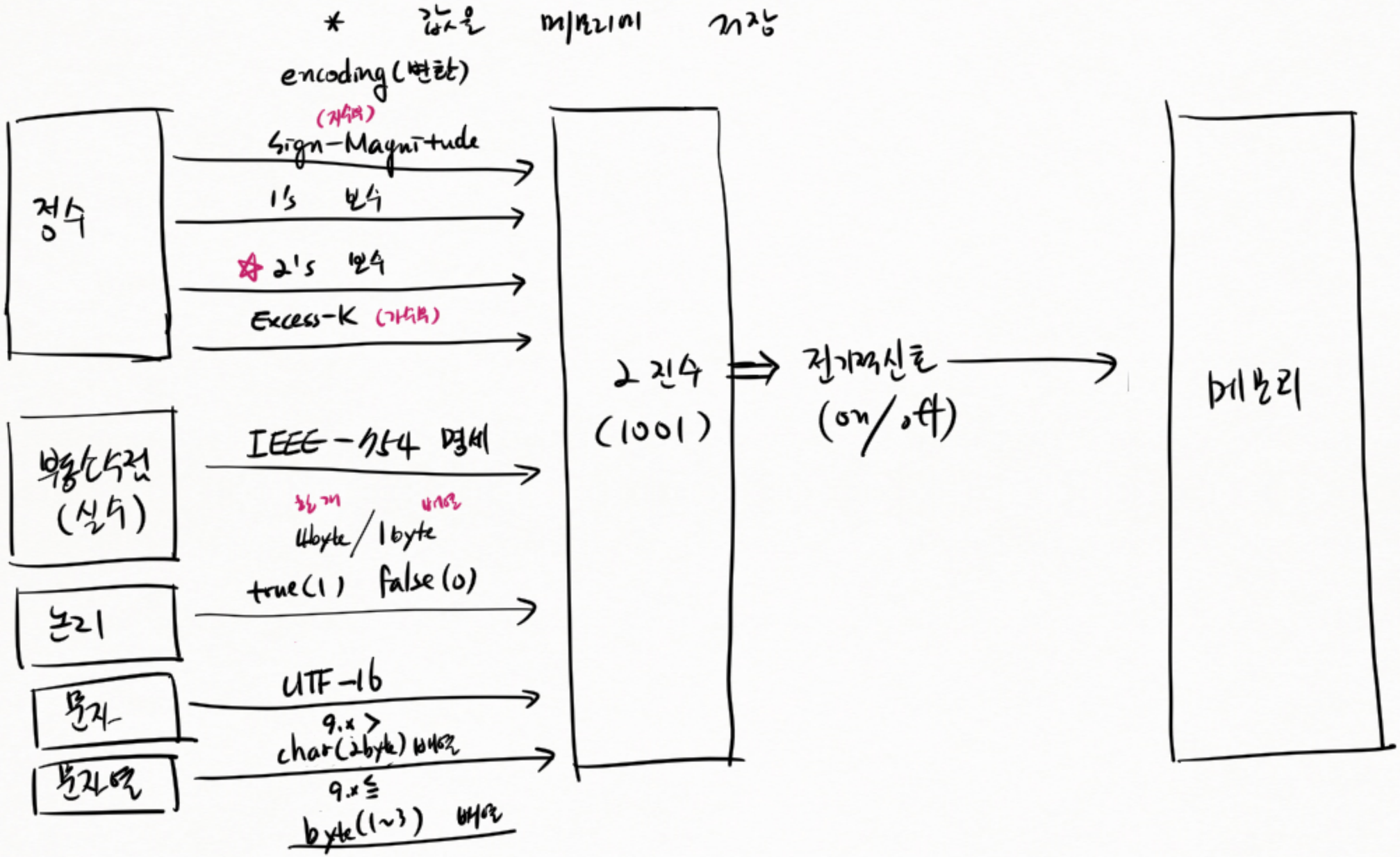
Literal

* 데이터 타입 & 리터럴

자바 기본 데이터 타입 (Primitive Data Type)

- 정수
  - → 4byte 리터럴 (int) 100
  - → 8 byte 리터럴 (long) 100L 100l
- 부동소수점
  - → 4 byte 리터럴 (float) ← 단정도 3.14f 3.14F
  - → 8 byte 리터럴 (double) ← 배정도 3.14
- 논리 → 4 byte 리터럴 true false
- 문자 → 2 byte 리터럴 'A' '가' ~~'AAA'~~
- 문자열 → 객체 "Hello"

⋮


* 값을 메모리에 기장
encoding(변환)
정수
(가수부)
Sign-Magnitude
1's 보수
2's 보수
Excess-K (가수부)
부동소수점
(실수)
IEEE-754 명세
4byte / 1byte
true(1) false(0)
논리
문자
UTF-16
9.x >
char(2byte) 배열
문자열
9.x ≤
byte(1~3) 배열
2진수
(1001)
전기적신호
(on/off)
메모리

* 정수 리터럴

int → 4Byte　　8byte ← long

10진수 : +100　　100L　100l

8진수 : -044

2진수 : 0B0101
0b0101

16진수 : 0XAC00
0xAC00

JavaScript 와 다르게
정수 리터럴의 메모리 크기가 있다
(표현할 수 있는 값의 범위가 있다)

100 (4byte 정수) : 약 -21억 ~ +21억

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

* 부동소수점 → IEEE-754 명세

+ 12.375 → 2진수 → 전기신호 → 메모리저장

1100

0.375
× 12
0.750 → 0
× 12
0.500 → 1
× 12
1.0 → 1

+

.011

12 . 375
⇓
1100 . 011 (2진수)
$= 1.100011 \times 2^3$
⇓
$X.\underline{100011} \times 2^{3}$

추출

⊕ 100011
(가수부)
⇓ Sign-magnitude
100011

3
(지수부)
⇓ Excess-k $= 2^{(8-1)}-1 = 128-1 = 127$
$3 + \text{bias값} = 3 + 127 = 130 = \underline{10000010}$

bias 값
‖
Excess-k 값

128	64	32	16	8	4	2	1

2735.2705 (10진수)
$= 2.7352705 \times 10^3$

0	1	0	0	0	0	0	1	0	1	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

부호비트 지수부 가수부

(4byte)
(1B3)
0x41460000 16진수
C1460000
(음수)

Variables

* 변수 선언 (delaration)

↳ 값을 담는 메모리를 ~~생성~~ 준비시키는 문법

데이터타입 변수명;

자바 원시 타입 primitive data type

- 정수
  - byte 1 byte
  - short 2 byte
  - int 4 byte
  - long 8 byte
- 부동소수점
  - float 4 byte
  - double 8 byte
- 논리
  - boolean ㉠/①
- 문자
  - char 2 byte

reference type → 그 밖에 → 클래스

storage location

```
int a;
```

```
a = 100;
```

assignment operator

(할당연산자, 배정연산자)

대입연산자

* 정수 변수

4byte 정수 메모리

4byte 정수 리터럴

int a = 100;

long b = 100;

8byte 4byte

short c = 100;

2byte 4byte

byte d = 100;

1byte 4byte

예외조건: 리터럴 값이 메모리 보다 크기가 크더라도 값을 수 있다면 허용한다!

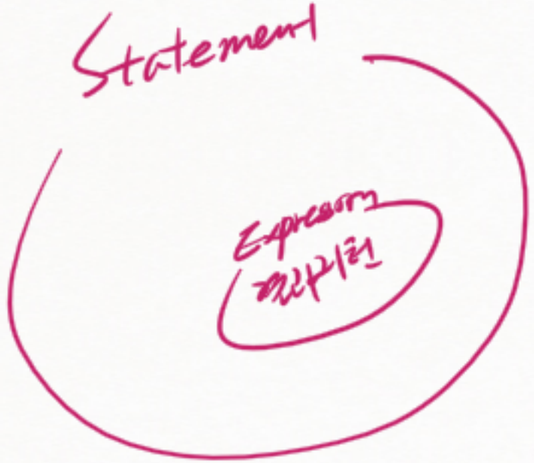
* l-value 와 r-value

l-value ↓ r-value ↓

반드시 메모리여야 함

```
a = 100;          // 리터럴, 변수
a = b * 3;        // expression
a = plus(100,200); // expression
                   //  ||
                   //  메서드 호출
```

~~100 = a;~~



* char 변수

char c;

UTF-16
||
문자코드(UCS2)를 저장하는 2byte 메모리

0~65535
범위의 값만 허락된다

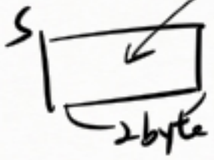
문자코드를 의미하는 정수값

c = 65;

System.out.println(c);
↓
A

short s;

-32768 ~ +32767
범위의 값만 허락된다.



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.

* ' ' 연산자

'A'

↓ return

65 ← A 문자에 대한 (UCS2 코드값 / UTF-16 / Unicode 2)

char c = 'A';

저장 ↑ ↓ 65

char c2 = 0xB62F;

c2 = '뜅';

저장 ↑ ↓ 0xB62F = 46622

c2 = '뜅' - 1;

내용

* 배열 문법

↳ 같은 종류의 메모리를 여러개 생성할 때 사용하는 문법

주소를 담는 변수 = reference 변수

☆ 데이터타입[] 변수명; ← 권장함

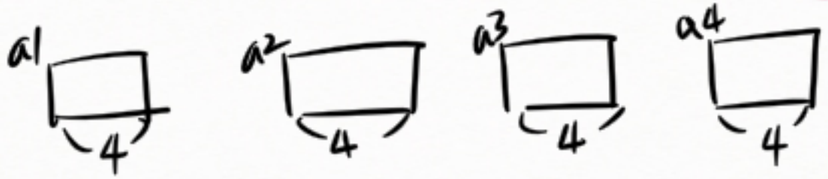
데이터타입 변수명[]; ← c style

변수명 = new 데이터타입[수량];

데이터타입[] 변수명 = new 데이터타입[수량];

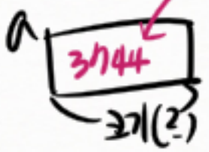
* 배열 선언

```
int a1, a2, a3, a4;
```



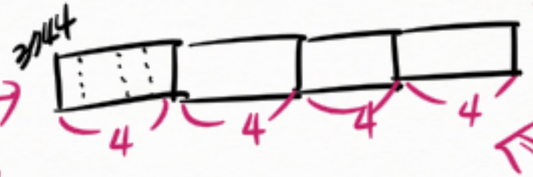
변수들이 개별적이기 때문에 연속된 메모리가 아니다!

```
int[] a;
```



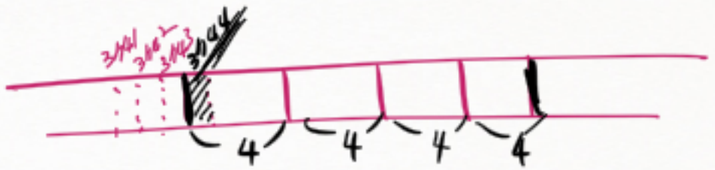
다른 메모리의 주소를 담을 반초 적절한 크기

```
a = new int[4];
```



연속된 메모리

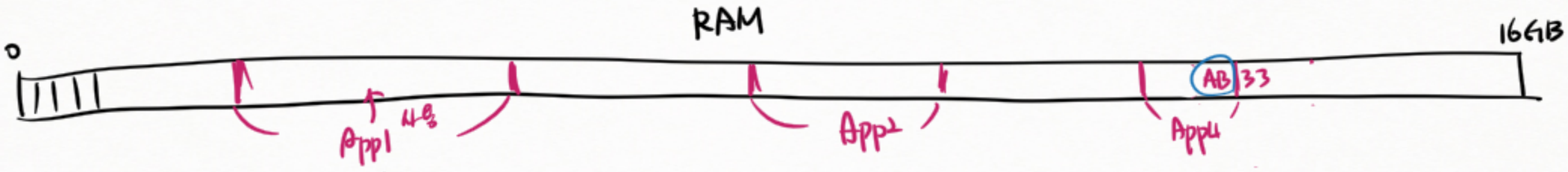
JVM은 Heap 영역에 연속된 int 타입의 메모리를 준비하고 리턴값은 준비한 메모리의 시작 주소이다.



new 명령을 통해 준비한 메모리 "인스턴스 (instance)"

예) int 배열의 인스턴스

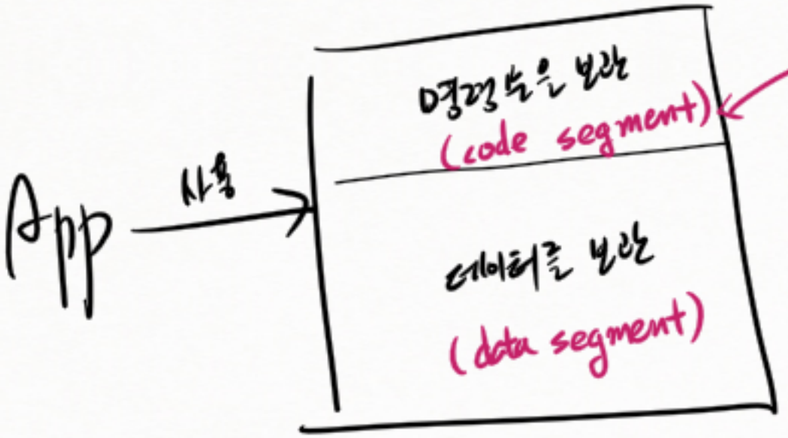
* RAM 과 메모리 주소



App. 을 실행하면
OS 가 App이 사용할 메모리를 지정한다

OS

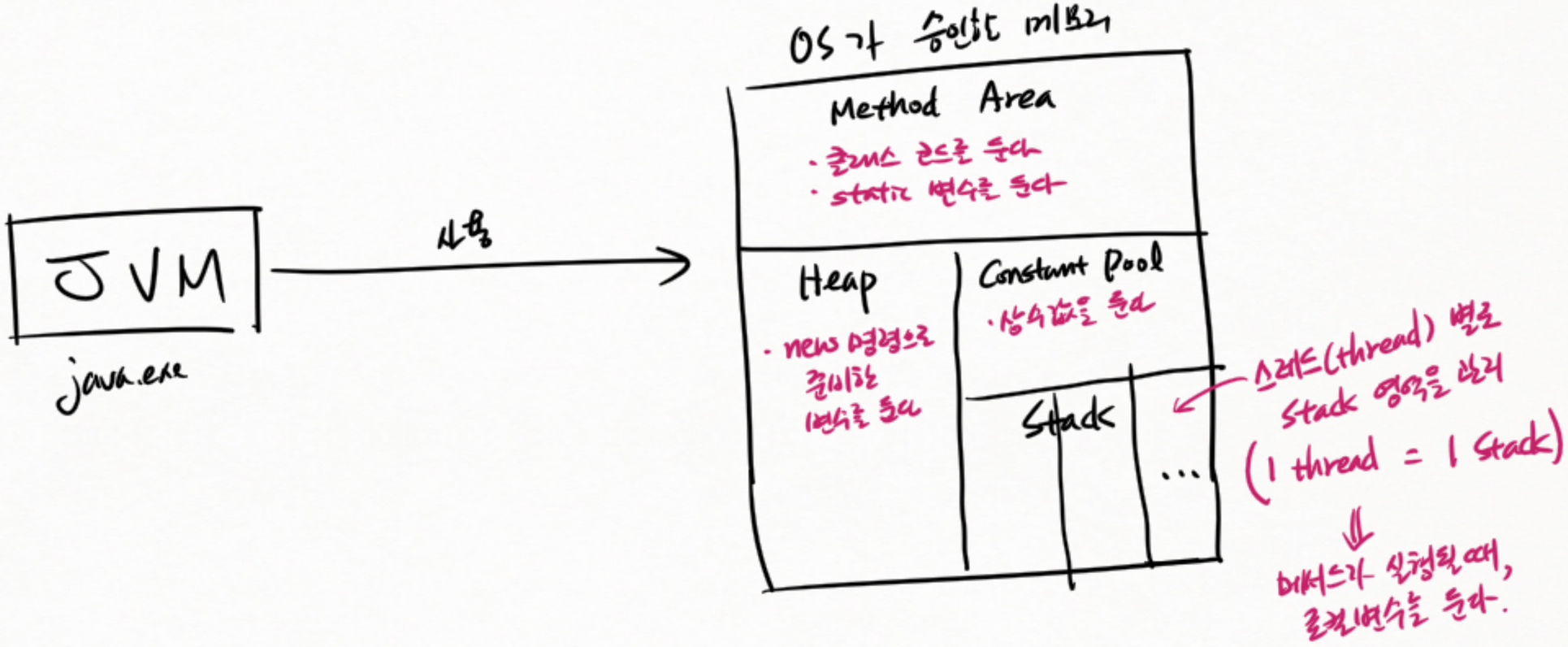
↳ App 의 실행이 종료되면
그 App.이 사용했던 메모리 영역은
다른 App.이 사용할 수 있도록 잠금이 해제된다.



OS로부터 사용을 승인 받은 메모리

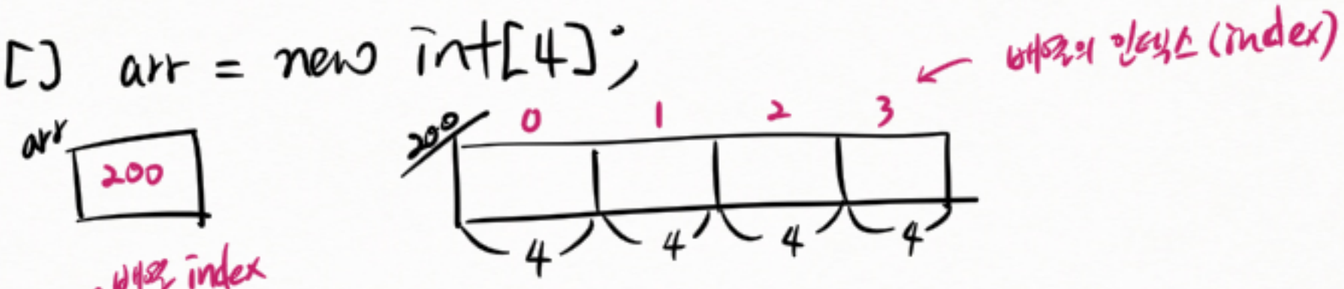
* 메모리로 연속된 바이트들의 1차원 배열이다.
↓
이해를 용이하게 보통 2차원으로 표현한다

* JVM 과 메모리


OS가 승인한 메모리
JVM
java.exe
사용
Method Area
· 클래스 로드를 돕다
· static 변수를 돕다
Heap
· new 명령으로 준비된 변수를 돕다
Constant Pool
· 상수값을 돕다
Stack
...
스레드(thread) 별로 Stack 영역을 관리
(1 thread = 1 Stack)
메서드가 실행될 때, 로컬변수를 돕다.

* 배열 메모리에 저장

```
int[] arr = new int[4];
```



배열 index

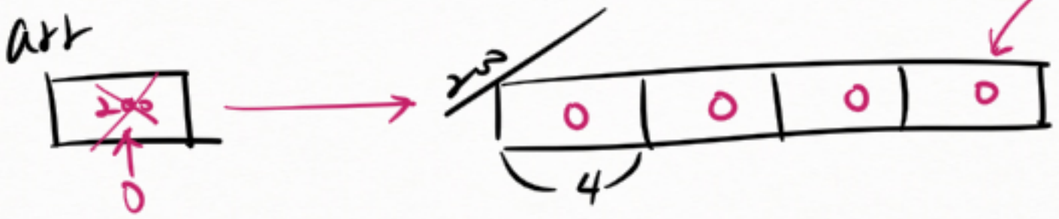
```
arr[0] = 100;
arr[1] = 200;
arr[2] = 300;
arr[3] = 400;
arr[4] = 500;
```

실행 오류 (runtime exception)
=
ArrayIndexOutOfBoundsException

* 배열 reference 변수와 null

↳ 배열 인스턴스의 주소를 담는 변수

```
int[] arr = new int[4];
```



인스턴스의 각 변수는 무조건 초기화 된다.

0으로

byte = 0
short = 0
int = 0
long = 0L
float = 0.0f
double = 0
boolean = false
char = '\u0000'

arr = ~~0~~; ← 레퍼런스의 주소를 초기화시킬때

null

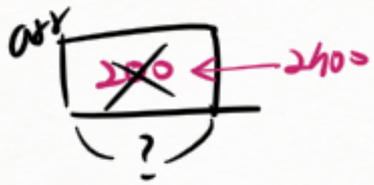
☆ int a;

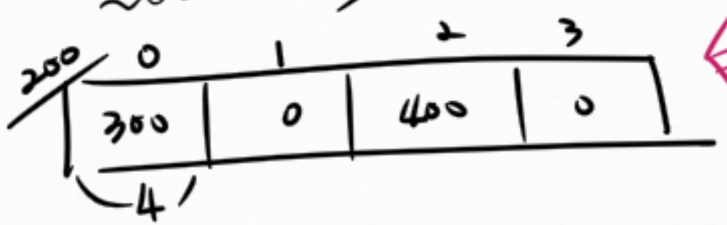
로컬 변수는 자동으로 초기화되지 않는다.
반드시 직접 초기화시켜야 한다.

* 배열 인스턴스와 가비지(garbage)

↳ 인스턴스의 주소를 잃어버려 사용할 수 없는 상태의 인스턴스

```
int[] ar = new int[4];
```



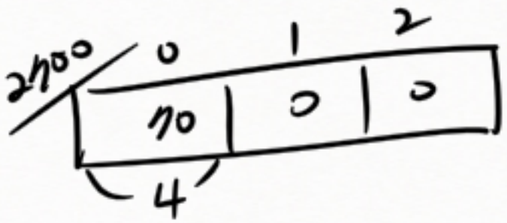


주소를 잃어버려 더이상 사용할 수 없는 인스턴스 "Garbage"

```
ar[0] = 300;
ar[2] = 400;
```

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

2400




```
ar[0] = 70;
```

2400

"Garbage Collector"에 의해 자동으로 메모리 해제된다

잔존장치를 줄이기 다른 용도로 재사용 할 수 있게 만든다.

언제 동작

① 메모리 부족 할 때

② cpu 한가할 때

* 가비지와 레퍼런스 카운트

JVM

↓ 레퍼런스 카운트 관리

주소	참조수
200	Garbage
300	T
400	—

```
int[] arr1 = new int[3];
```

arr1 [200] ← 400

200: [0 | 0 | 0]

```
int[] arr2;
```

arr2 [200] ← 300

```
arr2 = arr1;
```

```
int[] arr3 = new int[2];
```

arr3 [300]

300: [0 | 0]

```
arr1 = new int[4];
```

400: [0 | 0 | 0 | 0]

```
arr2 = arr3;
```

값. println(값)

작업수행에 필요한 값

작업을 수행

작업할 때 사용할 값

콘솔정보

System.out.println(no);

System: 도구함 (class)

out: 변수 (field)

println: 작업자 = 연산자 (method) = function

no: 파라미터

시스템 표준 출력장치

⇓

Eclipse의 console창

"번호: " + no

↓ 문자열 생성

"번호:" + "1"

↓ 새 문자열 생성

"번호: 1"

실행 도중에서
임시로 문자열이 생성된다.

↓ 전달

System.out.println( )

escape character

↳ 문자가 아니라 명령이다.

```
System.out.printf("번호: %d\n", no);
```

변수의 값 삽입

* 키보드 입력

System.in ← 키보드 데이터 : 사용자가 키보드의 키를 눌렀을 때
어떤 키를 눌렀는,
이때 caps lock 의 상태를 무엇인지,
shift 키의 상태는 무엇인지
⋮
키상태

Scanner keyScanner = new Scanner(⇓);

↖ 키보드에서 읽은 바이트를 가공한 문자 데이터
UTF-16 (UCS2)

☆ 개수제
||
데이터가 저장된 메모리
(값)

keyScanner.nextLine() ← keyScanner (값) 의 문자 데이터를 받아서
한 줄의 문자열을 만든 후 리턴한다.
(사용자가 엔터를 칠때까지
읽어 들인다)

⇓ 정수 값으로 바꾸고 싶다면

Integer.parseInt( )

⇓ 리턴

정수

☆ 객체와 메서드, 파라미터

키보드에서 읽은 문자 데이터

⇓

keyScanner.nextLine()

값

정보를 사용해서 작업 수행

⇓

Integer.parseInt(문자열값)

도구함 class

파라미터로 받은 값을 사용해서 작업 수행

외부 호출자에게 작업결과 리턴 ⇐ 객체.메서드(파라미터, 파라미터, …)

값을 읽고

값 사용

작업결과 저장

작업수행

값저장

파라미터에 들어 있는 값을 사용만 한다

* Date 클래스

OS의 시간 데이터

⇓ 읽어서

System.currentTimeMillis()

⇓ long 타입의 정수값

1970년 1월 1일 0시 0분 0초 이후에 경과된 시간

Date today = new Date(밀리초)

⇓ 가공

년, 월, 일 데이터

년, 월, 일

today.toString()

⇓ 리턴

"yyyy-MM-dd"

Month

분

↓

hh:mm:ss

yyyy-mm-dd

Minute

Method

↳ 기능 단위로 코드를 묶는 문법

↳ 재사용이 쉽다

↳ 코드가 블록 단위로 정리되어 관리가 쉬워진다.

* 여러 메서드가 변수 공유하기

```
class A {
    int a;

    void m1() {
        int a;
        a=200;
    }

    void m2() {
        a=100;
    }
}
```

a는 메서드에서 선언한 변수로 접근 불가!

* parameter 와 argument

parameter

```
int plus(int a, int b) {
    return a+b;
}
```

argument

```
int result = plus(100, 200);
```

* 가변 파라미터 배열

String[] arr = {"김구", "유관순", "안중근", "연개소문"};

arr
200

"김구"	"유관순"	"	"
0	1	2	3

hello(arr);
200

* call by value

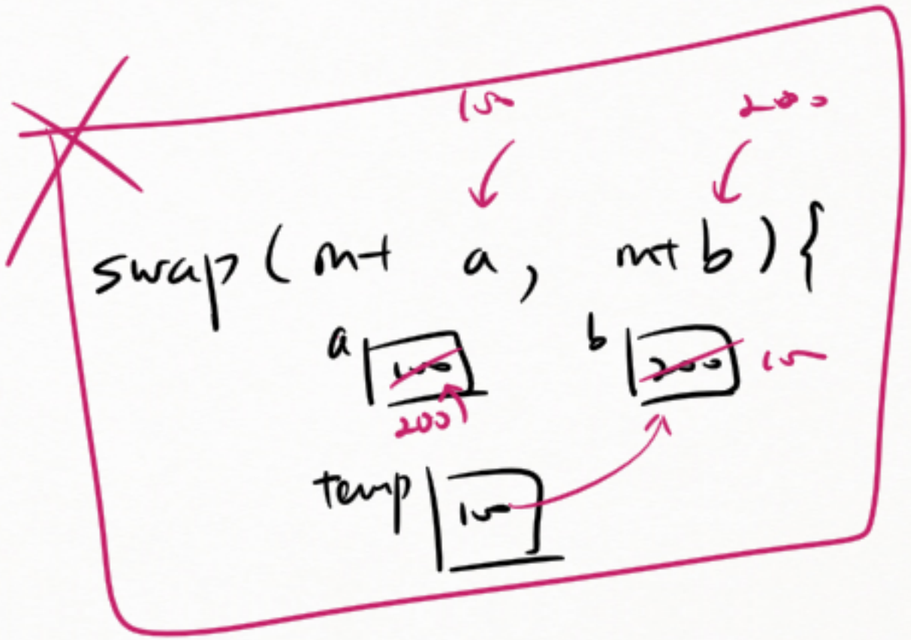
a [100] b [200]

swap(a, b);

100 200

int 값

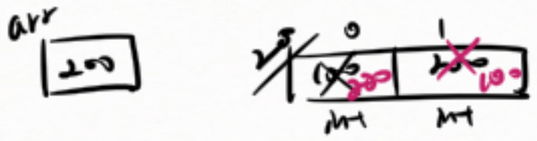
(primitive type의 값)



* call by reference

```
int[] arr = new int[]{100, 200};
```

생략 가능




```
swap(arr);
```

200

배열 주소
(reference)

200 ← main() 에서 만든 배열의 주소

```
swap(int[] arr) {
```

arr | 200 |

temp | 100 |

* call by value 와 메모리 상태

main() 실행

메서드를 호출하면

메서드가 사용할 로컬변수는 "스택 메모리 영역"에 준비된다.

호출 메서드의 로컬 변수가 놓인다

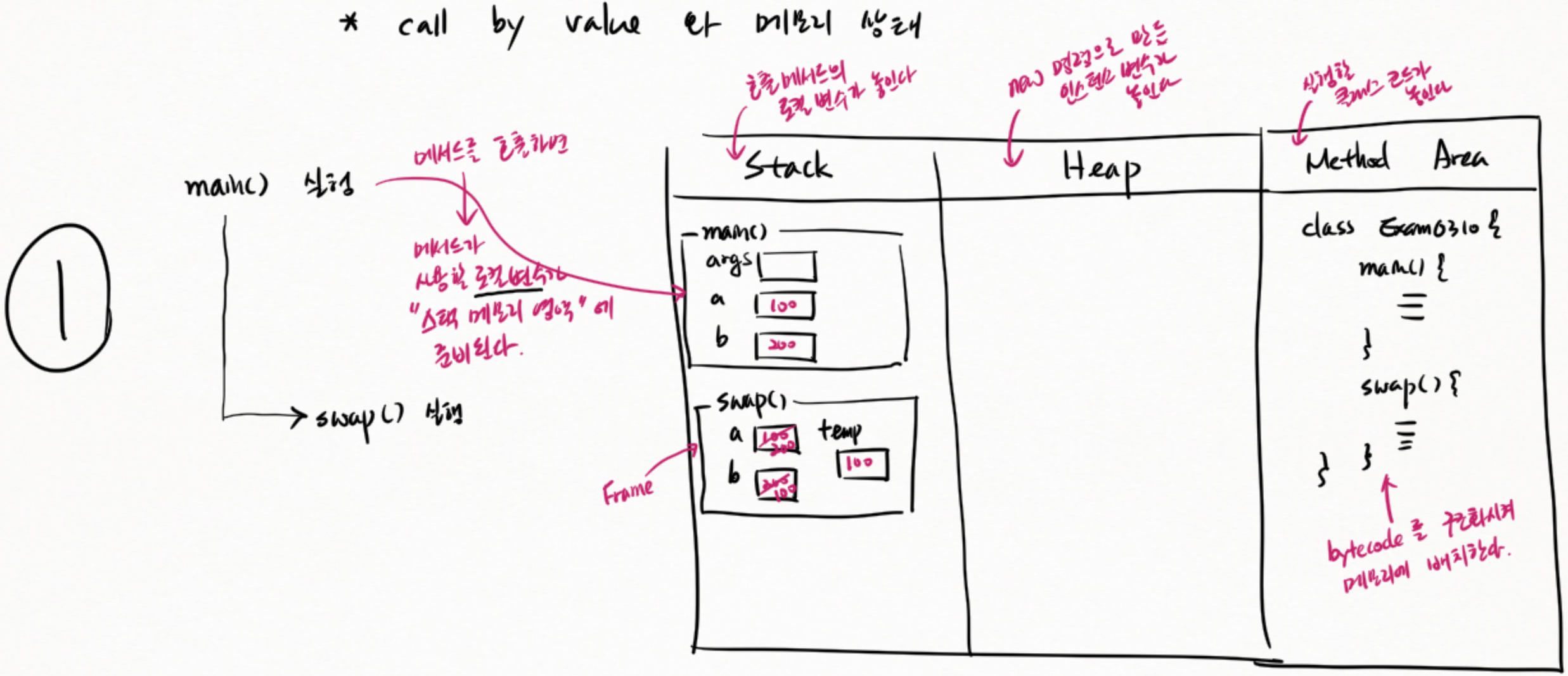
new 명령으로 만든 인스턴스 변수가 놓인다

실행할 클래스 코드가 놓인다

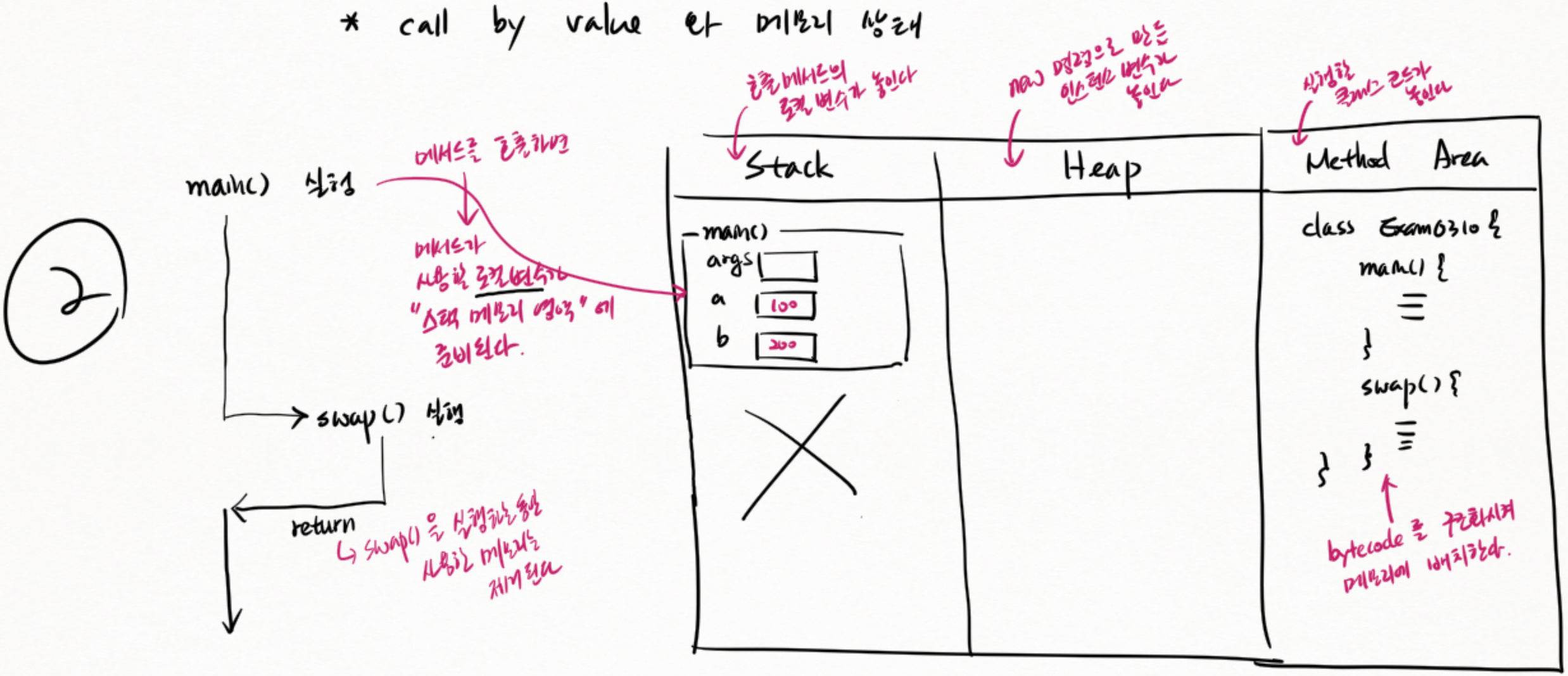
Stack	Heap	Method Area
main()		class Exam0310 {
args		main() {
a 100		≡
b 200		}
		swap() {
		≡
		} }

bytecode 를 구조화시켜 메모리에 배치한다.

* call by value 와 메모리 상태



* call by value 와 메모리 상태



* call by value 와 메모리 상태

3

main() 실행

swap() 실행

return

종료 끝

Stack	Heap	Method Area

✕		class Exam0310 {
		main() {
		≡
		}
		swap() {
		≡
		} }

종료 메서드의 로컬 변수가 놓인다

new 명령으로 만든 인스턴스 변수가 놓인다

실행할 클래스 코드가 놓인다

bytecode 를 구조화시켜 메모리에 배치한다.

* call by value 와 메모리 상태

4

main() 실행

swap() 실행

return

프로그램

→ JVM 실행 종료

JVM이 사용했던 메모리는 OS의 의무

가능해 회수된다.

해제