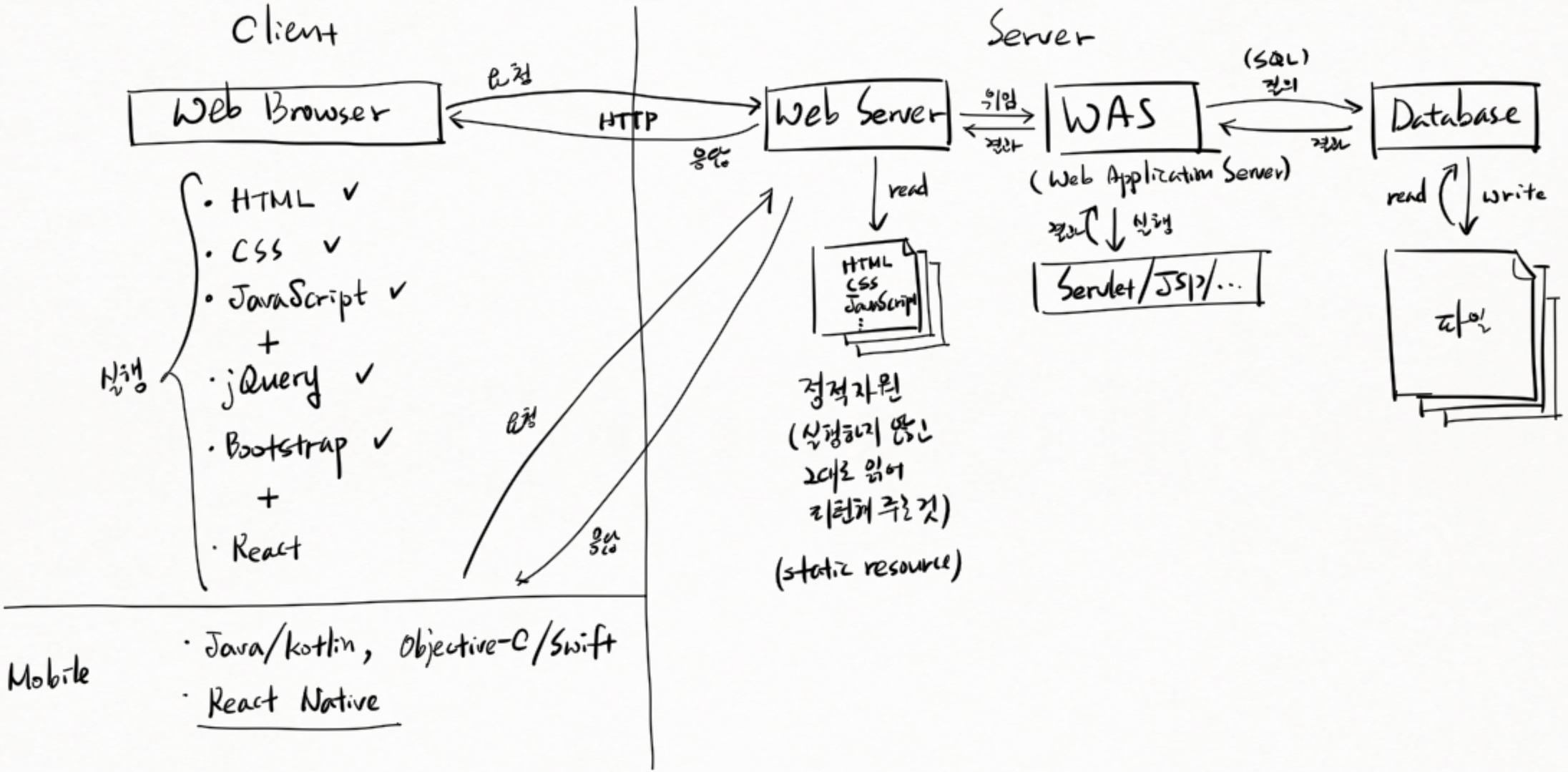
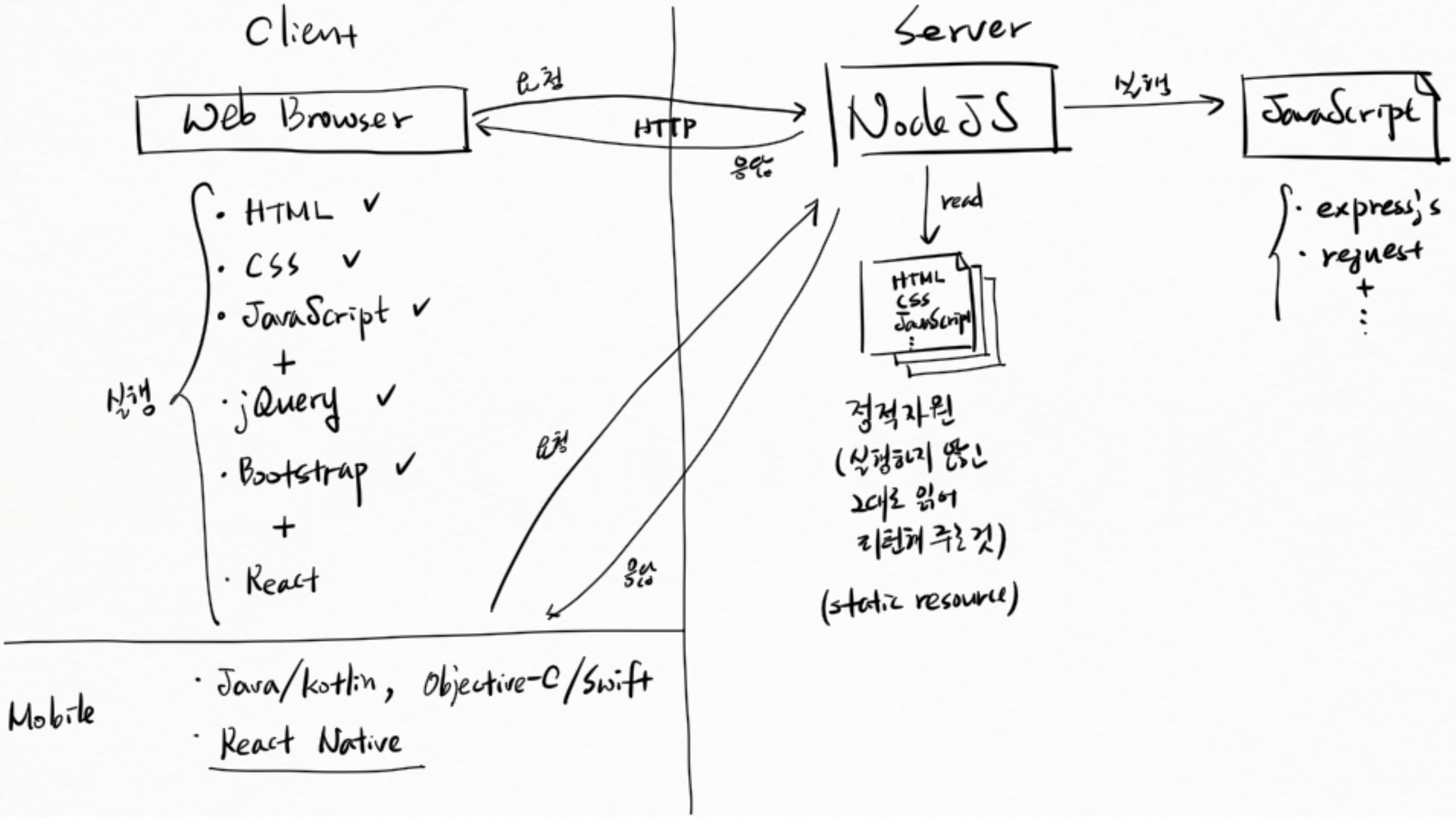
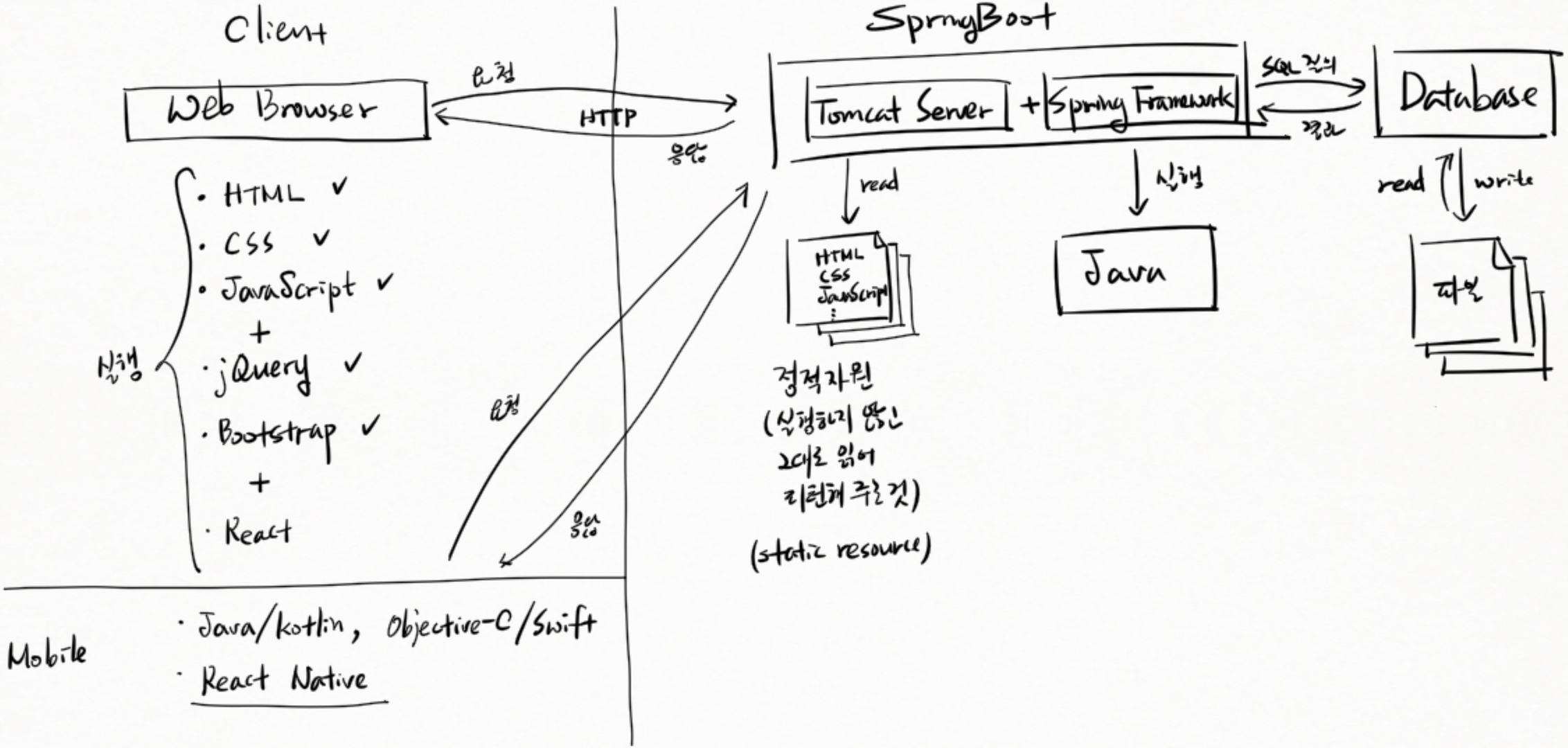
* Web Application Architecture by Java



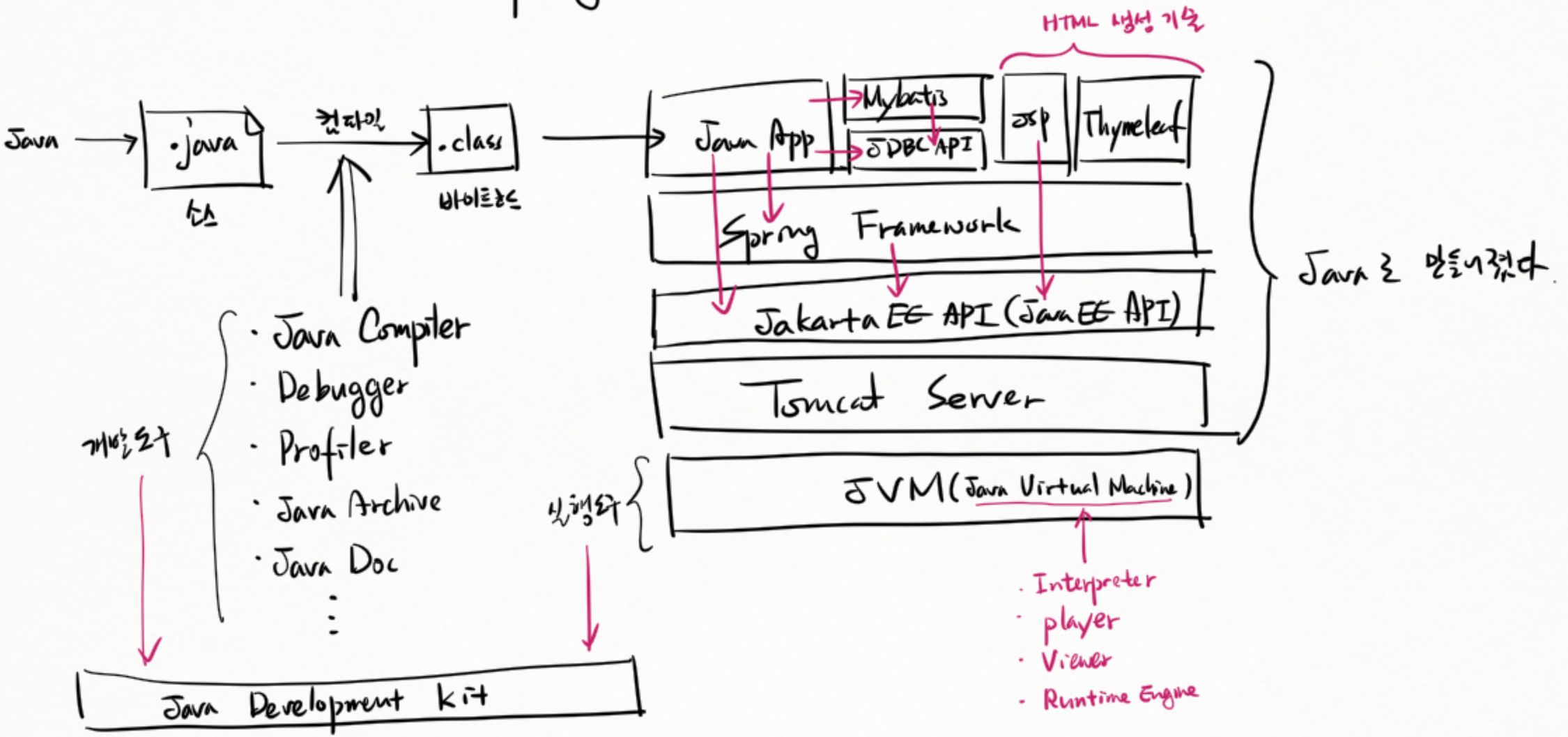
* Node.js Web Application Architecture



* 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

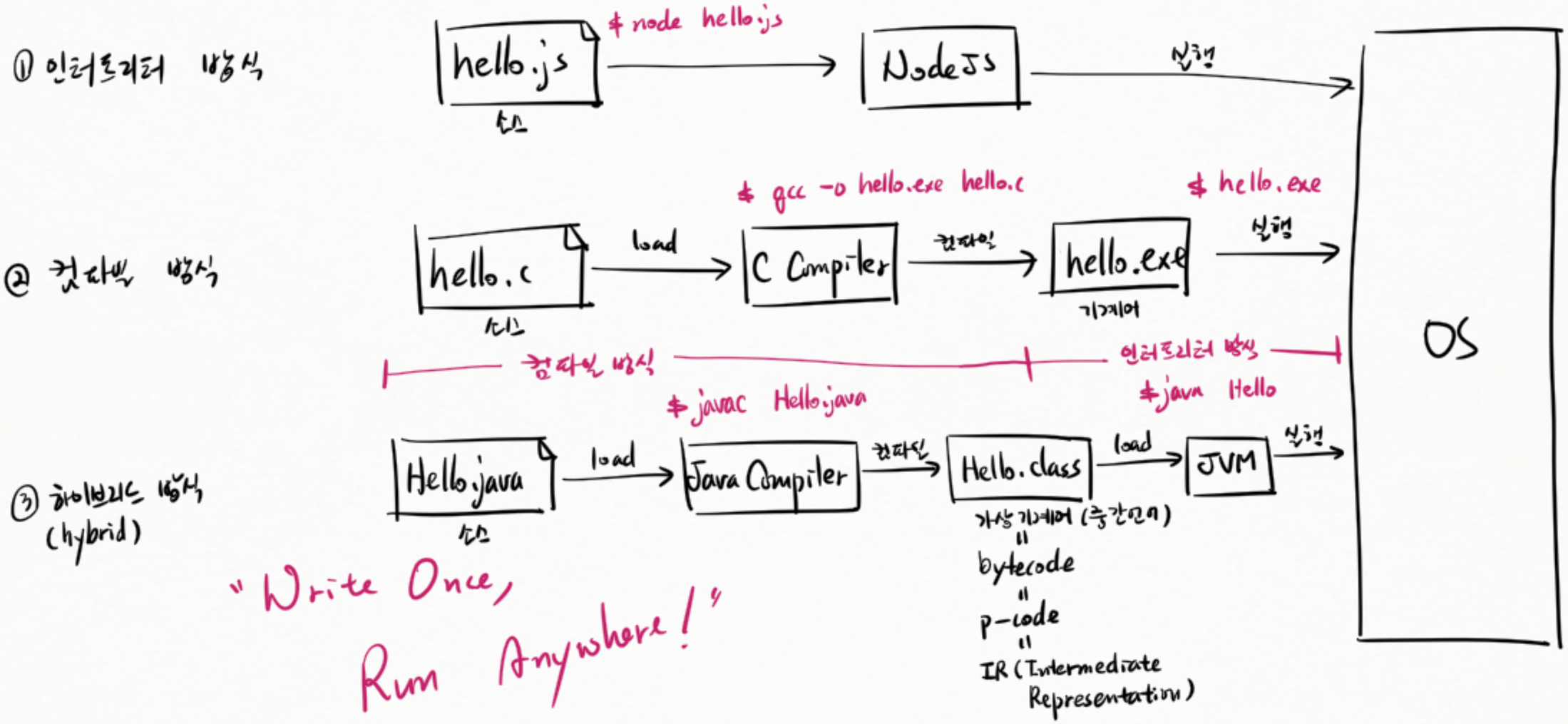
↳ · 소스 편집
· JDK를 이용한 빌드 작업, 테스팅 등
⋮
개발 전반의 작업을 수행할 수 있도록 도와주는 프로그램

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.

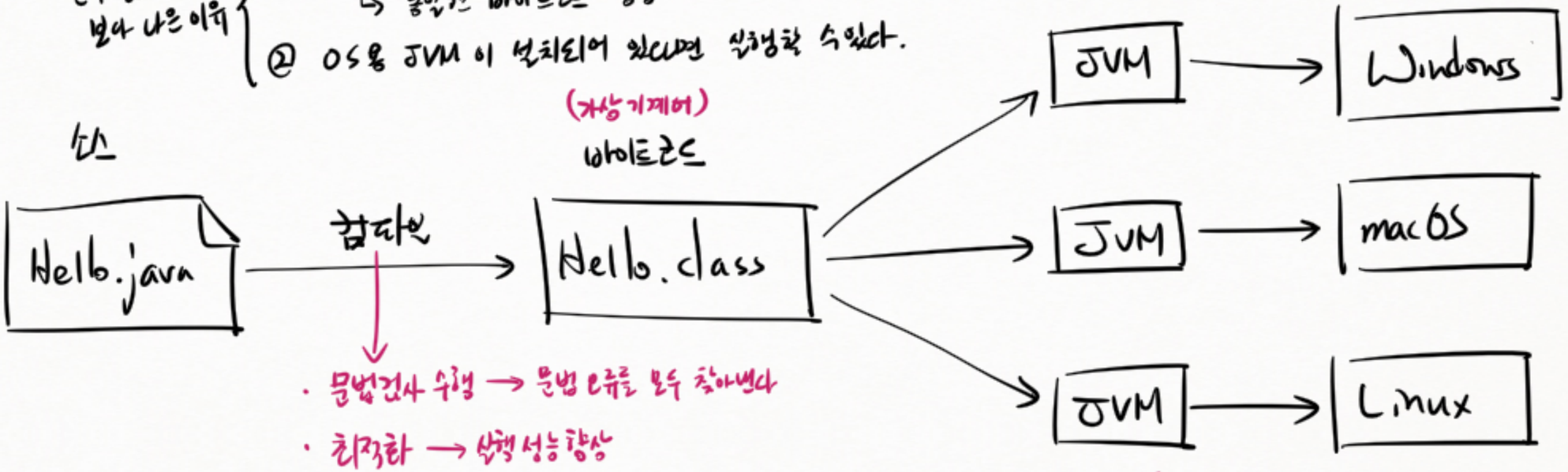
* App. 개발 및 실행



"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

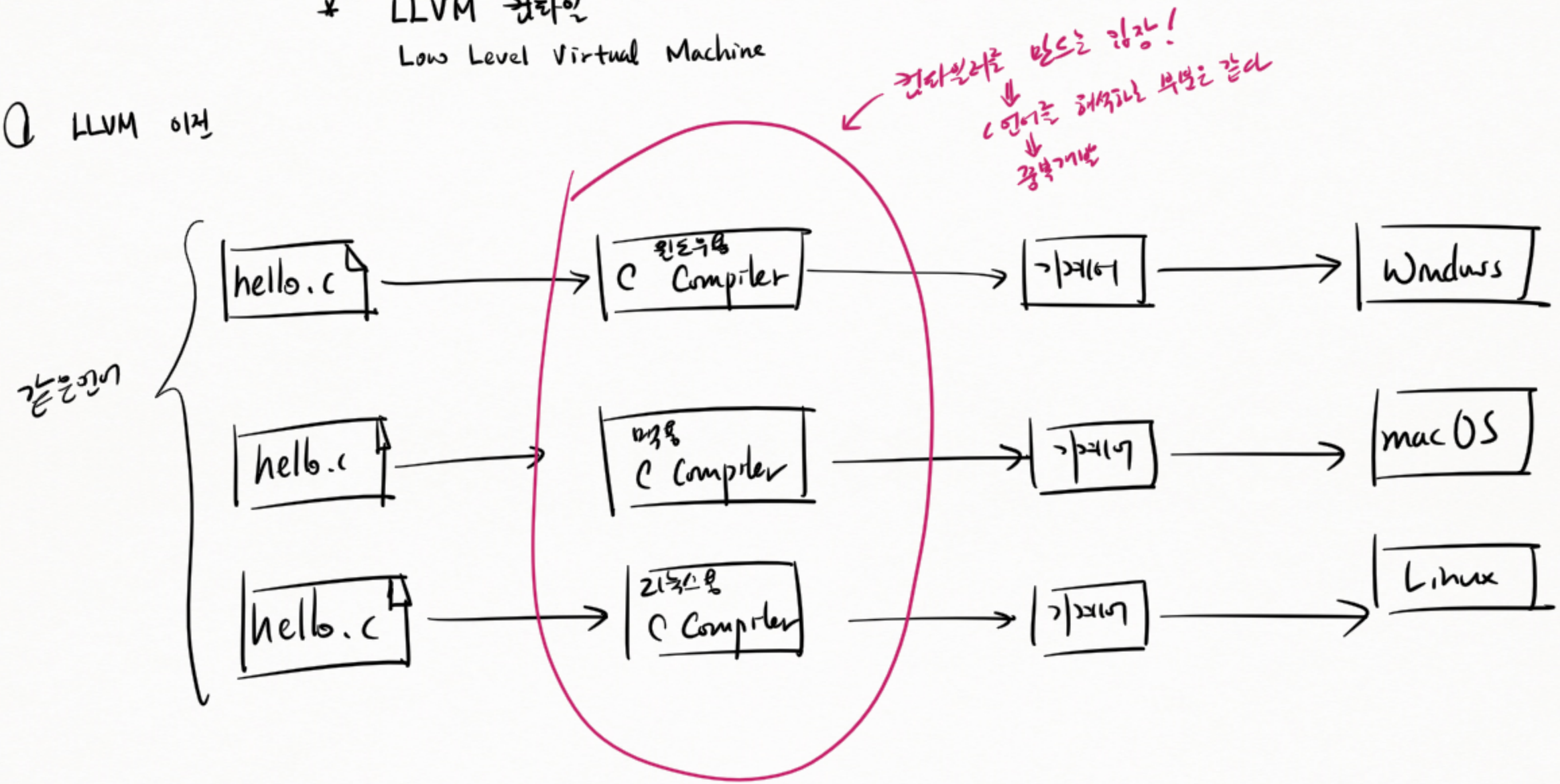
순수 인터프리터 방식보다 나은 이유
③ 컴파일 과정에 문법의 오류를 모두 찾아낸다
④ 완전 기계어가 아니라 기계어에 가까운 언어로 변환된 명령을 실행하기 때문에 소스에 작성된 명령을 실행하는 것보다 실행속도가 빠르다.

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

* LLVM 컴파일

Low Level Virtual Machine

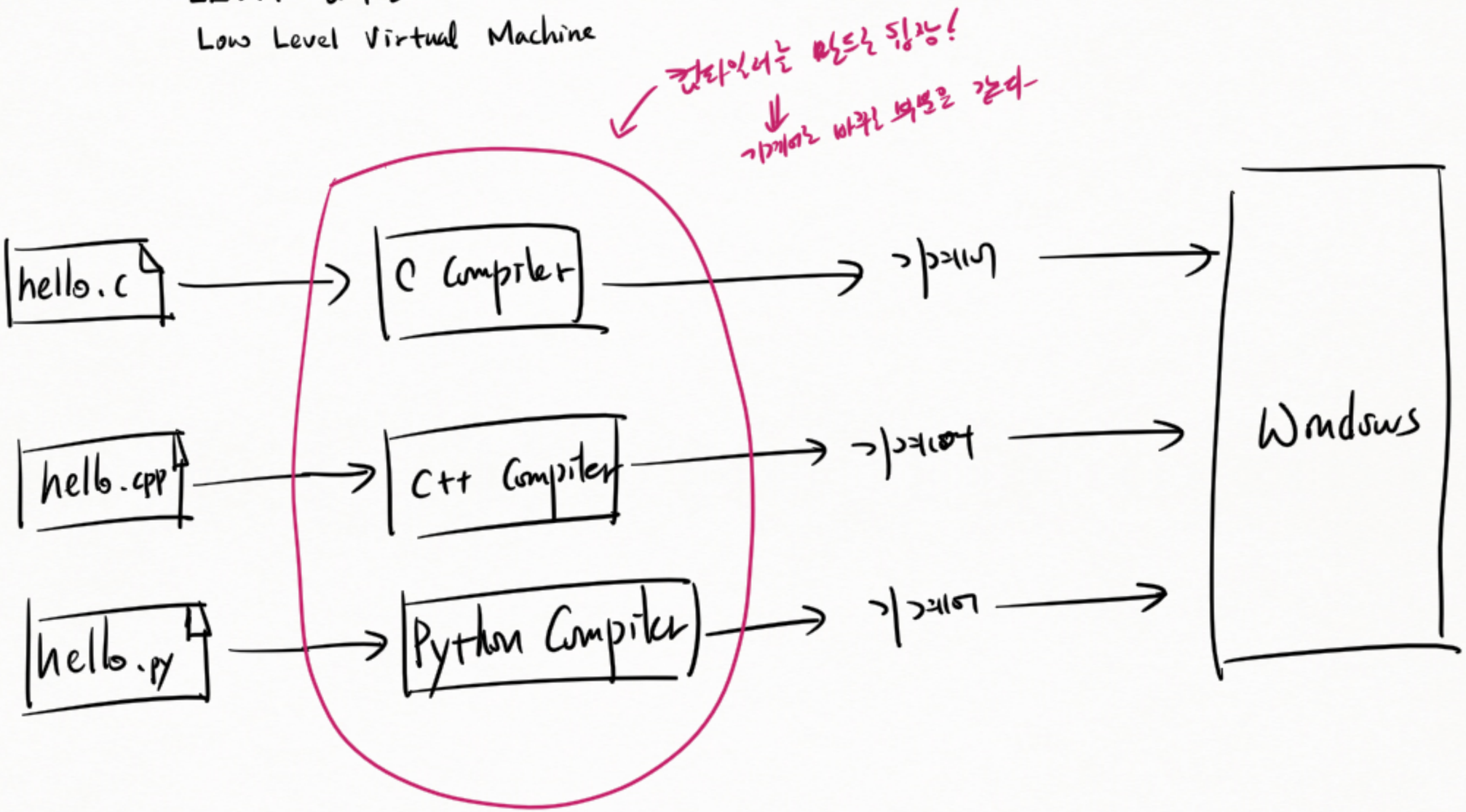
Q LLVM 이전



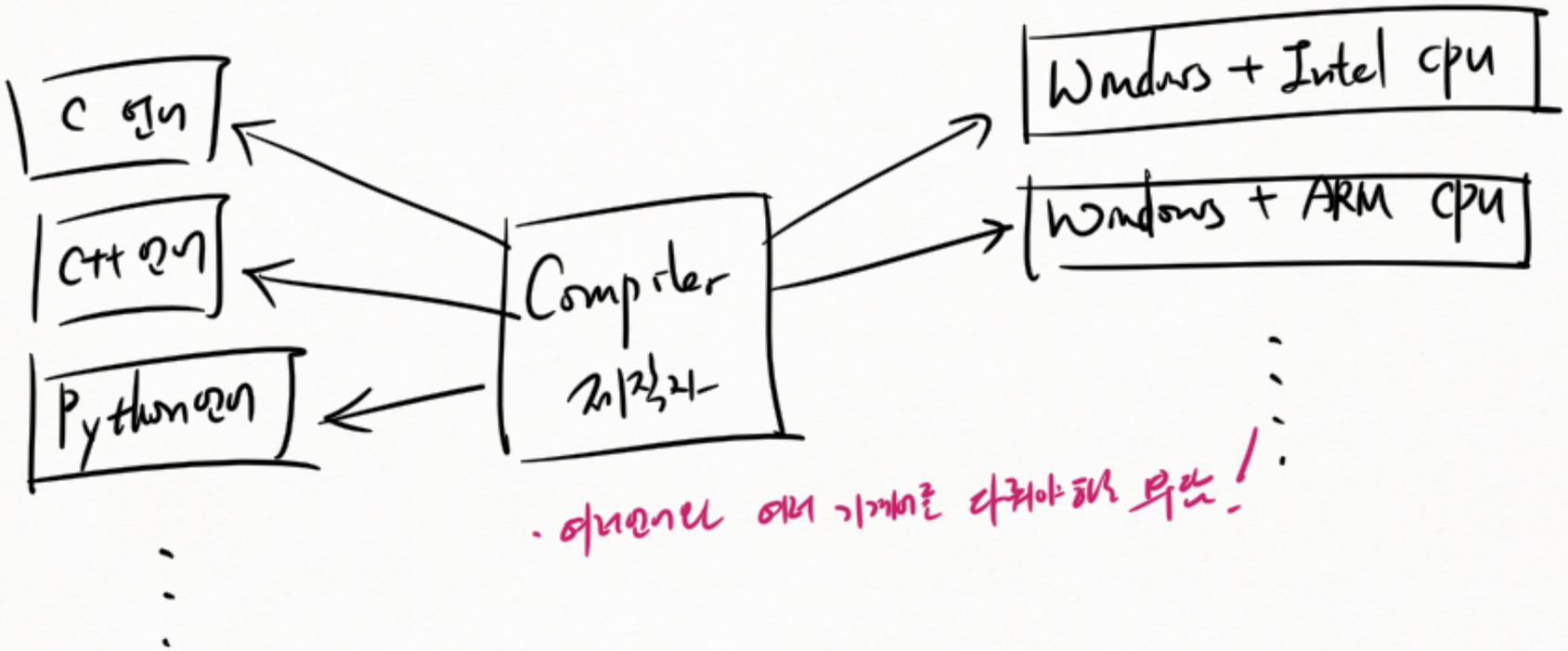
* LLVM 컴파일

Low Level Virtual Machine

① LLVM 이전



* 기존 방식의 문제점

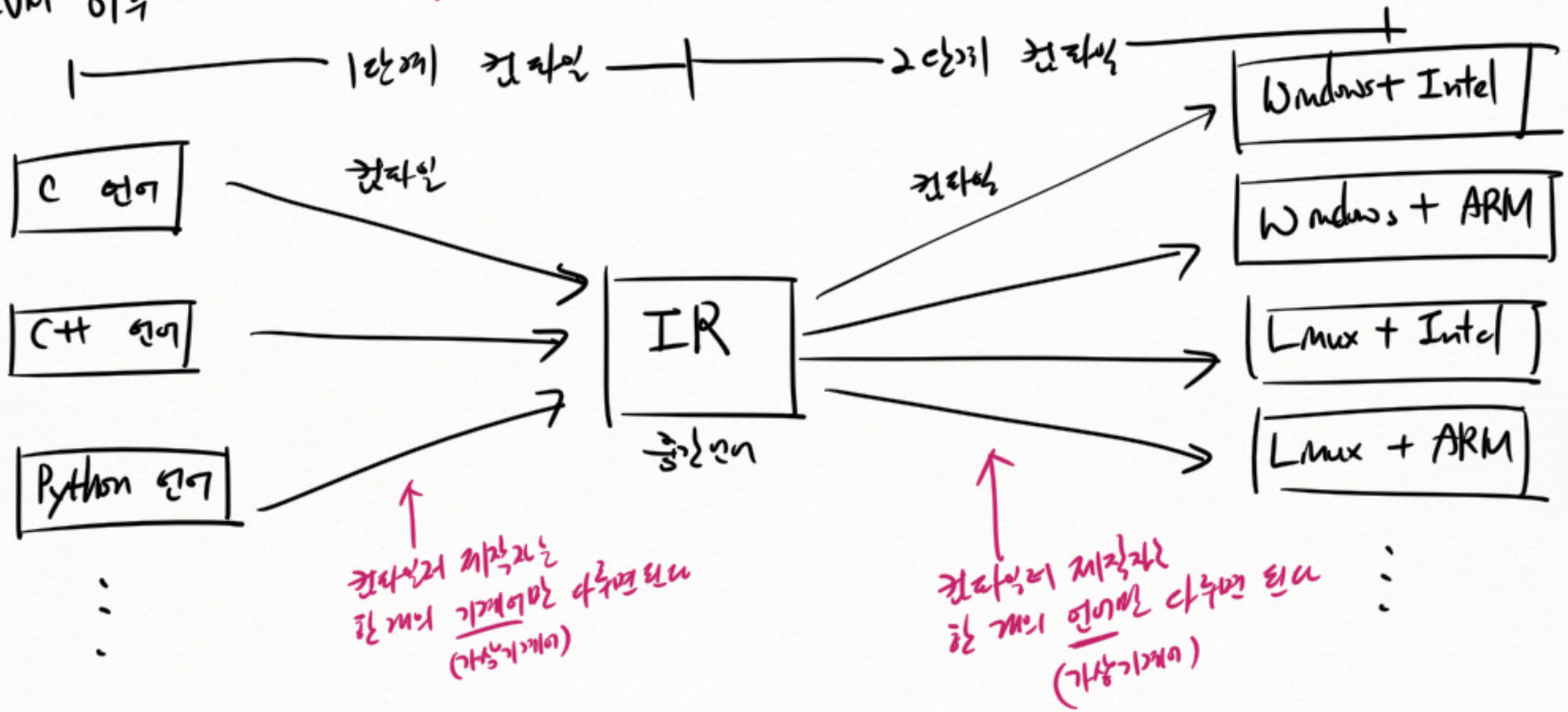


· 어떤언어냐 어떤 기계를 다뤄야 하는 문제!

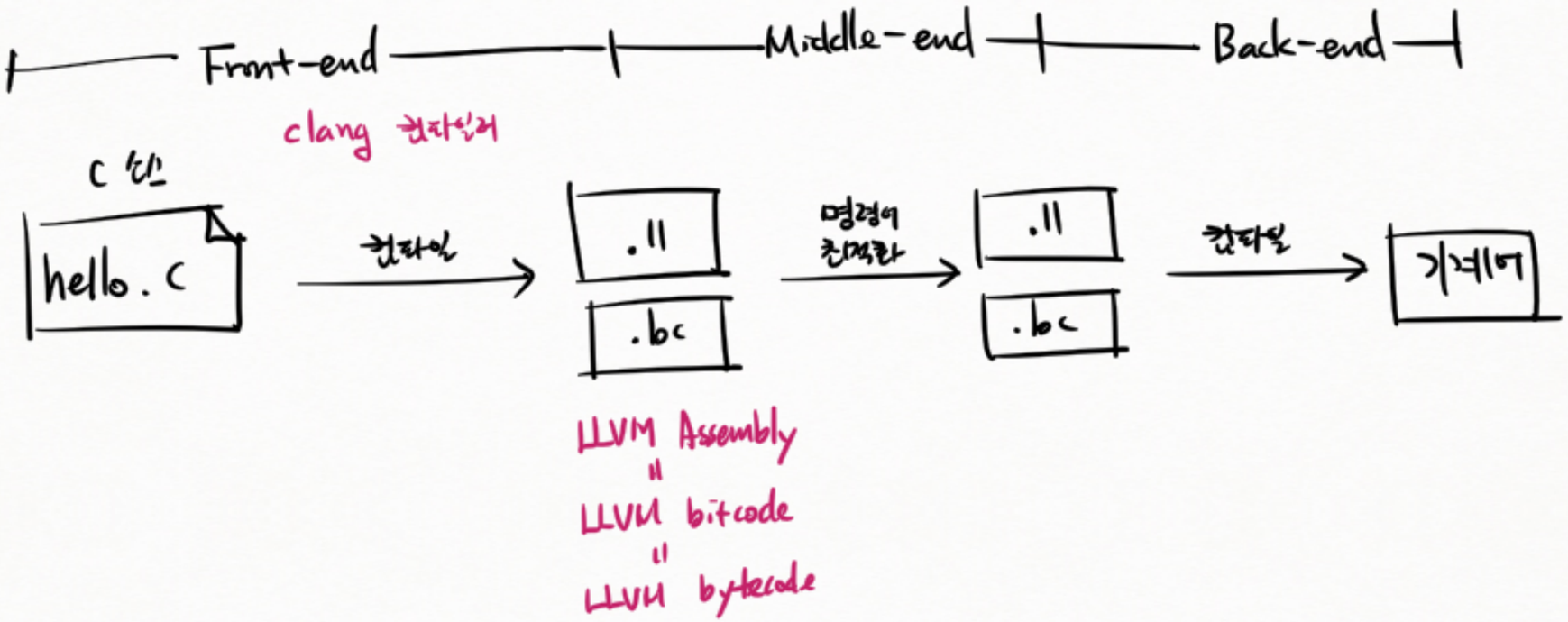
* LLVM 방식을 사용하는 이유

↳ 새 프로그래밍 언어나 컴파일러를 만들기 쉽다.

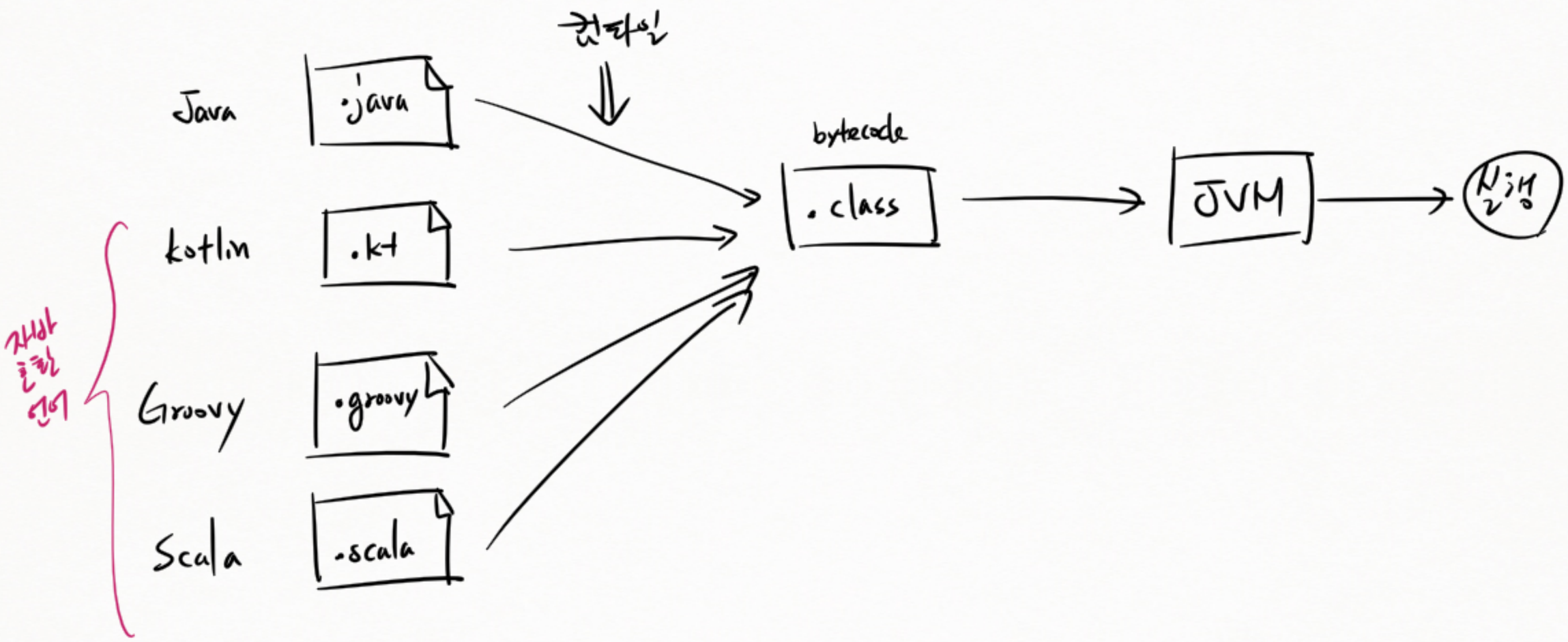
② LLVM 이후



* LLVM 컴파일 과정

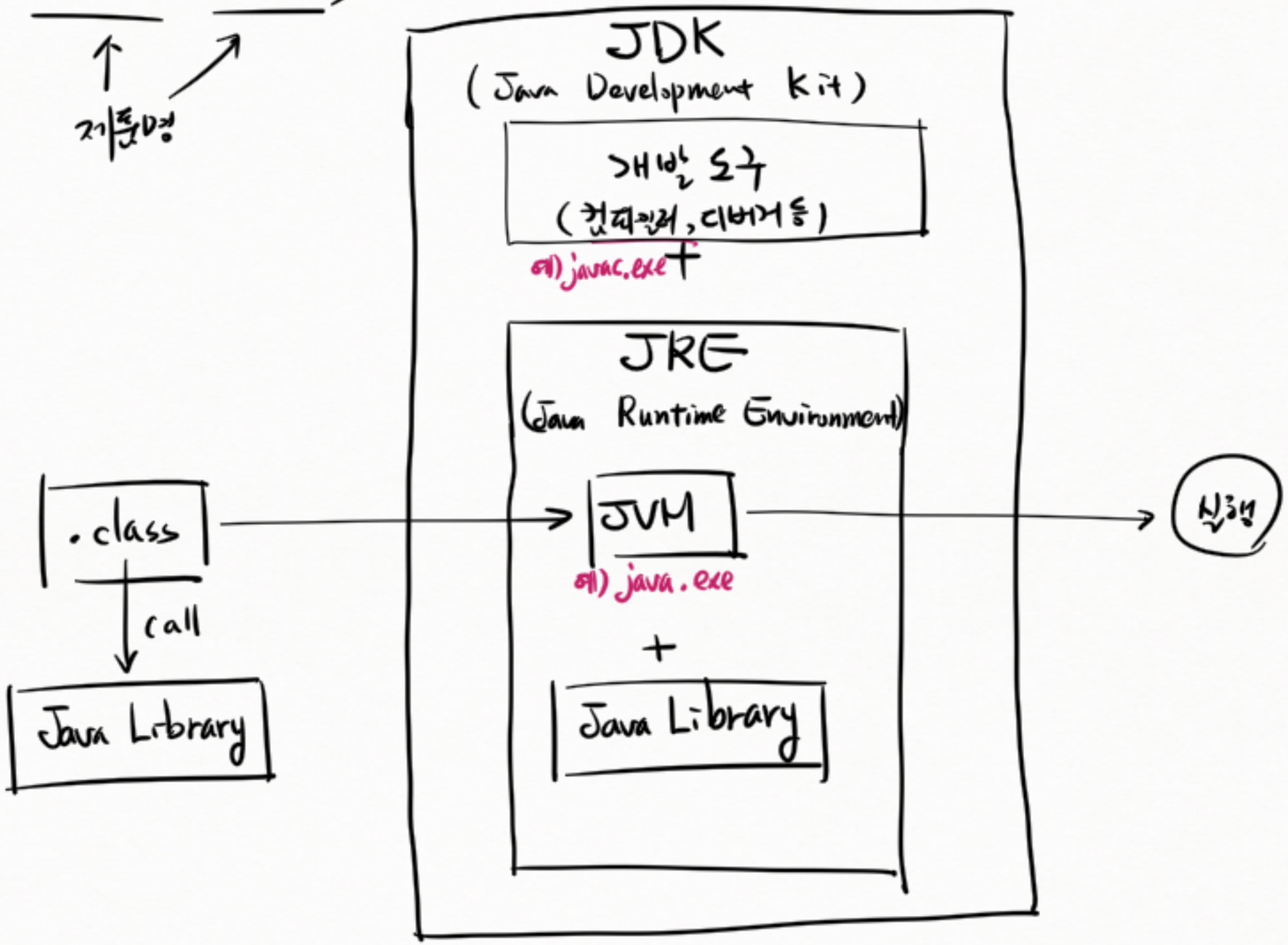

Front-end
Middle-end
Back-end
clang 컴파일러
C 언
hello.c
컴파일
.ll
.bc
LLVM Assembly
=
LLVM bitcode
=
LLVM bytecode
명령어 최적화
.ll
.bc
컴파일
기계어

* 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
```

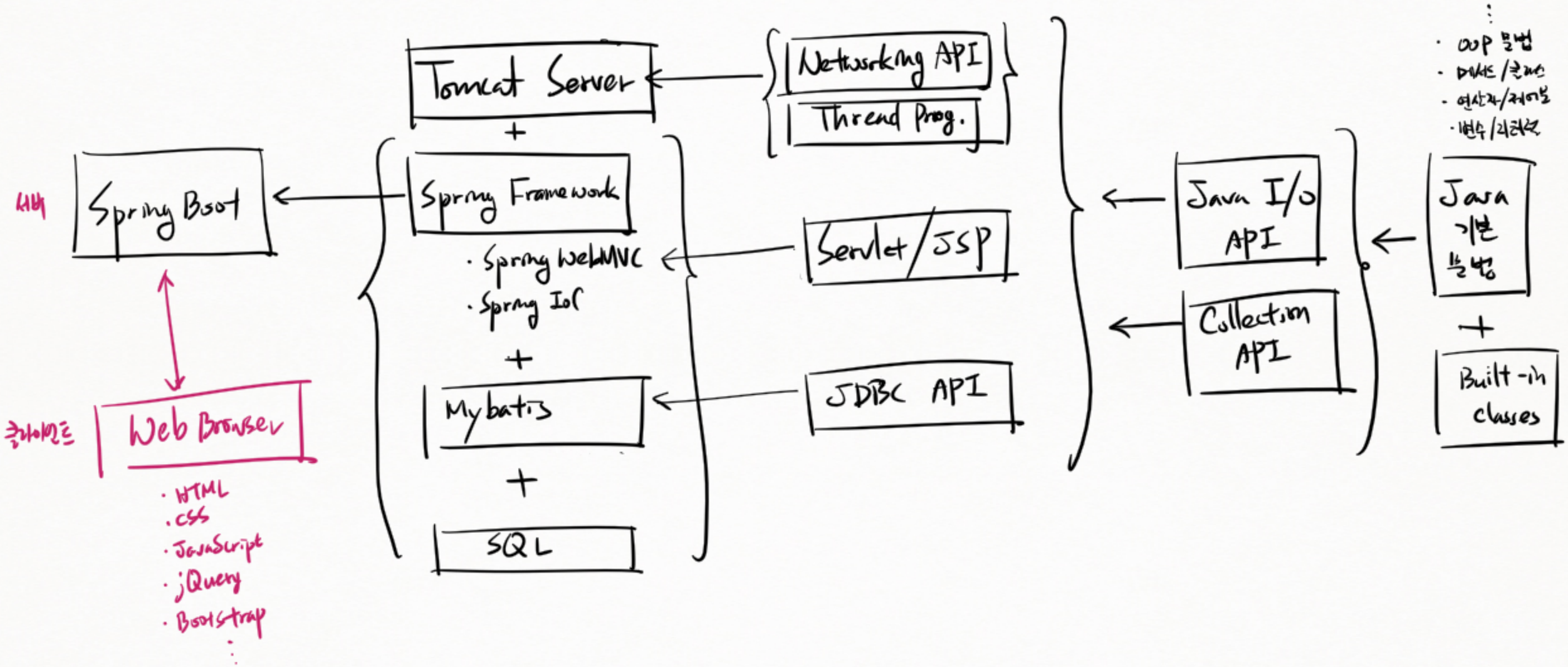
```
boot-app/                  ← 프로젝트 폴더
├ app/                     ← 하위 프로젝트. 메인 프로젝트
│  ├ src/
│  │  ├ main/
│  │  │  ├ java/           ← Java 소스 파일을 두는 폴더
│  │  │  └ resources/      ← 기타 파일을 두는 폴더
│  │  │                      (.properties, .xml, .html 등)
│  │  └ test/
│  │     ├ java/           ← 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

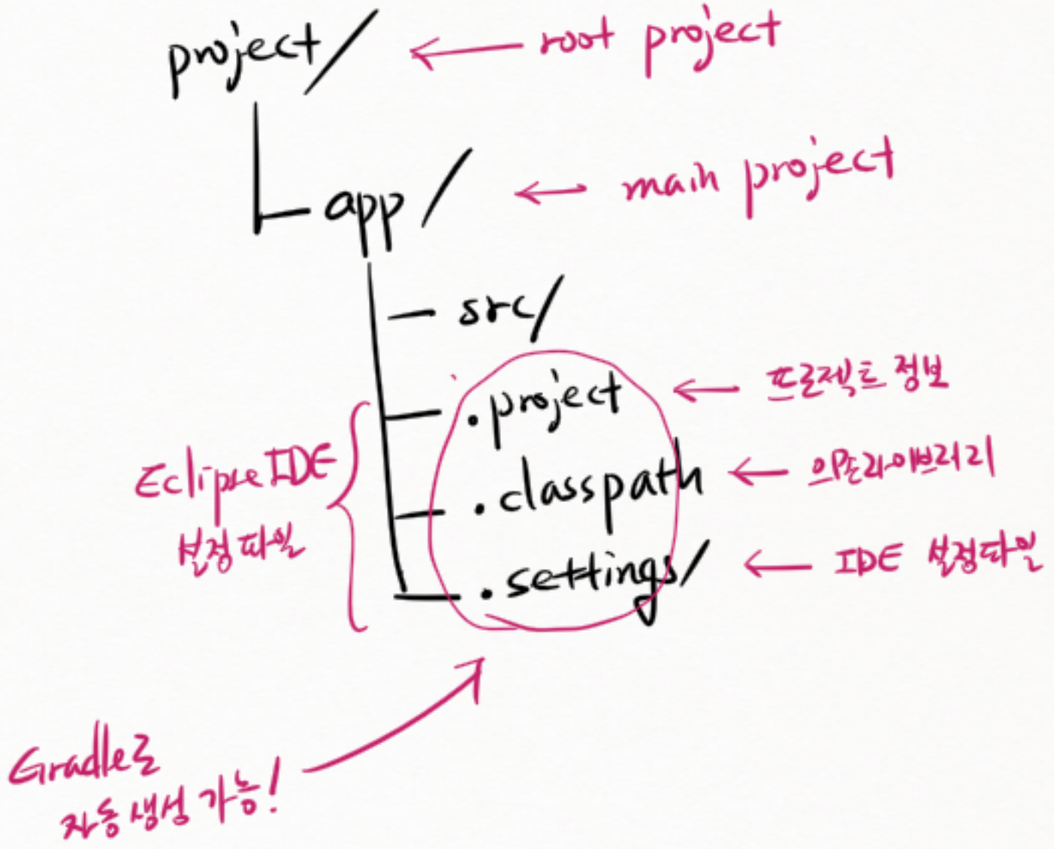
```
~/git/bitcamp-ncp/          ← Git Repository
    |— boot-app/            ← project 폴더
        |— app/             ← sub project (main)
            |— src/         ← 소스 폴더
            |— build.gradle ← Gradle 빌드 스크립트 파일
        |— settings.gradle  ← Gradle 빌드 보조 스크립트 파일
        |— gradle/          ← 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

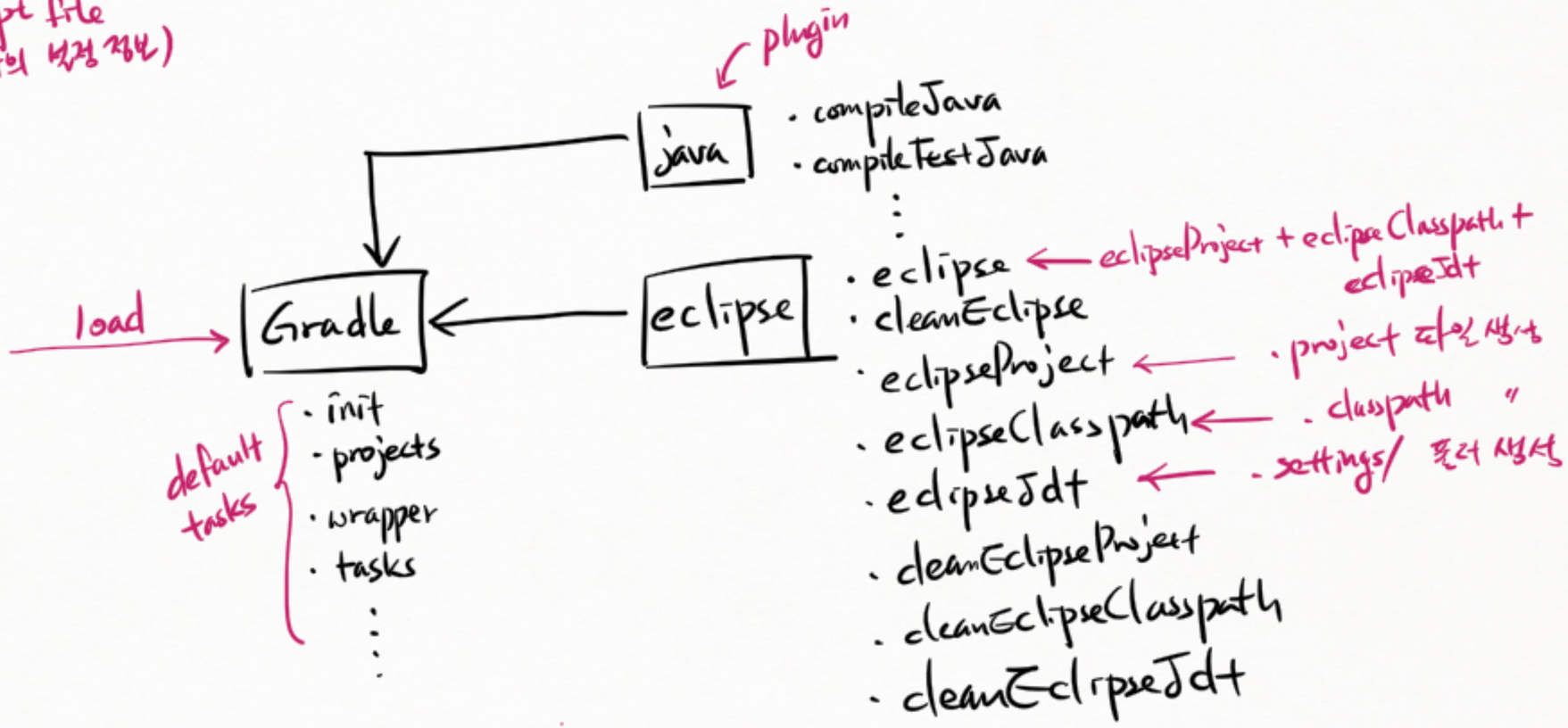


* Gradle 빌드 도구

[build.gradle]

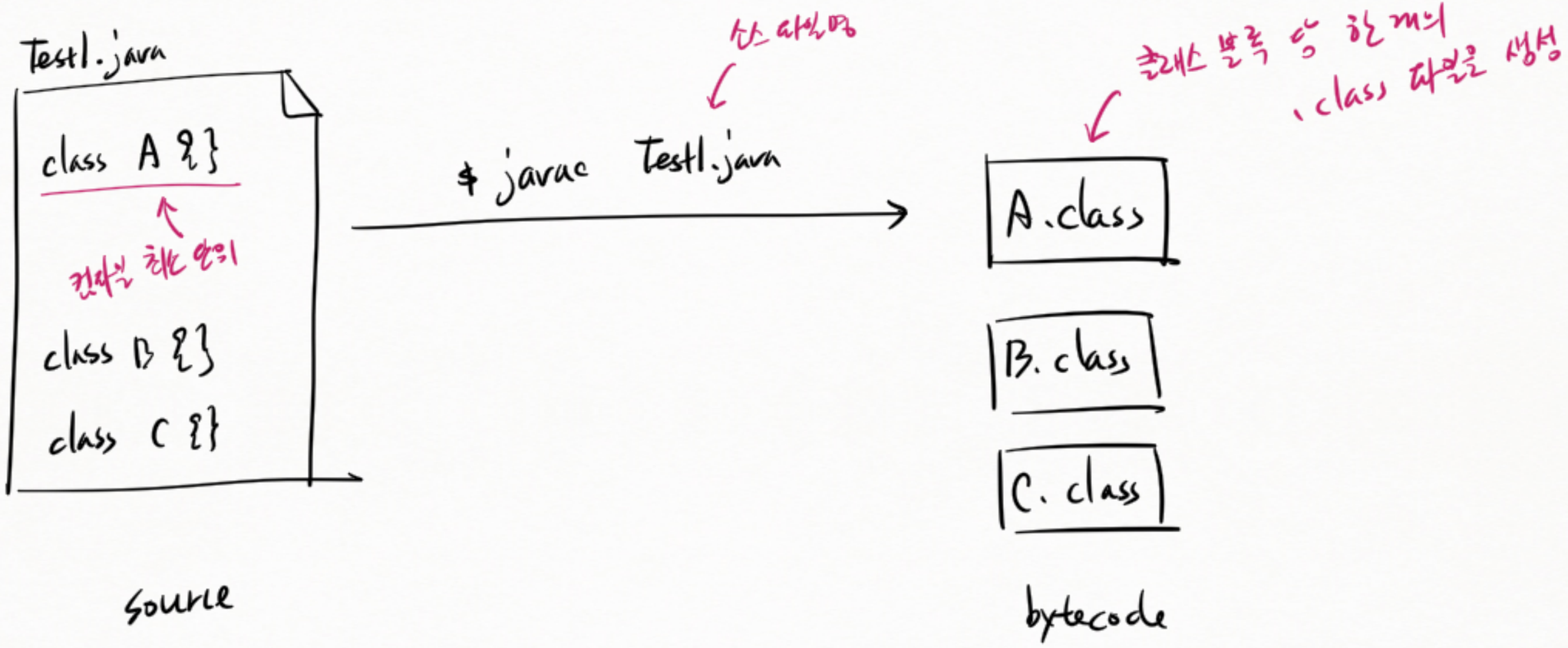
build script file
(빌드 도구의 설정 정보)

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



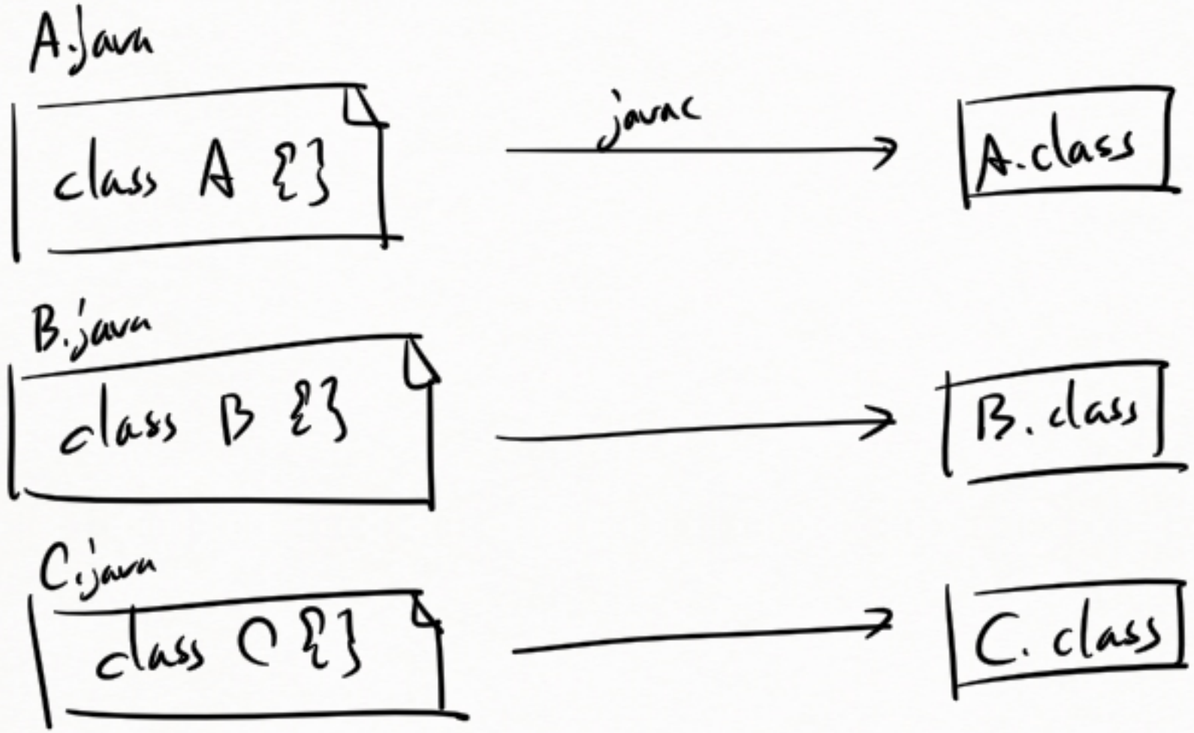
Java 문법

* .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 {}
```

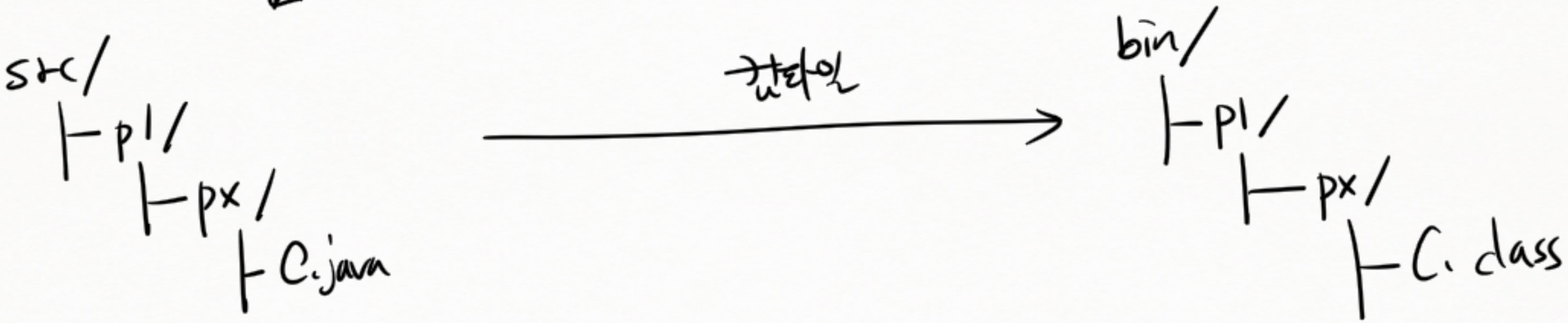
$ javac -d bin

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

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

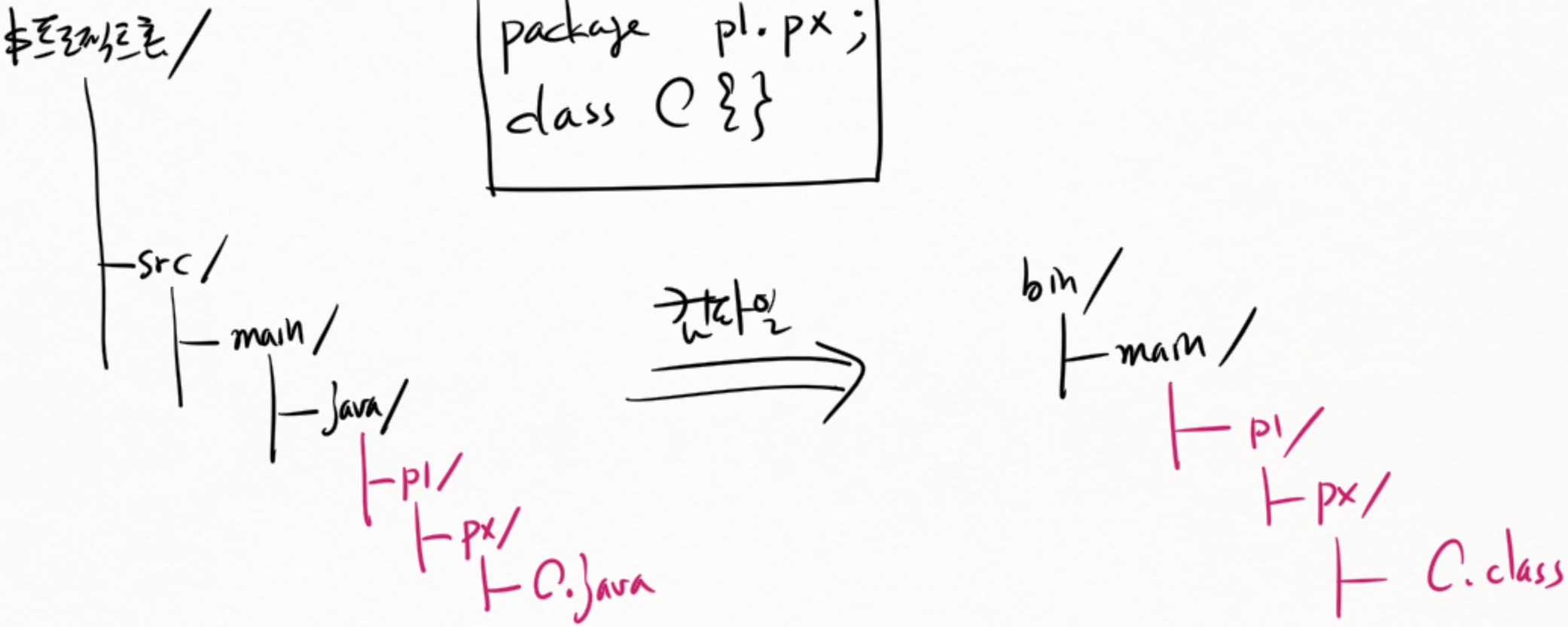
* 패키지와 디렉토리

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

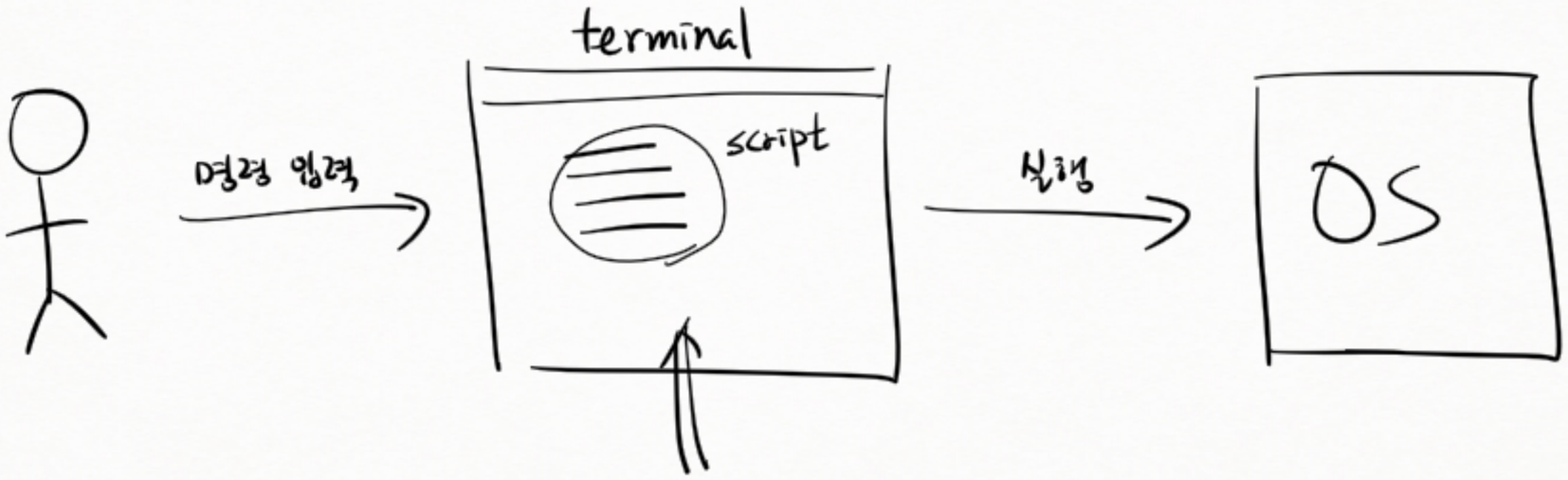


* Maven 프로젝트의 폴더 디렉토리 구조로 소스 파일 관리

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



* shell script



Windows : Batch script (.bat), PowerShell

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

* 클래스 블록의 공개 여부

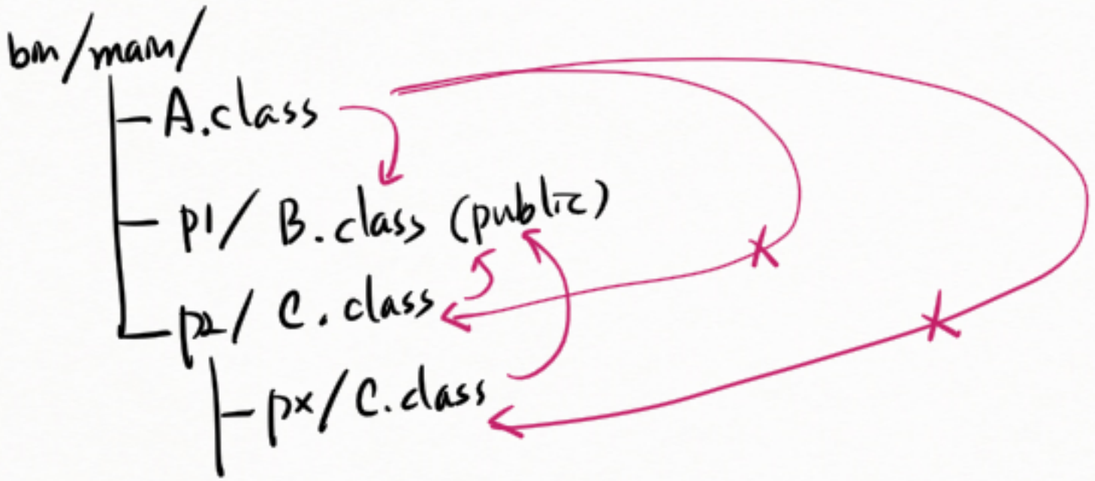
A.java

```
public class A { }
```

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

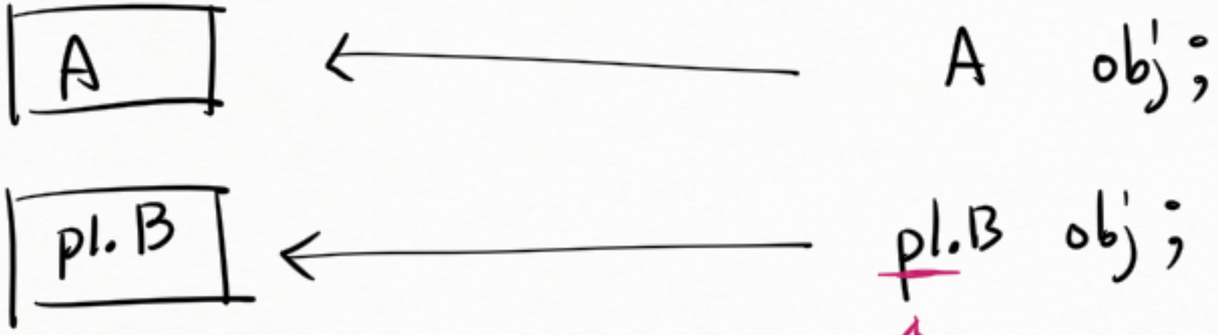
↳ 공개 클래스의 파일명은 반드시 클래스명과 같아야 한다.

* 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

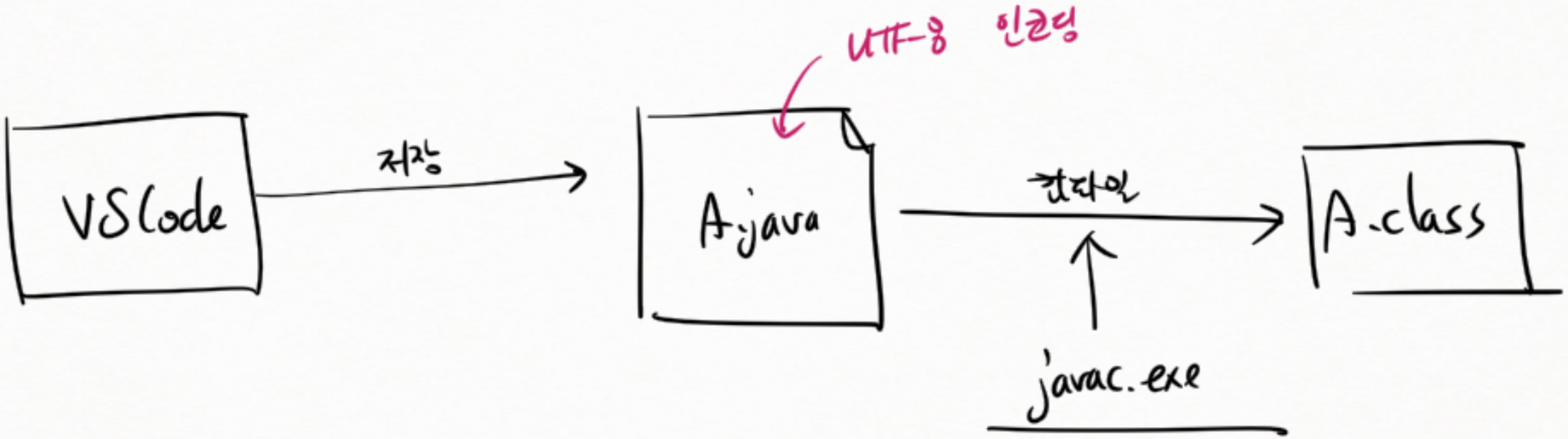
↳ 자바 컴파일러 와 JVM은 클래스를 찾을 이 옵션의 경로를 따라간다.

```
$ 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.

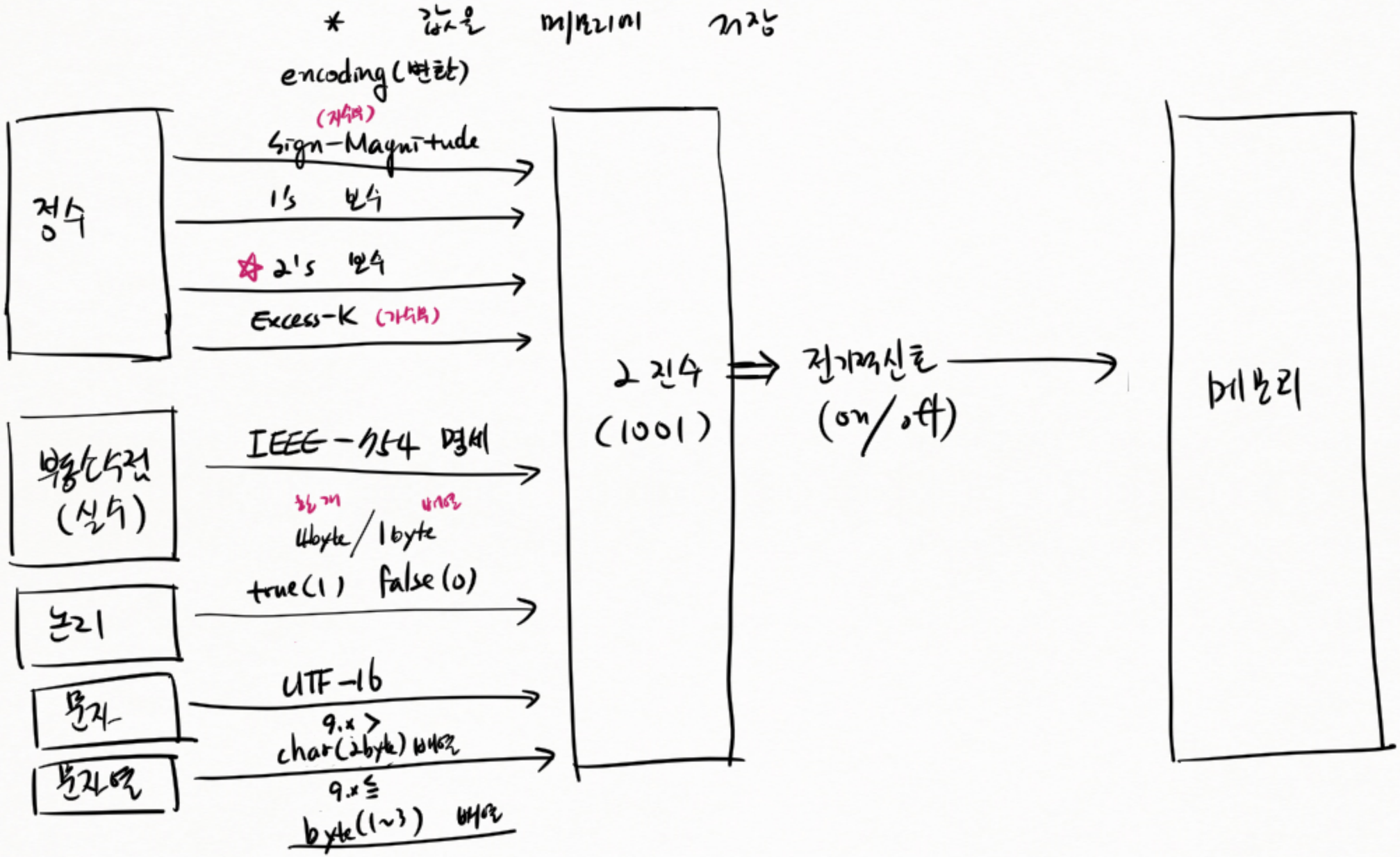
Literal

* 데이터 타입 & 리터럴

자바 기본 데이터 타입 (Primitive Data Type)

- 정수
  - → 4byte 리터럴 (int) 100
  - → 8 byte 리터럴 (long) 100L 100l
- 부동소수점
  - → 4byte 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 + bias값 $= 3 + 127 = 130 =$ 10000010

bias 값
‖
Excess-k 값

$$\boxed{\begin{aligned}&2735.2705\ (10진수)\\ &= 2.7352705 \times 10^3\end{aligned}}$$

128	64	32	16	8	4	2	1

부호비트	지수부	가수부
0	1 0 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 0

(4byte)
(4B)
0x41460000 — 16진수
C1460000
(음수)