

프로그래밍 언어 종류와 구분

화생방 스터디
강수아

목차

1. 개발 편의성에 의한 분류

A. 기계어

B. 어셈블리어

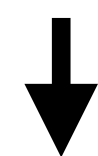
I. 명령어

2. 실행 방식에 의한 분류

3. 번역기

사람

이해 가능 여부



개발 편의성에 의한 분류

Low-Level Language

기계어

```
1000 1011 0100 0101 1111 1000
```

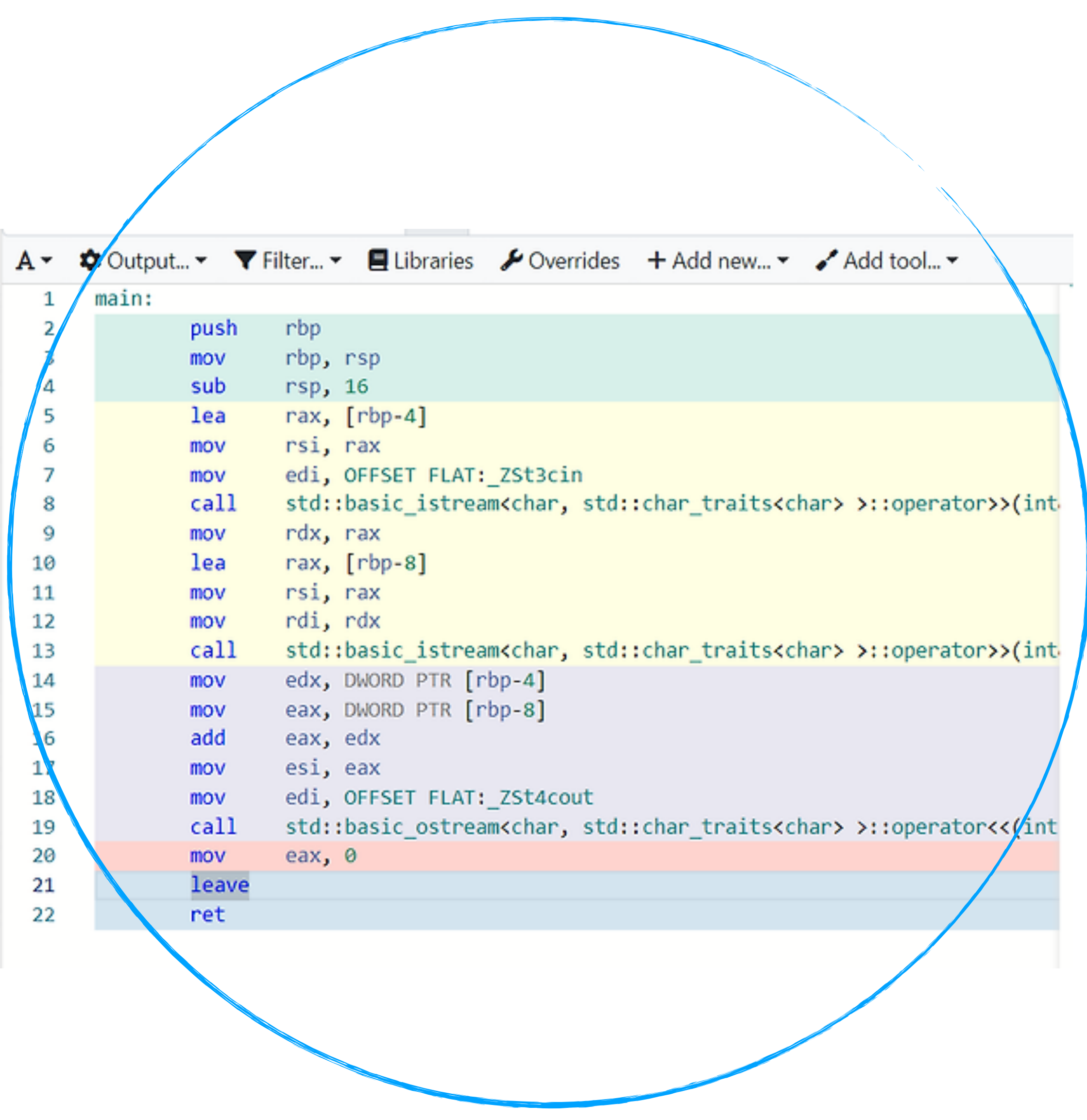
```
1000 0011 1100 0100 0000 1100
```

```
0000 0011 0100 0101 1111 1100
```

Low-Level Language

어셈블리어

```
1  #include <iostream>
2
3  int main()
4  {
5      int x, y;
6
7      std::cin >> x >> y;
8      std::cout << x + y;
9
10     return 0;
11 }
```

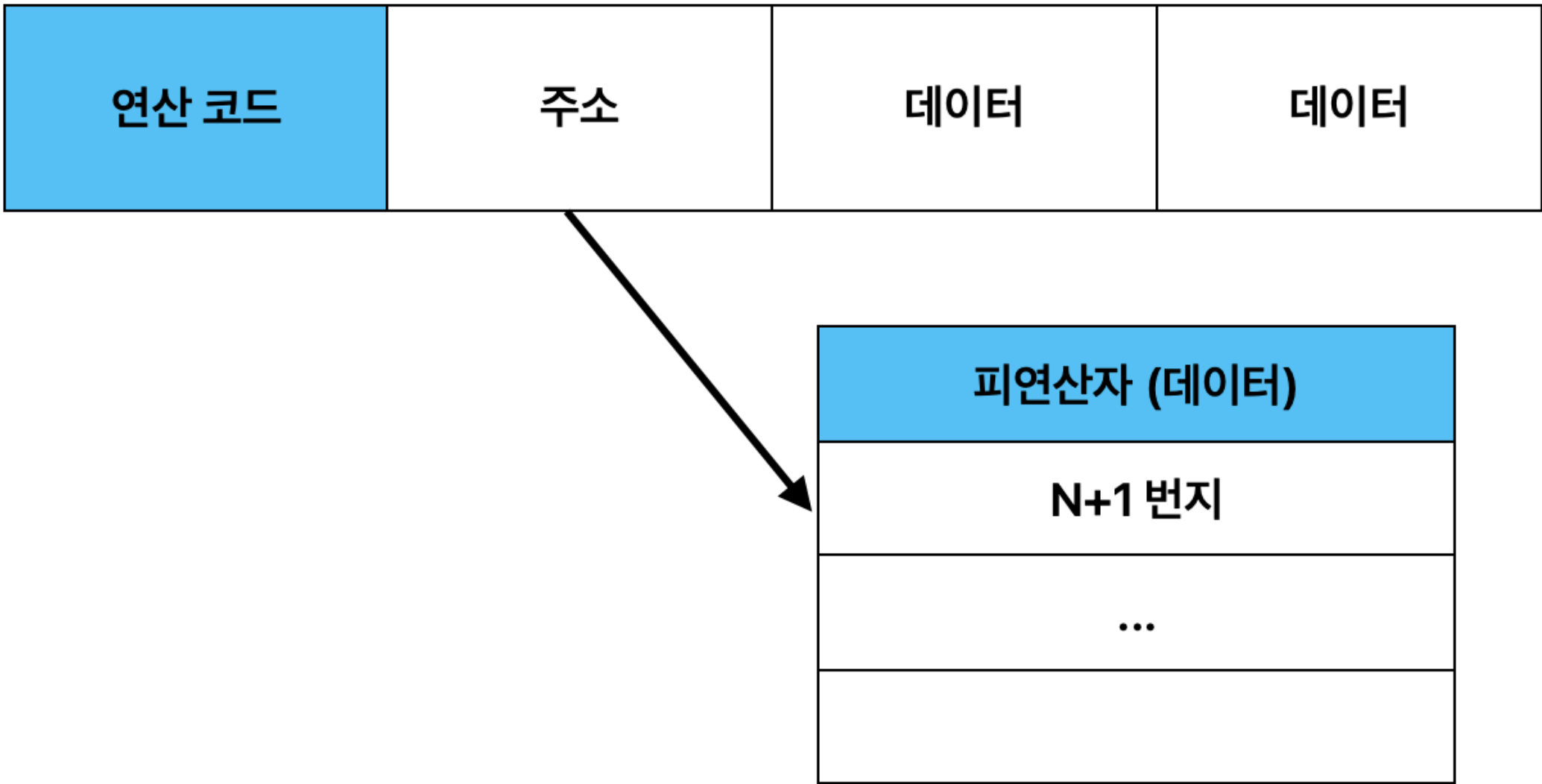
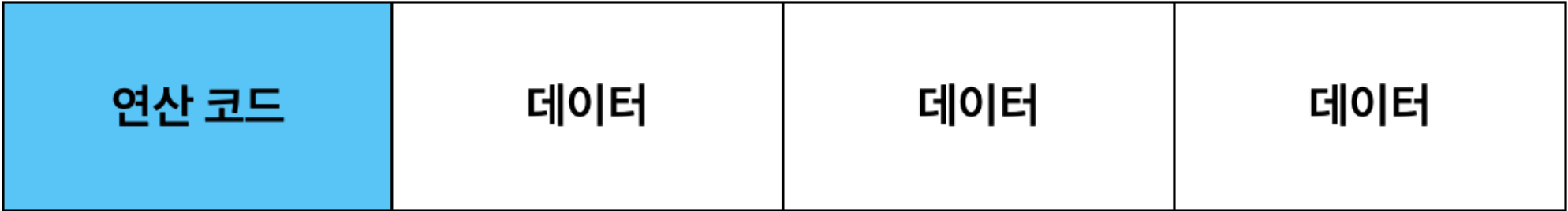


```
A ▾ ⚙ Output... ▾ ▼ Filter... ▾ 📖 Libraries 🔧 Overrides + Add new... ▾ 🛠 Add tool... ▾
1  main:
2      push    rbp
3      mov     rbp, rsp
4      sub     rsp, 16
5      lea     rax, [rbp-4]
6      mov     rsi, rax
7      mov     edi, OFFSET FLAT:_ZSt3cin
8      call    std::basic_istream<char, std::char_traits<char> >::operator>>(int
9      mov     rdx, rax
10     lea     rax, [rbp-8]
11     mov     rsi, rax
12     mov     rdi, rdx
13     call    std::basic_istream<char, std::char_traits<char> >::operator>>(int
14     mov     edx, DWORD PTR [rbp-4]
15     mov     eax, DWORD PTR [rbp-8]
16     add     eax, edx
17     mov     esi, eax
18     mov     edi, OFFSET FLAT:_ZSt4cout
19     call    std::basic_ostream<char, std::char_traits<char> >::operator<<(int
20     mov     eax, 0
21     leave
22     ret
```

Low-Level Language

명령어의 구조

'무엇을 대상으로, 어떤 작동을 수행하라'



Low-Level Language

명령어의 구조


데이터 전송

산술/논리 연산

제어 흐름 변경

입출력 제어

Etc.

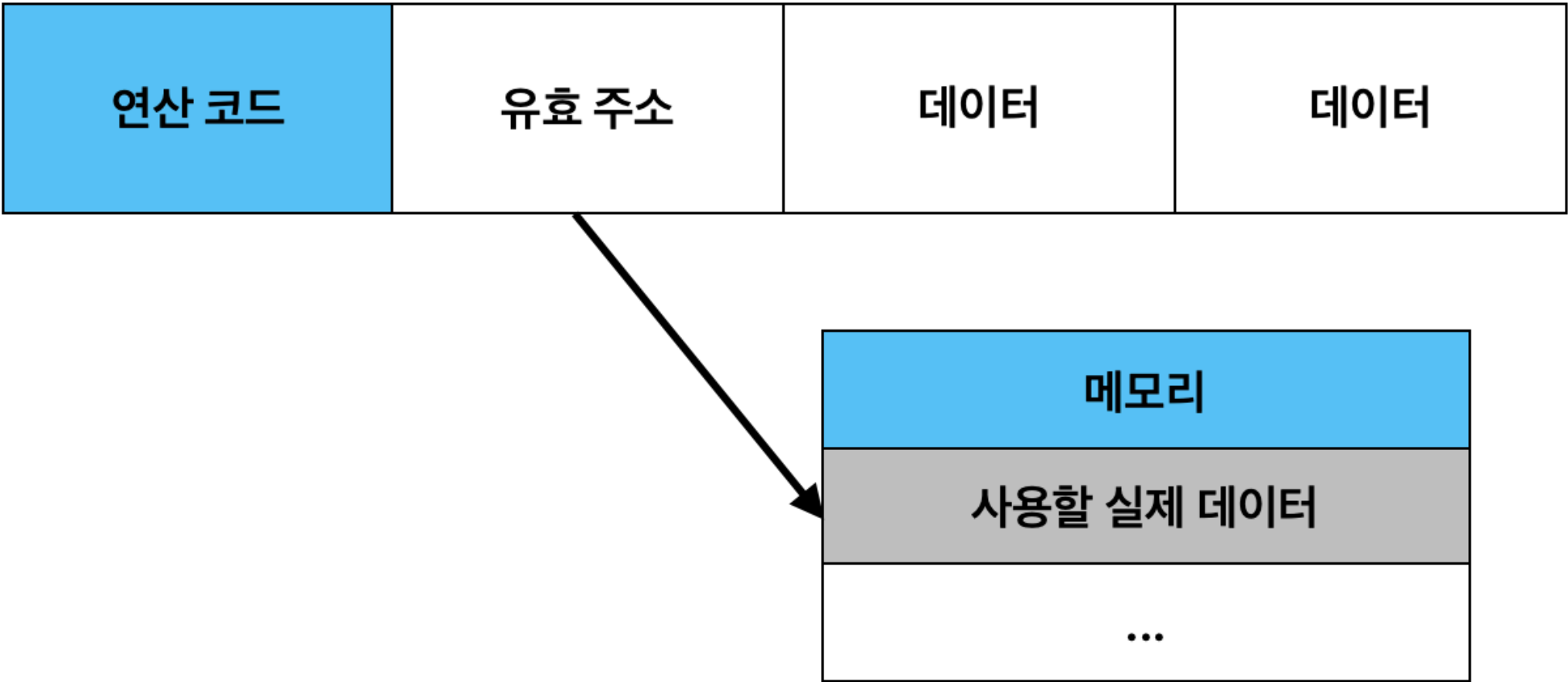


```
push    rbp
mov     rbp, rsp
sub     rsp, 16
lea     rax, [rbp-4]
mov     rsi, rax
mov     edi, OFFSET FLAT:_ZSt3cin
call    std::basic_istream<char, std::char_traits<char> >::operator>>(int&)
mov     rdx, rax
lea     rax, [rbp-8]
mov     rsi, rax
mov     rdi, rdx
call    std::basic_istream<char, std::char_traits<char> >::operator>>(int&)
mov     edx, DWORD PTR [rbp-4]
mov     eax, DWORD PTR [rbp-8]
add     eax, edx
mov     esi, eax
mov     edi, OFFSET FLAT:_ZSt4cout
call    std::basic_ostream<char, std::char_traits<char> >::operator<<(int)
mov     eax, 0
leave
ret
```

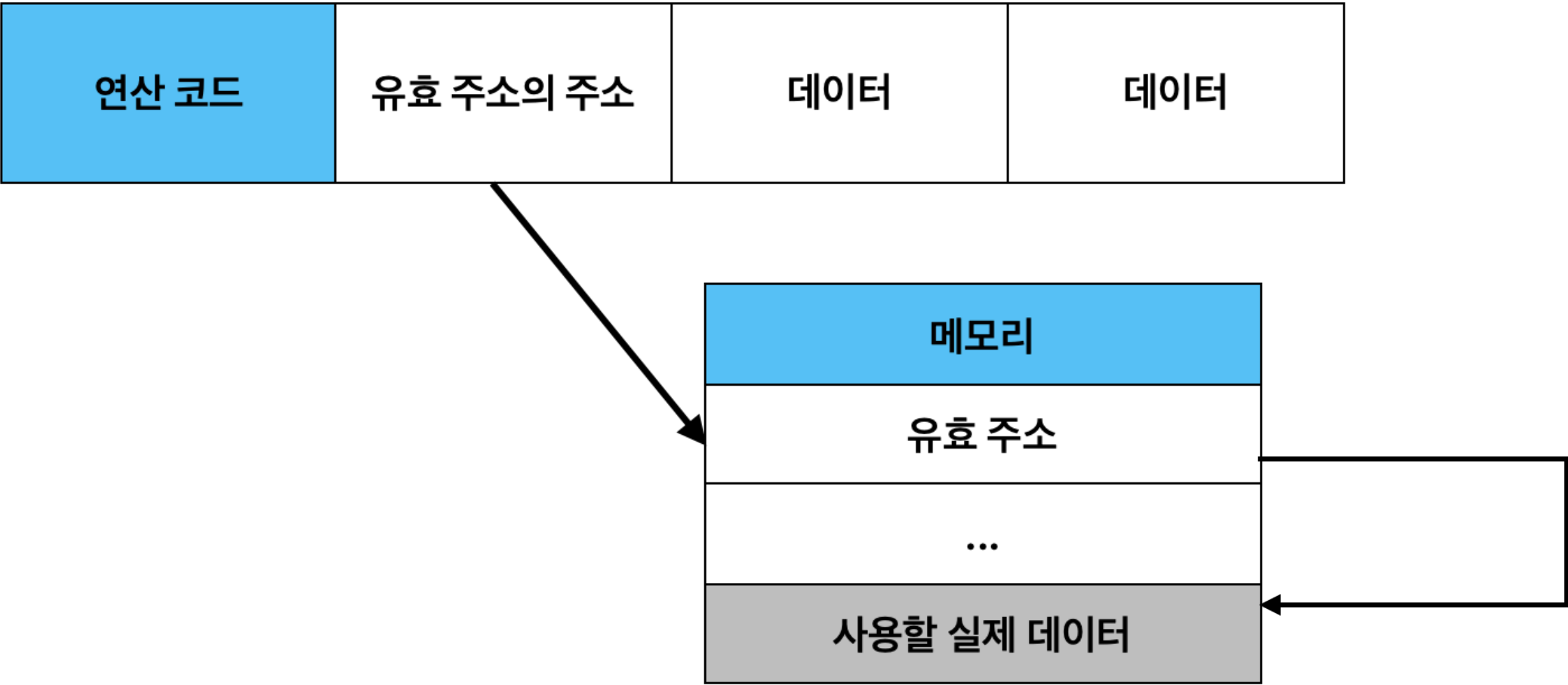
Low-Level Language

명령어 주소 지정 방식 1, 2, 3

1. 직접 주소 지정 방식



2. 간접 주소 지정 방식



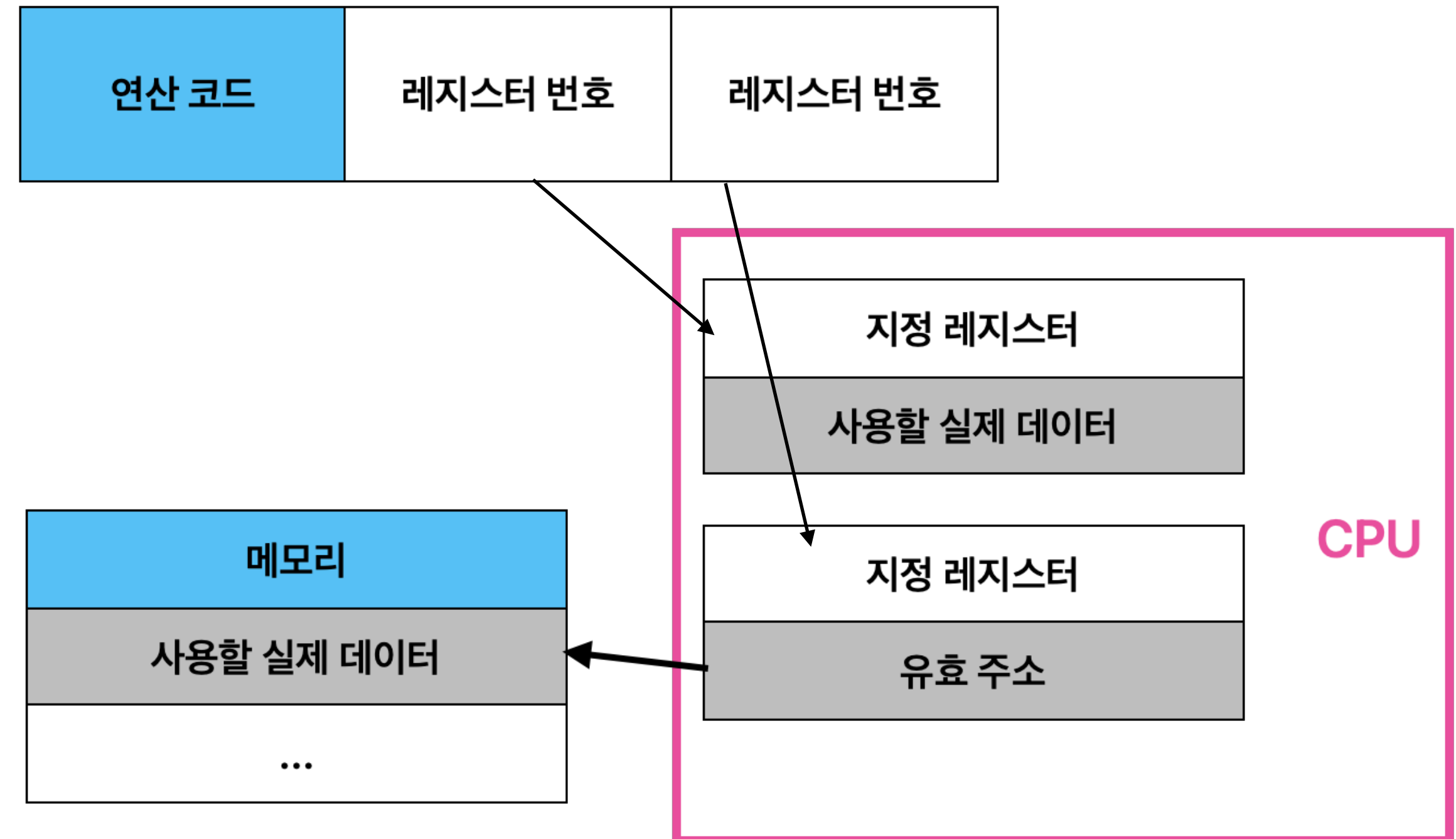
3. 즉시 주소 지정 방식

Low-Level Language

명령어 주소 지정 방식 4, 5

4. 레지스터 주소 지정 방식

5. 레지스터 간접 주소 지정 방식



Low-Level Language

명령어 주소 지정 방식

일반적으로 빠르고 메모리 효율성이 높은 순서

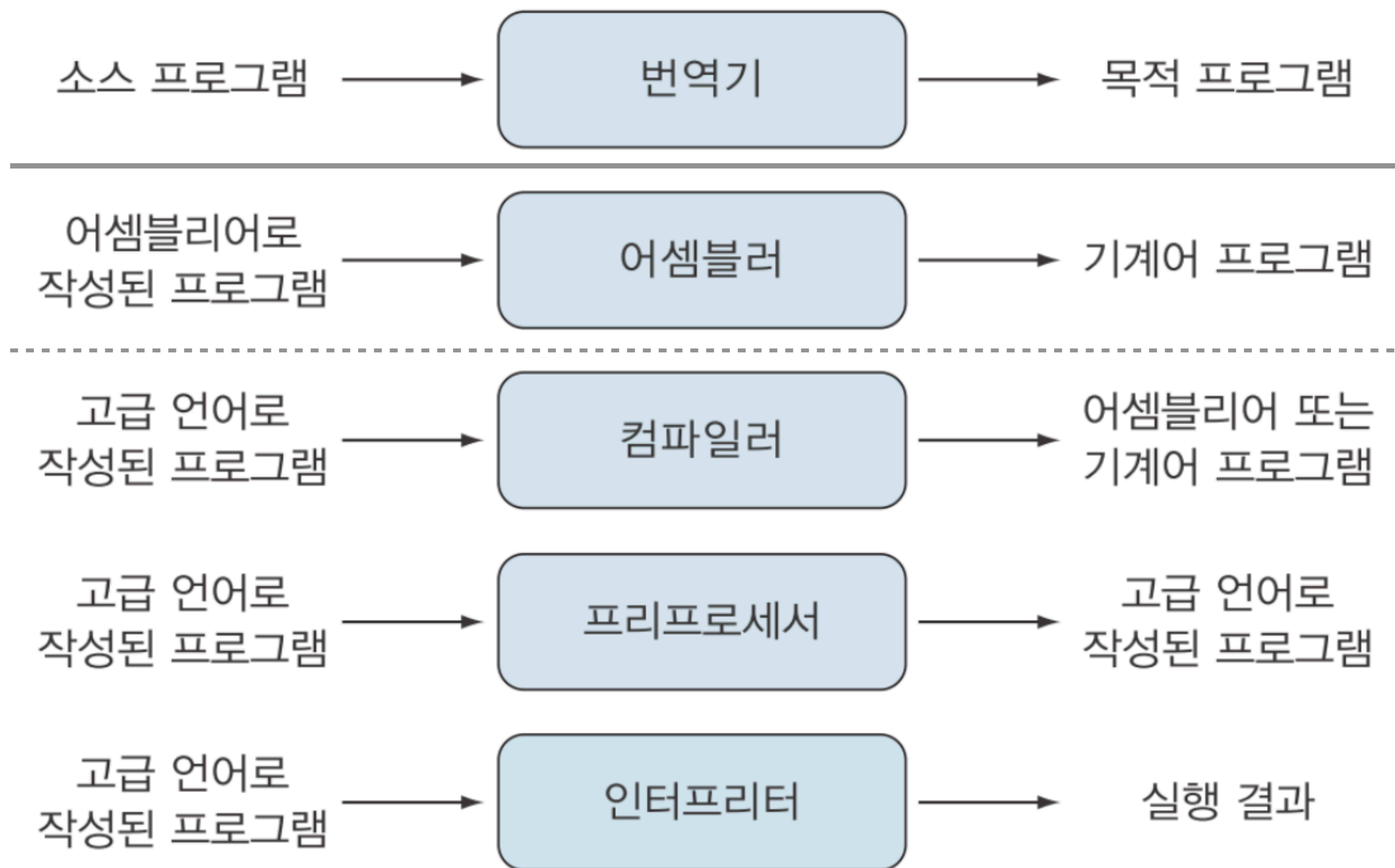
레지스터 간접 > 레지스터 > 간접 > 직접 > 즉시 주소

실행 방식에 의한 분류

High-Level Language

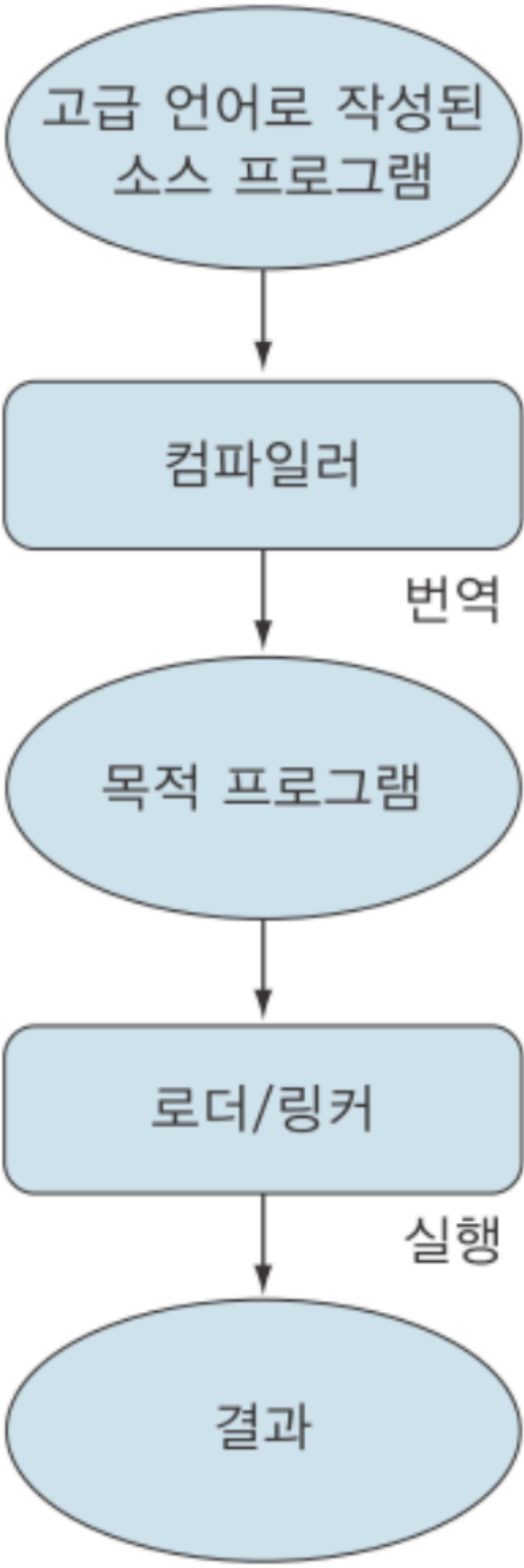
- 대표적인 방식
 - 컴파일 방식: 소스 코드 전체가 컴파일러에 의해 저급 언어로 번역
 - 인터프리터 방식: 소스 코드를 한 줄 씩 저급 언어로 번역

번역기 종류

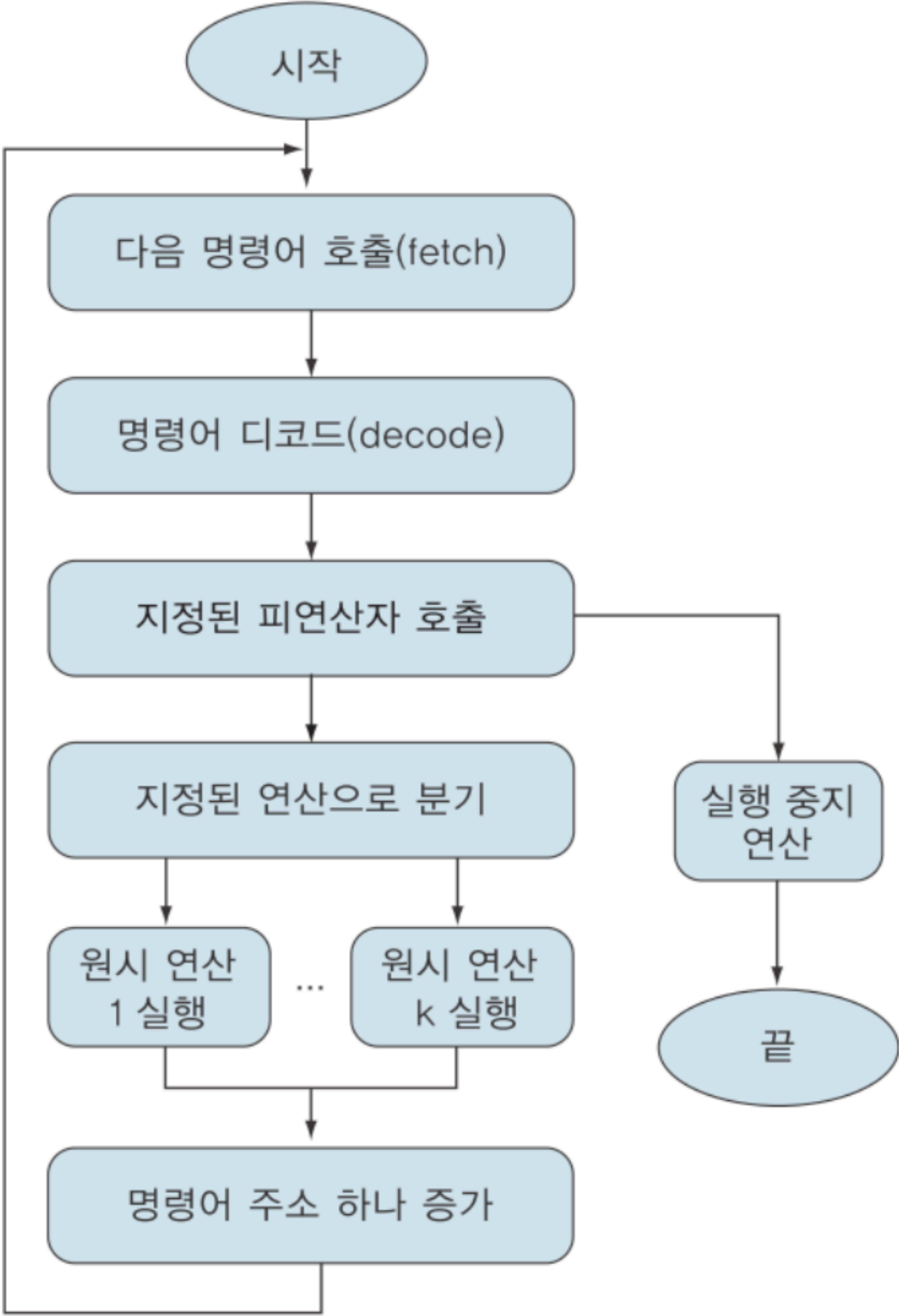


컴파일러와 인터프리터

컴파일러



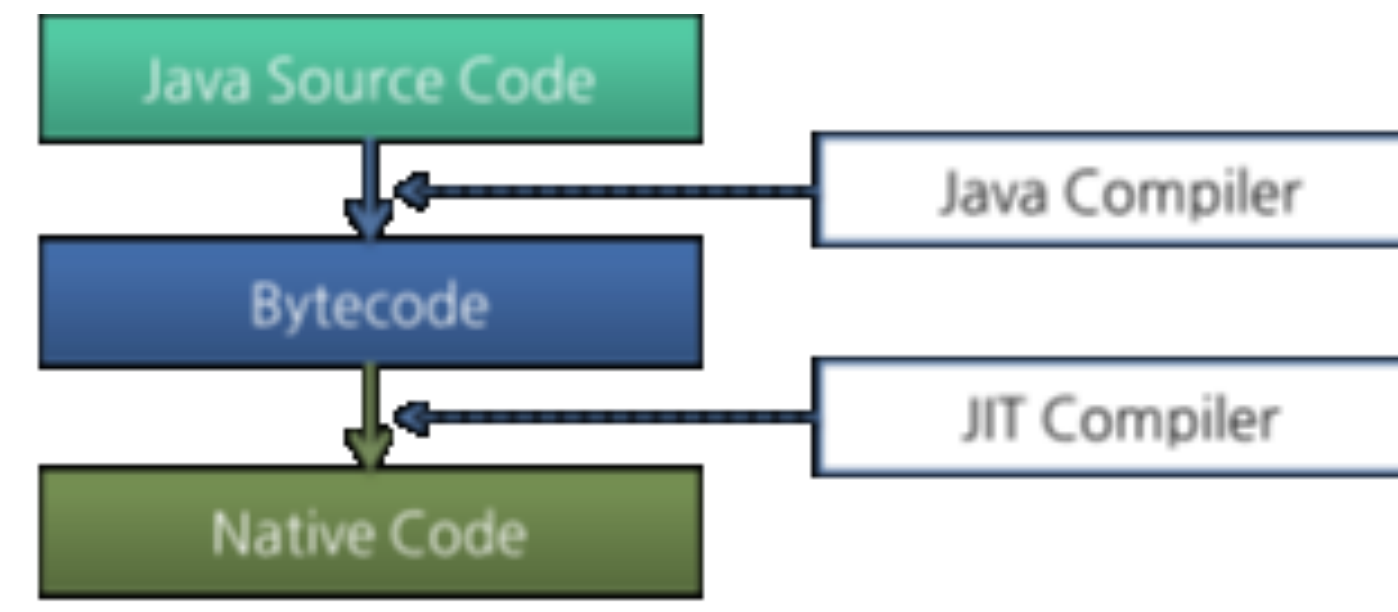
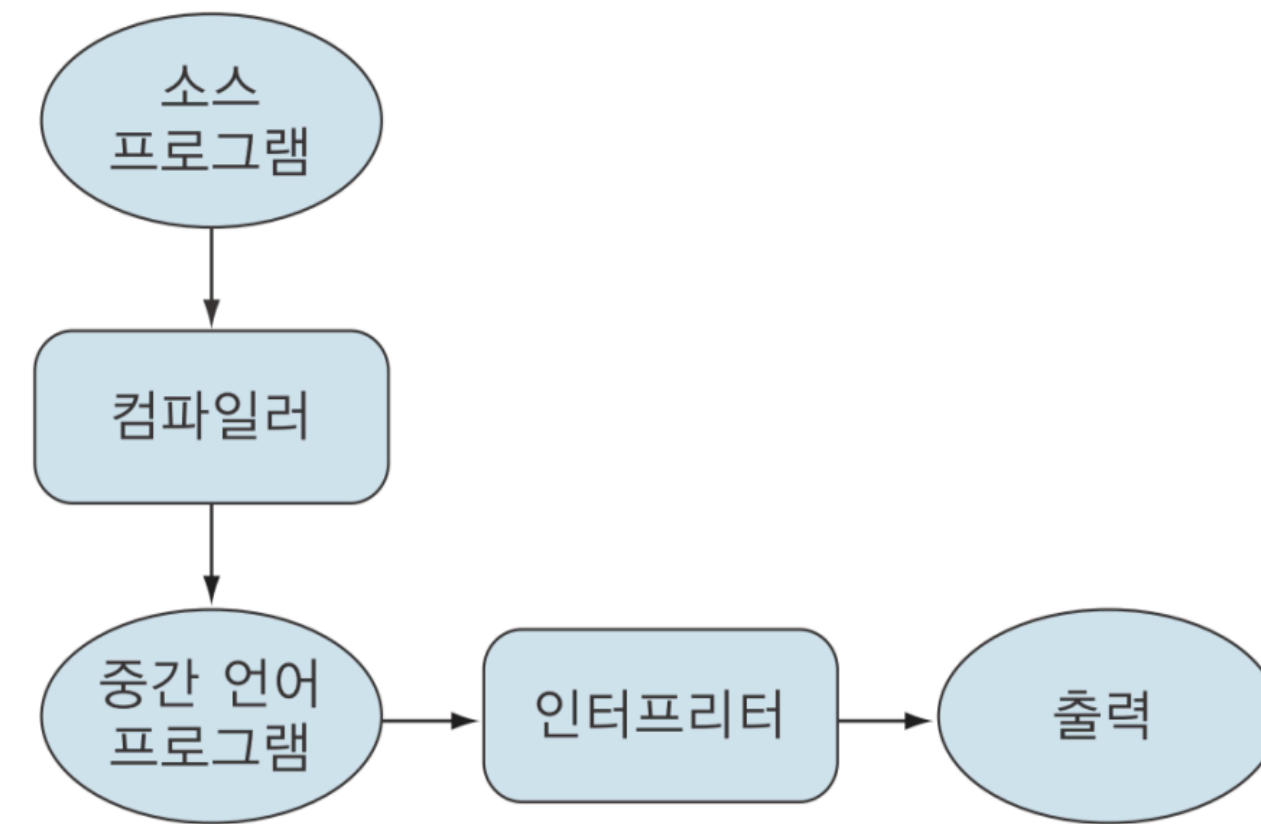
인터프리터



컴파일러와 인터프리터

혼합형 컴파일러

- Java, .Net Etc.
 - Bytecode Compiler
 - JIT(Just-In-Time Compiler)



Q & A

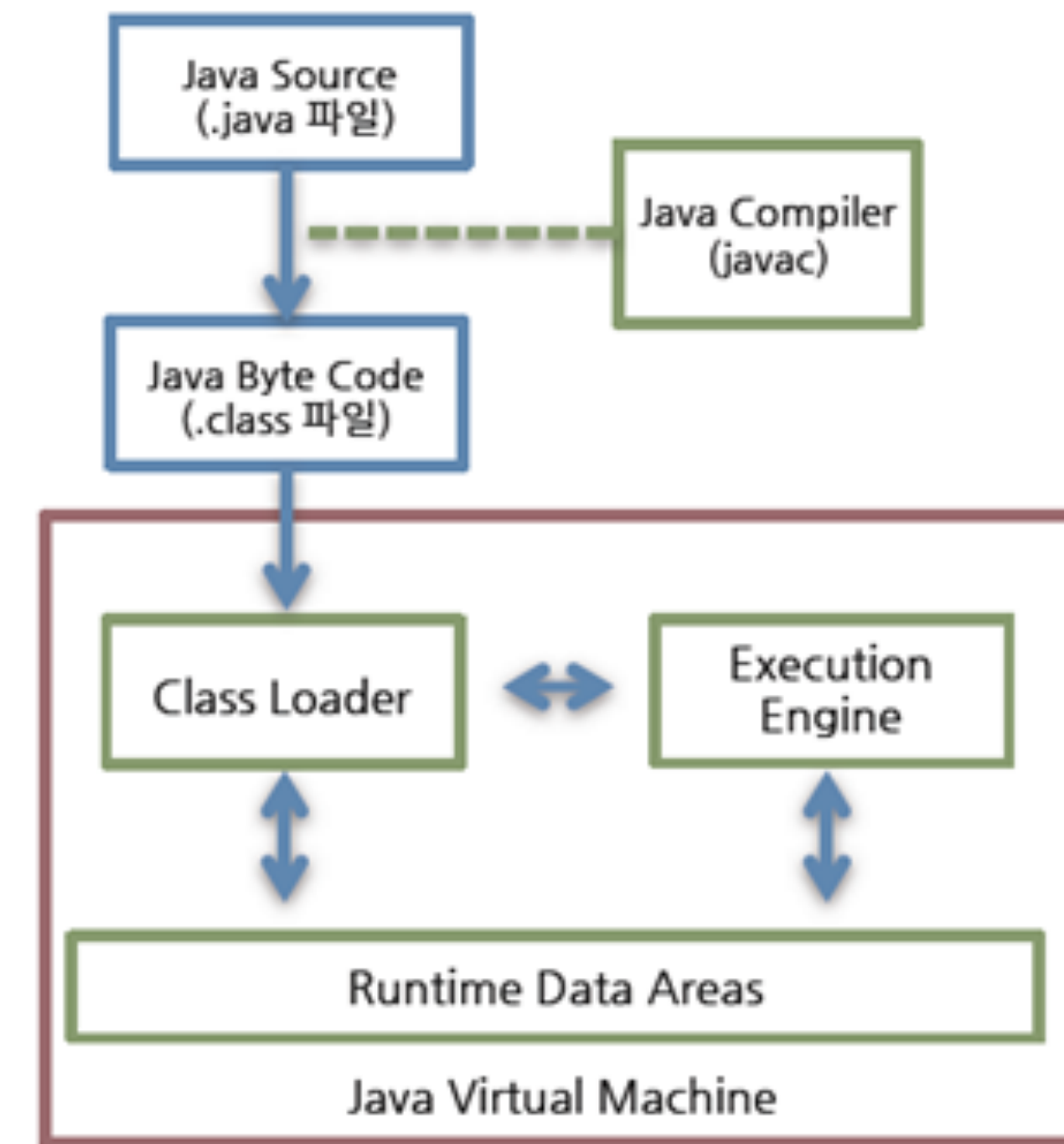
Appendix

Java 컴파일 과정

Java 컴파일 과정

JVM

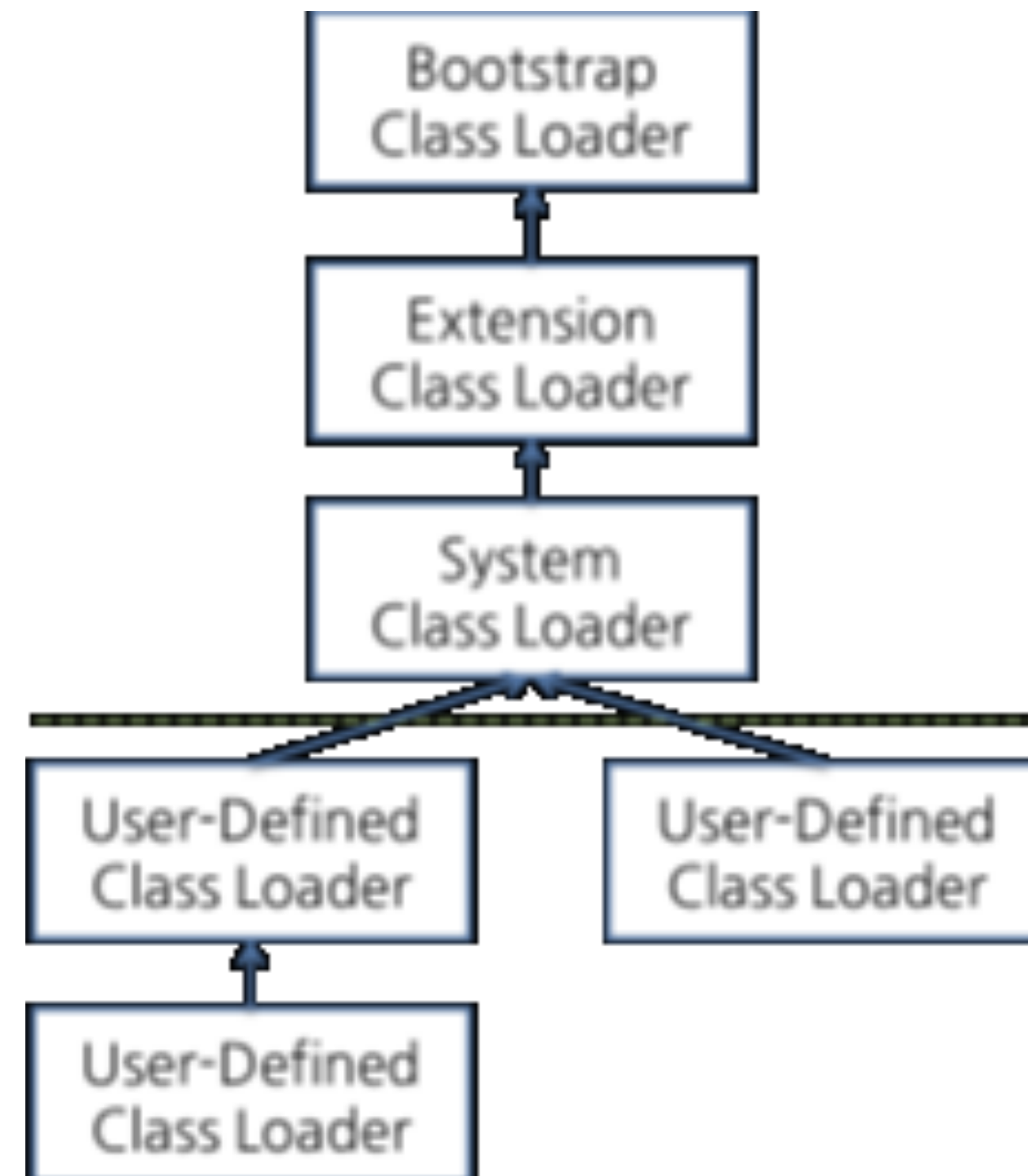
1. 개발자의 .java 작성
2. Java Compiler가 해당 소스파일 컴파일
3. 컴파일된 byte code를 JVM Class Loader에 전달
4. 필요한 Class들을 Load, Link하여 JVM 메모리에 올림
5. Execution Engine이 JVM 메모리상의 byte code들을 명령어 단위로 가져와 실행



Java 컴파일 과정

Class Loader

- 동적 로딩을 담당
- 특징
 - 1.계층 구조
 - 2.위임 모델
 - 3.가시성
 - 4.언로드 불가



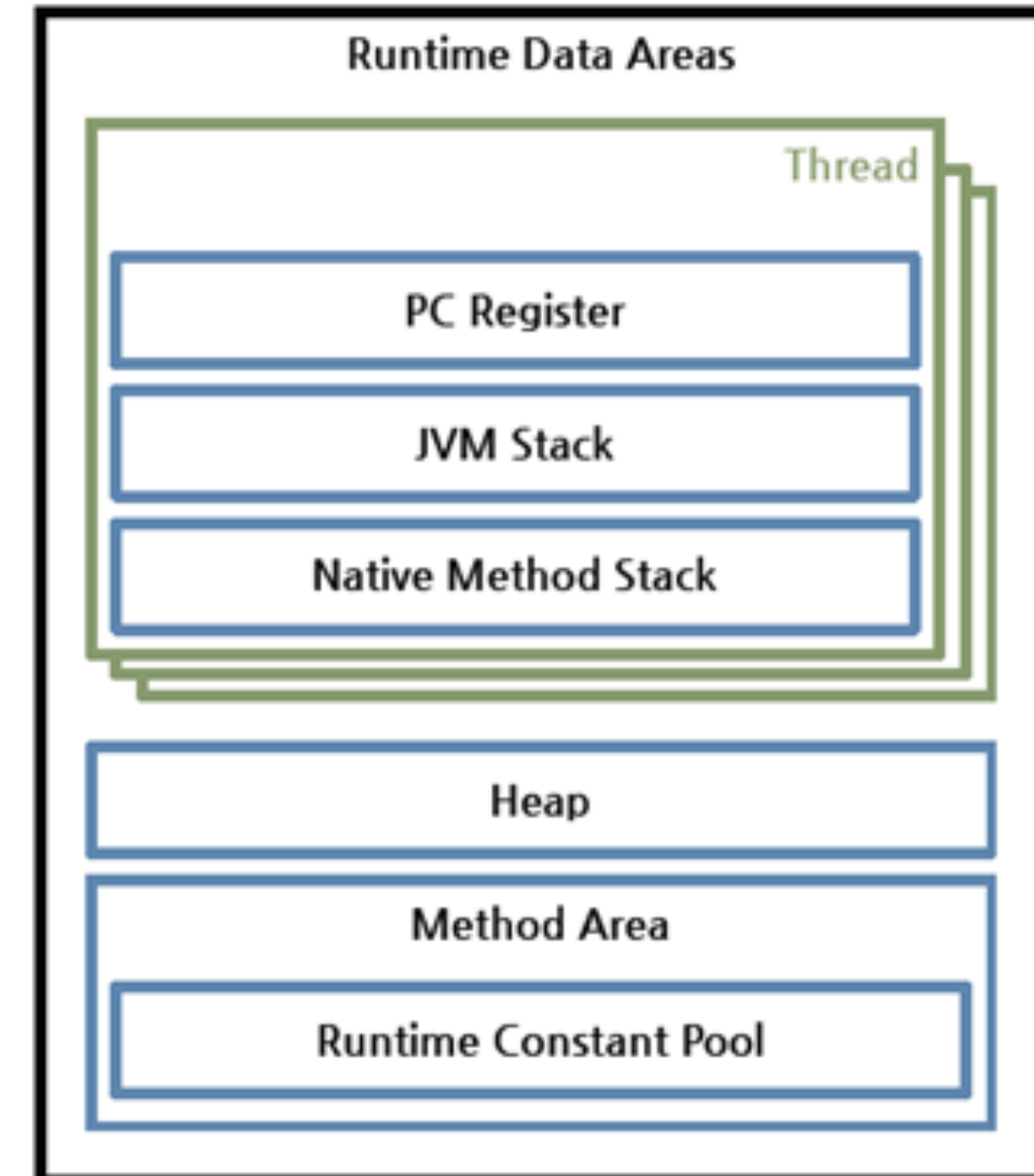
상위

하위

Java 컴파일 과정

Runtime Data Areas

- JVM이 OS 위에서 실행되며 할당받는 메모리 영역



Java 컴파일 과정

Executable Engine

- Byte code를 JVM 내부에서 컴퓨터가 실행할 수 있는 형태로 변경
- Java 성능 개선의 핵심점

- References

- NCS 학습 모듈 - 프로그래밍 언어 활용
- 전복대 - 컴파일러의 이해
- Naver D2 JVM Internal