

1장 - JVM은 무엇이며 자바는 어떻게 실행하는 것인가

출처 : <https://github.com/whiteship/live-study/issues/1>

목표

자바 소스 파일(.java)을 JVM으로 실행하는 과정 이해하기.

학습할 것

- JVM이란 무엇인가
- 컴파일 하는 방법
- 실행하는 방법
- 바이트코드란 무엇인가
- JIT 컴파일러란 무엇이며 어떻게 동작하는지
- JVM 구성 요소
- JDK와 JRE의 차이

들여가기에 앞서

오늘부터 백기선님의 자바 스터디를 같이 진행해보기로 했다.

동영상 강의만으론 제대로 이해하기가 어렵다는 이유와, 좀 더 깊게 언어를 이해하고 싶다는 이유다.

한 주차당 며칠이 걸릴지는 잘 모르겠지만, 최대한 하루만에 끝내려고 노력할 것이다.

자바는 WORA(Write Once Run Anywhere)을 원한다!

JVM이란 무엇인가

Java Virtual Machine, 자바 바이트코드를 실행할 수 있는 주체 (출처 : 위키백과)

자바 바이트코드는 JRE(Java Runtime Environment) 위에서 동작한다. 이 JRE에서 가장 중요한 요소는 **자바 바이트코드를 해석하고 실행하는 JVM**이다. JVM의 역할은 **자바 애플리케이션을 클래스 로더를 통해 읽어들이어서 자바 API와 함께 실행하는 것**이다.

자바 애플리케이션 해석을 클래스 로더 → 자바 API와 함께 실행 ?

JVM의 특징

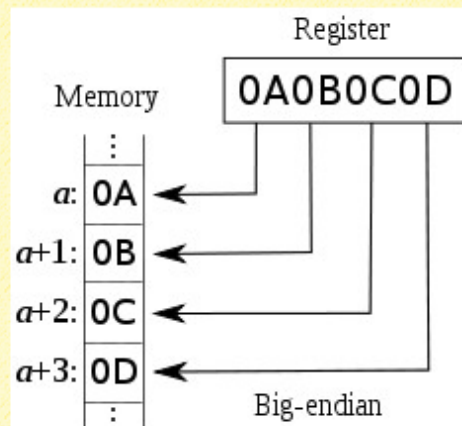
- **스택 기반의 가상 머신**
- ***심볼릭 레퍼런스** : 기본 자료형(primitive data type)을 제외한 모든 타입(클래스, 인터페이스)을 명시적인 메모리 주소 기반의 레퍼런스가 아니라, **심볼릭 레퍼런스를 통해 참조** 한다.
- **가비지 컬렉션**(GC, garbage collection) : 클래스 인스턴스는 사용자 코드에 의해 명시적으로 생성되고 가비지 컬렉션에 의해 자동으로 파괴된다
- **기본 자료형을 명확하게 정의하여 플랫폼 독립성 보장** : C/C++ 등의 전통적인 언어는 플랫폼에 따라 int형의 크기가 변한다. JVM은 **기본 자료형을 명확하게 정의하여 호환성을 유지** 하고 **플랫폼 독립성은 보장** 한다.
- **네트워크 바이트 오더(network byte order)** : 자바 클래스 파일은 네트워크 바이트 오더를 사용. 리틀 엔디안이나, 빅 엔디안 사이에서 플랫폼 독립성을 유지하려면 고정된 바이트 오더를 유지해야 하므로 네트워크 전송 시에 사용하는 바이트 오더인 네트워크 바이트 오더를 사용한다. 네트워크 바이트 오더는 **빅 엔디안** 이다.

▼ 빅 엔디안

(2) Big Endian

: 데이터가 상위 바이트 부터 메모리에 적재(Network Ordering)

가장 최상위 바이트(0A)가 가장 낮은 메모리 주소에 저장되는 방식



(ex) 0x12345678의 32비트 값을 Big Endian으로 표현하면 :

0x12 0x34 0x56 0x78

낮은 주소 --> 높은주소

- 플랫폼 독립성 보장을 위해 사용됨

이를 조사하면서 여러가지 의문의 들었다. 그에 대해 아래에 정리해본다.

? 스택 기반 레지스터 기반과의 차이점?

가상머신의 구현체는 명세서를 어떻게 구현하냐에 따라 여러 종류가 된다. 보통 물리적인 CPU에 의해 처리되는 동작을 흉내낼 수 있어야 하며, 아래와 같은 컨셉을 가진다.

[가상 머신의 필수요소]

- 소스 코드를 실행가능한 바이트코드로 변환한다.

- 명령어와 피연산자를 포함하는 데이터구조를 갖는다.
- 함수를 실행하기 위한 콜 스택
- IP(Instruction Pointer - 다음 실행할 곳을 지정하는 포인터)
- 가상 CPU - 다음 명령어를 fetch & 명령어 해석 & 명령 실행

[Stack 기반 가상머신]

- 대다수의 가상머신이 스택 기반
- 피연산자와 연산 후 결과를 스택에 저장
- 피연산자가 스택 포인터에 의해 암시적으로 처리됨
- 코드 작성과 컴파일이 쉽고 가상머신이 빠름

계산 과정이 귀찮고 복잡해 지더라도 하드웨어에 따라 레지스터의 개수가 다르기 때문에 다기종의 디바이스에 지원하기 위해 채택했다고 생각함

[레지스터 기반 가상머신]

- 실제 하드웨어와 비슷해서 코드 생성기가 코드를 생성하기 쉽고 빠르다.
 - 스택 기반 VM은 동일한 피연산자를 여러 번 푸시하지만, 레지스터 기반은 적절한 양의 레지스터를 할당하고 작업하여 작업량과 CPU 시간을 크게 줄일 수 있다.
- 스택으로 밀고 들어가는 오버헤드가 존재하지 않음
- 같은 연산을 할 때 더 적은 명령어로 연산 수행이 가능하다.
- 일부 최적화 가능

? 심볼릭 레퍼런스

- 프로그램 실행을 위한 API를 클래스가 직접 가지는 것이 아닌 이름을 통한 참조값을 이용해서 실행할 때 메모리 상에서 API를 호출할 수 있도록 이름을 주소로 대체하는 특징을 의미

출처 : <https://d2.naver.com/helloworld/1230>

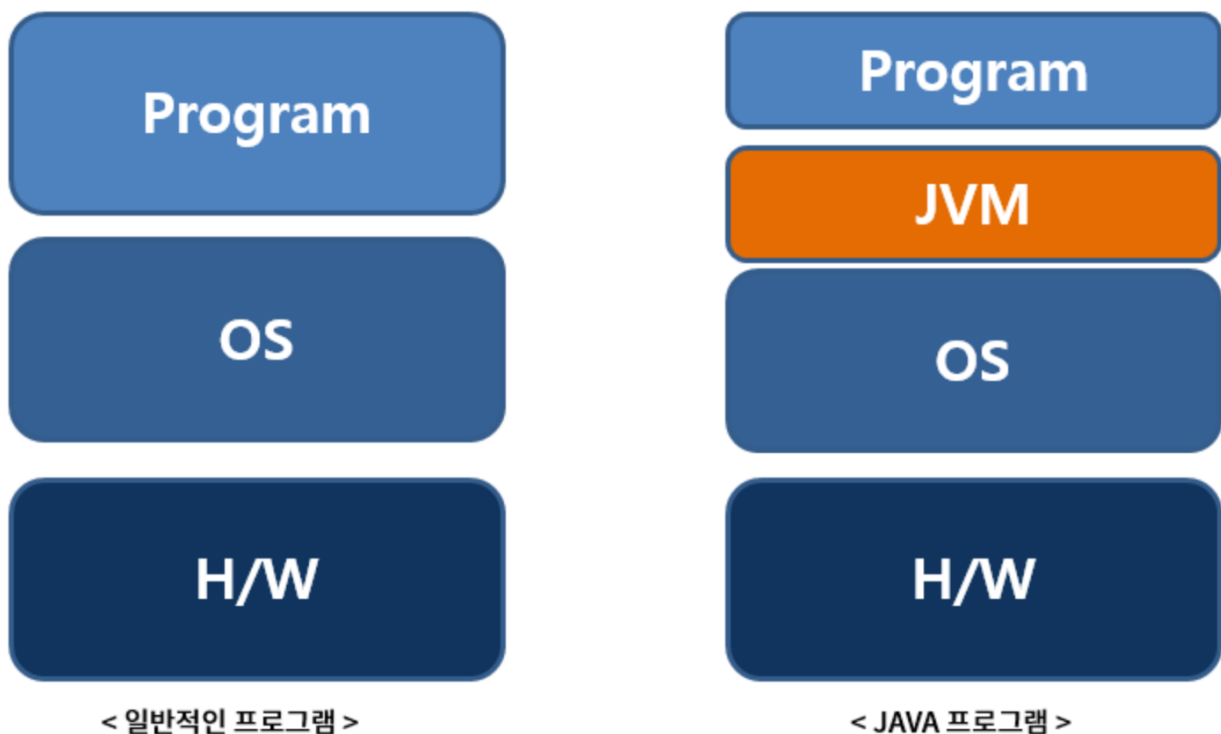
<https://s2choco.tistory.com/13>

컴파일 하는 방법

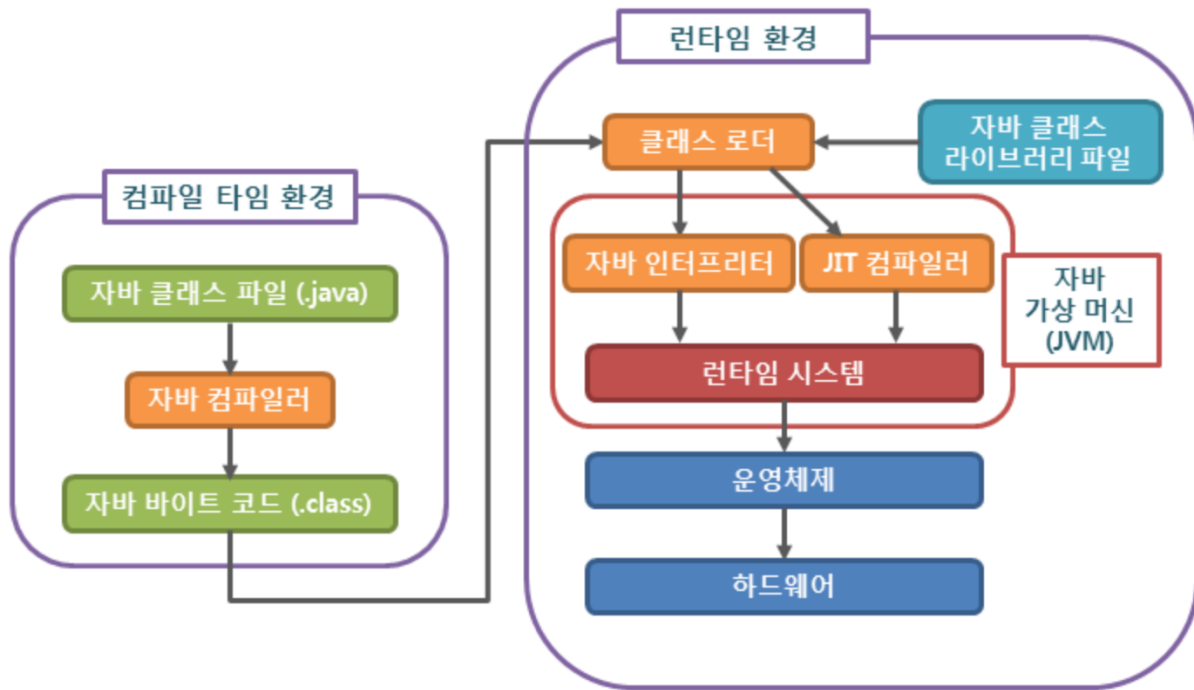
자바는 **OS에 독립적인** 특징을 가지고 있다. 그것을 가능하게 한 것이 JVM이다. 이를 어떻게 가능하게 했는지 자바 컴파일 과정을 통해 알아보자.

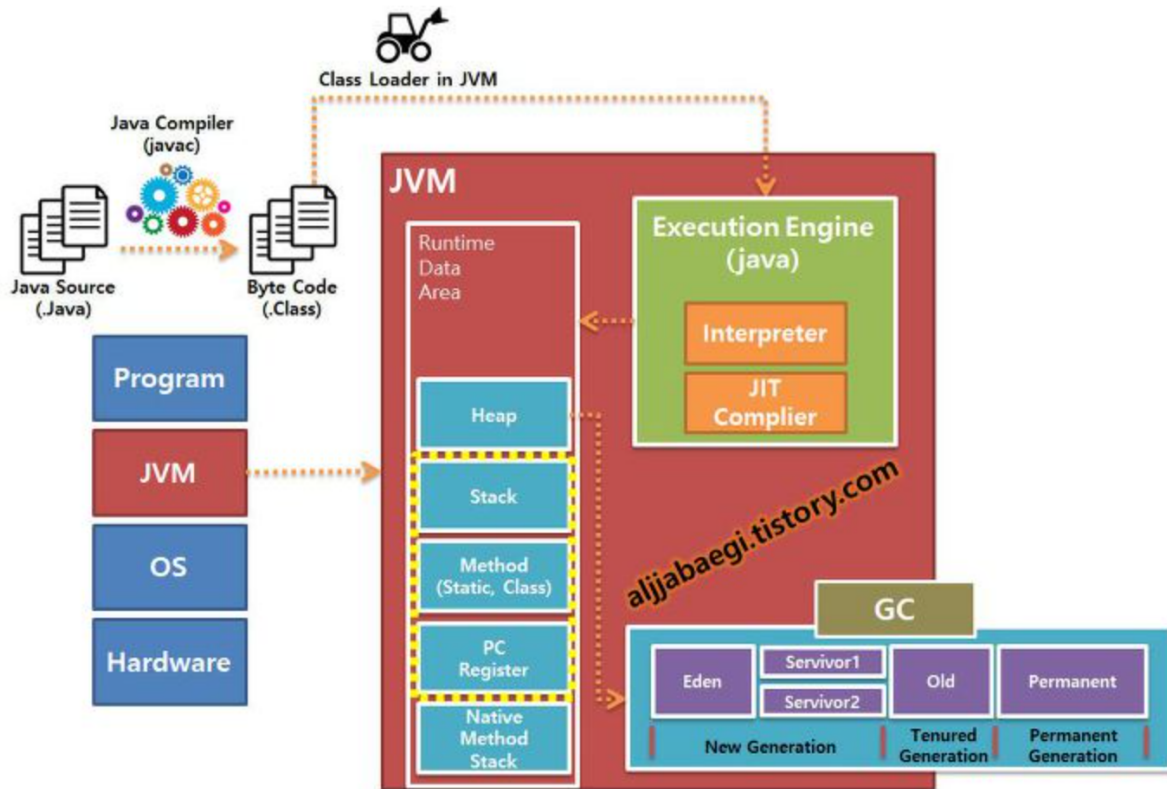
? 자바에서의 컴파일

- `.java` 를 `.class(byte code)` 로 바꾸는 과정



컴파일 순서

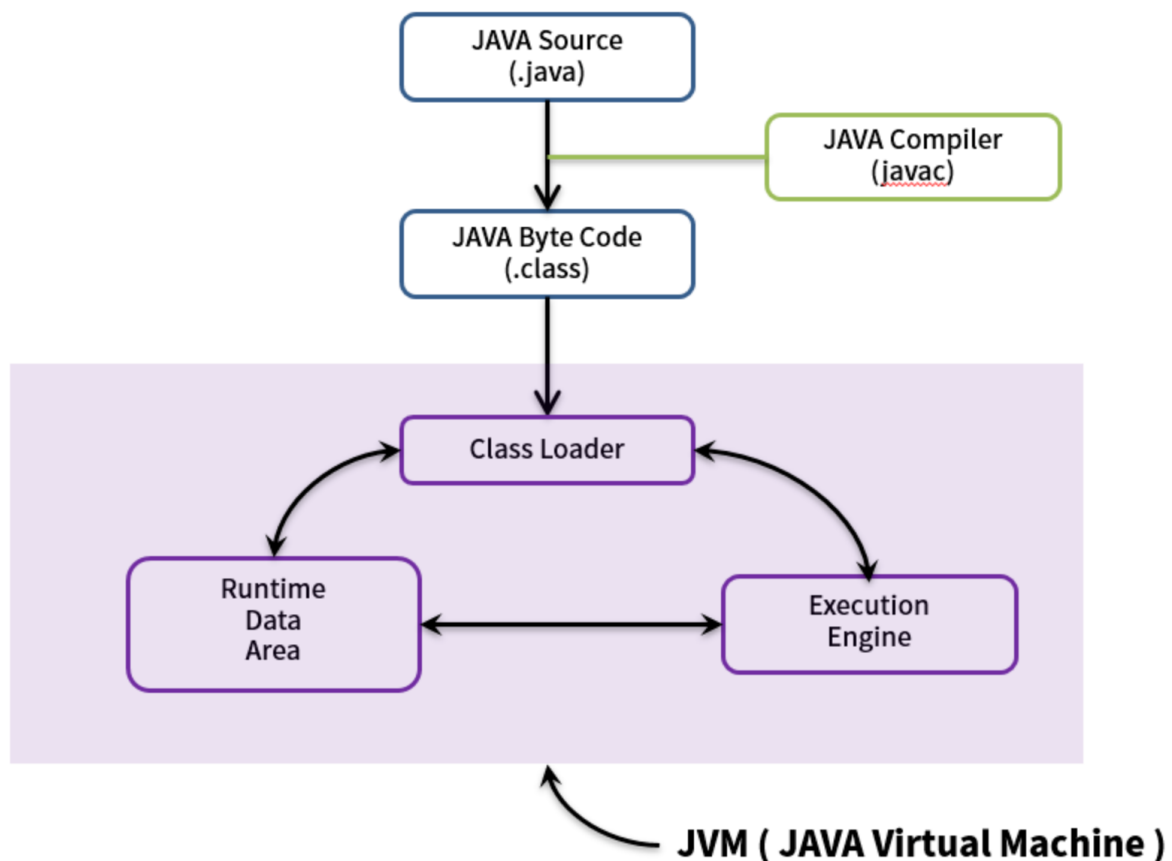




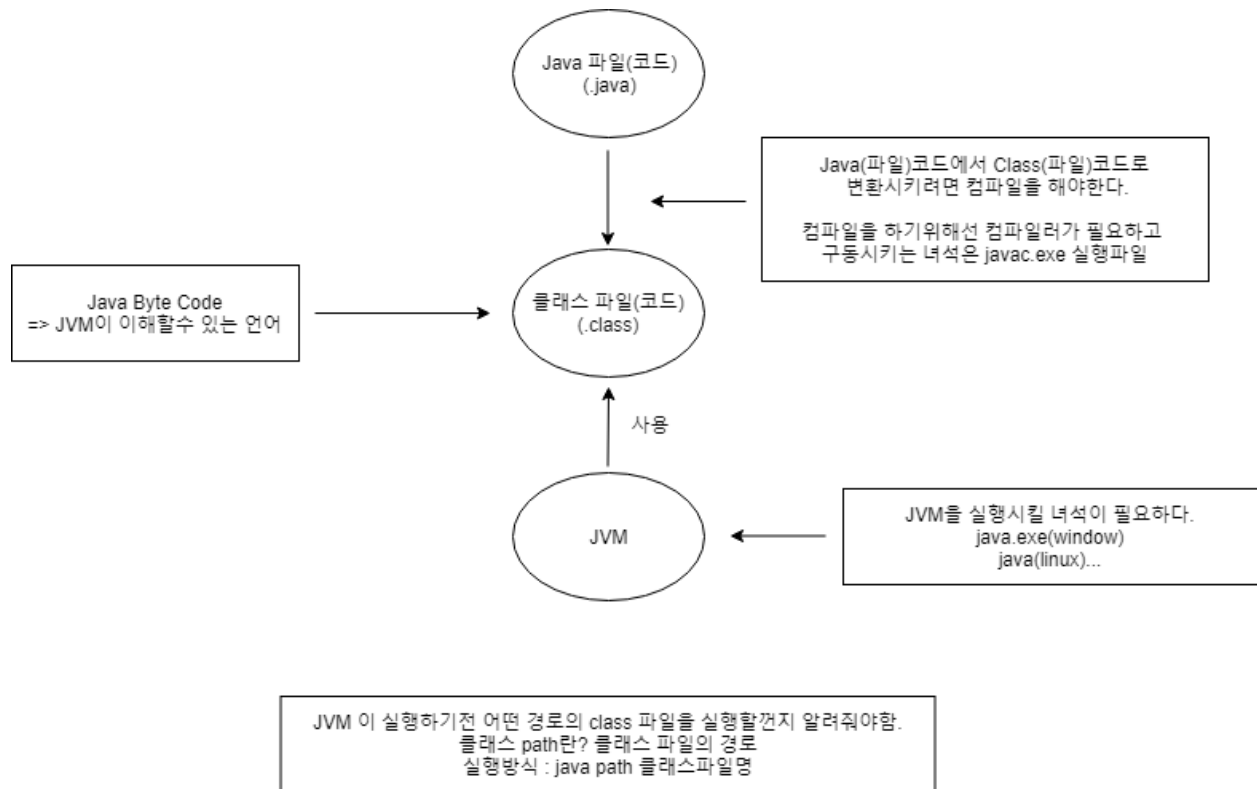
1. 자바 소스코드(.java)를 작성한다.
2. 자바 컴파일러(java Compiler)가 자바 소스파일을 컴파일한다.
 → 이때 나오는 파일은 자바 **바이트 코드(.class)**파일로 아직 컴퓨터가 읽을 수 없는 자바 가상 머신이 이해할 수 있는 코드이다. 바이트 코드의 각 명령어는 1바이트 크기의 **Opcode**와 추가 피연산자로 이루어져 있다.
3. 컴파일된 바이트 코드를 JVM의 클래스로더(Class Loader)에게 전달.
4. 클래스 로더는 동적로딩(Dynamic Loading)을 통해 필요한 클래스들을 로딩 및 링크하여 런타임 데이터 영역(Runtime Data area), 즉 JVM의 메모리에 올린다.
 - **클래스 로더** 세부 동작
 - a. **로드** - 클래스 파일을 가져와서 JVM의 메모리에 로드한다.
 - b. **검증** - 자바 언어 명세(Java Language Specification) 및 JVM 명세에 명시된 대로 구성되어 있는지 검사한다.
 - c. **준비** - 클래스가 필요로 하는 메모리를 할당(필드, 메서드, 인터페이스 등등)

- d. **분석** - 클래스의 상수 풀 내 모든 심볼릭 레퍼런스를 다이렉트 레퍼런스로 변경한다.
 - e. **초기화** - 클래스 변수들은 적절한 값으로 초기화한다. (static 필드)
5. **실행엔진(Execution Engine)**은 JVM 메모리에 올라온 바이트 코드들을 명령어 단위로 하나씩 가져와서 실행. 이때, 실행 엔진은 두 가지 방식으로 변경한다.
- a. **인터프리터** - 바이트 코드 명령어를 하나씩 읽어서 해석하고 실행한다. 하나하나의 실행은 빠르나, **전체적인 실행 속도가 느리다**는 단점을 가짐
 - b. **JIT 컴파일러 (Just-In-Time Compiler)** - 인터프리터의 단점을 보완하기 위해 도입된 방식. 바이트코드 전체를 컴파일하여 **바이너리 코드로 변경**하고 이후에는 해당 메시지를 더 이상 인터프리팅 하지 않고, **바이너리 코드로 직접 실행하는 방식**이다. 하나씩 인터프리팅하여 실행하는 것이 아니라 바이트 코드 전체가 컴파일된 바이너리 코드를 실행하는 것이기 때문에 **전체적인 실행속도는 인터프리팅 방식보다 빠르다**.

자바 코드(JAVA Code) 실행 과정



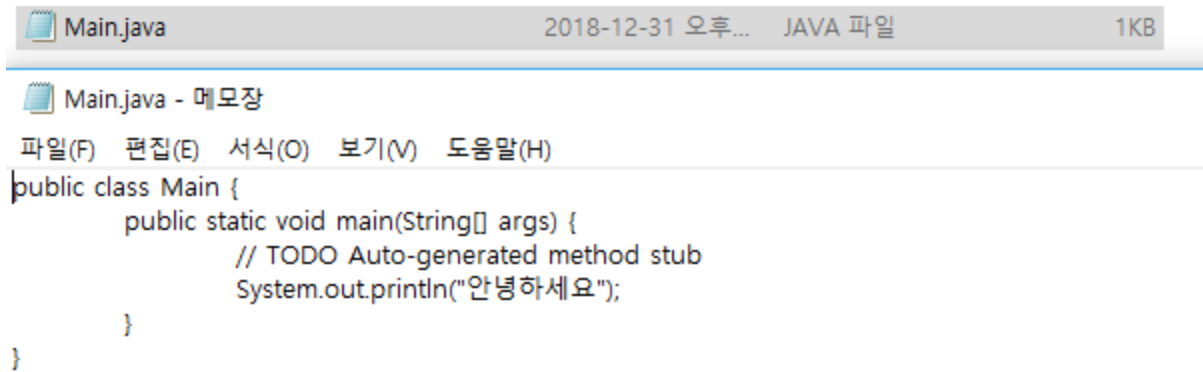
실행하는 방법



▼ 일반적으로 IDE에서 .java파일을 만들어 실행시키지만, 실무에선 .java파일을 IDE가 아닌 다른 환경에서 실행해야 하는 경우가 종종 있다.

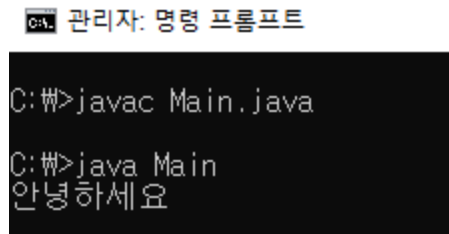
일반적인 방법

1. C 드라이브에 Main.java 파일 생성



2. cmd 창에서 컴파일과 실행

- 컴파일 - **javac** Main.java
→ Main.class 파일이 생성됨
- 실행 - **java** Main



package가 있을 때 (package Test.Now;가 있다고 가정)

1. Main.java 자바 파일에 package Test.Now; 를 추가함
2. 파일에 package가 있다면 Main.java 를 컴파일한 Main.class는 그 경로에 존재해야 하며, .class 파일을 실행시키기 위해선 package 경로가 붙어야 한다. (java Test.now.Main)
3. javac -d . Main.java
→ 현재 디렉토리(.)에 Main.java의 package 디렉토리를 만들어줘라는 의미

Main.java 파일에 package Test.Now; 추가한 후 javac Main.java를 하면 Main.class는 나오지만, 실행하면 오류가 발생한다.

- class 파일이 package의 경로에 존재하지 않는다.
- class 파일을 실행하기 위해선 package의 경로를 적어줘야 한다.

출처 : <https://yoonemong.tistory.com/183>

<https://superblo.tistory.com/entry/커맨드cmd에서-자바-컴파일하기-및-실행-방법2>

바이트코드란 무엇인가

WORA(Write Once Run Anywhere)를 구현하기 위해 JVM은 사용자 언어인 자바와 기계어 사이의 중간 언어인 자바 바이트코드를 사용한다. 이 자바 바이트코드가 자바 코드를 배포하는 가장 작은 단위이다.

▼ 재미있는 사례

자바 바이트코드에 대해 설명하기 전에 한 가지 사례를 살펴 보자. 이 사례는 개발 과정에서 실제로 발생한 것을 변경, 요약한 것이다.

현상

원래 잘 동작하던 애플리케이션이 라이브러리 업데이트 이후로 다음과 같은 오류를 내고 동작하지 않는다.

```
Exception in thread "main" java.lang.NoSuchMethodError: com.nhn.user.UserAdmin.addUser(Ljava/lang/String;)V
at com.nhn.service.UserService.add(UserService.java:14)
at com.nhn.service.UserService.main(UserService.java:19)
```

애플리케이션 코드는 변경하지 않았으며 다음과 같다.

```
// UserService.java
...
public void add(String userName) {
    admin.addUser(userName);
}
```

업데이트된 라이브러리 소스코드와 원래 소스코드는 다음과 같다.

```
// UserAdmin.java - 업데이트된 소스코드
...
public User addUser(String userName) {
    User user = new User(userName);
    User prevUser = userMap.put(userName, user);
    return prevUser;
}
// UserAdmin.java - 원래 소스코드
...
public void addUser(String userName) {
    User user = new User(userName);
    userMap.put(userName, user);
}
```

즉, 반환값이 없던 addUser() 메서드가 User 클래스 인스턴스를 반환하는 메서드로 변경되었다. 그러나, 애플리케이션 코드는 addUser() 메서드의 반환값을 사용하지 않으므로 변경하지 않았다. 보기에는 com.nhn.user.UserAdmin.addUser() 메서드는 여전히 존재하는 것 같은데 왜 NoSuchMethodError가 발생할까?

원인

애플리케이션 코드를 새로운 라이브러리로 다시 컴파일하지 않았기 때문이다. 즉, 애플리케이션 코드는 우리가 보기에는 반환값과 무관하게 메서드를 호출하고 있는 것 같지만, 실제로 컴파일된 클래스 파일은 반환값까지 지정된 메서드를 지칭한다.

이는 다음 오류 메시지를 살펴보면 확실히 알 수 있다.

```
java.lang.NoSuchMethodError: com.nhn.user.UserAdmin.addUser(Ljava/lang/String;)V
```

NoSuchMethodError는 "com.nhn.user.UserAdmin.addUser(Ljava/lang/String;)V"라는 메서드를 찾지 못해서 발생했다. 여기서 관심을 가질 부분은 "Ljava/lang/String;"과 마지막의 "V"이다. 자바 바이트코드의 표현에서 "L;"은 클래스 인스턴스이다. 즉, addUser()

메서드는 `java/lang/String` 객체 하나를 파라미터로 받는 메서드이다. 이 사례의 라이브러리에서는 파라미터가 변경되지 않았으므로 이 파라미터는 정상이다. 위 메시지 마지막의 "V"는 메서드의 반환값을 나타낸다. 자바 바이트코드 표현에서 "V"는 반환값이 없음을 의미한다. 즉, 위 오류 메시지는 `java.lang.String` 객체 1개를 파라미터로 받고 반환값은 없는 `com.nhn.user.UserAdmin.addUser`라는 메서드를 찾지 못했다는 의미이다.

애플리케이션 코드는 이전 라이브러리로 컴파일되었으므로, "V"를 반환하는 메서드를 호출하도록 class 파일에 기록되어 있지만, 새로 변경된 라이브러리에서 "V"를 반환하는 메서드는 없어지고, "Lcom/nnh/user/User;"를 반환하는 메서드가 추가되었기 때문에 `NoSuchMethodError`가 발생한 것이다.

참고

오류 자체는 새로운 라이브러리를 다시 컴파일하지 않았기 때문에 발생한 것이지만, 이 사례에서는 라이브러리 제공자의 잘못이 크다고 할 수 있다. `public`으로 공개된 메서드의 반환값이 없다가 `User` 클래스 인스턴스를 반환하도록 변경된 것이므로, 이것은 명백한 메서드 시그니처 변경이다. 즉, 라이브러리의 하위 호환성이 깨진 것이므로 라이브러리 제공자는 메서드가 변경되었다는 것을 반드시 사용자에게 알렸어야 한다.

JVM을 이야기 할 때 자바 바이트코드는 뺄 수 없다. **JVM**은 **자바 바이트코드를 실행하는 실행기**이다. 자바 컴파일러는 C/C++등의 컴파일러처럼 고수준 언어를 기계어, 즉 직접적인 CPU 명령으로 변환하는 것이 아니라, 개발자가 이해하는 **자바 언어를 JVM이 이해하는 자바 바이트코드로 번역한다**. 따라서 **자바 바이트코드는 플랫폼 의존적인 코드가 없기 때문에 JVM(정확하게 말하자면 같은 프로파일의 JRE)이 설치된 장비라면 CPU나 운영체제가 다르더라도 실행할 수 있고** (윈도우 PC에서 개발하여 컴파일한 클래스 파일을 리눅스 장비에서도 별다른 변경 없이 실행한다), 컴파일 결과물의 크기가 소스코드의 크기와 크게 다르지 않으므로 **네트워크로 전송하여 실행하기가 쉽다**.

클래스 파일 자체는 바이너리 파일이므로 사람이 이해하기 쉽지 않다. 이 점을 보완하기 위해 JVM 벤더들은 `javap`라는 역어셈블러(`disassembler`)를 제공한다. `javap`를 | 〇용한 결과물을 훑 | 자바 어셈블리라고 부른다. 앞의 사례에서 애플리케이션 코드 `UserService.add()` 메서드를 `javap -c` 옵션으로 역어셈블한 결과물은 다음과 같다.

```
public void add(java.lang.String);
Code:
```

```

0: aload_0
1: getfield #15; //Field admin:Lcom/nhn/user/UserAdmin;
4: aload_1
5: invokevirtual #23; //Method com/nhn/user/UserAdmin.addUser:(Ljava/lang/String;)V
8: return

```

이 결과물에서 addUser() 메서드를 호출하는 부분은 4번째줄인 "5: invokevirtual #23;"이다. 이는 23번 인덱스에 해당하는 메서드를 호출하라는 의미이며, 23번 인덱스의 메서드를 **javap 프로그램이 친절하게 주석**으로 달아주었다. invokevirtual은 자바 바이트코드에서 메서드를 호출하는 가장 기본적인 명령어의 OpCode(operation code)이다. 참고로, 자바 바이트코드에서 메서드를 호출하는 명령어 OpCode는 invokeinterface, invokespecial, invokestatic, invokevirtual의 4가지가 있으며 각각의 의미는 다음과 같다.

- **invokeinterface** - 인터페이스 메서드 호출
- **invokespecial** - 생성자, private 메서드, 슈퍼 클래스의 메서드 호출
- **invokestatic** - static 메서드 호출
- **invokevirtual** - 인스턴스 메서드 호출

자바 바이트코드의 명령어는 OpCode와 피연산자(Operand)로 분리할 수 있으며, invokevirtual과 같은 OpCode는 2바이트의 피연산자를 필요로 한다.

위의 애플리케이션 코드를 업데이트된 라이브러리로 다시 컴파일하여 다시 역어셈블하면 다음 결과를 얻을 수 있다.

```

public void add(java.lang.String);
Code:
0: aload_0
1: getfield #15; //Field admin:Lcom/nhn/user/UserAdmin;
4: aload_1
5: invokevirtual #23; //Method com/nhn/user/UserAdmin.addUser:(Ljava/lang/String;)Lcom/nhn/user/User;
8: pop
9: return

```

23번에 해당하는 메서드가 "Lcom/nhn/user/User;"를 반환하는 메서드로 변환됐다.

위의 역어셈블 결과물에서 **코드 앞의 숫자**는 무엇을 의미할까? 바로 **바이트 번호**이다. JVM이 실행하는 코드를 굳이 자바 "바이트"코드라고 하는 이유가 바로 이것일 것이다. 즉, 위의 `aload_0`, `getfield`, `invokevirtual` 같은 바이트코드 명령어 OpCode들은 1 바이트의 바이트 번호로 표현된다. `aload_0 = 0x2a`, `getfield = 0xb4`, `invokevirtual = 0xb6` 등이다. 따라서, 자바 바이트코드 명령어 OpCode는 최대 256개라는 점을 알 수 있다.

`aload_0`, `aload_1`과 같은 OpCode는 피연산자가 필요 없다. 따라서 `aload_0` 바로 다음 바이트가 다음 명령어의 OpCode가 된다. 그러나 `getfield`, `invokevirtual`은 2바이트의 피연산자가 필요하다. 따라서 첫 번째 바이트에 있는 `getfield`의 다음 명령어는 2바이트를 건너뛰는 네 번째 바이트에 기록된다. 위의 바이트코드를 Hex Editor로 보면 다음과 같다.

```
2a b4 00 0f 2b b6 00 17 57 b1
```

자바 바이트코드에서 클래스 인스턴스는 "L;", void는 "V"로 표시되는 것처럼 다른 타입들도 고유의 표현이 있다. 이 표현을 정리하면 다음과 같다.

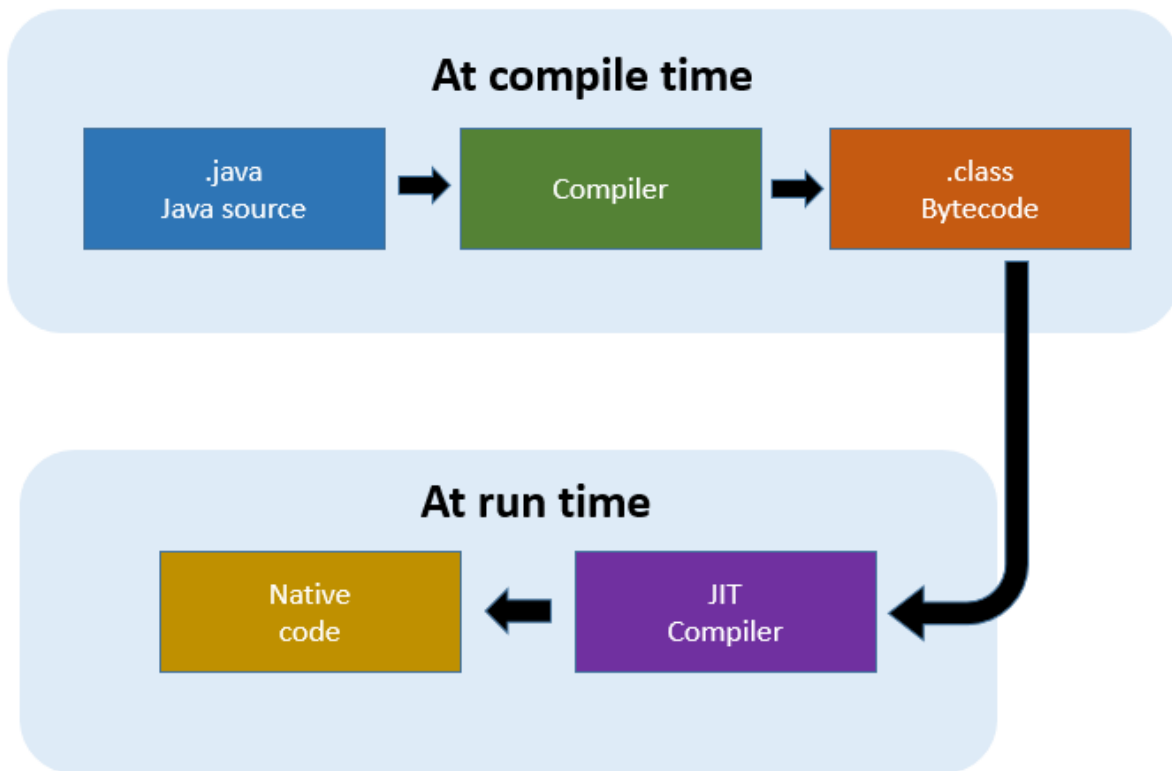
출처 : <https://d2.naver.com/helloworld/1230>

JIT 컴파일러란 무엇이며 어떻게 동작하는지

<https://www.notion.so/1-JVM-475cb3ef27c0487094d01450fb060804#0d8cfa70ff7348058660d9d2c3a180d1>

JIT(Just-in-Time) 컴파일러는 **바이트코드를 컴퓨터 프로세서(CPU)로 직접 보낼 수 있는 명령어 (기계어)**로 바꾸는 프로그램이다.

바이트코드를 읽어 빠른 속도로 기계어를 생성할 수 있다. 이런 기계어 변환은 **코드가 실행되는 과정에 실시간**으로 일어나며(그래서 Just-In-Time이다), 전체 코드의 필요한 부분만 변환한다. 기계어로 변환된 코드는 캐시에 저장되기 때문에 **재사용시 컴파일을 다시 할 필요가 없다**.



자바는 왜 느린가?? 🐢

자바의 성능에 대한 이슈는 항상 존재했었다. 아마, 다른 인터프리터 언어들이 하드웨어가 지금 처럼 뛰어나지 않았을 때에 굉장히 만연했던 이야기일 것이다.

자바가 느린 이유는 자바의 특징인 **인터프리터**를 하는 과정과, 그 전에 **컴파일 과정**을 한번 거치기 때문에 그러한 것인데, 이 점으로 여러 진영에서 공격을 당하고 있다.

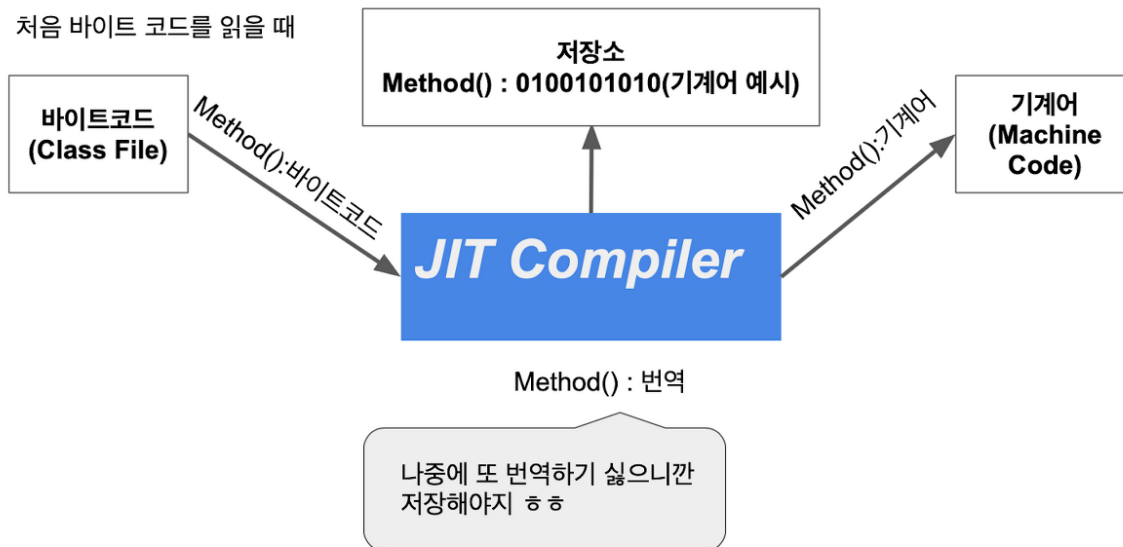
파이썬 vs 자바 성능 차이

- 바이트코드로 컴파일하는 과정이 성능에 영향을 미치는데, 파이썬은 그러지 않다 내용일 것이다.

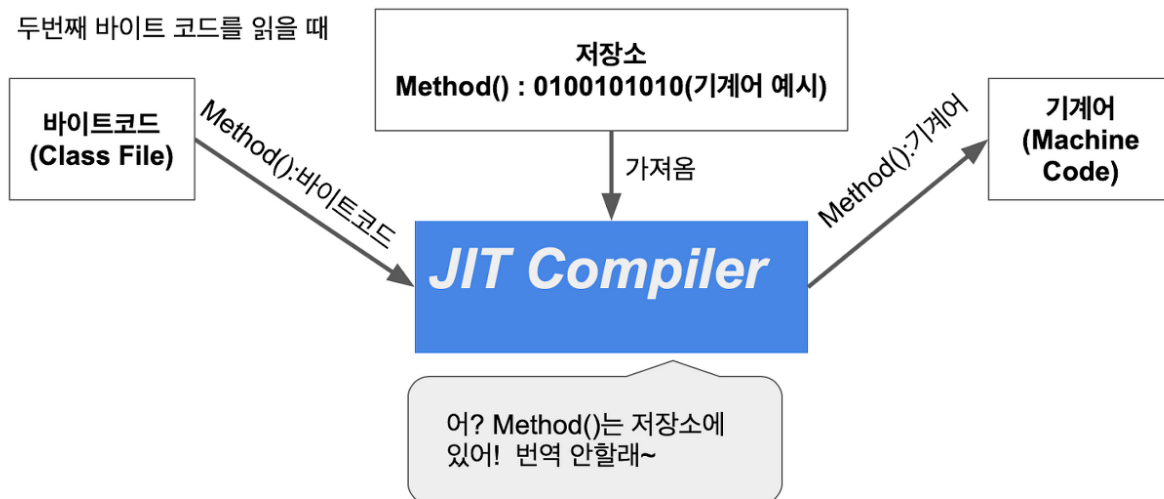
(1) 컴파일 방식 : 소스코드를 한꺼번에 컴퓨터가 읽을 수 있는 native machine (기계)어로 변환

(2) 인터프리터 방식 : 소스코드를 빌드시에 암것도 하지 않다, 런타임시에 한줄 한줄 읽어가며 변환

인터프리터 방식은 소스코드를 런타임시에 한줄 한줄 읽어서 느린데, 이를 해결하고자 JIT를 도입한 것이다.



- 저장소에 저장을 해서, 반복되는 것은 캐시를 사용하는 것 처럼 사용한다.



JIT 내부 동작에 대한 구체적인 설명

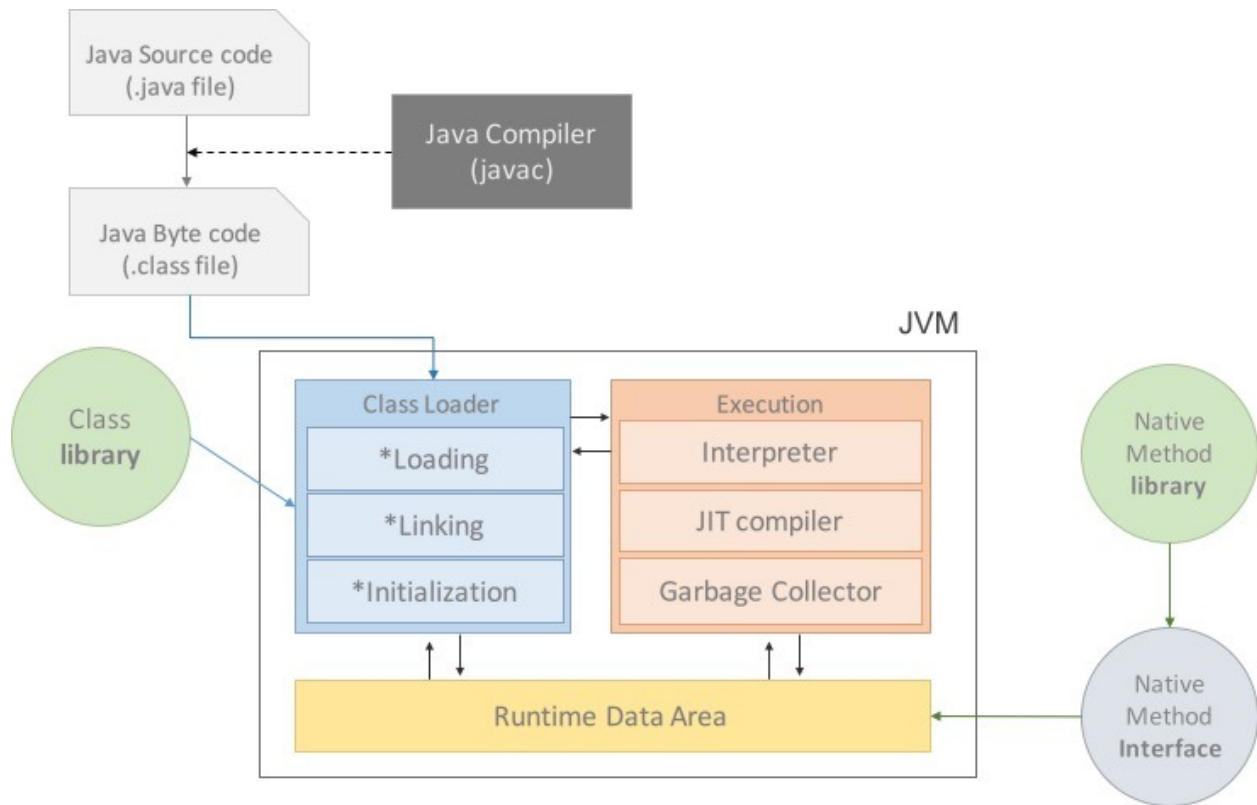
출처 : <https://catch-me-java.tistory.com/11?category=438116>

JVM 구성 요소

자바프로그램 실행 과정

1. 프로그램이 실행되면 JVM은 OS로부터 프로그램에 필요한 메모리를 할당받는다.
JVM은 이 메모리를 용도에 따라 여러 영역으로 나눠 관리한다.
2. 자바 컴파일러(`javac`)가 자바 소스코드(`.java`)를 읽어들이 자바 바이트코드(`.class`)로 변환
3. Class Loader를 통해 class파일들을 JVM으로 로딩한다.
4. 로딩된 class 파일들은 Execution engine을 통해 해석된다.
5. 해석된 바이트코드는 Runtime Data Areas에 배치돼 실질적인 수행이 이루어지게 된다.
이러한 실행과정 속에서 JVM은 필요에 따라 Thread Synchronization, GC같은 관리작업을 수행

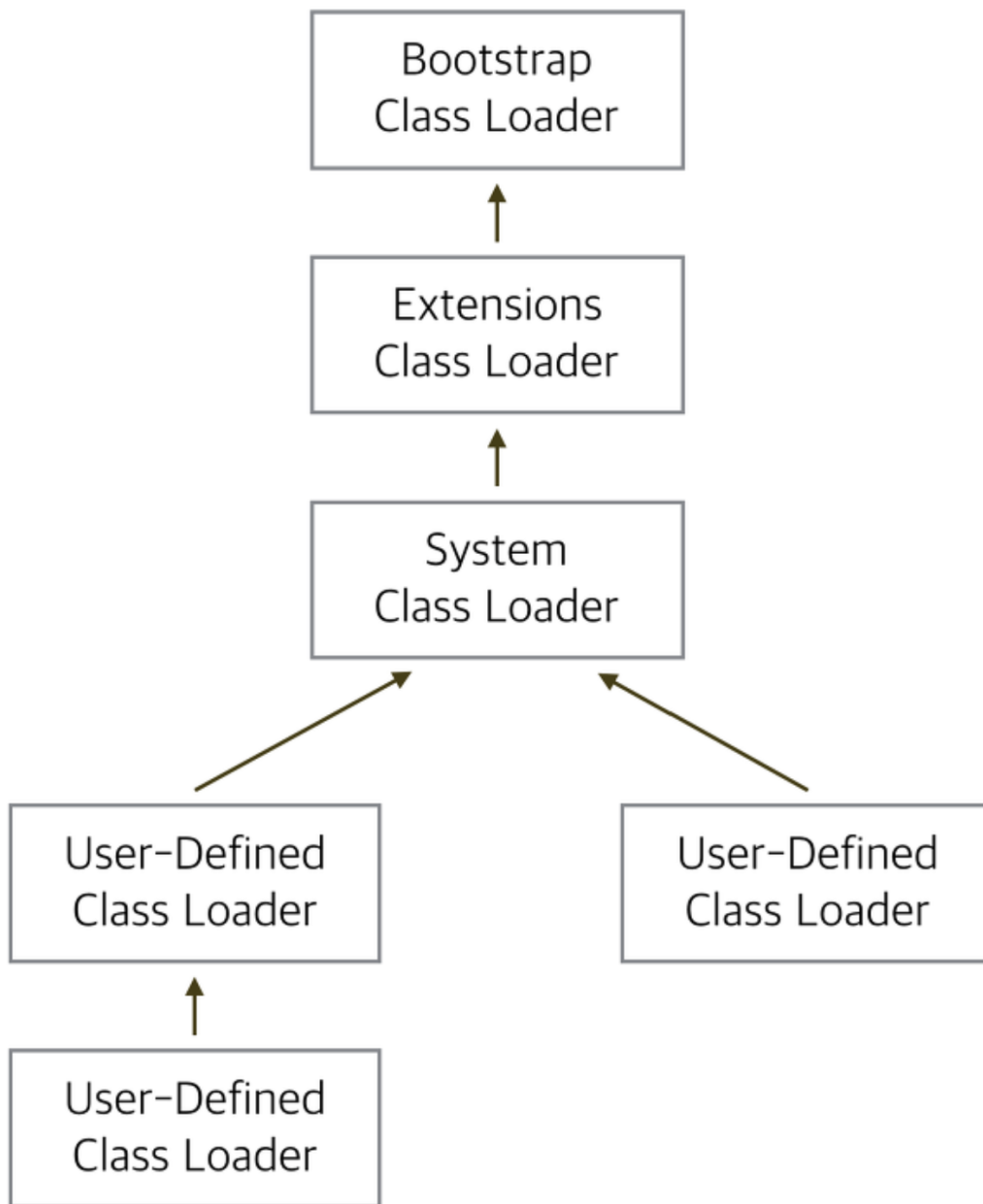
JVM은 크게 ClassLoader, GC, Runtime Data Area, Execute engine으로 나뉜다.



Class Loader

자바 컴파일러가 .java 파일을 컴파일하면 .class 파일(바이트 코드)가 생성되는데 이 클래스 파일들을 읽어 Execution Engine이 사용할 수 있도록 **Runtime Data Area 형태로 메모리에 적재**하는 역할을 한다.

클래스 로더의 구조



- **Bootstrap Class Loader** - JVM이 실행될 때 실행 되는 클래스 로더로, **\$JAVA_HOME/jre/lib** 에 있는 JVM을 실행할 때 **가장 기본이 되는 라이브러리들을** 로드 하는 클래스 로더입니다. 다른 클래스 로더와 다르게 자바가 아닌 **네이티브로 구현된** 클래스 로더입니다.
- **Extensions Class Loader** - 추가로 로딩되는 클래스들로 **\$JAVA_HOME/lib/ext/*.jar**에 있는 클래스들을 로드 합니다. 이 클래스들은 별도의 **CLASSPATH 설정 없이도** 로딩이 됩니다

다.

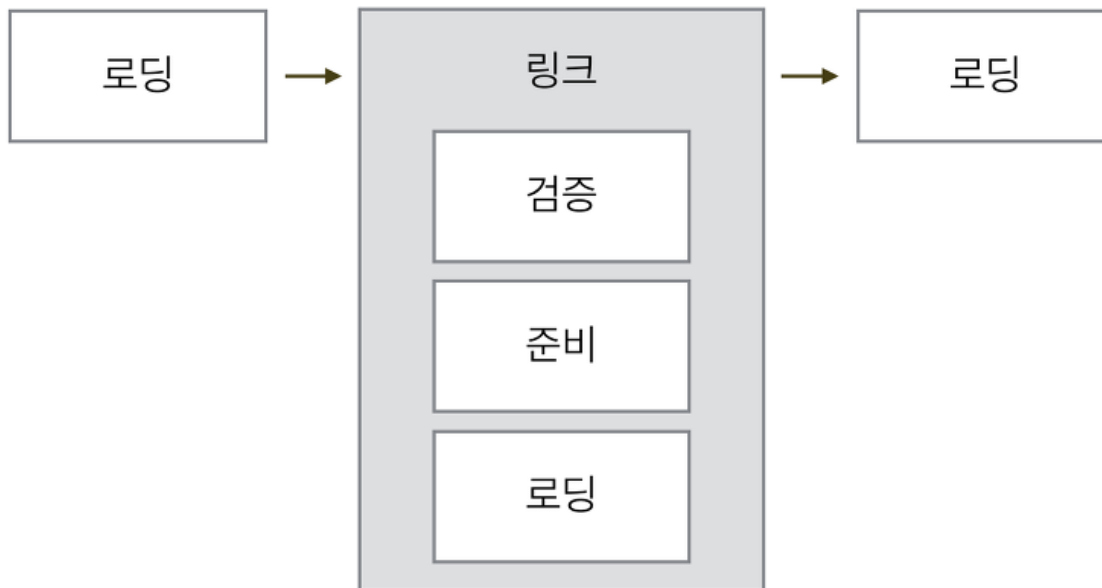
- **System Class Loader** - **CLASSPATH**에 정의 되거나 JVM 옵션에서 -cp, classpath에 설정된 클래스들을 로드 합니다.
- **User-Defined Class Loader** - 애플리케이션 코드에 사용자가 직접 정의해서 사용하는 클래스 로더입니다.

클래스 로더 설명

- 각 클래스 로더들은 부모와 자식 형태의 계층적인 모델을 취한다.
- 가장 최상위 로더는 Bootstrap Class Loader이다.
- 클래스를 로드 할 때에는 parent-first/child-last 순서로 로드한다.
- 캐시 → 부모 → 자식 순서로 클래스 로딩이 된다.
- 캐시에 해당 클래스가 없다면, 부모 클래스 로더 → 부모의 부모 클래스로 가서 로딩함
- 자식 → 부모의 클래스는 찾을 수 있지만 부모 → 자식은 못찾는다.
- 마지막으로 로드 된 클래스는 언로드 될 수 없다.

클래스 로딩 과정

자바의 클래스 로딩은 세부적으로 로딩, 링크, 초기화라는 세 단계 과정을 거친다.



- **로딩** : 클래스 파일을 바이트 코드로 읽어 **메모리로 가져오는** 과정
- **링크** : 가장 복잡한 과정으로, 읽어본 바이트 코드가 **자바 규칙을 따르는지 검증**하고, 클래스에 정의된 **필드, 메소드, 인터페이스**들을 나타내는 **데이터 구조를 준비**하며, 그 클래스가 **참조하는 다른 클래스를 로딩**한다.
- **초기화** : 슈퍼 클래스 및 정적 필드를 초기화한다.

클래스 로딩을 위한 JVM의 로딩 절차

1. 어떤 메소드를 호출하는 문장을 만났는데, 그 메소드를 가진 클래스 바이트코드가 아직 로딩된 적이 없다면, 곧바로 JVM은 JRE 라이브러리 폴더에서 클래스를 찾는다.
2. 없으면, CLASSPATH 환경변수에 지정된 폴더에서 클래스를 찾는다.
3. 찾았으면, 그 클래스 파일이 올바른지 바이트코드 검증한다.
4. 올바른 바이트코드라면 메소드영역으로 파일을 로딩한다.
5. 클래스 변수를 만들라는 명령어가 있으면 메소드 영역에 그 변수를 준비한다.

6. 클래스 블록이 있으면 순서대로 그 블록을 실행한다.
7. 이렇게 한번 클래스의 바이트코드가 로딩되면 JVM이 종료될때까지 유지된다.

Execution Engine

Execution Engine 이란?

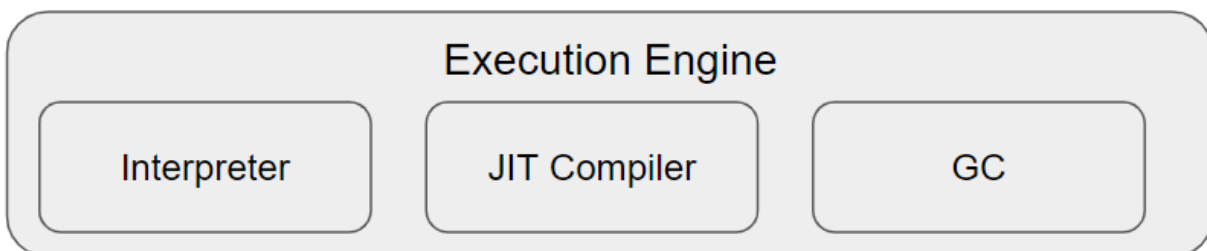
Class Loader에 의해 JVM으로 로드된 .class들은 Runtime Data Areas의 Method Area에 적재한다.

Execution Engine은 메모리에 할당된 바이트코드를 실행하는 역할이다.

바이트코드는 human-readable 형태를 띄는데, 이를 Machine이 읽을 수 있는 형태로 변환해 준다.

바이트코드를 한 줄 마다 읽으며 변환하는 작업을 거친다.

Execution Engine의 구성 요소



Interpreter

- 바이트 코드를 기계가 이해할 수 있도록 Native Code로 바꾸는 작업
- 중복되는 바이트 코드들에 대해서는 JIT 컴파일러를 사용한다

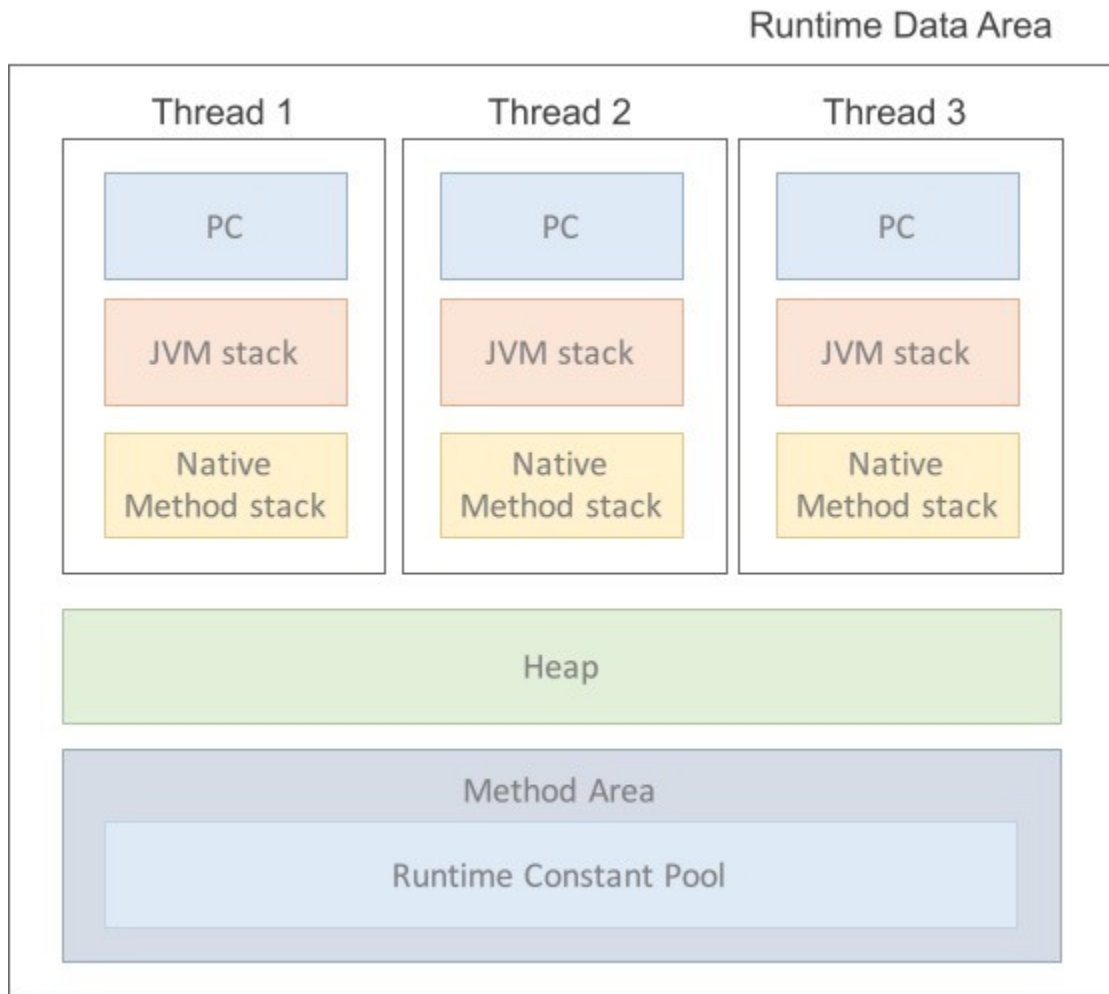
JIT (Just In Time) Compiler

Garbage Collector (GC) - 참조, 2

- Runtime Data Area의 Heap 영역의 더 이상 참조되지 않는 객체를 정리한다.

Runtime Data Area

프로그램을 수행하기 위해 OS에서 할당받은 메모리 공간



▼ PC register와 두개의 Stack은 각 Thread별로 생성, Method Area, Heap은 모든 Thread에 공유됨

1) PC Registers

- Thread가 생성될 때 마다 생기는 공간, **Thread가 어떤 명령을 실행하게 될지에 대한 부분 기록**
- JVM은 Stacks-Base 방식으로 작동하는데, CPU에 직접 Instruction을 수행하지 않고, **Stack에서 Operand를 뽑아내 이를 별도의 메모리 공간에 저장하는 방식**을 취한다. 이 때 **메모리 공간이 PC Register**이다.

2) JVM Stack 영역

- **Thread 제어**를 위해 사용되는 메모리 영역이다.
- **프로그램 실행과정에서 임시로 할당되었다가 메소드를 빠져나가면 바로 소멸되는** 특성이 있다.
- 각종 형태의 **변수, 임시 데이터, 스레드나 메소드 정보**를 저장한다.
- 메소드 호출 시마다 **각각의 스택 프레임(그 메서드만의 공간)**이 생성된다.
- 메소드 수행이 끝나면 **프레임별로 삭제**한다.
- 메소드 안에서 사용되는 **값들(local variable)**을 저장한다.
- 또 **호출된 메소드의 매개변수, 지역변수, 리턴 값 및 연산 시 일어나는 값들**을 임시 저장한다.

⚠ 주의!

멀티 Thread 프로그램의 경우 각 Thread가 자신의 Stack을 가지고는 있지만 Heap 영역은 공유하기 때문에, 프로그래밍시에 Thread-safe 하지 않는 이슈에 주의하며 프로그래밍을 해야 한다. 결론적으로 Heap 영역 자체가 Thread-safe 하지 않는 상태이다. Thread-safe 하게 객체를 생성하기 위해서는 Immutable한 객체를 설계하는 것이 좋다.

3) Native method stack

- 자바 프로그램이 컴파일되어 생성되는 바이트코드가 아닌 실제 실행할 수 있는 기계어로 작성된 프로그램을 실행시키는 영역이다.

- Java가 아닌 다른 언어로 작성된 코드를 위한 공간
- Java Native Interface(JNI)를 통해 바이트 코드로 전환하여 저장하게 된다.
- 일반 프로그램처럼 커널이 스택을 잡아 독자적으로 프로그램을 실행시키는 영역이다.
- 이 부분을 통해 C code를 실행시켜 커널에 접근할 수 있다.

4) Heap

- **new 연산자**로 생성된 객체와 배열을 저장한다.
- 사용자가 관리하는 인스턴스가 생성되는 공간. 객체를 동적으로 생성하면 인스턴스가 할당 됨
- 프로그램은 시작될 때 **미리 Heap 영역을 많이 할당**해 놓으며, 인스턴스와 인스턴스 변수가 저장된다.
- **레퍼런스 변수**의 경우 Heap에 인스턴스가 저장 되는것이 아니라 **포인터가 저장**된다.

| Heap 영역은 GC의 대상이 되는 영역 (아래의 Method Area도 포함)

Heap & Stack

5) Method Area (= Class area = Static area)

클래스 정보를 처음 메모리 공간에 올릴 때 초기화되는 대상을 저장하기 위한 메모리 공간

프로그램 실행 중 클래스가 사용되면 JVM은 해당 클래스 파일을 읽어서 분석하여 클래스의 인스턴스 변수, 메소드 코드 등을 Method Area에 저장한다. 이 때 클래스 변수도 이 영역에 함께 생성된다.

프로그램이 실행되면 모든 코드가 저장되어 있는 상태가 아니다. new 키워드를 통해 객체가 동적으로 생성되기 이전에는 텍스트일 뿐이다.

객체 생성 후에 메소드를 실행하게 되면 해당 클래스 코드에 대한 정보를 Method Area에 저장하게 된다. 저장되는 내역은 아래와 같다.

Type Information

Type은 클래스와 인터페이스를 통칭하는 것으로 이해하면 된다.

- Type의 전체 이름 (패키지명 + 클래스명)
- Type의 직계 하위 클래스 전체 이름
- Type의 클래스 / 인터페이스 여부
- Type의 modifier (public / abstract / final)
- 연관된 인터페이스 이름 리스트

Runtime Constant Pool

Type의 모든 상수 정보를 갖고 있다.

- Type, Field, Method로의 모든 Symbolic Reference 정보를 포함
- Constant Pool의 Entry는 배열과 같이 인덱스 번호를 통해 접근
- Object의 접근 등 모든 참조를 위한 핵심 요소

Field Information

Field는 인스턴스 변수를 가르킨다.

- Field Type
- Field modifier (public / private / protected / static / final / volatile / transient)

Class Variable

Class 변수는 static 키워드로 선언된 변수를 의미 한다.

- 모든 인스턴스에 공유되며 인스턴스가 없어도 직접접근이 가능하다.
- 이 변수는 인스턴스의 것이 아니라 클래스에 속하게 된다.
- 클래스를 사용 하기 이전에 이 변수들은 미리 메모리를 할당 받아 있는 상태가 된다.
- final class 변수는 상수로 치환 되어 Runtime Constant Pool에 값을 복사한다.

Static 변수는 Method Area의 Class Variable에 저장
기본형이 아닌 static 클래스형 변수는 레퍼런스 변수만 저장되고 실제 인스턴스는 Heap에 저장되어 있다. 그 후 이 인스턴스의 변수를 저장하기 위해 Heap에 메모리가 확보되고 Heap의 인스턴스가 Method Area의 어느 클래스 정보와 연결되는지 설정 하게 된다.

출처 : <https://asfirstalways.tistory.com/158>

클래스 로더의 구조

Execution Engine

자바 동적로딩

Runtime Data Area

참고 1

참고 2

JDK와 JRE의 차이



JRE (Java Runtime Environment)

OS위에서 실행되는 소프트웨어 계층, 자바 프로그램을 실행시키는데 필요한 class 라이브러리와 리소스들을 제공한다.

JRE란 ?

JRE는 JDK를 사용하여 생성된 자바 코드를 JVM에서 실행하는 데 필요한 라이브러리와 결합하고 나서 결과 프로그램을 실행하는 JVM의 인스턴스를 만든다. JVM들은 여러 OS에서 사용될 수 있으며 JRE로 작성된 프로그램은 모든 OS에서 실행된다. 이렇게 JRE는 Java 프로그램을 수정 없이 모든 OS에서 실행할 수 있게 한다.

JRE의 동작

JDK와 JRE는 상호 작용하여 거의 모든 OS에 Java 기반 애플리케이션을 원활하게 실행할 수 있는 지속 가능한 런타임 환경을 만든다. 아래에는 JRE의 구성을 나타낸다.

1) Class Loader

- 클래스 로더는 동적으로 자바 프로그램 실행에 필요한 모든 클래스들을 로드한다.
- Java 클래스는 필요할 때만 메모리에 로드되므로, JRE는 클래스 로더를 사용해 요청 시 프로세스를 자동화한다.

2) Bytecode verifier

- 인터프리터로 전달되기 전에 자바 코드의 포맷과 정확성을 보장한다.
- 코드가 시스템 무결성 혹은 액세스 권한을 위반하는 경우 클래스는 손상된 것으로 간주되어 로드되지 않는다.

3) Interpreter

- 바이트 코드가 성공적으로 로드된 후, Java Interpreter는 Java 프로그램을 기본 컴퓨터에서 기본적으로 실행할 수 있는 JVM 인스턴스를 만든다.

JRE 구성

JVM외에도 JRE는 Java 애플리케이션을 최대한 활용할 수 있는 다양한 지원 소프트웨어 도구와 기능으로 구성되어 있다.

Development toolkits

- Java 2D, AWT(Abstract Window Toolkit), Swing

Integration libraries

- CORBA(Java IDL), JDBC(Java Database Connectivity) API, Java Naming and Directory Interface (JNDI)

Language and utility libraries

- Collection Framework, Concurrency Utilities, Preferences API, Logging, Java Archive (JAR)

JDK

자바 어플리케이션 **개발에 필요한 도구들의 집합**이다. 모든 JDK는 JRE를 갖는다.

- 컴파일러, jdb(Java Debug), javadoc과 같은 도구도 있다.
- 프로그램을 생성하고 컴파일할 수 있다.

what is JRE

후기

평소 자바를 할 때 JDK를 깔고 사용한다는 것은 인식하곤 있었으나 구체적으로 궁금하진 않았다.

하지만 이에 대해 공부를 하며, 더 깊은것들이 궁금해졌고 찾아보는 시간이 되었다.

JVM이 왜 필요하고, Java가 어떤것을 목표로 JVM을 만들었는지에 대해 알게 되었을때 신기하고 재밌었다.

.java 파일에서 .class로, 그 다음 기계어로 바뀌는 과정을 알게 되었고 각 단계에서 어떤 동작이 이뤄지는지 잘 알게 되었다.

JVM, JRE, JDK 유튜브 link!