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

들어가기에 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.

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 실행**하는 JVM이다. JVM의 역할은 **자바 애플리케이션을 클래스 로더를 통해 읽어들여서 자바 API와 함께 실행**하는 것이다.

> 자바 애플리케이션 해석을 클래스 로더 → 자바 API와 함께 실행 ?

JVM의 특징

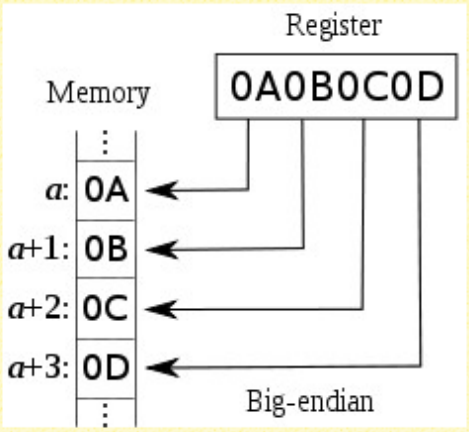
- **스택 기반**의 가상 머신
- ***심볼릭 레퍼런스** : 기본 자료형(primitive data type)을 제외한 모든 타입(클래스, 인터페이스)을 명시적인 메모리 주소 기반의 레퍼런스가 아니라, `심볼릭 레퍼런스를 통해 참조` 한다.
- **가비지 컬렉션**(GC, garbage collection) : 클래스 인스턴스는 사용자 코드에 의해 명시적으로 생성되고 가비지 컬렉션에 의해 자동으로 파괴된다
- **기본 자료형을 명확하게 정의하여 플랫폼 독립성 보장** : C/C++ 등의 전통적인 언어는 플랫폼에 따라 int형의 크기가 변한다. JVM은 **기본 자료형을 명확하게 정의**하여 **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

> 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 코드**를 실행가능한 **바이트코드로 변환**한다.
- **명령어**와 **피연산자**를 포함하는 **데이터구조**를 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**
- **피연산자**가 **스택 포인터**에 의해 **암시적으로 처리됨**
- **코드 작성**과 **컴파일**이 **쉽고** 가상머신이 **빠름**

[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 할당**하고 작업하여 **작업량과 CPU 시간을 크게 줄일 수 있다**.
- 스택으로 밀고 들어가는 **오버헤드가 존재하지 않음**
- 같은 연산을 할 때 **더 적은 명령어로 연산 수행**이 가능하다.
- **일부 최적화** 가능

❓ 심볼릭 레퍼런스

- 프로그램 실행을 위한 API를 클래스가 직접 가지는 것이 아닌 **이름을 통한 참조값**을 이용해서 실행할 때 메모리 상에서 API를 호출할 수 있도록 **이름을 주소로 대체하는 특징**을 의미

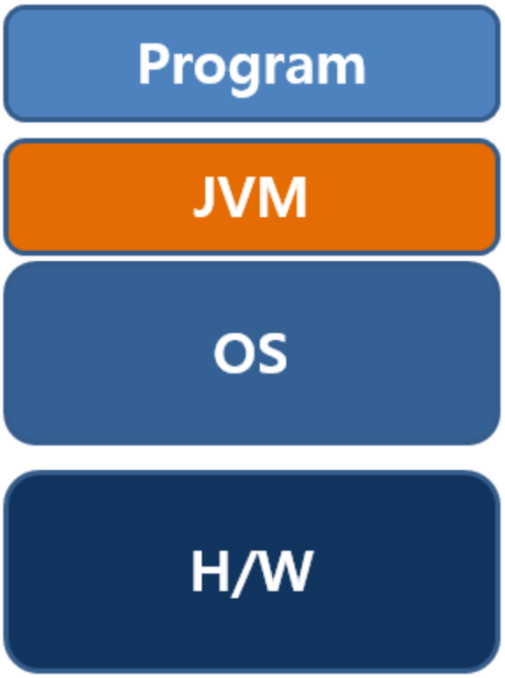
출처 : https://d2.naver.com/helloworld/1230

https://s2choco.tistory.com/13

컴파일 하는 방법

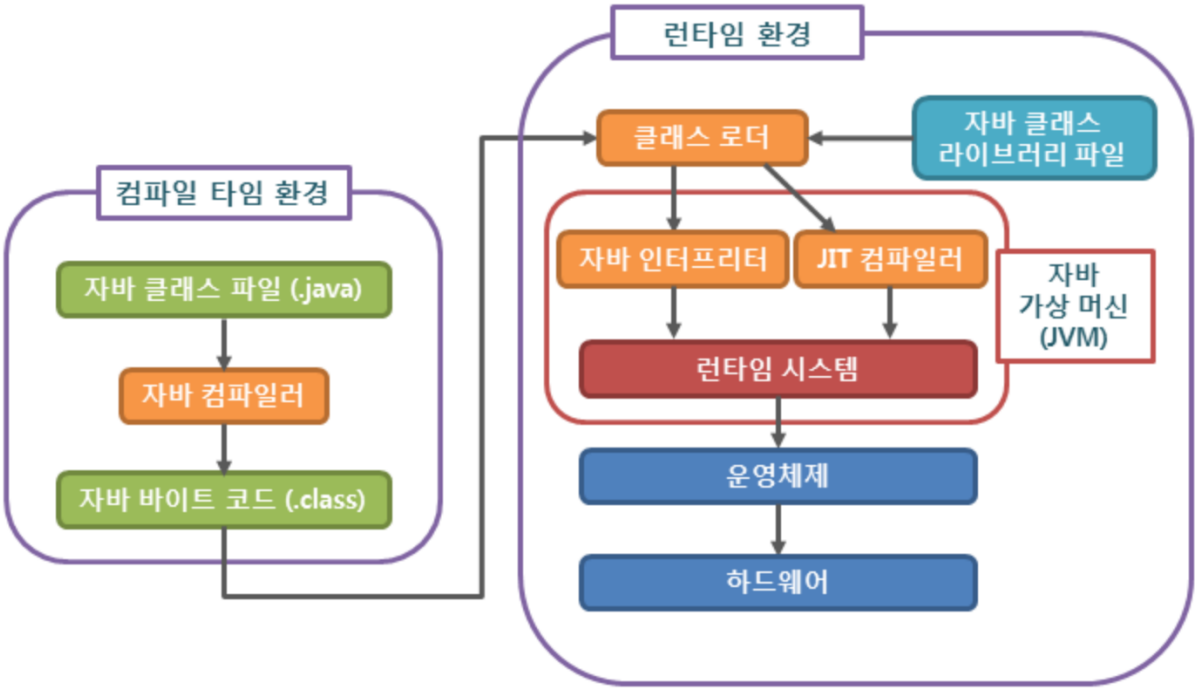
> 자바는 **OS에 독립적인** 특징을 가지고 있다. 그것을 가능하게 한 것이 JVM이다. 이를 어떻게 가능하게 했는지 자바 컴파일 과정을 통해 알아보자.

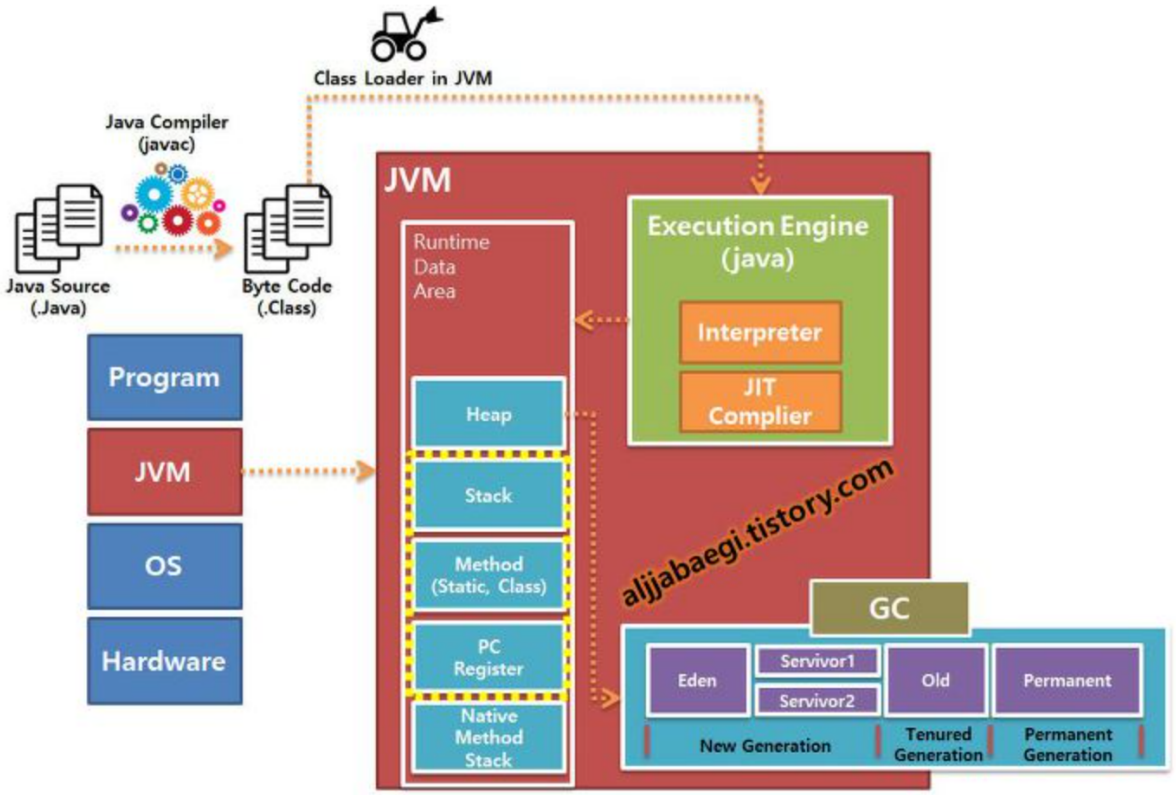
< 일반적인 프로그램 >



< JAVA 프로그램 >

컴파일 순서


런타임 환경
클래스 로더
자바 클래스
라이브러리 파일
컴파일 타임 환경
자바 클래스 파일 (.java)
자바 컴파일러
자바 바이트 코드 (.class)
자바 인터프리터
JIT 컴파일러
자바
가상 머신
(JVM)
런타임 시스템
운영체제
하드웨어


Class Loader in JVM
Java Compiler
(javac)
Java Source
(.Java)
Byte Code
(.Class)
Program
JVM
OS
Hardware
JVM
Runtime
Data
Area
Heap
Stack
Method
(Static, Class)
PC
Register
Native
Method
Stack
Execution Engine
(java)
Interpreter
JIT
Complier
alijabaegi.tistory.com
GC
Eden
Servivor1
Servivor2
Old
Permanent
New Generation
Tenured
Generation
Permanent
Generation

1. 자바 **소스코드(.java)를 작성**한다.
2. 자바 **컴파일러(java Compiler)**가 자바 **소스파일을 컴파일**한다.

   → 이때 나오는 파일은 자바 **바이트 코드(.class)파일**로 아직 **컴퓨터가 읽을 수 없는 자바 가상 머신이 이해할 수 있는 코드**이다. 바이트 코드의 각 명령어는 1**바이트 크기의 Opcode와 추가 피연산자**로 이루어져 있다.
3. 컴파일된 **바이트 코드**를 **JVM의 클래스로더(Class Loader)**에게 전달.
4. 클래스 로더는 **동적로딩(Dynamic Loading)**을 통해 **필요한 클래스**들을 **로딩 및 링크**하여 **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.

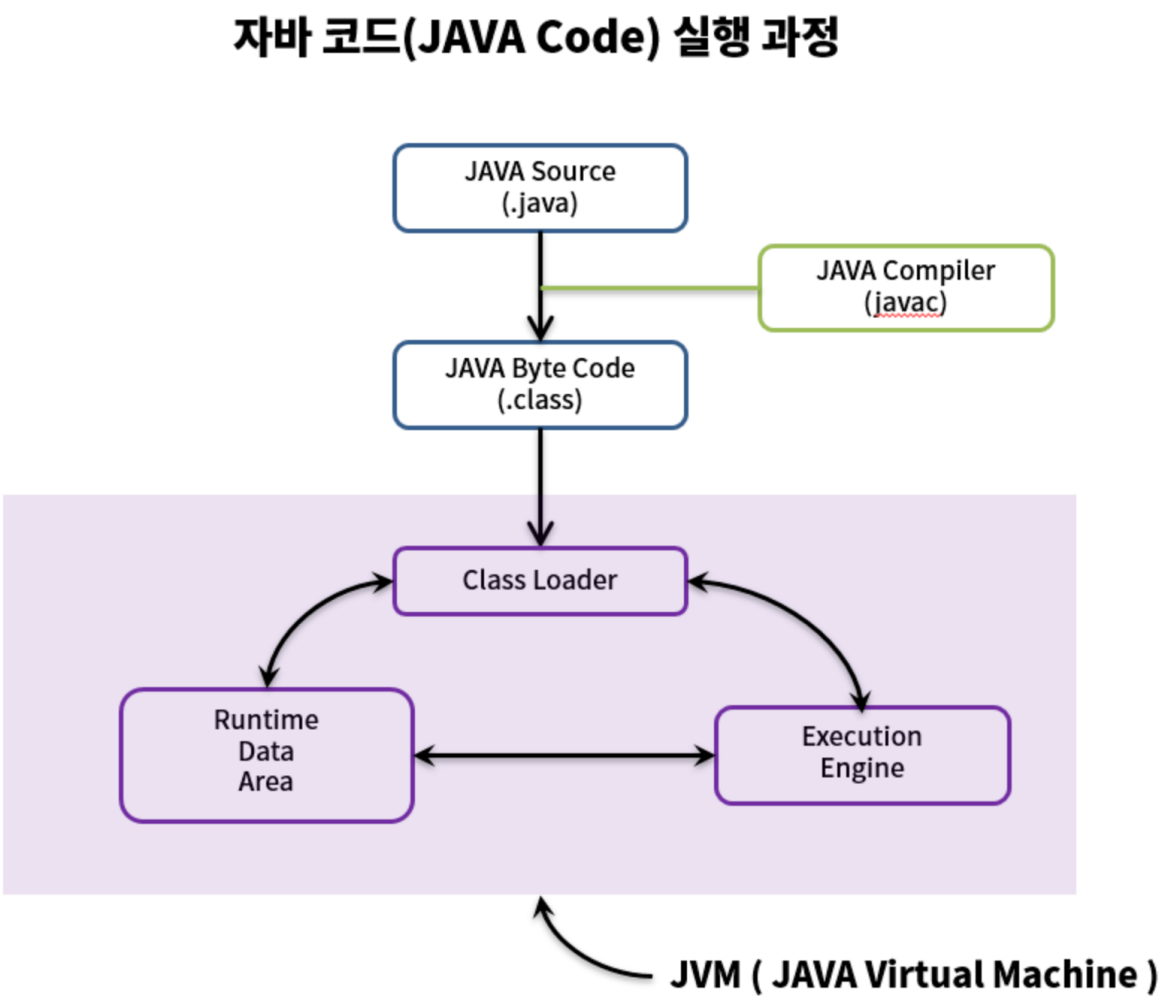
   c. **준비** - 클래스가 필요로 하는 메모리를 할당(필드, 메서드, 인터페이스 등등)

   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 실행**한다. 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 컴파일**하여 **바이너리 코드로 변경**하고 이후에는 해당 메서드를 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.**



실행하는 방법

바이트코드란 무엇인가

JIT 컴파일러란 무엇이며 어떻게 동작하는지

JVM 구성 요소

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

JDK와 JRE의 차이