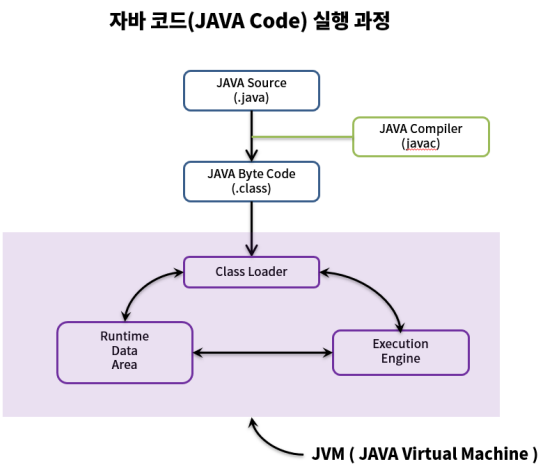
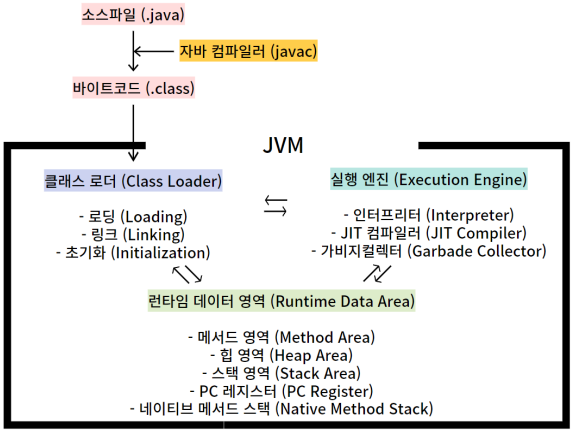
컴파일 과정





1. 자바 컴파일러(Java Compiler)가 자바 소스(.java)를 자바 바이트 코드(.class)로 컴파일

   ▼ 자바 바이트 코드(.class)

   - JVM이 이해할 수 있는 저수준 언어(반기계어), 컴퓨터는 이해 불가
   - 바이트 코드의 각 명령어는 1byte 크기의 opcode와 추가 피연산자로 이루어짐
   - 기존의 언어의 컴파일 결과로 생성되는 오브젝트파일(.obj)과 달리, 바이트코드(.class)는 모든 플랫폼의 JVM에서 실행가능

     (⇒ Java만 설치되어 있다면 Windows환경에서 생성된 바이트코드는 Mac 환경에서도 실행이 가능)
   - 바이트 코드 변환을 통해서 CPU 및 OS에 독립적으로 수행되지만, 기계어 코드를 직접 읽는 것보다 느림
   - *javap -c* (.class file) 명령을 통해 바이트 코드 확인 가능
   - *javap* 명령은 클래스 파일의 필드, 생성자, 메소드를 출력
2. 컴파일된 바이트 코드를 JVM의 클래스로더(Class Loader)에게 전달

3. 클래스 로더는 동적로딩(Dynamic Loading)을 통해 필요한 클래스들을 로딩 및 링크하여 런타임 데이터 영역(Runtime Data area), 즉 JVM의 메모리에 올림.

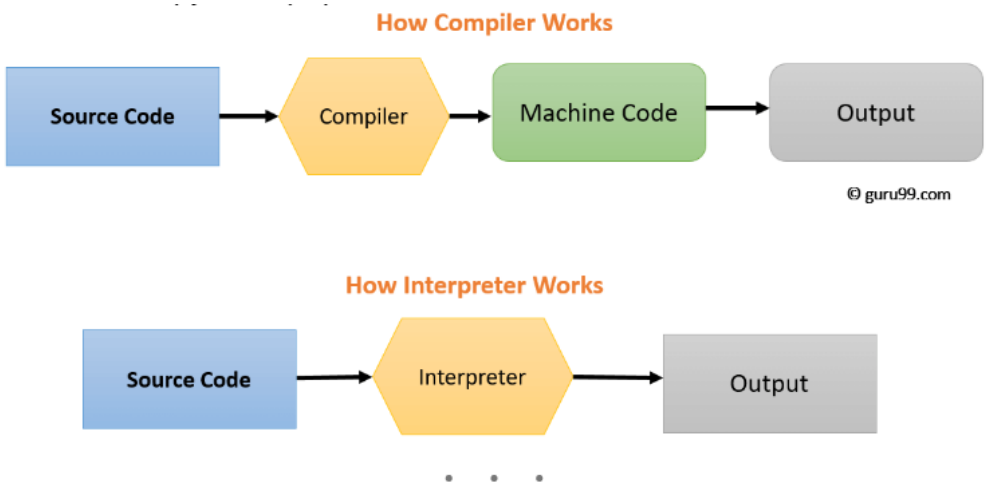
    ▼ 클래스 로더 세부 동작

    1. **로드** : 클래스 파일을 가져와서 JVM의 메모리에 로드
    2. **링크** : 자바 언어 명세(Java Language Specification) 및 JVM 명세에 명시된 구성 요소 검증. 클래스가 필요로 하는 메모리 할당 (필드, 메서드, 인터페이스 등)
    3. **분석** : 클래스의 상수 풀 내 모든 심볼릭 레퍼런스를 다이렉트 레퍼런스로 변경
    4. **초기화** : 클래스 변수들을 적절한 값으로 초기화 (static 필드)

4. 실행엔진(Execution Engine)은 JVM 메모리에 올라온 바이트 코드들을 명령어 단위로 하나씩 가져와서 실행.

    ▼ 실행 엔진의 두가지 방식

    ▼ 인터프리터 vs 컴파일러



- JVM은 기본적으로 인터프리터 방식 사용하고, 내부적으로 특정 메서드가 얼마나 자주 수행되는지 체크하고 일정 정도를 넘을 때만 JIT 컴파일러 방식을 사용

1. 인터프리터 방식 - 바이트 코드 한 줄씩 읽기
    - 바이트 코드 명령어를 하나씩 읽어서 해석하고 실행.
    - 하나하나의 실행은 빠르나, 전체적인 실행 속도가 느림.
2. JIT 컴파일러(Just-In-Time Compiler) 방식 - 한번에 컴파일하고 캐싱
    - 인터프리터의 단점을 보완하기 위해 도입된 방식

- 바이트 코드 전체를 컴파일하여 바이너리 코드로 변경하고 이후에는 해당 메서드를 더이상 인터프리팅 하지 않고, 바이너리 코드로 직접 실행하는 방식.
- 하나씩 인터프리팅하여 실행하는 것이 아니라 바이트 코드 전체가 컴파일된 바이너리 코드를 실행하는 것이기 때문에 전체적인 실행속도는 인터프리팅 방식보다 빠름.
- JIT Compiler에 의해 해석된 코드는 캐시에 보관하기 때문에 한 번 컴파일된 후에는 빠르게 수행됨
- but, 인터프리팅 방식보다는 초기 실행이 훨씬 오래 걸리므로 한번만 실행하면 되는 코드는 인터프리팅하는 것이 유리. (반복 실행되는 루프나 메서드 등은 JIT가 유리)

동적 로딩

- 프로그램 실행 중에 클래스를 동적으로 로드하는 기능. 런타임, 필요한 시점에 클래스를 로딩하여 사용할 수 있음.

링크

- 클래스나 메서드 등의 심볼(symbol)과 해당하는 실제 메모리 주소를 연결하는 과정
- 이를 수행하는 프로그램을 링커라고 함.

자바 언어 명세서 (Java Language Specification)

- JLS는 자바 언어를 위한 문법과 정상/비정상적인 규칙들을 보여줌
- 또한 정상적인 프로그램을 실행하기 위한 프로그램 방법들을 보여줌

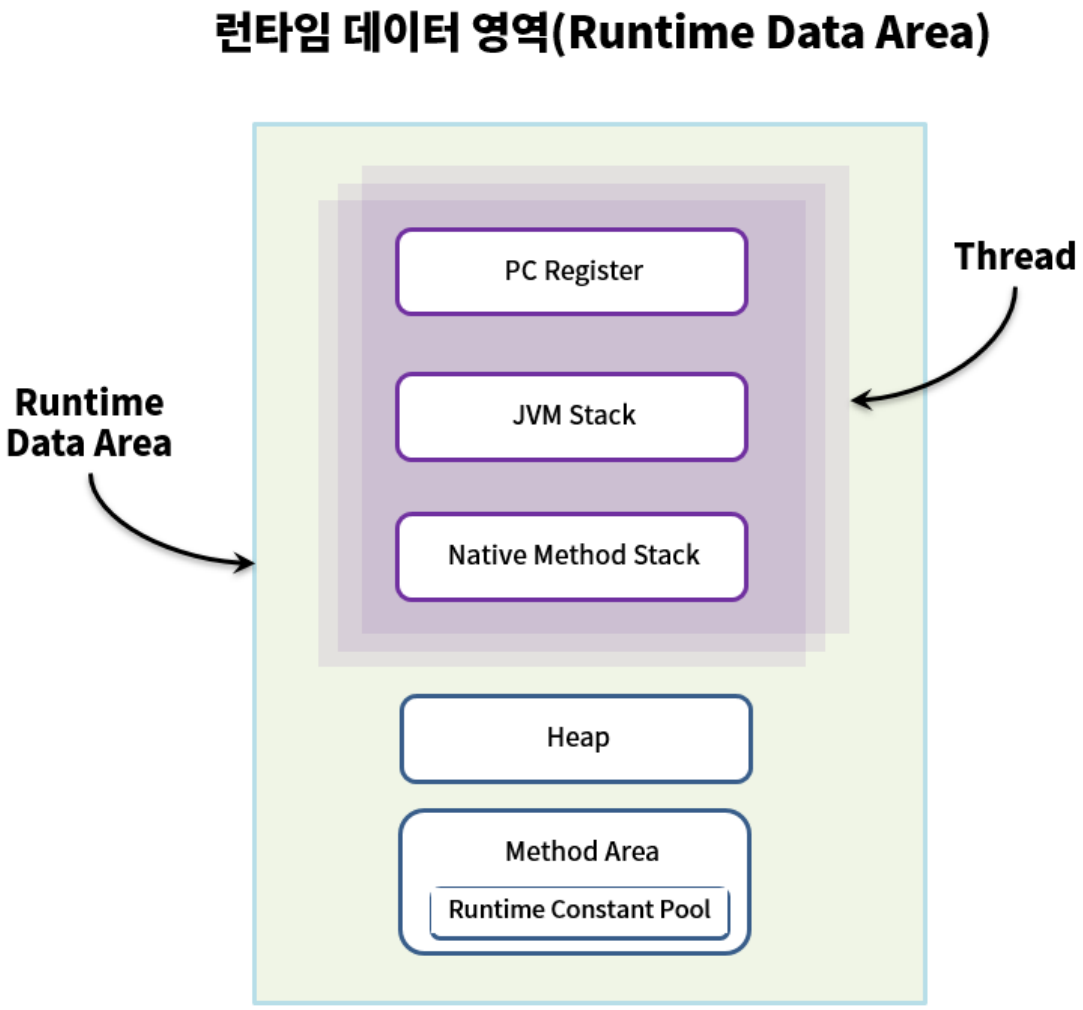
Symbolic Reference

- 실제 메모리 주소 값이 아닌, 참조하는 대상의 이름으로만 지정하고 있는 것이다
- JVM 클래스 로더가 Linking 단계의 분석(Resolving)에서 실제 주소값(다이렉트 레퍼런스)으로 변경한다

상수 풀 : 자바 클래스 파일(.class 파일) 내에 포함된 상수들의 모음

▼ 런타임 데이터 영역 : JVM이 프로그램 실행하는 동안 데이터 저장/관리 하는 공간

- pc레지스터, JVM 스택, 네이티브 메서드 스택은 스레드마다 하나씩 생성됨
- 힙, 메서드 영역은 모든 스레드가 공유해서 사용



- pc 레지스터 : 현재 수행 중인 명령의 주소를 가지며, 스레드가 시작될 때 생성됨
- JVM 스택 : 스택 프레임이라는 구조체를 저장하는 스택. 스레드가 시작될 때 생성됨.
- 네이티브 메서드 스택 : JAVA 외의 언어로 작성된 네이티브 코드를 위한 스택. JNI(JAVA Native Interface)를 통해 호출하는 C/C++ 등의 코드를 수행하기 위한 스택으로, 언어에 맞게 생성됨
- 힙 : 인스턴스 또는 객체를 저장하는 공간으로 가비지 컬렉션 대상. JVM 성능 등의 이슈에서 가장 많이 언급되는 공간.
- 메서드 영역 : JVM이 시작될 때 생성됨. JVM이 로딩 과정에서 읽어들인 각각의 클래스와 인터페이스에 대한 런타임 상수 풀, 필드와 메서드에 대한 정보, static 변수, 메서드의 바이트 코드 등을 저장.
    - 런타임 상수 풀 : JVM 동작에서 가장 핵심적인 역할을 수행하는 곳. 각 클래스와 인터페이스의 상수 뿐만 아니라, 메서드와 필드에 대한 모든 레퍼런스까지

담고 있는 테이블. JVM은 런타임 상수 풀을 통해 특정 메서드나 필드의 실제 메모리상 주소를 찾아서 참조함.

출처

https://steady-snail.tistory.com/67#자바_코드(JAVA_Code)_실행_과정

https://ssocoit.tistory.com/270

https://yummy0102.tistory.com/510