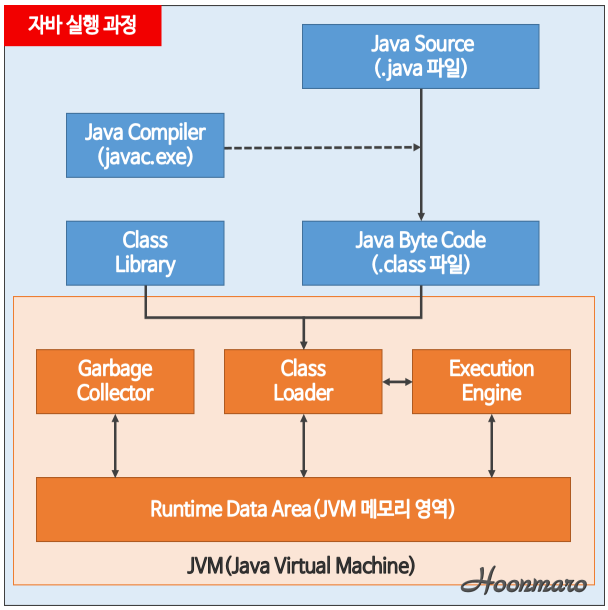
JVM 메모리 구조 및 동작원리

전체적인 자바 실행 과정



출처 : https://hoonmaro.tistory.com/19

1. 실행 할 클래스 파일을 jvm 메모리에 로드 후 초기화
2. 메소드, 클래스 변수들을 해당 메모리 영역에 배치
3. 클래스 로드가 끝난 뒤 main 메소드를 찾아가 지역변수, 객체변수, 참조변수를 스택에 push
4. 프로그램 실행

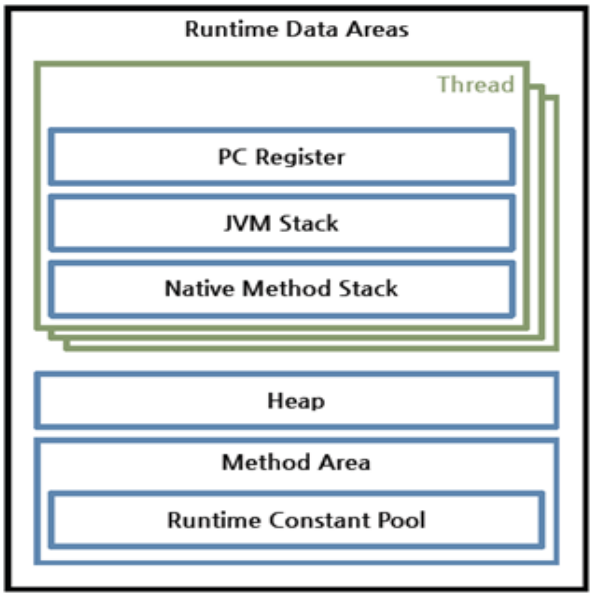
JVM 구조

Class Loader

- JVM 내로 클래스를 로드하고 링크를 통해 배치하는 작업을 수행하는 모듈.
- 컴파일 타임이 아닌 런타임 시 동적으로 클래스를 로드.
- 계층 구조
  - 클래스 로더끼리 부모-자식 관계를 이루어 계층 구조로 생성.
  - 최상위 클래스 로더는 부트스트랩 클래스 로더(Bootstrap Class Loader)
- 위임 모델
  - 계층 구조를 바탕으로 클래스 로더끼리 로드를 위임하는 구조로 동작
  - 클래스를 로드할 때 먼저 상위 클래스 로더를 확인
  - -> **상위 클래스 로더에 있다면** 해당 클래스를 사용하고,
  - -> **없다면** 로드를 요청받은 클래스 로더가 클래스를 로드한다.
- 가시성(visibility) 제한
  - 하위 클래스 로더는 상위 클래스 로더의 클래스를 찾을 수 있지만, 상위 클래스 로더는 하위 클래스 로더의 클래스를 찾을 수 없다.

- 언로드 불가
    - 클래스 로더는 클래스를 로드할 수는 있지만 언로드할 수는 없다. 언로드 대신, 현재 클래스 로더를 삭제하고 아예 새로운 클래스 로더를 생성하는 방법을 사용할 수 있다.

Runtime Data Areas



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

- PC Register

- 이중 pc 레지스터 라고도 함.
- Thread가 생성될 때 마다 생기는 공간
- Thread가 어떠한 명령을 실행하게 될 지에 대한 부분을 기록(현재 수행중인 JVM 명령 주소)
- CPU에 직접 Instruction을 수행하지 않고, Stack에서 Operand를 뽑아내 PC Register에 저장하는 방식

- Stack

- JVM stack과 native method stack이 존재함(여기서는 JVM stack만 설명함. 흔히 말하는 JVM의 stack은 JVM stack)
- 각 thread마다 하나씩 존재하며, thread가 시작될 때 할당됨.
- Stack Frame 이라는 구조체를 저장하는 스택
- JVM은 오직 JVM 스택에 스택 프레임을 추가하고 제거하는 동작만 수행함.
- 모든 primitive type 변수는 스택에 직접 값을 가진다.
- 메소드를 호출할 때마다 프레임을 추가하고 메소드가 종료되면 해당 프레임을 제거 (push, pop)
- 메소드 정보, 지역변수, 매개변수, 연산 중 발생하는 임시 데이터 저장
- 메소드 호출 시 생성되는 스레드 수행정보를 기록하는 Stack Frame을 저장
- **Stack Frame**
    - method가 수행될 때마다 하나의 stack frame이 생성되어 JVM stack에 추가
    - method가 종료되면 stack frame 제거
    - 지역 변수 배열, 피연산자 스택, 실행중인 method가 속한 클래스의 런타임 상수 풀에 대한 레퍼런스를 가짐
    - 지역 변수 배열, 피연산자 스택의 크기는 컴파일 시 결정되기 때문에, stack frame의 크기도 method에 따라 크기가 고정됨.

    - **지역 변수 배열**
        - 0부터 시작하는 인덱스를 가진 배열
        - 0은 method가 속한 클래스 인스턴스의 this reference임.
        - 1부터는 method에 전달된 파라미터들이 저장되며, 파라미터 이후에는 method의 지역변수들이 저장됨.
    - **피연산자 스택**
        - method의 실제 작업 공간
        - 각 method는 피연산자 스택과 지역 변수 배열 사이에서 데이터를 교환하고, 다른 method 호출 결과를 추가하거나 꺼냄.

- Heap

- JVM이 관리하는 프로그램 상에서 데이터를 저장하기 위해 런타임 시 동적으로 할당하여 사용하는 영역.
- new 연산자로 생성된 객체 또는 인스턴스와 배열을 저장한다.
- 힙 영역에 생성된 객체와 배열은 스택 영역의 변수나 다른 객체의 필드에서 참조한다.
- 참조하는 변수나 필드가 없다면 의미없는 객체가 되어 Garbage Collecting의 대상이 된다.
- 힙 영역의 사용기간 및 스레드 공유 범위
    - 객체가 더 이상 사용되지 않거나 명시적으로 null 선언 시
    - GC(Garbage Coolecting) 대상
    - 구성 방식이나 GC 방법은 JVM 벤더마다 다를 수 있음
    - 모든 스레드에서 공유

- Method(Static) Area

- 모든 Thread가 공유하는 영역
- JVM이 시작될 때 생성됨.
- JVM이 읽어들인 클래스와 인터페이스에 대한 런타임 상수 풀, 멤버 변수(필드), 클래스 변수(Static 변수), 생성자와 메소드, 메소드의 바이트코드 등을 저장
- JVM 벤더마다 다양한 형태로 구현
- **Runtime 상수 풀**
    - 클래스 파일 포맷에서 constant_pool 테이블에 해당하는 영역
    - JVM 동작에서 가장 핵심적인 역할을 수행하는 곳
    - Method(Static) Area 영역에 포함되는 영역
    - 각 클래스와 인터페이스의 상수 뿐 아니라, method와 필드에 대한 모든 레퍼런스까지 담고있는 테이블
    - JVM은 Runtime 상수 풀을 통해 해당 메서드나 필드의 실제 메모리 상 주소를 찾아서 참조함.

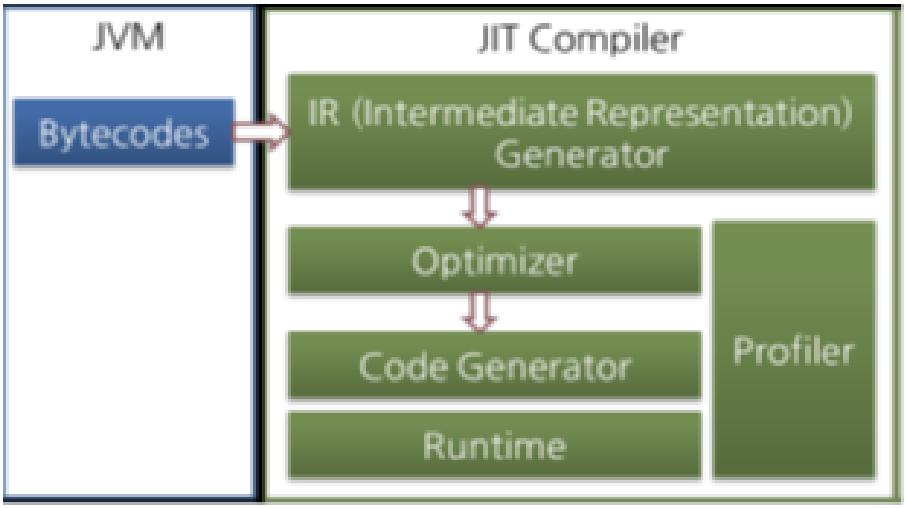
Execution Engine

- Java의 바이트 코드를 명령어 단위로 읽어서 실행(CPU가 기계 명령어를 하나씩 실행하는 것과 비슷)
- 바이트 코드의 각 명령어는 1바이트짜리 OpCode와 추가 피연산자로 이루어짐
- Execution Engine은 하나의 OpCode를 가져와서 피연산자와 함께 작업을 수행한 후, 다음 OpCode를 수행하는 식으로 동작
- 바이트 코드를 두 가지 방식으로 JVM 내부에서 기계가 실행할 수 있는 형태로 변경
- **인터프리터**

- 바이트 코드 명령어를 하나씩 읽어서 해석하고 실행
- 해석은 빠르나 결과의 실행은 느림.
- 즉, 바이트 코드라는 '언어'는 기본적으로 인터프리터 방식으로 동작

- **JIT(Just-In-Time) 컴파일러**
  - 인터프리터의 단점을 보완하기 위해 도입된 것.
  - 인터프리터 방식으로 실행하다가 적절한 시점에 바이트 코드 전체를 컴파일하여 네이티브 코드로 변경
  - 이후, 해당 메서드를 더 이상 인터프리팅 하지 않고 네이티브 코드(기계어)로 직접 실행
  - 컴파일된 네이티브 코드는 캐시에 보관하기 때문에 한 번 컴파일된 코드는 계속 빠르게 수행
  - 바이트 코드를 중간 단계의 표현인 IR(Intermediate Representation)로 변환하여 최적화를 수행
  - 그 후 네이티브 코드 생성.

> JIT 컴파일러가 컴파일 하는 과정은 바이트 코드를 하나 씩 인터프리팅 하는 것보다 훨씬 오래걸리므로, 한 번만 실행되는 코드라면 컴파일하지 않고 인터프리팅하는 것이 훨씬 유리하다.
> 따라서, JIT 컴파일러를 사용하넌 JVM들은 내부적으로 해당 메서드가 얼마나 자주 수행되는지 체크하고, 일정 정도를 넘을 때에만 컴파일을 수행함.



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

Garbage Collector

- 메모리 관리 기능을 자동으로 수행
- 더 이상 사용되지 않는 객체를 해제하는 방식으로 메모리 관리
- Garbage Collector는 기본적으로 두 가지 전제 하에 만들어짐.
  1. 대부분의 객체는 금방 접근 불가능 상태가 된다.
  2. 오래된 객체에서 젊은 객체로의 참조는 아주 적게 존재한다.
- 효율적인 GC를 위한 Oracle HotSpot VM에서의 공간 분리
  - Young 영역 : 새롭게 생성된 객체가 위치하는 공간. 이 영역에서 객체가 사라질 때 Minor GC가 발생한다고 말함.

- Old 영역 : 접근 불가능 상태로 되지 않아 Young 영역에서 살아남은 객체가 여기로 복사됨. 대부분 Young 영역보다 크게 할당하며, GC가 Young 영역보다 상대적으로 적게 발생. 이 영역에서 객체가 사라질 때 Major GC가 발생한다고 말함.