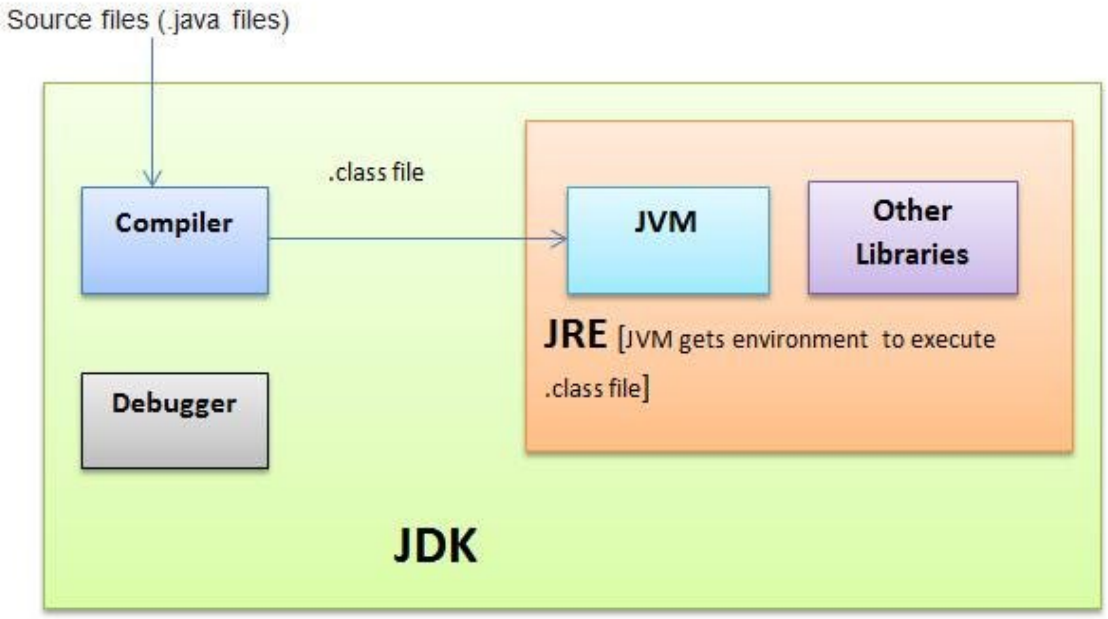
JVM (Java Virtual Machine)

✅ JVM은 **'자바를 실행하기 위한 가상 기계(컴퓨터)'** 이다.

- JVM의 역할은 자바 애플리케이션을 클래스 로더를 통해 읽어 들여 자바 API와 함께 실행하는 것이다.
- 메모리 관리, Garbage collection(가비지 컬렉션)을 수행한다.
- Stack 기반으로 동작한다.
- 자바 애플리케이션이 실행되기 위해서 반드시 필요한 요소로, 자바로 구성된 애플리케이션은 모두 JVM에서만 실행된다.
- 모든 자바 애플리케이션의 코드는 JVM을 거쳐 OS에 전달되기 때문에 OS에 독립적이다.
- 단, JVM은 OS에 종속적이므로 해당 OS에서 실행 가능한 JVM을 설치해야 한다.
- `JRE(Java Runtime Environment)`
  - 자바 응용프로그램 실행을 위한 최소의 환경
  - JVM + API(Application Programming Interface, library)
- `JDK(=JavaSE, Java Development Toolkit)`
  - Compiler를 비롯한 개발에 필요한 여러 도구 + JRE(실행 도구)
  - 컴파일러: java 소스를 byte code로 변환(단, 문법적인 오류가 없는 경우)
  - 인터프리터: byte code인 class 확장자 파일로 실행



출처: https://medium.com/@mannverma/the-secret-of-java-jdk-jre-jvm-difference-fa35201650ca

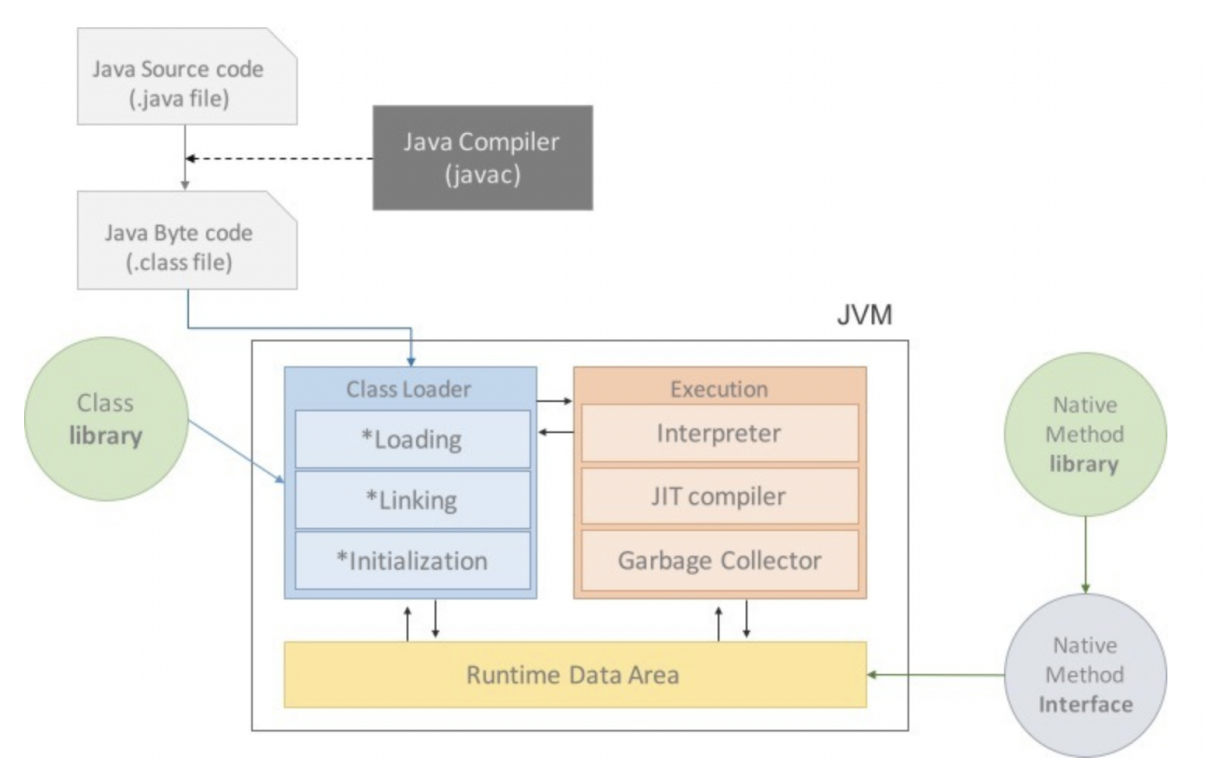
!? 왜 JVM을 알아야 할까?

- 한정된 메모리를 효율적으로 사용하여 최고의 성능을 내기 위해서!
- 메모리 효율성을 위해 메모리 구조를 알아야 한다.

💡 자바프로그램 실행 과정

1. 프로그램이 실행되면 JVM은 OS로부터 필요한 메모리를 할당받는다.
2. 자바 컴파일러(javac)가 자바 소스코드(.java)를 읽어 자바 바이트코드(.class)로 변환한다.
3. Class Loader를 통해 class 파일들을 JVM으로 로딩한다.
4. 로딩된 class 파일들은 Execution engine을 통해 해석된다.
5. 해석된 바이트코드는 Runtime Data Areas 에 배치되어 실질적인 수행이 이루어진다.

→ 이 과정에서 JVM은 필요에 따라 Thread Synchronization과 GC같은 관리작업을 수행한다.



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

💡 JVM 구성 & 메모리 구조

📌 Class Loader (클래스 로더)

- JVM 내로 바이트 코드(.class 파일)을 로드하고, 링크를 통해 배치하는 작업을 수행하는 모듈
- Runtime 시에 동적으로 클래스를 로드함
- .jar 파일 내에 저장된 클래스들을 JVM 위에 탑재하고 사용하지 않는 클래스들은 메모리에서 삭제 (컴파일러 역할)
- 자바는 동적코드로 컴파일 타임이 아니라 런타임에 참조함. 즉, 클래스를 처음으로 참조할 때 해당 클래스를 로드하고 링크하는 역할을 클래스 로더가 수행

📍 **동적 로드**
-
자바 프로그램이 실행될 때 모든 클래스가 한꺼번에 메모리에 로드되는 것이 아니라, **그 클래스가 실제로 필요한 시점에 메모리로 로드**

📍 **컴파일 타임 vs. 런타임**
- 자바의 소스 코드는 먼저 "컴파일 타임"에 컴파일러에 의해 바이트코드(.class 파일)로 변환. 하지만 이 바이트코드는 "런타임"에 자바 가상 머신(JVM)에 의해 실제로 실행됨. 여기서 중요한 점은,
클래스가 실제로 사용되는 시점(런타임)에 메모리로 로드된다는 것.

📌 Execution Engine (실행 엔진)

- 클래스를 실행시키는 역할
- 클래스 로더가 JVM 내의 Runtime Data Area에 바이트 코드(.class 파일)을 배치시키면 Execution Engine에 의해 실행
- 바이트 코드를 실제로 JVM 내부에서 기계가 실행할 수 있는 형태로 변경.
- 바이트 코드를 해석하는 두 가지 방식

💾 Interpreter (인터프리터)

- 바이트 코드를 명령어 단위로 한 줄씩 실행.
- 속도 느림

💾 JIT (Just-In-Time) 컴파일러

- 인터프리터의 단점을 보완하기 위해 도입된 컴파일러
- 인터프리터 방식으로 실행하다가 적절한 시점에 바이트 코드 전체를 네이티브 코드로 변환 → 그 다음부터 인터프리터는 네이티브 코드로 컴파일된 코드를 바로 사용
- 네이티브 코드는 캐시에 보관하기 때문에 한 번 컴파일된 코드는 빠르게 수행
- JIT 컴파일러가 컴파일하는 과정은 바이트코드를 인터프리팅하는 것보다 훨씬 오래걸리므로 한 번만 실행되는 코드라면 컴파일하지 않고 인터프리팅하는 것이 유리

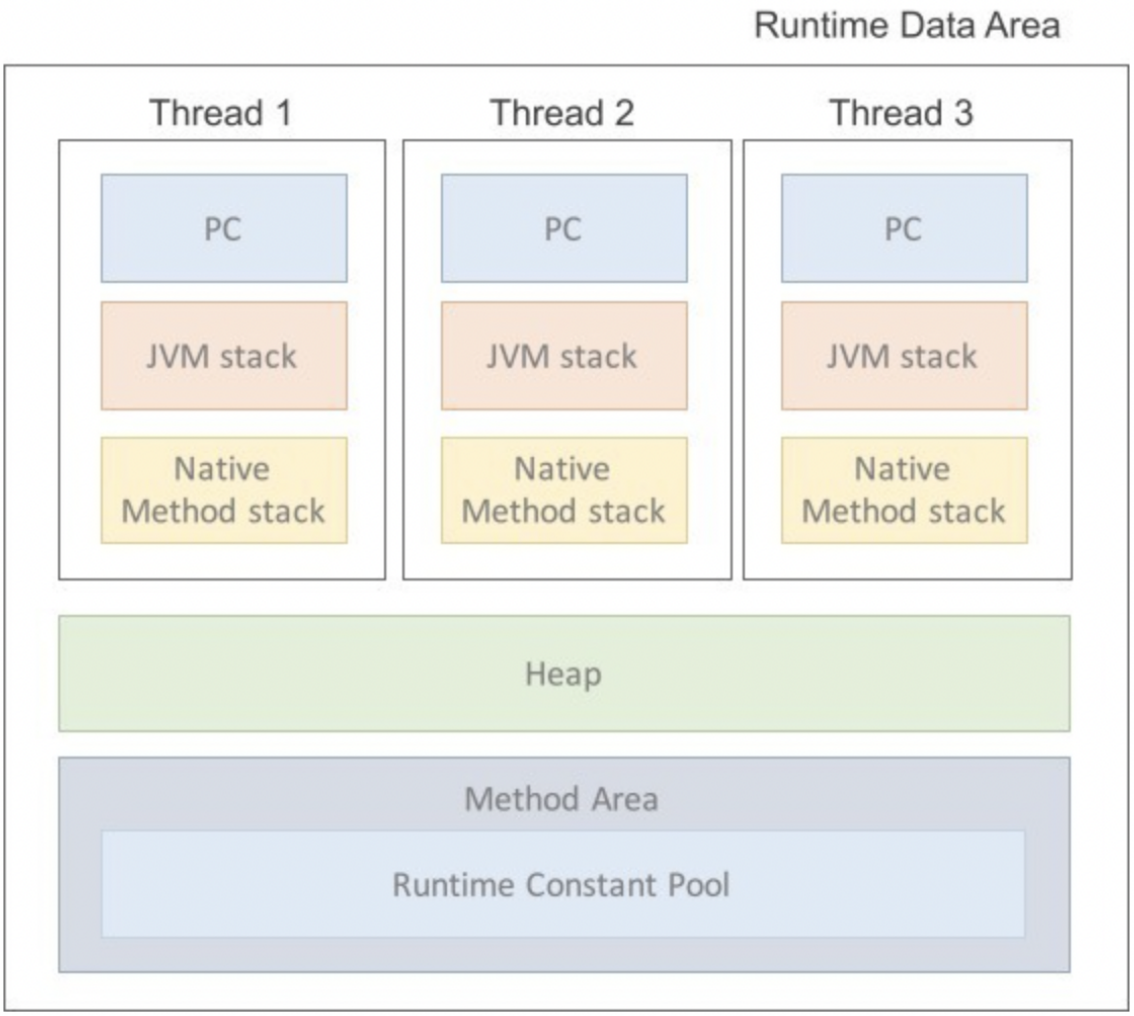
- **따라서 JIT 컴파일 사용 JVM들은 내부적으로 해당 메서드가 얼마나 자주 수행 되는지 체크하고 일정 정도를 넘을 때만 컴파일 수행**

💾 GC (Garbage Collector)

- GC를 수행하는 모듈 (쓰레드) 존재
- 힙 영역에서 사용되지 않는 객체들을 제거하는 작업을 의미

📌 Runtime Data Area

- 프로그램을 수행하기 위해 OS에서 할당 받은 메모리 공간
- Java는 멀티 스레드 환경으로 모든 스레드는 Heap, Method Area를 공유한다.



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

💾 PC Register

- JVM은 스택 기반의 가상 머신으로, CPU에 직접 접근하지 않고 스택에서 주소를 추출해 가져온다. 가져온 주소를 저장하는 곳이 PC Register.
- 따라서, 현재 어떤 명령을 실행해야 할 지에 대한 기록을 담당

💾 JVM Stack

- 호출된 메서드의 파라미터, 지역 변수, 리턴 값 및 연산 값 등이 저장되는 영역
- 프로그램 실행 시 임시로 할당되었다가 메서드를 빠져나가게 되면 소멸되는 특성의 데이터들이 저장되는 영역
- 메서드 호출 시마다 스택에 각각의 스택 프레임이 생성되고, 수행이 끝나면 스택 포인트에서 해당 프레임을 제거

💾 Native Method Stack

- Java 이외의 언어에 제공되는 Method의 정보가 저장되는 공간
- Java Native Interface를 통해 바이트 코드로 저장
- Kernel이 자체적으로 Stack을 잡아 독자적으로 프로그램을 실행시키는 영역

⬇️ JVM 메모리의 3가지 주요 영역

📍 Method Area

- 클래스에 대한 정보가 저장되는 공간
- 이 때 클래스의 클래스변수(class variable)도 함께 생성
- 클래스, 인터페이스, 메서드 필드 static 변수 등의 바이트 코드 보관
- 올라가는 정보
  - **Field Information**
    - 멤버 변수에 대한 정보 (이름, 타입, 접근 지정자 등)
  - **Method Information**
    - 메서드에 대한 정보 (이름, 리턴 타입, 파라미터, 접근 지정자 등)
  - **Type Information**
    - Class 인지 Interface 인지 혹은 Type의 속성, 이름, super class의 이름 등

        - 또한 Method Area에는 상수형을 저장하고 중복을 막는 Runtime Constant Pool이 존재

📍Heap

- 객체(instance)가 생성되는 공간. 즉, 인스턴스 변수(instance variable)들이 생성되는 공간
- **쓰레기 수집기(Garbage Collector, GC)**의 활동영역

📍 call stack 또는 execution stack (호출스택)

- 메서드의 작업에 필요한 메모리 공간을 제공
- 메서드의 작업 수행 중 발생하는 지역변수들과 연산의 중간결과 등을 저장
    - 메서드 호출 ➡️ 수행에 필요한 만큼 스택에 메모리 할당
    - 메서드 종료 ➡️ 사용했던 메모리 반환 및 스택에서 제거
    - 호출스택의 가장 위에 있는 메서드 〓 현재 실행 중인 메서드
    - 아래에 있는 메서드 〓 바로 위의 메서드를 호출한 메서드