# 1주차 과제 : JVM 은 무엇이며 자바 코드는 어떻게 실행하는 것인가

## 학습 목표

자바 소스 파일(.java)을 JVM 에서 동작하는 과정을 이해할 수 있다.

## 학습내용

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

## Advanced

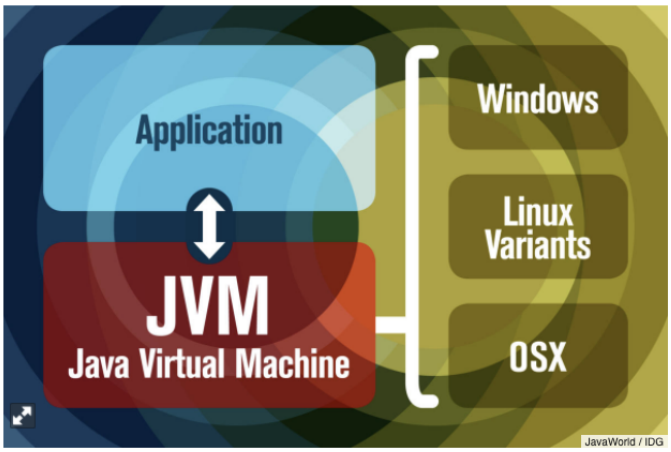
- GC란 무엇인가
- GC의 종류 및 각 GC별 동작 알고리즘
- cloud-native 환경에서의 JVM 관리의 특징

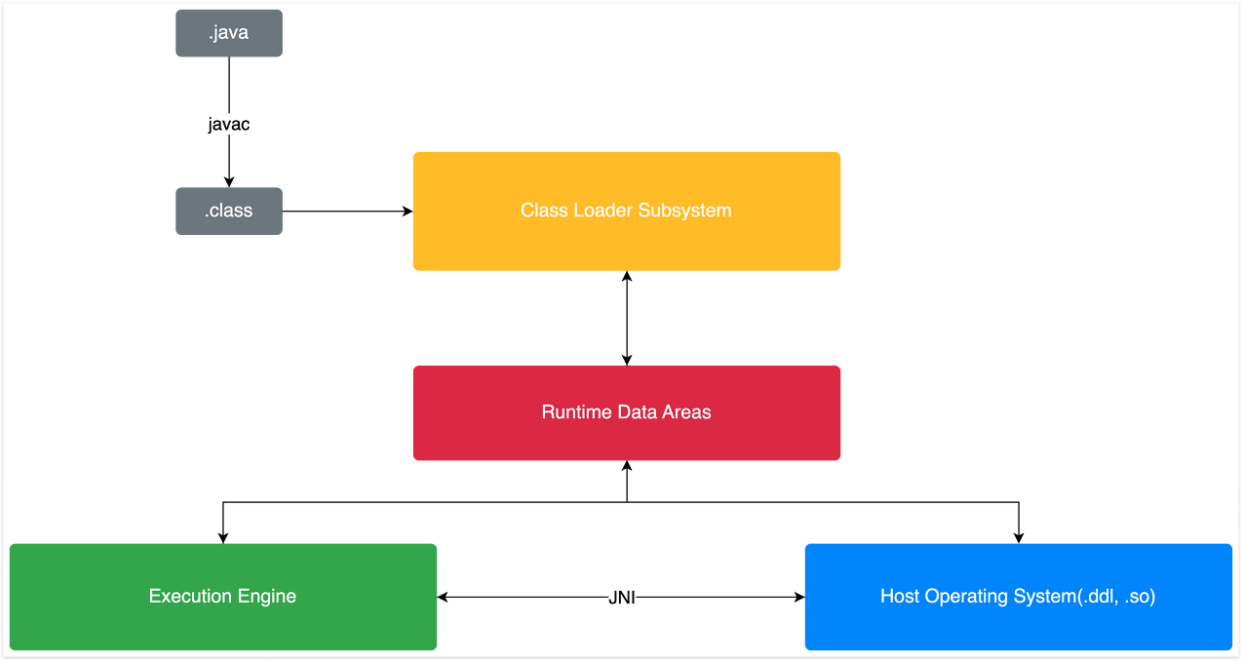
---

## JVM 이란 무엇인가

- 자바 컴파일러는 .java 파일을 .class 파일로 변환
- .class 파일에 포함된 바이트 코드는 기계어가 아니어서, OS 에서 즉시 실행이 안됨
- 그래서 **OS 가 바이트코드를 이해할 수 있게 기계어로 해석해주는 역할을 하는 것**이 `JVM` (JAVA Virtual Machine)

- JVM 위에서는 바이트코드가 OS 가 이해할 수 있도록 해석되기 때문에, 바이트코드는 JVM 위에서 **OS에 상관없이 실행**된다.
    - 따라서 JVM만 설치되어 있으면, **OS 에 종속적이지 않게 자바 코드 실행 가능**



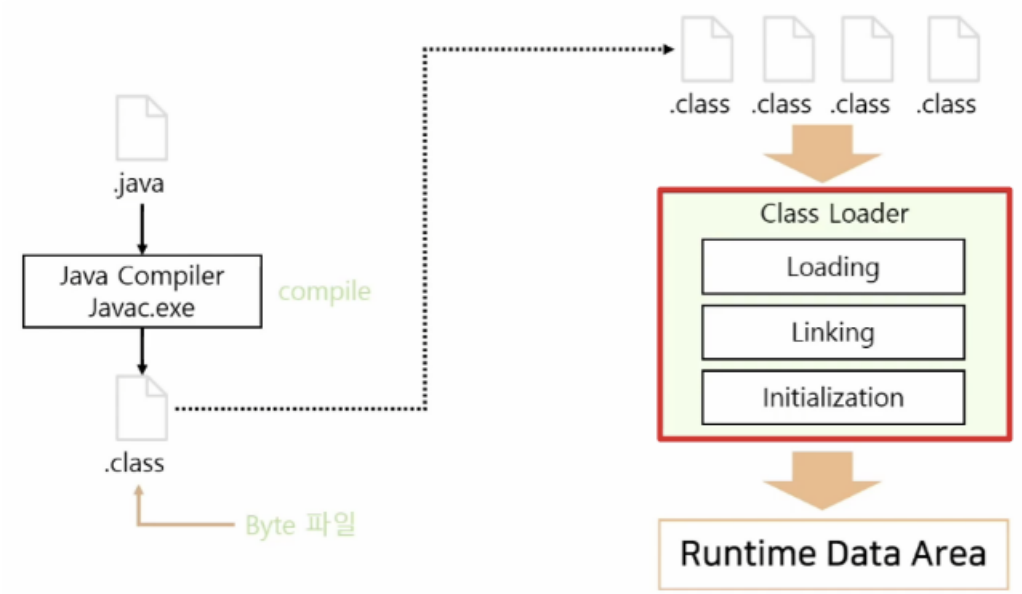

**JVM 이 바이트 코드를 해석하는 원리**

1. .java 파일이 javac (컴파일러) 에 의해 .class 로 컴파일됨

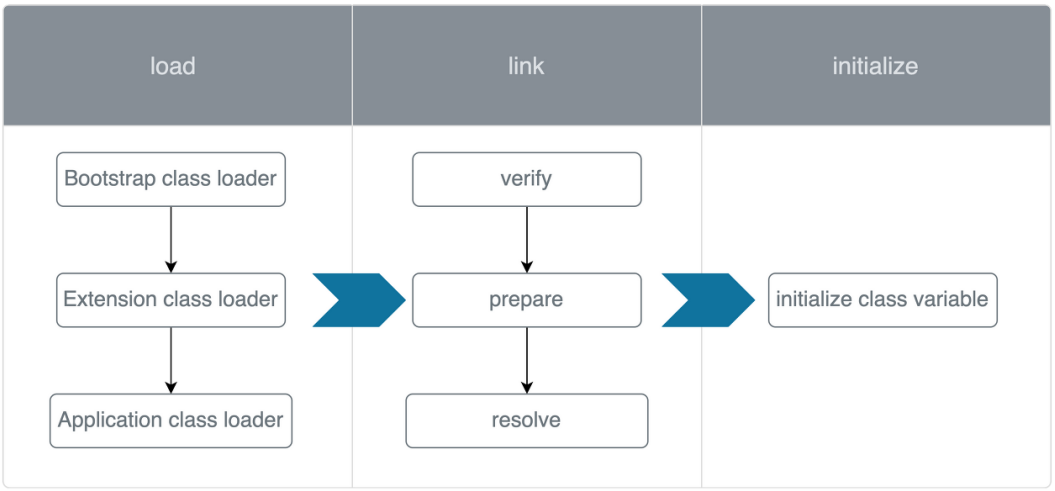
2. 클래스파일은 JVM 의 `Class Loader` 는 동적로딩을 통해 필요한 클래스들을 로딩/링크 하여 실제 메모리를 할당받아 관리하는 영역인 `Runtime Data Area` 에 올린다

    ▼ **클래스로더** : JVM 내로 클래스 파일들을 동적으로 로드하고, 링크를 통해 배치하는 작업을 수행하는 모듈 (로드된 클래스 코드를 엮어서 JVM 메모리 영역인 Runtime Data Area 에 배치)

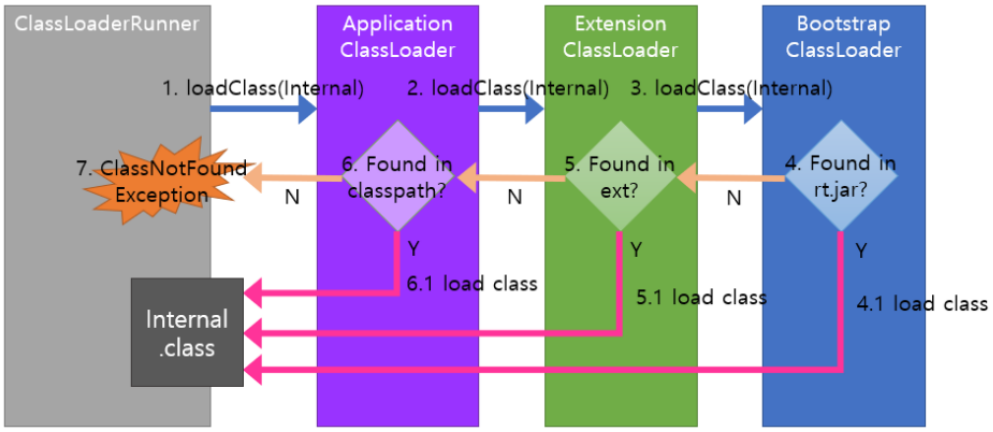
    - 바이트 코드를 3가지 클래스 로더로 로딩하여, 링크로 코드 내부의 레퍼런스를 연결하고 static 변수의 초기화 과정을 거쳐 실행 엔진이 사용할 수 있도록 JVM 메모리 영역에 적재하는 시스템




    - 클래스 파일의 로딩 순서




1. **Load : 클래스 파일을 가져와 JVM** 메모리에 로드 **하는 과정**




- 서로 상하관계에 있는 클래스 로더들이, 클래스 로딩을 요청할 때 상위 클래스 방향으로 위임하는 메커니즘이 존재
- 클래스 유형별로 계층을 구분하고, 로딩될 때 전부 다 로딩되지는 않고 해당 클래스 로더까지만 부분적으로 탐색하고 로딩할 수 있게 해주는 목적의 모델
- **Bootstrap class loader**
  - JAVA_HOME/lib에 있는 코어 자바 API를 제공 (jre/lib/rt.jar)
    - Object.class, String.class 등
  - 최상위 우선순위를 가진 클래스 로더

    - Native C 로 구현됨
- **Extension class loader**
    - JAVA_HOME/lib/ext 폴더 또는 java.ext.dirs 시스템 변수에 해당하는 위치에 있는 클래스를 읽는다 (jre/lib/ext/*.jar, java.ext.dirs/*.jar)
    - java 로 구현됨
- **Application class loader**
    - 앱 ClassPath(앱을 실행 할 때 주는 -classpath 옵션 또는 java.class.path 환경변수의 값에 해당하는 위치)에서 클래스를 읽는다 (-classpath, -cp, Manifest)
    - java 로 구현됨

2. **Link : 클래스 파일을 사용하기 위해 `검증` 하는 과정**
    - Verify : .class 파일 형식이 유효한지 체크 (클래스가 자바 언어 명세 및 JVM 명세에 명시된대로 구성되어있는지)
    - Prepare : 클래스 변수 (static 변수) 의 기본값에 따라 필요한 메모리를 할당, 클래스의 필드/메소드/인터페이스를 나타내는 데이터 구조 준비
    - Resolve (Optional) : 클래스의 상수 풀 내의 심볼릭 메모리 레퍼런스를 메소드 영역에 있는 실제 레퍼런스로 교체, 심볼릭 레퍼런스 ≠ 메모리 번지 참조가 아니라, **이름에 의한 참조를 말하는 것 like 바로가기 파일**
3. **Initialize : static 값들을 명시된 값으로 할당하는 과정**

3. Runtime Data Area 에 배치된 바이트 코드는 `Execution Engine`에 의해 실행된다

    ▼ **실행엔진 :** 실행 엔진은 클래스 로더를 통해 런타임 데이터 영역에 배치된 **바이트 코드를 명령어 단위로 읽어서 실행**

    - 바이트 코드를 실제 JVM 내부에서 기계가 실행할 수 있는 형태로 변경해줌
    - 이 때 실행 엔진은 Intepreter, JIT 컴파일러 2가지 방식을 혼합해서 바이트 코드를 실행함

4. 이 과정에서 Execution Engine 에 의해 GC 의 작동, Thread 동기화가 이루어진다

## 컴파일 하는 방법

- `javac` 명령어 사용하면, 클래스파일이 (ex) test.class) 생성됨
  - ex) javac [JAVA 파일명]  / javac test.java
- `javac` 로 자바 파일을 컴파일하면, 클래스파일이 생성되는 이유
  - C 또는 C++ 등으로 작성된 프로그램은 최종 결과물로 exe 파일을 만들어낸다. 사용자는 그냥 단순히 exe 프로그램을 실행하기만 하면 프로그램을 실행할 수 있다.
    - 물론 자바도 exe 프로그램으로 만들어 낼 수는 있지만 JVM이 exe에 포함되는 형식이어야 하므로 exe 파일이 무척이나 커진다는 단점이 있다.
  - C, C++ 등의 언어에서 만든 실행 파일은 JVM 같은 중간 단계를 거치지 않기 때문에 속도가 빠르다.
    - 하지만 운영체제마다 실행 파일을 따로 작성해야 한다는 단점이 있다.
  - 반대로 자바는 JVM이라는 중간 단계가 있으므로 C, C++ 등의 언어보다 속도가 느리다.
    - 하지만 한 번 작성한 클래스 파일은 어떤 운영체제에서도 사용할 수 있다는 장점이 있다.
    - 자바 또한 JIT just-in-time 을 도입하는 등의 발전을 거듭하여 실행 속도도 하드웨어를 직접 제어하는 경우만 아니라면 거의 C, C++를 따라잡았다.

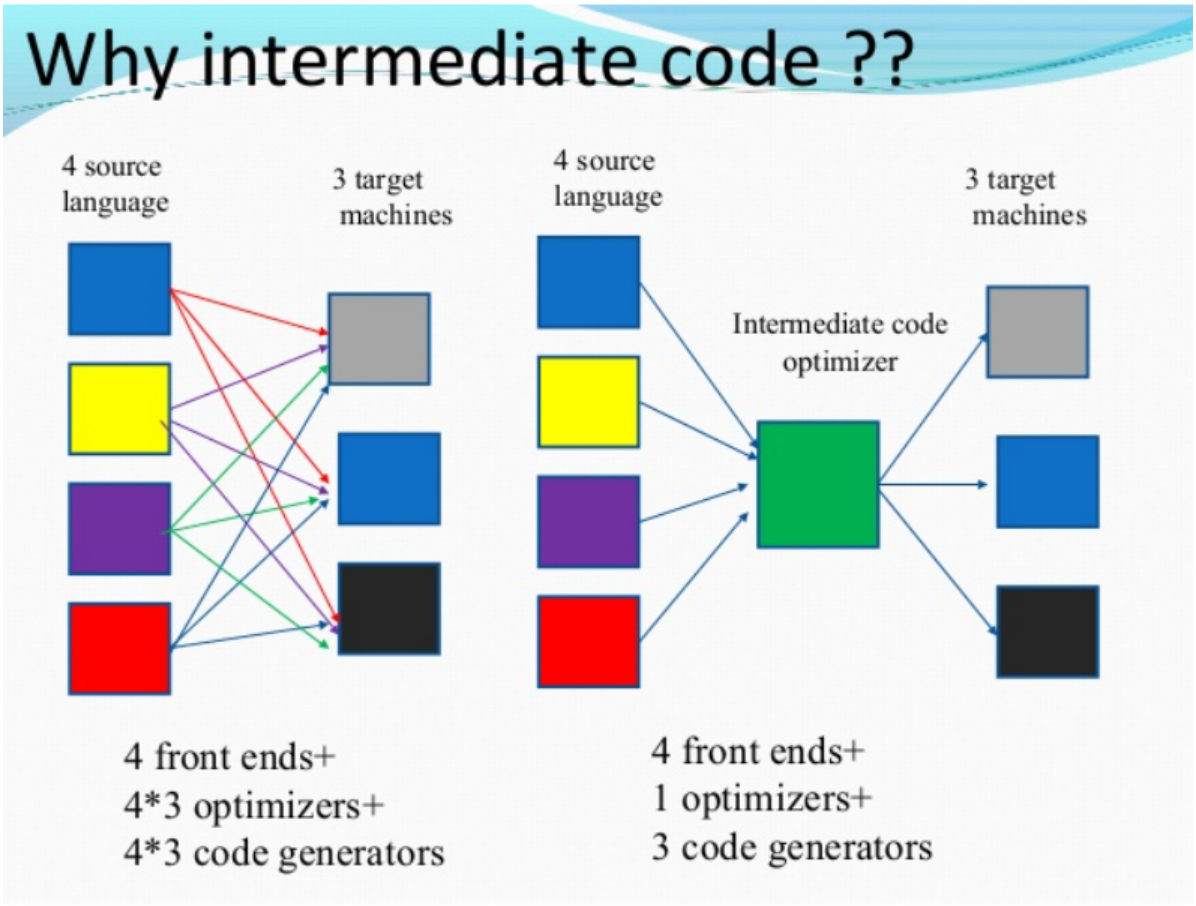
## 실행하는 방법

- `java` 명령어 사용 ex) java [클래스 파일명]  / java test
- java 명령어는 JRE 를 실행하는데, 이는 java 명령어의 **인자로 지정된 클래스를 실행하기 위한 자바 실행환경이 조성됨**을 의미한다.
- 인자로 지정된 클래스를 로딩하고 main() 메소드를 호출한다

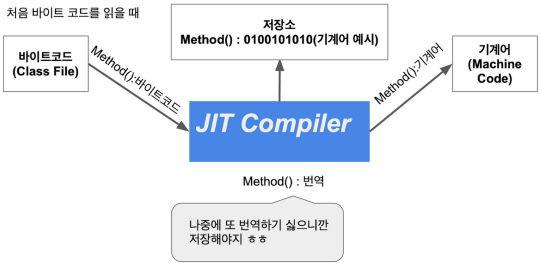
## 바이트코드란 무엇인가

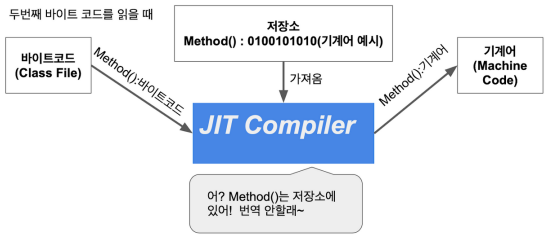
- 특정 하드웨어가 아닌 **가상 컴퓨터에서 돌아가는 실행 프로그램을 위한 이진 표현법** (특정 하드웨어에 종속적이지 않음)
- 가상머신이 이해할 수 있는 언어

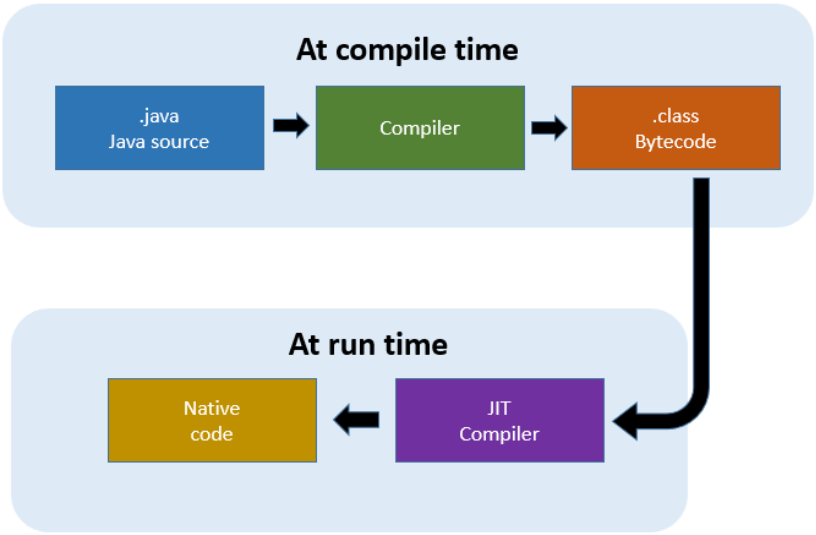
- 바이트코드는 다시 인터프리터나 JIT 컴파일러에 의해 바이너리 코드 (컴퓨터가 인식할 수 있는 0,1 로 구성된 이진코드) 로 변환됨
- 바이트 코드를 만드는 이유
    - 중간단계를 하나 두어서, 간접화를 통해 경우의 수를 낮추고 효율을 높이기 위해 중간 코드를 생성하는 것




## JIT 컴파일러란 무엇이며 어떻게 동작하는지



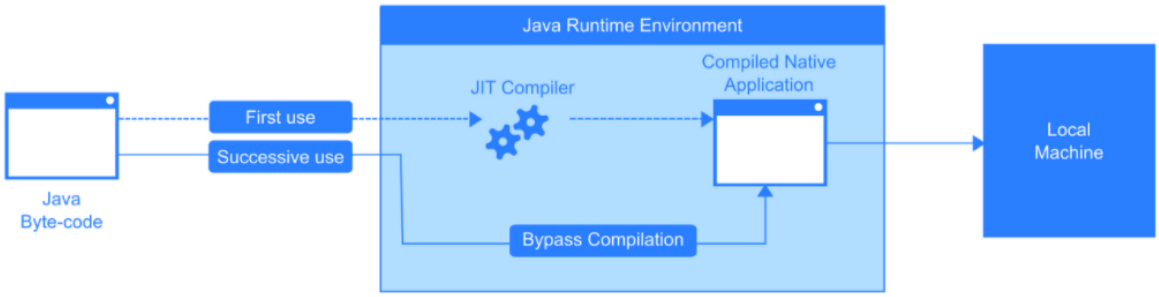



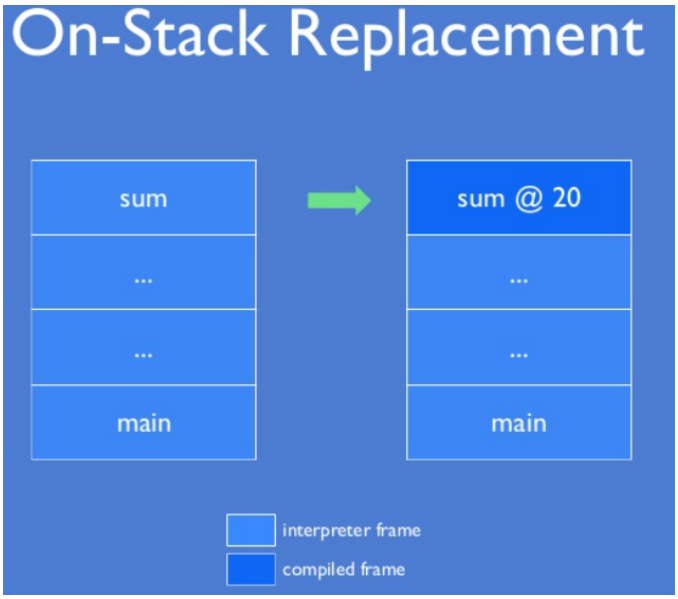

- 자바는 **바이트 코드로 한 번 컴파일**하고, **바이트코드를 인터프리터** 하는 방식 2가지를 진행한다
- C언어보다 자바가 느리다고 하는 이유
  - 바이트코드는 기계가 이해할 수 있는 기계어가 아니기 때문에, `인터프리터` 를 사용해서 **런타임 시** 한줄씩 읽어야 함. 따라서 **런타임 전** 소스코드를 미리 읽어서 기계어로 변환하는 `컴파일러` 방식보다 느릴 수 밖에 없음
  - 이러한 문제를 개선하기 위해, **번역된 코드는 캐싱**해두고, 다음에 똑같은 코드가 있다면 **번역하지 않고 캐싱된 값을 사용하여 매번 기계어 코드가 생성되는 것을 방지**해 인터프리터를 사용하는 시간을 단축시키는 것 (즉, 인터프리터와 컴파일러 방식을 적절히 혼합하여 속도 개선)
  - 바이트코드를 Native Code로 변환하는 데에도 비용이 소요되므로, JVM은 모든 코드를 JIT 컴파일러 방식으로 실행하지 않고 **인터프리터 방식을 사용하다 일정 기준이 넘**

**어가면 JIT 컴파일 방식으로 명령어를 실행하는 식으로 진행**

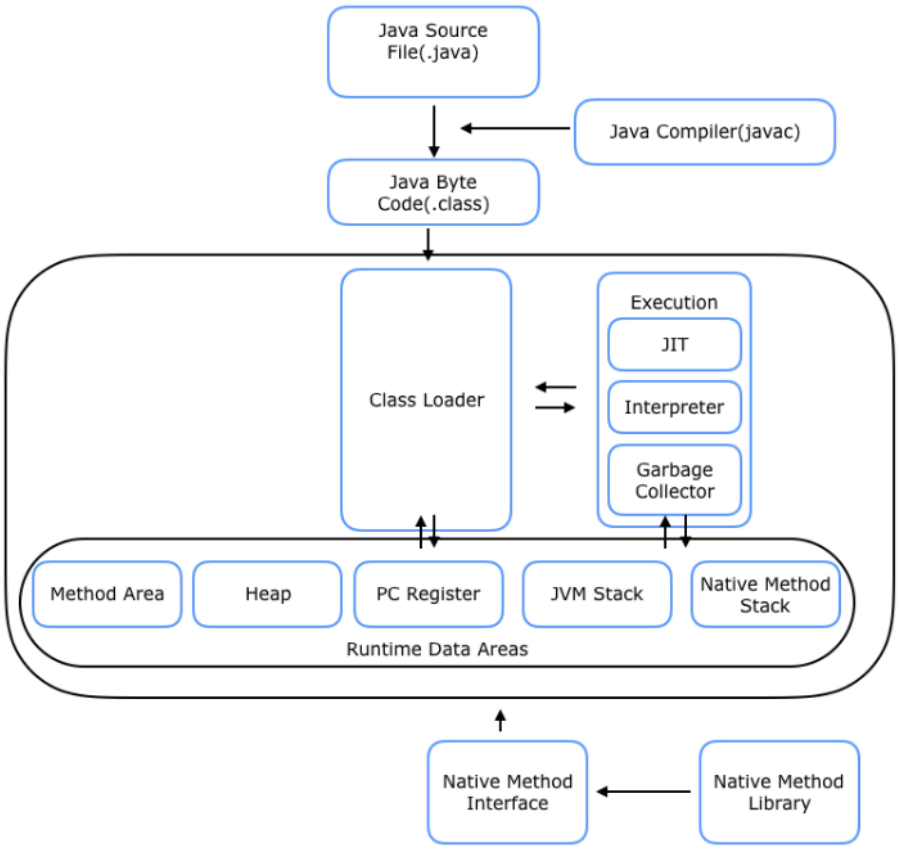
- 인터프리터와 컴파일러 방식을 어떻게 적절하게 혼합한 것인지




  - JIT 컴파일러의 Just In Time 은 '제 때' 라는 뜻이며, 이는 `자바 메소드를 호출할 때` 를 의미함. 하지만 자바 메소드는 처음 호출되자마자 컴파일이 되는 것이 아님
  - JVM이 호출되는 메서드 각각에 대해 호출마다 호출 횟수를 누적해서 그 횟수가 특정 수치를 초과 할 때 컴파일함. 즉, **얼마나 자주 호출되는지 검사한 후 '이제는 컴파일이 필요한 시점이다'**라고 판단하는 기준이 있는데, 이를 `컴파일 임계치`라고 한다
  - 컴파일 임계치는 아래 2가지를 합친 것을 말한다.
    - Method Entry Counter (JVM 내의 메소드가 호출된 횟수)
    - Back-Edge Loop Counter (메소드가 루프를 빠져나오기까지 돈 횟수)
  - 위의 2가지 카운터 합계를 확인하고 **메소드가 컴파일될 자격이 있는지 결정한 후, 자격이 있다면 해당 메소드는 컴파일 되기 위해 큐에서 대기**하게 되고, 이후 컴파일 스레드에 의해 컴파일됨
  - `OSR (On Stack Replacement)` : 메소드 스택 상에서 컴파일 된 버전을 바로 실행시키는 것
    - 스택 프레임을 컴파일 된 것으로 교체해서 속도를 개선한 것


On-Stack Replacement
sum
...
...
main
sum @ 20
...
...
main
interpreter frame
compiled frame

## JVM 구성 요소


Java Source File(.java)
Java Compiler(javac)
Java Byte Code(.class)
Class Loader
Execution
JIT
Interpreter
Garbage Collector
Method Area
Heap
PC Register
JVM Stack
Native Method Stack
Runtime Data Areas
Native Method Interface
Native Method Library

▼ **클래스 로더 (Class Loader) :** JVM 내로 클래스 파일(*.class)을 동적으로 로드하고, 링크를 통해 배치하는 작업을 수행하는 모듈

▼ **실행 엔진 (Execution Engine)**

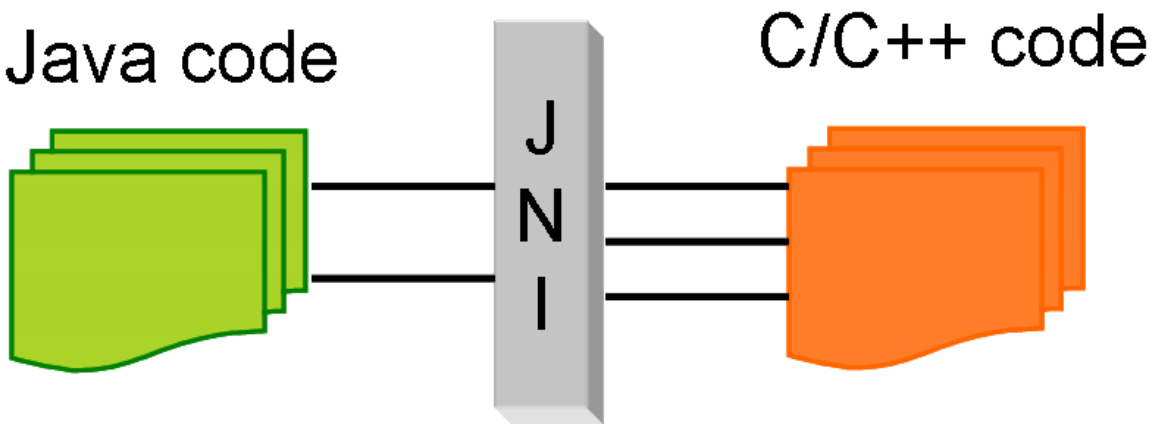
- Interpreter
  - 바이트 코드 명령어 하나씩 읽어 해석하고 바로 실행
  - JVM 내에서 바이트 코드는 기본적으로 인터프리터 방식으로 동작하기에, 같은 메소드라도 여러번 호출되면 매번 해석하고 수행해야돼서 속도가 느림
- JIT 컴파일러
  - 인터프리터 방식의 단점을 보완하기 위해, 반복된 코드를 발견하여 바이트 코드 전체를 컴파일하여 `Native Code` 로 변경하고, 이후에는 해당 메소드를 더이상 인터프리팅하지 않고 `캐싱` 해두었다가 Native Code로 직접 실행하는 방식
  - 하나씩 인터프리터로 실행하는 게 아니라, 컴파일 된 Native Code 를 실행하기 때문에 전체적인 속도는 인터프리팅 방식보다 빠르다
  - 바이트코드를 Native Code로 변환하는 데에도 비용이 소요되므로, JVM은 모든 코드를 JIT 컴파일러 방식으로 실행하지 않고 **인터프리터 방식을 사용하다 일정 기준이 넘어가면 JIT 컴파일 방식으로 명령어를 실행하는 식으로 진행**
- GC
  - Heap 메모리 영역에서 더이상 사용하지 않는 메모리를 자동회수
  - 일반적으로 자동으로 실행되지만, 단 GC(가비지 컬렉터) 가 실행되는 시간은 정해져 있지 않다.
  - 특히 Full GC 가 발생하는 경우, GC 를 제외한 모든 스레드가 중지되기 때문에 장애가 발생할 수 있다.

▼ **런타임 데이터 영역 (Runtime Data Area) :** JVM 의 메모리 영역으로, 자바 애플리케이션 실행 시 사용되는 데이터를 적재하는 영역

- **모든 스레드가 공유하는 영역**
- **각 스레드마다 생성되는 개별 영역**
- **Method Area (Runtime Constant Pool 포함)**
  - JVM 에서 읽어들인 클래스와 인터페이스에 대한 *런타임 상수 풀, 메소드와 필드, static 변수, 메소드 바이트 코드* 등을 포함

    - **Runtime Constant Pool**
        - 클래스 파일 constant_pool 테이블에 해당하는 영역
        - 클래스, 인터페이스 상수, 메소드, 필드에 대한 *모든 레퍼런스 저장*
        - JVM 은 런타임 상수 풀을 통해 *해당 메소드나 필드의 실제 메모리 상 주소를 찾아 참조*
    - JVM 시작 시에 생성되며 프로그램 종료 시까지 저장됨 / 명시적으로 null 선언 시 GC 대상 (구성 방식이나 GC 방법은 JVM 벤더마다 상이)
- **Heap**
    - 프로그램 상에서 데이터를 저장하기 위해 **런타임 시에 동적으로 할당하여 사용하는 메모리 영역**
    - new 연산자를 통해 생성한 객체 또는 배열을 저장
    - JVM 이 관리
    - 객체가 더이상 사용되지 않거나, 명시적으로 Null 선언 시 GC의 대상이 됨 (구성 방식이나 GC 방법은 JVM 벤더마다 상이)
- **Stack**
    - 선입후출 구조로, 메소드 호출 시 생성되는 수행정보를 기록하는 Frame 을 저장
    - 메소드 정보, 지역변수, 매개변수, 연산 중 발생하는 임시 데이터 저장
- **PC Register**
    - 현재 실행 중인 JVM 주소를 가짐
    - CPU 명령어 (instruction) 을 수행
    - CPU 명령어 수행하는동안 필요한 정보를 CPU 내 기억장치인 레지스터에 저장
    - 연산 및 결과값을 메모리에 전달하기 전, CPU 내의 기억장치
- **Native Method Stack**
    - Native Code : JAVA에서 부모가 되는 C언어나, C++, 어셈블리어로 구성된 코드
    - 자바 외 언어로 작성된 Native Code 를 위한 메모리
    - Java Native Interface 를 통해 바이트코드로 전환하여 저장하게 됨

▼ **JNI (Java Native Interface == 네이티브 메소드 인터페이스)** : 네이티브 메소드 라이브러리에 있는 네이티브 메소드를 사용하기 위한 인터페이스

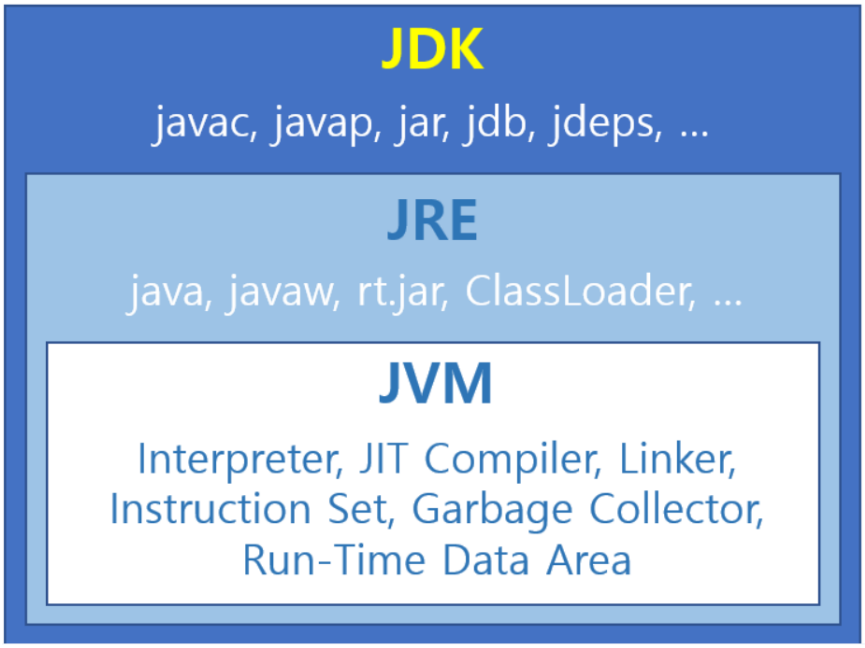



- native 키워드가 붙은 메소드의 이름과 동일한 인터페이스를 통해, 네이티브 메소드 내에 구현된 로직을 실행할 수 있음

▼ **네이티브 메소드 라이브러리** : 네이티브 메소드가 구현되어 있는 라이브러리

- 이곳에 접근하기 위해서는 네이티브 메소드 인터페이스를 거쳐야 함

## JDK와 JRE의 차이




- **JDK** : Java Development Kit
  - 컴파일러 (javac), 역어셈블러 (javap), 디버거 (jdb) 등 개발에 필요한 도구를 제공함
- **JRE** : Java Runtime Environment
  - 자바 실행 명령, 프로그램 실행 실패 등이 발생할 경우 대화 상자를 표시하는 시작 프로그램 파일(javaw.exe), 클래스로더와 바이트코드의 실행(java.exe)에 필요한 기본 라이브러리(rt.jar) 제공
- JDK 의 컴파일러를 사용해서 class 파일을 만들고,
  JRE 의 java 명령어를 사용해서 클래스파일의 바이트코드를 실행하도록 요청하면,
  JVM 이 구동되면서 바이트코드가 실질적으로 실행된다 (실제 OS에 메모리 할당/회수, 시스템 명령 호출 등..)

## 번외) 클래스파일과 바이트 코드

- 클래스 파일 ) 바이트 코드

1. **클래스 정보**: 클래스 파일에는 클래스의 이름, 상위 클래스 및 구현하는 인터페이스에 대한 정보
2. **필드 정보**
   - 클래스에 선언된 필드(멤버 변수)에 대한 정보가 클래스 파일에 포함

- 이 정보는 필드의 이름, 데이터 유형, 액세스 제어자 등을 포함

3. **메서드 정보**
   - 클래스에 정의된 메서드(함수)에 대한 정보도 클래스 파일에 저장
   - 이 정보에는 메서드 이름, 반환 유형, 매개 변수 목록, 액세스 제어자 등이 포함
4. **상수 풀(Constant Pool)**
   - 클래스 파일은 상수 풀이라는 특별한 데이터 구조를 포함
   - 상수 풀에는 클래스의 상수, 문자열 리터럴, 필드 및 메서드에 대한 참조 등이 저장됨
     → 이 상수 풀은 바이트 코드에서 **상수에 대한 참조를 효율적으로 관리**하는 데 사용됨
5. **바이트 코드** : 가장 중요한 부분으로, **클래스 파일에는 JVM에서 실행될 수 있는 바이트 코드가 포함되는데, 이 바이트 코드는 클래스의 메서드 내용**을 나타냄
6. 기타 부가 정보: 클래스 파일에는 메서드에 대한 예외 처리 정보, 디버깅 정보, 메타데이터 등 다양한 부가 정보가 포함될 수 있음

- 이러한 정보는 JVM이 클래스 파일을 로드하고 실행할 때 사용됨