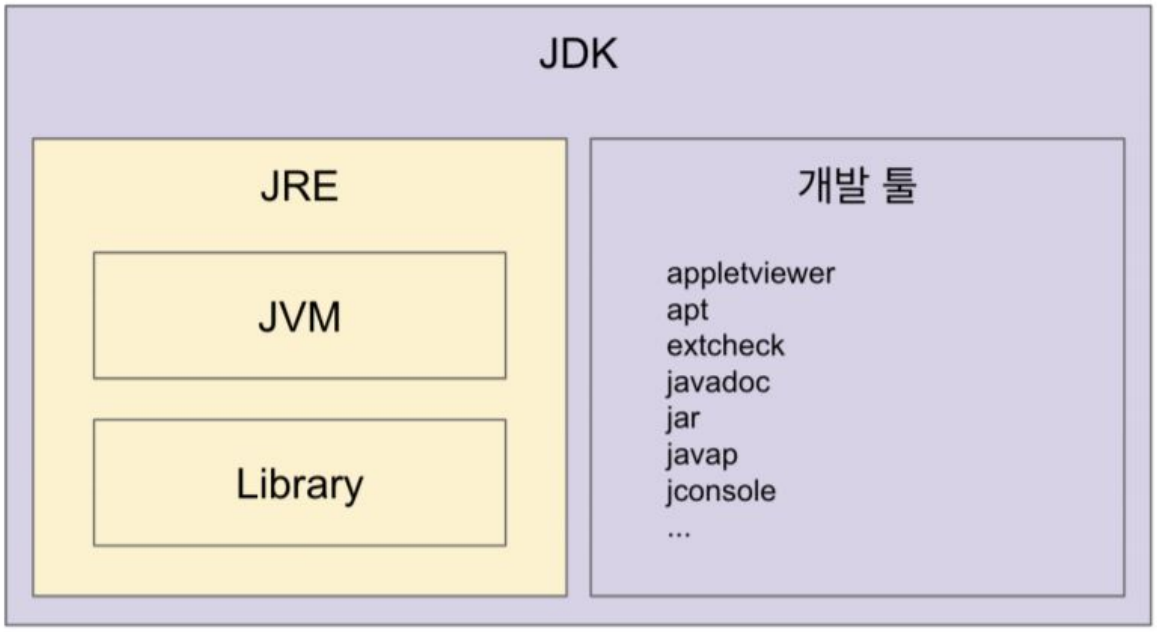
더 자바 코드를 조작하는 다양한 방법

[JVM 이해]



JVM

- Java Virtual Machine
- 자바 가상 머신으로 자바 바이트 코드(.class파일)를 OS에 특화된 코드로 변환(인터프리터와 JIT컴파일러)하여 실행한다
- 바이트코드를 실행하는 표준(JVM 자체는 표준)이자 구현체(특정 벤더가 구현한 JVM)
- JVM 벤더 : 아마존, 오라클, Azul
- 특정 플랫폼에 종속적

JRE

- Java Runtime Environment
- 목적 : 자바 애플리케이션을 실행
- JVM + 라이브러리
- JVM과 핵심 라이브러리 및 자바 런타임 환경에서 사용하는 프로퍼티 세팅이나 리소스 파일을 가지고 있다
- 개발 관련 도구는 포함X (그건 JDK 에서 제공)

JDK

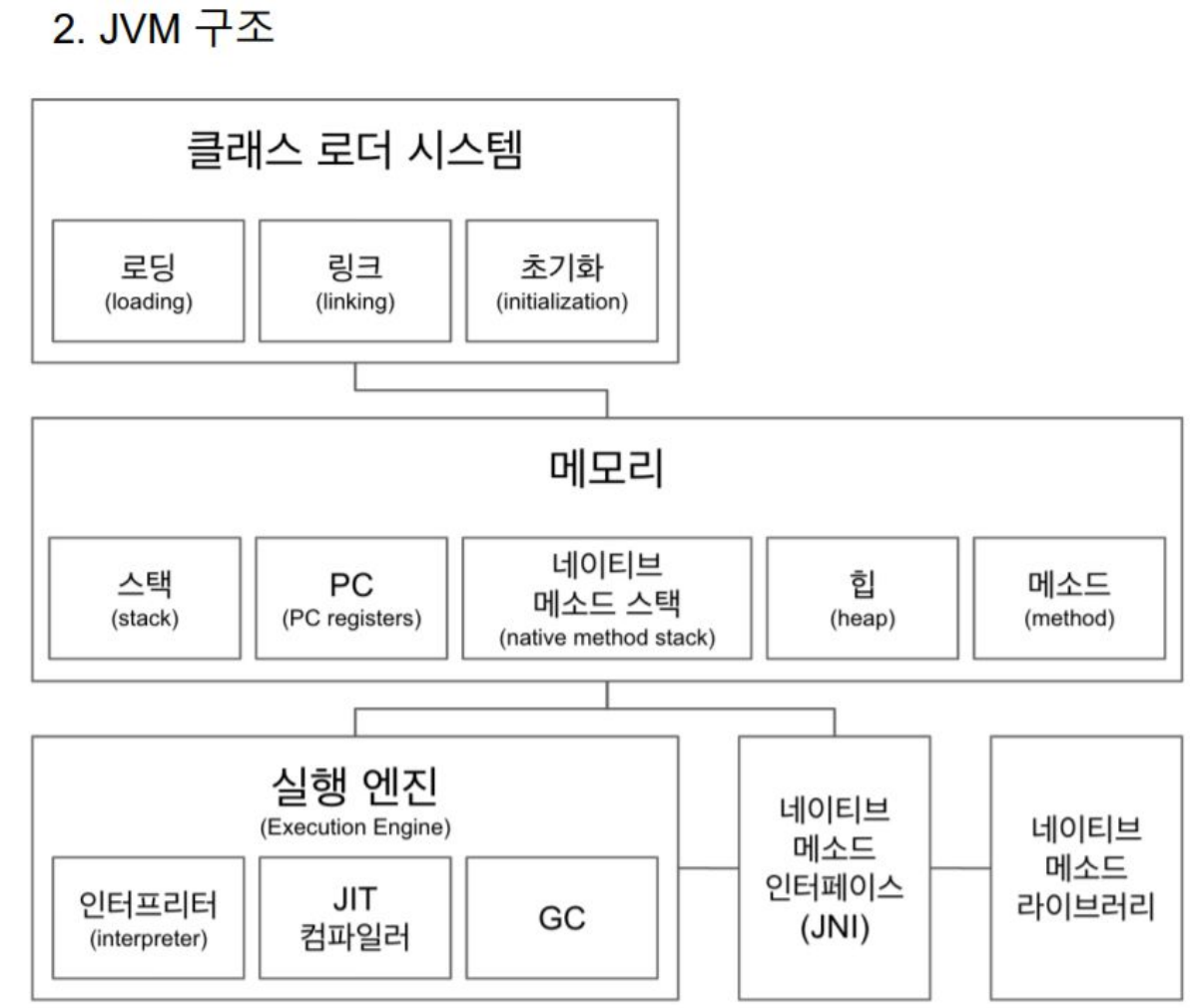
- Java Development Kit
- JRE + 개발에 필요한 툴
- 소스코드를 작성할 때 사용하는 자바 언어는 플랫폼에 독립적
- 오라클은 자바11부터는 JDK만 제공하며 JRE 는 따로 제공 X
- Write Once Run Anywhere

자바

- 프로그래밍 언어
- JDK에 들어있는 자바 컴파일러(javac)를 사용하여 바이트코드(.class)로 컴파일 가능
- 자바유로화 : Oracle JDK 11 부터 상용으로 사용시 유료

JVM 언어
- JVM 기반으로 동작하는 프로그래밍 언어
- 클로저, 그루비, JRuby, 코틀린 등

[JVM 구조]



클래스 로더 시스템
- .class 에서 바이트코드를 읽고 메모리에 저장
- 로딩 : 클래스 읽어오는 과정
- 링크 : 래퍼런스를 연결하는 과정
- 초기화 : static 값들 초기화 및 변수에 할당

메모리
- 메소드 영역에는 클래스 수준의 정보(클래스이름, 부모클래스 이름, 메소드, 변수) 저장. 공유자원이다
- 힙 영역에는 객체를 저장. 공유자원이다
- 스택/PC/네이티브 메소드 스택은 쓰레드

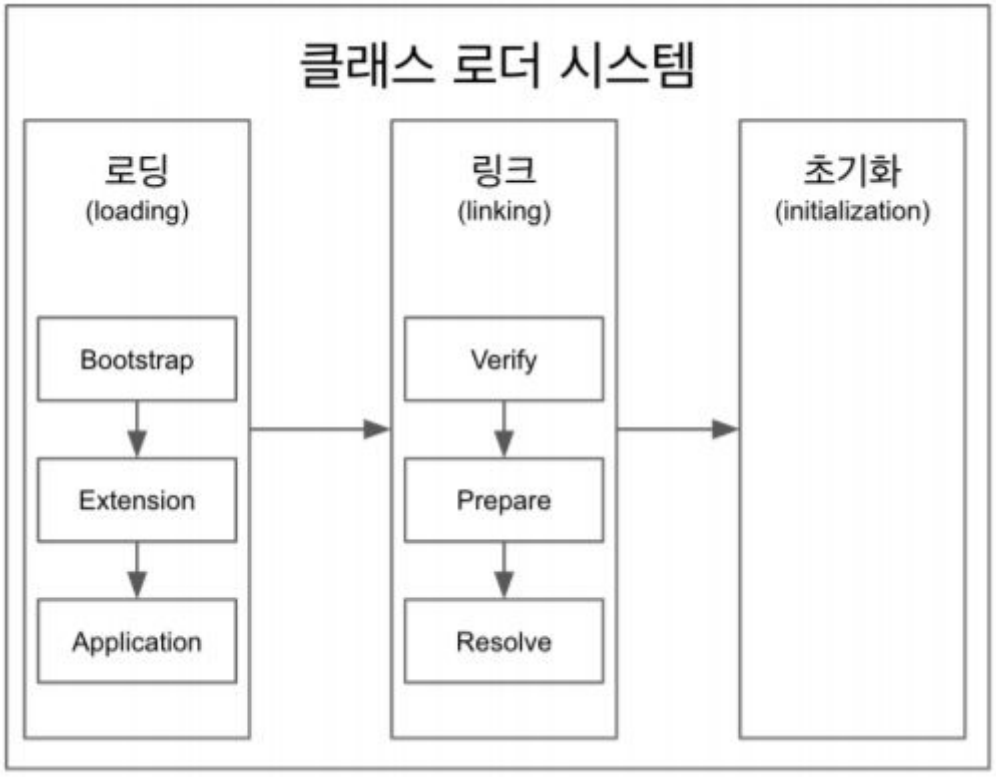
- 스택 영역에는 스레드 마다 런타임 스택을 만들고 그 안에 메소드 호출을 스택프레임이라 부르는 블럭으로 쌓는다 쓰레드를 종료하면 런타임 스택도 사라진다
- PC(Program Counter) 레지스터 : 쓰레드 마다 쓰레드 내 현재 실행할 스택프레임을 가리키는 포인터가 생성
- 네이티브 메소드 : native 키워드가 붙어 있고 구현을 자바가아닌 C나 C++로 구현

실행 엔진
- 인터프리터 : 바이트 코드를 한줄 씩 실행
- JIT 컴파일러 : 인터프리터 효율을 위해 인터프리터가 반복되는 코드를 발견하면 JIT 컴파일러로 반복되는 코드를 모두 네이티브 코드로 바꿔둔다. 그 다음 부터 인터프리터는 네이티브 코드로 컴파일된 코드를 바로 사용
- GC : 더 이상 참조되지 않는 개체를 모아 정리

#JNI
- Java Native Interface
- 자바 애플리케이션에서 C,C++ 어셈블리로 작성된 함수를 사용할 수 있는 방법 제공
- Native 키워드를 사용한 메소드 호출

네이티브 메소드 라이브러리
- C / C++ 로 작성된 라이브러리

[클래스 로더]



클래스 로더
- 로딩, 링크, 초기화 순으로 진행

- 로딩
    - 클래스로더가 .class 파일을 읽고 그 내용에 따라 적절한 바이너리 데이터를 만들어 “메소드” 영역에 저장
    - 이때 메소드영역에 저장되는 데이터
        - FQCN
        - 클래스 / 인터페이스 / 이늄
        - 메소드와 변수
    - 로딩이 끝나면 해당 클래스 타입의 Class 객체 생성하여 “힙” 영역에 저장
- 링크
    - Verify, Prepare, Resolve(optional) 3 단계로 나뉨
    - 검증 : class 파일 형식이 유효한지 체크
    - Preparation : 클래스 변수(static변수)와 기본값에 메모리
    - Resolve : 심볼릭 메소드 래퍼런스를 메소드 영역에 실제 래퍼런스로 교체
- 초기화
    - Static 변수의 값을 할당
- 클래스 로더는 계층 구조로 이루어져 있으며 3가지 클래스 로더 제공
    1) 부트 스트랩 클래스 로더
        - JAVA_HOME\lib에 있는 코어 자바 API를 제공한다. 최상위 우선순위를 가진 클래스 로더
    2) 플랫폼 클래스로더
        - JAVA_HOME\lib\ext 폴더 또는 java.ext.dirs 시스템 변수에 해당하는 위치에 있는 클래스를 읽는다.
    3) 애플리케이션 클래스로더
        - 애플리케이션 클래스패스(애플리케이션 실행할 때 주는 -classpath 옵션 또는 java.class.path 환경 변수의 값에 해당하는 위치)에서 클래스를 읽는다.

[바이트 조작 툴 활용 예]
- 프로그램 분석
    - 코드에서 버그 찾는 툴
    - 코드 복잡도 계산
- 클래스 파일 생성
    - 프록시
    - 특정 API 호출 접근제한
    - 스칼라 같은 언어의 컴파일러
- 그 밖에 자바 소스를 건드리지 않고 코드변경이 필요한 경우
    - 프로파일러
    - 최적화
    - 로깅
- 스프링 컴포넌트 스캔하는 방법
    - 컴포넌트 스캔으로 빈으로 등록할 후보 클래스 정보를 찾는데 사용
    - ClassPathScanningCandidateComponentProvider -> SimpleMetadataReader
    - ClassReader와 Visitor 사용해서 클래스에 있는 메타 정보를 읽어온다.

[리플렉션]

Class<T> 에 접근하는 방법
- 모든 클래스를 로딩한 다음 Class<T>의 인스턴스가 생긴다. “타입.class”로 접근 가능

- 모든 인스턴스는 getClass() 메소드를 가지고 있다. “인스턴스.getClass()”로 접근 가능
- 클래스를 문자열로 읽는 방법
    - Class.forName(“FQCN”)
    - 클래스패스에 해당 클래스가 없다면 ClassNotFoundException 발생

Class<T>를 통해 할 수 있는 것
- 필드 (목록) 가져오기
- 메소드 (목록) 가져오기
- 상위 클래스 가져오기
- 인터페이스 (목록) 가져오기
- 애노테이션 가져오기
- 생성자 가져오기

중요 애노테이션
- @Retention : 해당 어노테이션을 언제까지 유지할 것인가? 소스/클래스/런타임
- @Inherit : 해당 어노테이션을 하위 클래스까지 전달 할 것인가
- @Target : 어디에 사용할 수 있는가? 생성자/필드 등

리플렉션
- getAnnotations(): 상속받은 (@Inherit) 애노테이션까지 조회
- getDeclaredAnnotations(): 자기 자신에만 붙어있는 애노테이션 조회

[리플렉션 API]

Class 인스턴스 만들기
- Class.newInstance()는 deprecated 됐으며 이제부터는
- 생성자를 통해 만들어야 한다

생성자로 인스턴스 만들기
- Constructor.newInstance(params)

필드 값으로 접근/설정하기
- 특정 인스턴스가 가지고 있는 값을 가져오는 것이기 때문에 인스턴스 필요
- Field.get(object)
- Field.set(object, value)
- static 필드를 가져올 땐 object가 없어도 되니까 null을 넘김

메소드 실행하기
- Object Method.invoke(object, params)

리플렉션 사용시 주의
- 지나친 사용은 성능 이슈를 야기할 수 있다. 반드시 필요한 경우에만 사용할 것
- 컴파일 타임에 확인되지 않고 런타임 시에만 발생하는 문제를 만들 가능성이 있다.
- 접근 지시자를 무시할 수 있다

[]