

1장 - JVM은 무엇이며 자바는 어떻게 실행하는 것인가

출처 : <https://github.com/whiteship/live-study/issues/1>

목표

자바 소스 파일(.java)을 JVM으로 실행하는 과정 이해하기.

학습할 것

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

들여가기에 앞서

오늘부터 백기선님의 자바 스터디를 같이 진행해보기로 했다.

동영상 강의만으론 제대로 이해하기가 어렵다는 이유와, 좀 더 깊게 언어를 이해하고 싶다는 이유다.

한 주차당 며칠이 걸릴지는 잘 모르겠지만, 최대한 하루만에 끝내려고 노력할 것이다.

자바는 WORA(Write Once Run Anywhere)을 원한다!

JVM이란 무엇인가

Java Virtual Machine, 자바 바이트코드를 실행할 수 있는 주체 (출처 : 위키백과)

자바 바이트코드는 JRE(Java Runtime Environment) 위에서 동작한다. 이 JRE에서 가장 중요한 요소는 **자바 바이트코드를 해석하고 실행하는 JVM**이다. JVM의 역할은 **자바 애플리케이션을 클래스 로더를 통해 읽어들이어서 자바 API와 함께 실행하는 것**이다.

자바 애플리케이션 해석을 클래스 로더 → 자바 API와 함께 실행 ?

JVM의 특징

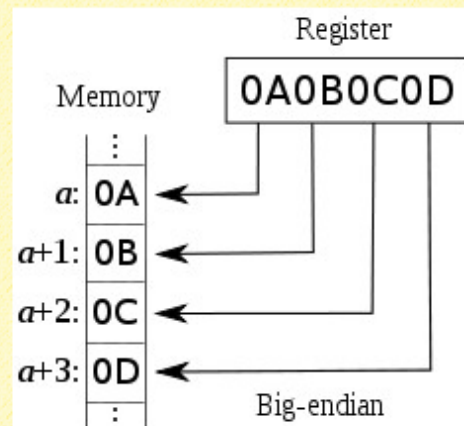
- **스택 기반의 가상 머신**
- ***심볼릭 레퍼런스** : 기본 자료형(primitive data type)을 제외한 모든 타입(클래스, 인터페이스)을 명시적인 메모리 주소 기반의 레퍼런스가 아니라, **심볼릭 레퍼런스를 통해 참조** 한다.
- **가비지 컬렉션**(GC, garbage collection) : 클래스 인스턴스는 사용자 코드에 의해 명시적으로 생성되고 가비지 컬렉션에 의해 자동으로 파괴된다
- **기본 자료형을 명확하게 정의하여 플랫폼 독립성 보장** : C/C++ 등의 전통적인 언어는 플랫폼에 따라 int형의 크기가 변한다. JVM은 **기본 자료형을 명확하게 정의하여 호환성을 유지** 하고 **플랫폼 독립성은 보장** 한다.
- **네트워크 바이트 오더(network byte order)** : 자바 클래스 파일은 네트워크 바이트 오더를 사용. 리틀 엔디안이나, 빅 엔디안 사이에서 플랫폼 독립성을 유지하려면 고정된 바이트 오더를 유지해야 하므로 네트워크 전송 시에 사용하는 바이트 오더인 네트워크 바이트 오더를 사용한다. 네트워크 바이트 오더는 **빅 엔디안** 이다.

▼ 빅 엔디안

(2) Big Endian

: 데이터가 상위 바이트 부터 메모리에 적재(Network Ordering)

가장 최상위 바이트(0A)가 가장 낮은 메모리 주소에 저장되는 방식



(ex) 0x12345678의 32비트 값을 Big Endian으로 표현하면 :

0x12 0x34 0x56 0x78

낮은 주소 --> 높은주소

- 플랫폼 독립성 보장을 위해 사용됨

이를 조사하면서 여러가지 의문의 들었다. 그에 대해 아래에 정리해본다.

? 스택 기반 레지스터 기반과의 차이점?

가상머신의 구현체는 명세서를 어떻게 구현하냐에 따라 여러 종류가 된다. 보통 물리적인 CPU에 의해 처리되는 동작을 흉내낼 수 있어야 하며, 아래와 같은 컨셉을 가진다.

[가상 머신의 필수요소]

- 소스 코드를 실행가능한 바이트코드로 변환한다.

- **명령어와 피연산자**를 포함하는 **데이터구조**를 갖는다.
- 함수를 실행하기 위한 **콜 스택**
- **IP**(Instruction Pointer - 다음 실행할 곳을 지정하는 **포인터**)
- **가상 CPU** - 다음 명령어를 **fetch & 명령어 해석 & 명령 실행**

[Stack 기반 가상머신]

- 대다수의 가상머신이 스택 기반
- 피연산자와 연산 후 결과를 **스택에 저장**
- **피연산자가 스택 포인터에 의해 암시적으로 처리됨**
- **코드 작성과 컴파일이 쉽고 가상머신이 빠름**

계산 과정이 귀찮고 복잡해 지더라도 하드웨어에 따라 레지스터의 개수가 다르기 때문에 다기종의 디바이스에 지원하기 위해 채택했다고 생각함

[레지스터 기반 가상머신]

- 실제 하드웨어와 비슷해서 코드 생성기가 **코드를 생성하기 쉽고 빠르다.**
 - 스택 기반 VM은 동일한 피연산자를 여러 번 푸시하지만, 레지스터 기반은 **적절한 양의 레지스터를 할당**하고 작업하여 **작업량과 CPU 시간을 크게 줄일 수 있다.**
- 스택으로 밀고 들어가는 **오버헤드가 존재하지 않음**
- 같은 연산을 할 때 **더 적은 명령어로 연산 수행**이 가능하다.
- **일부 최적화 가능**

? **심볼릭 레퍼런스**

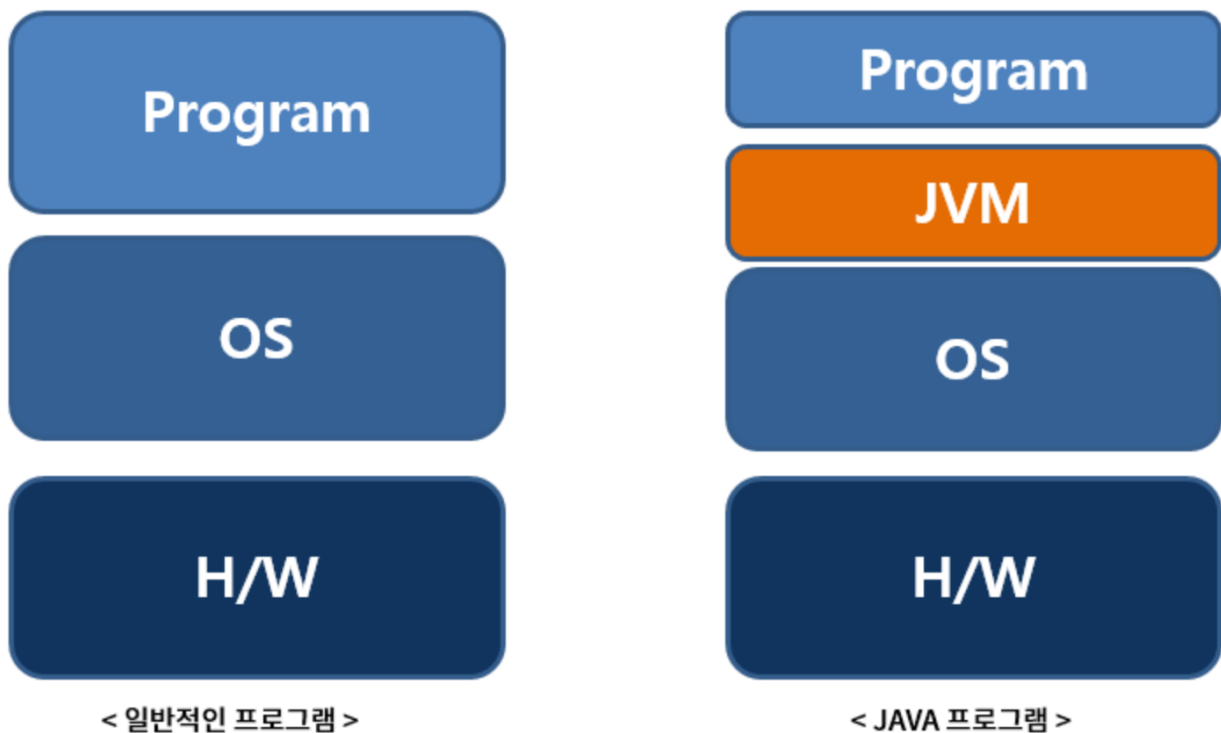
- 프로그램 실행을 위한 API를 클래스가 직접 가지는 것이 아닌 **이름을 통한 참조값**을 이용해서 실행할 때 메모리 상에서 API를 호출할 수 있도록 **이름을 주소로 대체하는 특징**을 의미

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

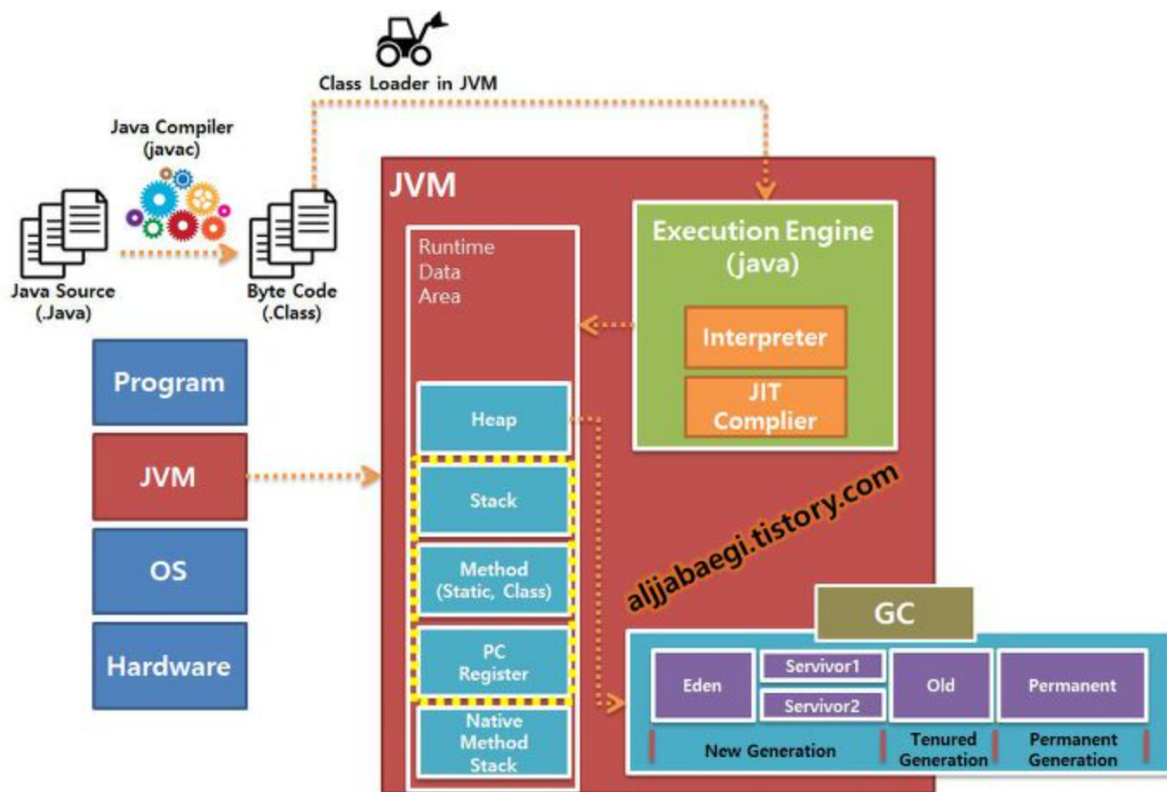
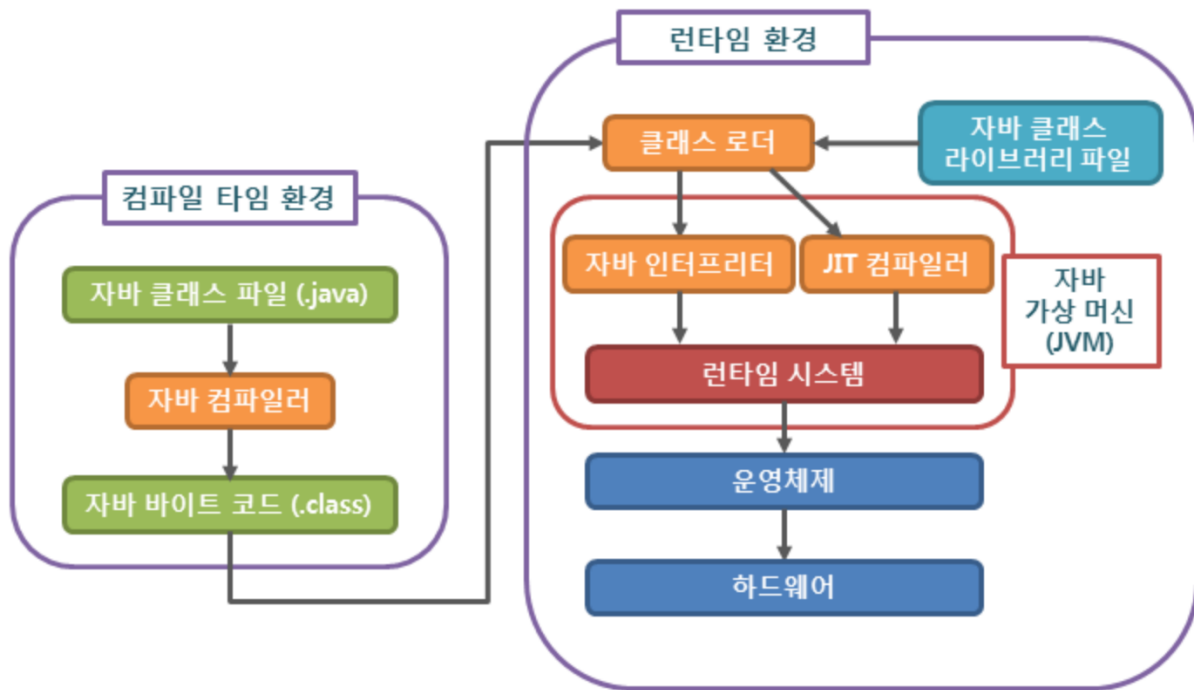
<https://s2choco.tistory.com/13>

컴파일 하는 방법

자바는 **OS에 독립적인** 특징을 가지고 있다. 그것을 가능하게 한 것이 JVM이다. 이를 어떻게 가능하게 했는지 자바 컴파일 과정을 통해 알아보자.

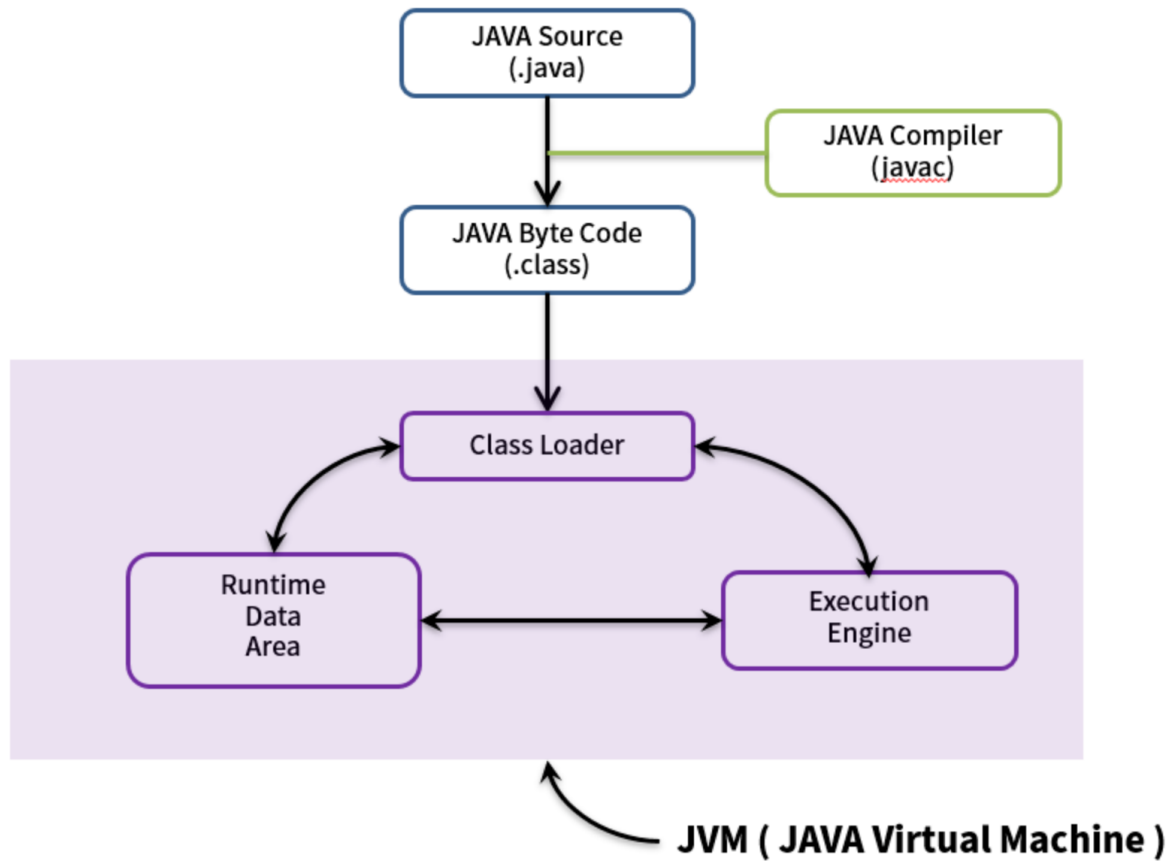


컴파일 순서

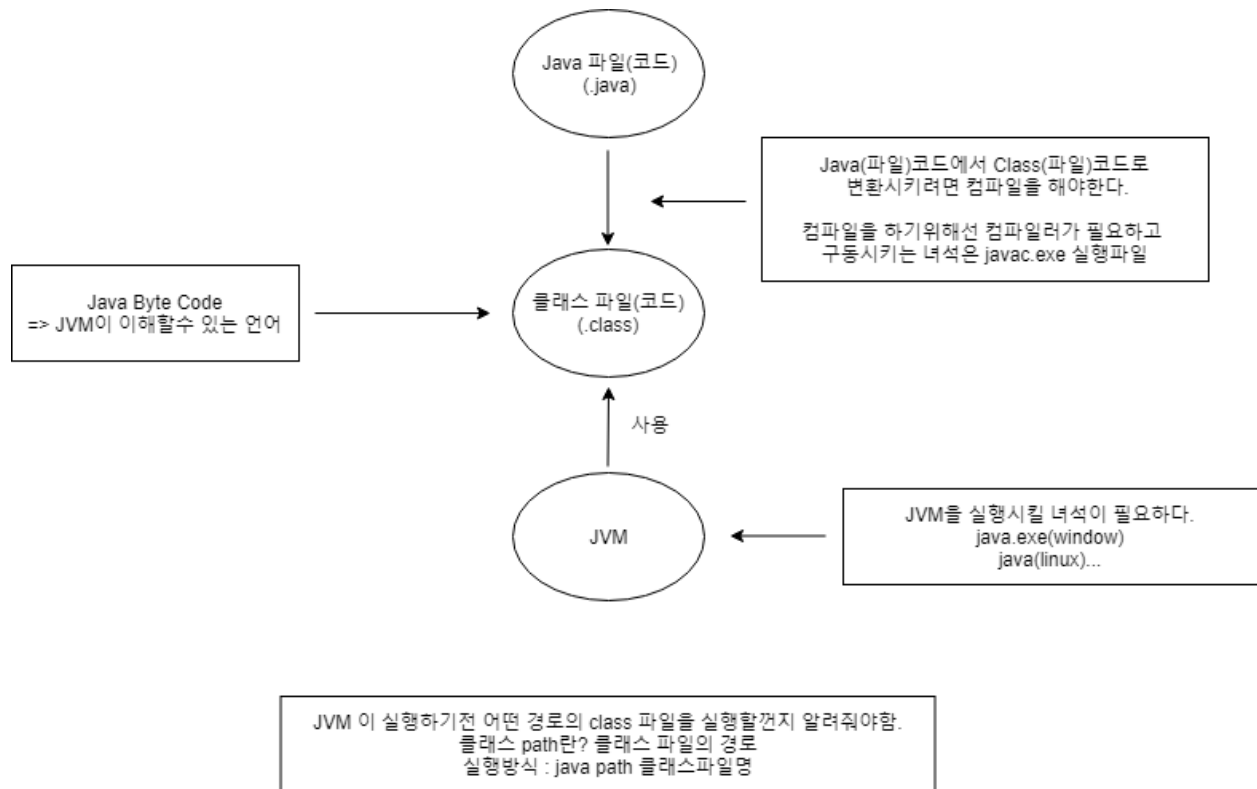


1. 자바 소스코드(.java)를 작성한다.
2. 자바 컴파일러(java Compiler)가 자바 소스파일을 컴파일한다.
 - 이때 나오는 파일은 자바 **바이트 코드(.class)**파일로 아직 컴퓨터가 읽을 수 없는 자바 가상 머신이 이해할 수 있는 코드이다. 바이트 코드의 각 명령어는 1바이트 크기의 **Opcode**와 추가 피연산자로 이루어져 있다.
3. 컴파일된 바이트 코드를 JVM의 클래스로더(Class Loader)에게 전달.
4. 클래스 로더는 동적로딩(Dynamic Loading)을 통해 필요한 클래스들을 로딩 및 링크하여 런타임 데이터 영역(Runtime Data area), 즉 JVM의 메모리에 올린다.
 - **클래스 로더** 세부 동작
 - a. **로드** - 클래스 파일을 가져와서 JVM의 메모리에 로드한다.
 - b. **검증** - 자바 언어 명세(Java Language Specification) 및 JVM 명세에 명시된 대로 구성되어 있는지 검사한다.
 - c. **준비** - 클래스가 필요로 하는 메모리를 할당(필드, 메서드, 인터페이스 등등)
 - d. **분석** - 클래스의 상수 풀 내 모든 심볼릭 레퍼런스를 다이렉트 레퍼런스로 변경한다.
 - e. **초기화** - 클래스 변수들은 적절한 값으로 초기화한다. (static 필드)
5. **실행엔진(Execution Engine)**은 JVM 메모리에 올라온 바이트 코드들을 명령어 단위로 하나씩 가져와서 실행. 이때, 실행 엔진은 두 가지 방식으로 변경한다.
 - a. **인터프리터** - 바이트 코드 명령어를 하나씩 읽어서 해석하고 실행한다. 하나하나의 실행은 빠르나, **전체적인 실행 속도가 느리다**는 단점을 가짐
 - b. **JIT 컴파일러 (Just-In-Time Compiler)** - 인터프리터의 단점을 보완하기 위해 도입된 방식. 바이트코드 **전체를 컴파일**하여 **바이너리 코드로 변경**하고 이후에는 해당 메서드를 더 이상 인터프리팅 하지 않고, **바이너리 코드로 직접 실행하는 방식**이다. 하나씩 인터프리팅하여 실행하는 것이 아니라 바이트 코드 전체가 컴파일된 바이너리 코드를 실행하는 것이기 때문에 **전체적인 실행속도는 인터프리팅 방식보다 빠르다**.

자바 코드(JAVA Code) 실행 과정



실행하는 방법



▼ 일반적으로 IDE에서 .java파일을 만들어 실행시키지만, 실무에선 .java파일을 IDE가 아닌 다른 환경에서 실행해야 하는 경우가 종종 있다.

일반적인 방법

1. C 드라이브에 Main.java 파일 생성

Main.java
2018-12-31 오후...
JAVA 파일
1KB

Main.java - 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

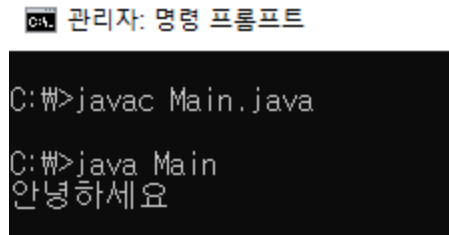
```

public class Main {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("안녕하세요");
    }
}

```

2. cmd 창에서 컴파일과 실행

- 컴파일 - **javac** Main.java
→ Main.class 파일이 생성됨
- 실행 - **java** Main



```
C:\>javac Main.java
C:\>java Main
안녕하세요
```

package가 있을 때 (package Test.Now;가 있다고 가정)

1. Main.java 자바 파일에 package Test.Now; 를 추가함
2. 파일에 package가 있다면 Main.java 를 컴파일한 Main.class는 그 경로에 존재해야 하며, .class 파일을 실행시키기 위해선 package 경로가 붙어야 한다. (java Test.now.Main)
3. javac -d . Main.java
→ 현재 디렉토리(.)에 Main.java의 package 디렉토리를 만들어줘라는 의미

Main.java 파일에 package Test.Now; 추가한 후 javac Main.java를 하면 Main.class는 나오지만, 실행하면 오류가 발생한다.

- class 파일이 package의 경로에 존재하지 않는다.
- class 파일을 실행하기 위해선 package의 경로를 적어줘야 한다.

출처 : <https://yoonemong.tistory.com/183>

바이트코드란 무엇인가

WORA(Write Once Run Anywhere)를 구현하기 위해 JVM은 사용자 언어인 자바와 기계어 사이의 중간 언어인 자바 바이트코드를 사용한다. 이 자바 바이트코드가 자바 코드를 배포하는 가장 작은 단위이다.

▼ 재미있는 사례

자바 바이트코드에 대해 설명하기 전에 한 가지 사례를 살펴 보자. 이 사례는 개발 과정에서 실제로 발생한 것을 변경, 요약한 것이다.

현상

원래 잘 동작하던 애플리케이션이 라이브러리 업데이트 이후로 다음과 같은 오류를 내고 동작하지 않는다.

```
Exception in thread "main" java.lang.NoSuchMethodError: com.nhn.user.UserAdmin.addUser(Ljava/lang/String;)V
at com.nhn.service.UserService.add(UserService.java:14)
at com.nhn.service.UserService.main(UserService.java:19)
```

애플리케이션 코드는 변경하지 않았으며 다음과 같다.

```
// UserService.java
...
public void add(String userName) {
    admin.addUser(userName);
}
```

업데이트된 라이브러리 소스코드와 원래 소스코드는 다음과 같다.

```
// UserAdmin.java - 업데이트된 소스코드
...
```

```

public UseraddUser(String userName) {
    User user =new User(userName);
    User prevUser = userMap.put(userName, user);
    return prevUser;
}
// UserAdmin.java - 원래 소스코드
...
publicvoidaddUser(String userName) {
    User user =new User(userName);
    userMap.put(userName, user);
}

```

즉, 반환값이 없던 addUser() 메서드가 User 클래스 인스턴스를 반환하는 메서드로 변경되었다. 그러나, 애플리케이션 코드는 addUser() 메서드의 반환값을 사용하지 않으므로 변경하지 않았다. 보기에는 com.nhn.user.UserAdmin.addUser() 메서드는 여전히 존재하는 것 같은데 왜 NoSuchMethodError가 발생할까?

원인

애플리케이션 코드를 새로운 라이브러리로 다시 컴파일하지 않았기 때문이다. 즉, 애플리케이션 코드는 우리가 보기에는 반환값과 무관하게 메서드를 호출하고 있는 것 같지만, 실제로 컴파일된 클래스 파일은 반환값까지 지정된 메서드를 지칭한다.

이는 다음 오류 메시지를 살펴보면 확실히 알 수 있다.

```

java.lang.NoSuchMethodError: com.nhn.user.UserAdmin.addUser(Ljava/lang/String;)V

```

NoSuchMethodError는 "com.nhn.user.UserAdmin.addUser(Ljava/lang/String;)V"라는 메서드를 찾지 못해서 발생했다. 여기서 관심을 가질 부분은 "Ljava/lang/String;"과 마지막의 "V"이다. 자바 바이트코드의 표현에서 "L;"은 클래스 인스턴스이다. 즉, addUser() 메서드는 java/lang/String 객체 하나를 파라미터로 받는 메서드이다. 이 사례의 라이브러리에서는 파라미터가 변경되지 않았으므로 이 파라미터는 정상이다. 위 메시지 마지막의 "V"는 메서드의 반환값을 나타낸다. 자바 바이트코드 표현에서 "V"는 반환값이 없음을 의미한다. 즉, 위 오류 메시지는 java.lang.String 객체 1개를 파라미터로 받고 반환값은 없는 com.nhn.user.UserAdmin.addUser라는 메서드를 찾지 못했다는 의미이다.

애플리케이션 코드는 이전 라이브러리로 컴파일되었으므로, "V"를 반환하는 메서드를 호출하도록 class 파일에 기록되어 있지만, 새로 변경된 라이브러리에서 "V"를 반환하는 메서드는 없어지고, "Lcom/nhn/user/User;"를 반환하는 메서드가 추가되었기 때문에 NoSuchMethodError가 발생한 것이다.

참고

오류 자체는 새로운 라이브러리를 다시 컴파일하지 않았기 때문에 발생한 것이지만, 이 사례에서는 라이브러리 제공자의 잘못이 크다고 할 수 있다. public으로 공개된 메서드의 반환값이 없다가 User 클래스 인스턴스를 반환하도록 변경된 것이므로, 이것은 명백한 메서드 시그너처 변경이다. 즉, 라이브러리의 하위 호환성이 깨진 것이므로 라이브러리 제공자는 메서드가 변경되었다는 것을 반드시 사용자에게 알렸어야 한다.

JVM을 이야기 할 때 자바 바이트코드는 뺄 수 없다. **JVM**은 **자바 바이트코드를 실행하는 실행기**이다. 자바 컴파일러는 C/C++등의 컴파일러처럼 고수준 언어를 기계어, 즉 직접적인 CPU 명령으로 변환하는 것이 아니라, 개발자가 이해하는 **자바 언어를 JVM이 이해하는 자바 바이트코드로 번역한다**. 따라서 **자바 바이트코드는 플랫폼 의존적인 코드가 없기 때문에 JVM(정확하게 말하자면 같은 프로파일의 JRE)이 설치된 장비라면 CPU나 운영체제가 다르더라도 실행할 수 있고** (윈도우 PC에서 개발하여 컴파일한 클래스 파일을 리눅스 장비에서도 별다른 변경 없이 실행한다), 컴파일 결과물의 크기가 소스코드의 크기와 크게 다르지 않으므로 **네트워크로 전송하여 실행하기가 쉽다**.

클래스 파일 자체는 바이너리 파일이므로 사람이 이해하기 쉽지 않다. 이 점을 보완하기 위해 JVM 벤더들은 javap라는 역어셈블러(disassembler)를 제공한다. javap를 이용한 결과물을 **훅** | 자바 어셈블리라고 부른다. 앞의 사례에서 애플리케이션 코드 UserService.add() 메서드를 javap -c 옵션으로 역어셈블한 결과물은 다음과 같다.

```
public void add(java.lang.String);
Code:
 0: aload_0
 1: getfield #15; //Field admin:Lcom/nhn/user/UserAdmin;
 4: aload_1
 5: invokevirtual #23; //Method com/nhn/user/UserAdmin.addUser:(Ljava/lang/String;)V
 8: return
```

이 결과물에서 addUser() 메서드를 호출하는 부분은 4번째줄인 **"5: invokevirtual #23;"**이다. 이는 23번 인덱스에 해당하는 메서드를 호출하라는 의미이며, 23번 인덱스의 메서드를 **javap 프로그램이 친절하게 주석으로** 달아주었다. invokevirtual은 자바 바이트코드에서 메서드를 호출하는 가장 기본적인 명령어의 OpCode(operation code)이다. 참고로, 자바 바이트코드에서

메서드를 호출하는 명령어 OpCode는 invokeinterface, invokespecial, invokestatic, invokevirtual의 4가지가 있으며 각각의 의미는 다음과 같다.

- **invokeinterface** - 인터페이스 메서드 호출
- **invokespecial** - 생성자, private 메서드, 슈퍼 클래스의 메서드 호출
- **invokestatic** - static 메서드 호출
- **invokevirtual** - 인스턴스 메서드 호출

자바 바이트코드의 명령어는 OpCode와 피연산자(Operand)로 분리할 수 있으며, invokevirtual과 같은 OpCode는 2바이트의 피연산자를 필요로 한다.

위의 애플리케이션 코드를 업데이트된 라이브러리로 다시 컴파일하여 다시 역어셈블하면 다음 결과를 얻을 수 있다.

```
public void add(java.lang.String);
Code:
0: aload_0
1: getfield #15; //Field admin:Lcom/nhn/user/UserAdmin;
4: aload_1
5: invokevirtual #23; //Method com/nhn/user/UserAdmin.addUser:(Ljava/lang/String;)Lcom/nhn/user/User;
8: pop
9: return
```

23번에 해당하는 메서드가 "Lcom/nhn/user/User;"를 반환하는 메서드로 변환됐다.

위의 역어셈블 결과물에서 **코드 앞의 숫자**는 무엇을 의미할까? 바로 **바이트 번호**이다. JVM이 실행하는 코드를 굳이 자바 "바이트"코드라고 하는 이유가 바로 이것일 것이다. 즉, 위의 aload_0, getfield, invokevirtual 같은 바이트코드 명령어 OpCode들은 1 바이트의 바이트 번호로 표현된다. aload_0 = 0x2a, getfield = 0xb4, invokevirtual = 0xb6 등이다. 따라서, 자바 바이트코드 명령어 OpCode는 최대 256개라는 점을 알 수 있다.

aload_0, aload_1과 같은 OpCode는 피연산자가 필요 없다. 따라서 aload_0 바로 다음 바이트가 다음 명령어의 OpCode가 된다. 그러나 getfield, invokevirtual은 2바이트의 피연산자가 필요하다. 따라서 첫 번째 바이트에 있는 getfield의 다음 명령어는 2바이트를 건너뛴 네 번째 바이트에 기록된다. 위의 바이트코드를 Hex Editor로 보면 다음과 같다.

```
2a b4 00 0f 2b b6 00 17 57 b1
```

자바 바이트코드에서 클래스 인스턴스는 "L;", void는 "V"로 표시되는 것처럼 다른 타입들도 고유의 표현이 있다. 이 표현을 정리하면 다음과 같다.

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

JIT 컴파일러란 무엇이며 어떻게 동작하는지

JVM 구성 요소

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

JDK와 JRE의 차이