아이템 25. 톱레벨 클래스는 한 파일에 하나만 담으라

- 자바 컴파일러가 하나의 소스파일에 여러 top-level 클래스들을 작성하는 것을 막지는 않지만 그렇게 하지 말아라
- 정 그러고 싶다면, static member class 를 사용하는 것을 고려해봐라

이번 아이템을 알아보기 전에 Java 컴파일에 대해서 알아봅시다

https://homoefficio.github.io/2019/01/31/Back-to-the-Essence-Java-%EC%BB%B4%ED%8C%8C%EC%9D%BC%EC%97%90%EC%84%9C-%EC%8B%A4%ED%96%89%EA%B9%8C%EC%A7%80-1/

참고자료: Back to the Essence - Java 컴파일에서 실행까지 - (1)

컴파일

- 전통적인 의미에서 컴파일이란?

  어떤 언어로 된 `소스코드` → 기계가 인식할 수 있는 `네이티브 코드`

- 자바에서의 컴파일?

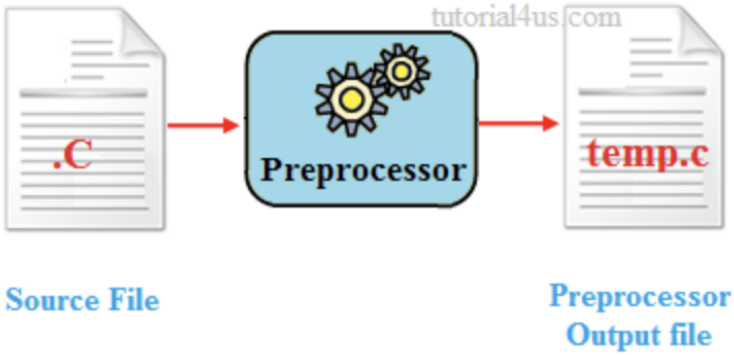
  (주로) `자바코드` → JVM 이 인식할 수 있는 JVM 명령어 코드 (`바이트 코드`)

  (드물게) `바이트 코드` → `네이티브 코드` 로 변환하는 과정을 일컬을 때도 있다. (ex) JIT 컴파일러

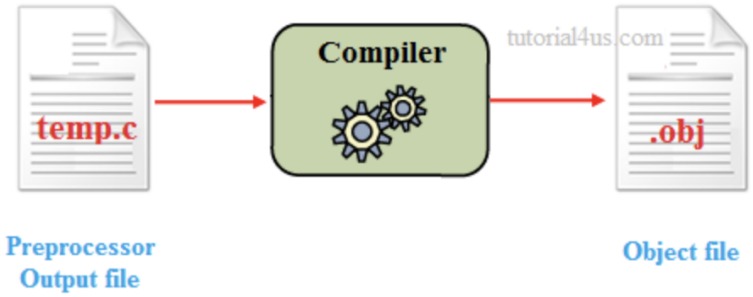
실행파일 생성 과정

- C 로 작성된 코드가 실행 파일을 만드는 과정

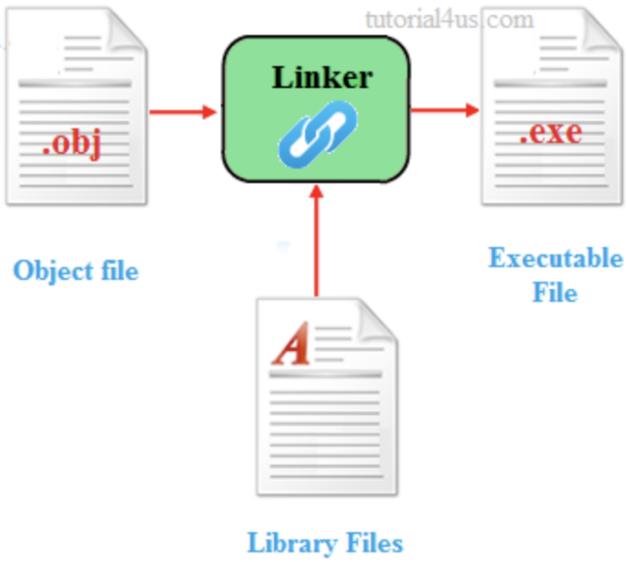
```
전처리 - 컴파일 - 어셈블리 - 링크 - 실행파일 생성
```



1. 전처리 (주석 제거, 매크로 인라인 화, include 파일 인라인 화)



2. 컴파일(어셈블리어 코드로 변환) + 어셈블리(기계어 코드로 변환) = 이 두 과정을 합쳐서 컴파일이라고 하기도 함



3. 링커가 기계어 코드와 공유 라이브러리 등 다른 코드를 합쳐서 최종 실행 파일 생성

- Java 로 작성된 코드가 실행 파일을 만드는 과정

  자바 소스 코드를 컴파일하는 과정이 몇 단계로 구성되는지는 구체적으로 스펙에 규정되어 있지는 않다

  ```
  자바 소스 코드 파일(.java) -> javac 컴파일러 -> JVM 바이트코드(.class)
  ```

  자바는 컴파일 결과로 나온 바이트코드가 JVM에 의해 실행되면서 네이티브 기계어 코드로 변환되므로, 프로그램 실행 전에 네이티브 기계어 코드를 만들어내는 어셈블리 단계가 없다고 볼 수 있다.

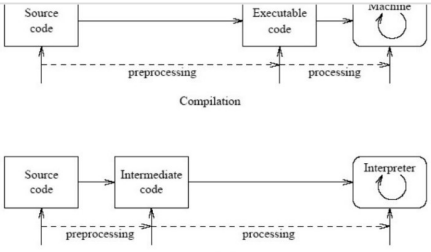
  마찬가지로 링크 단계도 프로그램 실행 전에 수행되지 않고 JVM에 의해 프로그램이 실행될 때 동적으로 수행된다

컴파일러와 인터프리터의 차이

컴파일러와 인터프리터의 차이

https://blog.naver.com/ehcibear314/221228096291 우선 윗 글을 먼저 읽고 오는 것이 좋다. 컴파일러, 인...

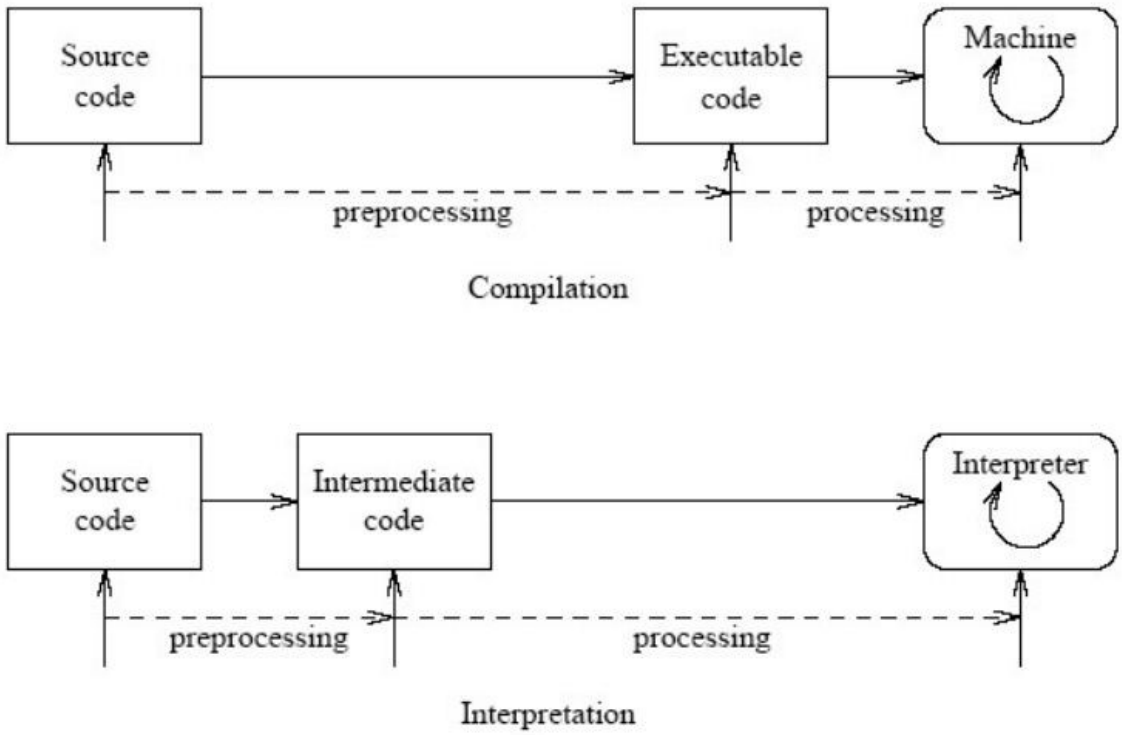
https://m.blog.naver.com/ehcibear314/221228200531



컴파일러나 인터프리터 둘 다 고레벨 언어를 기계어로 변환하는 것은 맞으나 과정에 차이가 있다

인터프리터 언어? 컴파일 언어?

Aa 이름	☑ 인터프리터 언어인가?	☑ 컴파일 언어인가?
Python	☑	☐
C, C++	☐	☑
Java	☑	☑



일반적인 컴파일러와 인터프리터의 차이

컴파일러

- 전체 소스코드를 보고 컴퓨터 프로세서가 실행 할 수 있는 기계어로 바로 변환

인터프리터

- 소스코드의 각 행을 연속적으로 분석하며 실행

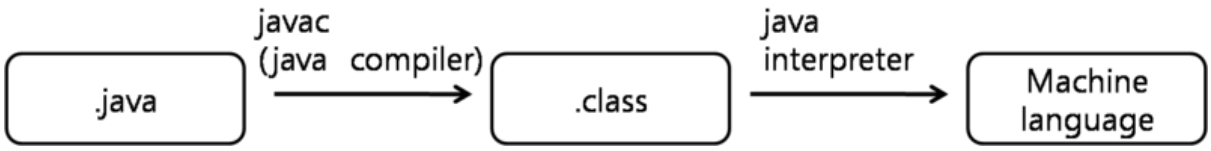
- 고레벨 언어를 바로 기계어로 변환
- 일반적으로 인터프리터에 비해 실행 시간이 빠르다
- 소스 코드가 유출되지 않아 보안에 좋다

- 고레벨 언어를 바로 기계어로 변환하지 않고 중간형태로 변환 후 각 행마다 실행, 이 중간 코드는 다른 프로그램에 의해 실행된다.
- 일반적으로 컴파일러에 비해 실행 시간이 느리다 (각 행마다 실행해야 해서?)
- 소스 코드가 유출될 수 있어 보안에 좋지 않다

지금에 와서는 인터프리터 방식이라고 해서 컴파일러에 의해 컴파일된 목적 프로그램에 비해서 그다지 속도상 뒤쳐지지 않게 되었고, 컴파일러와 인터프리터의 장점을 혼합하여 일차적인 컴파일을 한 뒤 인터프리터가 읽고 사용하는 방식으로 확장되어 원본 소스코드가 공개되는 상황도 방지할 수 있게 되었다

...고 합니다

자바를 통해 컴파일러와 인터프리터의 차이를 알아보자



1. 컴파일

   `.java` 파일을 자바 컴파일러 (javac) 가 `.class` 파일로 변환한다

   `.class` 는 바이트 코드로 쓰여 있다.

   ▼ 컴파일 이후 나오는 `.class` 바이트코드가 기계어인가?

   ㅇㅇ

   컴파일러는 기계가 실행할 수 있는 기계어로 변환하는 프로그램.

   그럼 컴파일의 결과물인 .class 바이트코드는 기계어인가?

   여기서 말하는 기계어가 반드시 하드웨어는 아님. JVM : Java Virtual Machine. 즉 JVM 도 Machine 임.

JVM 이 읽고 실행할 수 있는 기계어 = JVM 을 위한 기계어 = .class = 바이트 코드

2. 인터프리터

   `.class` 파일 내의 바이트 코드를 특정 환경의 기계에서 실행될 수 있도록 변환한다.

   IBM PC 에서 작성된 .class 파일을 가져다가 메킨토시에서도 실행할 수 있다는 것

   (C 의 경우 컴파일부터 다시 해야 한다)

언제 컴파일러 언어를 사용하고 언제 인터프리터 언어를 사용해야 하는가?

소스코드를 바로 기계어로 변환하는 컴파일러의 경우 프로그램이 작성된 기계상에서 실행할 때 매우 효율적이다.

→ 이는 대부분의 하드웨어 제어 시스템의 프로그래밍 언어가 C 인 이유이다.

→ 그러나 이와 동시에 기계 종류에 종속된다는 말

→ 인터프리터의 경우 플랫폼에 종속되지 않는다. But, 실행시간이 느리긴 하다

다시 원래 아이템으로 돌아와서

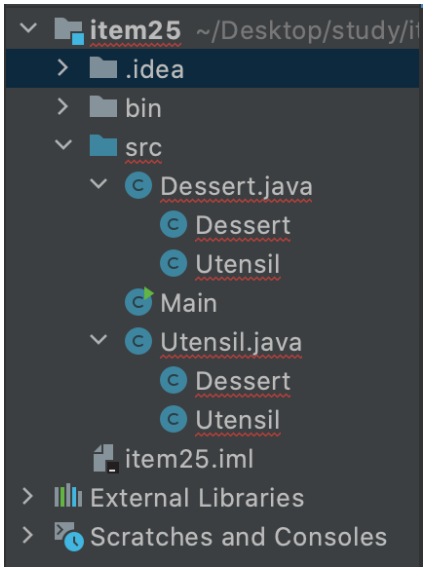
top-level class 에 클래스가 하나만 있어야 된다는데, 만약 두개 있으면 어떻게 되느냐?

클래스 정의가 중복될 때 문제가 생길 수 있다.

▼ Main.java

```
public class Main {
    public static void main(String[] args) {
        System.out.println(Utensil.NAME + Dessert.NAME);
    }
}
```

▼ Dessert.java - pot, pie




```java
class Utensil {
    static final String NAME = "pot";
}

class Dessert {
    static final String NAME = "pie";
}
```

▼ Utensil.java - pan, cake

```java
class Utensil {
    static final String NAME = "pan";
}

class Dessert {
    static final String NAME = "cake";
}
```

IDE 에서 여러가지를 해보자

`javac Main.java Dessert.java` → pancake ??.. 아님 potpie 나왔음

`javac Main.java` → pancake

`javac Main.java Utensil.java` → pancake

`javac Dessert.java Main.java` → potpie

한 파일에 여러 클래스가 존재한다면, 클래스가 중첩될 가능성이 있고, 중첩되는 경우 어떤 클래스 정의를 사용할지는 컴파일러에 전달되는 순서에 따라 다른 결과가 발생한다.

한 파일 내에 여러개의 탑 레벨 클래스를 놓고 싶다면 static member class 를 사용하면 어떨지 고려해봐라

```java
public class Test {
    public static void main(String[] args) {
        System.out.println(Utensil.NAME + Dessert.NAME);
    }

    private static class Utensil {
        static final String NAME = "pan";
    }
```

```
    private static class Dessert {
        static final String NAME = "cake";
    }
}
```