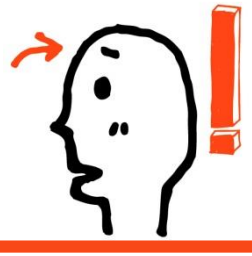
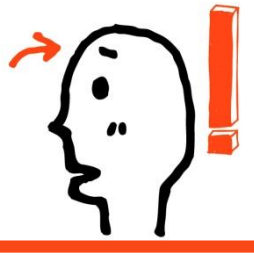
컴파일러와 인터프리터의 개념



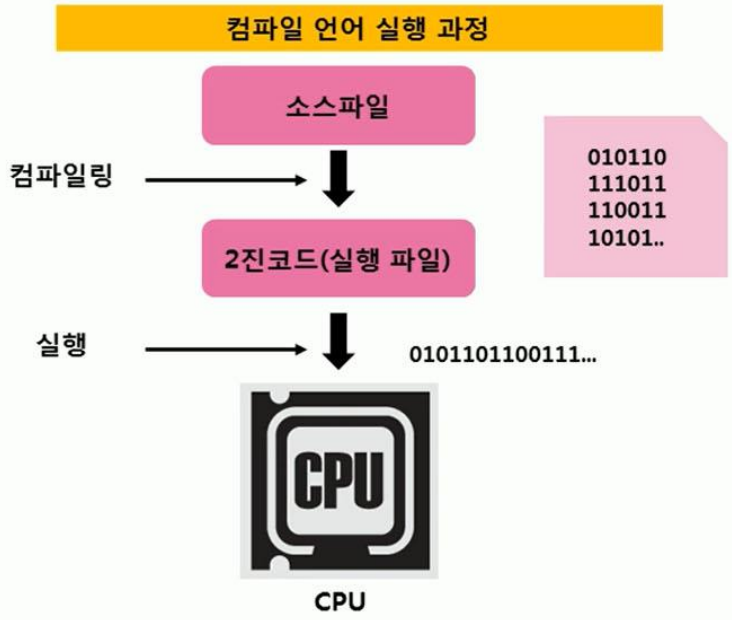
컴파일러와 인터프리터의 비교

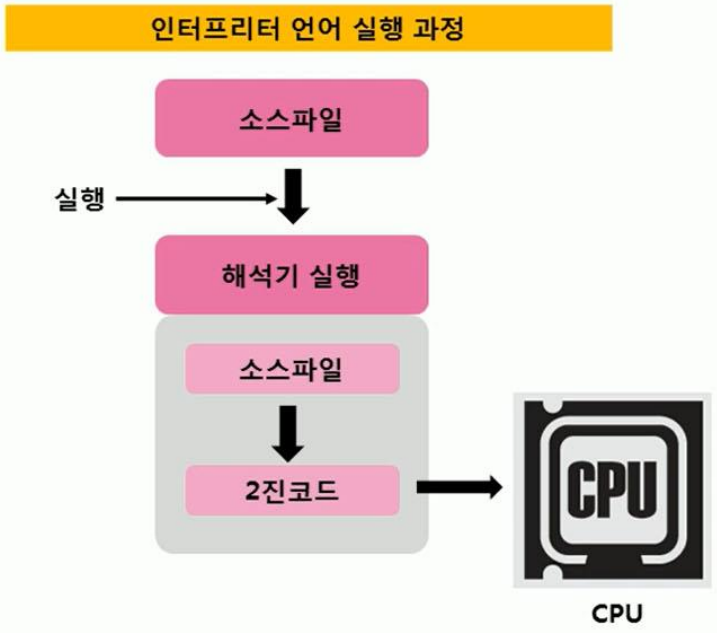
비교	컴파일러	인터프리터
개념	고급 언어로 작성된 소스 프로그램 전체를 목적 프로그램으로 번역 후 링킹 작업을 통해 실행 프로그램 생성	고급 언어로 작성된 소스 프로그램을 한 줄씩 번역하고 번역과 동시에 한 줄씩 즉시 실행
목적프로그램	생성함	생성하지 않음
번역 속도	속도 느림	속도 빠름
실행 속도	속도 빠름	속도 느림
장점	한번 번역 후에는 다시 번역하지 않아 실행 속도 빠름	소스 프로그램의 변화에 대한 반응이 빠름
단점	번역과 실행 과정이 있어 번역 시간이 오래 걸림	프로그램 실행 시 매번 번역해야 하므로 실행 속도 느림
사례	FORTRAN, PASCAL, C, C++	BASIC, LISP, PROLOG, SNOBOL

컴파일러와 인터프리터의 개념

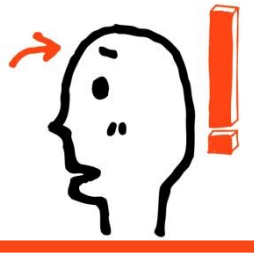


- 현재 사용 중인 고급 언어 실행 과정

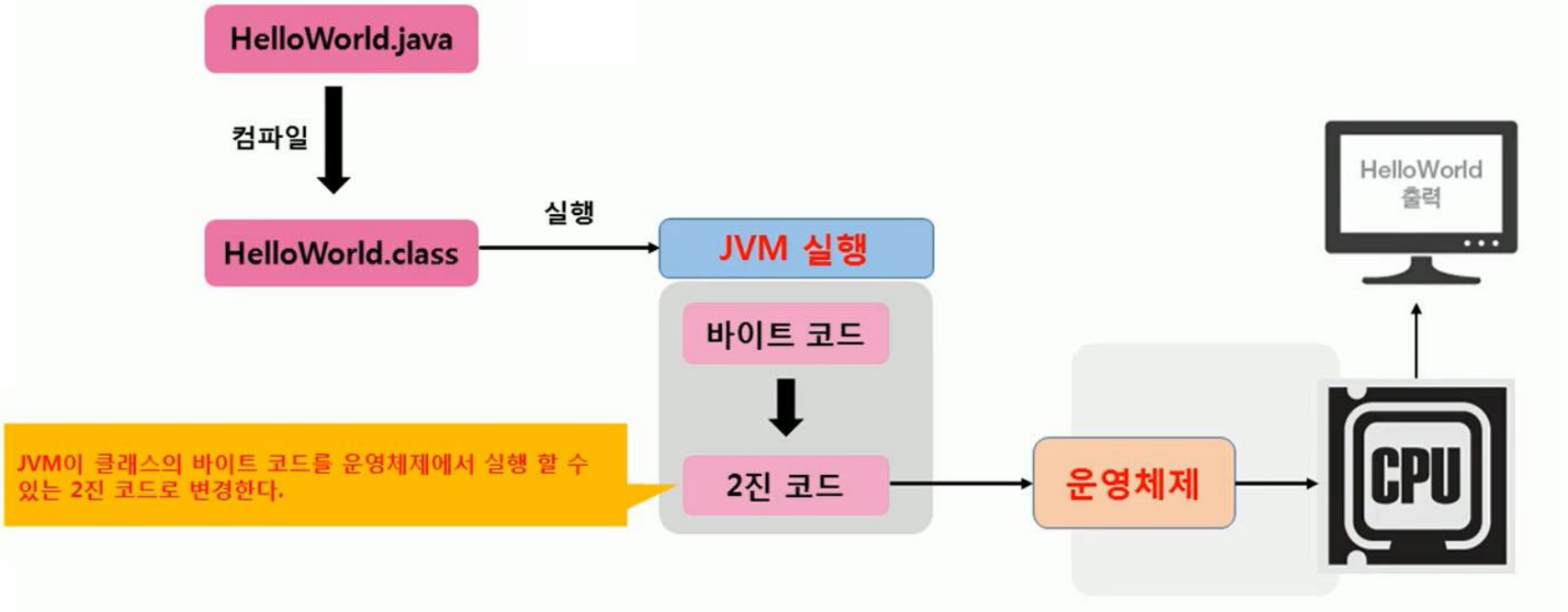




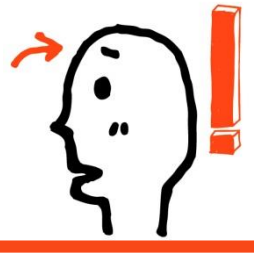
컴파일러와 인터프리터의 개념



- 자바 소스 실행과정



싱글톤

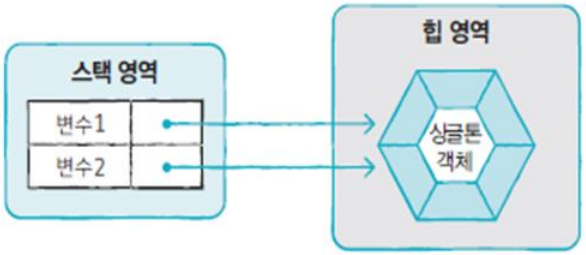


❖ 싱글톤 (singleton)

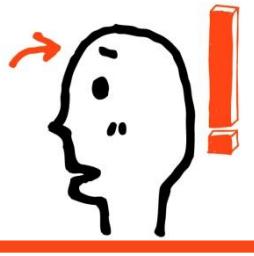
- 전체 프로그램에서 단 하나의 객체만 만들도록 보장하는 코딩 기법
- 싱글톤 작성 방법
  - 클래스 외부에서 new 연산자 통해 생성자 호출하는 것 불가하도록 private 접근 제한자 사용
  - 자신의 타입인 정적 필드 선언 후 자신의 객체 생성해 초기화
  - 외부에서 호출할 수 있는 getInstance() 선언
  - 정적 필드에서 참조하는 자신의 객체 리턴

```
public class 클래스 {
  //정적 필드
  private static 클래스 singleton = new 클래스();

  //생성자
  private 클래스() {}

  //정적 메소드
  static 클래스 getInstance() {
    return singleton;
  }
}
```

```
클래스 변수1 = 클래스.getInstance();
클래스 변수2 = 클래스.getInstance();
```



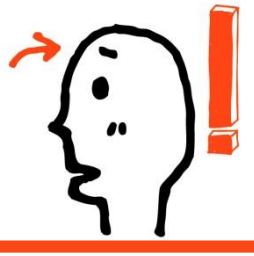
싱글톤



소스 코드 Singleton.java

```
package sec05.exam04;

public class Singleton {
  private static Singleton singleton = new Singleton();

  private Singleton() {}

  static Singleton getInstance() {
    return singleton;
  }
}
```

싱글톤



객체 소스 코드 SingletonExample.java

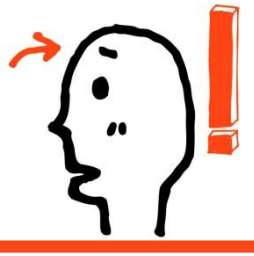
```
package sec05.exam04;

public class SingletonExample {
  public static void main(String[] args) {
    /*
    Singleton obj1 = new Singleton();
    Singleton obj2 = new Singleton();
    */

    Singleton obj1 = Singleton.getInstance();
    Singleton obj2 = Singleton.getInstance();

    if(obj1 == obj2) {
      System.out.println("같은 Singleton 객체입니다.");
    } else {
      System.out.println("다른 Singleton 객체입니다.");
    }
```

← 컴파일 에러

실행결과

```
같은 Singleton 객체입니다.
```

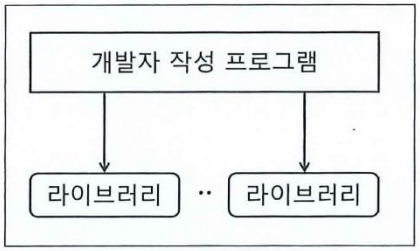
라이브러리



라이브러리란?

프로그램을 개발할 때 모든 기능을 개발자가 직접 작성하지는 않는다. 필요한 기능을 직접 작성할 수도 있지만 복잡한 로직이 담겨있는 기능의 경우 다른 사람이 작성한 모듈을 가져다 사용할 수 있다. 이때 다른 사람이 작성한 모듈을 라이브러리라고 한다. 혹은 자신이 개발한 클래스들을 라이브러리화할 수도 있다.

개발에서 자기가 직접 작성하는 부분이 갈수록 줄어들고 있다. 외부의 라이브러리를 잘 끌어다 쓰는 것이 생산성 및 프로그램의 품질을 높이는 손쉬운 방법 중의 하나이다.

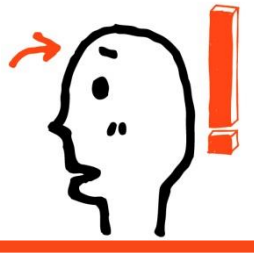


[도표] 프로그램 및 라이브러리

라이브러리라고 해서 별다른 것은 아니다. 컴파일돼있는 여러 클래스 파일을 확장자가 jar인 파일에 부가 정보와 함께 묶어놓은 형태다.

StringUtils라는 클래스가 라이브러리 jar로 제공되고 이를 사용해 다음의 프로그램을 작성한다고 하자.

라이브러리



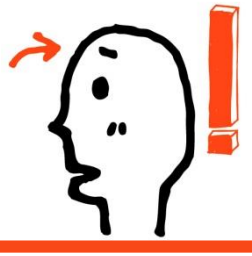
[예제] 라이브러리 사용 예제 – MyProgram.java

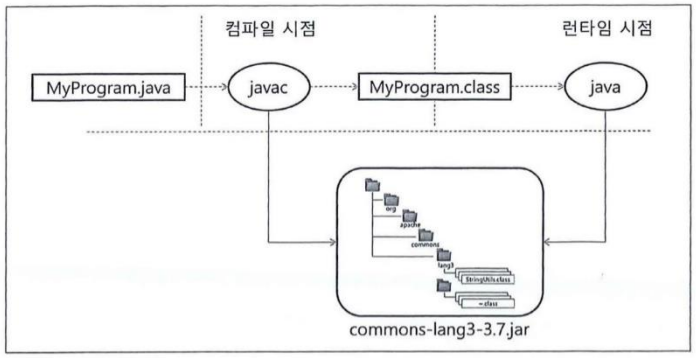
```
package com.javainhand;

import org.apache.commons.lang3.StringUtils;

public class MyProgram {

    public static void main(String[] args) {
        String str = "";
        boolean flag = StringUtils.isEmpty(str);
        System.out.println("StringUtils.isEmpty(str) : "+flag);
    }
}
```

jar 안에는 결국 클래스들이 묶여있다. 위 예에서 org.apache.commons.lang3.StringUtils를 import하고 있는데 이는 실제로 스트링 클래스에 대한 도움 기능을 모아놓은 commons-lang3-3.7.jar라는 라이브러리에 포함돼있다. 해당 라이브러리에 포함돼있는 클래스를 사용하는 코드는 여타 클래스를 사용하는 코드와 동일하다.

이렇게 jar 형태의 라이브러리를 사용하려면 첫째, 컴파일 타임 때 필요하고, 둘째, 실행 시간에 필요하다. 컴파일 타임 때 필요하다는 것은 javac 명령어를 사용해 확장자가 ~.java 파일을 ~.class로 컴파일할 때 해당 라이브러리를 명시해야 한다는 의미다. 실행 시간에 필요하다는 것은 java 명령어를 사용해 개발자가 작성한 프로그램을 실행할 때 해당 라이브러리를 명시해야 한다는 의미다.

라이브러리

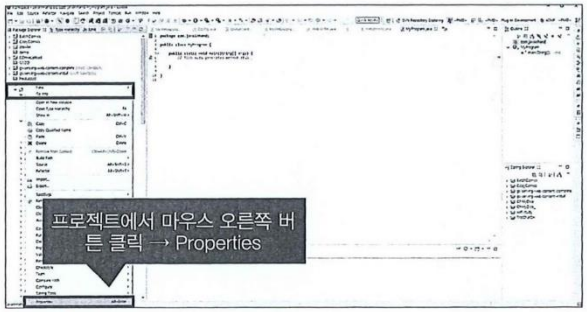




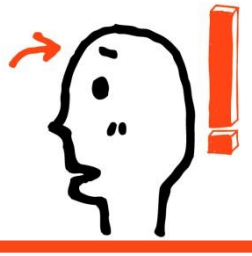
[도표] 라이브러리를 사용한 컴파일 및 실행

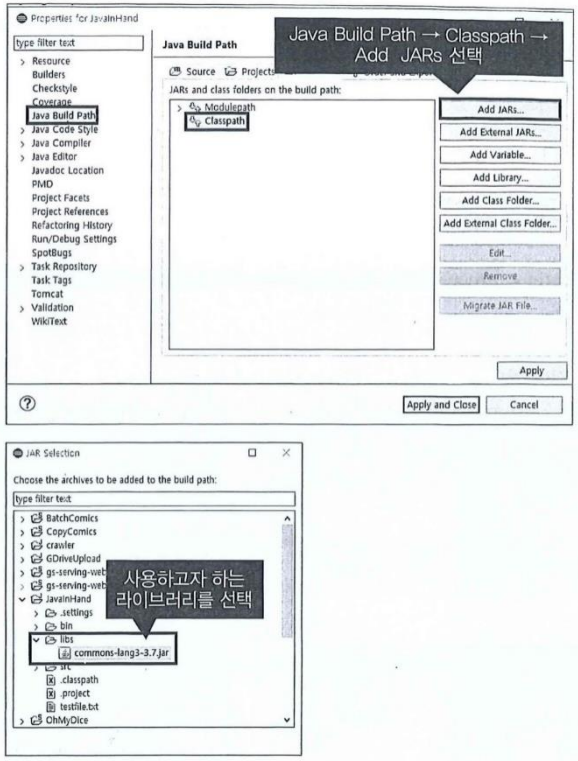
앞의 그림에서 알수 있듯이 컴파일 시점에는 컴파일러에게 참조하는 라이브러리를 알려야 하며, 실행 시점에는 가상 머신, 즉 java에게 참조하는 라이브러리를 알려야 한다. 이를 위해 javac와 java의 옵션을 사용해 추가되는 정보를 기술한다.

실무 프로젝트에서는 명령 창에서 javac나 java에 라이브러리 옵션을 주는 방식으로 프로젝트를 진행하지는 않는다. 대신 IDE나 Maven 등의 개발 환경에 라이브러리를 참조하도록 설정한다. 일단 이번에는 이클립스 환경에서 라이브러리를 추가해 보자.



라이브러리





[도표] 라이브러리 추가 작업

사용하려는 라이브러리를 주로 인터넷에서 다운로드해 특정 위치에 복사한 후 앞의 도표와 같은 작업을 수행한다. 앞의 예에서는 프로젝트 폴더 내에 libs라는 디렉터리를 만들고 그 안에 jar 파일을 내려받았다. 이렇게 이클립스 설정을 하면 우리는 이클립스를 통해서 컴파일하고 실행하기 때문에 해당 라이브러리에 대한 참조를 이클립스가 자동으로 수행한다.