5주차: 패키지

https://github.com/whiteship/live-study/issues/7

패키지

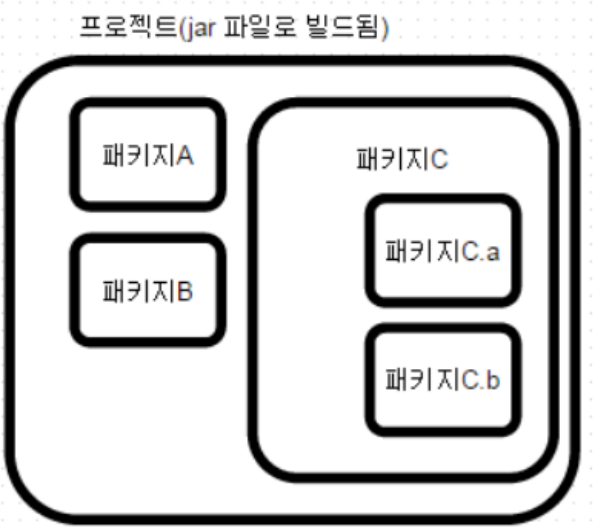
: 비슷한 성격의 자바 클래스들을 모아 놓은 자바의 **디렉토리**이다.

패키지는 만들어지면 디렉토리로 생성되고, 자바 프로젝트를 빌드하면 `프로젝트 디렉토리/dist` 디렉토리 밑에 `프로젝트명.jar` 파일이 생성되는데 이것은 프로젝트 내의 모든 컴파일된 패키지를 포함한 압축파일이다. 파일을 열어보면 이 프로젝트에 포함된 패키지들이 생성되어 있음을 알 수 있다.

+ dist 는 배포용 디렉토리. src의 코드가 컴파일된 결과물이고, 보통 SW에서 dist 는distribution(유통시키다, 배포하다) 의 약자라고 한다!
+ 비슷한 폴더명으로는 dest(destination) 이 있다.

jar 파일을 실행하기 위해선 아래 명령어를 실행하면 된다.

```
java -jar 프로젝트명.jar
```



일반적인 자바 프로젝트(jar 파일) 구조 / 이미지 출처: https://studymake.tistory.com/428

패키지를 사용하는 이유는?

1. 클래스의 **분류가 용이**하다. (비슷한 것 끼리 묶는다)
2. 패키지가 다르다면 **동일한 클래스명**을 사용할 수 있다.

패키지는 JAVA API 에서 제공하는 **Built-in packages** 와 사용자가 직접 생성하는 **User-defined packages** 로 나누어 볼 수 있다.

- User-defined Packages

사용자 지정 패키지 만들 때 `package` 키워드 사용한다.

- **package 키워드**

```
package 패키지명;
class 클래스명 { ... }
```

- Built-in Packages

짧게 Java API 에 대해 먼저 말하자면 Java API 는 자바 개발 환경에 포함되어 있는 클래스 라이브러리이다. 입력, 데이터베이스 프로그래밍 등을 관리하기 위한 구성 요소가 이에 포함되어 있다. (전체 구성요소는 오라클에서 확인 가능함.) 라이브러리는 패키지와 클래스로 나뉘는데, 단일 클래스를 가져오거나 전체 패키지를 가져올 수 있다.
라이브러리에서 클래스/패키지를 가져오기 위해서 `import` 키워드를 사용한다.

- **import 키워드**

```
import 패키지명.클래스명; // 패키지에서 특정 class 만 import
import 패키지명.*; // 패키지 전체 import
```

+ 찾아보니까 import 없이 사용할 수도 있긴하다.

  -> 패키지와 클래스를 모두 기술(**FQCN(Fully Qualified Class Name)**이라고 함)

이 경우 패키지 이름이 길거나, 사용해야 할 클래스 수가 많다면 코드가 난잡해 보이게 된다. 그래서 `import` 를 주로 사용한다. 하지만!!! import 써서 다 해결되는 거면 이게 있는 이유가 없으니까 **FQCN** 이 필요한 경우는, **서로 다른 패키지에 동일한 클래스 이름이 존재**하고, **두 패키지가 모두 import** 되어 있을 경우이다.

자바 컴파일러가 어떤 패키지에서 클래스를 로딩할지 결정할 수 없어 컴파일 에러가 발생한다.
저번에 다중 상속 얘기하면서 C++ 네임스페이스 쓰는 거랑 비슷하다고 했었는데 이것도 비슷하다!!

+ 그리고 빌트인 패키지는 import 문 없이도 그냥 사용가능하다! -> 왜..?

일단 Java API 에 속해있다는 사실을 기억해보면, Java API 는 소스 코드가 클래스 로더를 통해 읽어들여지고 자바 API 와 함께 실행되었었다! (1주차: JVM)

소스 코드가 실행되는 과정

1. JRE 가 자바 프로그램 실행 시 main() 찾기
2. main() 존재 시, Class Loader가 **목적 파일**(.class) **실행**
3. **static** 영역에 java.lang 패키지, import한 **패키지들 적재**
4. **stack** 영역에 main()**의 호출 정보**(stack frame) **적재** -> 변수 영역에 각 인자들 위치
5. main() **실행**
6. main() 의 } 를 만나면 JRE 가 JVM을 종료시켜 모든 메모리 제거

여기서 3번을 주목하면, java.lang같은 빌트인 패키지는 3에서 import 패키지들과 함께 적재가 된다.
즉, **이미 import 된 상태**라 보면 된다! lang 같은 애들은 java 가 자동으로 기술해주기 때문에 선언하지 않아도 된다!

+ **라이브러리 vs 모듈 vs 패키지**

추가로, Java API 가 클래스 라이브러리라고 했는데 라이브러리? 패키지? 그리고 파이썬에서는 **패키지**보다는 **라이브러리**와 **모듈**이란 용어를 많이 쓰는데 이 셋의 정확한 정의와 차이점이 궁금하다!

(in 파이썬)

- **라이브러리** = 여러 패키지와 모듈들을 모아놓은 것
- **패키지** = 특정 기능과 관련된 여러 모듈을 한 폴더 안에 넣어 관리. 이 폴더 = 패키지.
- **모듈** = 함수, 변수, 클래스를 모아놓은 것. 일반적으로 한 파일을 의미.

➔ 결론 : **라이브러리 ≥ 패키지 ≥ 모듈**

ex) 수학 풀이 프로그래밍 모음 라이브러리 ⊃ 구구단 패키지 ⊃ 2단을 출력하는 모듈

+ **프레임워크**

: 여러 패키지를 모아 하나의 프로그램을 구동할 수 있도록 한 묶음.

폴더 트리가 존재. 즉, 정해진 폴더 안에서 작업을 해야 프로그램이 돌아간다.

ex) 리액트 = frontend framework (react, react-dom, babel, webpack 등)

Django, Spring 등

+ 파이썬에서 `if __name__ == "__main__":` 쓰는 이유

파이썬은 **최초로 시작되는 스크립트 파일과 모듈의 차이가 없음**. 그래서 `__name__` 변수를 통해 현재 스크립트 파일이 시작점인지 모듈인지 판단한다.

`if __name__ == '__main__'` 은 현재 스크립트 파일이 프로그램의 시작점이 맞는지, 즉 스크립트 파일이 메인 프로그램으로 사용될 때와 모듈로 사용될 때를 구분하기 위한 용도이다.

얘를 사용하지 않으면, 파이썬은 import 해올 때, import 해오는 모듈을 모두 실행하기 때문에 원치 않은 코드가 실행될 수도 있다.

++ 파이썬은 왜 프로그램의 시작점이 정해져 있지 않나요..? 😟

: 파이썬이 처음 개발됐을 땐 리눅스, 유닉스에서 사용되는 스크립트 언어 기반이었기 때문에 프로그램의 시작점이 따로 정해져있지 않았다. 보통 리눅스/유닉스 스크립트 파일은 하나로 이루어져있기 때문에 파일 자체가 하나의 프로그램이다. -> 시작점이 따로 필요하지 않다!

하지만, C 언어나 Java 같은 애들은 처음부터 소스 파일을 여러 개 사용했기 때문에 시작 함수(main)를 따로 정해놔야함!

+ 자바도 모듈 있다! Java 9 부터 지원! 자바에서의 모듈이란? 왜 처음부터 안만들었음?

-> 모듈이란...

처음엔 필요성을 크게 느끼지 못했으나, 쓰다보니까 필요하니까 만들었음. 패키지로는 감당 안되는 애들이 발견됨.

첫째로, **패키지의 캡슐화(Encapsulation)가 완벽하지 않다.**

객체지향 프로그래밍에서 세부 구현을 숨기는 캡슐화는 필수적인 기법이다. 하지만 자바에서 클래스는 액세스 지정자로 멤버의 공개 여부를 마음대로 지정하는데 비해 패키지의 정보 은폐는 약하다. **클래스를 숨길 수는 있지만** 공개 또는 비공개 둘 중 하나만 선택할 수 있어 **여러 패키지에서 공용으로 사용할 클래스를 외부에 대해 숨길 수 없다**. 공용으로 사용할 클래스를 공개해 버리면 라이브러리 내부 뿐만 아니라 외부에서도 자유롭게 사용할 수 있다. 클래스는 패키지 내부로 숨길 수 있지만 패키지는 숨겨 두고 내부에서만 사용할 방법이 없다. 이럴 때는

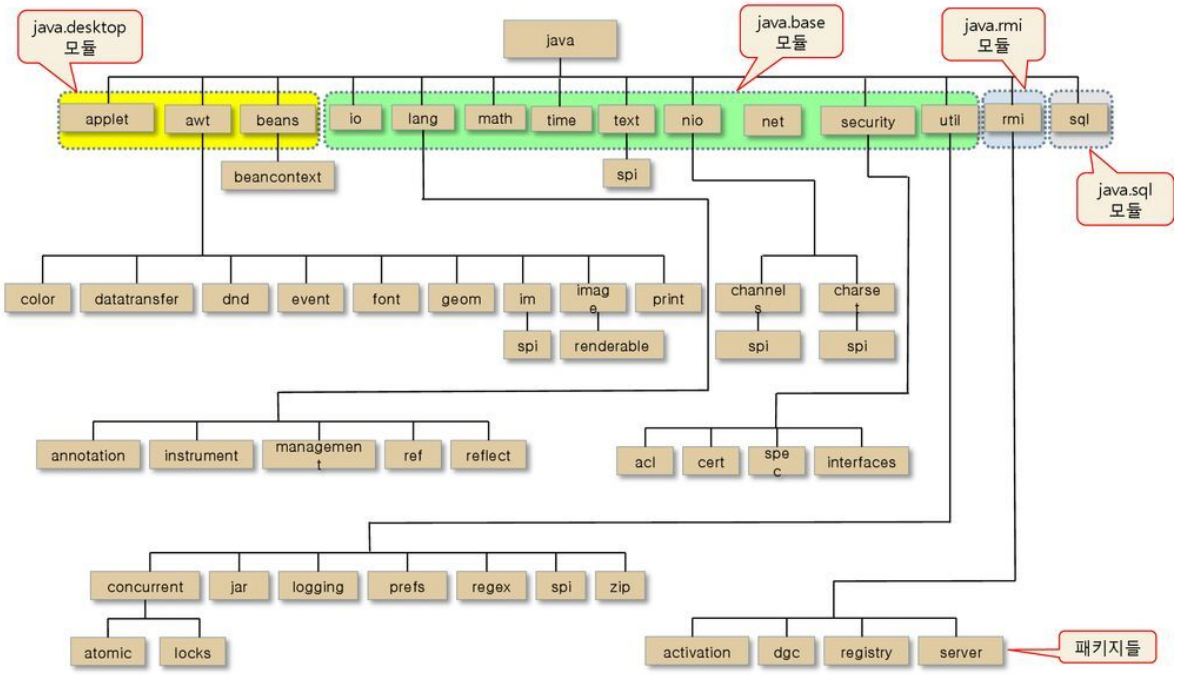
문서나 가이드를 통해 내부용 패키지는 사용하지 말라고 부탁 또는 협박(!)하는 수밖에 없는데 모든 개발자가 이런 권장 사항을 다 지키는 것은 아니다. 사용자가 내부 기능을 직접 사용하면 이후 자유롭게 유지, 보수할 수 없다. 그래서 **문법적으로 완전히 숨길 수 있는 강한 캡슐화가 필요**해졌다.

둘째로, **빌드 단계에서 프로그램 구동에 필요한 모든 클래스가 다 있는지 확인할 수 없다.** 자바는 동적 로딩을 통해 필요한 클래스를 실행중에 로드한다. 덕분에 시작이 빠르고 생성할 클래스를 선택할 수 있지만 실행 직후에는 누락 사실을 바로 알 수 없다는 문제가 있다. 클래스 누락을 방지하려면 실행중에 로드되는 클래스가 다 있는지 수작업으로 일일이 확인하는 수밖에 없다.

마지막으로, **런타임이 거대해져 배포가 어려워졌다.** 초기의 자바 플랫폼은 라이브러리의 모든 클래스를 rt.jar라는 단 하나의 파일에 통합하여 배포했다. rt는 런타임(Run Time)이라는 뜻이며 자바 프로그램이 실행되기 위한 모든 클래스가 포함되어 있는 자바 플랫폼 그 자체이다. 단일 파일이라 배포하기 쉽고 별 문제가 없었다. 그러나 자바 버전이 올라 가면서 rt.jar는 용량 60M에 2만개의 클래스를 포함하는 초대형 런타임이 되었고 앞으로는 더 늘어날 것이 뻔하다. PC에서는 별 무리가 없지만 임베디드 장비에 설치하기에는 너무 거대하다. 게다가 단일 파일이라 불필요한 것까지 한꺼번에 배포할 수밖에 없다. 꼭 **필요한 클래스만 추려 원하는 런타임을 생성**하고 **런타임과 프로그램을 같이 배포**할 수 있어야 할 필요가 생겼다.

이러한 문제들을 해결하기 위해 도입된 것이 **모듈화**다.

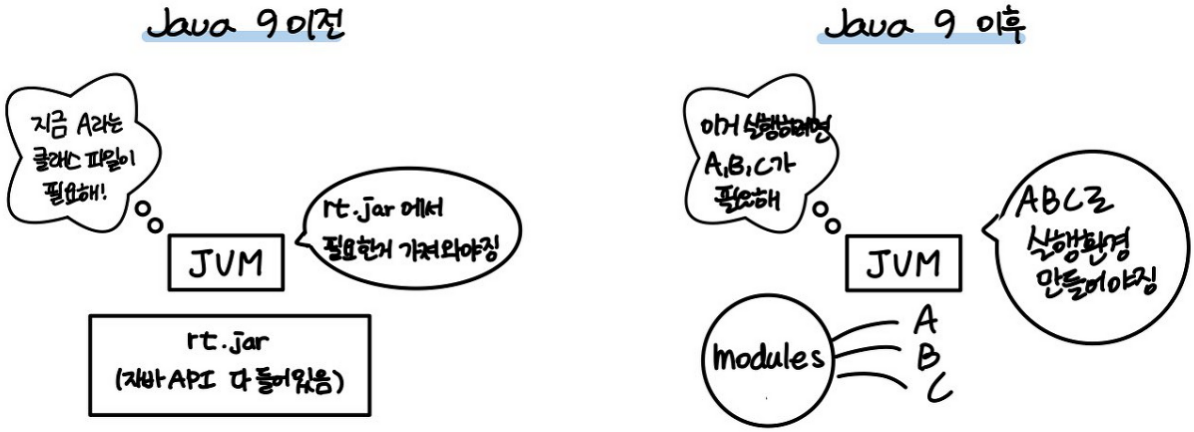
- Java 9부터 패키지들을 여러 개의 모듈로 **나눠서 제공**



자바 모듈과 패키지 구조 / 이미지 출처: https://hanbi97.tistory.com/231

~~구조 디렉토리 마다 하나씩 확인해보고 싶은데 일단 갈길이 머니까 다음ㅇ...~~

- jmods 디렉토리 안에 기본 모듈이 들어있고 모듈 파일(.jmod)에는 패키지들이 담겨있음.
- 실행시 **필요한 모듈만을 가져와** 실행 환경을 만들고 jlink 이용해 custom 실행 환경을 만들 수 있음.



이미지 출처: https://hanbi97.tistory.com/231

+ 패키지 이름 규칙 (with. 오라클)

1. 모든 패키지 이름은 클래스나 인터페이스와 혼동을 주지 않기 위해 소문자(lower case)로 지정
2. 일반적인 기업 : com.회사이름.패키지명 -> 해당 패키지는 회사.com의 개발자가 이 패키지를 만들었다~ (지역도 표기하고 싶은 경우 : com.회사이름.지역.패키지명)
3. 숫자로 시작하거나, '_' 과 '$'를 제외한 특수 문자 사용 금지
4. java 로 시작하는 패키지 금지(자바 표준 API에서만 사용)
5. int, static 등 자바 예약어 금지

+ 이대로 다 되면 좋겠지만... 안된다면 각 상황에 대한 대처 방식!

Legalizing Package Names

Domain Name	Package Name Prefix
hyphenated-name.example.org	org.example.hyphenated_name
example.int	int_.example
123name.example.com	com.example._123name

이미지 출처: [Java] package에 대해 알아봐요.

- 클래스패스

: Java 가 클래스를 사용하기 위해 탐색할 때, JVM 혹은 자바 컴파일러가 사용하는 파라미터로 클래스나 패키지를 찾을 때 기준이 되는 경로.

이미지 출처: 클래스패스(Classpath) :: 으뜸별

그러니까...

> JVM 이 프로그램 실행해야함
>
> ➔ Java Runtime 이 .class 파일에 포함된 명령을 실행해야함
> ➔ 이 .class 파일을 어디서 찾지???
> ➔ **classpath** 에 지정된 경로에서 찾을 수 있다~
> ➔ Java Runtime 이 이 **classpath** 에 지정된 경로 **모두 검색**함.
> ➔ 특정 클래스에 대한 코드가 포함된 .class 파일 찾음!
> ➔ 첫 번째로 찾은 파일 사용한다!

Classpath 를 지정하는 방법은 두 가지가 있다.

1. 환경 변수 CLASSPATH 사용
2. Java runtime 에 -classpath (-cp) 플래그 사용

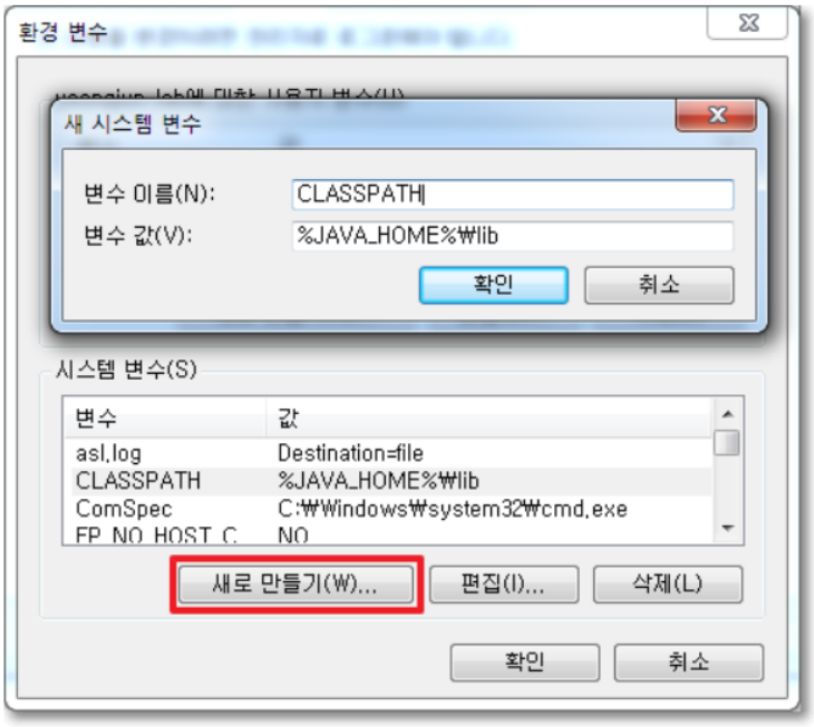
- **CLASSPATH 환경변수**

> **환경 변수**
> - 프로세스가 컴퓨터에서 동작하는 방식에 영향을 미치는 동적인 값들의 모임
> - 운영체제에서 자식 프로세스들을 생성할 때 참조하는 변수

라고 하는데.. 사실 이놈의 환경변수. 맨날 개발 환경 세팅할 때, @@@ : command not found 가 뜨면.. 아맞다 하고 확인해야하는 놈으로 좀 더 익숙하다.

좀 더 쉽게 설명한 글이 있어 링크를 가져와봤다! : https://m.blog.naver.com/zxwnstn/221521038431

즉, 환경 변수는 운영체제에서 자식 프로세스들을 생성할 때 참조하는 변수로 JVM 기반의 애플리케이션도 이 환경변수 값을 참고하여 동작한다!



이미지 출처: 클래스패스(Classpath) :: 으뜸별

대신 운영체제를 변경하면 클래스패스가 유지되지 않고 사라져서 **이식성 측면에서 좋지 않다**.

- **classpath 옵션**

다른 방법인 classpath 옵션이란

javac 명령을 통해 소스 파일을 컴파일 할 때, 컴파일러가 컴파일 하기 위해서 필요로 하는 참조할 클래스 파일들을 찾기 위해서 **컴파일 시 파일 경로를 지정해주는 옵션**이다.

```
javac -classpath <필요한 클래스 파일 위치> <소스 파일 위치>

// 만약 참조할 클래스 파일이 여러 개라면 세미콜론(;)으로 구분해줄 수 있다.
javac -classpath <필요한 클래스 파일 위치>;<필요한 클래스 파일 위치>; <소스 파일 위치>
```

+ **-classpath** 옵션은 java 와 javac 명령어에 모두 사용 가능.

  java : 런타임 때 .class 파일에 포함된 명령 실행하기 위해서 이 파일을 탐색할 경로를 알아야하기 때문에 사용 가능하다!

- **접근지시자**

~~*클래스랑 인터페이스 하면서 충분히 한 것 같은데..!*~~

- **참고 링크**
  - https://wikidocs.net/231
  - https://yeonyeon.tistory.com/191
  - https://www.w3schools.com/java/java_packages.asp
  - https://studymake.tistory.com/428
  - https://www.inflearn.com/questions/295097
  - https://scshim.tistory.com/221
  - https://yeonyeon.tistory.com/191
  - 모듈 vs 패키지 vs 라이브러리 vs 프레임워크
  - https://raspberrylounge.medium.com/%EC%9E%90%EB%B0%94%EC%97%90%EC%84%9C-%ED%8C%A8%ED%82%A4%EC%A7%80-package-%EC%99%80-%EB%AA%A8%EB%93%88-module-%EC%9D%98-%EC%B0%A8%EC%9D%B4%EC%A0%90-16b2eda177b4
  - https://aliencoder.tistory.com/20
  - https://blog-of-gon.tistory.com/223?category=864375
  - https://dodonam.tistory.com/107
  - https://hanbi97.tistory.com/231
  - https://raspberrylounge.medium.com/%EC%9E%90%EB%B0%94%EC%97%90%EC%84%9C-%ED%8C%A8%ED%82%A4%EC%A7%80-package-%EC%99%80-%EB%AA%A8%EB%93%88-module-%EC%9D%98-%EC%B0%A8%EC%9D%B4%EC%A0%90-16b2eda177b4
  - https://beststar-1.tistory.com/17
  - 7주차 과제 : 패키지
  - 자바 클래스패스(classpath)란?