6.7 모듈(Module)

모듈이란?

자바 9 부터 새롭게 지원되는 모듈(module)이란 관련 있는 패키지들과 해당 패키지들에서 사용하는 다양한 종류의 자원(resource)들을 묶어서 관리할 수 있는 좀 더 큰 개념의 패키지이다. 모듈에 대해서 자세히 설명하는 것은 이 책의 범위를 넘어가는 일이다. 여기서는 패키지가 있는데 더 큰 개념의 모듈이 필요한지 간략하게 알아보고, 모듈을 이용해서 간단한 프로그램을 만들어본다. 모듈에 대해서 좀 더 관심이 있다면 전문적으로 모듈에 대해서 설명하는 책을 살펴보기 바란다.

앞에서 패키지는 여러 클래스들을 폴더로 나누어서 관리하고 이름 충돌이 나지 않도록 만들었다고 했다. 패키지는 관련 있는 클래스들을 묶어서 관리할 수 있도록 했고 그 안에서 private, protected, public 등의 접근 제어성을 부여함으로써 패키지 내부에서만 사용하는 클래스와 외부에서 사용할 수 있는 클래스들을 구별시킬 수 있었다. 즉 패키지 내부에서 사용하는 클래스들은 protected 로 지정하면 외부에서는 볼 수 없게 되고, public 으로 지정된 클래스들은 다른 패키지에서도 사용할 수 있다. 패키지에서 이렇게 접근 제어자를 이용해서 가시성을 지정하는 것은 외부에서 사용해야만 하는 클래스들과 내부적으로만 사용해야 하는 구현 부분을 감추려는 캡슐화 때문이었다.

하지만 이런 패키지의 특성이 의도치 않게 추상화나 캡슐화를 해치는 경우가 있다. 예를 들어 자바에서 제공하는 클래스들은 대부분 java.util.ArrayList 나 java.awt.Point2D 처럼 java.packagename 형태로 제공되거나, 확장 패키지 개념으로 javax.packagename 형태로 제공되었다. 하지만 자바 8 이하에서는 sun.misc.BASE64Encoder 나 sun.misc.BASE64Decoder 같은 형태의 클래스들도 함께 제공되었다. 자바는 처음에 Sun Microsystems 에서 만들어졌으므로, 이 클래스들은 내부적으로 사용하기 위해서 만들어졌던 것으로 보인다 (공식적인 자바의 패키지라면 java.misc.BASE64Encoder 나 java.misc.BASE64Decoder 같은 형태로 지정되었을 텐데 그렇지 않았으니까). 그런데 public 으로 지정된 패키지의 클래스들은 모두에게 공개되어있다는 특성 때문에, 결국 Sun Microsystems 외부의 다른 프로그래머들이 해당 클래스들을 많이 사용했었다. 즉 내부적으로 사용하기 위해서 만들어졌던 패키지였지만, 패키지의 특성상 어쩔 수 없이 모든 자바 프로그래머에게 공개되었던 것이다. 다른 사람들이 지속적으로 사용하다 보니, 호환성을 중시하는 자바에서는 여태까지 이런 클래스들을 없애지 못하고 두고 있다가, 자바 9 버전부터는 java.util.Base64.Encoder 와 java.util.Base64.Decoder 클래스로 대체하고 해당 클래스들을 제거하였다.

여기서 패키지의 문제점과 모듈의 필요성이 드러난다. 앞에서 말한 것처럼 패키지 외부에서 클래스를 사용할 수 있도록 하려면 public 으로 클래스를 지정해야 한다. 문제는 public 으로 지정된 클래스들은 해당 패키지에 접근 가능한 모든 프로그래머들이 자유롭게 사용할 수 있게 되는 것이다. 즉 내부적으로 사용하기 위해서 만든 패키지였는데, 결국 외부 사용자에게도 고스란히 공개될 수 밖에 없게 되는 것이다. 이러한 문제를 해결하려면 함께 사용되는 클래스들을 모두 한 곳에 묶어 한 개의 큰 패키지를 만들어서 내부적으로 사용하는 클래스들을 해당 패키지 내에서만 사용하게 해야 할 수도 있다. 하지만 이렇게 한다면 관련된 클래스들을 묶기 위해서 제공했던 패키지의 의미가 퇴색될 수 밖에 없다.

이러한 문제를 해결하기 위해서 패키지들을 묶어서 한 개의 덩어리로 취급할 수 있도록 만들어진 것이 모듈이다. 모듈은 이렇게 내부적으로 사용하려고 만든 패키지들의 public 클래스들이 모두에게 공개되는 것들을 막을 수 있다. 즉 자바의 모듈은 여러 개의 패키지를 포함할 수 있으며, 그 중 특정 패키지만 모듈 밖에서 사용할 수 있도록 지정할 수 있다. 패키지의 클래스가 public 으로 지정되어

있다고 하더라고, 해당 패키지가 모듈 밖에 공개되지 않으면 외부 모듈에서는 사용할 수 없게 되는 것이다. 따라서 모듈은 패키지보다 더 큰 개념의 추상화 또는 캡슐화를 제공한다. 그렇다면 앞에서 보았던 예를 다시 생각해보자. 만약 자바에서 처음부터 모듈을 지원했다면, java.packagename 으로 되어 있는 패키지들은 모두 외부 모듈에서 사용할 수 있도록 하고, sun.packagename 에 패키지들은 내부적으로만 사용할 수 있도록 했다면, 앞에서 얘기했던 문제는 처음부터 발생하지 않을 수 있다.

모듈은 클래스나 패키지뿐만 아니라 다른 자원 또는 파일들을 포함할 수 있다고 했다. 패키지는 클래스를 묶어서 관리하는 목적으로만 만들어졌었다. 하지만 최근 프로그래밍 트렌드를 보면, 사용자 인터페이스와 코드를 분리해서 각각 다른 파일들로 저장해서 관리하기도 하고 또 프로그램에 따라서는 여러 가지 관련 자료들을 함께 포함해야 하기도 하는데, 기존 패키지에서는 이런 것들이 어려웠다. 따라서 새로운 프로그래밍 트렌드에 맞는 새로운 패키지 개념이 필요하게 되고, 그래서 만들어진 것이 모듈이다.

모듈은 자바 클래스 라이브러리가 너무 커지고만 있기 때문에 작게 모듈화시키기 위해서 제만들어지기도 했다. 자바의 클래스 라이브러리는 기존에 만들어진 자바 프로그램과의 호환성 문제 때문에 클래스를 줄이기는 어렵고 갈 수록 늘어만 가고 있는 중이다. 그리고 자바 8 까지 자바에서 제공하는 클래스들은 rt.jar 이라는 한 개 파일에 거의 들어 있었다. 앞으로 계속 해서 클래스들의 크기가 커진다면 rt.jar 파일의 크기 역시 계속해서 커질 수 밖에 없다. 이는 자바 프로그램에서 사용하는 클래스의 개수가 몇 개 안되더라도, 결국 대규모로 커진 rt.jar 에 대한 의존성을 제거할 수 없다는 얘기가 되고, 이는 다시 비효율적인 지속적으로 남게 된다. 그래서 자바 9 에서는 이러한 기본적인 자바 클래스 라이브러리를 몇 개의 모듈로 잘라서 필요한 부분만을 사용할 수 있도록 해주었다.

이렇게 다양한 목적을 가지고 모듈이라는 개념이 도입되었지만, 이 책에서는 패키지에 대한 접근성을 제한하는 목적의 모듈에 대해서만 간단하게 살펴보기로 한다. 아래는 기존 패키지에 비해 모듈을 사용함으로써 얻을 수 있는 장점에 대해서 나열한다.

- public 클래스를 숨길 수 있으므로 패키지에 비해 강한 캡슐화를 지원할 수 있다
- 모듈 단위로 좀 더 큰 개념의 추상화와 캡슐화를 지원함으로써 패키지 간의 관계를 모듈 단위로 정리할 수 있어 복잡한 의존성을 줄일 수 있다.
- 모듈은 컴파일된 자바 코드 외에 다른 리소스 파일들을 포함시킬 수 있다. 패키지는 자바 코드를 모아놓은 것들이었다. 하지만 모듈에서는 이미지 파일이나 XML 파일 같은 자료들을 모듈에 함께 포함시킬 수 있다.
- 자바에서는 관련된 클래스들을 묶고 패키지화 시키고, 다시 그 패키지들을 묶어서 모듈화 시킨다는 점에 주의하자. 일반적으로 응용 프로그램을 모듈에 넣는 것 보다는 자주 사용될 수 있는 라이브러리를 구축할 때 더 유용하다.

디폴트 모듈 (default modules)

모듈을 지원하기 시작한 자바 9 에서는 기존에 패키지 형태로만 제공되던 클래스들을 모듈화시켰고 모두 4 개의 큰 그룹으로 나누어서 지원한다. 컴파일된 자바 코드를 실행시킬 때 사용하는 java 프로그램을 사용하면 기본적으로 자바에서 제공하는 모듈들을 확인할 수 있다. 윈도우의 커맨드 창이나 맥 또는 리눅스의 터미널 창에서 아래처럼 실행시켜보자.

```
java --list-modules
```

화면에 자바에서 제공하는 모듈들이 아래 화면에서 보인 것처럼 출력된다.

화면에 나타난 모듈의 목록에서는 자바에서 제공하는 모든 모듈이 나타나지는 않지만, 기본적으로 자바에서 제공하는 모듈들은 아래 표에서 정리한 것처럼 크게 4 개 그룹으로 분류되어 있다.

모듈 그룹	설명	일부 모듈 이름
java	기존 자바 스탠다드 버전에서 제공했던 대부분의 패키지를 포함하는 모듈이다.	java.se, java.desktop, java.base, java.logging, java.xml 등
javafx	JavaFX 그래픽 사용자 인터페이스 관련 패키지들을 모아 놓은 것이다.	javafx.fx.base, javafx.controls, javafx.graphics, javafx,swing
jdk	java 나 javafx 에 속하지 않지만, JDK(Java Development Kit)에서 내부적으로 사용하는 패키지들이 여기에 속한다.	jdk.charsets, jdk.httpserver, jdk.internal.vm.compiler, jdk.xml.dom
Oracle	자바를 소유하고 있는 Oracle 관련해서 필요한 패키지들이다.	oracle.desktop, oracle.net

앞에서 만들었던 패키지를 모듈로 구성해본다. com.my.hello 패키지를 com.my.hello.module 에 넣고 패키지를 사용할 수 있도록 만들고 com.my.app 패키지를 com.my.app.module 모듈에 넣기로 한다.

모듈 생성하기

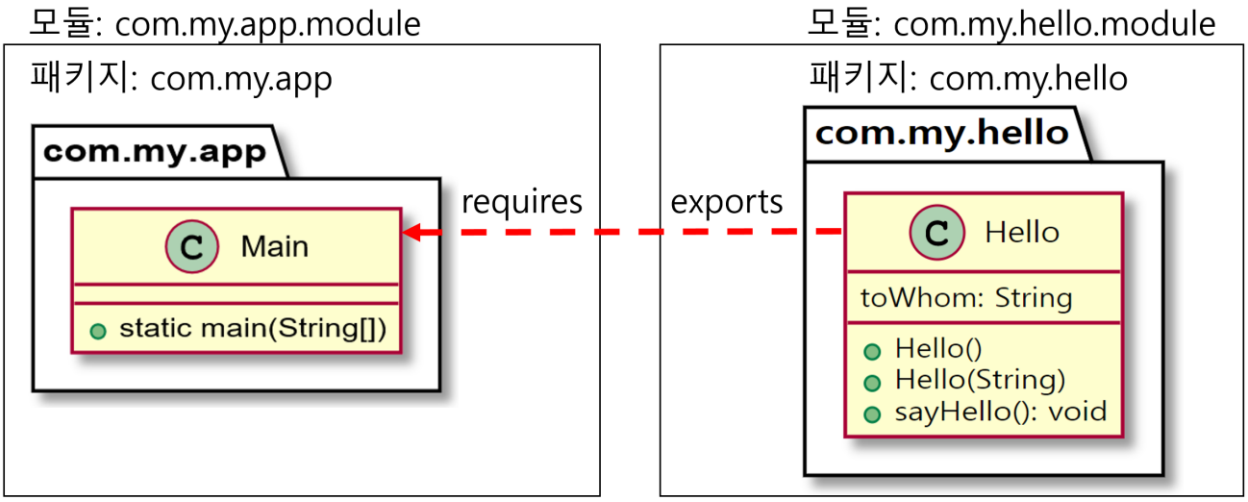
여기서 만들 모듈은 이미 앞에서 정해졌다. 아래에서 보인 두 개의 모듈을 구현할 것이다.

- com.my.hello.module
- com.my.app.module

com.my.hello.module 은 예전에 만들었던 "hello" 클래스를 모듈 형태로 구현한 것이다. 그리고 com.my.app.module 모듈은 com.my.hello.module 에서 제공하는 클래스의 객체를 생성하고 함수를 호출하여 인사를 하는 프로그램이다. 자바의 모듈은 한 개 이상의 패키지를 포함할 수 있는데,

기본적으로 모듈에 있는 패키지들은 해당 모듈 내부에서만 사용할 수 있도록 하여 강력한 캡슐화를 지원한다. 만약 패키지를 외부 모듈에서 사용할 수 있도록 하려면, 해당 패키지들은 명시적으로 엑스포트(exports) 되어야 한다. 반대로 다른 모듈에서 제공하는 패키지를 사용하는 모듈에서는 특정 모듈을 요구(requires)한다는 것을 명시적으로 지정하여 모듈간의 의존성을 정확하게 밝혀야 한다.

여기서는 아래 그림에서 보인 것처럼 com.my.hello.module 모듈에서는 Hello 클래스가 포함된 com.my.hello 패키지를 엑스포트(exports)한다. 그리고 com.my.app.module 모듈은 com.my.hello.module 모듈을 요구(requires)하고, 다시 그 안에 있는 com.my.app 패키지의 Main 클래스에서는 해당 클래스를 일반 패키지의 클래스를 사용하는 것처럼 임포트(import) 해서 사용한다.



모듈 생성 과정

프로그램에서 사용할 패키지의 클래스를 제공하는 com.my.hello.module 부터 모듈을 만들어보기로 한다. 모듈에 대해서 모든 내용을 설명하는 것은 이 책의 범위를 넘어간다. 여기서는 다른 곳에서 사용할 수 있는 패키지를 제공하는 모듈을 만들면서 필요한 최소한의 설명만을 할 것이다. 좀 더 자세한 내용은 자바 설명서를 참고한다.

아래는 다른 모듈에서 사용할 수 있는 패키지를 포함하는 모듈을 만드는 순서를 보인다.

- 모듈 이름 정하기
- 모듈 이름으로 폴더(디렉토리) 만들기
- 패키지 폴더 만들기
- 패키지에 들어갈 자바 클래스 코드 작성
- 모듈 서술자 파일(module descriptor file) 작성
- 모듈 컴파일
- 만약 필요하면 jar 파일 생성
- 모듈 사용

모듈 이름 정하기

패키지를 사용하는 이유 중에는 클래스의 이름이 충돌을 막기 위해서도 있다. 그래서 일반적으로 패키지의 이름은 본인 또는 프로그램을 작성하는 기업 홈페이지 URL 을 거꾸로 사용한다고 했다. 모듈도 비슷하게 이름을 정해야 한다. 다른 모듈 이름과 충돌하지 않을만한 이름으로 정해야 하며, 이

때문에 모듈도 패키지처럼 URL 을 거꾸로 사용하는 것을 권한다. 만약 개인적으로 프로그램을 만드는 것이라서 URL 이 없다면 최대한 겹치지 않을 이름을 지정하도록 한다. 그리고 모듈은 패키지처럼 소문자로 작성하는 것이 일반적이다. 여기서는 이미 앞에서 보였던 것처럼 com.my.hello.module 와 com.my.app.module 으로 정했다.

모듈 이름으로 폴더(디렉토리) 만들기

모듈 관련 코드들은 일반적으로 모듈 이름으로 만들어진 폴더에 넣는다. 서브 폴더가 계속 구성되는 패키지와는 달리 모듈 폴더는 단일 폴더로 만들어진다. 따라서 앞에서 보인 두 개의 모듈을 예로 생각하면 com.my.hello.module 과 com.my.app.module 폴더를 생성해야 한다. 아래는 윈도우에서 폴더를 생성하는 명령을 보인다.

```
mkdir com.my.hello.module
mkdir com.my.app.module
```

패키지 폴더 만들기

모듈과는 달리 패키지 폴더는 패키지 이름에 있는 마침표('.')를 기준으로 여러 단계의 서브 폴더들로 구성된다. 여기에서 구현될 모듈들은 com.my.hello 와 com.my.main 이라고 이름붙은 패키지를 각각 포함하고 있다. 아래 명령은 윈도우에서 각 모듈 폴더 안에 패키지 폴더를 만드는 것을 보인다.

```
mkdir com.my.hello.module\com\my\module\hello
mkdir com.my.app.module\com\my\main
```

아래는 폴더 구조를 보인다. 윈도우 명령창에서 tree 명령을 이용해서 디렉토리 구조를 출력하였다.

```
卅---src
    +---com.my.app.module
    |   卅---com
    |       卅---my
    |           卅---app
    卅---com.my.hello.module
        卅---com
            卅---my
                卅---hello
```

패키지에 들어갈 자바 클래스 코드 작성

이제 패키지에 들어갈 코드를 작성한다. 아래 코드를 입력해서 com.my.hello.module.java 에 저장한다.

```
// Hello.java
package com.my.hello;

public class Hello {
    String toWhom = "world";
    public Hello() {
    }
    public Hello(String toWhom) {
        this.toWhom = toWhom;
    }
```

```
    public void sayHello() {
        System.out.printf("hello %s!", toWhom);
    }
}
```

그리고 아래 코드는 com.my.app.module.java 에 저장한다.

```
// Main.java
package com.my.app;

import com.my.hello.Hello;

public class Main {
    public static void main(String[] args) {
        Hello hm = new Hello("ycho");
        hm.sayHello();
    }
}
```

모듈 서술자 파일(module descriptor file) 작성

코드를 작성했으면, 앞에서 얘기했던 것처럼 각 모듈에서 어떤 패키지를 외부에서 사용할 수 있도록 제공(exports)할 것인지 혹은 어떤 모듈을 사용(requires)하는지를 명시해야 한다. 이런 내용들을 명시하는 파일을 모듈 서술자 파일이라고 하며, 파일 이름은 module-info.java 라고 붙여야 하고 항상 모듈 이름으로 지정된 폴더에 존재해야 한다. 모듈 서술자 파일은 확장자가 java 라고 되어 있지만 내용은 자바 코드는 아니다. 서술자 파일용 특별 문법으로 작성되어야 하고 아래에서 보인 것처럼 module 이라는 키워드로 시작되며 모듈의 이름을 지정하고 블록 영역 안에 모듈 명령문을 넣는다.

```
module <모듈명> {
    <모듈 명령문>
    <모듈 명령문>
}
```

명령어	설명
module	모듈 선언
모듈명	모듈 이름을 지정, 모듈의 중복을 방지하기 작명할 필요, 여러 단어는 점으로 단어를 구분할 수 있음
모듈 명령문	exports, requires, opens, uses, provides 등이 있음. 여기서는 특정 패키지를 외부 모듈에서 사용할 수 있도록 제공하겠다고 지정하는 exports 와 외부 모듈에서 제공하는 패키지를 요구하는 requires 만 사용함

아래는 각각 com.my.hello.module 과 com.my.app.module 의 module-info.java 코드를 보인다. com.my.hello.module 은 com.my.hello 패키지를 외부에서 사용할 수 있도록 제공하는 모듈이다. 따라서 "exports 패키지명" 형태로 지정한다.

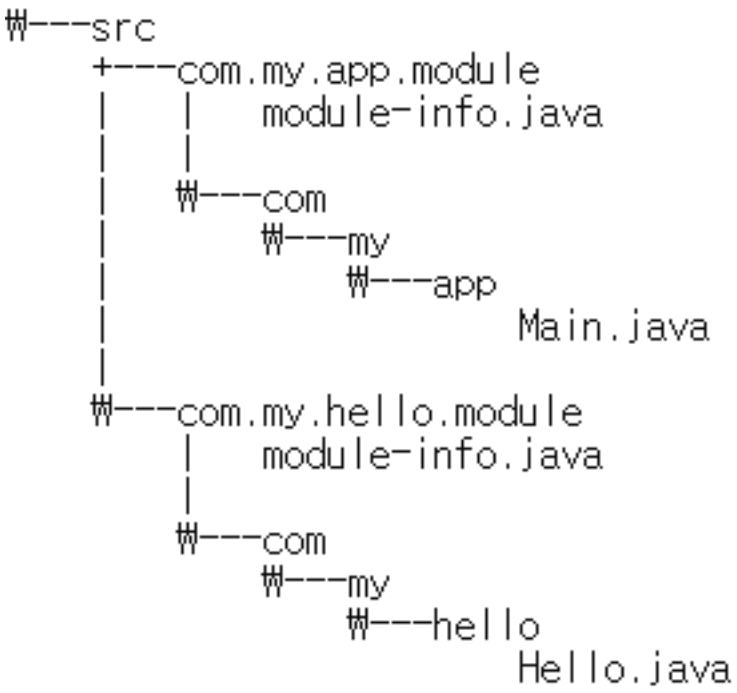
아래 코드는 com.my.hello.module 폴더에 저장한다.

```
module com.my.hello.module {
    exports com.my.hello;
}
```

아래 module-info.java 파일은 com.my.app.module 디렉토리에 저장한다. com.my.app.module 은 com.my.hello.module 에서 제공하는 com.my.hello 패키지를 사용해야 한다. 따라서 해당 모듈이 필요하다고 "requires 모듈명"으로 지정한다.

```
module com.my.app.module {
    requires com.my.hello.module;
}
```

아래 그림은 자바 소스 파일과 module-info.java 를 생성한 후의 폴더 구조를 보인다. 파일들의 위치를 자세히 확인해야 한다.



모듈 컴파일과 실행

아래에서 보인 것처럼 입력해서 모듈을 컴파일 해본다. 모듈을 컴파일하는 것은 일반 자바 코드를 컴파일하는 것과 좀 다른 부분들이 있다. 아래 표에서는 컴파일할 때 사용해야 하는 옵션 몇 가지를 설명한다.

옵션 | 설명 |
-module-path 디렉토리 | 컴파일된 모듈이 들어 있는 디렉토리를 지정함. 자바의 -classpath 옵션을 대체할 수 있음. -classpath 는 모듈에 대해서 모르기 때문에 모듈을 사용하지 않을 때 계속 사용할 수 있지만, -module-path 는 -classpath 의 역할을 하면서 모듈에 대해서 이해하고 처리할 수 있는 옵션임 |
-module-source-path 디렉토리 | 여러 개 모듈의 소스 코드가 들어있는 디렉토리를 지정함이 옵션이 사용될 때 지정된 디렉토리에는 모듈 이름으로 된 서브 디렉토리가 존재하고 각 모듈의 소스 코드가 기존 패키지 형태(서브 디렉토리)로 있어야 함자바 컴파일러는 모듈을 컴파일하면서 소스 코드가 들어있는 디렉토리 구조를 -d 옵션으로 지정한 결과물 디렉토리에 반영함. 즉 모듈명/패키지폴더 형태가 그대로 반영됨 |

앞에서 만들었던 모듈 코드를 컴파일하면서 여기서 설명한 옵션들이 어떻게 사용되는지 확인해보자. 먼저 패키지를 제공하는 모듈부터 아래에서 보인 것처럼 컴파일한다.

```
C:\code\ModuleTest>javac -d modules --module-path modules --module-source-
path src src\com.my.hello.module\com\my\hello\Hello.java
src\com.my.hello.module\module-info.java
```

다음은 여기서 사용된 옵션에 대한 설명이다. | 옵션 | 설명 | | -d modules | 컴파일된 코드(출력물)를 넣을 디렉토리를 지정함. modules 폴더에 컴파일된 코드를 넣도록 함 | | –module-path modules | 지금은 패키지를 제공하는 모듈만 컴파일하므로 이 옵션은 없어도 됨. 하지만 다른 모듈을 사용하는 경우와 동일하게 처리하기 위해 넣음 | | –module-source-path src | src 폴더에 있는 모듈 및 패키지 폴더 구조를 출력물 디렉토리에 반영함. src 폴더에 있는 com.my.hello.module 과 그 아래 서브 디렉토리 구조로 되어 있는 패키지 디렉토리를 modules 아래 똑같이 만들고 컴파일된 class 파일들을 저장함 | | src.my.hello.module.java src.my.hello.module-info.java | 모듈에서 사용되는 자바 소스 코드를 지정함 |

이번에는 앞에서 만든 모듈을 사용하는 Main.java 를 아래에서 보인 것처럼 컴파일 한다.

```
C:\code\ModuleTest>javac --module-path modules -d modules --module-source-
path src src\com.my.app.module\com\my\app\Main.java
src\com.my.app.module\module-info.java
```

컴파일된 모듈을 실행시키는 것도 기존 방법과는 다르다. 아래는 모듈 실행 방법을 보이고 옵션에 대해서 설명한다.

```
java --module-path "diretory" --module "ModuleName/ClassName"
```

옵션 | 설명 |
–module-path “directory” | 모듈이 존재하는 디렉토리 이름을 지정함 |
–module “module/classname” | main() 함수를 포함하는 클래스가 있는 모듈과 클래스 이름을 명시함 |

앞에서 만들었던 com.my.app.module 모듈에 있는 com.my.app 패키지의 Main 클래스를 실행시키는 코드는 아래와 같다. 컴파일된 모듈의 위치를 지정하기 위해 –module-path 옵션을 사용하였고 모듈을 실행시키기 위해 –module 옵션을 사용하였다.

```
C:\code\ModuleTest>java --module-path modules --module
com.my.app.module/com.my.app.Main
```

다음은 실행 결과를 보인다.

```
C:\code\ModuelTest>java --module-path modules --module
com.my.app.module/com.my.app.Main
hello ycho!
```