25. 빌드 및 배포

`javac`, `java`, `jar` 명령어

Java 개발에서 사용하는 대표적인 명령어인 `javac`, `java`, `jar`는 각각 **컴파일**, **실행**, **아카이브(패키징)** 역할을 수행한다. 이들은 JDK 설치 시 함께 제공되며, Java 애플리케이션 개발의 핵심적인 빌드 및 실행 도구다.

📌 1. `javac` — Java 컴파일러

✅ 개요

`javac`는 `.java` 파일을 컴파일하여 `.class` 바이트코드 파일을 생성한다. 이 `.class` 파일은 JVM이 실행할 수 있다.

✅ 기본 사용법

```
javac HelloWorld.java
```

→ `HelloWorld.class` 생성

✅ 주요 옵션

옵션	설명
`-d <디렉터리>`	`.class` 파일의 출력 경로 지정
`-cp` 또는 `-classpath`	참조할 외부 클래스 경로 지정
`-source` / `-target`	컴파일 소스/타겟 버전 지정
`-encoding UTF-8`	소스 파일 인코딩 지정

✅ 예시

```
javac -d out -encoding UTF-8 -cp "libs/*" src/com/example/Main.java
```

📌 2. `java` — Java 애플리케이션 실행기

✅ 개요

`java` 명령어는 컴파일된 `.class` 파일을 JVM 상에서 실행한다.

✅ 기본 사용법

```
java HelloWorld
```

→ `HelloWorld.class` 파일 실행

> **주의**: 확장자 `.class`를 쓰지 않는다.

✅ 주요 옵션

옵션	설명
`-cp` 또는 `-classpath`	실행 시 참조할 클래스 경로 지정
`-jar`	JAR 파일을 직접 실행
`-Xmx`, `-Xms`	JVM 메모리 설정
`-D<name>=<value>`	시스템 속성 설정

✅ 예시

```
java -cp out com.example.Main
java -Xmx512m -Denv=prod MainApp
```

📌 3. `jar` — Java Archive 도구

✅ 개요

`jar` 명령어는 여러 `.class` 파일과 리소스를 하나의 `.jar` 아카이브 파일로 묶는 도구다. 실행 가능한 JAR 파일을 만들 수도 있다.

✅ 기본 사용법

```
jar cf app.jar *.class
```

→ `app.jar` 생성

✅ 주요 옵션

옵션	설명
`c`	새 JAR 생성
`f`	파일 이름 지정
`v`	자세한 출력 (verbose)
`x`	JAR 파일 추출
`t`	JAR 파일 내용 보기
`m`	`MANIFEST.MF` 파일 지정
`e`	메인 클래스 지정 (`--create`와 함께 사용)

✅ 실행 가능한 JAR 생성

```
jar cfe app.jar com.example.Main -C out/ .
```

→ 실행 가능한 JAR 파일 생성

```
java -jar app.jar
```

📦 실행형 JAR 예제 전체 흐름

```
# 1. 컴파일
javac -d out src/com/example/Main.java

# 2. JAR로 묶기
jar cfe app.jar com.example.Main -C out .

# 3. 실행
java -jar app.jar
```

🧠 정리

명령어	역할	예시
`javac`	`.java` → `.class`	`javac Hello.java`
`java`	`.class` 실행	`java Hello`
`jar`	`.class` 묶기	`jar cfe app.jar Hello Hello.class`

Maven / Gradle 빌드 도구

Java 개발에서 **Maven**과 **Gradle**은 대표적인 **빌드 도구(build tool)**로, 단순한 컴파일/실행을 넘어서 **의존성 관리, 프로젝트 빌드, 테스트, 배포 자동화**까지 도와주는 핵심 도구다. 이 둘은 기능상 유사하지만 구조와 철학이 조금 다르다.

1️⃣ Maven 개요

🧩 특징

- XML 기반 설정 (`pom.xml`)
- Convention-over-Configuration (표준 디렉토리 구조)
- 빌드 생명주기(Lifecycle) 중심 설계

📁 기본 디렉토리 구조

```
project/
├── pom.xml
└── src/
    ├── main/
    │   └── java/
    │       └── com/example/App.java
    └── test/
        └── java/
            └── com/example/AppTest.java
```

📦 pom.xml 예시

```xml
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>myapp</artifactId>
  <version>1.0.0</version>

  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-context</artifactId>
      <version>5.3.20</version>
    </dependency>
  </dependencies>
</project>
```

🛠️ 주요 명령어

```
mvn compile          # 컴파일
mvn test             # 단위 테스트
mvn package          # JAR/WAR 빌드
mvn install          # 로컬 저장소에 설치
mvn clean            # target 디렉토리 제거
```

2️⃣ Gradle 개요

🧩 특징

- Groovy/Kotlin DSL 기반 설정 (`build.gradle`, `build.gradle.kts`)
- 선언적 + 프로그래밍 가능한 구조
- 속도 최적화(인크리멘털 빌드, 캐시 등)

📁 기본 디렉토리 구조

```
project/
├── build.gradle
└── src/
    ├── main/java/
    └── test/java/
```

📜 build.gradle 예시 (Groovy DSL)

```
plugins {
    id 'java'
}

group = 'com.example'
version = '1.0.0'

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework:spring-context:5.3.20'
    testImplementation 'junit:junit:4.13.2'
}
```

🛠️ 주요 명령어

```
gradle build         # 전체 빌드
gradle clean         # 빌드 파일 정리
gradle test          # 테스트 실행
gradle run           # 애플리케이션 실행 (애플리케이션 플러그인 필요)
```

> `./gradlew` → Gradle Wrapper: Gradle이 설치되어 있지 않아도 실행 가능하게 하는 스크립트

🔍 Maven vs Gradle 비교

항목	Maven	Gradle
설정 문법	XML (`pom.xml`)	Groovy/Kotlin DSL (`build.gradle`)
성능	일반적	빠름 (빌드 캐시, 병렬 처리 등)
유연성	낮음 (구조 고정)	높음 (스크립트 수준 제어 가능)
학습 곡선	단순	초반에 복잡하게 느껴질 수 있음
의존성 선언	`<dependency>` 태그	`implementation`, `testImplementation` 등
대중성	오래된 Java 프로젝트에서 널리 사용	최근 프로젝트에서 증가하는 추세

💡 어떤 걸 써야 할까?

- **Maven**
  - 표준화된 워크플로우를 원할 때
  - 대규모 팀 협업에서 일관성 있는 설정을 중시할 때
- **Gradle**
  - 빠른 빌드 속도와 높은 유연성을 원할 때
  - Kotlin DSL 및 스크립트 활용에 익숙할 때
  - Spring Boot 등 최신 프레임워크 기반일 때 추천

📦 실전: JAR 파일 만들기

Maven

```
mvn clean package
```

→ `target/myapp-1.0.0.jar`

Gradle

```
gradle build
```

→ `build/libs/myapp-1.0.0.jar`

JAR 생성 및 실행

Java에서 `.jar` 파일(JAVA ARCHIVE)은 클래스 파일, 리소스, 메타데이터를 하나로 묶은 압축 파일로, **배포 가능한 실행 패키지**로 자주 사용된다.

✅ 1. 기본 JAR 생성 (명령어 기반)

📁 예시 디렉토리

```
MyApp/
├── Main.java
└── Manifest.txt
```

🔹 Main.java

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, JAR!");
    }
}
```

🔹 컴파일

```
javac Main.java
```

→ `Main.class` 생성

🔹 Manifest 파일

```
Main-Class: Main
```

주의: 마지막 줄에 반드시 **개행문자(빈 줄)** 있어야 함!

🔹 JAR 생성

```
jar cfm MyApp.jar Manifest.txt Main.class
```

- `c`: create
- `f`: output file 지정
- `m`: 사용자 정의 `Manifest.txt` 사용

🔹 실행

```
java -jar MyApp.jar
```

✅ 2. Maven으로 JAR 생성

`pom.xml` 설정 (실행 메인 클래스 지정 포함)

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <version>3.2.0</version>
      <configuration>
        <archive>
          <manifest>
            <mainClass>com.example.Main</mainClass>
          </manifest>
        </archive>
      </configuration>
    </plugin>
  </plugins>
</build>
```

빌드 및 실행

```
mvn clean package
java -jar target/your-artifact-id-version.jar
```

✅ 3. Gradle로 JAR 생성

`build.gradle` 예시

```
plugins {
    id 'java'
    id 'application'
}

mainClassName = 'com.example.Main'
```

실행 JAR 만들기

```
gradle clean build
```

실행:

```
java -jar build/libs/your-app.jar
```

🛠️ JAR 안 구조 보기

```
jar tf MyApp.jar
```

📌 Fat JAR (의존성 포함 실행 JAR)

Maven 또는 Gradle로 라이브러리까지 포함한 단일 실행 파일 만들기:

Maven

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-shade-plugin</artifactId>
  <version>3.2.4</version>
  <executions>
    <execution>
      <phase>package</phase>
      <goals><goal>shade</goal></goals>
      <configuration>
        <transformers>
          <transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
```

```
            <mainClass>com.example.Main</mainClass>
          </transformer>
        </transformers>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Gradle (Shadow Plugin)

```
plugins {
  id 'com.github.johnrengelman.shadow' version '7.1.2'
}

shadowJar {
  manifest {
    attributes 'Main-Class': 'com.example.Main'
  }
}
```

```
gradle shadowJar
```

→ `build/libs/yourapp-all.jar`

🚀 JAR 실행 요약

구분	실행 명령어
일반 클래스 파일	`java Main`
JAR 실행	`java -jar MyApp.jar`
클래스 경로 지정	`java -cp myapp.jar com.example.Main`

모듈 기반 배포 (JLink, JPackage)

🧱 1. Java 모듈 시스템 기초 개념 정리

Java 9+부터는 `module-info.java`를 기반으로 **명시적 의존성**과 **모듈화**가 가능해졌다.
예시:

```
module com.example.app {
    requires java.base;
    requires java.sql;
    exports com.example.app;
}
```

🛠️ 2. JLink — 모듈 기반 JRE 이미지 생성기

`jlink`는 **자바 런타임 이미지(JRE)**를 내 애플리케이션에 **최소화된 형태로 포함**시킬 수 있게 해준다.
즉, **필요한 모듈만 포함한 JRE 생성**이 가능해서 전체 크기가 작아지고 성능도 최적화된다.

📦 사용 전 요구 조건

- 반드시 `module-info.java`가 있어야 함 (자바 모듈 시스템 기반)
- `javac`로 모듈 단위로 컴파일해야 함
- JDK 9 이상 필요

📁 프로젝트 구조 예시

```
myapp/
├── src/
|   └── com.example.app/
|       ├── module-info.java
|       └── Main.java
```

① 컴파일 (모듈로)

```
javac -d out \
      --module-source-path src \
      $(find src -name "*.java")
```

② JLink 이미지 생성

```
jlink \
  --module-path $JAVA_HOME/jmods:out \
  --add-modules com.example.app \
  --output myapp-runtime \
  --launcher run=com.example.app/com.example.app.Main
```

- `--add-modules`: 포함할 모듈 명시
- `--output`: 결과 디렉토리 (실행 환경)
- `--launcher`: 실행 진입점 지정

🏁 실행

```
./myapp-runtime/bin/run
```

✅ 생성된 디렉토리는 독립적인 실행 환경이므로, Java 설치 없이 실행 가능!

📦 3. JPackage — 운영체제 실행 파일 또는 설치 패키지 생성

`jpackage`는 JDK 14+에서 제공되고, Windows/Mac/Linux별 **실행파일(EXE, DMG, DEB, RPM 등)** 을 만드는 데 사용한다.

요구 사항

- `jlink`와 마찬가지로 모듈 기반이어야 함
- Java 14 이상

기본 사용법

```
jpackage \
  --input out \
  --name MyApp \
  --main-jar myapp.jar \
  --main-class com.example.Main \
  --type exe \
  --icon icon.ico \
  --dest dist
```

주요 옵션

옵션	설명
`--input`	클래스나 JAR 파일이 위치한 디렉토리
`--name`	애플리케이션 이름
`--main-jar`	실행할 JAR 파일
`--main-class`	진입점 클래스 (`module.name/ClassName`)
`--type`	`exe`, `msi`, `dmg`, `pkg`, `deb`, `rpm` 등
`--icon`	실행 아이콘 (선택사항)
`--dest`	생성물 저장 위치

> `--module`과 `--module-path`를 사용해 모듈 기반으로도 패키징 가능

✅ JLink vs JPackage 비교

항목	JLink	JPackage
목적	경량화된 JRE 이미지	배포 가능한 설치 파일 생성
출력물	독립적인 실행 디렉토리	실행파일 (exe, dmg 등)
플랫폼별 구분	없음 (JDK로 실행됨)	있음 (OS별 실행 파일 생성)

항목	JLink	JPackage
모듈 필수 여부	필수	기본적으로 필수
Java 버전	9 이상	14 이상

⬅ END 정리

도구	용도
`jlink`	필요한 모듈만 포함한 **JRE 이미지 생성**
`jpackage`	설치형 **실행 프로그램 또는 설치파일 생성**