I) Code organisation, compiling, packaging, execution and tools last-update 05.2025

Code ---

The code be organized in classes / Object types, the *.java files.

Packages ---

Object types are organized in packages. A package is a logical collection of types: some of them are visible outside the package, and some of them are not, depending on their scope.

Package names must be unique, and their name should follow a certain template. On your computer, a package is a hierarchy of directories. Each directory contains other directories and/or Java files.

This organization is important, because any Java object type can be identified uniquely using the package name and its own name.

(*)

The contents of one package can span across multiple subprojects, meaning that

if you have more than one subproject in your project, you can have the same package name in more than one subproject, containing different classes. A symbolic representation of this is depicted in Figure 3-4.

// .java .class .jar relation ---

The files with .java extension containing type definitions are compiled into files with .class extension that are organized according to the same package structure and (then) packaged into one or more(*) JARs (Java Archives).

A library ---

is a collection of jars containing classes used to implement a certain functionality.

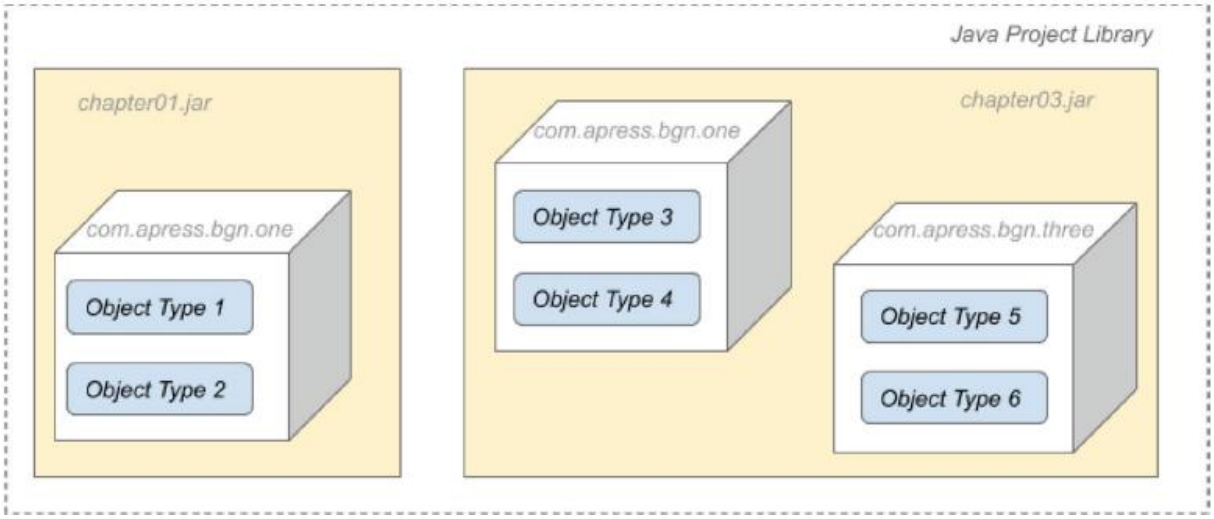

Figure 3-4. Example of package contents that span across multiple JARs (82)

classpath ---

A moderately complex Java application references one or more libraries.

To run the application, all its dependencies (all the JARs) must be on the classpath. It means that in order to run a Java application, a JDK, its dependencies (external jars), and the application jars are needed. Figure 3-5 depicts this quite clearly.

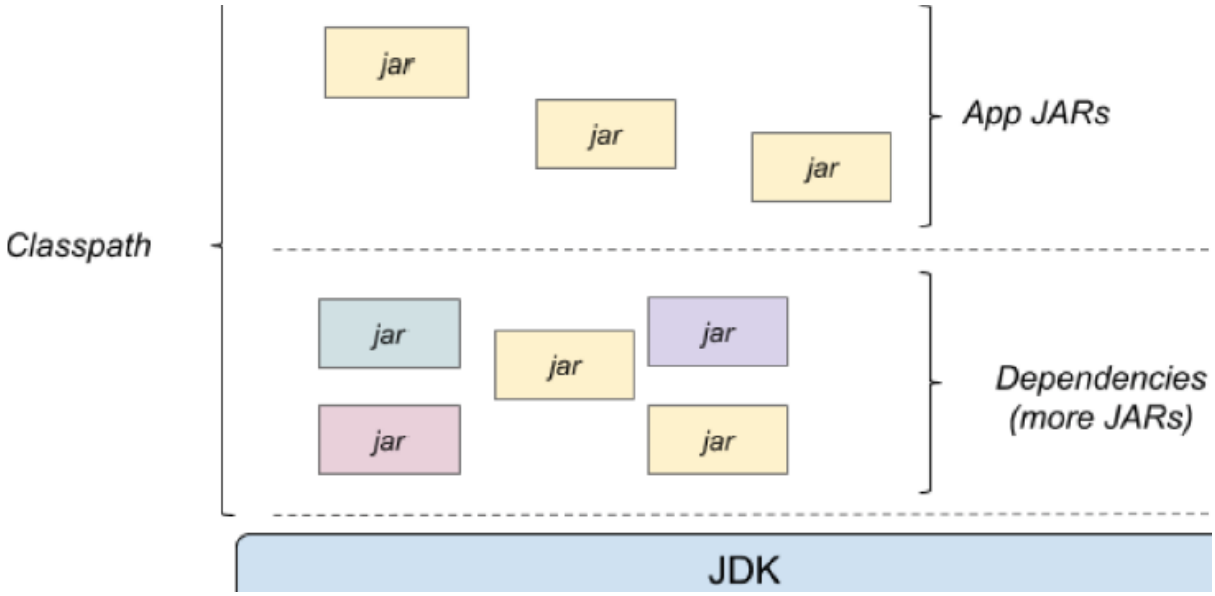

Figure 3-5. Application classpath

problem "JAR Hell" : The JARs that make up an application classpath are (obviously) not always independent of each other. This problem was resolved in Java 9 by introducing another level to organize packages: modules.

Module ---

A Java module is a way to group packages and configure more-granulated access to package contents. A Java module is a uniquely named, reusable group of packages and resources (e.g., XML files and other types of non-Java files) described by a file named module-info.java , located at the root of the source directory.

This file contains the following information:

• The module's name • The module's dependencies (that is, other modules this module depends on) • The packages the module explicitly makes available to other modules (all other packages in the module are implicitly unavailable to other modules) • The services the module offers • The services the module consumes • To what other modules it allows reflection • Native code • Resources • Configuration data

You can see the package declaration, but where is the module? Well, the module is an abstract concept, described by the *module-info.java* file. If you are configuring Java modules in your application, Figure 3-4 evolves into Figure 3-12.

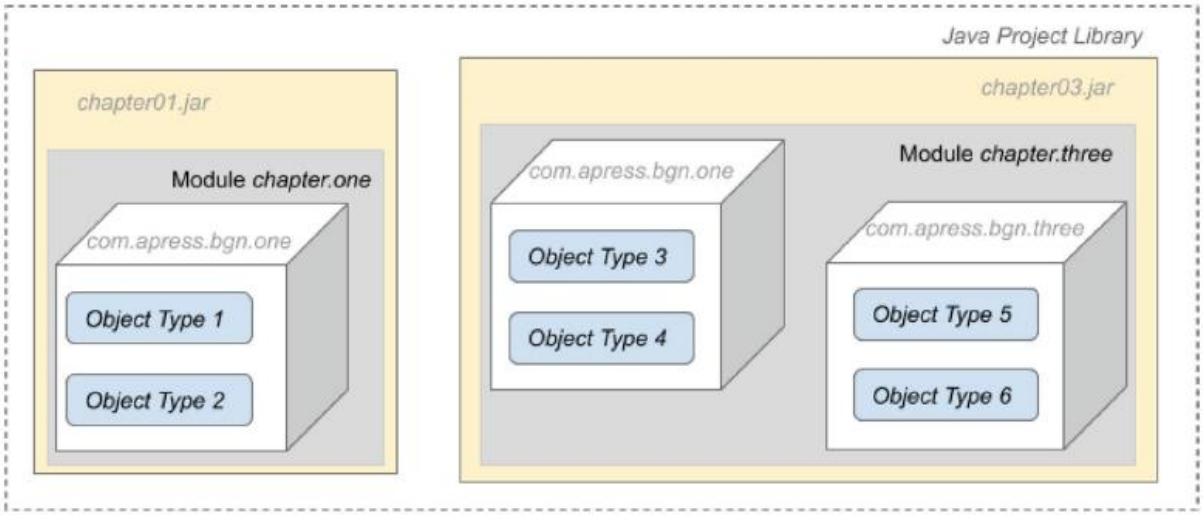

Figure 3-12. Java modules represented visually

Access Modifiers ---

access modifiers can be used to specify access to classes, and in this case we say that they are used at top-level.

At top-level only two access modifiers can be used: public and none.(84)

They also can be used to specify access to class members, and in this case they are used at member-level

At member-level two more modifiers can be applied, aside from the two previously mentioned: private and protected. (87)

II) Using commandos

Java-PROJECT ---

Multimodule-(Java-)Project ---

compiling code -> *.class

building/packaging => artifact *.jar

run Programm

III) project using MAVEN (tool for ...)

Maven-(Java-)PROJECT ---

Maven-Multimodule-(Java-)Project ---