JAR files

A **Java Archive (JAR)** is a platform-independent file format to pack multiple files together and distribute them as a single unit.
So it comes in handy if your application contains lots of files.

These are the main benefits of a JAR file:

- it can aggregate multiple files of different types;
- it is a compressed archive (with a **ZIP** algorithm) that reduces the size of the application and makes it easier to move it over a network;
- you can digitally sign it (this feature won't be discussed in this topic).

A JRE can start an application packed into a JAR, but to create a JAR you need to use a JDK.

The structure of a JAR file

A JAR file is simply an aggregation of bytecode files (.class), configuration files (e.g., .json, .xml), images, and even sound clips into a single compressed file. All files except bytecode files are usually called **resources**.

It is also recommended that a JAR file contains a special file named MANIFEST.MF in a special folder named META-INF. This file should describe the JAR file itself (a manifest is a kind of metadata): its version, the author, and so on.

Here is the example of a structure of a JAR file:

```
example.jar
 ├─── META-INF
|    └─── MANIFEST.MF
 ├─── second
|    ├─── Main.class
|    └─── MyIcon.png
 └─── third
      └─── another
           └─── OneMore.class
```

Usually, a JAR file has a set of .class files grouped by **packages**.

For now, just imagine a package as a directory or some nested directories. In our example, there are two packages: second and third/another.

Also, to represent a specific package name **dots** are used **instead of slashes** (so third.another notation would be correct in this case). Finally, there are some rules for naming the packages. For instance, a name can contain

letters and digits, but no hyphens (-).

As you can see, a JAR may have several packages and many .class files and/or resources in these packages.

The manifest file has a set of headers. The name and the value are separated by a colon (:). Take a look at a small example below:
Manifest-Version: 1.0
Created-By: 9.0.1 (Oracle Corporation)
Main-Class: second.Main
———————————— last line should end with a new line or a carriage return

The key feature here is the optional header Main-Class that defines the relative path of a class with the main method to start the application. The value shouldn't have the .class extension appended to the class name.

It's important to remember that the **last line** of the manifest file should end with a **new line or a carriage return**, or it will not be parsed properly.

Running a JAR file

There are two ways to run a JAR file, depending on whether you want to use the Main-Class header in the manifest file or not.

- if this header is not present or you want to specify the main class manually, run:

```
java -cp app-without-main-class-header.jar path.to.Main
```

The last parameter here is the full class name (with packages). The -cp option means **classpath**, i.e., paths to all the JARs which the JRE should scan for bytecode and resources. If you want, you can repeat multiple -cp path-to-Nth.jar pairs of parameters to provide the JRE with multiple different JAR files.

- if this header is present, then run:

```
java -jar app-with-main-class-header.jar
```

You can try executing both commands locally with these two JAR files (unpack top-level archives):

- a demo application without the Main-Class header (the path to main is myapp.Main);
- a demo application with the Main-Class header.

Both of them print the same line:
Hello, Java
Also, you can find some JAR files on the Internet and try running those. At the same time, you can view their internal structure – just replace .jar with .zip and

open these files as archives.

Conclusion

Now you know what a JAR is, what JAR files consist of, and how to run them on your computer. We have not discussed here how to create a JAR.

Usually, developers use build tools (like Maven or Gradle) or an IDE (like IntelliJ IDEA or Eclipse), which we cover in other topics.

You can also create them just by merging some files into a .zip archive and renaming it into a .jar file.
If you'd like to know more about the JAR file structure, read the specification.

Package Your Application: Open a terminal or command prompt, navigate to the directory containing your compiled .class files and the MANIFEST.MF file. Then run the following command to package your application into a JAR file:
bash

```
jar cfm YourJarFileName.jar MANIFEST.MF com/example/*.class
```

Replace YourJarFileName.jar with the desired name for your JAR file and com/example/*.class with the path to your compiled .class files. If your classes are in a different directory structure, adjust the path accordingly.
This command creates a JAR file named YourJarFileName.jar with the specified manifest file and includes all .class files in the specified package (com/example in this example).