Java Modules - Introduction by example

Why do we need Java Modules?

- Simplified: Dependency configuration via CLASSPATH is unreliable, if we are not cautious we can end up in JAR hell
  because we have no means to declare the applications dependencies explicitly.
  Current options to modularize applications
  provide no means to enforce loose coupling since there is no way to enforce dependencies.
- Modules are all about being able to specify an applications dependencies explicitly in order to prevent problems at
  runtime

Which options to modularize applications did we have before Java Modules?

- Inferfaces
- Packages
- JAR files

What is the issue with the existing options?

- Neither interfaces nor packages can be used to enforce loose coupling
  - Packages provide self-contained namespaces but they do not provide the means to create a hierarchy on language
    level since they do not provide the means to define which types they export to other packages on a fine grained
    level.
  - A type defined as public is globally public and thus can be accessed by any other package
  - JAR = zip files of packages and some meta information
    - provide no means to solve problem of packages
    - Do not allow to specify dependencies to other JAR files explicitly

What is the issue with the CLASSPATH?

The problem with the CLASSPATH is JAR Hell, which can manifest in different ways:

- Unexpressed dependencies
  - A JAR cannot express which other JARs it depends on in a way that the JVM will understand
  - This has to be managed by an external entity (developer, build tool, etc.)
- Transitive dependencies
  - Same as unexpressed dependencies but compounded for each JAR needed by the application
- Shadowing
  - Since classes will be loaded from the first JAR on the classpath to contain them, that variant will “shadow” all
    others and make them unavailable

What are the characteristics of Java Modules?

- Each module has a unique name
- Each module can contain packages, classes and interfaces
- Each module defines its dependencies to other modules explicitly
    - It defines which other modules it depends on
    - It defines which packages are exported to other modules
- Class look up is faster for compiler and JVM because of explicitly defined dependencies
- Both the compiler and the JVM ensure that only referenced and exposed types can be accessed
- Using the public keyword on a class does not necessarily mean that classes from other modules can access this class

Language extensions introduced by Java Modules

- New keywords
  - Module -> to define a module
  - Requires -> to describe dependencies to other modules
  - Exports -> to declare which packages of a module are visible for other modules
- This new keywords are the core elements used to define a module
- The definition of a module, also called a module descriptor is provided in a file named "module-info.java"

The four different types of Java Modules (I/II)

- Named Plattform Modules
  - Modules of the JDK
  - Modules of this type can not access the module path
- Named Application Modules
  - Modules which bundle applications or libraries
  - Modular JAR's with a module-info.class file
  - Modules of this type can access the module path and all modules on the module path except the unnamed module
  - Modules of this type can not access the CLASSPATH and in turn the classes on the class path

The four different types of Java Modules (I/II)

- Open Modules
  - Same characteristics as Named Application Modules
  - Modules of this type open all of their packages for reflection
- Automatic modules
  - JAR files without a module-info.class become automatic modules if they are put on the module path

0. Overview

1. We start in JAR Hell

- We somehow ended up with two JAR's containing `UserServiceImpl` in the same package on the `CLASSPATH`
- Which implementation of the user service is used depdends on which implementation comes first on the `CLASSPATH`
- Try switching the order in which `user-service-provider-1` and `user-service-provider-2` a listed as dependencies
  in the build.gradle of `user-service-consumer`.

2. We want to escape JAR Hell and enjoy the world of explicit dependencies of Java 14

- Since we have control over all the source code our project relies on, we decide to apply bottom-up migration approach
  with named modules
- If we do not have control over all ther source code we rely on we use automatic modules instead of named modules

To create an executable for the application execute: `jlink --module-path $JAVA_HOME/jmods:build/install/application/lib --add-modules application --launcher application=application/ch.resrc.application.Application --output build/application --compress=2`

Execute the executable of the application

This command executes the executable created via jlink:
`build/install/application/bin/application`

Compare size of JDK to size of executable

Display size of JDK: `du -h $JAVA_HOME`
Display size of executable: `du -h build/application`

List dependencies of modules (resp. JAR files) with jdeps

To analyze the dependencies of the modules execute: `jdeps`