MODULAR PROGRAMMING

OVERVIEW

- Modules
- Creating and running modular programs
- File module-info.java
- Discovering modules
- Named vs automatic vs unnamed modules
- JDK dependencies
- Migrating an application
- Creating a service

MODULES

- Big projects consists of many JAR files
- Projects depend on other projects, so JAR files typically need other JARs to run
- Version management of all these JARs can be tough
- Modules are comparable to JAR files, but what's accessible outside the module can be controlled by the developer
- Java Platform Module System (JPMS) groups related packages
- JPMS comes with command-line options for Java tools

MODULE

- Contains module-info.java file
- One or more Java packages
- Java modules can depend on other modules

OH NO!

Just imagine: Maven died. So did Gradle. And etc.

BENEFITS OF MODULES I – ACCESS CONTROL

- Traditional options for access control:
  - Private
  - Default
  - Protected
  - Public
  - But what if we'd wanted something that's accessible to some packages only? Without making it pulic?

BENEFITS OF MODULES I – ACCESS CONTROL

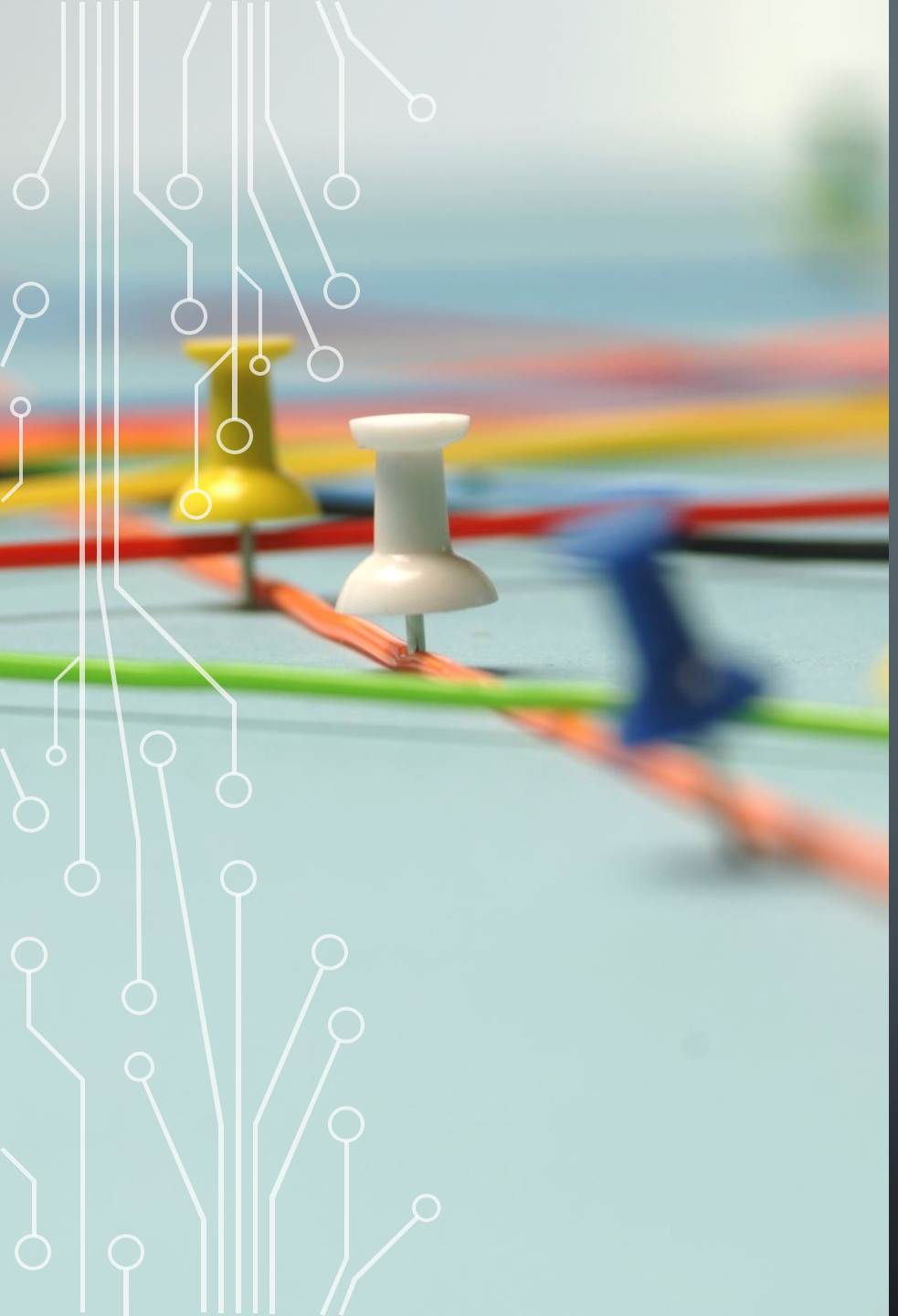
- Traditional options for access control:
  - Private
  - Default
  - Protected
  - Public
  - Protected and public accessible in other modules if the package that the public/protected members are in is exported

BENEFITS OF MODULES II – DEPENDENCY MANAGEMENT

- Libraries depend on other libraries
- Not including the right library can result in runtime exceptions
- Without Maven or Gradle, this is a hell to manage (with it's still sometimes though)
- Modules specify what dependencies they have and the program can check the module path and won't run without it
- No more double packages, a package is allowed to be supplied by one module only

BENEFITS OF MODULES III - PERFORMANCE

- During startup, only required modules need to be loaded during class loading
- Improves starting times



CREATING AND COMPILING MODULAR PROGRAMS

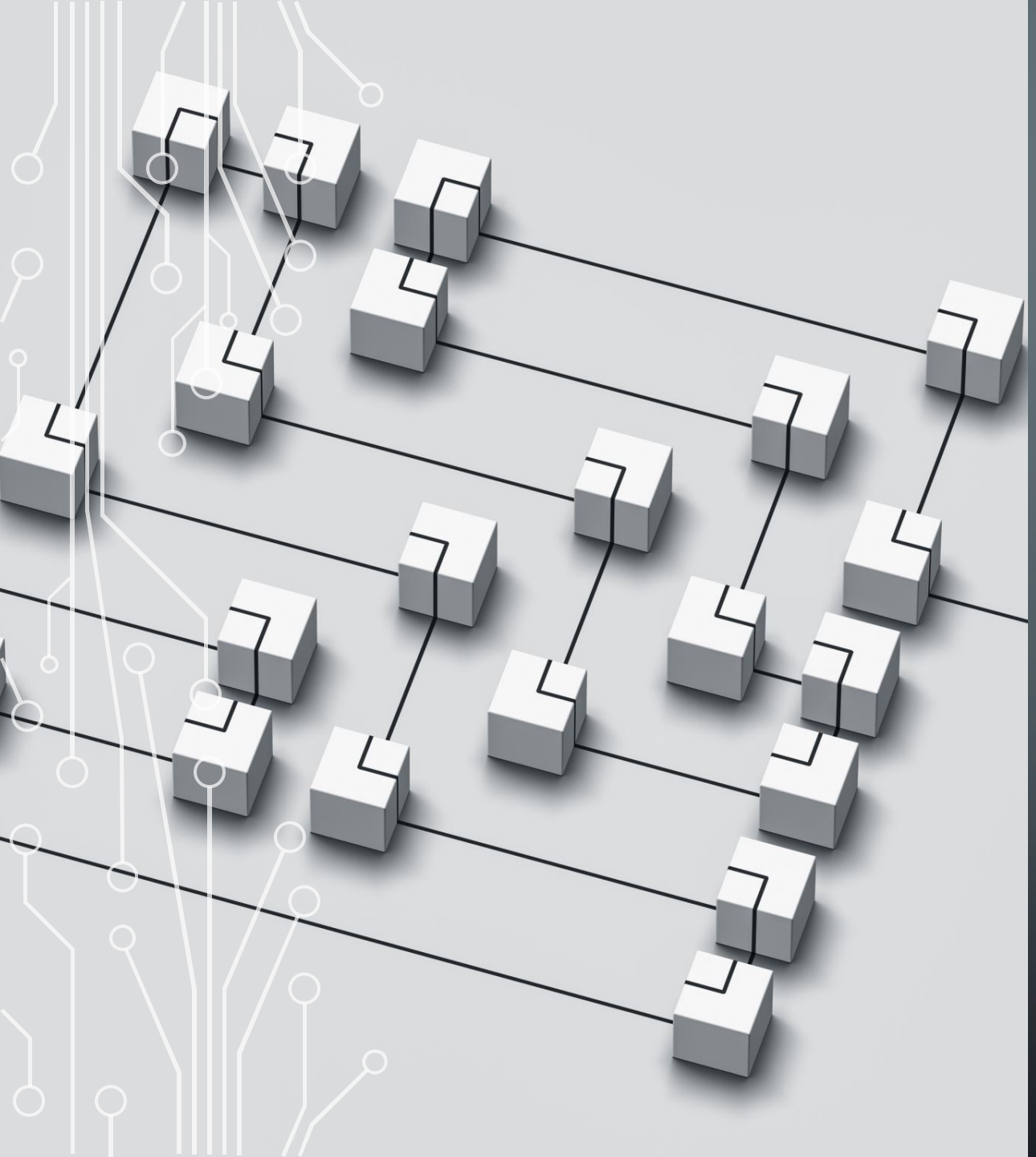
Ingredients:

- Files of your program
- Plus module-info.java at the root of the project

Compiling modules:

javac --module-path location-dependencies-files-directory -d directory-for-class-files some-file.java

Instead of --module-path we can use -p

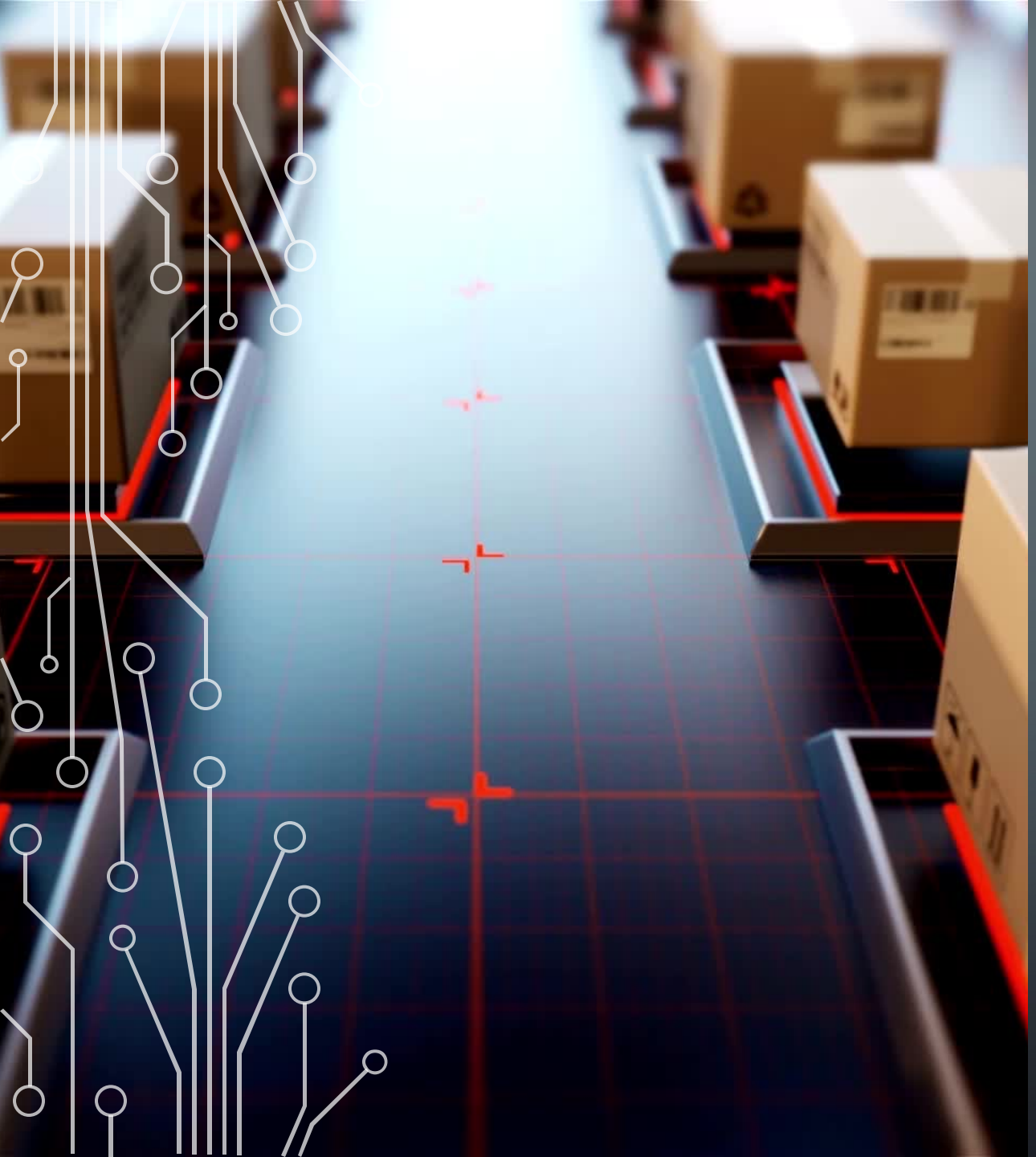


RUNNING MODULES

java --module-path location-module-dependencies --module module.name/nl.brightboost.example.ClassName

--module can be replaced with –m

--module-path can be replaced with -p



PACKAGE MODULES

- jar -cvf location-for-jar -C directory-to-be-packaged .

- Run jar:

java -p mp-location -m module.name/nl.brightboost.example.ClassName

MODULE-INFO.JAVA

- Basic module file that doesn't allow any outside of package access:

```
module name.of.module {
}
```

MODULE-INFO.JAVA

- Basic module file that allows outside of package access for exported package:

```
module name.of.module {
        exports nl.brightboost.example;
}
```

Exporting means that all public classes, interfaces and enums are exported

The public and protected fields in these are visible

MODULE-INFO.JAVA

- Basic module file that allows outside of package access for exported package to a specific module:

```
module name.of.module {
        exports nl.brightboost.example to nl.brighboost.something;
}
```

MODULE-INFO.JAVA

- Basic module file that allows outside of package access for exported package and requires another one:

```
module name.of.module {

        exports nl.brightboost.example;

        requires nl.brightboost.another;

}
```

EXERCISE

- Go ahead and create the modules on page 459 - 472

MODULE-INFO.JAVA

- Basic module file that allows outside of package access for exported package and requires another one:

```
module name.of.module {
        exports nl.brightboost.example;
        requires transitive nl.brightboost.another;
}
```

”requires transitive” means that anything that depends on this module also depends on the require transitive module

Same module cannot be required twice (so also not requires transitive and requires for the same module)

EXERCISE

- Go ahead and improve the dependencies by using requires transitive

OTHER MODULE-INFO.JAVA KEYWORDS

- Provides: specifies that a class provides an implementation of a service

provides nl.brightboost.example.SomeApi with nl.brightboost.example.SomeImplementation;

- Uses: Specifies that a module relies on a service

uses nl.brightboost.example.SomeApi;

- Opens: allows usage of reflection in the specified packages

opens nl.brightboost.example (to nl.brightboost.specific.package);

DISCOVERING MODULES I

- Describe module to know what a module requires and exports:

java -p location-of-module-jar --describe-module name.of.module

- Get available modules:

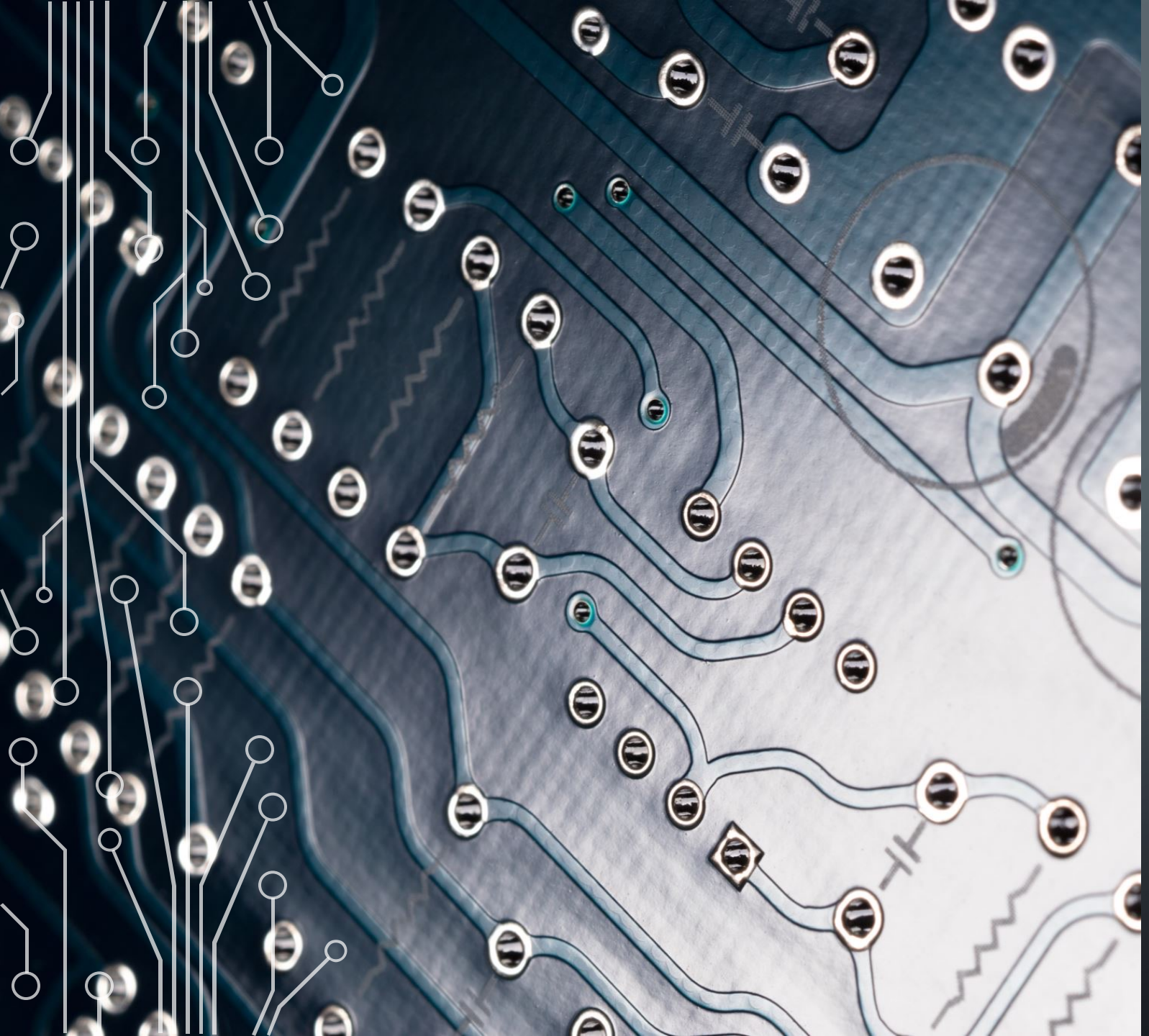
java --list-modules

java -p location-of-mod --list-modules

DISCOVERING MODULES II

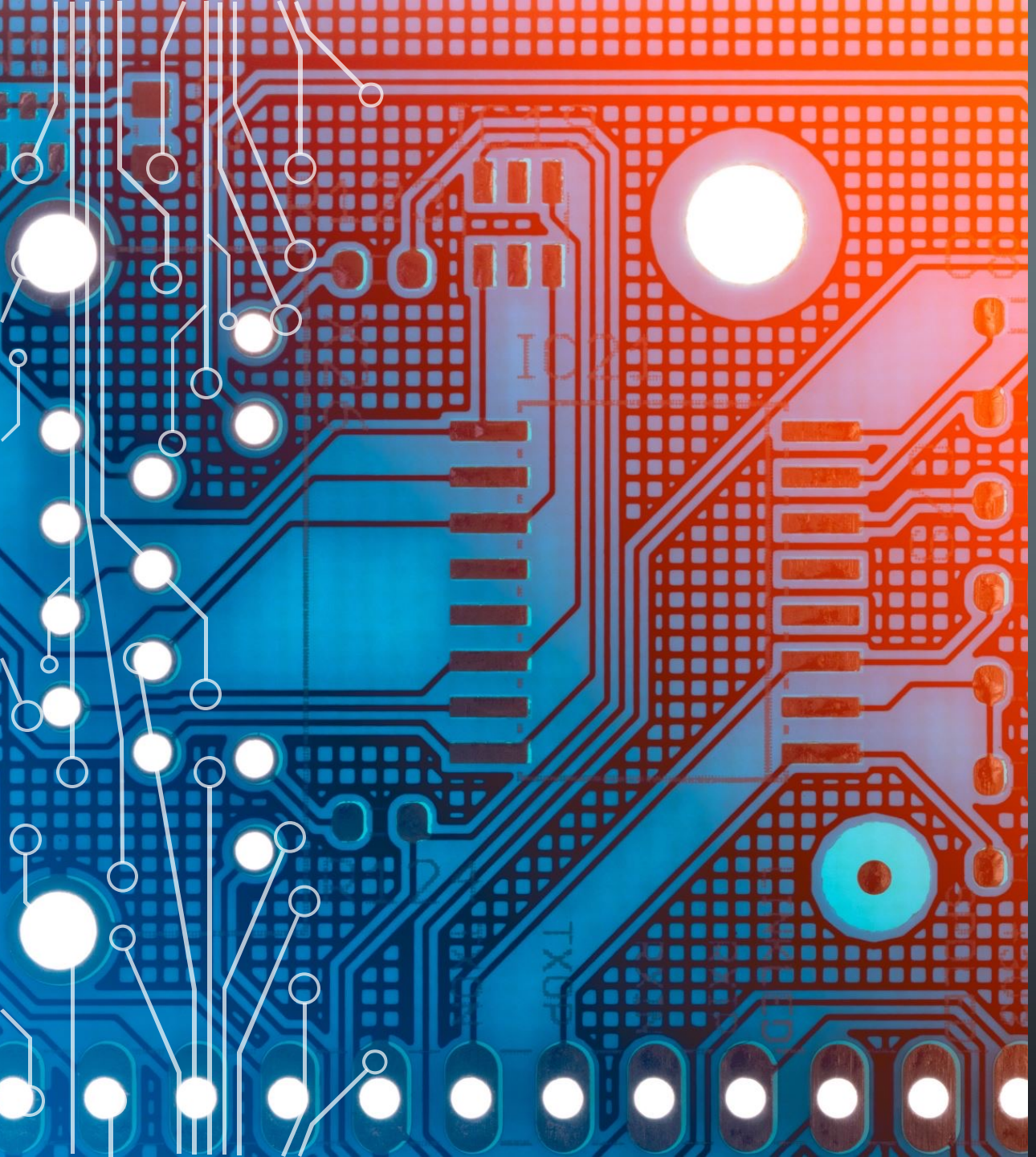
Dependencies in the module with jdeps (jdeps does not have the -p to abbreviate --module-path)

- jdeps (--summary) --module-path location-dependencies-mods location/name.jar



NAMED MODULES

- So far, we have seen named modules
- Names modules contain a module-info file
- Named modules appear on the module path

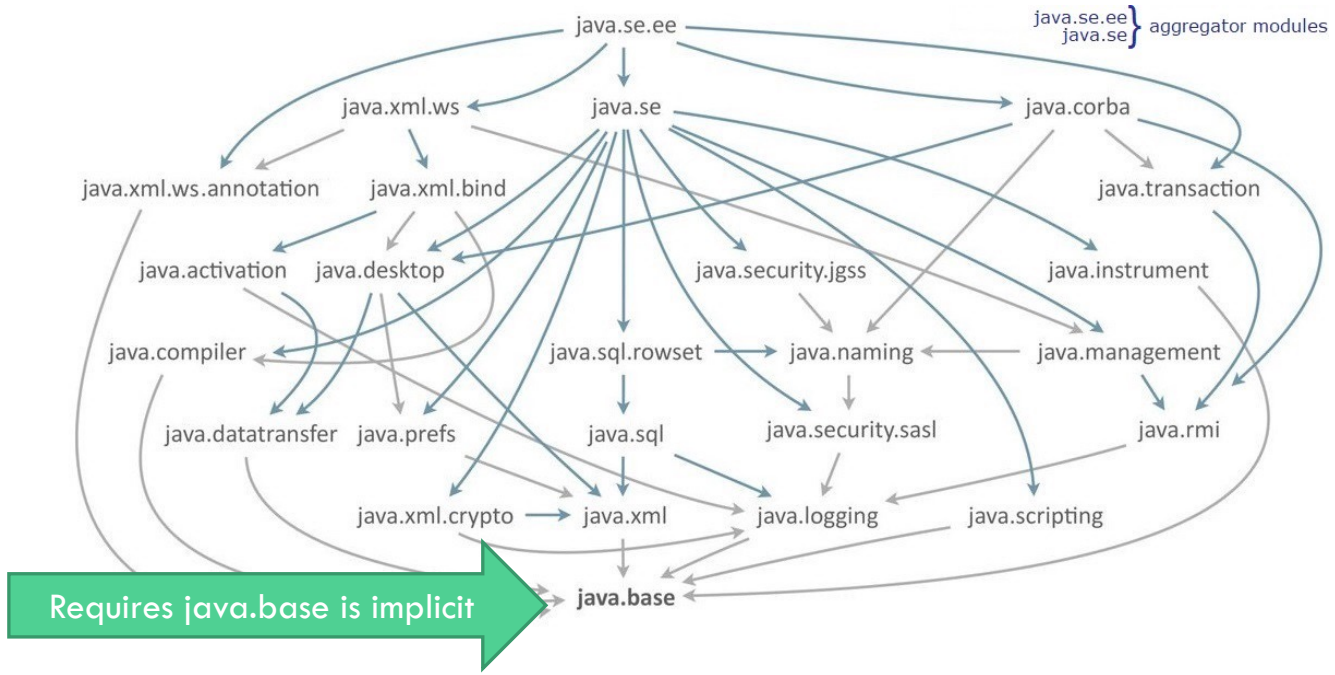


AUTOMATIC MODULES

- Appears on the module path but doesn't have a module-info
- Just a JAR file that's placed on the module-path
- Everything is being exported
- Name is automatically generated:
  - If available in MANIFEST.MF >> that name
  - Uses jar name, -.jar and –version. Dashes are replaced with dots. Double and leading/trailing dots are removed.

UNNAMED MODULES

- On the classpath
- If it contains a module-info file, it's ignored (since it's on the classpath)
- No exports to named or automatic modules
- It can use JARs on classpath and module path
- It's actually not really a module, just how Java was working before modules

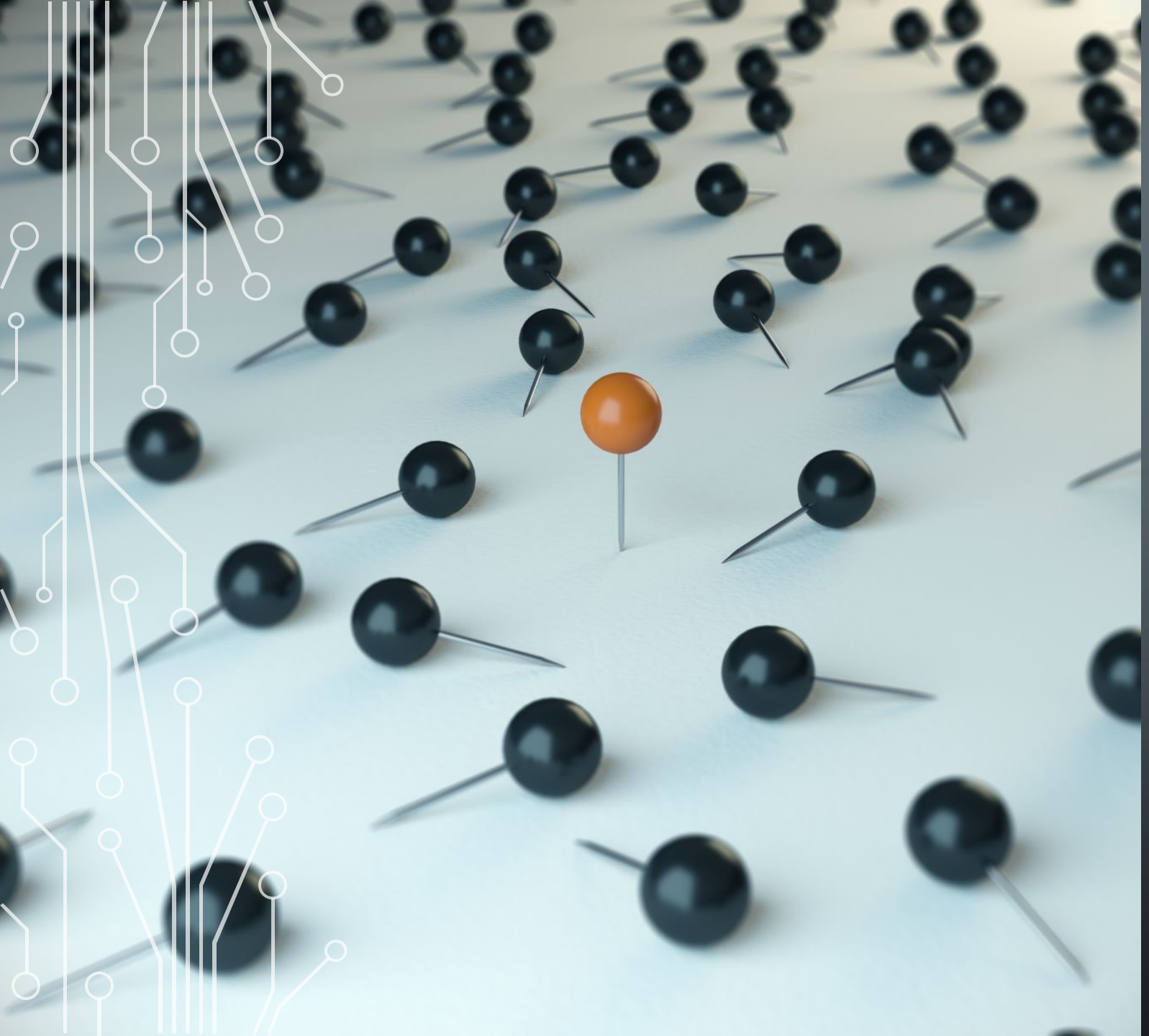


JDK DEPENDENCIES

HTTPS://MEDIUM.COM/@DVSINGH9/JAVA-MODULES-A-COMPLETE-GUIDE-PART-1-DEA3E6210177

MIGRATING AN APPLICATION I

- Ordering modules to find the dependencies
- Bottom-up migration:
  - Start with least dependencies
  - Add module-info and the necessary exports / requires
  - Move to module path, make sure others stay on the classpath until they're migrated
  - Next with least dependencies until done



MIGRATING AN APPLICATION II

- Top-down migration
  - Place all projects on the module path
  - Choose the one with most dependencies
  - Add module info, and exports and requires
  - Repeat with nextz most dependencies one, until done

SERVICES

- Service:
  - Interface
  - Classes used by the interface
  - And a way of looking up implementations of the interface
  - (Implementations are not part of the service)

CONSUMER

MODULE THAT USES A SERVICE

GETS AN IMPLEMENTATION VIA THE SERVICE LOCATOR

SERVICE PROVIDER

- Implementation of the service interface
- Multiple available ones at runtime is possible
- Module-info requires the interface, we don't export the implementation directly, instead we *provide* it in module-info.java:
  - provides interfaceName with className;

EXERCISE

- Follow along with the example in the book pages 823-830