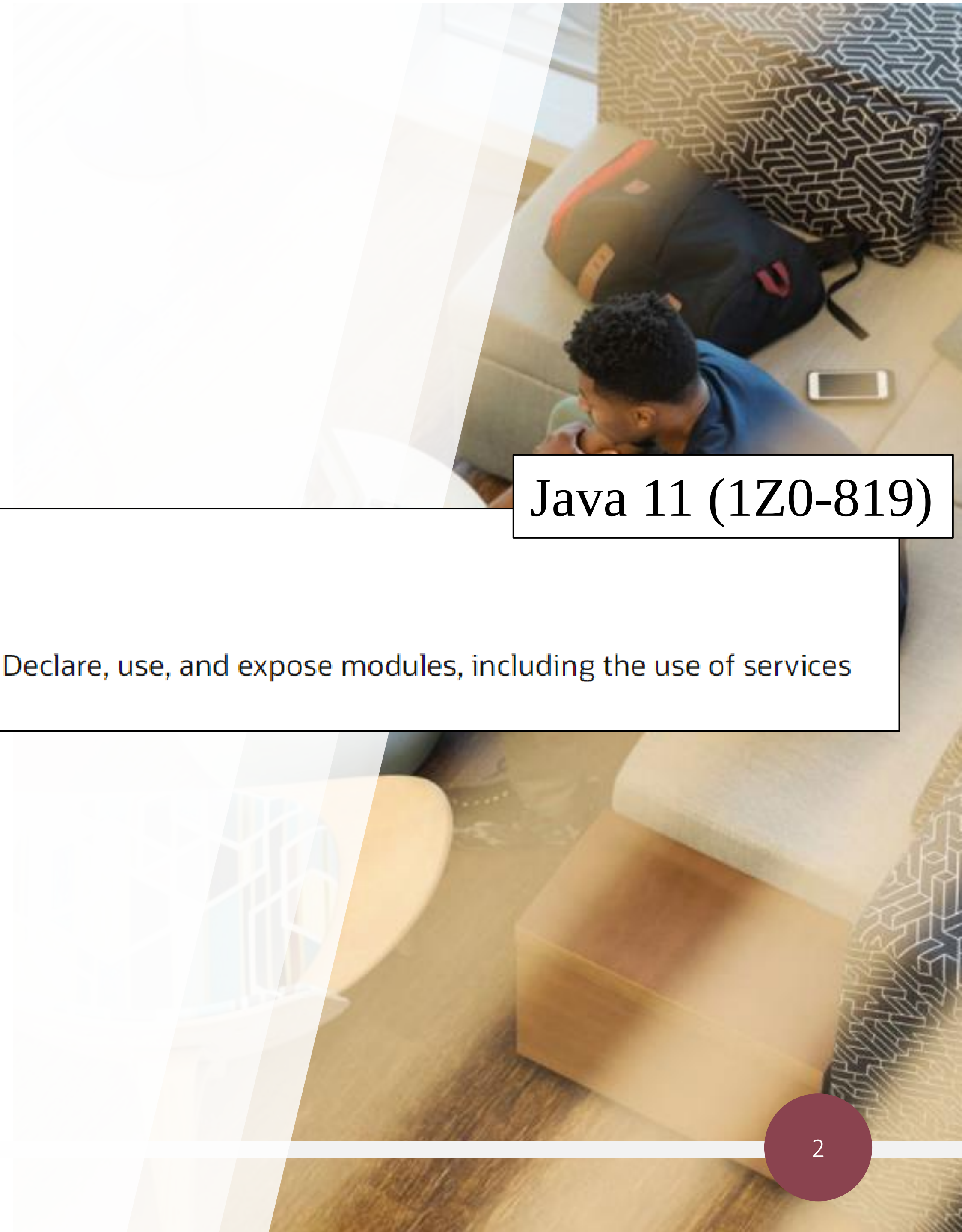
Modules

Modules



Java 11 (1Z0-819)

Java Platform Module System

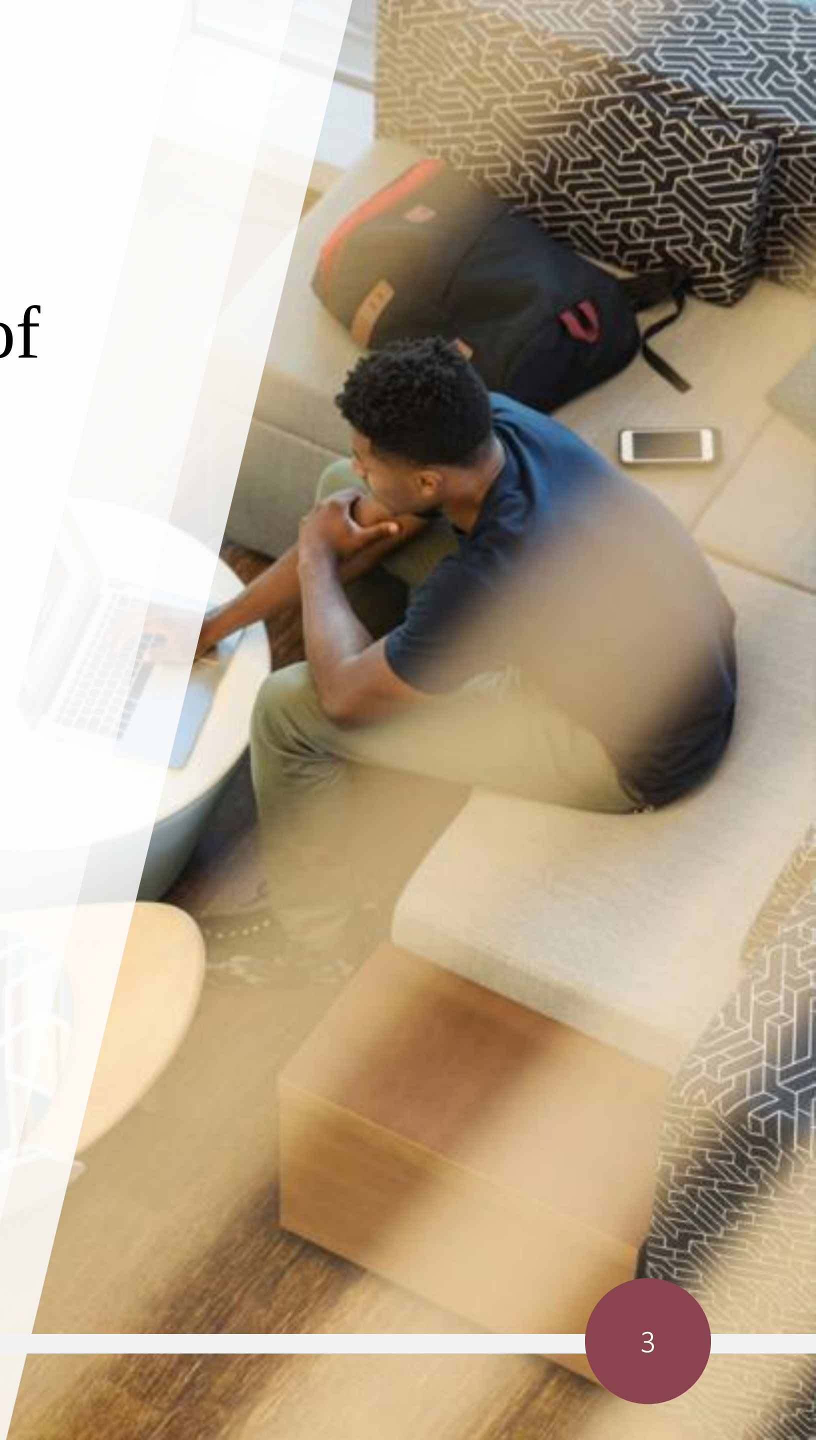
- ✓ Deploy and execute modular applications, including automatic modules
- ✓ Declare, use, and expose modules, including the use of services

Modules

- Modules, introduced in Java 9, provide an extra layer of encapsulation by enabling us to group related packages.

- From a high-level, modules enable us to specify well-defined boundaries/dependencies in our code base.

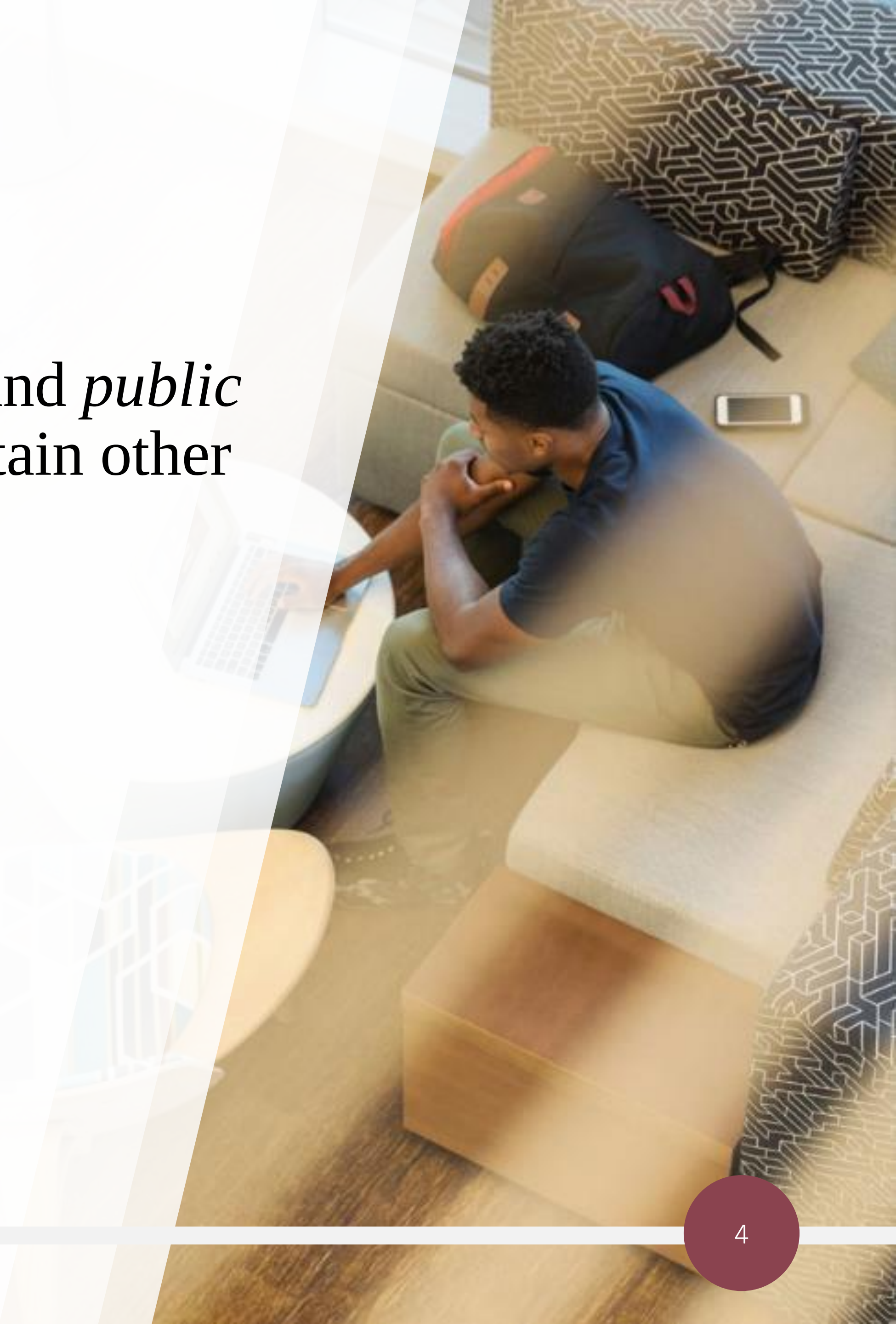
- As a developer, you can specify which packages are accessible outside the module and also the module dependencies (the other modules that the module itself depends upon).



Why Modules?

- Improved access control:
  - in addition to *private*, package-private, *protected* and *public* now we have the ability to restrict packages to certain other packages.

- Improved large-scale structure of applications:
  - boundaries/dependencies in your code base.
  - hide implementation details.
  - improved decoupling.

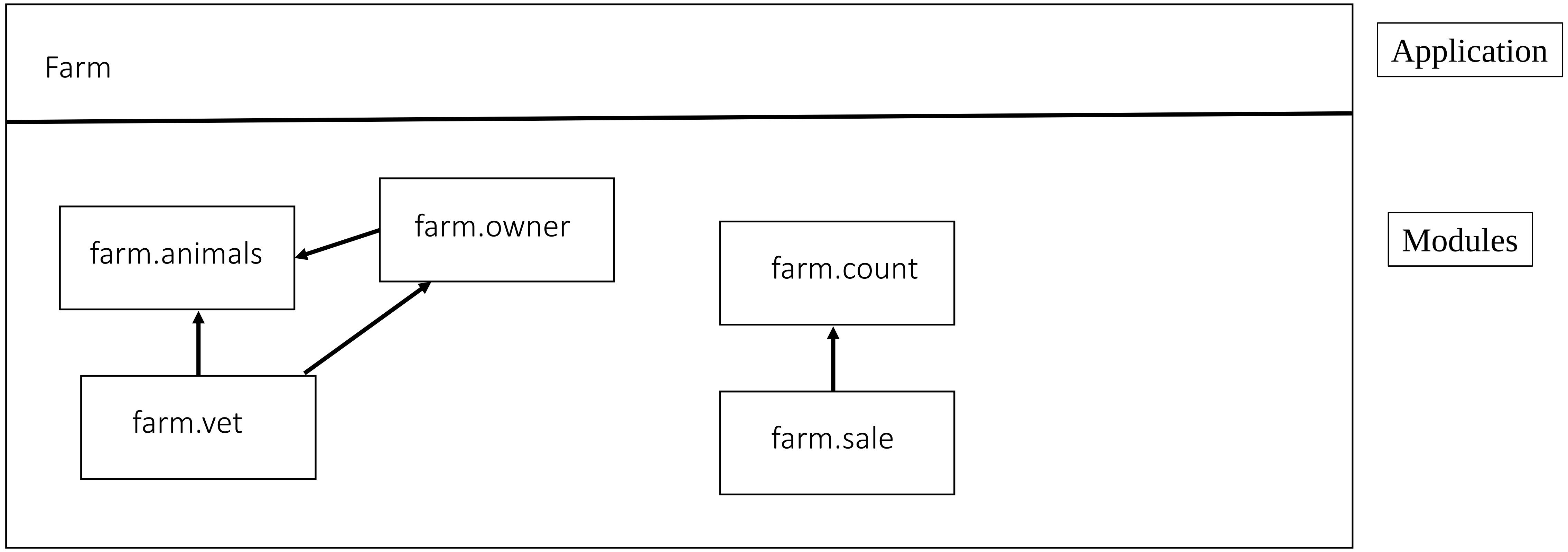


Why Modules?

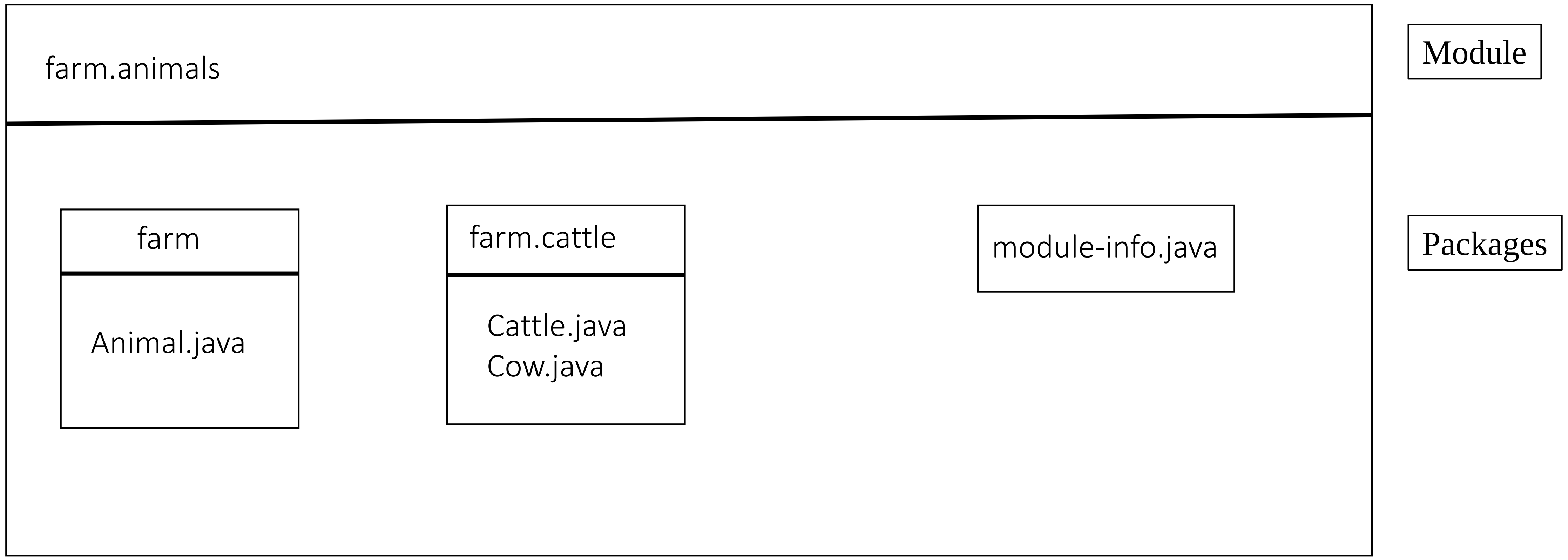
- Reduced build sizes:
  - as Java itself is now modularised, developers can specify which of the Java API modules are required for their custom applications; this results in smaller application sizes (a consideration for memory-constrained devices).

- Earlier detection of missing code:
  - prior to modules, missing classes/JAR files might only be realised at runtime, when the class/JAR file was being used for the first time.
  - as modules specify their dependencies - in a fully modular environment, the Java VM can check the whole dependency graph when it starts up; thus, any missing modules would be spotted much earlier.

Modular Application



Structure of a Module

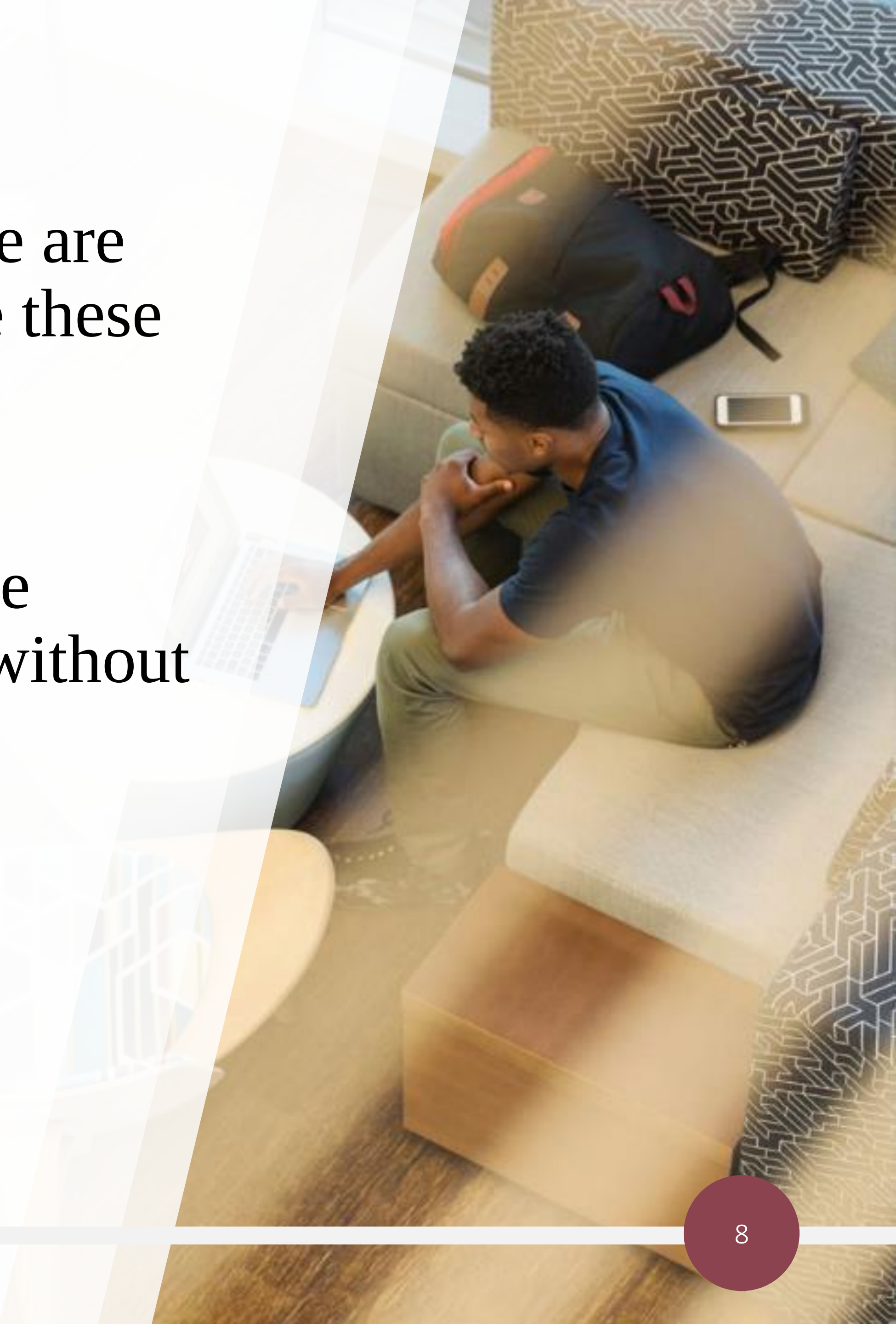


Modules

- By default, all classes in the packages in a module are strongly encapsulated i.e. no other module can use these classes, even if they are *public* classes.

- Package names and module names live in separate namespaces i.e. you can have the same identifiers without any conflict (name clash).

- A module contains one or more packages plus a *module-info.java* file.

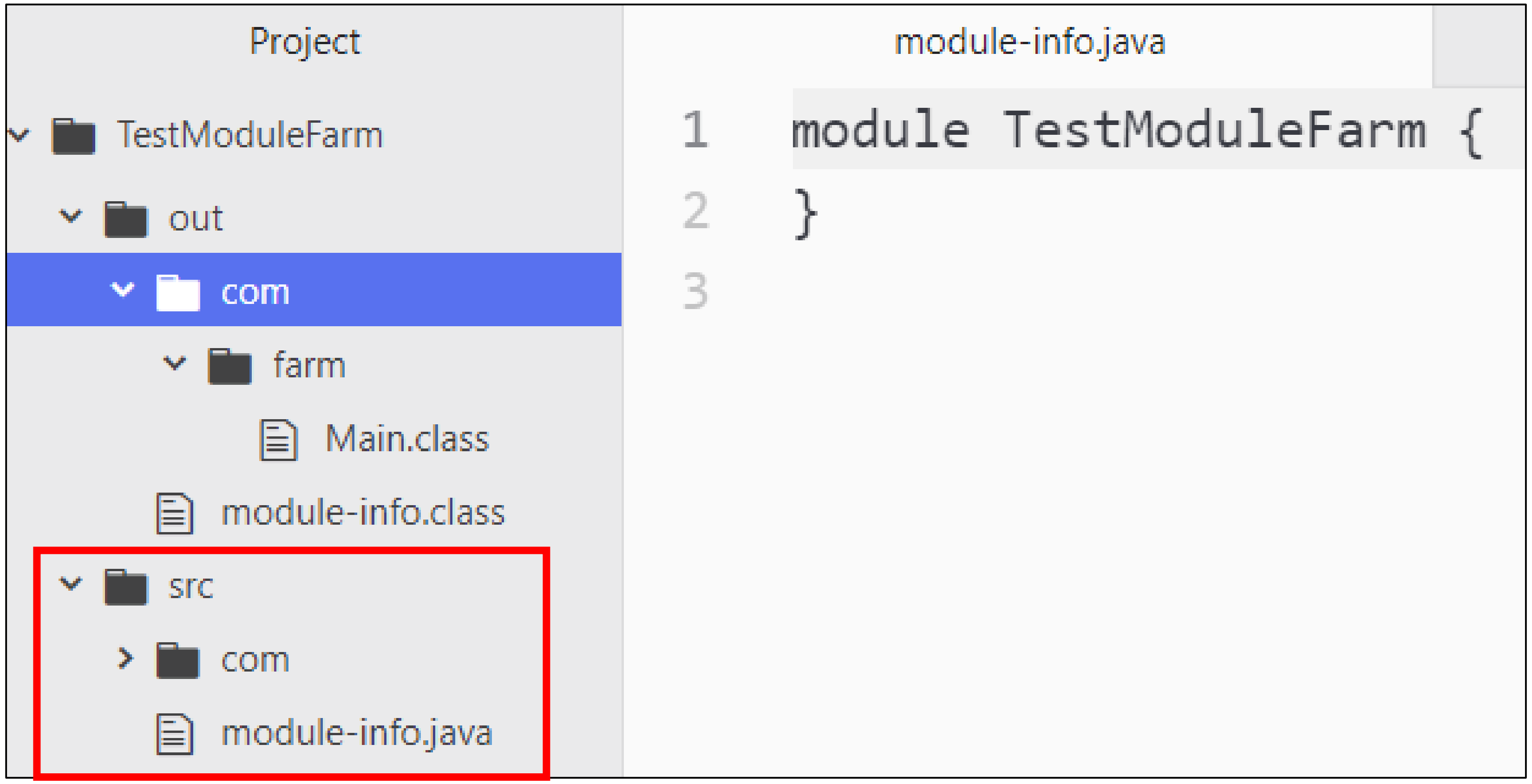


Modules

- When compiling a module, it is more convenient to use "module directories" as otherwise we would have to list all the *.java* source files (including *module-info.java)*.

- If you use a module directory, then the name of the module directory must match the name of the module in the *module-info.java* file.

- Place the *module-info.java* file in the root of the source tree of the module you are describing.



```
module TestModuleFarm {
}
```

javac (compiler) flags

Listing all the source files:	javac –**d** {dir} {all the java source files, including module-info.java}
Module directory(ies):	javac –**d** {dir} –**m** {module_name}, {module_name} **--module-source-path** {src_dir}
Dependencies to other custom (possibly 3rd party) modules:	**--module-path** or –**p**
Examples:	javac -d out src/com/farm/Main.java src/module-info.java javac -d out **-m** TestModuleFarm **--module-source-path** src javac -d out **--module** TestModuleFarm --module-source-path src javac –d {dir} –m {module_name} --module-source-path {src_dir} –**p** {dir e.g. “mods”}

Short Version	**Long Version**
-m	--module
-p	--module-path

java flags

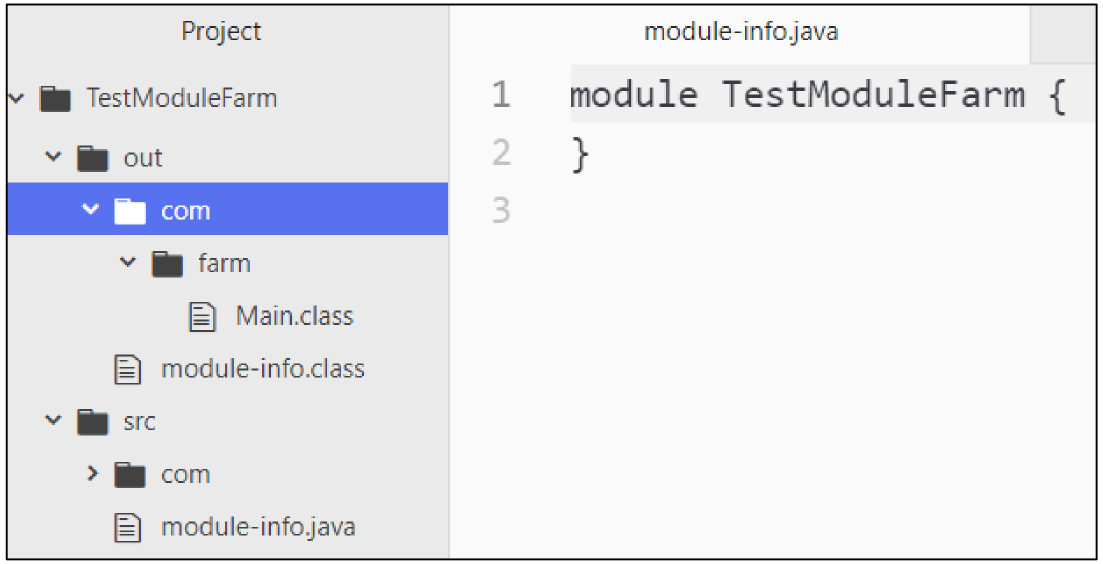
	java –**p** {module_path} –**m** {module}/{fully qualified main class}
Example:	java **-p** out **-m** TestModuleFarm/com.farm.Main
Example:	java **--module-path** out **--module** TestModuleFarm/com.farm.Main

Short Version	**Long Version**	**Comment**
-m	--module	-
-p	--module-path	The “classpath” for modules.

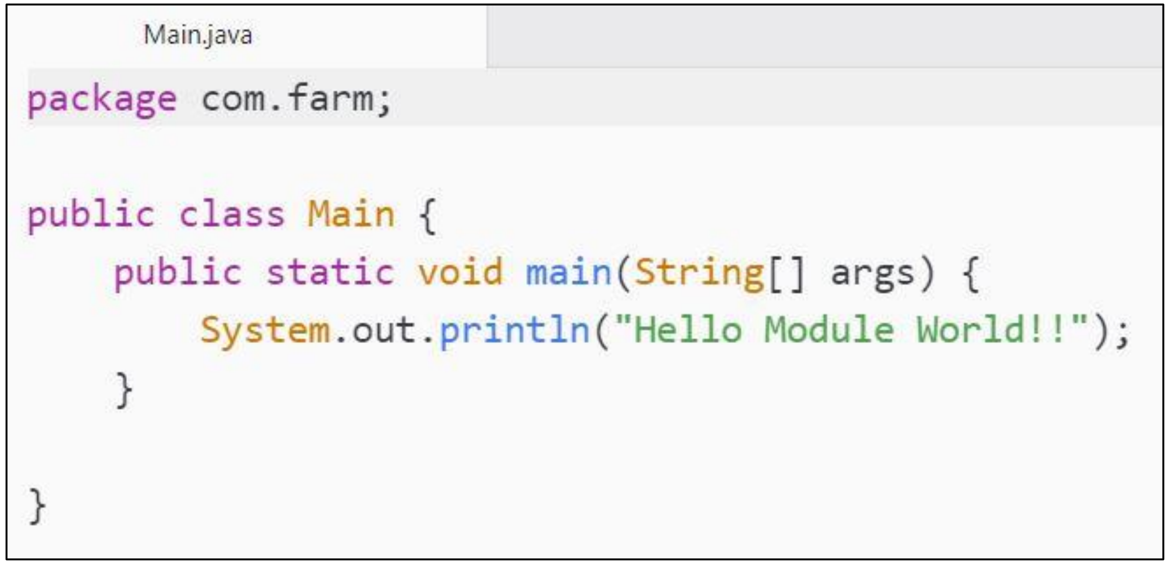
Flags Recap

Command	Flags	Question flags answer
javac	-d {dir}	where to put the class files?
	-m {module_name} --module {module_name}	what is the name of the module?
	--module-source-path {src_dir}	where is the module source code?
	-p {dir e.g. “mods”} --module-path {dir e.g. “mods”}	where are the custom modules we need (if any) ?
	Example:	javac -d out -m TestModuleFarm --module-source-path src
java	-p {module_path} --module-path {module_path}	where is the module?
	-m {module}/{fully qualified main class} --module {module}/{fully qualified main class}	what is the name of the module?
	Example:	java -p out -m TestModuleFarm/com.farm.Main

TestModuleFarm Listing all source files



```
module TestModuleFarm {
}
```

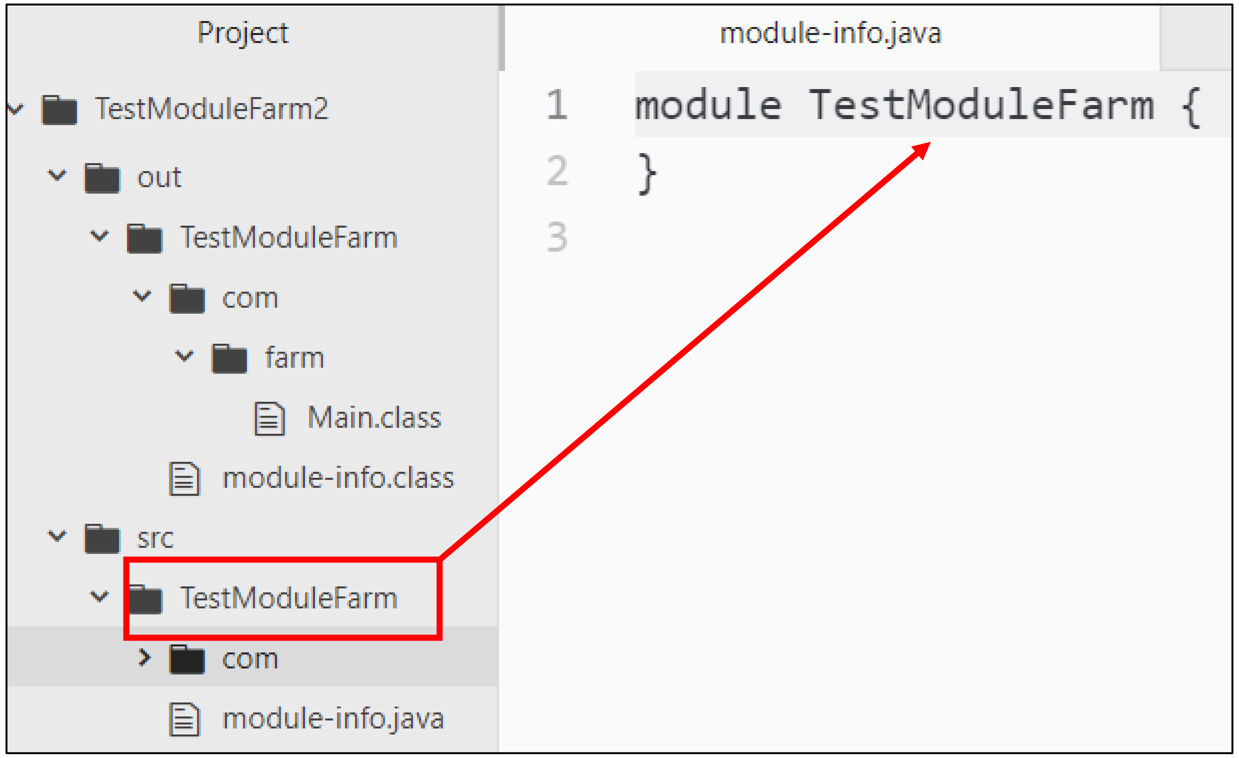


```
package com.farm;

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

}
```

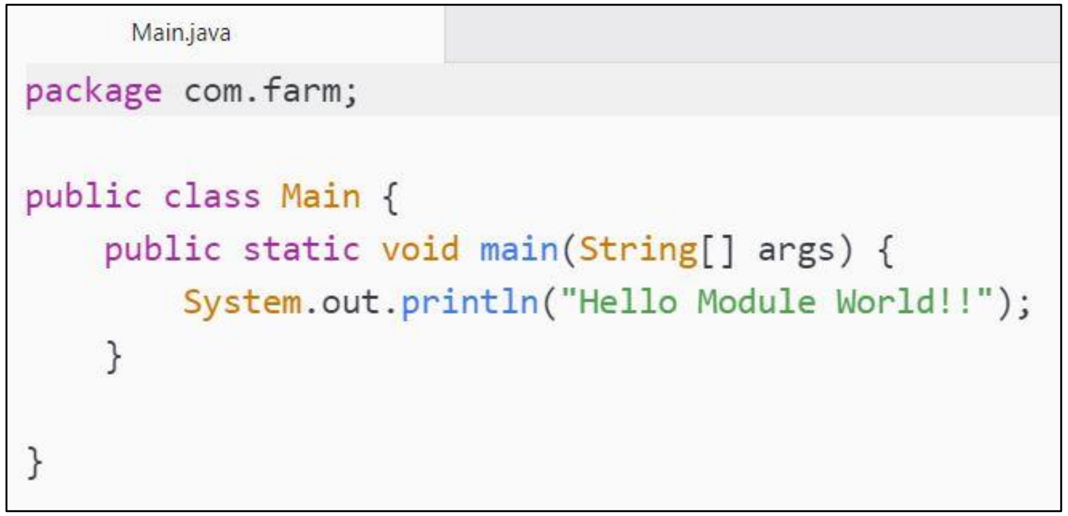
```
javac -d out src/com/farm/Main.java src/module-info.java
java -p out -m TestModuleFarm/com.farm.Main
```

```
Hello Module World!!
```

TestModuleFarm2
Module Directory



```java
module TestModuleFarm {
}
```



```java
package com.farm;

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

}
```

```
javac -d out --module TestModuleFarm --module-source-path src
java -p out -m TestModuleFarm/com.farm.Main
```

```
Hello Module World!!
```

Dependencies between Modules

- As stated earlier, by default, all classes in the packages in a module are strongly encapsulated i.e. no other module can use these classes, even if they are *public* classes.

- If you want a package to be visible outside the module then "export" it in the *module-info.java* file. The keyword *exports* is used.

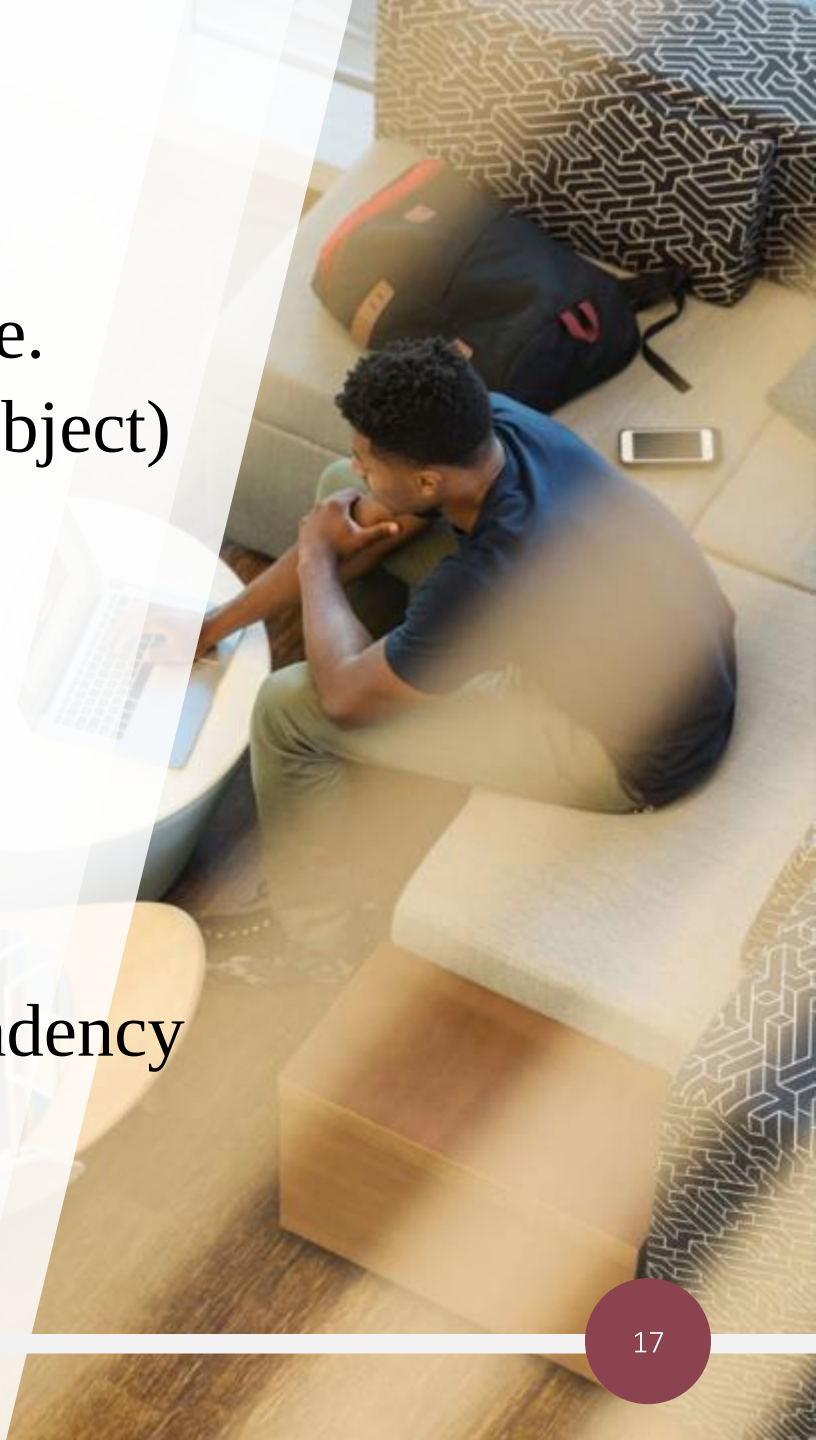
- If a module "depends" on another module, the keyword *requires* is used.

Dependencies between Modules

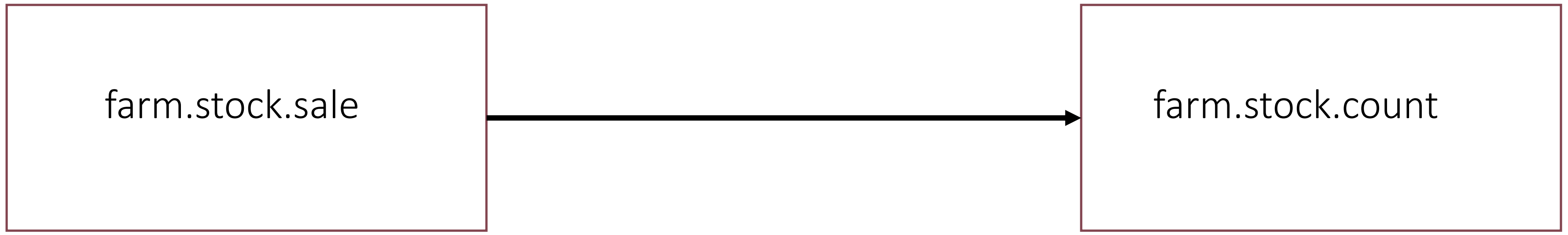
- All modules implicitly depend on the *java.base* module.
  - the module exports the *java.lang* package (String, Object)

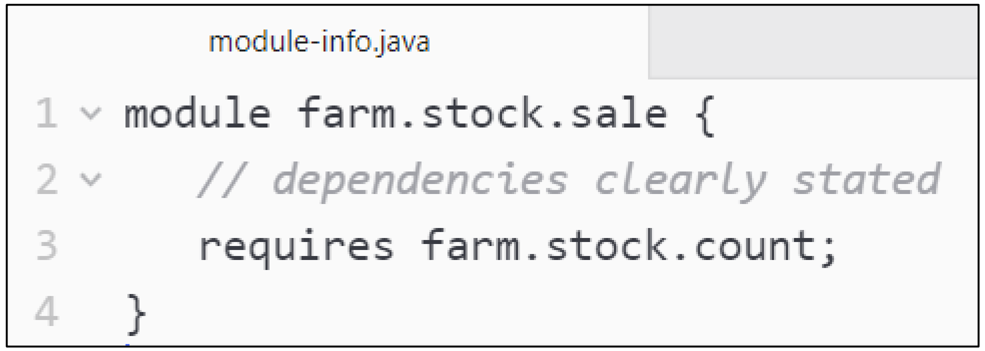
- Remember:
  - you ex**p**ort (*exports*) a **p**ackage
  - you **d**epend (*requires*) on a mo**d**ule

- The *requires* keyword between modules forms a dependency graph.

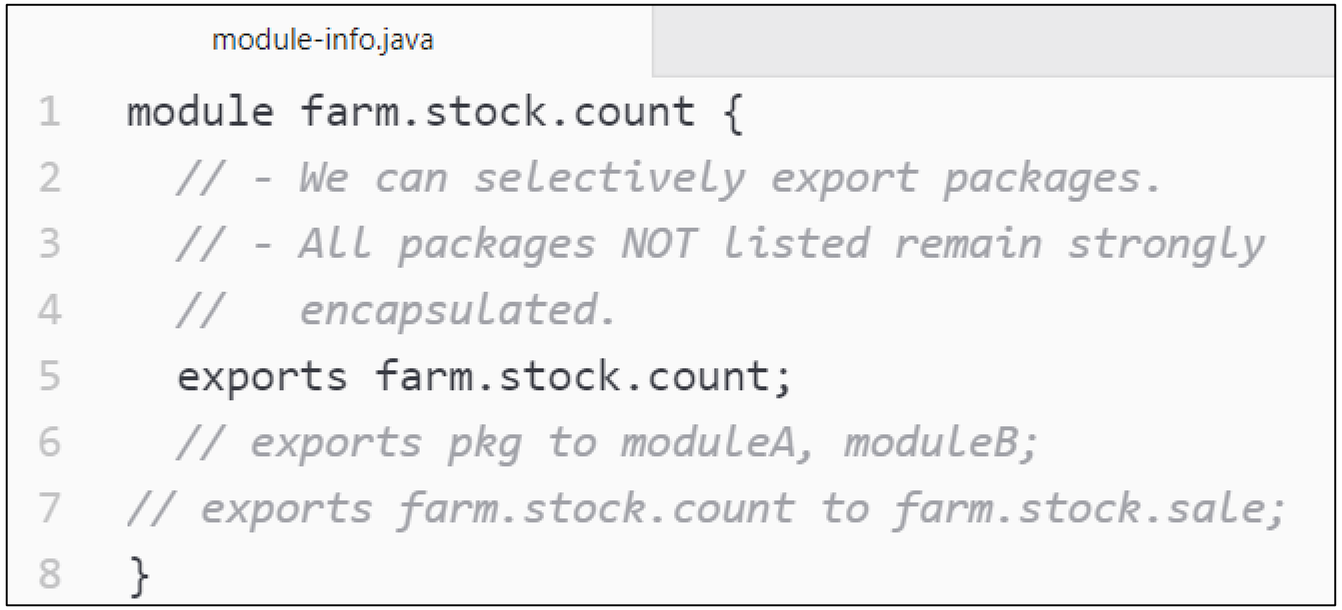


Dependencies between Modules




module-info.java

```
module farm.stock.sale {
    // dependencies clearly stated
    requires farm.stock.count;
}
```


module-info.java

```
module farm.stock.count {
  // - We can selectively export packages.
  // - All packages NOT listed remain strongly
  //   encapsulated.
  exports farm.stock.count;
  // exports pkg to moduleA, moduleB;
// exports farm.stock.count to farm.stock.sale;
}
```

1. Run SellThem example.
2. Jar version.

opens

- Frameworks such as Spring and Hibernate, frequently require access to our code at runtime via "reflection".
    - Class.forName("farm.stock.sale.SellThem").newInstance();

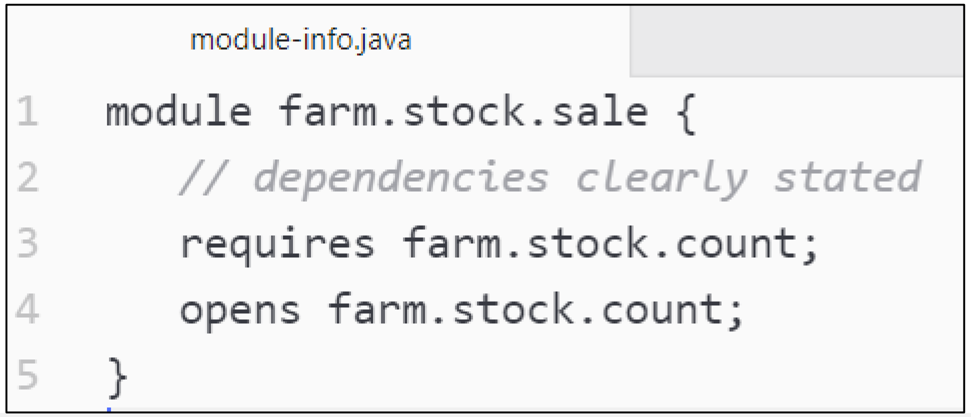
- Via reflection, calling code can override access control and as a result of this power, you must clearly state that you allow this type of access.

- Thus we want our types to be strongly encapsulated at compile-time but provide runtime access via reflection. The keyword *opens* provides this.

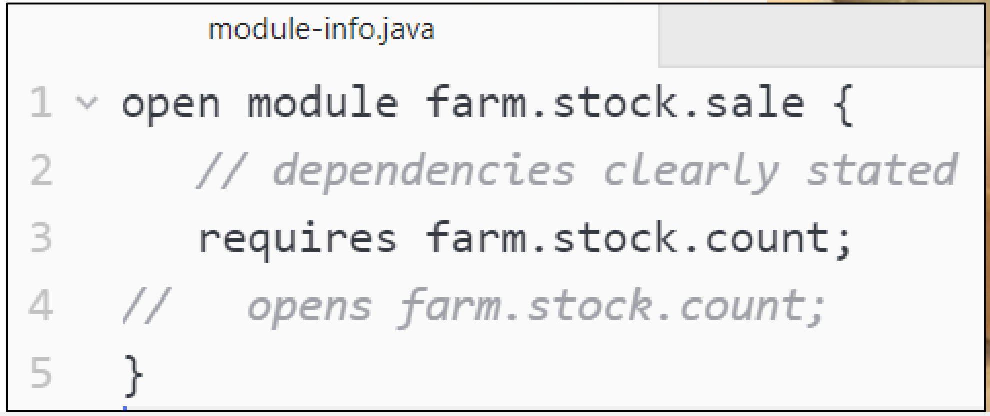
opens

- In your module-info.java file you state:
  - *opens packageName;*
  - *opens packageName to moduleName;*

- A module itself can be *open*. This automatically opens all packages inside the module.

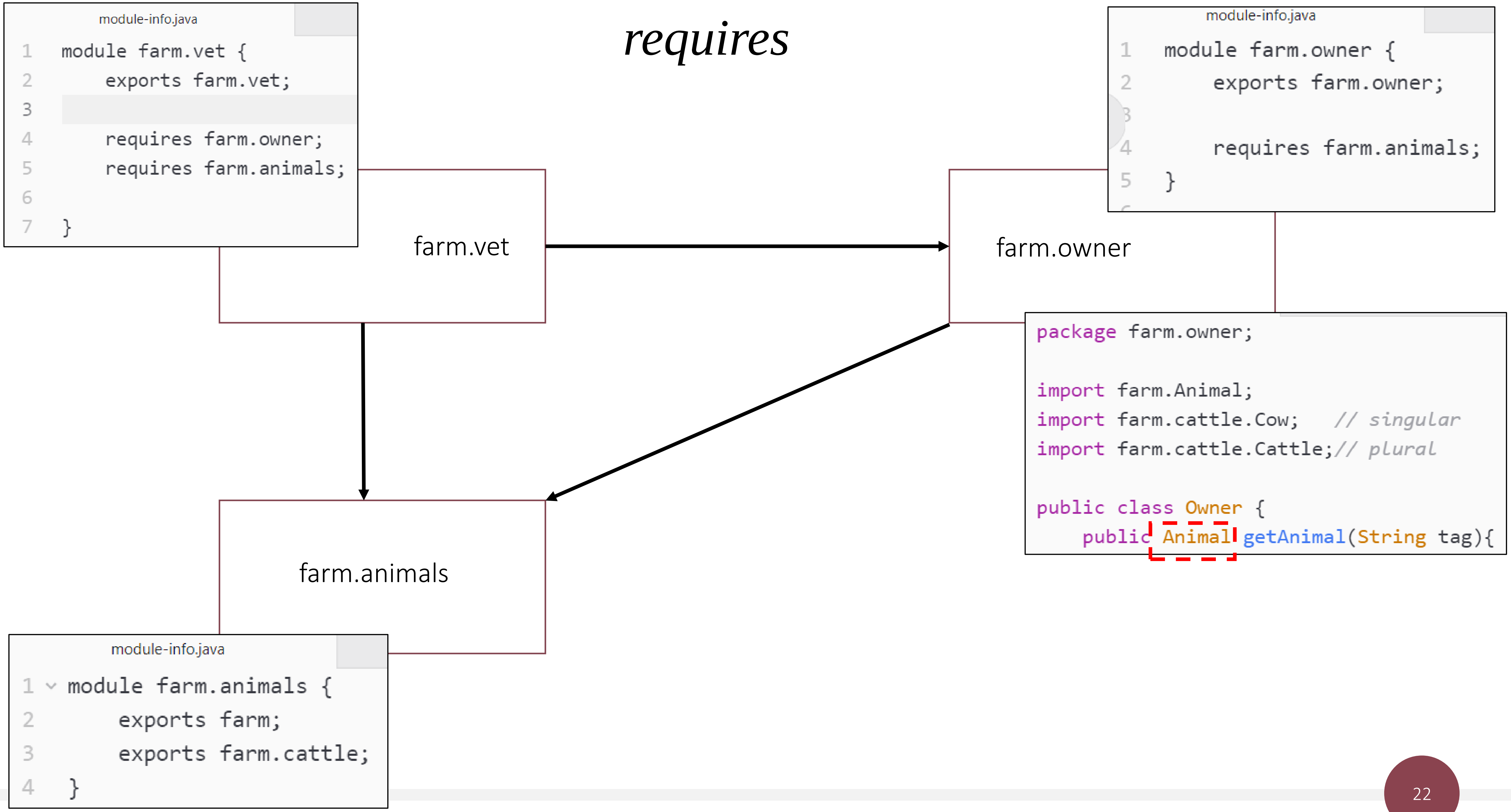

module-info.java

```
module farm.stock.sale {
   // dependencies clearly stated
   requires farm.stock.count;
   opens farm.stock.count;
}
```


module-info.java

```
open module farm.stock.sale {
   // dependencies clearly stated
   requires farm.stock.count;
//    opens farm.stock.count;
}
```

requires



module-info.java

```
module farm.vet {
    exports farm.vet;

    requires farm.owner;
    requires farm.animals;

}
```

module-info.java

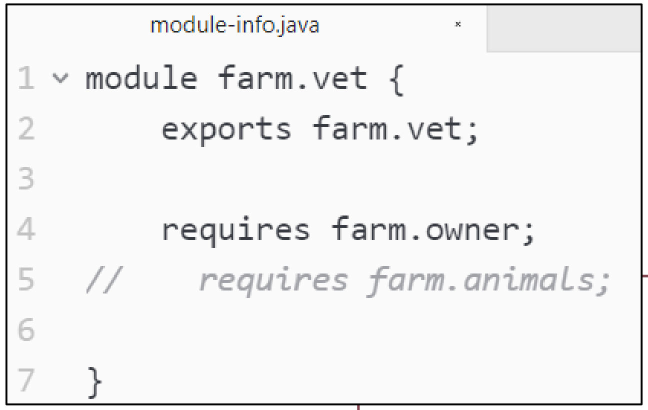
```
module farm.owner {
    exports farm.owner;

    requires farm.animals;
}
```

```
package farm.owner;

import farm.Animal;
import farm.cattle.Cow;    // singular
import farm.cattle.Cattle;// plural

public class Owner {
    public Animal getAnimal(String tag){
```

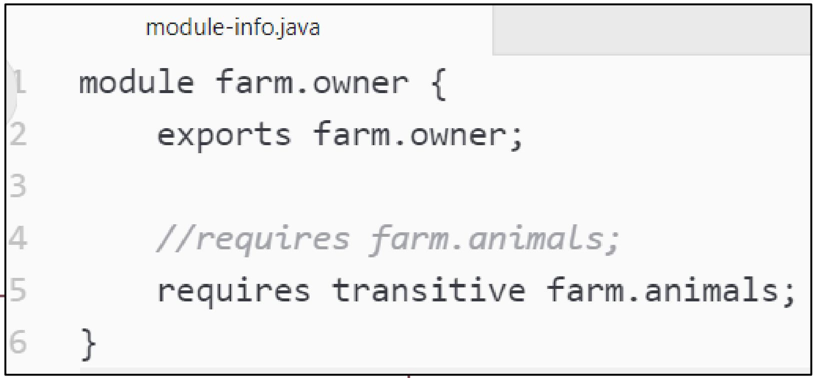
module-info.java

```
module farm.animals {
    exports farm;
    exports farm.cattle;
}
```

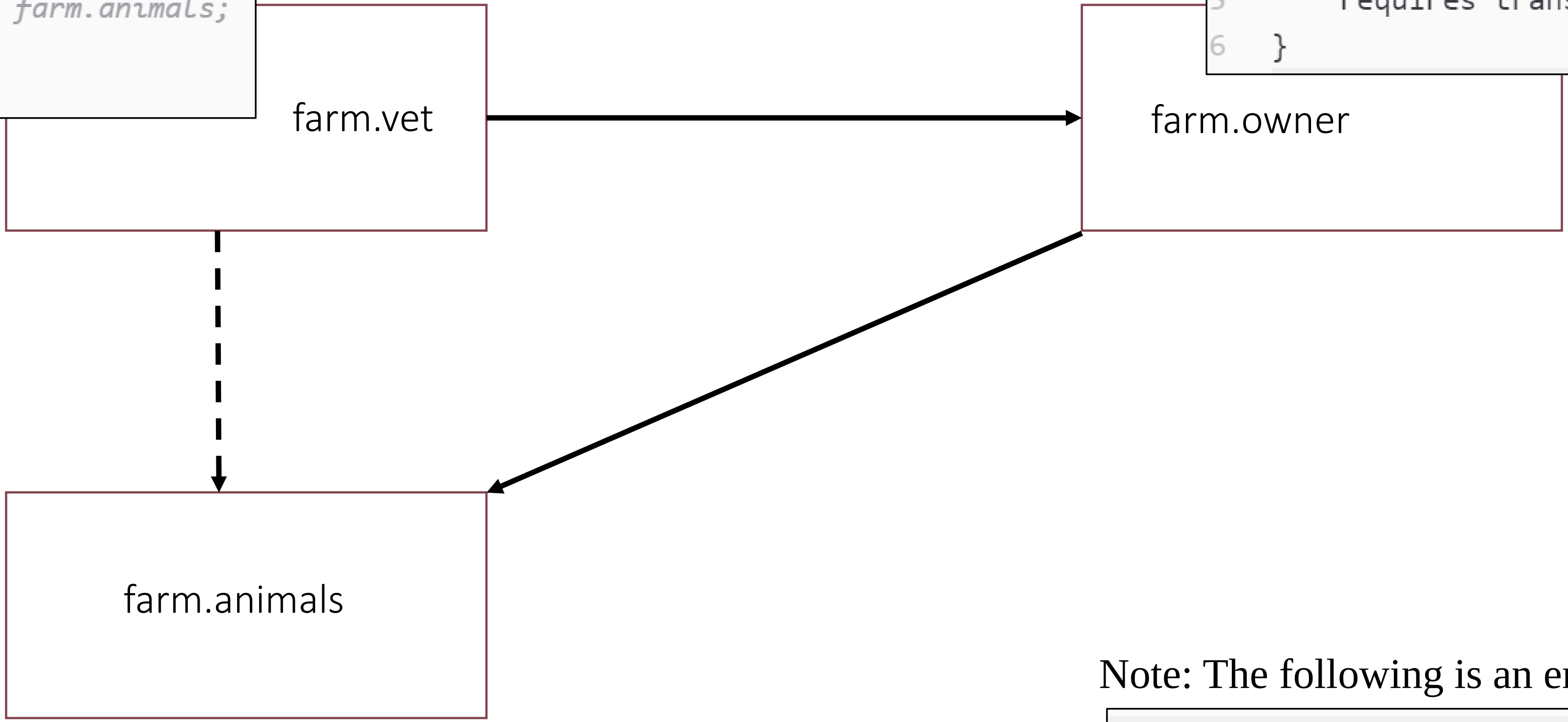
requires transitive



```
module farm.vet {
      exports farm.vet;

      requires farm.owner;
//    requires farm.animals;

}
```



```
module farm.owner {
      exports farm.owner;

      //requires farm.animals;
      requires transitive farm.animals;
}
```



Note: The following is an error:

```
requires farm.animals;
requires transitive farm.animals;
```

Operations on a Module

- Packaging a module
  - creating and executing a modular JAR (Java ARchives).
- Describing a module (*java* and *jar*).
- *jdeps* – discovering the dependencies in the module.
- *jmod*
  - native libraries
  - not executable
  - intended for use with the *jlink* tool to build a custom native image.

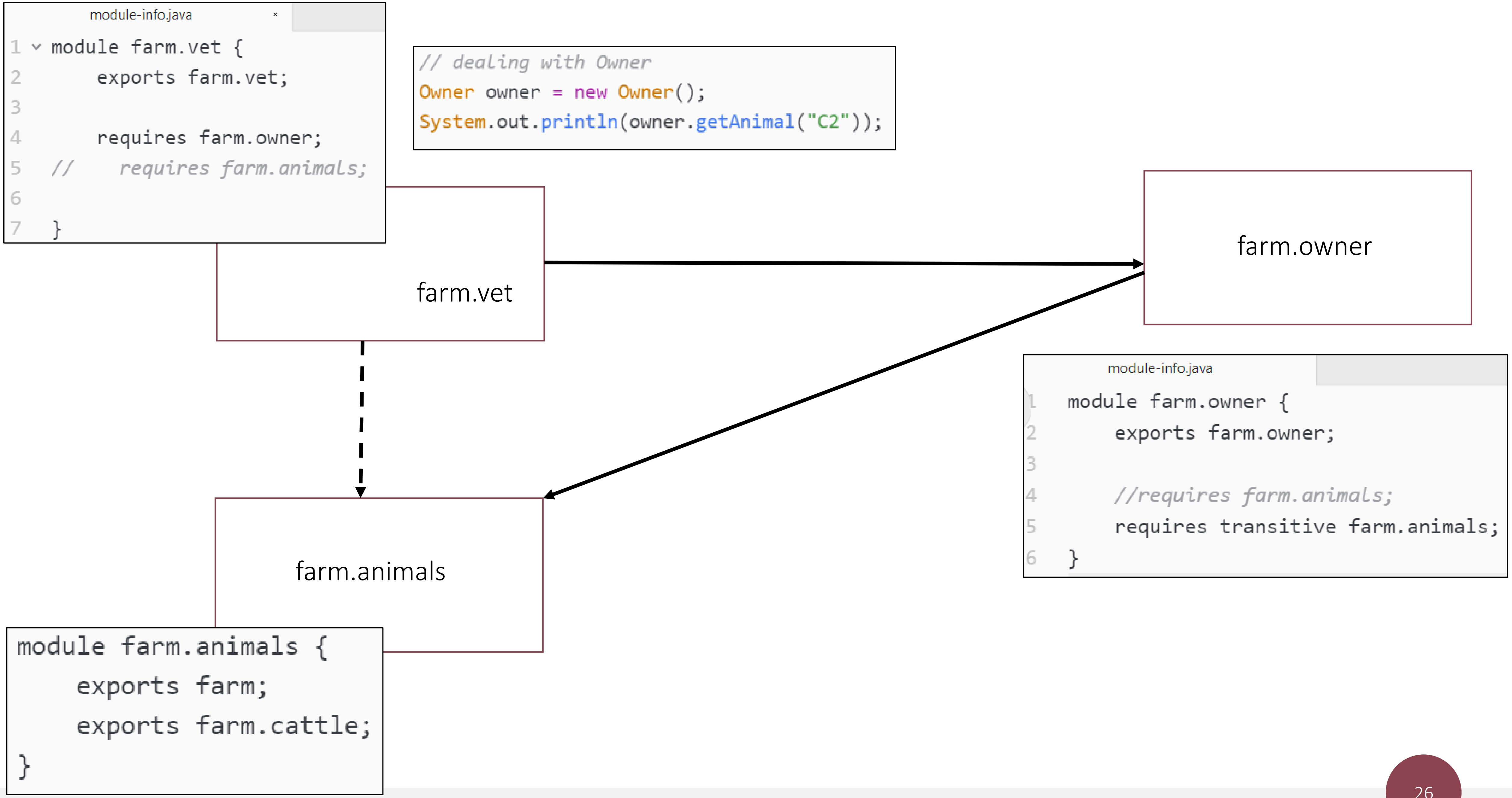
Services

- Modules are tightly coupled due to the use of *requires*, *exports* and the knowledge and creation of specific types.

- Services enable the decoupling of modules.

- Rather than modules directly coupling to other modules, there is now a middle layer, the service registry which abstracts the consumers from the producers.

- The *service (interface) contract*, which is hosted by the registry, is what binds the consumers and producers together.

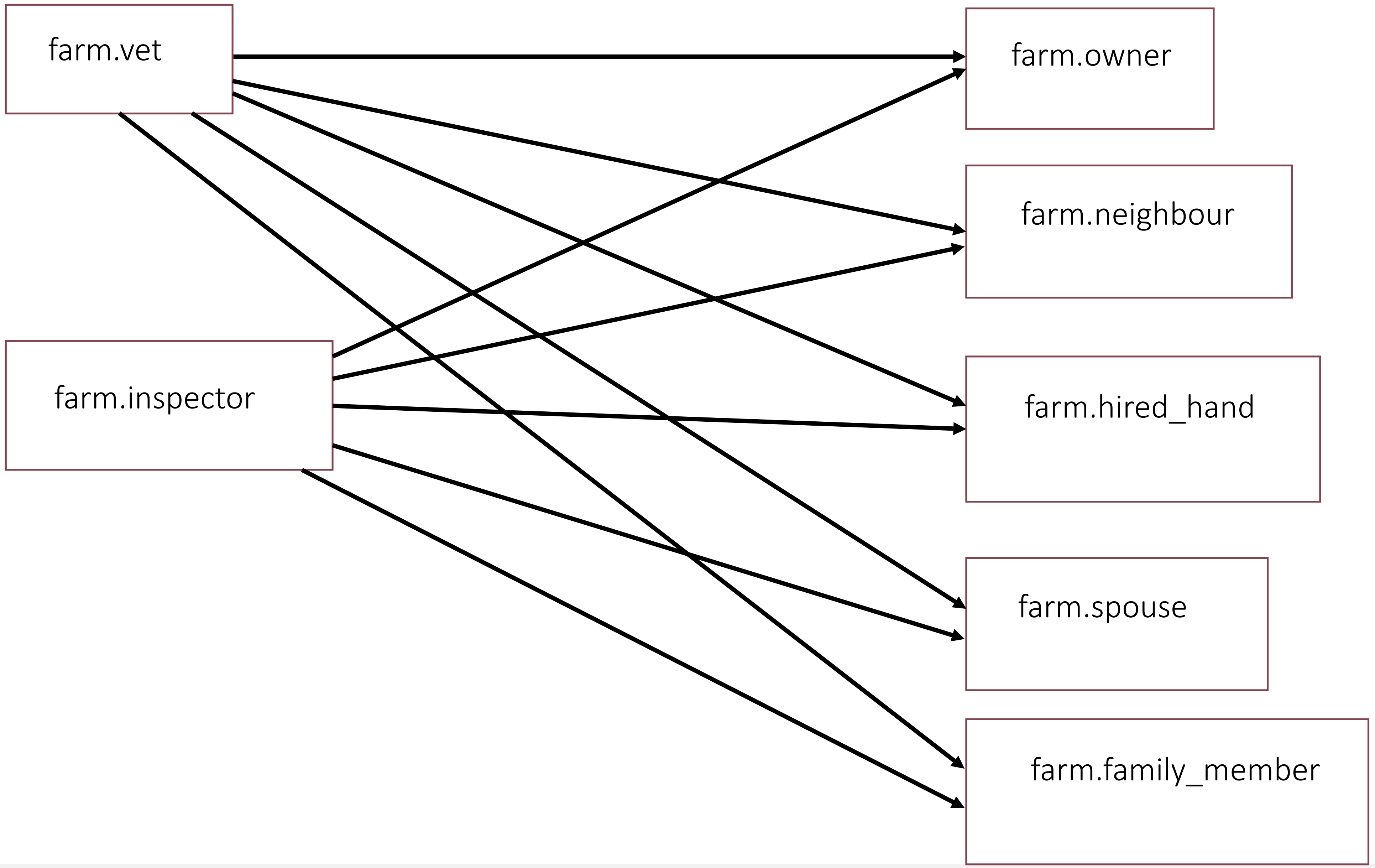


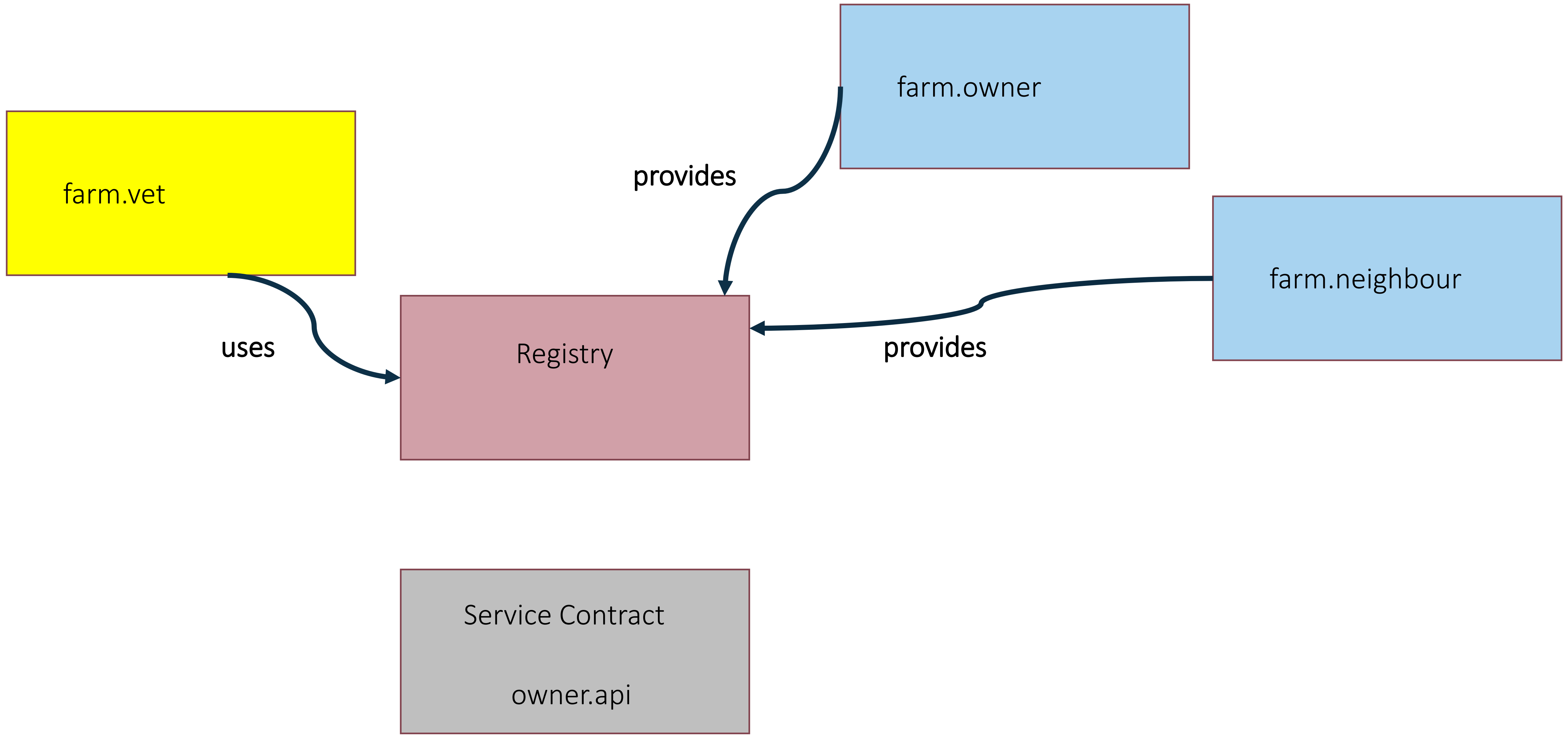
```
module farm.vet {
      exports farm.vet;

      requires farm.owner;
//     requires farm.animals;

}
```

```
// dealing with Owner
Owner owner = new Owner();
System.out.println(owner.getAnimal("C2"));
```

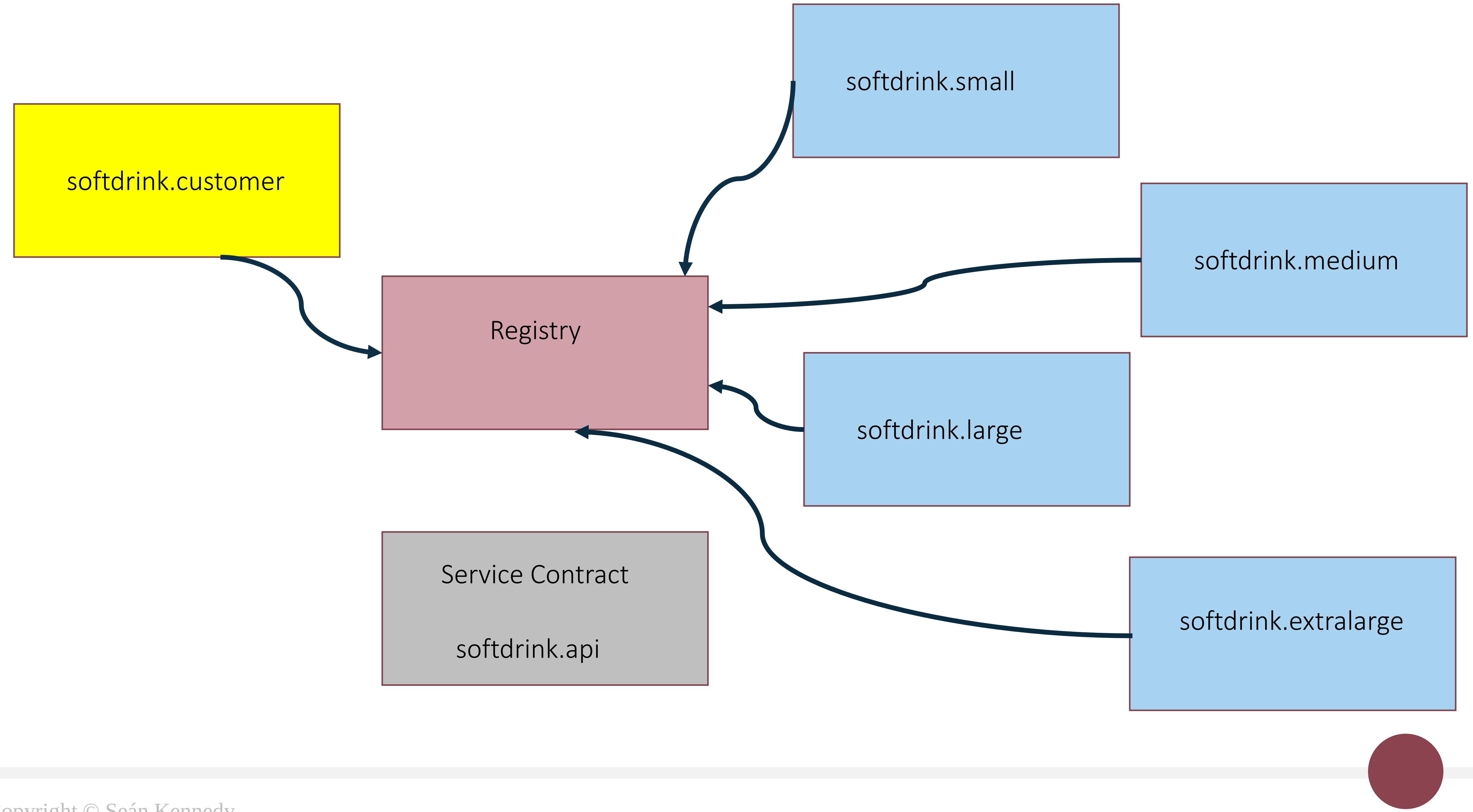
```
module farm.owner {
     exports farm.owner;

     //requires farm.animals;
     requires transitive farm.animals;
}
```

```
module farm.animals {
    exports farm;
    exports farm.cattle;
}
```


farm.vet
farm.inspector
farm.owner
farm.neighbour
farm.hired_hand
farm.spouse
farm.family_member

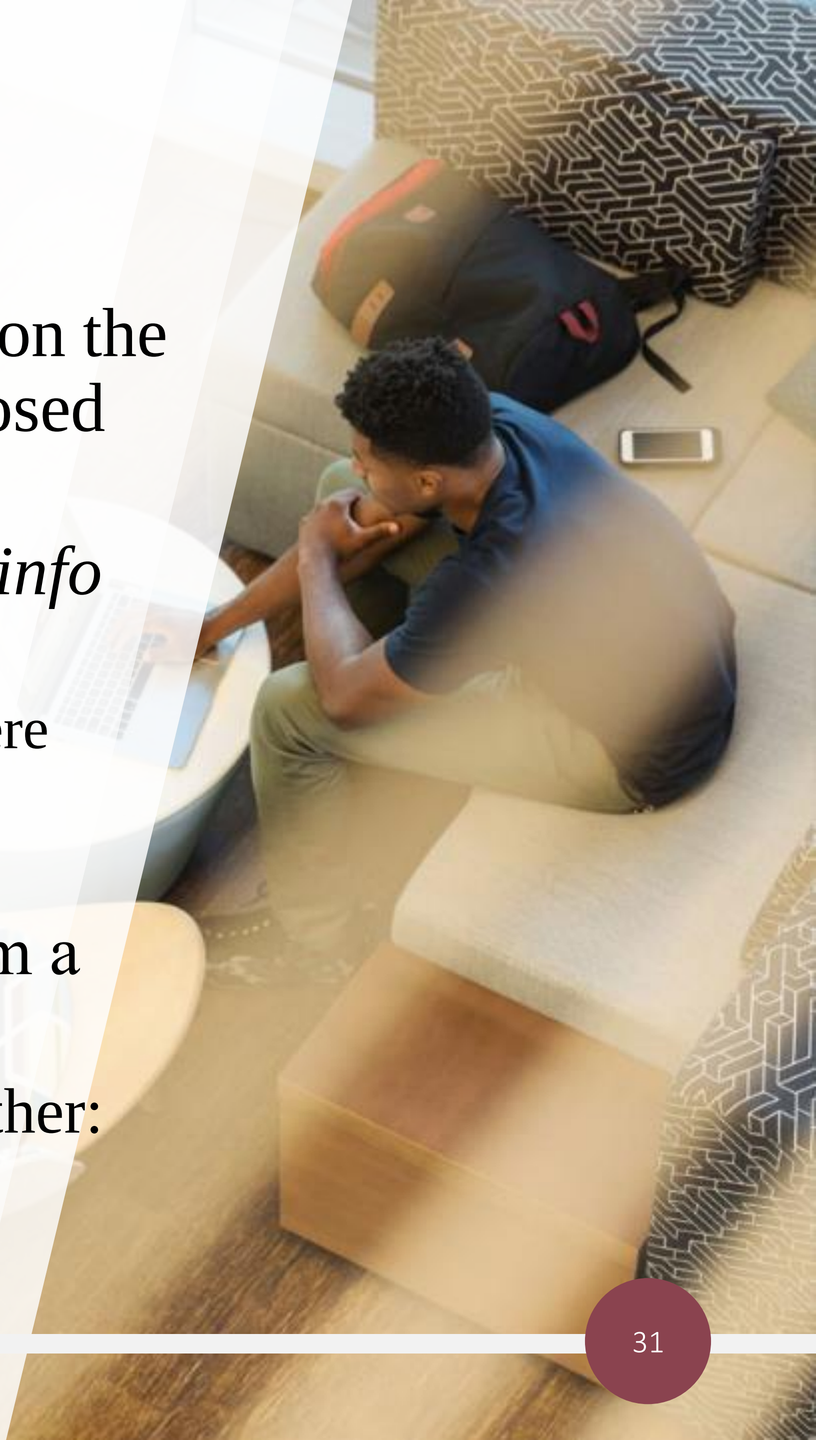

farm.owner
provides
farm.vet
farm.neighbour
uses
Registry
provides
Service Contract
owner.api

OwnerService example


softdrink.customer
Registry
softdrink.small
softdrink.medium
softdrink.large
softdrink.extralarge
Service Contract
softdrink.api

Types of Modules

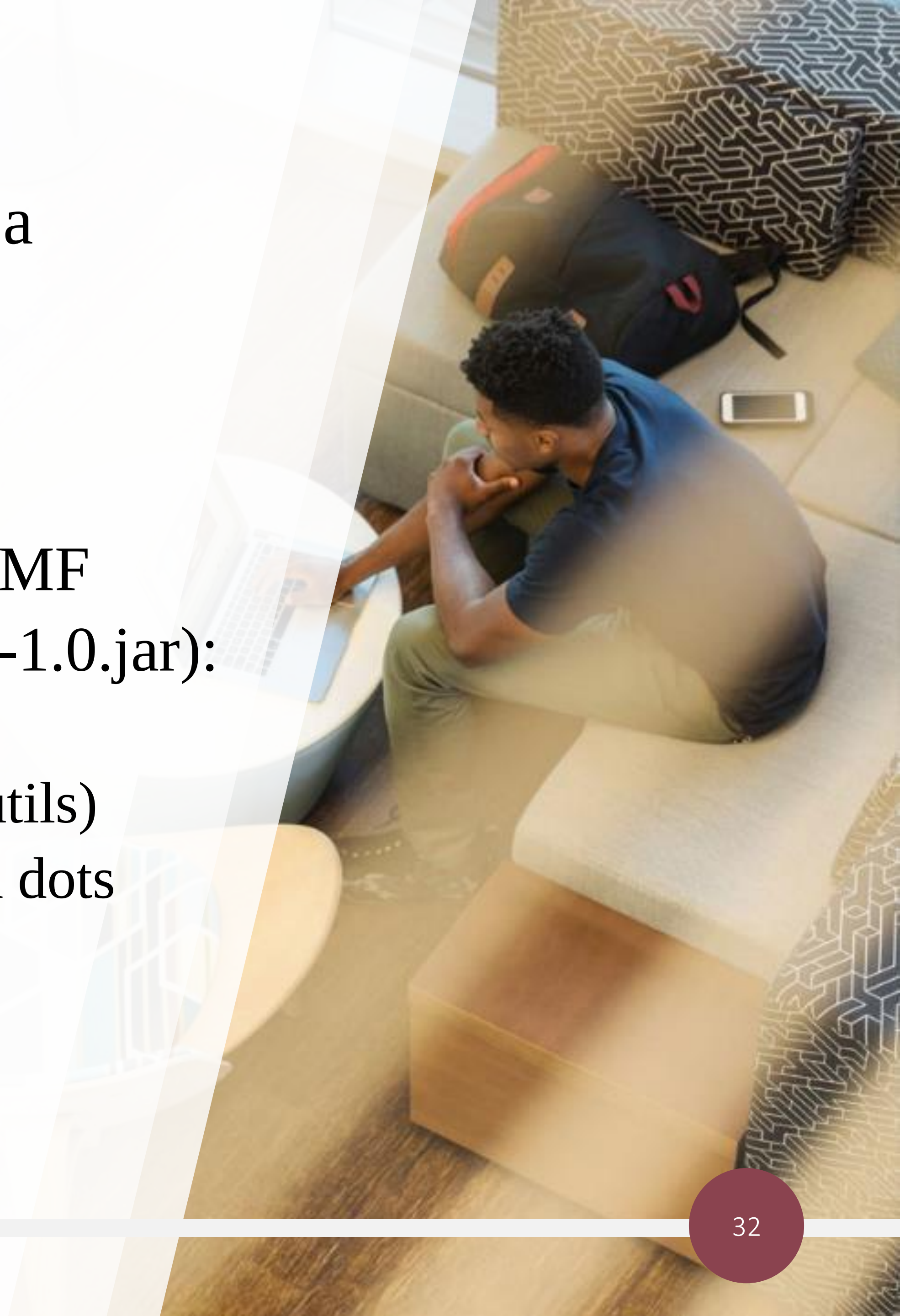
- There are 3 types of modules:
  - Named modules – contain a *module-info* file and appear on the module path, not the classpath i.e. --*module-path* as opposed to --*class-path*.
  - Unnamed modules – non-modular JAR's i.e. no *module-info* file (if present, it is ignored) and appear on the <u>classpath</u>.
    - think pre-modules i.e. the way Java worked before modules were introduced.
    - classpath code can read module path code but not vice vearsa.
  - Automatic modules – non-modular JAR's (typically from a 3rd party) that are on the <u>module path</u>
    - Java automatically creates named modules for them by either:
      - *Automatic-Module-Name* property in MANIFEST.MF
      - otherwise, a set of naming rules are applied



Automatic Modules

- Required when a modular application encounters a non-modular JAR on the module path.

- Rules for naming them:
  - *Automatic-Module-Name* property in MANIFEST.MF
  - otherwise, the following rules apply (e.g. text-utils-1.0.jar):
    - extension is dropped (text-utils-1.0)
    - version information at end of file is dropped (text-utils)
    - replace characters (other than letters/numbers) with dots (text.utils)
    - replace any sequence of dots with single dots
    - remove single dots from beginning and end



Automatic Modules

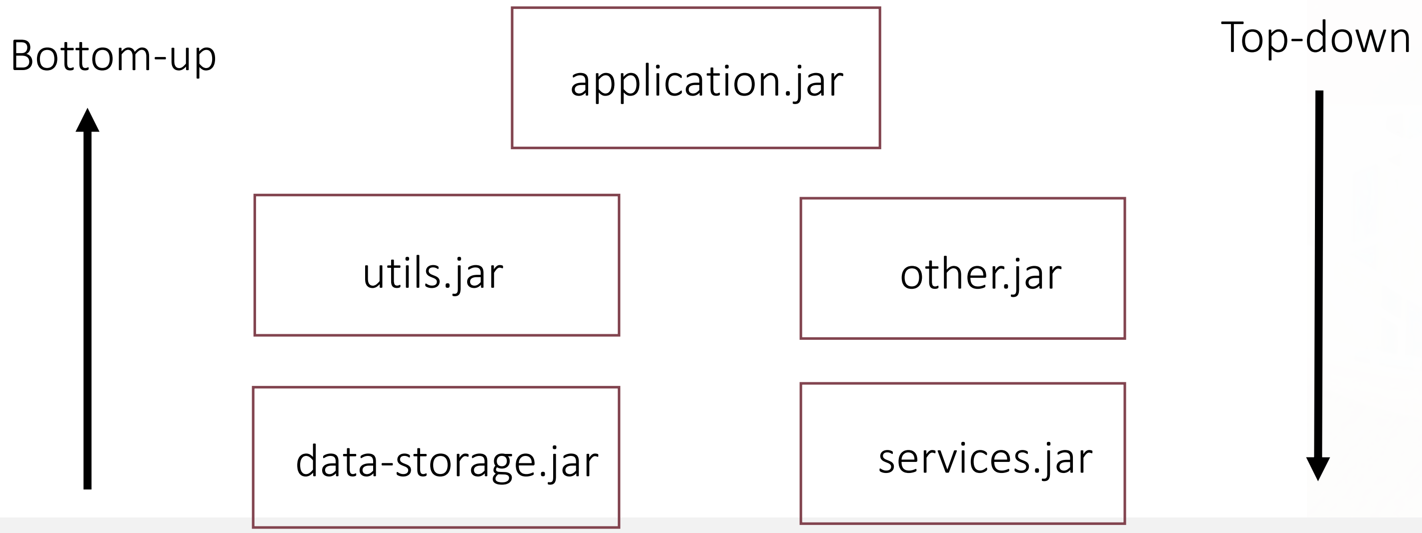
- All the packages in the automatic module are exported.

- The modular application (which is using the automatic module), will need to state that it *requires* the automatic module in its module-info file. This is why the naming rules are important.

Migration Strategies

- What if you want to migrate your classpath-based applications to modules?

- There are 2 strategies that you can follow:
  - Bottom-up
  - Top-down



Bottom-up

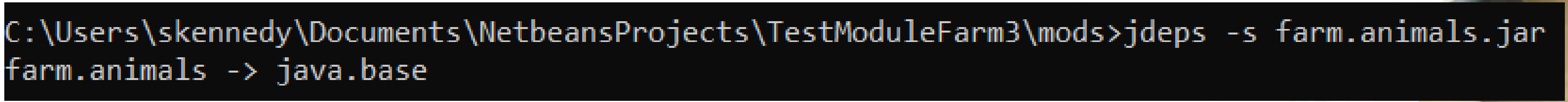
- Turning JAR files on the classpath into modules that we can put on the module path.

- Algorithm:
  1. Start at the “leaf” nodes i.e. nodes that have no dependencies other than the JDK.
  2. Create a *module-info.java* file for the JAR. Name the module. Ensure all packages required by other (higher-level) JAR’s are exported using the *exports* directive. In addition, all modules this JAR depends upon must have *requires* directives.
  3. Once migrated, this newly named module moves from the classpath to the module path.
  4. Repeat with the next lowest JAR until you finally have modularised the top level JAR i.e. you are done.

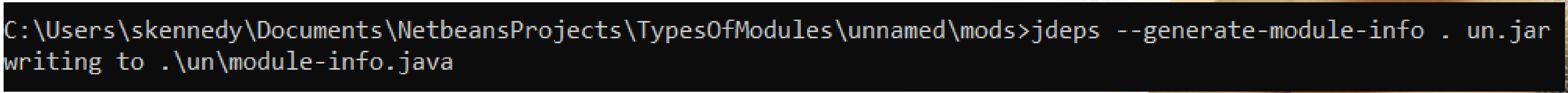
Bottom-up – using *jdeps*

- *jdeps* can be used to automatically generate the *module-info.java* file

- So, rather than having to analyse all the code to check for all dependencies, *jdeps* can do that for you.

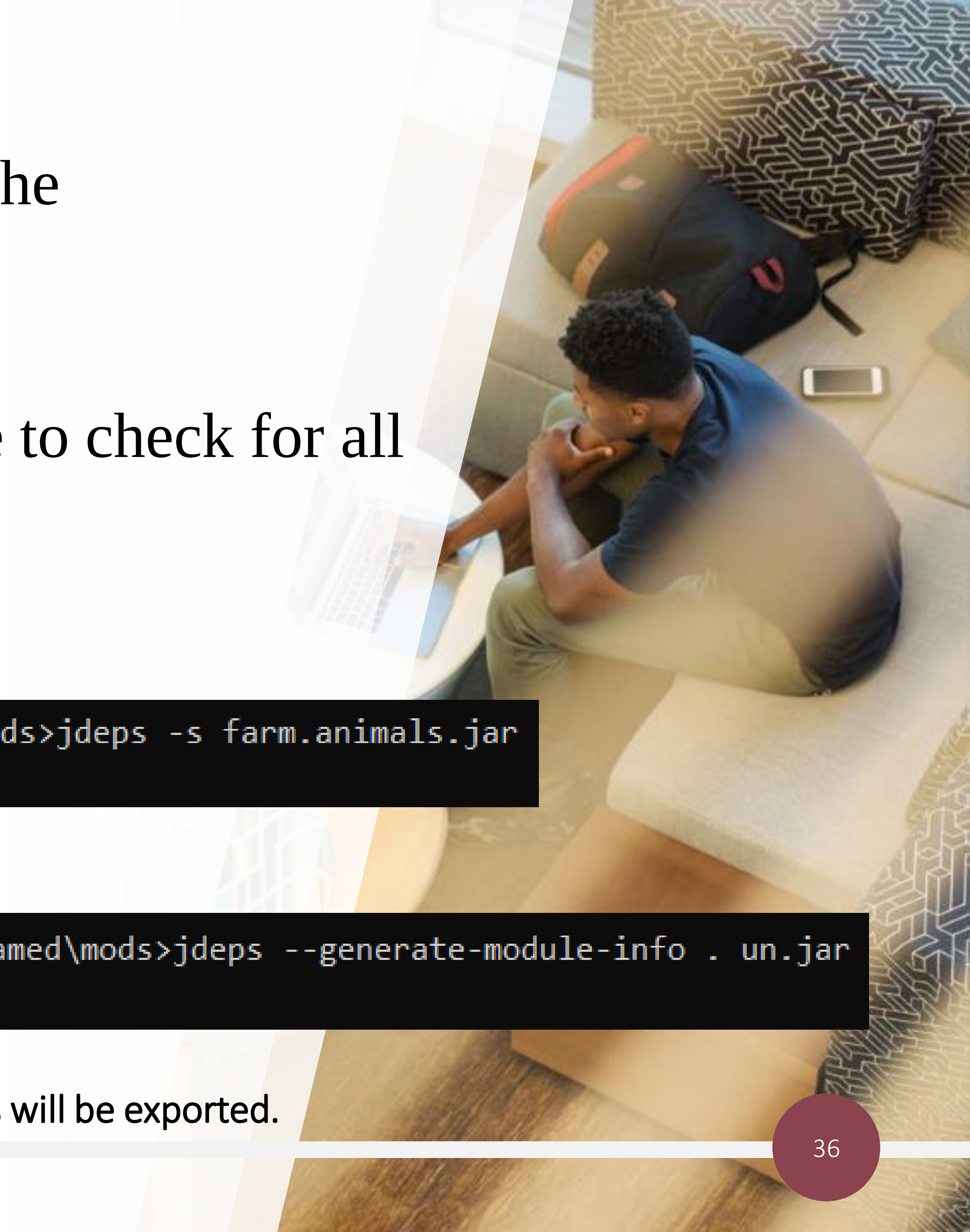


```
C:\Users\skennedy\Documents\NetbeansProjects\TestModuleFarm3\mods>jdeps -s farm.animals.jar
farm.animals -> java.base
```



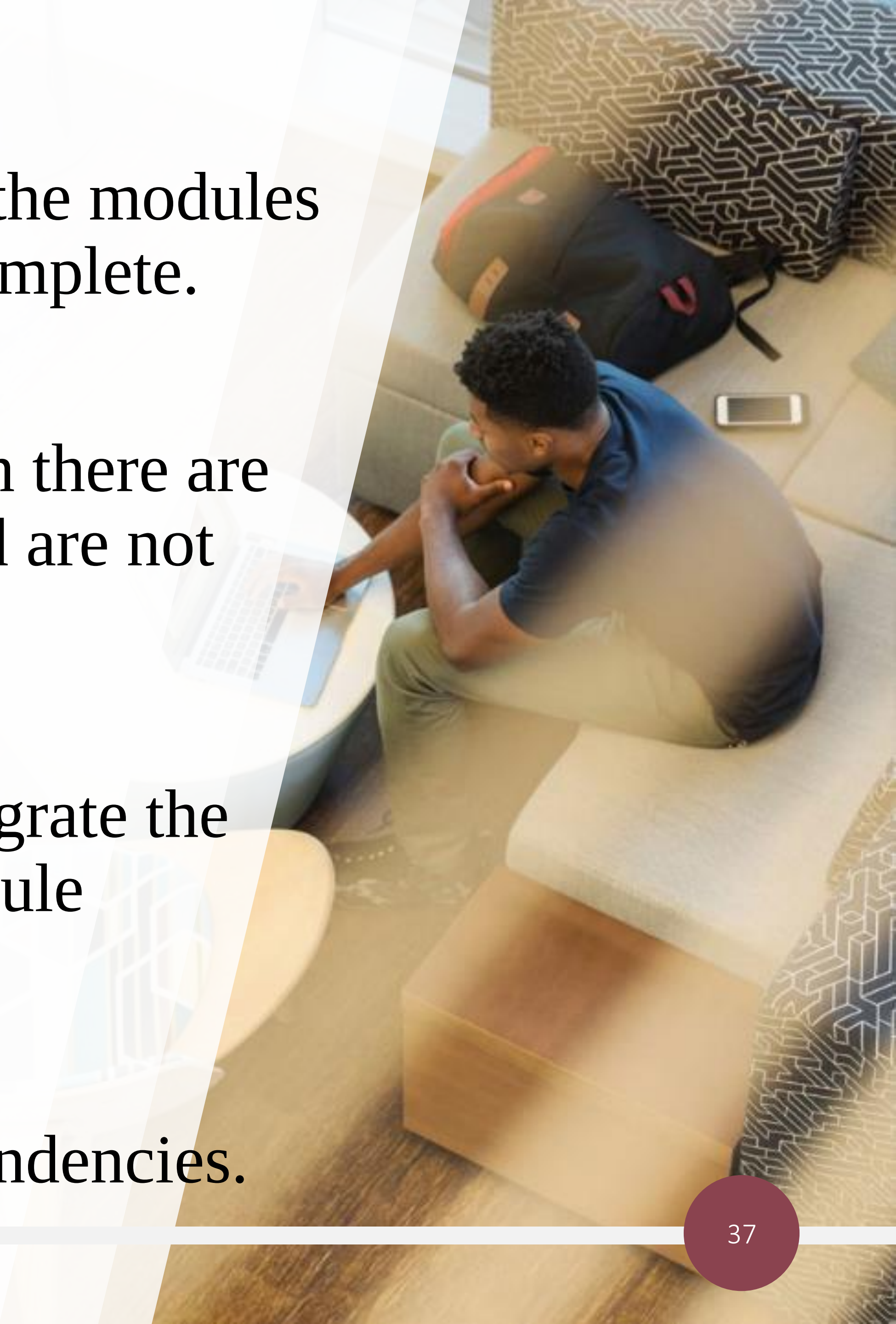
```
C:\Users\skennedy\Documents\NetbeansProjects\TypesOfModules\unnamed\mods>jdeps --generate-module-info . un.jar
writing to .\un\module-info.java
```

Careful! All packages will be exported.



Bottom-up

- The bottom-up approach yields benefits when all the modules are modularised and the top-level root module is complete.
- An issue arises with the bottom-up approach when there are external JAR's that we do not have control over and are not modularised yet.
- Do we wait for the external JAR developers to migrate the unnamed modules or do we try to maintain the module descriptors ourselves? Neither option is ideal.
- Constrained by the weakest link in the set of dependencies.



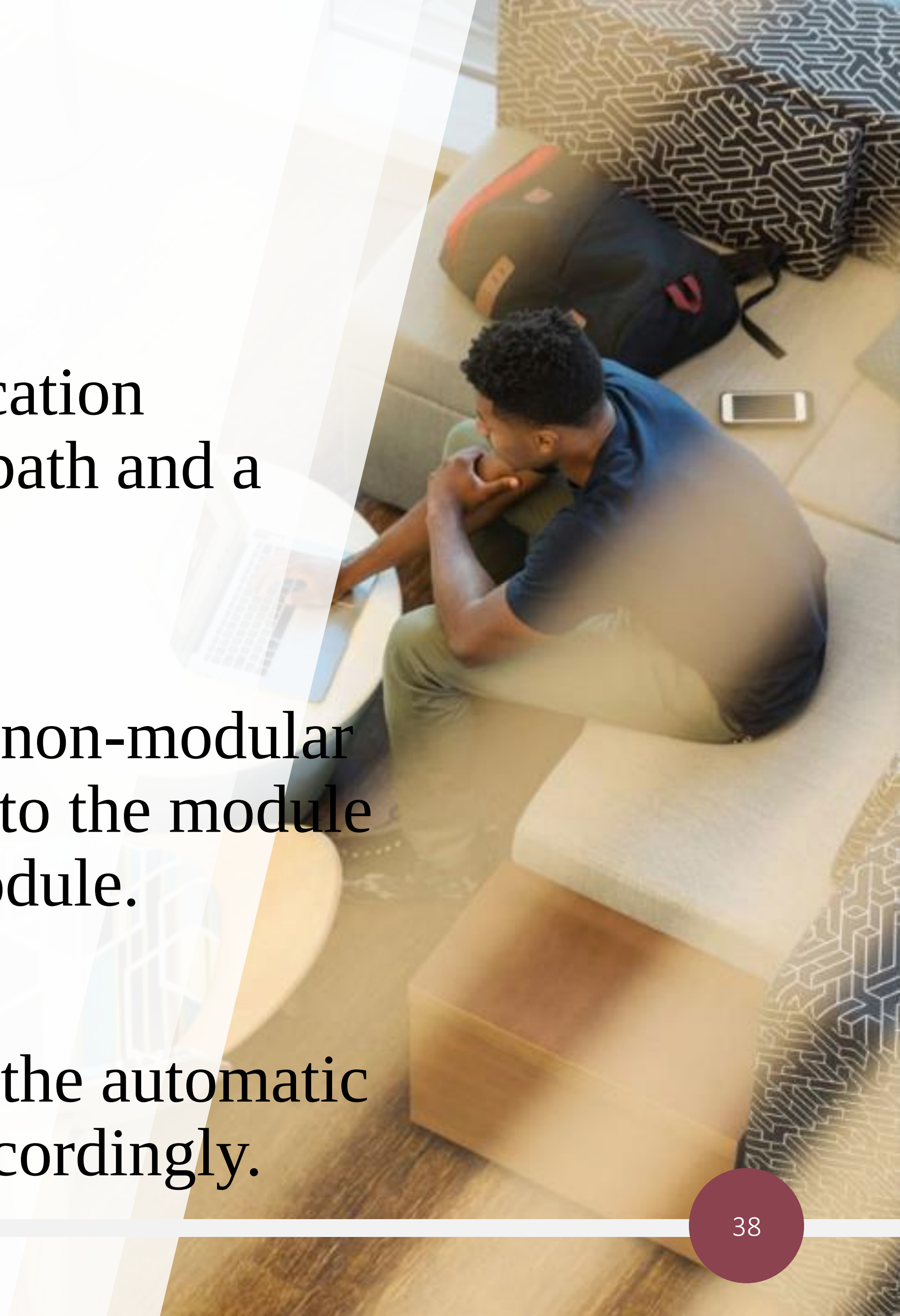
Top-down

- Proceeds in a top-down fashion from the root.

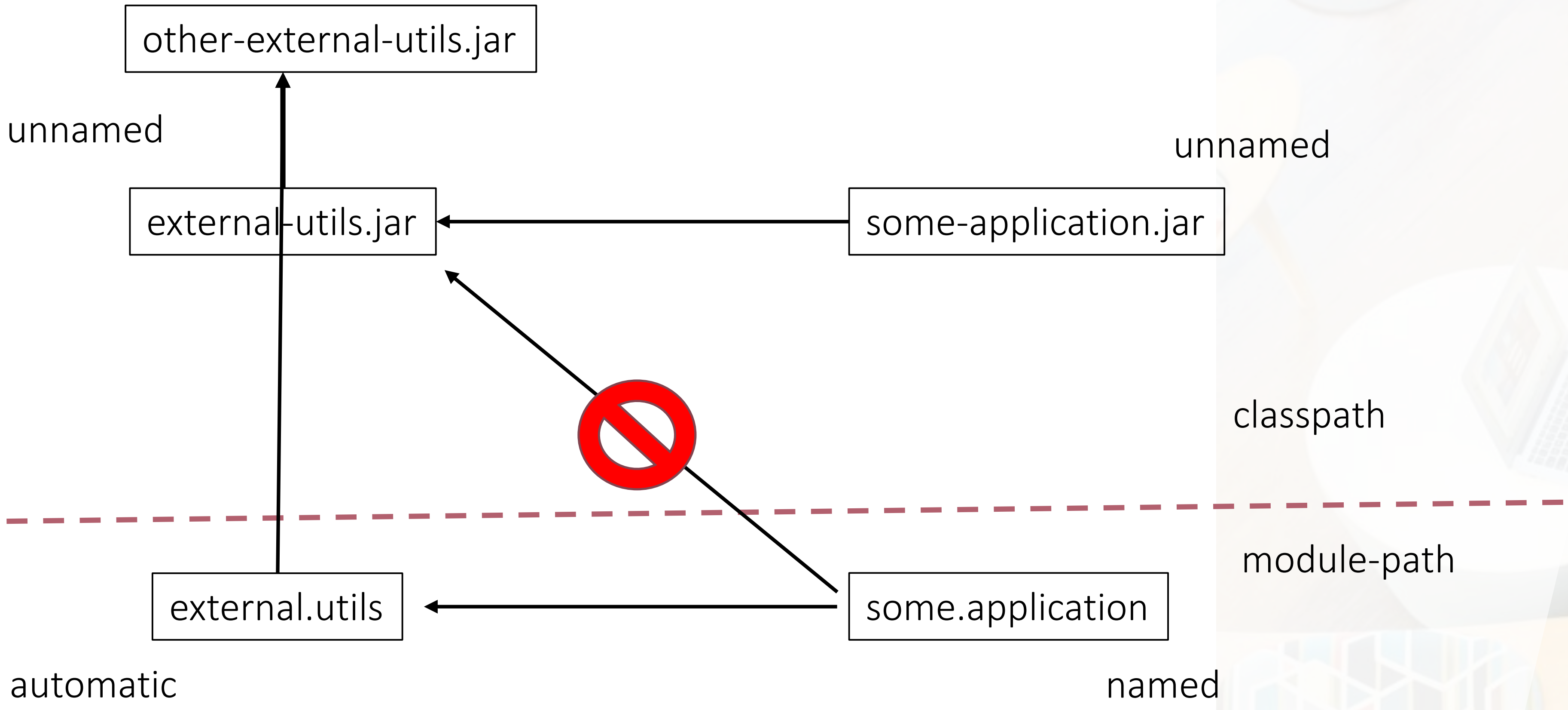
- Uses automatic modules - where a modular application encounters a non-modular JAR file on the module path and a module is generated for it automatically .

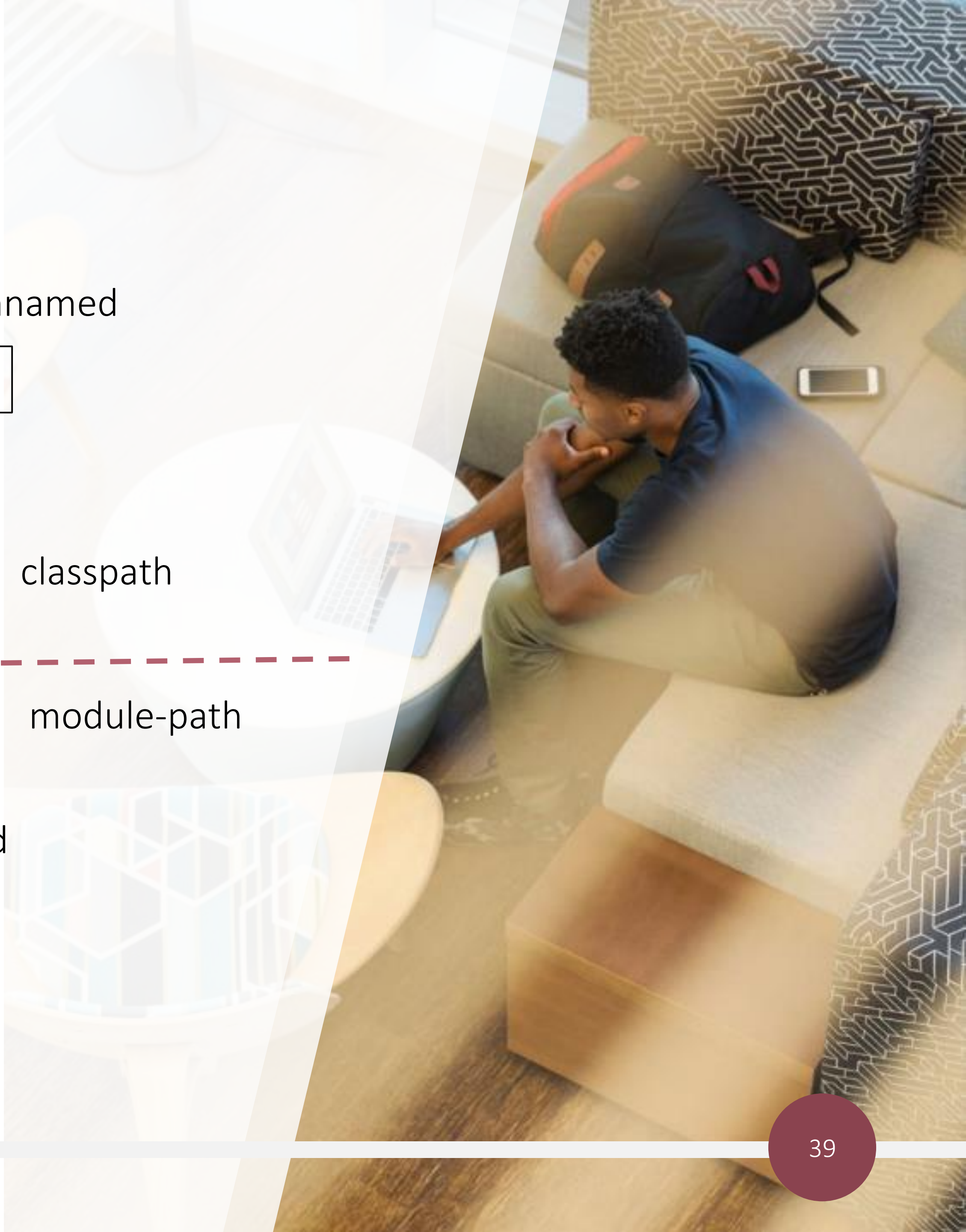
- Thus, we can simply move the unaltered external non-modular (i.e. unnamed) module/JAR file from the classpath to the module path; where it will be converted to an automatic module.

- All we need to know is what name Java will give the automatic module and modify our dependencies (*requires*) accordingly.

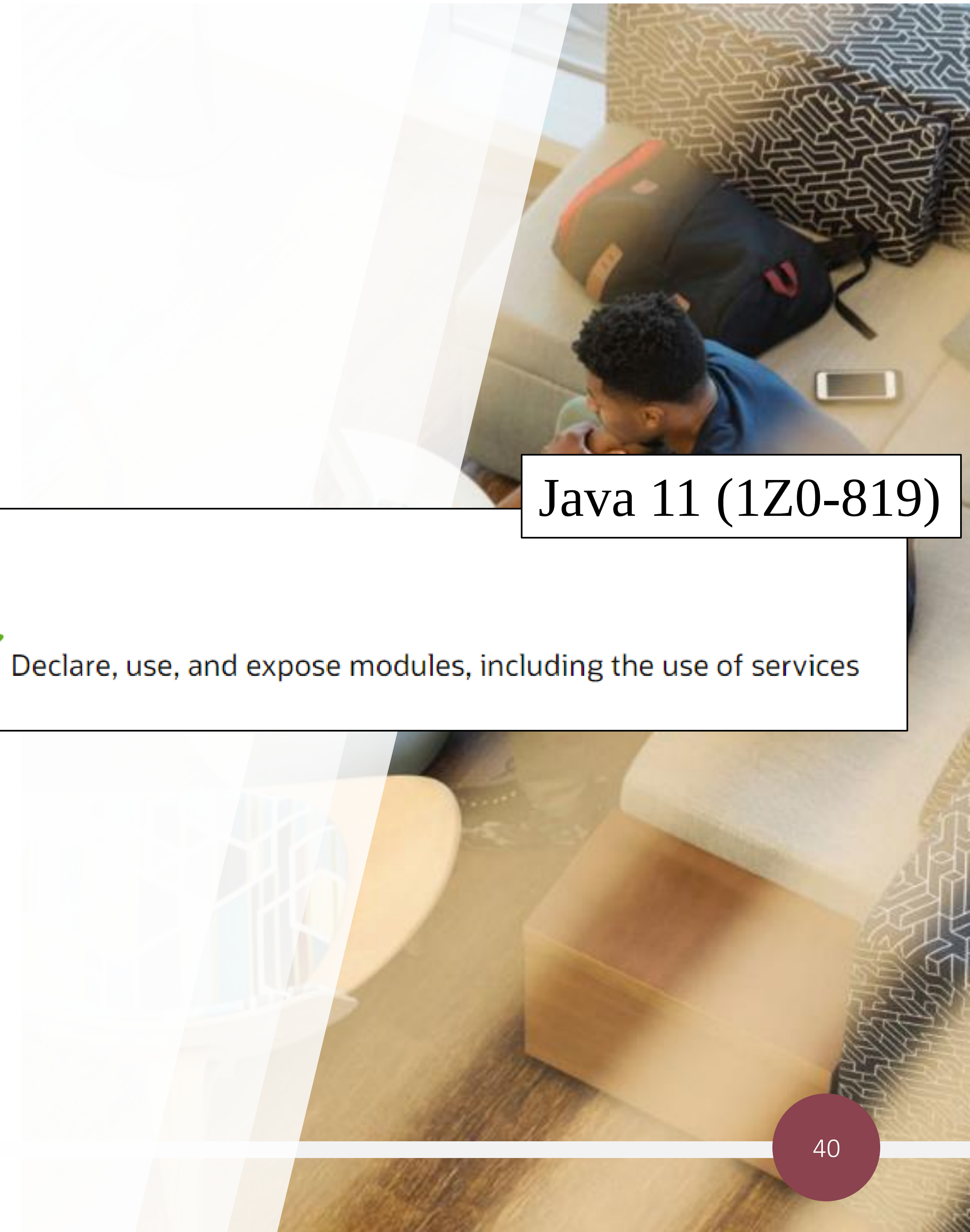


Top-down


other-external-utils.jar
unnamed
unnamed
external-utils.jar
some-application.jar
classpath
module-path
external.utils
some.application
automatic
named



Modules



Java 11 (1Z0-819)

Java Platform Module System

✓ Deploy and execute modular applications, including automatic modules

✓ Declare, use, and expose modules, including the use of services