Java SE 9

Modules

LUXOFT TRAINING

LXFT LISTED NYSE

Tools

- JDK 9 (Java Development Kit)
- IntelliJ IDEA

JDK 8 - about the code.

JDK 9 - about **deployment**.

Motivation

Project Jigsaw aims to design and implement a standard module system for the Java SE Platform and to apply that system to the Platform itself, and to the JDK.

Its primary goals are to make implementations of the Platform more easily scalable down to small devices, improve security and maintainability, enable improved application performance, and provide developers with better tools for programming in the large.

Motivation

Primary goals of project Jigsaw:

- Make it easier for developers to construct and maintain libraries and large applications
- Improve the security and maintainability of Java SE Platform implementations in general, and the JDK in particular
- Enable improved application performance
- Enable the Java SE Platform, and the JDK, to scale down for use in small computing devices and dense cloud deployments
- Work on Project Jigsaw began in August 2008 with an initial exploratory phase. Work on the design and implementation for Java 9 began in 2014.

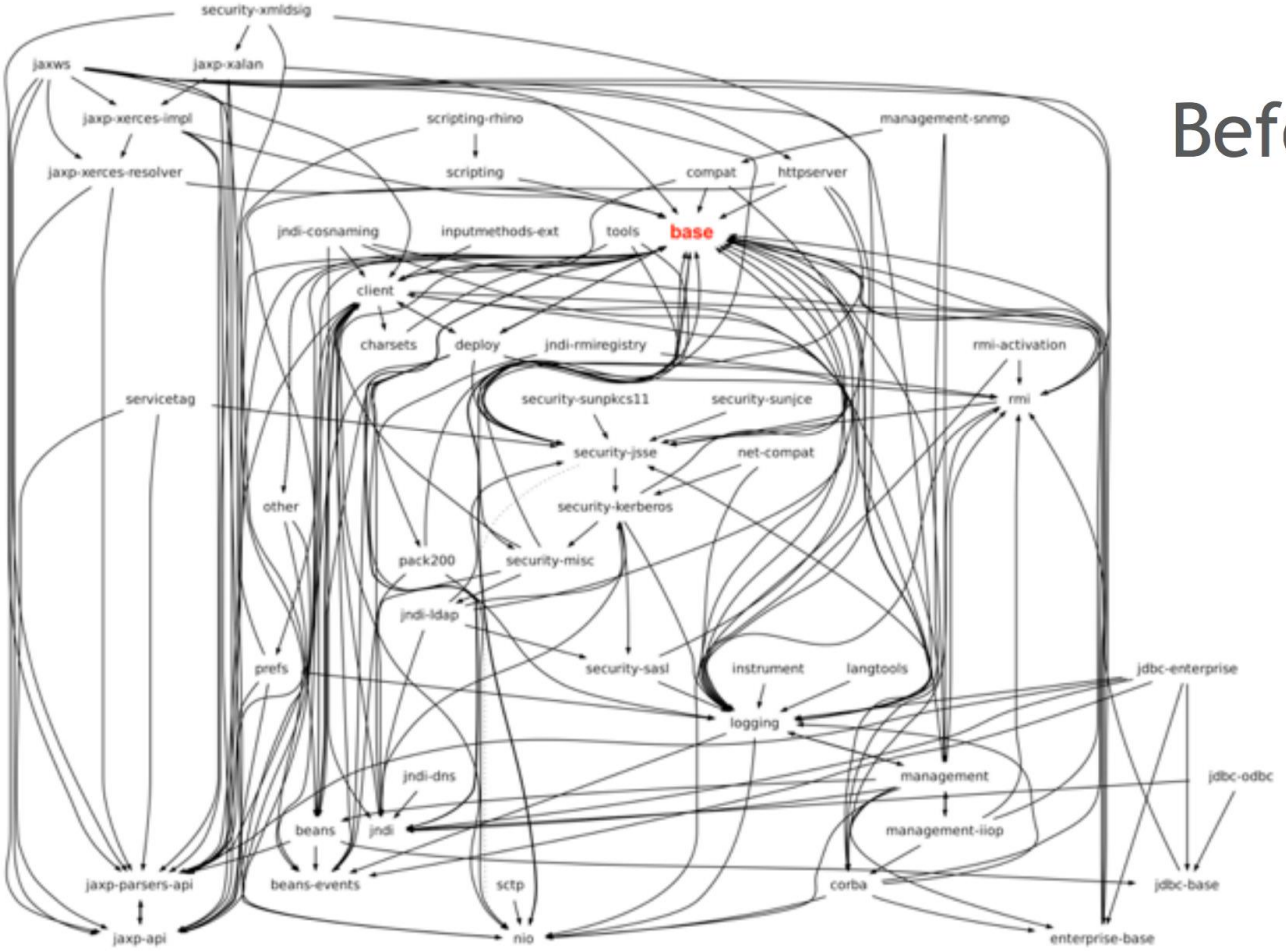
Motivation

- large jars (example rt.jar)

- lack of clarity on dependencies

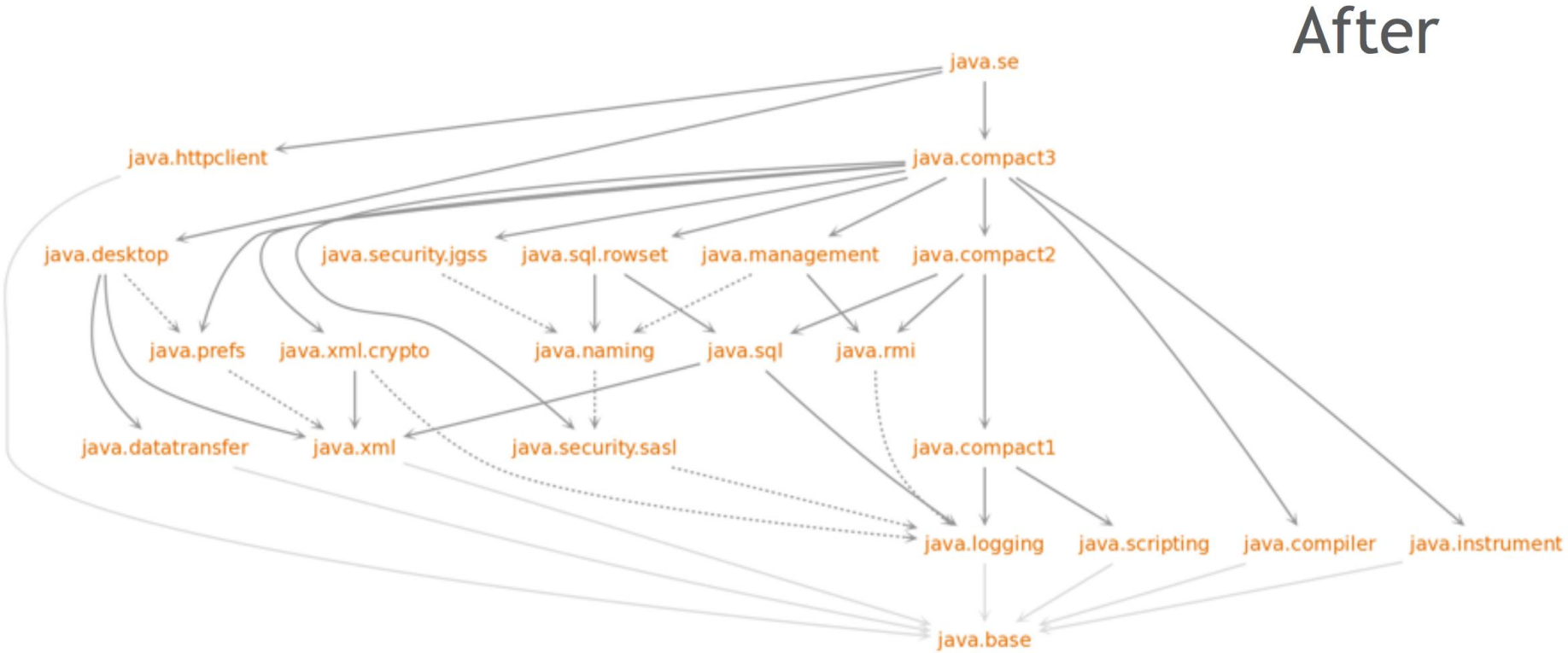
- sometimes public is too open

JDK 8



Before

JDK 9


After
java.se
java.httpclient
java.compact3
java.desktop
java.security.jgss
java.sql.rowset
java.management
java.compact2
java.prefs
java.xml.crypto
java.naming
java.sql
java.rmi
java.datatransfer
java.xml
java.security.sasl
java.compact1
java.logging
java.scripting
java.compiler
java.instrument
java.base

module - What is it?

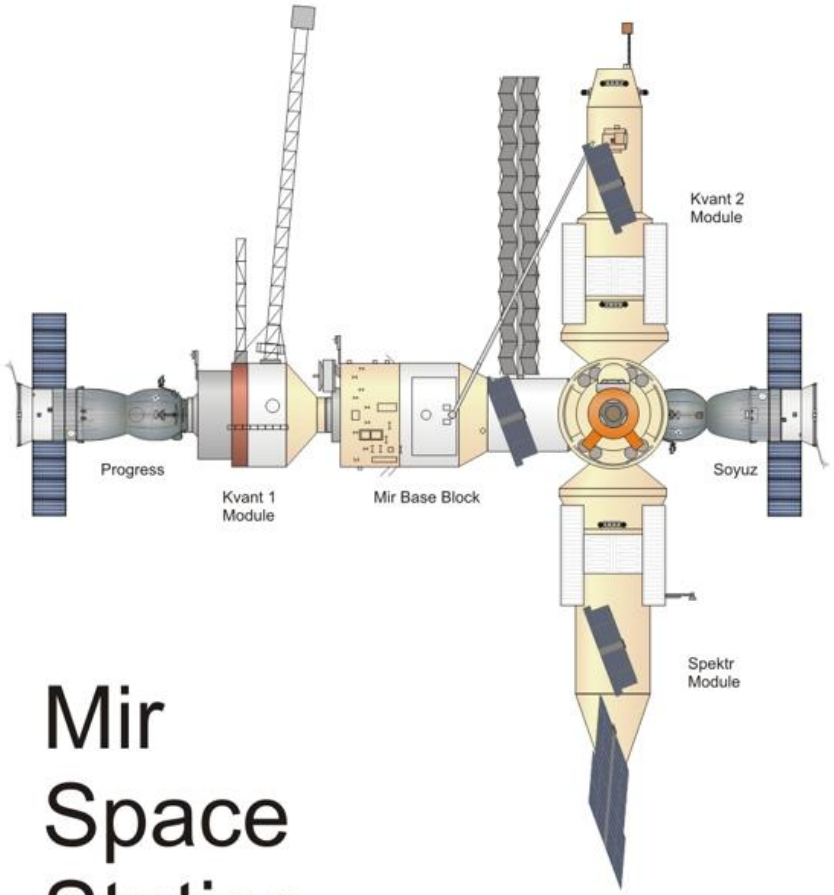
Module

Technically speaking, a module is the same concept as a jar, but more flexible in terms of access.

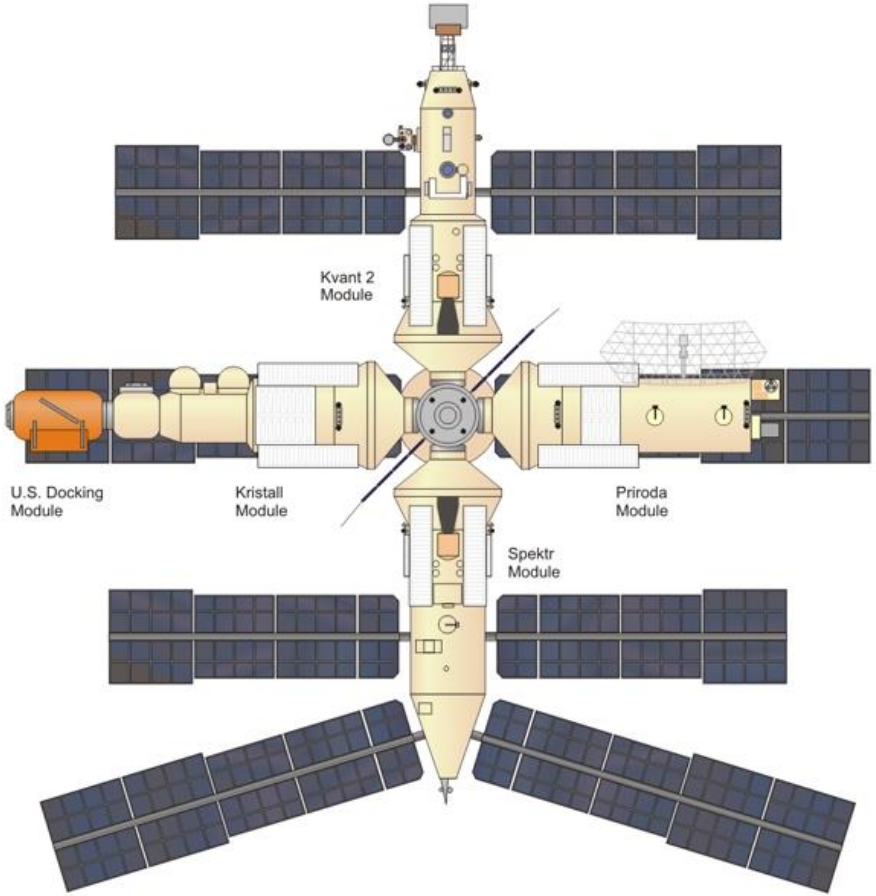
Before Java 9, we had access modifiers for each class. Now we also have access modifiers for each module.

Module

A module is a self-describing collection of code, data and some resources. It is a set of related packages, types (classes, abstract classes, interfaces) with code, data and resources.



Mir Space Station



HistoricSpacecraft.com

Accessibility JDK 1 - 8

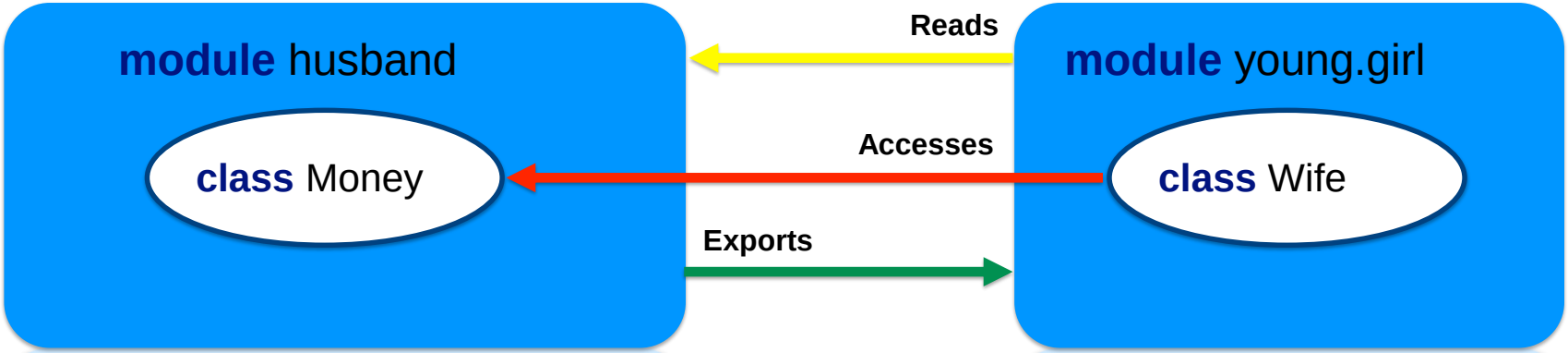
- private
- default (package private)
- protected
- public

Accessibility JDK 9

- private
- default (package private)
- protected
- **public only within a module**
- **public to specific modules**
- **public to everyone**

public - no longer means “**accessible**”

The Role of Readability




```
module husband
{
    exports com.man.love;
    exports com.money;
}
```

```
module young.girl
{
    requires husband;

    exports com.passion;
}
```

How to create a module?

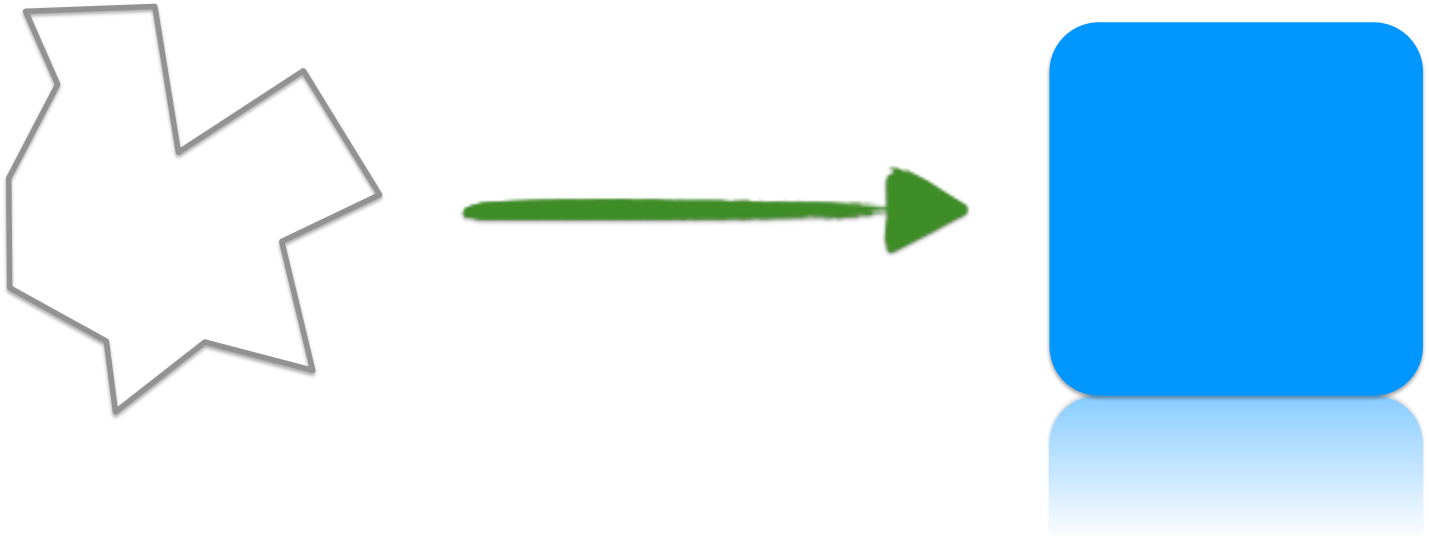
module-info.java

```
module husband
{
    exports com.man.love;
    exports com.money;
}
```

Enough theory. Will look at the examples from here.

1. How to use modular JDK.

Using a modular JDK



module-info.java

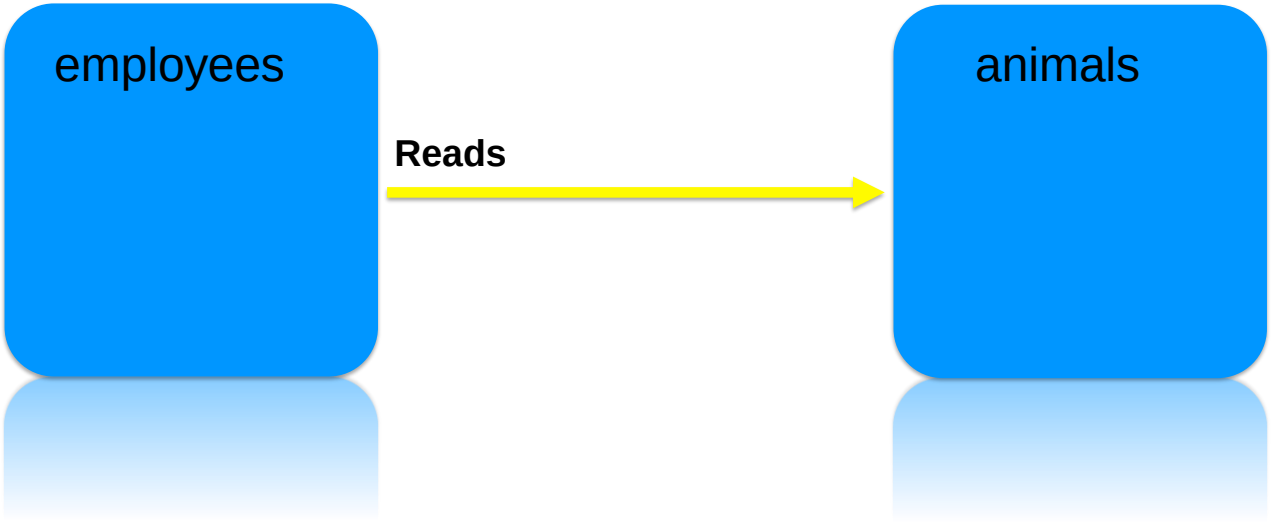
```
module dependent
{
    requires java.logging;
}
```

Task 1

Run your first Java 9 application. Project 1-first-steps

2. Relationship between two modules.

Relationship between modules



Relationship between modules

Define the list of modules required by your module using:

requires <module_name>;

Define the list of packages that your module exports using:

exports <package_name>;

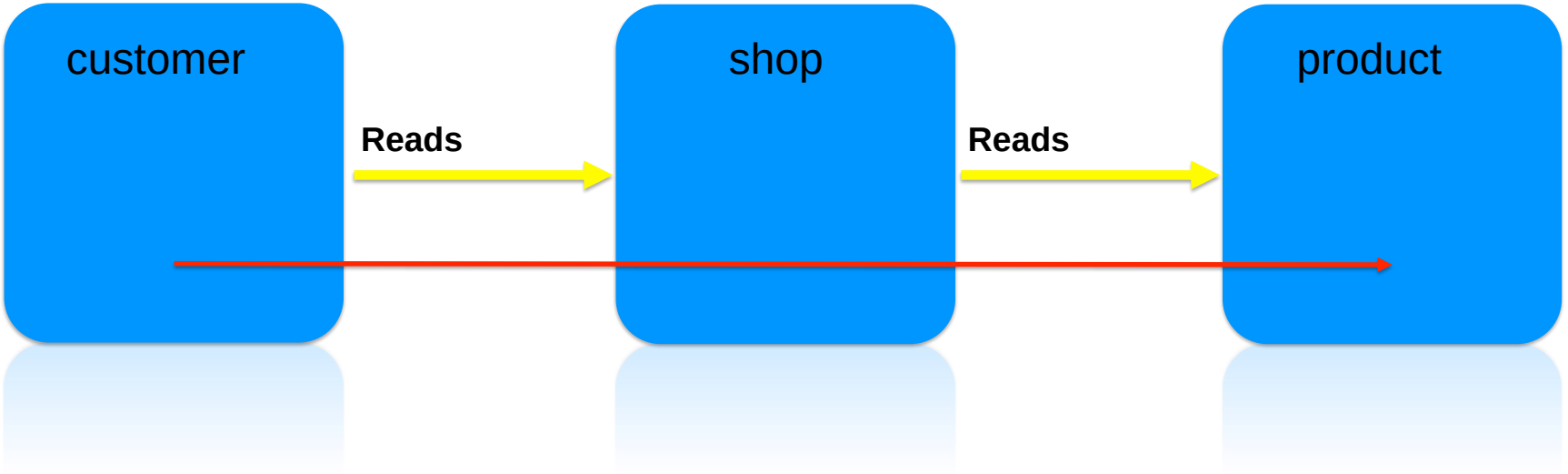
Limitations

You can require a package from one source only!

If two modules export a package with an identical name, the third module is allowed to require only one of the modules.

3. Relationship between three modules.

Relationship between modules



Will the customer read the product in this case?

Relationship between modules

requires transitive <module_name>;

Allows a module to also export modules it requires.

Task 2

Create an application with several modules.

Project 2-world-of-fun

4. Read the implementation.

Read the implementation

Will dealer **have access to implementations if** storage **does not export any?**

5. Services

Load implementation

In this case consumer **can get access to implementation via interface but you should also have** Provider **that returns the needed implementation?**

Load implementation

java.util.ServiceLoader - provides a standard way to load the **service** from another **module** or **jar**.

Service is represented by a single type, that is, a single interface or abstract class.

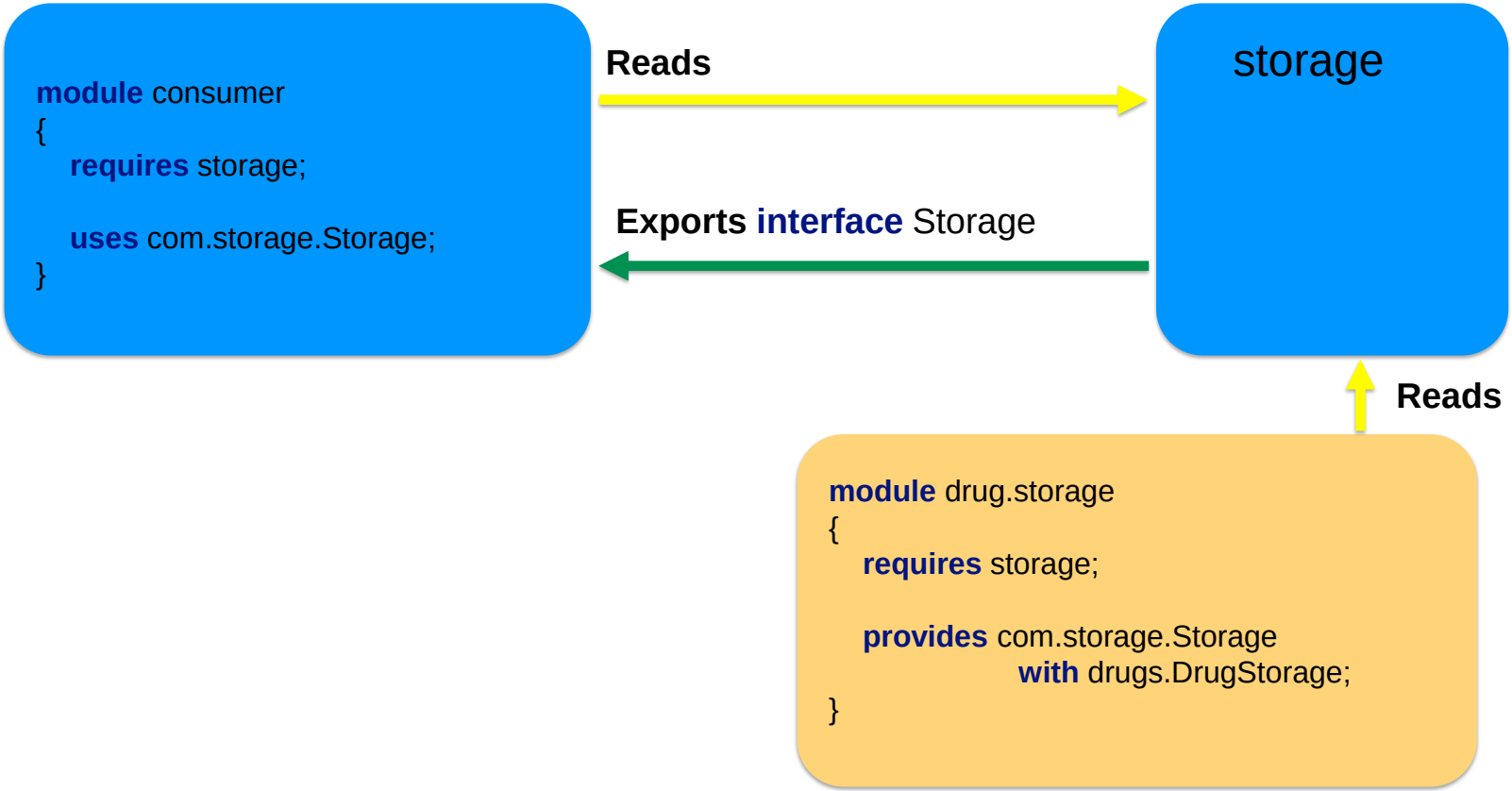
Load implementation

java.util.ServiceLoader - provides a standard way to load the **service** from another **module** or **jar**.

```
ServiceLoader<Storage> loader = ServiceLoader.load(Storage.class);
Optional<Storage> storage = loader.findFirst();

if (storage.isPresent())
{
    return storage.get().get(numberOfUnits);
}
```

Load implementation



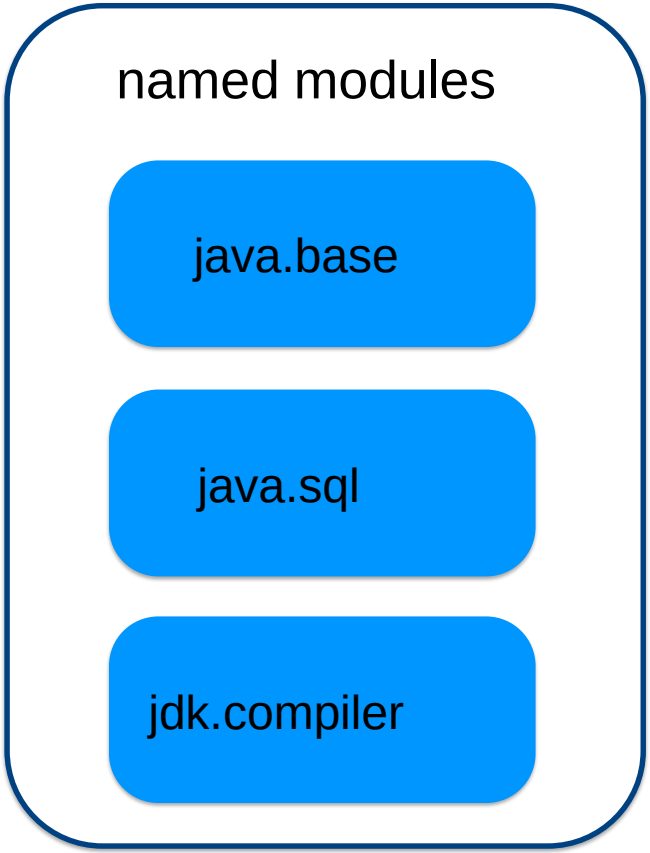
Task 3

Work with ServiceLoader

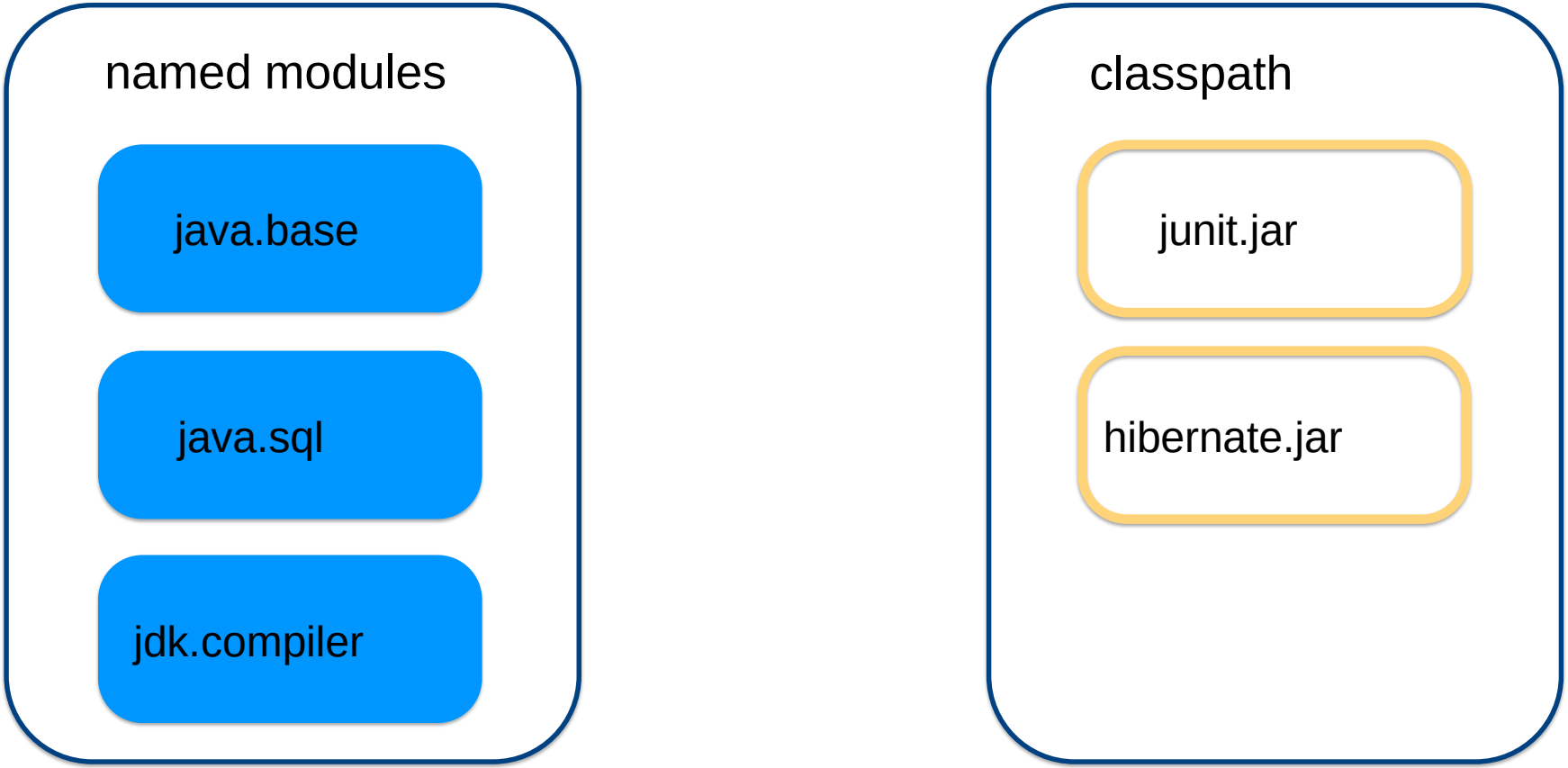
Project 3-storage

6. Unnamed modules

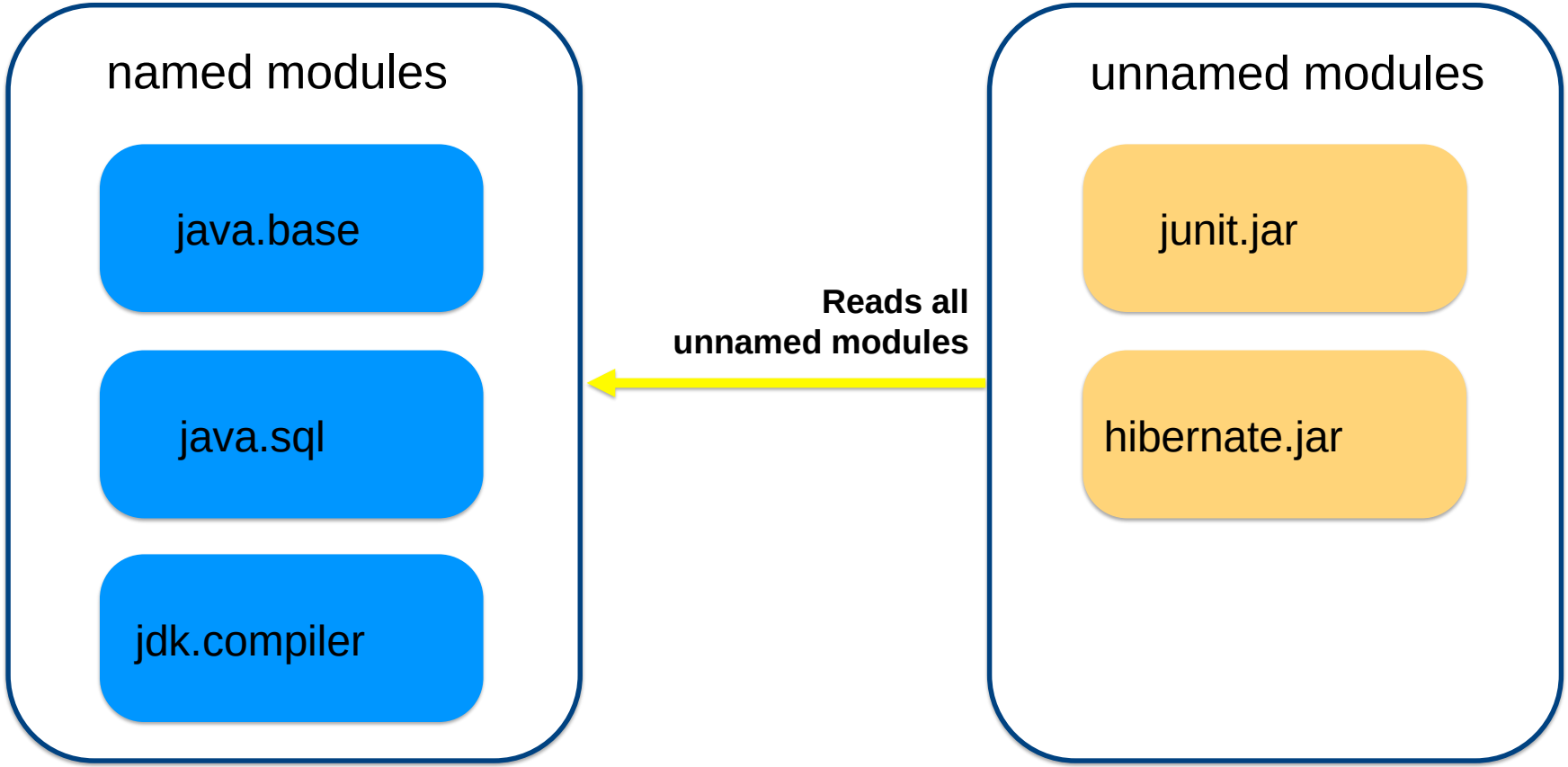
Unnamed modules



Modules before compilation

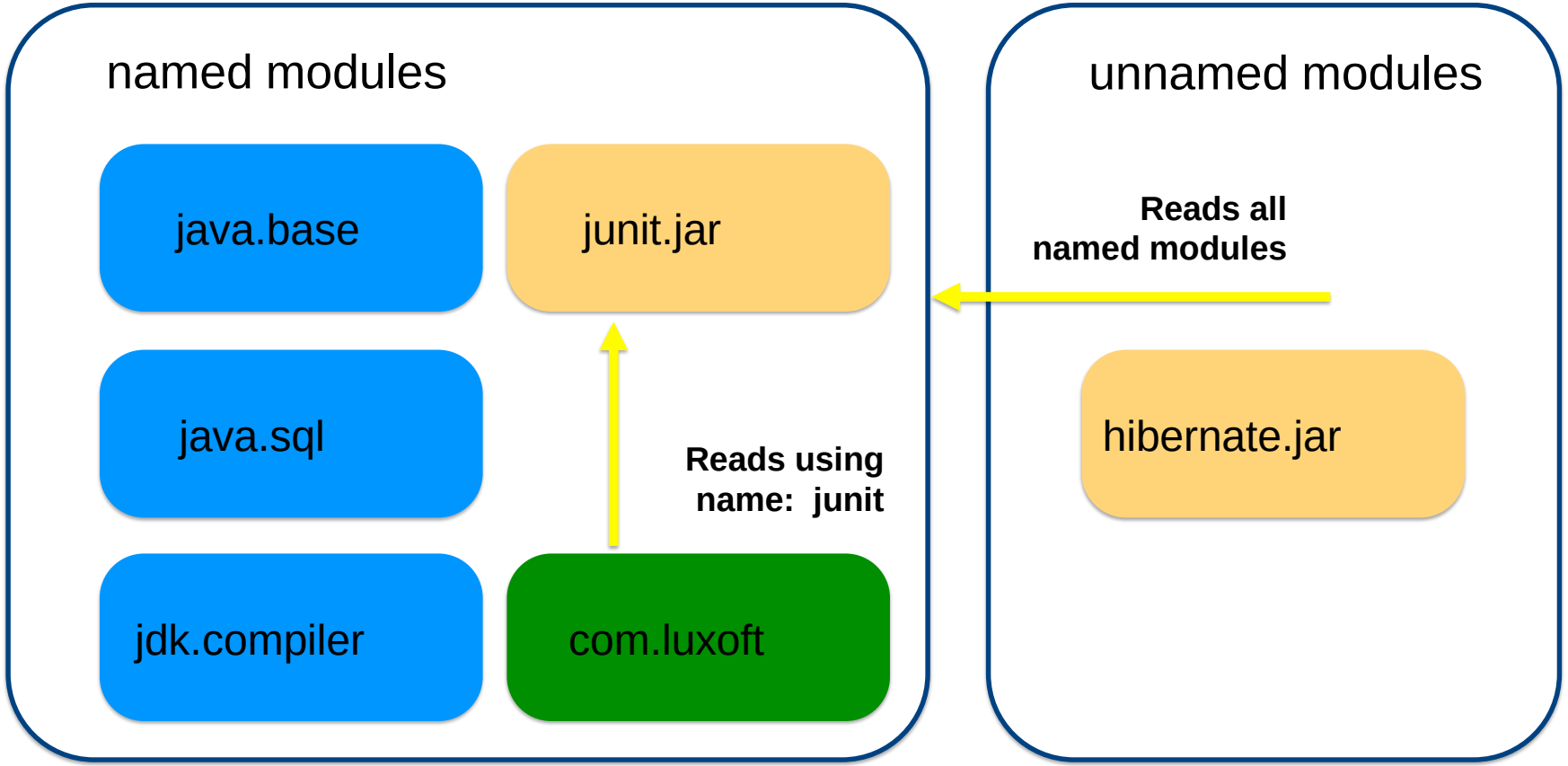


Modules after compilation



7. Automatic modules

Modules after compilation



Task 4

Add external jar as a module.

Project 4-automodule

8. jlink tool

jlink

Tool that can assemble and optimize a set of modules and their dependencies into a custom run-time image.

jlink

$ jlink --module-path <modulepath> --add-modules <modules> --output <path>

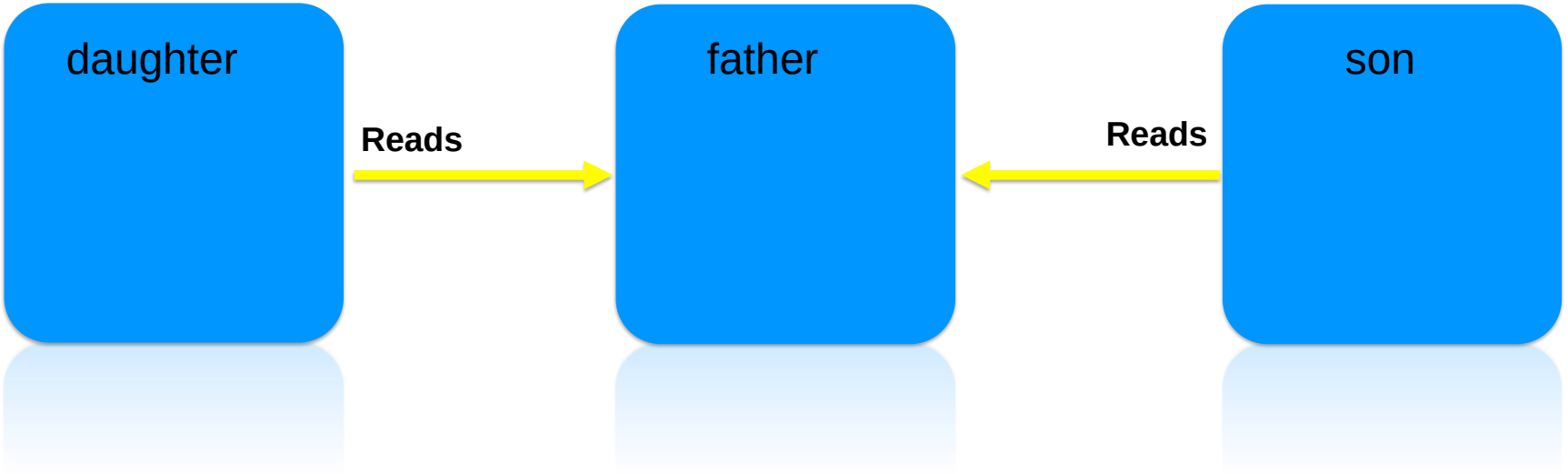
--module-path is the path where observable modules will be discovered by the linker; these can be modular JAR files, JMOD files, or exploded modules.

--add-modules names the modules to add to the run-time image; these modules can, via transitive dependencies, cause additional modules to be added.

--output is the directory that will contain the resulting run-time image.

9. Export package to specific modules only

Export to module



Export to module