11

Migrating to a Modular Application

Objectives

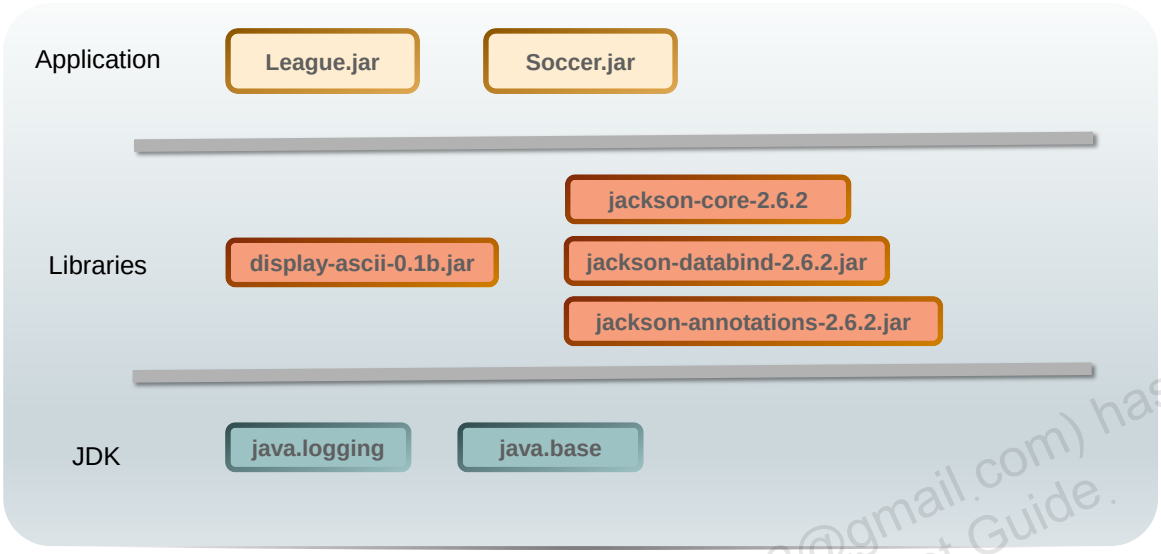
After completing this lesson, you should be able to:

- Use `jdeps` to check the dependencies of individual JARs in a Java SE 8 application
- Describe the difference between top-down and bottom-up migration
- Use the class path and the module path to run a Java SE 9 application
- Describe split packages and how they can occur
- Describe cyclic dependencies and a way to address them

Topics

- Application migration overview
- Top-down migration
- Bottom-up migration
- More on libraries
- Split packages
- Cyclic dependencies

The League Application



The graphic in the slide shows a typical application. It has three layers: the application JAR files at the top, the JDK at the bottom, and a number of library JAR files in the middle. In an application written prior to Java SE 9, all of these three layers comprise JAR files. To run them, they are simply added to the class path and loaded as needed by the JVM.

Assume that even though the application and library JARs were written using a Java version prior to SE 9, you now want to run them on as a modularized application.

Given this three-layer application, it's important to understand that while the JDK is modularized, JARs can be run on the module path or on the class path as desired. This flexibility allows you to start by migrating the lowest layer, the libraries, first, while still running the application JARs on the class path without modularizing them. You can also migrate the application JARs first and run these modularized JARs with library JARs that have not been modularized. This is covered next in this lesson.

In this and subsequent slides, modules will be shown without the `.jar` suffix. (Also, note that modules do not need to be placed in a JAR file to function as modules.)

Run the Application

Using the class path:

```
java -cp \
   dist/League.jar: \
   lib/Soccer.jar: \
   lib/Basketball.jar: \
   lib/display-ascii-0.1b.jar: \
   lib/jackson-core-2.6.2: \
   lib/jackson-databind-2.6.2.jar: \
   lib/jackson-annotations-2.6.2.jar \
   main.Main
```

Here's the entire application being run on the class path, exactly as you would do for an application running on Java SE 8. But assuming this is running on a modular JDK, something slightly different is going on here as the modular JDK works only with modules. What happens is that if a JAR file is placed on the class path, as shown in this example, the module system will essentially treat all the packages as if they're in a special module called the unnamed module.

The Unnamed Module

- All types must be associated with a module in Java SE 9.
- A type is considered a member of the unnamed module if it is:
  - In a package not associated with any module
  - Loaded by the application
- Unnamed modules:
  - Read all other modules
  - Export all their packages
  - Cannot have any dependencies declared on them
  - Cannot be accessed by a named module
    - A named module is one with a `module-info.java` file.

Topics

- Application migration overview
- Top-down migration
- Bottom-up migration
- More on libraries
- Split packages
- Cyclic dependencies

Top-down Migration and Automatic Modules

Top-down migration refers to the order in which two JARs may be migrated to corresponding modules (where the migration is one to one).

- Given JARs `a.jar` and `b.jar` where some classes in `a.jar` depend on some classes in `b.jar`, top-down migration involves migrating `a.jar` first.
  - As a consequence, `b.jar` must be run on the module path as otherwise it cannot be accessed by what is now module `a`.
    - If run on the class path, it would become an unnamed module, and named modules cannot access unnamed modules.
  - By running `b.jar` on the module path, it becomes an automatic module with a name and can be accessed by module `a`.
    - Module a can now, in its `module-info.java` file, declare its dependency on the automatic module created from `b.jar`.
    - For example, if "b" is the name given to this automatically generated module based on `b.jar`, module `a` can refer to this automatic module with the directive:
      - `requires b;`

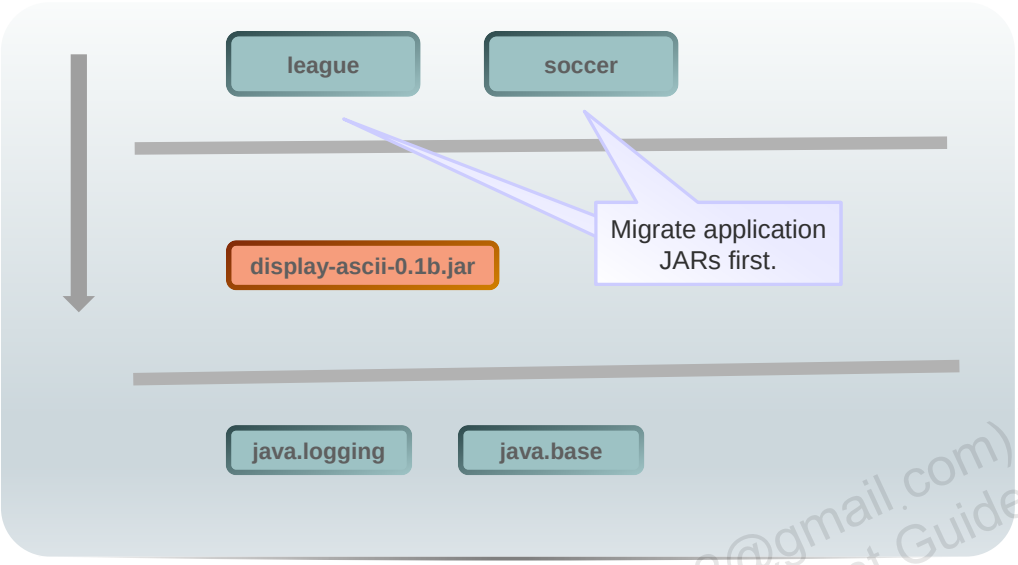
Note that the distinction here is that in the case of two JARs, `a.jar` and `b.jar`, top-down migration means first migrating the JAR that depends on the other JAR. Classes in `a.jar` depend on classes in `b.jar`, so top-down migration means migrating `a.jar` first.

Automatic Module

- Is a JAR file that does not have a module declaration and is placed on the module path
- Is a “real” module
- Requires no changes to someone else's JAR file
- Is given a name derived from the JAR file (either from its name or from metadata)
- Requires all other modules
- Can be required by other modules
- Exports all of its packages

Two unique things about automatic modules, that they require all other modules and that they export all their packages, are necessitated by the fact that they don’t have their own `module-info.java` file.

Top-Down Migration



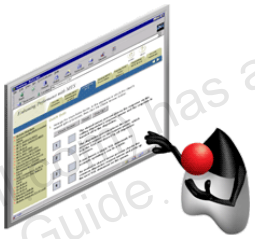
The graphic in the slide shows an example in which the migration has been performed top down. Here, the JAR files, `League.jar` and `Soccer.jar`, have been converted into two modules, `league` and `soccer`. This is done by adding a `module-info.java` file to the source code for each of the JAR files. This is shown in more detail in the following slides.

In this example, the JAR files, `League.jar` and `Soccer.jar`, were modularized at the same time, but this is still top-down migration because classes in `League.jar` depend on classes in `display-ascii-0.1b.jar`. Therefore, `display-ascii-0.1b.jar` will need to be become an automatic module.

Creating `module-info.java`—Determining Dependencies

Consider each JAR file that will become an application module—what does it require and what does it export?

- Module league requires?
- Module league exports?
- Module soccer requires?
- Module soccer exports?



Check Dependencies

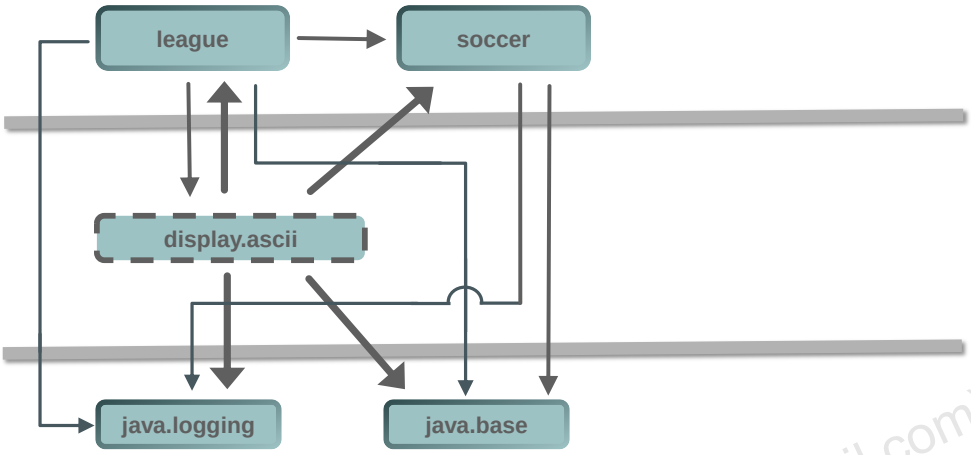
Run `jdeps` to check dependencies:

```
jdeps -s lib/display-ascii-0.1b.jar lib/Soccer.jar dist/League.jar
   League.jar -> lib/Soccer.jar
   League.jar -> lib/display-ascii-0.1b.jar
   League.jar -> java.base
   League.jar -> java.logging
   Soccer.jar -> java.base
   Soccer.jar -> java.logging
   display-ascii-0.1b.jar -> java.base
```

The `jdeps` command enables you to see what the dependencies are by looking at the JAR files. If `League.jar` becomes the `league` module, it will require the `soccer` module (created from `Soccer.jar`). But what about the requirement for `display-asci-0.1b.jar`? If you are performing top-down migration, the whole point is that you can migrate the application first, but use the libraries as they are. The next slides show how this is possible.

Library JAR to Automatic Module

Typical Application Modularized



This is the result. The `league` module can now require the `display.ascii` module, and the `display.ascii` module behaves like any other module, except that it requires every other module.

Note that in the graphics in this lesson, automatic modules are denoted with a dotted line rather than a solid one.

Topics

- Application migration overview
- Top-down migration
- Bottom-up migration
- More on Libraries
- Split packages
- Cyclic dependencies

Bottom-up Migration

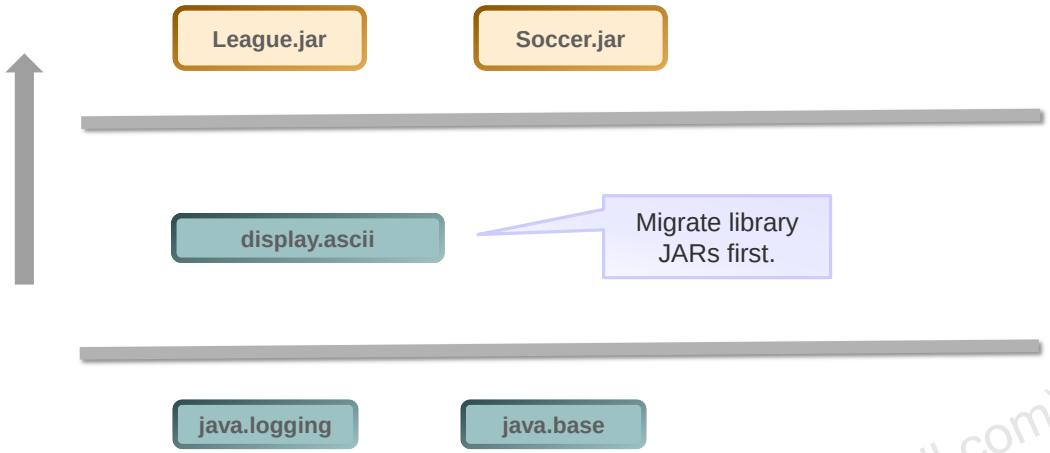
Bottom-up migration refers to the order in which two JARs may be migrated to corresponding modules (where the migration is JAR to module one to one).

Given two JARs `a.jar` and `b.jar`, where some classes in `a.jar` depend on some classes in `b.jar`:

- Bottom-up migration involves migrating `b.jar` first.
  - This migrated module, `b`, can now be run on the module path.
  - JAR `a.jar` can run on the class path as an unnamed module or on the module path as an automatic module.

Note that the distinction here is that in the case of two JARs, `a.jar` and `b.jar`, bottom-up migration means first migrating the JAR that has the dependency on it. Classes in `a.jar` depend on classes in `b.jar`, so bottom-up migration means migrating `b.jar` first.

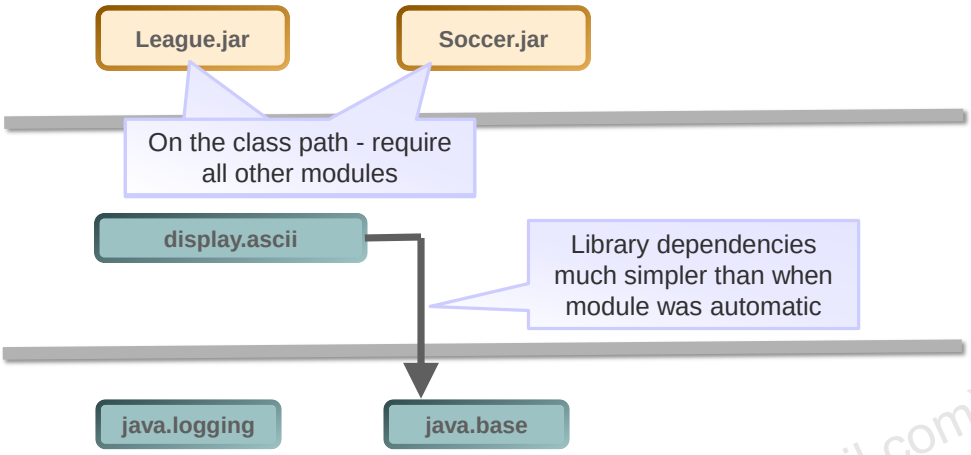
Bottom-Up Migration



The graphic in the slide shows bottom-up migration, where the library JAR, which was previously named `display-ascii-0.1b.jar`, has been converted to a module named `display.ascii`. The slide shows `League.jar` and `Soccer.jar` as not having been converted to modules.

Because the class path and the module path can be combined, the original application JARs can be run on the class path, while the `display.ascii` library module is run on the module path. It would also be possible to run `League.jar` and `Soccer.jar` on the module path—they would then run as automatic modules.

Modularized Library



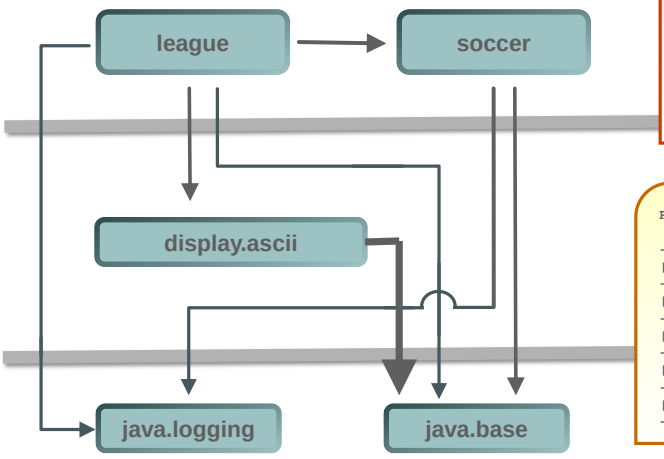
Run Bottom-Up Migrated Application

Note use of `--add-modules`

```
java -cp dist/League.jar:lib/Soccer.jar -p lib/display.ascii \
main.Main
java -cp dist/League.jar:lib/Soccer.jar -p mods/display.ascii \
   --add-modules display.ascii main.Main
```

You need to use the `--add-modules` option in the `java` command because `League.jar` is running as an unnamed module. The Java run time cannot determine what modules to resolve, so `display.ascii` must be added manually as shown in the slide.

Fully Modularized Application




```
java -p dist/League.jar: \
        lib/Soccer.jar: \
        lib/display.ascii.jar \
     -m League.jar/main.Main
```

```
Robins win the league!

-----------------------------------------------------------------
|          | Robins | Sparrows | Magpies | Crows | Points | Goals |
-----------------------------------------------------------------
| Robins   |  X     | 0 - 1    | 0 - 4   | 1 - 1 | 2      | 3     |
-----------------------------------------------------------------
| Sparrows | 2 - 1  |  X       | 1 - 1   | 1 - 0 | 8      | 7     |
-----------------------------------------------------------------
| Magpies  | 1 - 0  | 2 - 2    |  X      | 0 - 1 | 7      | 9     |
-----------------------------------------------------------------
| Crows    | 1 - 1  | 2 - 0    | 1 - 1   |  X    | 7      | 6     |
-----------------------------------------------------------------
```


Whether starting from the bottom-up or the top-down, the migration process is completed when all JARs have been converted, that is, both the application JARs and the library JARs. In the example in the slide, the library JAR (after being modularized) is `display.ascii.jar`. This assumes that this library JAR is under the control of the organization doing the migration. But in many cases, this may not be true, and you may have to either continue to use the library as an automatic module or wait until the library maintainer creates a modularized version of the library.

Module Resolution

Use `--show-module-resolution` with `--limit-modules` to limit output.

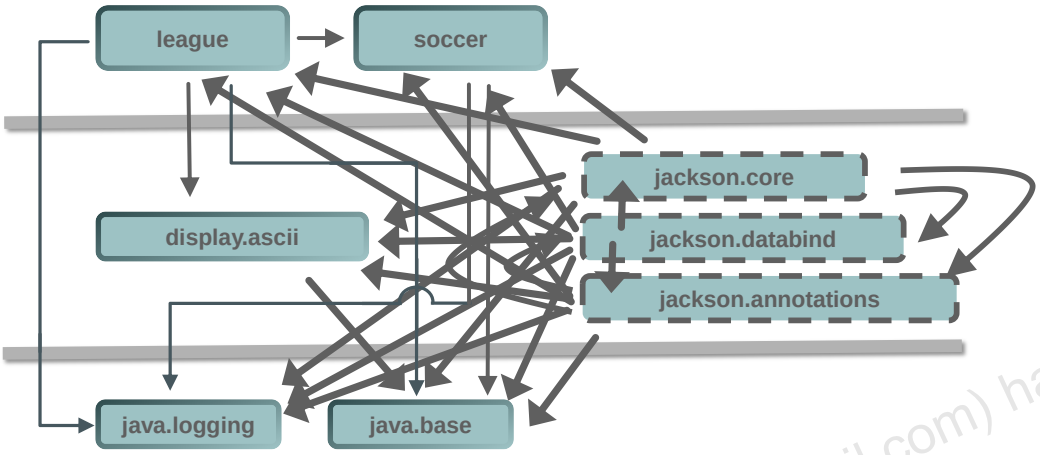
```
   java --limit-modules java.base,display.ascii,Soccer \
   --show-module-resolution -p mods:lib -m league/main.Main
root league file:///home/oracle/examples/mods/league/
   league requires display.ascii
 file:///home/oracle/examples/league/mods/display.ascii/
   league requires Soccer
 file:///home/oracle/examples/league/mods/soccer/
   league requires java.logging jrt:/java.logging
   soccer requires java.logging jrt:/java.logging
   java.base binds java.logging jrt:/java.logging
```

The `--show-module-resolution` option will list all the modules that are being resolved as the application starts up. If you know exactly what modules are required, you can limit the modules that are resolved and thus the output with the `--limit-module` option.

Topics

- Application migration overview
- Top-down migration
- Bottom-up migration
- More on Libraries
- Split packages
- Cyclic dependencies

More About Libraries



The slide shows an example where there are some additional libraries that remain as automatic modules. The libraries in question are the Jackson JSON libraries. Because there are three JARs and because automatic modules require all other modules transitively, the resultant module graph is a little messy. But it still has much more dependency information in the overall graph than the corresponding class path.

Run Application with Jackson Libraries

- Compile:

```
javac -p lib -d mods --module-source-path src-9 \
   $(find src-9 -name "*.java")
```

- Run:

```
java -p mods:lib -m league/main.Main
Unable to make field private soccer.SoccerTeam soccer.Soccer.homeTeam
accessible: module Soccer does not "opens soccer" to module
jackson.databind (through reference chain: soccer.Soccer[0])
Exception in thread "main" java.lang.reflect.InaccessibleObjectException:
...
```

You will find that many libraries use reflection. This will cause a problem because they won't be able to reflect on types within your application unless their packages have been exported. For types that are marked as private, exporting is not enough, and another directive `opens` is available.

`opens` is covered in more detail in the lesson titled "Working with the Module System."

Open Soccer to Reflection from Jackson Libraries

- Open the entire module.

```
open module soccer {
    requires java.logging;
    exports soccer;
    exports util;
}
```

- Open just the package needed to all modules or to a specific module (as shown).

```
module soccer {
    requires java.logging;
    exports soccer;
    exports util;
    opens soccer to jackson.databind;
}
```

Topics

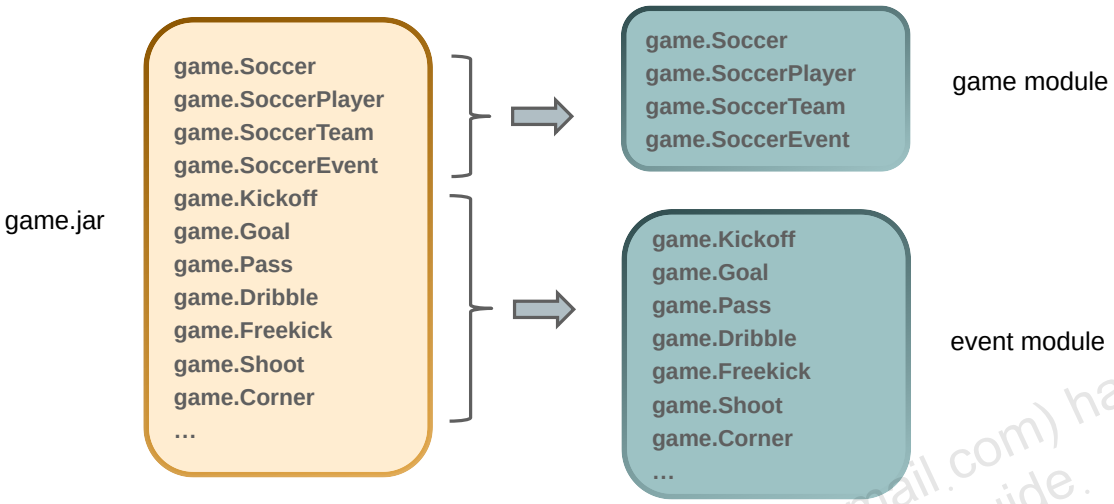
- Application migration overview
- Top-down migration
- Bottom-up migration
- More on Libraries
- Split packages
- Cyclic dependencies

Split Packages

How they can occur during migration

- Splitting Java 8 application into modules
- Converting a Java 8 application that was organized for access control

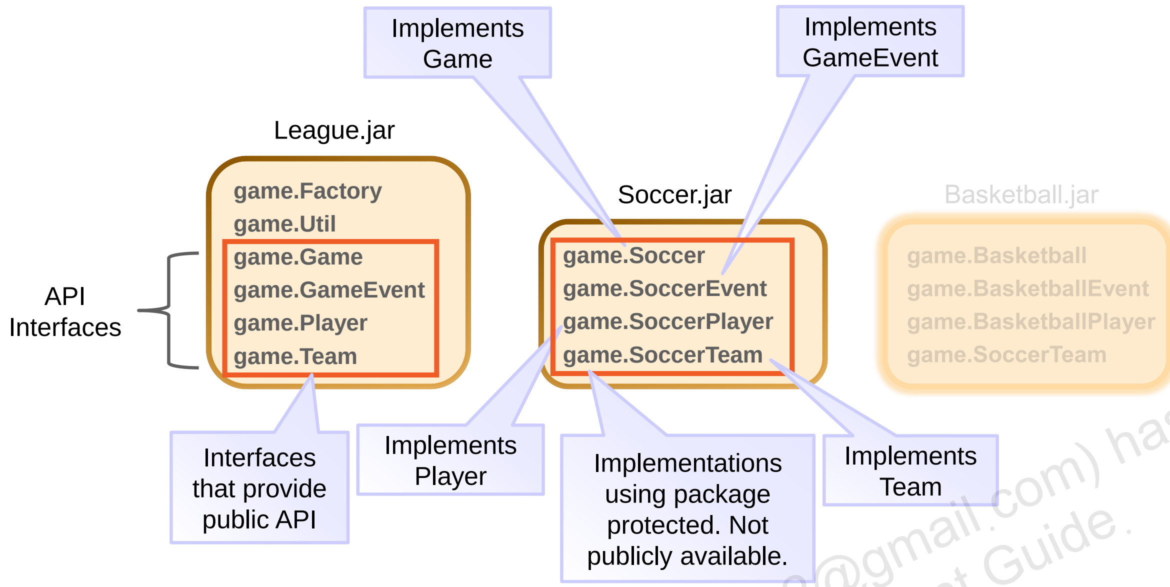
Splitting a Java 8 Application into Modules



In the example in the slide, the problem is caused by modifying a simply designed original Java SE 8 application so that it now uses modules in a logical way.

The solution is straightforward: refactor the classes to use a number of packages in a logical way, similar to the logical split shown by module in the diagram.

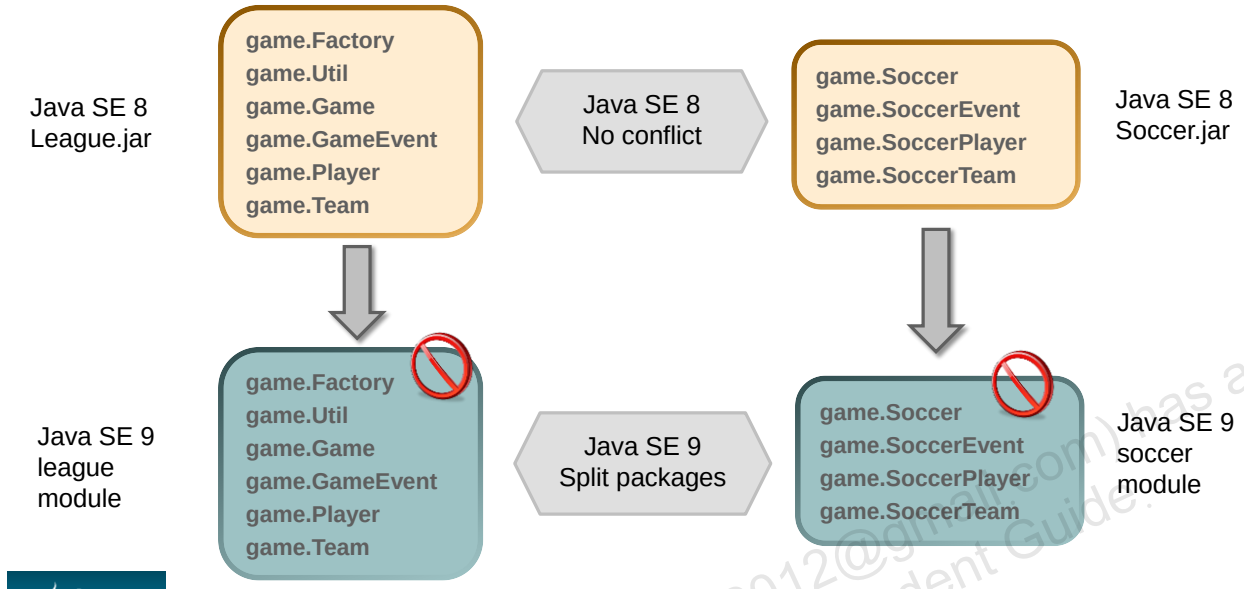
Java SE 8 Application Poorly Designed with Split Packages



This example illustrates a split package problem, where split packages are used intentionally to allow the classes in a JAR to implement interfaces in another JAR but not have the implementations public. The idea is that if a package name is reused in another JAR, the classes in that package could be given package access and, therefore, could be accessible to classes in the same package that are stored in another JAR, but be inaccessible to other classes (that are not in that package).

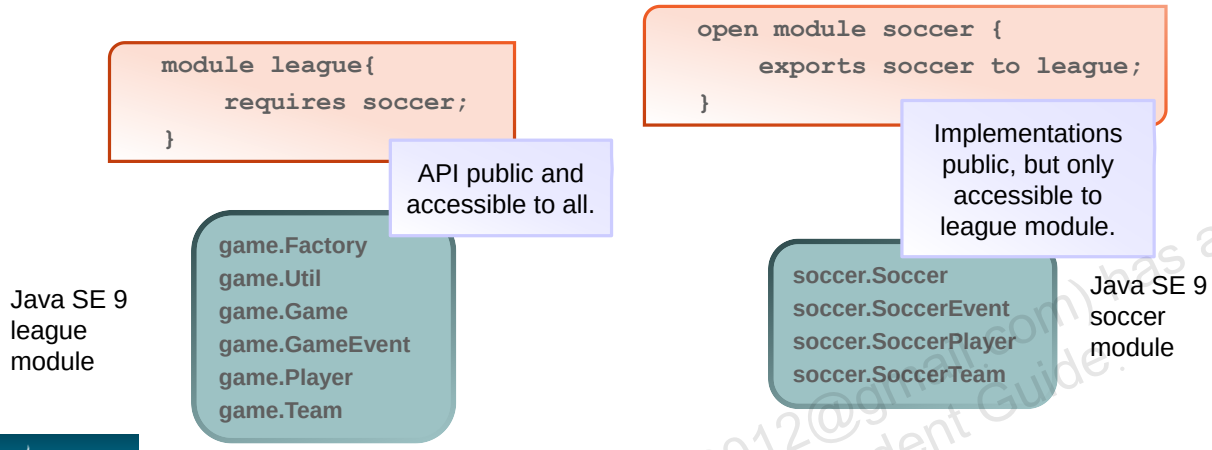
Note that this is not considered a good approach; it is a nonstandard and problematic attempt to achieve what is very easily achievable with modular Java.

Migration of Split Package JARs to Java SE 9



Addressing Split Packages

Encapsulation is achieved at the modular JAR level in Java SE 9 through the use of qualified export.

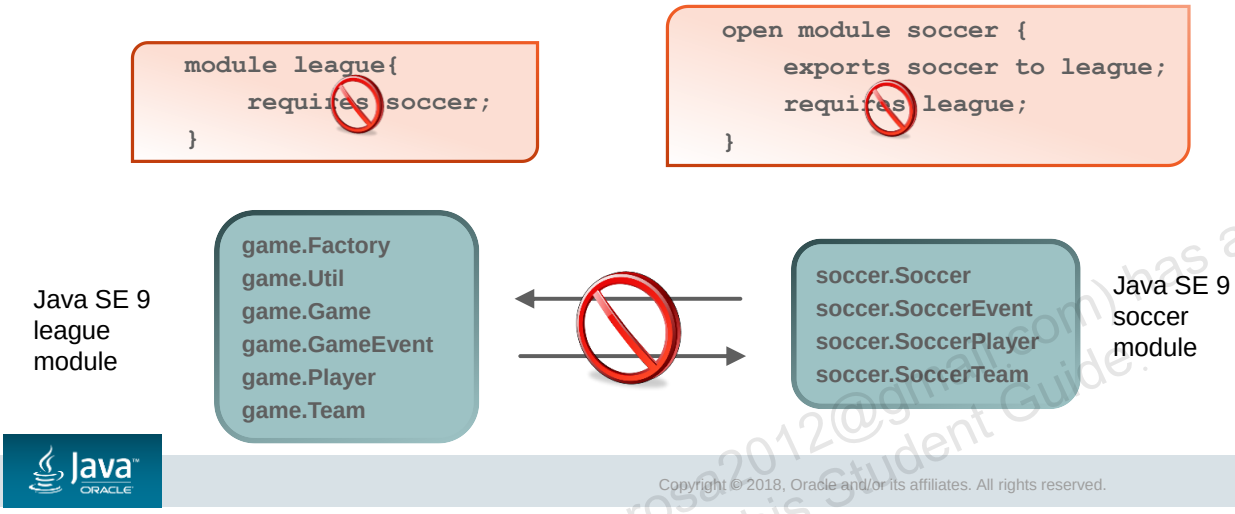


Topics

- Application migration overview
- Top-down migration
- Bottom-up migration
- More on Libraries
- Split packages
- Cyclic dependencies

Cyclic Dependencies

Cyclic module dependencies are not permitted in Java SE 9.

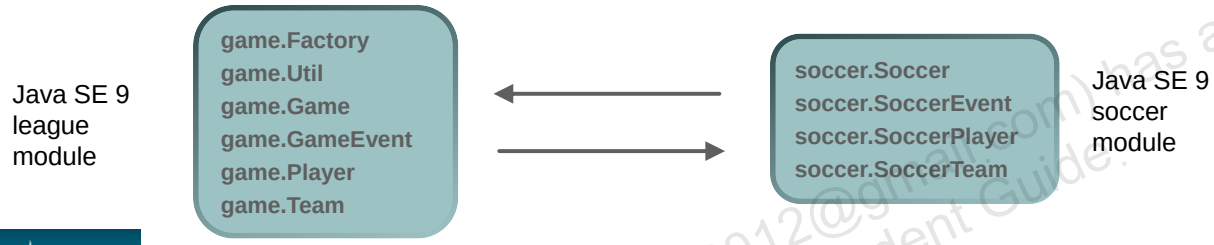


Migrating the previous split packages example will raise another issue. Implementation classes in the `soccer` module require access to the interfaces in the `league` module.

The `Factory` class in the `league` module needs access to implementation classes in the `soccer` module. This is a cyclic dependency and is not permitted in the module system.

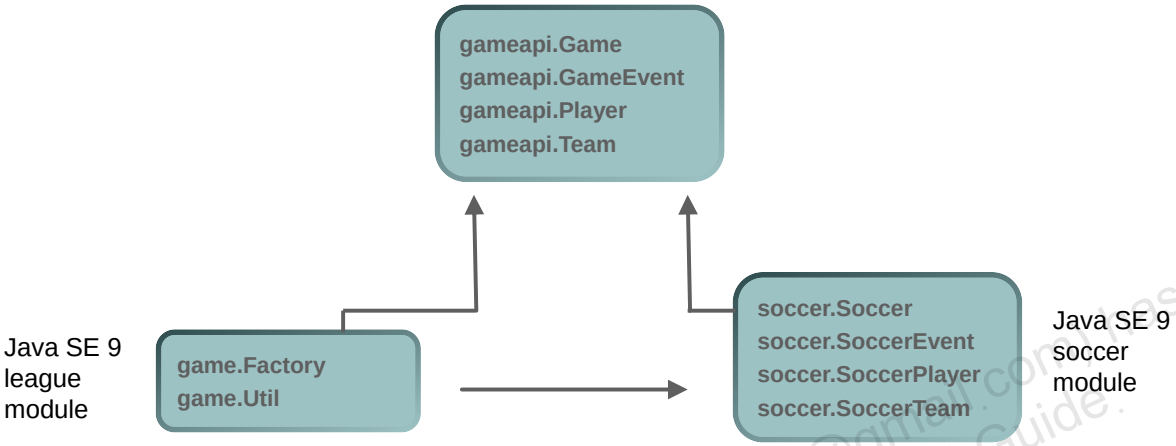
Addressing Cyclic Dependency 1

One possible approach is to remove the dependency soccer has on league…



Addressing Cyclic Dependency 2

…and instead create a new module that both league and soccer are dependent on.



The graphic in the slide shows one way of addressing this cyclic dependency. You can create a new module to hold the various interfaces of the API so that the cycle no longer exists. This is not a solution for every case, but often the consideration of where to put the interface types may help suggest an answer.

There is another answer though, and that is the use of services. This is covered in the lesson titled "Services."

Top-down or Bottom-up Migration Summary

Top-down or Bottom-up migration refers to the order in which JARs may be migrated to corresponding modules (where the migration is one to one JAR to module or modular JAR). Given two JARs `a.jar` and `b.jar`, where some classes in `a.jar` depend on some classes in `b.jar`:

- Top-down migration involves migrating `a.jar` first.
  - As a consequence, `b.jar` must become an automatic module as otherwise it cannot be accessed by what is now module `a`.
  - Both modules must be run on the module path.
- Bottom-up migration involves migrating `b.jar` first.
  - This migrated module, `b`, can now be run on the module path.
  - JAR `a.jar` can run on the class path as an unnamed module or on the module path as an automatic module.

Summary

In this lesson, you should have learned how to:

- Use `jdeps` to check the dependencies of individual JARs in a Java SE 8 application
- Describe the difference between top-down and bottom-up migrations
- Use the class path and the module path to run a Java SE 9 application
- Describe split packages and how they can occur
- Describe cyclic dependencies and a way to address them

Practice 11: Overview

This practice covers the following topics:

- Practice 11-1: Examining the League Application
- Practice 11-2: Using `jdeps` to Determine Dependencies
- Practice 11-3: Migrating the Application
- Practice 11-4: Adding a main Module
- Practice 11-5: Migrating a Library
- Practice 11-6: Bottom-Up Migration
- Practice 11-7: Adding the Jackson Library