History

We know that the basic unit of reuse in Java is `class`

- Inheritance => reuses behavior
- Interface => reuses abstractions

Classes are organized into packages.
Packages share code with each other through the `public` modifier, but since it's shared with every other package, it's challenging to visualize which package is reusing code from another package.

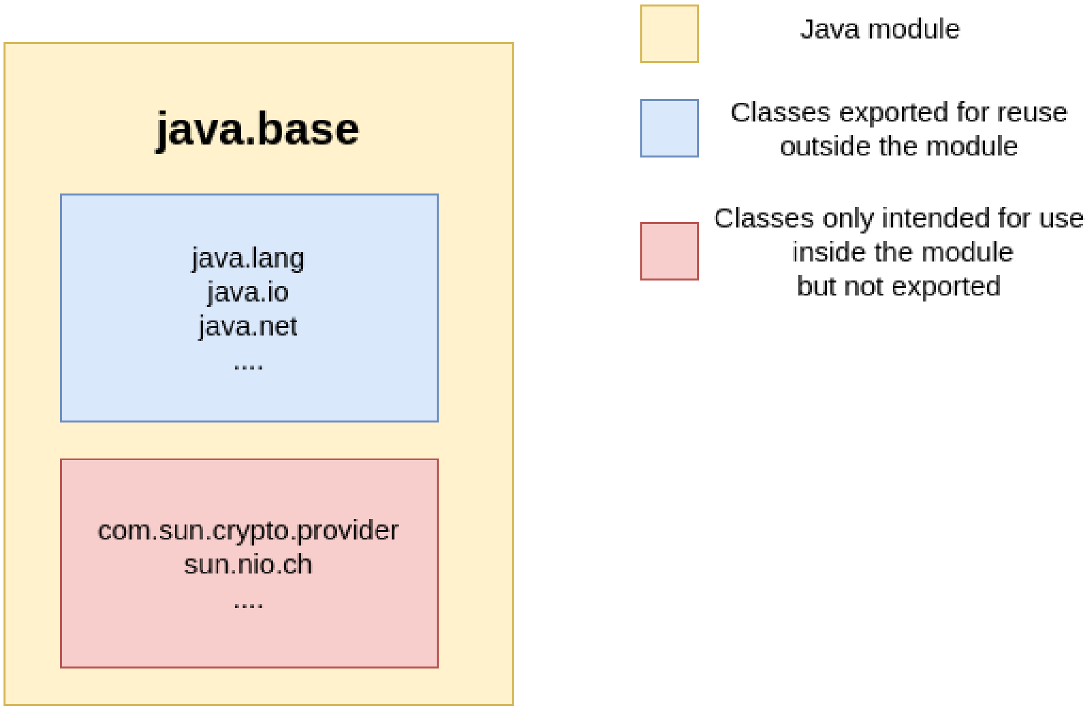
Hence, there was a need to organize packages to clarify dependencies. Through Project Jigsaw, Java 9 introduced modules. In java's context - A module is a set of packages designed for reuse, so the parts that you want to be reused can be and which you don't can't be

For listing these dependencies, have to create a separate file `module-info.java`

In short modules provide a nice way to encapsulate packages, and access only depends on whether the module exports/provides that particular class

---

Requires/Exports


java.base
java.lang
java.io
java.net
....
com.sun.crypto.provider
sun.nio.ch
....
Java module
Classes exported for reuse outside the module
Classes only intended for use inside the module but not exported

```
// module-info.java
module java.base {
     exports java.lang;
     exports java.io;
     exports java.net;
     Export java.util
}
```

In addition Use of the `public` can be granularized into 3 use cases w.r.t. Modules

1. Public to everyone - implementation provides static access [2]
   Eg - `exports helloWorld;`
2. Public but only to friend modules - The class is exported only to specific classes
   Eg - `exports helloWorld to Impl1;`
3. Public only within a module - meaning it hasn't been exported

`transitive`

Note that no sub packages are imported by default, however transitive access can be provided through the transitive keyword
Consider the following directive from the java.desktop module declaration:

```
requires transitive java.xml;
```

In this case, any module that reads java.desktop also implicitly reads java.xml.

module-info.java

```
module logger {
       Requires FileIO;
}
```

logger.java

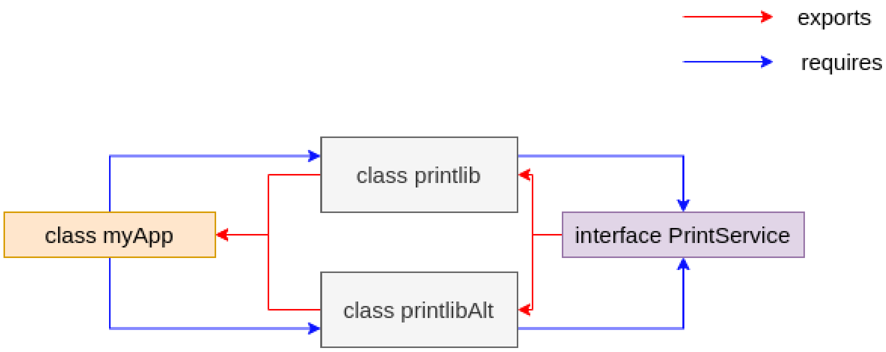
```
module logger {
       Requires FileIO;
}
```

Use/Provides

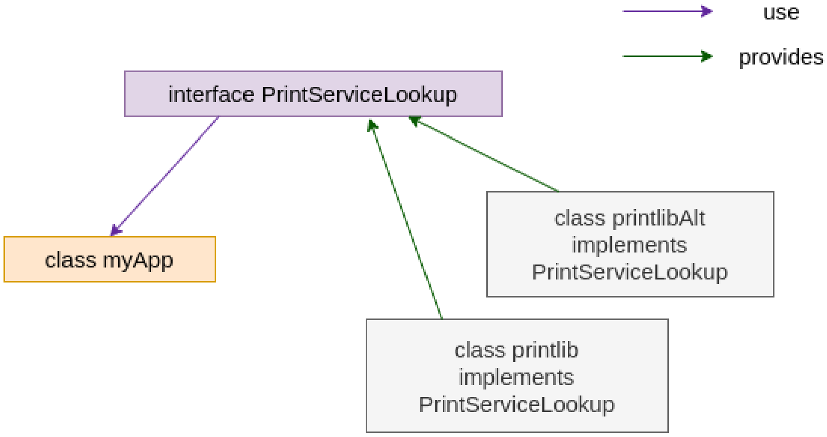
Follows the services design pattern consisting of:

1. One module requesting an implementation of an interface
2. One/More modules providing implementation of that interface
3. The interface itself

Consider an example case: `myApp` optionally requirements `printlib/printlibAlt` which act as potential optimizations to the printing process



Tight Coupling


use
provides
interface PrintServiceLookup
class myApp
class printlibAlt implements PrintServiceLookup
class printlib implements PrintServiceLookup

Loose Coupling

Above case: myApp being dependent on printlib/printlibAlt when their usage could be optional. Also if printService fails the whole build fails. Hence, we see that our elements are tightly coupled.

Below case: When PrintServiceLookup is called, it returns a list of attached classes, and application logic would take care of the rest. If does not find any other printers it can default to an action. Hence, the components are loosely coupled in their functionality.

Now, within myApp, one has to handle the application logic by having a default behaviour or finding the required class being provided by PrintServiceLookup

```
ServiceLoader<PrintServiceLookup> psls =
     ServiceLoader.load(PrintServiceLookup.class);

for (PrintServiceLookup psl : psls) {
            PrintService ps = psl.getDefaultPrintService();
     if (ps.isDocFlavorSupported(...)) return ps;
}

return DEFAULT_PRINT_SERVICE;
```

Usage

```
[open] module <name> {
      <main>
}
```

Main

```
requires [transitive] <package>
exports <package> [to <friend-package>]
uses <module-provider>
provides <module-provider> with <provider-implementation>
```

Advanced

Note: open cannot be combined with opens

```
open module <name> { ... }
opens <sub-module-name>
```

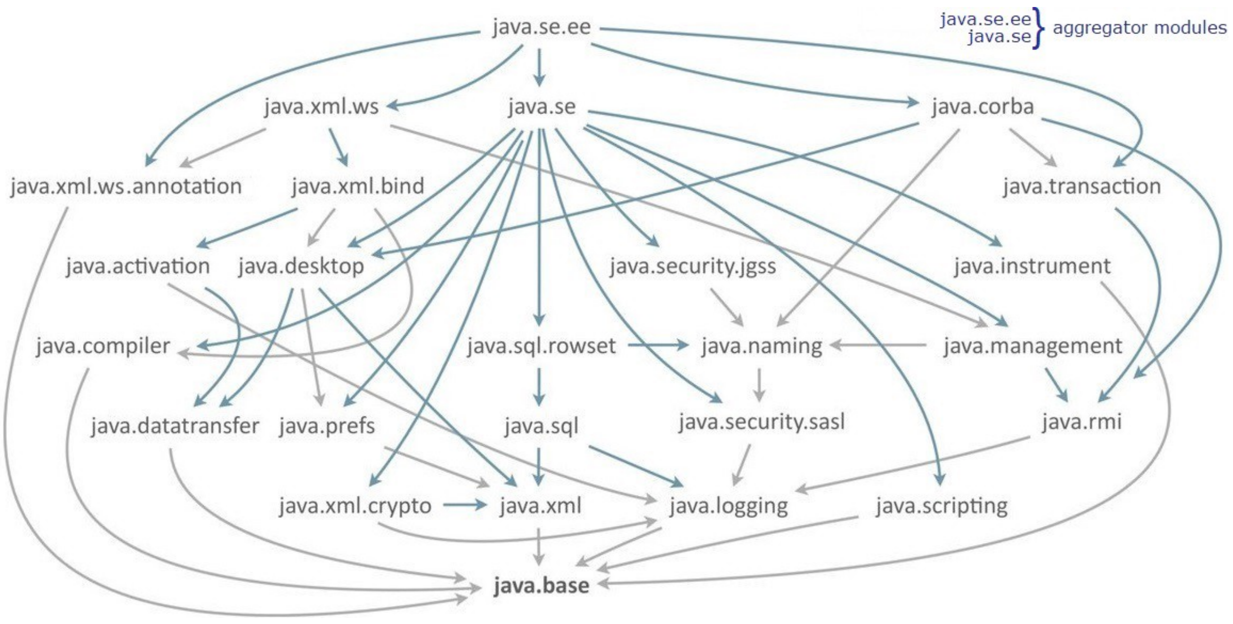
Advantages of adding a module system in the context of Java:

- No missing dependencies -> since it checks whether all the modules are available at compile time
- No cyclic dependencies
- No split packages -> Having two modules export the same package is also sometimes referred to as a split package. A split package means that the package's total content (classes) is split between multiple modules. This is not allowed.
- Smaller Application Distributables - by specifying exactly which set of classes we use, we know which classes won't be used.

Opinion: Designed for migration, hence hindered by how strong it could have been

1. Poor use of reflection
2. Automatic modules - `requires` for everything
3. Many projects use build systems with Maven

Java SE Modules



References

[1] https://openjdk.org/projects/jigsaw/spec/sotms/
[2] https://blogs.oracle.com/java/post/modular-development-with-jdk-9
[3] https://stackoverflow.com/questions/46482364/what-is-an-open-module-in-java-9-and-how-do-i-use-it

—---

http://www.cs.cmu.edu/~aldrich/wyvern/wyvern-guide.html

Wyvern thinks in terms of resources