Note about the Reflection API

Overview

- reflection, introspection
- allows for
  - obtaining information about classes, fields, methods
  - creating objects
  - calling methods
  - ...
- the package `java.lang.reflect`
- the class `java.lang.Class<T>`

java.lang.Class

- an instance of the class **Class** represents a class (interface, enum,...) in a running program
- primitive types also represented as instances of **Class**
- no constructor
- instances created automatically during loading the class code to JVM
  - classes are loaded to JVM when firstly used

java.lang.Class

- obtaining an instance of **Class**
  - `getClass()`
    - the method of the Object class
    - returns the class of the object on which was called
  - the class literal
    - `JmenoTridy.class`
    - the class for the given type
  - `Class.forName(String className)`
    - static method
    - returns the class of the given name
  - for primitive types
    - the static attribute TYPE on the wrapper classes
      - `Integer.TYPE`
    - the literal `class`
      - `int.class`

java.lang.Class

- class are loaded to JVM by a *classloader*
  - `java.lang.ClassLoader`
  - the standard classloader looks up classes in CLASSPATH
  - own classloaders can be created
  - `Class.forName(String className, boolean initialize, ClassLoader cl)`
    - loads the class by the given classloader and returns an instance of the Class
  - `getClassLoader()`
    - the method of Class
    - the classloader, which loaded the class

java.lang.Class: methods

- `String getName()`
  - returns the name of the class
  - for primitive types returns their names
  - for array returns a string beginning with the chars '[' (number of '[' corresponds to dimension) and then an identification of the element type
    Z..boolean, B..byte, C..char, D..double, F..float, I..int, J..long, S..short, Lclassname..třída nebo interface

```
String.class.getName() // returns "java.lang.String"
byte.class.getName()   // returns  "byte"
(new Object[3]).getClass().getName()
     // returns   "[Ljava.lang.Object;"
(new int[3][4][5][6][7][8][9]).getClass().getName()
     // returns   "[[[[[[[I"
```

java.lang.Class: methods

- `public URL getResource(String name)`
- `public InputStream getResourceAsStream(String name)`
  - reads a resource
    - image, ..., anything
  - data loaded by a classloader => loading by the same rules as loading classes
  - a name of the resource ~ a hierarchical name as of classes
    - dots replaced by `'/'`

java.lang.Class: methods

- is... methods
  - `boolean isEnum()`
  - `boolean isInterface()`
  - ...
- get... methods
  - `Field[] getFields()`
  - `Method[] getMethods()`
  - `Constructor[] getConstructors()`
  - ...
- ...

Usage of Reflection API

- information about code
- dynamic loading
- plugins
- proxy classes
- ...

jar

Overview

- creating archives composed of .class files
- JAR ~ Java Archive
- file
  - extension .jar
  - format – ZIP
  - file META-INF/MANIFEST.MF
    - description of the content
- usage – distribution of software
  - CLASSPATH can contain .jar files
  - .jar files can be directly executed
- can contain also other files than .class files
  - images
  - audio
  - anything else

Usage

- creating an archive

  ```
  jar cf file.jar *.class
  ```

  - creates the file.jar with all .class files
  - adds the MANIFEST.MF file to it

  ```
  jar cmf manifest file.jar *.class
  ```

  - creates the file.jar with the given MANIFEST file

  ```
  jar cf0 soubor.jar *.class
  ```

  - no compression

  - see documentation for other parameters

- API for working with jar files
  - java.util.jar, java.util.zip

MANIFEST.MF file

- list of tuples
  - name : value
  - inspired by the standard RFC822
- tuples can be grouped
  - groups separated by an empty line
  - main group (the first one)
  - groups for individual entries in the archive
- length of lines – max 65535
- end of lines
  - CR LF, LF, CR

MANIFEST.MF files

- main group
  - Manifest-Version
  - Created-By
  - Signature-Version
  - Class-Path
  - Main-Class
    - applications can be launched

    java -jar archive.jar
- other section
  - the first tuple

    Name: path_to_the_entry_in_the_archive

Jar and Ant

- the task **jar**
  - parameters
    - destfile, basedir, includes, excludes, manifest
  - inner elements
    - manifest
  - example

```
<jar destfile="${dist}/lib/app.jar"
       basedir="${build}/classes"
       excludes="**/Test.class"
  />

<jar destfile="test.jar" basedir=".">
    <include name="build"/>
    <manifest>
      <attribute name="Built-By" value="${user.name}"/>
      <section name="common/class1.class">
        <attribute name="Sealed" value="false"/>
      </section>
    </manifest>
  </jar>
```

java.util.jar

- similar to java.util.zip
- `JarInputStream`, `JarOutputStream`
  - children of ZipInputStream and ZipOutputStream
  - JarInputStream has the getManifest() method
- `JarEntry`
  - child of `ZipEntry`
  - obtaining attributes
- `Manifest`
  - the MANIFEST.MF file

Modules

Modules

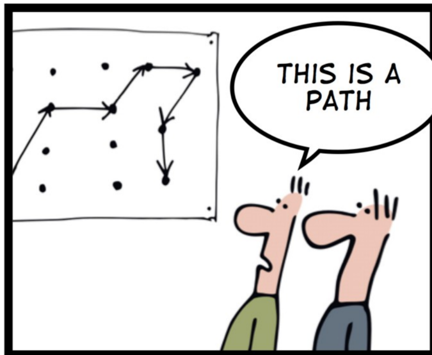
- a module
  - explicitely defines what is provided but also what is ***required***

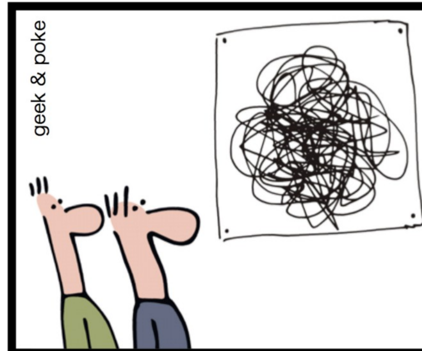
- why?
  - the *classpath* concept is “fragile”
  - no encapsulation

Modu

- a mod
  - exp ... so what is *req*
- why?
  - the
  - no e

GRAPH THEORY FOR GEEKS



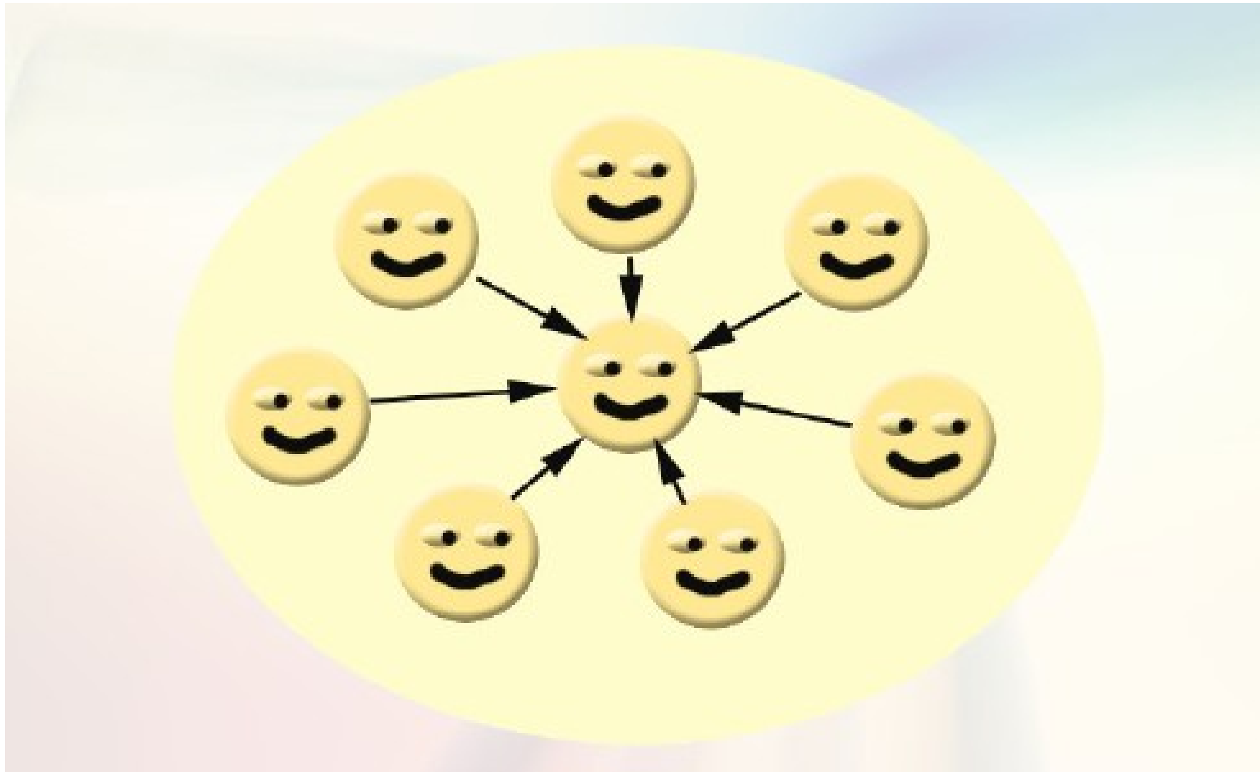


Modular apps – motivation

- why
  - applications get more complex
  - assembled from pieces
  - developed by distributed teams
  - complex dependencies
  - good architecture
    - know your dependencies
    - manage your dependencies

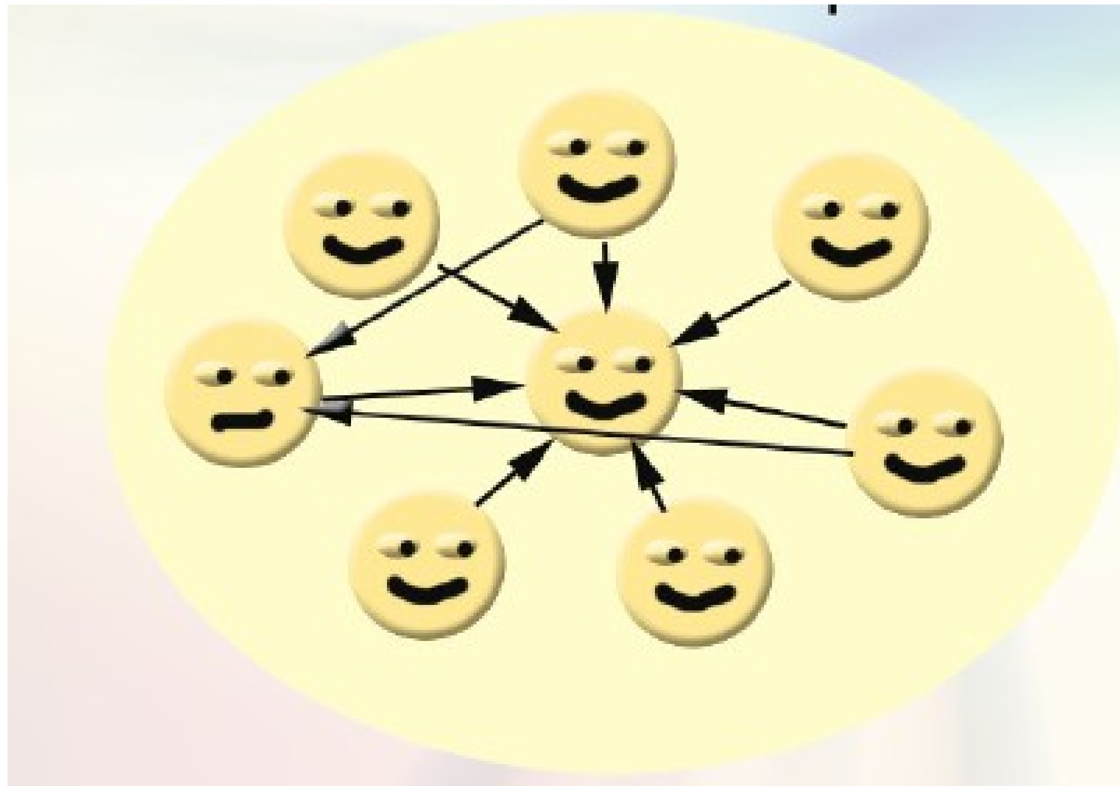
Modular apps – motivation

- Version 1.0 is cleanly designed...



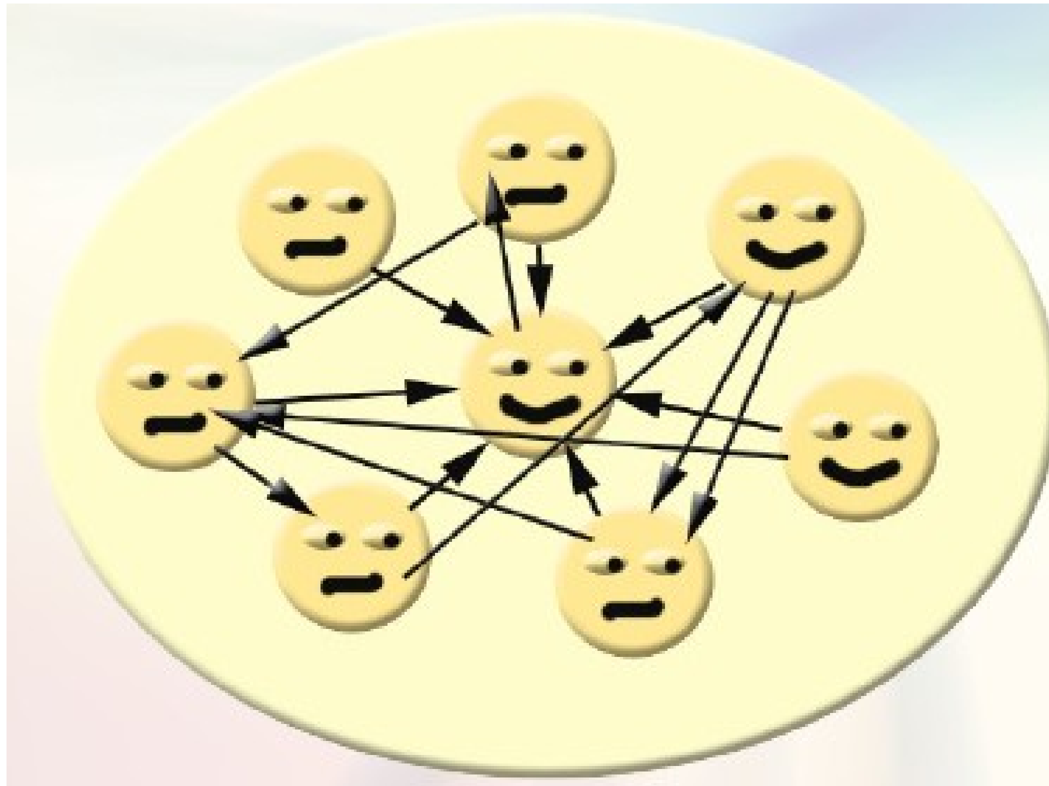
Modular apps – motivation

- Version 1.1...a few expedient hacks...we'll clean those up in 2.0



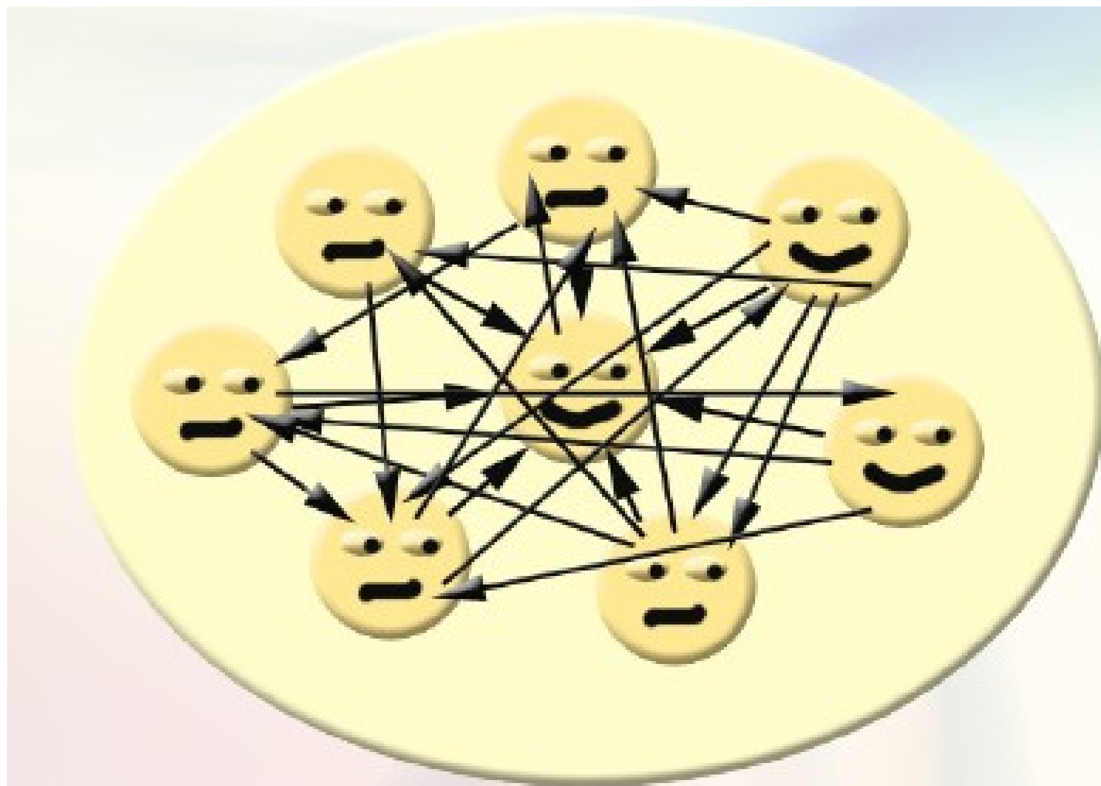
Modular apps – motivation

- Version 2.0...oops...but...it works!



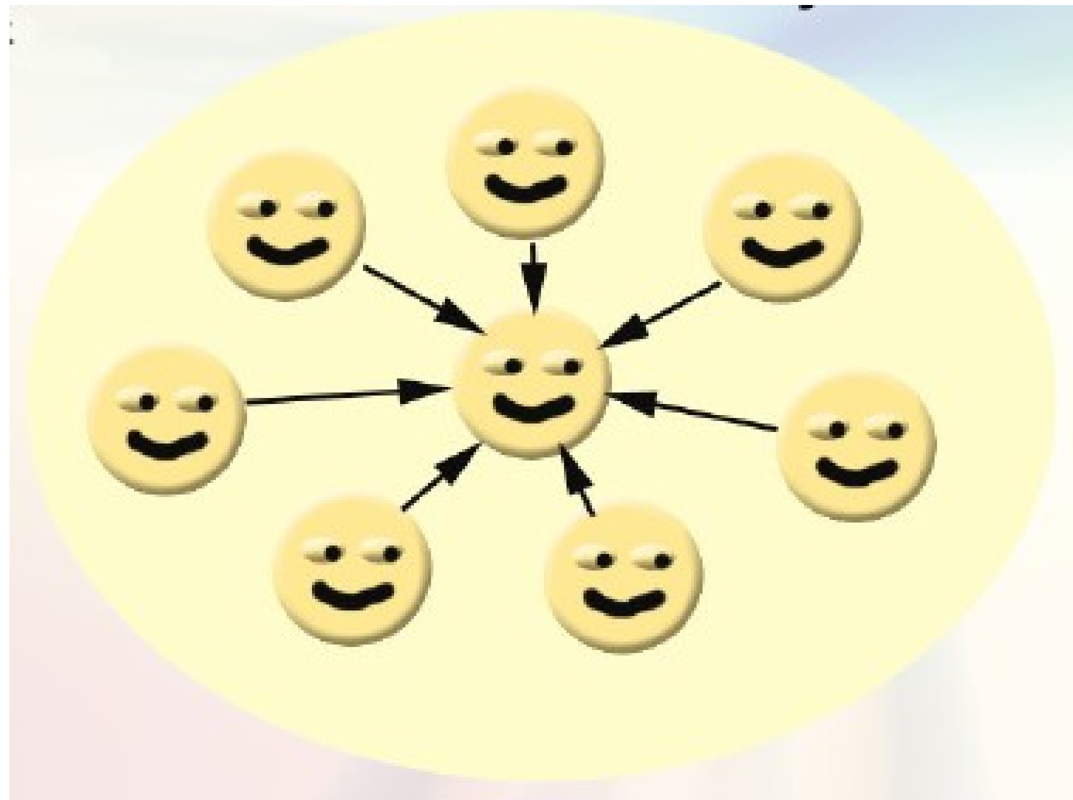
Modular apps – motivation

- Version 3.0...Help! Whenever I fix one bug, I create two more!



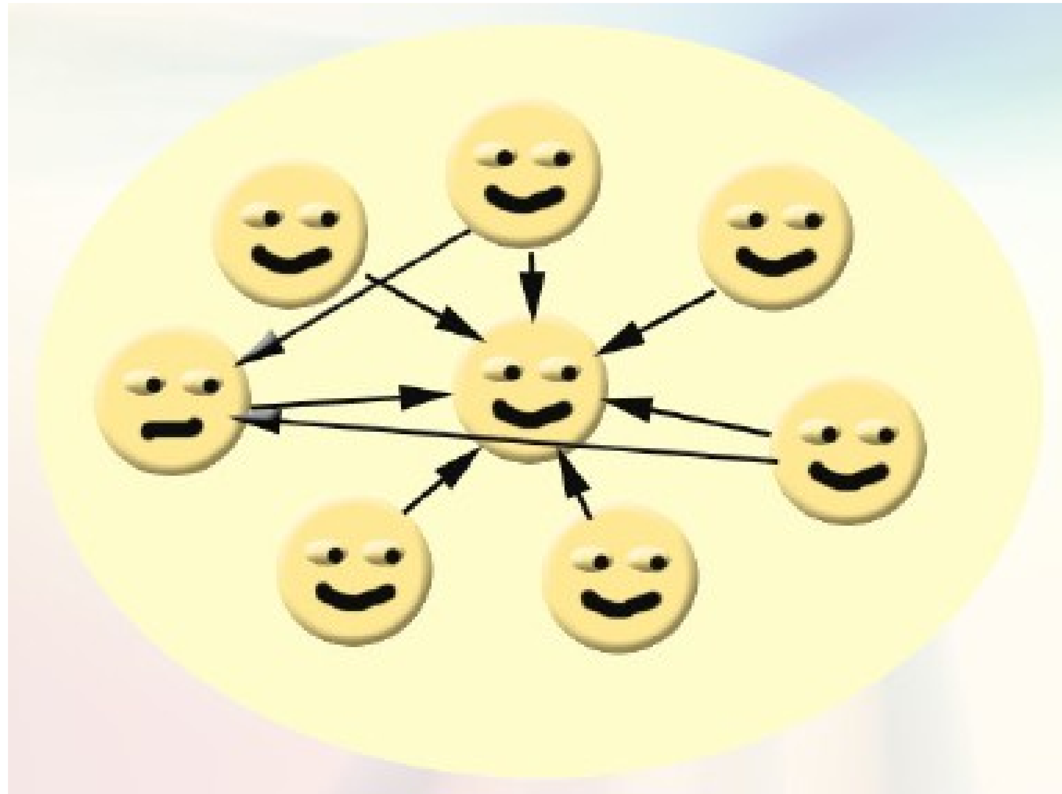
Modular apps – motivation

- Version 4.0 is cleanly designed. It's a complete rewrite. It was a year late, but it works...



Modular apps – motivation

- Version 4.1...does this look familiar?....



Module declaration

- module-info.java

```
module com.foo.bar {
    requires com.foo.baz;
    exports com.foo.bar.alpha;
    exports com.foo.bar.beta;
}
```

- modular artifact
  - modular JAR – JAR with module-info.class
  - a new format JMOD
    - a ZIP with classes, native code, configuration,...

Modules and JDK

- JDK std library modularized too
  - java.base – always „required“

```
module java.base {
    exports java.io;
    exports java.lang;
    exports java.lang.annotation;
    exports java.lang.invoke;
    exports java.lang.module;
    exports java.lang.ref;
    exports java.lang.reflect;
    exports java.math;
    exports java.net;
    ...
```

Module readability & module path

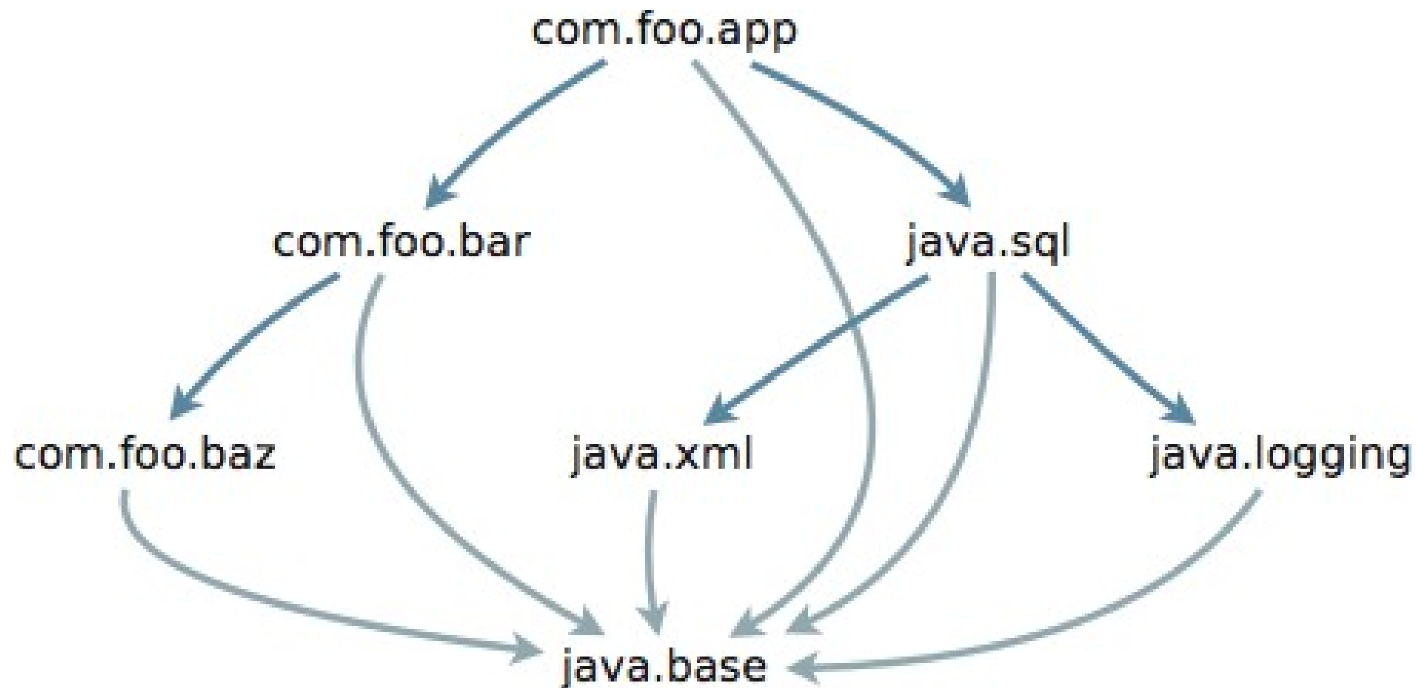
- When one module depends directly upon another

  Module ***reads*** another module (or, equivalently, second module is ***readable*** by first)

- ***Module path*** – eequivalent to classpath
  - but for modules
    - -p, --module-path

Module graph

```
module com.foo.app {
    requires com.foo.bar;
    requires java.sql;
}
```



Compatibility with “old” Java

- Classpath still supported
  - in fact – modules are “optional”
- Unnamed module
  - artefacts outside any module
    - “old” code
  - reads every other module
  - exports all of its packages to every other module

Slides version J12.en.2018.01
This slides are licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.