JAVA

Krátce o Reflection API

Přehled

- reflection, introspection
- umožňuje
  - zjišťování informací o třídách, atributech, metodách
  - vytváření objektů
  - volání metod
  - ...
- balík `java.lang.reflect`
- třída `java.lang.Class<T>`

java.lang.Class

- instance třídy **Class** reprezentuje třídu (interface, enum,...) v běžícím programu
- primitivní typy také reprezentovány jako instance třídy **Class**
- nemá žádný konstruktor
- instance vytvářeny automaticky při natažení kódu třídy do JVM
  - třídy jsou natahovány do JVM až při jejich prvním použití

java.lang.Class

- získání instance třídy **Class**
  - `getClass()`
    - metoda na třídě Object
    - vrátí třídu objektu, na kterém je zavolána
  - literál class
    - `JmenoTridy.class`
    - třída pro daný typ
  - `Class.forName(String className)`
    - statická metoda
    - vrátí třídu daného jména
  - pro primitivní typy
    - statický atribut TYPE na wrapper třídách
      - `Integer.TYPE`
    - literál `class`
      - `int.class`

java.lang.Class

- třídy do JVM natahuje *classloader*
  - `java.lang.ClassLoader`
  - standardní classloader hledá třídy v CLASSPATH
  - lze si napsat vlastní classloader
  - `Class.forName(String className, boolean initialize, ClassLoader cl)`
    - natáhne třídu daným classloaderem a vrátí objekt třídy Class
  - `getClassLoader()`
    - metoda na Class
    - classloader, kterým byla třída natažena

java.lang.Class: metody

- `String getName()`
  - vrátí jméno třídy
  - pro primitivní typy vrátí jeho jméno
  - pro pole vrátí řetězec začínající znaky [(tolik, kolik má pole dimenzí) a pak označení typu elementu Z..boolean, B..byte, C..char, D..double, F..float, I..int, J..long, S..short, Lclassname..třída nebo interface

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

java.lang.Class: metody

- `public URL getResource(String name)`
- `public InputStream getResourceAsStream(String name)`
  - načte nějaký „zdroj“
    - obrázky, ..., cokoliv
  - data načítá classloader => načítání se řídí stejnými pravidly jako načítání tříd
  - jméno „zdroje“ ~ hierarchické jméno jako u tříd
    - oddělovací tečky jsou nahrazeny lomítky `'/'`

java.lang.Class: metody

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

Použití Reflection API

- informace o kódu
- dynamické načítání
- pluginy
- proxy třídy
- ...

jar

Přehled

- vytváření archivů sdružujících .class soubory
- JAR ~ Java Archive
- soubor
  - přípona .jar
  - formát – ZIP
  - soubor META-INF/MANIFEST.MF
    - popis obsahu
- použití – distribuce softwaru
  - do CLASSPATH lze psát .jar soubory
  - lze přímo spouštět .jar soubory
- nemusí obsahovat jen .class soubory
  - obrázky
  - audio
  - cokoliv

Použití

- vytvoření archivu

```
jar cf soubor.jar *.class
```

  - vytvoří soubor.jar se všemi .class soubory
  - přidá do něj MANIFEST.MF soubor

```
jar cmf manifest soubor.jar *.class
```

  - vytvoří soubor.jar s daným MANIFEST souborem

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

  - nepoužije se komprese

  - pro další parametry viz dokumentaci

- práce s jar archivy v programu
  - java.util.jar, java.util.zip

MANIFEST.MF soubor

- seznam dvojic
  - jméno : hodnota
  - inspirováno standardem RFC822
- dvojice lze seskupovat do skupin
  - skupinu odděleny prázdným řádkem
  - hlavní skupina (první)
  - skupiny pro jednotlivé položky archivu
- délka řádků max. 65535
- konce řádků
  - CR LF, LF, CR

MANIFEST.MF soubor

- hlavní sekce
  - Manifest-Version
  - Created-By
  - Signature-Version
  - Class-Path
  - Main-Class
    - aplikace lze spouštět

    java -jar archiv.jar
- vedlejší sekce
  - první položka

    Name: cesta_k_položce_v_archivu

Jar a Ant

- task **jar** v Antu
  - parametry
    - destfile, basedir, includes, excludes, manifest
  - vnořené elementy
    - manifest
  - příklady

```
<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

- podobné jako java.util.zip

- `JarInputStream`, `JarOutputStream`
  - potomci ZipInputStream a ZipOutputStream
  - JarInputStream ma navíc metody getManifest()
- `JarEntry`
  - potomek `ZipEntry`
  - získávání atributů
- `Manifest`
  - reprezentace MANIFEST.MF souboru

Java

Moduly

Modularizace

- modul
  - explicitně definované co poskytuje i co ***požaduje***

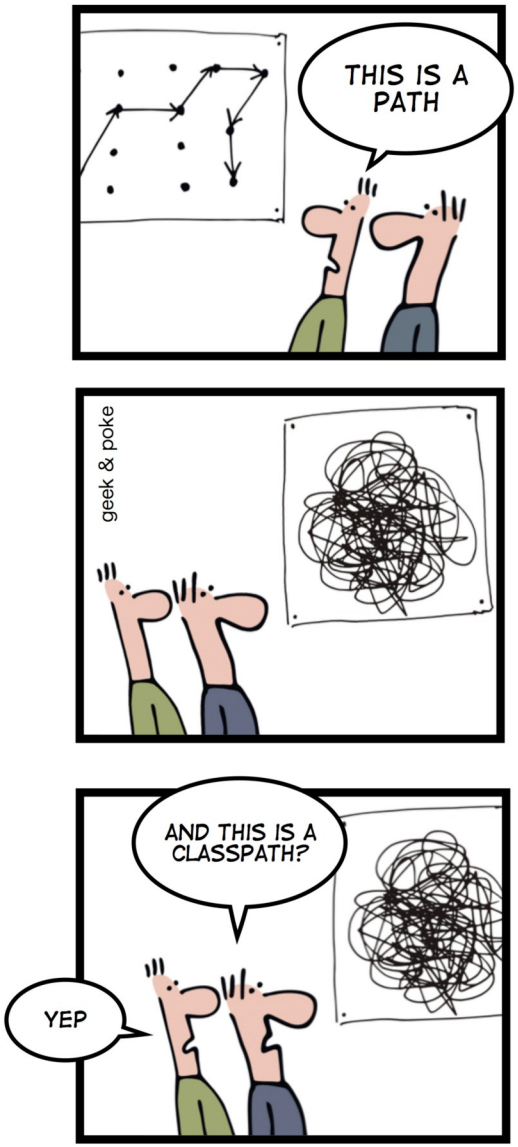
- proč
  - koncept *classpath* je „křehký“
  - chybí zapouzření

Modu

- modul
  - exp

ožaduje

- proč
  - kon
  - chyl

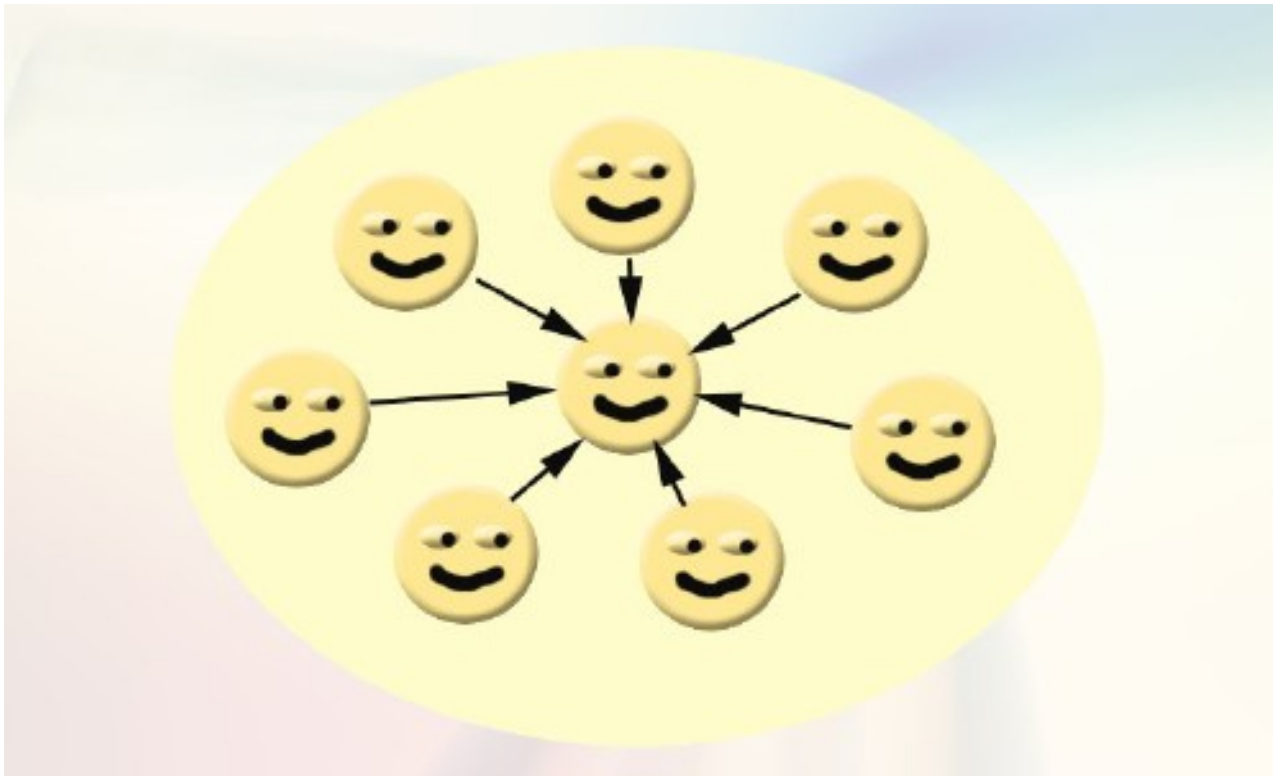


Modulární aplikace – motivace

- proč
  - aplikace více a více komplexní
  - skládání aplikací
  - vývoj v distribuovaných týmech
  - komplexní závislosti
  - dobrá architektura programu
    - ví o svých závislostech
    - spravuje závislosti

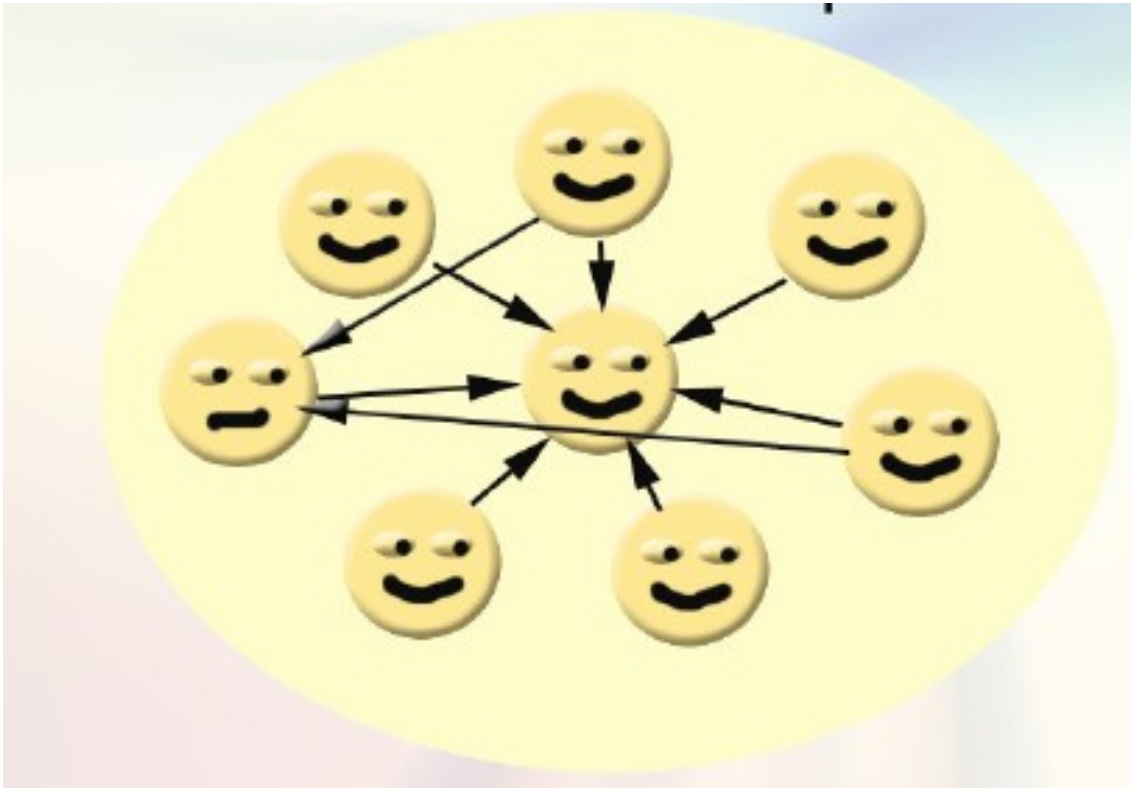
Modulární aplikace – motivace

- Verze 1.0 – vše dobře navrženo



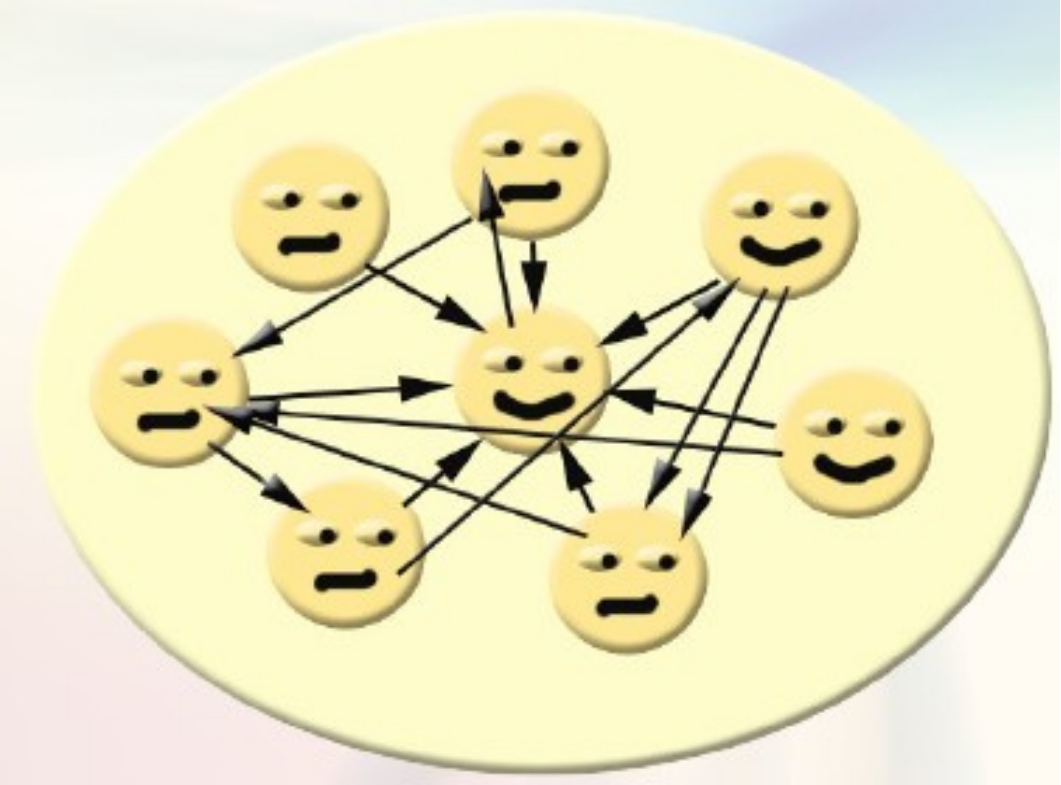
Modulární aplikace – motivace

- Verze 1.1...několik „vynálezavých hacků“...vyčistíme to ve 2.0



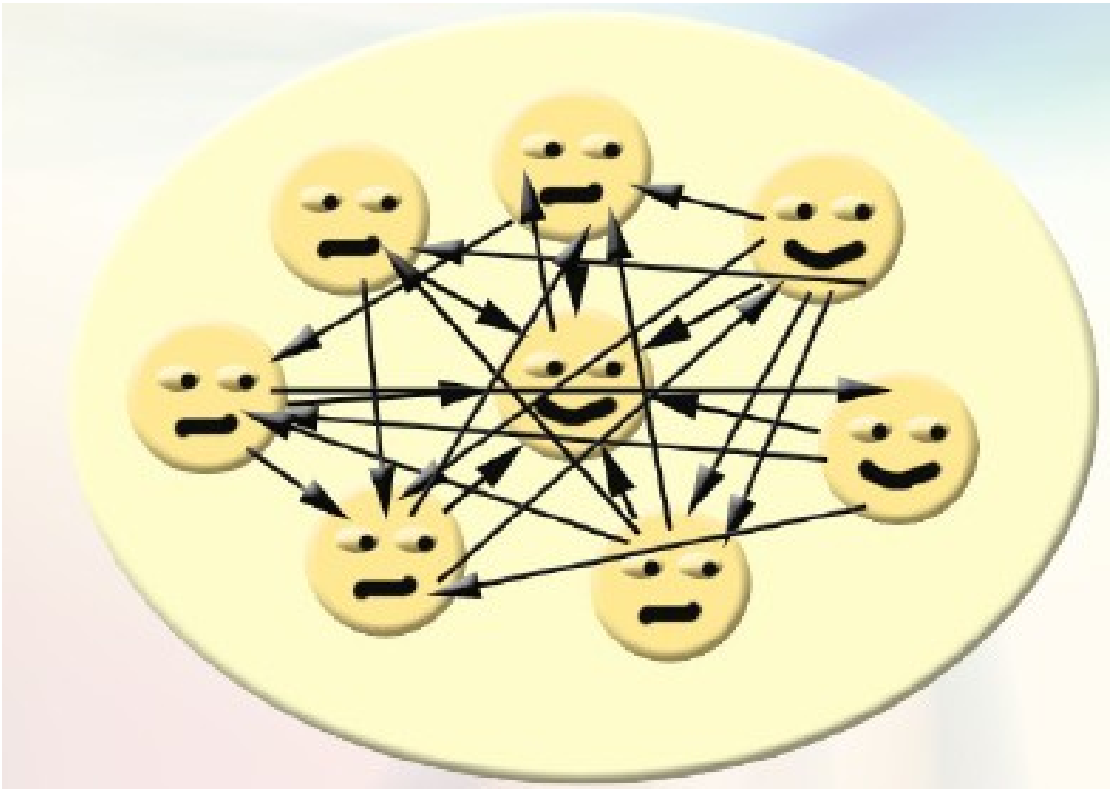
Modulární aplikace – motivace

- Verze 2.0...oops...ale...funguje to!



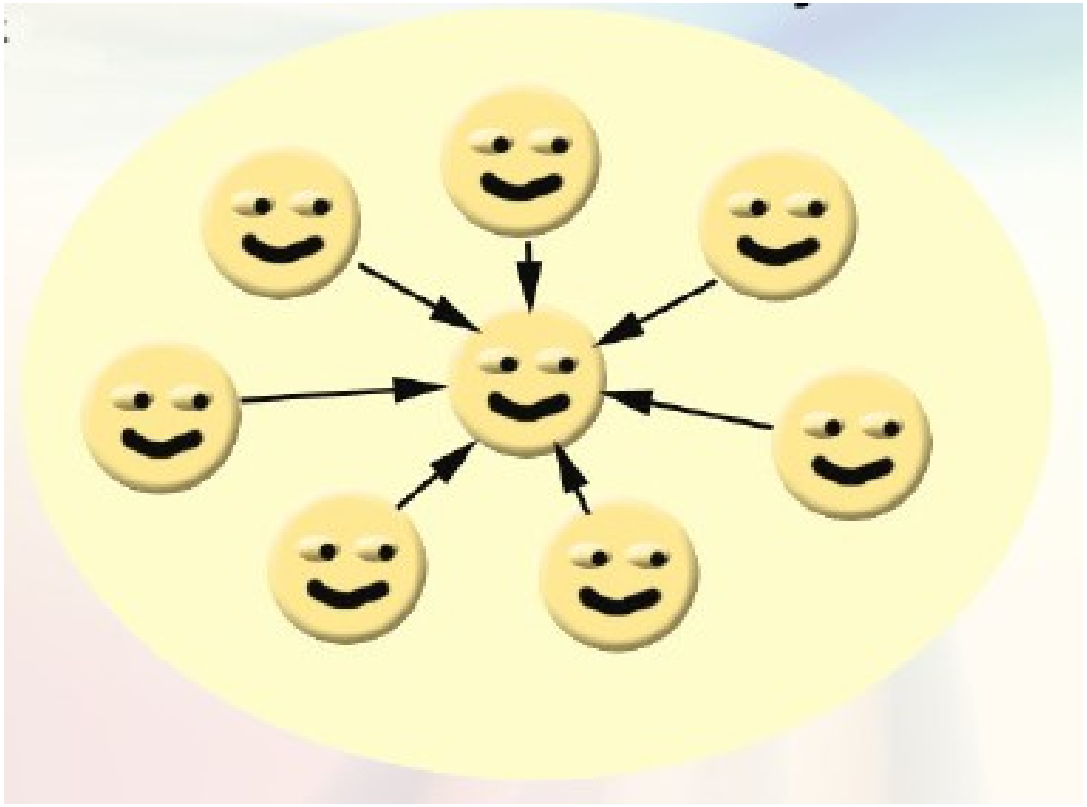
Modulární aplikace – motivace

- Verze 3.0 – Pomoc! Oprava jakékoliv chyby přinese dvě další chyby!



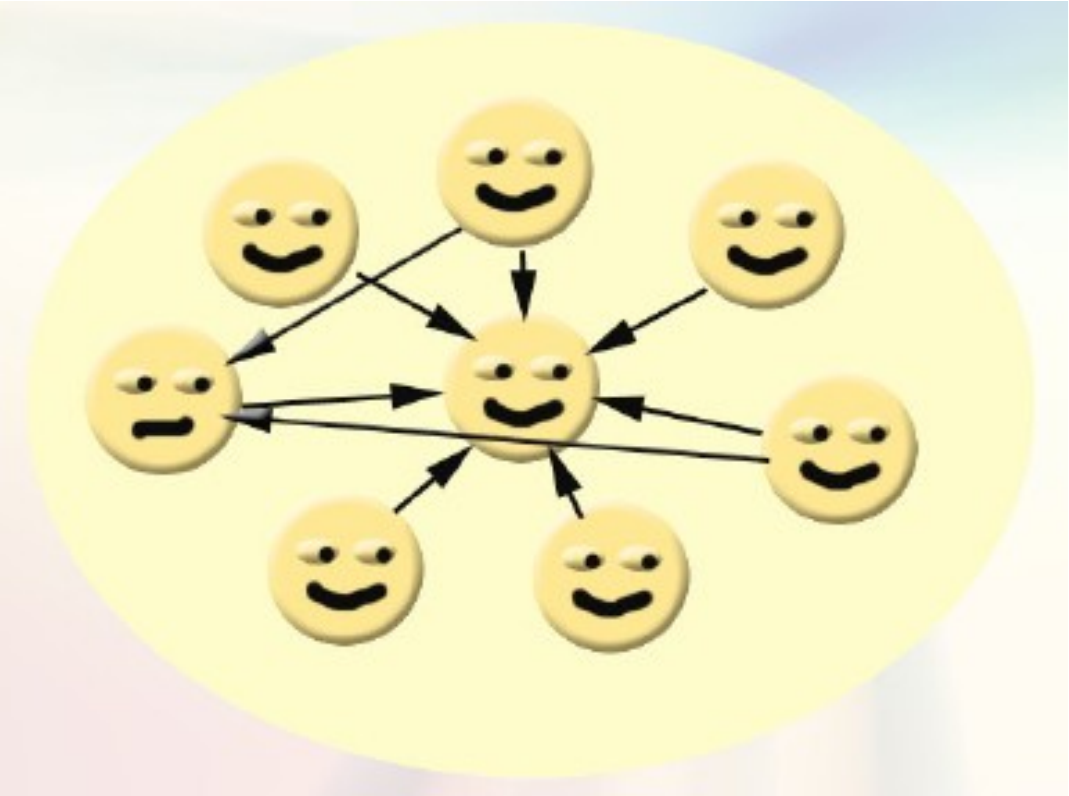
Modulární aplikace – motivace

- Verze 4.0 – vše dobře navrženo. Kompletně přepsano, trvalo to o rok delé, ale funguje to...



Modulární aplikace – motivace

- Version 4.1...tohle vypadá povědomě...



Deklarace modulu

- module-info.java

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

- modular artifact
  - modularní JAR – JAR obsahující module-info.class
  - nový formát JMOD
    - ZIP s třídami, nativním kódem, konfigurací,...

Moduly a JDK

- standardní knihovna JDK také modularní
  - java.base – vždy „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

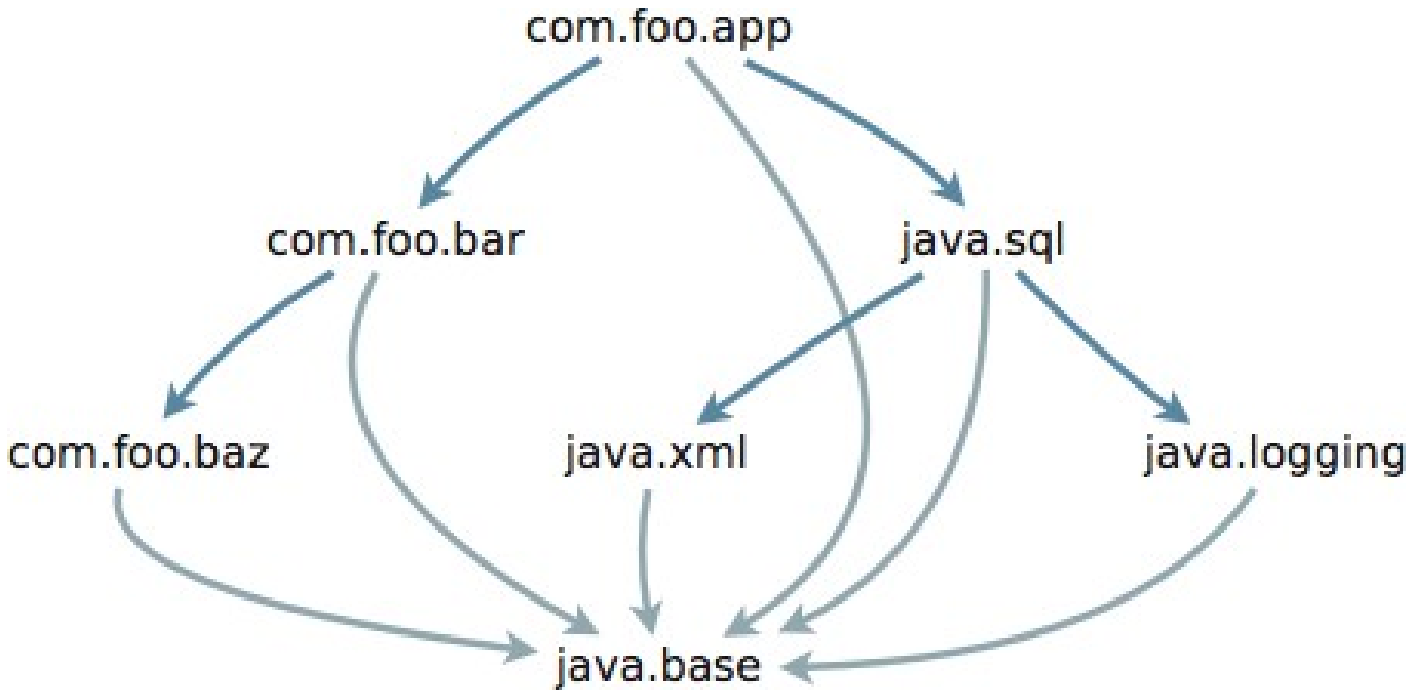
- Pokud modul přímo závisí na jiném modulu

  Modul ***čte (reads)*** jiný modul (nebo, jinak, druhý modul je ***čitelný (readable)*** prvním modulem)

- ***Module path*** – ekvivalent ke classpath
  - ale pro moduly
    - -p, --module-path

Module graph

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



Kompatibilita se „starou“ Javou

- Classpath stále podporováno
  - v podstatě jsou moduly „volitelné“

- Nepojmenovaný modul
  - cokoliv mimo jakýkoliv modul
    - „starý“ kód
  - čte jakýkoliv jiný modul
  - exportuje všechny svoje balíčky pro všechny jiné moduly