



# Alma Mater Studiorum-Università di Bologna Scuola di Ingegneria

---

## Moduli in Java Introduzione & esempi

*Corso di Laurea in Ingegneria Informatica*  
*Anno accademico 2021/2022*

**Prof. ENRICO DENTI**

*Dipartimento di Informatica – Scienza e Ingegneria (DISI)*



# MODULARIZZARE UN'APPLICAZIONE

---

- Modularizzare un'applicazione significa:
  - concettualmente stabilire quali e quanti moduli ci siano e *che dipendenze abbiano*
  - praticamente strutturare preventivamente l'app in *package* predisporre i vari **module-info.java** collocarli nella *directory base* dei package
- Non si può modularizzare un'applicazione senza package!
  - se l'applicazione non era strutturata su più package (orrore..)
  - ..oppure usava il package di default (doppio orrore..)
  - la prima cosa da fare è un bel *refactoring*!



# ESEMPIO 1: FRAZIONI (1/6)

- Il caso di studio **Frazione** fu sviluppato a inizio corso
  - niente package espliciti, tutto nel default package
- **Primo passo: refactoring** → ristrutturare con package
  - due package indipendenti (ma dal nome "affine"..)
  - `utils.math` per la classe **Frazione**
  - `utils.test` per il mini-main di prova

```
C:.\n└─utils\n    └─math\n        Frazione.class\n        Frazione.java\n    └─test\n        MyMain.class\n        MyMain.java
```



# ESEMPIO 1: FRAZIONI (2/6)

---

- Così riorganizzata, l'applicazione va compilata ed eseguita tradizionalmente, specificando (se occorre) il *class path*

- compilazione

```
javac utils\math\Frazione.java
```

```
javac utils\test\MyMain.java
```

- esecuzione (class path di default)

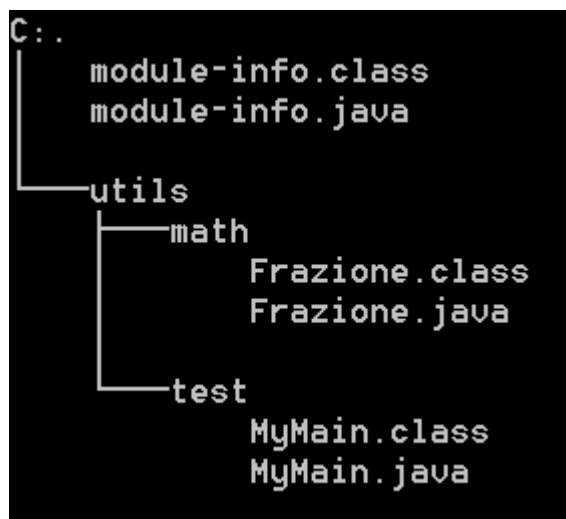
```
java utils.test.MyMain
```

oppure (class path esplicito):

```
java -cp . utils.test.MyMain
```

# ESEMPIO 1: FRAZIONI (3/6)

- **Secondo passo: modularizzare**
  - aggiunta del file **module-info.java** nella cartella base
  - nome modulo: **frazioni**  
volutamente diverso dai package (anche se fuori standard)  
per evidenziare dove si usi l'uno e dove l'altro
  - non esporta esplicitamente nulla, non richiede esplicitamente nulla



```
module frazioni { module-info.java
    // does not require anything
    // does not export anything
}
```



# ESEMPIO 1: FRAZIONI (4/6)

- Per compilare l'applicazione, occorre ora **compilare il modulo e tutti i package che esso usa:**

- compilazione

```
javac module-info.java utils\math\Frazione.java  
      utils\test\MyMain.java
```

- Per eseguire l'applicazione, occorre ora **specificare il *module path* e la *coppia modulo/classe* da eseguire**

- esecuzione con opzioni verbose (forma ***--optiontext***):

```
java --module-path .  
      --module frazioni/utils.test.MyMain
```

- esecuzione con opzioni shortcut (forma single-char ***-option***):

```
java -p . -m frazioni/utils.test.MyMain
```

# ESEMPIO 1: FRAZIONI (5/6)

- Per compilare l'applicazione occorre compilare il modulo e tutti i suoi sottomoduli

- compilazione con opzioni complete

```
javac --module-path Module path Module/class applicazione.java  
      --module Module/class  
      -p Module path  
      -m Module/class
```

- Per eseguire l'applicazione occorre specificare il *module path* e la *coppia modulo/classe* da eseguire

- esecuzione con opzioni complete (forma *--option*):

```
java --module-path module path modulo/classe modulo/classe  
     --module frazioni/utils.test.MyMain
```

- esecuzione con opzioni shortcut (forma single-char *-option*):

```
java -p module path -m modulo/classe modulo/classe  
      module path modulo/classe
```



# ESEMPIO 1: FRAZIONI (6/6)

- Ora tutto funziona:

```
Creata la frazione 3/12
Creata la frazione 1/4
Creata la frazione -1/8
Creata la frazione 1/8
Creata la frazione 4
Le frazioni 3/12 e 1/4 sono equivalenti
Le frazioni 3/12 e -1/8 non sono equivalenti
La frazione 3/12 ridotta ai minimi termini diventa 1/4
```





# FRAZIONI: UN PASSO IN PIÙ (1/5)

- Immaginiamo che il main sia diverso e usi alcuni elementi di grafica JavaFX

```
package utils.test;
import utils.math.Frazione;
import javafx.scene.control.Alert;
public class MyMain {
    public static void main(String[] args) {
        Alert alert = new Alert(Alert.AlertType.ERROR);
        Frazione frazione1 = new Frazione(3,12);
        ...
    }
}
```

- **JavaFX non fa parte del modulo `java.base`** richiesto ("importato") implicitamente di default, quindi *se non viene esplicitamente richiesto si ha errore di compilazione*



# FRAZIONI: UN PASSO IN PIÙ (2/5)

- Tentando di compilare come prima, si ottiene:

- compilazione

```
javac module-info.java utils\math\Frazione.java  
utils\test\MyMain.java
```

```
utils\test\MyMain.java:5: error: package  
javafx.scene.control is not visible  
import javafx.scene.control.Alert;  
                           ^  
  
(package javafx.scene.control is declared in module  
javafx.controls, but module frazioni does not read it)  
1 error
```

# FRAZIONI: UN PASSO IN PIÙ (3/5)

- Perché la compilazione abbia successo, occorre *richiedere esplicitamente il modulo di cui la classe **Alert** fa parte*
- Per analizzare le dipendenze si usa lo strumento **jdeps**

```
jdeps -s .  
frazioni -> java.base  
frazioni -> javafx.controls
```

-s fornisce  
output sintetico

- risulta che quella classe sia parte del modulo **javafx.controls**
- ergo, occorre inserire tale dipendenza in `module-info.java`

```
module frazioni {  
    requires javafx.controls;  
    // does not export anything  
}
```

module-info.java



# FRAZIONI: UN PASSO IN PIÙ (4/5)

---

- La compilazione ora ha successo:

```
javac module-info.java utils\math\Frazione.java  
utils\test\MyMain.java
```

- Di nuovo, per eseguire l'applicazione, occorre specificare il *module path* e la *coppia modulo/classe* da eseguire

```
java -p . -m frazioni/utils.test.MyMain
```

- Stavolta tutto fila e l'applicazione esegue normalmente.



# FRAZIONI: UN PASSO IN PIÙ (5/5)

- Senza `-s`, `jdeps` fornisce una analisi dettagliata:

```
jdeps .
frazioni [file:///C:/...lert/./]
  requires mandated java.base (@9.0.4)
  requires javafx.controls (@9.0.4)
utils.math    -> java.lang          java.base
utils.math    -> java.lang.invoke    java.base
utils.test    -> java.io             java.base
utils.test    -> java.lang           java.base
utils.test    -> java.lang.invoke    java.base
utils.test    -> javafx.scene.control javafx.controls
utils.test    -> utils.math          frazioni
```

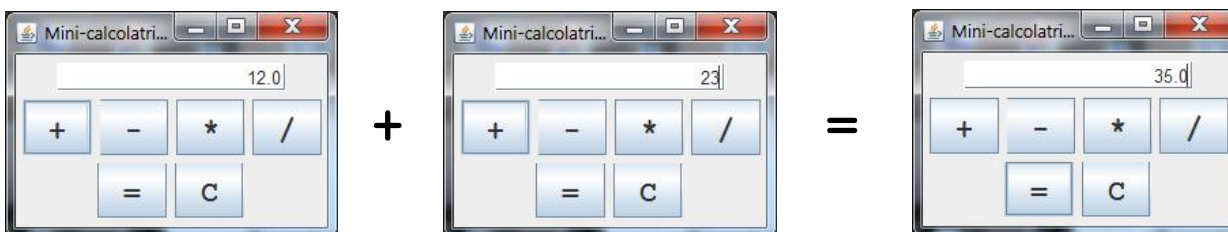
package richiesto

modulo che lo fornisce

# ESEMPIO 2:

## MINI-CALCOLATRICE IN SWING (1/3)

- **SwingCalculator** è una mini-calcolatrice scritta in Swing
  - package `calculator`



- Come JavaFX, anche AWT e Swing non sono inclusi nel modulo di default `java.base`
  - necessità di opportuna **requires** in fase di modularizzazione

```
module calc {  
    requires java.desktop;  
}
```

module-info.java



# ESEMPIO 2:

## MINI-CALCOLATRICE IN SWING (2/3)

- Tentativo di compilazione con `module-info` vuoto..

```
calculator\SwingCalculator.java:3: error: package java.awt is not
visible
import java.awt.*;
        ^
(package java.awt is declared in module java.desktop, but module
calc does not read it)

calculator\SwingCalculator.java:4: error: package javax.swing is
not visible
import javax.swing.*;
        ^
(package javax.swing is declared in module java.desktop, but module
calc does not read it)

calculator\SwingCalculator.java:5: error: package java.awt.event is
not visible
import java.awt.event.*;
        ^
(package java.awt.event is declared in module java.desktop, but
module calc does not read it)
...
```



# ESEMPIO 2:

## MINI-CALCOLATRICE IN SWING (3/3)

- Analisi dipendenze con `jdeps`

```
jdeps -s .  
. -> java.base  
. -> java.desktop  
  
jdeps .  
. -> java.base  
. -> java.desktop  
calculator -> java.awt          java.desktop  
calculator -> java.awt.event    java.desktop  
calculator -> java.io           java.base  
calculator -> java.lang         java.base  
calculator -> java.lang.invoke  java.base  
calculator -> javax.swing       java.desktop
```

- Soluzione:

<pre>module calc {     requires java.desktop; }</pre>	module-info.java
---	------------------





# MODULI AUTOMATICI

## ALGORITMO DI NAMING

- Un modulo automatico si chiama "circa" come il JAR, con *alcuni distinguo* per garantire che il nome ottenuto:
  - dev'essere un *valido nome di package* secondo la sintassi Java multi-livello (no caratteri illegali, no cifre iniziali, etc.)
  - non dev'essere legato a specifiche versioni (per quelle c'è l'attributo *version*)
- Questa specifica cattura l'idea che il nome del modulo debba rappresentarne lo scopo, non la versione
- Esempi
  - JAR `frazlib-12.jar` → modulo `frazlib` (attributo di versione: 12)
  - JAR `foo-bar-1.2.3-SNAPSHOT.jar` → modulo `"foo.bar"`  
(attributo di versione: "1.2.3-SNAPSHOT")



# MODULI AUTOMATICI

## ALGORITMO DI NAMING

- A tal fine, l'algoritmo di naming opera come segue:
  - se il nome contiene un trattino seguito da caratteri numerici
    - ossia, fa match con l'espressione regolare "`(\\d+ (\\. | $) )`"
  - allora il nome del modulo è solo la parte che precede il trattino
    - la parte successiva fa da numero di versione (o ignorata se ciò non è possibile)
  - poi, tutti i caratteri non alfanumerici sono sostituiti da puntini
    - gruppi di puntini solo collassati in un unico puntino
  - infine, tutti i puntini iniziali e finali vengono rimossi.
- E infatti, riconsiderando gli esempi:
  - JAR `frazlib-12.jar` → modulo `frazlib` (attributo di versione: 12)
  - JAR `foo-bar-1.2.3-SNAPSHOT.jar` → modulo `"foo.bar"`  
(attributo di versione: "1.2.3-SNAPSHOT")
  - JAR `2p.jar` → *nome illegale, modulo automatico non derivabile*



# MODULI AUTOMATICI POSSIBILI PROBLEMI...

- Se un JAR ha un nome *che non è un identificatore Java valido* (es. `2p.jar`), il tentativo di derivare un modulo automatico *fallirà*:

```
jar --file 2p.jar --describe-module
```

```
Impossibile derivare il descrittore di modulo per 2p.jar  
2p: Invalid module name: '2p' is not a Java identifier
```

- Possibili soluzioni:
  - se si è utente: cambiare nome al JAR ☺
  - se si è utente evoluto o creatore del JAR:
    - aggiungere al MANIFEST la riga **Automatic-Module-Name: *name***
  - se si è il creatore del JAR:
    - aggiungere al package un vero `module-info.java`



# MODULI AUTOMATICI .. E POSSIBILI SOLUZIONI

- Opzione 1: cambiare nome al JAR
  - ad esempio, da `2p.jar` a `twoP.jar`
  - `jar --file twoP.jar --describe-module`
  - Nessun descrittore di modulo trovato. Derivato modulo automatico.  
`twoP automatic (...)`
- Opzione 2: aggiornamento del JAR
  - riga da aggiungere: `Automatic-Module-Name: tuprolog.full`
  - file di testo con la riga extra: `manifest-automatic-name.txt`
  - invocazione del comando JAR in modalità update (**u**):  
`jar umf manifest-automatic-name.txt 2p.jar`
  - risultato: il modulo automatico ora viene estratto 😊
  - `jar --file 2p.jar --describe-module`
  - Nessun descrittore di modulo trovato. Derivato modulo automatico.  
`tuprolog.full automatic`

# MODULI AUTOMATICI .. E POSSIBILI SOLUZIONI

- Il manifest prima ...

```
1 Manifest-Version: 1.0
2 Ant-Version: Apache Ant 1.9.6
3 Created-By: 1.8.0_40-b27 (Oracle Corporation)
4 Implementation-Vendor: tuProlog team
5 Implementation-Title: tuProlog Engine
6 Implementation-Version: 3.2.1.0
7 Main-Class: alice.tuprologx.ide.GUILauncher
8 Class-Path: .
9
```

- ...e dopo

```
1 Manifest-Version: 1.0
2 Ant-Version: Apache Ant 1.9.6
3 Created-By: 1.8.0_40-b27 (Oracle Corporation)
4 Implementation-Vendor: tuProlog team
5 Implementation-Title: tuProlog Engine
6 Implementation-Version: 3.2.1.0
7 Main-Class: alice.tuprologx.ide.GUILauncher
8 Class-Path: .
9 Automatic-Module-Name: tuprolog.full
10
```

Non è stato necessario rifare il JAR con Java 9+ (es. Java 17)  
È bastato aggiornare il manifest.



# DA JAR CLASSICI A JAR MODULARI

---

- Più in generale, se necessario i vecchi JAR (non modulari) possono essere *aggiornati* a veri *JAR modulari*
- Procedura
  1. estrarre dal JAR tutte le dipendenze (eventualmente con `jdeps`)
  2. creare l'apposito `module-info.java`
  3. compilarlo con l'opzione `--patch-module`
  4. aggiornare il JAR aggiungendo la nuova dichiarazione di modulo con l'opzione `jar --update`



# ARGOMENTI AVANZATI

---

1. Export selettivi (**exports to**)
2. Dipendenze transitive (**requires transitive**)
3. Dipendenze a compile time (**requires static**)
4. Aperture a reflection (direttiva **opens**)
5. Generazione di applicazioni auto-contenute: **jlink**



# EXPORT SELETTIVI

- La direttiva **exports** può essere *selettiva*
  - è cioè possibile specificare *verso quali moduli si esporta*
  - con ciò, si limita e controlla a priori lo scope dell'esportazione
  - ma al contempo si inserisce una dipendenza verso un altro modulo

- Esempio

```
module ed.numbers {  
    exports myNumbers;  
    exports test to ed.main;  
}
```

- Scenari da verificare
  - chiunque deve poter accedere al package **myNumbers**
  - MA solo il modulo **ed.main** deve poter accedere al package **test**



# EXPORT SELETTIVI

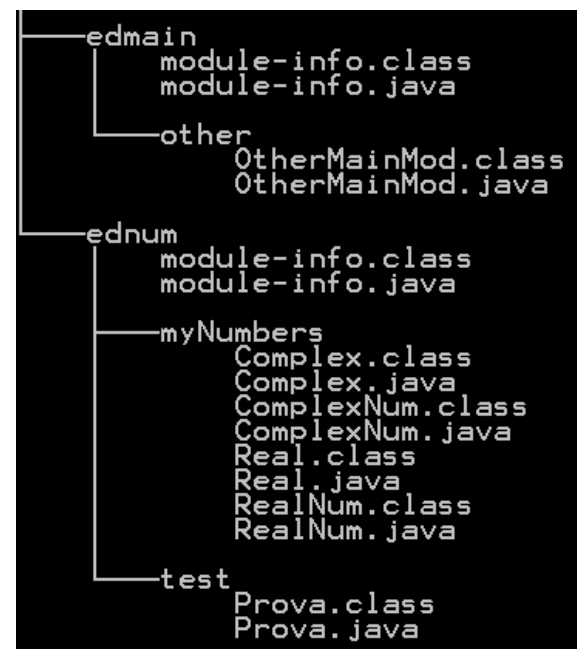
## ESEMPIO (1/4)

- Situazione

- il modulo **ed.main** contiene un package (non esportato) che richiede **ed.numbers**

```
module ed.numbers {  
    exports myNumbers;  
    exports test to ed.main;  
}
```

- non essendo esportato, il main non può accedere al package **test** e si ha errore:



```
javac -p "..\complex-real con interfacce (modularizzata)-modulo1"  
    module-info.java OtherMainMod.java
```

```
OtherMainMod.java:2: error: package test is not visible
```

```
import test.Prova;
```

```
    ^
```

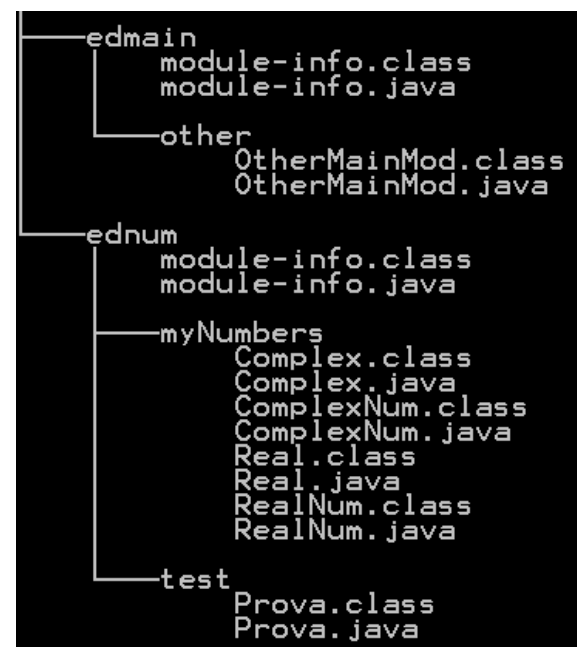
```
(package test is declared in module ed.numbers, which does not export it)
```



# EXPORT SELETTIVI

## ESEMPIO (2/4)

- Come vedremo, ora invece funzionerà
  - struttura a package, per pulizia
- Compilazione 1° modulo **ed.numbers**
  - giustamente viene emesso un warning perché il modulo destinazione, **ed.main**, non risulta visibile dalla cartella corrente
  - ma non è un problema..



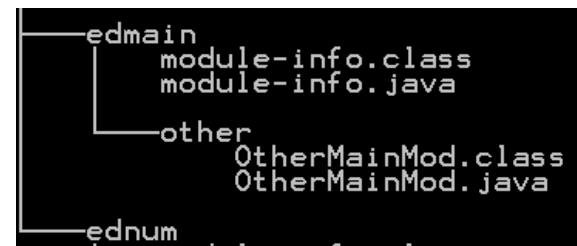
```
javac -p ednum ednum\module-info.java ednum\myNumbers\*.java ednum\test\*.java
module-info.java:3: warning: [module] module not found: ed.main
    exports test to ed.main;
                        ^
1 warning
```



# EXPORT SELETTIVI

## ESEMPIO (3/4)

- Come vedremo, ora invece funzionerà
  - struttura a package: per pulizia, la classe **OtherMainMod** va nel package **other**
- Compilazione 2° modulo **ed.main**



```
javac -p edmain;ednum edmain\module-info.java edmain\other\OtherMainMod.java
```

- Esecuzione
  - poiché la classe **OtherMainMod** fa parte del package **other**, che a sua volta appartiene al modulo **ed.main**, il lancio deve prevedere la corretta sintassi *per entrambi*:

```
java -p edmain;ednum -m ed.main/other.OtherMainMod
```

*modulename/qualifiedclassname*

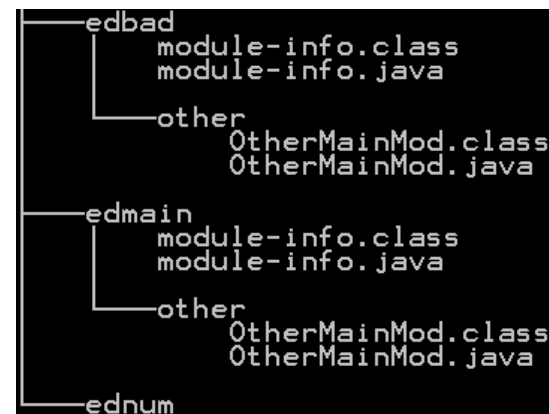
```
r1 + r2 = 21.64
c1 + c2 = -13.0+2.0i
c + c3 = -13.0+0.0i
c = r; c = 21.64
```



# EXPORT SELETTIVI

## ESEMPIO (4/4)

- Controprova
  - modulo **ed.bad**, clone di **ed.main**
- Compilazione 3° modulo **ed.bad**
  - ERRORE, perché il modulo **ed.numbers** esporta il package **test** solo a **ed.main** e non ad altri moduli



```
javac -p edbad;ednum edbad\module-info.java edbad\other\OtherMainMod.java
edbad\other\OtherMainMod.java:3: error: package test is not visible
import test.Prova;
      ^
```

```
(package test is declared in module ed.numbers, which does not export it to
module ed.bad)
```

```
1 error
```

# DIPENDENZE TRANSITIVE

- La direttiva **requires** di norma *non è transitiva*
  - il fatto che il modulo A richieda il modulo B e il modulo B richieda il modulo C *non significa* che il modulo A richieda implicitamente C
  - motivo: rendere esplicite tutte le dipendenze
  - talora però ciò può dar luogo a lunghe elencazioni tediose
- Per questo, la direttiva **requires transitive** consente di specificare una *dipendenza transitiva*
  - da usare quando è scontato che chi usi A abbia bisogno anche di B
  - ad esempio, poiché il modulo `javafx.controls` implica l'uso di `javafx.base`, la sua dichiarazione è transitiva:

```
module javafx.controls {  
    requires transitive javafx.base;  
    ...  
}
```



# DIPENDENZE TRANSITIVE

## SCENARI D'USO TIPICI

---

- Scenario 1: quando un package di un altro modulo è usato *in una API pubblica*
  - in tal caso, non si può presumere che l'utente dell'API pubblica debba conoscere le dipendenze "interne" del modulo
- Scenario 2: *moduli aggregatori*
  - sono moduli che contengono solo dipendenze transitive
  - aggregano selettivamente altri moduli per fornire una view / API
  - ad esempio, *il modulo `java.se`* è un aggregatore di molti altri, utile per l'utente non interessato all'articolazione fine interna del JRE

# DIPENDENZE STATICHE

- Di rado, ma può accadere che un modulo sia richiesto *a compile time* ma *non necessariamente a run time*
  - Scenario 1: usare una classe in un dato modulo nel caso esso sia presente, *o altrimenti fare qualcos'altro (che non la usa)*
  - Scenario 2: usare un'annotazione dichiarata in un altro modulo
- A questo serve la direttiva **requires static**
  - esempio di scenario del primo tipo:

```
...  
try {  
    new oracle.jdbc.driver.OracleDriver();  
    ...  
} catch (ClassNotFoundException e) {  
    ... // fai altro  
}
```

Se la classe non c'è,  
a runtime il modulo  
non è necessario



# ACCESSO TRAMITE REFLECTION

- Fino a Java 8, era di norma possibile *aggirare i qualificatori di accesso tramite reflection*
  - campi privati accessibili tramite il metodo `setAccessible(true)` a meno che un apposito security manager lo vietasse (raro)
  - chiara violazione di incapsulamento, a volte utile/necessaria per
    - white-box testing
    - sfruttare API non documentate ma "de facto standard"
- Da Java 9+, l'accesso ai campi non pubblici di una classe *appartenente a un modulo è invece impedito*
  - `setAccessible(true)` lancia `InaccessibleObjectException`
  - enforcing di *protezione e incapsulamento*
  - MA certe vecchie librerie ne hanno bisogno per funzionare... ☹
    - **IDEA: rilassare il vincolo in modo selettivo**





# ACCESSO TRAMITE REFLECTION

- La **direttiva opens** apre un **package** alla reflection
  - protezione e incapsulamento coniugate con *apertura controllata*
  - questo esempio consente al modulo **myapp** di accedere, tramite reflection, ai campi non pubblici dei tipi definiti nel **packageXYZ**

```
module myapp {  
  requires ...;  
  opens packageXYZ;  
  ...  
}
```

Naturalmente, per coerenza, solo un *package esportato* può essere anche *aperto* !

- Il **qualificatore open** apre **tutti i package** di un modulo

```
open module myapp {  
  requires ...;  
  ...  
}
```

Equivale a *esportare e aprire* tutti i package del modulo



# GENERAZIONE DI APPLICAZIONI AUTO-CONTENUTE

- Per ovviare alla pesantezza del JRE in applicazioni embedded, si può *generare una versione ridotta del JRE* contenente *le sole classi e i soli JAR effettivamente usati*
  - tale *runtime image* è solitamente *molto* più leggera
  - l'applicazione risultante può essere distribuita ed eseguita *in modo autonomo*: non richiede che l'utente installi preventivamente il JRE
- Lo strumento **jlink** genera l'applicazione embedded
  - analizza quali classi l'applicazione usi realmente e genera la *runtime image* corrispondente
  - tecnicamente, costruisce il *minimo insieme di moduli* che l'applicazione richiede per funzionare (mettendoli in `lib/modules`)
  - **NON funziona con JAR vecchio stile (non modularizzati)**
    - opera *solo* con modular JAR, file JMOD, o moduli "esplosi"



# jlink: PECULIARITÀ

- A differenza degli altri strumenti, **jlink** impone **requisiti extra sui nomi delle cartelle** in cui operare
  - se non si seguono le sue (strane) convenzioni, dà sempre errore
- In particolare, occorre che la cartella in cui ci si trova si chiami come il modulo
  - non basta che si chiami così la cartella "sottostante"
  - ciò dà luogo a una *struttura doppiamente nidificata* che a volte sembra un "assurdo" doppione (es. **frazioni/frazioni/...**)
  - (..che sia un bug dell'implementazione di Java 9..?)
    - Check: verificare con Java 17!



# LO STRUMENTO `jlink`

- Sintassi

```
jlink --module-path <modulepath>  
--add-modules <modules>  
--limit-modules <modules>  
--output <path>
```

Limita il link ai moduli citati, escludendo quelli eventualmente aggiunti per il tramite di *dipendenze transitive*.

- Esempio 1 (v. oltre):

```
jlink --output myout  
--module-path "C:...\jdk-17.0.1\jmods";frazioni  
--add-modules frazioni
```

- Esempio 2 (con alias di lancio):

```
jlink --output myout  
--module-path "C:...\jdk-17.0.1\jmods";frazioni  
--add-modules frazioni  
--launcher start=frazioni/frazioni.MyMain
```

# ESPERIMENTI CON `jlink` (1/4)

- **Esempio 1: modulo `frazioni`**

- una prima cartella `frazioni` contiene il `module-info`
- una seconda cartella `frazioni` al suo interno contiene il vero modulo



- **Compilazione (occhio al primo livello!):**

```
javac -d frazioni frazioni\module-info.java
frazioni\frazioni\Frazione.java
frazioni\frazioni\MyMain.java
```

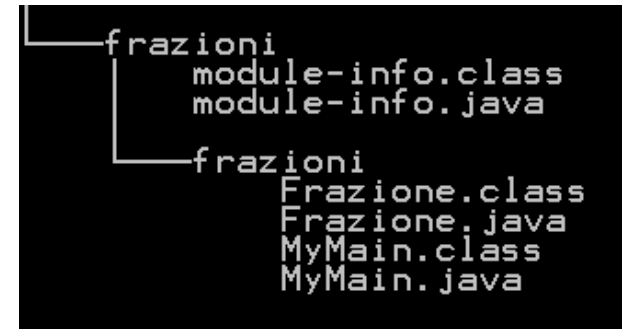
- **Generazione immagine**

- i moduli di sistema si trovano nella sottocartella **jmods** del JDK
- vogliamo l'immagine in **myout** (che non deve già esistere)

# ESPERIMENTI CON `jlink` (2/4)

- **Esempio 1: modulo `frazioni`**

- una prima cartella `frazioni` contiene il `module-info`
- una seconda cartella `frazioni` al suo interno contiene il vero modulo
- i moduli di sistema si trovano nella sottocartella **`jmods`** del JDK
- vogliamo l'immagine in `myout` (che non deve già esistere)
- predisponiamo anche un *alias di lancio* (opzione `--launcher`)



- **Generazione immagine**

```
jlink --output myout --module-path "C:\Program  
Files\Java\jdk-17.0.1\jmods";frazioni  
--add-modules frazioni  
--launcher start=frazioni/frazioni.MyMain
```

# ESPERIMENTI CON jlink (3/4)

- Risultato: 36 MB  
vs 215 MB del JRE

- Lancio:

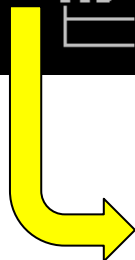
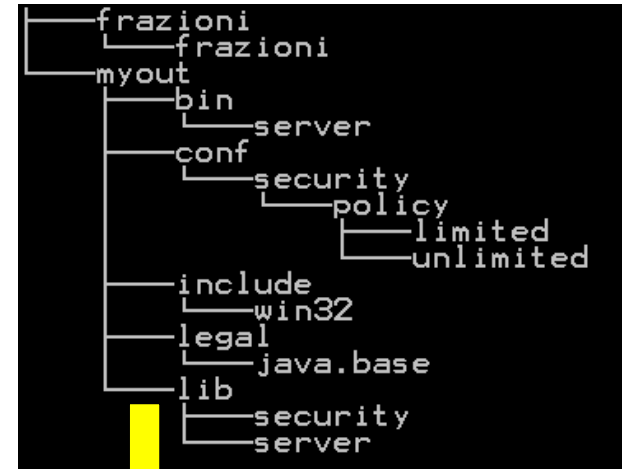
`myout\bin\start`

- il launcher

```
@echo off
set JLINK_VM_OPTIONS=
set DIR=%~dp0
"%DIR%\java" %JLINK_VM_OPTIONS% -m frazioni/frazioni.MyMain %*
```

- output:

```
>myout\bin\start
Creata la frazione 3/12
Creata la frazione 1/4
Creata la frazione -1/8
Creata la frazione 1/8
Creata la frazione 4
Le frazioni 3/12 e 1/4 sono equivalenti
Le frazioni 3/12 e -1/8 non sono equivalenti
La frazione 3/12 ridotta ai minimi termini diventa 1/4
```



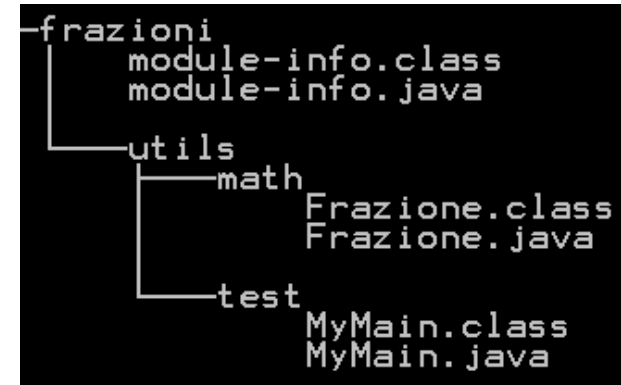
```
lib
  classlist
  jrt-fs.jar
  jvm.cfg
  jvm.lib
  modules
  tzdb.dat
  tzmappings
```

# ESPERIMENTI CON jlink (4/4)

- **Esempio 2: modulo frazioni**

*articolato in due package non omonimi*

- una prima cartella **frazioni** contiene il **module-info**
- una seconda cartella **utils** al suo interno contiene le cartelle **math** e **test**



- **Compilazione (occhio al primo livello!):**

```
javac -d frazioni frazioni\module-info.java
frazioni\utils\math\Frazione.java
frazioni\utils\test\MyMain.java
```

- **Generazione immagine (come prima, salvo il launcher):**

```
jlink ... --add-modules frazioni
--launcher start=frazioni/utils.test.MyMain
```