



Alma Mater Studiorum-Università di Bologna
Scuola di Ingegneria

Il problema della distribuzione (deployment) di un'applicazione o libreria

Corso di Laurea in Ingegneria Informatica
Anno accademico 2021/2022

Prof. ENRICO DENTI

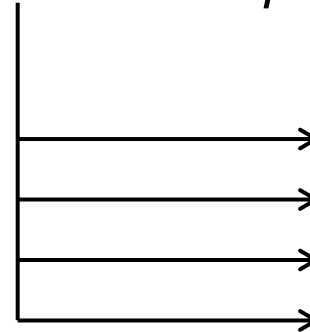
Dipartimento di Informatica – Scienza e Ingegneria (DISI)

IL DEPLOYMENT DEL SOFTWARE

- In questi linguaggi, non esiste più l'eseguibile monolitico.
- Come distribuire un'applicazione o una libreria *fatte di più classi* in modo pratico ed efficiente?
 - **IDEA: impacchettarle in un "archivio compresso"**
 - in questo modo, il file da distribuire è uno solo..
 - .. ma *internamente ogni entità mantiene la propria individualità*

In Java: **JAR (Java ARchive)**
(usato anche da Scala e Kotlin)

In C#: **.NET assembly**
(dal 2019: NuGet in .NET)



☺ **Non occorre scompattarli per usarli** ☺
L'infrastruttura li può usare *così come sono!*

IL DEPLOYMENT DEL SOFTWARE

Cosa occorre distribuire?

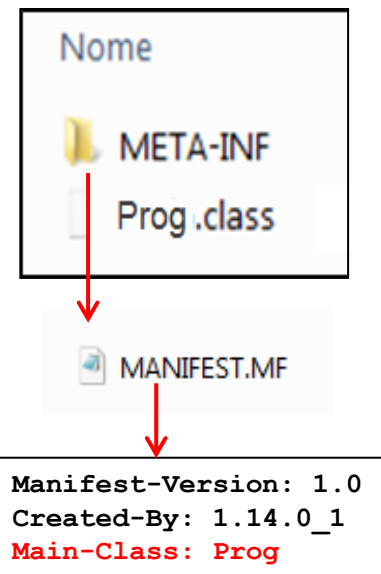
- Per **librerie**, bastano le classi
- Per **applicazioni**, occorre anche specificare *la posizione del main* (cioè, l'*entry point* dell'applicazione stessa)

- **In Java**: applicazioni e librerie sono distribuite *sempre come JAR* (con/senza specifica main)
- **In .NET**: applicazioni e librerie sono distribuite *con due tipi di assembly diversi: EXE o DLL* (sempre *DLL in .NET Core e .NET 5.0*)
- **In Scala, Kotlin**: sempre JAR, come in Java



JAVA: IL FORMATO JAR

- **JAR (Java ARchive)** è un file compresso ZIP
 - si apre con Unzip, 7-Zip, etc.
- che però contiene al suo interno una **speciale cartella META-INF con un file MANIFEST.MF**, che mantiene una serie di *informazioni extra*
 - per le applicazioni: *la posizione del main*
- Ciò rende possibile eseguire un'applicazione Java semplicemente *facendo doppio clic sul file JAR*, come in un eseguibile tradizionale
NB: con OpenJDK occorre prima configurare l'installazione



Gli archivi JAR si creano con lo strumento **jar incluso nel JDK.**

- È uno strumento a riga di comando: occorre apprenderne l'uso
- Analoghe funzionalità sono offerte dagli ambienti integrati (Eclipse)



CREARE UN FILE JAR (1/3)

Per creare un file JAR come archivio di classi (zip):

```
jar cf nomearchivio.jar classi
```

dove *classi* è

- un elenco di **classi** (.class) da includere
- o anche intere **cartelle** (pippo/*.class)

separate fra loro da "punto e virgola" in Windows

(Esempio: *.class;pippo/*.class)

o da "due punti" su Unix/Linux/Mac (*.class:pippo/*.class)

MA:

- un JAR così costruito va bene *per distribuire una libreria*
- *non è adatto a un'applicazione*, perché manca l'indicazione sulla posizione del main



JAVA: CREARE UN FILE JAR (2/3)

Per creare un file JAR eseguibile:

```
jar cmf info.txt nomeapp.jar classi
```

dove **info.txt** (che può chiamarsi in qualunque modo) è un *file di testo* che contiene la riga:

```
Main-Class: NomeclasseMain
```

seguita da una *riga vuota* (indispensabile!)

NB: da Java 9 in poi si può anche specificare la classe contenente il main *direttamente nella riga di comando*, senza file di testo ausiliari:

```
jar cef NomeclasseMain nomeapp.jar classi
```

Inoltre, sempre da Java 9, in alternativa ai comandi "mono-lettera" (c,e,m,f), si possono usare *comandi con nomi più descrittivi* introdotti da **--**, ad esempio:

```
jar --create --main-class NomeclasseMain --file nomeapp.jar classi
```



JAVA: CREARE UN FILE JAR (3/3)

Un JAR così creato è eseguibile

- perché contiene in sé l'indicazione sulla posizione del main

Come lo si esegue?

- o da riga di comando: `java -jar nomefile.jar`
- o semplicemente facendo *"doppio clic"* sul JAR stesso

In tal caso, però, l'applicazione sarà eseguita *senza aprire una finestra di terminale* → *se l'app non apre una finestra grafica (o non genera file)* non si vedrà niente!

IMPORTANTE: con OpenJDK, per far funzionare il «doppio clic» occorre prima completare la configurazione nel sistema operativo.
→ vedere le slide apposite



JAVA: ESEMPIO

Applicazione **MyApp** costituita da:

- `CodFisc.class`
- `Esempio.class` (*con il main*)

File .class già compilati
(solo da zippare)

CREAZIONE FILE JAR ESEGUIBILE

```
jar cmf infoApp MyApp.jar Esempio.class CodFisc.class
```

oppure, se si trovano tutti e soli nella stessa cartella:

```
jar cmf infoApp MyApp.jar *.class
```

Contiene la riga:
Main-Class: Esempio

MA se **Esempio** non è un'app grafica,
lanciando il JAR non vedremo niente



JAVA: ESEGUIRE UN FILE JAR

Per questo può essere utile eseguire un'applicazione distribuita come JAR *da riga di comando*

- Anziché

```
java NomeClasseColMain
```

- si può scrivere:

```
java -jar nomefile.jar
```

Questo approccio funziona anche se l'app non è grafica, perché sfrutta la finestra (già presente) del terminale da cui l'applicazione viene lanciata.

```
C>java -jar MyMain.jar  
1
```



C#: IL FORMATO "ASSEMBLY"

- In C#, compilazione e generazione dell'assembly sono *entrambe affidate al compilatore csc*
 - in .NET Core si utilizza il comando `dotnet` + argomenti
- Pertanto, è necessario specificare subito se si desidera:
 - un assembly per *applicazione eseguibile (con main)*
 - il file prodotto ha estensione **EXE** (default)
 - un assembly per *libreria (senza main)*
 - il file prodotto ha estensione **DLL**
 - da specificare con direttiva **`/target:library`**
- In .NET 5 (e .NET Core) lo strumento **NuGet** impacchetta DLL in modo del tutto simile ai JAR di Java
 - un pacchetto NuGet è un file ZIP con estensione **.nupkg** che contiene DLL + manifest



C#: CREARE UN "ASSEMBLY"

Per creare un assembly di *applicazione eseguibile*:

```
csc Classe1.cs Classe2.cs ...
```

Di default, **csc** presuppone che *il main sia nel primo file*: ergo produce un eseguibile di nome **Classe1.exe**

- per indicare un nome diverso: **csc /out:MyProg.exe ...**
- per indicare che il main è altrove: **csc /main:Classe2 ...**

Per creare un assembly di *libreria*:

```
csc /target:library Classe1.cs Classe2.cs ...
```

Di default, **csc** produce un eseguibile di nome **Classe1.dll**

- per indicare un nome diverso: **csc /out:MyLib.dll ...**



USARE UNA LIBRERIA

- Per usare una libreria (nostra o di altri), occorre sapere:
 - *come si compila un programma* che usi una libreria
 - *come si esegue un programma* che usi una libreria
- Quindi, *sia il compilatore sia l'infrastruttura* dovranno sapere *dove trovare* quella libreria
 - il compilatore, per accertarsi che il programma la *usi* correttamente
 - l'infrastruttura, per potersi *dinamicamente agganciare* ad essa

IN COMPILAZIONE:

- in Java: *opzione -cp*
- in C#: *opzione /reference*

IN ESECUZIONE:

- in Java: *opzione -cp*
(o specifica `Class-Path` nel file Manifest)
- in C#: nessuna opzione
(la libreria dev'essere in posizioni prestabilite)



USARE UNA LIBRERIA

Java

- libreria: **Lib.jar**
- cliente: **MyMain.java**

Compilazione

- se la libreria è nella stessa cartella:
`javac -cp Lib.jar MyMain.java`
- se la libreria è altrove:
`javac -cp ../Lib.jar MyMain.java`

Esecuzione

- se il main è una classe stand-alone:
`java -cp ../Lib.jar;. MyMain`
- se il main è in un JAR, la libreria va specificata nel corrispondente Manifest:
Class-Path: ../Lib.jar

C#

- libreria: **Lib.dll**
- cliente: **MyMain.cs**

Compilazione

- se la libreria è nella stessa cartella:
`csc /reference:Lib.dll MyMain.cs`
- se la libreria è altrove:
`csc /lib:... /reference:Lib.dll MyMain.cs`

Esecuzione

- semplicemente lanciando **MyMain.exe**
- la libreria può trovarsi solo nella cartella corrente o nella *Global Assembly Cache (GAC)*, una sorta di archivio centralizzato delle librerie.
Non sono ammesse altre posizioni.



ESEMPIO COMPLETO

OBIETTIVI

1. **Creare una libreria `CFLib`** costituita da una (unica) classe `CodFisc` che offre servizi per il calcolo e la verifica del codice fiscale
2. **Compilare un'applicazione `Prog`** che la usi
3. **Eseguire tale applicazione** in due scenari opposti
 - in presenza della libreria → funzionamento corretto
 - in assenza della libreria → errore a run-time

DEPLOYMENT

Verranno comunque distribuiti *due JAR (o assembly) separati*

- **NB: Java non consente di annidare archivi JAR uno dentro l'altro**, neppure usando opzioni come `Class-Path`: per superare questa limitazione occorrono tool di terze parti, come *One-Jar* (<http://one-jar.sourceforge.net>)
- In alternativa si può fare un JAR unico con dentro tutte le classi, *ma spesso la licenza d'uso della libreria altrui non lo consente (visibilità e riconoscibilità)*



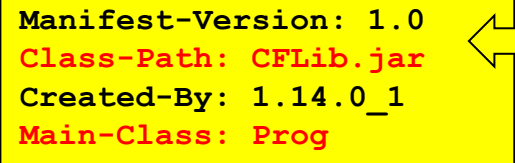
ESEMPIO COMPLETO: caso Java

1. Java: creazione libreria CFLib

```
javac CodFisc.java                (genera CodFisc.class)  
jar cf CFLib.jar CodFisc.class    (genera CFLib.jar)
```

2. Java: compilazione applicazione Prog

```
javac -cp CFLib.jar Prog.java      (genera Prog.class)  
jar cmf info.txt Prog.jar Prog.class (con Class-Path in info.txt)
```



```
Manifest-Version: 1.0  
Class-Path: CFLib.jar  
Created-By: 1.14.0_1  
Main-Class: Prog
```

3. Java: esecuzione applicazione Prog

```
java -cp CFLib.jar;. Prog          oppure  
java -jar Prog.jar                 (richiede comunque la presenza di CFLib.jar)
```

Java: se a runtime manca il JAR o la classe richiesta:

```
Exception in thread "main" java.lang.NoClassDefFoundError:  
CodFisc at Prog.main(Prog.java:3)
```



ESEMPIO COMPLETO: caso C#

1. C#: creazione libreria CFLib

```
csc /target:library /out:CFLib.dll CodFisc.cs
```

2. C#: compilazione applicazione Prog

```
csc /reference:CFLib.dll /out:Prog.exe Prog.cs
```

3. C#: esecuzione applicazione Prog

Prog.exe *(CFLib.dll deve essere nella stessa cartella)*

C#: se a runtime manca la DLL:

```
Unhandled Exception: System.IO.FileNotFoundException:
Could not load file or assembly 'CodFisc, Version=0.0.0.0'
or one of its dependencies.
Cannot find the file specified at Prog.Main(string[] args)
```


RIASSUMENDO...

Per produrre un JAR o un assembly:

JAVA: PROCESSO IN 2 TEMPI

- prima si compilano le classi
- poi si zippano nel JAR i file `.class` così ottenuti
(facendo includere nel *manifest* tutte le informazioni accessorie necessarie)

C# : PROCESSO UNICO

- si compilano le classi tutte insieme, generando un assembly EXE o DLL
- In .NET Core (o .NET 5.0): lo strumento NuGet costituisce il secondo passo

Struttura dell'applicazione risultante:

JAVA: (INSIEME DI) JAR

- l'applicazione è distribuita sotto forma di uno o più JAR
- non occorre decomprimere il JAR per eseguirlo
- se si distribuiscono più JAR zippati insieme (o tramite un installer), occorre unzippare / installare

C#: (INSIEME DI) ASSEMBLY

- l'applicazione distribuita sotto forma di assembly EXE + eventuali DLL
- non occorre decomprimere gli assembly per eseguirli
- se si distribuiscono più assembly zippati insieme, occorre unzippare / installare (in .NET Core → NuGet)