Alma Mater Studiorum-Università di Bologna
Scuola di Ingegneria

Gestione dell'I/O in Java
Parte 0: Introduzione

Corso di Laurea in Ingegneria Informatica

Anno accademico 2021/2022

Prof. ENRICO DENTI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

UN PO' DI STORIA (1)

- Il package originale: **`java.io`**
  - classe **`File`** come percorso (es. "**`/mnt/foo.txt`**")
  - concetto di *stream* come *canale unidirezionale* di I/O
  - architettura "a cipolla" basata sui due pattern *adapter* e *chain of responsibility*
    - package evoluto e arricchito negli anni
    - ancor oggi molto usato, rimane la base per tutto il resto
- Nuovi package per nuove astrazioni
  - Java 4: nuovo package **`java.nio`** con *nuove astrazioni* per scenari specifici (*buffer*, *charset, ..*)
    - ulteriormente arricchito in Java 5 & 6
  - Java 6: nuova astrazione *console*
    - semplifica nettamente l'I/O da console, astraendo dai dettagli

UN PO' DI STORIA (2)

- Java 7: NIO 2.0
  - nuovo package **`java.nio.file`**
    - gestione completa dell'I/O da file (incluso *zip file system*)
    - nuova architettura
    - nuove classi factory & libreria (**`Files`**, **`Paths`**)
    - nuovo entry point **`Path`** (al posto di **`File`**)
  - nuova sintassi per le eccezioni (*try-with-resources)* per semplificare gli scenari più tipici di gestione I/O
- Java 8
  - non aggiunge direttamente novità all'I/O
  - MA introduce lo *stream di operazioni* come nuova potente astrazione multi-uso

    → *Non confondere con gli stream di I/O !!*

Le basi: `File` & `Path`

L'ASTRAZIONE `File` IN `java.io`

`File` è una delle astrazioni-base del package **`java.io`**

- modella un file o una directory
  - il costruttore ha per argomento il nome del file o directory
- il file (o directory) modellato *può non esistere*
  - l'obiettivo è infatti rappresentare un qualunque file (o directory) *che possa esistere* nel sistema, non *"che esista ADESSO"*
- sono offerti metodi per
  - verificare se il file o directory esiste, o se è un file o una directory
  - operare sul file (delete, create, etc)
  - operare sulle directory (*mkdir*, *rmdir*, lista file contenuti, etc)
  - ispezionare e modificare i relativi diritti (read, write, execute)
- *obsoleta da Java 7 in poi, sostituita da **`Path`***
  - metodo `toPath` per favorire interoperabilità con nuovo codice

SEPARATORI DIPENDENTI DALLA PIATTAFORMA

`File` incapsula in *opportune costanti* le specificità della piattaforma in uso:

- **`File.separator`** (il separatore dei file)
  - '**`/`**' in Unix & Mac (nonché all'interno dei JAR)
  - '**`\`**' in Windows (ma non all'interno dei JAR)
- **`File.pathSeparator`** (il separatore dei percorsi)
  - '**`:`**' in Unix & Mac (nonché all'interno dei JAR)
  - '**`;`**' in Windows (ma non all'interno dei JAR)

ESEMPI `File` (1)

ESEMPIO 1: rinominare un file (dopo aver verificato che esista):

```
File f = new File("myfile.tmp");
if (f.exists()) {
  f.renameTo(new File("myfile.old"));
}
```

ESEMPIO 2: stampare tutti i file di una *directory*

```
File d = new File(".");
if (d.isDirectory()) {
  String[] allFilenames = d.list();
  for (String filename : allFilenames){
      System.out.println(filename);
  }
}
```

ESEMPI `File` (2)

ESEMPIO 3: stampare *solo alcuni file* di una directory:

```
File f = new File(".");
if (f.isDirectory()) {
  String[] someFilenames = f.list(new MyFilter());
  for (String filename : filenames){
      System.out.println(filename);
  }
}
```

Una *nostra* classe che implementi l'interfaccia **`FilenameFilter`**, il cui metodo

`boolean accept(File dir, String f)`

restituisca `true` se il file di nome **`f`** nella cartella **`dir`** va incluso in lista, `false` altrimenti.

ESEMPI `File` (3)

ESEMPIO 3: stampare *solo alcuni file* di una directory:

```
File f = new File(".");
if (f.isDirectory()) {
  String[] someFilenames = f.list(new MyFilter());
  for (String filename : someFilenames){
      System.out.println(filename);
  }
}
```

Ad esempio, per filtrare i soli file Java:

```
public class MyFilter implements FilenameFilter {
  public boolean accept(File dir, String name){
      return name.endsWith(".java");
  }
}
```

`File` – PERCHÉ SOSTITUIRLA?

`File` era semplice, ma il tempo ha fatto emergere *difetti*

- molti metodi non lanciavano eccezioni in caso di fallimento
  - esempio: se falliva l'eliminazione di un file, si riceveva solo una indicazione di fallimento ma non il motivo (file inesistente? diritti? altro?)
- **`rename`** non operava in modo consistente fra più piattaforme
- mancanza di supporto completo ai link simbolici
- supporto per i metadati (permessi, diritti, etc) minimale e inefficiente
- seri problemi di scalabilità in molti metodi cruciali
  - gravi problemi con directory di grandi dimensioni (rischio di blocco con lunghi elenchi di file, problemi di memoria con rischi di blocco del servizio)
- gestione inaffidabile in presenza di link simbolici circolari

Approccio troppo «basico», con problemi di scalabilità

Per migrare facilmente il vecchio codice alle nuove API: ***`toPath`***

LA NUOVA ASTRAZIONE `Path` IN `java.nio.file`

`Path` è il nuovo entry point del package `java.nio.file`

- sostituisce la precedente astrazione **`File`**
  - come **`File`**, modella un percorso che può anche *non esistere*
  - come **`File`**, risente della piattaforma sottostante per i separatori
  - a differenza di **`File`**, è un'interfaccia → *non ci sono costruttori*
  - la costruzione avviene tramite i metodi della factory **`Paths`**
  - metodo **`toFile`** per favorire *interoperabilità con vecchio codice*
- offre metodi per
  - recuperare informazioni sul file o directory
  - operare su percorsi (convertire un percorso in altro formato, unire due percorsi, creare un percorso fra due percorsi, ..)
  - confrontare percorsi
  - …

ESEMPIO `Path` (1)

Il metodo factory **`Paths.get`** costruisce un **`Path`** a partire

- da una stringa che rappresenta *un singolo percorso*
- da una serie di stringhe che rappresentano *pezzi di percorso*

```
Path p1 = Paths.get("/tmp/foo"); // factory method
Path p2 = Paths.get( System.getProperty("user.home"),
                     "logs", "foo.log");
System.out.println(p1); // toString
System.out.println(p1.getFileName());
System.out.println(p1.getName(0));
System.out.println(p1.getNameCount());
System.out.println(p1.subpath(0,2));
System.out.println(p1.getParent());
System.out.println(p1.getRoot());
```

```
\tmp\foo
foo
tmp
2
tmp\foo
\tmp
\
```

ESEMPIO `Path` (2)

Il metodo factory **`Paths.get`** costruisce un **`Path`** a partire

- da una stringa che rappresenta *un singolo percorso*
- con sintassi Windows o Unix
- *con percorsi relativi o assoluti*

```
Path p1 = Paths.get("sally\\bar"); // factory method

System.out.println(p1); // toString
System.out.println(p1.getFileName());
System.out.println(p1.getName(0));
System.out.println(p1.getNameCount());
System.out.println(p1.subpath(0,2));
System.out.println(p1.getParent());
System.out.println(p1.getRoot());
```

```
sally\bar
bar
sally
2
sally\bar
sally
null
```

ESEMPIO `Path` (3)

I percorsi possono essere

- *normalizzati* col metodo **`normalize`**

```
Path p1 = Paths.get("/home/./joe/foo");
Path p2 = Paths.get("/home/sally/../joe/foo");
Path p3 = Paths.get("enrico/temp");

System.out.println(p1);
System.out.println(p2);
System.out.println(p1.normalize());
System.out.println(p2.normalize());
System.out.println(p3);
System.out.println(p3.normalize());
```

```
\home\.\joe\foo
\home\sally\..\joe\foo
\home\joe\foo
\home\joe\foo
enrico\temp
enrico\temp
```

ESEMPIO `Path` (4)

I percorsi possono essere

- *risolti* uno rispetto a un altro (ossia, concatenati) col metodo **`resolve`**

```
Path p3 = Paths.get("enrico/temp");
System.out.println(p3.resolve("home/joe"));
System.out.println(p3.resolve("/home/joe"));
```

```
enrico\temp\home\joe
\home\joe
```

Impossibile "risolvere" un percorso relativo rispetto a un percorso assoluto *diverso*

```
System.out.println(Paths.get("/home/joe").resolve(p3));
System.out.println(Paths.get("home/joe").resolve(p3));
```

```
\home\joe\enrico\temp
home\joe\enrico\temp
```

ESEMPIO `Path` (5)

I percorsi possono essere

- *relativizzati* uno rispetto all'altro (de-concatenati) con **`relativize`**

```
Path p1 = Paths.get("home");
Path p2 = Paths.get("home/sally/bar");
System.out.println(p1.relativize(p2));
```

```
sally\bar
```

```
Path p3 = Paths.get("casa");
System.out.println(p1.relativize(p3));
System.out.println(p3.relativize(p1));
```

```
..\casa
..\home
```

In assenza di informazioni specifiche, due percorsi relativi si suppongono *riferiti allo stesso livello*, ossia ***fratelli***

ESEMPIO `Path` (6)

I percorsi possono essere

- *confrontati* con **`equals`** e **`compareTo`** (sono dei **`Comparable`**)
  - NB: che il confronto sia case-sensitive o no, *dipende dalla piattaforma*
- *confrontati per prefissi/suffissi* con **`startsWith`**, **`endWith`**

```
Path p2 = Paths.get("enrico");
Path p3 = Paths.get("enrico/temp");
Path p4 = Paths.get("enrico/temp");

System.out.println(p3.equals(p4));
System.out.println(p3.compareTo(p4));
System.out.println(p3.compareTo(p2));

Path p5 = Paths.get("enrico/Temp");
System.out.println(p3.equals(p5));
System.out.println(p3.compareTo(p5));
```

```
true
0
5
```

p3 > p2

```
true
0
```

Windows è case-insensitive

LA LIBRERIA & FACTORY `Files` IN `java.nio.file`

È una libreria (come **`Math`**) + factory →*solo funzioni statiche*

1. metodi per operare su file e directory
   - copiare o spostare file o directory
   - creare / eliminare file, directory, link simbolici
   - creare file o directory temporanee
   - verificare l'esistenza di file o directory
   - recuperare attributi di file o directory
2. metodi per leggere e scrivere piccoli file in blocco
   - leggere / scrivere tutti i byte di un PICCOLO file binario
   - leggere / scrivere tutti i caratteri di un PICCOLO file di testo
3. metodi factory per *creare stream di I/O*
   - per non dover creare a mano gli stream (come si è sempre fatto)
   - metodi NON ESSENZIALI, aggiunti per pura comodità

CASO 1: ESEMPI

VECCHIA VERSIONE con **File**

```
File f = new File("myfile.tmp");
if (f.exists()) f.renameTo(new File("myfile.old"));
else System.out.println("File " + f + " inesistente");
```

NUOVA VERSIONE con **Files** e **Path**

```
Path f = Paths.get("myfile.tmp");
try {
  if (Files.exists(f)) {
       Files.move(f, Paths.get("myfile.old"));
  }
  else {
       System.out.println("File " + f + " inesistente");
  }
}
catch(IOException e1){
  System.out.println("Errore durante rename");
}
```

CASO 1: ESEMPI

VECCHIA VERSIONE con **`File`**

```
File d = new File(".");
if (d.isDirectory()) {
   String[] allFilenames = d.list();
   for (String fn: filenames) System.out.println(fn);
}
```

VERSIONE SIMILE con **`Files`** e **`Path`**

Non è identica alla precedente, perché..

`Files.list` non restituisce più un array di stringhe, ma una cosa "strana".. nuova ☺

```
Path d = Paths.get("myfile.tmp");
try {
   if (Files.isDirectory(d)) {
      System.out.println("Contati "+ Files.list(d).count() +" file");
}
catch(IOException e){
   System.out.println("Errore durante listing directory");
}
```

LA LIBRERIA & FACTORY `Files` IN `java.nio.file`

Leggere e scrivere piccoli file «in blocco»

- **`readAllBytes`** legge un piccolo file binario
- **`readAllLines`** legge un piccolo file di testo (due metodi)

readAllBytes

```
public static byte[] readAllBytes(Path path)
                           throws IOException
```

Reads all the bytes from a file. The method ensures that the file is closed when all bytes have been read or an I/O error, or other runtime exception, is thrown.

Note that this method is intended for simple cases where it is convenient to read all bytes into a byte array. It is not intended for reading in large files.

readAllLines

```
public static List<String> readAllLines(Path path,
                                        Charset cs)
                                 throws IOException
```

Read all lines from a file. This method ensures that the file is closed when all bytes have been read or an I/O error, or other runtime exception, is thrown. Bytes from the file are decoded into characters using the specified charset.

This method recognizes the following as line terminators:

- \u000D followed by \u000A, CARRIAGE RETURN followed by LINE FEED
- \u000A, LINE FEED
- \u000D, CARRIAGE RETURN

Additional Unicode line terminators may be recognized in future releases.

Note that this method is intended for simple cases where it is convenient to read all lines in a single operation. It is not intended for reading in large files.

CASO 2: ESEMPI

Lettura di un PICCOLO file binario

```
public static void main(String args[]){
   try {
     byte[] bytes = Files.readAllBytes(Paths.get("xy.bin"));
     System.out.println("Letti " + bytes.length + " bytes");
   }
   catch(IOException e){
     System.out.println("Problema di lettura");
     System.exit(1);
   }
 }
```

```
Letti 44 bytes
```

se il file esiste ed è stato aperto regolarmente

```
Problema di lettura
```

se il file non esiste o non può essere aperto

MAI usare questo approccio per file "non piccoli" !

CASO 2: ESEMPI

Lettura di un PICCOLO file di testo

```
import java.nio.file.*;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.List;

public class LetturaDaPiccoloFileTesto {

 public static void main(String args[]){
    try {
      List<String> rows = Files.readAllLines(Paths.get("dante.txt"));
      System.out.println("Lette " + rows.size() + " righe");
    }
    catch(IOException e2){
      System.out.println("Altro problema di lettura");
      System.exit(2);
    }
 }

}
```

```
Lette 3 righe
```

se il file esiste ed è stato aperto regolarmente

```
Altro problema di lettura
```

se il file non esiste o non può essere aperto

LA LIBRERIA & FACTORY `Files` IN `java.nio.file`

Leggere e scrivere piccoli file «in blocco»

- **`write`** scrive un array di byte in un piccolo file binario
- **`write`** scrive una lista di stringhe in un piccolo file di testo

write

```
public static Path write(Path path,
                         byte[] bytes,
                         OpenOption... options)
                  throws IOException
```

Secondo argomento: array di byte

Writes bytes to a file. The `options` parameter specifies how the file is created or opened. If no options are present then this method works as if the `CREATE`, `TRUNCATE_EXISTING`, and `WRITE` options are present. In other words, it opens the file for writing, creating the file if it doesn't exist, or initially truncating an existing `regular-file` to a size of 0. All bytes in the byte array are written to the file. The method ensures that the file is closed when all bytes have been written (or an I/O error or other runtime exception is thrown). If an I/O error occurs then it may do so after the file has been created or truncated, or after some bytes have been written to the file.

Usage example: By default the method creates a new file or overwrites an existing file. Suppose you instead want to append bytes to an existing file:

```
Path path = ...
byte[] bytes = ...
Files.write(path, bytes, StandardOpenOption.APPEND);
```

write

```
public static Path write(Path path,
                         Iterable<? extends CharSequence> lines,
                         Charset cs,
                         OpenOption... options)
                  throws IOException
```

Secondo argomento: lista di stringhe

Write lines of text to a file. Each line is a char sequence and is written to the file in sequence with each line terminated by the platform's line separator, as defined by the system property `line.separator`. Characters are encoded into bytes using the specified charset.

The `options` parameter specifies how the file is created or opened. If no options are present then this method works as if the `CREATE`, `TRUNCATE_EXISTING`, and `WRITE` options are present. In other words, it opens the file for writing, creating the file if it doesn't exist, or initially truncating an existing `regular-file` to a size of 0. The method ensures that the file is closed when all lines have been written (or an I/O error or other runtime exception is thrown). If an I/O error occurs then it may do so after the file has been created or truncated, or after some bytes have been written to the file.

CASO 2: ESEMPI

Scrittura di un PICCOLO file binario

```
public static void main(String args[]){
   try {
     byte[] bytes = { 2, 15, 113, 127, -4};
     Files.write(Paths.get("mybytes.bin"), bytes,
                                        StandardOpenOption.CREATE);
     System.out.println("Scritti " + bytes.length + " bytes");
   }
   catch(IOException e){
     System.out.println("Problema di scrittura");
     System.exit(1);
   }
 }
```

```
Scritti 5 bytes
```

se il file è stato creato e scritto regolarmente

```
Problema di scrittura
```

se il file non si è potuto aprire o c'è stato qualche altro problema

MAI usare questo approccio per file "non piccoli" !

CASO 2: ESEMPI

Scrittura di un PICCOLO file di testo

```
public class ScritturaSuPiccoloFileTesto {

 public static void main(String args[]){
    try {
      List<String> righe = List.of("Nel mezzo", "del cammin di nostra vita",
                           "mi ritrovai", "per una selva oscura",
                           "che la diritta via", "era smarrita.");
      Files.write(Paths.get("dante2.txt"), righe);
      System.out.println("Scritte " + righe.size() + " righe ");
    }
    catch(IOException e2){
      System.out.println("Problema di scrittura");
      System.exit(2);
    }
 }

}
```

```
Scritte 6 righe
```

se il file è stato creato e scritto regolarmente

```
Problema di scrittura
```

se il file non si è potuto aprire o c'è stato qualche altro problema

CASO 3: ESEMPI

Gli esempi del tipo 3… li vedremo dopo ☺

Si tratta comunque una classica serie di factory methods:

- `newBufferedReader(Path p)`
- `newBufferedReader(Path p, Charset cs)`
- `newBufferedWriter(Path p, Charset cs, OpenOption... options)`
- `newBufferedWriter(Path p, OpenOption... options)`
- `newInputStream(  Path p, OpenOption... options)`
- `newOutputStream(  Path p, OpenOption... options)`

CURIOSITÀ: la notazione *varargs* (`Tipo...`) indica un numero variabile di argomenti di quel certo `Tipo` (da 0 in su)

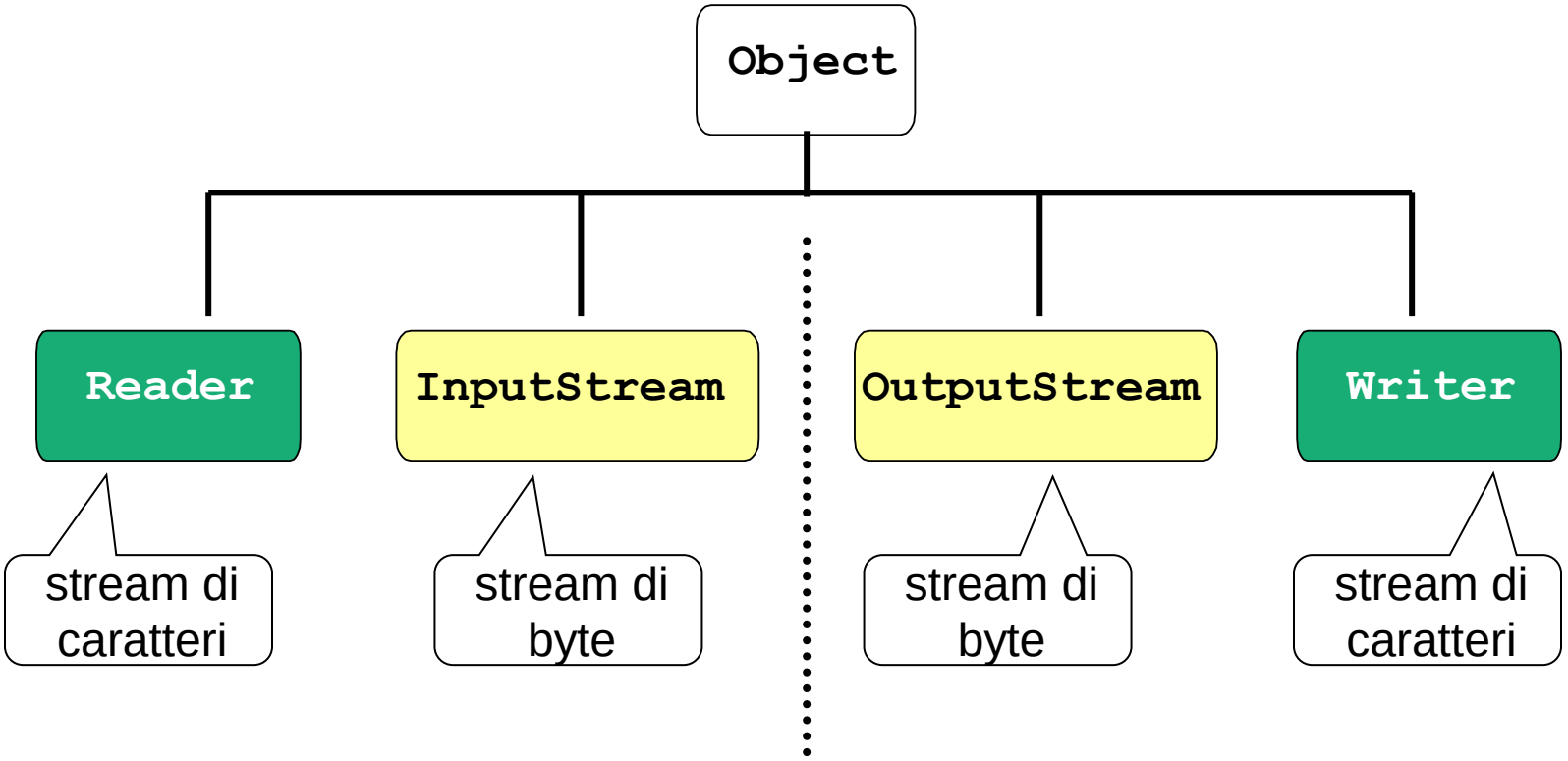
Concetti base

`java.io`: LE BASI

ASTRAZIONE CHIAVE: *stream di byte*

- un canale di comunicazione di uso generale
- *monodirezionale (o di input, o di output)*
- capace di trasferire *uno o più byte*
  - si può gestire *byte per byte* qualunque tipologia di I/O
- ma l'I/O di testo costituisce un *caso frequente,* meritevole di un supporto ad hoc → *stream di caratteri*
  - non sono indispensabili
  - ma sono ottimizzati per l'I/O a caratteri (linee di testo, Unicode..)
  - da Java 1.5, implementano l'interfaccia **`Readable`**

CLASSI BASE ASTRATTE



`java.io`: ARCHITETTURA

Il package **`java.io`** definisce i *concetti-base* e *l'architettura* per gestire l'I/O:

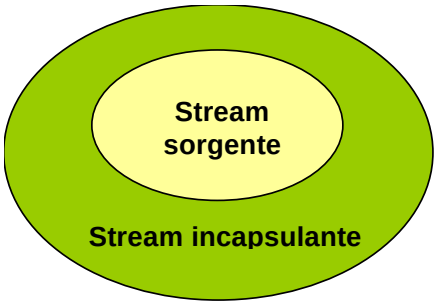
- da qualsiasi *sorgente*
- verso qualsiasi *destinazione*

in modo *configurabile.*

Incapsulare in un oggetto i dettagli di una data *sorgente* o di un *dispositivo di output*

Comporre uno stream personalizzato sovrapponendo a uno *stream "nucleo" altri strati* che lo adattino alle necessità del caso specifico

PATTERN Adapter + Chain of responsibility



APPROCCIO "A CIPOLLA"

Per questo, sia gli stream di byte, sia quelli di caratteri si dividono in *due grandi categorie*:

- stream destinati a incapsulare sorgenti fisiche di dati o dispositivi fisici di uscita
  - i loro costruttori hanno come argomento *il dispositivo che interessa*: file, connessioni di rete, array di byte,…
- stream di adattamento pensati per *adattare i precedenti* alle necessità, aggiungendo ulteriori funzionalità
  - i loro costruttori hanno come argomento *uno stream già esistente* (il nucleo da "avvolgere")