I File in Java

di Roberta Molinari

ESERCIZIO

Analizzando i 4 byte di un file il loro valore in Hex è

4D 69 61 6F

Quali sono i dati memorizzati?

Serializzazione

- La **Serializzazione** è un processo per salvare un oggetto in un supporto di memorizzazione o per trasmetterlo su una connessione di rete.
- Può essere un file in forma binaria o può utilizzare codifiche testuali (es. in XML) direttamente leggibili dagli esseri umani.
- Il suo scopo è trasmettere l'intero stato dell'oggetto in modo che esso possa essere successivamente ricreato nello stesso identico stato con il processo inverso: **Deserializzazione**.

Serializzazione in Java
tipi di file utilizzati

- classi ObjectInputStream e ObjectOutputStream definiscono streams di byte
- file di testo CSV (comma-separated values)
- file JSON (JavaScript Object Notation)
- file XML (eXtensible Markup Language)

Formato JSON

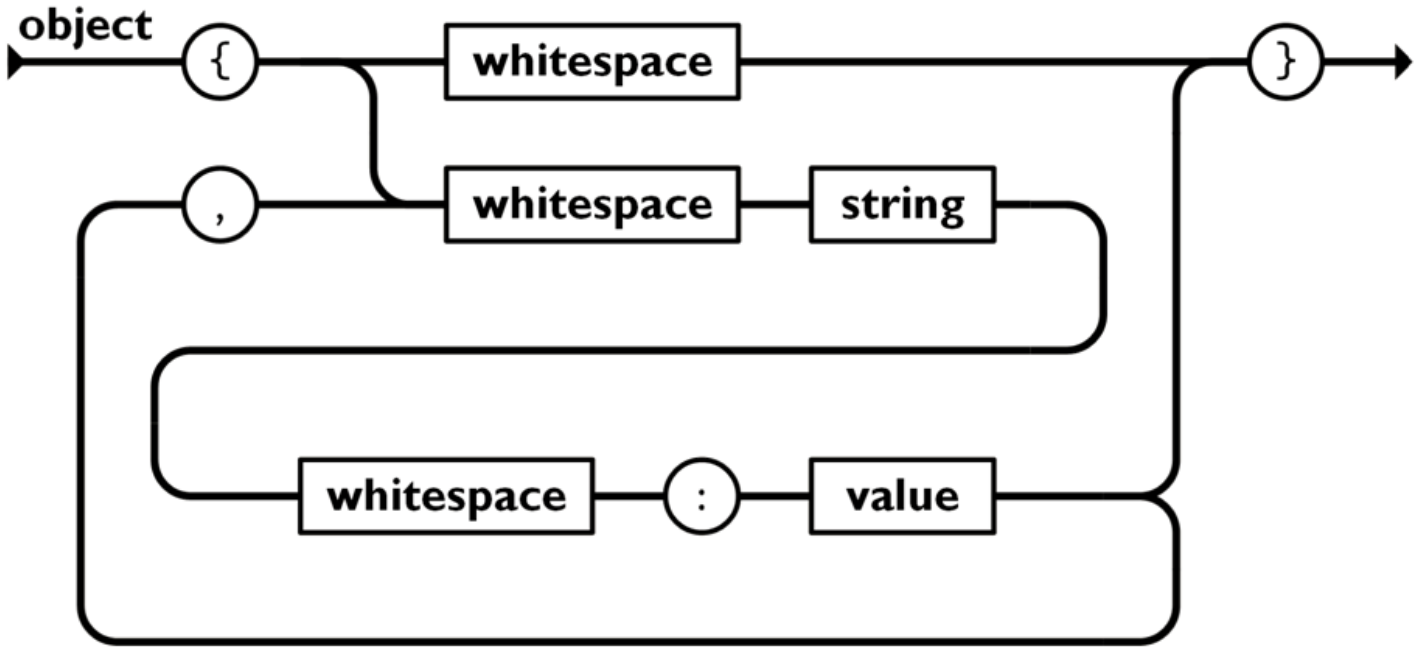
- **JSON** (JavaScript Object Notation) è un semplice formato per lo scambio di dati:
- per le persone è facile da leggere e scrivere
- per le macchine risulta facile da generare e analizzarne la sintassi.
- Si basa su un sottoinsieme del Linguaggio di Programmazione JavaScript (Standard ECMA-262 Terza Edizione - Dicembre 1999).
- è completamente indipendente dal linguaggio di programmazione, ma utilizza convenzioni conosciute dai programmatori di linguaggi della famiglia del C, come C, C++, C#, Java, JavaScript, Perl, Python, e molti altri.

Formato JSON

- JSON è basato su due strutture:
  - **OGGETTI**: insieme di **coppie nome/valore**. In diversi linguaggi, questo è realizzato come un oggetto, un record, uno struct, un dizionario, una tabella hash, un elenco di chiavi o un array associativo.
  - **ARRAY**: **elenco ordinato di valori**. Nella maggior parte dei linguaggi questo si realizza con un array, un vettore, un elenco o una sequenza.
- Queste sono strutture di dati universali: tutti i linguaggi di programmazione moderni li supportano in entrambe le forme. E' sensato che un formato di dati che è interscambiabile con linguaggi di programmazione debba essere basato su queste strutture.

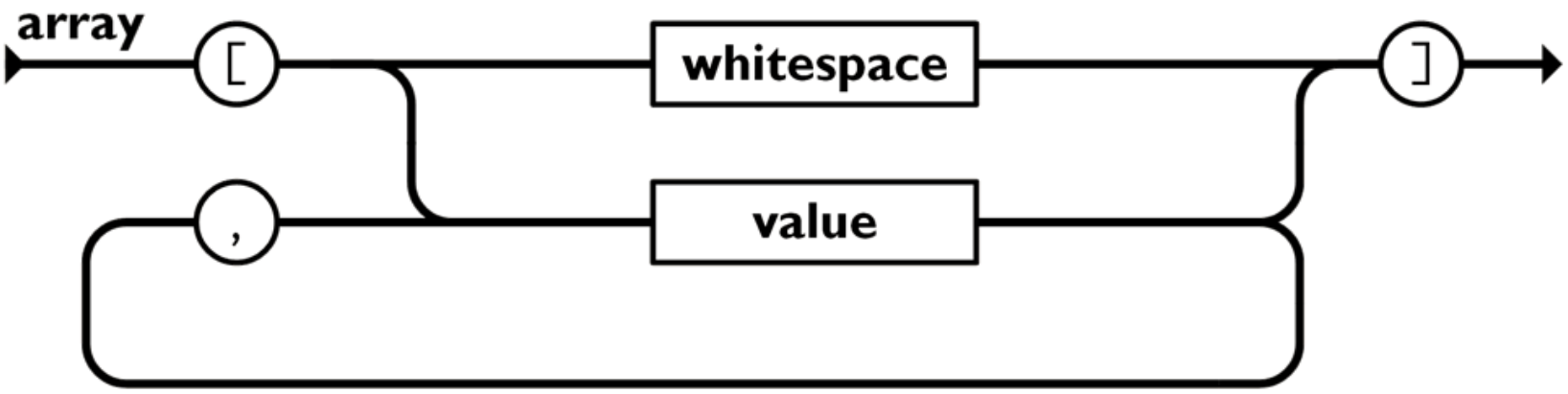
Formato JSON
Oggetto

- Un ***oggetto*** è una serie non ordinata di nomi/valori. Un oggetto inizia con **{** e finisce con **}**. Ogni nome è seguito da **:**due punti e la coppia di nome/valore sono separata da **,**virgola. (non c'è una virgola dopo l'ultimo elemento).



Formato JSON
Array

- Un ***array*** è una raccolta ordinata di valori. Un array comincia con **[**e finisce con **]**. I valori sono separati da **,**virgola (non c'è una virgola dopo l'ultimo elemento). Possono essere di tipo misto



Formato JSON
Valori

- Un ***valore*** può essere una stringa tra virgolette, o un numero, o vero **true** o falso **false** o nullo **null**, o un oggetto o un array.

- **Queste strutture possono essere annidate.**

- Per validare il file https://jsonlint.com/

Formato JSON
Esempio oggetto

```
public class MyJson {
  String myString = "my string";
  int myInt = 5;
  double[] myArrayOfDoubles =
       new double[] { 3.14, 2.72 };
 MyOtherJson objectInObject =
       new MyOtherJson();
}
```

- Questa è la relativa rappresentazione JSON di un oggetto della classe:

```
{
  "myString" : "my string",
  "myInt" : 5,
  "myArrayOfDoubles" : [ 3.14, 2.72 ],
  "objectInObject" : {}
}
```

Formato JSON
Esempio array

- Gli array possono anche contenere elementi con tipi misti, ad esempio:

```
[
  "red",
  51,
  true,
  null,
  {
       "state": "complete"
  } ,
  [1 , 3]
]
```

Formato JSON
Esempio array di oggetti

```
[
  {    "precision": "zip",
       "Latitude":  37.7668,     "Longitude": -122.3959,
       "Address":   "",     "City":      "SAN FRANCISCO",
       "State":     "CA",     "Zip":       "94107",
       "Country":   "US"
  },
  {     "precision": "zip",
       "Latitude":  37.371991,   "Longitude": -122.0260,
       "Address":   "",     "City":      "SUNNYVALE",
       "State":     "CA",     "Zip":       "94085",
       "Country":   "US"
  }
]
```

File di byte o binario

- Un **file binario o di byte** è un file che può contenere qualsiasi tipo di dati, codificato in codice binario a scopo di archiviazione o utilizzo (ad esempio file documenti che contengono testo formattato).
- Molti formati di file binari contengono parti che possono essere interpretate come testo, ma si distinguono per definizione dai file di testo veri e propri: un file binario è un file che non contiene solo semplice testo.

Es.

- file eseguibili
- Immagini
- Fogli di calcolo
- Documento word

Gli stream in Java

File

I **file** sono un insieme di dati memorizzati su memoria di massa. Sono gli "archivi elettronici".

Se il file è una successione di record con la stessa struttura si dice **file di record o strutturati**, se è una successione di righe di caratteri si parla di **file di testo**.

Il modo con cui un programma utilizza i dati memorizzati rappresenta il **tipo di accesso** al file, che può essere:

- **Sequenziale**
- **Diretto o random**

File di testo o binari

Il SO tratta i file in 2 modi, come:

- **File di testo**: ovvero come successione di caratteri, distinguendo una riga da un'altra grazie ad un carattere di fine riga. ('\n')
- **File binari**: successione di byte senza distinguere una riga da un'altra. (per i file di record, per poter interpretare i dati si deve conoscere la struttura)

Per poter trattare i dati memorizzati su un file, questi si devono trasferire dalla memoria di massa alla memoria centrale (operazione di *input*) o viceversa (operazione di *output*). Il trasferimento fisico è relativo non ad un singolo byte, ma ad un insieme di byte: **blocco fisico** (che, in generale, contiene 1 o più record - righe). Dal punto di vista del programmatore il trasferimento è relativo ad un solo record - riga.

File
operazioni

- **Apertura**:
  - si crea un collegamento tra MM e MC associando al **nome logico** del file (riconosciuto all'interno del programma) il **nome fisico** (riconosciuto a livello di SO)
  - Si riserva in MC un buffer per le operazioni di I/O
  - Si aggiornano le tabelle di gestione dei file per specificare le informazioni necessarie per l'individuazione dei dati su MM
- **lettura**: si trasferiscono i dati dalla MM al buffer relativo al file
- **scrittura**: si trasferiscono i dati dal buffer relativo al file alla MM
- **chiusura**: si aggiornano le tabelle di gestione file

File

classe File

- **java.io.File** – è una rappresentazione astratta di un file o di una directory. Permette di lavorare a livello più alto, come ad es. creare un file vuoto, eliminarlo…
- I metodi principali:
  - **exists**() restituisce true se il file esiste, false altrimenti.
  - **createNewFile**() crea effettivamente il file (vuoto) sul disco. Restituisce true se si riesce a creare il file, false se esiste già.
  - **delete**() elimina il file dal disco. Restituisce true se si riesce a eliminare il file, false altrimenti
  - **isFile**() restituisce true se si tratta di un file, false altrimenti
  - **isDir**() restituisce true se si tratta di una directory, false altrimenti.
  - **mkDir**() permette di creare una directory. Restituisce true se si riesce a creare la dir, false altrimenti
  - **renameTo**(File dest) rinomina il file con il nome di dest Restituisce true se tutto ok, false altrimenti

File
Creazione

- Il costruttore della classe File riceve in ingresso il path del file. La creazione di un'istanza di File non genera la creazione fisica del file sul disco fisso; occorre chiamare il metodo `createNewFile()` per crearlo fisicamente. Questo metodo restituisce a valore booleano: true se il file è stato creato correttamente e false se il file esiste già.
- Per **creare** un file nella dir del progetto

```
File f=new File("nome.txt");
f.createNewFile();
```

```
File f=new File("nome.txt");
if (f.exists())
      System.out.println("Il file esiste");
if (f.createNewFile())
      System.out.println("Il file è stato creato");
else
      System.out.println("Il file esiste già e non è stato
  creato");
```


20

File

Esistenza - Chiusura

- Per sapere se **esiste** un file si può fare direttamente

```
if (new File("nome.txt").exists())
```

- Per la **chiusura** (il file deve essere aperto)

```
bufferedReader.close();
```

Per ripulire il buffer e terminare la scrittura (o lettura) su disco si mette prima della chiusura di un file aperto in w/a

```
filewriten.flush();
```

- Le operazioni sui file possono generare l'eccezione **`FileNotFoundException`**
- Per conoscere il nome del file si usa il metodo **`getName()`**

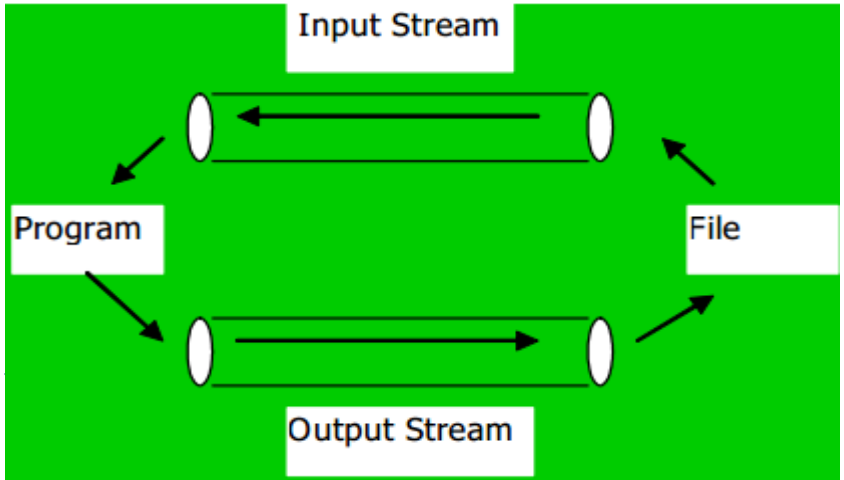
```
File file = new File("my.dat");
String s=file.getName();
```


21

Stream

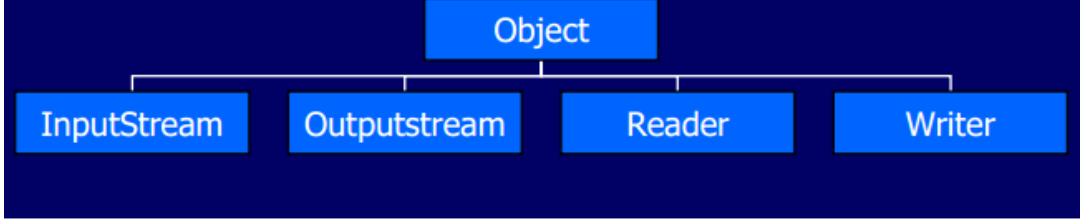
Uno **stream** o **flusso** è una sequenza ordinata di dati:

- **monodirezionale** (o di input, o di output)
- adatto a trasferire byte (o anche caratteri)
- **Buffer**: deposito di dati da (su) cui si può leggere (scrivere)
- **Canale**: connessione con entità in grado di eseguire operazioni di I/O; i canali includono i buffer, i file e i socket
- **Stream di Input:** i dati sono presi da una sorgente (file o tastiera o rete) e trasferiti al programma
- **Stream di Output:** i dati sono generati dal programma e trasferiti ad una destinazione (file o video o rete)



22

Stream e file



stream di byte stream di caratteri

I flussi permettono operazioni di I/O

Il relativo package è **java.io**, in cui ci sono 4 classi astratte di base.

I flussi si possono concatenare uno dopo l'altro, così da sfruttare il fatto che ogni stream vede i dati e li tratta in modo diverso con metodo propri.

In un flusso non è necessario conoscere la fonte per leggere un flusso di input come non è necessario conoscere il destinatario per poter inviare (scrivere) un flusso, pertanto un **file** può essere visto come un particolare flusso di dati scritto su memoria di massa.

I File in Java possono essere:

- **binari** (flusso di byte a 8 bit)
- di **testo** (flusso di caratteri)
- di tipi primitivi
- di oggetti

Questi tipi possono solo essere gestiti in modo sequenziale

La rappresentazione dei caratteri nei file

Java usa UTF-16 come rappresentazione interna dei caratteri, ma per la loro serializzazione su file usa una variazione non standard di UTF-8. Le differenze:

1. il carattere nullo (U+0000) viene rappresentato con due byte anziché uno, nello specifico come 11000000 10000000 (0xC0 0x80). In questo modo ci si assicura che nessuna stringa codificata venga troncata prematuramente perché contenente il byte *null* (0x00), interpretato da alcuni linguaggi di programmazione (il C) come terminatore della stringa.
2. l'UTF-8 standard rappresenta con 4 byte i caratteri al di fuori del BMP (Basic Multilingual Plane). In Java vengono prima rappresentati come coppie surrogate (come in UTF-16) e successivamente entrambi gli elementi della coppia vengono codificati in UTF-8. questo vuol dire che i caratteri che richiedono 4 byte per essere rappresentati in UTF-8, siano rappresentati in UTF-8 modificato con sequenze di 6 byte.

File

I **file** sono un insieme di dati memorizzati su memoria di massa. Sono gli "archivi elettronici".

Se il file è una successione di record con la stessa struttura si dice **file di record o strutturati**, se è una successione di righe di caratteri si parla di **file di testo**.

Il modo con cui un programma utilizza i dati memorizzati rappresenta il **tipo di accesso** al file, che può essere:

- **Sequenziale**
- **Diretto o random**

25

File di testo o binari

Il SO tratta i file in 2 modi, come:

- **File di testo**: ovvero come successione di caratteri, distinguendo una riga da un'altra grazie ad un carattere di fine riga. ('\n')
- **File binari**: successione di byte senza distinguere una riga da un'altra. (per i file di record, per poter interpretare i dati si deve conoscere la struttura)

Per poter trattare i dati memorizzati su un file, questi si devono trasferire dalla memoria di massa alla memoria centrale (operazione di *input*) o viceversa (operazione di *output*). Il trasferimento fisico è relativo non ad un singolo byte, ma ad un insieme di byte: **blocco fisico** (che, in generale, contiene 1 o più record - righe). Dal punto di vista del programmatore il trasferimento è relativo ad un solo record - riga.

File
operazioni

- **Apertura**:
  - si crea un collegamento tra MM e MC associando al **nome logico** del file (riconosciuto all'interno del programma) il **nome fisico** (riconosciuto a livello di SO)
  - Si riserva in MC un buffer per le operazioni di I/O
  - Si aggiornano le tabelle di gestione dei file per specificare le informazioni necessarie per l'individuazione dei dati su MM
- **lettura**: si trasferiscono i dati dalla MM al buffer relativo al file
- **scrittura**: si trasferiscono i dati dal buffer relativo al file alla MM
- **chiusura**: si aggiornano le tabelle di gestione file

27

File

classe File

- **java.io.File** – è una rappresentazione astratta di un file o di una directory. Permette di lavorare a livello più alto, come ad es. creare un file vuoto, eliminarlo…
- I metodi principali:
  - **exists**() restituisce true se il file esiste, false altrimenti.
  - **createNewFile**() crea effettivamente il file (vuoto) sul disco. Restituisce true se si riesce a creare il file, false se esiste già.
  - **delete**() elimina il file dal disco. Restituisce true se si riesce a eliminare il file, false altrimenti
  - **isFile**() restituisce true se si tratta di un file, false altrimenti
  - **isDir**() restituisce true se si tratta di una directory, false altrimenti.
  - **mkDir**() permette di creare una directory. Restituisce true se si riesce a creare la dir, false altrimenti
  - **renameTo**(File dest) rinomina il file con il nome di dest Restituisce true se tutto ok, false altrimenti

28

File
Creazione

- Il costruttore della classe File riceve in ingresso il path del file. La creazione di un'istanza di File non genera la creazione fisica del file sul disco fisso; occorre chiamare il metodo `createNewFile()` per crearlo fisicamente. Questo metodo restituisce a valore booleano: true se il file è stato creato correttamente e false se il file esiste già.
- Per **creare** un file nella dir del progetto

```
File f=new File("nome.txt");
f.createNewFile();
```

```
File f=new File("nome.txt");
if (f.exists())
      System.out.println("Il file esiste");
if (f.createNewFile())
      System.out.println("Il file è stato creato");
else
      System.out.println("Il file esiste già e non è stato
  creato");
```


29

File

Esistenza - Chiusura

- Per sapere se **esiste** un file si può fare direttamente

```
if (new File("nome.txt").exists())
```

- Per la **chiusura** (il file deve essere aperto)

```
bufferedReader.close();
```

Per ripulire il buffer e terminare la scrittura (o lettura) su disco si mette prima della chiusura di un file aperto in w/a

```
filewriten.flush();
```

- Le operazioni sui file possono generare l'eccezione **`FileNotFoundException`**
- Per conoscere il nome del file si usa il metodo **`getName()`**

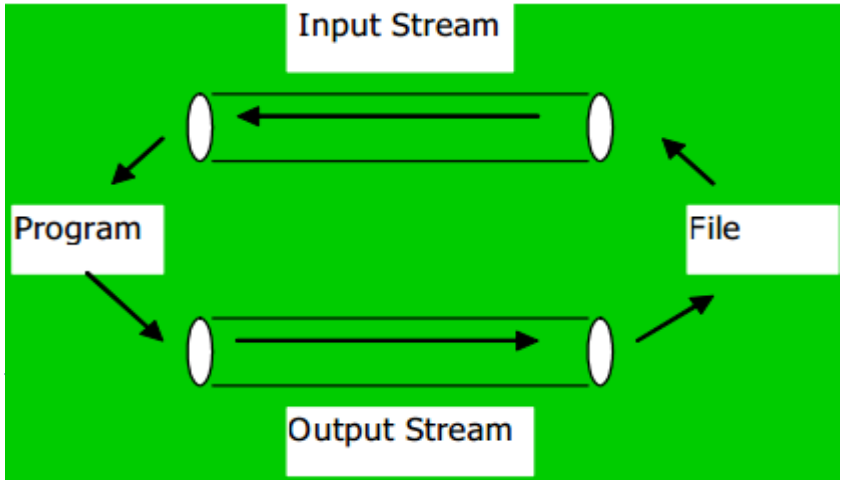
```
File file = new File("my.dat");
String s=file.getName();
```


30

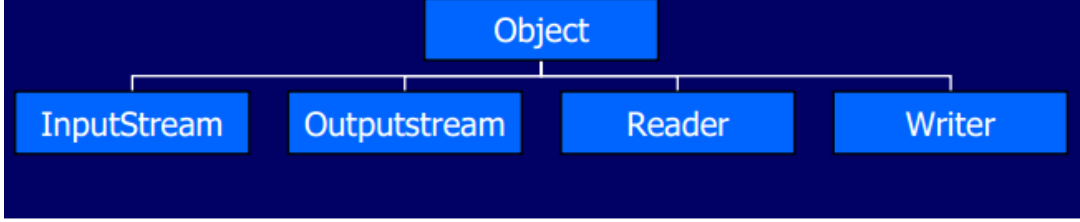
Stream

Uno **stream** o **flusso** è una sequenza ordinata di dati:

- **monodirezionale** (o di input, o di output)
- adatto a trasferire byte (o anche caratteri)
- **Buffer**: deposito di dati da (su) cui si può leggere (scrivere)
- **Canale**: connessione con entità in grado di eseguire operazioni di I/O; i canali includono i buffer, i file e i socket
- **Stream di Input:** i dati sono presi da una sorgente (file o tastiera o rete) e trasferiti al programma
- **Stream di Output:** i dati sono generati dal programma e trasferiti ad una destinazione (file o video o rete)



Stream e file



stream di byte stream di caratteri

I flussi permettono operazioni di I/O

Il relativo package è **java.io**, in cui ci sono 4 classi astratte di base.

I flussi si possono concatenare uno dopo l'altro, così da sfruttare il fatto che ogni stream vede i dati e li tratta in modo diverso con metodo propri.

In un flusso non è necessario conoscere la fonte per leggere un flusso di input come non è necessario conoscere il destinatario per poter inviare (scrivere) un flusso, pertanto un **file** può essere visto come un particolare flusso di dati scritto su memoria di massa.

I File in Java possono essere:

- **binari** (flusso di byte a 8 bit)
- di **testo** (flusso di caratteri)
- di tipi primitivi
- di oggetti

Questi tipi possono solo essere gestiti in modo sequenziale

32

La rappresentazione dei caratteri nei file

Java usa UTF-16 come rappresentazione interna dei caratteri, ma per la loro serializzazione su file usa una variazione non standard di UTF-8. Le differenze:

1. il carattere nullo (U+0000) viene rappresentato con due byte anziché uno, nello specifico come 11000000 10000000 (0xC0 0x80). In questo modo ci si assicura che nessuna stringa codificata venga troncata prematuramente perché contenente il byte *null* (0x00), interpretato da alcuni linguaggi di programmazione (il C) come terminatore della stringa.
2. l'UTF-8 standard rappresenta con 4 byte i caratteri al di fuori del BMP (Basic Multilingual Plane). In Java vengono prima rappresentati come coppie surrogate (come in UTF-16) e successivamente entrambi gli elementi della coppia vengono codificati in UTF-8. questo vuol dire che i caratteri che richiedono 4 byte per essere rappresentati in UTF-8, siano rappresentati in UTF-8 modificato con sequenze di 6 byte.

File

classi per file di testo

- **java.io.FileReader** – permette di leggere i caratteri contenuti in un file di testo, uno alla volta (non stringhe o numeri)
- **java.io.BufferedReader** – wrapper della classe FileReader che legge da un buffer in cui vengono raccolti i byte prima di trasmetterli realmente. Quando i dati vengono richiesti, vengono letti dal buffer tutti insieme, quindi le prestazioni migliorano notevolmente non dovendo ogni volta avere l'ok del sistema operativo per accedere al file system. Legge anche stringhe. I metodi che si utilizzano sono:
  - `int` **`read()`** legge un carattere restituendo il codice UNICODE. Restituisce -1 a fine file
  - `String` **`readln()`** legge una riga. Restituisce null a fine file

34

File

classi per file di testo

- **java.io.FileWriter** – permette di scrivere i caratteri in un file di testo uno alla volta (non stringhe o numeri)
- **java.io.BufferedWriter** – wrapper della classe FileWriter che utilizza un buffer in cui scrivere i byte del file. Periodicamente i caratteri vengono letti dal buffer e scritti fisicamente sul file, migliorando le prestazioni. Gestisce anche stringhe. I metodi che si utilizzano sono:
  - **`write`**`(Char c)` **`write`**`(String s)` scrive la stringa
  - **`newLine()`** aggiunge il carattere di fine riga (dipende dal SO o \n o \r o entrambi)
  - **`append`**`(String s)` scrive s in coda

  Per ripulire il buffer e terminare la scrittura (o lettura) su disco si usa il metodo **`flush`**`();`

35

File
classi per file di testo

- **Java.io.PrintWriter** – wrapper della classe FileWriter, che permette di scrivere qualunque dato Java, convertendolo automaticamente in stringa.
- È possibile memorizzaare nel file stringhe formattate. I metodi che si utilizzano sono:

- **`print`**`(String s)` **`println`**`(String s)` scrive la stringa aggiungendo o meno \n.
- **`printf(String format, Object... args)`** come la printf in C

```
PrintWriter pw = new
PrintWriter(new FileWriter("f.txt"));
pw.print("Un numero: ");
pw.println(Math.PI);
pw.print("Un oggetto: ");
pw.println(new java.awt.Rectangle(10,15,20,30));
```


36

Stream
Per input da tastiera

Si concatenano due flussi che gestiscono caratteri e flussi di input

```
InputStreamReader input= new InputStreamReader (System.in);
BufferedReader tastiera=new BufferedReader(input);
```

La prima istruzione crea lo stream `input` che riceve dall'oggetto `System.in` che rappresenta lo standard di input.

La seconda crea lo stream `tastiera` che si collega al primo e permette di accedere ai dati del flusso utilizzando il metodo `.readLine()`

37

File di testo
Creazione del file

- La creazione avviene con l'istanzazione di oggetti di tipo **`FileReader o FileWriter`** (a seconda del tipo di apertura che si vuole effettuare)

- Entrambe le classi hanno 2 costruttori:
  - **`FileReader/Writer`**`(File f);`
  - **`FileReader/Writer`**`(String nomeFile);`

```
File f= new File("nome.txt");
FileReader fR= new FileReader(f);
```

o direttamente

```
FileReader fR= new FileReader("nome.txt");
```


38

File di testo
Apertura del file

- Si concatenano due stream di caratteri, perchè le operazioni di scrittura/lettura più efficienti sono fornite dal 2^ flusso
- Per l'apertura in **lettura** (se non esiste dà errore)

```
FileReader f= new FileReader("nome.txt");
BufferedReader fRead = new BufferedReader(f);
```

- Per l'apertura in **scrittura** (se esite elimina i dati, se no lo crea)

```
FileWriter f= new FileWriter("nome.txt");
    //oppure FileWriter("nome.txt", false);
PrintWriter fWrite= new PrintWriter (f);
```

- Per l'apertura in **append** (se esite non elimina i dati, se no lo crea)

```
FileWriter f= new FileWriter("nome.txt", true);
PrintWriter fWrite = new PrintWriter (f);
```

- Per aprire un file in scrittura se non esiste, in append altrimenti

```
FileWriter f= new FileWriter("nome.txt", (new
  File("nome.txt")).exists());
```


39

File di testo
Lettura del file

```
FileReader f= new FileReader("nome.txt");
BufferedReader fRead = new BufferedReader(f);
```

- Lettura di righe

```
String s = fRead.readLine();//return null per EOF
while(s != null){
…
s = fRead.readLine();
}
```

- Lettura di caratteri (in Java sono codificati in UNICODE UTF-16)

```
Char c;
int cod = fRead.read();//return codice UNICODE,
while(cod != -1){       // return -1 per EOF
c = (char)cod;   //casting per avere il carattere
…
cod=fRead.read();
}
```

File di testo
Scrittura su file

```
FileWriter f= new FileWriter("nome.txt");
PrintWriter fWrite = new PrintWriter(f);
```

- Scrittura di righe con terminatore `'\n'`

```
fWrite.println(s1 + s2);
```

- Scrittura di stringhe, senza termine riga

```
fWrite.print(s1 + s2);
```

ATTENZIONE nei file scritti con un editor di testo il carattere di fine riga sono due (dipende dal SO, in Windows):

- '\n' (Line Feed in ASCII 13) seguito da
- '\r' (Carriage Return in ASCII 10)

- Esiste anche il metodo `newline()` che aggiunge un separatore di linea

41

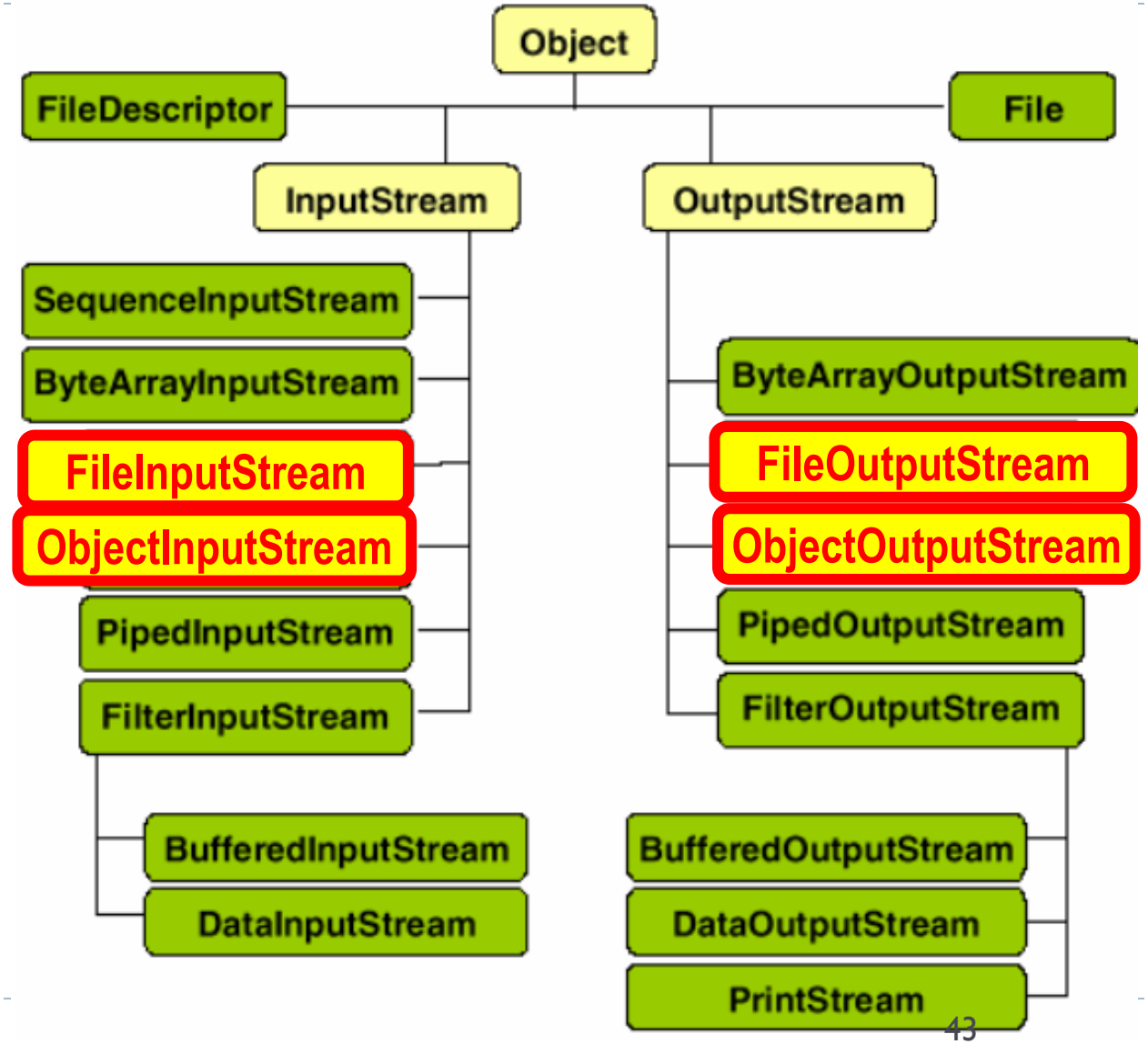
File di testo
Scrittura su file

```
BufferedWriter output =
    new BufferedWriter(new FileWriter(file));

String line = …;
output.append(line);
```


42

Classi stream di byte



43

File strutturati

- Si utilizzano dei file di byte per memorizzare dei record con una certa struttura
- Vengono usati principalmente per la **serializzazione** degli oggetti
- Per poterlo fare la classe degli oggetti da memorizzare deve implementare l'interface **`Serializable`** (che però non ha metodi), che permette di trasferire le istanze della classe sui flussi di input e output

  Per es.

```
class MioObj implements Serializable
```

Serializzazione è un processo per salvare un oggetto in un supporto di memorizzazione o per trasmetterlo su una connessione di rete. Può essere in forma binaria o può utilizzare codifiche testuali (es. in CSV, JSON, XML) direttamente leggibili dagli esseri umani. Lo scopo della serializzazione è di trasmettere l'intero stato dell'oggetto in modo che esso possa essere successivamente ricreato nello stesso identico stato dal processo inverso, chiamato **deserializzazione**.

File strutturati
Serializzazione

Le classi **ObjectInputStream** e **ObjectOutputStream** definiscono stream (basati su stream di byte) su cui si possono leggere e scrivere oggetti.

La scrittura e la lettura di oggetti va sotto il nome di **object serialization**, poiché si basa sulla possibilità di scrivere lo stato di un oggetto in una forma sequenziale, sufficiente per ricostruire l'oggetto quando viene riletto.

45

File strutturati
Serializzazione

La serializzazione di oggetti viene usata principalmente:

- Per fornire un meccanismo di **persistenza** ai programmi, consentendo l'archiviazione di un oggetto per poi riutilizzarlo in una successiva invocazione dello stesso programma. Si pensi ad esempio ad un programma che realizza una rubrica telefonica o un'agenda.
- Nel contesto di **invocazione remota di metodi** (**Remote Method Invocation** -- **RMI**) in cui due oggetti, che possono essere su macchine diverse, comunicano attraverso sockets e possono scambiarsi oggetti durante la comunicazione.

46

File strutturati
Serializzazione

Transient specifica che una variabile non deve essere serializzata, quindi il suo valore nella serializzazione viene posto a null. Una variabile transient non può essere final o static

- La serializzazione delle istanze di una classe viene gestita dal metodo **writeObject** della classe **ObjectOutputStream**. Questo metodo scrive automaticamente
  - la classe dell'oggetto;
  - la firma della classe;
  - I valori di tutte le variabili che non siano **transient** o **static**
  - gli oggetti raggiungibili, sia direttamente che transitivamente dall'oggetto che si sta serializzand
- Per realizzare la de-serializzazione si utilizza il metodo **readObject** della classe **ObjectInputStream**.

47

File strutturati
Apertura del file

Si concatenano due stream di byte, perchè le operazioni di scrittura/lettura più efficienti sono fornite dal 2^ flusso

- Per l'apertura in **lettura**

```
ObjectInputStream fIn= new ObjectInputStream(
                          new FileInputStream ("nome.dat"));
```

- Per l'apertura in **scrittura**

```
ObjectOutputStream fOut= new ObjectOutputStream(
                        new FileOutputStream("nome.dat"););
  //oppure FileOutputStream ("nome.dat", false)
```

- Per l'apertura in **append**

```
ObjectOutputStream fOut = new ObjectOutputStream(
                      new FileOutputStream("nome.dat", true));
```

File strutturati
Scrittura su file

- Scrittura di istanze di oggetti della classe `MioObj`

```
MioObj obj=new MioObj(…);

ObjectOutputStream fOut=new ObjectOutputStream (
                        new FileOutputStream("nome.dat"));

fOut.writeObject(obj);
```

Se si salva con **`writeObject`** un oggetto che ha puntatori ad altri oggetti, allora tutti gli oggetti raggiungibili, sia direttamente che transitivamente, vengono scritti nello stream, in modo da mantenere le relazioni tra di essi.

Il metodo **`writeObject`** lancia un'eccezione **`NotSerializableException`** se cerca di serializzare un oggetto che non implementa l'interfaccia **`Serializable`.**

49

File strutturati
Lettura del file

- Lettura di oggetti della classe `MioObj`

```
ObjectInputStream fIn= new ObjectInputStream(
                    new FileInputStream ("nome.dat"));

MioObj obj=null;
    try{
    …
    obj = (mioObj)fIn.readObject();
        //casting perchè restituisce un Object
    …
    }catch(ClassNotFoundException e){
        //se il casting fallisce
    }
```


50

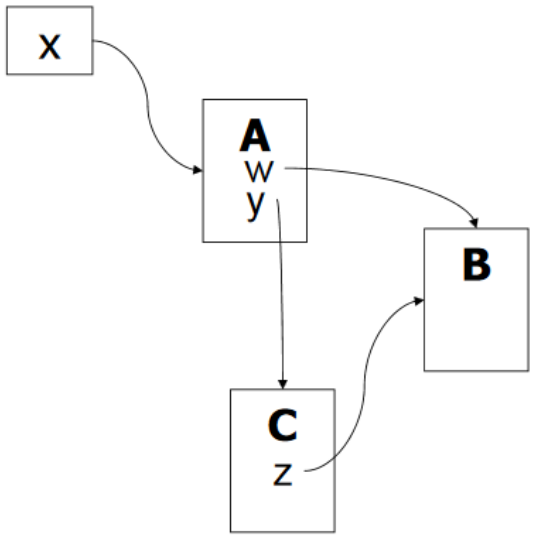
File strutturati
Lettura del file

- Lettura di più oggetti della classe `MioObj`. Attenzione gli oggetti devono essere riletti nell'ordine in cui erano stati scritti

```
ObjectInputStream fIn= new ObjectInputStream(
                    new FileInputStream ("nome.dat"));
MioObj obj = null;
while(true){ //se c'è più di un oggetto da leggere
    try{      …
    obj = (mioObj)fIn.readObject();
    //casting perchè restituisce un Object
    …
    }catch(EOFException e){ //raggiunto fine file
       break; //interrompe il ciclo
    } catch(ClassNotFoundException e){
        //se il casting fallisce
    }
```


51

File strutturati
Serializzazione di rete di oggetti



- **Efficienza**: ogni oggetto deve essere copiato sullo stream una sola volta
- **Consistenza**: devono essere mantenute le relazioni tra gli oggetti

- Ad ogni oggetto viene assegnato un numero di serie
- Quando si serializza un oggetto si verifica se tale oggetto è già stato serializzato in tal caso, si riferisce l'oggetto tramite il suo numero di serie; se no si serializza l'oggetto

52

File strutturati
Serializzazione di rete di oggetti

Oggetto	Numero di serie
A	1
B	2
C	3