Capitolo 17

Gestione di file in Java

Ogni insieme di informazioni residenti su memoria di massa costituisce in Java un flusso di dati che è denominato *stream*. Più in particolare un *input stream* costituisce una sequenza di byte che è sottoponibile a operazioni di lettura; simmetricamente un *output stream* definisce un flusso di byte sottoponibile a operazioni di scrittura. Responsabili della gestione delle operazioni di lettura da stream e di scrittura su stream sono i metodi di classi afferenti al package `java.io` e da esse derivate.

I *file* costituiscono le strutture di supporto per flussi di dati, quali sono considerate a livello di file system, e vengono gestite a livello globale dalla classe `File`. Le operazioni specifiche sul contenuto dei file dipendono dall'organizzazione e dal tipo di accesso per essi previsti. Il linguaggio Java prevede per i file le due tipologie di accesso sequenziale e diretto, distinguendo inoltre tra file formattati e non formattati.

17.1 Flussi e file sequenziali

I file ad *accesso sequenziale* di Java sono manipolati principalmente dai metodi delle classi `InputStream` e `OutputStream`, `FileInputStream` e `FileOutputStream`, `FilterInputStream` e `FilterOutputStream`, `DataInputStream` e `DataOutputStream`, le ultime due delle quali implementano le rispettive interfacce `DataInput` e `DataOutput`.

17.1.1 Classe "File"

Gli oggetti della classe `File` rappresentano file o directory afferenti al file system del calcolatore da gestire quali *entità atomiche*. Ognuno di essi è individuato da un *pathname*, assoluto oppure relativo alla directory di lavoro corrente, che è strutturato secondo le convenzioni del sistema.

I principali metodi della classe sono preposti alla creazione di file o directory, all'acquisizione dei loro pathname o delle directory di appartenenza, alla definizione dei privilegi di lettura e scrittura, alla loro ridenominazione o rimozione dal file system.

File

```
File (String path)
File (File dir, String name)

String getName()
String getPath()
boolean exists()

boolean isFile()
boolean isDirectory()
String[] list()

boolean mkdir()
boolean renameTo (File dest)
boolean delete()
```

- I costruttori della classe sono preposti alla creazione di oggetti di tipo `File`, afferenti al file system del calcolatore, il cui pathname può essere espresso con modalità distinte. Nella *prima forma* il parametro del costruttore definisce il pathname completo del file. Nella *seconda forma*, il costruttore richiede l'identificatore `name` del file e la directory `dir` di collocazione: qualora sia **`null`**, viene assunta quale directory di collocazione la directory corrente.
- I metodi `getName()` e `getPath()` forniscono, rispettivamente, l'identificatore del `File` a cui è inoltrato il messaggio oppure il pathname completo dello stesso.
- Il metodo `exists()` controlla l'esistenza o meno del file a cui è inoltrato il messaggio. Similarmente, i metodi `isFile()` e `isDirectory()` stabiliscono se l'oggetto destinatario del messaggio è un file oppure una directory.
- Il metodo `list()` è attivato in risposta a messaggi inoltrati a directory. Fornisce una lista dei file in essa presenti.
- Il metodo `mkdir()` assegna la struttura di directory all'oggetto `File` a cui è inoltrato il messaggio. In modo similare il metodo `renameTo()` ridenomina il file destinatario del messaggio, così che sia identificato dal pathname specificato dal parametro `dest`. Il metodo `delete()` rimuove il file destinatario: qualora si tratti di una directory, questa *deve* essere vuota.

Si è già evidenziato nel Paragrafo 16.5.1 come Java fornisca una classe di servizio, denominata `JFileChooser` e residente nel package `javax.swing`, tramite la quale l'utente è messo in condizione di selezione i file di interesse attraverso una finestra di dialogo con cui navigare nel file system.

La classe `SelezioneFile` *(Scheda 17.1b) fornisce un esempio di acquisizione di file operata ricorrendo all'impiego di* `JFileChooser`.

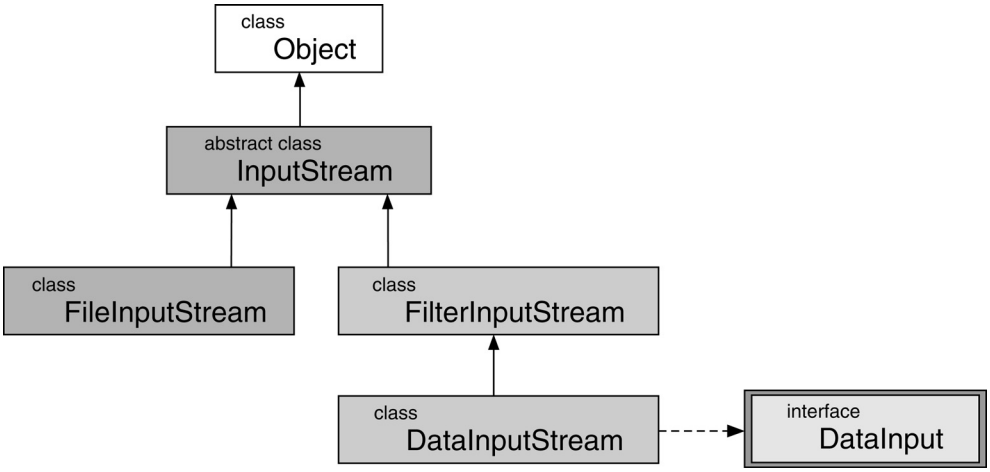


Figura 17.1 Gerarchia di classi e interfacce per la gestione dei file di input.

17.1.2 Classi "InputStream" e "OutputStream"

Si tratta di due classi astratte rappresentative di sequenze di byte. Gli oggetti della classe `InputStream` definiscono flussi in ingresso da cui è possibile leggere singoli byte o gruppi di byte. Simmetricamente, gli oggetti della classe `OutputStream` definiscono flussi in uscita su cui è possibile trascrivere byte. In entrambe le classi sono presenti metodi suscettibili di sollevare eccezioni della classe `IOException`. Costituiscono le classi radice delle corrispondenti classi destinate alla modellazione dei file *non formattati* (`FileInputStream`, `FileOutputStream`) e di quelli *formattati* (`DataInputStream`, `DataOutputStream`) implementanti le relative interfacce (`DataInput`, `DataOutput`). La Figura 17.1 evidenzia la gerarchia che lega le classi e l'interfaccia dei file in lettura, essendo analoga quella dei file in scrittura.

17.1.2.1 Classe "InputStream"

I metodi della classe `InputStream` causano la lettura di (gruppi di) byte e l'eventuale riposizionamento del puntatore a *stream* utilizzabili in lettura.

InputStream

```
abstract int read()
int read (byte b[])
int read (byte b[], int off, int len)
long skip (long n)
```

- Il metodo astratto `read()` definisce il modello della funzione che implementa le operazioni di lettura. Ogni applicazione ridefinente la classe ne deve fornire un'implementazione che restituisce il successivo carattere presente in ingresso sullo stream, con il vincolo che il raggiungimento della condizione di **`eof`** debba restituire il valore –1.

- I metodi `read()` forniscono l'implementazione di default dell'omonimo metodo astratto. Nella forma a *singolo parametro* il metodo acquisisce nell'array `b` un numero di caratteri pari alla lunghezza dell'array. Nella forma a *tre parametri* il metodo legge nell'array `b`, a decorrere dalla posizione `off`, un numero di caratteri al più pari al valore di `len`.
- Il metodo `skip()` scarta al più `n` caratteri dallo stream di ingresso restituendo il numero di caratteri effettivamente ignorati.

17.1.2.2 Classe "OutputStream"

I principali metodi della classe `OutputStream` modellano le operazioni di scrittura di (gruppi di) byte su *stream* utilizzabili in scrittura.

OutputStream

```
abstract void write (int b)
void write (byte b[])
void write (byte b[], int off, int len)
```

- Il metodo astratto `write()` definisce la funzione implementante le operazioni di scrittura. Ogni applicazione che ridefinisca la classe deve fornirne una propria implementazione.
- I metodi `write()` causano la trascrizione sullo stream dei caratteri dell'array `b`. Nella *seconda forma*, il metodo trascrive al più `len` caratteri a decorrere dalla posizione `off` dell'array.

17.1.3 Classi per file sequenziali non formattati

I file non formattati di Java sono costituiti da sequenze di byte che non subiscono alcuna conversione in fase di lettura o di scrittura.[1] Come si è già accennato, essi sono gestiti dalle classi `FileInputStream` e `FileOutputStream`. Gli oggetti della classe `FileInputStream`, sottoclasse di `InputStream`, modellano flussi per file sequenziali da aprire in lettura. In modo simmetrico gli oggetti della classe `FileOutputStream`, sottoclasse di `OutputStream`, definiscono flussi per file sequenziali utilizzabili in scrittura.

La connessione tra gli stream su cui Java effettua le operazioni di lettura/scrittura e i corrispondenti oggetti di classe `File` depositari delle informazioni lette o scritte avviene tramite i costruttori delle classi `FileInputStream` e `FileOutputStream`, ai quali i `File` di interesse sono passati quali parametri. Tali operazioni possono causare eccezioni della classe `FileNotFoundException`.

17.1.3.1 Classe "FileInputStream"

I metodi della classe `FileInputStream` causano la lettura di (gruppi di) byte e l'eventuale riposizionamento del puntatore a *file* aperti in lettura, oltre ad effettuarne la chiusura. Gran parte dei principali metodi della classe costituiscono riscritture, più specifiche ma con stessa semantica, degli omonimi metodi di lettura della corrispondente sovraclasse `InputStream` e pertanto non verranno qui ridefiniti.

[1] Per tale motivo i file non formattati vengono anche denominati file binari.

FileInputStream

```
FileInputStream (String name)
FileInputStream (File file)

int read ()
int read (byte b[])
long skip (long n)
void close()
```

- I costruttori `FileInputStream()` sono destinati alla creazione di stream utilizzabili in lettura e associati a file sequenziali, dei quali può essere fornito l'identificatore `name` oppure l'oggetto `file` che li sostiene.
- Il metodo `close()` è preposto alla chiusura di un file con rilascio delle risorse di sistema associate.

17.1.3.2 Classe "FileOutputStream"

I metodi della classe `FileOutputStream` causano la trascrizione di (gruppi di) byte su *file* aperti in scrittura, oltre ad effettuarne la chiusura. Gran parte dei principali metodi della classe costituiscono riscritture degli omonimi metodi della corrispondente sovraclasse `OutputStream` e non vengono ridefiniti.

FileOutputStream

```
FileOutputStream (String name)
FileOutputStream (File file)
FileOutputStream (String name, boolean append)

void write (int b)
void write (byte b[])
void write (byte b[], int off, int len)
void close()
```

- I costruttori `FileOutputStream()` sono destinati alla creazione di stream di uscita associabili a file sequenziali, dei quali può essere definito l'identificatore `name` oppure l'oggetto `file` che li sostiene. I file possono eventualmente essere aperti in estensione, piuttosto che in sola (ri)scrittura, qualora nel costruttore sia specificato il parametro `append` con valore **`true`**.
- Il metodo `close()` è preposto alla chiusura di un file con rilascio delle risorse di sistema associate.

Le applicazioni `CopiaFile` *(Scheda 17.1a) e* `SelezioneFile` *(Scheda 17.1b) forniscono esempi di gestione di file sequenziali non formattati, operata in entrambi i casi ricorrendo all'impiego delle classi* `FileInputStream` *e* `FileOutputStream` *ma utilizzando tecniche diverse per la selezione dei file.*

Scheda 17.1a - class CopiaFile

Argomenti

Copiatura di file non formattati.

```
import java.io.*;
public class CopiaFile {
   public static void main (String[] args)
                              throws IOException, EOFException {
      FileInputStream inpF = null;
      FileOutputStream outF = null;
      if (args.length != 2)
         System.out.println ("Omesso parametro per file");
      else {
         inpF = new FileInputStream (args[0]);
         outF = new FileOutputStream (args[1]);
         int ch = inpF.read();
         while (ch > 0) {
            outF.write (ch);
            ch = inpF.read();
         }
         inpF.close();
         outF.close();
      }
   }
}
```

La classe `CopiaFile` opera la copiatura di un file sorgente in un corrispondente file di destinazione. I due file sono rispettivamente individuati, a livello di codice, dall'identificatore `inpF`, di tipo `FileInputStream`, e dall'identificatore `outF` di tipo `FileOutputStream`. Gli identificatori dei file, quali sono noti al file system, vengono passati all'applicazione mediante gli argomenti `args[0]` e `args[1]` del metodo `main()`. Le eventuali eccezioni sono fatte rimbalzare all'interprete Java per la gestione di default.

Inizialmente la classe inizializza a **`null`** i puntatori `inpF` e `outF` agli stream dei file.

Inizializzazione oggetti

```
FileInputStream inpF = null;
FileOutputStream outF = null;
```

Successivamente viene effettuato un controllo sui parametri del metodo `main()`, al fine di garantire che all'applicazione siano passati esattamente due identificatori di file.

Controllo parametri del main()

```
if (args.length != 2)
   System.out.println ("Omesso parametro per file");
else { … }
```

In caso di corretta definizione degli stessi, l'applicazione procede creando gli stream dei due file mediante invocazione dei costruttori `FileInputStream()` e `FileOutputStream()` sui parametri `args[0]` e `args[1]` del metodo `main()`.

Inizializzazione dei puntatori

```
if (args.length != 2)
   . . .
else {
   inpF = new FileInputStream (args[0]);
   outF = new FileOutputStream (args[1]);
   . . .
}
```

Le operazioni successive copiano i byte del file sorgente `inpF` nel corrispondente file di destinazione `outF`. La trascrizione è basata su un costrutto **`while`**, la cui terminazione è stabilita dalla restituzione del valore –1 da parte del metodo `read()` applicato a `inpF`; la restituzione di tale valore corrisponde al raggiungimento della condizione di EndOfFile.

Copiatura del file sorgente

```
else {
   . . .
   int ch = inpF.read();
   while (ch > 0) {
      outF.write (ch);
      ch = inpF.read();
   }
   . . .
}
```

Seguono le consuete operazioni di chiusura dei file.

Chiusura dei file

```
else {
   . . .
   inpF.close();
   outF.close();
}
```

Scheda 17.1b - `class` SelezioneFile

Argomenti

Copiatura di file con selezione degli stessi tramite finestra di dialogo.
Classe coinvolta: `JFileChooser`.

```
import java.io.*;
import javax.swing.*;
public class SelezioneFile {
   public static void main (String[] args)
                              throws IOException, EOFException {
      JFileChooser dlgFile = new JFileChooser();
      int dato;
```

```
        FileInputStream inpF = null;
        FileOutputStream outF = null;
        if (dlgFile.showOpenDialog (null) ==
                              JFileChooser.APPROVE_OPTION) {
           File file = dlgFile.getSelectedFile();
           inpF = new FileInputStream (file);
        }
        if (dlgFile.showSaveDialog (null) ==
                              JFileChooser.APPROVE_OPTION) {
              File file = dlgFile.getSelectedFile();
              outF = new FileOutputStream (file);
        }
        do {
           dato = inpF.read();
           if (dato != -1)
              outF.write ((char) dato);
        } while (dato != -1);
        inpF.close();
        outF.close();
     }
  }
```

La classe `SelezioneFile` opera la copiatura di un file sorgente in un corrispondente file di destinazione. I due file sono rispettivamente individuati, a livello di codice, dall'identificatore `inpF`, di tipo `FileInputStream`, e dall'identificatore `outF` di tipo `FileOutputStream`. Entrambi i file sono selezionati nel file system mediante opportune finestre di dialogo.

Inizialmente il programma alloca il dispositivo di selezione `dlgFile` invocando il costruttore `JFileChooser()` dell'omonima classe del package `javax.swing` e inizializza a **`null`** i puntatori `inpF` e `outF` ai due file.

Inizializzazione oggetti

```
JFileChooser dlgFile = new JFileChooser();
FileInputStream inpF = null;
FileOutputStream outF = null;
```

Successivamente viene richiesta al dispositivo `dlgFile` l'apertura della finestra di selezione del file sorgente con il metodo `showOpenDialog()`. La selezione di un file nella finestra di dialogo, seguita dalla approvazione dell'operazione stessa (individuata dalla condizione `JFileChooser.APPROVE_OPTION`), causa l'assegnazione del file selezionato alla variabile `file` per il tramite del metodo `getSelectedFile()`. Tale `file` viene quindi passato quale argomento al costruttore `FileInputStream()` per la modellazione dell'opportuno stream per il file sorgente.

Apertura del dialogo per il file in lettura

```
if (dlgFile.showOpenDialog (null) == JFileChooser.APPROVE_OPTION) {
   File file = dlgFile.getSelectedFile();
   inpF = new FileInputStream (file);
}
```

In maniera del tutto analoga si procede per la modellazione dello stream di uscita `outF`, di classe `FileOutputStream`, avendo cura di invocare nel dispositivo `dlgFile` una finestra di definizione/salvataggio file con il metodo `showSaveDialog()`.

Apertura del dialogo per il file in scrittura

```
if (dlgFile.showSaveDialog (null) == JFileChooser.APPROVE_OPTION) {
   File file = dlgFile.getSelectedFile();
   outF = new FileOutputStream (file);
}
```

Le operazioni che seguono sono destinate alla copiatura dei byte contenuti nel file sorgente individuato da `inpF` nel corrispondente file di destinazione `outF`.

Copiatura del file sorgente

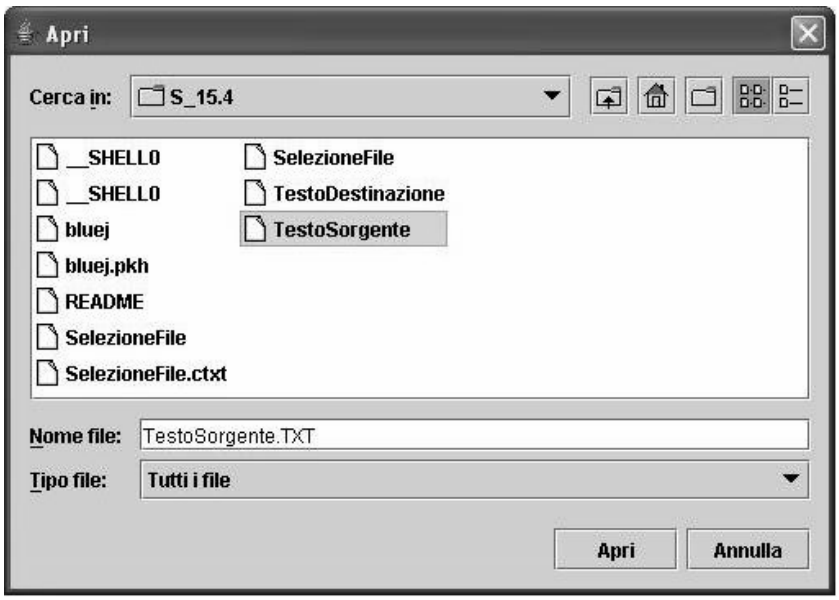
```
do {
   dato = inpF.read();
   if (dato != -1)
      outF.write ((char) dato);
} while (dato != -1);
```

Concludono il programma le consuete operazioni di chiusura di entrambi i file.

Chiusura dei file

```
inpF.close();
outF.close();
```

Possibile finestra di selezione del file sorgente:



17.1.4 Classi per file sequenziali formattati

I file formattati comprendono flussi di byte che sono rappresentativi di dati la cui notazione è quella binaria quando allocati in memoria, ma che usualmente sono rappresentati con la loro effettiva struttura indipendente dal sistema. I dati in questione richiedono pertanto opportune procedure di conversione quando trasferiti dalla memoria ad un supporto file e viceversa.

Le principali classi destinate alla gestione di file sequenziali formattati sono le classi `FilterInputStream` e `DataInputStream`, implementante l'interfaccia `DataInput`, per i file in lettura e le corrispondenti classi `FilterIOutputStream` e `DataOutputStream`, implementante l'interfaccia `DataOutput`, per i file in scrittura. Tali classi permettono di trattare in modo indipendente dalla piattaforma valori di tipo primitivo e oggetti di tipo `String`.

17.1.4.1 Interfacce "DataInput" e "DataOutput"

Le interfacce `DataInput` e `DataOutput` definiscono le intestazioni dei metodi utilizzabili per la lettura e la scrittura di tipi di dato primitivi. Costituiscono interfacce comuni sia alle classi `DataInputStream` e `DataOutputStream`, destinate alla gestione di file sequenziali, che alla classe `RandomAccessFile` responsabile della gestione dei file ad accesso diretto.

I principali metodi definiti dall'interfaccia `DataInput` sono preposti all'acquisizione di dati di tipo **`boolean`**, **`byte`**, **`short`**, **`char`**, **`int`**, **`long`**, **`float`** e **`double`**. Simmetricamente operano i metodi di `DataOutput` per la scrittura di dati afferenti agli stessi tipi summenzionati e per la scrittura di stringhe. La specifica completa dei metodi è rimandata alle classi `DataInputStream` e `DataOutputStream`.

17.1.4.2 Classi "FilterInputStream" e "FilterOutputStream"

Le classi `FilterInputStream` e `FilterOutputStream` definiscono le sovraclassi di tutte le classi che modellano stream di ingresso e di uscita che richiedono strumenti di filtraggio per la conversione dei dati trattati. Entrambe poggiano su preesistenti stream da gestire in lettura o scrittura, denominati *underlying input/output stream*, ai quali aggiungono ulteriori funzionalità.

Più in particolare la classe `FilterInputStream` sovrascrive tutti gli omonimi metodi di `InputStream` con nuove versioni destinate ad inoltrare le richieste di servizio ricevute ai corrispondenti metodi dello stream di ingresso sottostante.

In maniera analoga anche i metodi di `FilterInputStream` costituiscono riscritture degli omonimi metodi di `OutputStream`, che sono adatte ad inoltrare le richieste di servizio ricevute ai corrispondenti metodi dello stream di uscita sottostante.

17.1.4.3 Classi "DataInputStream" e "DataOutputStream"

Le classi utilizzabili dall'utente per la lettura e la scrittura su file di dati primitivi sono le classi `DataInputStream` e `DataOutputStream`. La classe `DataInputStream` definisce file sequenziali in lettura, implementando l'interfaccia `DataInput`. Analogamente la classe `DataOutputStream` definisce file sequenziali in scrittura e implementa l'interfaccia `DataOutput`.

DataInputStream

```
DataInputStream (InputStream in)

int skipBytes (int n)
boolean readBoolean()
byte readByte()
short readShort()
char readChar()
int readInt()
long readLong()
float readFloat()
double readDouble()
```

- Il costruttore `DataInputStream()` è destinato alla modellazione dello stream di ingresso da cui leggere dati residenti sul preesistente `InputStream` definito dal parametro `in`.
- Il metodo `skipBytes()` scarta esattamente `n` byte del sottostante stream di ingresso. Termina l'esecuzione dopo che tutti i byte specificati sono stati ignorati, oppure è stata identificata la terminazione dello stream, oppure è stata sollevata l'eccezione `IOException`.
- Il metodo `readBoolean()` legge *un* byte dallo stream di ingresso, trattando il valore *zero* a rappresentazione di **`false`** e ogni altro valore a rappresentazione di **`true`**. Il metodo `readByte()` legge *un* byte dallo stream di ingresso, interpretandolo come valore di tipo **`byte`**.
- I metodi `readShort()` e `readChar()` leggono *due* byte dallo stream di ingresso, interpretandoli rispettivamente come valore di tipo **`short`** oppure **`char`**.
- I metodi `readInt()` e `readLong()` leggono rispettivamente *quattro* e *otto* byte dallo stream di ingresso, interpretandoli come valore di tipo **`int`** oppure **`long`**.
- I metodi `readFloat()` e `readDouble()` leggono rispettivamente *quattro* e *otto* byte dallo stream di ingresso, interpretandoli come valore di tipo **`float`** oppure **`double`**.

DataOutputStream

```
DataOutputStream (OutputStream out)

void writeBoolean (boolean v)
void writeByte (int v)
void writeShort (int v)
void writeChar (int v)
void writeInt (int v)
void writeLong (long v)
void writeFloat (float v)
void writeDouble (double v)
void writeChars (String s)
```

- Il costruttore `DataOutputStream()` è destinato alla modellazione dello stream di uscita su cui trascrivere dati da far risiedere sul preesistente `OutputStream` definito dal parametro `out`.

- Il metodo `writeBoolean()` scrive *un* byte sullo stream di uscita, utilizzando il valore *zero* a rappresentazione di **`false`** e il valore *uno* a rappresentazione di **`true`**. Il metodo `writeByte()` trascrive il valore `v`, occupando un byte sullo stream di uscita.
- I metodi `writeShort()` e `writeChar()` trascrivono sullo stream di uscita il valore `v`, di tipo rispettivamente **`short`** e **`char`**, utilizzando *due* byte.
- I metodi `writeInt()` e `writeLong()` trascrivono sullo stream di uscita il valore `v`, di tipo rispettivamente **`int`** e **`long`**, utilizzando *quattro* oppure *otto* byte.
- I metodi `writeFloat()` e `writeDouble()` trascrivono sullo stream di uscita il valore `v`, di tipo **`float`** ovvero **`double`**, utilizzando *quattro* oppure *otto* byte.
- Il metodo `writeChars()` trascrive la stringa `s` sullo stream di uscita nella forma di una sequenza di caratteri. A tal fine effettua ripetute chiamate del metodo `writeChar()`.

La classe `RwTipiPrimitivi` *(Scheda 17.1c) fornisce un esempio di lettura e scrittura di tipi primitivi su file formattati.*

Scheda 17.1c - `class` RwTipiPrimitivi

Argomenti

Scrittura e lettura di tipi primitivi con file formattati.

```
import java.io.*;
public class RwTipiPrimitivi {
   public static void main (String[] args) throws IOException {
      DataOutputStream outF;
      DataInputStream inpF;
      outF = new DataOutputStream (new FileOutputStream
                                        ("FileDiProva"));
      outF.writeBoolean (false);
      outF.writeDouble (12.24);
      outF.writeChar ('a');
      outF.writeLong (172);
      outF.close();
      inpF = new DataInputStream (new FileInputStream
                                        ("FileDiProva"));
      System.out.println ("Valore boolean: " + inpF.readBoolean() );
      System.out.println ("Valore double : " + inpF.readDouble() );
      System.out.println ("Valore char : " + inpF.readChar() );
      System.out.println ("Valore long : " + inpF.readLong() );
      inpF.close();
   }
}
```

La classe `RwTipiPrimitivi` trascrive una serie di dati primitivi su uno stream di classe `DataOutputStream`, associandolo al file denominato `"FileDiProva"`. Lo stesso file viene successivamente letto al fine di controllare la correttezza dei dati precedentemente registrati.

Dopo aver dichiarato le variabili `inpF` e `outF`, con cui identificare `"FileDiProva"` in termini di flusso formattato durante le due fasi di scrittura e successiva lettura,

Dichiarazione oggetti

```
DataOutputStream outF;
DataInputStream inpF;
```

l'applicazione assegna a `outF` lo stream formattato di classe `DataOutputStream`, avente come stream sottostante lo stream di tipo `FileOutputStream` che è associato al file `"FileDiProva"` aperto in scrittura.

Apertura del file in scrittura

```
outF = new DataOutputStream (new FileOutputStream ("FileDiProva"));
```

Su detto stream vengono registrati, con formato indipendente dalla piattaforma, quattro valori di tipi primitivi diversi. A completamento delle operazioni di scrittura il file viene chiuso.

Registrazione di valori di tipo primitivo

```
outF.writeBoolean (false);
outF.writeDouble (12.24);
outF.writeChar ('a');
outF.writeLong (172);
outF.close();
```

Successivamente lo stesso `"FileDiProva"` viene aperto in lettura, associandolo allo stream di ingresso `FileInputStream` e filtrandolo attraverso lo stream `DataInputStream`.

Apertura del file in lettura

```
inpF = new DataInputStream (new FileInputStream ("FileDiProva"));
```

Nel nuovo contesto vengono espletate operazioni di lettura simmetriche delle precedenti operazioni di scrittura per il controllo dei valori registrati.

Lettura di valori di tipo primitivo

```
System.out.println ("Valore boolean: " + inpF.readBoolean() );
System.out.println ("Valore double : " + inpF.readDouble() );
System.out.println ("Valore char : " + inpF.readChar() );
System.out.println ("Valore long : " + inpF.readLong() );
inpF.close();
```

17.2 Flussi e file ad accesso diretto

I file ad accesso diretto sono modellati da oggetti della classe `RandomAccessFile`. In tali file è possibile scrivere o leggere dati in corrispondenza di specifiche posizioni del supporto. La classe implementa i metodi delle interfacce `DataInput` e `DataOutput` precedentemente trattate. La gerarchia delle classi e interfacce è riportata in Figura 17.2.

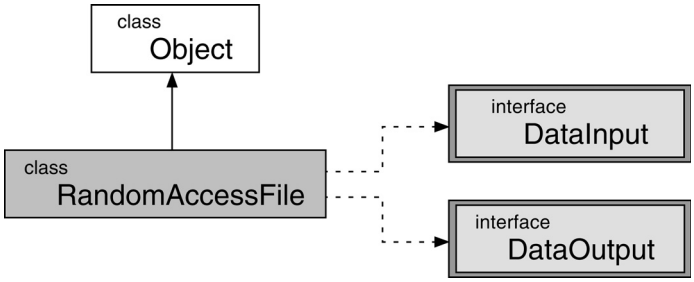


Figura 17.2 Gerarchia di classi e interfacce per la gestione dei file ad accesso diretto.

17.2.1 Classe "RandomAccessFile"

Oltre agli usuali metodi forniti per la scrittura e la lettura di dati di tipo primitivo, già visti per i file sequenziali, la classe `RandomAccessFile` include alcuni metodi per il posizionamento esplicito del puntatore al file in corrispondenza al quale effettuare le operazioni di interesse.

RandomAccessFile

```
RandomAccessFile (String name, String mode)
RandomAccessFile (File file, String mode)

long getFilePointer()
void seek (long pos)
void close()
```

- I costruttori `RandomAccessFile()` sono destinati alla costruzione di uno stream da cui leggere, e su cui opzionalmente scrivere, dati relativi al file individuato dall'identificatore `name` oppure definito dal parametro `file`. L'ulteriore parametro `mode` stabilisce se il file va aperto in sola lettura, nel caso il suo valore sia `"r"`, oppure sia in lettura che in scrittura qualora il valore sia `"rw"`.
- Il metodo `getFilePointer()` restituisce il valore corrente del puntatore al file, valutato conteggiando il numero di byte che lo separano dall'inizio dello stream.
- Il metodo `seek()` assegna al puntatore al file il valore `pos`, indicante la posizione del byte in corrispondenza al quale effettuare la successiva operazione di lettura o di scrittura. Un posizionamento del puntatore oltre l'estensione corrente del file non causa di per sé la variazione di quest'ultima: l'estensione viene modificata solamente da eventuali operazioni di scrittura eccedenti l'ultimo byte corrente del file.

La classe `FileRandom` *(Scheda 17.2a) fornisce un esempio di gestione di file ad accesso diretto.*

Scheda 17.2a - class FileRandom

Argomenti

Scrittura e lettura con file ad accesso diretto.

```
import java.io.*;
public class FileRandom {
   public static void main (String[] args)
                                    throws IOException, EOFException {
      FileOutputStream outF = new FileOutputStream ("FileDiProva");
      for (char ch = 'a'; ch <= 'z'; ch++)
         outF.write (ch);
      outF.close();
      RandomAccessFile inpF = new RandomAccessFile ("FileDiProva", "r");
      for (int k = 6; k >=2; k--) {
         inpF.seek (k);
         char ch = (char) inpF.readByte();
         System.out.println (ch);
      }
      inpF.close();
   }
}
```

La classe `FileRandom` fornisce un esempio d'uso di file ad accesso diretto. Inizialmente la classe genera un file sequenziale, denominato `"FileDiProva"` a livello di file system e individuato dalla variabile `outF` a livello di codice, in cui vengono scritti sequenzialmente i caratteri dell'alfabeto. Si osservi che l'uso del metodo `write()` comporta, per ciascun carattere, la scrittura del byte meno significativo della codifica UNICODE di `ch`.

Creazione del file con accesso sequenziale

```
FileOutputStream outF = new FileOutputStream ("FileDiProva");
for (char ch = 'a'; ch <= 'z'; ch++)
   outF.write (ch);
outF.close();
```

Successivamente lo stesso file viene aperto in lettura come file di tipo `RandomAccessFile` e identificato dall'oggetto `inpF`.

Apertura del file con accesso diretto

```
RandomAccessFile inpF = new RandomAccessFile ("FileDiProva", "r");
```

Dal file vengono quindi letti i caratteri da `'g'` a `'c'` per il tramite del metodo `readByte()`, dopo averli identificati per posizione mediante il metodo di localizzazione `seek()`. I caratteri sono trascritti sul dispositivo standard di uscita per controllo.

Lettura del file con accesso diretto

```
for (int k = 6; k >=2; k--) {
   inpF.seek (k);
   char ch = (char) inpF.readByte();
   System.out.println (ch);
}
inpF.close();
```