

17/11/20

Programmazione Orientata agli Oggetti

Flussi e File

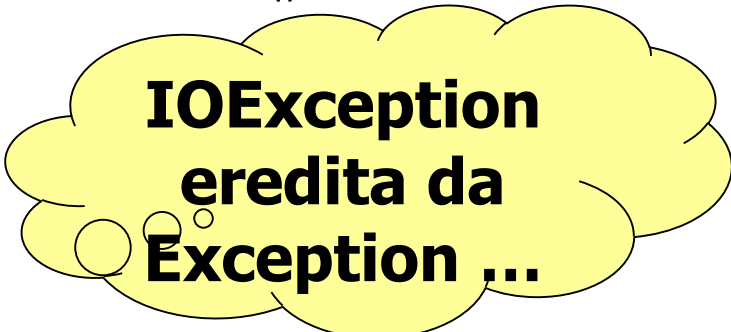
Libero Nigro

Premessa

- Dicesi **flusso** (o **stream**) una successione di dati *prelevati da* una certa sorgente o *forniti a* una certa destinazione
- La *sorgente* può essere: un **file**, la **tastiera**, una **connessione di rete** etc
- La *destinazione* può essere un **file**, il **video**, una **connessione di rete** etc
- Il package **java.io** consente di lavorare in modo uniforme con i flussi di ingresso/uscita indipendentemente dalla loro sorgente/destinazione
- Lo zoo delle classi di java.io è molto ricco (più di sessanta classi). Tuttavia è possibile imparare rapidamente a gestire i flussi più comuni attraverso esempi
- I flussi possono essere: **binari** (o non interpretati a priori), **tipati**, **testuali**, ad **oggetti**
- I file sono di norma acceduti in veste sequenziale: per leggere la 10-ma componente, bisogna comunque attraversare le prime 9.

Classi base per i flussi binari

- **InputStream** (classe astratta)
 - abstract int **read()** throws **IOException**
 - int read(byte[] b) – si basa su read()
 - int read(byte[] b, int off, int len) – si basa su read()
 - int available()
 - void close()
 - ...
- **OutputStream** (classe astratta)
 - abstract void **write**(int b) throws **IOException**
 - void write(byte[] b) – si basa su write(un byte)
 - void write(byte[] b, int off, int len) – si basa su write(...)
 - void flush()
 - void close()
 - ...



**IOException
eredita da
Exception ...**

- L'operazione `read()` è bloccante per il thread che la esegue, se nessun byte è disponibile al momento sullo stream in lettura
- `read()` ritorna un byte come intero tra 0 e 255
- Se il flusso è terminato, la `read()` ritorna -1. Per distinguere questo valore da tutti i “normali” valori di un byte, il tipo di ritorno di `read()` è appunto **int** e non **byte**
- L'operazione `write()` riceve come parametro un int in quanto, in generale, un'espressione che coinvolge byte è comunque di tipo int. Del risultato vengono presi unicamente gli 8 bit (byte) meno significativi, mentre i restanti 24 sono ignorati

Esempi di classi eredi concrete

- FileInputStream
- FileOutputStream

che consentono rispettivamente di lavorare in ingresso e uscita su file al “livello di byte”. La visione a byte è la più bassa possibile. Dopo tutto, *un qualunque file è sempre costituito da una successione di byte* (visione non interpretata del loro contenuto)

Copia di file

```
import java.io.*;
public class Copia{
    public static void main( String []args ) throws IOException {
        InputStream source=new FileInputStream("f1.dat");
        OutputStream dest=new FileOutputStream("f2.dat");
        int dato; //notare la dichiarazione
        for(;;){
            dato=source.read();
            if( dato== -1 ) break; //end of file di f1
            dest.write( dato );
        }//for
        source.close();
        dest.close();
    }//main
}//Copia
```

Il ciclo di lettura dal file source si può scrivere equivalentemente utilizzando il metodo available() che restituisce il numero di byte disponibili per la lettura nello stream

```
for(;;){
    if( source.available()==0 ) break;
    dato=source.read();
    dest.write( dato );
}//for
```

Crittografia (elementare)

```
import java.io.*;
public class Crittografia{
    public static void main( String []args ) throws IOException {
        if( args.length==0 ){ System.out.println("Attesa chiave"); System.exit(-1);}
        int chiave=Integer.parseInt( args[0] ); //assunta >=3
        InputStream source=new FileInputStream("c:\\poo-file\\source.dat");
        OutputStream dest=new FileOutputStream("c:\\poo-file\\dest.dat");
        int dato;
        for(;;){
            dato=source.read();
            if( dato== -1 ) break;
            dest.write( crittografa( dato, chiave ) );
        }
        source.close();
        dest.close();
    } //main
    static byte crittografa( int d, int chiave ){
        return (byte)(d+chiave );
    }
} //Crittografia
```

Per decrittare, si usa la stessa
chiave ma negativa

Flussi non bufferizzati

- Gli esempi di flussi/file considerati in precedenza sono (nominalmente) *unbuffered*, ossia scambiano dati (es. byte) direttamente con la mediazione di operazioni al livello del sistema operativo sottostante, che può accedere es. al disco al livello di ogni singolo byte (inefficiente)
- Java, tuttavia, offre anche la possibilità di utilizzare flussi/file mediante da un *buffer* la cui dimensione può anche essere specificata dal programmatore. In questo modo, le operazioni di accesso ai dati, in lettura/scrittura, avvengano *tramite* il buffer e possono risultare molto più efficienti
- Si nota che il disco *rigido* di un computer è progettato per essere utilizzato naturalmente lavorando a *blocchi di byte* (es. 256 o 512 byte per volta). Un blocco può dunque essere trasferito dal disco al buffer o viceversa, mentre il programma ottiene/scrive singoli dati dal/sul buffer.
- Si parla di *flushing* quando, in scrittura, si svuota il buffer trasferendo il suo contenuto su file. Dualmente in lettura

Flussi bufferizzati

- Si ribadisce che l'utilizzo di un buffer fa sì che scrivendo, ad es., un byte esso venga copiato sul buffer e non immediatamente sul file. Il contenuto del buffer verrà riversato sul file quando il buffer è pieno o quando si richiede un *flush()* sul flusso. La chiusura di un flusso comporta automaticamente il flushing del contenuto residuo del buffer. Alcune classi di file consentono, mediante un parametro di un costruttore, l'*auto-flushing*. Es. in scrittura, dopo ogni singola `println()`.
- Considerazioni simili si possono ripetere per le operazioni di lettura: il prossimo dato verrà prelevato dal buffer, se questo è non vuoto. Quando vuoto, il buffer viene riempito con dati provenienti dal file etc.
- Volendo lavorare con flussi bufferizzati binari, es. file, sono utili le classi **BufferedInputStream** e **BufferedOutputStream** che hanno un costruttore per specificare la dimensione del buffer, diversamente (il che va bene in molti casi) si utilizza una dimensione di default. L'impiego di tali classi è esemplificato di seguito:
InputStream in=new BufferedInputStream(new FileInputStream(nomefile));
- Dopo questo, si lavora su **in** così come si è visto in precedenza sulla versione non bufferizzata `FileInputStream`. E similmente in scrittura.

Le interfacce DataOutput e DataInput

DataOutput (parziale):

```
writeByte( int b )
writeInt( int i )
writeShort( short s )
writeLong( long l )
writeFloat( float f )
writeDouble( double d )
writeChar( char c )
writeBoolean( boolean b )
writeChars( String s ) ogni char 2 byte
writeUTF( String s )
// compact Unicode Text Format
```

Classi di flussi tipati:

DataOutputStream

– eredita da OutputStream e implementa DataOutput

DataInputStream

– eredita da InputStream e implementa DataInput

DataInput (parziale):

```
byte readByte();
int readInt()
short readShort()
long readLong()
float readFloat()
double readDouble()
char readChar();
boolean readBoolean()
String readUTF()
```

Più esattamente: **modified UTF-8**

che rappresenta alcuni caratteri

UNICODE con 1 solo byte, altri con
2 byte, altri con 3 byte

Creazione di un file di interi

```
import java.io.*;
import java.util.*;
public class Crea{
    public static void main( String []args )throws IOException {
        //per semplicita' non si usa la bufferizzazione
        DataOutputStream dos=new DataOutputStream(
            new FileOutputStream("c:\\poo-file\\f3.dat") ); //apre il file in scrittura
        System.out.println("Fornisci una serie di interi sino al primo 0");
        Scanner sc=new Scanner( System.in );
        int x=0;
        for(;;){
            System.out.print("int>");
            x=sc.nextInt();
            if( x==0 ) break;
            dos.writeInt( x );
        }
        dos.close(); //chiude file
    }
}
```

```
//visualizza contenuto di f3.dat
DataInputStream dis=new DataInputStream(
    new FileInputStream("c:\\poo-file\\f3.dat") ); //apre il file in lettura
System.out.println();
System.out.println("Contenuto del file");
for(;;){
    try{
        x=dis.readInt();
    }catch( EOFException e ){ break; }
    System.out.println( x );
}
}
//for
dis.close();
}
//main
}
}
//Crea
```

Esercizio: modificare il programma in modo che i flussi tipati siano bufferizzati

Osservazioni

- Nella specificazione di una costante stringa che esprime il nome con path di un file, la barra rovesciata va raddoppiata:

```
... new FileInputStream ("c:\\poo-file\\f3.dat")
```

Tale raddoppio non va specificato quando si fornisce es. da tastiera il nome di un file con path. Tutto ciò è legato al fatto che in un programma Java il carattere \ anticipa *sequence di escape*, es. "\n" che significa carriage-return/line-feed. Per esprimere che si desidera proprio il significato di \ occorre raddoppiarlo. Da input, invece, queste considerazioni non si applicano

- Per rendere più flessibile la classe Crea è conveniente non usare una costante stringa ma leggere ad es. da tastiera il nome del file da aggiornare
- Il file da aggiornare deve essere già esistente. Questa è una proprietà di tutti i file aperti in lettura
- Ovviamente, alla prima invocazione del programma il file è vuoto. È importante, da file system, creare il file senza un contenuto. Ad es. in una shell dos si può procedere come segue:
- c:\poo-file>copy con: f3.dat INVIO
CTRL-Z INVIO
che crea f3.dat di dimensione 0 byte.

- Una proprietà importante delle classi di java.io è la **composizione dei flussi**
- Ad es., un DataOutputStream è stato costruito a partire da un FileOutputStream. Il risultato è che il DataOutputStream ottenuto è un file tipato e non più un semplice file di “byte grezzi” (raw bytes)
- In realtà è possibile lavorare sul DataOutputStream ottenuto sia al livello di byte che più ad alto livello in veste tipata, es. come file di interi
- Il carattere tipato di un file non è comunque espresso in modo preciso dalle dichiarazioni, nè è importante l’eventuale estensione del nome del file. È sempre il programmatore che deve garantire che il file corretto è utilizzato da un programma

RandomAccessFile

- Si tratta di una **classe base**. Essa implementa le due interfacce `DataInput` e `DataOutput`
- Per polimorfismo, laddove è atteso ad es. un `DataOutputStream` si può equivalentemente passare un `RandomAccessFile` (raf)
- I file ad accesso diretto possono essere letti e scritti contemporaneamente
- **Attenzione:** non è possibile spostare gli elementi in un raf, ossia non è possibile inserire un nuovo elemento in un punto intermedio
- Un file ad accesso diretto può essere aperto a sola lettura (“r”) o in lettura-scrittura (“rw”) come mostrato di seguito
- In un raf è disponibile l’indicizzazione dei **byte** componenti. Gli indici possibili sono: `[0..length()-1]`. Metodi propri di `RandomAccessFile` sono:
- `long getFilePointer()`
 - ritorna la posizione della testina sul file, ossia un indice che può valere da 0 a `length()` (uno oltre la fine del file)
- `long length()`
 - ritorna il numero di byte del file
- `void seek(long pos)`
 - `pos` è atteso tra 0 e `length()`. Pone la testina all’inizio del byte di indice `pos`

- Una volta spostata la testina su una posizione pos del file, allora
 - se pos è all'inizio di un elemento, è possibile comandare una read/write tipata
 - se pos è alla fine del file (pos==length()) è possibile comandare solo una write tipata
- La testina si sposta automaticamente dopo un'operazione di read o write.
- Se il numero richiesto di byte (di un dato) non esiste, si solleva una EOFException erede di IOException

```
static boolean esiste( String nome, int x ) throws IOException{
    RandomAccessFile f=new RandomAccessFile( nome, "r" );
    int inf=0, sup=(int)(f.length()/4)-1; boolean result=false;
    for(;;){
        if( inf>sup ) break;
        int med=(inf+sup)/2;
        f.seek( med*4 );
        int elem=f.readInt();
        if( elem==x ){ result=true; break; }
        if( elem>x ) sup=med-1;
        else inf=med+1;
    }
    f.close();
    return result;
} //esiste
```

Ricerca binaria su
su un raf di interi ordinato

Insertion sort su file di interi

- Sia f un file di interi ordinato per valori crescenti
- Sia x un intero da aggiungere ad f rispettando l'ordine
- Si crea un file temporaneo tmp su cui si copiano tutti gli elementi di f minori di x, quindi si scrive x, quindi si copiano i restanti elementi di f
- Il programma assume che, alla fine dell'operazione di aggiornamento selettivo, il programmatore provveda, operando al livello di sistema operativo, a cancellare il file originario e a ridenominare il file temporaneo con il nome del file originario

```
import java.io.*;
import java.util.*;
public class AggiornamentoSelettivo{
    public static void main( String []args ) throws IOException{
        Scanner sc=new Scanner( System.in );
        System.out.print("nome file=");
        String nome=sc.nextLine();
        System.out.print("intero da aggiungere");
        int x=sc.nextInt();
        inserisci( nome, x );
    } //main
```


- Il file temporaneo `tmp` è stato mappato sul file fisico “`tmp`” del file system. Non avendo utilizzato il path completo per il file esterno, esso risiede nella *directory di default* (cioè la directory di lavoro).
- La directory di lavoro coincide con la *directory di progetto*.
- Pertanto, “`tmp`” viene creato es. nella directory **corso-poo-2020-2021** del workspace in uso.
- Le operazioni di “manutenzione” di sistema, ossia rimozione del file originario e ridenominazione del file temporaneo col nome del file originario, possono essere *anche* realizzate dall’interno del programma Java con la mediazione di oggetti di tipo **File** (si veda più avanti per i dettagli).

Flussi testuali

- Contengono caratteri stampabili (lettere, cifre, segni di punteggiatura, spazi, ...) più le marche di *fine linea*
- L'esatta composizione di una marca di fine linea dipende dal sistema operativo. Ad es. su Windows è la combinazione di due caratteri di controllo: carriage-return e line-feed. Una marca di fine linea è evocata in uscita dalla sequenza di escape `'\n'`
- Un flusso testuale può essere visto come un testo, ossia una successione di linee. È possibile ispezionare/modificare un file testo con un comune editor di testo (es. notepad di Windows)
- Esistono delle gerarchie di classi apposite per i flussi di testo, basate rispettivamente su **Reader** e **Writer**
- Due classi concrete spesso utilizzate sono **BufferedReader** e **PrintWriter**. **PrintWriter** può essere combinata con la classe **BufferedWriter** per ottenere la bufferizzazione durante le operazioni di uscita
- Interessante è anche la classe **PrintStream** che offre alcune semplificazioni rispetto a **PrintWriter**. `System.out` è un oggetto **PrintStream**

BufferedReader (lista parziale dei metodi)

```
String readLine()  
void close()
```

PrintWriter (parziale)

```
void print[ln]( String s )  
void print[ln]( tipo_di_base x );  
void println()  
void flush()  
void close()
```

Salvataggio dell'agenda

- Per esemplificare l'uso di file di tipo testo, si riconsidera la gerarchia di classi dell'agenda telefonica e si mostrano i metodi salva/ripristina, es. realizzati come metodi default nell'interfaccia Agenda:

```
public void salva( String nomeFile ) throws IOException{  
    PrintWriter pw=new PrintWriter( new FileWriter(nomeFile) );  
    for( Nominativo n: this )  
        pw.println( n ); //si scrive su pw il toString di n  
    pw.close();  
} //salva
```

- All'atto pratico può essere utile creare il **PrintWriter** in versione bufferizzata come segue:

```
PrintWriter pw=new PrintWriter(  
    new BufferedWriter( new FileWriter(nomeFile) ) );
```

Ripristino dell'agenda

```
public void ripristina(String nomeFile) throws IOException{
    BufferedReader br=new BufferedReader( new FileReader(nomeFile) );
    String linea=null;
    StringTokenizer st=null;
    LinkedList<Nominativo> tmp=new LinkedList<Nominativo>();
    //tmp e' utile per far fronte a malformazioni del file
    boolean okLettura=true;
    for(;;){
        linea=br.readLine();
        if( linea==null ) break; //eof di br
        st=new StringTokenizer(linea, " -");
        try{
            String cog=st.nextToken(); String nom=st.nextToken();
            String pre=st.nextToken(); String tel=st.nextToken();
            tmp.add( new Nominativo( cog, nom, pre, tel ) ); //aggiunge in coda
        }catch(Exception e){
            okLettura=false; break;
        }
    }
    br.close();
    if( okLettura ){
        this.svuota();
        for( Nominativo n: tmp ) this.aggiungi(n);
    }
    else throw new IOException();
} //ripristina
```

- **FileReader** associa ad un file di tipo testo un convertitore che trasforma caratteri ASCII generati sul file system locale (es. Win) in caratteri UNICODE richiesti da Java
- **FileWriter** associa ad un file di tipo testo un convertitore che trasforma caratteri UNICODE in caratteri ASCII richiesti dal file system locale (es. Win)
- Quando fallisce la lettura di una linea da un buffered reader, si ritorna una stringa null
- **FileReader** va sostituito con **InputStreamReader** quando il **BufferedReader** è “attaccato” alla tastiera.

Lettura di stringhe da tastiera

- Si può evitare l'uso della classe **Scanner** come segue:
BufferedReader **br**=
 new BufferedReader(new **InputStreamReader**(System.in));
System.out.print(“Fornisci una stringa=”);
String linea=**br**.readLine();

PrintStream (es. System.out)

- È capace di rendere un OutputStream (da cui deriva) idoneo per stamparvi dati primitivi in modo conveniente e testuale. Non solleva IOException. Piuttosto una situazione di errore setta un flag sull'oggetto PrintStream che è interrogabile col metodo checkError
- Costruttori esistono per creare un PrintStream con la capacità di **auto-flushing**: ad ogni println, o emissione di un byte-array o di una marca di fine linea '\n'
- I caratteri corrispondenti alla stampa di un dato primitivo sono emessi in forma di byte, codificati secondo le convenzioni del sistema operativo locale utilizzato. In altre parole, qui non serve ricorrere a FileWriter
- Lista di metodi (parziale):
PrintStream(File f), PrintStream(OutputStream out),
PrintStream(OutputStream out, boolean autoflush),
PrintStream(String nomefile)
void print[ln](tipo di dati primitivo o oggetto), //versioni overloaded
void printf(String format, Object ...), void printf(Locale l, String format, Object ...),
void flush(),
void close()
- Le usuali modalità d'uso di PrintStream sono quelle già note su System.out.

Flussi di Oggetti

- È possibile salvare il contenuto di una agendina anche utilizzando il concetto di *flusso di oggetti* e connesso meccanismo di **serializzazione**
- Si aggiunge alla testata della classe Nominativo che essa implementa altresì l'interfaccia **Serializable** (che è senza metodi, ossia è una *marker* interface).
- Considerato che gli oggetti interni a Nominativo (stringhe) sono essi stessi già serializzabili, diventa possibile per il compilatore Java far diventare un oggetto nominativo una sequenza di byte suscettibile di memorizzazione (*persistenza*) e ripristino.
- Il nome serializzazione deriva dal fatto che ad ogni oggetto (riferimento) viene associato un *numero seriale* univoco tale che se l'oggetto, per via di aliasing, dovesse essere re-incontrato durante lo stesso processo di serializzazione, solo il riferimento al suo numero seriale viene re-introdotto sul flusso di oggetti. In altre parole, l'uso dei numeri seriali permette di serializzare gli oggetti una volta sola quando si processa un grafo (ragnatela) di oggetti comunque complesso.
- **Vincolo:** la classe degli oggetti serializzati (qui Nominativo) non dovrebbe cambiare tra il momento in cui si realizza la serializzazione ed il momento in cui si ripristinano gli oggetti serializzati (de-serializzazione).

Classi per flussi di oggetti

- **ObjectOutputStream**
 - *estende **OutputStream** e implementa, tra l'altro, l'interfaccia **DataOutput***
 - `void writeObject(Object o)` //scrive **o** in forma serializzata
 - `void close()`
- **ObjectInputStream**
 - *estende **InputStream** ed implementa, tra l'altro, l'interfaccia **DataInput***
 - `Object readObject()` //legge e deserializza un oggetto
 - `void close()`

Salvataggio di una agendina mediante serializzazione

Per esemplificare, si re-implementano i metodi salva/ripristina della interfaccia Agendina usando flussi di oggetti.

```
public void salva( String nomeFile ) throws IOException{  
    ObjectOutputStream oos=  
        new ObjectOutputStream(  
            new FileOutputStream( nomeFile ) );  
    for( Nominativo n: this ){  
        oos.writeObject( n );  
    }  
    oos.close();  
} //salva
```

Ripristino di una agendina mediante serializzazione

```
public void ripristina( String nomeFile ) throws IOException {  
    ObjectInputStream ois=new ObjectInputStream(  
        new FileInputStream( nomeFile ) );  
  
    this.svuota();  
    Nominativo n=null;  
    for(;;){  
        try{  
            n=(Nominativo)ois.readObject();  
            this.aggiungi( n );  
        }  
        catch( ClassNotFoundException e1 ){ ois.close; throw new IOException(); }  
        catch( ClassCastException e2 ){ ois.close; throw new IOException(); }  
        catch( EOFException e3 ){ ois.close(); break; }  
    }  
} //for  
} //ripristina
```

Eventualmente ci si può cautelare,
prima di svuotare l'agenda, leggendo i
nominativi su una linked list come fatto in
precedenza

serialVersionUID

- Ovviamente è inevitabile che le classi evolvano nel tempo (es. si re-implementano alcuni metodi, si aggiungono/eliminano campi e/o metodi etc). Tutto ciò può determinare problemi a deserializzare oggetti precedentemente salvati, cioè relativi a una versione precedente della classe.
- Nei limiti del possibile, Java consente in qualche modo di mantenere “compatibilità” tra versioni diverse delle classi, rendendo possibile la deserializzazione, con responsabilizzazione del programmatore.
- Il programmatore può farsi generare il numero seriale unico associato ad una versione di una classe con l’utility (presente nel JDK)

serialver NomeClasse INVIO

- Lo strumento **serialver** (eventualmente anche in veste grafica se attivato con l’opzione –show) fornisce un valore long associato alla classe (trascurando i campi static e transient) es.

NomeClasse: static final long serialVersionUID=-1923124576134167197L;

- A questo punto, si può “forzare” la compatibilità di una nuova versione della classe con quella vecchia, mantenendo in essa il numero seriale della versione precedente.

```
class NomeClasse implements Serializable{  
    ...  
    public static final long serialVersionUID=-1923124576134167197L;  
    ...  
} //Nome classe
```

- La deserializzazione di un oggetto di classe NomeClasse *può ancora avvenire*, entro certi limiti, pur se la classe “è cambiata”.
- Se un campo ha cambiato tipo rispetto alla versione precedente, il processo di serializzazione in realtà è incompatibile
- Se l’oggetto serializzato ha più campi di quanti ne risultano nella nuova versione della classe, gli extra-campi vengono ignorati durante la deserializzazione.
- Se l’oggetto serializzato ha meno campi di quanti ne ha la nuova versione della classe, i nuovi campi dell’oggetto deserializzato verranno posti al loro valore di default (null per un oggetto, 0 per un campo numerico etc.) ciò che può dar fastidi nei metodi che si aspettano valori inizializzati opportunamente. La “cura” consiste nel customizzare la deserializzazione (si veda più avanti).

serialver ed Eclipse

- Nell'ambiente integrato Eclipse, non appena si specifica che una classe implementa **Serializable**, la classe viene associata ad un warning che ricorda che essa non dispone ancora del serialVersionUID.
- Cliccando sul warning è possibile richiedere l'inserimento in automatico del numero seriale (ciò che corrisponde ad eseguire il tool serialver) da parte del compilatore.

- Il meccanismo della serializzazione è piuttosto complesso anche se il suo utilizzo, in molti casi pratici, risulta semplice.
- Occorre tener presente che il ripristino degli oggetti da un flusso di oggetti deve avvenire *allo stesso modo* rispetto a come è stato effettuato il salvataggio. Salvando un array di 3 oggetti, al ripristino occorre prelevare in un “unico colpo” un array di 3 oggetti e non tre oggetti separatamente.
- Non vengono serializzati campi di un oggetto che siano etichettati static o transient. Il modificatore transient va usato, ad es., quando alcune variabili di istanza si riferiscono a classi non serializzabili. In questi casi è opportuno *customizzare la serializzazione*. Nella classe serializable si ridefiniscono i metodi writeObject() e readObject(), solo in versione private, come segue:
 - private void writeObject(ObjectOutputStream out) throws IOException{
 out.defaultWriteObject(); //per attivare il meccanismo di serializzazione di base
 *scritture su **out** customizzate*
} //writeObject

- private void readObject(ObjectInputStream in) throws IOException{
 in.defaultReadObject(); //per attivare il meccanismo di deserializzazione di base
 *letture da **in** customizzate*
} //readObject

- Una customizzazione “più radicale” consiste nell’implementare l’interfaccia Externalizable (anziché Serializable) e ridefinire i metodi public
 - public void readExternal(ObjectInput in) throws IOException
 - public void writeExternal(ObjectOutput out) throws IOException
- In questo caso, la classe definisce i **suoi** propri meccanismi per salvare/ripristinare lo stato dell’oggetto (facendosi carico **anche** dello stato della superclasse etc.) sul/dal flusso di oggetti. La serializzazione di base si limita a registrare il descrittore della classe sul flusso. Per tutto il resto, vale la ridefinizione di writeExternal().
- Durante la lettura di un oggetto esternalizzato, si garantisce che venga prima creato un oggetto della classe target con il *costruttore di default*; ogni altra inizializzazione dipende da quanto scritto in readExternal().
- L’esternalizzazione può essere preferibile, per ragioni di efficienza, alla serializzazione standard quando sono in gioco grandi quantità di dati.

Gestione di file e classe File

- La classe File consente di lavorare con i file dal punto di vista del file system, ad es. per verificare se un file esiste su directory, per modificare (cancellare, ridenominare etc.) un file su directory, per gestire il sistema di directory etc

```
File f=new File("f.dat");
```

```
if( f.exists() ) ...
```

```
if( f.delete() ) ... il file è stato effettivamente cancellato ...
```

```
f.renameTo( new File("g.dat") );
```

- Attenzione:** *prima di una delete, renameTo etc. occorre assicurarsi che i file siano chiusi*. In più, per ridenominare un file, occorre usare un nome di file non esistente. Se il file esiste, si può (se è lecito) prima cancellarlo. Vediamo la parte finale del metodo `inserisci(String nome, int x)` che realizza un aggiornamento selettivo su un file di interi:

```

...
tmp.writeInt( x ); //scrivi sicuramente x
if( flag ){
    for(;;){
        tmp.writeInt( y );
        pos=raf.getFilePointer();
        if( pos==raf.length() ) break;
        y=raf.readInt();
    }//for
} //if(flag)
tmp.close(); raf.close();
File f=new File(nome); f.delete();
File ff=new File("tmp");
ff.renameTo( f );
}
} //AggiornamentoSelettivo

```

Importante: il file temporaneo e il file di origine dovrebbero appartenere alla stessa directory; il metodo `renameTo(...)` non è in grado di spostare un file da un posto ad un altro del file system. In alternativa sono utili i metodi della classe di utilità `java.nio.Files`.

Ancora sulla classe File

- Altri metodi della classe File (lista parziale):

boolean isFile() – ritorna true se il file è un normale file, false altrimenti.

boolean isDirectory() – usa la tua fantasia.

deleteOnExit() – chiede che il file/directory venga cancellato quando termina il programma (esce la Java Virtual Machine – JVM).

String getAbsolutePath() – ritorna il path name assoluto di questo file/directory.

String getName() – ritorna il nome del file/directory del path name.

- Si nota che un oggetto File può essere passato ad un costruttore di una class stream (es. `FileInputStream`, etc.) in luogo della stringa del nome esterno del file (es. "f.dat").

Una classe `ObjectFile<T>` con lettura anticipata

T è atteso `Serializable`

void open(Modo modo) throws IOException

apre il file in accordo al modo, che può essere LETTURA o SCRITTURA (valori del tipo enum `Modo`)

void close() throws IOException usa la tua fantasia.

boolean eof() ritorna true se è stata raggiunta la fine fisica del file.

T peek() ritorna il prossimo elemento del file, se esiste, senza avanzare la testina sul file. Se `eof()` è true, solleva una `IOException`.

void get() throws IOException

avanza la testina alla prossima posizione del file; ridefinisce il prossimo elemento. Assume l'apertura in LETTURA. Se `eof()` è true, solleva una `IOException`.

void put(T o) throws IOException

scrive `o`, in forma serializzata, come ultimo elemento del file. Assume l'apertura in SCRITTURA.

String toString ritorna sotto forma di stringa il contenuto del file (supposto non di grandi dimensioni).

Fusione ordinata di due file ordinati

f1:

3	5	13	14	20	30
---	---	----	----	----	----

^

f2:

2	4	8	12
---	---	---	----

^

f3:

2

^

f1:

3	5	13	14	20	30
---	---	----	----	----	----

^

f2:

2	4	8	12
---	---	---	----

^

f3:

2	3	4	5	8	12
---	---	---	---	---	----

^

Il risultato finale è:

f3:

2	3	4	5	8	12	13	14	20	30
---	---	---	---	---	----	----	----	----	----

La classe MergeFile

```
package poo.file;
import java.util.*;
import java.io.*;

public class MergeFile{

    public static void main( String... args) throws IOException{
        System.out.println("Fusione ordinata di due file di interi f1 ed f2 in un file f3");
        Scanner sc=new Scanner( System.in );
        System.out.print("nome esterno di f1 = ");
        String nomeF1=sc.nextLine();
        System.out.print("nome esterno di f2 = ");
        String nomeF2=sc.nextLine();
        System.out.print("nome esterno file fusione f3 = ");
        String nomeF3=sc.nextLine();
    }
}
```

La classe MergeFile

```
ObjectFile<Integer> f1=new ObjectFile<>( nomeF1, ObjectFile.Modolo.LETTURA );
System.out.println("Contenuto di f1:");
System.out.println( f1 );
ObjectFile<Integer> f2=new ObjectFile<>( nomeF2, ObjectFile.Modolo.LETTURA );
System.out.println("Contenuto di f2:");
System.out.println( f2 );
ObjectFile<Integer> f3=new ObjectFile<>( nomeF3,ObjectFile.Modolo.SCRITTURA );
int x1, x2;
while( !f1.eof() && !f2.eof() ){
    x1=f1.peek(); x2=f2.peek();
    if( x1<x2 ){ //minimo proviene da f1
        f3.put( x1 ); f1.get();
    }
    else{ f3.put( x2 ); f2.get(); }
}
//gestione residuo su f1
while( !f1.eof() ){ f3.put( f1.peek() ); f1.get(); }
//gestione residuo su f2
while( !f2.eof() ){ f3.put(f2.peek() ); f2.get(); }
f1.close(); f2.close(); f3.close();
System.out.println("Contenuto di f3:");
System.out.println( f3 );
} //main
} //MergeFile
```


Cenni al package java.nio (New I/O)

Il package nio espone nuove classi e metodi che in alcune circostanze possono migliorare sensibilmente le prestazioni di programmi che usano flussi/file.

Due nuovi concetti introdotti sono **Channel** e **Buffer**. Un *channel*, rispetto ad un classico flusso di dati, rappresenta un file su cui è anche possibile sfruttare meccanismi propri del sottostante sistema operativo, es. capacità di *blocking*, *memory mapped i/o* etc. Un *buffer* è una struttura dati che organizza la lettura/scrittura di dati da/su disco, in modo da garantire un'interazione a *blocchi* con il device. In pratica un buffer nasconde un array e tre variabili di gestione: *position*, *limit* e *capacity*. In scrittura, *position* denota sin dove è stato scritto il buffer, *limit* coincide con *capacity* (capacità dell'array). In una fase di lettura, *limit* è posto all'ultimo valore assunto da *position* nella precedente fase di scrittura, e *position* (resettato) indica sin dove si è letto sino ad ora. Per predisporre un buffer a scrittura si esegue *clear()*. Per commutare su lettura (lettura dal buffer e scrittura su channel/file) si esegue *flip()*.

Classi di buffer: **ByteBuffer**, **IntBuffer**, **ShortBuffer**, **CharBuffer** etc.

Un esempio basato su memory-mapped file

Si considera un file di interi ad es. con un milione di interi (4MB), disordinato, soggetto a ricerca di un elemento x . Si vuole conoscere la posizione (indice) di x sul file, che vale -1 se x non è presente sul file.

Le classi `FileInputStream`, `FileOutputStream` e `RandomAccessFile`

ammettono un metodo per ottenere un channel dal file.

Si vuole studiare l'impatto sulle prestazioni della ricerca quando il file è usato come stream sequenziale, eventualmente buffered, e come memory-mapped di nio.

Si presenta un semplice programma che se il file non esiste lo crea, caricando in esso in modo random tutti gli interi da 0 a n (n escluso).

```
package poo.nio;
import java.io.BufferedInputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.EOFException;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.RandomAccessFile;
import java.nio.MappedByteBuffer;
import java.nio.channels.FileChannel;
import java.util.Random;
```

```
public class FileSearching{
    static final int n=1000000;
    public static void main( String[] args ) throws IOException{
        String nome="c:\\poo-file\\search.dat";
        File f=new File(nome);
        if( !f.exists() ) creaFile( nome );
        int x=new Random().nextInt(n); //genera un intero random tra 0 e n (n escluso)
        int pos=0; //la posizione di x sul file
```

```

long start=System.currentTimeMillis();
pos=linearSearchRandomAccessFile(nome,x);
long end=System.currentTimeMillis();
System.out.println("linearSearchRAF: pos="+pos+" of x="+x+
    " elapsed time="+((end-start))+ "msec");
start=System.currentTimeMillis();
pos=linearSearchDIS(nome,x); //DataInputStream
end=System.currentTimeMillis();
System.out.println("linearSearchDIS: pos="+pos+" of x="+x+
    " elapsed time="+((end-start))+ "msec");
start=System.currentTimeMillis();
pos=linearSearchBDIS(nome,x); //Buffered DIS
end=System.currentTimeMillis();
System.out.println("linearSearchBDIS: pos="+pos+" of x="+x+
    " elapsed time="+((end-start))+ "msec");
start=System.currentTimeMillis();
pos=linearSearchMemoryMappedFile(nome,x);
end=System.currentTimeMillis();
System.out.println("linearSearchMMF: pos="+pos+" of x="+x+
    " elapsed time="+((end-start))+ "msec");
} //main

```

```
static int linearSearchRandomAccessFile( String nome,int x ) throws IOException{
```

```
    RandomAccessFile f=new RandomAccessFile( nome, "r" );
```

```
    int result=-1, elem=0;
```

```
    for( int i=0; i<n; i++ ){//esempio
```

```
        f.seek( i*4 );
```

```
        elem=f.readInt();
```

```
        if( elem==x ){ result=i; break; }
```

```
    }
```

```
    f.close();
```

```
    return result;
```

```
//linearSearchRandomAccessFile
```

```

static int linearSearchDIS( String nome, int x ) throws IOException{
    DataInputStream f=
        new DataInputStream( new FileInputStream(nome) );
    int result=-1, elem=0, i=0;
    for(;;){
        try {
            elem=f.readInt();
            if( elem==x ){ result=i; break; }
        }catch( EOFException e ) { break; }
        i++;
    }
    f.close();
    return result;
}
}

```

```
static int linearSearchBDIS( String nome, int x ) throws IOException{
    DataInputStream f=new DataInputStream(
        new BufferedInputStream( new FileInputStream(nome) ) );
    int result=-1, elem=0, i=0;
    for(;;){
        try {
            elem=f.readInt();
            if( elem==x ){ result=i; break; }
        }catch( EOFException e ) { break; }
        i++;
    }
    f.close();
    return result;
} //linearSearchBDIS
```

```
static int linearSearchMemoryMappedFile( String nome, int x )  
                                           throws IOException{
```

```
    RandomAccessFile f=new RandomAccessFile( nome, "r" );  
    FileChannel channel= f.getChannel(); //ottiene il canale  
    int length=(int)channel.size();  
    MappedByteBuffer buffer= //ottiene un ByteBuffer e lo mappa  
        channel.map(FileChannel.MapMode.READ_ONLY, 0, length);  
    int result=-1, elem=0;  
    for( int i=0; i<n; i++ ){  
        elem=buffer.getInt(i*4);  
        if( elem==x ){ result=i; break; }  
    }  
    f.close();  
    return result;  
  
} //linearSearchMemoryMappedFile
```



```

static void creaFile( String nome ) throws IOException{
    DataOutputStream dos=
        new DataOutputStream( new FileOutputStream(nome) );
    boolean[] a=new boolean[n];
    int c=0, pos=0;
    Random r=new Random();
    while( c<n ){
        while( a[pos=r.nextInt(n)] );
        dos.writeInt(pos);
        c++; a[pos]=true;
    }
    dos.close();
} //creaFile

} //FileSearching

```

Performance di esecuzione

Ipotesi: il file già esiste per cui a nessun algoritmo *si fa pagare* il tempo di creazione del file (non trascurabile).

linearSearchRAF: pos=480380 of x=354539 elapsed time=4434msec

linearSearchDIS: pos=480380 of x=354539 elapsed time=4019msec

linearSearchBDIS: pos=480380 of x=354539 elapsed time=33msec

linearSearchMMF: pos=480380 of x=354539 elapsed time=16msec

Tali risultati dipendono ovviamente dal computer utilizzato e cambiano utilizzando un differente PC.

Tuttavia sono indicativi delle prestazioni relative degli algoritmi.

Altre letture

- Il processo di serializzazione/esternalizzazione e più in generale le problematiche dei flussi e file di Java, ivi compreso il package `java.nio`, possono essere approfonditi, ad es., su:
 - C. Horstmann, G. Cornell, *Core Java*, Vol. II, Advanced Features, Prentice-Hall