

15

Input-Output

Mirko Viroli
`mirko.viroli@unibo.it`

C.D.L. Ingegneria e Scienze Informatiche
ALMA MATER STUDIORUM—Università di Bologna, Cesena

a.a. 2018/2019



Goal della lezione

- Illustrare le API fornite da Java per l'I/O
- Descrivere alcune scelte progettuali e pattern
- Mostrare esempi di applicazione

Argomenti

- Classi per gestire file
- Classi per gestire Stream (di input/output)
- Serializzazione di oggetti
- Classi per gestire file di testo
- Pattern Decorator



Il problema dell'Input/Output

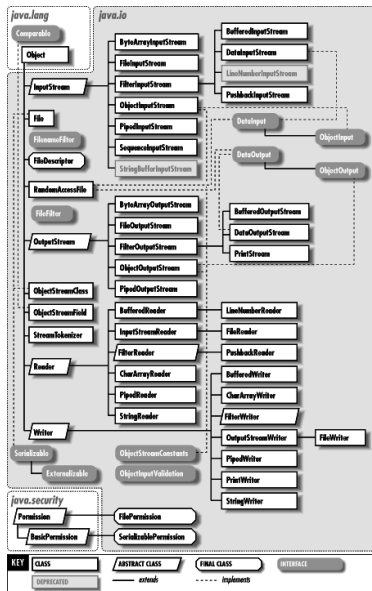
Uno dei problemi fondamentali per un sistema operativo

- Gestire le comunicazioni fra CPU e dispositivi affacciati sul BUS
 - ▶ Console, tastiera, mouse, dischi, rete, sensori, schermo
- Vi sono varie modalità di interazione possibili
 - ▶ sequenziale, random-access, buffered, per carattere/linea/byte/oggetto
- I sistemi operativi offrono vari meccanismi
 - ▶ file, I/O control interface, socket per il networking, video driver

La libreria `java.io.*`

- Fornisce i concetti di File e Stream di dati
- Consente una gestione flessibile dei vari aspetti
- È estesa con la libreria `java.nio`, che vedremo poi
- È la base per altre librerie avanzate per l'I/O (networking,...)
- I/O in ambiente a finestre sarà visto in seguito (GUI..)

La libreria java.io



I macro-elementi della libreria

Outline della lezione

- File
- Stream di ingresso e uscita
- File ad accesso “random”
- Stream di oggetti e serializzazione
- Reader e Writer di testi

- 1 File e loro proprietà
- 2 Input/OutputStream
- 3 Serializzazione di oggetti
- 4 Random access file
- 5 File di testo

File system

- Il file system è un modulo del S.O. che gestisce la memoria secondaria
- Maschera le diversità di dispositivi fisici (HD, CD, DVD, BR, SSD,...)
- Maschera le diversità di contenuti informativi (testi, filmati, archivi,...)
- Fornisce meccanismi per fornire prestazioni, concorrenza, robustezza

File

- Un file system contiene un insieme di **file**
- Un file ha un contenuto informativo, ossia un insieme di byte
 - ▶ interpretabili in vario modo (testi, programmi, strutture dati)
 - ▶ potrebbe essere un file virtuale, che mappa un dispositivo
 - ▶ un caso particolare è la directory (ossia una tabella di ID di file)
- Si ha una organizzazione gerarchica in cartelle (un file ha un path)
- Un file ha un ID, nome, percorso, diritti di accesso, dimensione, ...

La classe `java.io.File`

Usi

- Serve a identificare un preciso file su file systems
 - Permette di ottenere informazioni varie sul file
 - Permette di effettuare alcune operazioni complessive (cancellazione, renaming)
 - Permette di impostare alcune proprietà (se eseguibile, se scrivibile)
 - Permette di ottenere informazioni generali sul file systems
 - Permette di creare cartelle
- ⇒ non include operazioni per accedere al suo contenuto, ma vi si potrà agganciare uno stream



Classe java.io.File: pt1

```
1 public class File implements Serializable, Comparable<File> {
2
3     public File(String pathname) {...}
4     public File(String parent, String child) {...}
5     public File(File parent, String child) {...}
6
7     /* -- Path-component accessors -- */
8     public String getName() {...}
9     public String getParent() {...}
10    public File getParentFile() {...}
11    public String getPath() {...}
12
13    /* -- Path operations -- */
14    public boolean isAbsolute() {...}
15    public String getAbsolutePath() {...}
16    public File getAbsoluteFile() {...}
17    public String getCanonicalPath() throws IOException {...}
18    public File getCanonicalFile() throws IOException {...}
19
20    /* -- Attribute accessors -- */
21    public boolean canRead() {...}
22    public boolean canWrite() {...}
23    public boolean exists() {...}
24    public boolean isDirectory() {...}
25    public boolean isFile() {...}
26    public boolean isHidden() {...}
27    public long lastModified() {...}
28    public long length() {...}
```

Classe java.io.File: pt2

```
1  /* -- File operations -- */
2
3  public boolean createNewFile() throws IOException {...}
4  public boolean delete() {...}
5  public void deleteOnExit() {...}
6  public String[] list() {...}
7  public String[] list(FilenameFilter filter) {...}
8  public boolean mkdir() {...}
9  public boolean renameTo(File dest) {...}
10 public boolean setLastModified(long time) {...}
11 public boolean setReadOnly() {...}
12 public boolean setWritable(boolean writable, boolean ownerOnly) {...}
13 public boolean setWritable(boolean writable) {...}
14 public boolean setReadable(boolean readable, boolean ownerOnly) {...}
15 public boolean setReadable(boolean readable) {...}
16 public boolean setExecutable(boolean executable, boolean ownerOnly)
    {...}
17 public boolean setExecutable(boolean executable) {...}
18 public boolean canExecute() {...}
19
20 /* -- Disk usage -- */
21 public long getTotalSpace() {...}
22 public long getFreeSpace() {...}
23 public long getUsableSpace() {...}
24 }
```

java.io.File in azione (modella un path su File System)

```
1 public class UseFile {
2
3     public static final String FILE_NAME = "/home/mirko/aula/oop/15/prova.bin"
4         ;
5
6     private static Iterable<Method> accessors(Class<?> c) throws Exception {
7         final Collection<Method> list = new ArrayList<>();
8         for (final Method m : c.getMethods()) {
9             if (m.getParameterTypes().length == 0
10                 && m.getName().matches("is.*|get.*|can.*")) { // REGEX
11                 list.add(m);
12             }
13         }
14         return list;
15     }
16
17     public static void main(String[] args) throws Exception {
18         final File f = new File( args.length == 0 ? FILE_NAME : args[0]);
19         for (final Method m : accessors(File.class)) {
20             System.out.println(m.getName() + " " + m.invoke(f));
21         }
22     }
23 }
24
25 // REGEX: http://docs.oracle.com/javase/tutorial/essential/regex/
```



Esempio di output

```
1 getName prova.bin
2 getParent /home/mirko/aula/15
3 isAbsolute true
4 getCanonicalPath /home/mirko/aula/15/prova.bin
5 getPath /home/mirko/aula/15/prova.bin
6 getParentFile /home/mirko/aula/15
7 getAbsolutePath /home/mirko/aula/15/prova.bin
8 getAbsoluteFile /home/mirko/aula/15/prova.bin
9 getCanonicalFile /home/mirko/aula/15/prova.bin
10 canRead true
11 canWrite true
12 isDirectory false
13 isFile true
14 isHidden false
15 canExecute false
16 getTotalSpace 53616242688
17 getFreeSpace 14087458816
18 getUsableSpace 11357081600
19 getClass class java.io.File
```



Accedere al contenuto di un file

Come fare?

- Un file ha un contenuto informativo (potenzialmente di grosse dimensioni)
- Lo si potrebbe leggere (in vari modi)
- Lo si potrebbe scrivere (in vari modi)
- Il suo contenuto potrebbe essere interpretabile in vari modi

Alcuni di tali concetti sono condivisi con altri meccanismi

- File ottenibili da rete
- Networking
- Archivi su database
- Depositi di informazione in memoria

Il concetto di **input/output-stream** è usato come astrazione unificante

Outline

- 1 File e loro proprietà
- 2 Input/OutputStream**
- 3 Serializzazione di oggetti
- 4 Random access file
- 5 File di testo

Overview sugli InputStream e OutputStream in Java

InputStream e OutputStream

- Stream = flusso (di dati)
 - Di base, gestiscono stream leggibili vs. scrivibili binari (`byte`)
 - Sono classi astratte (e non interfacce..)
 - Gestiscono vari meccanismi, via “sottoclassi” e “decorazioni”, tra cui
 - ▶ Lettura/scrittura su file (`FileInputStream`)
 - ▶ Lettura/scrittura su memoria (`ByteArrayInputStream`)
 - ▶ Lettura/scrittura bufferizzata (`BufferedInputStream`)
 - ▶ Lettura/scrittura anche di valori primitivi (`DataInputStream`)
 - ▶ Lettura/scrittura anche di oggetti (`ObjectInputStream`)
- ⇒ ..e corrispondenti versioni Output

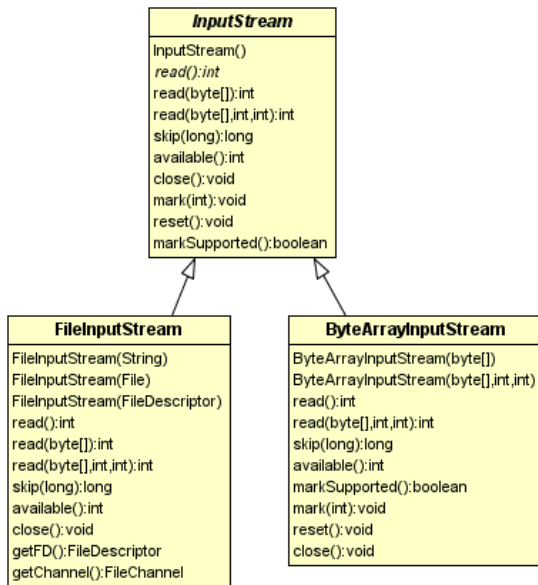
Altre classi, con gestione diversa

- File di testo (`Reader`, `Writer`, e specializzazioni)
- File ad accesso “random” (`RandomAccessFile`)

La classe java.io.InputStream

```
1 public abstract class InputStream implements Closeable {
2
3     // Reads the next byte (0 to 255, -1 is end-of-stream)
4     public abstract int read() throws IOException;
5
6     public int read(byte b[]) throws IOException {...}
7
8     public int read(byte b[], int off, int len) throws IOException
9     {...}
10
11     public long skip(long n) throws IOException {...}
12
13     public int available() throws IOException {...}
14
15     public void close() throws IOException {...}
16
17     public synchronized void mark(int readlimit) {...}
18
19     public synchronized void reset() throws IOException {...}
20
21     public boolean markSupported() {...}
22 }
```


FileInputStream e ByteArrayInputStream



Uso di ByteArrayInputStream

ByteArrayInputStream

- crea un `InputStream` a partire da un `byte[]`
- è un wrapper

```
1 import java.io.*;
2
3 public class UseByteArrayStream {
4
5     public static void main(String[] args) throws IOException {
6         final byte[] b = new byte[] { 10, 20, 30, 40, 50 };
7         final InputStream in = new ByteArrayInputStream(b);
8         int c;
9         try {
10             while ((c = in.read()) != -1) { // C-style
11                 System.out.println(c);
12             }
13         } finally { // assicura la chiusura anche con eccezioni
14             in.close();
15         }
16     }
17 }
```

Il costrutto try-with-resources

Costrutto try-with-resources

- vuole la creazione di un `java.lang.AutoCloseable` come primo argomento
- ne assicura la chiusura
- si possono opzionalmente aggiungere delle `catch` di eccezioni
- andrebbe sempre usato..

```
1 import java.io.*;
2
3 public class UseTryWithResources {
4
5     public static void main(String[] args) throws IOException {
6         final byte[] b = new byte[] { 10, 20, 30, 40, 50 };
7         int c;
8         try (final InputStream in = new ByteArrayInputStream(b)) {
9             while ((c = in.read()) != -1) { // C-style
10                 System.out.println(c);
11             }
12         }
13     }
14 }
```

Esempio StreamDumper

```
1 import java.io.*;
2
3 public class StreamDumper {
4
5     // rendo inaccessibile il costruttore
6     private StreamDumper() {}
7
8     public static void dump(InputStream input) throws IOException{
9         for (int c=0; (c = input.read()) != -1;) {
10             System.out.print(c+"\t");
11         }
12     }
13 }
```



UseStreamDumper – uso uniforme di vari InputStream

```
1 public class UseStreamDumper {
2
3     public static void main(String[] args) throws IOException {
4
5         final byte[] b = new byte[]{10,20,30};
6
7         try(
8             final InputStream input = new ByteArrayInputStream(b);
9             final InputStream input2 = new FileInputStream(UseFile.FILE_NAME);
10            final InputStream input3 = new InputStream(){ // An ad-hoc stream
11                private int count = 100;
12                private final Random r = new Random();
13                public int read() throws IOException {
14                    return (this.count-- > 0 ? r.nextInt(256) : -1);
15                }
16            };
17        ) {
18            StreamDumper.dump(input);
19            System.out.println();
20            StreamDumper.dump(input2);
21            System.out.println();
22            StreamDumper.dump(input3);
23            System.out.println();
24        }
25    }
26 }
```

La classe java.io.OutputStream

```
1 public abstract class OutputStream implements Closeable, Flushable{
2     /**
3      * ..The byte to be written is the eight
4      * low-order bits of the argument b. The 24
5      * high-order bits of b are ignored.
6      */
7     public abstract void write(int b) throws IOException;
8
9     public void write(byte b[]) throws IOException {...}
10
11     public void write(byte b[], int off, int len) throws
12     IOException {...}
13
14     public void flush() throws IOException {...}
15
16     public void close() throws IOException {...}
17 }
```

Stream di uscita – Duale all'InputStream

- Esistono anche le analoghe specializzazioni `ByteArrayOutputStream` e `FileOutputStream`

UseOutputStream

```
1 import java.io.*;
2 import java.util.Random;
3
4 public class UseOutputStream {
5
6     public static void main(String[] args) throws IOException {
7
8         try (
9             final OutputStream output = new FileOutputStream(UseFile.FILE_NAME);
10         ) {
11             // Aggiungo byte random
12             final Random r = new Random();
13             for (int i = 0; i < 100; i++) {
14                 output.write(r.nextInt(256));
15             }
16
17             // Aggiungo un array di byte
18             final byte[] b = new byte[] { 10, 20, 30, 40 };
19             for (int i = 0; i < 10; i++) {
20                 output.write(b);
21             }
22         }
23     }
24 }
25 }
```

UseOutputStream2 – qualche variante

```
1 import java.io.*;
2 import java.util.Random;
3
4 // Due varianti: uso di java.io.File, e catch con try-with-resources
5 public class UseOutputStream2 {
6
7     public static void main(String[] args) {
8
9         try (
10             final OutputStream output =
11                 new FileOutputStream(new File(UseFile.FILE_NAME));
12         ) {
13             // Aggiungo byte random
14             final Random r = new Random();
15             for (int i = 0; i < 100; i++) {
16                 output.write(r.nextInt(256));
17             }
18
19             // Aggiungo un array di byte
20             final byte[] b = new byte[] { 10, 20, 30, 40 };
21             for (int i = 0; i < 10; i++) {
22                 output.write(b);
23             }
24         } catch (final IOException e){
25             System.out.println("Something went wrong!");
26         }
27     }
28 }
```


Salvataggio di strutture dati: List<Byte>

```
1 public class ListOnFile {
2
3     public static void main(String[] args) throws IOException {
4         final List<Byte> list = new ArrayList<>(20); // Creo una lista random
5         final byte[] ar = new byte[20];
6         new Random().nextBytes(ar);
7         for (final byte b : ar) {
8             list.add(b);
9         }
10        System.out.println("Prima: " + list);
11        try (
12            final OutputStream file = new FileOutputStream(UseFile.FILE_NAME)
13        ) {
14            for (final byte b : list) { // La riverso su file
15                file.write(b);
16            }
17        }
18        try (
19            final InputStream file2 = new FileInputStream(UseFile.FILE_NAME)
20        ) {
21            final List<Byte> list2 = new ArrayList<>();
22            int c;
23            while ((c = file2.read()) != -1) { // Ricarico da file
24                list2.add((byte) c);
25            }
26            System.out.println("Dopo: " + list2);
27        }
28    }
29 }
```

Solo byte?

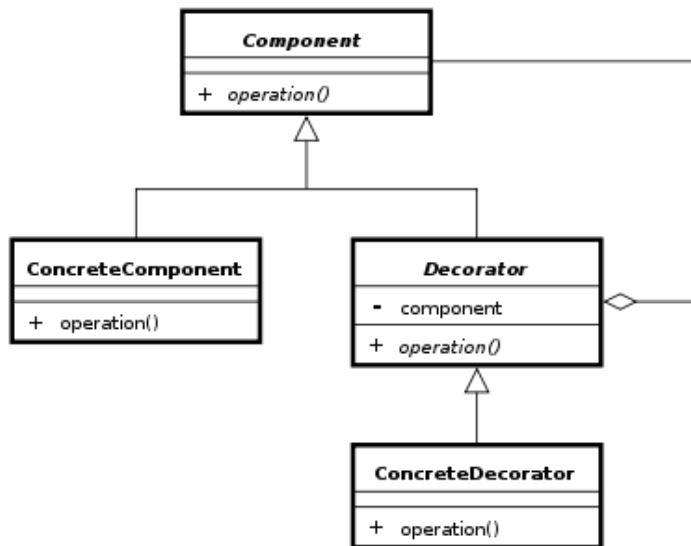
Problema..

- Fornire la possibilità che da un `InputStream` si possano leggere anche `int`, `long`, eccetera
- Ove possibile, ciò dovrebbe essere possibile da qualunque stream

Il concetto di decoratore

- Si definisce `DataInputStream` che estende `InputStream`
 - ▶ e similmente `DataOutputStream` che estende `OutputStream`
- Tale nuova classe comunque fa da wrapper per un `InputStream`, al quale delega le varie operazioni
- Un oggetto di tale nuova classe è un decoratore per quello interno.. visto che ne modifica il funzionamento
- Con questa tecnica è possibile decorare sia un `FileInputStream` che un `ByteArrayInputStream` – o altri

Decorazione, in generale



Decorazione, il caso di DataInputStream



DataInputStream

```
1 public interface DataInput {
2     void readFully(byte b[]) throws IOException {...}
3     boolean readBoolean() throws IOException {...}
4     byte readByte() throws IOException {...}
5     int readUnsignedByte() throws IOException {...}
6     short readShort() throws IOException {...}
7     int readUnsignedShort() throws IOException {...}
8     char readChar() throws IOException {...}
9     int readInt() throws IOException {...}
10    long readLong() throws IOException {...}
11    float readFloat() throws IOException {...}
12    double readDouble() throws IOException {...}
13    String readUTF() throws IOException {...} // non-standard UTF-8
14    ...
15 }
16
17 public class FilterInputStream extends InputStream {...}
18
19 public class DataInputStream
20     extends FilterInputStream implements DataInput{
21     ...
22     public DataInputStream(InputStream in){...}
23     ...
24 }
```

DataOutputStream

```
1 public class DataOutputStream extends OutputStream ... {
2     void writeBoolean(boolean v) throws IOException {...}
3     void writeByte(int v) throws IOException {...}
4     void writeShort(int v) throws IOException {...}
5     void writeChar(int v) throws IOException {...}
6     void writeInt(int v) throws IOException {...}
7     void writeLong(long v) throws IOException {...}
8     void writeFloat(float v) throws IOException {...}
9     void writeDouble(double v) throws IOException {...}
10    void writeBytes(String s) throws IOException {...}
11    void writeChars(String s) throws IOException {...}
12    void writeUTF(String str) throws IOException {...} // non-standard UTF-8
13 }
14
15 public class FilterOutputStream extends OutputStream {...}
16
17 public class DataOutputStream
18     extends FilterOutputStream implements DataOutput {
19     ...
20     public DataOutputStream(OutputStream out) {...}
21     ...
22 }
```



UseDataStream

```
1 import java.io.*;
2
3 public class UseDataStream {
4
5     public static void main(String[] args) throws IOException {
6
7         try (
8             final OutputStream file = new FileOutputStream(UseFile.FILE_NAME);
9             final DataOutputStream dstream = new DataOutputStream(file);
10         ) {
11             dstream.writeBoolean(true);
12             dstream.writeInt(10000);
13             dstream.writeUTF("Ciao");
14             dstream.writeDouble(5.2);
15         }
16
17         try (
18             final InputStream file2 = new FileInputStream(UseFile.FILE_NAME);
19             final DataInputStream dstream2 = new DataInputStream(file2);
20         ){
21             System.out.println(dstream2.readBoolean());
22             System.out.println(dstream2.readInt());
23             System.out.println(dstream2.readUTF());
24             System.out.println(dstream2.readDouble());
25         }
26     }
27 }
```

Altra decorazione: `BufferedInputStream`, `BufferedOutputStream`

Esigenza

- fornire una diversa implementazione interna dello stream
- non legge un byte alla volta, ma riempie un buffer
- questo aumenta le performance nell'accesso a file e rete
- come fornire la funzionalità in modo ortogonale al resto della gestione degli stream?

`BufferedInputStream`, `BufferedOutputStream`

- sono ulteriori decoratori, della stessa forma dei precedenti
- non aggiungono altri metodi
- per come sono fatti i decoratori, possono essere usati in “cascata” a `DataInputStream` e `DataOutputStream`

UseBufferedDataStream

```
1 import java.io.*;
2
3 public class UseBufferedDataStream {
4
5     public static void main(String[] args) throws IOException {
6         // dstream -> bstream -> file
7         try (
8             final OutputStream file = new FileOutputStream(UseFile.FILE_NAME);
9             final OutputStream bstream = new BufferedOutputStream(file);
10            final DataOutputStream dstream = new DataOutputStream(bstream);
11        ){
12            dstream.writeBoolean(true);
13            dstream.writeInt(10000);
14            dstream.writeDouble(5.2);
15            dstream.writeUTF("Prova"); // Scrive in rappresentazione UTF-16
16        }
17        // dstream2 -> bstream2 -> file2
18        try (
19            final InputStream file2 = new FileInputStream(UseFile.FILE_NAME);
20            final InputStream bstream2 = new BufferedInputStream(file2);
21            final DataInputStream dstream2 = new DataInputStream(bstream2);
22        ){
23            System.out.println(dstream2.readBoolean()); // Do not change order!!
24            System.out.println(dstream2.readInt());
25            System.out.println(dstream2.readDouble());
26            System.out.println(dstream2.readUTF());
27        }
28    }
```

UseBufferedDataStream2 – chaining dei costruttori

```
1 import java.io.*;
2
3 public class UseBufferedDataStream2 {
4
5     public static void main(String[] args) throws IOException {
6         // dstream -> bstream -> file
7         try (
8             final DataOutputStream dstream = new DataOutputStream(
9                 new BufferedOutputStream(
10                     new FileOutputStream(UseFile.FILE_NAME)));
11         ){
12             dstream.writeBoolean(true);
13             dstream.writeInt(10000);
14             dstream.writeDouble(5.2);
15             dstream.writeUTF("Prova"); // Scrive in rappresentazione UTF-16
16         }
17         // dstream2 -> bstream2 -> file2
18         try (
19             final DataInputStream dstream2 = new DataInputStream(
20                 new BufferedInputStream(
21                     new FileInputStream(UseFile.FILE_NAME)));
22         ){
23             System.out.println(dstream2.readBoolean()); // Do not change order!!
24             System.out.println(dstream2.readInt());
25             System.out.println(dstream2.readDouble());
26             System.out.println(dstream2.readUTF());
27         }
28     }
```

Decorazione, una visione complessiva



Altri decoratori di InputStream (..e OutputStream)

Sono molteplici, tutti usabili in combinazione

- `CheckedInputStream`: mantiene un “checksum” per verifica integrità
- `CipherInputStream`: legge dati poi processati dopo una cifratura
- `DeflateInputStream`: legge dati e li comprime in formato “deflate”
- `InflaterInputStream`: legge dati e li scompatta dal formato “deflate”
- `ProgressMonitorInputStream`: legge dati con possibilità di “unread”



Ancora sui decoratori

Pro e contro

- Sono un mix di polimorfismo e incapsulamento
- Consentono di comporre funzionalità in modo piuttosto flessibile
- Danno luogo a più flessibilità rispetto all'ereditarietà
- Più complicati da usare e comprendere

Con gli stream, è possibile comporre:

- Uno stream di sorgente dati: `FileInputStream`, `ByteArrayInputStream`, ..
- Uno (o più) stream di gestione interna: `BufferInputStream`, ..
- Uno stream di presentazione dati: `DataInputStream`, `ObjectInputStream`, ..



Outline

- 1 File e loro proprietà
- 2 Input/OutputStream
- 3 Serializzazione di oggetti**
- 4 Random access file
- 5 File di testo

Serializzazione di oggetti

Motivazioni

- Rendere gli oggetti persistenti, e trasferibili a istanze di JVM diverse
- Esempio: memorizzarli su file, su array di byte, trasferirli via rete
- Java Serialization
 - ▶ serializza con relativa semplicità strutture di oggetti anche complicate
 - ▶ a volte è fondamentale apportarvi correzioni ad-hoc

Ingredienti Serialization

- Interfaccia “tag” `java.io.Serializable`
- Classi `ObjectInputStream` e `ObjectOutputStream`
- Keyword `transient` per campi che non devono essere serializzati
- Metodi `readObject` e `writeObject` (e altri) per modificare la serializzazione di default per un oggetto, o per motivi di sicurezza
- Meccanismo UID per gestire versioni diverse delle classi

Diagramma UML con ObjectInputStream



Classe Person e l'interfaccia "tag" Serializable

```
1 public class Persona implements java.io.Serializable {
2
3     private static final long serialVersionUID = 567742502623265945L;
4     private final String nome;
5     private final int annoNascita;
6     private final boolean sposato;
7
8     public Persona(
9         final String nome,
10        final int annoNascita,
11        final boolean sposato
12    ) {
13        this.nome = nome;
14        this.annoNascita = annoNascita;
15        this.sposato = sposato;
16    }
17
18    public String toString() {
19        return this.nome + ":" + this.annoNascita + ":"
20            + (this.sposato ? "spos" : "non-spos");
21    }
22 }
```

Serializable – implementata già da molte classi, non Object

- da implementare per avere oggetti “automaticamente” serializzabili
- ciò non comporta alcun contratto da ottemperare



Classe UseObjectStream

```
1 public class UseObjectStream {
2
3     public static void main(String[] args) throws Exception {
4
5         // ostream -> bstream -> file
6         try (
7             final OutputStream file = new FileOutputStream(UseFile.FILE_NAME);
8             final OutputStream bstream = new BufferedOutputStream(file);
9             final ObjectOutputStream ostream = new ObjectOutputStream(bstream);
10        ){
11            ostream.writeInt(10000);
12            ostream.writeDouble(5.2);
13            ostream.writeObject(new java.util.Date()); // Classe serializ.
14            ostream.writeObject(new Persona("Rossi", 1960, false));
15        }
16
17        // ostream2 -> bstream2 -> file2
18        try (
19            final InputStream file2 = new FileInputStream(UseFile.FILE_NAME);
20            final InputStream bstream2 = new BufferedInputStream(file2);
21            final ObjectInputStream ostream2 = new ObjectInputStream(bstream2);
22        ){
23            System.out.println(ostream2.readInt());
24            System.out.println(ostream2.readDouble());
25            System.out.println(ostream2.readObject()); // carica il Date
26            System.out.println(ostream2.readObject()); // carica la Persona
27        }
28    }
```

Classe ObjectOutputStream: note

Note

- la `writeObject()`/`readObject()` fallisce se l'oggetto non è serializzabile
 - ▶ se la classe dell'oggetto non implementa `Serializable`
 - ▶ la la classe dell'oggetto ha un campo che sia un oggetto non serializzabile
- la `readObject()` fallisce se la classe dell'oggetto non è disponibile
- la `readObject()` fallisce se la classe dell'oggetto è una versione diversa



Come funzionano `writeObject()` / `readObject()`

`ObjectOutputStream.writeObject()`

- Lancia una eccezione se l'oggetto non è serializzabile
- Scrive sullo stream i campi dell'oggetto uno a uno (di tipi primitivi o serializzabili a loro volta)
- Si evitano i campi con modificatore `transient`
- In questo processo, si evita di scrivere due volte uno stesso oggetto

`ObjectInputStream.readObject()`

- Lancia una eccezione se non trova la classe o non è `compatibile`
- Chiama il costruttore senza argomenti della prima sopra-classe non serializzabile, e da lì in giù non chiama altri costruttori
- Ripristina il valore dei campi leggendoli dallo stream
- Lascia inalterati i campi `transient`

Il problema delle versioni di una classe: serialVersionUID

Problema

- Si serializza un oggetto, la classe viene modificata e ricompilata, e quindi si ritira su l'oggetto.. i dati sarebbero facilmente “corrupted”

Soluzione: ogni classe che implementa Serializable...

- .. deve fornire una costante “`long serialVersionUID`” che contiene un numero univoco per quella versione della classe
- Se non corrisponde a quello dell'oggetto caricato si ha eccezione

Fatti

- Se mancante Eclipse segnala warning. Può generarne uno a richiesta.
- Se mancante la JVM ne calcola uno suo ma è sconsigliato.
- Molti lasciano il campo al valore 1, non preoccupandosene.

I campi transient

I campi transient non vengono serializzati. In quali casi servono?

- Campi aggiunti per motivi di performance (p.e., caching di un calcolo), e che quindi possono essere ricostruiti a partire dagli altri campi
- Campi che contengono info specifiche sul run corrente della JVM (p.e., logs), e che quindi non avrebbero più senso quando l'oggetto viene recuperato dallo stream
- Campi che contengono oggetti comunque non serializzabili (p.e., Object), e che quindi porterebbero ad una eccezione
- Campi per i quali si vuole prevedere un meccanismo di serializzazione diverso



CPersona con caching toString, pt1

```
1 public class CPersona implements java.io.Serializable {
2
3     // Generated by Eclipse
4     private static final long serialVersionUID = -8985026380526620812L;
5
6     private String nome;
7     private int annoNascita;
8     private boolean sposato;
9     transient private String cachedToString = null;
10
11     public CPersona(String nome, int annoNascita, boolean sposato) {
12         this.nome = nome;
13         this.annoNascita = annoNascita;
14         this.sposato = sposato;
15     }
16
17     public String getNome() {
18         return nome;
19     }
20
21     public int getAnnoNascita() {
22         return annoNascita;
23     }
24
25     public boolean isSposato() {
26         return sposato;
27     }
```

CPersona con caching toString, pt2

```
1 public void setName(final String nome) {
2     this.nome = nome;
3     this.cachedToString = null;
4 }
5
6 public void setAnnoNascita(final int annoNascita) {
7     this.annoNascita = annoNascita;
8     this.cachedToString = null;
9 }
10
11 public void setSposato(final boolean sposato) {
12     this.sposato = sposato;
13     this.cachedToString = null;
14 }
15
16 private String computeToString() {
17     return this.nome + ":" + this.annoNascita + ":"
18         + (this.sposato ? "spos" : "non-spos");
19 }
20
21 public String toString() {
22     if (this.cachedToString == null) {
23         System.err.println("Log: The cache is empty...");
24         this.cachedToString = this.computeToString();
25     }
26     return this.cachedToString;
27 }
28 }
```


UseTransient

```
1 public class UseTransient {
2
3     public static void main(String[] args) throws Exception {
4
5         try (final ObjectOutputStream out = new ObjectOutputStream(
6             new FileOutputStream(UseFile.FILE_NAME))) {
7             final CPersona p = new CPersona("Rossi", 1960, false);
8             System.out.println("Prima stampa " + p); // cache vuota
9             System.out.println("Seconda stampa " + p); // cache non vuota
10            out.writeObject(new CPersona("Rossi", 1960, false));
11        }
12
13        System.out.println("Ri-carico l'oggetto... ");
14
15        try (final ObjectInputStream in = new ObjectInputStream(
16            new FileInputStream(UseFile.FILE_NAME))) {
17            final CPersona q = (CPersona) in.readObject(); //attenzione al cast!
18            System.out.println("Prima stampa " + q); // cache vuota
19            System.out.println("Seconda stampa " + q); // cache non vuota
20        }
21    }
22 }
23 }
```



Progettazione di una serializzazione ad-hoc

Serializzazione ad-hoc

- Il modello transient/non-transient a volte non è sufficiente
- A volte serve serializzare in modo diverso certi campi
- È possibile definire per la classe serializzabile i metodi
“`void readObject(ObjectInputStream in)`” e
“`void writeObject(ObjectOutputStream out)`”
- Se definiti, `ObjectInputStream` e `ObjectOutputStream` chiamano quelli

Dettagli

- Tali metodi possono cominciare con la chiamata a `defaultReadObject()/defaultWriteObject()`, per leggere i campi non-statici e non-transienti
- Si può quindi proseguire scrivendo/leggendo negli stream in input quello che si vuole

APersona: serializzazione ad-hoc per una data

```
1 public class APersona implements java.io.Serializable {
2     private static final long serialVersionUID = -8985026380526620812L;
3
4     private final String nome;
5     private transient Date lastUse = new Date();
6
7     public APersona(final String nome) {
8         this.nome = nome;
9     }
10
11     public void used() {
12         this.lastUse = new Date();
13     }
14
15     public String toString() {
16         return this.nome + ":"
17             + (this.lastUse == null ? "null" : this.lastUse.getTime());
18     }
19
20     private void writeObject(final ObjectOutputStream out) throws IOException {
21         out.defaultWriteObject(); // accetto il comportamento di default
22         System.err.println("writing");
23     }
24
25     // una sorta di costruttore..
26     private void readObject(final ObjectInputStream in) throws IOException,
27         ClassNotFoundException {
28         in.defaultReadObject();
29         System.err.println("reading");
30         this.lastUse = new Date(); // in lettura, ripristino la data corrente
31     }
```

UseAdHocSerialization

```
1 import it.unibo.apice.oop.p15io.files.UseFile;
2
3 public class UseAdHocSerialization {
4
5     public static void main(String[] args) throws Exception {
6
7         try (final ObjectOutputStream out = new ObjectOutputStream(
8             new FileOutputStream(UseFile.FILE_NAME))){
9             final APersona p = new APersona("Rossi");
10            p.used();
11            System.out.println(p);
12            out.writeObject(p);
13        }
14
15        System.out.println("Ri-carico l'oggetto... ");
16
17        try (final ObjectInputStream in = new ObjectInputStream(
18            new FileInputStream(UseFile.FILE_NAME))){
19            final APersona q = (APersona) in.readObject();
20            System.out.println(q);
21        }
22    }
23 }
```

Serializzazione ad-hoc per java.util.ArrayList, pt1

```
1 package java.util;
2
3 public class ArrayList<E> extends AbstractList<E>
4     implements List<E>, RandomAccess, Cloneable, java.io.Serializable
5 {
6     private static final long serialVersionUID = 8683452581122892189L;
7     private transient Object[] elementData;
8     private int size;
9
10    ...
11    private void writeObject(java.io.ObjectOutputStream s)
12        throws java.io.IOException{
13        // Write out element count, and any hidden stuff
14        s.defaultWriteObject();
15
16        // Write out array length
17        s.writeInt(elementData.length);
18
19        // Write out all elements in the proper order.
20        for (int i=0; i<size; i++)
21            s.writeObject(elementData[i]);
22    }
23 }
```



Serializzazione ad-hoc per java.util.ArrayList, pt2

```
1 private void readObject(java.io.ObjectInputStream s)
2     throws java.io.IOException, ClassNotFoundException {
3     // Read in size, and any hidden stuff
4     s.defaultReadObject();
5
6     // Read in array length and allocate array
7     int arrayLength = s.readInt();
8     Object[] a = elementData = new Object[arrayLength];
9
10    // Read in all elements in the proper order.
11    for (int i=0; i<size; i++)
12        a[i] = s.readObject();
13 }
14
15 }
```



Outline

- 1 File e loro proprietà
- 2 Input/OutputStream
- 3 Serializzazione di oggetti
- 4 Random access file**
- 5 File di testo

Classe RandomAccessFile

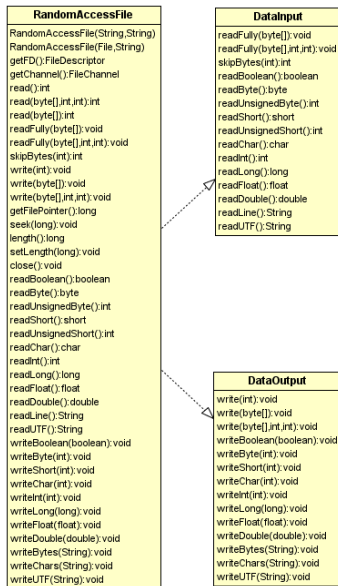
Motivazioni

- Alcuni file sono di grosse dimensioni, e non vengo letti/scritti per intero come nei casi visti finora
- Bensì si modifica qualche elemento “a metà”, o se ne aggiungono in fondo, o si legge un elemento in una data posizione

Classe RandomAccessFile

- Non è usata tramite InputStream o OutputStream
- Fornisce i metodi di DataInput e DataOutput
- Fornisce metodi aggiuntivi:
 - ▶ getFilePointer: torna la posizione corrente nel file
 - ▶ seek: imposta la nuova posizione nel file
 - ▶ length: torna la lunghezza del file
 - ▶ setLength: imposta la lunghezza del file

Classe RandomAccessFile



UseRandomAccessFile

```
1 public class UseRandomAccessFile {
2
3     public static void main(String[] args) throws IOException {
4
5         try (final RandomAccessFile raf = new RandomAccessFile(
6             UseFile.FILE_NAME, "rw")) { // read-write
7
8             for (int i = 0; i < 100000; i++) {
9                 System.out.println("writing: " + i);
10                raf.writeInt(i);
11            }
12            raf.seek(23000 * 4);
13            System.out.println("reading in position 23000*4: " + raf.readInt());
14            raf.setLength(800000);
15            System.out.println("extending the size");
16            raf.seek(123000 * 4);
17            System.out.println("reading in position 123000*4: " + raf.readInt());
18        }
19    }
20
21 }
```



Outline

- 1 File e loro proprietà
- 2 Input/OutputStream
- 3 Serializzazione di oggetti
- 4 Random access file
- 5 File di testo**

File di testo

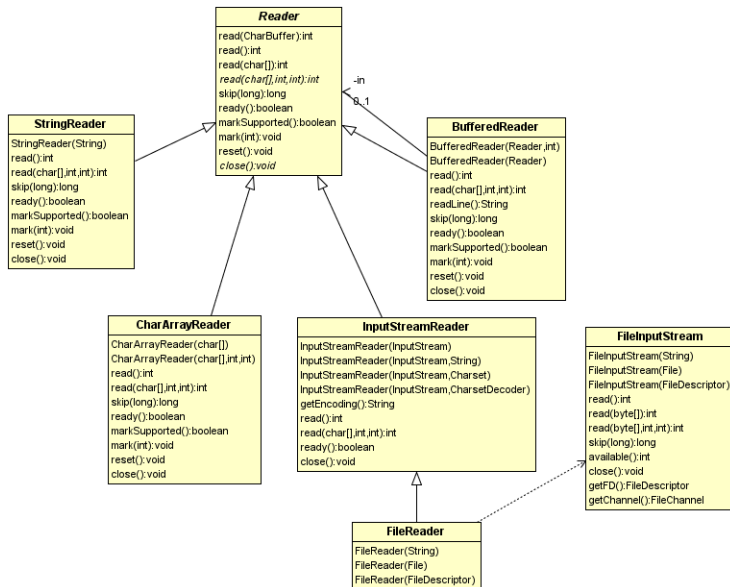
File binari vs file di testo

- Gli stream binari sono quelli visti, sono costituiti da sequenze di byte e il modo col quale si ottengono da questi altri tipi primitivi è standard
- Gli “stream di testo” hanno problematiche diverse:
 - ▶ sono sequenze di caratteri
 - ▶ la codifica (UTF-8, UTF-16, ASCII) potrebbe variare
 - ▶ la codifica potrebbe dettare anche i codici speciali di “a capo” etc.
- Questa gestione richiede classi specifiche

Reader (e Writer)

- Reader: la radice, con metodi per leggere caratteri, e linee di testo
- StringReader: decoratore per prelevare da String
- BufferedReader: decoratore per ottimizzare gli accessi
- InputStreamReader: decoratore che incapsula un InputStream
- FileReader: sua estensione per leggere da file via FileInputStream

UML classi relative ai reader (writer analoghi)



Esempio: UseReadersWriters

```
1 public class UseReadersWriters {
2
3     public static final String FILE_NAME = "/home/mirko/aula/oop/15/a.txt";
4
5     public static void main(String[] args) throws Exception {
6         try {
7             final BufferedWriter w = new BufferedWriter(new FileWriter(FILE_NAME))
8         } {
9             w.write("Prova");
10            w.newLine();
11            w.write("di file");
12            w.newLine();
13        }
14        try {
15            final BufferedReader r = new BufferedReader(new FileReader(FILE_NAME))
16        } {
17            System.out.println(r.readLine());
18            System.out.println(r.readLine());
19            System.out.println(r.readLine()); // null, indica la fine del file!
20        }
21        try {
22            final BufferedReader r = new BufferedReader(new FileReader(FILE_NAME))
23        } {
24            String line = null;
25            while( (line = r.readLine()) != null){
26                System.out.println(line);
27            }
28        }
```

Esempio: UseStreamReadersWriters

```
1 // Soluzione con specifica della codifica dei caratteri
2 public class UseStreamReadersWriters {
3
4     public static void main(String[] args) throws Exception{
5
6         try(final BufferedWriter w = new BufferedWriter(
7             new OutputStreamWriter(
8                 new FileOutputStream(UseReadersWriters.FILE_NAME),"UTF-16"))){
9             w.write("Prova");
10            w.newLine();
11            w.write("di file");
12            w.newLine();
13        }
14
15        try(final BufferedReader r = new BufferedReader(
16            new InputStreamReader(
17                new FileInputStream(UseReadersWriters.FILE_NAME),"UTF-16"))){
18            System.out.println(r.readLine());
19            System.out.println(r.readLine());
20            System.out.println(r.readLine()); // null
21        }
22    }
23 }
```



Il caso di: System.in e System.out

```
1 public class SystemInOut {
2
3     public static void main(String[] args) throws Exception{
4
5         final InputStream is = System.in;
6         final BufferedReader br = new BufferedReader(new InputStreamReader(is));
7         System.out.println(br.readLine()); // può lanciare una IOException
8
9         final PrintStream ps = System.out;
10        // una sotto-classe di OutputStream, che incapsula un Writer
11        ps.println("Un comando noto"); // non lancia eccezioni!
12        ps.format("Altro comando noto.. %d %f %s\n", 10,20.2,"prova");
13
14        // scrittura di file con PrintStream
15        final PrintStream ps2 = new PrintStream(UseReadersWriters.FILE_NAME);
16        ps2.println("prova");
17        ps2.print(10);
18        ps2.println();
19        ps2.close();
20    }
21 }
```



Riassunto classi

Identificazione di un file (o directory)

- File

Accesso random

- RandomAccessFile

Lettura di file dati (scrittura duale)

- FileInputStream + BufferedInputStream + DataInputStream
- FileInputStream + BufferedInputStream + ObjectInputStream

Lettura di file di testo (scrittura duale)

- FileReader + BufferedReader
- FileInputStream + InputStreamReader + BufferedReader

