16
Input-Output

Mirko Viroli
mirko.viroli@unibo.it

C.D.L. Ingegneria e Scienze Informatiche
Alma Mater Studiorum—Università di Bologna, Cesena

a.a. 2022/2023

Outline

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

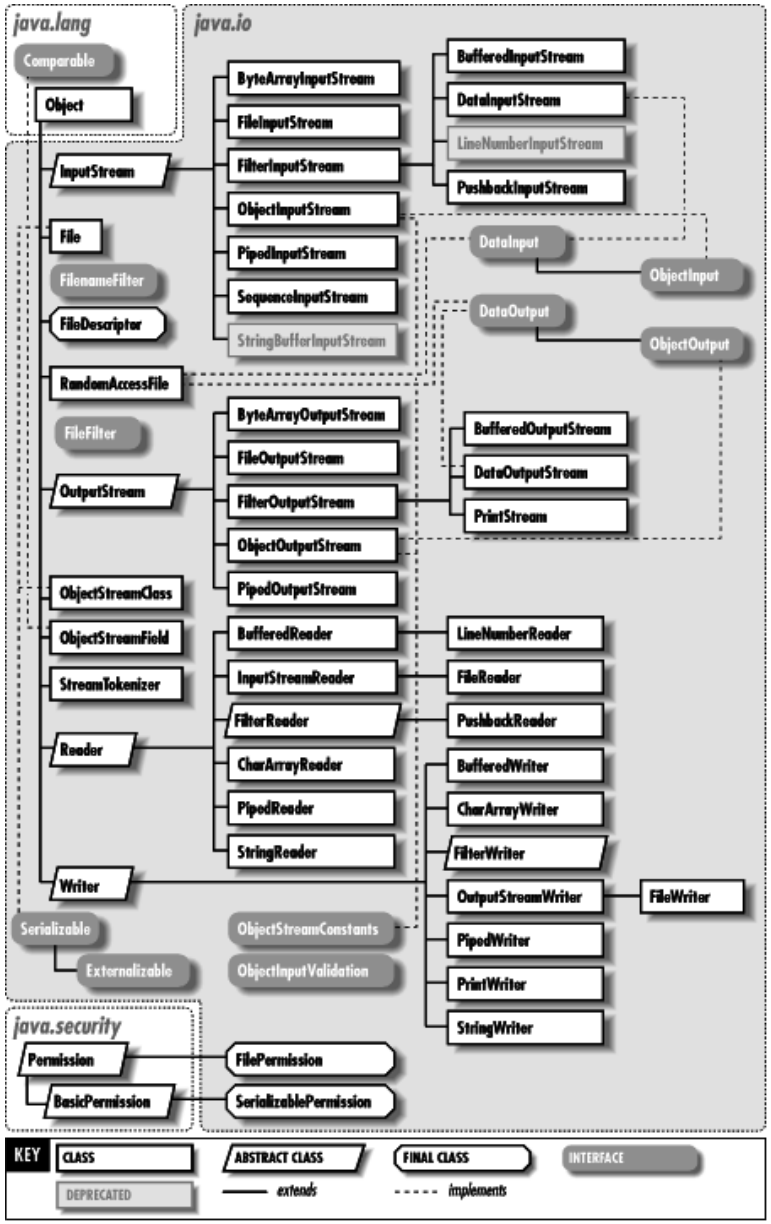
Uno dei problemi fondamentali per un sistema operativo

- 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 alcune funzionalità nella libreria `java.nio`, che vedremo poi
- È la base di librerie avanzate (networking,...), anche non-JDK (JSON,...)
- I/O con l'utente in ambiente a finestre è realizzato con le 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

Outline

I File

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

$\Rightarrow$ 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)

```java
public class UseFile {

  public static final String SEP = File.separator;
  // public static final String FILE_NAME =
  // "/home/mirko/aula/oop/Prova.bin"; // non portabile!!
  public static final String FILE_NAME =
    System.getProperty("user.home") + SEP + "aula" + SEP + "oop" + SEP + "
     Prova.bin";

  private static Iterable<Method> accessors(Class<?> c) throws Exception {
    final Collection<Method> list = new ArrayList<>();
    for (final Method m : c.getMethods()) {
      if (m.getParameterTypes().length == 0
          && m.getName().matches("has.|is.*|get.*|can.*")) { // REGEX
        list.add(m);
      }
    }
    return list;
  }

  public static void main(String[] args) throws Exception {
    final File f = new File( args.length == 0 ? FILE_NAME : args[0]);
    for (final Method m : accessors(File.class)) {
      System.out.println(m.getName() + " " + m.invoke(f));
    }
  }
}
// REGEX: http://docs.oracle.com/javase/tutorial/essential/regex/
```

Esempio di output

```
getName prova.bin
getParent /home/mirko/aula/oop
isAbsolute true
getCanonicalPath /home/mirko/aula/oop/prova.bin
getPath /home/mirko/aula/oop/prova.bin
getParentFile /home/mirko/aula/oop
getAbsolutePath /home/mirko/aula/oop/prova.bin
getAbsoluteFile /home/mirko/aula/oop/prova.bin
getCanonicalFile /home/mirko/aula/oop/prova.bin
canRead true
canWrite true
isDirectory false
isFile true
isHidden false
canExecute false
getTotalSpace 53616242688
getFreeSpace 14087458816
getUsableSpace 11357081600
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

- Risorse interne al classpath Java
- Networking e file di rete
- Archivi su database
- Depositi di informazione in memoria

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

Outline

Overview sugli `InputStream` e `OutputStream` in Java

`InputStream` e `OutputStream`

- Stream = flusso (di dati)
- Di base, gestiscono flussi binari (di `byte`) leggibili vs. scrivibili
- Sono classi astratte (e non interfacce...)
- Possono essere specializzate da "sottoclassi" e "decorazioni", tra cui
  - Per diverse sorgenti e destinazioni di informazione, ad esempio su file (`FileInputStream`) o su memoria (`ByteArrayInputStream`)
  - Per diversi formati di informazione, ad esempio valori primitivi (`DataInputStream`) o interi oggetti Java (`ObjectInputStream`)

  $\Rightarrow$ ...e corrispondenti versioni `Output`

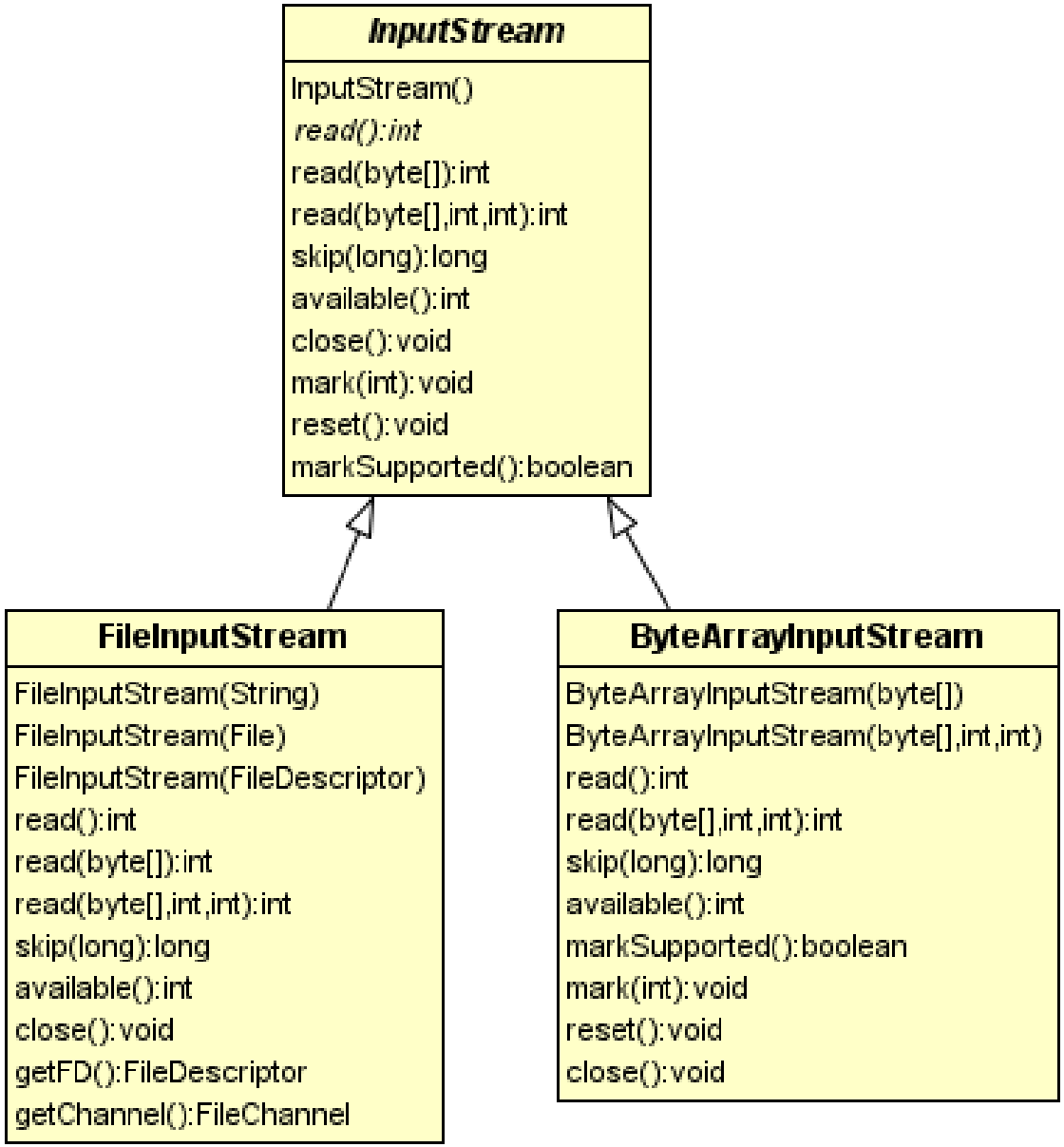
Tipicamente usati per alimentare altre classi

- File di testo (`Reader`, `Writer`, e specializzazioni)
- Librerie avanzate comunemente usate per l'accesso al file system tipicamente hanno metodi che accettano (`In`/`Out`)`putStream`

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     public long skip(long n) throws IOException {...}
11
12     public int available() throws IOException {...}
13
14     public void close() throws IOException {...}
15
16     public synchronized void mark(int readlimit) {...}
17
18     public synchronized void reset() throws IOException {...}
19
20     public boolean markSupported() {...}
21 }
```

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, -1, 40, -58 };
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   }
```

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 <code>b</code>. The 24
5       * high-order bits of <code>b</code> 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
       IOException {...}
12
13     public void flush() throws IOException {...}
14
15     public void close() throws IOException {...}
16 }
```

Stream di uscita – Duale all'InputStream

- Esistono anche le analoghe specializzazioni ByteArrayOutputStream e FileOutputStream

UseOutputStream

```java
import java.io.*;
import java.util.Random;

public class UseOutputStream {

  public static void main(String[] args) throws IOException {

    try (
      final OutputStream output = new FileOutputStream(UseFile.FILE_NAME);
    ) {
      // Aggiungo byte random
      final Random r = new Random();
      for (int i = 0; i < 100; i++) {
        output.write(r.nextInt(256));
      }

      // Aggiungo un array di byte
      final byte[] b = new byte[] { 10, 20, 30, 40 };
      for (int i = 0; i < 10; i++) {
        output.write(b);
      }
    }
  }

}
```

UseOutputStream2 – qualche variante

```java
import java.io.*;
import java.util.Random;

// Due varianti: uso di java.io.File, e catch con try-with-resources
public class UseOutputStream2 {

  public static void main(String[] args) {

    try (
      final OutputStream output =
        new FileOutputStream(new File(UseFile.FILE_NAME));
    ) {
      // Aggiungo byte random
      final Random r = new Random();
      for (int i = 0; i < 100; i++) {
        output.write(r.nextInt(256));
      }

      // Aggiungo un array di byte
      final byte[] b = new byte[] { 10, 20, 30, 40 };
      for (int i = 0; i < 10; i++) {
        output.write(b);
      }
    } catch (final IOException e){
      System.out.println("Something went wrong!");
    }
  }
}
```

Salvataggio di strutture dati: List<Byte>

```java
public class ListOnFile {

  public static void main(String[] args) throws IOException {
    final List<Byte> list = new ArrayList<>(20); // Creo una lista random
    final byte[] ar = new byte[20];
    new Random().nextBytes(ar);
    for (final byte b : ar) {
      list.add(b);
    }
    System.out.println("Prima: " + list);
    try (
      final OutputStream file = new FileOutputStream(UseFile.FILE_NAME)
    ) {
      for (final byte b : list) { // La riverso su file
        file.write(b);
      }
    }
    try (
      final InputStream file2 = new FileInputStream(UseFile.FILE_NAME)
    ) {
      final List<Byte> list2 = new ArrayList<>();
      int c;
      while ((c = file2.read()) != -1) { // Ricarico da file
        list2.add((byte) c);
      }
      System.out.println("Dopo: " + list2);
    }
  }
}
```

Solo `byte`?

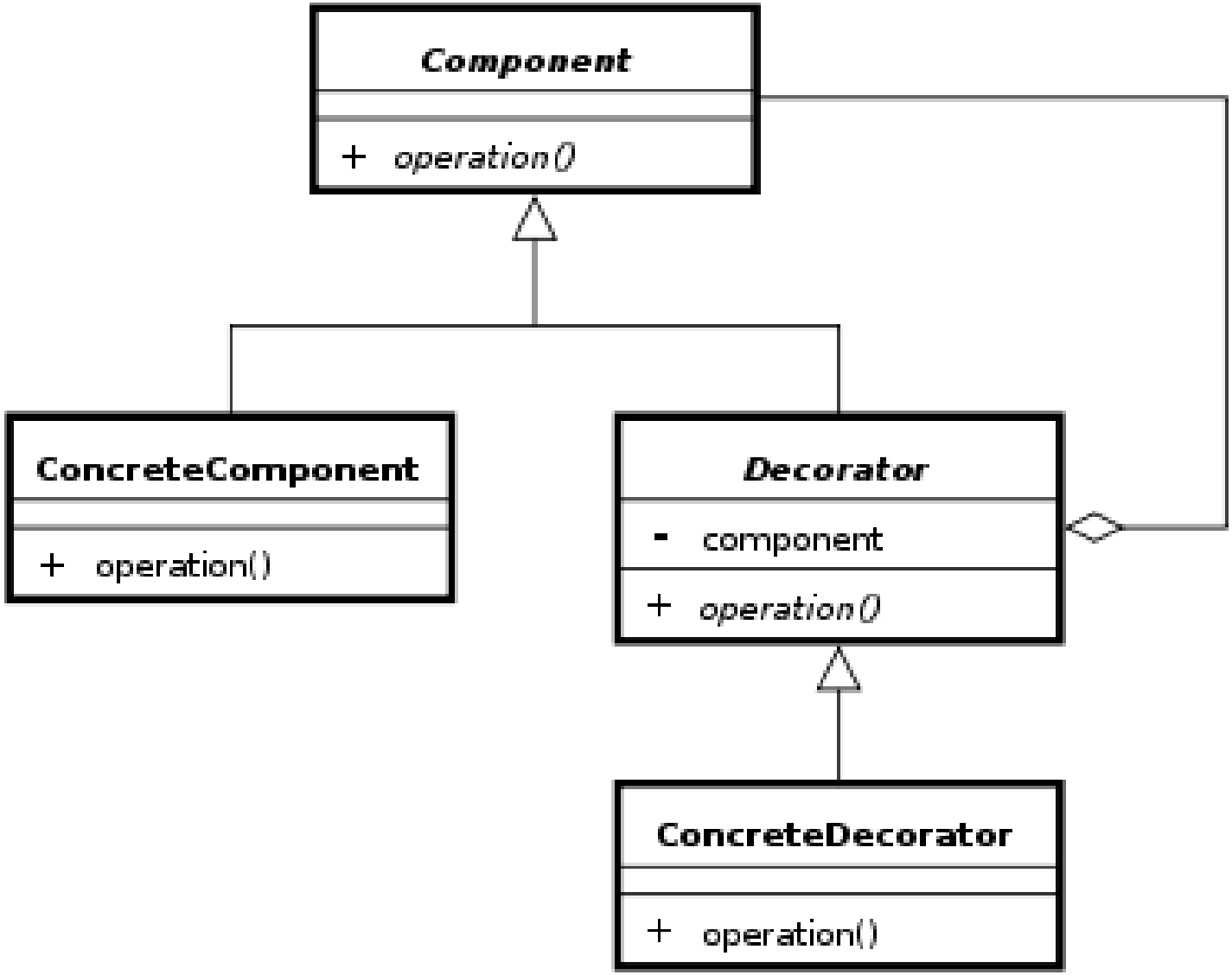
Problema...

- Poter leggere e scrivere da uno `Stream` anche `int`, `long`, eccetera

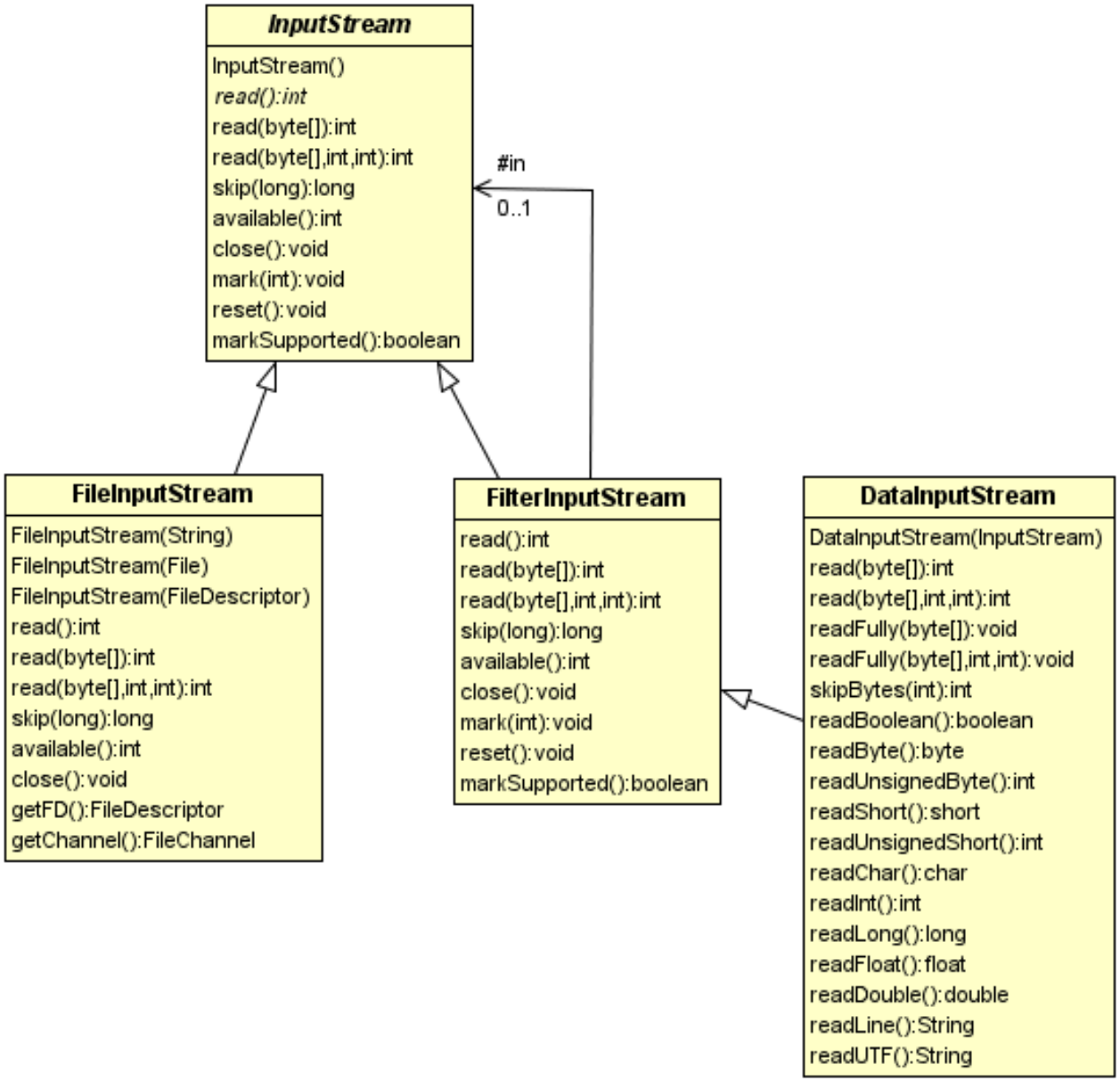
Il concetto di decoratore

- 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

```java
import java.io.*;

public class UseDataStream {

  public static void main(String[] args) throws IOException {

    try (
      final OutputStream file = new FileOutputStream(UseFile.FILE_NAME);
      final DataOutputStream dstream = new DataOutputStream(file);
    ) {
      dstream.writeBoolean(true);
      dstream.writeInt(10000);
      dstream.writeUTF("Ciao");
      dstream.writeDouble(5.2);
    }
    try (
      final InputStream file2 = new FileInputStream(UseFile.FILE_NAME);
      final DataInputStream dstream2 = new DataInputStream(file2);
    ){
      System.out.println(dstream2.readBoolean()); // Do not change order
      System.out.println(dstream2.readInt());
      System.out.println(dstream2.readUTF());
      System.out.println(dstream2.readDouble());
    }
  }
}
```

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

```java
import java.io.*;

public class UseBufferedDataStream {

    public static void main(String[] args) throws IOException {
    // dstream -> bstream -> file
    try (
      final OutputStream file = new FileOutputStream(UseFile.FILE_NAME);
      final OutputStream bstream = new BufferedOutputStream(file);
      final DataOutputStream dstream = new DataOutputStream(bstream);
    ){
      dstream.writeBoolean(true);
      dstream.writeInt(10000);
      dstream.writeDouble(5.2);
      dstream.writeUTF("Prova"); // Scrive in rappresentazione UTF-16
    }
    // dstream2 -> bstream2 -> file2
    try (
      final InputStream file2 = new FileInputStream(UseFile.FILE_NAME);
      final InputStream bstream2 = new BufferedInputStream(file2);
      final DataInputStream dstream2 = new DataInputStream(bstream2);
    ){
      System.out.println(dstream2.readBoolean()); // Do not change order!!
      System.out.println(dstream2.readInt());
      System.out.println(dstream2.readDouble());
      System.out.println(dstream2.readUTF());
    }
  }
```

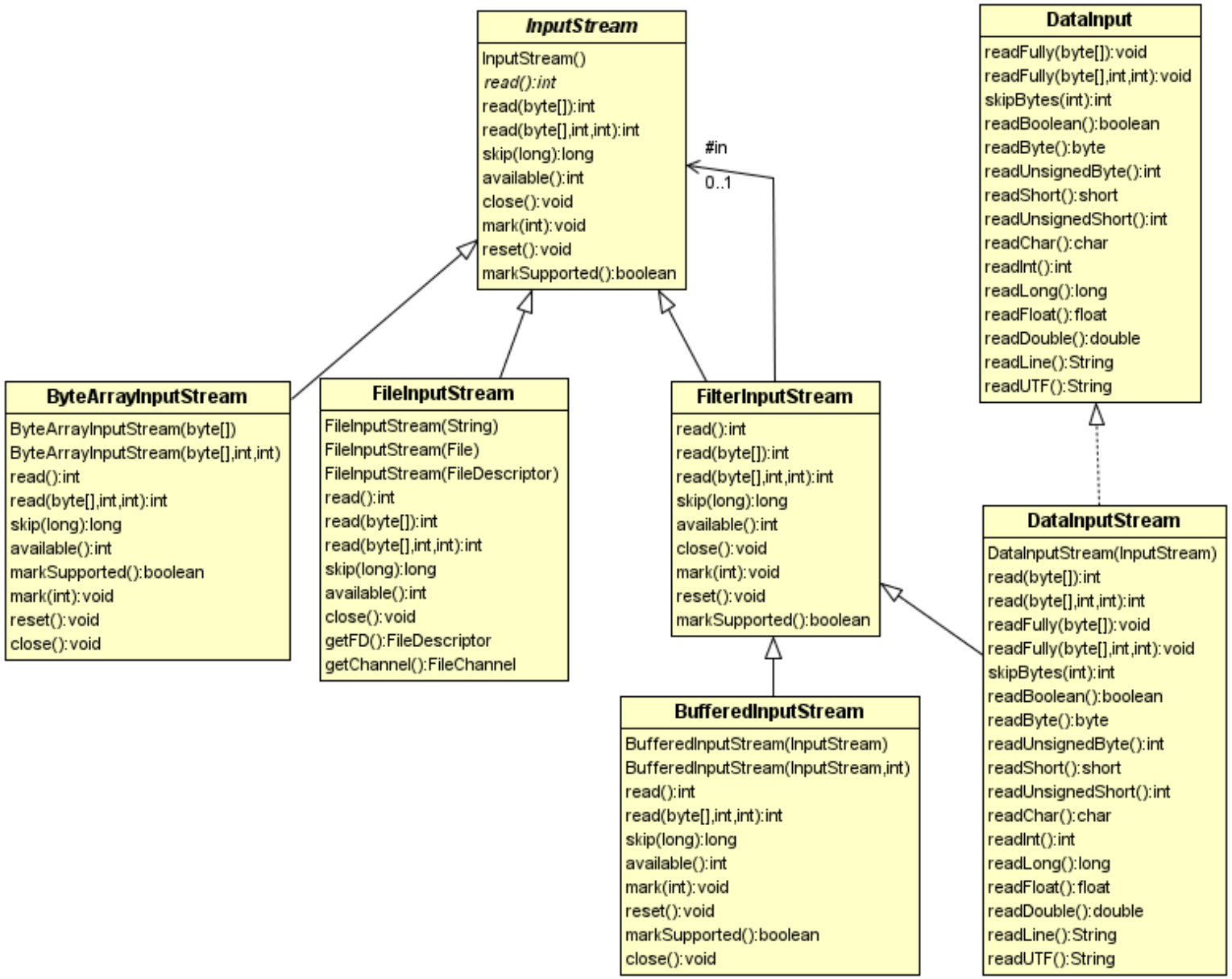
UseBufferedDataStream2 – chaining dei costruttori

```java
import java.io.*;

public class UseBufferedDataStream2 {

    public static void main(String[] args) throws IOException {
    // dstream -> bstream -> file
    try (
      final DataOutputStream dstream = new DataOutputStream(
        new BufferedOutputStream(
          new FileOutputStream(UseFile.FILE_NAME)));
    ){
      dstream.writeBoolean(true);
      dstream.writeInt(10000);
      dstream.writeDouble(5.2);
      dstream.writeUTF("Prova"); // Scrive in rappresentazione UTF-16
    }
    // dstream2 -> bstream2 -> file2
    try (
      final DataInputStream dstream2 = new DataInputStream(
        new BufferedInputStream(
          new FileInputStream(UseFile.FILE_NAME)));
    ){
      System.out.println(dstream2.readBoolean()); // Do not change order!!
      System.out.println(dstream2.readInt());
      System.out.println(dstream2.readDouble());
      System.out.println(dstream2.readUTF());
    }
  }
```

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

File ed encoding

Il contenuto dei file

Supponiamo utilizziate un `DataOutputStream` per scrivere in sequenza i numeri da 0 a 20 (escluso)

- Che cosa avete realmente scritto?
- avete scritto i byte da zero a 19
- In esadecimale, `0x000102030405060708090A0B0C0D0E0F10111213`
- **Non** avete scritto il testo `0123456789101112131415161718l9`

- I file sono sequenze di byte
- Per fare input/output occorre stabilire:
  - Una conversione dalla struttura dati che stiamo manipolando a sequenza di byte (encoding)
  - Una conversione da sequenza di byte a struttura dati (decoding)
- I `Data-Stream` offrono encoding e decoding per tipi primitivi
- Come trattare strutture dati più articolate?

Serializzazione di oggetti

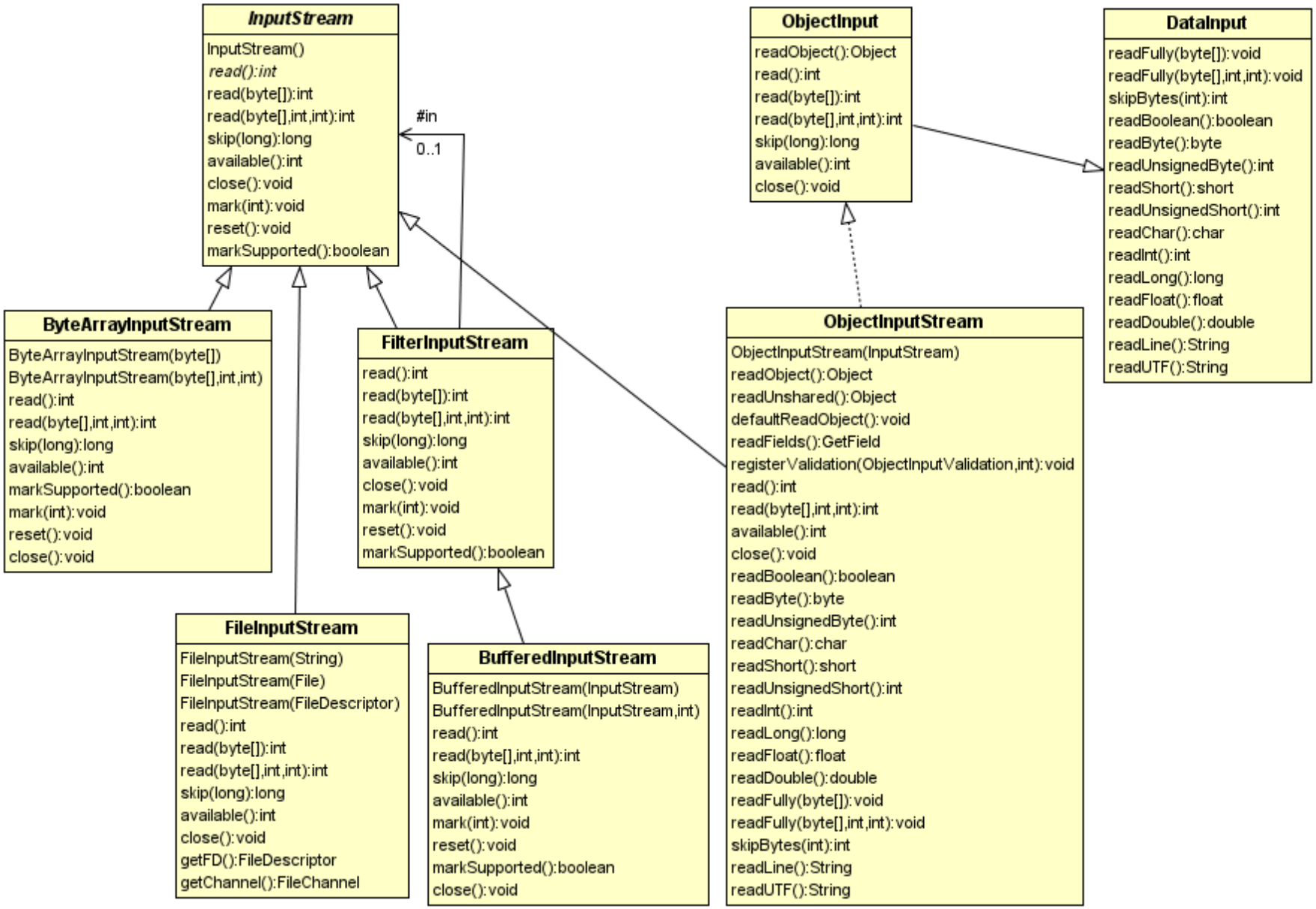
Motivazioni

- 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 Person implements java.io.Serializable {
2
3    private static final long serialVersionUID = 5677425026232659 45L;
4    private final String name;
5    private final int birthYear;
6    private final boolean married;
7
8    public Person(
9        final String name,
10       final int birthYear,
11       final boolean married
12   ) {
13     this.name = name;
14     this.birthYear = birthYear;
15     this.married = married;
16   }
17
18   public String toString() {
19     return this.name + ":" + this.birthYear + ":"
20         + (this.married ? "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

```java
public class UseObjectStream {

  public static void main(String[] args) throws Exception {

    // ostream -> bstream -> file
    try (
      final OutputStream file = new FileOutputStream(UseFile.FILE_NAME);
      final OutputStream bstream = new BufferedOutputStream(file);
      final ObjectOutputStream ostream = new ObjectOutputStream(bstream);
    ){
      ostream.writeInt(10000);
      ostream.writeDouble(5.2);
      ostream.writeObject(new java.util.Date()); // Classe serializ.
      ostream.writeObject(new Person("Rossi", 1960, false));
    }

    // ostream2 -> bstream2 -> file2
    try (
      final InputStream file2 = new FileInputStream(UseFile.FILE_NAME);
      final InputStream bstream2 = new BufferedInputStream(file2);
      final ObjectInputStream ostream2 = new ObjectInputStream(bstream2);
    ){
      System.out.println(ostream2.readInt());
      System.out.println(ostream2.readDouble());
      System.out.println(ostream2.readObject()); // carica il Date
      System.out.println(ostream2.readObject()); // carica la Persona
    }
  }
```

Classe `UseObjectStream`: 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

CPerson con caching toString, pt1

```
1  public class CPerson implements java.io.Serializable {
2
3    // Eclipse would ask to implement a serialVersionUID
4    //private static final long serialVersionUID = -898502638052662081 2L;
5
6    private String name;
7    private int birthYear;
8    private boolean married;
9    transient private String cachedToString = null;
10
11   public CPerson(String name, int birthYear, boolean married) {
12     this.name = name;
13     this.birthYear = birthYear;
14     this.married = married;
15   }
16
17   public String getName() {
18     return this.name;
19   }
20
21   public int getBirthYear() {
22     return this.birthYear;
23   }
24
25   public boolean isMarried() {
26     return this.married;
27   }
```

CPerson con caching toString, pt2

```
1   public void setName(final String name) {
2     this.name = name;
3     this.cachedToString = null;
4   }
5
6   public void setBirthYear(final int birthYear) {
7     this.birthYear = birthYear;
8     this.cachedToString = null;
9   }
10
11  public void setMarried(final boolean married) {
12    this.married = married;
13    this.cachedToString = null;
14  }
15
16  private String computeToString() {
17    return this.name + ":" + this.birthYear + ":"
18        + (this.married ? "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
2  public class UseTransient {
3
4    public static void main(String[] args) throws Exception {
5
6      try (final ObjectOutputStream out = new ObjectOutputStream(
7          new FileOutputStream(UseFile.FILE_NAME))){
8        final CPerson p = new CPerson("Rossi", 1960, false);
9        System.out.println("Prima stampa " + p); // cache vuota
10       System.out.println("Seconda stampa " + p); // cache non vuota
11       out.writeObject(new CPerson("Rossi", 1960, false));
12     }
13
14     System.out.println("Ri-carico l'oggetto... ");
15
16     try (final ObjectInputStream in = new ObjectInputStream(
17         new FileInputStream(UseFile.FILE_NAME))){
18       final CPerson q = (CPerson) in.readObject(); //attenzione al cast!
19       System.out.println("Prima stampa " + q); // cache vuota
20       System.out.println("Seconda stampa " + q); // cache non vuota
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

APerson: serializzazione ad-hoc per una data

```java
public class APerson implements java.io.Serializable {
  private static final long serialVersionUID = -8985026380526620812L;

  private final String name;
  private transient Date lastUse = new Date();

  public APerson(final String name) {
    this.name = name;
  }

  public void used() {
    this.lastUse = new Date();
  }

  public String toString() {
    return this.name + ":"
        + (this.lastUse == null ? "null" : this.lastUse.getTime());
  }

  private void writeObject(final ObjectOutputStream out) throws IOException {
    out.defaultWriteObject(); // accetto il comportamento di default
    System.err.println("writing");
  }

  // una sorta di costruttore..
  private void readObject(final ObjectInputStream in) throws IOException,
      ClassNotFoundException {
    in.defaultReadObject();
    System.err.println("reading");
    this.lastUse = new Date(); // in lettura, ripristino la data corrente
  }
```

UseAdHocSerialization

```
1  import it.unibo.apice.oop.p16io.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 APerson p = new APerson("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 APerson q = (APerson) 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 }
```

Note sulla serializzazione Java

Raramente utilizzata in applicazioni "vere"

Applicazioni vere tendono a non utilizzare `Object(In/Out)putStream`:

- Nessuna standardizzazione
- Poco efficiente in termini di spazio
- Dispendiosa in termini di performance
- Scarsa portabilità (solo da/a software Java)
- `readObject` / `writeObject` in qualche modo violano il linguaggio

Importanza di `Serializable`

Approcci diversi alla serializzazione spesso:

- Serializzano oggetti `Serializable` con campi `Serializable` e non `transient` (deve valere ricorsivamente)
- Prevedono sistemi custom per la serializzazione di oggetti diversi
- Non fanno uso di `readObject` / `writeObject`

Outline

Classe `RandomAccessFile`

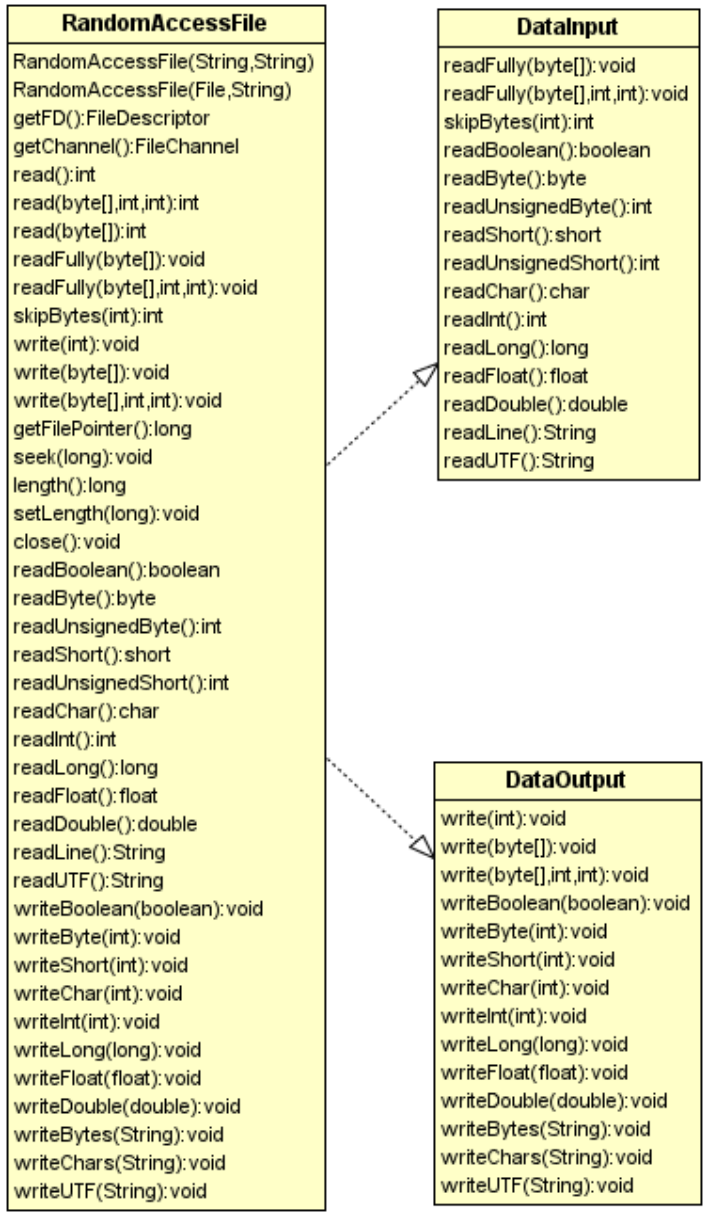
Motivazioni

- 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

Limitazione degli stream binari visti finora

Finora abbiamo visto stream binari, ed in particolare di oggetti Java

- Non esiste uno standard documentato
  - Gli standard descrivono in modo inequivoco un protocollo (in questo caso di de/serializzazione)
  - Difficile scrivere librerie per leggere e scrivere in quel formato
  - Bassa **standardizzazione**
- Non comprensibili da applicazioni non-Java
  - Oggetti serializzati non apribili da applicazioni in Javascript, Python, eccetera
  - Non adatti e.g., per applicazioni web
  - Bassa **portabilità**
- Non comprensibili se aperti in formato testuale
  - Non modificabili da umani
  - Non adatti e.g., per file di configurazione
  - Bassa **leggibilità** e quindi **intelligibilità**

Limitazione degli stream binari visti

Standardizzazione e portabilità

- Non direttamente ascrivibili al formato binario
- (che comunque non aiuta...)
- Esistono standard portabili binari, come ProtocolBuffers

Leggibilità e intelligibilità

- Problema comune a tutti i meccanismi finora visti
- Esistono standard per scrivere oggetti in formato testuale
- Problema da risolvere in due fasi:
  - Conversione da oggetto a stringa di testo e viceversa
  - Conversione da stringa di testo a file e viceversa

Conversione da oggetto a stringa di testo e viceversa

- Banale nel caso in cui l'oggetto sia una stringa
- Molto più complicato per strutture dati arbitrarie
- Passo necessario verso l'interoperabilità fra linguaggi

(Alcuni) Formati standard per la conversione di oggetti in testo

- JavaScript Object Notation – JSON (RFC 7149)
  - Nato in seno a JavaScript (che non c'entra nulla con Java)
  - Molto usato nel web
- Tom's Obvious, Minimal Language – TOML
  - Particolarmente indicato per file di configurazione
- YAML Ain't Markup Language – YAML
  - Superset di JSON dalla versione 1.2
  - Supporto per funzioni avanzate (e.g. anchoring)
  - Molto usato per file di configurazione complessi

Conversione stringa a file di testo

Problema risolto con tabelle di conversione (text encoding)
Nota: importante anche per la rappr. **in memoria** dei caratteri

Text Encoding

- ASCII (RFC 20)
  - l'encoding che usa il linguaggio C per i `char`
  - 1 byte per carattere (massimo 256 caratteri)
- UTF-8 (RFC 3629)
  - Standard di fatto sul web, encoding di default in Linux
  - encoding da usare per i sorgenti di codice
  - da 1 a 4 byte per carattere
  - Codifica 1.112.064 simboli
- UTF-16 (RFC 2781)
  - Encoding in memoria delle `String` in Java
  - 2 o 4 byte per carattere
  - Codifica 1.112.064 simboli
- ISO Latin (ISO/IEC 8859-1:1998)
  - Encoding di default del testo in Windows

File di testo

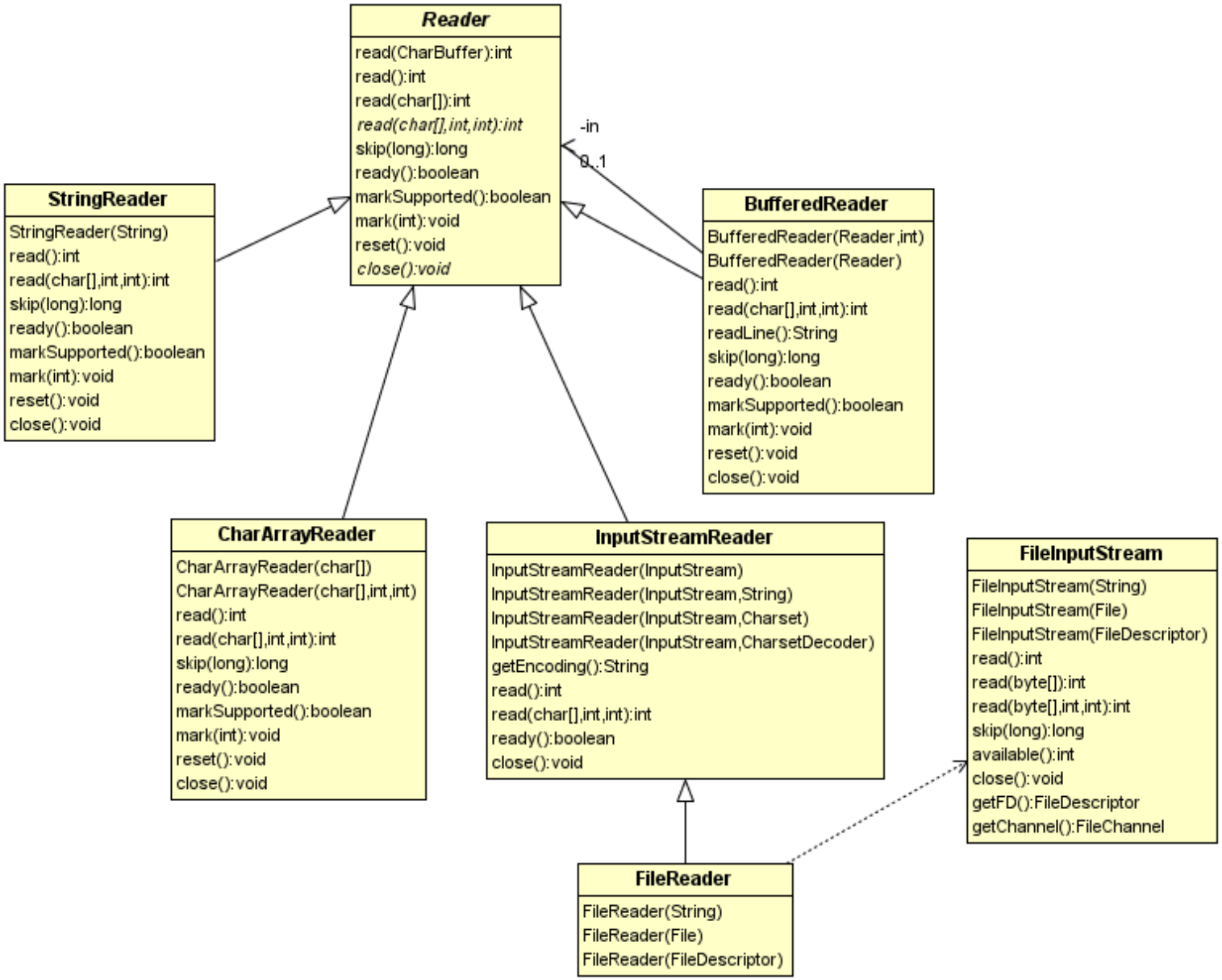
File binari vs file di testo

- 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

```java
public class UseReadersWriters {

  public static final String FILE_NAME = "/home/mirko/aula/oop/15/a.txt";

  public static void main(String[] args) throws Exception {
    try (
      final BufferedWriter w = new BufferedWriter(new FileWriter(FILE_NAME))
    ) {
      w.write("Prova");
      w.newLine();
      w.write("di file");
      w.newLine();
    }
    try (
      final BufferedReader r = new BufferedReader(new FileReader(FILE_NAME))
    ) {
      System.out.println(r.readLine());
      System.out.println(r.readLine());
      System.out.println(r.readLine()); // null, indica la fine del file!
    }
    try (
      final BufferedReader r = new BufferedReader(new FileReader(FILE_NAME))
    ) {
        String line = null;
        while( (line = r.readLine()) != null){
            System.out.println(line);
        }
    }
```

Esempio: UseStreamReadersWriters

```java
// Soluzione con specifica della codifica dei caratteri
public class UseStreamReadersWriters {

  public static void main(String[] args) throws Exception{

    try(final BufferedWriter w = new BufferedWriter(
        new OutputStreamWriter(
          new FileOutputStream(UseReadersWriters.FILE_NAME),"UTF-16"))){
      w.write("Prova");
      w.newLine();
      w.write("di file");
      w.newLine();
    }

    try(final BufferedReader r = new BufferedReader(
        new InputStreamReader(
          new FileInputStream(UseReadersWriters.FILE_NAME),"UTF-16"))){
      System.out.println(r.readLine());
      System.out.println(r.readLine());
      System.out.println(r.readLine()); // null
    }
  }
}
```

Il caso di: System.in e System.out

```java
public class SystemInOut {

  public static void main(String[] args) throws Exception{

    final InputStream is = System.in;
    final BufferedReader br = new BufferedReader(new InputStreamReader(is));
    System.out.println(br.readLine()); // può lanciare una IOException

    final PrintStream ps = System.out;
    // una sotto-classe di OutputStream, che incapsula un Writer
    ps.println("Un comando noto"); // non lancia eccezioni!
    ps.format("Altro comando noto.. %d %f %s\n", 10,20.2,"prova");

    // scrittura di file con PrintStream
    final PrintStream ps2 = new PrintStream(UseReadersWriters.FILE_NAME);
    ps2.println("prova");
    ps2.print(10);
    ps2.println();
    ps2.close();
  }
}
```

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`