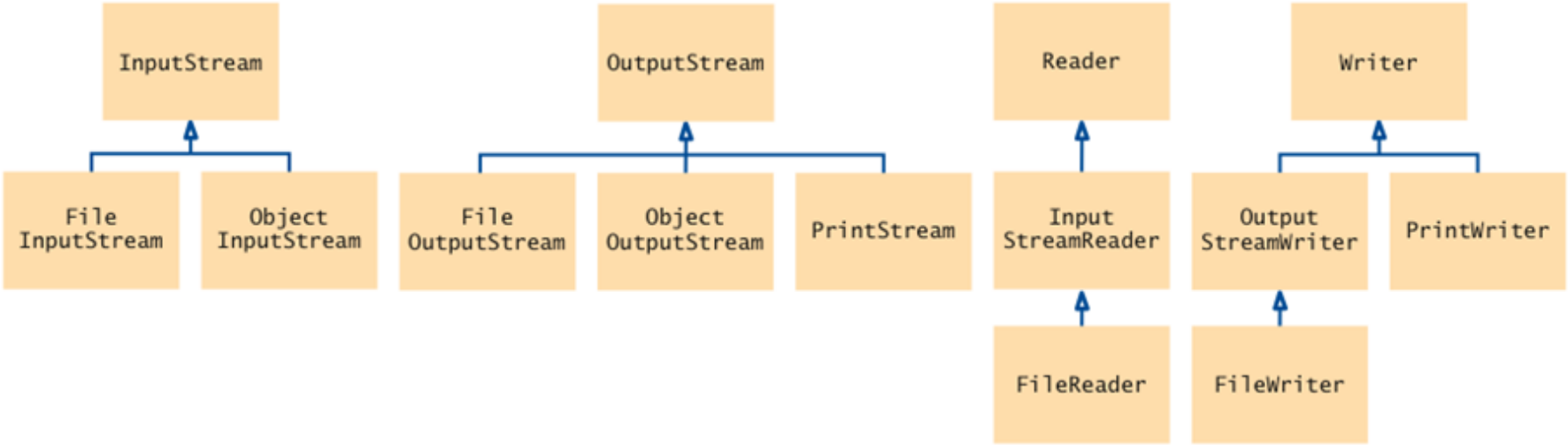
Il package `java.io` e i flussi

Il package `java.io`

- In Java input e output sono definiti in termini di flussi (stream)
  - Sequenze ordinate di dati
- Due tipi di flussi
  - Di dati binari (byte stream)
  - Di caratteri (character stream)
- Ciascun tipo di flusso è gestito da apposite classi



- Per dati binari, usare la classe **`InputStream`**
- Per caratteri, usare la classe **`Reader`**

- Flussi di output: hanno una destinazione
  - Per dati binari, usare la classe **`OutputStream`**
  - Per caratteri, usare la classe **`Writer`**
- Tutte queste classi sono nel package **`java.io`**
  - **`import java.io.*;`**

La classe `IOException`

- Utilizzata da molti metodi di **`java.io`** per segnalare condizioni di errore
- Un metodo solleva una **`IOException`** se si verifica un problema collegato al flusso di I/O
- Costruttori che ricevono come parametro il nome di un file/directory o un oggetto File possono lanciare una **`FileNotFoundException`** (sottoclasse di **`IOException`**)

Flussi Standard

Definiti dalla classe **`System`** in **`java.lang`**

- Standard input (tastiera): **`System.in`**
  - Di tipo **`InputStream`**
- Standard output (monitor): **`System.out`**
  - Di tipo **`PrintStream`**
- Standard error (per messaggi di errore): **`System.err`**
  - Di tipo **`PrintStream`**

La classe astratta InputStream

- Dichiara i metodi per leggere flussi binari da una sorgente specifica
- Alcuni metodi:
  - `public abstract int read() throws IOException`
  - `public void close() throws IOException`

Note su InputStream

- **`read`**
  - Legge un **`byte`** alla volta
  - Restituisce
    - un **`int`** (da 0 a 255) che rappresenta il byte letto
    - -1 se il file è terminato
- **`close`**
  - Chiude il flusso di input
  - Rilascia le risorse associate al flusso
  - Ulteriori operazioni sul flusso chiuso provocano una **`IOException`**
- E' importante chiudere il flusso d'input con il metodo **`close()`**

La classe FileInputStream

- Sottoclasse concreta di **`InputStream`**
  - **`public class FileInputStream extends InputStream`**

- Possiamo creare oggetti di questa classe
  - **`FileInputStream in = new FileInputStream("nomefile.bin");`**

Esempio: Contare i byte in un flusso

```
import java.io.*;

public class ContaByte {
   public static void main(String[] args) throws
        IOException {
      InputStream in = new
                  FileInputStream("nomefile.bin");
      int totale = 0;
      while (in.read() != -1)
           totale++;
      in.close();
      System.out.println("Il numero di byte è" + totale);
   }
}
```

Specificare il path di un file

- Quando si digita il path di un file ogni barra rovesciata va inserita due volte
  - Una singola barra rovesciata è un carattere di escape

```
InputStream in = new
    FileInputStream("C:\\nomedir\\nomefile.est");
```

La classe astratta OutputStream

- Dichiara i metodi per scrivere flussi binari in una destinazione specifica

- Alcuni metodi:
  - `public abstract void write(int b) throws IOException`
  - `public void close() throws IOException`

Note su OutputStream

- **`write`**
  - Scrive un byte alla volta
  - il byte è passato come argomento di tipo int
- **`close`**
  - Chiude il flusso di output
  - Rilascia le risorse associate al flusso
  - Ulteriori operazioni sul flusso chiuso provocano una **`IOException`**
- E' importante chiudere il flusso d'output con il metodo **`close()`**
  - chiusura garantisce scrittura

Classe PrintStream

- `PrintStream` è una classe concreta nella discendenza (sottoclasse) di `OutputStream`
- Aggiunge a `OutputStream` tutti i metodi per stampare convenientemente vari tipi di dati
  - Ad es. i metodi `print` e `println`
- Oggetto `System.out` è di tipo `PrintStream` e rappresenta il flusso standard di output (flusso binario)

La classe FileOutputStream

- Sottoclasse concreta di **`OutputStream`**
  - **`public class FileOutputStream extends OutputStream`**
- Possiamo creare oggetti di questa classe
  - **`FileOutputStream out = new FileOutputStream("nomefile.est");`**
- C'è anche un costruttore

`public FileOutputStream(File file, boolean append)`

La classe astratta **`Reader`**

- Dichiara i metodi per leggere flussi di caratteri da una sorgente specifica
- Alcuni metodi:
  - **`public int read() throws IOException`**
  - **`public abstract void close()`**
    `throws IOException`

Note su `Reader`

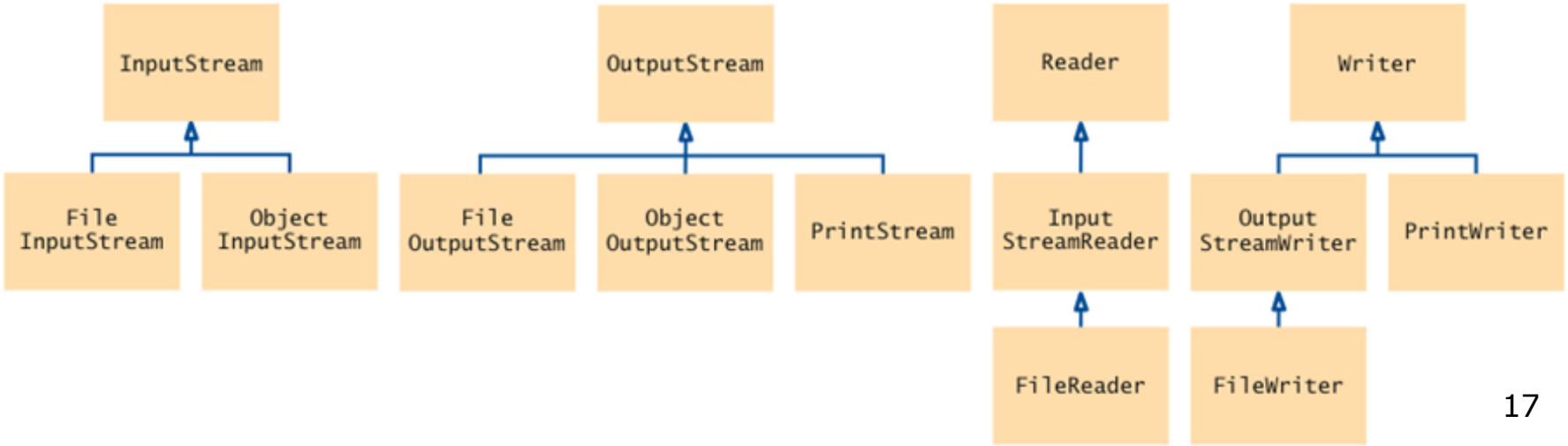
- **`read`**
  - Legge un carattere alla volta
  - Restituisce
    - un **`int`** (da 0 a 65535) che rappresenta il carattere letto
    - -1 se il file è terminato
- **`close`**
  - Chiude il flusso di caratteri
  - Rilascia le risorse associate al flusso
  - Ulteriori operazioni sul flusso chiuso provocano una **`IOException`**
- E' importante chiudere il flusso d'input con il metodo **`close()`**

Concretizzare `Reader`

- Dobbiamo convertire un flusso di input binario in un flusso di input di caratteri

- Si usa la classe **`InputStreamReader`**
  - E' una sottoclasse concreta di **`Reader`**
  - Il costruttore è

    `public InputStreamReader(InputStream in)`



Conversione tra flussi

- L'oggetto `System.in` è di tipo `InputStream`, possiamo convertirlo in un flusso di caratteri

```
InputStreamReader reader =
            new InputStreamReader(System.in);
```

- In generale:

```
InputStream in =
            new FileInputStream("nomefile.bin");
InputStreamReader reader =
            new InputStreamReader(in);
```

La classe `FileReader`

- Sottoclasse di **`InputStreamReader`**
  - `public class FileReader`
    `extends InputStreamReader`
- Costruttore richiede nome file
  - `FileReader reader =`
    `new FileReader("nomefile.txt");`
- Serve per leggere flussi di caratteri da un file
  - `char c = reader.read();`

Esempio: Contare i caratteri in un flusso

```
import java.io.*;

public class ContaCaratteri {
   public static void main(String[] args)
                                        throws IOException {
   Reader reader = new FileReader("nomefile.txt");
   int totale = 0;
   while (reader.read() != -1)
         totale++;
   reader.close();
   System.out.println("Il numero di caratteri è" +
                                                  totale);
   }
}
```

Usare un `FileReader`

- Gli oggetti della classe **`FileReader`** leggono un carattere per volta, ma spesso serve leggere intere linee.
- Si può pensare di usare un oggetto che compone stringhe a partire dai caratteri letti da un **`FileReader`**
- Si può usare la classe **`Scanner`**

```
FileReader reader = new FileReader("file.txt");
Scanner in = new Scanner(reader);
String inputLine = in.nextLine(); //lettura dati
```

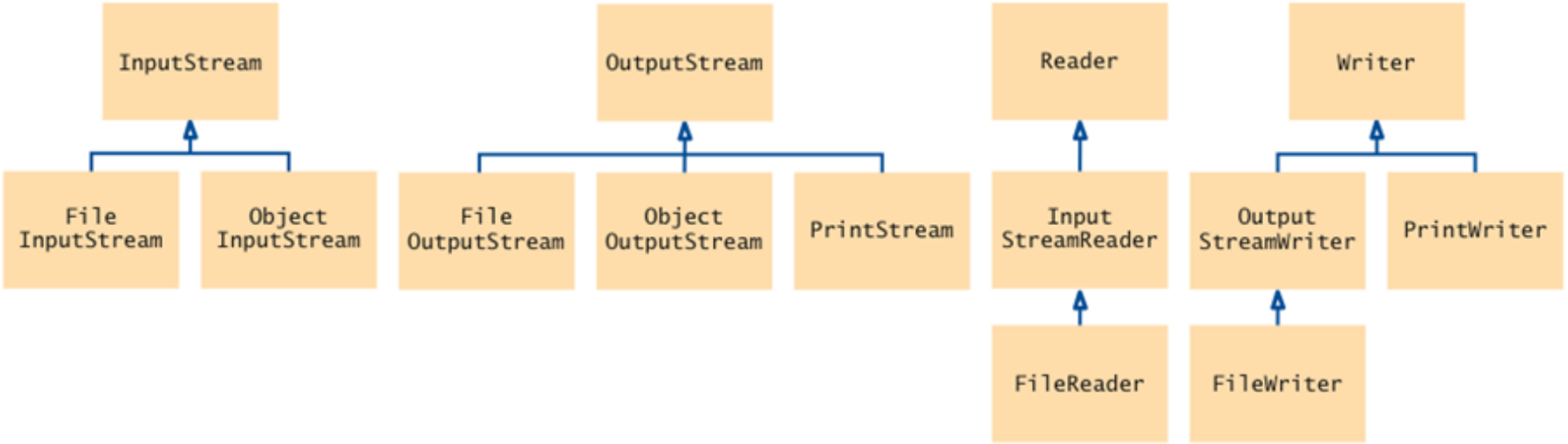
La classe astratta `Writer`

- Dichiara i metodi per scrivere flussi di caratteri verso una destinazione specifica

- Alcuni metodi:
  - `public void write(int c)`
    `throws IOException`
  - `public abstract void close()`
    `throws IOException`

Note su `Writer`

- **`write`**
  - Scrive un carattere alla volta
  - il carattere è passato come argomento di tipo int
- **`close`**
  - Chiude il flusso di output
  - Rilascia le risorse associate al flusso
  - Ulteriori operazioni sul flusso chiuso provocano una **`IOException`**
- E' importante chiudere il flusso d'output con il metodo **`close()`**
  - chiusura garantisce scrittura



- `public PrintWriter("output.txt")`

- Contiene tutti i metodi **`print`** e **`println`** di **`PrintStream`**
  - `PrintWriter out = new PrintWriter("output.txt");`
  - `out.println(2.75);`
  - `out.println(new BankAccount());`
  - `out.println("Hello, World!");`
- Quando si istanzia un oggetto **`PrintWriter`**:
  - se il file passato come parametro del costruttore esiste, allora viene svuotato del suo contenuto
  - se il file non esiste viene creato un file nuovo (vuoto) con il nome passato come parametro del costruttore

Esempio

- Scrivere un programma che legge tutte le righe di un file e le scrive in un altro file facendo precedere ogni riga dal suo numero

- File di input:

```
Mary had a little lamb
Whose fleece was white as snow.
And everywhere that Mary went,
The lamb was sure to go!
```

- File di output desiderato:

```
/* 1 */ Mary had a little lamb
/* 2 */ Whose fleece was white as snow.
/* 3 */ And everywhere that Mary went,
/* 4 */ The lamb was sure to go!
```

File LineNumberer.java

```
01: import java.io.FileReader;
02: import java.io.IOException;
03: import java.io.PrintWriter;
04: import java.util.Scanner;
05:
06: public class LineNumberer
07: {
08:    public static void main(String[] args)
09:    {
10:       Scanner console = new Scanner(System.in);
11:       System.out.print("Input file: ");
12:       String inputFileName = console.next();
13:       System.out.print("Output file: ");
14:       String outputFileName = console.next();
15:
16:       try
17:       {
```

File LineNumberer.java

```
18:          FileReader reader = new FileReader(inputFileName);
19:          Scanner in = new Scanner(reader);
20:          PrintWriter out = new PrintWriter(outputFileName);
21:          int lineNumber = 1;
22:
23:          while (in.hasNextLine())
24:          {
25:             String line = in.nextLine();
26:             out.println("/* " + lineNumber + " */ " + line);
27:             lineNumber++;
28:          }
29:
30:          out.close();
31:       }
32:       catch (IOException exception)
33:       {
34:          System.out.println("Error processing file:"
                + exception);
35:       }
36:    }
37: }
```

La classe `File`

- Rappresentazione astratta di un file
- Può essere utilizzato per manipolare file esistenti
- Creiamo un oggetto di tipo **`File`**

```
File inputFile = new File("input.txt");
```

```
(input.txt può non esistere, e questo comando non lo crea)
```

- Non possiamo leggere/scrivere direttamente dati da un oggetto di tipo **`File`**
- Dobbiamo costruire un oggetto di tipo **`FileReader`** o **`PrintWriter`**

```
FileReader reader = new FileReader(inputFile);
PrintWriter writer = new PrintWriter(inputFile);
```

La classe File: alcuni metodi

- `public boolean delete()`
  - Cancella il file restituendo true se la cancellazione ha successo
- `public boolean renameTo(File newname)`
  - Rinomina il file restituendo true se la ridenominazione ha successo
- `public long length()`
  - Restituisce la lunghezza del file in byte (zero se il file non esiste)
- `public boolean exists()`
  - Testa se il file o la directory denotata da questo File esiste

Ricapitoliamo con un esempio

```
import java.io.*;
public class Esempio {
 public static void main(String[] args)
                                        throws IOException {
        // SCRITTURA
   PrintWriter pw = new PrintWriter("C:\\HelloWorld.txt");
   pw.println("HELLO WORLD alla fine del file");
        // Chiusura File
   pw.close();
        // LETTURA
   FileReader fr = new FileReader("C:\\HelloWorld.txt");
   Scanner sc = new Scanner(fr);
   String s = sc.nextLine();
        // Chiusura File
   fr.close();
   System.out.println(s);
```

Esempio

```
  File f1 = new File("C:\\HelloWorld.txt");
  File f2 = new File("C:\\HelloWorld2.txt");
  f1.renameTo(f2);
       // Attenzione NON crea il file
  File f3 = new File("C:\\HelloWorld3.txt");
       // Per crearlo dovete darlo ad un Writer
  PrintWriter fw2 = new PrintWriter(f3);
 } // Fine main
} // Fine classe
```

Leggere pagine web

- Esiste una classe in Java per gestire gli indirizzi delle pagine Web, si chiama **`URL`**
- Fornisce un metodo **`openStream`** che restituisce un oggetto **`InputStream`**:

```
URL locator = new
   URL("http://bigjava.com/index.html");

InputStream in = locator.openStream();
```

- Se si vuole leggere caratteri possiamo:

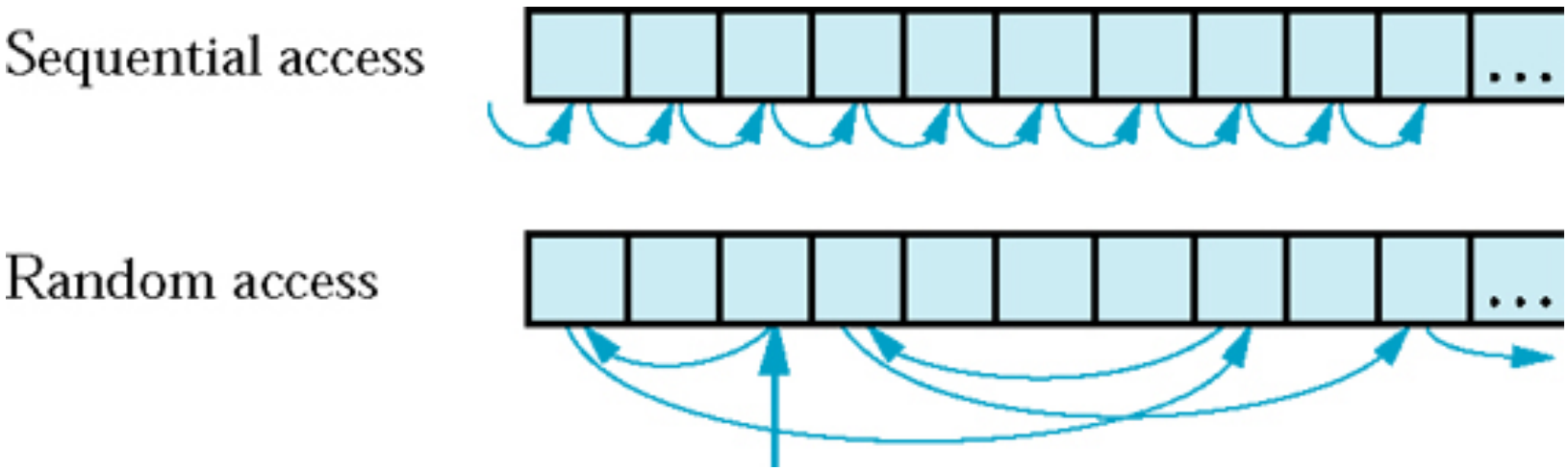
```
Scanner in = new
   Scanner(locator.openStream());
```

Accesso sequenziale e casuale

- Accesso sequenziale
  - Un file viene elaborato un byte alla volta, in sequenza
  - può essere inefficiente

- Accesso casuale
  - Possiamo accedere a posizioni arbitrarie nel file
  - Soltanto i file su disco supportano l'accesso casuale: `System.in` e `System.out` no
  - Ogni file su disco ha un puntatore di file che individua la posizione dove leggere o scrivere.

Accesso sequenziale e casuale



Accesso casuale

- Per l'accesso casuale al file, usiamo un oggetto di tipo **`RandomAccessFile`**
- Possiamo aprire il file in diverse modalità:
  - "**`r`**" apre il file in sola lettura; se viene usato un metodo di scrittura viene invocata una **`IOException`**
  - "**`rw`**" apre il file per lettura e scrittura. Se il file non esiste prova a crearlo.

- Es.:

```
RandomAccessFile f =
        new RandomAccessFile("bank.dat","rw");
```

Accesso casuale: metodi

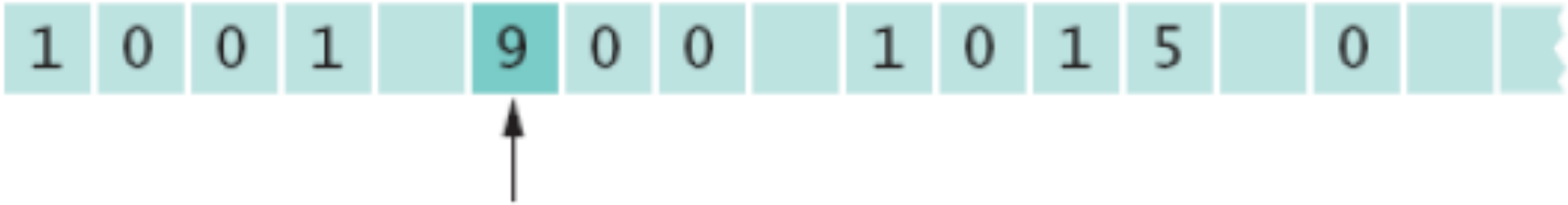
- **`f.read()`**
  - come **`read`** di **`InputStream`**, astratto, un byte alla volta
  - **`readLine()`**, **`readInt()`**, **`readDouble()`**, …
- **`f.write(b)`**
  - scrive il byte **`b`** a partire dalla posizione indicata dal puntatore
  - **`writeChars(String)`**, **`writeDouble(double)`**, **`writeInt(int)`**, …
- **`f.close()`** //chiude il file
- **`f.seek(n)`** //sposta il puntatore al byte di indice n
- **`int n = f.getFilePointer();`**
  - Fornisce la posizione corrente del puntatore nel file
- **`long fileLength = f.length();`**
  - Fornisce il numero di byte di un file

Esempio

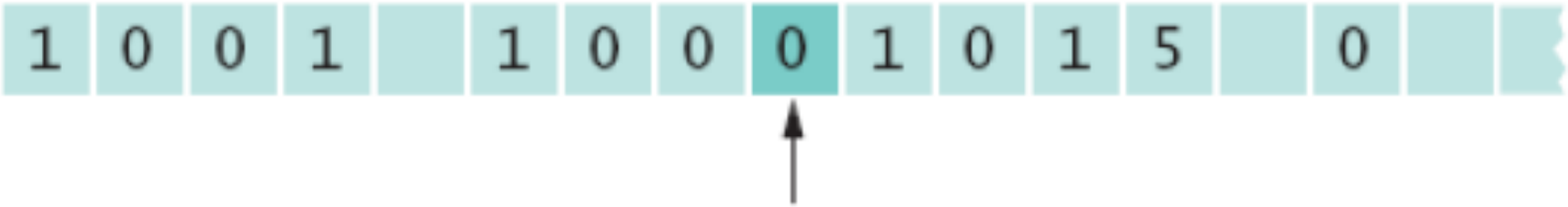
- Si vuole usare un **`RandomAccessFile`** per mantenere un insieme di oggetti **`BankAccount`**
- Il programma deve permettere di selezionare un conto e di effettuare un versamento
- Per manipolare un insieme di dati in un file occorre prestare attenzione a come i dati sono formattati
  - Supponiamo che memorizziamo un conto come un testo (String), ad esempio: conto 1001 ha saldo 900 e conto 1015 ha saldo 0

Esempio

- Vogliamo versare 100 nel conto 1001



- Se semplicemente scriviamo il nuovo valore si ha



Soluzione

- Per aggiornare un file:
  - Ogni valore deve avere uno spazio fissato sufficientemente grande
  - Ogni record ha la stessa taglia
    - E' facile individuare ogni record
  - Quando tutti i record hanno la stessa taglia è più facile memorizzare i numeri in binario

Note su RandomAccessFile

- **`RandomAccessFile`** memorizza i dati in binario
- **`readInt`** legge interi come sequenze di 4 bytes
- **`writeInt`** scrive interi come sequenze di 4 bytes
- **`readDouble`** e **`writeDouble`** usano 8 bytes

```
double x = f.readDouble();
f.writeDouble(x);
```

Esempio

- Determinare il numero di conti nel file

```
public int size() throws IOException
{
    return (int) (file.length() / RECORD_SIZE);
    // RECORD_SIZE is 12 bytes:
    // 4 bytes for the account number and
    // 8 bytes for the balance }
```

- Leggere l'n-esimo conto nel file

```
public BankAccount read(int n) throws IOException
{
   file.seek(n * RECORD_SIZE);
   int accountNumber = file.readInt();
   double balance = file.readDouble();
   return new BankAccount(accountNumber, balance);
}
```

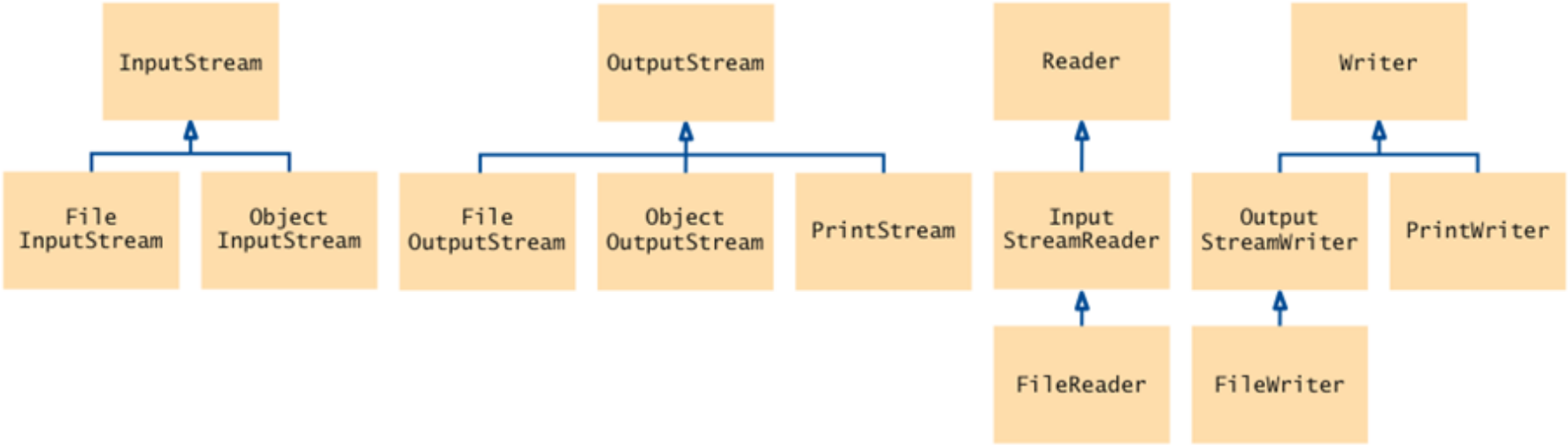
Esempio

- Scrivere nell'n-esimo conto del file

```
public void write(int n, BankAccount account)
                                    throws IOException {
   file.seek(n * RECORD_SIZE);
   file.writeInt(account.getAccountNumber());
   file.writeDouble(account.getBalance());
}
```

- Cerca l'indice di un conto nel file

```
public int find(int accountNumber) throws IOException {
   for (int i = 0; i < size(); i++){
     file.seek(i * RECORD_SIZE);
     int a = file.readInt();
     if (a == accountNumber)
        return i;
   }
   return -1; // conto non trovato
}
```



- Per scrivere un oggetto non dobbiamo prima decomporlo
- Per leggere un oggetto non dobbiamo leggere i dati separatamente e poi ricomporre l'oggetto

- Flussi in scrittura
  - Classe **`ObjectOutputStream`**
- Flussi in lettura
  - Classe **`ObjectInputStream`**

Serializzazione

- La memorizzazione di oggetti in un flusso viene detta serializzazione
  - Ogni oggetto riceve un numero di serie nel flusso
  - Se lo stesso oggetto viene salvato due volte la seconda volta salviamo solo il numero di serie
  - Numeri di serie ripetuti sono interpretati come riferimenti allo stesso oggetto
- Non vengono serializzate né le variabili statiche né le variabili d'istanza dichiarate **`transient`**.

Costruttore e `writeObject`

- L'oggetto da inserire nel flusso deve essere serializzabile altrimenti viene sollevata la **`NotSerializableException`**
  - Appartenere a una classe che implementa l'interfaccia **`Serializable`**
  - **`Serializable`** non ha metodi

```
MyClass mc = new MyClass(…);
ObjectOutputStream out =
     new ObjectOutputStream(new FileOutputStream("mc.dat"));
out.writeObject(mc); //MyClass implementa Serializable
```

Lettura: `readObject`

- Legge un `Object` da file
- Restituisce un riferimento a tale `Object`
- L'output necessita di un cast
- Può lanciare un'eccezione controllata di tipo `ClassNotFoundException`

```
ObjectInputStream in =
    new ObjectInputStream(new FileInputStream("mc.dat"));
MyClass mc = (MyClass) in.readObject();
```

File Serialtester.java

```
01: import java.io.File;
02: import java.io.IOException;
03: import java.io.FileInputStream;
04: import java.io.FileOutputStream;
05: import java.io.ObjectInputStream;
06: import java.io.ObjectOutputStream;
07:
08: /**
09:    This program tests serialization of a Bank object.
10:    If a file with serialized data exists, then it is
11:    loaded. Otherwise the program starts with a new bank.
12:    Bank accounts are added to the bank. Then the bank
13:    object is saved.
14: */
15: public class SerialTester
16: {
```

File Serialtester.java

```
17:    public static void main(String[] args)
18:          throws IOException, ClassNotFoundException
19:    {
20:       Bank firstBankOfJava;
21:
22:       File f = new File("bank.dat");
23:       if (f.exists())
24:       {
25:          ObjectInputStream in = new ObjectInputStream
26:                (new FileInputStream(f));
27:          firstBankOfJava = (Bank) in.readObject();
28:          in.close();
29:       }
30:       else
31:       {
32:          firstBankOfJava = new Bank();
33:          firstBankOfJava.addAccount(new
                BankAccount(1001, 20000));
```

File Serialtester.java

```
34:          firstBankOfJava.addAccount(new
                   BankAccount(1015, 10000));
35:       }
36:
37:       // Deposit some money
38:       BankAccount a = firstBankOfJava.find(1001);
39:       a.deposit(100);
40:       System.out.println(a.getAccountNumber()
                + ":" + a.getBalance());
41:       a = firstBankOfJava.find(1015);
42:       System.out.println(a.getAccountNumber()
                + ":" + a.getBalance());
43:
44:       ObjectOutputStream out = new ObjectOutputStream
45:             (new FileOutputStream(f));
46:       out.writeObject(firstBankOfJava);
47:       out.close();
48:    }
49: }
```

Esercizio 1

- Modificare la classe **`BankAccount`** in modo che lanci un'eccezione quando viene istanziato un conto con saldo negativo, quando viene versata una somma negativa e quando si tenta di prelevare una somma non compresa tra 0 e il saldo del conto
- Definire tre eccezioni diverse una per ogni situazione descritta al punto precedente (una deve essere controllata e le altre non controllate)
- Scrivere un programma di test che prende in input una scelta dell'utente le operazioni da eseguire
- Il programma di test deve catturare e gestire una delle eccezioni non controllate e lasciare le altre due non gestite

Costante `SerialVersionUID`

- Poiché la deserializzazione potrebbe essere fatta anche da un'altra classe che si trova su un'altra macchina, bisogna verificare che le classi usate da chi ha serializzato l'oggetto e chi lo sta deserializzando siano compatibili

- La costante `SerialVersionUID` associa alla classe un identificativo univoco di tipo `Long`

- E' fortemente consigliato definire l'ID e non farlo generare automaticamente

Costante SerialVersionUID

```
import java.io.Serializable;
public class Punto implements Serializable{
  private static final long serialVersionUID = 1L;
  private int x;
  private int y;
  public Punto(int x,int y){
    this.x=x;
    this.y=y;
  }
  public String toString(){
    return "Il punto ha coordinate "+x+" e "+y;
  }
}
```

Generazione `SerialVersionUID`

- Potete usare il programma SerialVer per generare un long casuale per il SerialVersionUID

- es. `serialver -show`

Riscrittura metodi readObject e writeObject

- In alcuni casi il comportamento di default dei metodi `readObject` e `writeObject` non è adeguato
- Java permette di personalizzare la serializzazione
- Se definito, il metodo `writeObject()` determina come l'oggetto deve essere salvato su ogni flusso, di solito per inserire informazioni aggiuntive

```
private void writeObject(ObjectOutputStream stream)
                                        throws IOException {
    // invocazione del comportamento di default
    stream.defaultWriteObject();
    // scrittura di altre info
  }
```

Riscrittura metodi readObject e writeObject

```
private void readObject(ObjectInputStream stream)
        throws IOException, ClassNotFoundException {
    // invocazione del comportamento di default
    stream.defaultReadObject();
    // letture di altre info e aggiornamento stato
  }
```