

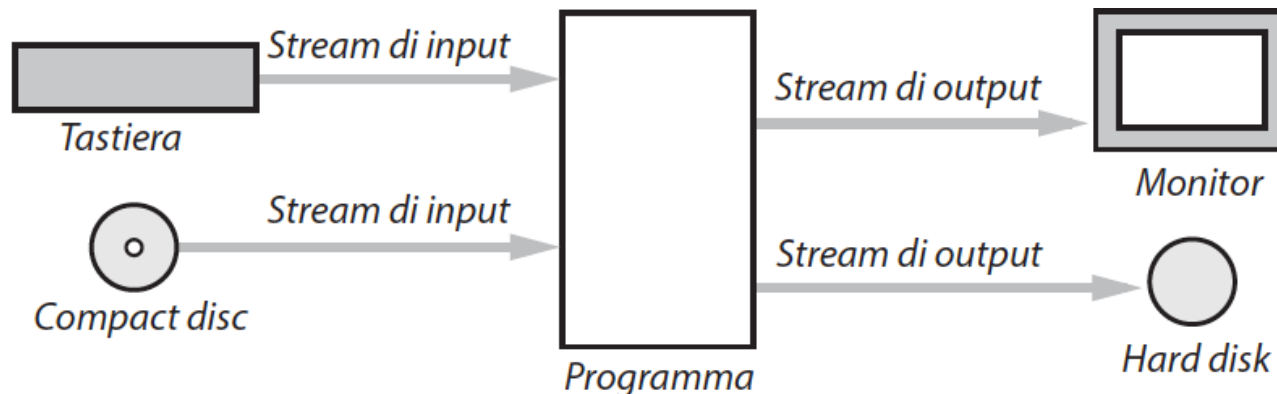
NOTE SU STREAM E FILE

Angelo Di Iorio

Università di Bologna

Stream

- In Java lettura e scrittura da e su file sono gestiti tramite flussi di dati (**stream**)
- Stesso meccanismo dalla lettura da tastiera e visualizzazione su terminale
- Uno **stream** può essere formato da caratteri, numeri o generici byte
- I **file** usano la stessa astrazione ma permettono **memorizzazione persistente** e di grandi quantità di dati



File: testo e binari

Java prevede la gestione di due tipi di file, **ognuno con i propri *stream* e metodi per processarli:**

- **File di testo**

- contenuto **human readable** (ed **editable**)
- byte interpretati come sequenze di codici ASCII o Unicode
- fine linea indicati da caratteri speciali

- **File binari**

- byte non interpretabili come codici ASCII
- es: file in linguaggio macchina, file in bytecode Java, oggetti serializzati
- gestione più efficiente ma non human-readable

Classi per manipolare file

- File testuali:
 - classi **Scanner** e **File** per leggere da file (JDK 1.5+)
 - classe **FileReader** per leggere da file (prima di JDK 1.5)
 - classe **FileWriter** per scrivere su file
- File binari:
 - classe **FileInputStream** per leggere
 - classe **FileOutputStream** per scrivere
- Queste (e altre) classi sono definite in `java.io`
- Vanno importate per essere utilizzate

Classe File

- La classe `File` fornisce un'astrazione per gestire in modo omogeneo i file e molti stream hanno costruttori che prendono in input istanze di questa classe
- Il costruttore di `File` prende in input una stringa con il nome o percorso del file.
- Due tipi di percorsi:
 - **Relativo**: calcolato a partire dalla directory di esecuzione del programma
 - **Assoluto**: calcolato a partire dalla directory radice del file-system
 - su sistemi Unix-like inizia con il simbolo “/”
 - su Windows solitamente “C:\”. NOTA: Windows usa backslash ma in Java è lecito usata sempre la notazione Unix anche su Windows

Alcuni metodi di File

```
String fileName = "src/stream/test.txt";  
File f = new File(fileName);  
if (f.exists()) {  
    System.out.println(f.getAbsolutePath());  
    if (f.canWrite())  
        System.out.println("Writable file");  
    if (f.isDirectory())  
        System.out.println("Ops, a directory!");  
}  
else  
    System.out.println("Are you sure the file exists?");
```

Percorsi e IDE

- Se non si usa un percorso assoluto, i file vengono cercati a partire dalla directory in cui è si esegue il programma
- **NOTA:** se usiamo un'IDE questa directory corrisponde alla radice del progetto
- Per questo motivo nell'esempio precedente il percorso iniziava con "`src/`" seguito dalle directory corrispondenti ai package
- Se lo stesso codice viene eseguito da linea di comando può non funzionare

Class e getResource(String s)

- Alternativamente si può usare il metodo `getResource(String s)` della classe `Class` che cerca una risorsa a partire dalla directory da cui è caricata la classe su cui è invocato
- In Java esiste infatti una **classe** che rappresenta una **classe nell'applicazione in esecuzione**
 - Espone i metodi per leggere le informazioni della classe (costruttori, metodi, proprietà, etc.)
 - Tra questi metodi `getResource(String s)`
- Da qualunque oggetto si può risalire alla classe con il metodo `getClass()` o con la proprietà `class` in classi statiche
- Dalle classi si può risalire al *classLoader* (come visto per Java FX)
- <https://docs.oracle.com/javase/8/docs/api/java/lang/class.html>


```
package stream;

import java.io.File;

public class HelloWorldFile {

    public static void main(String[] args) {

        File f = new File (
            HelloWorldFile.class.getResource("test.txt").getFile()
        );

        if (f.exists())
            System.out.println(f.getAbsolutePath());
        else
            System.out.println("Are you sure the file exists?");

    }
}
```

getResource() e getResourceAsStream()

- Nell'esempio precedente abbiamo costruito un oggetto `File` da una risorsa
- Molto spesso l'output del metodo `getResource()` può essere usato direttamente (ad esempio con le view FXML)
- Esiste anche `getResourceAsStream()` che ritorna appunto uno *stream*, identificato anche in questo caso con un percorso relativo rispetto alla directory che contiene la classe

```
public class FXMLAdminDemo extends Application {  
    public void start(Stage stage) throws Exception {  
        Parent root =  
            FXMLLoader.load(getClass().getResource("admin.fxml"));  
        ...  
    }  
}
```

I/O e file binari

- Alcuni stream sono specializzati per la gestione efficiente di file binari, cioè sequenze di byte
- `ObjectInputStream` e `ObjectOutputStream` sono le classi Java usate per leggere e scrivere dati di questo tipo (un byte alla volta)
- Se non è necessario visualizzare e/o modificare i file tramite editor di testo questi stream rappresentano una soluzione semplice ed efficiente per gestire file
- Il comportamento è molto simile a `Writer` e `Scanner` ma internamente le classi sono specializzate e ottimizzate per gestire questi contenuti

Listato 14.6 del libro

```
// import omessi
public class FileBinarioOutputDemo {
    public static void main(String[] args) {

        String nomeFile = "numeri.dat";

        try {
            ObjectOutputStream outputStream = new ObjectOutputStream(new
                FileOutputStream(nomeFile));
            Scanner tastiera = new Scanner(System.in);
            System.out.println("Inserire interi non negativi.");
            System.out.println("Negativo per terminare.");
            int unIntero;
            do {
                unIntero = tastiera.nextInt();
                outputStream.writeInt(unIntero);
            } while (unIntero >= 0);

            System.out.println("I numeri e il valore di terminazione");
            System.out.println("sono stati scritti nel file " + nomeFile);
            outputStream.close();
        }
    }
}
```

```
// import omessi
public class FileBinarioInputDemo {
    public static void main(String[] args) {
        String nomeFile = "numeri.dat";
        try {
            ObjectInputStream inputStream =
                new ObjectInputStream(new
                    FileInputStream(nomeFile));
            System.out.println("Lettura numeri non
                                negativi in " + nomeFile);

            int unIntero = inputStream.readInt();

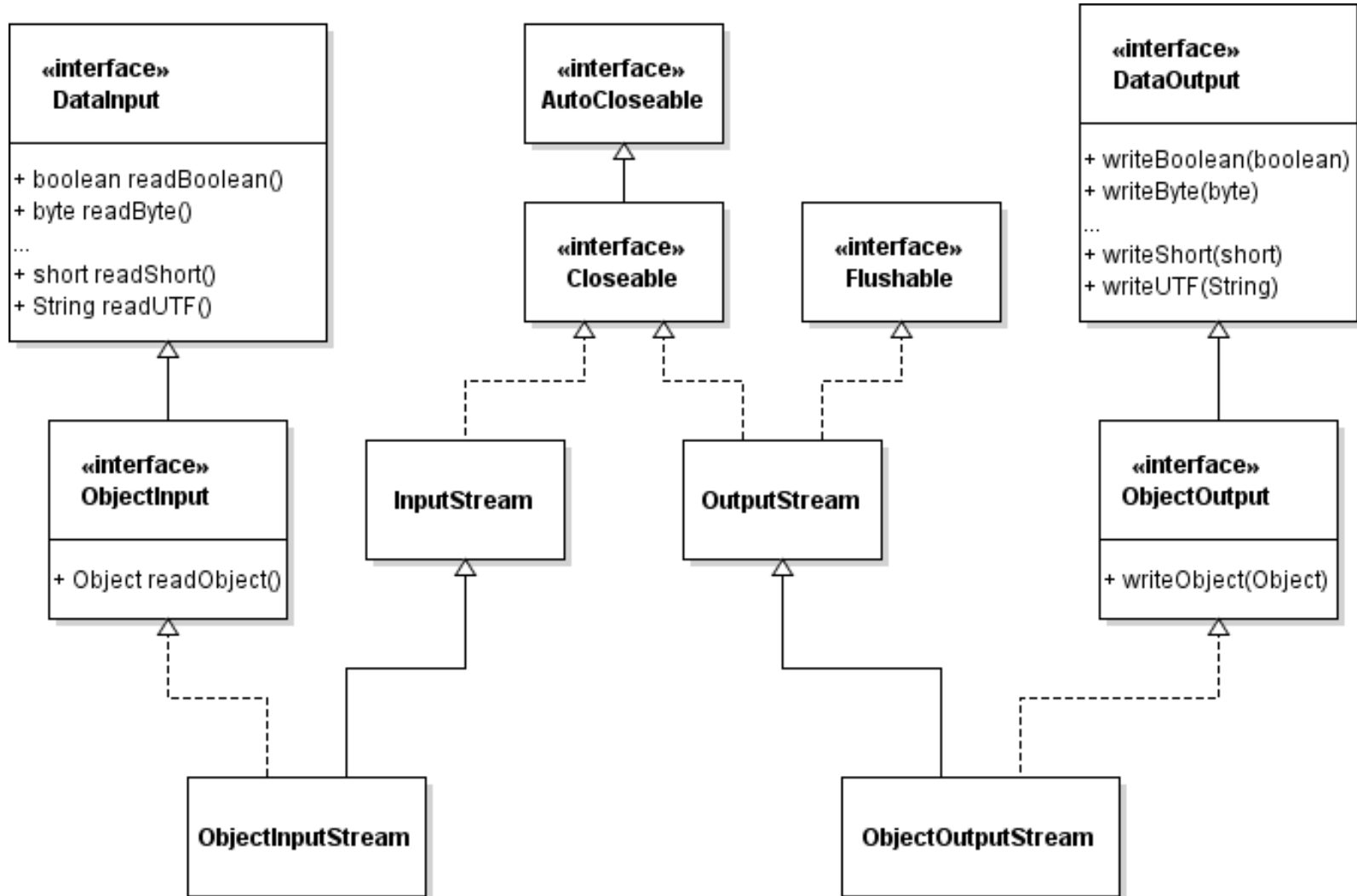
            while (unIntero >= 0) {
                System.out.println(unIntero);
                unIntero = inputStream.readInt();
            }

            System.out.println("Fine lettura dal file.");

            inputStream.close();
        }
    }
}
// blocchi catch omessi
```

Listato 14.7
del libro

ObjectInputStream e ObjectOutputStream



Serializzazione

- Java offre anche un modo semplice, chiamato appunto **object serialization**, per convertire un oggetto in una sequenza di byte che può essere quindi memorizzata in modo persistente su un file binario
- Questa conversione è possibile solo sugli oggetti **serializzabili** che implementano cioè l'interfaccia `Serializable`
- Questa interfaccia fa parte della libreria standard Java e va importata nel sorgente della classe che la implementa
- E' possibile poi usare i metodi `writeObject()` e `readObject()` degli stream `OutputStream` e `InputStream` così come visto finora per gli altri tipi di dato

Classe serializzabile

```
// import omessi
import java.io.Serializable;

public class Specie implements Serializable {

    private static final long serialVersionUID = 1L;

    private String nome;
    private int popolazione;
    private double tassoCrescita;

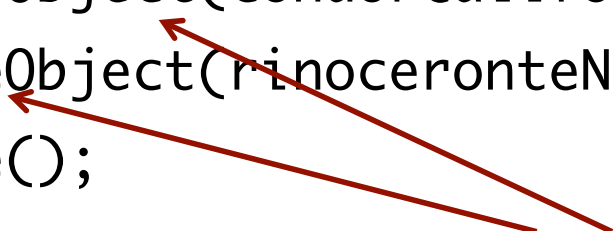
    // costruttore e metodi omessi ( incluso toString() )
```



```
// import omessi
public class IOoggettoClasseDemo {
    public static void main(String[]args) {
        String nomeFile = "specie.registrazioni";
        ObjectOutputStream outputStream = null;
        try {
            outputStream = new ObjectOutputStream(new
                FileOutputStream(nomeFile));
            // blocco catch omissso

            Specie condorCalifornia = new Specie("Condor
                della California", 27, 0.2);
            Specie rinoceronteNero = new Specie("Rinoceronte
                Nero", 100, 1.0);
            try {
                outputStream.writeObject(condorCalifornia);
                outputStream.writeObject(rinoceronteNero);
                outputStream.close();
            }
        }
    }
}

... // gestione eccezioni omessa
```



Serializzazione oggetti

Serializzazione e serial number

- La serializzazione di un oggetto memorizza:
 - informazioni sulla classe (nome e signature),
 - valori delle variabili di istanza, tranne quelle dichiarate come `transient` o `static`
 - insieme degli oggetti che l'oggetto referencia (**chiusura**). Questo insieme è noto come **object graph**. Tutti gli oggetti che lo compongono devono essere serializzabili
- La serializzazione associa un numero di versione alla classe (`serialVersionUID`) usato per verificare se mittente e ricevente hanno caricato la stessa classe
- Si memorizza anche un numero di serie per l'oggetto, che viene usato se si scrive più volte lo stesso oggetto sullo stream (si scrive solo il riferimento e non l'intero oggetto)

Classe serializzabile

```
// import omessi
import java.io.Serializable;

public class Specie implements Serializable {

    private static final long serialVersionUID = 1L;

    private String nome;
    private int popolazione;
    private transient double tassoCrescita;

    // costruttore e metodi omessi ( incluso toString() )
```

Serializzazione di array

- Poiché in Java gli array sono trattati come oggetti, la serializzazione può essere usata anche su interi array tramite `writeObject()` e `readObject()`

```
...
Specie[] unArray = new Specie[2];
unArray[0] = new Specie("Condor ", 27, 0.2);
unArray[1] = new Specie("Rinoceronte ", 100, 1.0);
String nomeFile = "array.dat";
try {
    ObjectOutputStream outputStream = new
    ObjectOutputStream(new FileOutputStream(nomeFile));
    outputStream.writeObject(unArray);
    outputStream.close();
}
```

...

Conclusioni

- Esistono inoltre anche altri stream specializzati che non vediamo ma che condividono il meccanismo generale di gestione di flussi di byte
- La serializzazione e questi stream sono usati da diverse API che gestiscono formati come CSV, JSON, etc.

