I file di testo

Prof. Viglietti Francesco
Www.in4matika.altervista.org

Flussi di Input/Output

Le informazioni presenti nella memoria secondaria (Memoria di Massa) sono indicate con il termine generico di **file**.

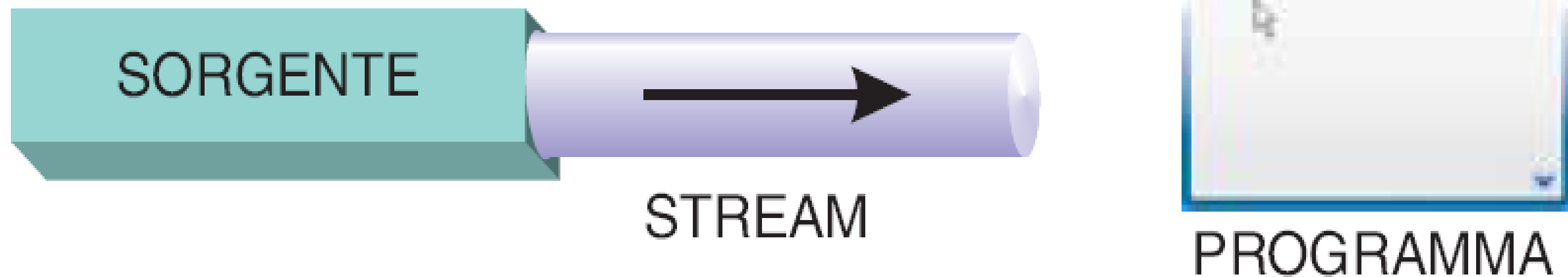
Le operazioni fondamentali :

- **Apertura→**si apre un collegamento tra M.C. e M.M. per eseguire operazioni sul file.
- **Lettura →**i dati vengono trasferiti dal file alla M.C.
- **scrittura→**i dati vengono trasferiti dalla M.C. al file.
- **chiusura→**si chiude il collegamento tra M.C. e file.

In generale, **operazioni di I/O** (Input/Output) sul file sono nel package ***java.io***.

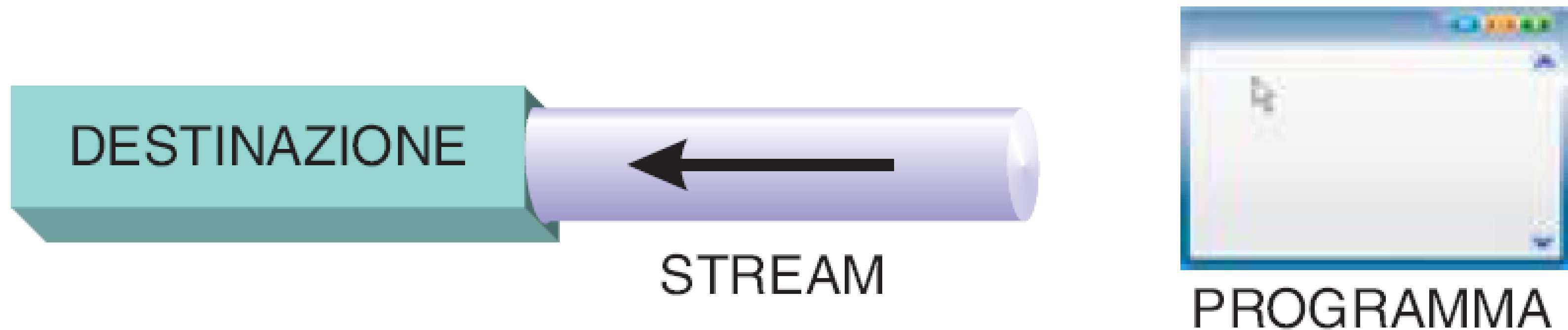
Java gestisce tutte le operazioni di I/O con il concetto di **Stream:** flusso di dati

Stream di input



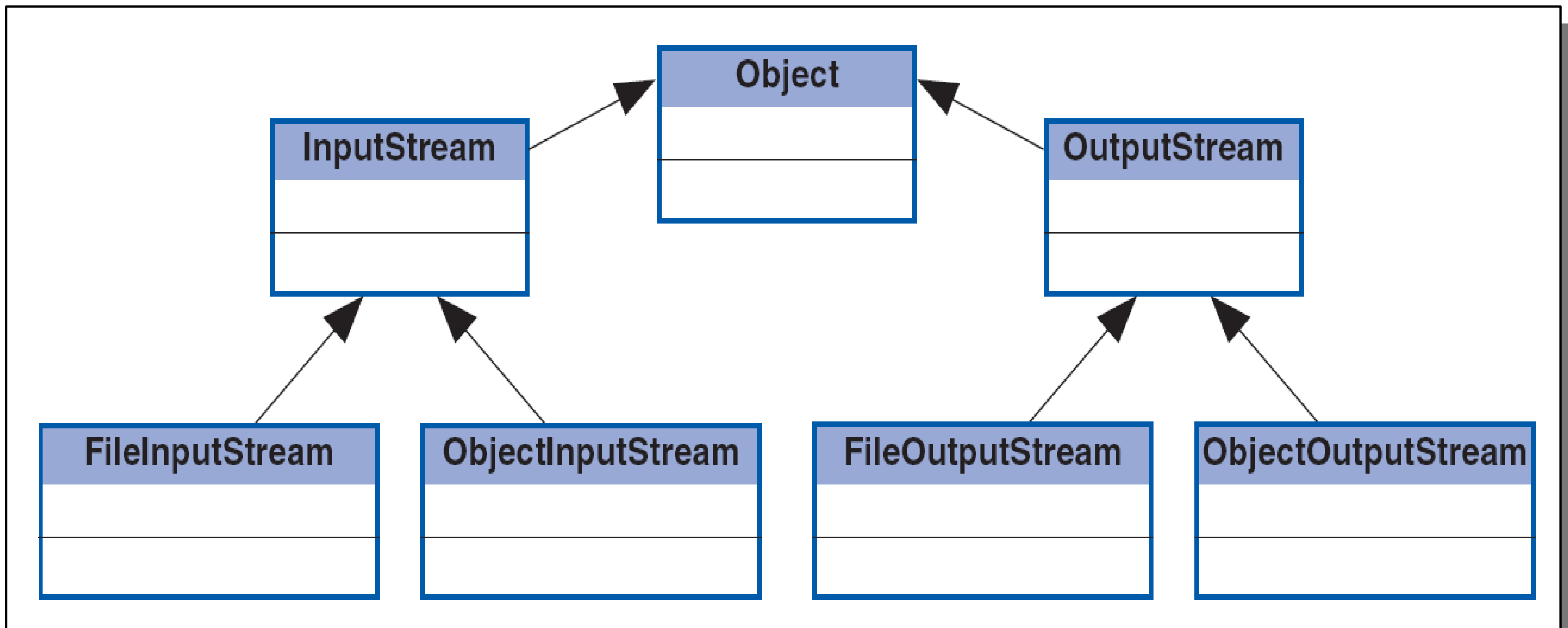
La sorgente di uno stream di input può essere ad esempio: la tastiera, un file su disco,informazioni di rete,...

Stream di output

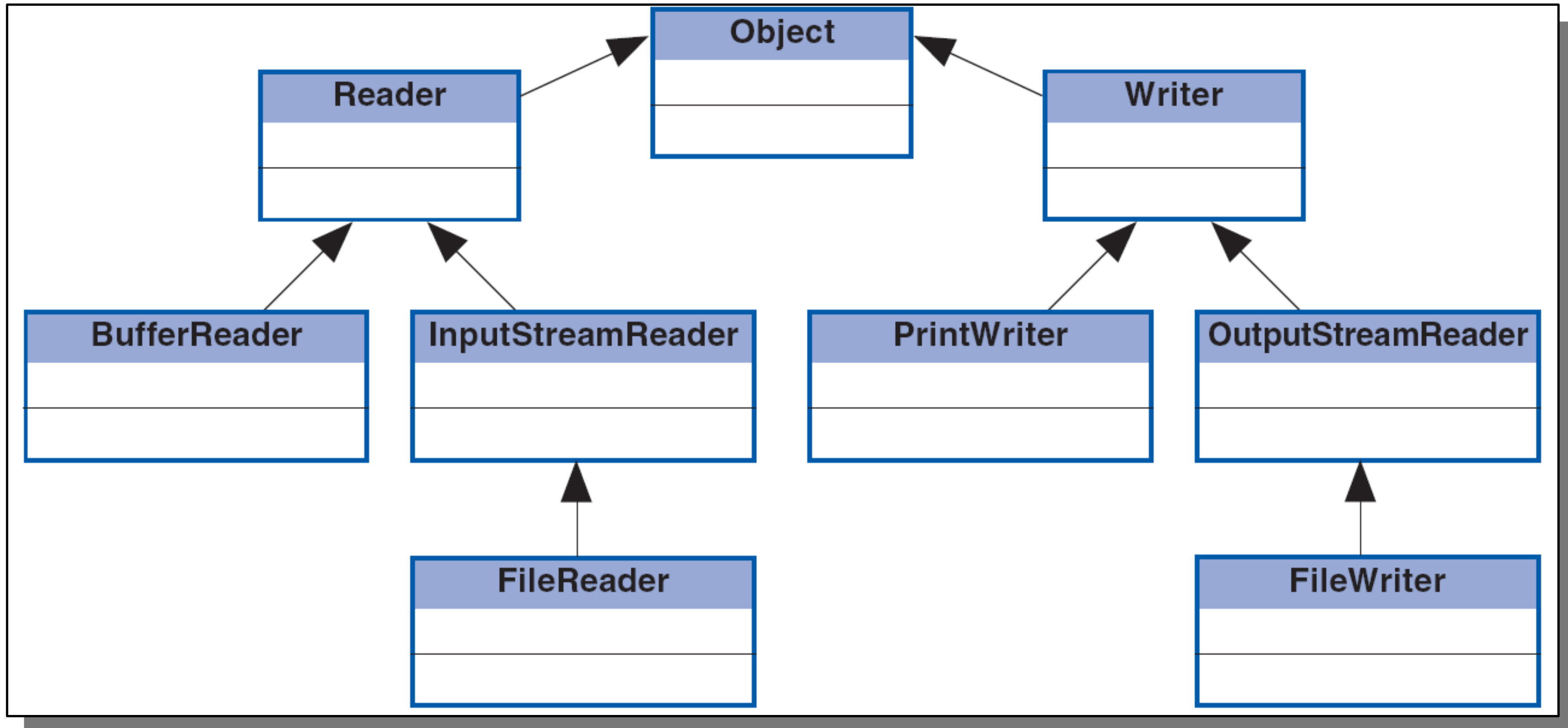


La destinazione di uno stream di output può essere ad esempio:
la stampante, un file su disco,il display, informazioni di rete,...

Classi basate sui byte



Classi basate sui caratteri

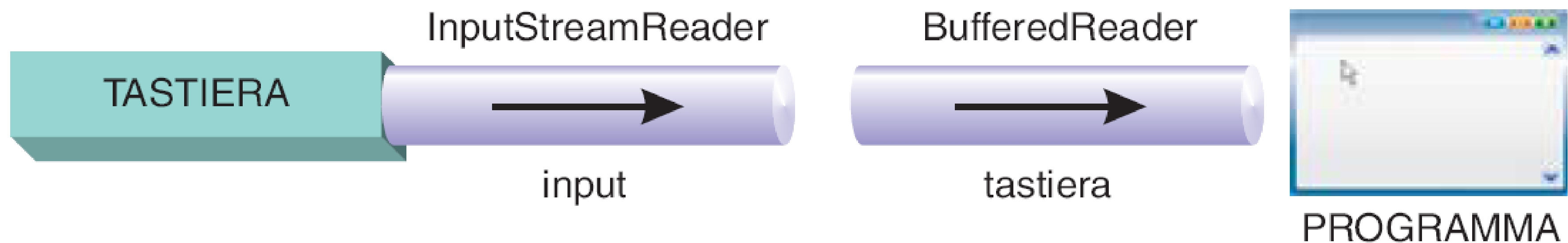


Lettura di dati da tastiera

Java può concatenare tra loro più stream. Ad esempio quello relativo allo standard input da tastiera con la lettura readLine()

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

BufferedReader tastiera = new BufferedReader(input);
```



In generale i file si dividono in due categorie:

- **File strutturati**: utilizzano classi basate sui byte e bisogna conoscere la struttura per poter interpretare il contenuto.
- **File di testo**: utilizzano classi basate sui caratteri, si possono leggere anche con il notepad (blocco note).

Gestione di file di testo

- Per i file di testo, ma per i file in generale, vengono usati due nomi, uno che identifica il file dal **OS** e l'altro che lo colleghi allo **stream**, che sparisce al termine dell'esecuzione.
- Viene usata una stringa che contiene il nome del file: ***String f=”out.txt”;*** Questo nome potrebbe anche essere dato in input da tastiera!
- L’**apertura** di un file di testo per le operazioni di output, cioè per la scrittura:

```
String nomeFile=”out.txt”;
PrintWriter outStream = null;
try{
outStream = new PrintWriter(nomeFile);//individuo il file sulla M.M.
}catch (FileNotFoundException e){
….//istruzioni per la generazione dell'eccezione
}
outStream.print(…);//scrive i dati nel file utilizzando le istruzioni…
outStream.close();//chiudere il file
```

Gestione di file di testo

- L'**apertura** di un file di testo per le operazioni di output, cioè per la scrittura, genera sempre un file vuoto! Se il file esistesse già lo sovrascriverebbe!
- Se volessi aggiungere dei dati ad un file di testo già esistente, allora dovrei aprirlo nel seguente modo, (riferendoci al codice dell'esempio precedente):

PrintWriter outStream=new PrintWriter (new FileOutputStream ("out.txt",true));

Il secondo argomento (**true**) passato al costruttore di **FileOutputStream** indica che si vogliono aggiungere dati al file, se questo esiste già!

Gestione di file di testo

- La **chiusura** di uno stream, sia di input che di output:

```
outStream.close();
```

- La **scrittura** di un file di testo che utilizza lo stream *PrintWriter* viene eseguita con i metodi **print** e **println.**

scrittura di file di testo

- Vi sono i metodi noi vedremoclasse PrintWriter ***con FileOutputStream,*** ad esempio:

```
String nomeFile=”out.txt”;
PrintWriter fw = null;
try {
    fw = new PrintWriter(new FileOutputStream(nomeFile,true));
}
catch(IOException e) {
    System.out.println("File non trovato");
}
fw.println(testo);
fw.close();
```

Lettura di file di testo con classe Scanner

- Vi sono i metodi della classe ***Scanner*** e della classe ***BufferedReader.*** Iniziamo dalla prima ad esempio:

```
String nomeFile=”out.txt”;
Scanner inStream = null;
….
try{
inStream = new Scanner(new File(nomeFile));//apertura file
}catch (FileNotFoundException e){
….//istruzioni per la generazione dell'eccezione
}
//stampa di tutti i dati del file di testo!
while(inStream.hasNextLine()){
String riga=inStream.nextLine();
System.out.print(riga);
}
//chiudere il file
inStream.close();
```

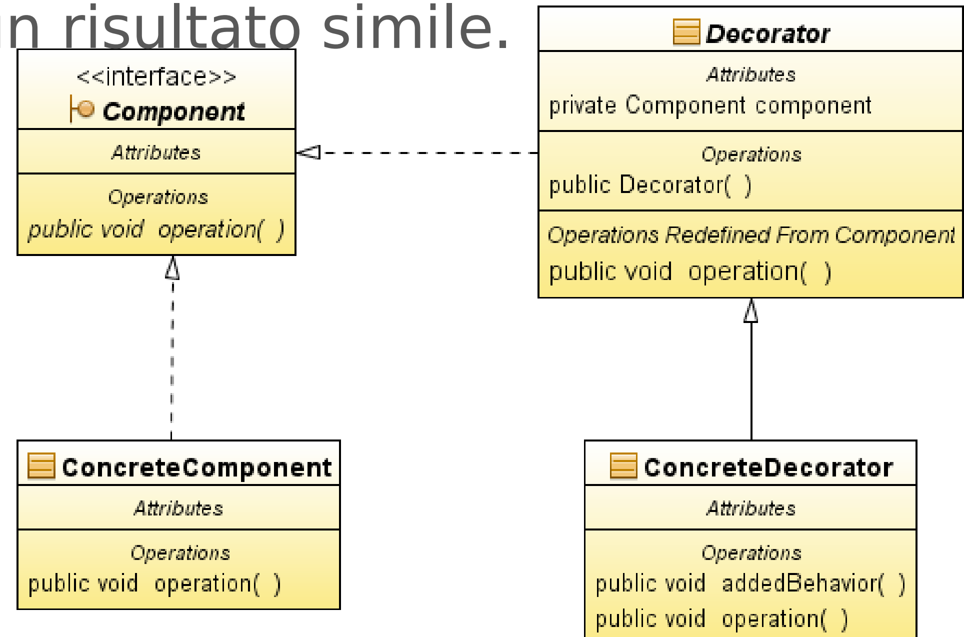
Lettura di file di testo classe BufferedReader

Però fino a java 5 era l'unica classe che gestiva i file.

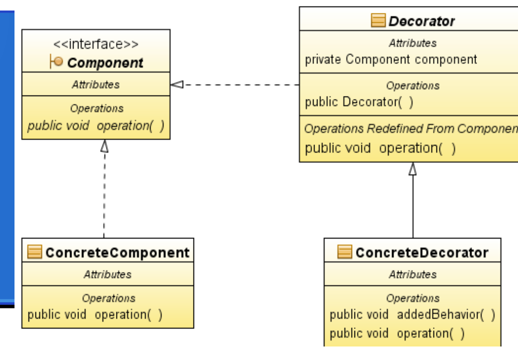
```
String nomeFile=”out.txt”;
BufferedReader inStream = null;
….
try{
//apertura file
inStream = new BufferedReader(new FileReader(nomeFile));
}catch (FileNotFoundException e){
….//istruzioni per la generazione dell'eccezione
}
//stampa di tutti i dati del file di testo!
try{
String riga=inStream.readLine();
while(riga!=null){
    System.out.print(riga);
    riga=inStream.readLine();}
inStream.close();//chiude il file
}
catch(IOException e){...//gestione eccezione...}
```

Pattern Decorator

Il Decorator è un pattern definito dalla GoF e fa parte dei pattern strutturali. Lo scopo principale di questo pattern è quello di aggiungere dinamicamente delle funzionalità alle classi evitando una proliferazione di sottoclassi. Il fatto di creare una gerarchia non è infatti sbagliato, ma ora vediamo un modo differente per ottenere un risultato simile.



diagramma



Nell'immagine precedente si vede l'interfaccia Component, che è il componente base, il quale definisce soltato il metodo operation. La classe che implementa questa interfaccia è ConcreteComponent, dove si trova la prima definizione di operation.
Per aggiungere delle funzionalità a questo Component si utilizza il pattern Decorator, definendo una classe astratta Decorator che implementa Component ed inoltre ha un riferimento a Component.
La classe che estende questo Decorator, è ConcreteDecorator, che implementa il metodo operation e al suo interno ed esegue il seguente codice:

Codice...

```
public void operation() {
   component.operation();
   addedBehaviour();
}
```

In questo modo si “decora” il comportamento del metodo operation di Component con un metodo (**addedBehaviour**).

Questo complesso meccanismo, può essere trovato anche nelle API per l’I/O in Java.

La classe astratta di base che è **InputStream**, che definisce una serie di metodi per l’I/O (read, close, skip). La classe **FilterInputStream**, estende InputStream, viene costruita con un un’istanza di InputStream e permette di aggiungere decorazioni i suoi metodi. Vediamo un semplice esempio di codice relativo a questo pattern.

Di seguito abbiamo un’interfaccia **Writer** e un’implementazione di quest’ultima

Esempi I/O

```java
package com.javastaff.pattern.decorator;

 public interface Writer {

public void presentation();

}
```

```java
package com.javastaff.pattern.decorator;
public class ConcreteWriter implements Writer {
	public void presentation() {
		System.out.println("sono un Oggetto");
	}
}
```

ConcreteWriter è una classe con un metodo che stampa una stringa. Ora andiamo a definire la classe astratta Wdecorator, che ci permette di decorare l'interfaccia Writer e tutti gli oggetti che la implementano

```java
package com.javastaff.pattern.decorator;
public abstract class Wdecorator implements Writer {
public Writer writer;
}
```

Esempi I/O

nell'ultima classe della slide precedente si è definito un attributo di tipo Writer, ora si vedrà come questo verrà utilizzato nella classe che effettivamente decora Writer.

```
package com.javastaff.pattern.decorator;
public class GentleWdecorator extends Wdecorator {
	public GentleWdecorator(Writer writer) {
		this.writer = writer;
	}

	public String getGentleGreeting() {
		return "Salve a tutti,\n";
	}

	public void presentation() {
		System.out.println(getGentleGreeting());
		writer.presentation();
	}
}
```

Si è "decorato" il metodo presentation di Writer aggiungendo una stringa a quello che viene scritto da Writer. Questo esempio ci fa capire meglio come utilizzare questo pattern