Capitolo 3

Polimorfismo ed API in Java

3.1 Polimorfismo

Uno dei migliori pregi dati da un Object Oriented Language come Java è appunto il Polimorfismo. Nello scorso capitolo abbiamo già visto qualche semplice caso anche se non è stato sottolineato il loro significato. Ad esempio si è implementato costruttori che svolgono diverse operazioni anche se presentano lo stesso nome. Infatti questa è appunto la forma più semplice di polimorfismo: due metodi possono avere lo stesso nome a patto che abbiano input e output di tipo diversi o che appartengano a classi diverse; questo non è sempre possibile in diverse tipi di linguaggi. Ma allo stesso tempo si è visto come, in qualche modo, classi possano essere descritte in modo gerarchico, definendo dipendenze tra di loro. In maniera generale, tutte le loro funzioni vengono definite dall'opportuno utilizzo delle parole chiavi: **extends**, **implements**, **interface**, **abstract** e **super**, più l'opzionale annotazione **@Override**.

Queste permettono di considerare una classe come un sottoinsieme di un'altra e quindi ne eredita la maggior parte delle capacità a meno di specializzarsi in una particolare operazione. Un esempio calzante è quello di voler definire il mondo animale; una classe base della descrizione sarà quella classe che contiene la descrizione di tutte le informazioni comune a tutti gli animali. Dopo di che sarà possibile creare un'altra classe che estende la prima e che descrive le peculiarità di un mammifero. Oppure un'altra ancora che estende la prima e descrive un pesce piuttosto che un insetto. Tuttavia sarà poi possibile creare una classe che descrive l'uomo, che estenderà quella dei mammiferi; e così via. Grazie a questo meccanismo la classe degli umani continuerà ad avere tutte le caratteristiche che hanno tutti i mammiferi e gli animali senza bisogno di doverle ridefinire ogni volta. Un altro esempio più completo è quello delle ontologie configurabili da Protege editor. Infatti se provate a creare una qualsiasi ontologia e poi cliccate su:

Tools $\rightsquigarrow$ *Generate-ProtegeOWL Java Code*

noterete che alcune classi Java vengono create in modo da descrive l'ontologia specificata. All'interno di esse si trovano pochissime istruzioni perché le diverse entità ontologiche vengono descritte come dipendenze tra classi distinte. Per ogni dubbio riguardante il polimorfismo questo rimane uno dei migliori modi per capirlo a pieno perché permette di fare delle modifiche logiche sull'ontologia e automaticamente vedere i cambiamenti che subisce il codice. Alternativamente, una spiegazione altrettanto semplice ed esaudiente si può trovare nella sezione dedicata nel tutorial di oracle[24] . Per capire a pieno il modo in cui le parole chiave nominate sopra vengono usate c'è bisogno di conoscere qual'è il loro significato, per questo verranno analizzate una ad una.

L'operazione **extends** è forse quella più usata e importante. La sua sintassi di utilizzo è ad esempio:

```
1  public class Dog extends Canine{
2      ...
```

```
    }
```

dove non c'è nessun tipo di limitazione sulla forma delle classi: Dog e Canine; cioè qualsiasi classe può estendere ed essere estesa da qualsiasi altra. Quello che questo tipo di istruzione permette di fare è quello di ereditare tutti i metodi, e gli attributi che non sono privati, della classe *Canine*. Quindi se, ad esempio questa è definita come:

```
    public class Canine extends Animal{
        ...
        public void howl(){ //ululare
            ...
        }
        ...
    }
```

e a sua volta:

```
    public class Animal{
        ...
        public void eat(){
            ...
        }
        ...
    }
```

risulterà possibile usare questi due metodi all'interno della classe Dog senza doverli riscrivere, quindi:

```
    public class Dog extends Canine{
        ...
        public void foodFound(){
            this.howl();
            ...
            this.eat();
        }
        ...
    }
```

In questo scenario può risultare conveniente usare anche il comando **super(...)** che sta ad indicare che la classe utilizza lo stesso costruttore di quella estesa. Quindi ad esempio se modifichiamo la classe Animal di prima in:

```
    public class Animal{
        ...
        // constructur
        public Animal( Date dateOfBorn){
            born( dateOfBorn);
        }
        ...
        public void eat(){
            ...
        }
        ...
    }
```

potremmo pensare di usare lo stesso costruttore per tutti gli animali visto che tutti nascono in qualche modo. Quindi l'esempio di prima diventerebbe:

```
public class Canine extends Animal{
    ...
    // constructor
    public Dog( Date dateOfBorn){
        super( dateOfBorn);
    }
    ...
    public void howl(){ //ululare
        ...
    }
    ...
}
```

```
public class Dog extends Canine{
    ...
    // constructor
    public Dog( Date dateOfBorn){
        super( dateOfBorn);
    }
    ...
    public void foodFound(){
        this.howl();
        ...
        this.eat();
    }
    ...
}
```

Così facendo il costruttore per qualsiasi tipo di oggetto Animale risulta lo stesso, indipendentemente che sia un cane o meno. Questo favorisce anche la possibilità di creare l'oggetto cane in maniera più flessibile rispetto ai precedenti esempi. Infatti in questo è corretto fare un'istanza della classe come:

```
Doog fuffi = new Doog( fuffiBirthDay);
Doog poseidone = new Caine( poseidoneBithDay);
Doog pippo = new Animal( pippoBirthDay);
```

Visto che tutti i Dog sono anche Animal e Canine. Ovviamente il contrario non è accettato e trattato come un'errore dal compilatore.

Un'altra parola chiave utile se si vogliono fare questo tipo di operazioni è l'annotazione **@Override** che viene utilizzata per sovrascrivere un metodo che altrimenti sarebbe descritto in un'altra classe. Per esempio consideriamo un particolare tipo di Cane che è particolarmente giovane non ancora in grado di ululare e capace di bere solamente latte. In questo caso lo si potrebbe descrivere attraverso la classe

```
public class JungDog extends Dog{
    ...
    // constructor
    public JungDog( Date dateOfBorn){
        super( dateOfBorn);
    }
```

```
        ...
        @Override
        public void howl(){
            // non fare niente
        }
        ...
        @Override
        public void eat(){
            drinkMilk();
        }
        ...
    }
```

Ancora una volta non vi è nessuna limitazione sul tipo di metodo che può essere sovrascritto. In java, di default ogni classe estende la classe Object; per questo tutti gli elementi disponibili sono anche degli oggetti. Infatti, ogni classe eredita alcuni metodi comuni a tutti gli oggetti; ad esempio: toString(), getClass(), wait(), notify()... una lista completa delle loro definizione è consultabile attraverso la JavaDoc a questo indirizzo[25] .

Continuando nella lista le prossime due parole chiave si allontano un po' da questo tipo di esempi. **abstract** sta ad indicare la presenza di un metodo o di una classe di cui non si è in grado di dare un'implementazione. Per cui la si definisce in termini di parametri in ingresso e in uscita in modo da poter continuare a creare il codice che tiene di conto di quel determinato metodo anche se al momento non presenta nessuna implementazione. Lo svantaggio di questo comando è che la classe non può essere stanziata con il comando *new*. Il vantaggio è che permette di sviluppare un'implementazione parziale lasciando alcune implementazioni a terzi. Un esempio di questo tipo di comando è:

```
    public abstract class MyAbstractClassName{
        // definisci gli attributi
        ....
        public MyClassName( .. ){
            // definisci il costruttore
            ...
        }
        ...
        private String myMethod1( .. ){
            // definisci il metodo
            ...
        }
        ...
        public abstract Boolean myMethod2( String st1, Integer n);
        // questo metodo non ha il corpo ma solo la definizione
        // non ho gli elementi per definirla e quindi la lascio astratta
        ...
    }
```

Visto che la classe è incompleta, per creare una sua istanza c'è bisogno di completarla ad esempio con:

```
    public class MyClassName extends MyAbstractClassName{
        @Override
        public Boolean myMethod2( String st1, Integer n){
            // implementa il metodo
            ...
        }
    }
```

Ovviamente questa classe è soggetta a tutti i comportamenti che l'istruzione extends comporta ma dovrà almeno implementare tutti i metodi astratti. Ora sarà possibile stanziare con il comando new la classe MyClassName e usufruire così anche di tutte le capacità contenute nella classe MyAbstractClassName.

Infine le parole chiave **interface** e **implements** sono legate tra loro e vengono utilizzate per definire la forma di una certa classe senza però darne nessuna implementazione. Utile nel momento in cui classi di natura diverse devono contenere gli stessi tipi di metodi da trattare in modo coerente per ognuna di loro. Un esempio, tratto dal tutoria oracle[26] ,di un interfaccia è ad esempio:

```
1  public interface Predator {
2          boolean chasePrey(Prey p);
3          void eatPrey(Prey p);
4  }
```

Da notare, che come per le classi astratte i metodi contenuti non presentano nessuna istruzione e quindi nessun body. Tuttavia a differenze delle classi astratte non è possibile fare alcuna operazione, le uniche istruzioni ammesse sono quelle di definizione dei metodi a meno del costruttore. Per rendere la classe stanziabile attraverso il comando *new* c'è bisogno di implementarla attraverso un'ulteriore classe; ad esempio:

```
1  public class Lion implements Predator {
2      // definsci gli attributi ed il costruttore
3      ....
4
5      @Override
6      public boolean chasePrey(Prey p) {
7              // definisci un'implementazione
8      }
9
10     @Override
11     public void eatPrey (Prey p) {
12             // definisci un'implementazione
13     }
14
15     ...
16 }
```

ora sarà possibile stanziare la classe nei seguenti modi:

```
1  Predator pre = new Lion();
2  Lion lion = new Lion();
```

L'utilizzo dei package è finalizzato a indirizzare i file testuali che descrivono classi diversi dentro una comune sotto cartella di *src*. Questa organizzazione gerarchica viene utilizzata per tenere un certo ordine logico e pratico all'interno di progetti complessi; è grazie a questo ulteriore servizio di modularità che si riesce a caricare attraverso un IDE come Eclipse librerie complesse senza nemmeno accorgercene. Tuttavia, è importante tenere a mente che questi meccanismi non possono essere utilizzati in un contesto static.

3.2 API: Application Programming Interfaces

Una delle operazioni che si effettua più spesso durante la stesura di un codice è quella di importare librerie esterne da poter utilizzare nel codice. Queste librerie altro non sono che una serie di classi scritte da un terzo sviluppatore, documentate attraverso una JavaDoc e pronte per essere utilizzate. Il loro utilizzo dipenda da come sono state progettate e non è ancora ben standardizzato, comunque se ne possono descrivere di due tipi. Una che richiede di usare istruzioni di comando simili a quelle viste negli esempi del Capitolo 1, l'altra che utilizza meccanismi polimorfi per configurare un generico comportamento che, altrimenti, è predefinito di default. Le prime sono generalmente più facili da utilizzare ma richiedo un numero di linee di codice da scrivere notevolmente superiori alle seconde.

Per cercare di impara ad usare librerie esterne c'è bisogno di sapere intuitivamente come sono strutturate ma, più importante, c'è bisogno di sapere quale parte della documentazione è bene consultare per risolvere un determinato problema. Considereremo ora una semplicissima collezione di classi che implementano gli oggetti necessari per scrivere e leggere da file. Una capacità decisamente interessante per elaborare un grande numero dati senza saturare la ram, a discapito della velocità. Oppure per stampare informazioni da poter riconsultare in seguito, magari dopo che un algoritmo di intelligenza virtuale, o un esperimento con sensori, ha funzionato per ore. La libreria che propongo di usare può essere semplicemente schematizzata attraverso la formalizzazione UML[27] (Unified Modeling Language). Tutto il modello del software parte dalla definizione dell'*interface* FileManager che raccoglie la definizione delle operazioni elementari che un qualsiasi oggetto, che agisce sul file, deve avere. Sono presenti inoltre 3 classi *abstract* che *implements* questo tipo di interfaccia: CommonFileOperations, LazyReader e LazyWriter; non consideriamo queste ultime due per il momento ma concentriamoci solamente sulla prima. Questa implementa tipi di operazioni che sono comuni sia nel caso della scrittura che in quello della lettura. In questo caso particolare gestisce la stringa testuale che rappresenta l'indirizzo in cui risiede il file e l'operazione di notifica di eventuali errori, dovuti dalla mancanza del file ad esempio. Tuttavia in questa classe si è deciso di non implementare alcuni metodi che dipendono dal tipo di operazione che si vuole dare, che quindi rimangono astratti. Due ulteriori classi: FileReader e FileWriter *extends* CommonFileOperations ereditando così i metodi già implementati. Quest ultime classi implementano due diverse operazioni di apertura e chiusura del file perché si basano su due oggetti forniti da java diversi: BufferedReader e BufferedWriter (vedi qui per alcuni esempi[28]). Ora i metodi di tipo astratto, per ogni rispettiva classe, sono quelli descritti dal nome: manipulateFile(). Da notare che anche gli unici metodi astratti per le classi LazyReader e LazyWriter sono le stesse. Questo perché questa API è stata realizzata con l'intento di indicare allo sviluppatore che intende usarla, che deve creare una sua classe (chiamata Reader e Writer in questo caso) la quale *extends* una tra le classi FileReader, FileWriter, LazyReader e LazyWriter. Cosi facendo questa dovrà solamente implementare l'ultimo metodo rimasto ancora astratto mentre tutte le restanti funzioni sono già pronte per essere utilizzate. Solitamente questo tipo di descrizione è fornita attraverso una documentazione dedicata che vedremo a breve.

3.3 Esercizio 3.0: Importare una Libreria Esterna su Eclipse

Solitamente le librerie sono composte da un unico file di esenzione .jar; tipicamente richiedono un controllo accurato nel verificare che le versioni siano compatibili per evitare problemi di instabilità nel software. In generale per importare una libreria di questo tipo basta creare una cartella di nome lib all'interno del progetto voluto che si trova nel workspace di eclipse. All'interno di questa cartella collezioneremo tutte le librerie utilizzate solo in quel particolare progetto. Quindi copiamo il file FileManager-SimpleAPI.jar all'interno di questa directory. Dopo di che cliccare con il tasto destro del mouse sopra l'icona che identifica questo progetto sull'albero delle directory presente a sinistra della finestra di java; ora:

Properties $\rightsquigarrow$ Java Build Path $\rightsquigarrow$ Libraries $\rightsquigarrow$ Add JARs..

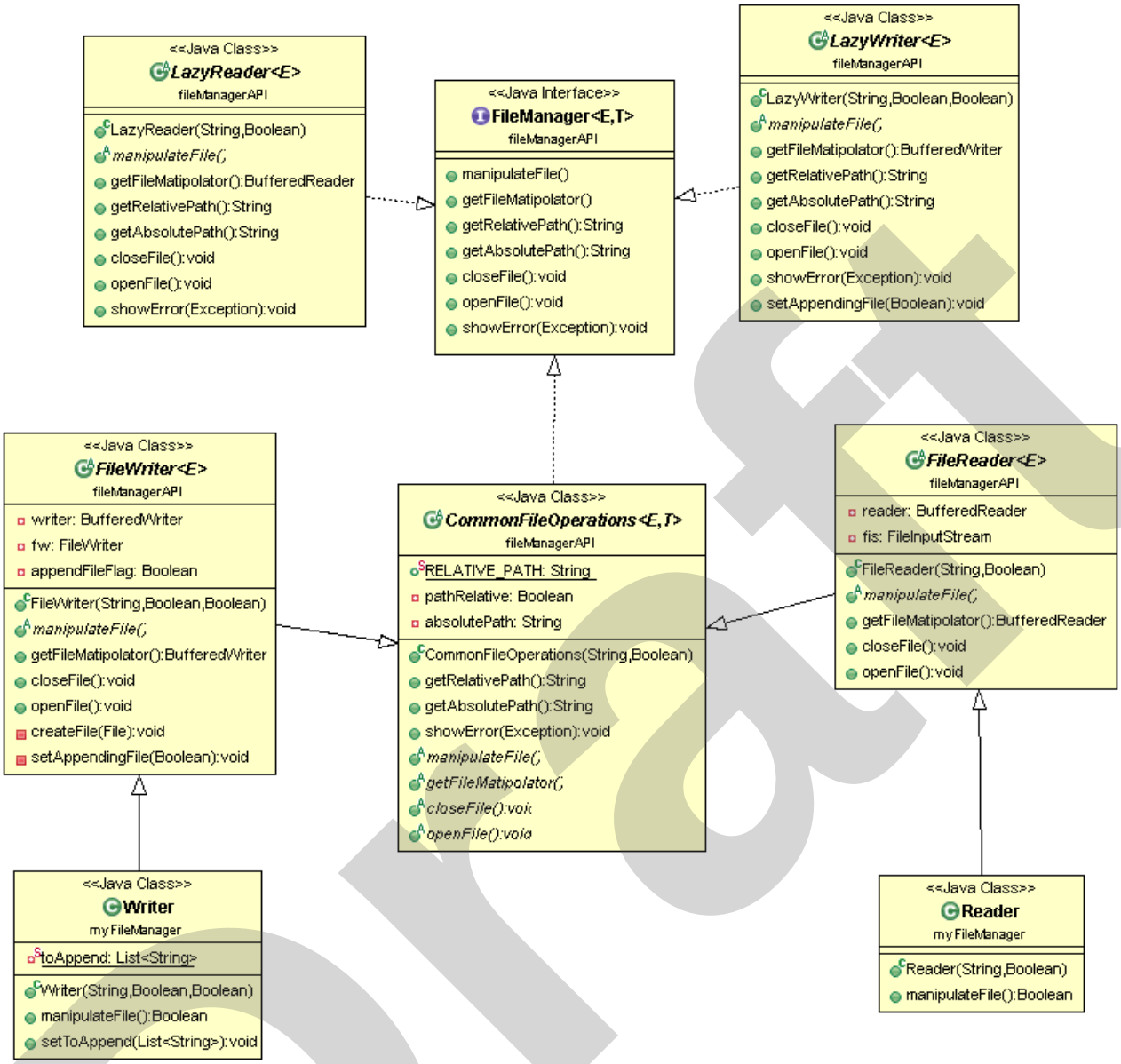

Figura 3.1: UML schema di un programma polimorfo per leggere e scrivere su File. Esempio di semplice API

e navigate fino all'interno della cartella lib poco prima creata. Nel caso in cui questa non sia ancora presente e consigliabile uscire dalla finestra delle proprietà e cliccare su:

$$\text{File} \rightsquigarrow \text{Refresh}$$

stando attenti di avere ancora il progetto selezionato sull'albero a sinistra della schermata principale. Quindi tornate ancora sull'opzione Ad JARs e terminate quello che prima non era possibile fare. Una volta confermata l'aggiunta del nuovo pacchetto può rivelarsi utile includere non solo i file eseguibili, ma anche quelli sorgenti

(quelli testuali) e la documentazione Java. Per fare questo è necessario espandere la libreria appena aggiunta cliccando sulla piccola freccia che si trova a sinistra del nome. Dopo di che fare doppio click sulla voce source attachment e navigare fino allo stesso file .jar aggiunto in precedenza. Compiere la stessa operazione per la voce Java Doc location ed inserire la posizione dello stesso file .jar attraverso l'opzione: Java Doc in archive. Utilizzate il bottone Validate per essere sicuri che Eclipse riconosca la posizione all'interno dell'archivio. Se questa operazione dà risultato negativo, inseritela manualmente attraverso l'opzione: Path within archive. Cliccate ok sia su questa finestra, che su quella delle proprietà. Ora la libreria dovrebbe essere stata importata completamente, per assicurarsi di ciò basta andare in un qualsiasi metodo di una qualsiasi classe all'interno di quel progetto e digitare una riga di comando corretta che sia formata dal nome di una classe presente nel pacchetto esterno. Nel nostro esempio basterà scrivere:

```
1    LazyWriter a;
```

Dopo questa operazione il nome della classe sarà sottolineato in rosso, avvicinarsi con il mouse fino a che non compare una finestra e controllare che sia presenta la voce: Import NomeClasse (nomePacchetto). In questo esempio: Import LazyWriter (fileManagerApi). Se questo accade vuole dire che la classe è riconosciuta all'interno del progetto, basterà cliccare su quella voce per aggiungere automaticamente una riga di comando sulle prime righe del file, che indicano che questa classe ha bisogno della classe LazyWriter, ed eliminare il segnale di errore presente in precedenza.

Per poter vedere il codice associato a quella determinata classe basterà premere ctr+mouse sinistro per essere automaticamente indirizzati alla sua definizione (questo vale per tutti gli oggetti presenti). Da qui si può non solo vedere le linee di comando, ma anche la definizione della documentazione che è scritta in un linguaggio dedicato molto simile all'html. Solitamente la documentazione viene inserita nel pacchetto jar già compilata, ma se lo si vuole generare nuovamente basterà andare sulla barra in alto della finestra principale e cliccare su:

Project $\rightsquigarrow$ Generate Java Doc

Da qui si apre una finestra su cui potete spuntare le classi per cui creare la documentazione e da cui dovete inoltre impostare la cartella dentro la quale salvare la documentazione generata. Basterà cliccare su Finish per far partire la sua compilazione e quindi la creazione. La documentazione può essere consultata aprendo con un browser i file (di solito index.html) presenti nella cartella dedicata alla documentazione. Un'altra alternativa è quella di andare nella scheda chiamata appunto Javadoc che si trova sulla stessa finestra dove c'è la console. Qui è visualizzata in modo sintetico e lo stesso tipo di informazioni vengono date avvicinandosi con il mouse ad un qualsiasi nome presente nel codice. Tuttavia il metodo consigliato è quello di aprirla tramite la terza icona a partire da destra presente nella scheda JavaDoc vista in precedenza. Facendo così si apre un'interfaccia che contiene tutte le informazioni indispensabili per utilizzare la libreria importata.

3.4 Esercizio 3.1: Scrittura e lettura su file

Come esercizio si propone di leggere la documentazione e di capire a pieno la struttura del codice. Solo quando il precedente punto è stato risolto si chiede di implementare le due classi Writer e Reader introdotti nello schema precedente. In queste classi si dovrà implementare almeno il metodo astratto, seguendo le indicazioni date dalla documentazione, che implementi come le linee testuali vengano manipolate sia in scrittura che in lettura al file. Come prima prova, implementare una logica semplice per assicurarsi che il basso livello del software funzioni (ad esempio copiare il contenuto di un file testuale all'interno di un altro). E buona norma creare una cartella all'interno del progetto denominata files e mettere qui tutti i file che volete considerare. Inoltre, tenete bene a mente che se si perde un file a questo livello di utilizzo del calcolatore non è possibile ricuperarlo. Per risolvere l'esercizio avrete bisogno anche di un metodo *main* all'interno di una classe da cui far partire l'esecuzione e gestire gli oggetti di tipo Writer e Reader appena implementati. Un aggiunta opzionale a questo esercizio è

quella di aggiungere un valore Boolean in modo da poter decidere se copiare incollare il contenuto di un file da un altro oppure fare un'operazione di tipo taglia e incolla.

3.5 Esercizio 3.2: Miglioramento delle capacità di una classe

Per vedere quanto questo tipo di programmazione è flessibile cercate di usare il primo codice dove però le classi Writer e Reader non espandono più FileWriter e FileReader ma LazyWriter e LezyReader. Concentratevi sul riconoscere come due implementazioni diverse degli stessi metodi danno risultati completamente diversi anche senza cambiare completamente tutta la strutta. Ovviamente questo esempio è molto elementare ma il concetto di base comunque non cambia. Come ultimo passo opzionale si propone di consultare le risorse in rete per cercare di aggiungere capacità alle classi Writer e Reader (nuovamente configurate in modo da espandere le classi FileWriter e FileReader). In particolare cercati di aggiungere metodi modulari in modo da poter essere anche in grado di eliminare e rinominare un file.

Capitolo 8

Appendice A: File Manager API

di seguito sono riportati i codici sorgenti utilizzati per implementare l'archivio jar introdotto nell'esercizio 3. Qui si utilizza l'interfaccia gia' analizzata nella sezione 7.3. In più e' presente:

```
1  package fileManagerAPI;
2
3  import java.io.FileNotFoundException;
4  import java.io.IOException;
5  /**
6   *
7   * This class implements some common methods defined in {@link FileManager}.
8   * In particular it has been designed to care about the initialization and storage
9   * of the directory path of the file, both in terms of relative and absolute path.
10  * It also implements a basic error notification through the command {@code e.printStackTrace();}.
11  * Finally it delegates the implementation of the other methods of the type FileManager
12  * through the modifier {@code abstract}
13  *
14  * @author Buoncompagni Luca
15  *
16  * @param <E> generic returning type of the method {@link #manipulateFile()}
17  * @param <T> generic returning type of the method {@link #getFileMatipolator()}
18  *
19  * @see FileManager
20  */
21 public abstract class CommonFileOperations<E, T> implements FileManager<E, T> {
22         // constants
23         /**
24          * describe the directory path with respect to the folder in which the software is running.
25          * It is based on the command: <br>{@code RELATIVE_PATH = System.getProperty("user.dir");}
26          */
27         public static String RELATIVE_PATH = System.getProperty("user.dir") +
      System.getProperty("file.separator");
28
29         private Boolean pathRelative = null; // is path relative?
30         private String absolutePath = null; // contains the absolute path
31
32         /**
33          * Constructor to initialize the class with a path that can be either relative or absolute.
```

```
34          * Where a relative path is the directory address starting for the folder in which
35          * the program is actually running. While absolute path is the directory address starting
36          * for the system root folder.
37          *
38          * @param path is the directory path to the file in relative or absolute notation.
39          * @param isRelative if it is true, it identifies that the parameter {@code path}
40          * defines a relative path. Otherwise, if it is false, it denotes that the parameter
41          * {@code path} is an absolute address.
42          */
43         public CommonFileOperations( String path, Boolean isRelative){
44                 this.pathRelative = isRelative;
45                 if( pathRelative){ // is true
46                         this.absolutePath = RELATIVE_PATH + path;
47                 } else { // is false
48                         this.absolutePath = path;
49                 }
50         }
51
52         @Override
53         public String getRelativePath() {
54                 // elimino la sottostringa uguale alla path relativa sostituendula con niente
55                 String relativePath = absolutePath.replace(RELATIVE_PATH, "");
56                 return( relativePath);
57         }
58
59         @Override
60         public String getAbsolutePath() {
61                 return absolutePath;
62         }
63
64         @Override
65         public void showError(Exception e) {
66                 e.printStackTrace();
67         }
68
69         @Override
70         public abstract E manipulateFile();
71
72         @Override
73         public abstract T getFileMatipolator();
74
75         @Override
76         public abstract void closeFile() throws IOException;
77
78         @Override
79         public abstract void openFile() throws FileNotFoundException, IOException;
80
81 }
```

```
1  package fileManagerAPI;
2
3  import java.io.BufferedWriter;
```

```
4  import java.io.File;
5  import java.io.IOException;
6  import java.util.ArrayList;
7  import java.util.List;
8
9  /**
10  * This class implements the operations used to be able to write lines into a file.
11  * Moreover, it propagates the implementation of the method {@link #manipulateFile()}
12  * through the modifier {@code abstract}. To Note that the generic type of data {@code T},
13  * defined in {@link FileManager} and propagate in {@link CommonFileOperations} has been
14  * fixed in this class to be an {@link BufferedWriter} object.
15  *
16  * A call to the function {@link #openFile()} will generate the proper initialization of the
17  * data returned by {@link #getFileMatipolator()} with respect to the parameters given
18  * in inputs to the constructor. Since file manipulator is of rime BufferedWriter it is
19  * possible to just use:
20  * <br><code>
21  *       &nbspString line = "something to write"<br>
22  *   &nbspgetFileMatipolator().write( line);<br>
23  *  <br></code>
24  * To make permanent the changes over the file {@link #closeFile()} should be called. This
25  * will also effect the value of the writer: {@link #getFileMatipolator()}. It must
26  * be reinitialized to be used again.
27  * </code>
28  * @author Buoncomapagni Luca
29  *
30  * @param <E> generic type of data returned by the method {@link #manipulateFile()}.
31  *
32  * @see CommonFileOperations
33  * @see FileManager
34  *
35  */
36 public abstract class FileWriter< E> extends CommonFileOperations< E, BufferedWriter>{
37
38         private BufferedWriter writer = null;
39         private java.io.FileWriter fw = null;
40         private Boolean appendFileFlag = false;
41         private List<String> toAppend = new ArrayList< String>();
42
43         /**
44          * Constructor which just call the construction of {@link CommonFileOperations}
45          * and uses the third parameter to set the method {@link #setAppendingFileType(Boolean)}.
46          *
47          * @param path is the directory path to the file in relative or absolute notation.
48          * @param isRelative if it is true, it identifies that the parameter {@code path}
49          * defines a relative path. Otherwise, if it is false, it denotes that the parameter
50          * {@code path} is an absolute address.
51          * @param appendFile if it is true that the new lines will be written at the end of
52          * the file. Otherwise, if it is false, the file will be replaced with an empty one, and
53          * than the data will be written on it.
54          */
55         public FileWriter(String path, Boolean isRelative, Boolean appendFile) {
56                 super(path, isRelative);
```

```
57                  this.setAppendingFileType( appendFile);
58          }
59
60          @Override
61          public abstract E manipulateFile();
62
63          @Override
64          public BufferedWriter getFileMatipolator() {
65                  return writer;
66          }
67
68          @Override
69          public void closeFile() throws IOException {
70                  writer.close();
71                  fw.close();
72          }
73
74          @Override
75          public void openFile() throws IOException {
76                  File f = new File( this.getAbsolutePath());
77                  if( ! f.exists()){
78                          createFile( f);
79                  }
80
81                  fw = new java.io.FileWriter( this.getAbsolutePath(), appendFileFlag);
82                  writer = new BufferedWriter( fw);
83          }
84
85          /**
86           * Create a new file given an initialized object of type {@link File}.
87           * It is also automatically called by the method {@link #openFile()}
88           * when the given directory does not contains any file with such name.
89           * Namely, if it does not exist it will be created
90           *
91           * @param f description of the file to create.
92           * @throws IOException
93           */
94          public static void createFile( File f) throws IOException{
95                  f.createNewFile();
96          }
97
98          private void setAppendingFileType( Boolean append){
99                  this.appendFileFlag = append;
100         }
101
102         public List<String> getToAppend(){
103                 return( toAppend);
104         }
105
106         public void setToAppend( List< String> lines){
107                 toAppend = lines;
108         }
109 }
```

```java
package fileManagerAPI;


import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.List;


/**
 * This class implements the operations used to be able to read the lines from a file.
 * Moreover, it propagates the implementation of the method {@link #manipulateFile()}
 * through the modifier {@code abstract}. To Note that the generic type of data {@code T},
 * defined in {@link FileManager} and propagate in {@link CommonFileOperations} has been
 * fixed in this class to be an {@link BufferedReader} object.
 * <br><br>
 *
 * A call to the function {@link #openFile()} will generate the proper initialization of the
 * data returned by {@link #getFileMatipolator()} with respect to the parameters given
 * in inputs to the constructor. Since file manipulator is of rime BufferedReader it is
 * possible to loop along all the lines of a file using:
 * <br><code>
 *       &nbspString line = getFileMatipolator().readLine();<br>
 *       &nbspWhile( line != null){<br>
 *            // do something <br>
 *              ....<br>
 *           &nbspline = getFileMatipolator().readLine();<br>
 *    }<br>
 * To make permanent the changes over the file {@link #closeFile()} should be called. This
 * will also effect the value of the reader: {@link #getFileMatipolator()}. It must
 * be reinitialized to be used again.
 * </code>
 *
 * @author Buoncompagni Luca
 *
 * @param <E> generic type of data returned by the method {@link #manipulateFile()}.
 *
 * @see CommonFileOperations
 * @see FileManager
 */
public abstract class FileReader<E> extends CommonFileOperations<E, BufferedReader>{

        // attributes
        private BufferedReader reader = null;
        private FileInputStream fis = null;

        List< String> lines = new ArrayList<String>();

        /**
```

```
54          * Constructor which just call the construction of {@link CommonFileOperations}
55          * and does not process any further the data.
56          *
57          * @param path is the directory path to the file in relative or absolute notation.
58          * @param isRelative if it is true, it identifies that the parameter {@code path}
59          * defines a relative path. Otherwise, if it is false, it denotes that the parameter
60          * {@code path} is an absolute address.
61          */
62         public FileReader(String path, Boolean isRelative) {
63                 super(path, isRelative);
64         }
65
66         @Override
67         public abstract E manipulateFile();
68
69         @Override
70         public BufferedReader getFileMatipolator(){
71                 return( reader);
72         }
73
74         @Override
75         public void closeFile() throws IOException {
76                 reader.close();
77                 fis.close();
78         }
79
80         @Override
81         public void openFile() throws FileNotFoundException {
82                 // ottieni un puntatore al file
83                 File f = new File( this.getAbsolutePath());
84                 fis = new FileInputStream( f);
85                 // inizializa l'oggetto reader
86                 InputStreamReader  isr = new InputStreamReader( fis);
87                 reader = new BufferedReader( isr);
88         }
89
90         public void setLines( List<String> lines){
91                 this.lines = lines;
92         }
93
94         public List<String> getLines(){
95                 return( lines);
96         }
97 }
```

8.1 Java Documentation

Contents

Chapter 1

Package fileManagerAPI

Package Contents *Page*

1.1 Interface FileManager

This interface contains the basic methods used to manipulate a generic file.

1.1.1 Declaration

public interface FileManager

1.1.2 All known subinterfaces

LazyWriter (in 1.6, page 15), LazyReader (in 1.5, page 13), FileWriter (in 1.4, page 10), FileReader (in 1.3, page 8), CommonFileOperations (in 1.2, page 5)

1.1.3 All classes known to implement interface

LazyWriter (in 1.6, page 15), LazyReader (in 1.5, page 13), CommonFileOperations (in 1.2, page 5)

1.1.4 Method summary

closeFile() This method is used to close all the connection with the file.
getAbsolutePath() This method can be used to retrieve the absolute path of the considered path.
getFileMatipolator() This method is used to get an initialized object which is able to perform operations over the file.
getRelativePath() This method is used to get a simplified file path, as long as it is in the same folder (or sub folders) in which the software is running.
manipulateFile() Do something on the file.
openFile() This method opens a connection with a file.
showError(Exception) This method shows informations about possible exceptions generate during the file manipulations.

1.1.5 Methods

- **closeFile**
  `void` **closeFile**`() throws java.io.IOException`
  - **Description**
    This method is used to close all the connection with the file. Possible changes on the file may be available in the file only when the file has been closed.
  - **Throws**
    * `IOException.` –
- **getAbsolutePath**
  `java.lang.String` **getAbsolutePath**`()`
  - **Description**
    This method can be used to retrieve the absolute path of the considered path.
  - **Returns** – the directory path starting for the system root.
- **getFileMatipolator**
  `java.lang.Object` **getFileMatipolator**`()`
  - **Description**
    This method is used to get an initialized object which is able to perform operations over the file.
  - **Returns** – a generic object used to manipulate the file.
- **getRelativePath**
  `java.lang.String` **getRelativePath**`()`
  - **Description**
    This method is used to get a simplified file path, as long as it is in the same folder (or sub folders) in which the software is running.

  - **Returns** – the path starting from the folder where the program is running.

- **manipulateFile**
  `java.lang.Object` **manipulateFile**`()`
  - **Description**
    Do something on the file. This method need to open, manipulate and close the file. It also has to consider possible input/output Exception that this may cause.
  - **Returns** – a generic type of data to notify that all operations has been correctly performed. Or it may return a data generated by the manipulations.

- **openFile**
  `void` **openFile**`() throws java.io.IOException, java.io.FileNotFoundException`
  - **Description**
    This method opens a connection with a file. It is not possible to operate on it before to call this procedure.
  - **Throws**
    * `IOException.` –
    * `FileNotFoundException.` –

- **showError**
  `void` **showError**`(java.lang.Exception e)`
  - **Description**
    This method shows informations about possible exceptions generate during the file manipulations.
  - **Parameters**
    * `e` – the type of exception occurs.

1.2 Class CommonFileOperations

This class implements some common methods defined in (in 1.1, page 3). In particular it has been designed to care about the initialization and storage of the directory path of the file, both in terms of relative and absolute path. It also implements a basic error notification through the command e.printStackTrace();. Finally it delegates the implementation of the other methods of the type FileManager through the modifier abstract

1.2.1 See also

- `FileManager` (in 1.1, page 3)

1.2.2 Declaration

public abstract class CommonFileOperations
extends java.lang.Object
implements FileManager

1.2.3 All known subclasses

FileWriter (in 1.4, page 10), FileReader (in 1.3, page 8)

1.2.4 Field summary

RELATIVE_PATH describe the directory path with respect to the folder in which the software is running.

1.2.5 Constructor summary

CommonFileOperations(String, Boolean) Constructor to initialize the class with a path that can be either relative or absolute.

1.2.6 Method summary

closeFile()
getAbsolutePath()
getFileMatipolator()
getRelativePath()
manipulateFile()
openFile()
showError(Exception)

1.2.7 Fields

- public static java.lang.String **RELATIVE_PATH**
  - describe the directory path with respect to the folder in which the software is running. It is based on the command:
    RELATIVE_PATH = System.getProperty("user.dir");

1.2.8 Constructors

- **CommonFileOperations**
  `public` **CommonFileOperations**`(java.lang.String` **path**`, java.lang.Boolean` **isRelative**`)`
  - **Description**
    Constructor to initialize the class with a path that can be either relative or absolute. Where a relative path is the directory address starting for the folder in which the program is actually running. While absolute path is the directory address starting for the system root folder.
  - **Parameters**
    - `path` – is the directory path to the file in relative or absolute notation.
    - `isRelative` – if it is true, it identifies that the parameter path defines a relative path. Otherwise, if it is false, it denotes that the parameter path is an absolute address.

1.2.9 Methods

- **closeFile**
  `void` **closeFile**`() throws java.io.IOException`
  - **Description copied from FileManager** (**in 1.1, page 3**)
    This method is used to close all the connection with the file. Possible changes on the file may be available in the file only when the file has been closed.
  - **Throws**
    * `IOException.` –
- **getAbsolutePath**
  `java.lang.String` **getAbsolutePath**`()`
  - **Description copied from FileManager** (**in 1.1, page 3**)
    This method can be used to retrieve the absolute path of the considered path.
  - **Returns** – the directory path starting for the system root.
- **getFileMatipolator**
  `java.lang.Object` **getFileMatipolator**`()`
  - **Description copied from FileManager** (**in 1.1, page 3**)
    This method is used to get an initialized object which is able to perform operations over the file.
  - **Returns** – a generic object used to manipulate the file.
- **getRelativePath**
  `java.lang.String` **getRelativePath**`()`
  - **Description copied from FileManager** (**in 1.1, page 3**)
    This method is used to get a simplified file path, as long as it is in the same folder (or sub folders) in which the software is running.
  - **Returns** – the path starting from the folder where the program is running.
- **manipulateFile**
  `java.lang.Object` **manipulateFile**`()`
  - **Description copied from FileManager** (**in 1.1, page 3**)
    Do something on the file. This method need to open, manipulate and close the file. It also has to consider possible input/output Exception that this may cause.
  - **Returns** – a generic type of data to notify that all operations has been correctly performed. Or it may return a data generated by the manipulations.
- **openFile**
  `void` **openFile**`() throws java.io.IOException, java.io.FileNotFoundException`
  - **Description copied from FileManager** (**in 1.1, page 3**)
    This method opens a connection with a file. It is not possible to operate on it before to call this procedure.

- **Throws**
  - `IOException.` –
  - `FileNotFoundException.` –

- **showError**
  `void` **showError**`(java.lang.Exception` **e**`)`
  - **Description copied from FileManager** (**in 1.1, page 3**)
    This method shows informations about possible exceptions generate during the file manipulations.
  - **Parameters**
    - `e` – the type of exception occurs.

1.3 Class FileReader

This class implements the operations used to be able to read the lines from a file. Moreover, it propagates the implementation of the method (in 1.3.6, page 10) through the modifier abstract. To Note that the generic type of data T, defined in (in 1.1, page 3) and propagate in (in 1.2, page 5) has been fixed in this class to be an object.

A call to the function (in 1.3.6, page 10) will generate the proper initialization of the data returned by (in 1.3.6, page 9) with respect to the parameters given in inputs to the constructor. Since file manipulator is of rime BufferedReader it is possible to loop along all the lines of a file using:

```
 &nbspString line = getFileMatipolator().readLine();
 &nbspWhile( line != null){
      // do something
       ....
     &nbspline = getFileMatipolator().readLine();
  }
```

`To make permanent the changes over the file (in 1.3.6, page 9) should be called. This will also effect the value of the reader: (in 1.3.6, page 9). It must be reinitialized to be used again.`

1.3.1 See also

- `CommonFileOperations` (in 1.2, page 5)
- `FileManager` (in 1.1, page 3)

1.3.2 Declaration

public abstract class FileReader
extends fileManagerAPI.CommonFileOperations (in 1.2, page 5)

1.3.3 Constructor summary

FileReader(String, Boolean) Constructor which just call the construction of (in 1.2, page 5) and does not process any further the data.

1.3.4 Method summary

closeFile()
getFileMatipolator()
getLines()
manipulateFile()
openFile()
setLines(List)

1.3.5 Constructors

- **FileReader**
  `public` **FileReader**`(java.lang.String` **path**`, java.lang.Boolean` **isRelative**`)`
  - **Description**
    Constructor which just call the construction of (in 1.2, page 5) and does not process any further the data.
  - **Parameters**
    * `path` – is the directory path to the file in relative or absolute notation.
    * `isRelative` – if it is true, it identifies that the parameter path defines a relative path. Otherwise, if it is false, it denotes that the parameter path is an absolute address.

1.3.6 Methods

- **closeFile**
  `void` **closeFile**`() throws java.io.IOException`
  - **Description copied from FileManager** (**in 1.1, page 3**)
    This method is used to close all the connection with the file. Possible changes on the file may be available in the file only when the file has been closed.
  - **Throws**
    * `IOException.` –
- **getFileMatipolator**
  `java.lang.Object` **getFileMatipolator**`()`
  - **Description copied from FileManager** (**in 1.1, page 3**)
    This method is used to get an initialized object which is able to perform operations over the file.
  - **Returns** – a generic object used to manipulate the file.

- **getLines**
  `public java.util.List` **getLines**`()`
- **manipulateFile**
  `java.lang.Object` **manipulateFile**`()`
  - **Description copied from FileManager** (**in 1.1, page 3**)
    Do something on the file. This method need to open, manipulate and close the file. It also has to consider possible input/output Exception that this may cause.
  - **Returns** – a generic type of data to notify that all operations has been correctly performed. Or it may return a data generated by the manipulations.
- **openFile**
  `void` **openFile**`() throws java.io.IOException, java.io.FileNotFoundException`
  - **Description copied from FileManager** (**in 1.1, page 3**)
    This method opens a connection with a file. It is not possible to operate on it before to call this procedure.
  - **Throws**
    * `IOException.` –
    * `FileNotFoundException.` –
- **setLines**
  `public void` **setLines**`(java.util.List` **lines**`)`

1.3.7 Members inherited from class CommonFileOperations

`fileManagerAPI.CommonFileOperations` (in 1.2, page 5)

- `public abstract void` **closeFile**`() throws java.io.IOException`
- `public String` **getAbsolutePath**`()`
- `public abstract Object` **getFileMatipolator**`()`
- `public String` **getRelativePath**`()`
- `public abstract Object` **manipulateFile**`()`
- `public abstract void` **openFile**`() throws java.io.FileNotFoundException, java.io.IOException`
- `public static` **RELATIVE_PATH**
- `public void` **showError**`(java.lang.Exception` **e**`)`

1.4 Class FileWriter

This class implements the operations used to be able to write lines into a file. Moreover, it propagates the implementation of the method (in 1.4.6, page 12) through the modifier abstract. To Note that the generic type of data T, defined in (in 1.1, page 3) and propagate in (in 1.2, page 5) has been fixed in this class to be an object. A call to the function (in 1.4.6, page 12) will generate the proper initialization of the data returned by (in 1.4.6, page 12) with respect to the parameters given in inputs to the constructor. Since file manipulator is of rime BufferedWriter it is possible to just use:

```
&nbspString line = "something to write"
&nbspgetFileMatipolator().write( line);
```

To make permanent the changes over the file (in 1.4.6, page 12) should be called. This will also effect the value of the writer: (in 1.4.6, page 12). It must be reinitialized to be used again.

1.4.1 See also

- `CommonFileOperations` (in 1.2, page 5)
- `FileManager` (in 1.1, page 3)

1.4.2 Declaration

public abstract class FileWriter
extends fileManagerAPI.CommonFileOperations (in 1.2, page 5)

1.4.3 Constructor summary

FileWriter(String, Boolean, Boolean) Constructor which just call the construction of (in 1.2, page 5) and uses the third parameter to set the method .

1.4.4 Method summary

closeFile()
createFile(File) Create a new file given an initialized object of type .
getFileMatipolator()
getToAppend()
manipulateFile()
openFile()
setToAppend(List)

1.4.5 Constructors

- **FileWriter**
  `public` **FileWriter**`(java.lang.String` **path**`, java.lang.Boolean` **isRelative**`, java.lang.Boolean` **appendFile**`)`
  - **Description**
    Constructor which just call the construction of (in 1.2, page 5) and uses the third parameter to set the method .
  - **Parameters**
    - `path` – is the directory path to the file in relative or absolute notation.
    - `isRelative` – if it is true, it identifies that the parameter path defines a relative path. Otherwise, if it is false, it denotes that the parameter path is an absolute address.
    - `appendFile` – if it is true that the new lines will be written at the end of the file. Otherwise, if it is false, the file will be replaced with an empty one, and than the data will be written on it.

1.4.6 Methods

- **closeFile**
  `void` **closeFile**`() throws java.io.IOException`
  - **Description copied from FileManager** (**in 1.1, page 3**)
    This method is used to close all the connection with the file. Possible changes on the file may be available in the file only when the file has been closed.
  - **Throws**
    * `IOException.` –
- **createFile**
  `public static void` **createFile**`(java.io.File f) throws java.io.IOException`
  - **Description**
    Create a new file given an initialized object of type . It is also automatically called by the method (in 1.4.6, page 12) when the given directory does not contains any file with such name. Namely, if it does not exist it will be created
  - **Parameters**
    * `f` – description of the file to create.
  - **Throws**
    * `java.io.IOException` –
- **getFileMatipolator**
  `java.lang.Object` **getFileMatipolator**`()`
  - **Description copied from FileManager** (**in 1.1, page 3**)
    This method is used to get an initialized object which is able to perform operations over the file.
  - **Returns** – a generic object used to manipulate the file.
- **getToAppend**
  `public java.util.List` **getToAppend**`()`
- **manipulateFile**
  `java.lang.Object` **manipulateFile**`()`
  - **Description copied from FileManager** (**in 1.1, page 3**)
    Do something on the file. This method need to open, manipulate and close the file. It also has to consider possible input/output Exception that this may cause.
  - **Returns** – a generic type of data to notify that all operations has been correctly performed. Or it may return a data generated by the manipulations.
- **openFile**
  `void` **openFile**`() throws java.io.IOException, java.io.FileNotFoundException`

– **Description copied from FileManager** (**in 1.1, page 3**)
This method opens a connection with a file. It is not possible to operate on it before to call this procedure.

– **Throws**
* `IOException.` –
* `FileNotFoundException.` –

- **setToAppend**
`public void` **setToAppend**`(java.util.List` **lines**`)`

1.4.7 Members inherited from class CommonFileOperations

`fileManagerAPI.CommonFileOperations` (in 1.2, page 5)

- `public abstract void` **closeFile**`() throws java.io.IOException`
- `public String` **getAbsolutePath**`()`
- `public abstract Object` **getFileMatipolator**`()`
- `public String` **getRelativePath**`()`
- `public abstract Object` **manipulateFile**`()`
- `public abstract void` **openFile**`() throws java.io.FileNotFoundException, java.io.IOException`
- `public static` **RELATIVE_PATH**
- `public void` **showError**`(java.lang.Exception` **e**`)`

1.5 Class LazyReader

This class implements an extremely lazy reader object. It would not do nothing except of writing stupid thinks. The generic data type T defined in (in 1.1, page 3) is here set to be a object. This class leave unimplemented the abstract method (in 1.5.6, page 15).

1.5.1 See also

– `FileManager` (in 1.1, page 3)

1.5.2 Declaration

public abstract class LazyReader
extends java.lang.Object
implements FileManager

1.5.3 Constructor summary

LazyReader(String, Boolean)

1.5.4 Method summary

closeFile()
getAbsolutePath()
getFileMatipolator()
getRelativePath()
manipulateFile()
openFile()
showError(Exception)

1.5.5 Constructors

- **LazyReader**
  `public` **LazyReader**`(java.lang.String s, java.lang.Boolean b)`

1.5.6 Methods

- **closeFile**
  `void` **closeFile**`() throws java.io.IOException`
  - **Description copied from FileManager** (**in 1.1, page 3**)
    This method is used to close all the connection with the file. Possible changes on the file may be available in the file only when the file has been closed.
  - **Throws**
    * `IOException.` –
- **getAbsolutePath**
  `java.lang.String` **getAbsolutePath**`()`
  - **Description copied from FileManager** (**in 1.1, page 3**)
    This method can be used to retrieve the absolute path of the considered path.
  - **Returns** – the directory path starting for the system root.
- **getFileMatipolator**
  `java.lang.Object` **getFileMatipolator**`()`
  - **Description copied from FileManager** (**in 1.1, page 3**)
    This method is used to get an initialized object which is able to perform operations over the file.
  - **Returns** – a generic object used to manipulate the file.
- **getRelativePath**
  `java.lang.String` **getRelativePath**`()`
  - **Description copied from FileManager** (**in 1.1, page 3**)
    This method is used to get a simplified file path, as long as it is in the same folder (or sub folders) in which the software is running.
  - **Returns** – the path starting from the folder where the program is running.

- **manipulateFile**
  `java.lang.Object` **manipulateFile**`()`
  - **Description copied from FileManager** (**in 1.1, page 3**)
    Do something on the file. This method need to open, manipulate and close the file. It also has to consider possible input/output Exception that this may cause.
  - **Returns** – a generic type of data to notify that all operations has been correctly performed. Or it may return a data generated by the manipulations.
- **openFile**
  `void` **openFile**`() throws java.io.IOException, java.io.FileNotFoundException`
  - **Description copied from FileManager** (**in 1.1, page 3**)
    This method opens a connection with a file. It is not possible to operate on it before to call this procedure.
  - **Throws**
    * `IOException.` –
    * `FileNotFoundException.` –
- **showError**
  `void` **showError**`(java.lang.Exception e)`
  - **Description copied from FileManager** (**in 1.1, page 3**)
    This method shows informations about possible exceptions generate during the file manipulations.
  - **Parameters**
    * `e` – the type of exception occurs.

1.6 Class LazyWriter

This class implements an extremely lazy writer object. It would not do nothing except of writing stupid thinks. The generic data type T defined in (in 1.1, page 3) is here set to be a object. This class leave unimplemented the abstract method (in 1.6.6, page 17).

1.6.1 See also

– `FileManager` (in 1.1, page 3)

1.6.2 Declaration

public abstract class LazyWriter
extends java.lang.Object
implements FileManager

1.6.3 Constructor summary

LazyWriter(String, Boolean, Boolean)

1.6.4 Method summary

closeFile()
getAbsolutePath()
getFileMatipolator()
getRelativePath()
manipulateFile()
openFile()
setAppendingFile(Boolean)
showError(Exception)

1.6.5 Constructors

- **LazyWriter**
  `public` **LazyWriter**`(java.lang.String` **s**`, java.lang.Boolean` **b**`, java.lang.Boolean` **a**`)`

1.6.6 Methods

- **closeFile**
  `void` **closeFile**`() throws java.io.IOException`
  - **Description copied from FileManager** (**in 1.1, page 3**)
    This method is used to close all the connection with the file. Possible changes on the file may be available in the file only when the file has been closed.
  - **Throws**
    * `IOException.` –
- **getAbsolutePath**
  `java.lang.String` **getAbsolutePath**`()`
  - **Description copied from FileManager** (**in 1.1, page 3**)
    This method can be used to retrieve the absolute path of the considered path.
  - **Returns** – the directory path starting for the system root.
- **getFileMatipolator**
  `java.lang.Object` **getFileMatipolator**`()`
  - **Description copied from FileManager** (**in 1.1, page 3**)
    This method is used to get an initialized object which is able to perform operations over the file.
  - **Returns** – a generic object used to manipulate the file.
- **getRelativePath**
  `java.lang.String` **getRelativePath**`()`
  - **Description copied from FileManager** (**in 1.1, page 3**)
    This method is used to get a simplified file path, as long as it is in the same folder (or sub folders) in which the software is running.

- **Returns** – the path starting from the folder where the program is running.

- **manipulateFile**
  `java.lang.Object` **manipulateFile**`()`
  - **Description copied from FileManager** (**in 1.1, page 3**)
    Do something on the file. This method need to open, manipulate and close the file. It also has to consider possible input/output Exception that this may cause.
  - **Returns** – a generic type of data to notify that all operations has been correctly performed. Or it may return a data generated by the manipulations.

- **openFile**
  `void` **openFile**`() throws java.io.IOException, java.io.FileNotFoundException`
  - **Description copied from FileManager** (**in 1.1, page 3**)
    This method opens a connection with a file. It is not possible to operate on it before to call this procedure.
  - **Throws**
    - `IOException.` –
    - `FileNotFoundException.` –

- **setAppendingFile**
  `public void` **setAppendingFile**`(java.lang.Boolean` **append**`)`

- **showError**
  `void` **showError**`(java.lang.Exception` **e**`)`
  - **Description copied from FileManager** (**in 1.1, page 3**)
    This method shows informations about possible exceptions generate during the file manipulations.
  - **Parameters**
    - `e` – the type of exception occurs.

Chapter 2

Package Networking

Package Contents *Page*

Classes

2.1 Class DataSet

2.1.1 Declaration

public class DataSet
extends java.lang.Object

2.1.2 Constructor summary

DataSet(double[][], double[])
DataSet(List, List)

2.1.3 Method summary

getArrayLabels()
getData()
getLabel(Integer)
getLabels()
getMatrixData()
getMatrixLabels()
getMatrixSample(Integer)
getNumberOfFeatures()
getNumberOfLabel()
getNumberOfSample()
getSample(Integer)
toString()

2.1.4 Constructors

- **DataSet**
  `public` **DataSet**`(double[][]` **data**`, double[]` **label**`)`
- **DataSet**
  `public` **DataSet**`(java.util.List` **data**`, java.util.List` **label**`)`

2.1.5 Methods

- **getArrayLabels**
  `public double[][]` **getArrayLabels**`()`
- **getData**
  `public double[][]` **getData**`()`
- **getLabel**
  `public double` **getLabel**`(java.lang.Integer` **idx**`)`
- **getLabels**
  `public double[]` **getLabels**`()`
- **getMatrixData**
  `public Jama.Matrix` **getMatrixData**`()`
- **getMatrixLabels**
  `public Jama.Matrix` **getMatrixLabels**`()`
- **getMatrixSample**
  `public Jama.Matrix` **getMatrixSample**`(java.lang.Integer` **idx**`)`
- **getNumberOfFeatures**
  `public java.lang.Integer` **getNumberOfFeatures**`()`
- **getNumberOfLabel**
  `public java.lang.Integer` **getNumberOfLabel**`()`
- **getNumberOfSample**
  `public java.lang.Integer` **getNumberOfSample**`()`
- **getSample**
  `public double[]` **getSample**`(java.lang.Integer` **idx**`)`
- **toString**
  `public java.lang.String` **toString**`()`

Index