# Gestione File, CSV, Classi Wrapper

## Gestione dei file in Java

In java possiamo manipolare (leggere/scrivere) file in maniera: testuale e binaria.

Per quello che ci interessa nel corso, utilizzeremo la modalità testuale.

In seguito vedremo come:

- Aprire file
- Leggere file
- Scrivere file

> 🤔 **Appunti online utilissimi per questi argomenti:**
>
> https://www.w3schools.com/java/java_files.asp

## Apertura di un file con Scanner

Per manipolare un file abbiamo già **utilizzato l’oggetto scanner:**

```
Scanner scan = new Scanner(System.in);
```

Che abbiamo costruito passando System.in (input da console) come File (ricordiamo che Java scrive su un file i comandi in console e poi questo viene eseguito dal terminale)

**Possiamo passare altri parametri al costruttore di Scanner, come ad Esempio un File da manipolare:**

```
Scanner scan = new Scanner(new File("C:\\Users\\...pathname..\\file.txt"));

// Occhio! File è un oggetto a sè che serve per dire a Java
// "Questo è un File che si trova all'indirizzo x"
```

🤔 **Occhio!**

L’**inizializzazione** di Scanner **può fallire** → bisogna gestire la possibile eccezione.

```
Scanner scan = null;

try{

  scan = new Scanner(new File("...pathname..."));

}catch(Exception e){

  e.printStackTrace();
  // Gestione eccezione

}
```

🤔 **Occhio!**

Bisogna **sempre chiudere il file quando non mi serve più:**

```
scan.close(); // Così si chiude il file
```

🤔 **Trucchetto:**

Visto che il pathName di una cartella è una roba lunghissima, tipo /home/daniel/eclipse-workspace/PlayG … etc … , se **si vogliono manipolare più file della stessa cartella**, conviene salvarsi il pathName lungo in un variabile String, ed esprimere i file come “concatenazione” del pathName ed il nome dei file:

```
// Ho file1.txt, file2.txt, file3.txt .... etc

String path = "/home/daniel/eclipse-workspace/PlayG/src/";

// Ora, se li volessi aprire tutti, invece di fare:
Scanner scan = new Scanner( new File ("/home/daniel/eclipse-workspace/PlayG/file1.txt"));

// E poi magari cambiare file
Scanner scan = new Scanner( new File ("/home/daniel/eclipse-workspace/PlayG/file2.txt"));
// Etc ...

// MOLTO MEGLIO fare:

Scanner scan = new Scanner( new File (path + "file1.txt") );
```

## Leggere con Scanner

Mi basta utilizzare i **soliti metodi di Scanner**, per leggere il file Aperto con Scanner.

```
//Una volta aperto il file con scanner come sopra

// Per leggere una riga:
String c = scan.nextLine();
System.out.println(c);

// Per leggere tutto il file:
// Si usa il valore boolean hasNextLine() -> true se il file ha ancora righe
// Da leggere!
```

```
while( console.hasNextLine() ) {

  String c = console.nextLine();
  System.out.println(c);

}
```

## Scrivere con PrintWriter (o FileWriter)

Per scrivere su file prima di tutto lo apro con un oggetto PrintWriter:

```
// Scrittura del file con PrintWriter
PrintWriter fileOut = null;

try {

  fileOut = new PrinWriter(pathName + "file.txt");

}catch(Exception e){

  e.printStrackTrace();
  // Gestione dell' eccezione
  // ....
}

// Aggiungo una linea di testo
fileOut.println("Hello, friend");
fileOut.println("~Mr. Robot");

// Anche lui deve essere chiuso alla fine
fileOut.close();
```

🤔 **Se il file non viene trovato?**

A differenze della lettura (dove se il file non viene trovato abbiamo un'eccezione), **se il file in scrittura non viene trovato, allora viene creato da 0.**

⚠️ **Nel codice precedente c’è un problema!**

Questo codice va bene, quando voglio **sovrascrivere un file.** La nuova scrittura cancella il contenuto precedente del file.

**Se vogliamo mantenere i dati vecchi E SCRIVERNE ANCHE DI NUOVI, dobbiamo modificare il codice come segue:**

```
// Cambio la costruzione di PrintWriter
fileOut = new PrintWriter( new FileOutputStream(pathName + "prova.txt", true) );

// Prima di tutto gli passo un oggetto FileOutputStream e non un path
// E poi un secondo argomento "true", che sta per "voglio fare l'append"
// del nuovo contenuto oppure "non voglio fare l'append"
```

---

## Problema dell’ e-commerce

Scrivere e leggere informazioni da una fonte (che sia un database, un file, qualsiasi cosa), **genera problemi di consistenza:**

Se mentre sto cercando di comprare qualcosa da Amazon

→ tipo un libro di cui ci sono solo 2 pezzi rimasti

Succede che **qualcuno compra tutti i libri disponibili**, io posso cercare (se non viene gestito a livello di programmazione) di **comprare un oggetto che non c’è.**

Quando qualcuno scrive un file, questo **deve essere aggiornato anche per tutte le altre persone che stanno cercando di scrivere quel file.**

🤔 Questa cosa è abbastanza avanzata → per ora non ce ne occuperemo (è una curiosità utile da sapere).

## Suddivisione logica in chunk e token

Logicamente, il contenuto di un File può essere suddiviso in :

**Chunk** → pezzo di file, generalmente riga

**Token** → un elemento di un Chunk, generalmente un parola

**Tutto quello che abbiamo visto fino ad ora:**

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/98ed21c7-a64b-4a15-909d-8166dfaa277b/Intro_gestione_file.java

## File CSV

CSV è un formato file, che serve per **Immagazzinare informazioni.**

E’ un po’ come il nonno di un file Excel:

```
"informazione1", "informazione2", "informazione3"
"informazione4", "informazione5", "informazione6"
"informazione7", "informazione8", "informazione9"
"informazione10", "informazione11", "informazione12"
```

Ogni **riga** corrisponde ad un **Chunk** (come la riga Excel)

Ed ogni **parola tra gli apici** corrisponde ad un **Token** (come la casella Excel).

I file **CSV** sono utili per salvare **informazioni semplici** (ad esempio i punti che fa ogni giocatore in un gioco tipo Pinball).

Posso con Java → **dividere tutto il contenuto in Chunk, Token** e manipolare il file CSV come un “database semplicissimo per salvare informazioni”.

> 🤔 Link UTILISSIMO per capire bene la manipolazione di CSV:
>
> https://www.javatpoint.com/how-to-read-csv-file-in-java

---

## Esercizio

Creare le seguenti classi:

**classe Auto:**

- *Targa
- *Tipologia Utilitaria, Sportiva
- *Produttore

- *Costruttori, set e get

**classe Garage (classe container):**

*Array di n veicoli

*Costruttore

*add Auto

**Task**

Creare un’istanza di Garage a partire dal contenuto di un file CSV.

> 🤔 Suggerimento:
>
> Dividi il contenuto del file in Chunks e Token usando la funzione split dopo la lettura di una riga del CSV

**Soluzione:**

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/aae90184-0558-4350-87eb-d576199a8129/Esercizio_CSV_1.zip

---

## Classi Wrapper

Sono degli oggetti utilizzati per **inglobare** dei tipi primitivi ed **“estenderne le capacità”.**

**Motivazione:**

Mi permette non solo di avere il valore associato al tipo primitivo (ad esempio 10), ma anche **dei metodi della classe Wrapper.**

**Esempio:**

```
// Intero normale (è il normalissimo tipo primitivo int)
int i = 10;

// La classe wrapper
Integer i = new Integer(10);

// Ho tanti metodi interessanti tipo
/*
  i.compate(j);
  i.value();
  etc ... (e ne posso anche definire altri)
*/
```

🤔 **Terminologia:**

Passare un tipo primitivo ad una classe Wrapper **prende il nome di “Boxing” del tipo primitivo.**

Tirare fuori il valore **da una Wrapper** ad un primitivo, **si dice “fare Unboxing”:**

```
//Boxing
int i = 10;
Integer n = new Integer(i);

// Unboxing
int nn = n.intValue();

// AutoBoxing -> Qua sto facendo un Unboxing
// implicito, con cast tra tipi Integer -> int
int nnn = n;

// vale anche il contrario
Integer x = 10;
```

Ho una vasta libreria di funzioni statiche già pronte con **tutte le classi Wrapper.**

Tra queste, alcune molto importanti sono **le funzioni di parse():**

```
//Parse di una stringa in un intero, ossia:
//Prendo una stringa e la interpreto (casto) in un intero

int x = Integer.parseInt("1234");

// Sono persino overloadate, ad esempio (sempre parseInt() ):
int s = Integer.parse("1234567891011", inizio, fine, radice);
```

**le funzioni “is_a”:**

```
// Ci sono funzioni che restituiscono un boolean se il dato passato è
// di un certo tipo "is_a"

char b = '3';
Character.isDigit(b); // Lo chiamo così perchè ricordiamo che è statico

char b = 'A';
Character.isUpperCase(b);
```

🤔 Nel corso abbiamo visto → Character, Integer, Double etc …

---

## Esercizio

Implementa il metodo di Integer parseInt() → con un metodo statico grezzo.

```
public class MainWrap {

  public static void main(String[] args) {

    System.out.println( MainWrap.mioParser("1234") );

  }

  public static int  mioParser(String str) {

    // esponente del 10
    int potenza = str.length() - 1;
    int risultato = 0;

    for (int i = 0; i < str.length(); i++) {

      // Tolgo all'ASCII corrente l'offset dello 0
      int current = ( (int) str.charAt(i)  - 48 );

      // Moltiplico * 10 ^ potenza -> numerazione posizionale
      risultato += (  current * Math.pow(10, potenza) );

      potenza--;

    }
```

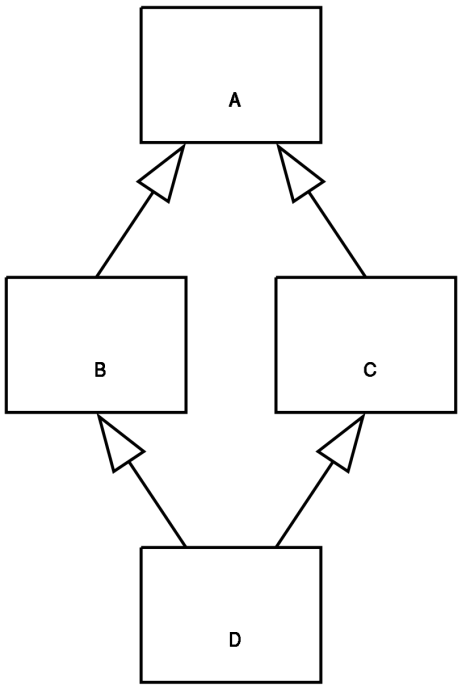
```
    return risultato;
  }

}
```

---

## Problema del diamante

---

Questo è un problema abbastanza famoso. Per spiegarlo **guardare l’immaginina sotto:**




Questo vuol dire B eredita A, C eredita A, D eredita B e C e quindi anche A (un bel casino)

**Problema!** Se due classi B e C ereditano dalla classe A, e la classe D eredita sia da B che da C: **se un metodo in D chiama un metodo definito in A, da quale classe viene ereditato?**

E’ ambiguo → non si sa da chi lo eredita!

🤔 Per questo in Java **non esiste l’ereditarietà multipla!** (si taglia la testa al toro insomma).

Se proviamo a farlo → **Errore**

🤔 Per una spiegazione più esaustiva:

https://italiancoders.it/ereditarieta-multipla-in-c/

---

## Classi Abstract

Sono delle classi che hanno la possibilità di **dichiarare dei metodi abstract:** ossia dei metodi che **devono per forza essere overridati** nelle classi figlie della classe Abstract.

In generale, i comportamenti “particolari” che ho quando parlo di classe Abstract sono:

- Se una classe A eredita un’ altra classe Abstract B **tutti i metodi dichiarati come abstract nella classe B** devono essere implementati (@overridati) nella classe A.
- Un classe Abstract **non definisce un tipo.** Se ho una abstract A, non posso **istanziare un oggetto di tipo A.**
- Può avere metodi non astratti e tutte le altre caratteristiche di una classe.

🤔 **Occhio!**

Le classi che ereditano le classi **abstract soffrono del problema del diamante,** in quanto ereditano anche **i metodi *non astratti* per intero (quindi l'ambiguita di "da chi eredito cosa" rimane.**

L'ereditarietà multipla di classi Abstract **rimane proibita da Java.**

🤔 **Visibilità protected**

Ora che abbiamo classi genitore e classi figlie, possiamo utilizzare **dei dati protected come dati membro della classi genitore.**

Se una classe genitore contiene questo tipo di dati, questi saranno visibili solo **dalle sue classi figlie.**

---

## Interfacce

Sono delle classi con **solo metodi astratti.**

E' l'equivalente dei metodi virtuali di C++ → **ogni classe che eredita l' Interfaccia deve implementare la *sua versione di TUTTI i metodi dell'interfaccia.***

In questo modo evito l'ambiguità del **problema del diamante.**

Posso infatti **ereditare da più interfacce.**

---

## Abstract vs Interfacce a livello logico

Quando uso le Abstract e quando le Interfacce?

### Le interfacce

→ Se voglio ereditarietà multipla.

→ Quando voglio creare “famiglie di classi dello stesso tipo”.

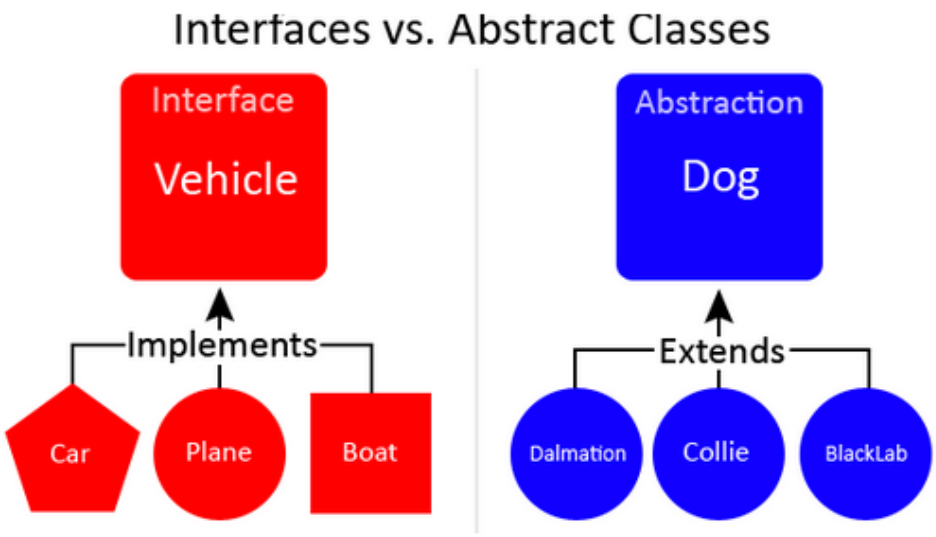
→ Quando quello che sto implementando nella classe figlia è un concetto molto “astratto”

### Le abstract

→ Se voglio semplicemente specializzare una classe

> 🤔 **TRUCCO!!!!**
>
> Come scelgo tra una abstract ed una interfaccia???
>
> Prima di tutto le scelgo se devo **specializzare il comportamento di una classe.**
>
> Poi mi faccio la seguente domanda:
>
> La classe Genitore, ha senso anche se non venisse specializzata?
> Se la risposta è sì → Abstract.
> Se la risposta è no → Interfaccia.

### Come qui sotto praticamente:




🤔 Le interfaces sono molto importanti per le Servlet (quando le vedremo).

---

**Esercizio**

---

Implementare la situazione seguente:

Classe Padre → Player con **attributi**:

- Attacco
- Difesa

**E metodi:** Un metodo astratto con un “Attacca”

Classi Figlio → Tank, Guerriero e Mago. Che implementano la loro versione di Attacca.

**Soluzione**

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/edceb8df-5548-4169-a8cb-bf5015c53596/Game_abstract.zip

---

## ArrayList

Vediamo ora una struttura dati chiamata **Lista.**

La lista è un “array” che cresce / decresce se aggiungo o tolgo metodi.
E’ un **array “dinamico”.**

Ci sono poi diversi **Oggetti** come ArrayList, tabella di Hash etc … che sono **degli oggetti che definiscono il modo in cui si manipola e si accede alla Lista dinamica.**

**ArrayList** → è un oggetto che definisce i metodi di accesso e manipolazione sulla Lista.

```
//Sintassi
ArrayList<tipoDato> nomeLista = new ArrayList<tipoDato>(dimensione_iniziale);


// Esempio di un arrayList di stringhe
ArrayList<String> l = new ArrayList<String>();

// Posso fare un upcasting sicuro
List<String> l = new ArrayList<String>();
```

Implementa vari metodi per gestire la lista

```
ArrayList<int> interi = new ArrayList<int>();

// Posso aggiungere un elemento
interi.add(10);

// Rimuovere
interi.remove(10);
```

I vari metodi per aggiungere, eliminare, modificare etc … sono qui:

https://www.w3schools.com/java/java_arraylist.asp

---

**Prossimi argomenti:**

Altre Strutture dati + Classe Generics + MySQL e SQL