

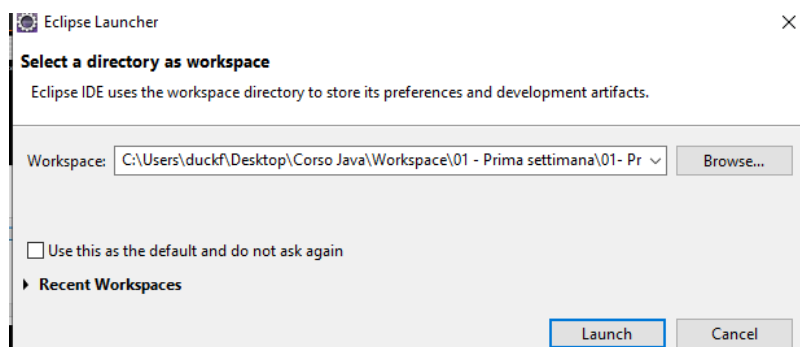
## Sommario

1.0 – INTRODUZIONE A ECLIPSE .....	2
1.1 CREARE NUOVO PROGETTO .....	3
2. JAVA .....	6
2.1 Sintassi e cose utili .....	7
2.1.2 – Operatori matematici .....	7
2.1.3 – Operatori logici .....	7
2.1.4 – Convertire il case .....	7
2.1.5 – Modulo .....	7
2.2 I tipi .....	8
2.2.1 – Lo scope di una variabile .....	9
2.3 Le variabili .....	9
2.4 Principi della programmazione .....	10
2.5 Convertire string in numeri .....	10
2.6 Lo scanner e gli input dell'utente .....	10
2.7 Secondo principio della programmazione – la selezione .....	12
2.7.1 – If piatto .....	13
2.7.2 – <b>If else</b> .....	13
2.7.3 – If annidati .....	14
2.7.4 – Operatore ternario .....	14
2.7.5 – <b>Switch</b> .....	15
2.8 Terzo principio della programmazione – iterazione .....	16
2.8.1 – Ciclo While .....	16
2.8.2 – Ciclo do while .....	17
2.8.3 – Ciclo for .....	17
2.8.4 – Differenze tra diversi cicli .....	18
2.8.5 – Ciclo for each .....	18
2.9 Leggere file .....	19
2.10 – I vettori .....	21
2.10.1 Split .....	24
2.11 INTRODUZIONE AGLI OGGETTI .....	25
2.11.1 Struttura del progetto .....	28
2.11.2 Visibilità delle proprietà e i metodi .....	28
2.12 PRINCIPI DELLA PRORAMMAZIONE A OGGETTI .....	30
2.13 ArrayList .....	30
2.13.1 ArrayList e tipi boxati .....	32

2.14 Costruttori .....	32
2.15 Extends .....	34
2.16 Polimorfismo dell'oggetto .....	36
2.16.1 Polimorfismo dell'oggetto – override.....	36
2.16.2 Polimorfismo dell'oggetto – overload .....	36
2.17 Far vedere a Java il tipo concreto di un oggetto .....	37
2.18 Aggregatori .....	38
2.19 Abstract .....	38
2.19.1 Metodi abstract .....	39
2.20 Getters and Setters.....	40
2.21 Proprietà e metodi statici .....	42
2.22 Gestione delle eccezioni (try/catch).....	44
2.23 Mappe.....	47
2.24 LE INTERFACCE.....	48
3.0 MYSQL.....	49
3.1 Creare un database .....	52
3.2 Queries sql.....	54
3.3 Modificare le tabelle.....	54

## 1.0 – INTRODUZIONE A ECLIPSE

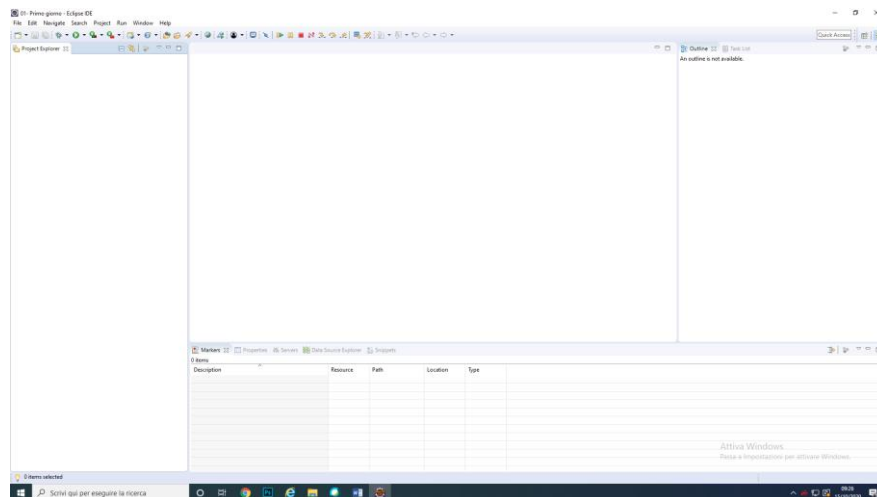
**Eclipse** è l'ide (integrated development environment) che useremo nel corso. Un IDE è un ambiente di sviluppo.



All'avvio, Eclipse ha bisogno di selezionare un **workspace**, ovvero lo spazio dove vengono salvati tutti i dati di ciò che poi viene eseguito.

Importante è lasciare bianca la spunta di default, in modo che ogni volta si possa selezionare la cartella per il workspace. Una volta selezionata la cartella lancio.

Eclipse crea in automatico la cartella metadata, in cui sono salvati i dati che Eclipse usa per lavorare.



Nella finestra a sinistra, il **project explorer**, si vedono i progetti che ho.

Outline e task list al momento non ci interessano, quindi possiamo toglierli per lasciare spazio al codice. Allo stesso modo la finestra sotto.

Le schede fondamentali sono:

- Quella del codice;
- La console, in cui vedo l'esecuzione del mio codice, il risultato.

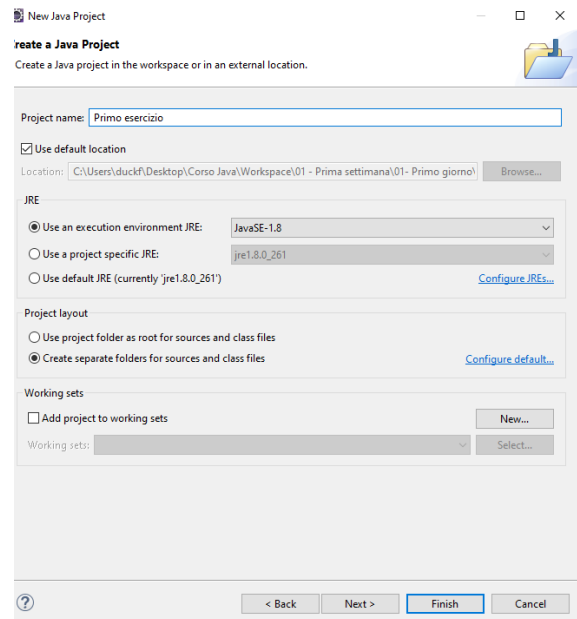
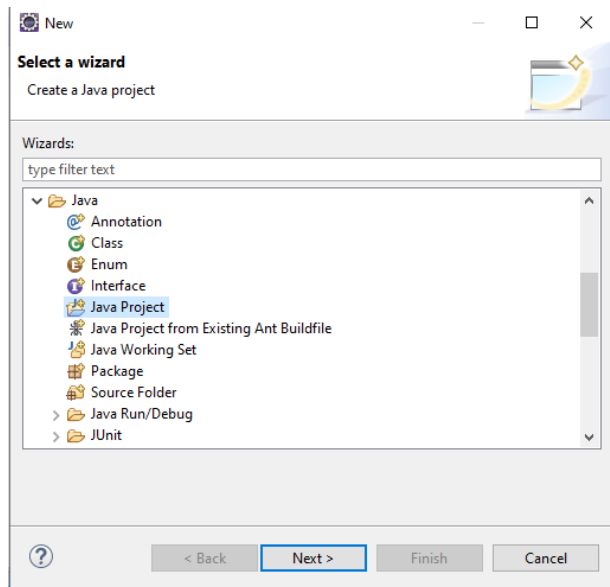
Una **virtual machine** è una macchina virtuale che mi permette di eseguire il codice. Il risultato dell'esecuzione si vede nella console.

## 1.1 CREARE NUOVO PROGETTO

Per creare un nuovo progetto devo fare:

file -> new -> other

A questo punto ci sono vari tipi di progetti che vengono dati in automatico, noi dobbiamo fare "java project", siccome non c'è subito in vista, dobbiamo cercare in "altro".

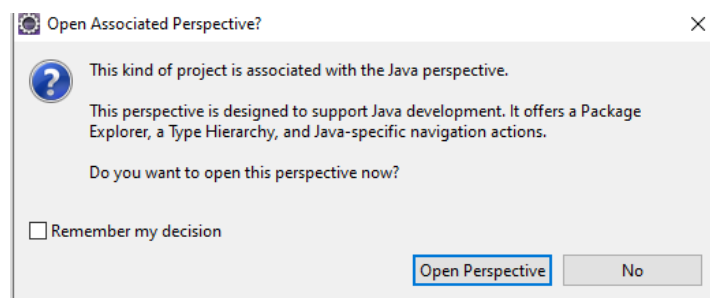


Per ora devo:

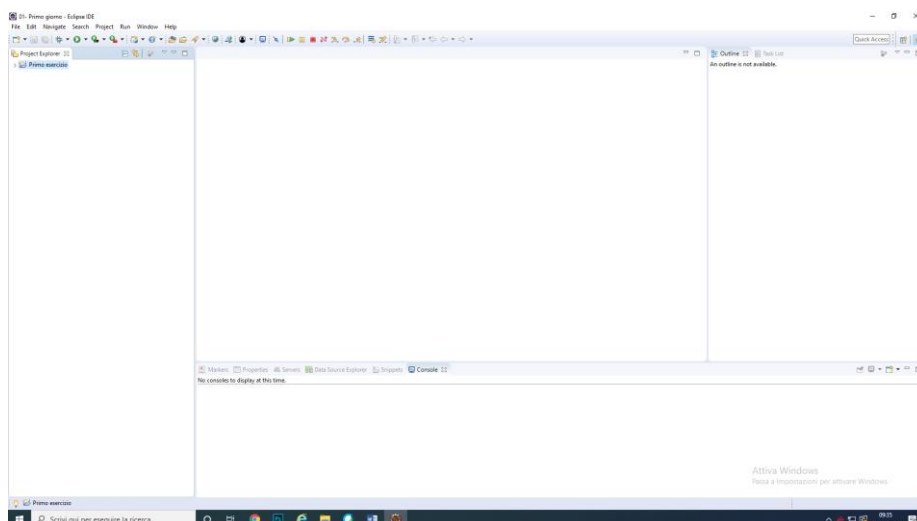
1- Impostare il nome del progetto;

2- Usare "default location"= salva il progetto utilizzando la cartella di riferimento del workspace, data quando ho aperto Eclipse.

A questo punto lancio e viene fuori questo messaggio:








Dice che chi fa progetti in Java, normalmente usa un workspace diverso. Chiede se voglio che apra la scrivania come dovrebbe essere di default. Diciamo di no, così viene mantenuta la scrivania come l'abbiamo impostata prima.



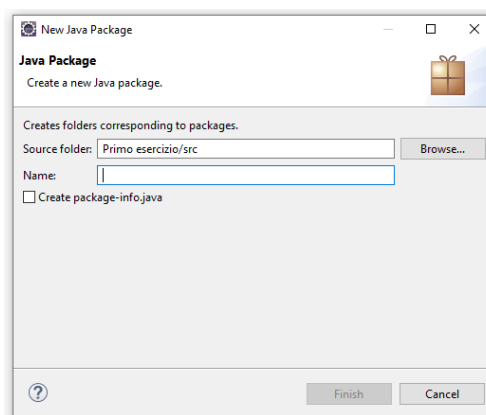
Siamo a questo punto: abbiamo creato il primo grande contenitore, la cartella in cui saranno salvati i dati.

La cartella è strutturata in questo modo:

	.settings	15/10/2020 09:33	Cartella di file	
	bin	15/10/2020 09:33	Cartella di file	
	src	15/10/2020 09:33	Cartella di file	
	.classpath	15/10/2020 09:33	File CLASSPATH	1 KB
	.project	15/10/2020 09:33	File PROJECT	1 KB

Dentro a “src” vanno messi tutti i file che creo.

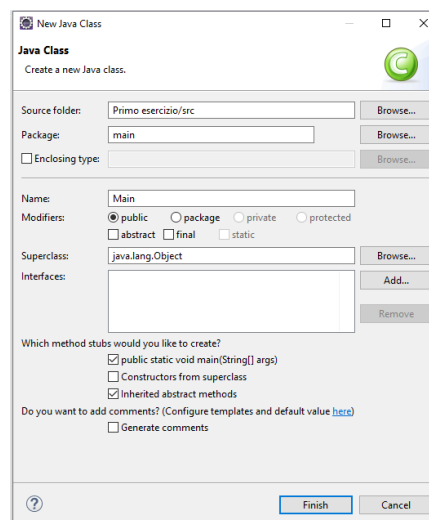
La cartella “Jre system library” contiene le librerie installate, la virtual machine ecc.



Creo un nuovo **package**, ovvero un contenitore. Devo dire dove inserisco il package e poi inserire il nome.

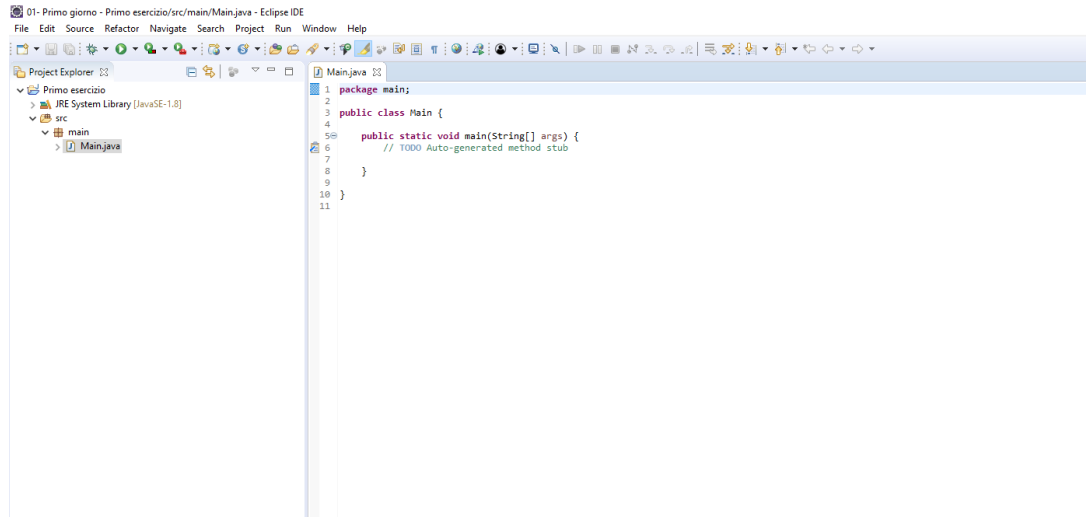
Creo un package di nome “main” dentro alla cartella “src”. Per convenzione il nome dei package è sempre scritto con la lettera iniziale minuscola.

Dentro al package “main” creo una nuova classe:



La classe invece va nominata sempre con la lettera iniziale maiuscola, per convenzione. Per ora non tocco altro. Devo solo spuntare il “Public static void main”.

A questo punto ho creato la classe:



Il package diventa marrone, perché ora contiene una classe.

Una **classe** è un contenitore, in questo caso si tratta di una **classe di avvio**.

“Public static void main args”= è un codice che permette di eseguire questo file. Vuol dire che tutto ciò che sta dentro alle graffe può essere eseguito. Per questo si chiama “classe d’avvio”.

```
1 package main;
2
3 public class Main
4 {
5     public static void main(String[] args)
6     {
7         System.out.println("Ciao mondo!");
8     }
9 }
10
```

```
<terminated> Main [Java Application] C:\Program Files\Java\jre1.8.0_261\bin\javaw.exe (15 ott 2020, 09:52:09)
Ciao mondo!
|
```

In console mi compare il risultato.

## 2. JAVA

## 2.1 Sintassi e cose utili

- Con `//` apro un commento in una sola riga.
- Con `/* */` apro e chiudo un commento lungo più righe.
- `+=` serve a sommare il valore precedente di una variabile a nuove cose.
- `\n` serve per andare a capo.

### 2.1.2 – Operatori matematici

- `==` indica uguaglianza;
- `>=` indica maggiore o uguale;
- `<=` indica minore o uguale;
- `>` indica maggiore;
- `<` indica minore;
- `!=` indica diverso;

Attenzione, `==` vale per i numeri e i boolean ma non per le String. Se bisogna fare una comparazione tra String si possono usare:

- `.equals` per comparazione esatta;
- `.equalsIgnoreCase` per comparazione che non tiene conto di maiuscole o minuscole;

### 2.1.3 – Operatori logici

- `||` sta per “or”, basta che una delle due condizioni sia vera per rendere vera la condizione;
- `&&` sta per “and”, entrambe le condizioni devono essere vere per rendere vera la condizione;

### 2.1.4 – Convertire il case

- `.toLowerCase()` = trasforma tutte le lettere di una parola in minuscolo;
- `.toUpperCase()` = trasforma tutte le lettere di una parola in maiuscolo;

```
if(destinazione.equalsIgnoreCase("oslo"))
    costoBiglietto = 180;
```

In questo modo Java confronta il contenuto della stringa senza badare al modo in cui è stata scritta.

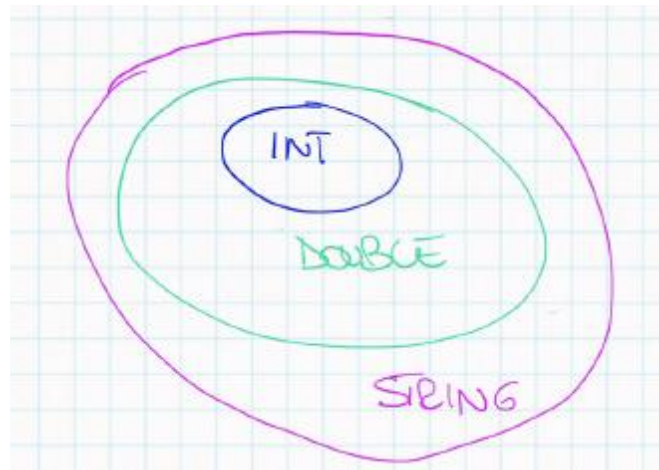
### 2.1.5 – Modulo

```
String numeriPari = "";
String numeriDispari = "";
for(int i = 0; i < numeri.length; i++) {
    if((i % 2) == 0)
        numeriPari += numeri[i];
    else
        numeriDispari += numeri[i];
}
```

Qui abbiamo un esempio di modulo ( $i \% 2$ ). Il %, a differenza della divisione /, restituisce solo il resto di una divisione.

In questo modo posso determinare se un numero è pari (resto = 0) o dispari (resto != 0).

## 2.2 I tipi



In Java abbiamo vari tipi di dati:

- **String** = dati di tipo testuale, è l'insieme più grande di dati, che contiene tutti gli altri;
- **int** = numeri interi;
- **double** = numeri decimali
- **char** = singola lettera.
- **boolean** = è un tipo di variabile che può restituire solo due valori (true o false). Si usa di solito quando ci aspettiamo che l'utente risponda sì/no, vero/falso, positivo/negativo.

```
// Dichiarazione  
boolean verifica;
```

```
// Inizializzazione  
verifica = false;
```

Solitamente un boolean viene sempre inizializzato a "false", in questo modo, se poi viene inserito in un if, sarà la condizione dell'if a determinarne l'eventuale cambiamento di valore.



```
// Dichiarazione
int numero;
boolean verifica;

// Inizializzazione
numero = 10;
verifica = false;

// Calcolo
if(numero > 10)
    verifica = true;

// Output
System.out.println(verifica);
```

```
String esempioStringa = "Ciao";
int esempioInt = 3;
double esempioDouble = 2.3;
char esempioChar = 'i';
```

Una variabile può contenere un solo tipo alla volta. Può essere che due tipi diversi vengano concatenati ma il risultato è una String.

```
String esempioConcat = "la mia età è" + 32 + "anni";
```

### 2.2.1 – Lo scope di una variabile

```
case "marinara" :
    prezzo = 4;
    int numero = 10;
    System.out.println(numero);
break;
case "esplosiva" :
    System.out.println(pizza);
    numero = 12;
    System.out.println(numero);
    prezzo = 10;
```

È lo scopo della variabile. La variabile in questo caso si chiama allo stesso modo ma in realtà è dentro al singolo case. Per averla in ogni case dovrei inizializzarla fuori dallo switch. Lo scope quindi è la visibilità di una variabile, per capire quando la posso utilizzare. Lo scope è il punto in cui quella variabile c'è, esiste e la posso usare.

## 2.3 Le variabili

```
String esempioStringa = "Ciao";  
int esempioInt = 3;  
double esempioDouble = 2.3;  
char esempioChar = 'i';
```

Questi sono esempi di variabili.

Importante è distinguere:

- **Dichiarazione** della variabile = dichiaro una variabile di un certo tipo e con un certo nome, a livello di macchina sto occupando uno spazio nella memoria del computer;
- **Inizializzazione** della variabile = è il momento in cui la variabile viene effettivamente creata, cioè le viene assegnato un valore iniziale. Non ho più solo la scatola ma inserisco qualcosa nella scatola.

```
String esempioStringa;  
  
esempioStringa = "Ciao!";
```

## 2.4 Principi della programmazione

- **Principio di sequenzialità** = Java legge il programma in sequenza, dalla prima all'ultima riga.
- **Metodo di lavoro DICO** = dichiarazione, inizializzazione, calcolo, output.
- **Selezione** = in un sistema comune dobbiamo avere la possibilità di gestire il dato in entrata e valutarlo prima di autenticarlo.
- **Iterazione** = ripetere un'istruzione o un blocco di codice finché una condizione è vera.
- **DRY** = don't repeat yourself.

## 2.5 Convertire string in numeri

- **Integer.parseInt()** = serve a convertire una stringa numerica in integer;
- **Double.parseDouble()** = serve a convertire una stringa numerica in Double.

## 2.6 Lo scanner e gli input dell'utente

Lo **Scanner** è un elemento di Java che permette al programma di stare in ascolto. Può ascoltare la tastiera, un DB, un file, ecc.

Lo Scanner quindi rileva gli input provenienti dall'esterno. Tutti gli input che provengono dall'esterno sono sotto forma di String.

Lo Scanner non è una cosa che Java conosce di default, ogni volta devo farglielo imparare. Bisogna importare una libreria dello Scanner.

```

1 package main;
2
3 import java.util.Scanner;
4
5 public class GeometriaInput
6 {
7     public static void main(String[] args)
8     {
9         // Chiedere all'utente come si chiama
10        // Chiedere all'utente la misura della base e
11        // dell'altezza del rettangolo.
12        // Calcolare e stampare la misura dell'area
13
14        // Dichiarazione
15
16        Scanner tastiera;
17
18    }
19 }
20

```

In questo caso vedo che ho importato il pacchetto corretto.

Alcune librerie vengono importate di default per ogni progetto Java. Altre vanno importate all'occorrenza, lo Scanner è uno degli elementi da importare.

Per ora però l'ho solo dichiarato, bisogna iniziarlo:

```

// Inizializzazione
tastiera = new Scanner(System.in);

```

Questa è la riga per inizializzare lo Scanner, così Java sa che deve mettersi in ascolto dell'input proveniente dalla tastiera dell'utente. Il fatto che la variabile si chiami "tastiera" è parlante ma non è necessario. Una volta eseguita la riga, Java rimane in attesa dell'input dell'utente.

Questa riga serve per gli input da tastiera, se voglio leggere input da DB o altre fonti sarà diverso.

```

// Dichiarazione
Scanner tastiera;
String nomeUtente;
double baseRettangolo, area;
int altezzaRettangolo;
String risposta;

// Inizializzazione
tastiera = new Scanner(System.in);
risposta = "";

System.out.println("Come ti chiami?");
nomeUtente =tastiera.nextLine();

```

Con "tastiera.nextLine()" dico di leggere l'input che l'utente scrive dopo la linea precedente.

```

System.out.println("Ciao " + nomeUtente + "inserisci il valore della base in cm!");
baseRettangolo = Double.parseDouble(tastiera.nextLine());

```

In questo caso faccio inserire un valore numerico, però tutti gli input sono String, perciò avrò bisogno di convertirlo.

```
int nuovoNumero =tastiera.nextInt();  
double nuovoNumero2 =tastiera.nextDouble();  
|
```

Potrei fare anche in questo modo. Queste due righe fanno la stessa cosa del `parseInt` e `parseDouble`, però lo fanno peggio, nel senso che con `parseInt` e `parseDouble` viene presa la stringa e viene convertita.

Con `nextInt` e `nextDouble` invece viene convertita direttamente senza preoccuparsi di cosa ci sia dentro, il che può portare a errori. Quindi meglio usare `parseInt` e `parseDouble`.

```
tastiera.close();|
```

Fondamentale alla fine del programma è inserire questa riga, per indicare a Java che deve smettere di attendere input.

Il canale di immissione dati tra esterno e interno va chiuso perché si consuma una risorsa inutilmente. Se si termina il programma ma la tastiera non è chiusa, potrebbe dare problemi in futuro.

Questo è un esempio di programma completo:

```
public static void main(String[] args)  
{  
    // Chiedere all'utente come si chiama  
    // Chiedere all'utente la misura della base e  
    // dell'altezza del rettangolo.  
    // Calcolare e stampare la misura dell'area  
  
    // Dichiarazione  
  
    Scanner tastiera;  
    String nomeUtente;  
    double baseRettangolo, area;  
    int altezzaRettangolo;  
    String risposta;  
  
    // Inizializzazione  
  
    tastiera = new Scanner(System.in);  
    risposta = "";  
  
    System.out.println("Come ti chiami?");  
  
    nomeUtente =tastiera.nextLine();  
  
    System.out.println("Ciao " + nomeUtente + " inserisci il valore della base in cm!");  
    baseRettangolo = Double.parseDouble(tastiera.nextLine());  
  
    System.out.println("La base del rettangolo è " + baseRettangolo + ". Inserisci il valore dell'altezza in cm!");  
    altezzaRettangolo = Integer.parseInt(tastiera.nextLine());  
  
    System.out.println("L'altezza del rettangolo è " + altezzaRettangolo);  
  
    // Calcolo  
  
    area = baseRettangolo * altezzaRettangolo;  
  
    risposta = "L'area del rettangolo di base " + baseRettangolo + " e altezza " + altezzaRettangolo + " è: " + area;  
  
    System.out.println(risposta);  
  
    // Chiudo ascolto input  
    tastiera.close();  
}
```

## 2.7 Secondo principio della programmazione – la selezione

In un sistema comune dobbiamo avere la possibilità di gestire il dato in entrata e valutarlo prima di autenticarlo, come una password.

```
// If
if(nome.equals("tomas")) {
    risposta = "benvenuto!";
}
```

Ecco un esempio di **if**, abbiamo:

- **Condizione** = la prima parte, se è vera allora si esegue la seconda parte.
- **Codice da eseguire.**

In questo caso si può fare senza le graffe, in questo modo:

```
// If
if(nome.equals("tomas")) |
    risposta = "benvenuto!";
```

Questo perché c'è solo una riga di codice dopo.

Ci sono diversi tipi di **if**, vediamoli:

### 2.7.1 – If piatto

```
if(destinazione.equals("londra"))
    costoBiglietto = 120;
if(destinazione.equals("parigi"))
    costoBiglietto = 110;
if(destinazione.equals("oslo"))
    costoBiglietto = 180;
```

Questo tipo di if è detto **if piatto**. Possono esistere contemporaneamente diverse condizioni e il programma le scorrerà una alla volta per verificarne la veridicità. Se una condizione è vera, allora eseguirà il codice ad essa legato, altrimenti passerà oltre.

Ogni condizione viene verificata indipendentemente dalle altre, anche nel caso in cui solo la prima sia vera.

Bisogna risolvere il però un problema, quello del **case sensitive**. Rimando al capitolo 2.1.1.3.

### 2.7.2 – If else

```

    if(eta >= 18)
        risposta = "maggiorenne";
    else
        risposta = "NO";

    System.out.println(risposta);

```

È un altro esempio di selezione: se la condizione nell'if è vera, allora viene eseguito il codice dell'if. Altrimenti (**else**) viene eseguito il codice dell'else.

Il blocco if else è un blocco unico, un elemento. A ogni if corrisponde un else. Posso avere un if senza else ma non un else senza if.

### 2.7.3 – If annidati

```

    if(eta > 0 && eta < 120)
        if(eta >= 18)
            risposta = "maggiorenne";
        else
            risposta = "minorenne";
    else
        risposta = "Hai un'età inferiore a zero oppure superiore a 120";

```

Questo è un esempio di **if annidato**.

### 2.7.4 – Operatore ternario

```

Scanner tastiera = new Scanner(System.in);
System.out.println("Quanti anni hai?");
int eta = Integer.parseInt(tastiera.nextLine());

```

```

String risposta = eta >= 18 ? "Maggiorenne" : "Minorenne";

```

Un operatore ternario è un modo diverso di scrivere un blocco if-else semplice.

In pratica è come scrivere un blocco if-else in una sola riga:

- String risposta -> significa che dichiaro una variabile di tipo String che si chiama "risposta";
- = -> significa che assegno a risposta il risultato del ternario;
- eta >= 18 -> indica che la condizione corrisponde a if(eta>=18);
- ? -> il punto di domanda indica l'istruzione che il programma esegue nel caso in cui la condizione è vera;
- "Maggiorenne" -> indica il valore che verrà assegnato a risposta se la condizione eta >= 18 è vera;
- "Minorenne" -> indica il valore che verrà attribuito a risposta se la condizione eta >= 18 è falsa.

Si possono creare ternari con più istruzioni ma non è consigliabile. Si possono creare ternari con all'interno altri ternari ma non è consigliabile.

Servono quindi in tutti i casi in cui c'è un blocco di codice if-else che ha una sola istruzione in caso di condizione vera e una sola istruzione in caso di condizione falsa.

```
Scanner tastiera = new Scanner(System.in);
System.out.println("Quanti anni hai?");
int eta = Integer.parseInt(tastiera.nextLine());

//I boolean sono perfetti per i ternari perché possono assegnare
//Di conseguenza o sono true o sono false!
boolean verificaEta = eta >= 18 ? true : false;

String risposta = verificaEta ? "Maggiorenne" : "Minorenne";
```

I boolean sono perfetti per i ternari perché possono assegnare un solo valore alle variabili. Di conseguenza o sono "true" o sono "false".

## 2.7.5 – Switch

Lo **switch** è un esempio di selezione che ricorda molto gli if piatti ma si comporta in maniera leggermente diversa e introduce nuove possibilità.

```
switch(pizza) {
case "margherita":
    // qui scrivo il codice da eseguire nel caso in cui
    // il valore di pizza sia margherita
    prezzo = 5;
    break;
case "marinara":
    prezzo = 4;
    break;
case "esplosiva":
    prezzo = 10;
    break;
default:
    prezzo = 15;
    break;
}
```

Lo switch richiede un **parametro**, ossia un valore da inserire all'interno delle parentesi tonde.

Attenzione: lo switch fa una comparazione con quel parametro ma non può eseguire dei calcoli. Non si può scrivere switch(3+1), lo switch non calcola.

Switch(pizza) significa che lo switch compara il valore di pizza con quelli presenti nei vari **case**. Quindi nel caso di "margherita" è come se scrivesse "pizza.equals("margherita")".

Importante è che alla fine di ogni case ci sia un **break**. Ogni case funziona come un programma a sé stante. I case non comunicano tra loro e non possono passarsi i valori o le variabili a un case all'altro.

Va inserito per forza il break perché termina il case e fa uscire il programma dallo switch. Se invece non si mette il break, si va fino alla fine dello switch.

Non è possibile inserire "equalsIgnoreCase()" dentro allo switch. Quindi si risolve con "toLowerCase()" o "toUpperCase()". Vedere capitolo 2.1.1.3.

```

switch(pizza.toLowerCase()) {
case "margherita":
    // qui scrivo il codice da eseguire nel caso in cui
    // il valore di pizza sia margherita
    prezzo = 5;
    break;
case "marinara":
    prezzo = 4;
    break;
case "esplosiva":
    prezzo = 10;
    break;
default:
    prezzo = 15;
    break;
}

```

Conviene metterlo direttamente al parametro, così non modifica il valore della variabile.

## 2.8 Terzo principio della programmazione – iterazione

Iterare significa ripetere un'istruzione o un blocco di codice finché una condizione è vera.

Le iterazioni in Java prendono anche il nome di **cicli**.

### 2.8.1 – Ciclo While

```

while(numero > 0 ) {
    System.out.println(numero);
    numero --;
}

```

Si legge: finché il valore di numero è maggiore o uguale a zero, esegui le istruzioni tra le parentesi graffe. La parte tra le parentesi tonde prende il nome di **condizione di iterazione**.

Questo codice viene eseguito se la condizione di iterazione è vera, cioè se il valore di numero è maggiore o uguale a zero.



Se la condizione è vera, il programma stampa il valore di numero. Per evitare un ciclo infinito, è necessario che il valore di numero diminuisca o comunque cambi. In questo caso diciamo che il programma deve diminuire di uno il valore di numero, dopo averlo stampato.

“Numero --” è uguale a scrivere “numero -1”. Lo stesso vale con il “++”;

il ciclo **while** può ripetersi **da 0 volte a infinite volte**.

### 2.8.2 – Ciclo do while

Il ciclo **while** cicla da zero a infinite volte e verifica subito che la condizione di iterazione sia vera. Se è vera, entra nel ciclo ed esegue il codice, altrimenti no.

```
int numero = 10;

while(numero > 0) {
    System.out.println(numero);
    numero --;
}
```

Diverso è il caso del ciclo **do while**:

```
do {
    System.out.println(numero);
    numero --;
}
while(numero > 10);
```

La differenza è che il ciclo do-while cicla almeno una volta per il principio di **sequenzialità**.

In pratica il ciclo do-while esegue il codice prima di verificare la condizione di iterazione. Se poi vede che la condizione è vera, ripete il ciclo, altrimenti si ferma.

### 2.8.3 – Ciclo for

```
public static void main(String[] args)
{
    int[] numeri = new int[5];

    numeri[0] = 10;
    numeri[1] = 10;
    numeri[2] = 10;
    numeri[3] = 10;
    numeri[4] = 10;

    int somma = 0;

    for(int i = 0; i < numeri.length; i++) {
        somma += numeri[i];
    }

    System.out.println(somma);
}
```

Questo è un esempio di ciclo for:

- `int i = 0` -> dichiaro e inizializzo una variabile `i`;
- `i < numeri.length` -> condizione di ripetizione, ripeti finché il valore di `i` è minore della grandezza del vettore;
- `i++` -> incremento, a ogni ciclo aumento di una unità il valore dell'indice.

Il ciclo `for` è specifico per i vettori. Gestisco il contatore, condizione e incremento tutti in solo posto. Inoltre il contatore ha come scope solo il `for`, a differenza del `while`.

```
Scanner tastiera = new Scanner(System.in);
System.out.println("Quanti numeri vuoi considerare?");
int grandezza = Integer.parseInt(tastiera.nextLine());

int[] numeri = new int[grandezza];

for(int i = 0; i < numeri.length; i++)
{
    numeri[i] = i;
}

String numeriPari = "";
String numeriDispari = "";
for(int i = 0; i < numeri.length; i++) {
    if((i % 2) == 0)
        numeriPari += numeri[i];
    else
        numeriDispari += numeri[i];
}

System.out.println(numeriPari + "\n" + numeriDispari);
tastiera.close();
```

Questo è un esempio di come possiamo usare il ciclo `for` per inserire i valori dentro a un vettore.

#### 2.8.4 – Differenze tra diversi cicli

- `Do – while` -> l'unico che ha la condizione di iterazione alla fine, quindi cicla almeno una volta. Utile quando non so quante volte dovrò ciclare, ad esempio quando devo chiedere all'utente di inserire un numero ignoto di libri. Cicla quindi da 1 a infinite volte.
- `While` -> utile quando devo leggere le righe di un file con `hasNextLine()`. Cicla da 0 (nel caso in cui la condizione non venga mai rispettata) a infinite volte.
- `For` -> il ciclo che si usa soprattutto con i vettori. Ha un indice con scope locale e non globale. Indice, condizione e incremento sono raggruppati nello stesso posto. Cicla da 0 a infinite volte. Posso volendo mettere una condizione che non tenga conto dell'indice, ma perderebbe di significato e lo renderebbe più simile a un `while`.

#### 2.8.5 – Ciclo `for each`

Ultimo tipo di iterazione. Il ciclo **for** permette di ripetere un'istruzione o un insieme di istruzioni per un numero definito di volte e al suo interno sfrutta un indice. Se ad esempio devo ripetere un'istruzione solo sugli oggetti che si trovano in posizione pari, è utile il ciclo for. Ma se ad esempio devo stampare e ripeto la stessa operazione indipendentemente dalla posizione per ogni oggetto dell'insieme, posso utilizzare il **ciclo for each**.

```
public String stampaProdottoForEach()
{
    String ris = "";
    //for(Prodotto nomeOggetto : prodotti) si legge:
    for(Prodotto nomeOggetto : prodotti)
        ris += nomeOggetto.toString() + "\n-----\n";
    return ris;
}
```

“for(Prodotto nomeOggett : prodotti)” si legge:

- For() -> indica iterazione;
- Prodotto -> indica il tipo di prodotto da ciclare;
- nomeOggetto -> è il nome che viene dato alla variabile. Possiamo dare il nome che vogliamo;
- : -> indicano “all’interno di”;
- prodotti -> indica l’insieme nel quale si sviluppa il ciclo.

Cicla per ogni oggetto di tipo “Prodotto” all’interno di “prodotti”.

Mentre il ciclo for cicla per indici, il ciclo for each **cicla per oggetti**.

```
//Immaginiamo di avere un vettore di String e di volerlo ciclare
String[] nomi = {"Nome1", "Nome2", "Nome3"};

for(int i = 0; i < nomi.length; i++)
    System.out.println(i + " " + nomi[i]);

for(String s : nomi)
    System.out.println(s);
```

Siccome il for each cicla per oggetti, non posso utilizzare i tipi primitivi. Non posso usare int ma devo usare Integer, non posso usare double ma devo usare Double.

## 2.9 Leggere file

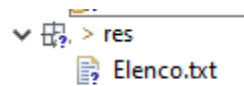
```
public class Lettura_File_01 {
    public static void main(String[] args) throws FileNotFoundException
    {
        String percorsoFile = "";

        Scanner file = new Scanner(new File(percorsoFile));

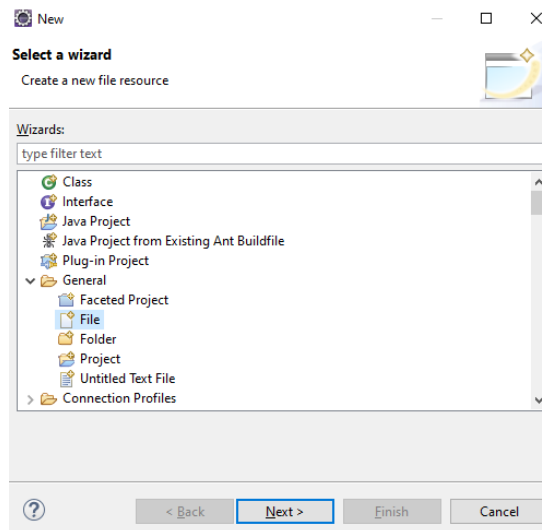
    }
}
```

Questo è il modo in cui si legge un file, dicendo a Java che se non trova il file deve crashare.

Per prima cosa si crea un package “res” con dentro un file.txt.



Per creare un nuovo file bisogna andare in “new” e selezionare in “general” -> “file”:



Il file sarà così:

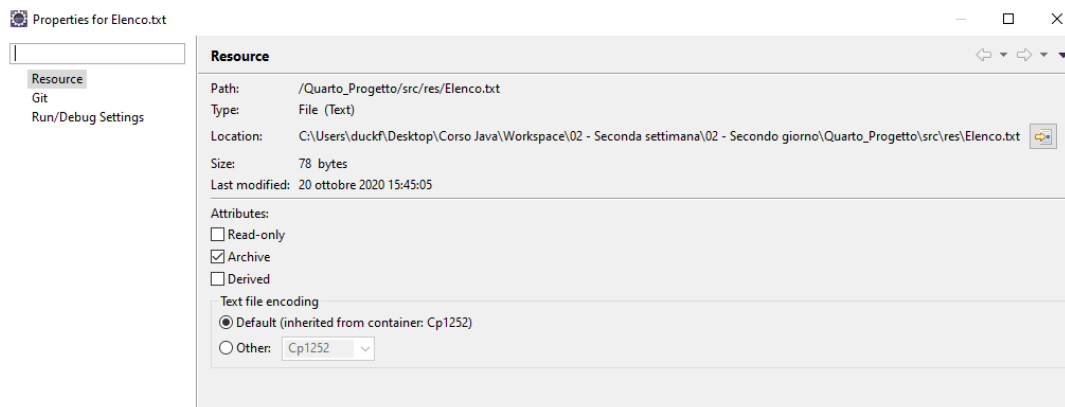
```
1 Il signore degli anelli
2 Alien
3 Atto di forza
4 Ghostbusters
5 Ritorno al futuro
```

```
Scanner file = new Scanner(new File(percorsoFile));
```

Con questa riga importiamo lo Scanner e il File.

La variabile “percorsoFile” indica il path del file che vogliamo leggere. Come facciamo a recuperare il path?

Vado sul file che mi interessa, poi faccio tasto destro e clicco su “properties”. Lì devo copiare la “location”:



Dopo aver copiato “location”, incollo in “percorsoFile”, ottenendo questo:

```
String percorsoFile = "C:\\Users\\duckf\\Desktop\\Corso Java\\Workspace\\02 - Seco
```

Cosa importante: devo copiare direttamente all'interno degli apici, perché in questo modo mette in automatico 2 backslash. Se invece copio fuori dagli apici e poi ce li metto attorno, ne fa solo uno, cosa che può dare problemi perché si confonde con l'a capo (\n).

Una volta aperto il file, questo è ciò che facciamo per leggerlo e stampare le righe in console:

```
String percorsoFile = "C:\\Users\\duckf\\Desktop\\Corsi\\  
Scanner file = new Scanner(new File(percorsoFile));  
  
String riepilogo = "";  
  
while(file.hasNextLine()) {  
    riepilogo += "- " + file.nextLine() + "\n";  
}  
  
System.out.println(riepilogo);  
file.close();
```

Con un ciclo while leggo una riga dopo l'altra e le salvo in una variabile "riepilogo", in seguito le stampo, una volta finite le linee.

Cosa importante è la condizione del while: **file.hasNextLine()**.

## 2.10 – I vettori

Un vettore è un insieme ordinato di elementi dello stesso tipo (un vettore di numeri interi può contenere solo numeri interi per esempio).

Un vettore è sempre di tipo **vettore**, ciò che sta al suo interno può essere String, double, int, ecc. ma il vettore sarà sempre solo di tipo vettore.

```
// Dichiaro il vettore  
String[] nomi;  
  
// Creo il vettore (inizializzazione)  
nomi = new String[3];
```

Questo è un vettore:

- "String" indica il tipo di variabile che può contenere il vettore "nomi";
- [ ] è il simbolo che indica il vettore, è di tipo vettore;
- "nomi" è il nome del vettore di tipo vettore che può contenere solo elementi di tipo String.

La prima riga è la dichiarazione del vettore, poi abbiamo la creazione (o inizializzazione) che avviene con la seconda riga:

- "nomi" è il nome del vettore;
- "new" serve per creare il vettore;

- “String[3]” indica il contenuto (String) e la dimensione [3]. [3] indica la dimensione del vettore, ogni vettore è un **insieme finito**. Per poter creare un vettore è sempre necessario specificare la dimensione.

Un vettore può essere vuoto, contenere alcuni elementi, oppure essere pieno. Non è detto che il suo contenuto debba essere sempre pieno o non vuoto.

Un vettore è un insieme **ordinato** di elementi dello stesso tipo. Per ordinare gli elementi al suo interno, ho bisogno di un **indice**. Ogni elemento all’interno del vettore sarà legato al proprio indice. Il valore dell’indice parte da **zero**. Se l’indice parte da zero e il vettore può contenere al massimo 3 elementi, il primo elemento ha indice 0.

Dire che l’ultimo elemento in questo caso ha indice 3 genera un errore in Java. Se a Java chiedete l’elemento con indice 3 è come se steste chiedendo la terza caramella a un sacchetto che ne contiene solo 2.

```
nomi[0] = "Primo nome";
```

- nomi[0] si legge: la posizione zero del vettore nomi di tipo vettore contenente String.
- = “Primo nome” indica il valore della posizione zero del vettore “nomi”.

```
nomi[0] = "Primo nome";
```

```
nomi[1] = "Secondo nome";
```

```
nomi[2] = "Terzo nome";
```

```
System.out.println(nomi[0]);
```

Per stampare il contenuto di una posizione del vettore devo chiamare il numero del vettore e la posizione a cui corrisponde il contenuto che mi interessa.

Scrivere “nomi[0]” oppure “primo nome” è la stessa cosa. Perché “primo nome” è il contenuto di tipo String corrispondente alla posizione zero del vettore (nomi[0]). Quindi se io chiedo a java di stampare nomi[0] lui stamperà il contenuto a quella posizione.

Se dovessi associare un valore a nomi[3] e provare a stamparlo Java genererebbe un errore chiamato:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
    at main.Vettori_01.main(Vettori_01.java:19)
```

Che significa che sto chiedendo qualcosa oltre i limiti consentiti dal vettore. Sto chiedendo il quarto elemento in un vettore con solo 3 elementi.

Una volta specificata la dimensione del vettore, resta tale.

```
String nomi[];
nomi = new String[3];

nomi[0] = "Nome 1";
nomi[1] = "Nome 2";
nomi[2] = "Nome 3";

System.out.println(nomi[0]);

nomi[0] = "Pippo";

System.out.println(nomi[0]);
```

In questo modo riassegno il valore a una delle posizioni del vettore.

```
/ prezzi.length
```

Questo è molto utile perché restituisce la grandezza di un vettore, ossia la sua dimensione.

```
int[] numeri;

numeri = new int[] {1,2,3,4,5,6,7,8,9,10};
```

Questo è un altro modo per scrivere i vettori, definendo da subito il contenuto. In questo modo, Java capisce da solo la grandezza del vettore. Invece di scrivere la grandezza, scriviamo il contenuto tra le graffe.

La grandezza del vettore viene determinata dagli elementi tra le { }, che in questo caso sono 10.

Inoltre, in questo modo i 10 elementi sono già stati ordinati all'interno del vettore in base a come sono stati presentati.

```
int[] numeri;

String numeriPari = "";
String numeriDispari = "";

numeri = new int[] {1,2,3,4,5,6,7,8,9,10};

for(int i = 0; i < numeri.length; i++) {
    if((i % 2) == 0)
        numeriPari += numeri[i];
    else
        numeriDispari += numeri[i];
}

System.out.println(numeriPari + "\n" + numeriDispari);
```

Questo è un esempio.

Altra cosa importante è che posso inserire i valori direttamente dal for:

```

Scanner tastiera = new Scanner(System.in);
System.out.println("Quanti numeri vuoi considerare?");
int grandezza = Integer.parseInt(tastiera.nextLine());

int[] numeri = new int[grandezza];

for(int i = 0; i < numeri.length; i++)
{
    numeri[i] = i;
}

String numeriPari = "";
String numeriDispari = "";
for(int i = 0; i < numeri.length; i++) {
    if((i % 2) == 0)
        numeriPari += numeri[i];
    else
        numeriDispari += numeri[i];
}

System.out.println(numeriPari + "\n" + numeriDispari);
tastiera.close();

```

Importante in questo caso è il modulo, "%", che serve a restituire il resto di una divisione. È l'unico modo che si ha per verificare se un numero è pari o dispari.

### 2.10.1 Split

```

String[] parola = {"c", "i", "a", "o"};

String parolaa = "ciao";

String[] prova = parolaa.split("");

for(int i = 0; i < parola.length; i++) {
    System.out.println(prova[i]);
}

String stampa = "";
String stampaContrario = "";

for(int i = 0; i < parola.length; i++) {
    stampa += parola[i];
}

System.out.println(stampa);

for(int i = (parola.length - 1); i >= 0; i--) {
    stampaContrario += parola[i];
}

System.out.println(stampaContrario);

```

Possiamo vedere le String come vettori di char, quindi possiamo giocare coi char.

```

String parola = "supercalifragilistichespiralidoso";
String[] vettore = parola.split("");

```

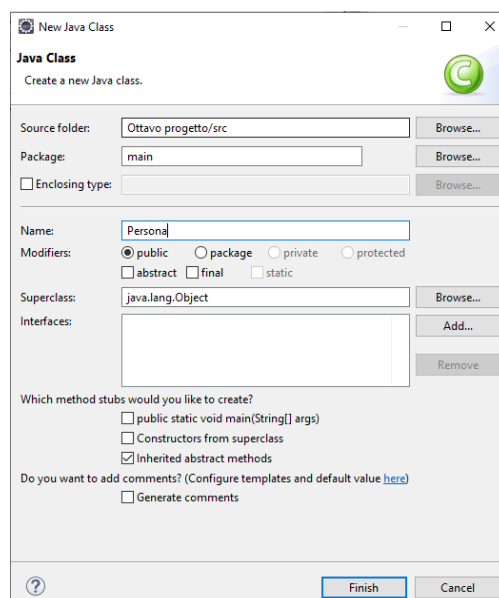


**Split()** è un metodo per le String che permette di creare un vettore da una String.

```
//Double.MIN_VALUE indica il valore più basso che un double  
//può assumere  
//Si, funziona anche per il max -> |  
double massimo = Double.MIN_VALUE;
```

## 2.11 INTRODUZIONE AGLI OGGETTI

Per ora abbiamo sempre lavorato col main, che è la classe di avvio. Quelle che vediamo ora solo le classi **modello**.



Creo una nuova classe, questa volta non è una classe di avvio, quindi non dobbiamo flaggare public static void.

```

1 package main;
2
3 public class Persona {
4
5 }
6

```

Questo è l'aspetto di una classe **modello**, rispetto alla classe di avvio cambia il fatto che non ha la stringa di avvio. A differenza della classe di avvio, non ha il main. La classe modello serve come istruzioni per la creazione di oggetti di quella classe.

La nostra classe modello Persona, servirà per creare oggetti di tipo Persona.

```

1 package main;
2
3 public class Persona {
4     |
5     public String nome;
6     public int eta;
7
8 }
9

```

“nome” ed “eta” sono due variabili inserite all'interno di una classe modello.

È corretto dire che ogni persona ha un nome e un'età? Sì, di conseguenza possiamo dire che tutti gli oggetti di tipo Persona avranno come **proprietà** sia nome, sia eta.

È corretto dire che ogni Persona ha lo stesso valore di nome ed eta? No. Ogni oggetto di classe Persona avrà come proprietà “nome” ed “eta”, ma non tutti gli oggetti avranno necessariamente gli stessi valori di “nome” ed “eta”.

Ogni oggetto di tipo Persona avrà le stesse proprietà dell'oggetto ma potrebbe avere dei valori diversi per le singole proprietà. Tutte le Persone hanno “nome” ed “eta” come proprietà. Non tutte le Persona si chiamano ad esempio Luca.

**L'oggetto è la concretizzazione della classe. La classe è l'astrazione.** Una classe modello è l'insieme delle proprietà comuni a ogni oggetti di quella classe.

```

1 package main;
2
3 public class MainPersona
4 {
5     |
6     public static void main(String[] args)
7     {
8         Persona p;
9
10    }
11 }

```

Qua ho creato una classe d'avvio. Dichiaro una variabile “p” di tipo Persona. “p” indica un oggetto della classe Persona. Il tipo di un oggetto viene definito in base alla classe a cui appartiene. L'oggetto quindi è la concretizzazione della classe a cui appartiene. L'oggetto è il caso concreto.

Adesso però va inizializzato.

```
Persona p;  
  
p = new Persona();
```

Così si inizializza l'oggetto.

"p" è un oggetto (o una variabile, o una scatola) che contiene tutte le proprietà dell'oggetto, comuni a tutti gli oggetti di tipo Persona.

Un oggetto viene inizializzato quando viene scritto **new**.

```
Persona p;  
  
p = new Persona();  
  
p.nome = "Tizio";  
System.out.println(p.nome);
```

Qui sto dando un nome a "p". Non è detto che tutti gli oggetti abbiano ogni proprietà della classe, possono anche essere vuote. Ad esempio al momento "p" non ha un valore di età. Non vuol dire però che quella proprietà non esista. Vuol dire che è null o 0.

All'interno della classe si può inserire tutto ciò che è riferito all'oggetto. Ci sono ad esempio i **metodi**, che sono dei programmi a sé:

```
public class Libro  
{  
    public String autore;  
    public String titolo;  
    public String genere;  
    public Double prezzo;  
  
    public String stampaLibro() {  
        String ris = "- " + autore + ", " + titolo + ", " + genere + ", " + prezzo + "€";  
        return ris;  
    }  
}
```

"stampaLibro" è un metodo. Quello che succede all'interno di Libro, rimane oscuro al main. Il main non sa cosa succede all'interno del metodo "stampaLibro" e non importa.

"public String stampaLibro()" viene definita **firma del metodo**, cosa contiene?

- **Public** = il livello di visibilità;
- **String** = il tipo di ritorno;
- **stampaLibro** = il nome del metodo;

Tutti i metodi devono sempre indicare il tipo di ritorno, ogni metodo ritorna qualcosa. "stampaLibro()" è un metodo che ritorna un valore calcolato di tipo String e come tale può e verrà considerato ovunque verrà utilizzato.

```
Viaggio[] viaggi;
```

Questo è un **vettore di oggetti**. Ogni oggetto Viaggio ha le stesse proprietà e ogni oggetto appartiene allo stesso **tipo**. Siccome un vettore è un insieme ordinato e finito di elementi dello stesso **tipo**, è possibile inserire gli oggetti all'interno dei vettori creando **vettori di oggetti**.

```
System.out.println("Quanti viaggi vuoi inserire?");
int nViaggi = Integer.parseInt(tastiera.nextLine());

Viaggio[] viaggi = new Viaggio[nViaggi];
Viaggio v;
int indice = 0;

//String riepilogo = "";
//int counter = 1;

double sommaPrezzi = 0;

while(indice < nViaggi) {
    v = new Viaggio();

    System.out.println("Inserisci la destinazione:");
    v.destinazione = tastiera.nextLine();
    System.out.println("Inserisci il prezzo base:");
    v.prezzoBase = Double.parseDouble(tastiera.nextLine());
    System.out.println("Inserisci il periodo migliore:");
    v.periodoMigliore = tastiera.nextLine();

    viaggi[indice] = v;

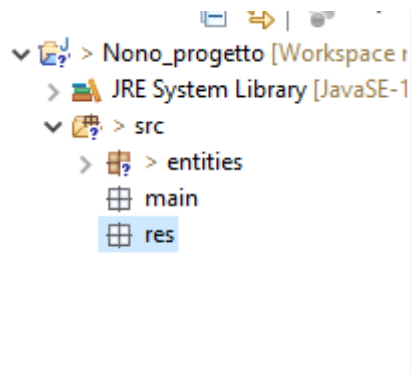
    indice ++;
}

String elenco = "";

for(int i = 0; i < viaggi.length; i++) {
    elenco += (i+1) + "- " + viaggi[i].stampaViaggio() + "\n";
    sommaPrezzi += viaggi[i].prezzoBase;
}
```

Qui sto inserendo i dati in un vettore di oggetti, un vettore di Viaggi.

### 2.11.1 Struttura del progetto



Da ora in poi struttureremo così i progetti:

- **Entities** -> ci sono le classi modello;
- **Main** -> ci sono le classi d'avvio;
- **Res** -> ci sono i file che dobbiamo leggere.

### 2.11.2 Visibilità delle proprietà e i metodi

```

package entities;

public class Studente
{
    // Proprietà dell'oggetto: le proprietà sono dati
    // ai quali vengono assegnati dei valori
    String nome;
    int eta;

    //Metodo: restituisce un valore di tipo string
    //I metodi sono dati che vengono calcolati
    // Ciò che differenzia proprietà da metodi sono le parentesi tonde
    public String scheda() {
        return nome + " " + eta;
    }
}

```

Qui abbiamo una classe modello, con proprietà e metodo.

Con la struttura a cartelle vista prima, bisogna importare le classi che sono in cartelle diverse, in questo modo:

```

1 package main;
2
3 import entities.Studente;
4
5 public class MainScuola_01
6 {
7     public static void main(String[] args)
8     {
9         Studente s;
10    }
11 }
12

```

Le proprietà e i metodi possono avere diversi livelli di visibilità:

- **Public** = visibile in tutto il progetto;
- **Package** = vuol dire che è visibile solo all'interno del package;
- **Private** = vuol dire che è visibile solo all'interno della classe.

Se non viene specificato nulla, il livello di visibilità è da considerarsi **package**.

```

public String nome;
public int eta;

```

Qui mettiamo "public" perché siano visibili in tutto il progetto.

I livelli di visibilità sono uno degli aspetti che rientrano in un principio della programmazione a oggetti detto **incapsulamento**, significa avere il controllo di quali parti del codice vengono viste dall'esterno e quali no.

Da ora stiamo parlando di programmazione a oggetti, perché non stiamo più programmando in modo classico ma a oggetti.

Quando scrivo un metodo devo sapere il tipo di ritorno, il nome e anche il livello di visibilità:

```

public String scheda() {
    return nome + " " + eta;
}

```

Un'altra definizione di metodo è: **blocchi che permettono di fare operazioni o assegnare valori alle proprietà.**

## 2.12 PRINCIPI DELLA PRORAMMAZIONE A OGGETTI

1. **Principio dell'incapsulamento** = significa avere il controllo di quali parti del codice vengono viste dall'esterno e quali no.
2. **Ereditarietà** = indica il legame tra classe e padre e classe figlia. Una classe padre può avere n figli, una classe figlia deve per forza avere uno e un solo padre. Non esiste ereditarietà multipla.
3. **Polimorfismo dell'oggetto**

## 2.13 ArrayList

Abbiamo visto che un vettore è un insieme ordinato e finito di elementi dello stesso tipo. Abbiamo visto che un vettore può contenere String, int, double, ecc e perfino oggetti.

Un vettore è ordinato perché ogni elemento al suo interno è identificato in modo univoco da un indice. Un vettore è finito perché è limitato alla sua dimensione.

Sarebbe più utile in certi casi lavorare con un insieme che si adatta al contenuto e che quindi aumenta o diminuisce la dimensione a seconda degli elementi al suo interno.

Per ottimizzare questo tipo di ragionamento possiamo ricorrere agli **ArrayList**, degli insiemi **ordinati** di elementi dello **stesso tipo**. Che differenza c'è rispetto ai vettori? Che l'ArrayList non è finito, è una specie di vettore dinamico che adatta la sua dimensione al contenuto.

**Un ArrayList è un insieme ordinato di elementi dello stesso tipo. Ordinato ma non finito.**

```
Scanner tastiera = new Scanner(System.in);
String risposta;
String titolo;
do {
    System.out.println("Inserisci un titolo");
    titolo = tastiera.nextLine();

    System.out.println("Vuoi inserire un nuovo titolo?");
    risposta = tastiera.nextLine();

} while(risposta.equalsIgnoreCase("no"));
```

Vediamo che in questo caso sarebbe utile avere una dimensione indefinita, proviamo quindi a creare un ArrayList.

```
ArrayList<String> elenco;
```

Quando dichiaro un ArrayList devo mettere tra le parenti uncinate il tipo di elemento che andrò a inserire.

Inoltre devo importarlo:

```
import java.util.ArrayList;
```

Lo devo importare perché sto creando un oggetto e ha una classe da importare. Dopo averlo dichiarato, lo creo:

```
elenco = new ArrayList<String>();
```

Se si volesse conoscere la dimensione dell'ArrayList, non si potrebbe utilizzare ".length" come per i vettori ma si dovrebbe ricorrere a ".size()".

```
System.out.println("Dimensione elenco: " + elenco.size());
```

"size()" è un metodo dell'oggetto di tipo ArrayList.

Per inserire un elemento all'interno dell'ArrayList, bisogna utilizzare ".add()", un altro metodo dell'oggetto di classe ArrayList.

```
elenco.add(titolo);
```

In questo modo si aggiunge un elemento all'interno dell'elenco. "(titolo)" indica l'elemento da aggiungere all'interno dell'elenco. Titolo in questo caso prende il nome di **parametro**. ".add()" è un primo esempio di metodo con passaggio di parametri o **metodo parametrizzato**.

A questo punto, dopo aver riempito l'ArrayList, lo ciclo per stamparne il contenuto:

```
String stampa = "";

for(int i = 0; i < elenco.size(); i++) {
    stampa += (i+1) + "- " + elenco.get(i) + "\n";
}

System.out.println(stampa);
```

Non cambia molto rispetto ai vettori, essenzialmente 2 cose:

- Al posto di ".length" si usa ".size()";
- Al posto di "elenco[i]" si usa "elenco.get(i)".

Il risultato è lo stesso.

Il programma quindi risulterà così:

```
.....
ArrayList<String> elenco;

elenco = new ArrayList<String>();
System.out.println("Dimensione elenco: " + elenco.size());

Scanner tastiera = new Scanner(System.in);
String risposta;
String titolo;
do {
    System.out.println("Inserisci un titolo");
    titolo = tastiera.nextLine();

    elenco.add(titolo);
    System.out.println("Dimensione elenco: " + elenco.size());
    System.out.println("Vuoi inserire un nuovo titolo?");
    risposta = tastiera.nextLine();
} while(!risposta.equalsIgnoreCase("no"));

String stampa = "";

for(int i = 0; i < elenco.size(); i++) {
    stampa += (i+1) + "- " + elenco.get(i) + "\n";
}

System.out.println(stampa);
```

Per rimuovere un elemento posso fare “.remove()”.

```
//Dopo aver stampato una volta l'elenco delle parole, decido di rimuovere
//la parola con indice 1 - Parola 2
parole.remove(1);
```

### 2.13.1 ArrayList e tipi boxati

Finora abbiamo sempre usato “int” come variabile per i numeri interi. “int” indica un **tipo primitivo**, come char, double e boolean. All’interno degli ArrayList tuttavia non è possibile inserire tipi primitivi ma tipi detti **boxati**. Il **tipo boxato** di “int” è “Integer”, che abbiamo già visto. Allo stesso modo, se volessimo inserire un double, dovremmo farlo per il suo tipo boxato.

**Un tipo primitivo è un tipo predefinito di Java.**

**Un tipo boxato è composto da tipi primitivi.**

Per capire bene la differenza, pensare a char (tipo primitivo) e a String (tipo boxato).

```
ArrayList<String> parole = new ArrayList();
ArrayList<Integer> numeri = new ArrayList();
```

### 2.14 Costruttori

Quando creiamo una classe modello, Java dà per scontato che con quella classe andremo a creare degli oggetti.

A tal proposito, Java regala un metodo implicito che si chiama **costruttore**. Il costruttore è un metodo particolare, grazie al quale è possibile **costruire / creare** degli oggetti del tipo della classe.

```
public class Libro
{
    public String titolo;
    public String autore;
    public int prezzo;

    public String scheda() {
        return titolo + " " + autore + " " + prezzo + "€";
    }
}

public Libro(){};
```

Questo è un costruttore. Il costruttore va necessariamente scritto in modo uguale al nome della classe. se la classe si chiama Libro, il costruttore si chiamerà Libro.

```
l = new Libro();
```



In realtà il costruttore lo abbiamo sempre usato qui, nella creazione di un oggetto di un certo tipo, perché invochiamo il metodo “Libro()”, ovvero il costruttore.

In una classe possono esserci infiniti costruttori.

```
public Libro()  
{  
    |  
};
```

Questo è il costruttore vuoto che ci regala Java automaticamente. Se noi facciamo un nuovo costruttore, il costruttore vuoto automatico si disattiva.

Il costruttore deve sempre essere **public**, non ha senso farlo private, perché non potrei chiamarlo al di fuori della classe.

Come funziona il costruttore?

```
public Libro(String titolo, String autore, int prezzo)  
{  
    |  
};
```

Creiamo un nuovo costruttore. Tra le parentesi tonde inseriamo i **parametri** che il metodo riceverà. Non mi interessa da dove arriveranno (utente, file, ecc) ma è **fondamentale l'ordine** in cui vengono scritti. Se dovessero arrivare le variabili in ordine diverso, il costruttore non le accetterebbe.

All'interno del costruttore cosa succede?

```
public Libro(String titolo, String autore, int prezzo)  
{  
    this.titolo = titolo;  
    this.autore = autore;  
    this.prezzo = prezzo;  
};
```

Vengono associati i parametri che arrivano dall'esterno con le proprietà dell'oggetto.

Con **this** faccio capire che si tratta delle proprietà dell'oggetto. Per identificare in modo univoco le proprietà dell'oggetto dai parametri nel caso in cui abbiano lo stesso nome, si usa **this**.

“**this.titolo**” indica la proprietà dell'oggetto. “Titolo” indica invece il parametro in ingresso.

Se la proprietà e il parametro si chiamano in modi diversi non è necessario usare “this” per differenziarli.

```
|  
l = new Libro();
```

A questo punto sul main mi dà errore perché nella dichiarazione di prima non sono contenuti i parametri, devo cambiarla perché il costruttore non è più quello base. Da oggi in poi il costruttore vuoto è meglio non usarlo.

```
l = new Libro(parametroAutore, parametroAutore, parametroPrezzo);
```

Così è corretto.

```
l.titolo = parametroTitolo;
l.autore = parametroAutore;
l.prezzo = parametroPrezzo;
```

Questa parte non ci serve più. in questo modo non servono più i nomi delle proprietà dell'oggetto. Nel modo nuovo il Main non ha più accesso alle proprietà dell'oggetto. Il main passa al costruttore tre parametri, poi il costruttore fa l'associazione.

```
Libro l1 = new Libro(parametroTitolo, parametroPrezzo);
System.out.println(l1.scheda());
```

In questo caso ho definito un altro costruttore. Posso averne infiniti, Java riconosce automaticamente dai parametri il costruttore giusto.

## 2.15 Extends

Siccome a livello logico ogni Studente è una Persona, con Persona condivide una serie di proprietà e volendo anche qualche metodo. "Studente extends Persona" significa che Studente è figlio di Persona.

Studente è anche Persona ma non è detto che tutti gli oggetti di tipo Persona siano anche Studente.

```
public class Persona {
    public String nome;
    public String cognome;
    public String sesso;
    public int eta;

    public Persona(String nome, String cognome, String sesso, Integer eta) {
        this.nome = nome;
        this.cognome = cognome;
        this.sesso = sesso;
        this.eta = eta;
    }

    public String scheda() {
        return nome + " " + cognome + " " + sesso + " " + eta;
    }
}
```

**Extends** è la parola chiave di uno dei principi della programmazione a oggetti, chiamato **ereditarietà**.

Ereditarietà è un principio della programmazione a oggetti per il quale andiamo a definire un rapporto di parentela 1 a n. una classe padre può avere n figli, una classe figlia può avere solo un padre. La classe figlia non eredita tutto ciò che c'è nella classe padre, se qualcosa è private non lo eredita.

Per poter creare una classe figlia, devo esplicitare all'interno della classe figlia il costruttore della classe padre.

```
public class Studente extends Persona {
    public String nome;
```

Ereditarietà indica il legame tra classe e padre e classe figlia. Una classe padre può avere n figli, una classe figlia deve per forza avere uno e un solo padre. Non esiste ereditarietà multipla.

C'è anche il livello di visibilità **protected**, che permette di vedere proprietà o metodi solo nelle classi che si trovano nello stesso package (come package) oppure alla classi **figlie**.

Una classe figlia può vedere le proprietà protected del padre indipendentemente dal package in cui si trova.

```
public class Persona {
    protected String nome;
    protected String cognome;
    protected String sesso;
    protected int eta;

    public Persona(String nome, String cognome, String sesso, Integer eta) {
        this.nome = nome;
        this.cognome = cognome;
        this.sesso = sesso;
        this.eta = eta;
    }
}
```

```
public class Studente extends Persona
{
    public int classe;
    public double mediaIta;
    public double mediaMate;
    public double mediaFra;
    public double mediaIng;
    public double mediaTot;
    public boolean esito = true;
    public boolean erasmus = false;
    public int insuf = 0;

    public Studente(String nome, String cognome, String sesso, Integer eta, Integer classe, Double mediaIta, Double mediaMate, Double mediaFra, Double mediaIng)
    {
        super(nome, cognome, sesso, eta);
        this.classe = classe;
        this.mediaIta = mediaIta;
        this.mediaMate = mediaMate;
        this.mediaFra = mediaFra;
        this.mediaIng = mediaIng;
        this.mediaTot = (mediaIta + mediaMate + mediaFra + mediaIng)/4;
    }
}
```

Così è come va fatta la classe figlia. “**Super**” si riferisce al costruttore di Persona, ossia al costruttore della classe padre.

(nome, cognome, sesso, eta) -> indica quali parametri non verranno gestiti da Studente ma verranno gestiti dal costruttore della classe padre, in questo caso da Persona.

Il costruttore di Studente riceve comunque tutti i parametri necessari alla costruzione dell'oggetto Persona. In più riceve anche quelli specifici per la classe Studente. All'interno del costruttore, i parametri per la costruzione di Persona vengono mandati alla classe padre, mentre gli altri vengono assegnati alle proprietà dell'oggetto direttamente nella classe figlia.

**Super** indica il costruttore della classe padre.

```
public String scheda() {
    String esitoStu = this.esito == true ? "Promosso" : "Bocciato";
    String eras = this.erasmus == true ? "Ammesso" : "Non ammesso";

    String risp = super.scheda()
        + "---- Classe: " + classe + "\n" +
        + "---- Media italiano: " + mediaIta + "\n" +
        + "---- Media matematica: " + mediaMate + "\n" +
        + "---- Media francese: " + mediaFra + "\n" +
        + "---- Media inglese: " + mediaIng + "\n" +
        + "---- Media totale: " + mediaTot + "\n" +
        + "---- Esito: " + esitoStu + "\n" +
        + "---- Erasmus: " + eras + "\n" +
        + "---- Numero insuf: " + insuf + "\n" +
    return risp;
}
```

Posso anche richiamare proprietà e metodi della classe padre con “**super.metodo()**”.

Le classi padre proteggono le classi figlie, quindi non fanno vedere le proprietà e i metodi delle classi figlie.

## 2.16 Polimorfismo dell'oggetto

Nuovo principio della programmazione: **polimorfismo dell'oggetto**.

"Persona p = new Studente()", si legge in questo modo:

- Persona -> indica il **tipo formale** dell'oggetto;
- p -> indica l'oggetto;
- Studente -> indica il tipo **concreto** dell'oggetto.

Java vedrà sempre gli oggetti per il loro tipo formale. Persona protegge in questo modo la classe figlia. "p" è un oggetto formalmente di classe Persona, concretamente di classe Studente.

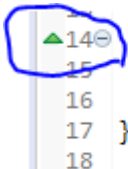
### 2.16.1 Polimorfismo dell'oggetto – override

Il polimorfismo dei metodi, nel caso dell'**override**, significa che il toString() che si trova in una classe (per es. Persona) sovrascrive il toString() della classe padre (in questo caso Object).

Per fare **override** bisogna scrivere la stessa firma del metodo in classi diverse:

```
public String toString() {  
    return nome + " " + eta;  
}
```

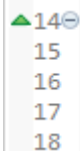
Questa "public String toString()" è uguale nella classe Object. Per capire se facciamo override bisogna vedere questo:



```
14 public String toString() {  
15     return nome + " " + eta;  
16 }  
17 }  
18 }
```

Andando sopra questo triangolo viene fuori che è un override.

**Override = firma uguale in classi diverse.**



```
14 overrides java.lang.Object.toString() {  
15     return nome + " " + eta;  
16 }  
17 }  
18 }
```

### 2.16.2 Polimorfismo dell'oggetto – overload

```

public double media() {
    return (mediaIta + mediaMate + mediaIng + mediaFra) / 4;
}

public double media(int nMaterie) {
    return (mediaIng + mediaFra) / nMaterie;
}

```

Questo è un esempio di **overload**, sono due metodi con lo stesso nome ma firma diversa nella stessa classe.

I due metodi hanno lo stesso nome, lo stesso tipo di ritorno ma nel primo caso non ci sono i parametri e nel secondo sì. Questo basta a Java per vederli in modo diverso.

Come può apparire diverso un metodo con lo stesso nome? Grazie ai parametri: cambiando l'ordine dei tipi con cui vengono passati al metodo.

La **firma del metodo** è il nome del metodo ed eventualmente i parametri. L'ordine dei parametri cambia la firma del metodo.

All'interno di una stessa classe, posso avere infiniti costruttori che permettono di creare oggetti in altrettanti infiniti modi. Attenzione, il polimorfismo del costruttore non c'entra nulla con il polimorfismo dell'oggetto!

## 2.17 Far vedere a Java il tipo concreto di un oggetto

```

Persona stud = new Studente("Marco", 28, 7.5, 8.6, 8, 6.7);

```

"stud" è di tipo formale Persona e noi sappiamo che Java vedrà sempre gli oggetti secondo il loro tipo formale. Persona tuttavia non permette l'accesso alle proprietà che sono presenti solo all'interno di Studente.

Persona ad esempio permette l'accesso al metodo "toString()" perché è presente anche all'interno della classe Persona. La classe Studente, semplicemente lo sovrascrive (override).

Ma come possiamo accedere ad esempio a "mediaIta", che è una proprietà specifica di Studente che non esiste in Persona? È necessario fare in modo che Java veda "stud" come un oggetto Studente e non come un oggetto Persona.

```

if(stud instanceof Studente) {
}

```

Per prima cosa va verificata l'appartenenza di "stud" alla classe Studente. **InstanceOf** si può tradurre come "appartiene a". Quindi, se "stud" appartiene anche alla classe Studente.

Se entro dentro all'if vuol dire che "stud", oltre a essere una Persona, appartiene anche alla classe Studente.

Il secondo step è convincere Java che si tratti di un oggetto Studente. È impossibile convincere Java, quindi faccio così:

- Dichiaro un oggetto di tipo Studente (tipo formale Studente).

- creo un oggetto Studente.

```
if(stud instanceof Studente) {  
    Studente stud1;  
    stud1 = (Studente) stud;  
}
```

Creo un oggetto Studente che è uguale a “stud” sotto forma di Studente. A questo punto, se volessi avere accesso al valore di mediaIta, dovrei chiamare la proprietà tramite un oggetto Studente.

Questa operazione prende il nome di **casting**, che è quando cerchiamo di mostrare a Java un oggetto per il suo tipo concreto.

```
if(stud instanceof Studente) {  
    Studente stud1;  
    stud1 = (Studente) stud;  
  
    mediaItaStud = stud1.mediaIta;  
    System.out.println(mediaItaStud);  
}
```

Il casting è il modificare il punto di vista di Java su un oggetto o qualcosa (ad esempio variabili). Modificare cioè il modo in cui Java vede quella cosa.

Il casting non è sempre necessario ma è obbligatorio quando ho bisogno di richiamare una proprietà o un metodo che è specifico esclusivamente di una classe e per tanto devo fare in modo che Java veda quell'oggetto per il suo tipo concreto.

## 2.18 Aggregatori

Tutte le operazioni che prima venivano svolte nel main, si trasformano in metodi nella **classe aggregatore**. La classe aggregatore serve a incapsulare perché pone un nuovo livello tra classi e main.

## 2.19 Abstract

Una classe astratta è una classe che non può essere istanziata. Non è possibile fare “new” su una classe astratta.

```
public abstract class Prodotto  
{
```

La classe astratta mi permette di non avere oggetti di tipo generico. Mi permette di evitare che oggetti di tipo troppo generico vengano creati.

Attenzione, il tipo formale degli oggetti sarà sempre “Prodotto”, in questo caso, ma concretamente non avrò nessun oggetto di tipo “Prodotto”.

A livello strutturale non cambia niente, a livello logico non abbiamo più il rischio di avere oggetti generici. Possiamo ancora dichiarare oggetti di tipo Prodotto, in linea con l'incapsulamento, ma non potremo più istanziarli (fare new). Se una classe non è abstract di default, è concreta.

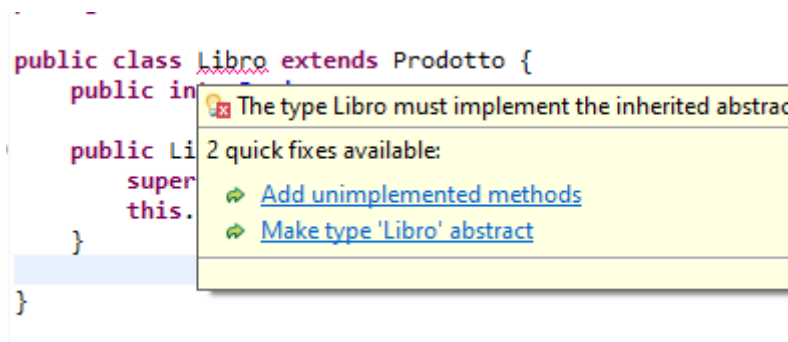
### 2.19.1 Metodi abstract

```
public abstract String toString();
```

Un metodo si compone di:

- Firma -> nome e parametri (occhio all'ordine);
- Corpo -> il corpo è il codice messo tra le graffe.

Un **metodo astratto non ha corpo**. Non ha importanza nella classe padre ma nella classe figlia. Un metodo astratto in una classe padre è un **obbligo** per il figlio, cioè il figlio, se il padre ha un metodo astratto, è obbligato a implementare il metodo. Se c'è un metodo astratto nella classe padre, il figlio è obbligato a implementarlo.



La classe figlia senza metodo infatti dà errore.

```
public class Libro extends Prodotto {
    public int nPagine;

    public Libro (String titolo, double prezzo, int
        super(titolo, prezzo);
        this.nPagine = nPagine;
    }

    @Override
    public String toString() {
        // TODO Auto-generated method stub
        return null;
    }
}
```

Così viene corretto in automatico.

Implementare vuol dire semplicemente dare un corpo al metodo.

Libro è una classe figlia di "Prodotto", siccome "Prodotto" ha un metodo abstract, "Libro" è obbligata a implementare il metodo abstract del padre.

Non importa come lo implementa, l'importante è che la classe figlia implementi tutti i metodi abstract della classe padre.

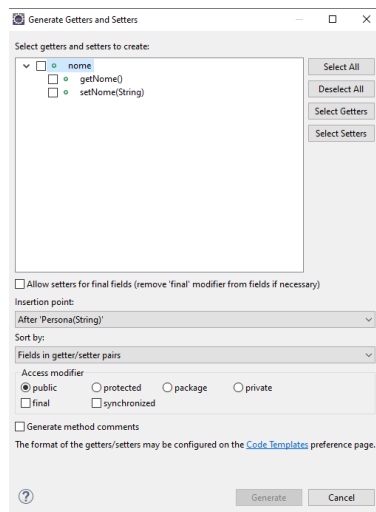
Stiamo implementando un metodo della classe padre, quindi invece di fare override, Java introduce una nuova keyword: **implements** (triangolino bianco):

```
10  
11 implements entities.Prodotto.toString(  
12 // TODO Auto-generated method st  
13 return null;  
14 }  
15
```

Nel main in questo modo non dobbiamo più castare.

Una classe astratta non contiene per forza metodi astratti. Una classe che contiene anche un solo metodo astratto, deve essere astratta.

## 2.20 Getters and Setters



Funzione che serve per generare dei metodi.

Il getter è un metodo pubblico che ritorna la stringa nome.

```
private String nome;  
  
public Persona(String nome)  
{  
    this.nome = nome;  
}  
  
public String getNome() {  
    return nome;  
}
```

Ora posso mettere le proprietà private e far sì che il nome sia accessibile solo come valore (solo get) e non modificabile.



Il metodo `getNome()` restituisce il valore della proprietà `nome`. In questo modo la proprietà può essere privata e siamo sicuri che chiunque ne abbia bisogno non acceda direttamente alla proprietà ma semplicemente al suo valore.

Il metodo `getNome()` ma in generale i metodi `Get` di Java, sono solitamente utilizzati per leggere.

Da oggi in poi le proprietà dell'oggetto saranno sempre private e l'unico modo per accedere al loro valore sarà attraverso il `get`.

I metodi `set()` servono per impostare. In questo caso, i metodi `set()` ricevono un parametro dall'esterno e si occupano dell'associazione di tale parametro con la proprietà dell'oggetto.

Il metodo `setNome(String nome)` riceve un parametro di tipo `String` e lo associa alla proprietà dell'oggetto `nome` esattamente come fa il costruttore:

```
public void setNome(String nome) {  
    this.nome = nome;  
}
```

A questo punto posso riformulare il costruttore in questo modo:

```
public Persona(String nome)  
{  
    setNome(nome);  
}
```

Mentre prima l'associazione del valore con il param era diretta, ora passa per il metodo `set`. Il costruttore riceve i parametri ma non li associa direttamente alle proprietà dell'oggetto. Per l'associazione chiede aiuto al metodo `set`.

```
public void setNome(String nome) {  
    if(nome.length() > 0)  
        this.nome = nome;  
    else  
        this.nome = "ERRORE - CAMPO VUOTO";  
}
```

Possiamo fare questa cosa molto comoda.

```
public String getNome() {  
    return nome;  
}  
  
public void setNome(String nome) {  
    if(nome.length() > 0)  
        this.nome = nome;  
    else  
        this.nome = "ERRORE - CAMPO VUOTO";  
}
```

Come vediamo la differenza tra i due è il **void**. I metodi `void` sono metodi particolari che non necessariamente hanno `return`. Un metodo `void` serve per modificare lo stato dell'oggetto.

Che cos'è lo stato dell'oggetto? È **l'insieme dei valori delle proprietà dell'oggetto in un dato istante**.

I metodi void sono metodi che hanno ritorno ma che servono a modificare i valori delle proprietà dell'oggetto.

## 2.21 Proprietà e metodi statici

**Static** indica una proprietà o un metodo di classe.

Una proprietà o un metodo di classe, se visibile, può essere richiamato indipendentemente dall'esistenza dell'oggetto. Se una proprietà dell'oggetto per esistere ha bisogno dell'oggetto, una proprietà di classe, per esistere, ha bisogno della classe.

```
public static int etaMinima = 0;
```

Static = **di classe**.

```
public class Main {  
    public static void main (String[] args) {  
  
        Persona p;  
        System.out.println(Persona.etaMinima);  
  
        p = new Persona("Luca");  
        System.out.println(p.toString());  
    }  
}
```

Come si vede, la proprietà della classe può essere richiamata anche prima che venga istanziato l'oggetto, perché si riferisce alla classe. chiedo alla classe il valore di quella proprietà, non all'oggetto.

La proprietà di classe esiste a prescindere dall'oggetto.

Ovviamente per usare una proprietà di classe, bisogna importare la classe.

```
Persona.LunghezzaMinimaNome = -1;
```

Così si sovrascrive la proprietà di classe.

Le proprietà di classe, a differenza di quelle dell'oggetto, vengono scritte in maiuscolo.

```
public static int ETAMINIMA = 0;  
public static int LUNGHEZZAMINIMANOME = 2;  
public static int LUNGHEZZAMASSIMANOME = 120;
```

Il valore di una proprietà di classe static può essere modificato. Se invece non si vuole dare la possibilità di modifica al valore impostato, invece di una proprietà static, si crea una proprietà **static final**:

```
public static final int LUNGHEZZAMASSIMANOME = 120;
```

Il valore di una proprietà static final non può essere modificato, se non alla riga in cui viene creata.

Possiamo, con le proprietà di classe, andare a verificare i valori delle proprietà dell'oggetto prima di creare l'oggetto, in modo che venga creato solo se rispetta le condizioni.

Le proprietà static e static finale permettono di verificare prima l'oggetto e i valori dei parametri.

Dopo aver visto le proprietà di classe, vediamo come vengono utilizzate in modo più pratico:

```
package util;  
    Static_Final_02/src/main/Main.java  
public abstract class Vik {  
    // In questa classe inseriremo tutto quello che può servire per fare  
    // controlli preventivi o successivi alla creazione di un oggetto  
  
    //La classe VIK, conterrà solo ed esclusivamente proprietà e metodi  
    // STATIC o STATIC FINAL  
  
    public static int MINCARATTERI = 2;  
    public static int MAXCARATTERI = 120;  
  
    public static boolean controllaString(String parametro) {  
        boolean ris = false;  
        if (parametro.length() > MINCARATTERI)  
            ris = true;  
        return ris;  
    }  
}
```

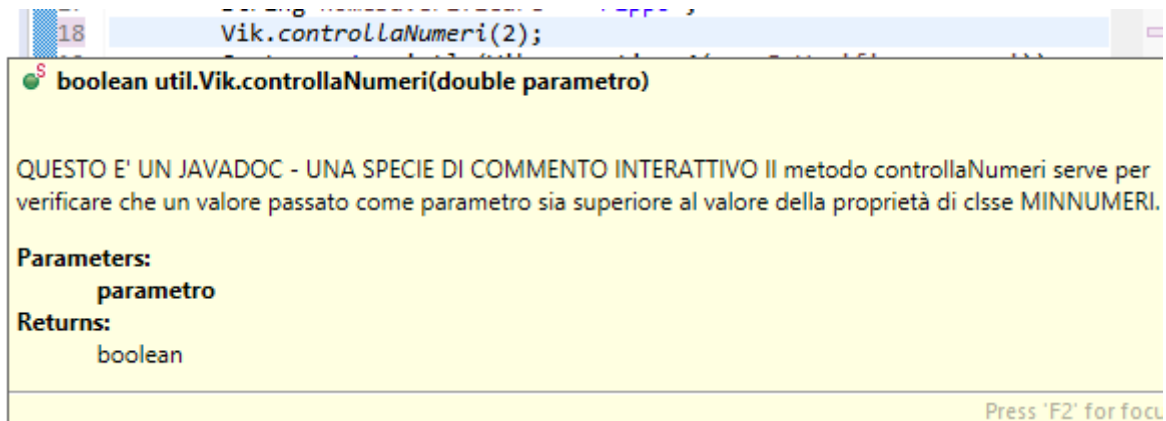
Creo una classe in un package util. A questo punto posso controllare la creazione dell'oggetto di tipo Persona in questo modo:

```
Persona p;  
String valoreNome = "Luca";  
  
if(Vik.controllaString(valoreNome))  
    p = new Persona(valoreNome);
```

A questo punto il controllo è fatto prima della creazione dell'oggetto.

```
/**  
 * QUESTO E' UN JAVADOC - UNA SPECIE DI COMMENTO INTERATTIVO  
 * Il metodo controllaNumeri serve per verificare che un valore pas  
 * come parametro sia superiore al valore della proprietà di classe  
 * MINNUMERI.  
 * @param parametro  
 * @return boolean  
 */
```

Questo è fondamentale per annotare i metodi.



Così quando riutilizzo non devo ogni volta ricordarmi come funziona.

## 2.22 Gestione delle eccezioni (try/catch)

Ci sono 2 tipi di eccezioni:

- Quelle che derivano da un problema logico (quando facciamo un errore logico noi programmatori, ad esempio creiamo un loop infinito).
- Eccezione runtime.

```

Main_Eccezione_01.java
4
5 public class Main_Eccezione_01
6 {
7     public static void main(String[] args)
8     {
9         Scanner tastiera = new Scanner(System.in);
10        System.out.println("Inserisci un numero");
11        int numero = Integer.parseInt(tastiera.nextLine());
12
13        System.out.println(numero);
14        tastiera.close();
15    }
16 }
17

```

Markers Properties Servers Data Source Explorer Snippets Console

<terminated> Main\_Eccezione\_01 [Java Application] C:\Program Files\Java\jre1.8.0\_261\bin\javaw.exe (05 nov 2020, 11:41)

Inserisci un numero

asdad

Exception in thread "main" java.lang.NumberFormatException: For input string: "asdad"

at java.lang.NumberFormatException.forInputString(Unknown Source)

at java.lang.Integer.parseInt(Unknown Source)

at java.lang.Integer.parseInt(Unknown Source)

at main.Main\_Eccezione\_01.main(Main\_Eccezione\_01.java:11)

Come gestisco una cosa del genere?

Se l'utente inserisce un valore che non riesce a essere trasformato in un numero da `Integer.parseInt()`, il programma genera un'eccezione. L'eccezione in questione si chiama `NumberFormatException` e il programmatore non può in alcun modo prevederla.

Tuttavia può gestirla attraverso un blocco **try/catch**.

Il blocco **try** tenta di eseguire un codice. La riga qui sotto viene definita **codice caldo**, perché è quella che genera l'eccezione:

```
try {  
    int numero = Integer.parseInt(tastiera.nextLine());  
    System.out.println(numero);  
}
```

Nel **catch** metto nelle parentesi tonde il tipo di exception che potrebbe verificarsi, in questo caso `NumberFormatException`:

```
try {  
    int numero = Integer.parseInt(tastiera.nextLine());  
    System.out.println(numero);  
} catch (NumberFormatException e) {  
}
```

Bisogna pensare all'eccezione come al sintomo di una malattia: in base ai sintomi viene definita una cura all'interno del catch.

All'interno del catch, definiamo il comportamento del programma nel caso in cui si presenti quell'eccezione:

```
Scanner tastiera = new Scanner(System.in);  
System.out.println("Inserisci un numero");  
  
try {  
    int numero = Integer.parseInt(tastiera.nextLine());  
    System.out.println(numero);  
} catch (NumberFormatException e) {  
    System.out.println("Hai inserito un valore non valido");  
}
```

Questo è il codice completo.

"e" è il nome che abbiamo dato al nostro oggetto. Come tutti gli oggetti, ha delle proprietà e dei metodi. Se voglio ad esempio stampare comunque il messaggio di errore quando si presenta l'exception, posso usare il metodo `printStackTrace()`.

```
    } //Fine di catch  
    catch (Exception e)  
    {  
        System.out.println("Eccezione non riconosciuta");  
    }
```

```

2 //Se si volesse stampare il messaggio di errore dell'Exception si potrebbe
3 //utilizzare getMessage()
4 System.out.println("Errore: " + e.getMessage());
5
6 } //Fine di catch
7 catch(Exception e)
8 {
9     //Exception e gestisce tutte le possibili eccezioni
10    //E' importante tuttavia specificare sempre l'eccezione o le eccezioni
11    //più probabili per avere dei feedback mirati da parte del programma
12    //e stabilire soluzioni adeguate.
13    System.out.println("Eccezione non riconosciuta");
14 }

```

```

public static void main(String[] args) {
    Scanner file = new Scanner(new File(""));

    // throws FileNotFoundException significa che il programma non gestisce l'eccezione
    // FileNotFoundException. All'interno di un programma infatti, una classe può in certi
    // casi decidere di non gestire l'eccezione ma di rimandare ad altri.
    // In pratica puoi gestire l'eccezione o fare scaricabarile.
}

```

Throws FileNotFoundException significa che non viene gestita l'eccezione.

```



public static void main(String[] args) {
    Scanner file = new Scanner(new File(""));

    // throws FileNotFoundException
    // FileNotFoundException. A
    // casi decidere di non ges
    // In pratica puoi gestire
}

```

Unhandled exception type FileNotFoundException

2 quick fixes available:

-  [Add throws declaration](#)
-  [Surround with try/catch](#)

Press 'F2' for focus

Abbiamo due opzioni. La prima è quella in cui non viene gestita l'eccezione. La seconda è quella in cui viene gestita e il codice diventa così:

```

public static void main(String[] args) {
    try {
        Scanner file = new Scanner(new File(""));
    } catch (FileNotFoundException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}

```

## 2.23 Mappe

Le mappe sono una **coppia chiave valore**. Consideriamo la chiave come una **proprietà**, il valore come il **valore** della chiave.

```
Map<String,String> libro;
```

Quando si dichiara una mappa, bisogna specificare il tipo della chiave e il tipo del valore.

Map, come Scanner, va importato.

Inizializziamo ora la mappa libro. Map rappresenta il **tipo formale** di libro. Il tipo **concreto** potrà essere **HashMap** o **LinkedHashMap**.

```
Map<String,String> libro;  
  
libro = new HashMap<String, String>();
```

Così inizializziamo. Anche HashMap deve essere importato.

Proviamo ad aggiungere una coppia chiave/valore alla mappa libro appena creata. Per aggiungere valori alla mappa si utilizza il metodo **.put()** che riceve entrambi i campi, sia chiave che valore, sotto forma di parametri.

```
10      // Le mappe sono una coppia chiave valore  
11  
12      Map<String,String> libro;  
13  
14      libro = new HashMap<String, String>();  
15  
16      libro.put("titolo", "Il signore degli anelli");  
17  
18      System.out.println(libro);  
19  }  
20 }
```

<terminated> Mappe\_01 [Java Application] C:\Program Files\Java\jre1.8.0\_261\bin\javaw.exe  
{titolo=Il signore degli anelli}

Proviamo a stampare le chiavi e i valori della mappa libro:

```
17      System.out.println(libro);  
18  
19      System.out.println(libro.keySet() + " " + libro.values());  
20  
21  }
```

<terminated> Mappe\_01 [Java Application] C:\Program Files\Java\jre1.8.0\_261\bin\javaw.exe (06 r  
{titolo=Il signore degli anelli}  
[titolo] [Il signore degli anelli]

Le mappe lavorano con coppie chiave/valore. Non è possibile per una mappa non avere una chiave associata a un valore e viceversa. Solitamente si utilizzano come coppie sempre String, String, perché anche se dovessi ricevere un valore numerico, posso gestirlo e trasformarlo all'occorrenza.

```
// Creo un insieme che contenga mappe
// Siccome le mappe sono oggetti, posso inserirle in un ArrayList o in un
// Vettore
ArrayList<Map<String,String>> elenco;
```

Elenco è un ArrayList che conterrà oggetti di tipo Map<String,String>.

Attenzione perché un oggetto di tipo Map<String,String> è diverso da un oggetto di tipo Map<String,Integer> ad esempio.

Se non ci interessa l'ordine non c'è differenza tra HashMap e LinkedHashMap. L'HashMap ritorna i dati in modo disordinato ma non è un disordine causale, segue un algoritmo.

## 2.24 LE INTERFACCE

Con le interfacce creo un livello in più di incapsulamento, in modo che il main non comunichi direttamente con l'aggregatore ma che abbia l'interfaccia come intermediario.

Un'interfaccia può essere implementata da infinite classi. Non ci sono problemi a implementare un'interfaccia.

Inoltre posso creare diverse interfacce.

Differenze principali tra interfacce e classi astratte:

- Fino a Java 1.7 la differenza più grande è che l'interfaccia non può avere le proprietà e quindi tutto ciò che ne consegue (niente stato dell'oggetto, niente istanziamento dell'oggetto).



## 3.0 MYSQL

Un database è un archivio per archiviare dati spesso complicati.

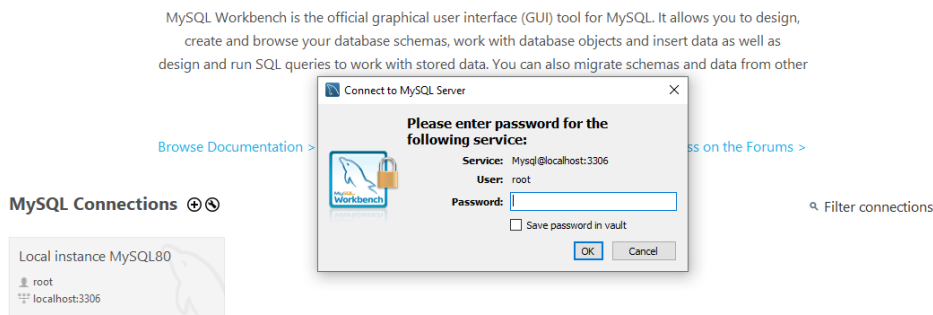
**Dbms** = database manage system. Noi utilizziamo Mysql, cioè un sistema di gestione di base di dati. È un software.

Mysql workbench non è un sistema di gestione ma è il programma che ci permette di utilizzare il sistema di gestione. Un database è un archivio che permette di gestire, manipolare e interrogare informazioni.

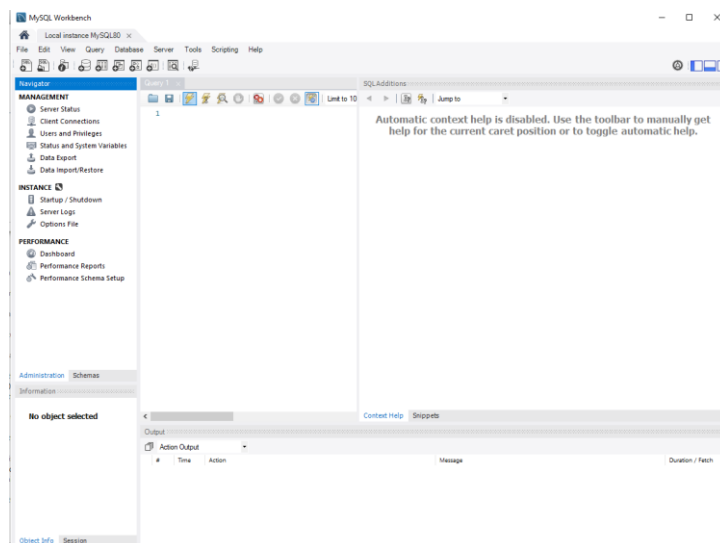
Un dbms è un software che permette di creare, manipolare e interrogare il database. Il nostro dbms è Mysql.

**Un database è un archivio che permette di gestire, organizzare e interrogare informazioni.**

**Un dbms è il software che permette di creare, manipolare e interrogare il db.**






Creiamo una connection tra noi e il database. C'è bisogno di 2 valori: user e password. Inserisco "root" e "root" e si apre questo:






## MANAGEMENT

-  [Server Status](#)
-  [Client Connections](#)
-  [Users and Privileges](#)
-  [Status and System Variables](#)
-  [Data Export](#)
-  [Data Import/Restore](#)


## INSTANCE

-  [Startup / Shutdown](#)
-  [Server Logs](#)
-  [Options File](#)

## PERFORMANCE

-  [Dashboard](#)
-  [Performance Reports](#)
-  [Performance Schema Setup](#)

Andando a sinistra su server status vedo tutti i dati del server:



Connection Name  
**Local instance MySQL80**

Host: **DESKTOP-GRQLM6T**  
Socket: **MySQL**  
Port: **3306**  
Version: **8.0.22 (MySQL Community Server - GPL)**  
Compiled For: **Win64 (x86\_64)**  
Configuration File: **C:\ProgramData\MySQL\MySQL Server 8.0\my.ini**  
Running Since: **Wed Nov 11 07:51:54 2020 (2:33)**

[Refresh](#)

### Available Server Features

Performance Schema:	<input checked="" type="radio"/> On	Windows Authentication:	<input type="radio"/> Off
Thread Pool:	<input type="radio"/> n/a	Password Validation:	<input type="radio"/> n/a
Memcached Plugin:	<input type="radio"/> n/a	Audit Log:	<input type="radio"/> n/a
Semisync Replication Plugin:	<input type="radio"/> n/a	Firewall:	<input type="radio"/> n/a
SSL Availability:	<input checked="" type="radio"/> On	Firewall Trace:	<input type="radio"/> n/a

### Server Directories

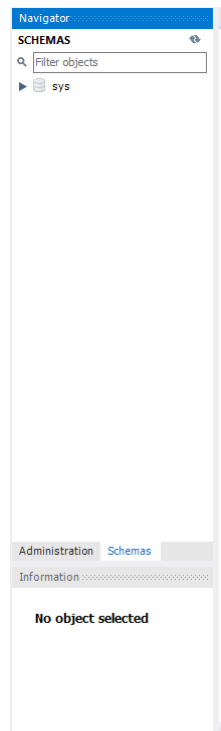
Base Directory:	C:\Program Files\MySQL\MySQL Server 8.0\
Data Directory:	C:\ProgramData\MySQL\MySQL Server 8.0\Data\
Disk Space in Data Dir:	19.62 GB of 255.21 GB available
Plugins Directory:	C:\Program Files\MySQL\MySQL Server 8.0\lib\plugin\
Tmp Directory:	C:\WINDOWS\SERVIC~1\NETWOR~1\AppData\Local\Temp
Error Log:	<input checked="" type="radio"/> On .\DESKTOP-GRQLM6T.err
General Log:	<input type="radio"/> Off
Slow Query Log:	<input checked="" type="radio"/> On DESKTOP-GRQLM6T-slow.log

### Replication Slave

: this server is not a slave in a replication setup

La porta in automatico è 3306.

Sulla sinistra vediamo anche gli schemas:

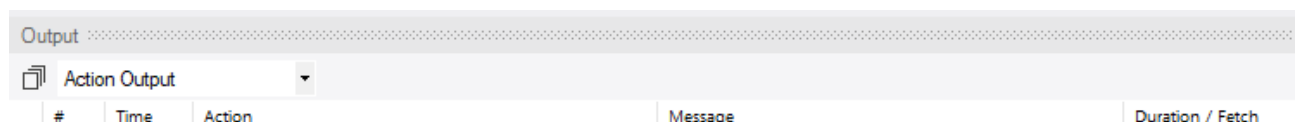


Il database può avere un numero illimitato di connessioni. La connessione è il ponte che connette al db. Una connessione può avere un limite di grandezza. A ogni connection corrispondono n database.

Le **queries** sono interrogazioni che facciamo al database. Una **query** è un'interrogazione, vale una volta sola, una volta lanciata è finita. Tutte le volte che scriviamo una query, quella query vale il momento in cui viene lanciata. Se chiediamo l'elenco degli studenti a un database, appena restituisce la risposta, il database dimentica la domanda.

Il database restituisce risposte alle singole domande.

Non tiene i dati in memoria perché i dati si riaggiornano in continuazione.

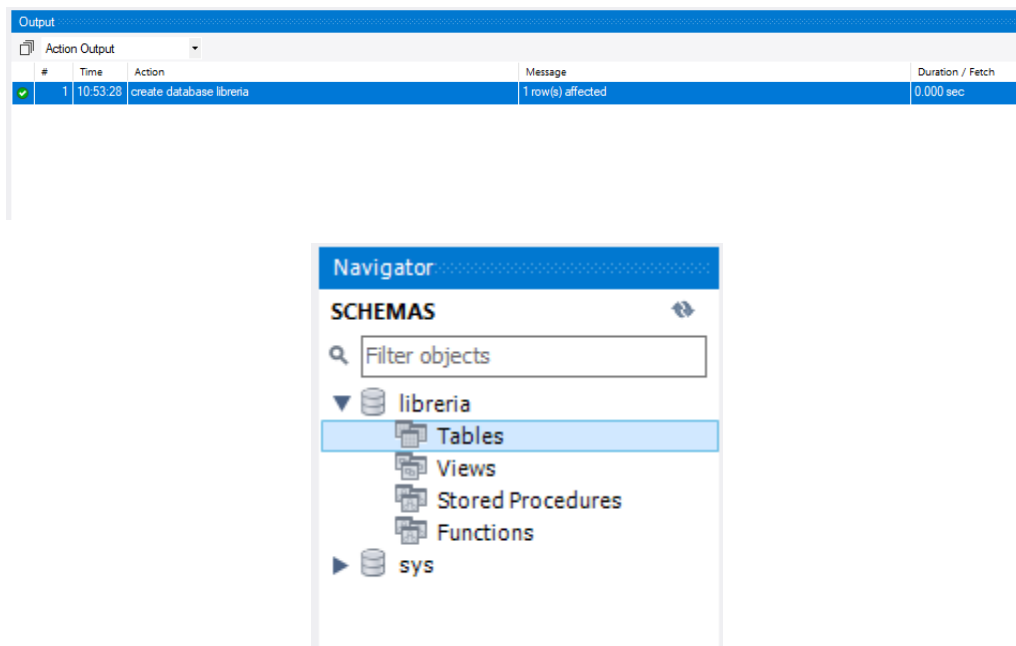


Nella parte sotto si vede l'esito delle interrogazioni.

### 3.1 Creare un database

```
1 -- Questo è un commento, proviamo a creare un db
2 • create database libreria;
3 |
```

Così si crea un database, eseguiamo l'istruzione e otteniamo questo:



A questo punto sono connesso a un server, al cui interno c'è un database libreria.

**Create** è la keyword che serve a creare qualsiasi struttura del database. **Database** è la keyword che permette di interagire col database.

Una volta creato il database devo ricordarmi di fare in modo che venga utilizzato, attraverso la keyword **use**.

```
1 • use libreria;

-- Creo le tabelle
create table libri
( -- Dentro alle parentesi inserisco i campi della tabella
  id int,
  titolo varchar(256),
  autore varchar(256),
  nPagine int,
  genere varchar(256),
  prezzo float
);
```

In questo modo creo una tabella. Devo specificare il tipo dei campi, in varchar si indica il numero di caratteri. Per i double possiamo usare anche float.

```

create table libri
( -- Dentro alle parentesi inserisco i campi della tabella
  id int primary key auto_increment,
  titolo varchar(60),
  autore varchar(60),
  nPagine int,
  genere varchar(10),
  prezzo float
);

```

All'id metto anche **primary key**: la primary key di una tabella è il valore che non può essere vuoto e non può essere ripetuto.

Aggiungo anche **auto\_increment** per far sì che il valore di id aumenti automaticamente a ogni nuovo record.

Una volta creata la tabella, ci inserisco i dati così:

```

insert into libri values
(1,'Il signore degli anelli','Tolkien',1250,'Fantasy',25),
(2,'Locke and Key','Joe Hill',600,'Horror',15),
(3,'La storia infinita','Michael Ende',300,'Fantasy',10),
(4,'Il condominio','Ballard',180,'Sci-Fi',12);

```

```

29 -- Finora ho sempre solo costruito senza richiedere effettivamente
30 -- qualcosa. Ora provo a interrogare il db
31 • select * from libri;

```

id	titolo	autore	nPagine	genere	prezzo
1	Il signore degli anelli	Tolkien	1250	Fantasy	25
2	Locke and Key	Joe Hill	600	Horror	15
3	La storia infinita	Michael Ende	300	Fantasy	10
4	Il condominio	Ballard	180	Sci-Fi	12
5	Watch	Alan Moore	NULL	NULL	35
6	V for Vendetta	Alan Moore	NULL	NULL	28

Con \* poi posso selezionare tutti i record della tabella.

## 3.2 Queries sql

```
select * from prodotti;
```

Questa è la query base per restituire tutti i dati della tabella.

```
select * from prodotti where prodotti.quantita < 6;
```

Così viene imposta una condizione alla restituzione dei dati, con la parola **where**.

```
select merci.nome from Merci where merci.quantita < 5 and merci.reparto = "cartoleria";
```

Qui vediamo come si seleziona soltanto una colonna della tabella.

## 3.3 Modificare le tabelle

```
alter table prodotti  
add column prezzoFinale double  
after dataCambio;
```

In questo modo si cambia una tabella già esistente, aggiungendo una colonna nuova, specificandone il nome e la posizione.

```
update prodotti set prodotti.iva = 20 where prodotti.id > 0;  
update prodotti set prodotti.iva = 4 where prodotti.tipo = "alimentari" and prodotti.id > 0;  
update prodotti set prodotti.iva = 10 where prodotti.tipo = "casalinghi" and prodotti.id > 0;
```

In questo modo si aggiornano dei record, modificando i valori relativi a una determinata colonna.