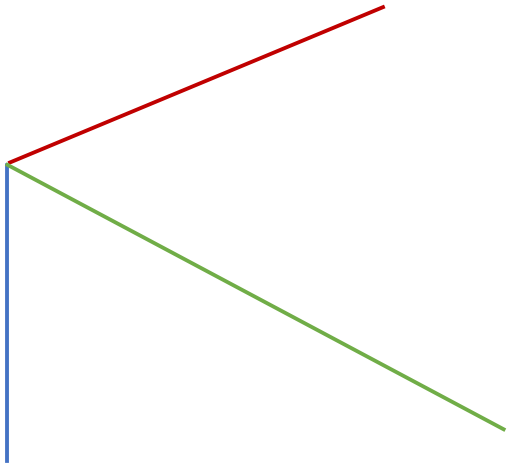
2019



INTRODUZIONE ALLA PROGRAMMAZIONE

WWW.FILOWEB.IT

FILIPPO BRUNELLI

Introduzione alla programmazione

Capitolo I
COSA SONO I PROGRAMMI ..... Pag. 2
TIPOLOGIA DI LINGUAGGI DI PROGRAMMAZIONE..... Pag. 2
TECNICHE DI PROGRAMMAZIONE: PROCEDURALE, STRUTTURATA E AD OGGETTI ..... Pag. 4
CONCETTI BASE: BIT, BYTE E NUMERAZIONE BINARIA..... Pag. 6
CONCETTI BASE: LE VARIABILI ..... Pag. 7
CONCETTI BASE GLI ARRAY ..... Pag. 8
CONCETTI BASE: LE CLASSI..... Pag. 8

Capitolo II
COSA CI SERVE E PERCHÉ CI SERVE ..... Pag. 10
IL PRIMO PROGRAMMA: LA COMPILAZIONE ..... Pag. 10
IL PRIMO PROGRAMMA: SPIEGAZIONE ..... Pag. 11
IL PRIMO PROGRAMMA: PERSONALIZZIAMO LA RISPOSTA! ..... Pag. 13
IL PRIMO PROGRAMMA: FACCIAMO DOMANDE E OTTENIAMO RISPOSTE! ..... Pag. 14
IL SECONDO PROGRAMMA: FACCIAMO I CONTI E TIRIAMO LE SOMME ..... Pag. 15
ALTRI TIPI DI OPERATORI DI BASE
..... Pag. 16
FACCIAMO DELLE SCELTE ..... Pag. 17
REITERIAMO LE ISTRUZIONI: I CLICLI ..... Pag. 20

Capitolo III
NON ABBIAMO ANCORA FINITO CON LE CLASSI ..... Pag. 22
OVERLOAD DEI METODI ..... Pag. 26
LE CLASSI STATICHE ..... Pag. 27
EREDITARIETÀ DELLE CLASSI
..... Pag. 28
POLIMORFISMO ..... Pag. 29
TIRIAMO LE SOMME ..... Pag. 31

Capitolo IV
ALLEGATI: PRINCIPALI COMANDI DOS A RIGA DI COMANDO ..... Pag. 32
ALLEGATI: IMPOSTARE IL PATH DI WINDOWS ..... Pag. 33
ALLEGATI: LE VARIABILI ..... Pag. 34

CAPITOLO I

COSA SONO I PROGRAMMI

Un programma non è altro che una serie di istruzioni, scritte in uno specifico linguaggio, da fare eseguire ad un computer.
Facciamo l'esempio di dover dare delle indicazioni stradali ad una persona. Diremmo qualcosa del tipo:
Vai avanti fino al semaforo, gira a destra e prosegui per tutta la via. Quando arrivi in piazza parcheggia che sei arrivato.
Ecco! Un programma si comporta allo stesso modo.
Nel nostro esempio abbiamo dato le istruzioni in Italiano, ma nel caso il nostro interlocutore fosse stato, ad esempio, inglese non avremmo potuto dirle in italiano ma nella sua lingua o in una lingua che entrambi conosciamo.
Anche nel mondo dei computer esistono tante lingue per fare i programmi e come nel mondo reale ogni lingua ha una sua grammatica, un suo lessico ed una sua sintassi e costruzione logica delle frasi e dei "dialetti".

TIPOLOGIA DI LINGUAGGI DI PROGRAMMAZIONE

Per "comunicare" con i computer i primi programmatori usavano scrivere i comandi direttamente nella lingua dei computer 01010110. Ecco che, ad esempio, scrivere 00110100 poteva voler dire "LOAD 8" dove 0011 è la rappresentazione interna del codice operativo LOAD mentre 0100 è il numero 8 in binario.
Se si voleva scrivere, ad esempio, la frase "Ciao Filippo" sullo schermo si scriveva qualcosa del tipo:

```
00000000  7f 45 4c 46 01 01 01 00  00 00 00 00 00 00 00 00
00000010  02 00 03 00 01 00 00 00  80 80 04 08 34 00 00 00
00000020  c8 00 00 00 00 00 00 00  34 00 20 00 02 00 28 00
00000030  04 00 03 00 01 00 00 00  00 00 00 00 00 80 04 08
00000040  00 80 04 08 9d 00 00 00  9d 00 00 00 05 00 00 00
00000050  00 10 00 00 01 00 00 00  a0 00 00 00 a0 90 04 08
00000060  a0 90 04 08 0e 00 00 00  0e 00 00 00 06 00 00 00
00000070  00 10 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00000080  ba 0e 00 00 00 b9 a0 90  04 08 bb 01 00 00 00 b8
00000090  04 00 00 00 cd 80 b8 01  00 00 00 cd 80 00 00 00
000000a0  43 69 61 6f 20 46 69 6c  69 70 70 6f 00 00 00 00
000000b0  73 68 73 74 72 74 61 62  00 2e 74 65 78 74 00 2e
000000c0  64 61 74 61 00 00 00 00  00 00 00 00 00 00 00 00
000000d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00

000000f0  0b 00 00 00 01 00 00 00  06 00 00 00 80 80 04 08
00000100  80 00 00 00 1d 00 00 00  00 00 00 00 00 00 00 00
00000110  10 00 00 00 00 00 00 00  11 00 00 00 01 00 00 00
00000120  03 00 00 00 a0 90 04 08  a0 00 00 00 0e 00 00 00
00000130  00 00 00 00 00 00 00 00  04 00 00 00 00 00 00 00
00000140  01 00 00 00 03 00 00 00  00 00 00 00 00 00 00 00
00000150  ae 00 00 00 17 00 00 00  00 00 00 00 00 00 00 00
00000160  01 00 00 00 00 00 00 00
```

Dove la linea scritta in rosso corrisponde a "Ciao Filippo".
Questo tipo di programmazione era molto difficile ed era molto vincolato al tipo di macchina usato.
Il passaggio successivo fu quello di "semplificare" questo metodo di programmazione con l'introduzione dell'assembly, un linguaggio molto simile al linguaggio macchina ma "più semplice" anche se sempre legato al tipo di hardware.
Per scrivere un programma in assembly si scrive un file di testo che poi viene dato in pasto ad un altro programma (chiamato assembler) che restituisce un eseguibile (ovvero un programma che il computer può eseguire); a seconda dell'assembler utilizzato si doveva usare uno specifico "dialetto".

Ciao Filippo in assembly per il compilatore NASM	Ciao Filippo in assembly per il compilatore INTEL

`org  0x100`
`  mov  dx, msg`
`  mov  ah, 9`
`  int  0x21`
`  mov  ah, 0x4c`
`  int  0x21`
`  msg  db 'Ciao Filippo', 0x0d, 0x0a, '$'`
	`Model small`
	`Stack 100h`
	`.data`
	`  hw db  "ciao Filippo", 13, 10, '$'`
	`.code`
	`.startup`
	`  mov ax, @data`
	`  mov ds, ax`
	`  mov dx, offset hw`
	`  mov ah, 09h`
	`  int 21h`
	`  mov ax, 4c00h`
	`  int 21h`
	`End`

Come si vede entrambi i metodi sono molto simili, fanno riferimento agli stessi indirizzi di memoria e funzioni ma sono scritti in un "dialetto" differente l'uno dall'altro: per entrambi **mov dx** sposta il messaggio nell'indirizzo di memoria dx, **mov ah,9** richiamala funzione per stampare, **int 0x21** richiamo il servizio DOS, **db** contiene il messaggio e **'$'** serve a indicare che il messaggio è terminato; **Stack 100h** e **org 0x100** indicano, che il programma deve essere caricato all'indirizzo 0x0100 in quanto compilando il programma diventerà il programma ciao.com (e non exe!). Le differenze sono nella sintassi. Da un punto di vista "grammaticale" invece vediamo che per scrivere un testo in NASM lo sacchiudo tramite gli apici (' ') mentre in INTEL tramite le virgolette (" ").
Con la crescita dell'uso dei computer ci si accorse ben presto che i linguaggi assembly facevano si risparmiare tempo rispetto al linguaggio macchina, ma che, se andavano bene per i comandi base di un calcolatore, diventava complesso gestire un programma troppo evoluto.
Tra gli anni '50 e '60 del XX° secolo si iniziarono a sviluppare linguaggi più complessi, che risultavano più facili da apprendere e da usare e che facevano risparmiare tempo ai programmatori. Si preferì cioè dare priorità alla realizzazione di un programma piuttosto che alla sua esecuzione.
Nascono così quelli che vengono chiamati "linguaggi di alto livello": più un linguaggio si avvicina al linguaggio macchina più viene chiamato di basso livello, mentre più si allontana più è di alto livello.
I linguaggi di alto livello necessitano un compilatore o un interprete che sia in grado di "tradurre" le istruzioni del linguaggio di alto livello in istruzioni macchina di basso livello eseguibili dal computer.
Un **compilatore** è simile ad un assemblatore ma molto più complesso perché ogni singola istruzione di un linguaggio di alto livello corrisponde a più istruzioni in linguaggio macchina e quindi deve "tradurre" in maniera più complicata. Il risultato sarà un programma eseguibile (solitamente .exe).
Un **interprete**, invece, è un programma che è in grado di tradurre un file (solitamente di testo) contenente un programma scritto in un linguaggio di programmazione, e di farlo eseguire dal computer.
Come conseguenza avremmo che un programma interpretato sarà molto più lento nell'esecuzione di un programma compilato.
Linguaggi di alto livello sono, ad esempio, il Pascal, il C ed il Basic, molto usati negli anni '70.

Ciao Filippo in Pascal	Ciao Filippo in C	Ciao Filippo in Basic

`program ciaofilippo;`
`begin`
`  writeln('Ciao Filippo');`
`  readln;`
`end.`
	`#include <stdio.h>`
	`int main()`
	`{`
	`  printf("Ciao Filippo");`
	`  return 0;`
	`}`
		`CLS`
		`PRINT "Ciao Filippo"`

Come si vede, scrivere un programma in questo modo semplifica, e di molto, sia i passaggi necessari sia un eventuale modifica.
Negli ultimi anni si è sviluppato un nuovo approccio ai linguaggi di programmazione, l'uso di bytecode.
Il bytecode è un linguagio intermedio: il codice sorgente viene compilato in un formato intermedio (chiamato appunto bytecode), il quale a sua volta viene interpretato da una Virtual Machine, che ha il compito di interpretare "al volo" le istruzioni bytecode in istruzioni per il processore. Questa metodologia permette di creare programmi che hanno un grande livello di portabilità e al contempo velocità. Basta avere la macchina virtuale installata sul proprio computer (non importa se il sistema sia Windows, Mac, Linux o altro) e il programma bytecode verrà eseguito.
La macchiana virtuale più famosa è sicuramente la Java Virtual Machine che "traduce" il bytecode (solitamente un file .class) in linguaggio macchina tramite una procedura chiamata *just in time*.
Anche la microsoft (tramite il frameworks .Net) ha creato il suo strumento di sviluppo, anche più complesso ed articolato di quello di Sun, dato che .Net può interagire con diversi linguaggi, il più famoso ed utilizzato dei quali è C# (# si legge sharp).

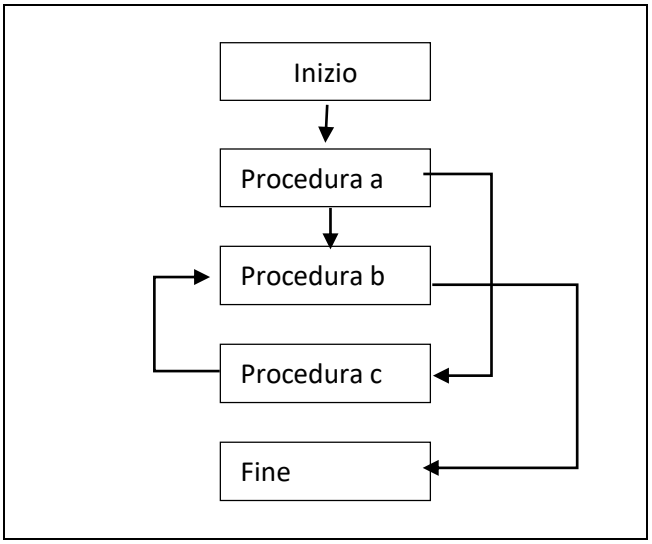
Ciao Filippo in Java	Ciao Filippo in C#

`public class ciaofilippo {`
 	
`    public static void main(String[] args) {`
`        System.out.println("Ciao Filippo");`
`    }`
 	
`}`
	`using System;`
	`class ciao`
	`    {`
	`        static void Main()`
	`        {`
	`            Console.WriteLine("Ciao Filippo");`
	`        }`
	`    }`

TECNICHE DI PROGRAMMAZIONE: PROCEDURALE, STRUTTURATA E AD OGGETTI

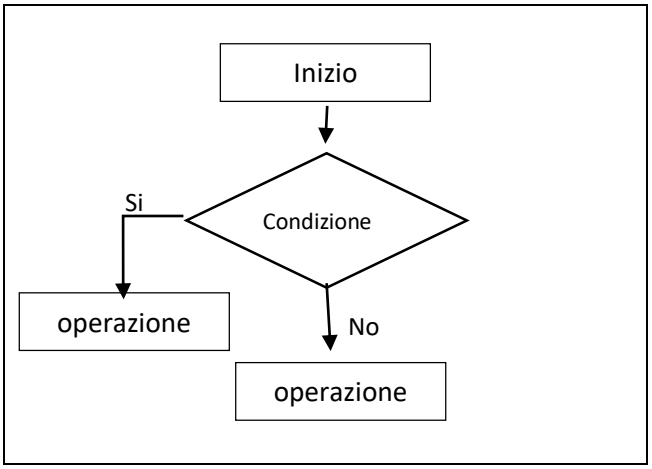
Abbiamo visto cosa sono i linguaggi di programmazione, ma non abbiamo ancora parlato delle tecniche che si usa per programmare. Anche qua, come nei vari linguaggi, possiamo suddividere le tipologie macro aree: Procedurale, Strutturata e Orientata agli Oggetti.

La **programmazione Procedurale** è una delle prime tipologie di programmazione utilizzate. Fa ricorso a diversi "salti" all'interno del programma per richiamare subroutine e scambiare con loro i dati. Il codice viene scritto passo dopo passo ed eseguito partendo dall'alto (la prima riga) e scendendo. La programmazione procedurale, facendo largo uso di subroutine crea dei programmi difficili da leggere e da correggere.



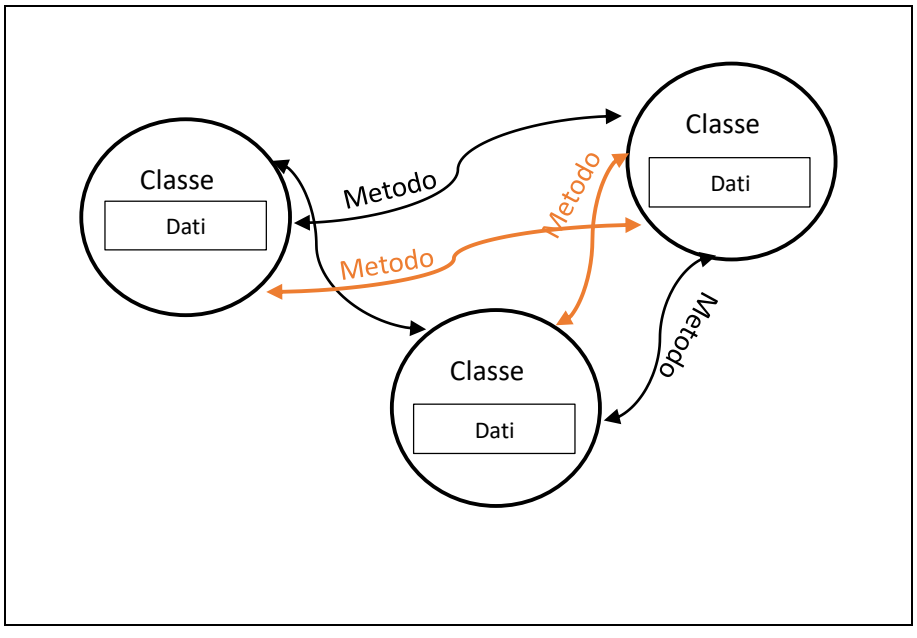
La programmazione Procedurale è considerata ormai deprecabile; linguaggi che ne facevano uso erano, ad esempio, i primi Basic ed il Cobol .

La **programmazione strutturata**, evoluzione di quella procedurale, pur continuando ad avere un approccio top-down enfatizza maggiormente la separazione dei dati trattati rispetto alle operazioni del programma stesso. L'esecuzione delle istruzioni non è più fatta obbligatoriamente in ordine sequenziale ma è gestita con un flusso logico e condizionata con loop, sequenze e decisioni (if, switch, while, ecc.) basate sul valore dei dati.



Esempi di linguaggi che usano questo tipo di programmazione sono il Pascal, il C, l'Ada,il Fortran, il Visual Basic fino alla versione 6.

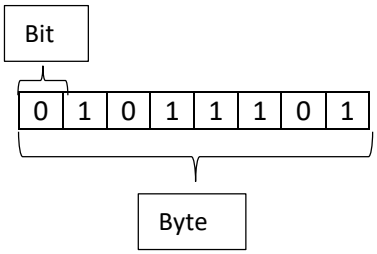
I linguaggi di **programmazione ad oggetti** sono, ad oggi, i più utilizzati. In questo tipo di programmazione i programmi sono scritti come una serie di "oggetti" che comunicano tra loro. Questi oggetti prendono il nome di Classi e sono visti come un "contenitore" che combina i dati con le operazioni (azioni) che possono essere eseguite su di essi. Ogni oggetto ha delle proprietà e delle azioni che possono essere eseguite su di essi.
Diversamente dai metodi di programmazione precedenti gli OPP (Object Oriented Programming) non hanno un metodo di programmazione dall'alto al basso, ma hanno un punto di inizio (main) e tutti gli oggetti che vengono definiti con le loro proprietà e come interagiscono tra di loro.



Esempi di linguaggi che usano questo tipo di programmazione sono Java e C#, C++, VB.Net.

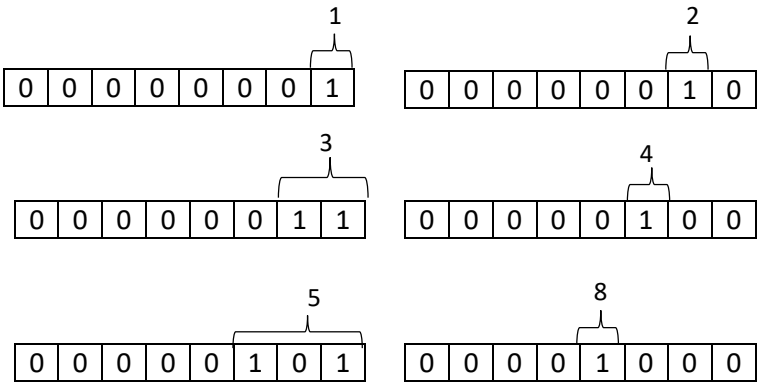
CONCETTI BASE: BIT, BYTE E NUMERAZIONE BINARIA

Abbiamo visto come i primi linguaggi di programmazione utilizzavano un linguaggio macchina per scrivere i programmi, inserendo una serie di 01010101. Questo tipo di numerazione prende il nome di binario.
Il sistema binario è un sistema di numerazione a base 2. L'uso di questo tipo di numerazione si era reso necessario in quanto il più simile al concetto di acceso/spento, vero/falso tipico degli elaboratori elettronici moderni (almeno fino all'arrivo dei computer quantistici!).
Ma a cosa serve oggi conoscere il sistema binario con linguaggi di programmazione evoluti?
Tra tutti i sistemi numerazione, dopo il sistema decimale, quello binario è quello che si incontra maggiormente nella programmazione in quanto sta alla base dei concetti di bit e byte.
Nei computer il concetto di 1 e 0 corrisponde alla presenza o all'assenza di tensione sul circuito; ciascun valore di 0 o di 1 è chiamato bit che corrisponde alla fusione delle parole binary digit (cifra binaria).
Presi da soli non hanno molto senso gli 0 o gli 1, ma messi in gruppi possono trasmettere delle informazioni.
Nei primi processori si usavano gruppi di 4 bit, poi si è passati a 7 per finire a 8bit (i famosi computer a 8 bit di fine anni '70 inizio '80). È di quel periodo la convenzione che un **byte** corrisponde ad un insieme di 8 bit.



Due byte fanno una **Word** (16 bit); una **Dword** (invece) non indica, in programmazione, 16bit ma 32 se riferita a processori a 16 bit mentre indica 64 se riferita a processori a 32bit.

Esempi di numerazione binaria.



Tramite la codifica ASCII, nel 1968 viene codificato un insieme di 7bit nei caratteri standard latini, così il numero 65 corrisponde al carattere **A**, mentre il numero 97 corrisponde al carattere **a**; per lo spazio il codice corrispondente è il numero 32.
Nel 1992 è stata ideata la codifica UTF-8 che usa 8bit ed è l'evoluzione del sistema ASCII che può usare un maggior numero di caratteri.

Ultima nota da considerare è che essendo in base 2 la numerazione binaria 1 Mb (1 Megabyte) dovrebbe corrispondere a 1024 KByte e così per i suoi multipli; in effetti così è stato fino a poco tempo fa, quando, per non far confusione agli utenti meno preparati (ignoranti) i produttori di Hard Disk decisero di arrotondare a potenze di dieci invece che di due e di sfruttare tale cosa per fini commerciali. Ecco che 1 Mb diventa 1000 Kbyte e viene coniata una nuova nomenclatura con parole tipoa Kib (Kibibit), MiB megabyte, e così per tutti i multipli, ecco perché acquistando un Hard disk da un Terabyte acquistiamo in realtà la possibilità di scrivere una quantità inferiore del 7.4% di quello che si aspetta!

CONCETTI BASE: LE VARIABILI

Il concetto di variabile quando si programma è una delle basi fondamentali. Possiamo paragonare una variabile ad una scatola dentro la quale metteremo i nostri dati che andremo ad elaborare.
Posso, ad esempio usare la variabile **a** per contenere il valore **10** e la variabile **b** per il valore **5** per finire con il mettere la somma di **a+b** dentro la variabile **c**.

```
a=10
b=5
c=a+b
```

A seconda del linguaggio di programmazione la variabile cambia nel suo modus operandi.
In assembly, ad esempio, corrisponde ad un indirizzo di memoria nel quale andiamo a mettere un valore; abbiamo visto all'inizio con l'esempio "Ciao Filippo" che mettiamo il valore di msg nel registro Dx (move Dx, msg).
In linguaggi più evoluti bisogna "dichiarare" la variabile e specificarne il tipo (testo, numero intero, numero a virgola mobile, valore boleano, ecc.). In C++, ad esempio, posso dichiarare:

```
int a;
```

per definire una variabile di tipo intero e verrà riservato in memoria lo spazio per questo tipo di variabile.
I numeri interi così dichiarati sono composti da 4 byte e possono quindi contenere numeri da -2.147.483.648 a 2.147.483.648, mentre se definisco **uint** (u sta per unsign ovvero senga segno) i numeri andranno da 0 a 4.294.967.295 (ovvero il doppio).
Il programmatore potrà sapere in ogni momento in quale cella di memoria è puntata una determinata variabile e richiamarla anche direttamente da lì. Questo è anche uno dei motivi per cui in alcuni linguaggi bisogna "distruggere" le variabili dopo averle utilizzate in modo da non tenere la memoria occupata.
Dichiarare una variabile, ad esempio, di tipo **string** vuol dire creare uno spazio di memoria che accetterà una sequenza di caratteri e il valore occupato sarà, in questo caso, variabile, mentre se dichiaro una variabile di tipo **byte** potrà contenere valori da 0 a 255 (dimensione 1 byte) mentre una di tipo **Bool** dichiaro una variabile di tipo boleano (sempre 1 byte) potrà contenere solo valori true o false (vero o falso).
È importante sapere che tipo di variabile sto usando perché non si possono fare operazioni su variabili che contengono tipi diversi di dati.
Nei linguaggi compilati è necessario che le variabili siano dichiarate affinché il compilatore o l'interprete sappia che tipo di dato si sta usando e la quantità di memoria da utilizzare.
In linguaggi di livello più astratto, l'unico modo di trovare una variabile dichiarata è il suo nome in quanto il suo indirizzo di memoria è allocato dinamicamente dal linguaggio e dal sistema operativo.

CONCETTI BASE: LE COSTANTI

Le costanti possono essere definite come delle "scatole" il cui contenuto non cambia durante tutta l'esecuzione del programma.
A differenza delle variabili quando si crea una costante, il compilatore sa che non deve leggere il valora ma ha già il valore definito.
Un esempio di costante potrebbe essere

```
const double Pi =3.141592653589
```

dove double indica il tipo di dato numerico a 8 byte con virgola (a differenza di int che indica un intero).
Note di programmazione: posso usare anche il valore float (che è a 4bit) per un valore numerico a virgola mobile; in questo caso avrò una precisione minore (7 cifre contro le 16 del double) ma con un uso minore di risorse.
Nell'uso del tipo di variabile è importante decidere il tipo di variabile in base all'uso che ne viene fatto.

È infatti inutile dichiarare una variabile double per il pi greco se devo solo calcolare l'area di un cerchio per un problema di matematica scolastico (sarà sufficiente un float con il valore 3.141592) ma se devo fare dei calcoli di ingegneria o che necessitano precisione userò un double con maggior precisione!

CONCETTI BASE GLI ARRAY

Può capitare di dover inserire un'aggregazione di valori dello stesso tipo. In questo caso, invece che creare moltissime variabili (es. p1, p2,p3… p_n) si ricorre a quelle che vengono chiamate variabili strutturate, o più comunemente **array**.
Un array non è altro che un insieme di variabili, tutte con lo stesso nome e dello stesso tipo i cui elementi sono disposti in memoria in maniera consecutiva.
Ad esempio posso definire l'array animale composto da tre valori

```
string[] animale= new string[3];
           animale[0] = "cane";
           animale[1] = "gatto";
           animale[2] = "ornitorinco";
```

Non entriamo nei tecnicismi ma diciamo subito che se leggo il valore di animale[1] avrò come risultato gatto.
Gli array sono presenti praticamente in tutti i linguaggi di programmazione e, come in una matrice matematica, può avere più dimensioni:

```
int[,] array2d = new int[3, 4];
          array2d[0,0] = 1;
          array2d[0,1] = 2;
          array2d[2,2] = 4;
```

in questo caso il nostro array possiamo immaginarlo come una griglia di 3x4 caselle contenenti valori numerici interi.

CONCETTI BASE: LE CLASSI

Le classi sono una proprietà specifica dei linguaggi orientati ad oggetti; il concetto di classe è molto astratto e capirlo (soprattutto per chi viene da un approccio strutturale) non è sempre semplicissimo. In questa parte non ci occupiamo di come si creano le classi in uno specifico linguaggio ma cercheremo di spiegare in maniera più semplice possibile cosa sono.
Possiamo considerare la classe come il modello sul quale si costruisce l'oggetto. Facciamo un esempio.
Creo una classe chiamata automobile che avrà come "proprietà" (per adesso usiamo il termine proprietà anche se in maniera non corretta per spiegare il concetto di classe) la marca, il modello ed il colore.
L'oggetto MiaAutomobile avrà le "prorietà" della classe automobile ma le caratteristiche della mia auto

MiaAutomobile.marca = "Alfaromeo"
MiaAutomobile.modello = "Giulietta"
MiaAutomobile.colore = "Nero"

Fin qua tutto chiaro? Spero di si!
Ogni classe contiene degli attributi (ovvero i dati) e dei metodi (le funzioni che manipolano i dati).
Ad esempio posso creare un metodo che all'inserimento dei dati restituirà la scritta "la tua auto è bella"

Esempio di classe automobile in C#

```
using System;

class PrimoEsempio
{
    public static void Main()
    {
        automobile MiaAutomobile;
        MiaAutomobile = new automobile ();

        MiaAutomobile.marca = "Alfaromeo";
        MiaAutomobile.modello = "Giulia";
        MiaAutomobile.colore = "nero";

                        Console.WriteLine(MiaAutomobile.marca);
                        Console.Write(MiaAutomobile.modello);
                        Console.Write(" di colore ");
                        Console.Write(MiaAutomobile.colore);
                        MiaAutomobile.risposta();
    }
}
public class automobile
{
     public string marca;
     public string modello;
     public string colore;

    public void risposta()
    {
        System.Console.WriteLine(" .La tua auto è bella.");
    }

}
```

Le operazioni che possono essere utilizzate dalla classe corrente o anche da altre prendono il nome di metodi.

Altre caratteristiche delle classi, quali l'ereditarietà, il polimorfismo, ecc. saranno trattati separatamente; per adesso è importante che si capisca il concetto di classe.
Ricapitolando, quindi, la mia classe è la classe automobile, mentre il mio oggetto si chiama MiaAutomobile, che creo tramite:

```
automobile MiaAutomobile;
MiaAutomobile = new automobile ();
```

Note: per creare l'oggetto MiaAutomobile potevo anche scrivere

```
automobile MiaAutomobile = new automobile ();
```

CAPITOLO II

COSA CI SERVE E PERCHÉ CI SERVE

Tra i tanti linguaggi disponibili useremo C# di Microsoft con il relativo frameworks su di un sistema windows. Perché questa scelta?
Innanzitutto, così come Java, C# è un moderno linguaggio ad oggetti, ha una semantica molto simile al C, C++ e Java; grazie a .Net Core si possono sviluppare applicazioni per praticamente qualsiasi piattaforma. A differenza di Java C# è standard ECMA (2001) ed Iso (2003). Per ultimo ma non per questo meno imporante, per compilare in C# non si deve necessariamente comprare un compilatore si possono usare sia editor che compilatori free: **csc.exe** è presente nel frameworks di .Net con windows mentre per altri sistemi si può usare **.Net Core**, mentre visual studio code (sempre multipiattaforma) è un ottimo editor gratuito.

> **Note:** Ci sono diverse versioni del Frameworks .Net. Non è detto che un programma scritto in .Net 3.5 funzioni se è installato solamente .Net 4.0 quindi è sempre meglio tenere anche le versioni meno aggiornate di .Net se si usano programmi datati.

C# è una sintesi delle migliori caratteristiche degli altri linguaggi: è potente come C e C++, completo come Java e semplice da programmare come Delphi e Visual Basic.
Per lavorare con C# si può tranquillamente scaricare la versione gratuita di Visual Studio, ricorrere al progetto Mono (https://www.mono-project.com/) disponibile per Linux, Mac, Windows o semplicemente un editor di testo (anche se consigliamo di usare Visual Studio Code) e compilare a riga di comando.
Per gli esempi che seguiranno useremo il sistema operativo Windows che è uno dei più usati (non consideriamo per adesso Android che è prettamente un sistema operativo per dispositivi mobili e quindi difficile da usare per sviluppare programmi).
Si richiede, inoltre, una conoscenza di base del sistema operativo Windows, dell'uso del prompt dei comandi e dell'organizzazione dei file sul disco fisso tramite la conoscenza e utilizzo delle direcotry tramite shell.
Per prima cosa verifichiamo di aver installato il frameworks .Net e quale versione.
Apriamo la cartella *C:\Windows\Microsoft.NET\Framework* (si suppone che windows sia installato nel disco C:\ !) e vediamo le versioni di Net installate.
Il nostro compilatore si trova all'interno di una di queste cartelle (ad esempio \v4.0.30319) con il nome csc.exe.
Nella stessa cartella esiste anche vbc.exe che è il compilatore Visual Basic .Net, che a differenza del suo predecessore Visual Basic 6 è un linguaggio ad oggetti.
Il comando csc.exe ha diverse opzioni, per vederle è sufficiente scrivere **csc.exe /?** e notiamo subito che ci permette, tra le altre cose, do compilare anche delle librerie (.DLL), dei moduli che possono essere aggiunti ad altri assembly e moltissime altre opzioni.

IL PRIMO PROGRAMMA: LA COMPILAZIONE

A questo punto, possiamo provare a scrivere e compilare il nostro programma: auto.exe.
Apriamo il nostro editor di testo preferito (no Word, Wordpad o altri editor simili che formattano il testo!) e scriviamo il programma automobili che abbiamo visto quando abbiamo introdotto il concetto di classe e che andiamo a salvare (meglio se in una cartella facilmente raggiungibile!) con il nome auto.cs .
Apriamo il prompt dei comandi e posizioniamoci nella cartella che ho creato.
Scriviamo:

```
C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe auto.cs
```

Se tutto è corretto verrà compilato il programma auto.exe. Verifico tramite il comando **dir** che sia presente nella directory e lancio auto.exe

Note: L'esempio utilizza il frameworks 4.5, nel caso si utilizzi un'altra versione utilizzare il percorso relativo a quella versione (ed. C:\Windows\Microsoft.NET\Framework\v3.5\csc.exe).
Volendo si può inserire il percorso del frameworks direttamente nei path di windows.

IL PRIMO PROGRAMMA: SPIEGAZIONE

Nella prima riga noto subito l'uso della direttiva **using**.

```
using System;
```

Organizzazione dei programmi c# usa i namespaces. Il namespaces, in questo caso, è usato come un sistema di organizzazione "external", ovvero un modo per presentare gli elementi di programma che vengono esposti ad altri programmi. La direttiva using è fornita per facilitare l'uso di questi namespaces.
Il namespaces sono l'equivalente dei pacages di Java.
La direttiva using ha 3 diversi utilizzi; in questo caso viene utilizzata per accedere ai membri statici di un tipo senza dover qualificare l'accesso con il nome del tipo.
Se non avessi specificato **using System** ad inizio avrei dovuto scrivere:

```
System.Console.WriteLine(MiaAutomobile.marca);
```

Mentre se avessi specificato **using System.Console** avrei potuto scrivere:

```
WriteLine(MiaAutomobile.marca);
```

Console è praticamente una Classe del name space System.
Anche se sembra creare confusione il richiamare o meno un name space all'inizio può invece risultare molto comodo nel caso si usino due name space che hanno una o più classi con lo stesso nome al loro interno: in questo modo posso richiamare uno soltanto dei due namespace o nemmeno uno e utilizzare la classe con lo stesso nome.

```
NamespaceA.classePippo();
NamespaceB.classePippo();
```

WriteLine è un "comando" che serve a scrivere, come dice il nome, una riga, mentre **Write** serve a scrivere senza andare a capo.
Nel nostro esempio abbiamo sia l'uso di WriteLine che l'uso di Write che sintatticamente sono simili: tra parentesi metto ciò che voglio che venga scritto; nel caso di una stringa da me immessa la racchiudo tra le virgolette.
Da un punto di vista sintattico notiamo che ogni istruzione termina con il carattere ";".
Questo vuol dire che non c'è alcuna differenza tra lo scrivere:

```
using System;
```

o srivere

```
using
    System;
```

Le parentesi graffe, invece, sono usate per racchiudere blocchi di istruzioni e nella seconda riga abbiamo:

```
class PrimoEsempio
        {
        }
```

La parola chiave **class** è usata per creare una classe.
Notiamo che non termina con un punto e virgola, e questo accade perché la definizione di classe non si limita alla sola dichiarazione. Tutto ciò che è racchiuso tra le parentesi graffe è ciò che la classe deve fare.
Note: *C# è case sensitive il che vuol dire che PrimoEsempio e primoesempio sono 2 cose diverse!*

Proseguendo troviamo:

```
public static void Main()
```

Partiamo dalla fine della riga per spiegare cosa abbiamo scritto: **Main()**.
Questo rappresenta il punto d' ingresso del programma.

All'avvio del programma Main() è il primo metodo chiamata e in un programma può essere presente uno ed uno soltanto punto d'ingresso. Se in un'applicazione sono presenti più punti d'ingresso Main() quando la si compila bisogna compilarla indicando quale di questi è il punto d'ingresso.

```
C:\Windows\Microsoft.NET\Framework\v4.0.30319\csc.exe auto.cs /main:PrimoEsempio
```

Il metodo Main() ha molte caratteristiche che vedremo più durante il corso, per adesso notiamo che tutto quello che è racchiuso all'interno della parentesi graffe di Main() è quello che verrà eseguito dal nostro programma.
Void che precede il punto di ingresso Main() indica che il metodo (blocco di codice che contiene una serie di istruzioni) non restituirà nessun valore. Non importa che venga scritto che l'auto è bella in quanto non è considerato un valore che può essere riutilizzato.
Se il metodo avesse dovuto, ad esempio, ritornare un numero intero dovevo scrivere int al posto di Void e specificare l'intero restituito tramite il comando return

```
using System;
class PrimoEsempio
{
    public static int Main()
    {
        automobile MiaAutomobile = new automobile ();

         MiaAutomobile.marca = "Alfaromeo";
         MiaAutomobile.modello = "Giulia";
         MiaAutomobile.colore = "nero";
                        Console.WriteLine(MiaAutomobile.marca);
                        Console.Write(MiaAutomobile.modello);
                        Console.Write(" di colore ");
                        Console.Write(MiaAutomobile.colore);
                        MiaAutomobile.risposta();
                        return 5;
    }
}
public class automobile
{
     public string marca;
     public string modello;
     public string colore;

    public void risposta()
    {
        System.Console.WriteLine(" la tua auto è bella.");
    }
}
```

Se provo a compilare il mio programma adesso e a lanciarlo in esecuzione vedo che a schermo non cambia nulla, ma in realtà mi torna un valore 5 che posso, ad esempio, leggere in un batch file o in uno script.
Static indica che il metodo agisce nel contesto della classe ma non necessariamente su una specifica istanza di oggetto; se applico, ad esempio static ad una classe non potrò usare parola chiave new per creare una variabile del tipo di classe.

Vediamo che dentro il metodo Main() la prima cosa che facciamo è il creare un oggetto MiaAutomobile dalla classe automobile

```
automobile MiaAutomobile = new automobile ();
```

A questo punto facciamo un salto a vedere la classe automobile e ad analizzarla.
Innanzi tutto vediamo che vengono dichiarate 3 variabili del tipo string (per poter inserire del testo) con la proprietà public.

```
     public string marca;
     public string modello;
     public string colore;
```

La parola **public** è un modificatore di accesso. I modificatori di accesso sono parole chiave usate per specificare l'accessibilità dichiarata di un membro o di un tipo, il più usato è sicuramente public che permette un accesso non limitato, gli altri sono private, protected, internal e protected internal. Di seguito le specifiche per ognuno di loro:

- **public:** L'accesso non è limitato.
- **private**: L'accesso è limitato al tipo contenitore.
- **protected:** L'accesso è limitato alla classe o ai tipi derivati dalla classe che li contiene.
- **internal:** L'accesso è limitato all'assembly corrente.
- **protected internal:** L'accesso è limitato all'assembly corrente o ai tipi derivati dalla classe che li contiene.
- **private protected:** L'accesso è limitato alla classe o ai tipi derivati dalla classe che li contiene all'interno dell'assembly corrente.

Proviamo a cambiare l'accesso alla variabile marca da public a private e a ricompilare il file

```
private string marca;
```

Vediamo che il compilatore csc.exe ci restituirà un errore:

```
       auto.cs(9,24): error CS0122: 'automobile.marca' è inaccessibile a causa del livello di
protezione.
       auto.cs(21,21): (Posizione del simbolo relativo all'errore precedente)
       auto.cs(12,56): error CS0122: 'automobile.marca' è inaccessibile a causa del livello di
protezione.
       auto.cs(21,21): (Posizione del simbolo relativo all'errore precedente)
```

Questo succede perché dichiarando la variabile private impostiamo l'accesso a quest'ultima solamente alla classe che la contiene.
I livelli di accesso sono disponibili, come si vede anche per le classi e per i metodi.
Da notare che fino a quando non viene creato l'oggetto MiaAutomobile le variabili non sono allocate in memoria; nel momento che creo l'oggetto viene creato lo spazio di memoria necessario per contenere le variabili!
Proseguendo il programma risulta molto semplice da capire: inseriamo nelle variabili marca, modello, colore dell'oggetto MiaAutomobile i relativi valori e scriviamo a testo i valori.
Per ultimo invoco l'azione risposta dell'oggetto MiaAutomobile

```
MiaAutomobile.risposta();
```

che fa scrivere a video la frase "La tua auto è bella".

IL PRIMO PROGRAMMA: PERSONALIZZIAMO LA RISPOSTA!

Bello ma questo metodo non è molto comodo dato che necessita ogni volta che cambiamo auto dobbiamo anche il dover ricompilare tutto.
Cosa fare allora? Passiamo i valori al nostro programma!
Modifichiamo il punto d'ingresso al nostro programma in questo modo:

```
public static void Main(string[] args)
```

Cosa vuol dire questa modifica? Vediamo che abbiamo aggiunto tra parentesi la scritta **string[] args**, in questo modo diciamo al compilatore che il programma dovrà ricevere dei parametri di tipo stringa quando viene lanciato.
Per vedere cosa succede modifichiamo anche l'assegnazione delle variabili del tipo di auto (marca modello e colore) come segue.

```
MiaAutomobile.marca = args[0];
MiaAutomobile.modello = args[1];
MiaAutomobile.colore = args[2];
```

In questo modo andiamo a recuperare 3 valori dalla variabile args di tipo string che viene definita in main(string[] args).
Compiliamo il nostro programma (se non si vuole perdere quello precedente possiamo salvarlo compilarlo con un altro nome ad esempio autob.cs e autob.exe) e lanciamolo con il comando:

```
autob.exe audi coupé nera
```

e vediamo che la risposta del nostro programma sarà conforme ai parametri che abbiamo inserito.

Note: Nel primo capitolo abbiamo introdotto il concetto di array, nel nostro programma args[] ne è un esempio.

IL PRIMO PROGRAMMA: FACCIAMO DOMANDE E OTTENIAMO RISPOSTE!

Se fino a qua è tutto chiaro possiamo fare un passo avanti e lasciare che sia il programma a chiederci quale marca, modello e colore di macchina ci piaccia.
In effetti un programma che non ci permette di inserire dei dati è un programma molto limitato, mentre uno che richiede i parametri a riga di comando può essere scomodo.
Per fare questo usiamo il comando **ReadLine()**.
ReadLine si comporta come WriteLine() solo che invece di scrivere una linea la legge.
Sostituiamo quindi

```
MiaAutomobile.marca = "Alfaromeo";
```

Con

```
MiaAutomobile.marca = Console.ReadLine();
```

In questo modo il nostro programma inserirà il contenuto che scriviamo sulla tastiera nella variabile marca di MiaAutomobile quando premo il tasto INVIO.
Per semplicità modifichiamo il nostro programma auto come segue:

```
using System;

class PrimoEsempio
{
    public static void Main()
    {
        automobile MiaAutomobile = new automobile ();

         Console.WriteLine("Che marca di automobile ti piace?");
         MiaAutomobile.marca = Console.ReadLine();
         Console.Write("Quale modello di ");
         Console.Write(MiaAutomobile.marca);
         Console.Write(" ti piace?");
         MiaAutomobile.modello = Console.ReadLine();
         Console.Write("Quale colore vorresti per la tua ");
         Console.Write(MiaAutomobile.marca);
         Console.Write(" ?");
         MiaAutomobile.colore = Console.ReadLine();
                        Console.Write(MiaAutomobile.marca);
                        Console.Write(" ");
                        Console.Write(MiaAutomobile.modello);
                        Console.Write(" di colore ");
                        Console.Write(MiaAutomobile.colore);
                        MiaAutomobile.risposta();
    }
}
public class automobile
{
     public string marca;
     public string modello;
     public string colore;

    public void risposta()
    {
        System.Console.WriteLine(". La tua auto è bella!");
    }
}
```

Non penso che servano molte spiegazioni per questo: il programma, una volta compilato e lanciato in esecuzione chiede all'utente che marca, modello e colore preferisce e scrive una risposta.

Note: abbiamo rimosso i parametri da Main() in quanto andiamo a "chiedere" i dati e non a doverli inserire come argomenti.

IL SECONDO PROGRAMMA: FACCIAMO I CONTI E TIRIAMO LE SOMME.

I computer, inizialmente, sono nati con lo scopo di semplificare i calcoli che gli scienziati dovevano fare.
Nel nostro primo programma, tuttavia, non abbiamo ancora fatto una sola operazione, il che non è giusto.
Chiariti i principi basilari vediamo quindi di iniziare a far di conto….
Creiamo un nuovo programma che chiameremo calcola.cs e scriviamo:

```
using System;

class calcola
{
    public static void Main(string[] args)
    {
        int a = int.Parse(args[0]); // Converto il primo valore in intero //
        int b = int.Parse(args[1]); // Converto il secondovalore in intero //
        int somma = a+b;
        int sott= a-b;
        int molt = a*b;
        int div = a/b;
        int modu = a%b;
        /*eseguo tutte le operazioni principali
        sono 5: + - * / % */
            Console.WriteLine("Somma dei numeri: " + somma);
            Console.WriteLine("Sottrazione dei numeri: " + sott);
            Console.WriteLine("Moltiplicazione dei numeri: " + molt);
            Console.WriteLine("Divisione dei numeri: " + div);
            Console.WriteLine("Modulo (resto della divisione) dei numeri: " + modu);

    }
}
```

Vediamo che abbiamo messo un po' di carne al fuoco….
La prima cosa che vorrei far notare è l'uso del metodo parse. Questo metodo ci permette di convertire una stringa passata come parametro in un numero (in questo caso intero). Questo ci serve perché i valori che passiamo al nostro programma sono sempre stringhe; non importa che io scriva 1, 3 o qualsiasi numero, per il programma è considerato una stringa e non un numero. Tramite parse viene estratto il valore interessato nell'array che passo e si ottiene un numero intero (int). Nel caso volessi ottenere un numero in virgola avrei potuto usare float.Parse o double.Parse.
A questo punto abbiamo i due valori numerici dentro le variabili a e b. Creiamo poi le operazioni che ci interessano.
In questo caso, le principali 5 operazioni: Somma (+), Sottrazione (–), moltiplicazione (*), divisione (/) e modulo (%) il resto della divisione.
Notiamo poi che quando scrivo i risultati uso il valore + come aggregatore che prende il nome di operatore di aggregatore di stringhe.
Il carattere + può essere infatti usato come operazione matematica se usata tra valori numerici ma anche come aggregatore se usato per valori di tipo stringa o misti come nel nostro caso.
Ecco che avremo che la scritta della somma sarà:
Somma dei numeri: 5 (o qualunque altro numero risulterà);
se avessimo usato lo stesso metodo nell'esempio precedente potevamo scrivere:

```
Console.Write(MiaAutomobile.marca + " " + MiaAutomobile.modello + " " + " di colore " +
MiaAutomobile.colore);
```

Per finire vediamo i commenti.
I **commenti** sono importanti quando si sviluppa un programma. In C# ci sono due modi per scrivere i commenti:
Il primo consiste nel racchiuderlo tra i simboli **//** quando sono tutti su di un'unica riga

```
int a = int.Parse(args[0]); // Converto il primo valore in intero //
```

Mentre se sono su più righe tra i simboli **/*** e ***/**

```
/*eseguo tutte le operazioni principali
        sono 5: + - * / % */
```

ALTRI TIPI DI OPERATORI DI BASE

I Oltre agli operatori aritmetici di base C# (così come altri linguaggi) ci mette a disposizione anche altri tipi di operatori. I primi che consideriamo sono gli **operatori di confronto**.
Come dice in nome stesso servono a confrontare due valori tra di loro e danno come risultato un valore booleano (vero o falso / true or false).
Gli operatori di confronto sono:

Operatore	Significato	Esempio
==	Uguale a	2==3 -> false 2==2 -> true
>	Maggiore	2>3 -> false 3>2 -> true
>=	Maggiore o uguale	2>=2 ->true 2>=3 -> false
<	Minore	2<3 -> true 3<1 ->false
<=	Minore o uguale	3<=3 ->true 3<=-7 -> false
!=	Diverso	3!=6 -> true 3!=3 -> false A!=B -> true

Esistono poi gli **operatori logici** (AND, OR e NOT) che in C# si rappresentano come

Operatore	Significato	Esempio
&&	AND logico	A=2 && b=6
||	OR logico	A=2 || b=6
!	NOT logico	A!=b

Cerchiamo di spiegare questi operatori in maniera semplice.
Il primo **AND** (&&) confronta due operazioni o due operatori di confronto e restituisce true se e solo se entrambe le condizioni risultano vere.
Nel nostro esempio è come se avessimo detto: *se A=2 e b=6 allora…*
L'operatore **OR** (||) invece, mi restituisce true solo se uno delle due condizioni risulta vera ma non entrambe!
Nel nostro esempio è come se avessimo detto: *se A=2 o b=6 allora….*
L'ultimo **NOT** (!) infine mi restituisce true se la condizione non è vera. Nel nostro esempio è come se avessimo detto: *se A non è b allora….*

Gli operatori di **incremento** (++) e di **decremento** (--) sono altri due operatori molto usati in C#. Sono rappresentati in forma prefissa o postfissa alla variabile.
Se scrivo, ad esempio: a++ equivale a scrivere a+1 mentre se scrivo a-- equivale a scrivere a-1

Vediamo degli esempi:

```
int a = 10;
int b = 5;
int c= a++ + b; // c= a+5=15; a=10+1
int d= ++a +b; //a=11+1=12 ; d=12+5=17
```

Nella prima somma c ha il valore di 15 in quanto il valore di a viene sommato dopo le altre operazioni.
Nel secondo esempio (a ha assunto il valore di 11 dall'operazione di precedente) prima aggiungo 1 ad a e poi sommo con b.

Gli ultimi operatori che consideriamo sono quelli di **assegnazione operazione**.

Operatore	Esempio	Significato
+=	a+=b	a=a+b
-=	a-=b	a=a-b
*=	a*=b	a=a*b
/=	a/=b	a=a/b
%=	a%=b	a=a%b

Questi sono i principali operatori; per gli altri si consiglia di vedere la documentazione ufficiale.

```
using System;

class compara
{
    public static void Main(string[] args)
    {
        int a = int.Parse(args[0]); // Converto il primo valore in intero //
        int b = int.Parse(args[1]); // Converto il secondovalore in intero //
              Console.WriteLine("a uguale a b:" );
              Console.WriteLine(a==b);
              Console.WriteLine("a maggiore di b:" );
              Console.WriteLine( a>b);
              Console.WriteLine("a minore di b:" );
              Console.WriteLine( a<b);
              Console.WriteLine("a diverso di b:" );
              Console.WriteLine( a!=b);
    }
}
```

Esempio di uso degli operatori di confronto. Compiliamo il programma e eseguiamolo come abbiamo eseguito il programma calcola.

FACCIAMO DELLE SCELTE.

Le istruzioni condizionali permettono di fare delle scelte all'interno di un programma o di una classe; le istruzioni condizionali sono presenti in tutti i linguaggi di programmazione ed in C# esistono due costrutti fondamentali, ovvero **switch** e **if/else**.

Il costrutto if/else viene usato per valutare se una espressione è vera o falsa, e di conseguenza far applicare delle scelte al programma.

Vediamo un esempio:

```
using System;

class calcola
{
    public static void Main(string[] args)
    {
        int a = int.Parse(args[0]); // Converto il primo valore in intero //
        int b = int.Parse(args[1]); // Converto il secondovalore in intero //

              if(a>b)
              {
                    Console.WriteLine ("A è maggiore di B");
              }
              else
              {
                    Console.WriteLine ("A è minore o uguale a B");
              }
    }
}
```

La sintassi di if/else è la seguente:

```
if (espressione)
 { blocco operazioni }
Else
 { blocco operazioni }
```

All'interno delle espressioni posso usare anche gli operatori logici :

```
if (espressione1 && espressione2)
 { blocco operazioni }
Else
 { blocco operazioni }
```

Possiamo modificare il nostro programma in questo modo :

```
if(a>b && a!=b)
        {
            Console.WriteLine ("A è maggiore e diverso da B ");
        }

        else
        {
             Console.WriteLine ("A è minore o uguale a B");
        }
```

In questo modo la condizione vuole che a sia maggiore di b e che a diverso da b, altrimenti si passa alla seconda condizione (si poteva usare anche >= o <= ma a noi interessava l'uso degli operatori logici).

Il costrutto **switch** rappresenta una valida alternativa al costrutto if/else.
Possiamo considerare switch è come un insieme di istruzioni if/else, una lista di possibilità, con un'azione per ogni possibilità, ed un'azione facoltativa di default.
La sintassi è

```
switch (espressione)
{
  case valore1:
```

```
    istruzioni;
    [break; | goto case n; | goto default;]

  case valore2:
    istruzioni;
    [break; | goto case n; | goto default;]

  [...]

  Default:
    istruzioni;
    break;
}
```

Dove case indica le varie opzioni con le relative istruzioni, **break** indica la fine dell'esecuzione delle operazioni switch, **goto** indica un salto condizionato e **default** le operazioni di default nel caso non sia valida nessuna delle condizioni selezionate in case.
Vediamo un esempio.

```
using System;

class  scegli
{
    public static void Main()
    {
        Console.WriteLine("Scegli un numero da 1 a 5:");
        string n=Console.ReadLine();
        int a = int.Parse(n); // Converto il valore di n da stringa ad intero //

       switch (a)
        {
        case 1:
            Console.WriteLine("Hai premuto 1");
            break;

        case 2:
            Console.WriteLine("Hai premuto 2");
            break;

        case 3:
            Console.WriteLine("Hai premuto 3 e ti ho fatto uno scherzo:");
            goto case 1;

        case 4:
        case 5:
            Console.WriteLine("Hai premuto 4 o 5");
            break;

        default:
                Console.WriteLine("Non so cosa hai premuto");
        break;
        }
    }
}
```

Notiamo che switch permette, come in case 4 e case 5 di raggruppare più scelte con un'unica risposta e di impostare dei salti all'interno delle risposte come in case 3 dove si salta al case 1.

Nel caso si usassero stringhe anziché numeri le condizioni vanno racchiuse tra virgolette.

```
using System;

class  scegli
{
    public static void Main()
    {
        Console.WriteLine("Come ti chiami?");
        string nome=Console.ReadLine();

       switch (nome)
        {
        case "Filippo":
            Console.WriteLine("Ciao Filippo");
            break;
        case "Marco":
            Console.WriteLine("Ciao Marco");
            break;
        default:
                Console.WriteLine("Ciao Estraneo");
        break;
        }
    }
}
```

REITERIAMO LE ISTRUZIONI: I CLICLI.

Come tutti i linguaggi di programmazione anche C# mette a disposizione la possibilità per il programmatore di reiterare i comandi tramite cicli.
While, Do… while e For sono i comandi che C# mette a disposizione per creare i cicli.

While serve per reiterare una serie di azioni finché una certa condizione non risulterà vera; la sintassi è:

```
While (espression) {ciclo di operazioni}
```

Do while è molto simile a while solo che, essendo la condizione while alla fine, il blocco di istruzioni verrà eseguito almeno una volta. La sintassi é:

```
Do {ciclo di operazion} While (espression);
```

Esempio di do…while

```
using System;
class  primaiterazione
{
    public static void Main()
    {
       int i = 0;
       do
       {
           Console.WriteLine(i);
           i++;
       }
        while (i<10);
    }
}
```

Esempio di While

```
using System;
class  secondaiterazione
{
    public static void Main()
    {
       int i = 0;
       while (i<10)
       {
           Console.WriteLine(i);
           i++;
       }

    }
}
```

Il ciclo **for**, invece, permette di eseguire una serie di istruzione mentre la condizione è vera.
La sua sintassi è:

```
for (variabile iteratore; condizione; iteratore)
    {
        ciclo di operazioni
    }
```

Capisco che vista così sembri quasi una formula alchemica misteriosa ma un esempio ci aiuterà a capire

```
using System;
class  ciclo
{
    public static void Main()
    {
       for (int i=10; i>0; i--)
       {
           Console.WriteLine(i);
       }

    }
}
```

Per finire questo capitolo diamo un'occhiata ad un operatore che C# si porta come eredità dal C e dal C++, l'operatore ternario.

L'**operatore ternario** è un operatore speciale la cui sintassi è:

```
espressione-condizione ? espressione1 : espressione2 ;
```

Per chi conosce Excel l'operatore ternario si comporta come il comando "se" in una cella.

```
using System;

class  ternario
{
    public static void Main()
    {
      int a=5;
      string risposta= a>0 ? "a è maggiore di 0" : "a è minore di 0";
      Console.WriteLine(risposta);

    }
}
```

Nel caso in esempio se il valore di a è maggiore di 0 il valore della stringa diventa "a è maggiore di 0" altrimenti diventa "a è minore di 0"

CAPITOLO III

NON ABBIAMO ANCORA FINITO CON LE CLASSI

In C#, come in tutti i linguaggi ad oggetti le classi sono la cosa più importante.
Fino ad ora ne abbiamo visto degli esempi anche per i programmi più semplici e siamo giunti ad un punto nel quale dobbiamo approfondire le caratteristiche delle classi.
Consideriamo il nostro primo programma auto ed in particolare la costruzione dell'oggetto MiaAutomobile

```
   automobile MiaAutomobile = new automobile ();
```

Vediamo che la parola chiave new è seguita dal nome della classe con due parentesi tonde. Questo è perché automobile è un metodo speciale chiamato **costruttore**.
Un costruttore non viene esplicitamente definito in una classe quando la si crea ma esiste anche se non svolge nessuna funzione viene semplicemente invocato quando si crea una nuova istanza per una classe, come nel nostro esempio.
Posso utilizzare i costruttori anche per inizializzare o invocare alcuni campi di una classe senza il dover accedervi direttamente.
Modifichiamo la nostra classe automobile come segue:

```
}
public class automobile
{
     private string marca;
     private string modello;
     private string colore;
     private byte anni;
     private float pagata;

        public automobile(
            string varmarca,
            string varmodello,
            string varcolore,
            byte varanni,
            float varpagata
            )
        {
            marca = varmarca;
            modello = varmodello;
            colore = varcolore;
            anni = varanni;
            pagata = varpagata;
        }

    public float valorenuovo()
    {
        return pagata - anni * 100;
    }
    public void risposta()
    {
        Console.WriteLine("marca:...........: " + marca );

        Console.WriteLine("modello:.........: " + modello );
        Console.WriteLine("colore:..........: " + colore );
        Console.WriteLine("pagata:..........: " + pagata );
        Console.WriteLine("anzianità:.......: " + anni );
        Console.WriteLine("Valore attuale:..: " + valorenuovo() );
    }

}
```

Come vediamo il costruttore è simile a un metodo che ha lo stesso nome della classe.
Spesso i costruttori sono utilizzati per inizializzare tutti o alcuni dati della classe e poter accedere ad essi dall'esterno.
Vediamo che nel nostro esempio abbiamo impostato le variabili nella classe come private e quindi non sarebbe possibile accedervi direttamente al difuori di essa. Tramite il costruttore possiamo avere accesso invece ad essi, infatti il costruttore accetta i parametri che li assegna ai membri interni.

```
public automobile(
            string varmarca,
            string varmodello,
            string varcolore,
            byte varanni,
            float varpagata
            )
        {
            marca = varmarca;
            modello = varmodello;
            colore = varcolore;
            anni = varanni;
            pagata = varpagata;
        }
```

Un'altra caratteristica è l'introduzione del metodo visualizza che mostra il contenuto dei dati membro (ricordiamo che precedentemente il metodo mostrava solamente una frase!).

```
public void risposta()
    {
        Console.WriteLine("marca:.....: " + marca );

        Console.WriteLine("modello:.....: " + modello );
        Console.WriteLine("colore:.....: " + colore );
        Console.WriteLine("pagata:.....: " + pagata );
        Console.WriteLine("anzianità:.....: " + anni );
        Console.WriteLine("Valore attuale:..: " + valorenuovo() );
     }
```

Ci stiamo avvicinando a quello che nella OOP è definito incapsulamento.
Adesso, quando creiamo l'oggetto MiaAutomobile posso scrivere:

```
public static void Main(string[] args)
    {
        automobile MiaAutomobile = new automobile (
                                                    "audi",
                                                    "coupe",
                                                    "nero",
                                                    12,
                                                    65000
                                                   );
        MiaAutomobile.risposta();
    }
```

In questo modo non è più necessario accedere direttamente ai dati membri della classe (che sono comunque privati), perché attraverso il costruttore è possibile inizializzarli.

Vediamo il programma completo che chiameremo: costruttore.cs

```
using System;

class costruttore
{
    public static void Main(string[] args)
    {
        automobile MiaAutomobile = new automobile (
                                                    "audi",
                                                    "coupe",
                                                    "nero",
                                                    12,
                                                    65000
                                                   );
        MiaAutomobile.risposta();
    }
}
public class automobile
{
    private string marca;
     private string modello;
     private string colore;
     private byte anni;
     private float pagata;

        public automobile(
            string varmarca,
            string varmodello,
            string varcolore,
            byte varanni,
            float varpagata
            )
        {
            marca = varmarca;
            modello = varmodello;
            colore = varcolore;
            anni = varanni;
            pagata = varpagata;
        }

    public float valorenuovo()
    {
        return pagata - anni * 100;
    }
    public void risposta()
    {
        Console.WriteLine("marca:..........: " + marca );
        Console.WriteLine("modello:........: " + modello );
        Console.WriteLine("colore:.........: " + colore );
        Console.WriteLine("pagata:.........: " + pagata );
        Console.WriteLine("anzianità:......: " + anni );
        Console.WriteLine("Valore attuale:..: " + valorenuovo() );
    }
}
```

In una classe posso avere due costruttore diversi con lo stesso nome, ma che si distinguono solamente per i parametri che vendono inviati.

```
public automobile(
            string varmarca,
            string varmodello
            )
        {
            marca = varmarca;
            modello = varmodello;
            colore = "Indifferente";
            anni = 1;
            pagata = 1;
        }
```

In questo caso abbiamo 2 valori che vengono passati (marca e modello) e gli altri che rimangono di default. Lo stesso risultato lo si poteva ottenere inizializzando direttamente i valori di default nei membri

Prima Versione

```
using System;

class costruttore
{
    public static void Main(string[] args)
    {
        automobile MiaAutomobile = new automobile (

"audi",
"coupe",
"nero",
12,
65000
);
        MiaAutomobile.risposta();

        Console.WriteLine("------------------");

        automobile MiaAutomobile2 = new automobile (

"Alfa Romeo",
"Giulia"
                                                );
        MiaAutomobile2.risposta();
    }
}
public class automobile
{
    private string marca;
     private string modello;
     private string colore;
     private byte anni;
     private float pagata;

        public automobile(
            string varmarca,
            string varmodello,
            string varcolore,
            byte varanni,
            float varpagata
            )
        {
            marca = varmarca;
            modello = varmodello;
            colore = varcolore;
            anni = varanni;
            pagata = varpagata;
        }
        public automobile(
            string varmarca,
            string varmodello
            )
        {
            marca = varmarca;
            modello = varmodello;
            colore = "Indifferente";
            anni = 1;
            pagata = 1;
        }
    public float valorenuovo()
    {
        return pagata - anni * 100;
    }
    public void risposta()
    {
        Console.WriteLine("marca:" + marca );
        Console.WriteLine("modello: + modello );
        Console.WriteLine("colore: " + colore );
        Console.WriteLine("pagata: " + pagata );
        Console.WriteLine("anzianità: " + anni );
        Console.WriteLine("Valore: " + valorenuovo()
);
    }
}
```

Seconda Versione

```
using System;

class costruttore
{
    public static void Main(string[] args)
    {
        automobile MiaAutomobile = new automobile (

"audi",
"coupe",
"nero",
12,
65000
);
        MiaAutomobile.risposta();

        Console.WriteLine("------------------");

        automobile MiaAutomobile2 = new automobile (

"Alfa Romeo",
"Giulia"

);
        MiaAutomobile2.risposta();
    }
}
public class automobile
{
    private string marca;
     private string modello;
     private string colore = "indifferente";
     private byte anni = 1;
     private float pagata = 0;

        public automobile(
            string varmarca,
            string varmodello
            )
        {
            marca = varmarca;
            modello = varmodello;
        }

    public float valorenuovo()
    {
        return pagata - anni * 100;
    }
    public void risposta()
    {
        Console.WriteLine("marca:" + marca );
        Console.WriteLine("modello: + modello );
        Console.WriteLine("colore: " + colore );
        Console.WriteLine("pagata: " + pagata );
        Console.WriteLine("anzianità: " + anni );
        Console.WriteLine("Valore: " + valorenuovo()
);
}
```

This è una parola chiave molto importante che tra le altre caratteristiche ha quella di poter essere utilizzata come istanza corrente di una classe.
Modifichiamo la classe automobile come segue:

```
public class automobile
{
     private string marca;
     private string modello;
     private string colore;
     private byte anni;
     private float pagata;

        public automobile(
            string marca,
            string modello,
            string colore,
            byte anni,
            float pagata
            )
        {
            this.marca = marca;
            this.modello = modello;
            this.colore = colore;
            this.anni = anni;
            this.pagata = pagata;
        }


    public float valorenuovo()
    {
        return pagata - anni * 100;
    }
    public void risposta()
    {
        Console.WriteLine("marca:.........: " + marca );
        Console.WriteLine("modello:.......: " + modello );
        Console.WriteLine("colore:........: " + colore );
        Console.WriteLine("pagata:........: " + pagata );
        Console.WriteLine("anzianità:.....: " + anni );
        Console.WriteLine("Valore attuale:..: " + valorenuovo() );
    }
}
```

Vediamo che in questo modo posso usare gli stessi nome sia per i parametri della classe che per i costruttori.
Per capirci all'interno del costruttore `this.marca` rappresenta il campo marca della classe mentre `marca` è il parametro passato al costruttore.

OVERLOAD DEI METODI

L'overloading dei metodi permette di usare lo stesso nome per diversi metodi all'interno delle classi (ricordiamo che Un metodo è un blocco di codice che contiene una serie di istruzioni).
Si può fare questa operazione solo nel caso che la signature (firma) sia diversa, quando per firma si intende un differente numero o tipo di parametri.

```
    public void risposta(int valore)
    {
        ...
    }
```

```
    public void risposta(string valore)
    {
        ...
    }

    public void risposta(string valore, int numero)
    {
        ...
     }
```

Attenzione: non è possibile definire due metodi con lo stesso nome e gli stessi parametri anche se il valore di ritorno è diverso.
Uno degli usi più comuni dell'overloading è quello di gestire i parametri di default per i parametri dei metodi.
Nell'esempio del costruttore potremmo scrivere:

```
public float valorenuovo()
    {
        return pagata - anni * 100;
    }

public float valorenuovo(int svalutazione)
    {
        return pagata - anni * svalutazione;
    }
```

In questo caso avrò due diversi risultati per valorenuovo in base al passaggio o meno di un valore al metodo.
In questo modo posso scegliere se richiamare il metodo passando un parametro o non passandolo.

```
MiaAutomobile.valorenuovo();
MiaAutomobile.valorenuovo(250);
```

GET / SET

Una buona norma, salvo casi particolari, è quella di definire i dati membro di una classe come private.
Tra i vari metodi per accedervi il più sicuro è quello di usare le proprietà definite da C# **get** e **set**.
Una proprietà è costituita da un modificatore, un tipo, un nome e dall'implementazione per l'accesso (get) o l'accesso (set).

```
        public float pagata
        {
            get { return this.pagata; }
            set { this.pagata = value; }

        }
```

Nell'esempio abbiamo usato una proprietà per accedere al dato membro pagata
Quindi posso accedere ai dati membro tramite:

```
    float valoreN = MiaAutomobile.pagata; // leggo il valore e lo metto nella variabile valoreN//
    MiaAutomobile.pagata = 25000 // inserisco un valore //
```

LE CLASSI STATICHE

Applicare la parola static ad una classe ha la capacità di modificarne alcune proprietà:

- Impedisce la possibilità di crearne un'istanza (oggetti) e rimuove il costruttore predefinito.
- Ne impedisce l'ereditarietà (vedremo dopo cosa si intende)
- Per accedere ai membri della classe, non essendoci oggetti, si utilizza il nome della classe stessa.
- Tutte le proprietà e i metodi della classe devono essere statici.

Questo tipo di classe prendere il nome di **classe statica**.
Una classe statica può essere usata come un contenitore adatto per insiemi di metodi che funzionano solo sui parametri di input e non devono ottenere o impostare campi di istanza interni. Ad esempio, nella libreria di classi .NET Framework, la classe statica System.Math contiene diversi metodi che eseguono operazioni

matematiche, senza alcun requisito per archiviare o recuperare dati univoci per una particolare istanza della classe Math.

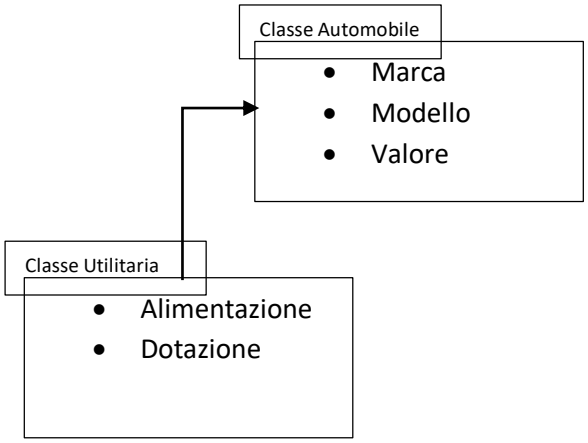
```
using System;
public static class clstatica
{
    // Non c'è un costruttore quindi non ci può essere un istanza
    public static int contatore { get; set; }
    public static int GetCount()
    {
        return contatore;
    }
}

public class statica
{
    static void Main(string[] args)
    {
        clstatica.contatore++;
        Console.WriteLine(clstatica.GetCount());
    }
}
```

EREDITARIETÀ DELLE CLASSI

L'ereditarietà è uno dei concetti fondamentali della programmazione ad oggetti e quindi anche in C#.
Grazie all'ereditarietà una classe può ereditare i dati membro ed i metodi di un'altra classe.
Per creare una classe derivata è sufficiente far seguire al suo nome il nome della classe base preceduto da due punti come nell'esempio seguente:

```
class utilitaria : automobile
```

Vediamo di spiegarci meglio: abbiamo due classi: la classe automobile e la classe utilitaria. Le auto utilitarie è un tipo di auto nella realtà e allo stesso modo la classe utilitaria appartiene alle automobili….



Ogni utilitaria sarà quindi anche un'automobile con in più due proprietà: alimentazione e dotazione.
Posso quindi istanziare MiaAutomobile come sempre

```
utilitaria MiaAutomobile = new utilitaria();
```

e devo passare i relativi argomenti.

```
utilitaria MiaAutomobile = new utilitaria(
                                "Fiat",
                                "500",
                                7500,
                                "GPL",
                                "Base"
      );
```

Vediamo un esempio completo…..

```
using System;
class  eredita
{
    public static void Main()
    {
      utilitaria MiaAutomobile = new utilitaria(
                                 "Fiat",
                                 "500",
                                 7500,
                                 "GPL",
                                 "Base"
      );

    MiaAutomobile.risposta();

    }
}

public class automobile
{
     public string marca;
     public string modello;
     public float valore;

      public automobile (
                        string marca,
                        string modello,
                        float valore
                       )
    {
        this.marca=marca;
        this.modello=modello;
        this.valore=valore;
    }


}

class utilitaria : automobile
{
        private string alimentazione;
        private string dotazioni;

        public utilitaria (
                                 string marca,
                                 string modello,
                                 float valore,
                                 string alimentazione,
                                 string dotazioni
                                 ) : base (marca, modello, valore)
        {
            this.alimentazione=alimentazione;
            this.dotazioni=dotazioni;

        }

         public void risposta()
    {
        Console.WriteLine("marca:.....: " + marca );
        Console.WriteLine("modello:.....: " + modello );
        Console.WriteLine("colore:.....: " + valore );
        Console.WriteLine("pagata:.....: " + alimentazione );
        Console.WriteLine("anzianità:.....: " + dotazioni );

    }
}
```

Come si vede nella classe utilitaria che eredita da automobile i dati, marca, modello e valore vendono dichiarati ma solo tramite il comando ": base" che vengono passati alla classe che eredita. Se noi togliessimo

```
: base (marca, modello, valore)
```

Otterremmo un errore di compilazione
Ricapitolando:

- Creiamo la classe base automobile con i membri privati o pubblici;
- creiamo un costruttore (vedi capitoli precedenti);
- Creiamo la classe che eredita;
- Creiamo un suo costruttore e, con l'aggiunta del comando "base", prendiamo i membri dalla classe principale;

Facile?

POLIMORFISMO

Nella programmazione ad oggetti il polimorfismo è il terso dei paradigmi fondamentali.
Mediante il polimorfismo è possibile definire diversi comportamenti per uno stesso metodo per le classi derivate: due classi possono contenere lo stesso metodo, con le stesse firme, ma con un comportamento diverso.
Per realizzare il polimorfismo si usano due parole chiave particolari: **Virtual e Override**.
Prendiamo quest'esempio di codice:

```
public class automobile
{
     ….

     public virtual float costo()
    {
       return valore+10;

    }



}

class utilitaria : automobile
{
       ….

     public override float costo()
    {
       return valore+300;

    }
```

Vediamo che abbiamo lo stesso metodo per due classi , con la stessa firma ma che eseguono due operazioni differenti: se automobile è normale il valore è uguale al valore piu10 (Euro?!), se utilitaria è uguale a valore più 300!

TIRIAMO LE SOMME

Lo scopo di questo tutorial, fin dal suo concepimento è stato quello di introdurre i concetti base di una programmazione moderna.
Se andassimo avanti diventerebbe un tutorial sul C#, per cui riteniamo giusto fermare qua le spiegazioni sul linguaggio che ci è servito per capire i concetti di Classe, ereditarietà, polimorfismo, variabili, ecc.
Chi volesse approfondire il linguaggio C# può tranquillamente consultare la guida in linea microsoft o uno dei tanti manuali disponibil o qualcuna delle varie riviste in circolazione sulla programmazione.
I concetti introdotti possono essere applicati sui principali linguaggi di programmazione con solo qualche modifica alla sintassi o a qualche regola.
Speriamo che il linguaggio utilizzato in queste pagine non sia risultato troppo difficile da capire e che via abbia introdotto al modno della programmazione in maniera semplice ed incuriosito nel voler approfondire l'argomento.
Ringraziamo tutti voi che ci avete seguito fino a qua.

CAPITOLO IV

ALLEGATI: PRINCIPALI COMANDI DOS A RIGA DI COMANDO.

CD: permette di visualizzare il nome della directory corrente o di spostarsi in un'altra

```
       CD /D [unità:][percorso]
       CD ..
  ..    Indica che si desidera passare alla directory padre.
```

COPY: copia file in un'altra posizione

```
       COPY [/D] [/V] [/N] [/Y] [/-Y] [/Z] [/L] [/A | /B] origine [/A | /B]
            [+ origine [/A | /B] [+ ...]] [destinazione [/A | /B]]

         origine       Indica i file da copiare.
         /A            Indica un file di testo ASCII.
         /B            Indica un file binario.
         /D            Consente la creazione del file di destinazione come
                       decrittografato destinazione  specifica la directory
                       e/o il nome dei nuovi file.
         /V            Verifica che i nuovi file siano stati copiati correttamente.
         /N            Utilizza nomi file brevi, se disponibili, durante
                       la copia di file con nome lungo.
         /Y            Non chiede la conferma prima di sovrascrivere
                       un file di destinazione esistente.
         /-Y           Chiede la conferma prima di sovrascrivere
                       un file di destinazione esistente.
         /Z            Copia i file dalla rete in modalità riavviabile.
         /L            Se l'origine è un collegamento simbolico, copia nella
                       Destinazione il collegamento anziché il file rappresentato
                       dal collegamento.
```

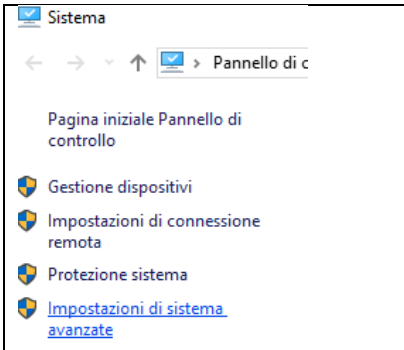
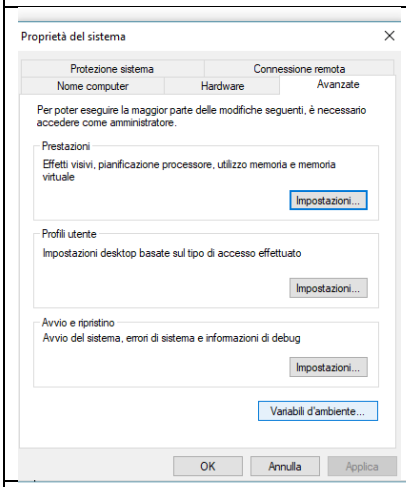
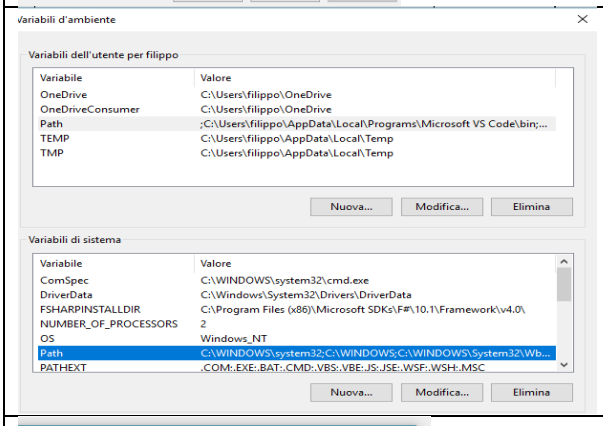
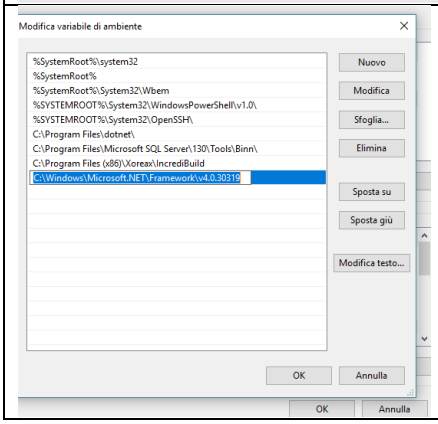
DIR: Mostra l'elenco dei file e delle directory presenti nella directory

```
DIR [unità:][percorso][nomefile] [/A[[:]attributi]]
  [/P] [/Q] [/R] [/S] [/X]
  [unità:][percorso][nomefile]
              Specifica unità, directory e/o file da elencare.

  /A          Visualizza i file con gli attributi specificati.
  attributi    D  Directory                  R  File sola lettura
               H  File nascosti              A  File archivio
               S  File di sistema            I  File non indicizzati
               L  Reparse point             - Prefisso per negare l'attributo
  /P          Mostra elenco una schermata per volta.
  /Q          Mostra il proprietario del file.
  /R          Visualizza flussi di dati alternativi del file.
  /S          Mostra file della directory e delle relative sottodirectory.
  /X          Mostra i nomi brevi generati per i nomi con formato lungo.
              Il formato è come quello di /N con nomi
              abrevi prima del nome esteso. Se non ci sono nomi brevi
              vengono visualizzati spazi bianchi.
  /4          Mostra l'anno in formato 4 cifre
```

ALLEGATI: IMPOSTARE IL PATH DI WINDOWS.

Andare su

`Pannello di controllo\Tutti gli elementi del Pannello di controllo\Sistema`

	Selezionare Impostazioni di sistema avanzate
	Sul tab Avanzate selezionare Variabili d'ambiente
	Su variabili di sistema selezionare Path e quindi modifica
	Selezionare nuovo e quindi scrivere il percorso del frameworks (es: C:\Windows\Microsoft.NET\Framework\v4.0.30319) dove si trova il file csc.exe nella riga dove mi si posiziona il cursore.

Confermare i cambiamenti ed uscire. Se il caso riavviare il sistema!

ALLEGATI: LE VARIABILI.

Tipo	Dimensione (bit)	Valori
sbyte	8	da -128 a 127
byte	8	da 0 o 255
short	16	da -32.768 a 32.767
ushort	16	da 0 a 65.535
char	16	Un carattere Unicode da 0 a 65.535
int	32	da -2.147.483.648 a 2.147.483.647
uint	32	da 0 a 4.294.967.295
long	64	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
ulong	64	0 a 18.446.744.073.709.551.61
float	32	7 cifre da 1,5 x 10-45 a 3,4 x 1038
double	64	da 5,0 x 10^{-324} a 1,7 x 10^{308}
decimal	128	28-29 cifre decimali da 1,0 x 10^{-28} a 7,9 x 10^{28}
bool	32	true o false (vero o falso)
string	16 * lunghezza stringa	

- nell'uso delle variabili non si deve mai usare una parola riservata del linguaggio;
- se il linguaggio è case sensitive (come in C#) attenzione alle maiuscole e minuscole;
- si pò usare il simbolo _ (underscore) ma non caratteri speciali come: &, ?, !.