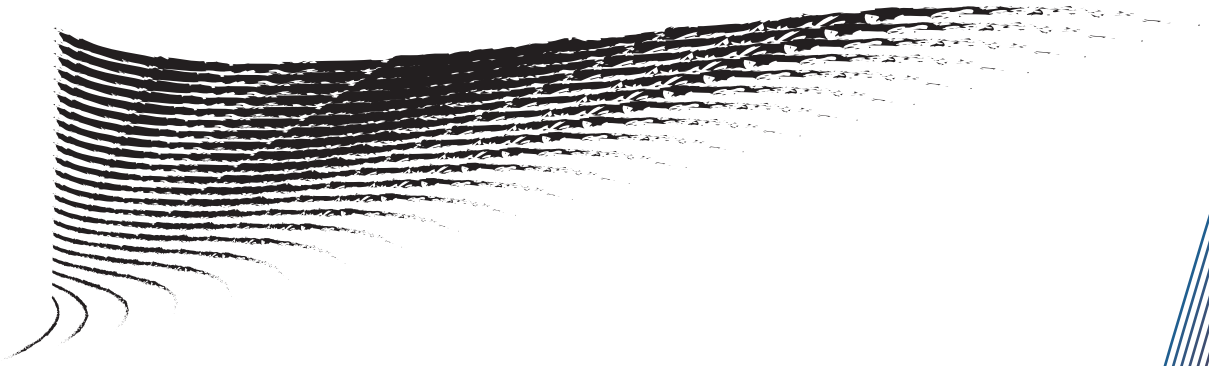
INTRODUZIONE ALLA PROGRAMMAZIONE CON L'USO DI C#



(c) Filippo Brunelli
www.filoweb.it

INDICE

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 (" ").

Note: L'uso di **.com** come estensione era usato nei vecchi computer, in particolare nel sistema operativo DOS erano programmi non rilocabili di dimensione non superiore a 64 K. Oggigiorno la memoria viene gestita in modo "virtuale" ed il concetto di "rilocazione" è del tutto diverso da quello del vecchio DOS i file COM non esistono più per gli applicativi

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 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"`

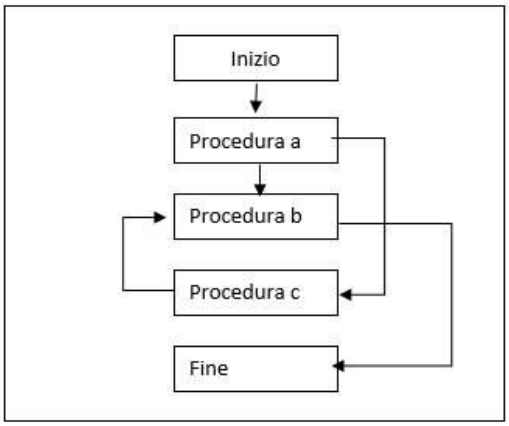
Come si vede, scrivere un programma in questo modo semplifica, e di molto, sia i passaggi necessari sia un eventuale modifica.
Negli ultimi anni si è sviluppato un nuovo approccio ai linguaggi di programmazione, l'uso di ***bytecode***.
Il bytecode è un linguagio intermedio: il codice sorgente viene 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* di Sun Microsystemsc che "traduce" il bytecode (solitamente un file .class) in linguaggio macchina tramite una procedura chiamata *just in time*.
Il vantaggio del bytecode di Sun è che un programma scritto in Java sarà eseguibile su qualsiasi macchina anche se a scapito della velocità di elaborazione (che dipende dalle caratteristiche del computer).
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");`
	`  }`
	`  }`

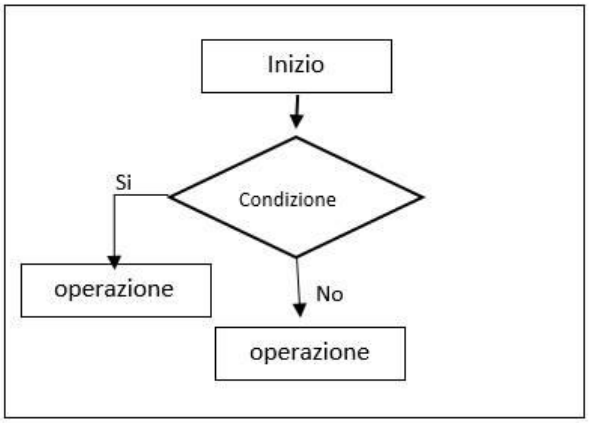
TECNICHE DI PROGRAMMAZIONE: PROCEDURALE, STRUTTURATA E AD OGGETTI

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.



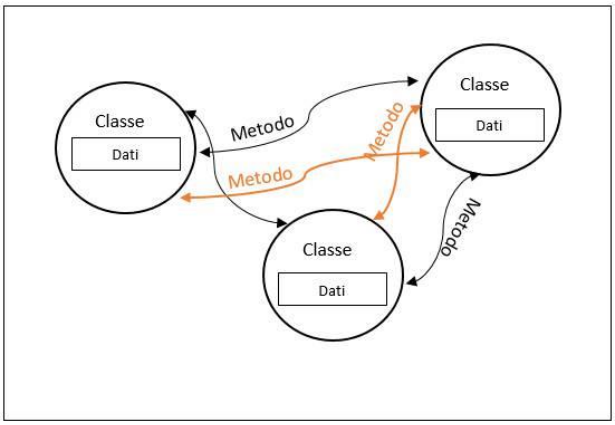
La **programmazione Procedurale** è una delle prime tipologie di programmazione utilizzate. Fa ricorso a diversi "salti" all'interno del programma per richiamare subroutine e scambiare con loro i dati. Il codice viene scritto passo dopo passo ed eseguito partendo dall'alto (la prima riga) e scendendo.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

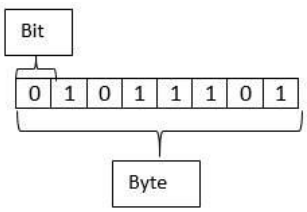
Esempi di linguaggi che usano questo tipo di programmazione sono il Pascal, il C, l'Ada, il Fortran, il Visual Basic fino alla versione 6.



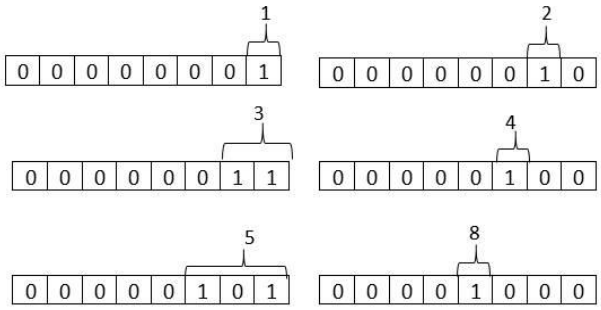
I linguaggi di **programmazione ad oggetti** sono, ad oggi, i più utilizzati. In questo tipo di programmazione i programmi sono scritti come una serie di "oggetti" che comunicano tra loro. Questi oggetti prendono il nome di Classi e sono visti come un "contenitore" che combina i dati con le operazioni (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.



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 senza 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.
Due esempi di costante (in C# e in Java) sono:

```
const double Pi =3.141592653589;
final int 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: 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… pn) 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" la marca, il modello ed il colore.
Da questa classe creo poi l'oggetto MiaAutomobile avrà le "prorietà" della classe automobile ma le caratteristiche della mia auto:

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

Come nella vita reale, o quasi, quindi.
Fin qua tutto chiaro? Spero di si!
Ogni classe contiene degli attributi (ovvero i dati) e dei metodi (le funzioni che manipolano i dati).
Vediamo, ad esempio, come appare una classe nel linguaggio C#, nella quale creo un metodo, che all'inserimento dei dati, restituirà la scritta "la tua auto è bella":

```
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.");
    }
}
```

Ricapitolando, quindi, la mia classe è la classe **automobile**, mentre il mio oggetto, che genero da lei, si chiama **MiaAutomobile**, e viene creato (in C#) tramite:

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

Le proprietà dell'oggetto sono la marca, il modello ed il colore mentre la risposta "La tua auto è bella" viene chiamato "metodo".
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.

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, 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, di compilare anche delle librerie (**.DLL**) ovvero 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.
Apriamo il nostro editor di testo preferito (no Word, Wordpad o altri editor simili che formattano il testo!, si notepad, visual studio code, notepad+, ecc.) 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 e scriviamo:

> **Note:** L'esempio utilizza il frameworks 4.5, nel caso si utilizzi un'altra versione utilizzare il percorso relativo a quella versione. Volendo si può inserire il percorso del frameworks direttamente nei path di windows.

```
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.

IL PRIMO PROGRAMMA: SPIEGAZIONE

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

```
using System;
```

Organizzazione dei programmi c# usa i ***NameSpaces***. La direttiva **using** serve per facilitare l'uso di questi namespaces che sono l'equivalente dei pacages di Java.
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 diversi!

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 chiamato 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 il 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 dovessi, ad esempio, applicare static ad una classe non potrò usare parola chiave new per creare una variabile del tipo di classe (vedremo di approfondire questo concetto più avanti).
Dentro il metodo Main() vediamo che la prima cosa che facciamo è il creare un oggetto *MiaAutomobile* dalla classe automobile:

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

A questo punto è doveroso fare un salto a vedere la classe automobile e ad analizzarla.
Innanzi tutto vediamo che vengono dichiarate 3 variabili del tipo string (string sta per stringa ed indica variabili dove i valori sono di 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.

Per carpire cosa vuol dire questo 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 una serie di errori:

```
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 che, ogni volta che cambiamo auto, dobbiamo anche dover ricompilare tutto.
Cosa fare allora?
Semplice: *passiamo i valori al nostro programma*!
Modifichiamo il punto d'ingresso del programma in questo modo:

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

Cosa vuol dire questa modifica?
Con l'aggiunta tra parentesi della scritta **string**[] **args** diciamo al compilatore che il programma dovrà ricevere dei parametri di tipo stringa quando viene lanciato.
Dovremo poi modificare anche l'assegnazione delle variabili del tipo di auto (marca modello e colore) per poter ricevere i dati in ingresso:

```
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.
Vediamo che i dati sono letti in maniera sequenziale ed assegnati al nostro programma.

> **Note:** se avessi scritto Main(int[] args) i valori sarebbero stati numeri interi e non stringhe

IL PRIMO PROGRAMMA: FACCIAMO DOMANDE E OTTENIAMO RISPOSTE!

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, è giunto quindi il momento di fare un passo avanti e lasciare che sia il programma a chiederci quale marca, modello e colore di macchina ci piaccia.
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 e non ci resta altro che creare le operazioni che ci interessano, in questo caso, le principali 5 operazioni: Somma (+), Sottrazione (–), moltiplicazione (*), divisione (/) e modulo (%) ovvero il resto della divisione.

Da notare 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);
```

Finiamo con i commenti!
I **commenti** sono importanti quando si sviluppa un programma perché aiutano a capire chi legge il programma (e spesso anche chi lo fa) cosa succede in quel punto.
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

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, 0 o 1).

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.

- **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…*
- **OR** (||) 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….*
- **NOT** (!) 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

```
int a = 10;
int b = 5;
int c= a++ +b;
int d= ++a +b;
```

Nella prima somma il valore di **c** si ha dal valore di **a** che viene sommato dopo le altre operazioni.
Nel secondo esempio per il valore di **d** 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.

Vediamo un esempio che ci illustra la comparazione di due valori:

```
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);
    }
}
```

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 sua 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 tramite **While**, **Do… while** e **For**.

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);
       }
    }
 }
```

Assegnamo il valore 10 alla variabile i e, finchè i>0, la scriviamo su di una riga e ne diminuiamo il valore ad ogni ciclo.

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

DOVE ERAVAMO RIMSTI

Nella lezione precedente abbiamo introdotto i concetti base della programmazione, le classi e abbiamo iniziato a compilare programmi scritti in C#.
Vediamo adesso di approfondire alcuni concetti sulle classi e ampliarne la nostra conoscenza….

LE CLASSI DERIVATE E L'EREDITARIETÀ

L'**ereditarietà** è uno dei concetti base della programmazione orientata agli oggetti; tramite questa proprietà diamo la possibilità ad una classe (detta *classe derivata*) di ereditare da un'altra (la *classe base*) le variabili, i metodi, le proprietà e di estendere.
Se torniamo al nostro esempio delle automobili possiamo avere una classe generale automobile e una (o più) classi che "ereditano" dalla classe principale le proprietà e ne aggiugono altre.
Vediamo un esempio:

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

Nell'esempio abbiamo la **classe utilitaria** che eredita dalla classe base (automobile) le proprietà e le estende aggiungendo alimentazione e dotazioni.

In C# la relazione di ereditarietà tra le classi si esprime usando i due punti (:)

```
class utilitaria : automobile
```

Una classe derivata può avere una sola classe di base diretta ma le sue proprietà sono transitive: se la classe C viene derivata dalla classe B e la classe B viene derivata dalla classe A, la classe C eredita i membri dichiarati nella classe B e nella classe A; se non voglio che questo accada devo dichiare la classe da cui eredita **sealed**.
Possiamo considerare una classe derivata come una "specializzazione" della sua classe base!
Per accedere ai membri della classe di base dall'interno di una classe derivata utilizziamo la parola chiave **base ()** dove all'interno della parentesi mettiamo i membre della classe base.

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

Vorrei che ci soffermassimo ancora un attimo su queto programma. Abbiamo dichiarato le variabili presenti nella classe automobile come private. Per fare in modo che sia possibile utilizzarle al difuori della classe utilizzaimo quello che viene chiamato **costruttore**.
Normalmente se non si specifica un costruttore per la classe, C# ne definisce uno per impostazione predefinita che crea istanze dell'oggetto e imposta le variabili dei membri sui valori predefini. In parole semplici, quando io creo un oggetto (tramite la parola chiave new) da una classe non faccio altro che utilizzare un costruttore (istanzio una classe).
Un costruttore è **un metodo** il cui nome è identico al nome del tipo relativo. La firma (ovvero i parametri che sono all'interno delle parentesi) di questo metodo include solo il nome metodo e l'elenco dei parametri, ma non include un tipo restituito:

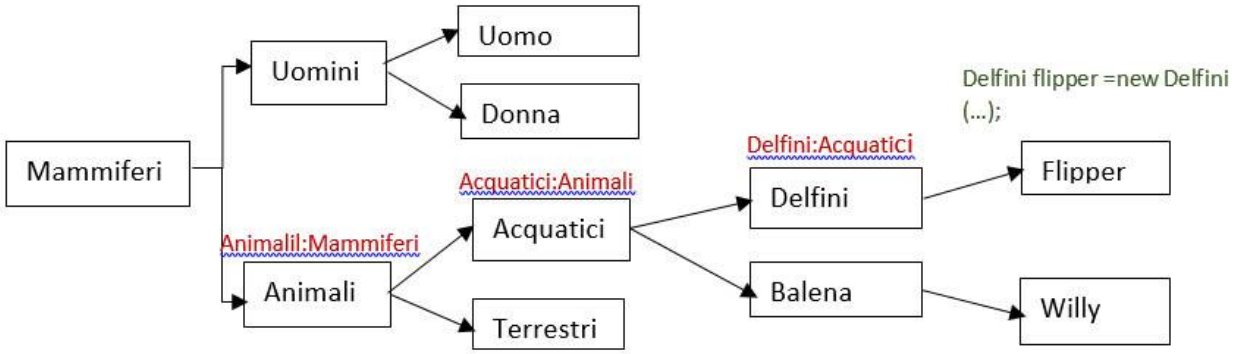
```
    public automobile ( string marca,  string modello,  float valore )
```

Quando scrivo ***automobile*** *MiaAutomobile = new* ***automobile*** *();* non faccio altro che utilizzare il costruttore.
Nel nostro esempio, quindi, tramite il costruttore automobile creiamo un metodo che ci permette di utilizzare le proprietà della nostra classe anche se sono definite private.

In fine tramite la parola chiave **.this** facciamo riferimento (in questo caso) all'istanza corrente della classe

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

Nella creazione del mio costruttore devo inserire le proprietà che voglio passare all'interno delle parentesi e poi richiamarli nel costruttore stesso.
Il concetto di eretità è molto importante: pensiamo ad esempio ad una calsse generale che definiamo mammiteri, da questa deriviamo le classi uomini e animali, e da uomini possiamo derivare uomo e donna, da animali terrestri e acquatici, ecc.



IL POLIMORFISMO

La parola polimorfismo vuol dire più forme ed è proprio questo che permette di fare ad una medesima entità astratta; mediante il polimorfismo in C# (e negli altri linguaggi di programmazione ad oggetti) è possibile definire diversi comportamenti per il medesimo metodo a più classi derivate.
Consideriamo la nostra classe automobile con le sole proprietà: *marca, modello e valore*.

Alla nostra classe aggiungiamo il **metodo costo** che aumenta il valore di 10€.
Dalla nostra classe automobile vogliamo derivare due differenti classi: utilitaria e berlina, con la differenza che nell'utilitaria il costo sarà del valore dell'auto più 300€ mentre per la berlina sarà di 5500€.

Creiamo quindi la nostra classe automobile con il metodo costo come da esempio qua sotto:

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

Notiamo subito che nel metodo costo abbiamo inserito la parola chiave **virtual**.
Tramite virtual comunichiamo al compilatore che le clasi derivate possono creare una differente versione del metodo.

Quando creo la mia classe derivata utilitaria, nella creazione del metodo inserisco la parola chiave override che serve a sovrascrivere il metodo:

```
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 override float costo() { return valore+300; }
}
```

Ugualmente per la classe berlina ed il metodo costo, ma con un valore diverso:

```
class berlina : automobile {
        private string alimentazione;
        private string dotazioni;
        public berlina (string marca, string modello, float valore, string alimentazione, string
dotazioni) : base (marca, modello, valore) {
            this.alimentazione=alimentazione;
            this.dotazioni=dotazioni;
        }
     public override float costo() { return valore+5500; }
}
```

Ecco che quando istanzio le classi avrò risultati diversi a seconda che sia berlina o utilitaria:

```
using System;
  class  Poli {
    public static void Main() {
      utilitaria MiaAutomobile = new utilitaria("Fiat", "500", 7500, "GPL", "Base");
      berlina TuaAutomobile = new berlina("BMW", "M5", 75000, "Diesel", "Luxury" );

        float costo1 = MiaAutomobile.costo();
        string marca1 = MiaAutomobile.marca;
        string modello1 = MiaAutomobile.modello;
        Console.WriteLine(marca1+ " " + modello1 + " - Costo: " + costo1);
```

```
        float costo2 = TuaAutomobile.costo();
        string marca2 = TuaAutomobile.marca;
        string modello2 = TuaAutomobile.modello;
        Console.WriteLine(marca2+ " " + modello2 + " - Costo: " + costo2);
    }
}

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

class utilitaria : automobile {
        private string alimentazione;
        private string dotazioni;
        public utilitaria (string marca, string modello, float valore, string alimentazione, stri
ng dotazioni) : base (marca, modello, valore) {
            this.alimentazione=alimentazione;
            this.dotazioni=dotazioni;
        }

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

class berlina : automobile {
        private string alimentazione;
        private string dotazioni;
        public berlina (string marca, string modello, float valore, string alimentazione, string
dotazioni) : base (marca, modello, valore) {
            this.alimentazione=alimentazione;
            this.dotazioni=dotazioni;
        }

     public override float costo() {
       return valore+5500;
    }
}
```

Note: il valore del metodo costo viene recuperato tramite la creazione di una variabile che legge il valore.
Porevamo anche scrivere *Console.WriteLine()* all'interno del metodo ma volevamo mostrare come recuperare un valore.

LE CLASSI ASTRATTE

Il concetto di classe astratta è molto generico e per capirlo possiamo fare un esempio se creiamo, ad esempio la classe forma dalla quale possiamo derivare le classi quadrato, triangolo, rettangolo, ecc.
La classe astratta è un tipo particolare di classe che non può essere inizializzata (non si può usare la parola chiave NEW) per creare degli oggetti; ne consegue che una classe astratta deve essere per forza derivata e, la classe derivata dalla classe astratta, deve implementare i membri astratti facendone l'**override** o, in alternativa, potrebbe anche non implementarli tutti, ma in questo caso, deve essere a sua volta astratta.
Per creare una classe astratta la si dichiara con la parola chiave **abstract** e, al suo interno, possono essere definite variabili, metodi e proprietà, proprio come abbiamo visto nelle lezioni precedenti dedicate alle classi.

Vediamo un esempio:

```
using System;
class EsempioAstratta {
    public abstract class Forma {
        protected string aNome;
        protected int aLati;
        public Forma(string Nome, int nLati) {
            aNome = Nome;
            aLati = nLati;
        }
        public string Nome {
            get { return aNome; }
        }
        public int nLati {
            get { return aLati; }
        }

        public abstract double Perimetro { get; }
        public abstract double Area { get; }
        public abstract void Faiqualcosa();
        }
    public class Quadrato : Forma {
            private int mLato;
                public Quadrato(int Lato) : base("Quadrato", 4) {
                      mLato = Lato;
       }
                public override double Perimetro  { get { return mLato * 4; } }
                public override double Area { get { return mLato * mLato; } }
                public override void Faiqualcosa() { }
        }
    public static void Main() {
        Console.WriteLine("Creo una un quadrato di lato 7");
        Forma fig1 = new Quadrato(7);
        Console.WriteLine("La figura creata è un " + fig1.Nome);
        Console.WriteLine("Il suo perimetro :" + fig1.Perimetro);
        Console.WriteLine("La sua area :" + fig1.Area);
        Console.ReadLine();
        }
}
```

Come si vede, abbiamo creato la classe astratta che chiamiamo forma e, da lì, creiamo la forma quadrato. Possiamo aggiungere la figura triangolo che fa sempre uso della classe forma:

Vediamo il listato completo:

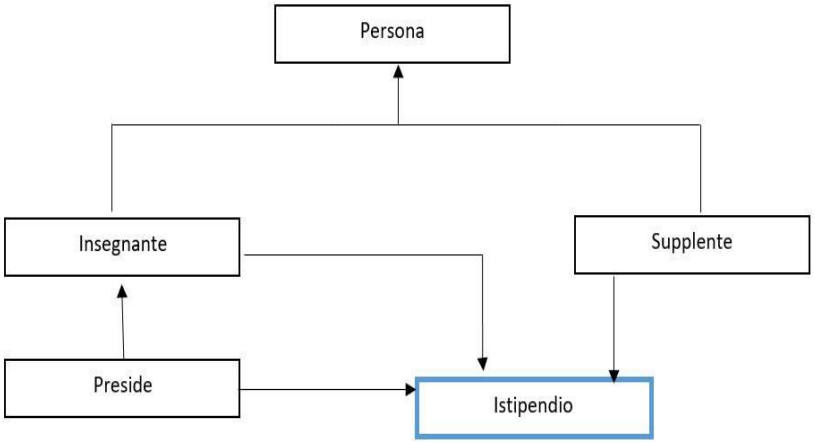
```
using System;

class EsempioAstratta {
    public abstract class Forma {
        protected string aNome;
        protected int aLati;

        public Forma(string Nome, int nLati) {
            aNome = Nome;
            aLati = nLati;
        }
        public string Nome {
            get { return aNome; }
        }
        public int nLati {
            get { return aLati; }
        }
```

```
        public abstract double Perimetro { get; }
        public abstract double Area { get; }
        public abstract void Faiqualcosa();
        }

    public class Quadrato : Forma {
            private int mLato;
                public Quadrato(int Lato) : base("Quadrato", 4)
                    { mLato = Lato; }
                public override double Perimetro
                    { get { return mLato * 4; } }
                public override double Area
                    { get { return mLato * mLato; } }
                public override void Faiqualcosa() {
                }
        }

    public class Triangolo : Forma {
            private double mLato1, mLato2, mLato3;
            private double mBase, mAltezza;
            public Triangolo(double Lato1, double Lato2, double Lato3, double Base, double Altezz
a) : base("Triangolo", 3) {
                mLato1 = Lato1;
                mLato2 = Lato2;
                mLato3 = Lato3;
                mBase = Base;
                mAltezza = Altezza;
            }
            public override double Perimetro {
                get { return mLato1 + mLato2 + mLato3; }
            }
            public override double Area  {
                get { return ((mBase * mAltezza) / 2); }
            }

            public override void Faiqualcosa() {
            }
        }

    public static void Main() {
        Console.WriteLine("Creo un quadrato di lato 7");

        Forma fig1 = new Quadrato(7);
               Console.WriteLine("La figura creata è un " + fig1.Nome);
               Console.WriteLine("Il suo perimetro :" + fig1.Perimetro);
               Console.WriteLine("La sua area :" + fig1.Area);
               Console.WriteLine("Creo un triangolo con lati 3, 4 e 5, base 4 e altezza 3");

        Forma  fig2 = new Triangolo(3, 4, 5, 4, 3);
               Console.WriteLine("La figura creata è un " + fig2.Nome);
               Console.WriteLine("Il suo perimetro :" + fig2.Perimetro);
               Console.WriteLine("La sua area :" + fig2.Area);
              Console.ReadLine();
        }
}
```

Ricapitolando possiamo dire che le classi astratte possono essere considerate come super-classi che contengono metodi astratti, progettate in modo che le sotto-classi che ereditano da esse ne "estenderanno" le funzionalità implementandone i metodi. Il comportamento definito da queste classi è "generico". Prima che una classe derivata da una classe astratta possa essere istanziata essa ne deve implementare tutti i metodi astratti.
Un esempio di classe astratta potrebbe essere una classe 'Database' che implementa i metodi generici per la gestione di un db ma non è specifica per nessun formato particolare e quindi non ha senso istanziarla. Da questa si derivano ad esempio le classi 'MySqlDb' o 'AccessDb' che sanno come aprire il db in questione.
Un caso particolare di classi astratte le **interfacce**, dichiarate tramite la parola chiave interface.

LE INTERFACCE

Le interfacce sono molto simili alle classi astratte, in quanto anch'essa definisce metodi e proprietà astratte ma, a differenza delle classi astratte, non troveremo l'implemetazione di alcun metodo o proprietà.
Possiamo definire un' interfaccia come "contratto" tra chi utilizza una classe e chi la implementa e comprende una serie di *signature* di metodi o proprietà.
Consideriamo il seguente diagramma:



Insegnante, Preside e Supplente sono tutte classi che ereditano dalla classe persona. In più abbiamo che preside è anche un insegnante, mentre supplente non lo è. Se definisco il metodo **calcolaStipendio** nella classe Insegnante questo verrà ereditato anche dalla classe Preside ma non dalla classe Supplente.
Quindi abbiamo una relazione di ereditarietà che mi lega insegnante e preside ma nulla che mi lega supplente anche se quest'ultima ha comunque al suo interno un metodo chiamato calcolaStipendio.
Tramite l'interfaccia Istipendio abbiamo la possibilità di instaurare un legame tra tutte queste classi.
Per fare questo definiamo l'interfaccia Istipendio che contiene il metodo calcolaStipendio, creando quello che viene chiamato un "contratto" che le tre classi devono rispettare.

```
interface Istipendio {
    float calcolaStipendio();
}
```

Mentre le classi avranno la seguente struttura:

```
class Insegnante : Persona, Istipendio {
    //.....
    //.....
    public float calcolaStipendio () {
            return ore * pagaoraria;
        }
}
```

Come si può vedere calcolaStipendio, definito nell'interfaccia Istipendio è stato implementato all'interno di Insegnante
Per implementarla all'interno di Supplente possiamo scrivere

```
class Supplente : Persona, Istipendio {
    //.....
    //.....
    public float calcolaStipendio () {
            return pagagiorno * giornilavorati;
        }
}
```

La prima cosa che notiamo è che nel caso di Supplente il metodo calcolaStipendio restituisce la paga in base ai giorni lavorati e non le ore come nel caso dell'insegnante.

Gli oggetti li creiamo, poi, passando per l'interfaccia:

```
class  interfaccia {
 public static void Main() {
       Istipendio Insegnante = new Insegnante ("Mario","Bianchi","Matematica",25,6 );
       Istipendio Preside = new Preside ("Gina","Lava","Italiano",30,12 );
       Istipendio Supplente = new Supplente ("Eta","Beta","Storia e Filosofia",20,3 );
       Console.writeline("Stipendio insegnante: " + Insegnante.calcolaStipendio());
       Console.writeline("Stipendio preside: " + Preside.calcolaStipendio());
```

```
        Console.writeline("Stipendio supplente: " + Supplente.calcolaStipendio());
    }
}
```

Un'interfaccia può anche definire proprietà oltre che metodi; in questo caso ogni classe che implementa l'interfaccia dovrà contenere un'implementazione della proprietà definita.

```
interface Istipendio  {
    float Stipendio {
        get; set;
    }
}
```

In questo caso la classe insegnante devono obbligatoriamente implementare la proprietà Stipendio

```
class Insegnante : Persona, Istipendio {
 public float stipendio;
    public float stipendio {
        get { return stipendio; }
        set { stipendio=value; }
    }

    public float calcolaStipendio () {
            return ore * pagaoraria;
        }
}
```

Mentre in C# non è consentito derivare da più di una classe è, invece, consentito implementare più di una singola interfaccia semplicemente inserendole una dietro l'altra separate da una virgola.

```
class Insegnante : Persona, Interfaccia1, Interfaccia2 {
   //......
    }
```

NOTE: Per convenzione i nomi delle interfacce dovrebbero iniziare con al I maiuscola.

LE INTERFACCE E LE CLASSI ASTRATTE: DIFFERENZE

Le classi astratte e le interfacce sono molto simili tra loro e possono confondere le idee a chi si avvicina per la prima volta a questi concetti ed al polimorfismo della programmazione ad oggetti. Anche se ad un primo sguardo può sembrare vero vi sono delle differenze che è bene tenere a mente.
La prima è che una classe astratta è una classe che non può essere istanziata, mentre un'intefaccia non è una classe e non ha un'implementazione; al suo interno contiene solo la firma dei metodi che dovranno essere implementati nelle classi che la ereditano.
Un'altra differenza è l'ereditarietà: le interfacce permettono di utilizzare l'ereditarietà multipla in C#, che diversamente non sarebbe possibile con l'uso delle classi solamente.
L'interfaccia è più indicata se si hanno più classi che devono avere alcuni metodi con lo stesso nome, ma non la stessa implementazione. La classe astratta è, invece, indicata per progetti che hanno più classi che condividono alcuni metodi.

LE STRUTTURE

L'idea che sta dietro alle strutture è quella di definire un mezzo che permette di definire tipi dalle caratteristiche simili ai tipi base. Le strutture possono essere considerate una più leggera alternativa alle classi alle quali assomigliano come queste, infatti, anche le strutture possono contenere costruttori, variabili metodi e proprietà.
Le differenze, invece sono che le strutture non possono ereditare da altre strutture o classi e sono rappresentate per valori (come i tipi base).
Per creare una struttura si usa la parola chiave **struct** e può essere creata come una normale classe:

```
using System;
class  Struttura {
    struct Coordinate {
```

```
        public int x, y, z;
        public Coordinate(int px, int py, int pz){
        x = px;
        y = py;
        z = pz;
        }
        public override string ToString() {
            return "Coordinate:" + x + "," + y + "," + z;
        }
    }

    public static void Main() {
        Coordinate c = new Coordinate(3,5,10);
        Console.WriteLine("---- SINGOLI VALORI ------------");
        Console.WriteLine(c.x);
        Console.WriteLine(c.y);
        Console.WriteLine(c.z);
        Console.WriteLine("---- RISPOSTA DAL PROGRAMMA ----");
        Console.WriteLine(c);
       }
}
```

Nella creazione dell'esempio qui sopra vediamo che è possibile inizializzare i membri dello struct accessibili dall'esterno solo usando un costruttore con parametri (vedi capitolo III lezione precedente).

ENUMERAZIONI

Le enumerazioni possono essere considerate un'alternativa elegante all'uso delle costanti e permettono di creare variabili che contengono un numero limitato di valori.
Per creare un'enumerazione si usa la parola chiave **enum** e la sua sintassi è:

[*modificatore d'accesso*] **enum** [*:tipo base*] { *lista dei valori* }

Dove ogni elemento memorizzato è rappresentato da un numero intero (int) e, dove, il primo elemento è 0, il secondo è 1, ecc.
L'uso delle enumerazioni è tra i più svariati, vediamone alcuni esempi:

```
public enum Pulsante
   {
     On = 1,
     Off = 0
   }
```

```
public enum Pulsante : short
   {
     On,
     Off
   }
```

```
enum Giorni
  {
    Lunedì = 0,
    Martedì = 1,
    Mercoledì = 2,
    Giovedì = 3,
    Venerdì = 4,
    Sabato = 5,
    Domenica = 6
  }
```

```
enum Taglia
   {
     S = 0,
     L = 15,
     XL = 20,
     XXL = 25
   }
```

Quando un'enumerazione non definisce dei valori (come nel secondo esempio del pulsante) i valori numerici saranno assegnati automaticamente partendo da 0 (On=0, Off=1).

Consideriamo l'esempio dove usiamo l'enumerazione per creare una corrispondenza tra la taglia di reggiseno italiana e quella Inglese:

```
using System;
  class  enumeratore {
    enum Reggiseni  {
       Seconda = 32,
       Terza = 34,
       Quarta = 36,
       Quinta = 38,
```

```
       Sesta =40
   }

    public static void Main() {
        Reggiseni miataglia;
        miataglia = Reggiseni.Terza;
        int tg = (int) Reggiseni.Terza;
        Console.WriteLine("Nella misura italiana la taglia " + miataglia + " corrisponde alla tag
lia " + tg +" nelle misure UK");
    }
}
```

Importante nell'esempio la sintassi per convertire il valore numerico dell'enumerazione.

```
        int tg = (int) Reggiseni.Terza;
```

È possibile definire delle Enumerazioni dove a una variabile possono essere assegnati più valori contemporaneamente, per fare questo, basta assegnare i diversi valori dell'Enumerazione alle potenze di 2.

```
using System;
 class  enumeratore {
   [FlagsAttribute]
   enum Taglia {
     S = 1,
     L = 2,
     XL = 4,
     XXL = 8,
     FORTE =16
   }
    public static void Main() {
        Taglia granditaglie = Taglia.XXL | Taglia.FORTE;
        Console.WriteLine(granditaglie);
    }
}
```

In questo esempio vediamo che granditaglie mi restituisce come output XXL, Forte. Questo è possibile tramite la parola chiave opzionale [*FlagsAttribute*]; se proviamo a compilare il programma eliminandolo avremo come risultato 24, ovvero la somma dei valori XXL e FORTE (8+16).
Nell'ultimo esempio vediamo come creare una lista di corrispondenze tra la misura di reggiseno italiana e inglese tramite l'uso delle matrici o **array**.
Le parole chiave .**GetValue** e .**GetNames** servono rispettivamente a recuperare il valore numerico e la corrispondenza del nostro enumeratore

```
using System;
class  enumeratore {
    enum Reggiseni {
       Seconda = 32,
       Terza = 34,
       Quarta = 36,
       Quinta = 38,
   }
    public static void Main() {
       int[] valore = (int[])Enum.GetValues(typeof(Reggiseni));
       string[] Misura = Enum.GetNames(typeof(Reggiseni));
        for (int i = 0; i < valore.Length; i++)
        {
            Console.WriteLine(Misura[i] + "=" + valore[i]);
        }
    }
}
```

CAPITOLO IV

VALORI E RIFERIMENTI

Per eseguire le applicazioni in modo ottimizzato C# (come altri linguaggi quali Java, C++, Vb.net ecc.) divide la memoria in due aree denominate **Stack** ed **Heap**.
Semplificando possiamo dire che lo Stack è un'area di memoria utilizzata per l'esecuzione dei metodi e viene utilizzato appunto per passare i parametri ai metodi e per memorizzare provvisoriamente i risultati restituiti dalla loro invocazione.
L' Heap è un'area di memoria utilizzata per l'allocazione degli oggetti istanziati e su di esso opera il garbage collector (raccoglitore di spazzatura) che è in grado di rilevare gli oggetti inutilizzati e distruggerli.
Consideriamo il programma seguente:

```
using System;

   class memoria {
       public static void aumenta (int n) {
           ++n;
       }

    public static void Main() {
      int n = 1;
      aumenta(n);
      Console.WriteLine(n);
    }
}
```

Ci si aspetterebbe che il risultato fosse 2 in quanto viene invocato il metodo aumenta che incrementa il numero n di 1, ma in realtà non accade questo.
L'operazione di incremento non avviene sul numero n che viene memorizzato nello Stack, ma su una copia del valore originale mentre quello che poi mostro in ConsoleWriteLine(n) è il valore dichiarato n = 1.

Quando invoco un metodo, invece, è presente un riferimento al valore e di conseguenza ogni modifica sarà visibile anche all'esterno del metodo.
Lo schema sotto aiuta a capire il concetto

Stack		Heap
Int n = 1; Int x = 3; Int x = y;	Int n = 1; Int x = 3; Int x = y; numero nu = new numero ();	nu

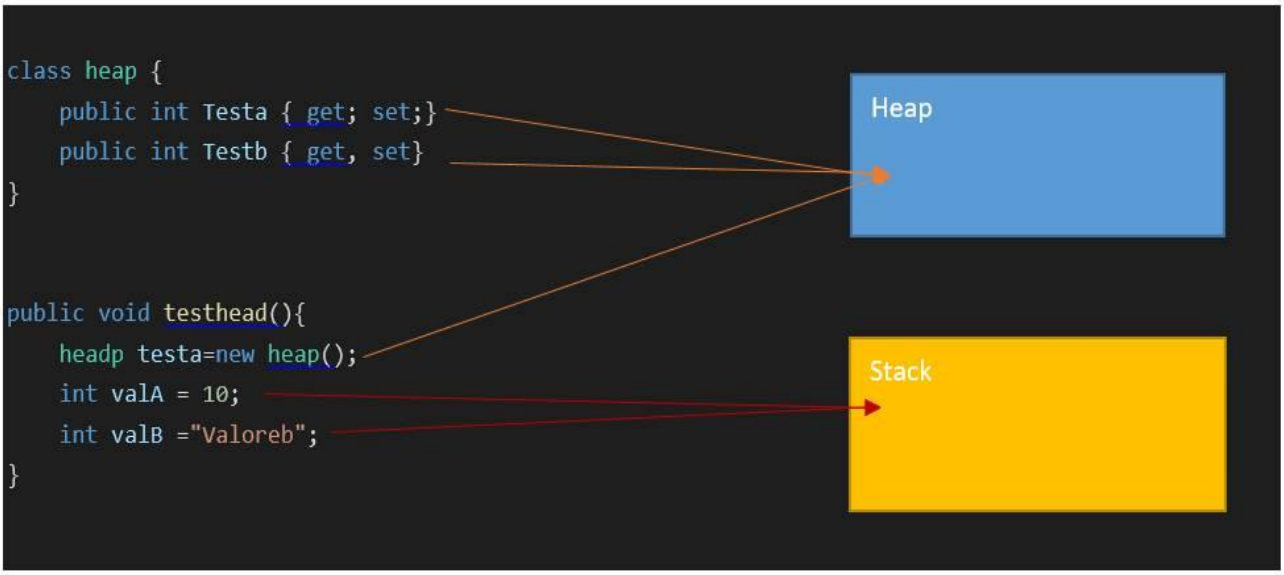
```
using System;

    class memoria {
       public static void aumenta (Numero nu) {
          nu.n++;
       }
        //…
       //…
    }

      public static void Main() {
      Numero nu = new Numero (1);
    }
}
```

Possiamo utilizzare la parola chiave *ref* per definire dei parametri per un metodo che accettano dei riferimenti anziché dei valori.

```
using System;
    class memoria {
       public static void aumenta (ref int n) {
           ++n;
       }

    public static void Main() {
      int n = 1;
      aumenta(ref n);
      Console.WriteLine(n);
    }
}
```

In questo modo il nostro risultato, adesso, sarà 2 dato che, avendo preceduto il parametro di n del metodo cambia con la parola ref abbiamo assegnato al metodo un riferimento alla variabile originale, quindi ogni modifica fatta viene fatta b che si trova in main.
Notiamo infine che, affinché funzioni è necessario che la parola ref sia usata anche quando il metodo viene invocato.

Riassumendo diciamo che *Stack viene utilizzato per l'allocazione della memoria statica e Heap per l'allocazione dinamica della memoria*, entrambi memorizzati nella RAM del computer.
Le variabili allocate nello stack vengono archiviate direttamente nella memoria e l'accesso a questa memoria è molto veloce e la sua allocazione viene gestita al momento della compilazione del programma. Quando una funzione o un metodo chiama un'altra funzione che a sua volta chiama un'altra funzione ecc., l'esecuzione di tutte quelle funzioni rimane sospesa fino a quando l'ultima funzione non ne restituisce il valore. Lo stack è sempre riservato in un ordine **LIFO** (*last in first out*), l'ultimo blocco riservato è sempre il blocco successivo da liberare. Questo rende davvero semplice tenere traccia dello stack, liberare un blocco dallo stack non è altro che regolare un puntatore.
Le variabili allocate sull'heap hanno la memoria allocata in fase di esecuzione e l'accesso a questa memoria è più lenta, ma la dimensione dell'heap è limitata solo dalla dimensione della memoria virtuale. È possibile allocare un blocco in qualsiasi momento e liberarlo in qualsiasi momento. Ciò rende molto più complesso tenere traccia di quali parti dell'heap sono allocate o libere in un dato momento.



PAROLA CHIAVE OUT

La parola chiave **out** offre molte possibilità tra le quali la possibilità che gli argomenti vengono passati per riferimento come la parola chiave ref, ma, differenza di quest'ultima, non richiede l'inizializzazione della variabile prima di essere passato.

```
using System;
      class Outesempio {
      static public void Main() {
        int i;
         Somma(out i);
        Console.WriteLine(i);
    }
    public static void Somma(out int i) {
        i = 30;
        i += i;
    }
}
```

Come vediamo passando il parametro con out non ho dovuto inizializzare i come avrei dovuto fare con ref dove avrei dovuto scrivere

```
int i = 0; //inizializzazione
```

Un altro campo di utilizzo di out è quello di poter restituire più valori di ritorno ad un metodo (che altrimenti ne restituisce solamente uno) come nell'esempio a seguire:

```
using System;
class Outesempio {
    static public void Main() {
        int i;
        string nome;
         Somma(out i, out nome);
        Console.WriteLine(nome + " ha detto " + i);
    }
    public static void Somma(out int i, out string nome) {
        i = 30;
        nome="Filippo";
        i += i;
    }
}
```

IL CASTING

Il casting è un modo per convertire i valori da un tipo a un altro.
Quando la conversione non comporta perdita di informazioni (come nel caso di una variabile *int* in una *double*) viene chiamato **implicito**.

```
using System;
    class CastingImplicito {
        static public void Main() {
          int x = 1;
          double y=x;
          Console.WriteLine(x + "->" + y);
    }
}
```

Quando il casting non è sicuro si chiama esplicito dato che il programmatore dichiara in maniera esplicita che vuole fare l'assegnazione in ogni caso, come se si volesse convertire un Double in intero.
La sintassi in questo caso richiede che il tipo di variabile target venga specificata tra parentesi.

```
using System;
    class CastingEsempio {
          static public void Main() {
            double x = 5.9;
            int y= (int) x;
            Console.WriteLine(x + "->" + y);
    }
}
```

Il cast esplicito può essere fatto solo tra tipi compatibili, quindi

```
        double x = 5.9;
        string y= (string) x;
```

mi restituisce un errore ***CS0030: Cannot convert type 'int' to 'string' 'string'***.

BOXING E UNBOXING

Abbiamo detto che in C# tutto è considerato come un oggetto, questo consente di assegnare ad una variabile un tipo object una qualsiasi altra variabile.
Questa operazione è chiamata di **boxing** è può essere eseguita come nell'esempio ed è sempre eseguita implicitamente:

```
int n = 4;
object Objn = n;
```

Il processo inverso prende invece il nome di unboxing ed è sempre eseguita in maniera esplicita attraverso Cast:

```
int n = 4;
object Objn = n;
int x = (int) Objn;
```

Quando definiamo un metodo che accetta in input un oggetto, ogni volta che viene passato un tipo che rappresenta un valore, il processo di boxing avviene in maniera automatica.
Un esempio di questo processo che abbiamo usato fino ad adesso è stato Console.Writeline.

```
int n = 4;
Console.WriteLine(n);
```

In questo caso la variabile n viene sottoposta ad un processo di boxing prima di essere passata.
Trattare ogni variabile come un oggetto ha come risultato che una variabile allocata sullo stack, potrà essere spostata nella memoria heap tramite l'operazione di boxing, e viceversa, dallo heap allo stack, mediante l'unboxing.
Bisogna però prestare attenzione all'uso di queste conversioni, in quanto boxing e unboxing sono processi onerosi dal punto di vista del calcolo. La conversione boxing di un tipo valore comporta infatti l'allocazione e la costruzione di un nuovo oggetto. A un livello inferiore, anche il cast richiesto per la conversione unboxing è oneroso dal punto di vista del calcolo.

CAPITOLO V

GLI ARRAY COSA SONO E COME SI CREANO

Mentre in altri linguaggi meno evoluti l'array è solamente un insieme di valori dello stesso tipo di una variabile in C#, dove tutto è un oggetto comprese le variabili, sono un insieme omogeneo di oggetti. Essendo un oggetto del tipo System.Array presenta una serie di metodi e proprietà che ne facilitano la gestione come, ad esempio, la ricerca di un valore all'interno di un array monodimensionale ordinato, copiare una parte di un array in un altro, ordinarne gli elementi o invertirne l'ordine e molto altro.
Per dichiarare un array la sintassi è:

tipo [] *nome*;

ad esempio un dichiarando

int [] *elenco*;

Dichiaro che la variabile elenco di tipo intero è un array; in questo modo al momento dell'istanziazione dell'array devo dichiararne dimensioni:

```
        int [] elenco;
        elenco = new int [7];
```

Quindi posso scrivere più semplicemente:

```
        int [] elenco = new int [7];
```

In questo modo abbiamo creato un array che contiene sette interi e posso assegnare i valori come nell'esempio sotto.

```
using System;
   class arrai {
    public static void Main() {
        int[] elenco = new int[7];
            elenco[0] = 1;
            elenco[1] = 2;
            elenco[2] = 3;
            elenco[3] = 4;
            elenco[4] = 5;
            elenco[5] = 6;
            elenco[6] = 7;
            Console.WriteLine(elenco[4]);
    }
}
```

Se volessi creare un array di valori string è sufficiente scrivere:

```
 string[] Anome = new string[3];
          Anome[0]="Filippo B.";
          Anome[1]="Mario C.";
          Anome[2]="Cristina G.";
 Console.WriteLine(Anome[1]);
```

È possibile inizializzare la matrice al momento della dichiarazione, in questo caso non è necessario inizializzare la matrice specificandone il numero di dimensioni che vengono dedotte dai valori passati.

```
   int[] Dichiarati = {5,8,2,4,9} ;
   Console.WriteLine(Dichiarati[1]);
```

Fino ad ora abbiamo visto array composti da una sola dimensione, come una riga, ma si possono creare array a n dimensioni. Questo tipo di array è anche chiamato array rettangolare in quanto ogni riga ha la stessa lunghezza.
Per creare un array multidimensionale al momento dell'istanziazione si deve dichiarare il numero di righe e di colonne oltre che il tipo.

```
int[,] tabella = new int[2,4];
```

Nell'esempio sopra abbiamo creato una tabella composta da 4 colonne per 2 righe. Posso inserire i valori allo stesso modo di come li inserisco in un array monodimensionale.

```
using System;
   class arrai {
    public static void Main() {
        int[,] tabella = new int[2,4] {{2,4,6,1},{10,5,9,0}};
        Console.WriteLine(tabella[1,2]);
    }
}
```

Nell'esempio sopra il risultato sarà 9 (l'array conta da 0 a n-1) ovvero il valore della riga 1 colonna 2

	0	**1**	**2**	**3**
0	2	4	6	1
1	10	5	**9**	0

Posso anche non dichiarare la dimensione dell'array, in questo caso viene creata la sua dimensione in modo automatico all'inserimento dei parametri:

```
  int[] aNoind;
        aNoind = new int[] { 1, 3, 5, 7, 9 };
        Console.WriteLine(aNoind[1]);
```

Nel caso volessi sapere la lunghezza di un array uso il metodo .Lenght e .GetLength.
Il primo recupera la lunghezza dell'array, mentre .GetLength viene utilizzato per la lunghezza del sottoarray.

```
        int[] lungo = new int[4] {10,15,22,8};
        Console.WriteLine(lungo.Length);
```

Oltre alle matrici monodimensionali e multidimensionali esistono anche le matrici irregolari. Queste matrici sono matrici i cui elementi sono costituiti da matrici; questo tipo di matrici sono chiamate anche jaggedArray.

```
using System;
   class arrai {
     public static void Main() {
        int[][] mMatrice = new int[3][];
        mMatrice[0] = new int[3] {1,3,5,};
        mMatrice[1] = new int[2] {2,4};
        mMatrice[2] = new int[2] {6,7};
        Console.WriteLine(mMatrice[0][1]);
     }
}
```

Notiamo che prima di poter usare mMatrice, è necessario che siano stati inizializzati i relativi elementi; ognuno degli elementi è costituito da una matrice unidimensionale di Integer. Il primo elemento è una matrice di 3 Integer, il secondo ed il terzo sono matrice di 2 Integer.

Fino ad ora abbiamo visto array di oggetti noti (int, string), ma cosa succede nel caso di array di oggetti generici, ovvero tipi rappresentati per riferimento? Semplice, visto che ogni elemento contiene un valore Null deve essere istanziato singolarmente. Vediamo di capire bene questo concetto considerando la seguente classe:

```
public class auto {
            private String Marca,Modello;
            public auto (String Marca, String Modello) {
                this.Marca = Marca;
                this.Modello = Modello;
            }
            public string marca { get { return Marca; } }
            public string modello { get { return Modello;}  }
            public override string ToString(){
                return marca + " " + modello;
            }
        }
```

Se volessimo creare un array con questa classe dobbiamo istanziare singolarmente ogni elemento in questo modo:

```
  public static void Main() {
        auto [] macchina = new auto[3];
        macchina[0]= new auto ("Fiat","500");
        macchina[1]= new auto ("Audi","A6");
        macchina[2]= new auto ("Wv","Maggiolino");
           Console.WriteLine(macchina[2]);
    }
```

Ricapitolando quello che abbiamo visto fino ad ora delle matrici sappiamo che:

- Una matrice può essere unidimensionale, multidimensionale o irregolare.
- Il numero di dimensioni e la lunghezza di ogni dimensione sono definiti durante la creazione dell'istanza della matrice. Questi valori non possono essere modificati per la durata dell'istanza.
- I valori predefiniti degli elementi numerici della matrice sono impostati su zero, mentre gli elementi di riferimento sono impostati su null.
- Le matrici sono a indice zero. Una matrice con n elementi viene indicizzata da 0 a n-1.
- Gli elementi di una matrice possono essere di qualsiasi tipo, anche di tipo matrice.

LAVORARE CON GLI ARRAY

Una volta capito cosa sono e come funzionano gli array vediamo come possiamo utilizzarli al meglio.

Scorrere una lista di array

La prima cosa che vogliamo fare con un array è quella di scorrerne i contenuti; per fare questo possiamo utilizzare un ciclo *for* già visto nella precedente lezione (pagina 21) oppure l'istruzione *foreach* che permette di accedere agli elementi di una collection o di un array.
Vediamo entrambi gli esempi:

Ciclo FOR

```
using System;
   class arrai {
    public static void Main() {
        int[] elenco = new int[7];
            elenco[0] = 1;
            elenco[1] = 2;
            elenco[2] = 3;
            elenco[3] = 4;
            elenco[4] = 5;
            elenco[5] = 6;
            elenco[6] = 7;
            for (int i=0; i < elenco.Length; i++) {
                Console.WriteLine(elenco[i]);
            }
    }
}
```

Ciclo FOREACH

```
using System;
   class arrai {
    public static void Main() {
        int[] elenco = new int[7];
            elenco[0] = 1;
            elenco[1] = 2;
            elenco[2] = 3;
            elenco[3] = 4;
            elenco[4] = 5;
            elenco[5] = 6;
            elenco[6] = 7;
            foreach (int i in elenco){
                    Console.WriteLine(i);
                }
    }
}
```

Come si vede la lunghezza è uguale solo che con l'istruzione foreach non devo conoscere la lunghezza del mio array

Ordinare un array

La classe array mette a disposizione due metodi principali per ordinare una lista ***Sort*** e ***Reverse***.
Il primo permette di ordinare gli elementi di un array se questi sono di tipo base:

```
using System;
   class arrai {
    public static void Main() {
        int[] elenco = new int[7] {4,3,2,9,1,8,7};
        Array.Sort(elenco);
              foreach (int i in elenco) {
                   Console.WriteLine(i);
              }
  }
}
```

Reverse, invece inverte l'ordine degli elementi in un array

```
using System;
   class arrai {
    public static void Main() {
        int[] elenco = new int[7] {4,3,2,9,1,8,7};
        Array.Reverse(elenco);
               foreach (int i in elenco) {
                    Console.WriteLine(i);
                }
    }
}
```

Nulla vieta di usare i due metodi insieme prima ordinandoli e poi invertendone l'ordine:

```
     Array.Sort(elenco);
     Array.Reverse(elenco);
```

Copiare un array in un altro

Gli array, essendo dichiarati, sono tipi riferimento (non sono tipi valore) per cui vengono allocati nell'heap quindi non sono dati temporanei (come quelli memorizzati nello stack).
Quindi scrivendo:

```
        int[] elenco = new int[7] {4,3,2,9,1,8,7};
        int[] elenco2 = elenco;
```

Siccome i due array fanno riferimento alla stessa area di memoria, se si modifica un elemento di elenco, si modifica anche l'elemento corrispondente di elenco2, e viceversa. Per eseguire una copia di un array e lavorare sulla copia in modo indipendentente dall'array originale conviene utilizzare uno dei due metodi nella classe *System.Array*: **CopyTo**, **Copy** e **Clone**.

Il metodo **CopyTo** consente di copiare un array in un altro array a partire da un indice indicato.

```
using System;
   class arrai {
    public static void Main() {
        int[] elenco = new int[7] {4,3,2,9,1,8,7};
        int[] elenco2 = new int[elenco.Length];
        elenco.CopyTo(elenco2, 0);
        Console.WriteLine(elenco2[1]);
    }
}
```

Il metodo **Copy** consente di copiare un array in un altro array;

```
using System;
   class arrai {
    public static void Main() {
        int[] elenco = new int[7] {4,3,2,9,1,8,7};
        int[] elenco2 = new int[elenco.Length];
        Array.Copy(elenco, elenco2, elenco.Length);
        Console.WriteLine(elenco2[4]);
    }
}
```

Il metodo .**Clone** serve per eseguire la clonazione di un array; questo metodo restituisce un oggetto e quindi necessita di un casting esplicito

```
using System;

  class arrai {
    public static void Main() {
        int[] elenco = new int[7] {4,3,2,9,1,8,7};
        int[] elenco2 = new int[elenco.Length];
        elenco2 = (int[])elenco.Clone();
        Console.WriteLine(elenco2[3]);
    }
}
```

Si possono eliminare gli elementi di un array tramite il metodo **Array.Clear**; la sintassi di Array.Clear è semlice

Array.Clear (*array*, *posizione di inizio*, *numero di elementi da cancellare*)

```
Array.Clear(elenco, 2, 5);
        Foreach (int i in elenco) {
                    Console.writeline(i);
                }
```

L'output sopra sarà: 4,3,0,0,0,0,0

GLI ARRAYLIST

Una delle llimitazioni più fastidiose degli array la loro dimensione fissa che deve essere dichiarata al momento della loro creazione. Per fortuna ci viene in aiuto la classe ArrayList che supera questa ed altre limitazioni permettendo di creare array che hanno una dimensione che cambia dinamicamente e che possono accedere ad elementi di qualsiasi tipo.

Per poter utilizzare ArrayList bisogna far riferimento al **NameSpace System.Collections**.

```
using System;
using System.Collections;
   class arrai {
    public static void Main() {
        ArrayList lista = new ArrayList();
    }
}
```

Come si vede nell'esempio sopra non abbiamo avuto bisogno di dichiarare la dimensione dell'array lista.
A questa lista posso aggiungere in maniera dinamica valori semplicemente con il metodo .**Add**:

```
        ArrayList lista = new ArrayList();
            lista.Add("Filippo");
            lista.Add(50);
            lista.Add(null);
```

in questo tipo di Array è importante sapere quanti sono gli elementi che lo compongono e per questo ci viene in aiuto il metodo **.Count**

```
ArrayList lista = new ArrayList();
            lista.Add("Filippo");
            lista.Add(50);
            lista.Add(null);
Console.WriteLine(lista.Count);
```

Infine, anche per ArrayList, come per gli array normali è possibile scorrerne gli elementi tramite i ***cicli for*** o ***foreach***.

I principali metodi per ArrayList sono:

.Add(Object**)**	Aggiunge un oggetto all'ArrayList
.BinarySearch(Int32,Int32,Object, IComparer**)**	Cerca un elemento nell'intero elenco ordinato usando l'operatore di confronto predefinito e restituisce l'indice in base zero dell'elemento
.Clear()	Rimuove tutti gli elementi da ArrayList.
.Clone()	Crea una copia superficiale di ArrayList.
.CopyTo(Array**)**	Copia ArrayList o una parte di esso in una matrice unidimensionale.
.CopyTo(Array, Int32**)**	Copia l'intero oggetto ArrayList in un oggetto Array compatibile unidimensionale, a partire dall'indice specificato della matrice di destinazione.
.Reverse()	Come negli array normali inverte l'ordine degli elementi
.Reverse(Int32, Int32**)**	Inverte l'ordine degli elementi in un range
.Sort()	Riordina gli elementi di un array
.ToArray()	Copia gli elementi di ArrayList in una nuova matrice Object.

```
using System;
using System.Collections;
   class arraiList {
     public static void Main() {
        ArrayList lista = new ArrayList();
            lista.Add("Filippo");
            lista.Add("Mario");
            lista.Add("Francesco");

        int indice = lista.BinarySearch("Filippo");
        Console.WriteLine(indice);

        lista.Sort();
        lista.Reverse();

        int Nuovoindice = lista.BinarySearch("Filippo");
        Console.WriteLine(Nuovoindice);
    }
}
```

Nell'esempio sopra abbiamo utilizzato BinarySearch prima su un array diordinato e poi ordinato ed invertito. Vediamo che il risultato dell'indice dove si trova la nostra ricerca cambia a seconda dell'ordinamento desiderato.

Essendo solo una piccola guida orientativa alla programmazione con l'uso di C# per la lista completa dei metodi e molto altro ancora, si consiglia di guardare online il sito web di Microsoft all'indirizzo: https://docs.microsoft.com/it-it/dotnet/api/system.collections.arraylist?view=netframework-4.8.

CAPITOLO VI

LE STRINGHE

Nei moderni linguaggi di programmazione le stringhe rappresentano uno dei tipi dati più importanti.
C# mette a disposizione la classe String per la manipolazione e la gestione delle stringhe.
La classe di String è definita nella libreria di classi di base .NET **System.String** rappresenta il testo come una serie di caratteri Unicode e fornisce metodi e proprietà per lavorare con le stringhe.
Un'altra caratteristica di questa classe è che è una classe sealed, quindi non può essere ulteriormente derivata ed al suo interno implementa le interfacce IComparable, IClonable, IConvertible, IEnumerable ee di conseguenza la classe String ha metodi per clonare una stringa, confrontare stringhe, concatenare stringhe e copiare stringhe. Gli oggetti stringa sono immutabili, ovvero non possono essere modificati una volta creati. Tutti i metodi String e gli operatori C# che sembrano modificare una stringa in realtà restituiscono i risultati in un nuovo oggetto stringa.

CREAZIONE DELLE STRINGHE

La classe String ha diversi costruttori che accettano una matrice di caratteri o byte.
Il codice seguente crea una stringa da un array di caratteri.

```
using System;
   class Stringhe {
    public static void Main() {
        char[] Caratteri = { 'F', 'i', 'l', 'o', 'w', 'e', 'b','.','i','t' };
        string nome = new string(Caratteri);
        Console.WriteLine(nome);
    }
}
```

Sebbene sia valido questo non è certo uno dei metodi più semplici per creare una stringa; molto più semplice è definire una variabile di tipo stringa e assegnare un valore racchiudendolo tra le virgolette come abbiamo fatto fino ad ora quando abbiamo lavorato con le stringhe:

```
     string nome = "Filoweb.it";
     Console.WriteLine(nome);
```

Qualunque cosa io scriva dichiarandola come stringa sarà una stringa, anche se scrivo un numero e quindi, nell'esempio che segue, non posso fare operazioni di alcun tipo su anno o numero senza prima convertirlo nel corrispondente numerico (int o double).

```
        string anno = "2019";
        string numero = "33.23";
```

Un altro metodo i creare stringhe è mediante la concatenazione tramite l'operatore +; questo metodo permette di creare una stringa partendo da più stringhe.

```
    public static void Main() {
        string primolivello = ".it";
        string secondolivello = "https://www.filoweb";
        string completo=secondolivello+primolivello;
        Console.WriteLine(completo);
    }
```

Si può anche creare una stringa da un intero, o qualunque altro oggetto tramite il metodo ToString che converte un oggetto nella relativa rappresentazione di stringa, in modo che sia adatto per la visualizzazione.
Modifichiamo il programma precedente come segue e vediamo il risultato:

```
        string primolivello = ".it";
        string secondolivello = "filoweb";
        string indirizzo="https://www.";
        int numero = 20;
        string completo=indirizzo+secondolivello+primolivello;
```

```
        completo="il sito web " + completo + " è attivo da " +numero.ToString() + " anni";
        Console.WriteLine(completo);
```

Nell'esempio sopra abbiamo convertito il numero intero in stringa e l'abbiamo concatenata.
Certo nell'esempio precedente non era strettamente necessario convertire il numero intero in stringa per poterlo concatenare ma rende l'idea di come funzioni il metodo.
Abbiamo detto che gli oggetti stringa sono immutabili. Nell'esempio sopra la creazione di:

```
        completo="il sito web " + completo + " è attivo da " +numero.ToString() + " anni";
```

permette di crea una nuova stringa che contiene il contenuto delle due stringhe combinato. Il nuovo oggetto viene assegnato alla variabile *completo* e l'oggetto originale assegnato a *completo* viene rilasciato per l'operazione di Garbage Collection perché nessun'altra variabile contiene un riferimento a tale oggetto.

NOTE: la modifica della stringa corrisponde alla creazione di una nuova stringa quindi bisogna prestare attenzione quando si creano riferimenti alle stringhe: se si crea un riferimento a una stringa e quindi si modifica la stringa originale, il riferimento continuerà a puntare all'oggetto originale anziché al nuovo oggetto creato quando la stringa è stata modificata.

AGGIUNGERE, RIMUOVERE, SOSTITUIRE

Abbiamo visto come concatenare due o più stringhe, ma possiamo anche inserire una stringa in una posizione specificata in un'altra stringa. Il metodo che si usa è **String.Insert** che inserisce una stringa specificata in una posizione di indice specificata in un'istanza.

```
using System;
   class Stringhe {
    public static void Main() {
        string nome = "Fippo";
        string mancante = nome.Insert(2, "li");
        Console.WriteLine(mancante.ToString());
    }
}
```

Nell'esempio sopra abbiamo una stringa nella quale aggiungiamo le lettere "li" dopo 2 caratteri.
Visto che il risultato è un oggetto lo dobbiamo convertire in stringa tramite .ToString.

Un altro metodo che può essere utile è **.Remove(***Int32***)**. Questo metodo non rimuove realmente i caratteri dopo *n* caratteri, ma crea e restituisce una nuova stringa senza quei caratteri.

```
mancante = nome.Remove(3);  // "rimuove" tutto quello che segue il 3° carattere
Console.WriteLine(mancante.ToString());
```

.Remove permette anche di "rimuovere" solo una parte dei caratteri di una stringa:

```
nome="Filippo";
mancante=nome.Remove(2,3); // ottengo come risultato Fipo
Console.WriteLine(mancante.ToString());
```

L'ultimo di questo gruppo di metodi che analizziamo è **.Replace(***Char, Char***)** che permette di sostituire uno o più caratteri all'interno di una stringa. In verità, anche qua non viene realmente sostituito ma creata una nuova stringa.

```
        string nome="Filippo";
        string mancante=nome.Replace("i","I");
        Console.WriteLine(mancante.ToString());
```

Come risultato le lettere "i" minuscole verranno sostituite con le lettere "I" maiuscole.

FORMATTAZIONE DI UN NUMERO

Tramite il metodo String.Format è possibile formattare una stringa in maniera personalizzata. Può risultare particolarmente utile nella formattazione di numeri con specifiche caratteristiche come nel caso di Currency (valori monetari), Esponenziali, Percentuali, ecc.

```
using System;
   class Stringhe {
    public static void Main() {
        double d=123456.789;
        String c = String.Format("{0:C}",d); // converte in formato valuta
        Console.WriteLine(c);
        String p = String.Format("{0:p}",d);// converte in formato %
        Console.WriteLine(p);
    }
}
```

L'esempio di sopra converte il numero prima in valuta (123.456,79) e poi in percentuale 12.345.678,90%
La formattazione della valuta (così come i numeri decimale e con virgola) dipendono dalle *impostazioni locali* per lo sviluppo di codice; così se imposto come cultura europea avrò la formattazione con il . per le migliaia e la virgola per i decimali, se imposto una cultura anglosassone sarà viceversa.
Per fare questo uso la classe **.CultureInfo**.
Nel seguente esempio creiamo due diversi outoput per il nostro valore, quello di default (Europeo) e quello Nordamericano.

```
using System;
   class Stringhe {
    public static void Main() {
        System.Globalization.CultureInfo Us = new System.Globalization.CultureInfo("en-US");
        double d=123456.789;
        String c = String.Format("{0:C2}",d); // converte in formato valuta Eu
        Console.WriteLine(c);
        String p = String.Format(Us,"{0:C}",d); // converte in formato valuta Us
        Console.WriteLine(p);
    }
}
```

Note: La classe *.CultureInfo* specifica un nome univoco per ogni lingua, in base allo standard RFC 4646. Il nome è una combinazione di un codice di impostazioni cultura minuscole ISO 639 2-lettera associato a una lingua e un codice di sottocultura maiuscolo ISO 3166 2-lettera associato a un paese o un'area geografica.

Principalmente è si usa **.CultureInfo** quando si devono formattare valute, numeri e date soprattutto se si devono sviluppare programmi che devono essere usati sia in Europa che in Nord America o Regno Unito dove le formattazioni appunto di questi elementi sono differenti.

Il seguente programma aiuta ad individuare la cultura di default sul sistema in uso:

```
using System;
using System.Globalization;
   class Culturamia {
    public static void Main() {
         Console.WriteLine("La tua cultura attuale è {0}.", CultureInfo.CurrentCulture.Name);
    }
}
```

STRINGA DI FORMATO COMPOSTO

Negli esempi precedenti abbiamo visto l'uso di formati composti o segnaposti; una stringa di formato composto e un elenco di oggetti che vengono usati come argomenti
La sintassi è: **{** *indice[n][:formatString]***}**
Le {} sono obbligatorie, l'indice è un numero che parte da 0, e formatString invece identifica la formattazione che viene usata per l'elemento.

```
using System;
   class  SegnaPosto {
    public static void Main() {
         string nome ="Filippo";
         string frase;
         frase = String.Format("Ciao {0} sono le {1:hh:ss}. Ricordati di chiamare {2}", nome, Dat
eTime.Now, "Antonio" );
        Console.WriteLine(frase);
    }
}
```

Come si vede il primo indice viene sostituito dalla variabile nome, il secondo l'ora (che vine o formattata) e per finire un valore.

L'INTERPOLAZIONE DI STRINGHE

Il metodo visto sopra, sebbene semplice ed intuitivo è difficile presenta alcuni inconvenienti. Il primo è che se, per errore, passiamo meno parametri di quanti sono i segnaposti, otteniamo un errore a runtime quindi la compilazione del programma non segnala nulla. Il secondo inconveniente è che se abbiamo molti segnaposti la lettura del codice diventa difficile.
Dalla versione 6 di C# grazie all'interpolazione le cose si fanno più semplici:

```
using System;
   class  SegnaPosto {
    public static void Main() {
         string nome ="Filippo";
         string frase;
         frase= $"Ciao {nome}. Come va?";
        Console.WriteLine(frase);
    }
}
```

Aggiungendo il carattere $ all'inizio della stringa viene abilitata l'interpolazione che permette di inserire le variabili tramite l'uso di parentesi graffe.
Con questo metodo il codice è più leggibile in quanto il segnaposto non dichiara solo il posto ma anche da dove il valore proviene.

ALTRI METODI

I metodi sono molti e come sempre si consiglia di guardare il sito di riferimento Microsoft. Ma noi accenniamo, velocemente ad altri metodi importanti.

.Length permette di sapere la lunghezza di una stringa

```
string nome="Filippo";
       Console.WriteLine(nome.Length);
```

. Substring estrae una sottostringa da una stringa madre. Nell'esempio estra dal 8° carattere fino al 14°

```
using System;
   class Stringhe {
    public static void Main() {
        string s = "https://www.filoweb.it è un buon sito internet";
        string sb = s.Substring(8, 14); // restituisce www.filoweb.it
        Console.WriteLine(sb);
    }
}
```

.ToCharArray() serve per convertire una stringa in una sequenza di caratteri da inserire in un array.

```
string nome="Filippo";
       char[] Caratteri = nome.ToCharArray();
       foreach (char ch in Caratteri) { Console.WriteLine(ch);  }
```

.Equals(*String,String***)** Server per confrontare due stringhe

```
string nome1="Filippo";
string nome2="Pamela";
        if (String.Equals(nome1, nome2))
             Console.WriteLine("Sono uguali");
        else
            Console.WriteLine("Non sono uguali");
```

.Compare(*String,String***)** Serve a comparare la lunghezza di due stringhe e ha tre valori di ritorno:

- ***<0*** la prima stringa è più corta della seconda;
- ***0*** entrambe le stringhe sono uguali;
- ***>0*** la prima stringa è maggiore della seconda;

```
string nome1="Filippo";
string nome2="Pamela";
       Console.WriteLine(String.Compare(nome1, nome2));
```

.ToUpper() e **.ToLower()** convertono, rispettivamente, una stringa in caratteri maiuscoli o minuscoli

```
string nome1="Filippo";
       Console.WriteLine(nome1.ToUpper());
       Console.WriteLine(nome1.ToLower());
```

.Trim() serve per rimuovere gli spazi vuoti da una stringa; si possono usare due metodi:

- **Trim(***Char[]***)**Rimuove tutte le occorrenze iniziali e finali di un set di caratteri specificato in un array dall'oggetto String corrente.
- **Trim()** Rimuove tutti gli spazi vuoti iniziali e finali dall'oggetto String corrente.

```
string nome1="   Filippo è filoweb.it   ";
       Console.WriteLine(nome1.Trim());
string nome2="***Filippo è filoweb.it...";
       Console.WriteLine(nome2.Trim('.','*'));
```

.StartsWith(String**)** e **.EndsWith(**String**)** verificano se una stringa inizia o finisce come un'altra stringa e restituiscono un valore booleano.

```
using System;
   class Stringhe {
    public static void Main() {
        string s = "https://www.filoweb.it è un buon sito internet";
        bool risposta = s.StartsWith("https");
        Console.WriteLine(risposta); // restituisce true
    }
}
```

LE SEQUENZE DI ESCAPE

Le sequenze di escape sono una combinazione di caratteri che, poste in una stringa, sono utilizzate per specificare azioni come il ritorno a capo, le tabulazioni, gli apici ecc. le sequenze di caratteri sono costituite da una barra rovesciata (\ ) seguita da una lettera o da una combinazione di cifre:

\' Virgoletta singola
\" Virgoletta doppia
**** Barra rovesciata
\0 Null
\b Backspace
\n Nuova riga
\r Ritorno a capo
\t Tabulazione orizzontale

```
        string stringa = "Tab 1\tTab2 2\tTab3";
        Console.WriteLine(stringa);
        stringa = "Io sono \"Filippo\"";
        Console.WriteLine(stringa);
        stringa = "Io invece \r\tvado a capo";
        Console.WriteLine(stringa);
```

```
c:\testC>stringhe3.exe
Tab 1   Tab2 2  Tab3
Io sono "Filippo"
        vado a capo
```

CAPITOLO VII

I DATABASE

I sistemi di database sono diventati irrinunciabili per la maggior parte dei programmi: tutti i software aziendali e quasi tutti i videogiochi si affidano ai database che non sono altro che un sistema organizzato per raccogliere i dati.
Uno dei principali tipi di database è quello relazionale, dove più tabelle sono messe in relazione tra di loro.
La tabella è l'elemento di partenza di ogni database ed è un insieme di righe e colonne dove ogni colonna contiene un dato relativo alla cosa che stiamo descrivendo e ogni riga corrisponde ad una istanza della cosa.
Se parlassimo di persone avremmo che ogni riga corrisponde ad una persona ed ogni colonna corrisponde ad una caratteristica della persona (nome, cognome, età, ecc.). In termini di database ogni colonna è un campo, ogni riga un record.

id	nome	cognome	eta	indirizzo	*Fare clic per aggiungere*
1	Filippo	Brunelli	46	Via le mani dal	
2	Pamela	Paolini	47	Via di torno	
3	Maurizio	Antonioni	52	Via Dos 2	
4	Alessandro	Alessandrovicl	25	Via Vai	
5	Giocanna	Colpodisole	30	Via il Calcare	
6	Noto	Ignoto	22	Via da qua	

Un database può essere composto di più tabelle. Ciò che rende un database relazionale è la presenza di legami fra le tabelle o di relazioni appunto.
Per i nostri esperimenti useremo un database creato con ACCESS (per chi volesse approfondire si consiglia vivamente di visitare il link https://www.filoweb.it/appunti.aspx la sezione ACCESS) contenente una sola tabella composta dai campi:

Id	Numeratore automatico
nome	Testo (String)
cognome	Testo (String)
eta	Numero (Int)
Indirizzo	Testo (String)

Creiamo quindi la nostra tabella e salviamola, per comodità, nella cartella dove stiamo creando il nostro progetto.

C# ED I DATABASE

Attraverso il Frameworks .NET C# permette di collegarsi a diversi database e per fare questo utilizza lo spazio nomi **System.Data** che fornisce accesso a classi che rappresentano l'architettura ADO.NET.
Il Name Sapce System.Data è composto da Classi, Interfacce, Enumerazioni e Delegati.
I principali Name Space per interfacciarsi con i vari database sono:

- **System.Data.Sqlclient** offre il provider di dati .NET Framework per SQL Server
- **System.Data.Odbc** offre il provider di dati .NET Framework per ODBC
- **System.Data.Oledb** offre il provider di dati .NET Framework per OLE DB
- **System.Data.Oracleclient** offre il provider di dati .NET Framework per Oracle

Per i nostri progetti che ci collegano ad Access useremo il il provider OLE DB che permette di connettersi ai database Sql Server, Access, documenti XML e molti altri.
Tra tutti i metodi che .Net mette a disposizione noi considerimo DataReader e DataSet.
Ecco un esempio di come creare la prima connessione ad un database:

```
using System;
using System.Data;
using System.Data.OleDb;
  class database {
    public static void Main() {
    OleDbConnection myConn = new OleDbConnection("Provider=Microsoft.ACE.OLEDB.12.0; Data Source=
C:\\testC\\database.accdb");
    myConn.Open();
            Console.WriteLine("Connessione Aperta");
    myConn.Close();
    Console.WriteLine("Connessione Chiusa");
    }
}
```

NOTE: Negli esempio utilizziamo, per comodità **OleDb**; per i database SQL basta cambiare il nome delle classi mettendo al posto di tutti i OleDB SQL quindi: ***SQLConnection***, ***SQLCommand***, ***SQLDataReader***, ecc., così come per utilizzare Oracle, o altri database supportati. Per le connessioni ai vari database si consiglia di leggere le relative istruzioni per costruire le stringhe di connessione

DATAREADER

DataReader utilizza un flusso di dati di sola lettura e forward, il che vuol dire che recupera il record da databse e lo memorizza nel buffer di rete fornendolo ogni volta che lo si richieste; questo metodo necessita di una connessione aperta durante la sua elaborazione ed è particolarmente indicato per manipolare grandi quantità di dati e quando la velocità è un fattore essenziale.

LEGGERE I RECORD IN UNA TABELLA

La prima classe del nostro Name Space che analizziamo è **OleDbConnection** che rappresenta una connessione aperta ad un'origine dati; al momento dell'istanziazione è norma indicare una stringa che rappresenta l'origine dati.

```
  OleDbConnection myConn = new OleDbConnection("STRINGA DI CONNESSIONE");
```

La seconda classe che ci interessa è **OleDbCommand** che serve per eseguire un comando SQL sulla nostra origine dati. Anche in questo caso al momento dell'instanziazione passo come parametri il mio comando SQL seguito dalla connessione:

```
    OleDbCommand myCmd = new OleDbCommand("COMANDO SQL", "STRINGA DI CONNESSIONE");
```

Infine la terza classe che ci interessa è la classe **OleDbDataReader** che serve a leggere un flusso di dati di una riga e va associata al comando OleDB:

```
    OleDbDataReader myReader = myCmd.ExecuteReader();
```

Con queste tre classi possiamo leggere i dati del nostro database!
Per estrapolare il contenuto della nostra tabella useremo I metodo **.GetInt32()** che server per recuperare un dato intero da un flusso di dati, mentre il metodo **.GetString()** lo estrae sotto forma di stringa.
Il metodo **.Read** della classe OleDbDataReader, in fine, serve a spostare il puntatore al record successivo.
Quando le operazioni sono concluse bisogna sempre ricordarsi di chiudere le connessioni ed i comandi aperti tramite **.Close()**.

Ecco il nostro programma per leggere il database:

```
using System;
using System.Data;
using System.Data.OleDb;

  class database {
    public static void Main() {
    string strConn = "Provider=Microsoft.ACE.OLEDB.12.0; Data Source=C:\\testC\\database.accdb";
    string strSQL = "SELECT id, nome, cognome, eta FROM nomi";

    OleDbConnection myConn = new OleDbConnection(strConn);
```

```
    OleDbCommand myCmd = new OleDbCommand(strSQL, myConn);
      myConn.Open();
    OleDbDataReader myReader = myCmd.ExecuteReader();

    while (myReader.Read()) {
        Console.WriteLine("ID:" + myReader.GetInt32(0)); // Primo record Numero intero
        Console.WriteLine("Nome: \t" + myReader.GetString(1)); // Secondo record Stringa
        Console.WriteLine("Cognome:" + myReader.GetString(2)); // Terzo record Stringa
        Console.WriteLine("Età: \t" + myReader.GetInt32(3));  // Quarto record Numero intero
     }

    myReader.Close();
    myConn.Close();
    Console.ReadKey();
    }
}
```

Come si vede il flusso di dati da OleDbDataReader mi restituisce una serie di dati sequenziali che leggo da 0 a 3 come quando leggo i dati che arrivano da una matrice e di ogni uno devo sapere il tipo di dato per poterlo estrapolare (inter, sringa, boleano, ecc.).

SCRIVERE UN RECORD IN UNA TABELLA

Per poter scrivere dei dati in un database utilizzeremo le proprietà **.CommandText** e **.CommandType** del NameSpace OleDbCommand.
.CommandText serve per ottenere o imposta l'istruzione SQL da eseguire all'origine dati.
Un esempio di utilizzo è:

.CommandText = "*SELECT [colonna],[colonna], ecc FROM [tabella] ORDER BY [colonna]*";

.CommandType serve, invece, ad impostare un valore che indica come viene interpretata la proprietà CommandText.
Il meto **.ExecuteNonQuery** della classe OleDbCommand esegue un'istruzione SQL nella proprietà Connection

Ecco il programma che permette di inserire delle righe nel nostro database.

```
using System;
using System.Data;
using System.Data.OleDb;
  class database {
    public static void Main(string[] args) {
     string nome = args[0];
     string cognome = args[1];
     int eta = int.Parse(args[2]);
     string strConn = "Provider=Microsoft.ACE.OLEDB.12.0; Data Source=C:\\testC\\database.accdb";

        OleDbConnection myConn = new OleDbConnection(strConn);
        OleDbCommand myCmd = new OleDbCommand();

        myCmd.CommandType = CommandType.Text;
        myCmd.CommandText = "INSERT INTO nomi (nome, cognome, eta) " + "VALUES ('"+nome+"', '"+co
gnome+"',"+eta+")";
        myCmd.Connection = myConn;

        myConn.Open();
            myCmd.ExecuteNonQuery();
        myConn.Close();
        Console.WriteLine("Valore inserito");
        Console.ReadLine();
    }
}
```

NOTE: Tramite ExecuteNonQuery è possibile eseguire operazioni di catalogo su di un database che non si limitano solo inserire dati nelle righe, ma anche creare tabelle, aggiungere righe, eliminare tabelle e colonne, ecc.

Supponendo di averlo compilato con il nome aggiungi.exe ecco il nostro programma in esecuzione :

```
c:\testC>csc aggiungi.cs
Compilatore Microsoft (R) Visual C# versione 3.100.119.28106 (58a4b1e7)
Copyright (C) Microsoft Corporation. Tutti i diritti sono riservati.


c:\testC>aggiungi Mosca Alata 31
Valore inserito
```

MODIFICARE UN RECORD IN UNA TABELLA

Per modificare i dati, per quanto concerne le classi da utilizzare, si usano gli stessi metodi che sono utilizzati per inserirli all'interno di un database con la sola modifica della query utilizzata:

```
myCmd.CommandText = "UPDATE nomi SET nome='" + nome + "', cognome='" + cognome +"', eta=" + eta +
" WHERE id=" + id;
```

Nella query si usa il comando **UPDATE** e la clausola *WHERE* che identifica l'ID univoco che è associato al nome. Se non scrivo l'id corretto, nel nostro esempio che segue, mi viene generato un errore

```
using System;
using System.Data;
using System.Data.OleDb;
using System.Text;

  class database {
    public static void Main(string[] args) {
     int id = int.Parse(args[0]);
     string nome = args[1];
     string cognome = args[2];
     int eta = int.Parse(args[3]);
     string strConn = "Provider=Microsoft.ACE.OLEDB.12.0; Data Source=C:\\testC\\database.accdb";

        OleDbConnection myConn = new OleDbConnection(strConn);
        OleDbCommand myCmd = new OleDbCommand();

        myCmd.CommandType = CommandType.Text;
        myCmd.CommandText = "UPDATE nomi SET nome='" + nome + "', cognome='" + cognome +"', eta="
 + eta +" WHERE id=" + id;
        myCmd.Connection = myConn;
        myConn.Open();
            myCmd.ExecuteNonQuery();
        myConn.Close();
        Console.WriteLine("ID " + id +" modificato correttamente");
        Console.ReadLine();
    }
}
```

Una volta compilato il nostro programma (e chiamato modifica.exe) per eseguirlo sarà sufficiente lanciarlo con il riferimento all'ID da modificare

```
ID:7
Nome:   Pinco
Cognome:Pallo
Età:    65
ID:9
Nome:   Mosca
Cognome:Alata
Età:    31

c:\testC>modifica 9 Marina Stella 22
```

CANCELLARE UN RECORD IN UNA TABELLA

Per eliminare un record si usa la clausola **DELETE** al posto di update

```
 myCmd.CommandText = "DELETE FROM nomi WHERE id=" + id;
```

e al programma viene trasferito solo l'ID che si vuole eliminare; se volessi invece eliminare tutti i record che contendono un nome simile a "filippo" (quindi Filippo, filippo, FILIPPO,, ecc.) modifico la query così:

```
 myCmd.CommandText = "DELETE FROM nomi WHERE nome LIKE'%filippo%'";
```

TROVARE DEI RECORD IN UNA TABELLA

Per cercare un record all'interno di un database utilizzo lo stesso metodo che si utilizza leggere i record da una tabella modificando la query in:

```
string strSQL = "SELECT id, nome, cognome, eta FROM nomi WHERE nome LIKE'%" + nome + "%'";
```

In questo modo passo al programma il parametro nome che cerco nella tabella. La clausola **LIKE** serve per cercare qualcosa che assomiglia e il carattere % che precede e segue equivale al carattere jolly *
(per i riferimenti alle query si consiglia di guardare la lezione su access su **WWW.FILOWEB.IT**).
Dato che nome è di tipo stringa devo inserire la ricerca tra due apici ' ', se avessi cercato per ID o per età non sarebbe stato necessario:

```
string strSQL = "SELECT id, nome, cognome, eta FROM nomi WHERE eta=" + eta;
```

Nell'esempio di query sopra avrei ottenuto la lista di tutti i nominativi che hanno età corrispondente alla mia richiesta
Ecco il programma completo.

```
using System;
using System.Data;
using System.Data.OleDb;
  class database {
    public static void Main(string[] args) {
    string nome = args[0];
    string strConn = "Provider=Microsoft.ACE.OLEDB.12.0; Data Source=C:\\testC\\database.accdb";
    string strSQL = "SELECT id, nome, cognome, eta FROM nomi WHERE nome LIKE'%" + nome + "%'";

    OleDbConnection myConn = new OleDbConnection(strConn);
    OleDbCommand myCmd = new OleDbCommand(strSQL, myConn);
    myConn.Open();
    OleDbDataReader myReader = myCmd.ExecuteReader();
    while (myReader.Read()) {
        Console.WriteLine("ID:" + myReader.GetInt32(0));
        Console.WriteLine("Nome: \t" + myReader.GetString(1));
        Console.WriteLine("Cognome:" + myReader.GetString(2));
        Console.WriteLine("Età: \t" + myReader.GetInt32(3));
     }

    myReader.Close();
    myConn.Close();
    Console.ReadLine();
    }
}
```

DATASET

L' oggetto **DataSet** è un oggetto che corrisponde ad una rappresentazione in memoria di un database. Al suo interno contiene l'oggetto **DataTable**, che corrisponde alle tabelle del database, che contiene a sua volta l'oggetto **DataColumn**, che definisce la composizione delle righe chiamate **DataRow**.
I vantaggi di usare un DataSet è che permette di definire relazioni tra le DataTable di un DataSet, allo stesso modo di quanto avviene in un database.
Una delle principali differenze tra un DataReader (visto precedentemente) e un DataSet è che il primo manterrà una connessione aperta al database fino a quando non la si chiude, mentre un DataSet sarà un oggetto in memoria. Questo porta come conseguenza che un DataSet risulta più *pesante* di un DataReader.

Un DataReader è di tipo forward alla lettura di dati mentre un DataSet permette di spostarti avanti e indietro e manipolare i dati.
Una delle caratteristiche aggiuntive dei DataSet è che possono essere serializzati e rappresentati in XML (l'uso dei file XML sarà oggetto del prossimo capitolo). mentre i DataReader non possono essere serializzati.
Se però si ha una grande quantità di righe da leggere dal database un DataReader è preferibile ad un DataSet che carica tutte le righe, occupa memoria ed influenzare scalabilità.
Uno dei principali oggetti utilizzati da DataSet è **DataAdapter** che funge da ponte tra un Dataset ed il Database.

LEGGERE I RECORD IN UNA TABELLA TRAMITE DATASET

Abbiamo introdotto i dataset e abbiamo visto che caricano in memoria i risultati di una query; una volta fatto questo sarà sufficiente scorrere i dati come si fa con un normale array…

```
using System;
using System.Data;
using System.Data.OleDb;
  class database {
     public static void Main() {
     string strConn = "Provider=Microsoft.ACE.OLEDB.12.0; Data Source=C:\\testC\\database.accdb";
     string strQuery =" Select * from nomi";

        OleDbConnection myConn = new OleDbConnection(strConn);
        OleDbDataAdapter myAdapter = new OleDbDataAdapter(strQuery, myConn);
        DataSet myDataSet = new DataSet();
        myConn.Open();

        myAdapter.Fill(myDataSet, "nomi");
        DataTable tblNomi = myDataSet.Tables["nomi"];

        foreach (DataRow drnomi in tblNomi.Rows) {
                Console.WriteLine("ID:" + drnomi ["ID"]);
                Console.WriteLine("Nome: \t" + drnomi ["nome"].ToString() );
                Console.WriteLine("Cognome:" + drnomi ["cognome"].ToString() );
                Console.WriteLine("eta:" + drnomi ["eta"]);
                Console.WriteLine("----------------------");
            }
           myAdapter.Dispose();
    }
}
```

Come si vede abbiamo caricato i contenuti della tabella all'interno di tblNomi tramite **DataAdapter** e poi fatto scorrere.
Un'altra cosa che si nota è che in questo caso non dobbiamo preoccuparci di aprire e chiudere una connessione in quanto viene tutto gestito dal DataSet.

SCRIVERE UN RECORD IN UNA TABELLA TRAMITE DATASET

Il metodo per aggiungere una nuova riga di dati ad un database è simile a quello visto sopra per leggerle.
Anche qua creiamo un DataSet contenente la tabella che ci interessa, poi usiamo il metodo **.Add** per aggiungere una nuova riga passando una matrice di valori, tipizzata come Object.
Importante è ricordarsi di utilizzare il metodo **.Update** per aggiornare la tabella del nostro database una volta che sono stati inseriti i dati del DataSet in memoria.
Un altro metodo importante è **.MissingSchemaAction** che indica quale azione eseguire quando si aggiungono dati all'oggetto DataSet e risultano mancanti gli oggetti.
Nel nostro caso **MissingSchemaAction.AddWithKey** aggiunge l'id (che è un valore tipo int autoincrementale univoco) automaticamente.

Altre opzioni per MissingSchemaAction possono essere:

- Add: Aggiunge le colonne necessarie per completare lo schema;
- Ignore: Vengono ignorate le colonne supplementari;
- Error: Se il mapping della colonna specificata risulta mancante, verrà generato l'oggetto InvalidOperationException.

```
using System;
using System.Data;
using System.Data.OleDb;

class database {
    public static void Main(string[] args) {
     string nome = args[0];
     string cognome = args[1];
     int eta = int.Parse(args[2]);
     string strConn = "Provider=Microsoft.ACE.OLEDB.12.0; Data Source=C:\\testC\\database.accdb";
     string strQuery =" Select * from nomi";

    OleDbConnection myConn = new OleDbConnection(strConn);
    OleDbDataAdapter myAdapter = new OleDbDataAdapter(strQuery, myConn);
    DataSet myDataSet = new DataSet();
    DataRow myDataRow;

    OleDbCommandBuilder myBuilder = new OleDbCommandBuilder(myAdapter);

    myAdapter.MissingSchemaAction = MissingSchemaAction.AddWithKey
    myAdapter.Fill(myDataSet, "nomi");
        myDataRow = myDataSet.Tables["nomi"].NewRow();
        myDataRow["nome"] = nome;
        myDataRow["cognome"] = cognome;
        myDataRow["eta"] = eta;

         myDataSet.Tables["nomi"].Rows.Add(myDataRow);
    myAdapter.UpdateCommand = myBuilder.GetUpdateCommand();
    myAdapter.Update(myDataSet,"nomi");
   myAdapter.Dispose();
    Console.WriteLine("Valore inserito");
    }
}
```

Come si vede, inserendo i dati in questo modo è più facile evitare errori anche se si ha un utilizzo maggiore di memoria.

MODIFICARE UN RECORD IN UNA TABELLA TRAMITE DATASET

Per modificare un record tramite DataSet è sufficiente caricare in memoria i record e tramite:

```
myDataRow = myDataSet.Tables["nomi"].Rows.Find(id);
```

recuperare l'ID (che deve essere un numero univoco autoincrementale) della riga che voglio modificare e sovrascriverne i contenuti.

```
using System;
using System.Data;
using System.Data.OleDb;
  class database {
     public static void Main(string[] args) {
     int id = int.Parse(args[0]);
     string nome = args[1];
     string cognome = args[2];
     int eta = int.Parse(args[3]);
     string strConn = "Provider=Microsoft.ACE.OLEDB.12.0; Data Source=C:\\testC\\database.accdb";
     string strQuery =" SELECT * FROM nomi WHERE id=" + id;

    OleDbConnection myConn = new OleDbConnection(strConn);
    OleDbDataAdapter myAdapter = new OleDbDataAdapter(strQuery, myConn);
    DataSet myDataSet = new DataSet();
```

```
    DataRow myDataRow;

    OleDbCommandBuilder myBuilder = new OleDbCommandBuilder(myAdapter);

    myAdapter.MissingSchemaAction = MissingSchemaAction.AddWithKey;
    myAdapter.Fill(myDataSet, "nomi");

        myDataRow = myDataSet.Tables["nomi"].Rows.Find(id);
        myDataRow["nome"] = nome;
        myDataRow["cognome"] = cognome;
        myDataRow["eta"] = eta;

        myAdapter.UpdateCommand = myBuilder.GetUpdateCommand();
        myAdapter.Update(myDataSet,"nomi");
        myAdapter.Dispose();
        Console.WriteLine("Valore Modificato");
    }
}
```

TROVARE UN RECORD TRAMITE DATASET

Per recuperare uno o più record in un DataSet è sufficiente modificare la stringa della query che si usa per legere i dati:

```
string strQuery =" Select * from nomi WHERE nome LIKE '%" + nome + "%'";
```

NOTE: per una spiegazione rapida dell'uso delle istruzioni SQL si consiglia di guardare i riferimenti online

DATASET O DATAREADER?

Prima di considerare quale sia migliore o meno tra le due soluzioni dobbiamo considerare cosa richiede la nostra applicazione, la velocità di esecuzione, la scalabilità, le dimensioni di dati, ecc.
Sia l'uso di DataSet che di DataReader sono molto frequenti nelle applicazioni .Net per recuperare i dati da un database.
Possiamo fare un piccolo raffronto veloce, per vedere quando usare uno o l'altro:

DataSet	DataReader
• Applicazione windows • Dati non troppo grandi • Restituzione di più tabelle • Il risultato è da serializzare • Architettura disconnessa • Per inviare attraverso i livelli • Memorizzazione nella cache dei dati • Non è necessario aprire o chiudere la connessione	• Applicazione web • Dati di grandi dimensioni • Restituzione di più tabelle • Per un accesso rapido ai dati • Deve essere esplicitamente chiuso • Il valore del parametro di output sarà disponibile solo dopo la chiusura • Restituisce solo una riga dopo la lettura

DataReader è uno stream di sola lettura e forward. Recupera il record da databse e memorizza nel buffer di rete e fornisce ogni volta che lo si richiede. DataReader rilascia i record mentre la query viene eseguita e non attende l'esecuzione dell'intera query. Pertanto è molto veloce rispetto al set di dati. Se è necessario un accesso di sola lettura ai dati, è utile eseguire spesso operazioni all'interno di un'applicazione utilizzando un DataReader.
Con **DataSet** i valori vengono caricati in memoria significa lavorare con un metodo disconnesso ovvero che non necessita di una connessione aperta durante il lavoro sul set di dati; quando si popola un elenco o si recuperano enormi quantità di record è meglio utilizzare pertanto Data Reader. In un'applicazione Web in cui centinaia di utenti potrebbero essere connessi, la scalabilità diventa un problema: se si prevede che questi dati vengano recuperati e per l'elaborazione DataReader potrebbe accelerare il processo in quanto recupera una riga alla volta e non richiede le risorse di memoria richieste dal DataSet.
Proprio come un database DataSet, d'altronde, è costituito da un insieme di tabelle e relazioni tra di loro, insieme a vari vincoli di integrità dei dati sui campi delle tabelle.

CAPITOLO VIII

XML

XML è un linguaggio di markup creato dal World Wide Web Consortium (W3C) per definire una sintassi per la codifica dei documenti.
A differenza di altri linguaggi di markup, come ad esempio come HTML, l' XML non ha un linguaggio di markup predefinito ma consente agli utenti di creare i propri simboli di marcatura per descrivere il contenuto del documento, creando un insieme di simboli illimitato e auto-definente.
L'XML è oggi molto utilizzato anche come mezzo per l'esportazione di dati tra diversi DBMS, è usato nei file di configurazione di applicazioni e sistemi operativi ed in molti altri settori.
Per fare alcuni esempi il formato XML viene usato per gli RSS Feed, per l'interscambio delle fatture elettroniche, lo scambio dati tra dati tra diversi portali e software gestionali, per la formattazione nei documenti nel formato Open Document, per le confeme delle PEC, ecc..

Un file XML non è altro che un file di testo che deve rispettare alcune regole:

- l'intestazione (<?xml version="1.0" encoding="UTF-8"?>) deve comprendere l'idendificazione del file XML appunto, la versione e l'encoding dei caratteri
- Deve esistere uno e soltanto uno elemento Root
- Ogni Tag apertodeve essere chiuso

Ecco come posono apparire i file XML contenente i nostri dati

```xml
<?xml version="1.0" encoding="UTF-8"?>
<nomi>
    <persona>
       <id>1</id>
       <nome>Filippo</nome>
       <cognome>Brunelli</cognome>
       <eta>47</eta>
    </persona>
    <persona>
       <id>2</id>
       <nome>Pamela</nome>
       <cognome>Paolini</cognome>
       <eta>48</eta>
    </persona>
    <persona>
       <id>3</id>
       <nome>Francesco</nome>
       <cognome>Assisi</cognome>
       <eta>838</eta>
    </persona>
    <persona>
       <id>4</id>
       <nome>Mario</nome>
       <cognome>Mari</cognome>
       <eta>33</eta>
    </persona>
</nomi>
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<nomi>
    <persona id=”1”>
       <nome>Filippo</nome>
       <cognome>Brunelli</cognome>
       <eta>47</eta>
    </persona >
    <persona id=”2”>
       <nome>Pamela</nome>
       <cognome>Paolini</cognome>
       <eta>48</eta>
    </persona >
    <persona id=”3”>
       <nome>Francesco</nome>
       <cognome>Assisi</cognome>
       <eta>838</eta>
    </persona >
    <persona id=”4”>
       <nome>Mario</nome>
       <cognome>Mari</cognome>
       <eta>33</eta>
    </persona >
</nomi>
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<nomi>
    <persona >
       <id>1</id>
       <nome>Filippo</nome>
       <cognome>Brunelli</cognome>
       <eta>47</eta>
       <indirizzi>
         <via>Via Vai </via>
         <n>12</n>
       </indirizzi>
    </persona >
    <persona >
       <id>2</id>
       <nome>Pamela</nome>
       <cognome>Paolini</cognome>
       <eta>48</eta>
       <indirizzi>
         <via>Via dai guai </via>
         <n>2</n>
        </indirizzi>
    </persona >
    <persona >
       <id>3</id>
       <nome>Francesco</nome>
       <cognome>Assisi</cognome>
       <eta>838</eta>
       <indirizzi>
          <via>Via cielo</via>
          <n>7</n>
       </indirizzi>
    </persona >
</nomi>
```

La **root** è il tag nomi, mentre i **nodi** sono persona; i sotto nodi sono quelli contenuti all'interno dei nodi come mostrato nell' ultimo esempio.

XML E .NET

C#, tramite il framework .Net, ha a disposizione diversi metodi per leggere e manipolare i dati XML, proprio come per i database relazionali e le principali classi sono **XmlDocument** e **XmlReader**.
XmlDocument, si comporta come un DataSet: legge l'intero contenuto XML in memoria e quindi consente di spostarti avanti e indietro come si preferisce, o di interrogare il documento utilizzando la tecnologia **XPath**.
XmlReader è invece simile alla classe DataReader: risulta più veloce e meno dispendioso in termini di memoria e consente di scorrere il contenuto XML un elemento alla volta, permettendo di esaminare il valore e quindi passare all'elemento successivo.
I principali Name Space messi a disposizione da .Net per utilizzare il formato XML sono: **System.Xml**, **System.Xml.Schema**, **System.Xml.Serialization**, **System.Xml.XPath** e **System.Xml.Xsl** per supportare le classi XML.

LEGGERE I DATI TRAMITE XMLDOCUMENT

Ecco un esempio tramite l'uso di XmlDocument:

```
using System;
using System.Xml;
using System.Xml.XPath;
    class xmluno {
        static void Main() {
            XmlDocument xmlDoc = new XmlDocument();
            xmlDoc.Load("c:\\testC\\esempiXML.xml");
            XmlNodeList lista = xmlDoc.SelectNodes("/nomi/persona");
            string nome;
            string cognome;
            string eta;
            foreach (XmlNode ls in lista) {
                        nome = ls["nome"].InnerText;
                        cognome = ls["cognome"].InnerText;
                        eta = ls["eta"].InnerText;
                        Console.WriteLine(nome + " " + cognome + " " + eta);
                }
            }
        }
```

NOTE: Solitamente un file XML (essendo un file di testo) difficilmente è di grandissime dimensioni e caricarlo in memoria non costituisce un grosso problema, per cui utilizzare XmlDocument non è necessariamente sinonimo di calo di prestazioni.

Abbiamo visto che si utilizza la classe XmlNodeList per selezionare un nodo del nostro documento XML.
XmlNodeList ha due proprietà fondamentali:

- **XmlNode.ChildNodes** che restituisce un XmlNodeList oggetto contenente tutti gli elementi figlio del nodo.
- **XmlNode.SelectNodes** che restituisce un XmlNodeList oggetto contenente una raccolta di nodi che corrispondono alla query XPath.

Nel nostro caso abbiamo utilizzato la seconda proprietà per ottenere la lista dei dati contenuti nel nodo persona, caricarli in memoria in memoria e poi, tramite un ciclo foreach mostrarli.
Se avessi modificato la riga nel seguente modo:

```
XmlNodeList lista = xmlDoc.SelectNodes("/nomi/persona/indirizzi");
```

ed il ciclo in:

```
                nome = ls["via"].InnerText;
                cognome = ls["n"].InnerText;
                Console.WriteLine(nome +" "+ cognome );
```

avrei avuto la lista degli indirizzi.
Per una lista completa delle proprietà visitare il sito microsoft: https://docs.microsoft.com/it-it/dotnet/api/system.xml.xmlnodelist?view=netframework-4.8

ESTRARRE UN DATO DA UN NODO

Il metodo più semplice per estrarre uno o più dati tramite XmlDocument è quello di inserire tra parentesi quadre nella stringa del nodo:

```
XmlNodeList lista = xmlDoc.SelectNodes("/nomi/persona[nome='Pamela']");
```

Un altro metodo è quello, di scorrere la lista fino a quando non si trova il dato cercato e, a quel punto, mostrarlo

```
foreach (XmlNode ls in lista) {
                if (ls["nome"].InnerText=="Pamela") {
                nome = ls["nome"].InnerText;
                cognome = ls["cognome"].InnerText;
                eta = ls["eta"].InnerText;
                Console.WriteLine(nome +" "+ cognome + " " +eta);
                }
            }
```

RIMUOVERE UN NODO

Per rimuovere un nodo da un file XML si usa la classe XmlNode; questa classe astratta che permette l'accesso ad un nodo del file Xml.
La classe XmlNode contiene una serie di proprietà tra le quali le principali sono:

.AppendChild(Node)	Aggiunge il nodo specificato alla fine dell'elenco dei nodi figlio del nodo corrente.
.Clone()	Crea un duplicato del nodo.
.InsertAfter(1° Nodo, 2° Nodo)	Inserisce il 2°nodo immediatamente dopo il 1° nodo
.InsertBefore(1° Nodo, 2° Nodo)	Inserisce 2° nodo immediatamente prima il 1° nodo
.RemoveAll()	Rimuove tutti gli elementi figlio e/o gli attributi del nodo corrente.
.RemoveChild(Nodo)	Rimuove il nodo figlio specificato.

```
using System;
using System.Xml;
using System.Xml.XPath;
    class xml2 {
        static void Main() {
            string fileXml = @"c:\testC\esempiXML.xml";
            XmlDocument xmlDoc = new XmlDocument();
            XPathNavigator navigator = xmlDoc .CreateNavigator();
            xmlDoc.Load(fileXml);
               XmlNode Nd = xmlDoc.SelectSingleNode("/nomi/persona[nome='Nuovo']");
            xmlDoc.DocumentElement.RemoveChild(Nd);
            xmlDoc.Save(fileXml);
        }
    }
```

Come si vede è sufficiente caricare il nodo desiderato e utilizzare la proprietà **.RemoveChild**

NOTE: La stringa fileXml contiene il caratere speciale \; ci sono due modi per poterlo inserire, uno consiste nell'inserirlo 2 volte (\\) dove il primo baskslash indica che seguirà un carattere speciale (vedi capitolo sulle stringhe); l'altro metodo è denominato *"Caratteri alla lettera"* che consiste nell'inserire il carattere @ prima della stringa (@"c:\testC\esempioXML.xml"). Tramite i carattere @ anteposto alla stringa è possibile inseire i caratteri speciali senza necessita di inserire \ prima di questi ultimi.

AGGIUNGERE UN NODO

Per aggiungere un nodo utilizzerò sempre la classe XmlNode ma il metodo **.AppendChild()** dopo aver selezionato il nodo padre che mi interessa.
Se il nuovo nodo è già presente nell'albero, viene rimosso dalla posizione originale e aggiunto alla posizione di destinazione.

Un esempio può aiutare a capire meglio:

```
using System;
using System.Xml;
using System.Xml.XPath;
    class xml2 {
        static void Main() {
            string fileXml = @"c:\testC\esempiXML.xml";
            XmlDocument xmlDoc = new XmlDocument();
            xmlDoc.Load(fileXml);
            XmlNode node = xmlDoc.CreateNode(XmlNodeType.Element, "persona", null);
            XmlNode nodeId = xmlDoc.CreateElement("id");
            nodeId.InnerText = "5";
            XmlNode nodeNome = xmlDoc.CreateElement("nome");
            nodeNome.InnerText = "Nuovo";
            XmlNode nodeCognome = xmlDoc.CreateElement("cognome");
            nodeCognome.InnerText = "Nome";
            XmlNode nodeEta = xmlDoc.CreateElement("eta");
            nodeEta.InnerText = "100";

            //aggiungo i nodi
            node.AppendChild(nodeId);
            node.AppendChild(nodeNome);
            node.AppendChild(nodeCognome);
            node.AppendChild(nodeEta );

            //aggiung i nodi all'XML
            xmlDoc.DocumentElement.AppendChild(node);
            xmlDoc.Save(fileXml);
        }
```

Come si vede è sufficiente creare i nodi tramite

XmlNode *oggetto* = *xmlDoc*.**CreateElement**("*nome_nuovo_nodo*");

All'interno del nodo padre ed assegnare un valore tramite **.**InnerText
E per finire aggiungo i nodi con il metodo **.AppendChild(***oggetto***)**

NOTE: Negli ultimi esempi visti abbiamo visto la proprietà **.Save**(*nome_File*) e la proprietà .Load(*nome_File*) della classe XmlDocument; queste proprietà servono, come dice il nome a salvare su disco e caricare in memoria un documento XML; è Importante, quando si usa la proprietà **.Save** che il file non sia in uso o sia aperto da un altro programma o utente in accesso esclusivo!

XPATH

XPath (*XML Path Language*) è un linguaggio standard che utilizza una sintassi non XML che è stato sviluppato per fornire un modo flessibile per scorrere un documento XML e trovarne i nodi ed i contenuti.
C# ed il frameworks .Net permette di utilizzare XPath all'interno dei propri progetti tramite lo spazio nomi:

```
using System.Xml.XPath;
```

Abbiamo visto l'utilizzo di questo spazio nomi negli esempi precedenti ma, poiché questo è solo un veloce tutorial generico, esamineremo solo le espressioni XPath di base e il loro significato.
Dato che XPath vede un documento XML come un albero di nodi e il suo scopo è proprio quello di individuare nodi e insiemi di nodi all'interno di questo albero una cosa importante da ricordare quando lo si utilizza è il contesto in cui ci si trova quando si tenta di utilizzare l'espressione.
Cosideriamo quindi che:
nodo radice: Il nodo radice dell'albero non è lo stesso elemento radice del documento infatti il nodo radice dell'albero contiene l'intero documento (compreso l'elemento radice; i commenti; le istruzioni di elaborazione presenti prima del tag iniziale o dopo il tag finale), quindi ha come figlio l'intero documento e non ha padre.

nodi elemento: hanno come padre il nodo radice o un altro nodo elemento. Come figli ha eventuali sotto nodi, testo e istruzioni contenute al suo interno. Gli attributi (<persona id="1">)non sono figli di un nodo elemento.
nodi attributo: hanno come padre un altro elemento, ma non si considera figlio di quell'elemento. Per accedervi ci vuole una richiesta di attributo.
Ad esempio /nomi/persona/nome applicato sul nostro file XML otterrà come risultato:

```
Element='<nome>Filippo</nome>'
Element='<nome>Pamela</nome>'
Element='<nome>Francesco</nome>'
Element='<nome>Mario</nome>'
```

Gli attributi degli elementi possono essere selezionati tramite XPath utilizzando il carattere @ seguito dal nome dell'attributo.
Posso quindi applicare, ad esempio, /nomi/persona[@id='3'] per selezionare il nodo specifico in:

```
<persona id="1">
    <nome>Filippo</nome>
    <cognome>Brunelli</cognome>
    <eta>47</eta>
</persona>
<persona id="3">
    <nome>Mario</nome>
    <cognome>Mari</cognome>
    <eta>33</eta>
</persona>
<persona id="2">
    <nome>Francesco</nome>
    <cognome>Franchi</cognome>
    <eta>60</eta>
</persona>
```

Una delle classi più importanti nel lo spazio nomi *System.Xml.XPath* è sicuramente **XPathNavigator** che fornisce un set di metodi usati per modificare nodi e valori in un documento XML ; per poter fare ciò è però necessario che l'oggetto XPathNavigator sia modificabile, quindi la proprietà *CanEdit* deve essere true.
La classe **XPathNodeIterator** fornisce i metodi per eseguire un'iterazione in un set di nodi creato come risultato di una query XPath (Usando il linguaggio XML Path) tramite l'utilizzo di un cursore di tipo forward-only di sola lettura e quindi utilizza il metodo **.MoveNext**; il set di nodi viene creato in base all'ordine con cui è riportato nel documento, quindi la chiamata a questo metodo consente di spostarsi al nodo successivo nell'ordine del documento. XPathNodeIterator consente di ottenere informazioni dal nodo corrente tramite **.Current .Name** e .**Current.Value**.

Ecco un esempio di come estrarre dei dati tramite XPath:

```
using System;
using System.Xml;
using System.Xml.XPath;
    class xmluno {
        static void Main() {
            XmlDocument xmlDoc = new XmlDocument();
            xmlDoc.Load("c:\\testC\\esempiXML.xml");
            XPathNavigator navigator=xmlDoc.CreateNavigator();
            XPathNodeIterator nodo=navigator.Select("/nomi/persona/nome");
              while (nodo.MoveNext()) {
                    Console.WriteLine(nodo.Current.Name +": " + nodo.Current.Value);
                }
            }
        }
```

Modificando la riga

```
XPathNodeIterator nodo=navigator.Select("/nomi/persona[nome='Filippo'] ");
```

otterremo tutti i dati del nodo persona il cui nome corrisponde a Filippo.

```
c:\testC>xml6
persona: 1FilippoBrunelli47Via Vai 12
```

Un'altra proprietà interessante della classe XPathNavigator è **.OuterXml** che ottiene il markup che rappresenta i tag di apertura e di chiusura del nodo corrente e dei relativi nodi figlio.

```
using System;
using System.Xml;
using System.Xml.XPath;
    class xmluno {
        static void Main() {
            XmlDocument xmlDoc = new XmlDocument();
            xmlDoc.Load("c:\\testC\\esempiXML.xml");
            XPathNavigator navigator=xmlDoc.CreateNavigator();
            Console.WriteLine(navigator.OuterXml);
            }
        }
```

LEGGERE I DATI TRAMITE XMLREADER

Vediamo adesso come visualizzare un documento XML con l'utilizzo della classe **XmlReader**:

```
using System;
using System.Xml;
    class xml2
    {
        static void Main()
        {
            using (XmlReader reader = XmlReader.Create(@"c:\testC\esempiXML.xml"))
            {
                while (reader.Read())
                {
                    if (reader.IsStartElement())
                    {
                        switch (reader.Name.ToString())
                        {
                            case "nome":
                                Console.Write(reader.ReadString());
                                break;
                            case "cognome":
                                Console.Write(" " + reader.ReadString());
                                break;
                            case "eta":
                                Console.WriteLine(" " + reader.ReadString());
                                break;
                            case "via":
                                Console.Write("ABITA in " + reader.ReadString());
                                break;
                            case "n":
                                Console.WriteLine(" n.° " + reader.ReadString());
                                break;
                        }
                    }
                }
            }
        }
    }
```

Come si vede viene letto il flusso di dati riga per riga e a seconda del nodo viene intrapresa una relativa azione.
Notiamo la presenza dell'azione **.IsStartElement()** che serve a verificare se il nodo di contenuto corrente è un tag di inizio; se questo è vero iniziamo una serie si switch (vedere la prima parte del corso) per leggere i contenuti dei nodi e dei sotto nodi. Anche se questo metodo sembra più macchinoso del precedente, in determinati casi, può offrire un controllo maggiore del risultato.

CAPITOLO IX

FILES E DIRECTORY

Sebbene si debba considerare superfluo il dover spiegare cosa siano i file e le directory (o cartelle) consideriamo velocemente di ripassare il significato di questi due termini in informatica:

- FILE: Il termine file viene utilizzato per riferirsi a un contenitore di informazioni/dati in formato digitale, tipicamente presenti su un supporto digitale di memorizzazione opportunamente formattato in un determinato file system. In altra parole, ogni singolo oggetto archiviato in memoria di massa viene definito FILE. Un file può essere un programma eseguibile, un documento di testo, un'immagine, un filmato, un audio una pagina Web, o tanto altro ancora.
- DIRECTORY: Le directory (o cartelle) sono dei "contenitori" che permettono di organizzare in maniera ordinata e gerarchica i files.

Quando si sviluppa un programma o un app sovente capita di dover creare dei file, siano essi di configurazione o per memorizzare dati in maniera temporanea o altro; nel frameworks .Net lo spazio dei nomi che si occupa di gestire i file e flussi di dati, che ne consente la lettura e la scrittura, che fornisce il supporto anche per le directory di base è **System.IO**.
Lo spazio dei nomi System.IO ha varie classi che vengono utilizzate per eseguire numerose operazioni con i file, come la creazione e l'eliminazione, la lettura o la scrittura, la chiusur, ecc.
Le principali classi sono:

- **BinaryReader**: Legge i dati da un flusso binario.
- **BinaryWriter**: Scrive dati in formato binario.
- **BufferedStream**: è una memoria temporanea per un flusso di byte.
- **Directory**: Aiuta a manipolare una struttura di directory.
- **DirectoryInfo**: Utilizzato per eseguire operazioni su directory.
- **DriveInfo**: Fornisce informazioni per le unità.
- **File**: Aiuta a manipolare i file.
- **FileInfo**: Utilizzato per eseguire operazioni sui file.
- **FileStream**: Utilizzato per leggere e scrivere in qualsiasi posizione in un file.
- **MemoryStream**: Utilizzato per l'accesso casuale ai dati in streaming archiviati in memoria.
- **Path**: Esegue operazioni sulle informazioni sul percorso.
- **StreamReader**: Utilizzato per leggere caratteri da un flusso di byte.
- **StreamWriter**: Viene utilizzato per scrivere caratteri in uno stream.
- **StringReader**: Viene utilizzato per la lettura da un buffer di stringhe.
- **StringWriter**: Viene utilizzato per la scrittura in un buffer di stringhe.

Quando si intende memorizzare le informazioni in un file, occorre analizzare la tipologia e il numero di informazioni, perpoterle organizzare, in modo da riuscire a leggerle successivamente per poterle reinterpretare.

LEGGERE E SCRIVERE UN FILE

La classe **StreamReader** permette di leggere un file come fosse un flusso di caratteri.

```
using System;
using System.IO;
    class miofile {
       static void Main() {
            StreamReader sr = new StreamReader("c:\\testC\\esempiXML.xml");
            String line = sr.ReadToEnd();
            Console.WriteLine(line);
            sr.Close();
        }
    }
```

Nell'esempio sopra il utilizziamo il metodo della classe SreamReader per caricare un file (in questo caso il nostro file XML) nella stringa line. Questa classe legge i caratteri da un flusso di byte fino alla fine del file (**.ReadToEnd()**) in una particolare codifica solitamente predefinita che è la codifica UTF-8, a meno che non sia specificato diversamente.
Per istanziare lo StreamReader con un'altra codifica la si scrive dopo la strigna del fiel separandola con una virgola:

```
 StreamReader sr = new StreamReader("c:\\testC\\esempiXML.xml", System.Text.Encoding.ASCII);
```

La classe **StreamWriter** si usa, invece, per scrivere un file di testo:

```
using System;
using System.IO;
    class miofile {
       static void Main() {
       StreamWriter scrivi= new StreamWriter(File.Open("c:\\testC\\test.txt", FileMode.Create));
           scrivi.Write("Ciao Mondo");
           scrivi.Write((char)10);
           scrivi.Write("Io sono Filippo");
           scrivi.Close();

        }
    }
```

Anche per StreamWriter posso inserire un encoding allo stesso modo che per lo StreamReader.
FileMode, invece Specifica le modalità di apertura di un file da parte del sistema operativo; i suoi metodi sono principali, che ci interessano, sono:

- **Append**: apre il file, se esiste, e si sposta alla fine del file oppure crea un nuovo file.
- **Create**: specifica che il sistema operativo deve creare un nuovo file. ATTENZIONE: Se il file esiste verrà sovrascritto.
- **CreateNew**: specifica che il sistema operativo deve creare un nuovo file

Una volta aperto (o creato) il file è sufficiente "Scrivere" quello che si vuole nel vile tramite il metodo Write; scrivendo il codice ASCII 10 inserisco un accapo nel mio file di testo.
Proviamo a modificare FileMode.Create con FileMode.Append e vediamo che aggiungiamo nuove righe al file che abbiamo creato:

```
StreamWriter scrivi= new StreamWriter(File.Open("c:\\testC\\test.txt", FileMode.Append));
```

NOTE: È buona norma ricordarsi sempre di chiudere il file aperto tramite il metodo **.Close()**.

A differenza di quanto accade con i file di testo, i contenuti dei file binari non sono consultabili con la semplice apertura del file con un editor di testo. Per creare e leggere file binari utilizziamo le classi **BinaryReader** e **BinaryWriter**. Queste due classi sono dei lettori e scrittori di *stream* di tipo binario che realizzano operazioni di input/output su un *FileStream* di tipo formattato.
Permettono cioè la lettura e la scrittura di *Int, Float, Double, String,* ecc. La classe **FileStream** permette di realizzare un semplice flusso di byte dall'applicazione al file viceversa.

Consideriamo ora il programma:

```
using System;
using System.IO;
    class miofile {
          public static void Main() {
                    FileStream fs = File.Create("c:\\testc\\test.bin");
                    BinaryWriter bw = new BinaryWriter(fs);
                        int numero = 10;
                        double nDoppio = 1500.34;
                        string Stringa = "Filippo ";
                    bw.Write(numero);
                    bw.Write(nDoppio);
                    bw.Write(Stringa);
                bw.Close();
                fs.Close();
                }
    }
```

Compiliamo ed eseguiamo il programma quindi proviamo ad aprire il nostro file con blocnote, il risultato sarà:

Filippo□ÂõÂ(\q—@

Compreso della riga vuota. Come si vede è qualcosa di molto differente da quello che abbiamo scritto.
Per leggere il nostro file, ora dobbiamo creare un programma apposito che legga i dati convertendoli nel valore corrispondente:

```
using System;
using System.IO;

    class miofile {
    public static void Main() {
            FileStream fs = File.OpenRead("c:\\testc\\test.bin");
            BinaryReader br = new BinaryReader(fs);
                Console.WriteLine(br.ReadInt32());
                Console.WriteLine(br.ReadString());
                Console.WriteLine(br.ReadDouble());
             br.Close();
            fs.Close();
        }
    }
```

Non solo devo sapere che tipo di dato leggo ma anche la loro posizione nella sequenza dello stream; se infatti provo a modificare la lettura dello stream:

```
                Console.WriteLine(br.ReadInt32());
                Console.WriteLine(br.ReadDouble());
                Console.WriteLine(br.ReadString());
```

mi si genererà un errore in quanto cerco di leggere una stringa all'interno di una stringa Double.
Se consideriamo un file ODT (*Open Document Text*) esso è binario in quanto è un set compresso di file XML, ma i file XML all'interno sono considerati file di testo. Quindi quando voglio leggere un file Open Office Write devo aprirlo come file binario.

LAVORARE CON FILES E DIRECOTRY

Come la classe **.File** anche la **.FileInfo** viene usata da C# per lavorare con i files.
La classe .File permette di effettuare operazioni sui file come lo spostarli, cancellarli e altro ancora; se volessi, ad esempio spostare un file è sufficiente utilizzare il metodo **.Move()**, indicanto la posizione ed il file sia di origine che di destinazione:

```
using System;
using System.IO;
    class miofile {
    public static void Main() {
         File.Move(@"c:\testc\test.txt",@"c:\testc\nuova\test.txt"); }
    }
```

Il metodo .Move() può essere utilizzato anche per rinominare un file, sarà sufficiente *spostarlo* nella stessa directory cambiandone il nome:

```
            File.Move(@"c:\testc\test.txt",@"c:\testc\test2.txt");
```

Per copiare un file si utilizza il metodo **.Copy()** la cui sintassi è simile a quella di .Move():

```
            File.Copy(@"c:\testc\test.txt",@"c:\testc\nuova\test.txt");
```

Per cancellare un file, si usa invece il metodo **.Delete():**

```
           File.Delete(@"c:\testc\nuova\test.txt");
```

La classe **FileInfo** viene usata principalmente per recuperare le informazioni da unfile e dispone di diverse proprietà, la cui lista si può tranquillamente vedere sul sito microsoft; noi vediamo adesso solamente un semplice esempio esplicativo:

```
using System;
using System.IO;
    class miofile {
    public static void Main() {
            FileInfo fInfo = new  FileInfo(@"c:\testc\esempiXML.xml");
            Console.WriteLine("Nome:\t\t" + fInfo.Name);
            Console.WriteLine("Nome Completo:\t" + fInfo.FullName);
            Console.WriteLine("Directory:\t" + fInfo.Directory);
            Console.WriteLine("Nome Directory:\t" + fInfo.DirectoryName);
            Console.WriteLine("Solo lettura:\t" + fInfo.IsReadOnly);
            Console.WriteLine("Creazione:\t" + fInfo.CreationTime);
            Console.WriteLine("Ultimo Accesso:\t" + fInfo.LastAccessTime);
            Console.WriteLine("Ultima Modifica:" + fInfo.LastWriteTime);
            Console.WriteLine("Creazione:\t" + fInfo.CreationTime);  }
    }
```

Come si vede la procedura per recuperare informazioni da un file è molto semplice e non necessita di essere spiegata è sufficiente sapere quali sono i metodi che utilizza.
Oltre che a leggere le informazioni di un file posso utilizzare **.FileInfo** per creare un file (tramile la classe **.FileStream**)

```
           System.IO.FileInfo fInfo = new System.IO.FileInfo(@"c:\testc\creo.txt");
           FileStream fileStrm = fInfo.Create();
```

o per cancellarlo:

```
            System.IO.FileInfo fInfo = new System.IO.FileInfo(@"c:\testc\creo.txt");
            fInfo.Delete();
```

Allo stesso modo, la classe **.DirectoryInfo** permette di lavorare con le directory:

```
using System;
using System.IO;
    class miofile {
    public static void Main() {
            DirectoryInfo dinfo = new DirectoryInfo(@"c:\testc");
            DirectoryInfo dinuova = new DirectoryInfo(@"c:\testc\nuova");
            DirectoryInfo dinuova2 = new DirectoryInfo(@"c:\testc\nuova2");
            Console.WriteLine(dinfo.Name);
            dinuova.Create();
            dinuova2.Create();
            dinuova2.Delete(); }
    }
```

Creo tre oggetti: dinfo, dinuova, dinuova2 relative alle directory interessate (i parametri tra parentesi) e poi per ciascuno oggetto applico il metodo interessato.
Se invece volessi recuperare la lista dei file presenti in una cartella utilizzo il metodo **.GetFiles()** della classe:

```
DirectoryInfo dinfo = new DirectoryInfo(@"c:\testc");
            foreach (var fi in dinfo.GetFiles())  {
                Console.WriteLine(fi.Name); }
```

Il metodo .GetFiles() permette anche di recuperare una lista di file in base ad una stringa di ricerca:

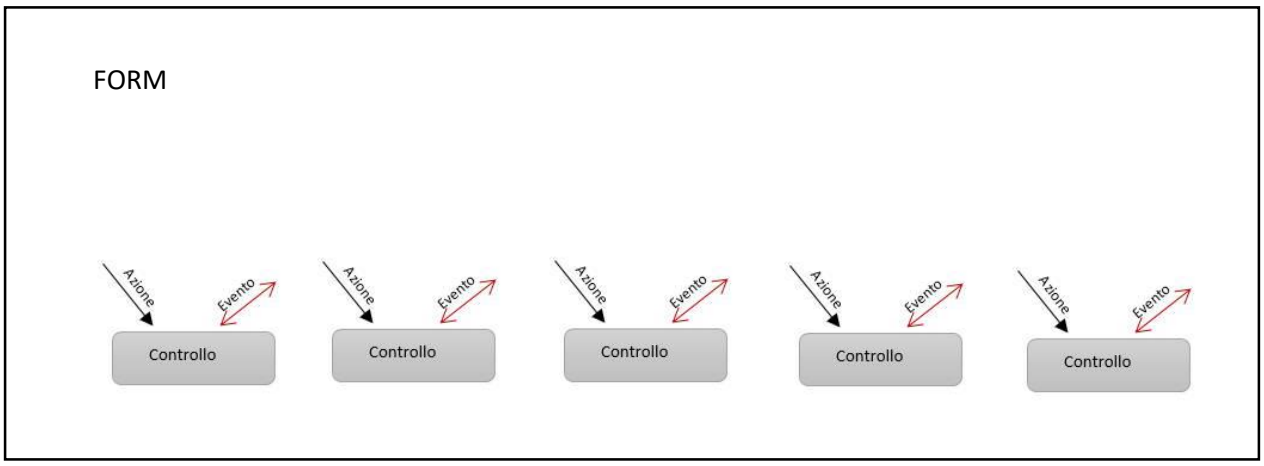
```
foreach (var fi in dinfo.GetFiles("*.exe"))
```

CAPITOLO X

LE FINESTRE

Il 6 aprile 1992 la Microsoft rilascia il programma Windows 3.1. All'epoca era poco più di un file manager a finestre evoluto più che un sistema operativo a differenza di altri sistemi che si trovavano in atri computer come Apple, Acron Archimedes, Amiga, Atari St, ecc.
Da allora l'utilizzo di finestre e mouse è diventato imperativo nei programmi e in c# la classe che si occupa di questo prende il nome di **Form** e si trova nel *Name Space* **System.Windows.Forms**.
Possiamo paragonare un Form ad un foglio bianco dove il programmatore inserisce i controlli, i pulsanti, i menù e tutto quello che serve per creare una GUI (Graphics User Interface) per consentire all'utente un utilizzo più semplice del programma.
Il metodo più semplice per progettare programmi che utilizzano i Form è sicuramente Visual Studio (che è possibile scaricare gratis nella versione Community), ma nulla ci vieta di creare interfacce grafiche con il nostro Frameworks e il nostro editor preferito.

A differenza di quanto avveniva nei programmi visti fino ad ora dove la comunicazione tra l'utente ed il programma avveniva tramite metodi di ingresso ed uscita (domanda -> acquisizione dati -> elaborazione) nei programmi con interfaccia grafica è una serie di eventi su oggetti che determina l'elaborazione dell'azione da compiere.



Se ad esempio un'utente clicca su un bottone questo può cambiare colore o nome, producendo l'effetto visuale appropriato, e contemporaneamente genererà un evento (OnClick).
Un controllo è composto da **proprietà** (che ne determinano l'aspetto, le caratteristiche, ecc.) e da **eventi** che ne determinano il comportamento (OnClick, MouseHover, MouseLeave, ecc.)
In un windows form possono essere utilizzati moltissimi componenti e controlli che vanno dal semplice bottone alla casella di testo, dalla DataGrid (una griglia che contiene dati) al browser web; per una lista dei controlli da usare si consiglia di visitare i siti microsoft ufficiali :
https://docs.microsoft.com/it-it/dotnet/framework/winforms/controls/controls-to-use-on-windows-forms
https://docs.microsoft.com/it-it/dotnet/framework/winforms/controls/windows-forms-controls-by-function

IL PRIMO FORM

Consideriamo adesso il seguente programma:

```
using System;
using System.Windows.Forms;
   class MiaForm: Form {
                    public MiaForm(){
   }
   public static void Main(){
                Application.Run( new MiaForm() );
                }
   }
```

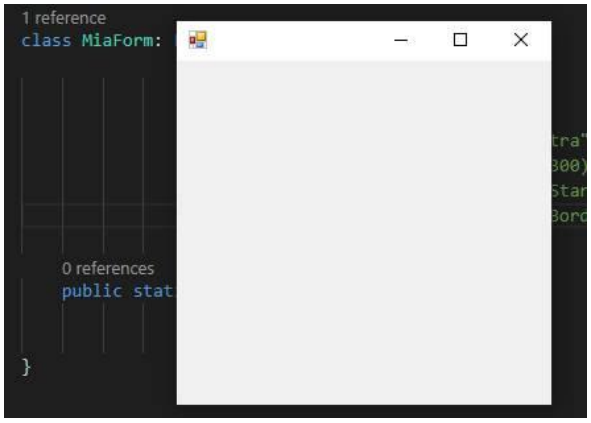
Il namespace **System.Windows.Forms** contiene le classi alle quali appartengono i controlli: *Form, TextBox, Label, Button,* eccetera.
Vediamo poi che abbiamo creato la classe MiaForm derivandola dalla classe Form (per derivare una classe vedere la prima parte del corso), che contiene le caratteristiche di una finestra windows base.
Una prima novità che compare e:

```
Application.Run(new MiaForm());
```

Che equivale a scrivere:

```
MainForm formPrincipale = new MiaForm();
Application.Run(formPrincipale);
```

dove il metodo **Run()** svolge la funzione di dare l'avvio vero e proprio al programma; quindi creo l'oggetto formPrincipale dalla classe MiaForm e lo eseguo.
Compiliamo il nostro programma e lanciamolo, vediamo che viene creata una finestra (piccola) senza null'altro.



Ricapitolando diciamo che, rispetto alle applicazioni console in una Windows Form:

- la classe principale deriva dalla classe Form;
- la classe principale definisce un costruttore, il quale viene invocato automaticamente quando viene creato il form;
- il metodo Main() contiene una sola istruzione, nella quale viene creato un form, il cui riferimento viene passato come argomento al metodo Run() della classe Application e che determina l'esecuzione vera e propria del programma.

Certo così la non ha molto senso, iniziamo con il personalizzare la nostra finestra.
Per prima cosa aggiungiamo lo spazio nomi System.Drawing, che ci permetterà di utilizzare il costrutto **Size** per impostare le dimensioni della nostra finestra tramite due valori interi separati da una virgola che esprimono il numero di pixel orizzontali e verticali del Form.

Definiamo poi le proprietà della nostra finestra dandole un titolo, un bordo, posizionandola al centro dello schermo e dandole l'opacità.

```
public MiaForm() {
    this.Opacity = .85;
    this.Text = "La mia finestra";
    this.Size = new Size(800,600);
    this.StartPosition = FormStartPosition.CenterScreen;
    this.FormBorderStyle=FormBorderStyle.Sizable;
}
```

La proprietà **.StartPosition** serve, appunto, a determinare la posizione iniziale del Form ed i parametri accettati sono:

- **CenterParent:** Il form risulta centrato rispetto al relativo form padre.
- **CenterScreen:** Il form viene visualizzato al centro dello schermo con le dimensioni specificate come dimensioni del form.
- **Manual:** La posizione del form è determinata dalla proprietà Location.
- **WindowsDefaultBounds:** Il form viene visualizzato nella posizione predefinita di Windows, con i limiti determinati dalle impostazioni predefinite di Windows.
- **WindowsDefaultLocation:** Il form viene visualizzato nella posizione predefinita di Windows, con le dimensioni specificate come dimensioni del form.

La proprietà **.FormBorderStyle** serve a determinare il tipo di bordo:

- **Fixed3D:** bordo tridimensionale fisso.
- **FixedDialog:** bordo spesso e fisso, di stile simile a quello di una finestra di dialogo.
- **FixedSingle:** bordo a riga singola fisso.
- **FixedToolWindow:** bordo della finestra degli strumenti non ridimensionabile. Non viene visualizzata alcuna finestra degli strumenti nella barra delle applicazioni o nella finestra visualizzata quando l'utente preme ALT+TAB.
- **None:** nessun bordo.
- **Sizable:** bordo ridimensionabile.
- **SizableToolWindow:** bordo ridimensionabile della finestra degli strumenti. Non viene visualizzata alcuna finestra degli strumenti nella barra delle applicazioni o nella finestra visualizzata quando l'utente preme ALT+TAB.

La proprietà **.Text**, in questo caso è usata per scrivere un testo nel mordo superiore del Form, ovvero il titolo, mentre la proprietà **.Opacity** esprime (tramite valori che vanno da 0 a 1) il grado di trasparenza.

IL PRIMO FORM: I CONTROLLI

Adesso che abbiamo creato il nostro contenitore (Form) vediamo di inserire i primi controlli; abbiamo detto che il Name Space System.Windows.Forms contiene molti controlli ma, tra tutti, i più utilizzati sono sicuramente:

- **Label:** I controlli Label vengono utilizzati per mostrare un testo che non può essere modificato dall'utente.
- **TextBox:** vengono usate per ottenere l'input da parte dell'utente o per visualizzare il testo; al suo interno il testo non risulta formattato (per questo si utilizza il *RichTextBox*).
- **Button:** consente all'utente di eseguire le operazioni desiderate facendo clic su di esso.

Ogni controllo dispone di proprietà e permette di eseguire delle azioni.

Poiché nel momento stesso in cui il form viene visualizzato l'interfaccia dev'essere già completa i controlli devono essere impostati ed inseriti nell'interfaccia del costruttore.

```
public MiaForm(){
                        this.Opacity = 0.95;
                        this.Text = "La mia finestra";
                        this.Size = new Size(800,600);
                        this.StartPosition = FormStartPosition.CenterScreen;
                        this.FormBorderStyle=FormBorderStyle.Sizable;

                        Label label1 = new Label(); //creo label
                          label1.BorderStyle = BorderStyle.FixedSingle;
                          label1.Text="Inserisci il tuo nome";
                          label1.Size = new Size (200,30);
                          label1.Location = new Point(300,100);
                          label1.TextAlign = ContentAlignment.MiddleCenter;
                          Controls.Add(label1); //aggiungo label1 al form

                        TextBox text1 = new TextBox(); // creo un textbox
                          text1.Size = new Size (200,30);
                          text1.Text = "scrivi qua il tuo nome";
                          text1.Location = new Point (300,150);
                          Controls.Add(text1); //aggiungo text1 al form

                        Button button1 = new Button(); // creo un pulsante
                          button1.Text="Premimi!";
                          button1.Size = new Size (200,60);
                          button1.Location = new Point(300,180);
                          button1.TextAlign = ContentAlignment.MiddleCenter;
                          Controls.Add(button1); //aggiungo button1 al form

                    }
```

Vediamo che abbiamo creato i nostri controlli come oggetti dalle rispettive classi e per ogni controllo abbiamo assegnato delle proprietà.
L'accesso alle proprietà di un controllo si ottiene sempre nello stesso modo:

```
nome-controllo.nome-proprietà = valore
```

Per le varie proprietà si consiglia di consultare le risorse online del sito microsoft:
https://docs.microsoft.com/it-it/dotnet/api/system.windows.forms.label?view=netframework-4.8
https://docs.microsoft.com/it-it/dotnet/api/system.windows.controls.textbox?view=netframework-4.8
https://docs.microsoft.com/it-it/dotnet/api/system.windows.forms.button?view=netframework-4.8

IL PRIMO FORM: GLI EVENTI

In .Net il concetto di evento si riferisce a qualcosa che è accaduto o che sta accadendo e che è influenza l'esecuzione del programma come ad esempio la pressione di un tasto sulla tastiera, di uno dei pulsanti del mouse, il passare sopra un oggetto con il puntatore, eccetera.
È bene ricordare che non sempre gli eventi corrispondo ad un'azione dell'utente, ad esempio l'apertura di un Form è un evento ma non è generato dall'utente ma come conseguenza dell'apertura del programma (che invece è un evento generato dall'utente), mentre la chiusura del programma, o del Form, è un evento reale generato dall'utente.
Semplificando possiamo dire che un evento reale è qualcosa che viene rilevato dall'interfaccia e che corrisponde ad una "risposta" da parte del programma. Un evento è qualcosa di potenziale, che non produce nulla di per sè se non la possibilità, da parte del programmatore, di agganciare una parte di codice a quell'evento. Per fare questo si usa un **gestore di evento**, che è un metodo che viene eseguito automaticamente in risposta all'evento stesso, e che definisce due parametri e non ritorna alcun valore la cui sintassi è:

```
void nome-metodo (object sender, Classe-informazioni-evento e)
```

Il parametro sender (che sta per «mandante») rappresenta un riferimento al controllo che ha sollevato l'evento, mentre e rappresenta l'evento che ha ricevuto (ad esempio la pressione del tasto del mouse).
Una volta creato l'evento che viene generato non di deve fare altro che attaccare un gestore di evento a un evento fornendo al metodo (nel nostro esempio) button1_Click la delega per gestire l'evento Click sul Bottone:

```
button1.Click +=new EventHandler(button1_Click);
```

```
public MiaForm(){
    this.Opacity = 0.95;
    this.Text = "La mia finestra";
    this.Size = new Size(800,600);
    this.StartPosition = FormStartPosition.CenterScreen;
    this.FormBorderStyle=FormBorderStyle.Sizable;

    Label label1 = new Label(); //creo label
      label1.BorderStyle = BorderStyle.FixedSingle;
      label1.Text="Inserisci il tuo nome";
      label1.Size = new Size (200,30);
      label1.Location = new Point(300,100);
      label1.TextAlign = ContentAlignment.MiddleCenter;
      Controls.Add(label1); //aggiungo label1 al form

    TextBox text1 = new TextBox(); // creo un textbox
      text1.Size = new Size (200,30);
      text1.Text = "scrivi qua il tuo nome";
      text1.Location = new Point (300,150);
      Controls.Add(text1); //aggiungo text1 al form

    Button button1 = new Button();
      button1.Text="Premimi!";
      button1.Size = new Size (200,60);
      button1.Location = new Point(300,180);
      button1.TextAlign = ContentAlignment.MiddleCenter;
      Controls.Add(button1); //aggiungo button1 al form
      button1.Click +=new EventHandler(button1_Click); //Gestore di Evento

    void button1_Click(object sender, System.EventArgs e) {
    string nome = Convert.ToString(text1.Text);
    MessageBox.Show("Ciao " + nome );
    }
}
```

Il tutto, ovviamente all'interno del mio Form.

NOTE: I nomi sender ed e vengono utilizzati principalmente per convenzione e sono arbitrari e qualunque valore va altrettanto bene

Se volessi aprire più form sarà sufficiente creare un nuovo bottone che, come azione, istanzia una nuova classe *MiaForm* o una nuova form ed esegue quindi l'azione **.Show()**.

```
    Button button2 = new Button();
      button2.Text="APRE FORM";
      button2.Size = new Size (200,60);
      button2.Location = new Point(100,180);
      button2.TextAlign = ContentAlignment.MiddleCenter;
      Controls.Add(button2); //aggiungo button1 al form
      button2.Click +=new EventHandler(button2_Click);

    Button button3 = new Button();
      button3.Text="APRE UGUALE";
      button3.Size = new Size (400,60);
      button3.Location = new Point(100,180);
      button3.TextAlign = ContentAlignment.MiddleCenter;
```

```
                        Controls.Add(button3); //aggiungo button1 al form
                        button3.Click +=new EventHandler(button3_Click);

                    void button2_Click(object sender, System.EventArgs e) {
                        Form2 MiaForm2 = new Form2();
                        MiaForm2.Show();
                    }

                    void button3_Click(object sender, System.EventArgs e) {
                        MiaForm FormUguale = new MiaForm();
                        FormUguale.Show();
                    }
                }

class Form2: Form {
                public Form2(){
                    this.Text = "La mia 2° finestra";
                    this.Size = new Size(800,600);
                    this.StartPosition = FormStartPosition.CenterScreen;
                    this.FormBorderStyle=FormBorderStyle.Sizable;
                    }
                 }
```

Tutto all'interno del Form principale in questo caso.

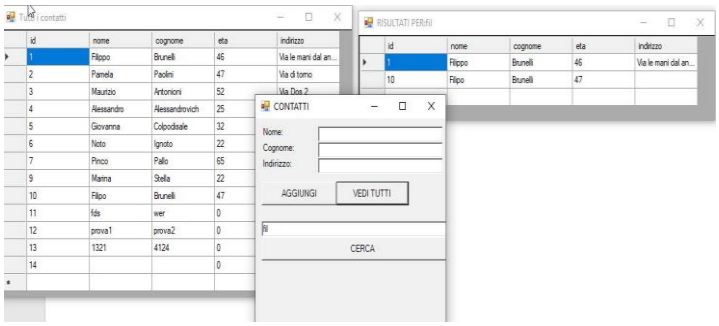
All'interno del secondo Form posso creare nuovi controlli ed utilizzare gli stessi nomi del primo senza che questo crei problemi:

```
class Form2: Form {
                public Form2(){
                    this.Text = "La mia 2° finestra";
                    this.Size = new Size(800,600);
                    this.StartPosition = FormStartPosition.CenterScreen;
                    this.FormBorderStyle=FormBorderStyle.Sizable;
                    Button button1 = new Button();
                      button1.Text="CHIUDIMI";
                      button1.Size = new Size (200,60);
                      button1.Location = new Point(300,180);
                      button1.TextAlign = ContentAlignment.MiddleCenter;
                      Controls.Add(button1); //aggiungo button1 al form
                      button1.Click +=new EventHandler(button1_Click);
                    void button1_Click(object sender, System.EventArgs e) {
                     this.Close();
                    }
                }
            }
```

Per chiudere un form posso usare l'azione **.Close()**.

CONCLUSIONI: UN PROGRAMMA COMPLETO

Abbiamo visto come creare un form e inserirvi dei controlli. Certo crearlo in questo modo, anche se non impossibile è molto complicato, per questo è meglio affidarsi ad un IDE come visual studio.
Nell'ultimo esempio creeremo un semplicissimo programma che andrà ad interfacciarsi con il nostro database che abbiamo usato nelle lezioni precedenti e farà uso di solo tre semplici Form per vedere quando diventa complicato creare un programma.
I form saranno:



- 1 form principale che permette di inserire un nuovo contatto (solo nome, cognome e indirizzo per comodità) ed aprire gli altri form
- 1 form che apre tutti i contatti
- 1 form che apre solo le ricerche per nome

Iniziamo con i Name System che utilizzeremo:

```
using System;
using System.Windows.Forms;
using System.Drawing;
using System.Data;
using System.Data.OleDb;
```

Come si vede la nostra form di partenza sarà la più complessa da scrivere in quanto contiene diversi controlli. Per prima cosa creiamo il nostro form come abbiamo visto fino ad ora. Questo form non dovrà avere dimensioni eccessive visto che i controlli che contiene saranno pochi:

```
public MainForm(){
                        this.Text = "CONTATTI";
                        this.Size = new Size(320,300);
                        this.StartPosition = FormStartPosition.CenterScreen;
                        this.FormBorderStyle=FormBorderStyle.Sizable;

                    }
```

Iniziamo quindi a creare al suo interno i controlli di cui abbiamo bisogno:
3 label, 4 text box e 3 pulsanti:

```
                        Label label1 = new Label();
                        label1.Text="Nome:";
                        label1.Size = new Size (90,20);
                        label1.Location = new Point(10,10);
                        Controls.Add(label1);

                        TextBox text1 = new TextBox();
                        text1.Size = new Size (200,30);
                        text1.Location = new Point (100,10);
                        Controls.Add(text1);

                        Label label2 = new Label();
                        label2.Text="Cognome:";
                        label2.Size = new Size (90,20);
                        label2.Location = new Point(10,30);
                        Controls.Add(label2);

                        TextBox text2 = new TextBox();
                        text2.Size = new Size (200,30);
                        text2.Location = new Point (100,30);
                        Controls.Add(text2);

                        Label label3 = new Label();
                        label3.Text="Indirizzo:";
                        label3.Size = new Size (90,20);
                        label3.Location = new Point(10,50);
                        Controls.Add(label3);

                        TextBox text3 = new TextBox();
                        text3.Size = new Size (200,30);
                        text3.Location = new Point (100,50);
                        Controls.Add(text3);

                        TextBox text4 = new TextBox();
                        text4.Text="Scrivi qua il nome da cercare e premi cerca";
                        text4.Size = new Size (320,30);
                        text4.Location = new Point (10,130);
                        Controls.Add(text4);

                        Button button1 = new Button();
                        button1.Text="AGGIUNGI";
                        button1.Size = new Size (120,30);
                        button1.Location = new Point(10,80);
                        button1.TextAlign = ContentAlignment.MiddleCenter;
```

```
                        Controls.Add(button1);
                        button1.Click +=new EventHandler(button1_Click);

                        Button button2 = new Button();
                        button2.Text="VEDI TUTTI";
                        button2.Size = new Size (120,30);
                        button2.Location = new Point(125,80);
                        button2.TextAlign = ContentAlignment.MiddleCenter;
                        Controls.Add(button2);
                        button2.Click +=new EventHandler(button2_Click);

                        Button button3 = new Button();
                        button3.Text="CERCA";
                        button3.Size = new Size (320,30);
                        button3.Location = new Point(10,150);
                        button3.TextAlign = ContentAlignment.MiddleCenter;
                        Controls.Add(button3);
                        button3.Click +=new EventHandler(button3_Click);
```

Ad ogni bottone andremo ad associare il rispettivo evento al click.
Il bottone button1 che si occupa di inserire un nuovo record è quello che ha il maggior numero di righe di codice:

```
void button1_Click(object sender, System.EventArgs e) {

    string nome = Convert.ToString(text1.Text);
    string cognome = Convert.ToString(text2.Text);
    string indirizzo = Convert.ToString(text3.Text);

    string strConn = "Provider=Microsoft.ACE.OLEDB.12.0; Data Source=C:\\testC\\database.accdb";
    OleDbConnection myConn = new OleDbConnection(strConn);
    OleDbCommand myCmd = new OleDbCommand();
    myCmd.CommandType = CommandType.Text;
    myCmd.CommandText = "INSERT INTO nomi (nome, cognome, indirizzo) " + "VALUES ('" + nome + "',
'" + cognome + "', '" + indirizzo + "')";
    myCmd.Connection = myConn;
    myConn.Open();
       myCmd.ExecuteNonQuery();
    myConn.Close();

    MessageBox.Show("DATI INSERITI NEL DATABASE");
    text1.Text="";
    text2.Text="";
    text3.Text="";
    }
```

Come si vede non facciamo altro che recuperare i valori presenti nei tre textbox ed inserirli all'interno del database (vedere lezione sui database).
Una volta che l'operazione è stata fatta viene visualizzato un messaggio che avverte l'utente della corretta esecuzione dell'operazione e viene "azzerato" il contenuto dei tre textbox semplicemente impostandone il valore a niente **text1.Text=""**.

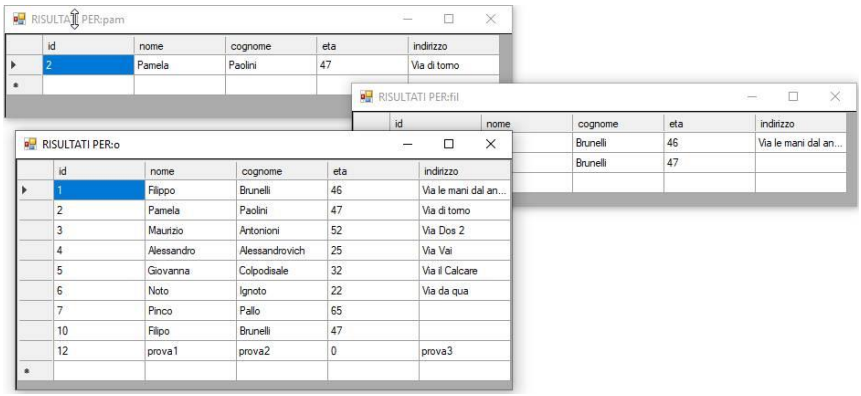
L'evento associato al bottone numero 2, invece, si limita ad istanziare un nuovo oggetto form2 ed a mostrarlo:

```
void button2_Click(object sender, System.EventArgs e) {
                        Form2 MiaForm2 = new Form2();
                        MiaForm2.Show(); }
```

Mentre il bottone 3, ha il compito di istanziare un nuovo oggetto Form dalla classe Cerca inviandovi il valore del text4, ovvero quello che dovrà cercare.

```
 void button3_Click(object sender, System.EventArgs e) {
                        Cerca Ricerca = new Cerca(text4.Text);
                        Ricerca.Show(); }
```

Questo mi permetterà di avere più finestre, una per ogni ricerca fata:



Il form2 ci presenterà un nuovo controllo chiamato **DataGridView** che è uno dei più comodi controlli presenti nel frameworks .Net per la visualizzazione dei dati.

```
class Form2: Form {
 public Form2(){
   this.Text = "Tutti i contatti";
   this.Size = new Size(600,600);
   this.StartPosition = FormStartPosition.CenterScreen;
   this.FormBorderStyle=FormBorderStyle.Sizable;

   BindingSource bSorgente = new BindingSource();

   DataGridView dataGridViewNomi = new DataGridView();
   dataGridViewNomi.Size = new Size (790,590);
   Controls.Add(dataGridViewNomi);

   string strConn = "Provider=Microsoft.ACE.OLEDB.12.0; Data Source=C:\\testC\\database.accdb";
   string strQuery =" Select * from nomi";
   OleDbConnection myConn = new OleDbConnection(strConn);
   OleDbDataAdapter myAdapter = new OleDbDataAdapter(strQuery, myConn);

   DataSet myDataSet = new DataSet();
           myAdapter.Fill(myDataSet, "nomi");
               DataTable tblNomi = myDataSet.Tables["nomi"];
           myAdapter.Dispose();

           bSorgente.DataSource = myDataSet.Tables["nomi"];
           dataGridViewNomi.DataSource = bSorgente;
                 }
}
```

La classe **DataGridView** visualizza dati in una griglia personalizzabile in formato tabulare come un foglio di excel. Il suo utilizzo è molto semplice tanto che, nel nostro caso è stato sufficiente associarla ad un dataset tramite il componente BindingSource che funge da canale e da origine dati per altri controlli ai quali è associato.
Nel nostro caso creiamo una sorgente (bsource)

```
BindingSource bSorgente = new BindingSource();
```

che colleghiamo al nostro dataset

```
bSorgente.DataSource = myDataSet.Tables["nomi"];
```

e poi al DataGridView

```
dataGridViewNomi.DataSource = bSorgente
```

Come si vede il processo è molto semplice e veloce.
Per quanto riguarda i DataSet si consiglia, se non si ricorda il loro uso, di ripassare la lezione sull'uso dei database in C#.

Il controllo DataGridView lo si crea come qualunque altro controllo istanziando la classe, ed allo stesso modo ne si settano le proprietà:

```
   DataGridView dataGridViewNomi = new DataGridView();
   dataGridViewNomi.Size = new Size (790,590);
   Controls.Add(dataGridViewNomi);
```

Per una più esaustiva descrizione della classe, visto la complessità e l'elasticità che la caratterizzano si consiglia di consultare la documentazione ufficiale Microsoft all'indirizzo: https://docs.microsoft.com/it-it/dotnet/api/system.windows.forms.datagridview?view=netframework-4.8

Note: il controllo DataGridView è altamente configurabile ed estendibile e fornisce molte proprietà, metodi ed eventi per personalizzare l'aspetto e il comportamento tanto che meriterebbe lui da solo uno solo capitolo.

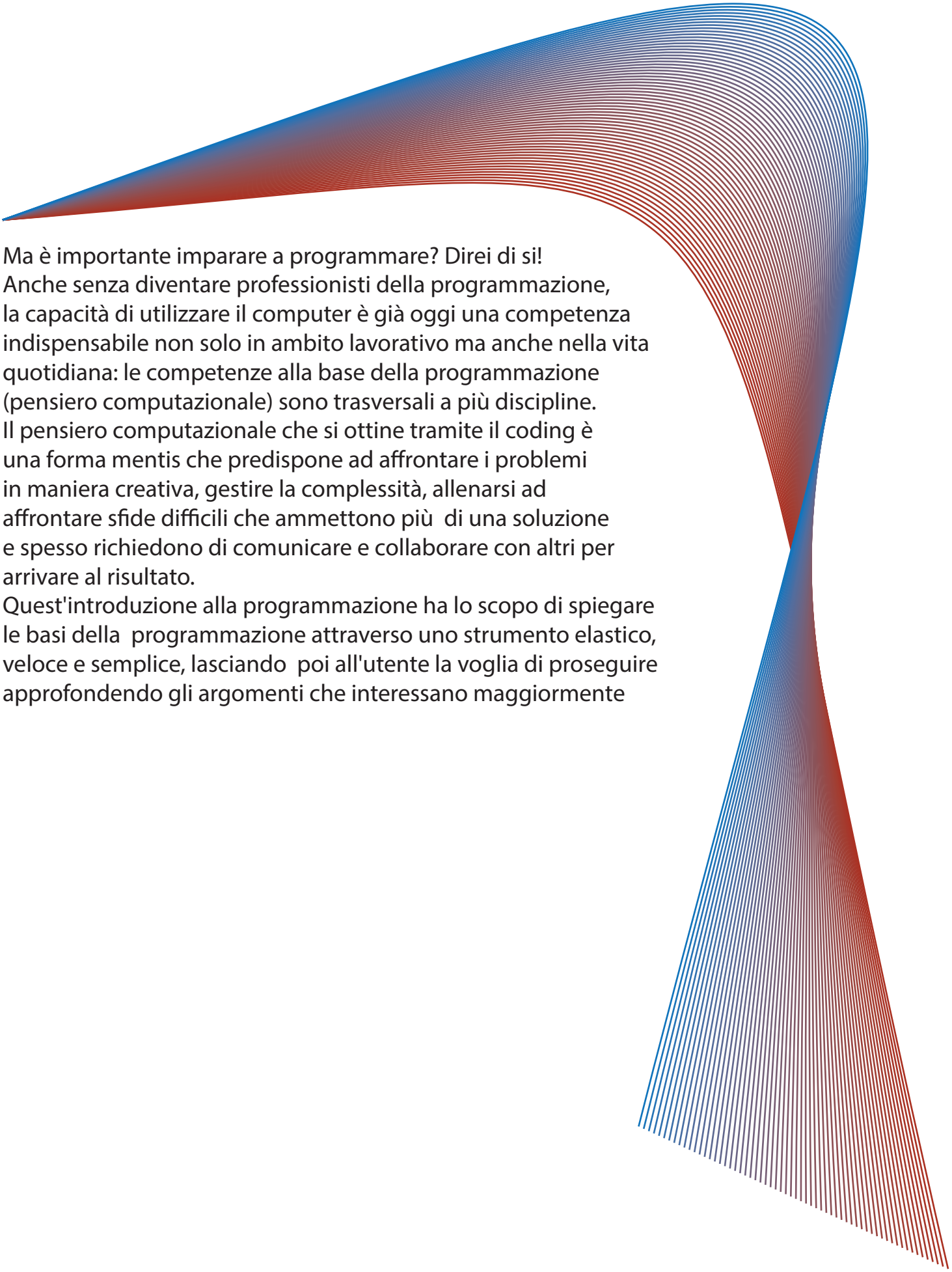
L'ultimo form, per finire, non fa altro che creare nuovamente un DataGridView ma, questa volta, si cercheranno solamente i parametri che vengono passati dal form1:

```
class  Cerca: Form {
            public Cerca(string ricerca){
                this.Text = "RISULTATI PER:" + ricerca;
                this.Size = new Size(600,600);
                this.StartPosition = FormStartPosition.CenterScreen;
                this.FormBorderStyle=FormBorderStyle.Sizable;

                DataGridView dataGridViewNomi = new DataGridView();
                BindingSource bsource = new BindingSource();

                dataGridViewNomi.Size = new Size (750,590);
                Controls.Add(dataGridViewNomi);

    string strConn = "Provider=Microsoft.ACE.OLEDB.12.0; Data Source=C:\\testC\\database.accdb";
    string strQuery =" SELECT * FROM nomi WHERE nome LIKE '%"+ricerca+"%' OR cognome LIKE '%" + r
icerca +"%'";
                OleDbConnection myConn = new OleDbConnection(strConn);
                OleDbDataAdapter myAdapter = new OleDbDataAdapter(strQuery, myConn);
                DataSet myDataSet = new DataSet();
                myAdapter.Fill(myDataSet, "nomi");
                DataTable tblNomi = myDataSet.Tables["nomi"];
                myAdapter.Dispose();

                bsource.DataSource = myDataSet.Tables["nomi"];
                dataGridViewNomi.DataSource = bsource;
                 }
        }
```

Come si vede il programma è molto semplice e abbiamo utilizzato la maggior parte dei concetti spiegati nelle lezioni precedenti.

CONSIDERAZIONI FINALI

Perché bisognerebbe imparare a programmare?
Diciamo che creare un programma, anche se semplice, è una sfida mentale ma anche un processo creativo che può dare molte soddisfazioni sia che lo si faccia per lavoro o per piacere e ci dà la possibilità di non essere sempre dipendenti da altri oltre che portarci ad una più completa conoscenza del computer: sviluppare il pensiero computazionale vuol dire sviluppare un processo mentale per la risoluzione di problemi che permette di operare a diversi livelli di astrazione del pensiero e che si può applicare anche nella vita di tutti i giorni. Per finire poi, diciamoci la verità, al giorno d'oggi, dove tutto è controllato dai computer sapre programmare, è un po' come avere dei super poteri…!



Ma è importante imparare a programmare? Direi di si!
Anche senza diventare professionisti della programmazione, la capacità di utilizzare il computer è già oggi una competenza indispensabile non solo in ambito lavorativo ma anche nella vita quotidiana: le competenze alla base della programmazione (pensiero computazionale) sono trasversali a più discipline.
Il pensiero computazionale che si ottine tramite il coding è una forma mentis che predispone ad affrontare i problemi in maniera creativa, gestire la complessità, allenarsi ad affrontare sfide difficili che ammettono più di una soluzione e spesso richiedono di comunicare e collaborare con altri per arrivare al risultato.
Quest'introduzione alla programmazione ha lo scopo di spiegare le basi della programmazione attraverso uno strumento elastico, veloce e semplice, lasciando poi all'utente la voglia di proseguire approfondendo gli argomenti che interessano maggiormente