"QUALUNQUE ALGORITMO PUÒ ESSERE IMPLEMENTATO UTILIZZANDO TRE SOLE STRUTTURE, LA SEQUENZA, LA SELEZIONE E IL CICLO, DA APPLICARE RICORSIVAMENTE ALLA COMPOSIZIONE DI ISTRUZIONI ELEMENTARI".

Sequenza : le istruzioni sono eseguite secondo l'ordine in cui sono scritte;
Selezione o Condizione : esiste una condizione da valutare e due possibili gruppi di istruzioni da eseguire; in funzione del valore della condizione, si sceglie un blocco oppure l'altro;
Iterazione : consente la ripetizione di un blocco di istruzioni, per un certo numero di volte, definito a seconda dell'obiettivo da raggiungere.

INTRODUZIONE ALLA PROGRAMMAZIONE CON L'USO DI C# II° PARTE

WWW.FILOWEB.IT
FILIPPO BRUNELLI

Introduzione alla programmazione con l'uso di C# II° parte

INDICE

CAPITOLO I
DOVE ERAVAMO RIMSTI 2
LE CLASSI ASTRATTE 2
LE INTERFACCE 4
LE INTERFACCE E LE CLASSI ASTRATTE: DIFFERENZE 6
LE STRUTTURE 7
ENUMERAZIONI 7

CAPITOLO II
VALORI E RIFERIMENTI 9
PAROLA CHIAVE OUT 10
IL CASTING 11
BOXING E UNBOXING 11

CAPITOLO III
GLI ARRAY COSA SONO E COME SI CREANO 13
LAVORARE CON GLI ARRAY 16
GLI ARRAYLIST 18

CAPITOLO IV
LE STRINGHE 20
CREAZIONE DELLE STRINGHE 20
AGGIUNGERE, RIMUOVERE, SOSTITUIRE 21
FORMATTAZIONE DI UN NUMERO 22
STRINGA DI FORMATO COMPOSTO 22
L'INTERPOLAZIONE DI STRINGHE 23
ALTRI METODI 23
LE SEQUENZE DI ESCAPE 25

CAPITOLO I

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 e ampliare la nostra conoscenza….

LE CLASSI ASTRATTE

Il concetto di classe astratta è molto generico e 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:

```
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()
                    { }
        }
```

Vediamo il listato completo:

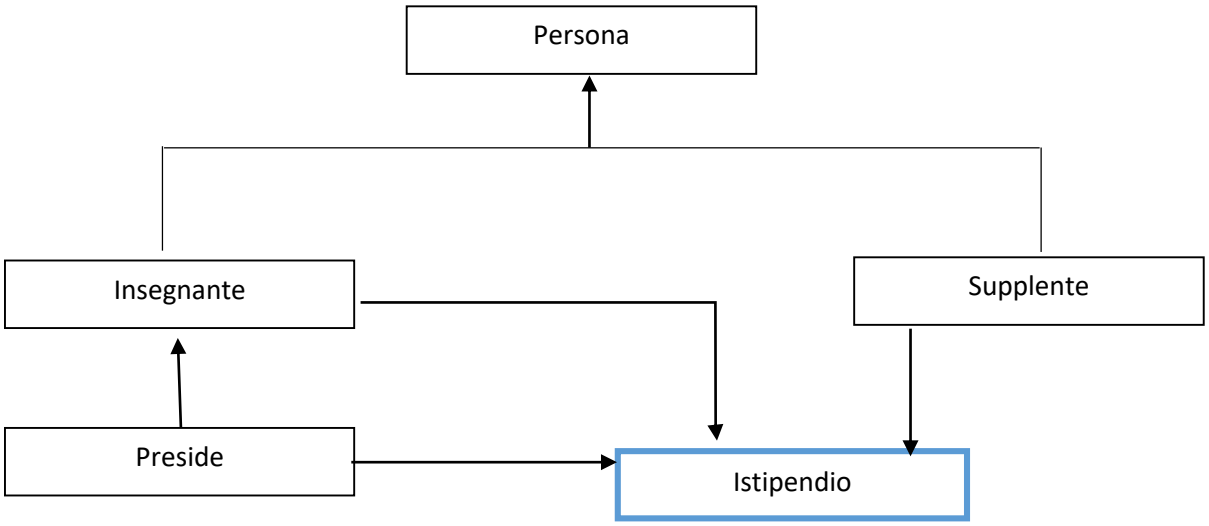
```
using System;
class EsempioAstratta
{
    public abstract class Forma
    {
        protected string aNome;
        protected int aLati;

        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; possono essere considerate una più leggera alternativa alle classi alle quali assomigliano: come le classi 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, il costruttore senza parametri implicito (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).

```
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");

    }
}
```

Nell'esempio sopra usiamo l'enumerazione per creare una corrispondenza tra la taglia di reggiseno italiana e quella Inglese; importante 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 .**GetName**s 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 II

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 III

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 IV

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 tuo intero numero 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 BENE: 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 e le tabulazioni, la creazione di apici o caratteri speciali; le sequenze di caratteri sono costituite da una barra rovesciata (\ ) seguita da una lettera o da una combinazione di cifre.

Le principali sequenze di escape che utilizzeremo sono:

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

```
using System;
   class Stringhe {
    public static void Main() {
        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

c:\testC>
```