

## *Modello di programmazione procedurale*

Anno 2018/2019

## Indice generale

<b>1</b>	<b>Introduzione.....</b>	<b>4</b>
1.1	Programmazione imperativa.....	4
1.2	Un esempio di programma imperativo.....	4
<b>2</b>	<b>Modello di programmazione procedurale.....</b>	<b>7</b>
2.1	Definizione di un metodo: intestazione e corpo.....	7
2.2	Chiamata (esecuzione) di un metodo.....	8
2.3	Modello procedurale: un primo esempio.....	8
2.3.1	Ambito di visibilità: variabili locali e variabili globali.....	9
2.4	Uso di variabili globali: locale “vs” globale.....	10
2.5	Conclusioni.....	12
2.5.1	Problemi dell’attuale versione.....	12
<b>3</b>	<b>Implementare funzioni “parametrizzate” .....</b>	<b>13</b>
3.1	Parametrizzare una funzione.....	15
3.1.1	Metodi con parametri e valore restituito.....	15
3.1.2	Uso del metodo ValoreMedio().....	16
3.1.3	Parametrizzare la funzione “numero alunni sopra la media” .....	16
3.2	Separare elaborazione da visualizzazione.....	17
3.3	“Eliminare” le variabili globali non necessarie.....	17
3.4	Conclusioni.....	18
<b>4</b>	<b>Funzioni “accessorie” .....</b>	<b>19</b>
4.1	Funzioni di input.....	19
4.1.1	Rendere esplicito l’intento di un’istruzione.....	20
4.2	Parametrizzare funzioni di output.....	20
4.2.1	Generalizzare una funzione.....	21
<b>5</b>	<b>Frequently Asked Questions.....</b>	<b>22</b>
5.1	Parametri e variabili locali di un metodo possono avere lo stesso nome? .....	22
5.2	I nomi dei parametri devono corrispondere al nome degli argomenti nell’istruzione di chiamata?.....	22
5.3	Il tipo degli argomenti deve essere uguale al tipo dei parametri?.....	22

5.4	Nella chiamata, gli argomenti devono essere variabili?.....	23
5.5	Due metodi possono avere lo stesso nome?.....	23
5.6	Una variabile locale può avere lo stesso nome di una variabile globale? 23	
5.7	Un metodo che definisce dei parametri e/o restituisce un valore può usare variabili globali?.....	24
5.8	Due variabili locali dichiarate in blocchi distinti possono avere lo stesso nome?.....	25
5.9	Il valore restituito da un metodo deve essere utilizzato?.....	25
5.10	Il valore restituito (se usato) deve essere assegnato a una variabile?...26	
5.11	Un metodo può avere più istruzioni “return”?.....	26
5.12	Un metodo “void” può avere una o più istruzioni “return”?.....	26
5.13	Un metodo può restituire un vettore?.....	27
5.14	I parametri di un metodo possono essere modificati?.....	27
5.15	Modificando i parametri si modificano gli argomenti corrispondenti?..28	

# 1 Introduzione

Il tutorial introduce i *metodi* e il loro ruolo nel *modello di programmazione procedurale*. Partirò da un riepilogo sulla *programmazione imperativa*; quindi mostrerò come questo modello sia limitato e dunque adatto soltanto a programmi molto semplici. Successivamente introdurrò i metodi e mostrerò che consentono di:

- Separare le funzioni del programma, migliorando l'organizzazione e la comprensibilità del codice.
- Favorire l'incapsulamento e il riutilizzo delle funzioni del programma.

## 1.1 Programmazione imperativa

Da wikipedia (adattato):

*Nella programmazione imperativa il programma è inteso come un insieme di istruzioni, ciascuna delle quali può essere pensata come un'ordine impartito a...*

Dunque, il programma è composto da una lista di istruzioni, alcune delle quali – *if*, *while*, *for*, *switch*, ... – servono a modificare il flusso di esecuzione, normalmente sequenziale.

Questo semplice modello di programmazione è in grado di esprimere qualunque algoritmo, non importa quanto complesso. Nonostante ciò si tratta di un modello limitato, il cui utilizzo, anche a fronte di problemi semplici, produce programmi complicati, poco leggibili e difficilmente modificabili.

Di seguito lo metterò alla prova con un semplice problema di programmazione.

## 1.2 Un esempio di programma imperativo

Dato in input l'elenco delle altezze (in cm) degli alunni di una classe, realizza un programma che:

- 1 Visualizzi i dati inseriti.
- 2 Calcoli l'altezza media.
- 3 Calcoli il numero di alunni con un'altezza superiore alla media.

Implementerò separatamente le funzioni del programma. (Esistono implementazioni alternative, che accorpano alcune funzioni.)

```
static void Main(string[] args)
{
    Console.WriteLine("Inserisci n° alunni:");
    int numAlunni = int.Parse(Console.ReadLine());
    double[] altezze = new double[numAlunni];
    for (int i = 0; i < altezze.Length; i++)
    {
        Console.WriteLine("Altezza n°{0}", i + 1);
        altezze[i] = int.Parse(Console.ReadLine());
    }
}
```

```

Console.WriteLine();
for (int i = 0; i < altezze.Length; i++)
{
    Console.WriteLine("{0}", altezze[i]);
}
double somma = 0;
for (int i = 0; i < altezze.Length; i++)
{
    somma = somma + altezze[i];
}
double altezzaMedia = somma / altezze.Length;
Console.WriteLine("\nAltezza media : {0}", altezzaMedia);
int numAlunniSopraMedia = 0;
for (int i = 0; i < altezze.Length; i++)
{
    if (altezze[i] > altezzaMedia)
        numAlunniSopraMedia++;
}
Console.WriteLine("\nN° alunni alti : {0}", numAlunniSopraMedia);
}

```

Si tratta di un programma molto semplice; inoltre, l'aver usato nomi significativi per le variabili rende il codice leggibile e facile da modificare. Ma ha un difetto: *è un unico blocco di codice che implementa più funzioni.*

Uno dei principi fondamentali della programmazione afferma che le funzioni di un programma dovrebbero essere implementate separatamente le une dalle altre. È un principio ingegneristico generale, che riguarda la progettazione di qualsiasi dispositivo: suddividerlo in più componenti, ognuno dotato di una funzione specifica, che collaborano al funzionamento generale.

Ispirandomi a questo principio, mostro nuovamente il programma, questa volta evidenziando le funzioni implementate:

```

static void Main(string[] args)
{
    Console.Write("Inserisci n° alunni:");
    int numAlunni = int.Parse(Console.ReadLine());
    double[] altezze = new double[numAlunni];

    for (int i = 0; i < altezze.Length; i++)
    {
        Console.Write("Altezza n°{0}", i + 1);
        altezze[i] = int.Parse(Console.ReadLine());
    }

    Console.WriteLine();
    for (int i = 0; i < altezze.Length; i++)
    {
        Console.WriteLine("{0}", altezze[i]);
    }
}

```

Inserimento  
altezze

Visualizzazione  
altezze

<pre>double somma = 0; for (int i = 0; i &lt; altezze.Length; i++) {     somma = somma + altezze[i]; } double altezzaMedia = somma / altezze.Length; Console.WriteLine("\nAltezza media : {0}", altezzaMedia);</pre>	Calcolo altezza media
<pre>int numAlunniSopraMedia = 0; for (int i = 0; i &lt; altezze.Length; i++) {     if (altezze[i] &gt; altezzaMedia)         numAlunniSopraMedia++; } Console.WriteLine("N° alunni alti : {0}", numAlunniSopraMedia);</pre>	Calcolo n° alunni sopra la media
}	

Sulla base di questa schematizzazione del codice, che evidenzia le funzioni del programma, procederò ad applicare il *modello procedurale di programmazione*.

## 2 Modello di programmazione procedurale

Il *modello di programmazione procedurale* estende il *modello imperativo*. Da wikipedia:

*La programmazione procedurale consiste nel creare dei blocchi di codice identificati da un nome... Questi sono detti sottoprogrammi...*

L'idea è che ogni funzione del programma sia implementata mediante un blocco di codice distinto: la *procedura*, o *sotto programma*. In C#, i sotto programmi sono chiamati *metodi*.

Un metodo rappresenta dunque un'*unità di codice* che è possibile utilizzare mediante il suo nome. Come vedremo, l'uso di metodi è fondamentale; infatti:

- 1 Forniscono un "significato" al codice. Un blocco di codice, di per sé, non trasmette alcun significato. Il nome di un metodo (purché appropriato) trasmette l'intento (*ordina*, *ricerca*, *inserisci*, etc) del procedimento eseguito. Rende dunque il programma comprensibile, più semplice da scrivere e da correggere..
- 2 Sono alla base del principio di *incapsulamento*. Un metodo consente di eseguire un procedimento senza dipendere dalla sua implementazione. Ad esempio, si può ordinare un vettore semplicemente eseguendo il metodo `Ordina()`. Successivamente, si può decidere di impiegare un altro algoritmo senza che il resto del programma debba essere modificato.
- 3 Sono alla base del processo di "riusabilità" del codice, poiché rendono semplice il reimpiego di una funzione, nello stesso programma e in altri programmi.

### 2.1 Definizione di un metodo: intestazione e corpo

Segue la sintassi (semplificata) della definizione di un *metodo*:

```
class Program
{
    static <tipo> <nome_metodo>(<parametri>) firma, o intestazione del metodo
    {
        ... corpo del metodo
    }
}
```

Un metodo è dunque composto da un'*intestazione* e un *corpo*. L'intestazione stabilisce la sintassi, e dunque i vincoli, sull'uso del metodo. Viene chiamata anche *firma*<sup>1</sup>, e non a caso; infatti, non possono esistere due metodi con la stessa firma, poiché devono differenziarsi per il nome e/o la lista dei parametri.

Il *corpo* del metodo definisce il blocco di codice da eseguire quando il metodo viene "chiamato".

1 Per essere precisi, l'elemento **<tipo>** non fa parte della firma del metodo.

## 2.2 Chiamata (esecuzione) di un metodo

L'esecuzione del metodo viene convenzionalmente definita *chiamata* (del metodo). L'istruzione di chiamata deve rispettare la sintassi:

```
<nome_metodo>(<argomenti>)
```

Gli *argomenti* (o *parametri attuali*), se esistono, devono corrispondere nel numero e nel tipo ai *parametri* dichiarati nell'intestazione del metodo (*parametri formali*).

In questo contesto, il codice contenente la chiamata al metodo viene convenzionalmente definito *codice chiamante*, o *metodo chiamante* se si fa riferimento all'intero metodo.

## 2.3 Modello procedurale: un primo esempio

Torniamo al programma proposto in 1.2; l'obiettivo è modificarlo usando il modello di programmazione procedurale. Esistono varie soluzioni, la più semplice delle quali è definire un metodo per ogni funzione del programma. Ma ciò solleva dei problemi.

Considera le funzioni di inserimento e visualizzazione. Per implementarle mediante dei metodi basta "estrarre" i due blocchi di codice da `Main()` e assegnare loro un nome:

```
static void Main(string[] args)
{
    InserisciAltezze();
    VisualizzaAltezze();
    ...
}

static void InserisciAltezze()                Inserimento altezze
{
    Console.Write("Inserisci n° alunni:");
    int numAlunni = int.Parse(Console.ReadLine());
    double[] altezze = new double[numAlunni]; // "altezze" è locale al metodo

    for (int i = 0; i < altezze.Length; i++)
    {
        Console.Write("Altezza n°{0}", i + 1);
        altezze[i] = int.Parse(Console.ReadLine());
    }
}

static void VisualizzaAltezze()              Visualizzazione altezze
{
    Console.WriteLine();
    for (int i = 0; i < altezze.Length; i++) // in questo blocco "altezze" non esiste!
    {
        Console.WriteLine("{0}", altezze[i]);
    }
}
```

Ma la nuova versione del programma non è corretta. Il problema è connesso all'uso della variabile `altezze` e al suo **ambito di visibilità**.



### 2.3.1 Ambito di visibilità: variabili locali e variabili globali

C'è una regola che riguarda la parte di codice in cui è possibile usare una variabile<sup>2</sup>:

*una variabile può essere usata soltanto nel blocco di codice all'interno del quale è dichiarata*

Questa regola definisce l'**ambito di visibilità** della variabile, e proprio a causa di questa, ad esempio, che l'indice di un ciclo `for()` può essere usato soltanto dentro il blocco di codice del ciclo.

Nel programma, la variabile `altezze` viene dichiarata nel metodo `InserisciAltezze()` e dunque non può essere usata nel metodo `VisualizzaAltezze()`. Si dice che la *variabile è locale al metodo nel quale è dichiarata*.

Poiché la stessa variabile deve essere usata in più metodi, è necessario dichiararla in un blocco di codice che li comprende tutti: la classe `Program`.

```
class Program
{
    static double[] altezze; // "altezze" è visibile (globale) ovunque nel programma

    static void Main(string[] args)
    {
        InserisciAltezze();
        VisualizzaAltezze();
        ...
    }

    static void InserisciAltezze()
    {
        Console.WriteLine("Inserisci n° alunni: ");
        int numAlunni = int.Parse(Console.ReadLine());
        altezze = new double[numAlunni];
        ...
    }

    static void VisualizzaAltezze()
    {
        Console.WriteLine();
        for (int i = 0; i < altezze.Length; i++)
            ...
    }
}
```

Nota bene: la dichiarazione è preceduta dalla parola `static`; ciò riguarda qualunque variabile o metodo definiti a livello di classe<sup>3</sup>.

Adesso, la variabile `altezze` si dice **globale**, poiché è accessibile ovunque all'interno di `Program`. Si dice anche che la variabile è *condivisa (shared)* dai tutti i metodi di `Program`.

2 La questione è in realtà un po' più complessa, e non riguarda soltanto le variabili.

3 Anche qui la questione è in realtà più complessa.

## 2.4 Uso di variabili globali: locale “vs” globale

L'operazione compiuta con `altezze` deve essere replicata con tutte le variabili utilizzate in più di un metodo. Occorre pertanto definire globale anche `altezzaMedia`, poiché viene calcolata nel metodo `CalcolaAltezzaMedia()` e utilizzata in `CalcolaNumAlunniSopraMedia()`:

```
class Program
{
    static double[] altezze;
    static double altezzaMedia;

    static void Main(string[] args)
    {
        InserisciAltezze();
        VisualizzaAltezze();
        CalcolaAltezzaMedia();
        CalcolaNumAlunniSopraMedia();
    }

    static void InserisciAltezze() {...}

    static void VisualizzaAltezze(){...}

    static void CalcolaAltezzaMedia()
    {
        double somma = 0;
        for (int i = 0; i < altezze.Length; i++)
        {
            somma = somma + altezze[i];
        }
        altezzaMedia = somma / altezze.Length;
        Console.WriteLine("\nAltezza media : {0}", altezzaMedia);
    }

    static void CalcolaNumAlunniSopraMedia()
    {
        int numAlunniSopraMedia = 0;
        for (int i = 0; i < altezze.Length; i++)
        {
            if (altezze[i] > altezzaMedia)
                numAlunniSopraMedia++;
        }
        Console.WriteLine("N° alunni alti : {0}", numAlunniSopraMedia);
    }
}
```

A questo punto è legittimo chiedersi se non sia vantaggioso avere solo variabili globali, in modo da non porsi il problema di dove dichiararle. La risposta è no! Infatti:

*ogni variabile deve essere dichiarata nel blocco di codice dove è necessaria*

Esistono vari motivi per questo:

- Aumenta la leggibilità del codice, poiché la dichiarazione e l'uso di una variabile sono collocati nella stessa "porzione" di programma. Il codice che usa la variabile fornisce un contesto che ne chiarisce il ruolo.
- Evita un "affollamento" di dichiarazioni in testa al programma, con l'effetto di nascondere le variabili importanti, utilizzate in molte parti del codice.
- Evita *effetti collaterali* indesiderati: utilizzare la stessa variabile in metodi diversi per scopi diversi, col rischio di introdurre bug nel programma.

Dunque: una variabile dovrebbe essere dichiarata globale soltanto se è strettamente necessario utilizzarla in due o più metodi.

Sulla base di queste considerazioni, le variabili `numAlunni`, `Somma` e `numAlunniSopraMedia` devono restare locali ai rispettivi metodi:

```
class Program
{
    static double[] altezze;
    static double altezzaMedia;

    static void Main(string[] args)
    {
        InserisciAltezze();
        VisualizzaAltezze();
        CalcolaAltezzaMedia();
        CalcolaNumAlunniSopraMedia();
    }

    static void InserisciAltezze()
    {
        Console.Write("Inserisci n° alunni: ");
        int numAlunni = int.Parse(Console.ReadLine());
        altezze = new double[numAlunni];
        ...
    }

    static void VisualizzaAltezze(){...}

    static void CalcolaAltezzaMedia()
    {
        double somma = 0;
        for (int i = 0; i < altezze.Length; i++)
        {
            somma = somma + altezze[i];
        }
        altezzaMedia = somma / altezze.Length;
        Console.WriteLine("\nAltezza media : {0}", altezzaMedia);
    }
}
```

```

static void CalcolaNumAlunniSopraMedia()
{
    int numAlunniSopraMedia = 0;
    for (int i = 0; i < altezze.Length; i++)
    {
        if (altezze[i] > altezzaMedia)
            numAlunniSopraMedia++;
    }
    Console.WriteLine("N° alunni alti : {0}", numAlunniSopraMedia);
}
}

```

## 2.5 Conclusioni

Consideriamo il risultato ottenuto dopo aver applicato il modello di programmazione procedurale. La versione del programma presentata in 1.2 è un blocco monolitico di codice che implementa quattro funzioni. Nella nuova versione, le funzioni sono chiaramente delineate.

Il contenuto di `Main()` è ora estremamente semplice e, soprattutto, esprime chiaramente l'intento del codice. Di fatto, appare come una "scaletta" delle operazioni da svolgere; operazioni i cui dettagli sono implementati attraverso i vari metodi del programma.

I metodi hanno un nome significativo, che esprime l'intento della funzione implementata. Sono brevi e semplici da comprendere. (È possibile "abbracciarne" il codice con uno sguardo, senza dover "scrollare" la schermata.)

I metodi mettono in pratica il *principio di incapsulamento*: usare una funzione (il calcolo dell'altezza media, ad esempio), semplicemente facendo riferimento al suo nome. Ciò semplifica il codice e, entro certi limiti, consente di isolare le funzioni tra loro e di modificarne l'implementazione senza influenzare il resto del programma.

### 2.5.1 Problemi dell'attuale versione

Esistono ancora due aspetti da considerare, che riguardano i metodi `CalcolaAltezzaMedia()` e `CalcolaNumAlunniSopraMedia()`.

Innanzitutto, entrambi implementano una duplice funzione, calcolare e visualizzare il risultato del calcolo. Questo viola il principio: *un metodo dovrebbe implementare una sola funzione*. (Un indizio di questa violazione: il nome di entrambi non riflette completamente la funzione svolta.)

Seconda cosa, sono gli unici metodi a utilizzare la variabile globale `altezzaMedia`. Nella sostanza, dunque, questa variabile non è realmente globale, poiché interessa soltanto un sottoinsieme del programma. Ma, allo stato attuale, dichiararla globale è l'unico modo per poter implementare i due procedimenti in metodi distinti.

### 3 Implementare funzioni “parametrizzate”

Considera una nuova versione del problema (ho evidenziato le parti aggiuntive in grassetto):

Dato in input l'elenco delle altezze (in cm) e **i pesi (in kg)** degli alunni di una classe, realizza un programma che:

- 1 Visualizzi i dati inseriti.
- 2 Calcoli l'altezza media **e il peso medio**.
- 3 Calcoli il numero di alunni con un'altezza superiore all'altezza media **e il numero di alunni con un peso superiore al peso medio**.

Per quanto riguarda la gestione, l'input e la visualizzazione dei dati non c'è molto su cui riflettere:

```
class Program
{
    static double[] altezze;
    static double[] pesi;
    static double altezzaMedia;

    static void Main(string[] args)
    {
        InserisciDatiAlunni();
        VisualizzaDatiAlunni();
        ...
    }
    static void InserisciDatiAlunni()
    {
        Console.Write("Inserisci n° alunni: ");
        int numAlunni = int.Parse(Console.ReadLine());
        altezze = new double[numAlunni];
        pesi = new double[numAlunni];
        for (int i = 0; i < altezze.Length; i++)
        {
            Console.WriteLine("\nAlunno n°{0}", i + 1);
            Console.Write("Altezza: ");
            altezze[i] = int.Parse(Console.ReadLine());
            Console.Write("Peso   : ");
            pesi[i] = int.Parse(Console.ReadLine());
        }
    }

    static void VisualizzaDatiAlunni()
    {
        Console.WriteLine("\n{0,7}{1,7}", "Altezza", "Peso");
        for (int i = 0; i < altezze.Length; i++)
        {
            Console.WriteLine("{0,7}{1,7}", altezze[i], pesi[i]);
        }
    }
}
```

```
...  
}
```

Per quanto riguarda i punti 2) e 3), invece, si possono immaginare due alternative: aggiungere nuove funzioni – peso medio e alunni con peso superiore alla media – oppure modificare quelle esistenti. Seguono entrambe le soluzioni, per brevità riferite soltanto al calcolo del valore medio:

```
class Program  
{  
    static double[] altezze;  
    static double[] pesi;  
    static double altezzaMedia;  
    static double pesoMedio;  
    ...  
  
    // soluzione 1: aggiunta della funzione "calcolo peso medio"  
    static void CalcolaPesoMedio()  
    {  
        double somma = 0;  
        for (int i = 0; i < pesi.Length; i++)  
        {  
            somma = somma + pesi[i];  
        }  
        pesoMedio = somma / pesi.Length;  
        Console.WriteLine("\nPeso medio : {0}", pesoMedio);  
    }  
  
    // soluzione 2: integrare calcolo peso medio a funzione esistente (altezza media)  
    static void CalcolaMediaAltezzaEPeso()  
    {  
        double sommaPesi = 0;  
        double sommaAltezze = 0;  
        for (int i = 0; i < altezze.Length; i++)  
        {  
            sommaAltezze = sommaAltezze + altezze[i];  
            sommaPesi = sommaPesi + pesi[i];  
        }  
        altezzaMedia = sommaAltezze / altezze.Length;  
        pesoMedio = sommaPesi / pesi.Length;  
        Console.WriteLine("\nAltezza e peso medi : {0} e {1}", altezzaMedia, pesoMedio);  
    }  
}
```

In realtà esiste una terza soluzione, che evita gli inconvenienti delle prime due, che sono:

- Duplicare il codice (il calcolo dei valori medi richiede lo stesso procedimento).
- Accorpare due funzioni nello stesso metodo (in realtà tre, perché il metodo incorpora anche la visualizzazione dei valori medi).

## 3.1 Parametrizzare una funzione

Considera le funzioni di calcolo dell'altezza media e del peso medio:

### Calcolo altezza media

```
static void CalcolaAltezzaMedia()
{
    double somma = 0;
    for (int i= 0; i < altezze.Length; i++)
    {
        somma = somma + altezze[i];
    }

    altezzaMedia = somma / altezze.Length;
    ...
}
```

### Calcolo peso medio

```
static void CalcolaPesoMedio()
{
    double somma = 0;
    for (int i = 0; i < pesi.Length; i++)
    {
        somma = somma + pesi[i];
    }

    pesoMedio = somma / pesi.Length;
    ...
}
```

I procedimenti sono identici, cambiano soltanto le variabili coinvolte.

Ebbene, è possibile implementare un metodo senza vincolarlo a specifiche variabili. Ciò consente di impiegarlo tutte le volte in cui è necessaria una determinata funzione, *indipendentemente dalle variabili alle quali deve essere applicata*.

### 3.1.1 Metodi con parametri e valore restituito

Un metodo con *parametri* implementa un procedimento senza stabilire a quali variabili applicarlo; al loro posto usa dei parametri, specificati tra parentesi nell'intestazione.

Un metodo con *valore restituito* (o *valore di ritorno*) restituisce il risultato di un procedimento senza stabilire dove memorizzarlo o come usarlo; mediante l'istruzione `return` stabilisce il valore restituito.

Sulla scorta di queste affermazioni, ecco come scrivere un metodo che restituisce il valore medio di un vettore di `double`:

```
static double ValoreMedio(double[] valori)
{
    double somma = 0;
    for (int i= 0; i < valori.Length; i++)
    {
        somma = somma + valori[i];
    }

    double media = somma / valori.Length;
    return media;
}
```

Nota bene: ho evidenziato il parametri e la variabile che memorizza il valore restituito per facilitare il confronto con i metodi `CalcolaAltezzaMedia()` e `CalcolaPesoMedio()`.

Due considerazioni:

- Ho usato dei nomi generici (`ValoreMedio`, `valori` e `media`), perché la funzione implementata non riguarda più altezze, i pesi, etc, ma una elenco generico di `double`.
- Il tipo del valore restituito deve essere compatibile con il tipo dichiarato nell'intestazione del metodo.

### 3.1.2 Uso del metodo `ValoreMedio()`

È nell'istruzione di chiamata che si stabilisce a quali variabili sarà applicato il metodo:

```
class Program
{
    static double[] altezze;
    static double[] pesi;
    static double altezzaMedia;
    static double pesoMedio;

    static void Main(string[] args)
    {
        InserisciDatiAlunni();
        VisualizzaDatAlunni();

        // lo applica al vettore "altezze" e memorizza il risultato in "altezzaMedia"
        altezzaMedia = ValoreMedio(altezze);

        // lo applica al vettore "pesi" e memorizza il risultato in "pesoMedio"
        pesoMedio = ValoreMedio(pesi);

        ...
    }
    ...
}
```

### 3.1.3 Parametrizzare la funzione “numero alunni sopra la media”

Anche i procedimenti che calcolano il numero di alunni più alto e più pesante possono essere parametrizzati mediante un metodo:

```
static int NumeroValoriSopraMedia(double[] valori, double media)
{
    int conta = 0;
    for (int i = 0; i < valori.Length; i++)
    {
        if (valori[i] > media)
            conta++;
    }
    return conta;
}
```



Ecco come usarlo:

```
static void Main(string[] args)
{
    ...
    int numAlunniAlti = NumeroValoriSopraMedia(altezze, altezzaMedia);
    int numAlunniPesanti = NumeroValoriSopraMedia(pesi, pesoMedio);
    ...
}
```

## 3.2 Separare elaborazione da visualizzazione

La parametrizzazione delle due funzioni – valore medio e numero valori sopra la media – consente di separare il calcolo del risultato dalla sua visualizzazione:

```
class Program
{
    static double[] altezze;
    static double[] pesi;
    static double altezzaMedia;
    static double pesoMedio;

    static void Main(string[] args)
    {
        InserisciDatAlunni();
        VisualizzaDatAlunni();

        altezzaMedia = ValoreMedio(altezze);
        pesoMedio = ValoreMedio(pesi);
        Console.WriteLine("\nAltezza e peso medi : {0} e {1}", altezzaMedia, pesoMedio);

        int numAlunniAlti = NumeroValoriSopraMedia(altezze, altezzaMedia);
        int numAlunniPesanti = NumeroValoriSopraMedia(pesi, pesoMedio);

        Console.WriteLine("\nAlunni più alti della media: {0}", numAlunniAlti);
        Console.WriteLine("\nAlunni più pesanti della media: {0}", numAlunniPesanti);
    }
    ...
}
```

Si tratta di una questione importante; si implementa un metodo che restituisce un risultato proprio per separare il calcolo di un valore dall'uso che se ne fa. In questo modo si può usare lo stesso metodo in scenari diversi, poiché l'uso che viene fatto del risultato viene stabilito nel codice chiamante e non all'interno del metodo.

## 3.3 “Eliminare” le variabili globali non necessarie

L'uso di metodi con parametri consente di dichiarare locali delle variabili che prima dovevano essere obbligatoriamente globali. È il caso delle variabili `altezzaMedia` e `pesoMedio`.

```

class Program
{
    static double[] altezze;
    static double[] pesi;
    static double altezzaMedia;
    static double pesoMedio;

    static void Main(string[] args)
    {
        InserisciDatAlunni();
        VisualizzaDatAlunni();

        double altezzaMedia = ValoreMedio(altezze);
        double pesoMedio = ValoreMedio(pesi);
        Console.WriteLine("\nAltezza e peso medi : {0} e {1}", altezzaMedia, pesoMedio);

        int numAlunniAlti = NumeroValoriSopraMedia(altezze, altezzaMedia);
        int numAlunniPesanti = NumeroValoriSopraMedia(pesi, pesoMedio);

        Console.WriteLine("\nAlunni più alti della media: {0}", numAlunniAlti);
        Console.WriteLine("\nAlunni più pesanti della media: {0}", numAlunniPesanti);
    }
    ...
}

```

### 3.4 Conclusioni

La possibilità di definire dei parametri e di restituire un valore consente di implementare funzioni indipendenti dalle variabili alle quali vengono applicate. Ciò è particolarmente utile per quei procedimenti di carattere generale: somma, media, ricerca, minimo, massimo, ordinamento, etc. Si tratta di procedimenti utilizzabili in molti scenari, dunque conviene parametrizzarli in modo da poterli reimpiegare più volte, sia nello stesso programma, che in programmi distinti.

Un altro "indizio" che suggerisce la possibilità di parametrizzare i metodi è quando una variabile globale viene usata soltanto da alcuni di essi. In questo caso è opportuno valutare la possibilità di modificare i metodi, aggiungendo un parametro ed evitando dunque l'uso diretto della variabile globale. Lo scopo è quello di eliminare la variabile globale, trasformandola in una variabile locale.

## 4 Funzioni “accessorie”

Finora ho parlato di “funzioni del programma”, intendendo procedimenti necessari a soddisfare le richieste del problema; ma nella programmazione è molto comune individuare funzioni utili, se non necessarie, alla realizzazione del programma, anche se non strettamente attinenti al problema. Di seguito fornisco alcuni esempi.

### 4.1 Funzioni di input

Considera quante volte hai scritto un simile codice:

```
Console.Write(<messaggio>);  
<tipo> <variabile> = <tipo>.Parse(Console.ReadLine());  
...
```

(Ovviamente, visualizzando un messaggio significativo e dichiarando una variabile specifica.)

Ebbene, questo semplice frammento di codice rappresenta in realtà una funzione: l’input di un valore, preceduto dalla visualizzazione di un messaggio per l’utente. È conveniente implementarla mediante dei metodi:

```
static double ReadDouble(string prompt)  
{  
    Console.Write(prompt);  
    return double.Parse(Console.ReadLine());  
}  
  
static double ReadInt(string prompt)  
{  
    Console.Write(prompt);  
    return int.Parse(Console.ReadLine());  
}
```

Segue una nuova versione di `InserisciDatiAlunni()` che usa i nuovi metodi:

```
static void InserisciDatiAlunni()  
{  
    int numAlunni = ReadInt("Inserisci n° alunni: ");  
    altezze = new double[numAlunni];  
    pesi = new double[numAlunni];  
    for (int i = 0; i < altezze.Length; i++)  
    {  
        Console.WriteLine("\nAlunno n°{0}", i + 1);  
        altezze[i] = ReadDouble("Altezza: ");  
        pesi[i] = ReadDouble("Peso : ");  
    }  
}
```

I due metodi forniscono diversi vantaggi:

- Semplificano il codice di input dei dati.
- Rendono il codice più chiaro, perché esprimono esplicitamente l'intento dell'istruzione: leggere un `double` o un intero.
- Incapsulano una funzione, che può essere successivamente modificata senza influenzare il resto del codice.

L'ultimo punto è particolarmente interessante. Si può immaginare di realizzare una nuova versione dei metodi che, nel caso di input non valido, lo chiede nuovamente, invece di far "crashare" il programma. Una volta realizzata, questa sarebbe automaticamente adottata in tutto il programma.

#### 4.1.1 *Rendere esplicito l'intento di un'istruzione*

Non esiste frammento di codice troppo breve che non valga la pena incapsulare in un metodo, purché rappresenti una funzione significativa. Considera le seguenti istruzioni, che scrivi normalmente per sospendere il programma:

```
Console.ReadLine()
```

```
Console.ReadKey()
```

In realtà, entrambi i metodi hanno la funzione di accettare l'input dell'utente, ma, in un certo contesto, vengono usati per sospendere il programma. Ebbene, perché non rendere specifico questo intento?

```
static void Pausa()
{
    Console.ReadKey();
}
```

## 4.2 Parametrizzare funzioni di output

Supponi di dover visualizzare spesso a delle specifiche coordinate; quindi di dover scrivere frequentemente il codice:

```
Console.SetCursorPosition(<colonna>, <riga>);
Console.Write(<messaggio>);
```

Ebbene, può essere utile incapsularlo in un metodo:

```
static void Write(int x, int y, string text)
{
    Console.SetCursorPosition(x, y);
    Console.Write(text);
}
```

## 4.2.1 Generalizzare una funzione

Supponi di realizzare un programma che, tra le altre cose, visualizza una cornice con le dimensioni dello schermo. È appropriato implementare questa funzione mediante un metodo:

```
static void Cornice()
{
    int y2 = Console.WindowHeight - 1;
    int x2 = Console.WindowWidth - 1;
    for (int x = 0; x <= x2; x++)
    {
        Write(x, 0, "*");
        Write(x, y2, "*");
    }
    for (int y = 0; y <= y2; y++)
    {
        Write(0, y, "*");
        Write(x2, y, "*");
    }
}
```


(Nota bene, ho utilizzato il metodo `Write()` definito in precedenza).

Questo approccio, però, non sfrutta la possibilità di parametrizzare il procedimento, in modo da poterlo adattare in diversi contesti, per usare un altro carattere per la cornice, oppure per disegnare una cornice di dimensioni qualsiasi e in qualsiasi posizione dello schermo.

È dunque utile definire un metodo con parametri:

```
static void Cornice(int posX, int posY, int width, int height, string ch)
{
    int x2 = posX + width - 1;
    int y2 = posY + height - 1;

    for (int x = posX; x <= x2; x++)
    {
        Write(x, posY, ch);
        Write(x, y2, ch);
    }
    for (int y = posY; y <= y2; y++)
    {
        Write(posX, y, ch);
        Write(x2, y, ch);
    }
}
```

Nota bene: è l'identico procedimento del metodo precedente, ma, parametrizzandolo, è stato "slegato" da un uso specifico e reso adatto ad essere utilizzato in svariati contesti. Ad esempio, la seguente istruzione disegna una cornice di  sulla parte alta dello schermo:

```
Cornice(0, 0, Console.WindowWidth, 3, "-");
```

## 5 Frequently Asked Questions

Finora ho trattato i metodi senza soffermarmi sulle regole che riguardano la loro definizione e il loro impiego. Di seguito prendo in considerazione alcune di queste regole, facendolo sotto forma di F.A.Q. e cioè di domande comuni alle quali fornisco una risposta e una spiegazione.

### 5.1 Parametri e variabili locali di un metodo possono avere lo stesso nome?

**NO.** Sarebbe come dichiarare la stessa variabile due volte nello stesso blocco di codice.

### 5.2 I nomi dei parametri devono corrispondere al nome degli argomenti nell'istruzione di chiamata?

**NO.** Nel verificare la corretta corrispondenza tra gli argomenti e i parametri, il linguaggio guarda soltanto alla compatibilità tra i tipi; il nome è irrilevante. Ad esempio:

```
static void Main(string[] args)
{
    double[] numeri = { 1, 2, 3, 4 };
    double somma = SommaValori(numeri); // ok
}

static double SommaValori(double[] valori) {...}
```

Nota bene: l'argomento ha il nome `numeri`, mentre il parametro si chiama `valori`.

### 5.3 Il tipo degli argomenti deve essere uguale al tipo dei parametri?

**(strettamente) NO.** Deve essere *compatibile*, che non è esattamente la stessa cosa; infatti vale la regola utilizzata per l'assegnazione. Nell'istruzione di assegnazione:

**<variabile> = <espressione>;**

il tipo dell'espressione deve essere compatibile verso quello della variabile. Per questo, ad esempio, il seguente codice è valido:

```
int a = 10;
double n = a * 2; // tipo dell'espressione è int (double <- int è ok)
```

Infatti, il tipo `int` è compatibile verso il tipo `double`. Ma non vale l'inverso:

```
double a = 10;
int n = a * 2; // tipo dell'espressione è double (int <- double è scorretto)
```

Infatti, il tipo `double` non è compatibile verso il tipo `int`. (Nota bene: in tutto questo, il valore delle variabili è irrilevante.)

Nel passaggio "argomenti → parametri" accade lo stesso: ogni argomento viene assegnato al parametro corrispondente applicando le regole dell'assegnazione.

## 5.4 Nella chiamata, gli argomenti devono essere variabili?

**NO.** Poiché nel passaggio “argomenti → parametri” valgono le regole dell’assegnazione. Dunque, l’argomento deve essere un’espressione compatibile con il parametro corrispondente.

Nel seguente esempio, il metodo `Cubo()` viene chiamato quattro volte:

```
static void Main(string[] args)
{
    double n = 2;
    int a = 20;
    double cubo = Cubo(n);           // variabile double : OK
    cubo = Cubo(a);                  // variabile int : OK
    Console.WriteLine(Cubo(30));     // costante int : OK
    Console.WriteLine(Cubo("20"));   // costante stringa : errore!
}

static double Cubo(double a)
{
    return a * a * a;
}
```

## 5.5 Due metodi possono avere lo stesso nome?

**SI.** Due o più metodi possono avere lo stesso nome; è una tecnica definita *overloading*. Al contrario, non possono avere la stessa *firma*, e cioè nome + numero e/o tipo dei parametri.

Nota bene, il tipo del valore restituito non contribuisce alla *firma*. Ad esempio:

```
static void MetodoA(int a) {...}
static void MetodoB() {...}
static int MetodoC() {...} // errore: la sua firma collide con quella di MetodoB!
```

`MetodoA` ha una *firma* univoca, ma `MetodoB` e `MetodoC` hanno la stessa *firma*, poiché la differenza tra `void` e `int` nel tipo del metodo non viene considerata.

## 5.6 Una variabile locale può avere lo stesso nome di una variabile globale?

**SI.** È uno dei vantaggi dell’incapsulamento: ciò che è dichiarato dentro un metodo non influenza ciò che sta fuori di esso.

Se una variabile locale ha lo stesso nome di una globale, si dice che la prima “nasconde” la seconda. Ciò significa che, dentro al metodo, l’uso della variabile fa riferimento a quella locale e non a quella globale.

## 5.7 Un metodo che definisce dei parametri e/o restituisce un valore può usare variabili globali?

**SI.** Il linguaggio non pone vincoli sul contenuto di un metodo, purché rispetti le regole sintattiche.

Semmai, la questione è un'altra: si definisce un metodo con parametri per implementare una *funzione parametrizzata*, se il metodo elabora delle variabili globali, ecco che può essere usato soltanto in quegli scenari che prevedono tali variabili.

Ad esempio, nel seguente codice il metodo `NumAltezzeSopra()` restituisce il numero di elementi del vettore `altezze` superiori a un certo valore, espresso come parametro:

```
class Program
{
    static double[] altezze;
    static void Main(string[] args)
    {
        altezze = new double[] { 1.76, 1.80, 1.82, 1.90 }; // sostituisce input dei dati
        ...
        int alti = NumAltezzeSopra(1,77);
        Console.WriteLine(altri); //-> 3
    }

    static int NumAltezzeSopra(double altezza)
    {
        int contaAlti = 0;
        for (int i = 0; i < altezze.Length; i++)
        {
            if (altezze[i] > altezza)
                contaAlti++;
        }
        return contaAlti;
    }
}
```

Il metodo funziona, ma è utilizzabile soltanto con il vettore `altezze`, nonostante implementi una funzione facilmente parametrizzabile e dunque riutilizzabile:

```
class Program
{
    static double[] altezze;
    static void Main(string[] args)
    {
        altezze = new double[] { 1.76, 1.80, 1.82, 1.90 }; // sostituisce input dei dati
        ...
        int alti = NumValoriSopra(altezze, 1,77);
        Console.WriteLine(altri); //-> 3
    }
    static int NumValoriSopra(double[] valori, double beraglio)
    {
        int conta = 0;
```



```

    for (int i = 0; i < valori.Length; i++)
    {
        if (valori[i] > bersaglio)
            conta++;
    }
    return conta;
}

```

Ciò detto, si possono immaginare funzioni applicabili a uno specifico contesto, ma che conviene comunque "semi parametrizzare" per poterne adattare il funzionamento. In questo caso ha senso definire un metodo con parametri, ma che usa ugualmente delle variabili globali.

## 5.8 Due variabili locali dichiarate in blocchi distinti possono avere lo stesso nome?

**SI.** Se i due blocchi sono allo stesso livello (nessuno contiene l'altro).

**NO.** Se uno dei due blocchi contiene l'altro.

Nell'esempio che segue sono dichiarate due variabili `i`, una a livello di metodo, l'altra nel secondo ciclo `for`:

```

static void VisualizzaClassi(string[] classeA, string[] classeB)
{
    int i;
    for (i = 0; i < classeA.Length; i++)
    {
        Console.WriteLine(classeA[i]);
    }
    for (int i = 0; i < classeB.Length; i++) // errore: "i" non può essere redichiarata!
    {
        Console.WriteLine(classeA[i]);
    }
}

```

## 5.9 Il valore restituito da un metodo deve essere utilizzato?

**NO.** Il valore può essere ignorato, ma ci si dovrebbe interrogare sul perché si decide di farlo.

Nel seguente esempio, le chiamate a due metodi distinti ignorano il valore restituito; ma mentre nel secondo caso si tratta di una scelta consapevole, nel primo caso è con tutta probabilità un bug:

```

static void Main(string[] args)
{
    Cubo(20); //ignorare il valore è probabilmente un bug
    ...
    Console.ReadKey(); //il tasto premuto non ha importanza
}

```

Chiamare `Cubo()` per poi non utilizzare il risultato prodotto non ha alcun senso! Mentre chiamare `ReadKey()` per sospendere l'esecuzione del programma è legittimo, poiché, in questo caso, quale sia il tasto premuto dall'utente non ha alcuna importanza.

## 5.10 Il valore restituito (se usato) deve essere assegnato a una variabile?

**NO.** Il valore *può* essere assegnato a una variabile, come può essere utilizzato in un'espressione, o passato direttamente come argomento a un altro metodo.

Ad esempio:

```
static void Main(string[] args)
{
    double cubo = Cubo(10);           // assegna a variabile "cubo"
    Console.Write(Cubo(30));          // passa come argomento a metodo Write()
    double ris = Cubo(30) * 10        // lo usa in una espressione
}
```

## 5.11 Un metodo può avere più istruzioni “return”?

**SI.** È utile quando il procedimento può terminare in base a più condizioni. Un classico esempio è rappresentato dal metodo di ricerca di un elemento. Il seguente metodo cerca il parametro `bersaglio` nel vettore, restituendo la sua posizione, o -1 se non lo trova:

```
static int IndiceDi(string[] elementi, string bersaglio)
{
    for (int i = 0; i < elementi.Length; i++)
    {
        if (elementi[i] == bersaglio)
            return i;
    }
    return -1;
}
```

L'alternativa è utilizzare una variabile nella quale memorizzare la posizione di `bersaglio`, inizializzandola a -1. Ma l'uso di un doppio `return` rende il codice più semplice e performante.

## 5.12 Un metodo “void” può avere una o più istruzioni “return”?

**SI.** È utile quando l'esecuzione di parte o di tutto il procedimento del metodo dipende da una condizione. Ovviamente, l'istruzione `return` non deve specificare alcun valore.

Ad esempio, il seguente metodo visualizza un elenco di nominativi, oppure un messaggio informativo se l'elenco è vuoto:

```
static void VisualizzaNomi(string[] nomi)
{
    if (nomi.Length == 0)
    {
        Console.WriteLine("L'elenco è vuoto");
        return;
    }
    for(int i = 0; i < nomi.Length; i++)
    {
        Console.WriteLine(nomi[i]);
    }
}
```

Nota bene: è possibile scrivere il metodo senza l'istruzione `return`, utilizzando una `if...else`. In alcuni casi, comunque, è più semplice utilizzare `return` e anticipare la terminazione del metodo.

## 5.13 Un metodo può restituire un vettore?

**SI.** Un metodo può restituire un valore di tipo qualsiasi, e quindi anche un *array*.

Spesso, un metodo simile è utile quando occorre implementare un procedimento che restituisca un nuovo vettore a partire da uno esistente. Il seguente esempio, dato un vettore di numeri interi, ne restituisce un altro contenente i numeri in posizione pari:

```
static int[] NumeriPosizionePari(int[] numeri)
{
    int[] posPari;
    //calcola dimensione del vettore risultato
    if (numeri.Length % 2 == 0)
        posPari = new int[numeri.Length / 2];
    else
        posPari = new int[numeri.Length / 2 + 1];

    int n = 0;
    for (int i = 0; i < numeri.Length; i = i+2)
    {
        posPari[n] = numeri[i];
        n++;
    }
    return posPari;
}
```

## 5.14 I parametri di un metodo possono essere modificati?

**SI.** I parametri di un metodo sono equivalenti a variabili locali; dunque, possono essere utilizzati come qualsiasi variabile.

È comunque ritenuta una pratica sconsigliata, poiché si userebbe il parametro con un intento diverso da quello per il quale è definito: fornire l'input al metodo.

## 5.15 Modificando i parametri si modificano gli argomenti corrispondenti?

**DIPENDE** dal tipo dei parametri. Da una parte ci sono gli *array*, dall'altra ci sono i rimanenti tipi.<sup>4</sup>

Poiché nel passaggio "argomenti → parametri" valgono le regole dell'assegnazione, se il parametro è tra i tipi primitivi – `byte`, `int`, `double`, `bool`, `char`, `string` – le modifiche non si riflettono sugli argomenti. Se il parametro è un *array*, allora sì.

Per dimostrarlo prendiamo in considerazione i tipi `int` e `int[]`. Nel codice seguente applico due operazioni simili a due variabili intere e a due variabili *array*:

```
int a = 10;
int b = a;           //a: 10           b:10
a = 20;              //a: 20           b:10
Console.WriteLine("{0}, {1}", a, b); //->20, 10

int[] vetA = { 1, 2, 3 };
int[] vetB = vetA;   //vetA e vetB: 1, 2, 3
vetA[0] = 100;       //vetA e vetB: 100, 2, 3
Console.WriteLine("{0}, {1}", vetA[0], vetB[0]); //->100, 100
```

Appare evidente che interi e *array* (di qualunque tipo) si comportano in modo diverso.

Gli interi non richiedono particolari spiegazioni. Dopo l'assegnazione, due variabili intere hanno lo stesso valore, ma restano comunque oggetti distinti; la modifica di uno non influenza l'altro, e viceversa.

Con gli *array* funziona diversamente. L'assegnazione:

```
int[] vet2 = vet1;
```

Non produce un nuovo vettore, `vetB`, contenente una copia degli elementi di `vetA`; fa sì che le due variabili "referenzino" lo stesso vettore. In pratica, `vetB` diventa un alias di `vetA`, e viceversa.

Tutto ciò ha profonde implicazioni, che riguardano anche i metodi. Nel seguente esempio, il metodo `AzzeraNegativi()` azzerà gli elementi negativi di un vettore di interi:

```
static void Main(string[] args)
{
    int[] numeri = { 1, -3, 5, -6 };
    AzzeraNegativi(numeri);    //-> "numeri" e "valori" referenziano lo stesso vettore
    for (int i = 0; i < numeri.Length; i++)
    {
        Console.Write("{0} ", numeri[i]);
    }    //-> 1, 0, 5, 0
}

static void AzzeraNegativi(int[] valori)    //-> "valori" diventa un alias dell'argomento
{
    for (int i = 0; i < valori.Length; i++)
```

<sup>4</sup> La questione è più complessa. Insieme agli *array* ci sono altri tipi di dati che, da questo punto di vista, si comportano nello stesso modo.

```
{  
    if (valori[i] < 0)  
        valori[i] = 0;  
}  
}
```

Il codice dimostra che modificando gli elementi del parametro `valori`, si modificano gli elementi del vettore `numeri`.

(Ciò dimostra che passare un vettore a un metodo è un'operazione estremamente efficiente, poiché, in realtà, non viene creata alcuna copia del vettore originale.)