Metodi

Modello di programmazione procedurale

Anno 2018/2019

Indice generale

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:

> *La programmazione imperativa è un modello di programmazione secondo il quale un programma è inteso come un insieme di istruzioni, ciascuna delle quali può essere pensata come un'ordine...*

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 procedimento computabile, non importa quanto complesso. Nonostante ciò, si tratta di un modello limitato, il cui utilizzo, anche a fronte di problemi piuttosto 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.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]);
    }
    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 grosso difetto: *è un unico blocco di codice che implementa più funzioni*.

Uno dei principi più importanti 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());
    }
```

Inserimento altezze

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

Visualizzazione altezze

```
    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);
```

Calcolo altezza media

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

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 è un modello di programmazione che consiste nel creare dei blocchi di codice identificati da un nome… Questi sono detti sottoprogrammi…*

L'idea, dunque, 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*, 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".

2.2 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 delle altezze. Per implementarle mediante dei metodi basta "estrarre" i due blocchi di codice da `Main()` e assegnare loro un nome:

```
static void Main(string[] args)
{
    // Esegue i due metodi
    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***.

2.2.1 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[1]:

> *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.

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

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.Write("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[2].

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`.

2.3 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. Pertanto occorre definire globale anche `altezzaMedia`, poiché viene calcolata nel metodo `CalcolaAltezzaMedia()` e utilizzata in `CalcolaNumAlunniSopraMedia()`:

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

2 Anche qui la questione è in realtà più complessa.

```
    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 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 dichiara nel blocco di codice dove è necessaria*

Per vari motivi:

- Aumenta la leggibilità del codice, poiché la dichiarazione e l'uso di una variabile sono collocati nella stessa "porzione" di programma.
- 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);
    }
}
```

2.4 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 delineate chiaramente.

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 sul resto del programma.

2.4.1 Problemi dell'attuale versione

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

Innanzi tutto entrambi implementano una duplice funzione, calcolare e visualizzare il risultato del calcolo. Questo viola il principio che afferma: *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 alunno 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)
    {
        InserisciAltezze();
        VisualizzaAltezze();
        ...
    }
    static void InserisciAltezze()
    {
        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 VisualizzaAltezze()
    {
        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 immagine 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 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: integrazione 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:

- 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 processare 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 in quale variabile memorizzarlo; al suo posto usa l'istruzione `return`, che 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 le 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)
    {
        InserisciAltezze();
        VisualizzaAltezze();

        // 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 le funzioni che calcolano il numero di alunni più alto e più pesante della media può essere parametrizzato mediante un unico 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 usare il metodo:

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