Moduli

Organizzare il codice e incapsulare funzioni mediante i moduli

Anno 2018/2019

Indice generale

1 Introduzione

Il tutorial fornisce un'introduzione all'implementazione dei *moduli*. Partirò con un riepilogo sulla *programmazione procedurale*, considerando nuovamente la funzione dei metodi. Quindi mostrerò come l'uso dei *moduli* estenda questo modello, consentendo di:

- Semplificare l'organizzazione del codice e lo sviluppo del programma.
- Favorire l'incapsulamento e il riutilizzo di determinate funzioni.

1.1 Programmazione "procedurale"

Da wikipedia:

> *La programmazione procedurale è un paradigma di programmazione che consiste nel creare dei blocchi di codice identificati da un nome… Questi sono detti sottoprogrammi, procedure o funzioni, a seconda del linguaggio...*

Un metodo rappresenta dunque un'*unità di codice* che è possibile utilizzare mediante il suo nome. I metodi:

1. Forniscono un "significato" al codice. Un blocco di codice non trasmette alcun significato. Il nome di un metodo (purché appropriato) trasmette l'intento (*ordina*, *ricerca*, *inserisci*, etc) di un procedimento.
2. Sono alla base del principio di *incapsulamento*. Un metodo consente di eseguire un procedimento senza dipendere da esso. 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 ne semplificano il reimpiego.

Nonostante la sua semplicità, questo modello di programmazione consente di creare programmi complessi; infatti, è possibile implementare metodi che chiamano altri metodi, che ne chiamano altri ancora, e così via.

Resta il fatto che un metodo rappresenta un singolo procedimento, ma esistono funzioni che, per essere convenientemente implementate, richiedono unità di codice di livello superiore. Si tratta dunque di poter gestire un insieme di metodi (ed eventualmente variabili) come una singola unità di codice, separata dal resto del programma e dunque dalle altre unità di codice.

Una unità di codice simile si chiama *modulo*.

2 Moduli: definizione e uso

I moduli sono costrutti che consentono di isolare un gruppo di costanti, variabili e metodi – chiamati *membri* del *modulo* - dentro un contenitore.

2.1 Definizione di un *modulo*: sintassi

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

```
static class <nome>
{                                          variabili e costanti (opzionali)
    <modificatore> static <tipo> <variabile>;
    <modificatore> const <tipo> <costanta> = <valore>;
    ...
                                                          metodi (opzionali)
    <modificatore> static <tipo> <nome metodo>(<argomenti>) {...}
    <modificatore> static <tipo> <nome metodo>(<argomenti>) {...}
    ...
}
```

Ricapitolando:

- Il *modulo* è dichiarato attraverso il costrutto `static class`.
- Può definire zero o più variabili (o costanti) e zero o più metodi. Sia le variabili che i metodi devono essere dichiarati `static`.

(Nota bene: eccetto che per l'assenza della parola chiave `static`, anche `Program` rientra nella definizione di *modulo*.)

Come vedremo più avanti, attraverso il *modificatore* di accesso – `public` o `private` – è possibile consentire o vietare l'accesso ai membri del *modulo*.

2.1.1 Nomenclatura

Per convenzione, il nome del *modulo* e i nomi dei metodi dovrebbero rispettare la nomenclatura "CamelCase", e cioè cominciare con la maiuscola.

2.2 Accesso ai membri del modulo

All'interno del *modulo* l'uso di variabili e metodi avviene normalmente; dall'esterno, invece, occorre qualificarli con il nome del modulo.

<nome_modulo>.<nome_membro>

Nel seguente esempio definisco un *modulo*, `Demo`, che definisce una variabile e un metodo.

Program

```
static void Main(string[] args)
{
    string s = Demo.Saluto();
    Console.WriteLine(s); //-> hello!
}
```

DemoModulo

```
static class Demo
{
    private static string testo = "hello!";
```

```
    public static string Saluto()
    {
        return testo;
    }
}
```

Nota bene: in `Main()`, l'uso del metodo `Saluto()` richiede che sia specificato il nome del *modulo*.

2.2.1 "Importare" i membri di un modulo: usare i nomi semplici

Se un *modulo* viene usato in modo intensivo, la sintassi `<modulo>.<metodo>` può diventare verbosa; per semplificare il codice è possibile *importare* i membri, in modo da poterli utilizzare come se facessero parte del *modulo* corrente. Per farlo occorre usare l'istruzione

```
using static <namespace.nome_modulo>
```

Nell'esempio seguente mostro come importare i membri del modulo `Console`:

```
using static System.Console;        //importa i membri del modulo Console
...
static void Main(string[] args)
{
    Console.WriteLine("Hello!");    //usa sintassi qualificata: <modulo>.<metodo>
    WriteLine("Hello!");            //usa sintassi semplice: <metodo>
    Console.ReadKey();              //usa sintassi qualificata: <modulo>.<metodo>
    ReadKey();                      //usa sintassi semplice: <metodo>
}
```

Non si dovrebbe abusare di questa possibilità, riservandola ai *moduli* predefiniti, come `Console` e `Math`. Infatti, facendolo si rende indistinguibile la chiamata a un metodo del *modulo* corrente da quella di un metodo di un altro *modulo*.

2.3 Conclusioni

La definizione e l'uso dei *moduli* è semplice; meno semplice è comprendere il loro ruolo, che è quello di estendere il modello di programmazione procedurale in modo da poter:

- Raggruppare metodi, variabili e costanti che hanno una relazione tra loro.
- Implementare funzioni che richiedono la collaborazione di più metodi.
- Implementare funzioni d'uso generale, favorendo il loro uso in più programmi.
- Incapsulare tali funzioni, rendendo possibile modificare la loro implementazione senza produrre effetti indesiderati sul resto del programma.

Di seguito prenderò in considerazione alcuni scenari e mostrerò come possano essere affrontati mediante l'implementazione di un *modulo*.

3 Moduli d'uso generale

Per comprendere l'utilità dei *moduli* si può cominciare considerando quelli predefiniti, come `Math` e `Console`. Il primo definisce le funzioni matematiche di base; `Console` implementa la gestione dell'input/output di un'applicazione console. Riporto un breve frammento di codice di entrambi:

```
public static class Math
{
    private static double doubleRoundLimit = 1e16d;
    private const int maxRoundingDigits = 15;
    ...
    public static double Round(double value, int digits) {...}
    ...
}
```

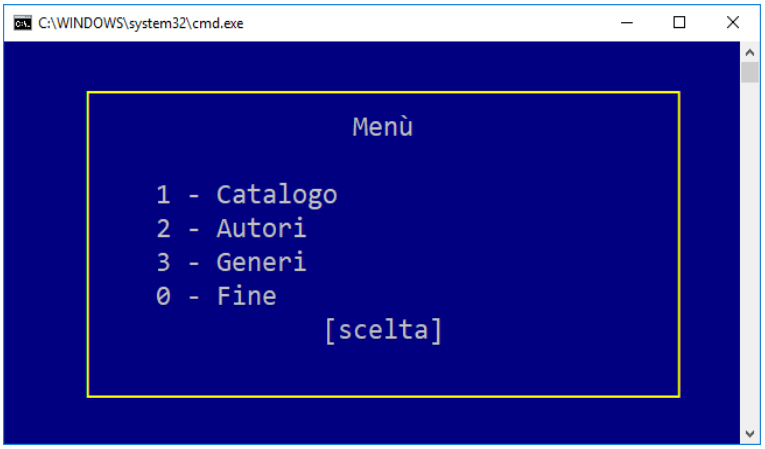
```
public static class Console
{
    private const int DefaultConsoleBufferSize = 256;
    private const short AltVKCode = 0x12;
    ...
    public static void Write(string format, object arg0) {...}
    ...
}
```

I due *moduli* implementano delle funzioni generali, utilizzabili in molti programmi; è uno dei vantaggi derivati dal creare *moduli*: favorire il riutilizzo del codice.

Di seguito fornisco un esempio in tal senso.

3.1 Gestione "avanzata" dello schermo

Immagina di dover produrre un'applicazione console per gestire le funzioni di una biblioteca e di voler realizzare un'interfaccia sofisticata. Lo *screen shot* mostra l'immagine di un ipotetico menù:



Per semplificare lo sviluppo della UI decido di implementare alcuni metodi di supporto, nella fattispecie un metodo per disegnare una cornice e alcuni metodi di visualizzazione. Poi, naturalmente, ci sono i metodi che rispondono ai requisiti del problema: visualizzazione catalogo libri, visualizzazione generi e autori, inserimenti dati, ricerca, etc.

```
class Program
{
    static void Main(string[] args){...}
    static void ShowCatalog() {...}
    static void ShowAuthors() {...}
    static void ShowGenres() {...}
    static void InsertBook() {...}
    ...

    static void Frame(int x1, int y1, int width, int height) {...} // disegna cornice
    static void Write(int x, int y, int width, string text) {...}  // centra testo
    static void Write(int x, int y, string text) {...}
    ...
}
```

Visivamente, ho separato i metodi che implementano i requisiti (gestire una biblioteca) da quelli di supporto alla visualizzazione, perché i due gruppi hanno una distinta funzione. Ebbene, la creazione di un *modulo* consente di formalizzare questa separazione, collocando i metodi di visualizzazione in un'unità di codice separata.

3.1.1 Creazione del modulo Screen

Per creare un *modulo* occorre aggiungere una nuova classe al progetto e prefissarla con la parola `static`. Nell'esempio, i metodi sono stati già creati in `Program`; dunque basta spostarli nel modulo:

```
static class Screen
{
    public static void Frame(int x1, int y1, int width, int height) {...}
    public static void Write(int x, int y, int width, string text) {...}
    public static void Write(int x, int y, string text) {...}
    public static string StringOf(char ch, int length) {...}  // usato in Frame()
    ...
}
```

La suddivisione del programma in due *moduli* (di fatto lo è anche `Program`) ha dei vantaggi:

1. Migliora l'organizzazione del codice, evitando un "affollamento" eccessivo all'interno dello stesso *modulo*. Ciò rende il codice più leggibile e facilmente modificabile; in una parola: gestibile.
2. Evita "collisioni" nei nomi. Ogni *modulo* ha i propri *membri* (costanti, variabili, metodi) e non è possibile usare per errore i membri di un altro *modulo*.
3. Guida lo sviluppo: si aggiunge un metodo (o si dichiara una variabile) a un *modulo* per estendere o modificare la funzione che implementa. È la funzione del modulo che da significato alle variabili e ai metodi che definisce.

L'ultimo punto è particolarmente importante: non si crea un *modulo* semplicemente per mettere ordine nel programma, scegliendo arbitrariamente i metodi da collocarvi; si crea un *modulo* per favorire l'implementazione di una funzione, separandola dalle altre funzioni del programma.

3.2 Accessibilità dei membri del *modulo*: modificatori di accesso

Di default, i membri di un *modulo* sono inaccessibili per gli altri *moduli*, sono cioè privati. Per renderli utilizzabili occorre che siano pubblici, e cioè decorati con il ***modificatore di accesso*** `public`:

```
static class Screen
{
    public static void Frame(int x1, int y1, int width, int height) {...}
    public static void Write(int x, int y, int width, string text) {...}
    public static void Write(int x, int y, string text) {...}
    public static string StringOf(char ch, int length) {...}   // usato in Frame()
    ...
}
```

3.2.1 Rendere i membri inaccessibili: private

Considera il metodo `StringOf()`: non condivide alcunché con la funzione del *modulo*, che è quella di supporto alla visualizzazione. Infatti, viene usato internamente dal metodo `Frame()`. In questo caso è opportuno che il metodo resti privato, e cioè inaccessibile agli altri *moduli*:

```
static class Screen
{
    ...
    private static string StringOf(char ch, int length) {...}
    ...
}
```

(Il *modificatore di accesso* `private` è quello predefinito, dunque può essere omesso.)

3.2.2 Interfaccia pubblica del modulo

L'*interfaccia pubblica* di un *modulo* è l'insieme dei membri pubblici; in pratica, è ciò che il resto del programma "vede" del *modulo*. È l'interfaccia pubblica a definire la funzione del *modulo*.

Ciò che non fa parte della funzione del *modulo*, anche se è utile alla sua implementazione, *deve essere privato*! È il caso del metodo `StringOf()`, utile nell'implementazione degli altri metodi, ma che non riguarda la funzione di visualizzazione.

3.3 Favorire la riusabilità del modulo

Alcuni *moduli* nascono per rispondere alle esigenze di uno specifico programma, ma, data la loro generalità, possono essere utilizzati anche in altri scenari. È il caso del *modulo* `Screen`.

La funzione di `Screen` non ha niente a che vedere con le operazioni di una biblioteca, poiché definisce metodi che facilitano la realizzazione di una UI avanzata. Per questo motivo può essere impiegato in tutti quei programmi per i quali si ritiene utile questa funzione.

In genere, per rendere un *modulo* utilizzabile in più scenari, è opportuno implementarlo in modo che i programmi possano adattarne il funzionamento alle proprie esigenze. Questo significa:

- Aggiungere operazioni non previste inizialmente, ma comunque correlate alla funzione del *modulo*.
- Generalizzare le operazioni esistenti, in modo da renderle flessibili.

Segue un frammento della nuova versione di `Screen`:

```
static class Screen
{
    public const char LeftUp = '┌';
    public const char LeftBottom = '└';
    public const char LeftMedium = '├';

    public const char RightUp = '┐';
    public const char RightBottom = '┘';
    public const char RightMedium = '┤';

    public const char HorLine = '─';
    public const char VerLine = '│';

    public const char Filled = '█';
    public const char Space = ' ';

    public static void Clear(int x1, int y1, int width, int height,
                             ConsoleColor color = ConsoleColor.Black) {...}

    public static void Frame(int x1, int y1, int width, int height,
                             ConsoleColor color = ConsoleColor.Gray)
    {
        ConsoleColor currentColor = Console.ForegroundColor;
        Console.ForegroundColor = color;
        int y2 = y1 + height;
        int x2 = x1 + width-1;
        string line = StringOf(HorLine, width-1);
        Write(x1 + 1, y1, line);
        for (int y = y1+1; y < y2; y++)
        {
            Write(x1, y, VerLine);
            Write(x2, y, VerLine);
        }
        Write(x1 + 1, y2, line);
```

```
        Write(x1, y1, LeftUp);
        Write(x2, y1, RightUp);
        Write(x1, y2, LeftBottom);
        Write(x2, y2, RightBottom);
        Console.ForegroundColor = currentColor;
    }

    public static void HorizontalLine(int x1, int y1, int width) {...}

    public static void VerticalLine(int x1, int y1, int height) {...}

    public static void Write(int x, int y, int width, Alignment alignment, string text)
    {...}


    public static void Write(int x, int y, string text) {...}
    ...
}

enum Alignment
{
    Left,
    Center,
    Right
}
```

Ho evidenziato le parti aggiunte allo scopo di rendere `Screen` più versatile:

- Costanti simboliche che rappresentano i caratteri usati per disegnare una cornice. Queste migliorano l'implementazione di `Frame()` e, all'occorrenza, sono utilizzabili dagli altri *moduli*.
- Metodo `Clear()`, che consente di "ripulire" un'area rettangolare dello schermo.
- Metodi `HorizontalLine()` e `VerticalLine()`, che consentono di disegnare linee orizzontali e verticali.
- In `Clear()` e `Frame()` possibilità di specificare un colore. Si tratta di un parametro con valore di *default*: se non specificato, viene usato un colore predefinito.
- In `Write()` il parametro `alignment`, che consente di stabilire l'allineamento del testo all'interno della lunghezza specificata: sinistra, centrato, destra.

Le modifiche proposte sono soltanto un esempio; quando un *modulo* diventa di utilità generale, è abbastanza comune farlo evolvere aggiungendo nuove funzionalità e potenziando quelle esistenti.

4 Incapsulare una funzione dell'applicazione

`Screen` è un esempio di *modulo* d'utilità generale, il cui impiego va oltre il singolo programma. Infatti potrebbe essere creato al di fuori di uno scenario specifico, anticipando le esigenze di chi deve realizzare applicazioni console con una UI avanzata. È ciò che è stato fatto con i moduli `Console` e `Math`. Il primo viene impiegato in tutte le applicazioni console; il secondo in tutti i programmi che richiedono operazioni matematiche.

In molti scenari, però, la funzione svolta dal *modulo* è strettamente legata a quella del programma in cui viene usato. In questo caso si crea un *modulo* per implementare una funzione specifica all'interno della funzione più generale del programma.

4.1 Campionato *moto gp*: separare elaborazione da interfaccia utente

Considera il seguente problema, ripreso dal tutorial **Record**:

> Realizzare un programma che gestisca i piloti iscritti al campionato di MotoGP. Il programma deve prevedere:
>
> 1. Caricamento dei piloti: nominativo, scuderia, punteggio.
> 2. Visualizzazione dei piloti in ordine di classifica.
> 3. Data una scuderia, visualizzare i piloti che vi appartengono.
> 4. Dato il nome di un pilota, restituire le informazioni memorizzate su di esso.

L'analisi dei dati porta a rappresentare il concetto di pilota mediante il *record* `Pilota`:

```
class Pilota
{
    public string Nominativo;
    public string Scuderia;
    public double Punteggio;
}
```

Stabilito come rappresentare i dati, occorre implementare le funzioni del programma. Un approccio molto comune è quello di tenere separate le funzioni che manipolano i dati da quelle di visualizzazione e gestione dell'input dell'utente. Per formalizzare questa separazione, si può implementare le prime in un nuovo *modulo*, lasciando le seconde in `Program`.

4.2 Modulo MotoGP

Il *modulo* `MotoGP` ha la funzione di gestire i piloti e le operazioni su di essi. Il *modulo* non dipende in alcun modo dall'interfaccia utente e dunque non utilizza i metodi del *modulo* `Console`.

Segue l'interfaccia pubblica:

```
static class MotoGP
{
    ...   // membri privati del modulo
    public static Pilota[] ElencoPiloti() {...}
    public static void AggiungiPilota(Pilota p) {...}
    public static Pilota[] PilotiScuderia(string scuderia) {...}
    public static void OrdinaPerClassifica() {...}
    public static Pilota PilotaPerNominativo(string nominativo) {...}
}
```

Nota bene: non ho specificato la lista che memorizza l'elenco dei piloti, perché questa non fa parte dell'interfaccia pubblica. È un aspetto molto importante: il resto del programma non ha alcun bisogno di conoscere e manipolare la struttura dati contenente i piloti, saranno i metodi del *modulo* a farlo.

La decisione di separare la funzione del *modulo* (gestire le operazioni sui piloti) dal modo in cui è implementata rispetta uno dei principi fondamentali della programmazione: il principio di ***incapsulamento***.

4.2.1 Implementazione del modulo

Di seguito mostro una parte dell'implementazione:

```
static class MotoGP
{
    private static List<Pilota> piloti = new List<Pilota>();

    public static Pilota[] ElencoPiloti()
    {
        return piloti.ToArray();
    }

    public static void AggiungiPilota(Pilota p)
    {
        piloti.Add(p);
    }

    public static Pilota[] PilotiScuderia(string scuderia)
    {
        var ris = new List<Pilota>();
        foreach (var p in piloti)
        {
            if (p.Scuderia == scuderia)
                ris.Add(p);
        }
        return ris.ToArray();
    }
    ...
}
```

Nota bene: la lista `piloti` è privata e dunque inaccessibile al codice esterno. Ciò ha due vantaggi:

1. Evita che possa essere modificata impropriamente. Spetta al *modulo* gestire i piloti, e dunque soltanto ai metodi del *modulo* modificare la lista.
2. Consente, in futuro, di utilizzare un'altra struttura dati per memorizzare i piloti.

Entrambi i punti riguardano il principio di *incapsulamento*. La suddivisione del programma in due funzioni – gestione piloti e gestione UI – non deve essere soltanto formale, ma sostanziale: il modulo `Program` *non* deve gestire i piloti, il modulo `MotoGP` *non* deve gestire la UI.

4.3 Uso del modulo MotoGP

Segue un'implementazione minimale di `Program`, contenente il codice necessario a verificare il corretto funzionamento del *modulo* `MotoGP`.

```
static void Main(string[] args)
{
    GeneraPiloti();  //sostituisce l'input dell'utente
    MotoGP.AggiungiPilota(new Pilota
    {
        Nominativo = "Beep, Beep",
        Scuderia = "A.C.M.E.",
        Punteggio = 400
    });

    MotoGP.OrdinaPerClassifica();
    Visualizza(MotoGP.ElencoPiloti());

    Pilota[] pilotiFerrari = MotoGP.PilotiScuderia("Honda");
    Visualizza(pilotiFerrari);

    Pilota p = MotoGP.PilotaPerNominativo("Rossi, Valentino");
    if (p != null)
    {
        Console.WriteLine("\n{0}", p.Nominativo);
        Console.WriteLine("Scuderia: {0}\nPunti    : {1}", p.Scuderia, p.Punteggio);
    }

    Console.ReadKey();
}

static void Visualizza(Pilota[] piloti)
{
    Console.WriteLine("\n{0,-25}{1,-10}{2,5}", "Nominativo", "Scuderia", "Punti");
    foreach (var p in piloti)
    {
        Console.WriteLine("{0,-25}{1,-10}{2,5}", p.Nominativo, p.Scuderia, p.Punteggio);

    }
}
static void GeneraPiloti() {...}  //sostituisce l'input dei dati
```

5 Moduli di “utilità”

Seguono alcuni esempi di *moduli* che implementano cosiddette funzioni di “utilità”, favorendo la scrittura di codice semplice e facilmente manutenibile.

5.1 Insiemi di costanti simboliche

Nel codice non si dovrebbero mai usare le costanti letterali – numeri, stringhe, caratteri – perché ne riducono la leggibilità e ne complicano la manutenibilità; al loro posto si dovrebbero usare le costanti simboliche. Quando più costanti simboliche fanno riferimento allo stesso soggetto, può essere utile collocarle in un *modulo*.

Ad esempio, il *modulo* `FrameChars` (ripreso da 3.3) definisce un insieme di costanti carattere utili a visualizzare i bordi di una cornice:

```
static class FrameChars
{
    public const char LeftUp = '┌';
    public const char LeftBottom = '└';
    public const char LeftMedium = '├';

    public const char RightUp = '┐';
    public const char RightBottom = '┘';
    public const char RightMedium = '┤';

    public const char HorLine = '─';
    public const char VerLine = '│';
}
```

La loro collocazione in un *modulo* semplifica la visualizzazione di cornici e linee in qualsiasi programma richieda questa funzione.

5.1.1 Raggruppare costanti simboliche di uno specifico programma

`FrameChars` è un *modulo* di utilità generale, riutilizzabile in molti programmi; ma può essere utile creare un *modulo* per raggruppare le costanti simboliche utilizzate in uno specifico programma.

Supponi che un programma debba eseguire dei comandi inseriti dall’utente, tra l’insieme di quelli elencati: `"COPY"`, `"DELETE"`, `"MOVE"`, `"RENAME"`. Nel codice, invece di usare direttamente le costanti letterali, conviene definire delle costanti simboliche e collocarle in un *modulo*:

```
static class UserCommand
{
    public const string Copy = "COPY";
    public const string Delete = "DELETE";
    public const string Move = "MOVE";
    public const string Rename = "RENAME";
}
```

In questo modo:

- Si evita la possibilità di introdurre bug nel codice: riutilizzare la stessa costante letterale, scrivendola in modo diverso.
- Si fruisce dell'*intellisense*.
- Si migliora la leggibilità, rendendo esplicito che le costanti fanno riferimento allo stesso soggetto: un comando dell'utente.
- Si favorisce la manutenibilità: in seguito è possibile modificare le costanti, ad esempio utilizzando lettere minuscole, senza dover cambiare il codice che le usa.

Tutto ciò non significa che ogni qual volta un programma usa delle costanti simboliche, si debba collocarle in un *modulo*. È utile farlo quando queste fanno riferimento a una specifica funzione.

5.2 Conversione di *enum* da / verso stringa

Quando un programma deve elaborare dei valori che ricadono all'interno di un insieme stabilito, si definisce normalmente un *enum* che rappresenti i possibili valori. In molti casi, comunque, ad ogni valore deve corrispondere una stringa, inserita dall'utente e/o visualizzata in output. Occorre dunque scrivere del codice che converte da *enum* a stringa e viceversa, ed eventualmente del codice che verifichi che a una determinata stringa corrisponda un valore dell'*enum*. Gli *enum* forniscono entrambe le funzioni, ma non sono particolarmente adattabili. In questo caso può essere vantaggioso incapsulare l'intera gestione dell'*enum* in un *modulo*.

Supponi, in un programma che gestisce il registro scolastico, di dover rappresentare il sesso degli studenti. Il *modulo* seguente implementa le operazioni:

- Conversione da stringa: `Parse()`.
- Conversione in stringa: `ToString()`.
- Verifica che una stringa rappresenti un valore *enum* valido: `IsValid()`.

```
public enum Genere
{
    Maschile,
    Femminile
}

public static class GenereHelper
{
    const string Maschile = "Uomo";
    const string Femminile = "Donna";

    public static Genere Parse(string genere)  //Genere <- stringa
    {
        switch (genere)
        {
            case Maschile: return Genere.Maschile;
            case Femminile: return Genere.Femminile;
            default: throw new FormatException("Valore non riconosciuto");
```

```
        }
    }

    public static string ToString(Genere genere) //stringa <- Sesso
    {
        return genere == Genere.Maschile ? Maschile : Femminile;
    }

    public static bool IsValid(string valore)
    {
        return valore == Maschile || valore == Femminile;
    }
}
```

Nota bene: ho volutamente scelto una rappresentazione stringa diversa dal corrispondente valore *enum*: `"Uomo"` → `Maschile` ; `"Donna"` → `Femminile`. Questo per dimostrare la flessibilità di questo approccio (usando `Enum.Parse()` non sarebbe stato possibile.)

Segue un frammento di codice che mostra l'uso del *modulo*:

```
static void Main(string[] args)
{
    string genereStr = "Uomo"; //sostituisce l'input
    if (!GenereHelper.IsValid(genereStr))
    {
        Console.WriteLine("Valore sconosciuto!");
        return;
    }
    Genere genere = GenereHelper.Parse(genereStr);
    //...
    Console.WriteLine(GenereHelper.ToString(genere)); // -> "Uomo"
}
```

6 Frequently Asked Questions

Di seguito riepilogo le regole sulla definizione e l'uso dei *moduli*, facendolo sotto forma di F.A.Q, e cioè di domande comuni alle quali fornisco una risposta e una spiegazione.

6.1 I *moduli* possono definire soltanto dei metodi?

NO. Un modulo può definire metodi, variabili e costanti simboliche[1].

6.2 Metodi e variabili di un *modulo* devono essere statici?

SI. In caso contrario il compilatore segnalerà un errore. (Vedi punto successivo.)

6.3 Nella definizione, il nome del modulo deve essere decorato con static?

In teoria **NO**, in pratica **SI**.

`Program`, che è un *modulo* a tutti gli effetti, dimostra che la parola `static` non è obbligatoria; *ma è assolutamente consigliabile usarla!* Infatti, in sua assenza, il linguaggio non impone l'uso di `static` nella definizione di metodi e variabili, cosa che è necessaria alla corretta definizione dei membri del *modulo*.

6.4 Un *modulo* può essere composto soltanto da variabili?

SI. Si tratta comunque di uno scenario poco comune. (Vedi punto successivo.)

6.5 Un *modulo* può essere composto soltanto da costanti simboliche?

SI. Può essere utile per definire un'insieme di opzioni o valori predefiniti. Si tratta di una funzione tipica degli *enum*, ma a volte è necessaria la flessibilità offerta dai *moduli*. (Vedi 5.1)

6.6 E' giusto implementare mediante un *modulo* una funzione di visualizzazione?

SI. Non ci sono vincoli sul codice di un *modulo*; l'importante è che implementi una specifica funzione. In 3.1 porto l'esempio di un *modulo* completamente dedicato alla visualizzazione.

6.7 Quanti *moduli* è possibile definire in un programma?

Quanti si vuole. Più è complessa la funzione generale del programma, più è utile e comune suddividerla in sotto funzioni, ognuna implementata da un *modulo*.

(È lo stesso principio adottato per i metodi, soltanto ad un livello superiore.)

1 In realtà un modulo può definire anche *proprietà* ed *eventi*, i quali sono costrutti del linguaggio che non abbiamo ancora affrontato.