Strong enum types

Anno 2017/2018

Indice generale

1 Introduzione

In questo tutorial considero alcuni scenari nei quali è necessario rappresentare dati che hanno un numero predefinito di valori. Alcuni esempi: il genere (maschile/femminile), i giorni della settimana, lo stato civile, un codice, uno stato di errore. Per questi tipi di dati si usa il termine *strong enum types*. (D'ora in avanti *enum types*.)

Prima di introdurre degli esempi, mostro alcune soluzioni normalmente adottate in questi scenari, evidenziandone i limiti.

1.1 Rappresentare il dato mediante il tipo string

Considera l'ipotesi di gestire un elenco di impiegati, per ognuno dei quali sono specificati il nome e il sesso. Spesso, la scelta più intuitiva per rappresentare il sesso cade sul tipo `string`:

```
public class Impiegato
{
    public string Nominativo { get; set; }
    public string Sesso { get; set; }      //->"Uomo"/"Donna"
}

...
class Program
{
    static void Main(string[] args)
    {
        List<Impiegato> impiegati = new List<Impiegato>();
        //... inserimento dei dati sugli impiegati
        var uomini = ElencoUomini(impiegati);
    }

    static List<Impiegato> ElencoUomini(List<Impiegato> impiegati)
    {
        var ris = new List<Impiegato>();
        foreach (var imp in impiegati)
        {
            if (imp.Sesso == "Uomo")
                ris.Add(imp);
        }
        return ris;
    }
}
```

Questa soluzione è semplice da implementare, ma presenta delle pesanti controindicazioni: usando delle constanti letterali (`"Uomo"` e `"Donna"`) si lega il codice al valore letterale dei dati e si ostacolano eventuali operazioni di *refactoring*. Ad esempio, cosa accadrebbe se si decidesse di accettare i valori: `"M"`/`"F"`, `"maschio"`/`"femmina"`, etc? Sarebbe necessario correggere manualmente il codice ovunque questi valori fossero utilizzati.

Inoltre, non possiamo avvalerci dell'*intellisense*; è facile commettere errori di scrittura, sui quali il compilatore (e l'analizzatore sintattico dell'IDE) non può aiutarci.

Ad esempio:

```
if (imp.Sesso == "Umo")
   ris.Add(imp);
```

1.2 Utilizzare delle costanti simboliche stringa

L'uso di costanti simboliche risolve il problema della presenza di costanti letterali nel codice.

```
public static class Sesso
{
    public const string Maschile = "Uomo";
    public const string Femminile = "Donna";
}
```

```
...
  if (imp.Sesso == Sesso.Maschile)
     ris.Add(imp);
...
```

Questa soluzione:

- Disaccoppia i nomi dai valori che rappresentano.
- Centralizza la definizione dei valori, consentendo dunque di utilizzare i nomi ovunque nel programma.
- Consente di fruire dell'*intellisense*.
- Permette il *refactoring*, per la modifica dei nomi delle costanti o delle costanti stesse.

1.2.1 Validare i valori

Il tipo `string` non valida di per sé i valori memorizzati. Si può ovviare a questa mancanza aggiungendo la logica necessaria al modulo `Sesso`:

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

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

È responsabilità del codice applicativo utilizzare il metodo `Valido()` per verificare la correttezza dei valori e usare le constanti `Maschile` e `Femminile` per esprimerli. Resta il fatto che non esiste alcun vincolo al riguardo, poiché il tipo `string` può rappresentare qualsiasi stringa.

Ciò riconduce al "peccato originale" di questa scelta: il sesso di un impiegato *non è una stringa*, anche se può essere rappresentato come tale.

1.3 Usare gli enum

Quando si tratta di memorizzare un dato che cade in un insieme di valori predefiniti, l'uso di un *enum* rappresenta la scelta più naturale, poiché introduce un nuovo tipo che identifica chiaramente i valori ammissibili.

```
public enum Sesso
{
    Maschile,
    Femminile
}
...

public class Impiegato
{
    public string Nominativo { get; set; }
    public Sesso Sesso { get; set; }
}
...

static List<Impiegato> ElencoUomini(List<Impiegato> impiegati)
{
    var ris = new List<Impiegato>();
    foreach (var imp in impiegati)
    {
        if (imp.Sesso == Sesso.Maschile)
            ris.Add(imp);
    }
    return ris;
}
```

Naturalmente, se occorre gestire l'input e/o l'output dei dati è necessario scrivere del codice che converta verso `string`. Non è buona norma lasciare questo compito al codice applicativo, anche perché non è detto che la rappresentazione stringa debba coincidere con il nome della costante *enum* corrispondente:

```
public static class SessoHelper
{
    const string Maschile = "Uomo";
    const string Femminile = "Donna";
    public static Sesso Parse(string sesso)
    {
        switch (sesso)
        {
            case Maschile: return Sesso.Maschile;
            case Femminile: return Sesso.Femminile;
            default: throw new FormatException("Formato non valido");

        }
    }
```

```
    public static string ToString(Sesso sesso)
    {
        return sesso == Sesso.Maschile ? Maschile : Femminile;
    }

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

(Nota bene: ho aggiunto anche un metodo di validazione.)

1.3.1 Problemi legati all'uso degli enum

L'uso di *enum* è preferibile rispetto alle stringhe, poiché introduce un nome che identifica un determinato concetto, il sesso in questo caso; ma si tratta di un tipo che non pone vincoli stringenti:

- Gli *enum* sono semplicemente dei "wrapper" del tipo `int` (in generale dei tipi integrali). Infatti, le costanti *enum* sono dei valori interi.
- Non consentono di implementare alcuna logica, che dunque deve essere definita altrove; ma è responsabilità del codice applicativo utilizzarla, poiché il tipo *enum* in sé non può imporre vincoli in tal senso.

In sostanza, non essendo realmente dei tipi di dati, gli *enum* non consentono di implementare i due aspetti fondamentali di un tipo: garantire che gli insiemi di valori ammissibili e le operazioni eseguibili su di essi siano sempre coerenti con il tipo implementato.

Di seguito introduco gli *strong enum types*: hanno le caratteristiche degli *enum* e allo stesso tempo sfruttano le possibilità offerte dalla OOP.

2 Esempio n° 1: Gender

Un principio della programmazione *object oriented* stabilisce: per ogni concetto appartenente al dominio del problema dovrebbe essere definito un tipo all'interno del programma. Quest'ultimo, mediante l'applicazione di *incapsulamento* e *invariante di tipo*, deve garantire che i valori e le operazioni eseguite su di essi siano sempre coerenti con il concetto rappresentato.

Il tipo `Gender` ha lo scopo di incapsulare il concetto del genere, implementando le funzioni di:

- Definizione dei due valori ammissibili.
- Validazione dei valori: un `Gender` può avere soltanto uno tra i due valori ammissibili.
- Creare un valore a partire da una stringa
- Fornire la conversione in stringa.
- Confrontare due valori per uguaglianza.

Segue un'implementazione:

```
public struct Gender:IEquatable<Gender>
{
    public static readonly Gender Male = new Gender(MALE);  //valore di default
    public static readonly Gender Female  = new Gender(FEMALE);

    const int MALE = 0;
    const int FEMALE = 1;

    private readonly int value; // default -> MALE
    public static Gender Parse(string text)
    {
        // se necessario si possono implementare più formati
        text = text.ToUpper();
        if (text == "M")
            return Male;
        if (text == "F")
            return Female;

        throw new FormatException("Il valore specificato...");
    }

    private Gender(int value)
    {
        this.value = value;
    }

    public static bool operator == (Gender left, Gender right)
    {
        return left.value == right.value;
    }

    public static bool operator != (Gender left, Gender right)
    {
```

```
        return left.value != right.value;
    }

    public override string ToString()
    {
        return value == MALE ? "M": "F";
    }
    public override int GetHashCode()
    {
        return value;
    }

    public bool Equals(Gender g)
    {
        return value == g.value;
    }
}
```

Vi sono alcune considerazioni da fare:

- Il tipo è una struttura ed è immutabile, caratteristica comune tra i tipi che implementano dei valori.
- I valori ammissibili, `Male` e `Female`, sono in pratica delle costanti (essendo campi *readonly*).
- Non definisce un costruttore pubblico; si può valorizzare una variabile `Gender` assegnando una delle due costanti.
- Definisce un metodo `Parse()`, che allinea `Gender` ai tipi predefiniti. (Sarebbe opportuno implementare anche `TryParse()`, o comunque un metodo di validazione che non sollevi eccezioni.)
- Definisce il metodo `ToString()` per la conversione verso stringa.
- `Gender` stabilisce il proprio valore predefinito (`Male`); è il valore assegnato alle variabili globali non inizializzate. (Vedi paragrafo **Struttura "vs" classe**)
- Implementa l'interfaccia `Iequatable<>`.

L'uso di questo tipo garantisce che una variabile non potrà mai avere un valore diverso da `Male` o `Female`.

2.1 Uso del tipo Gender

Dal punto di vista del codice applicativo, `Gender` si usa come un *enum*, con la differenza che incapsula internamente la validazione dei valori utilizzati e fornisce la loro rappresentazione in stringa:

```
public class Impiegato
{
    public string Nominativo { get; set; }
    public Gender Sesso { get; set; }
}
```

```
...

static List<Impiegato> ElencoUomini(List<Impiegato> impiegati)
{
    var ris = new List<Impiegato>();
    foreach (var imp in impiegati)
    {
        if (imp.Sesso == Gender.Male)
            ris.Add(imp);
    }
    return ris;
}
```

C'è però una limitazione: le variabili `Gender` (come qualunque altra classe o struttura) non possono essere usate negli `switch`. La versione 7 di C# potenzia le funzionalità del costrutto `switch`, ma richiede comunque una sintassi piuttosto complicata. In sostanza, non possiamo scrivere come faremmo se `Gender` fosse un *enum*:

```
Gender gender = Gender.Parse("M");
...

switch (gender)
{
    case Gender.Male: Console.WriteLine("Maschio");break;
    case Gender.Female: Console.WriteLine("Maschio"); break;
}
```

Ma dobbiamo utilizzare la nuova funzionalità di *pattern matching* con la clausola `when`:

```
Gender gender = Gender.Parse("M");
...

switch (gender)
{
    case Gender g when g == Gender.Male: Console.WriteLine("Maschio");break;
    case Gender g when g == Gender.Female: Console.WriteLine("Maschio"); break;
}
```

2.2 Struttura "vs" classe

Nell'implementare un *enum type* è leggittimo chiedersi se sia preferibile utilizzare strutture, piuttosto che classi. In generale le strutture, soprattutto in caso di tipi con una piccola occupazione di memoria, sono una soluzione ottimale, poiché producono una minor "pressione" sul *garbage collector*. Ebbene, nel caso di *enum types* immutabili, come dimostrerò più avanti, la differenza tra strutture e classi è irrilevante.

D'altra parte, l'uso di strutture solleva un'altra questione: è possibile usare variabili globali non inizializzate, le quali sono impostate al valore predefinito. Occorre garantire che il valore di default rientri tra quelli ammissibili.

In `Gender`, il valore di default è `Male`. (`Female` sarebbe stato lo stesso: basta invertire i valori delle costanti `MALE` e `FEMALE`.)

```
Gender sesso; //-> Gender.Male
```

Si tratta di una scelta legittima, ma discutibile, poiché il valore memorizzato nella variabile non emerge chiaramente dal codice. D'altra parte è lo stesso comportamento dei tipi predefiniti e, soprattutto, degli *enum*.

Ovviamente, se `Gender` fosse una classe l'intera questione perderebbe di importanza, poiché una variabile globale sarebbe inizializzata al valore `null`.

3 Esempio n° 2: RequestType

In uno scenario di comunicazione client-server che usa il formato testo, i messaggi inviati dal client contengono un campo che identifica il tipo di richiesta:

- `LGI → Login`
- `LGO → Logout`
- `ERQ → Echo request`
- `MSG → Message`

Si vuole implementare un *enum type* che incapsuli il tipo della richiesta. Segue una soluzione che implementa il tipo mediante una classe[1]:

```
public class RequestType
{
    //NB: questa dichiarazione deve precedere il resto della classe
    static readonly Dictionary<string, RequestType> requestList =
                                        new Dictionary<string, RequestType>();

    public static readonly RequestType Login = new RequestType("LGI");
    public static readonly RequestType Logout = new RequestType("LGO");
    public static readonly RequestType EchoRequest = new RequestType("ERQ");
    public static readonly RequestType Message = new RequestType("MSG");

    private string text;
    public static RequestType Parse(string text)
    {
        if (requestList.TryGetValue(text, out RequestType rq))
            return rq;

        throw new FormatException("Richiesta non valida");
    }

    private RequestType(string text)
    {
        this.text = text;
        requestList.Add(text, this);
    }

    public override string ToString()
    {
        return text;
    }
}
```

Vi sono molte differenze nell'implementazione rispetto al tipo `Gender`. L'idea sottostante a questa soluzione è la memorizzazione in un dizionario dell'elenco delle costanti del tipo. L'inserimento di un valore nel dizionario viene eseguito nel costruttore, invocato nella definizione delle costanti.

1 Il metodo **Parse()** sfrutta una caratteristica di C# 7 e dunque non è compilabile (ma è facilmente modificabile) con la versione 6 del linguaggio.

Perché questa particolare "configurazione" possa funzionare, è necessario che il dizionario sia creato prima della definizione delle costanti, altrimenti il costruttore proverebbe ad aggiungere il valore a un dizionario inesistente.

Si possono immaginare delle varianti. Ad esempio, nell'attuale versione il campo privato `text` è necessario soltanto per l'implementazione di `ToString()`; altrimenti, la chiave memorizzata nel dizionario è sufficiente. Al posto del dizionario si può usare una lista; in questo caso è obbligatorio definire il campo `text`, altrimenti il metodo `Parse()` non sarebbe in grado di trovare l'oggetto corrispondente.

3.1 Classe "vs" struttura

Mettendo a confronto `RequestType` e `Gender` nascono due questioni:

- Esistono differenze di performance tra le due soluzioni? (Corollario: usare una classe mette "pressione" al *garbage collector*?)
- Perché `RequestType` non definisce la logica necessaria per gestire l'uguaglianza tra due valori?

Le due questioni sono in realtà strettamente correlate.

`RequestType` non penalizza affatto il *garbage collector*, poiché gli unici oggetti creati sono le quattro costanti (oltre al dizionario, naturalmente). Considera il seguente codice:

```
string[] cmd = { "LGI", "LGO", "ERQ", "MSG" };
RequestType[] richieste = new RequestType[1000];

for (int i = 0; i < richieste.Length; i++)
{
    richieste[i] = RequestType.Parse(cmd[i % 4]);
}
```

Apparentemente crea 1000 oggetti, ma non è così; *crea mille riferimenti ai quattro oggetti che rappresentano le costanti del tipo.*

Sulla base di questo si capisce che `RequestType` non ha bisogno di gestire le operazioni di uguaglianza, per il semplice motivo che il confronto tra due oggetti avverrà, *e deve avvenire*, tra due *reference*. Diversamente da quanto accade con `Gender`, se due valori sono uguali significa che sono lo stesso oggetto; si parla infatti di *object identity* e non di *value equality*.

```
RequestType rq1 = RequestType.Login;
RequestType rq2 = RequestType.Parse("LGI");   // -> restituisce un riferimento a Login
bool uguali = rq1 == rq2; // -> true (rq1 e rq2 referenziano lo stesso oggetto!)
```

4 Esempio n° 3: Direction

Nel terzo scenario l'uso di un semplice *enum* appare la scelta perfetta: non è richiesta alcuna conversione verso `string`, né è richiesta una logica di validazione.

In un videogioco 2D occorre gestire la direzione di movimento dei personaggi: sinistra, destra, in alto, in basso. Il cambio di direzione di un personaggio è sottoposto a un preciso vincolo: non può invertire la propria direzione, ma soltanto modificarla di 90°. Inoltre deve essere possibile cambiare la direzione in modo casuale, ma sempre rispettando il vincolo suddetto.

4.1 Uso di un *enum*

In un caso del genere, la prima scelta cade immediatamente su un *enum*:

```
public enum Direction
{
    Left,
    Right,
    Up,
    Down
}
```

Dopodiché si implementano i metodi che modificano le variabili di tipo `Direction`, implementando le funzioni e i vincoli richiesti.

Si tratta di un approccio procedurale, che dunque non sfrutta le possibilità offerte dalla OOP: incapsulare dati e comportamento in un oggetto.

4.2 Tipo Direction

L'obiettivo è quello di implementare un tipo che incapsula: le quattro direzioni, le operazioni di cambio direzione e il vincolo al quale deve sottostare il cambio suddetto. In questo caso decido di implementare un tipo mutabile[2]:

```
public struct Direction
{
    public static readonly Direction Left = new Direction(0);  // default
    public static readonly Direction Right = new Direction(1);
    public static readonly Direction Up = new Direction(2);
    public static readonly Direction Down = new Direction(3);

    private int direction; // default -> Left
    private static Random rd = new Random();

    private Direction(int direction)
    {
        this.direction = direction;
    }
```

2 Ho omesso gli operatori **==** e **!=**, che comunque non sono richiesti dal codice applicativo.

```
    public void TurnRandom()
    {
        if (direction < 2)
            direction = rd.Next(2, 4);
        else
            direction = rd.Next(0, 2);
    }

    public void TurnLeft()
    {
        if (direction != Right.direction)
            direction = Left.direction;
    }

    public void TurnRight()
    {
        if (direction != Left.direction)
            direction = Right.direction;
    }

    public void TurnUp()
    {
        if (direction != Down.direction)
            direction = Up.direction;
    }

    public void TurnDown()
    {
        if (direction != Up.direction)
            direction = Down.direction;
    }
}
```

(Nota bene: i metodi `Turn<direzione>()` modificano l'oggetto, ma soltanto se la nuova direzione non rappresenta un'inversione a 180°.)

Essendo `Direction` una struttura, le variabili globali non inizializzate avranno il valore di default `Left` (che corrisponde al valore zero nel campo privato `direction`).

Segue un frammento di codice che usa il tipo:

```
static Direction heroDirection;  // default -> Direction.Left
...

static void GetCommands()
{
    if (Console.KeyAvailable == false)
        return;

    ConsoleKeyInfo ki = Console.ReadKey(true);
    switch (ki.Key)
    {
        case ConsoleKey.LeftArrow: heroDirection.TurnLeft(); break;
        case ConsoleKey.RightArrow: heroDirection.TurnRight(); break;
```

```
        case ConsoleKey.UpArrow: heroDirection.TurnUp(); break;
        case ConsoleKey.DownArrow: heroDirection.TurnDown(); break;
        case ConsoleKey.Spacebar: heroDirection.TurnRandom();break;
    }
}
```

4.3 Uso di una classe

Nel caso di un *enum type* mutabile, l'uso di una classe non è un approccio funzionante. Infatti, gli oggetti che rappresentano le costanti del tipo (`Left`, `Right`, `Up` o `Down`) sarebbero referenziati dalle variabili usate dal programma e, attraverso di esse, modificati.

Ipotizza che `Direction` sia una classe e considera il seguente codice:

```
var dir1 = Direction.Left; // -> dir1 referenzia Left
dir1.TurnDown();           // -> dir1 diventa Down (ma ciò vale anche per Director.Left)
var dir2 = Direction.Left; // -> Down
```

Dopo la prima istruzione, la variabile `dir1` referenzia l'oggetto `Left`. (il campo `direction` vale 0). La seconda istruzione modifica lo stato di `dir1`, e dunque modifica anche lo stato di `Left`: in sostanza, la costante `Left` si è trasformata nella costante `Down`! (Il campo `direction` vale 3)

4.3.1 Implementazione immutabile mediante una classe

Rendendo `Direction` immutabile è possibile utilizzare una classe, tra l'altro producendo un codice estremamente compatto.

```
public class Direction
{
    public static readonly Direction Left = new Direction();
    public static readonly Direction Right = new Direction();
    public static readonly Direction Up = new Direction();
    public static readonly Direction Down = new Direction();

    static Random rnd = new Random();
    public Direction TurnRandom()
    {
        int n = rnd.Next(2);
        if (this == Left || this == Right)
            return (n == 0) ? Up : Down;

        return (n == 0) ? Left : Right;
    }

    public Direction TurnLeft()
    {
        return (this != Right)? Left: this;
    }

    public Direction TurnRight()
    {
```

```
        return (this != Left) ? Right: this;
    }

    public Direction TurnUp()
    {
        return (this != Down) ? Up : this;
    }

    public Direction TurnDown()
    {
        return (this != Up) ? Down : this;
    }
}
```

Il codice offre alcuni spunti interessanti. Innanzitutto, la classe non definisce alcun campo privato! Non ne ha bisogno, poiché tutto ciò che distingue una variabile da un'altra è il suo *reference* e non lo stato dell'oggetto che referenzia (tale stato non esiste.)

I metodi `Turn<direzione>()` non modificano lo stato dell'oggetto (non c'è!), ma restituiscono un *reference* alla nuova direzione, oppure l'attuale *reference* se questa non può essere modificata.

Il confronto tra due variabili `Direction` non richiede l'*override* di `==` e `!=`, poiché il confronto per uguaglianza viene eseguito sui *reference* e non sugli oggetti.

Segue un frammento di codice che usa la classe:

```
static Direction heroDirection = Direction.Left;
...

static void GetCommands()
{
    if (Console.KeyAvailable == false)
        return;

    ConsoleKeyInfo ki = Console.ReadKey(true);
    switch (ki.Key)
    {
        case ConsoleKey.LeftArrow: heroDirection = heroDirection.TurnLeft(); break;
        case ConsoleKey.RightArrow: heroDirection = heroDirection.TurnRight(); break;
        case ConsoleKey.UpArrow: heroDirection = heroDirection.TurnUp(); break;
        case ConsoleKey.DownArrow: heroDirection = heroDirection.TurnDown(); break;
        case ConsoleKey.Spacebar: heroDirection = heroDirection.TurnRandom();break;
    }
}
```