Appunti di Informatica: C#

Ares Rebatto

February 29, 2024

Contents

1 Programmazione ad oggetti

1.1 Metodi d'estensione

I metodi d'estensione sono utili nel momento in cui vogliamo "aggiungere" dei metodi a una classe già esistente, che sia tra quelle predefinite dal linguaggio o una classe creata da noi, senza dover fare una classe derivata dalla prima. Vediamo un esempio utilizzando le classi int e string già presenti nel linguaggio:

```
int num = 4;
int scndNum = 3;

string up = "HELLO WORLD";
string  low = "hello world";

Console.WriteLine(num.IsEven()); //true
Console.WriteLine();
Console.WriteLine(scndNum.IsEven()); //false
Console.WriteLine();
Console.WriteLine(up.IsUpperCase()); //true
Console.WriteLine();
Console.WriteLine(low.IsUpperCase()); //false
```

```
static class MetodiDestensione{
 public static bool IsUpperCase(this string s) => s == s.ToUpper();

 public static bool IsEven(this int n) => n%2==0;
}
```

Osserviamo la classe MedotiDestensione. Quando si voglio creare dei metodi d'estensione, bisogna farlo dentro una classe statica apposita, dentro alla quale poi ci possiamo scrivere tranquillamente tutti i metodi d'estensione per tutte le classi che vogliamo e questi dovranno essere, chiaramente, a loro volta statici e come parametro dovranno accettare almeno un'istanza della classe di cui vogliamo estendere il metodo, ma il parametro va richiesto con la keyword this davanti al tipo in modo da evidenziare che si parla dell'istanza corrente(E sostanzialmente ci permette di avere la sintassi, come la potete vedere sopra, num.IsEven() invece di MetodiDestensione.IsEven(num)).

Nel nostro caso abbiamo fatto due metodi d'estensione: uno per le stringhe che verifica se la stringa data in input è scritta in CapsLock o meno, mentre l'altro metodo è per gli interi e verifica se il parametro dato è pari.

Sopra alla classe andiamo semplicemente a testare il funzionamento dei metodi d'estensione che abbiamo scritto

1.2 Classi astratte

Le classi astratte sono un particolare tipo di classe che non può essere istanziata e rientra in quel capitolo del C# che fa riferimento alla riutilizzabilità del codice. Si parla di classi astratte perché si alza il livello di astrazione rispetto alle classi semplici o le classi madri e queste vengono utilizzate quando ho la necessità di modellare degli oggetti simili tra loro, ma non uguali.
In caso si decidesse di usare una classe astratta, bisognerebbe anche capire quali caratteristiche accomunano gli oggetti che vogliamo modellare e quali invece li caratterizzano singolarmente, in modo da capire come strutturare il codice.

Si, ma come mai se abbiamo detto che non possiamo creare delle istanze di questa classe, allora parliamo di modellazione di oggetti? Il fatto è che non si possono creare istanze direttamente dalla classe astratta, ma questo non significa che non possiamo generarle affatto: per utilizzare le classi astratte, infatti, abbiamo bisogno di creare delle classi figlie da cui poi istanzieremo i nostri oggetti. Anche per questo si parlava di caratteristiche comuni e individuali: quelle comuni andranno inserite all'interno della classe astratta, mentre quelle individuali si inseriranno dentro quelle derivate.
Ma vediamo come si scrive una classe astratta prendendo un esempio direttamente dal sito della Microsoft:

```
Square quadrato = new Square(4);

quadrato.CalcolaPerimetro(quadrato, 5);
Console.WriteLine(quadrato.Perimetro);



public abstract class Shape
{
 public virtual int Perimetro{get; set;}
 public abstract int Side{get;set;}

 public abstract int GetArea();

 public virtual void CalcolaPerimetro(Shape shape, int len){
  Perimetro = shape.Side *len;
 }

}

public class Square : Shape
{
 public override int Side{get;set;}

 public Square(int n) => Side = n;
```

```
 public override int GetArea() => Side * Side;

}
```

Osserviamo immediatamente una cosa: la differenza tra la keyword abstract e virtual per i membri della classe. Asbtract indica che il membro che stiamo marchiando come tale deve necessariamente essere implementato nella classe figlia, altrimenti ci dà un errore evidenziandoci la mancata implementazione. E' importante evidenziare come, in effetti, nella classe astratta, un metodo abstract non viene per niente implementato: di fatto il metodo GetArea ha solo la firma e manca di un qual si voglia corpo.
La keyword virtual indica invece come sia possibile fare l'override nella classe derivata, ma non sia obbligatorio farlo: di fatto la properties Perimetro è stata marchiata con virtual, ma non viene sovrascritta nel metodo Square.

Per riassumere, quindi, una classe astratta non può essere istanziata direttamente, ma deve essere ereditata, può avere sia membri astratti, cui implementazione deve avvenire forzatamente nella classe ereditata, sia membri virtuali, che sono opzionalmente sovrascrivibili nella classe figlia.

E' anche molto importante dire che una classe astratta non può essere modificata col modificatore Sealed, poiché l'idea di una classe astratta è proprio quella di essere ereditata, mettendosi agli antipodi di quello che si propone di fare Sealed.

1.3 Interfacce

In C# non esiste la possibilità di avere l'ereditarietà multipla come avviene in altri linguaggi: per ovviare a questo problema la Microsoft ha introdotto, già nella prima versione del linguaggio, **le Interfacce**.
Queste vengono definite non come classi, ma come **contratti**: questo è dovuto al fatto che quello che scriviamo al loro interno è solo la firma dei metodi e non la loro implementazione, che andrà poi fatta all'interno di una classe figlia.

```
public interface IAccelerable{
    void Accelera(int valore);
    void Accelera();
}
```

In questo esempio dichiariamo un'interfaccia, le quali vanno sempre dichiarate con una I all'inizio, che contiene la firma di due metodi che si chiamano entrambi Accelera, ma di cui uno richiede un parametro intero in input.

```
public class Veicolo : IAccelerable, IComparable<Veicolo>
{
```

```
 public string Modello { get; set; }
 public int Velocita { get; set; }
 public Propulsore Propulsione { get; set; }


 public Veicolo()
 {
  Velocita = 0;
  Propulsione = Propulsore.Nessuno;
 }
 public Veicolo(string m)
 {
  Modello = m;
 }

 public virtual void Accelera()
 {
  Velocita++;
  //throw new NotImplementedException();
 }




 public virtual void Accelera(int valore)
 {
  throw new NotImplementedException();
 }

 public int CompareTo(Veicolo altroVeicolo)
 {
  return this.Modello.CompareTo(altroVeicolo.Modello);
 }

 public override string ToString()
 {
  return $"Questa {Modello} ha una velocità di: {Velocita}";
 }
}

public enum Propulsore
{
 Nessuno,        //0
 Benzina,
 Gasolio,
 Gpl,
```

```
 Metano,
 Elettrico,
 Muscolare
}
```

In questa classe abbiamo 3 properties di cui una è l'istanza di un'enum. Due costruttori in overload e poi va a fare l'implementazione, mantenendolo comunque come virtuale, del metodo Accellera presente in IAccelerable aumentando la properties velocità.
Avviene anche l'implementazione per il metodo Accelera che richiede un interno in input, ma questa volta lancia un'eccezione per segnalare che non vi è nessuna implementazione per il metodo in questione; il dubbio che potrebbe sorgere a questo punto è che potremmo semplicemente non implementarlo piuttosto che lanciare un'eccezione: in realtà quando facciamo derivare una classe da un'interfaccia, dobbiamo implementare tutti i suoi metodi.
Fa infine l'override del metodo ToString presente già nella classe predefinita Object.
La parte più importante si trova in cima, affianco al nome della classe: vediamo infatti che questa classe è figlia di due interfacce, ovvero la nostra IAccelerable e un'interfaccia predefinita del C# che è IComparable, che ci permette di fare il Sort per una lista di oggetti istanziati da una classe scritta da noi.

```
public class Auto:Veicolo
{
 public Auto()
 {
 }

 public Auto(string m)
 :base (m)
 {
 }
 public override void Accelera()
 {
  if (Propulsione == Propulsore.Elettrico)
  Velocita += 10;
  else
  Velocita += 5;
 }
}

public class Bicicletta:Veicolo
{
 public Bicicletta()
 {
 }
```

```
 public Bicicletta(string m)
 :base (m)
 {
 }
 public override void Accelera()
 {
  Velocita += 1;
 }
}
```

Ecco qui il punto saliente del tutto: in queste due classi vediamo come l'implementazione del metodo Accelera è differente. Andiamo a scrivere il Main e osserviamo le conseguenze di quello che abbiamo scritto:

```
List<Veicolo> veicoli = new List<Veicolo>();

//Costruttore implicito: perché funzioni la classe deve implementare il costruttore di defau
Veicolo ve = new Veicolo { Modello = "Tesla", Propulsione = Propulsore.Elettrico };
Veicolo vf = new Auto { Modello = "Tesla", Propulsione = Propulsore.Elettrico };
Veicolo va = new Auto { Modello = "Ford" };
Veicolo vb = new Bicicletta { Modello = "Bianchi", Propulsione = Propulsore.Muscolare };
```

Si crea una lista che contiene istanze della classe Veicolo e gli si inseriscono oggetti di tipo Veicolo, che però istanziamo col costruttore della classi figlie: questo avviene perché la classe madre può contenere tutte le istanze della classe figlia.
Questo ci permette di avere una lista di oggetti dello stesso tipo, ma che in base al tipo della classe figlia che hanno, accelerano in maniera differente: quelle che abbiamo istanziato col costruttore di Auto accelererà di 10, se ha un propulsore elettrico, ogni volta che richiamiamo il metodo e di 5 se non ha il propulsore elettrico. Quelle che invece abbiamo istanziato col costruttore Bicicletta accelereranno di 1 e così anche quelle istanziate con il costruttore di Veicolo.
Potrebbe sembrare una cosa piuttosto sciocca, ma se ci pensate, quanto tempo ci avreste messo a fare una cosa del genere con la programmazione strutturata? Molto e avreste usato molte più linee di codice, oltre che avere un algoritmo poco ottimizzato.

Andiamo ora per mezzo di un foreach a stampare tutti gli elementi della lista come segue:

```
foreach (var item in veicoli)
{
 item.Accelera();
 Console.WriteLine(item);
}

veicoli.Sort();
```

```
Console.WriteLine();

foreach (var item in veicoli)
{
 Console.WriteLine(item);
}
```

Se osserviamo l'output, vediamo che è il seguente:

```
 Questa Tesla ha una velocità di: 1
 Questa Ford ha una velocità di: 5
 Questa Bianchi ha una velocità di: 1

 Questa Bianchi ha una velocità di: 1
 Questa Ford ha una velocità di: 5
 Questa Tesla ha una velocità di: 1
```

Spiegare i primi 3 non è difficile: vengono stampati gli elementi, secondo le indicazioni del metodo ToString in Veicolo, in ordine rispetto a come li abbiamo inseriti dentro la lista.

Allora che cosa è successo gli ultimi 3? Che ordine ha seguito per stampare gli elementi? Osserviamo che nel codice abbiamo richiamato un metodo Sort sulla nostra lista, ovvero quel metodo che con le liste o array di oggetti delle classi predefinite del linguaggio, le ordina in ordine crescente, ma come ha fatto questo a ragionare su una classe scritta da noi? In base a cosa ha scelto l'ordine? Molto semplicemente glielo abbiamo detto noi nel metodo CompareTo scritto nella classe Veicolo. Riprendiamo il suo codice:

```
public int CompareTo(Veicolo altroVeicolo)
{
 return this.Modello.CompareTo(altroVeicolo.Modello);
}
```

Il metodo Sort, fa uso del metodo CompareTo che trova nella classe degli oggetti della lista che deve riordinare: noi abbiamo quindi scritto la sua implementazione sfruttando a nostra volta il CompareTo del tipo della properties Modello, ovvero String, che compara le stringe dicendo quale viene prima tra le due messe a comparazione usando l'ordine alfabetico. In effetti il tutto corrisponde con il nostro output: Bianchi comincia con B, quindi viene prima della F di Ford, che a sua volta viene prima della T di Tesla.

Osserviamo meglio però come funziona il metodo **CompareTo**. Questo metodo ritorna un numero intero, tipicamente si parla di -1, 0 o 1, in base al risultato della comparazione.

- -1 lo ritorna quando l'istanza corrente(quindi quella col this) precede quella passata come parametro.

- 0 lo ritorna se le due istanze sono nella stessa posizione.
- 1 lo ritorna se l'istanza specificata come parametro al metodo precede l'istanza corrente.