Capitolo 11
Le classi – ereditarietà, polimorfismo, modificatori

Dopo l'incapsulamento, che abbiamo visto nel paragrafo precedente, incontriamo ora un secondo concetto cardinale della programmazione a oggetti, l'ereditarietà. Si tratta, detto molto semplicemente, della possibilità di definire delle classi (derivate o sottoclassi) che discendono da una di livello gerarchicamente superiore (che chiameremo superclasse o base), una specie di relazione padre – figlio, se vogliamo. In questo senso, a logica, la classe che discende da una superclasse dovrebbe essere una implementazione, un raffinamento di questa. La classe che eredita dalla base può usufruire di tutti i membri non privati del suo chiamiamolo antenato, potendoli anche modificare oltre ovviamente ad averne di propri. E' evidente l'importanza, la comodità e la potenza di questo concetto che potremo estendere anche alle interfaccie, quando ne parleremo. Si possono creare lunghe sequenze di discendenti partendo da una classe e via via collegando via ereditarietà una serie di classi di livelli successivi. Ad esempio non è difficile intuire la comodità di variare una classe e vedere le modifiche diffuse nelle classi derivate. Ma insomma le possibilità sono veramente numerose, l'esperienza porterà ad apprezzare questo meccanismo.

In C# l'operatore che definisce l'ereditarietà è il : (due punti). Non ci sono quindi coinvolte nuove parole chiave. Concettualmente avremo quindi una situazione di questo tipo, nella sua forma più semplice:

```
classe A
{
dati pubblici della classe A
}

classe B : A
{
dati della Classe B
}
```

Cosa significa: abbiamo definito la classe A che avrà la funzione di padre. A ha i suoi metodi pubblici. Poi abbiamo definito la classe B (figlio) la quale oltre che dei suoi dati potrà disporre, in funzione della ereditarietà, anche di quelli della classe A. Banalmente si intuisce che l'ereditarietà si esprime, genericamente come:

class <classe derivata> : <classe base>

C# supporta l'ereditarietà singola, ovvero ogni classe può discendere da una sola classe, diversamente da C++, Python, Perl e altri linguaggi che supportano l'ereditarietà multipla e similmente invece a Delphi, PHP, SmallTalk e Java, per citarne alcuni. Il dibattito fra i fautori della ereditarietà multipla e quella singola è ancora blandamente aperto e non è qui il caso di parlarne se non per dire che anche l'approccio deciso dagli sviluppatori di C# nulla toglie in espressività e potenzialità al linguaggio. Di solito si argomenta che l'ereditarietà multipla porta si a benefici rappresentativi e consente di scrivere codice molto compatto ma, in pratica, introduce tali e tante complicazioni (la più famosa è certamente il cosiddetto "diamond problem" ma anche difficoltà di lettura e debug) che gli svantaggi alla lunga superano i vantaggi. Personalmente sono in qualche modo attratto dall'ereditarietà multipla, ma del resto non mi è mai capitato di dover risolvere problemi tali da richiedere per forza l'uso di questo approccio. C#, come Java permette tuttavia di ereditare da più di una interfaccia.

Vediamo ora un esempio per rompere il ghiaccio:

C# Esempio 6.1

```
1   using System;
2
3   class A
4   {
5       public int x = 1;
6       public void scrivi()
7       {
8           Console.WriteLine("Ciao");
9       }
10  }
11
12  class B : A
```

```
13  {
14      public int y = 2;
15  }
16
17
18  class test
19  {
20      public static void Main()
21      {
22          A a = new A();
23          B b = new B();
24          Console.WriteLine(a.x);
25          Console.WriteLine(b.x + " " + b.y);
26          b.scrivi();
27      }
28  }
```

L'ereditarietà è realizzata alla riga 12 mentre la 25 e la 26 ci mostrano chiaramente, tramite l'istanza b, che la classe B dispone del membro x e del metodo scrivi.

Già da questa breve introduzione si può capire quanto siano importanti i modificatori applicati ai vari campi. Questo discorso, qui appena accennato, è fondamentale da ricordare quando si mette in piedi una architettura di un qualche complessità. Tipicamente ogni membro dichiarato pubblico nella classe base sarà pubblico anche nella sua derivata e questo è un fatto di cui tenere conto in quanto, come conseguenza, sarà consentito un accesso praticamente indiscriminato.
L'esatto opposto avviene se definiamo private i membri della classe padre. Come è noto il modificatore private applicato ai membri rende impossibile la manipolazione di questi al di fuori della classe a cui i membri stessi appartengono. Quindi i derivati ottengono l'eredità si di un membro private ma, al contrario della classe base, non possono accedere ad esso. Anche questo è un aspetto importante da ricordare quando si progetta una classe e si si prevede di derivare da essa altre classi. La conseguenza può essere la imprevista indisponbilità di un dato che magari si riteneva di poter usare.

Vediamo di mettere queste semplici idee in pratica:

```
class A
{
    private static int x = 1;
    public void scrivi()
    {
        int y = 3 + x;
        Console.WriteLine(y);
    }
}
class B : A
{
    public int y = 2 + x;
}
```

In questo caso non è possibile compilare in quanto la classe B eredita dalla classe A il membro x che è private in A e quindi non può essere utilizzato nella somma y = 2 + x. Se x fosse public in A allora non ci sarebbero problemi. In questo caso tuttavia x sarebbe accessibile in modo indiscriminato. Se questo non fosse desiderato un buon compromesso, come avrete intuito, è ricorrere al modificatore protected che garantisce l'accesso al membro sia da parte della classe base che della sua derivata e da eventuali altre successive.

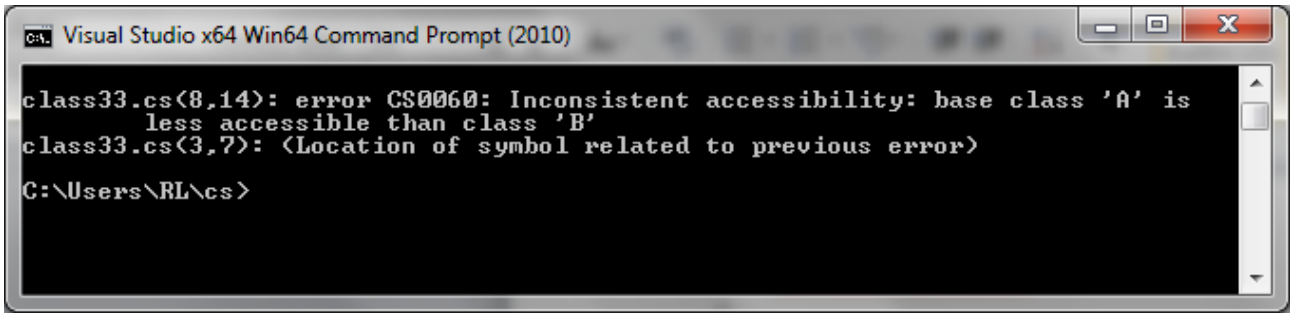
Va inoltre sottolineato **che i costruttori non vengono ereditati;** quest'ultimo fatto ha una sua logica, infatti i costruttori servono alla classe cui appartengono è rischioso propagarli. Tuttavia, nelle situazioni in cui risulta utile, è possibile per una sottoclasse richiamare un costruttore della superclasse di appartenenza

Le particolarità dell'ereditarietà tutta via non finiscono qui; consideriamo il seguente esempio:

```
class A
{
    protected int x = 1;
}

public class B : A
{
    public int y = 2;
}
```

Se cerchiamo di compilare questo codice ne otterremo un errore:



Questo perchè di default, ovvero in assenza di modificatori, la classe A è Internal, ovvero ha un grado di visibilità inferiore rispetto a B che è public. E dal momento che B è una sorta di sottoinsieme di A essa non può avere caratteristiche più generali, per così dire. La soluzione consiste ovviamente nel porre anche A public oppure B internal. Il contrario, ovvero A public e B internal invece funziona, come è facile provare. E le particolarità non finiscono qui: vediamo un altro esempio, abbastanza tipico e diffuso:

C# Esempio 6.1

```
1   public class A
2   {
3       public A(int x)
4       {
5           this.x = x;
6       }
7       private int x;
8   }
9
10  public class B : A
11  {
12  }
13
14  public class Test
15  {
16      public static void Main()
17      {
18          B b = new B();
19      }
20  }
```

Questo codice non va bene e il compilatore ci spiega il perchè:

class68.cs(10,14): error CS1729: 'A' does not contain a constructor that takes 0 arguments

è chiaro: la classe B non ha un costruttore e viene richiamato quello di default che a sua volta cerca di chiamare la classe base A senza passare parametri ma questa classe ha un suo costruttore per cui risulta evidente l'errore. Anche provare a rimediare ponendo la riga 18 come segue:

B b = new B(5);

non può funzionare anzi il problema, come dovrebbe essere evidente, peggiora perchè B non prevede parametri. A questo punto la domanda è: come sistemare le cose? Una soluzione è quella di dotare A di un costruttore di default avente 0 parametri:

```
public class A
{
    public A()
    {}

    public A(int x)
    {
        this.x = x;
    }
    private int x;

}
```

Quindi attenzione anche alla questione costruttori.

A titolo di curiosità sottolineo un fatto che, in fondo, fa capire come il concetto di ereditarietà sia radicato in C#; possiamo infatti affermare che ogni classe che creiamo deriva da una classe base. Laddove questo non sia esplicitato possiamo sempre intuire una derivazione che parte da object, ovvero scrivere:

```
class miaclasse
{
}

equivale a:

class miaclasse: object
{
}
```

Piuttosto interessante è anche il meccanismo di **upcasting** e di **downcasting**; la presenza della parola "cast" ci fa subito pensare ad una forma di conversione e così è in effetti. Il primo di questi meccanismi si basa sull'idea che la classe base è giocoforza più "generica" della classe da essa derivata che consiste in un raffinamento della base. In questo senso una istanza della classe derivata può essere ricondotta alla classe base. Quindi un oggetto di un dato tipo viene ricondotto tramite l'upcasting ad un tipo per così dire più generale. L'upcasting viene sempre permesso dal compilatore. Ovviamente il downcasting fa esattamente il contrario ed è una operazione, come vedremo, un po' più delicata.
Vediamo di affrontare la questione passo passo perchè non è proprio immediata. Definiamo una classe A che sarà la base ed una classe B che da A deriva. Dal momento che A è più generica di B quest'ultima si può considerare alla stregua di un sottoinsieme della precedente. Questo significa che se posso usare A in un certo ambito potrò certamente usare lì anche B. Il seguente programma, che non compila come vedremo, illustra il concetto:

C# Esempio 6.1

```
1   using System;
2   public class A
3   {
4       public int x = 1;
5   }
6
7   class B : A
8   {
9       public int y = 2;
10  }
11
12  class test
13  {
14      public static void Main()
15      {
16          A a1 = new A();
17          B b1 = new B();
```

```
18          A a2 = new B();
19          B b2 = new A();
20      }
21  }
```

Le righe interessanti iniziano dalla 17. Questa, come 18, è del tutto normale, creaiamo due oggetti, a1 e b1 che hanno come riferimento A e B. La riga 19 introduce un esempio di upcasting. L'oggetto creato è di tipo B ma il suo riferimento è la classe A il che può essere in quanto A è più generico rispetto a B. Un oggetto di tipo B è certamente anche di tipo A. La riga 20 invece non compila in quanto un oggetto di tipo A può non essere contenuto in B, sarebbe un downcast da A verso B e quindi non accettabile. Le operazioni di downcast devono essere ottenute attraverso un meccanismo esplicito. Quindi b2 non potrà utilizzare y. Affichè ciò avvenga bisogne effettuare il downcast in modo esplicito e quindi le ultime righe diventeranno ad esempio:

```
A a1 = new A();
B b1 = new B();
A a2 = new B();
B b3 = (B)a2;
```

L'ultima riga, quella in rosso, effettua il downcast verso B. Come abbiamo detto, tuttavia, non sempre il downcast è permesso. Per usare un'altra spiegazione infatti possiamo dire che l'upcasting sposta un oggetto specializzato in un ambito più generale il downcasting fa l'opposto e non è detto che questa maggiore specializzazione sia permessa. In questo caso si può verificare una eccezione a runtime. Ad esempio:

C# Esempio 6.1

```
1   using System;
2
3   public class A
4   {
5       public int x = 1;
6   }
7
8   class B : A
9   {
10      public int y = 2;
11  }
12
13  class test
14  {
15      public static void Main()
16      {
17          A a0, a1 = new A();
18          B b0, b1 = new B();
19          A a2 = new B();
20          B b3 = (B)a2;
21  //      b0 = a1;
22          b0 = (B)a1;
23      }
24  }
```

La riga 21 se non commentate non compila mentre la 22 dà luogo ad una eccezione di cast in fase di runtime a1 non può essere riversato nel tipo B. Per cercare di evitare queste situazioni è possibile ricorrere all'operatore **is** per verificare la natura di un oggetto. Da ultimo segnalo che è anche possibile utilizzare **as** per effettuare un downcast.

Una volta viste le sfaccettature dell'ereditarietà passiamo ad analizzare il terzo cardine della programmazione OOP ovvero il polimorfismo. Si tratta della capacità da parte degli oggetti di modificare il pronatteseprio comportamento dipendentemente dalla classe di appartenenza. In pratica, quando una classe eredita da una superclasse possiamo modificare i membri ottenuti in eredità. Le parole chiave per il polimorfismo sono essenzialmente 3: **new**, **virtual** e **override**.
Prima di proseguire coi formalismi però cerchiamo di capire quando si può rendere necessario questo benedetto polimorfismo.Gli esempi possibili infiniti, uno tipico, classico, che troverete anche altrove, è quella della persona-

lavoratore: ovvero da un insieme generico “persone” definiamo un sottoinsieme “lavoratore” che ha un insieme di caratteristiche comuni, tra cui lo stipendio. Quindi:

```
PERSONA -> nome, cognome, sesso, età, residenza.
LAVORATORE: PERSONA -> datore di lavoro, stipendio.
OPERAIO: LAVORATORE -> mansione
IMPIEGATO: LAVORATORE -> ufficio
```

Abbiamo definito un tipo generico, Persona che ha certi campi base. Da esso discende la classe definita come Lavoratore che ai membri che eredita da Persona aggiunge il datore di lavoro e lo sipendio che andrà calcolato secondo certi criteri. Poi distinguiamo due tipi di lavoratore ovvero impiegati e operai i quali avranno altre caratteristiche. Entrambi condividono la funzione di calcolo dello stipendio che dovrà però essere diversa tra l’uno e l’altro. In questo caso interviene il polimorfismo che permette avere un comportamento differente nei due casi per quanto concerne un membro ereditato, appunto lo stipendio. Questo meccanismo in breve ci permette di definire membri in classi derivate aventi lo stesso nome di quelli della classe base ma con un comportamento differente.

L’esempio è banale, probabilmente ne troverete tanti migliori sulla rete e nei testi ma dovrebbe essere sufficiente a rendere l’idea. Il polimorfismo è perfetto per questo situazioni dove il comportamento di parti comuni cambia al cambiare della classe.

Come abbiamo detto esistono due modi di gestire il polimorfismo a seconda che si sappia in anticipo quali saranno i metodi che ne saranno oggetto oppure no.
Nel caso dell’esempio sappiamo che è il calcolo dello stipendio ad essere oggetto di differenti comporamenti a seconda della tipologia del lavoratore. Per cui potremo ricorrere alla coppia di parole **virtual** e **override**. La prima può essere applicata a metodi, proprietà ed indicizzatori e indica la predisposizione dell’entità alla quale viene applicata ad essere modificata nell’ambito di una classe derivata. In questo senso virtual è irrinunciabile nel senso che non può essere omessa.
Partiamo da un esempio:

C# Esempio 6.1

```
1   using System;
2
3   public class Persona
4   {
5       protected string nome;
6       protected string cognome;
7       protected string eta;
8       protected string residenza;
9   }
10
11  public class Lavoratore : Persona
12  {
13      protected string datorelavoro;
14      protected virtual float stipendio()
15      {
16          float st = 1000;
17          return st;
18      }
19  }
20
21  public class Operaio : Lavoratore
22  {
23      protected string mansione;
24      protected override float stipendio()
25      {
26          float st = 1200;
27          return st;
28      }
29  }
30
31  public class Impiegato : Lavoratore
32  {
33      protected string ufficio;
```

```
34      protected override float stipendio()
35      {
36          float st = 1500;
37          return st;
38      }
39  }
40
41  class test
42  {
43      public static void Main()
44      {
45      }
46  }
```

Questo programma realizza, in forma molto semplice e grossolana, lo riconosco, l'esempio di cui sopra; la classe Lavoratore prevede un metodo stipendio che viene definito attraverso la parola virtual; in questo modo è possibile tramite override utilizzarne una forma modificata nelle due classi derivate Operaio e Impiegato. L'esempio potrebbe essere migliore, ok, ma mi auguro che il concetto sia abbastanza chiaro. La keyword override può essere applicata a metodi, indicizzatori, eventi, proprietà e ha appunto il preciso ed unico scopo di permettere una nuova implementezione di una entità di dei tipi citati che sia stata ereditata. L'assenza di virtual darebbe origine ad un chiaro messaggi d'errore da parte del compilatore. Se invece omettiiamo la parola override il compilatore lancia un warning che ci avvisa che un membro conflitta con uno che stiamo ereditando. Questo è ammissibile ma, ovviamente rende il codice difficilmente gestibile, di qui l'avviso da parte del compilatore; non è un errore ma non è nemmeno conforme ad una corretta programmazione. Il comportamento a run-time è lo stesso che si avrebbe se ci fosse la parola new, che vedremo adesso; infatti proprio il compilatore in questo caso ci invita ad adoperare override o anche **new** la quale keyword costituisce il secondo metodo per realizzare il polimorfismo da un punto di vista pratico. Questo secondo sistema è più, come dire, netto rispetto al precedente. In quanto permette la ridefinzione anche per quanto riguarda ad esempio la parametrizzazione e il tipo laddove con virtual e override è queste caratteristiche non possono essere modificate; questo aspetto è importante da ricordare. Per quanto riguarda l'uso di new possiamo esporre un esempio modificando il codice precedente come segue:

```
public class Lavoratore : Persona
{
    protected string datorelavoro;
    protected float stipendio()
    {
        float st = 1000;
        return st;
    }
}

public class Operaio : Lavoratore
{
    protected string mansione;
    protected new float stipendio()
    {
        float st = 1200;
        return st;
    }
}

public class Impiegato : Lavoratore
{
    protected string ufficio;
    public new float stipendio()
    {
        float st = 1600;
        return st;
    }
}
```

Come si vede è sparita la parola override che non è più necessaria.
Cosa usare nella pratica? La coppia override + virtual è nata con il polimorfismo e per il polimorfismo, si potrebbe dire. Il metodo basato sulla keyword new permette evidentemente di intervenire anche su codice, su classi, create magari

da altri autori e non modificabili. Quindi si pone in evidenza per risolvere problemi pratici quale appunto la necessità di esercitare il polimorfismo in presenza di classi su cui non è possibile intervenire.

C# ci consente anche di utilizzare la versione di un membro appartenente ad una classe base nell'ambito di una derivata. Questo è permesso tramite la parola **base** premessa a quella ad esempio del metodo che desideriamo recuperare.
Anche qui vediamo un semplice esempio:

C# Esempio 6.1

```
1   using System;
2
3   public class Saluto1
4   {
5       public virtual void s1()
6       {
7           Console.WriteLine("Saluto riga 1");
8       }
9   }
10
11  public class Saluto2 : Saluto1
12  {
13      public override void s1()
14      {
15          base.s1();
16          Console.WriteLine("Saluto riga 2");
17      }
18  }
19
20  class test
21  {
22      public static void Main()
23      {
24          Saluto2 s2 = new Saluto2();
25          s2.s1();
26      }
27  }
```

La riga 15 realizza quanto detto: il metodo s1 nell'ambito di Saluto2 effettua un override rispetto all'analogo in Saluto1 ma richiama, appunto tramite **base** il metodo nella classe da cui deriva il metodo s1 di Saluto1.
Una interessante differenza relativo all'uso di new piuttosto che dell'accoppiata virtual + override la possiamo notare attraverso il seguente esempio che presento in parallelo:

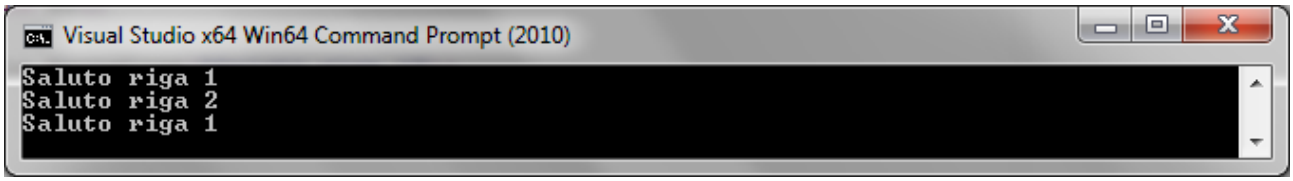
C# Esempio 6.1

```
1   using System;                           using System;
2   public class Saluto1                    public class Saluto1
3   {                                       {
4       public void s1()                        public virtual void s1()
5       {                                       {
6           Console.WriteLine("Saluto riga          Console.WriteLine("Saluto riga
7   1");                                    1");
8       }                                       }
9   }                                       }
10
11  public class Saluto2 : Saluto1          public class Saluto2 : Saluto1
12  {                                       {
13      public new void s1()                    public override void s1()
14      {                                       {
15          base.s1();                              base.s1();
16          Console.WriteLine("Saluto riga          Console.WriteLine("Saluto riga
17  2");                                    2");
18      }                                       }
19  }                                       }
20
21  class test                              class test
```

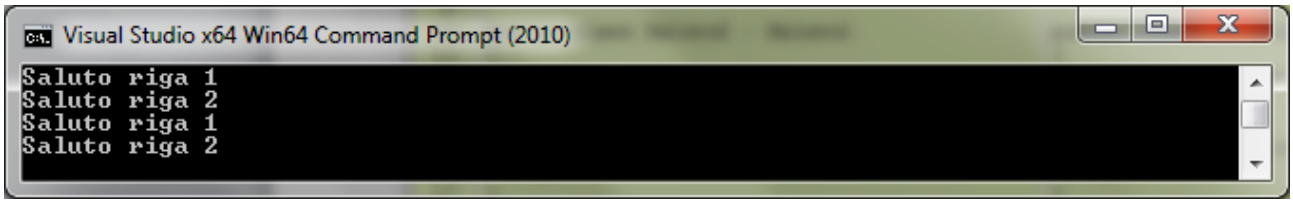
```
22  {
23      public static void Main()
24      {
25          Saluto2 s2 = new Saluto2();
26          s2.s1();
27          ((Saluto1)s2).s1();
        }
    }
```

```
{
    public static void Main()
    {
        Saluto2 s2 = new Saluto2();
        s2.s1();
        ((Saluto1)s2).s1();
    }
}
```

L'output si presenta così per l'esempio di sinistra



Mentre per quello di destra abbiamo:



In pratica attraverso il metodo che fa uso di new possiamo richiamare il metodo della classe base anche attraverso un cast come mostrato alla riga 25; la stessa istruzione nell'esempio di destra non sortisce lo stesso effetto come è evidente.

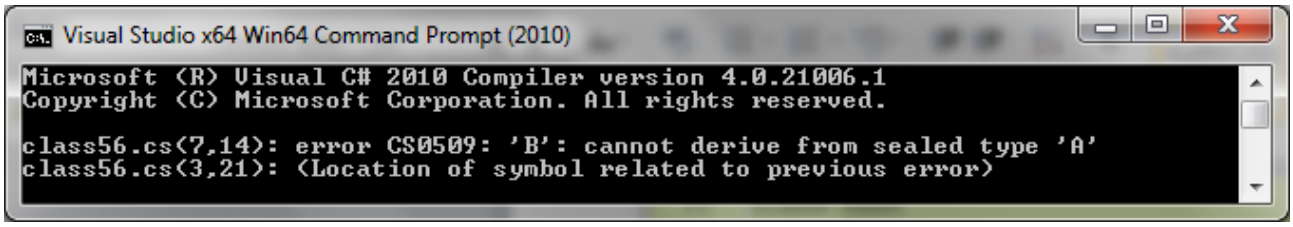
ALTRI MODIFICATORI DI CLASSE

Esistono altra parole chiave che rendono più completo l'uso delle classi modificandone alcune caratteristiche. La prima di queste parole è **sealed** che di fatto impedisce che si possa ereditare dalla classe alla quale viene applicata.

C# Esempio 6.1

```
1   using System;
2
3   public sealed class A
4   {
5   }
6
7   public class B : A
8   {
9   }
10
11  class test
12  {
13      public static void Main()
14      {
15      }
16  }
```

In questo esempio il tentativo di derivare dalla classe A fallisce proprio per la presenza della clausola sealed. Il compilatore rileva questa problematica in maniera molto chiara:



Un esempio per così dire nobile di classe sealed lo troviamo nel framework e lo avevamo già anticipato, si tratta della classe string, della quale possiamo creare tutte le istanze che vogliamo ma dalla quale non possiamo derivare per evitare che si generino delle varianti non standard di un tipo importante come le stringhe. A volte vi sono motivazioni di carattere commerciale che spingono a definire una classe sealed in quanto non si vuole che qualcuno derivi e crei delle varianti di una nostra classe. Va infine sottolineato che una classe sealed gode di alcune ottimizzazioni da parte del compilatore.
Il prossimo esempio ci illustra un altro comportamento peculiare del compilatore di fronte ad una classe sealed:

C# Esempio 6.1

```
1   using System;
2
3   public sealed class A
4   {
5       protected int x = 0;
6   }
7
8   class test
9   {
10      public static void Main()
11      {
12      }
13  }
```

In questo avremo un warning a compile time (new protected member declared in sealed class) in quanto è inutile dichiarare un membro protected in una classe sealed dal momento che il modificatore protected permette l'accesso alle classi derivate che non possono evidentemente esistere.
Il modificatore sealed può essere visto come implicitamente applicato alle strutture, dal momento che anche da esse non si può derivare nulla.

Molto interessante è il modificatore **abstract**. Questo come vedremo è assai utile per la messa in pratica del concetto di polimorfismo e ne costituisce un prezioso alleato essendo invece un po' l'antitesi al modificatore sealed appena incontrato. In realtà abstract si può applicare non solo alle classi ma anche ai metodi o agli indicizzatori, alle proprietà ma è utilizzabile solo nellambito di classi astratte. Ovvero un metodo astratto può esistere solo in una classe astratta. Evidentemente un metodo astratto non avrà un suo "corpo". Tornando a noi, una classe definita abstract può soltanto essere utilizzata come base per altre classi da essa evidentemente derivate. Al suo interno membri astratti sono privi di implementazione, come già detto e devono essere quindi sviluppati all'interno delle classi derivate, realizzando in questo un comportamento pienamente polimorfico. Per evidente contraddizione una classe abstract non può essere contestualmente anche sealed. E' altrettanto chiaro che una classe abstract non può essere istanziata direttamente, non avrebbe senso e il compilatore non accetterebbe un tentativo simile. Il suo scopo naturale, alla fin fine, è quello di funzionare come un **template**, ponendosi in alternativa alle interfaccie, come vedremo quando parleremo di quell'argomento evidenziando le opportune differenze
Ecco un primo passo nel mondo delle classi astratte:

C# Esempio 6.1

```
1   using System;
2
3   public abstract class Quattrolati
4   {
5       abstract public int Area();
6   }
7
8   public class Quadrato : quattrolati
```

```
 9  {
10  }
11
12  class test
13  {
14      public static void Main()
15      {
16      }
17  }
```

Questo programma non compila, in particolare il compilatore ci avvisa che la classe Quadrato non implementa tutti i metodi abstract di Quattrolati; quest'ultima classe è appunto un template, ovvero le classi che da essa deriveranno dovranno per forza implementare, a loro modo, il metodo Area. Più in generale una classe derivata da una base astratta deve obbligatoriamente implementare tutti i metodi stratti della classe base, nessuno escluso. Un esempio funzionante è il seguente:

C# Esempio 6.1

```
 1  using System;
 2
 3  public abstract class Quattrolati
 4  {
 5      abstract public int Area();
 6  }
 7
 8  public class Quadrato : Quattrolati
 9  {
10      int lato = 0;
11
12      public Quadrato (int n)
13      {
14          lato = n;
15      }
16
17      public override int Area()
18      {
19          return lato * lato;
20      }
21  }
22
23  public class Rettangolo : Quattrolati
24  {
25      int lato1 = 0;
26      int lato2 = 0;
27
28      public Rettangolo (int n1, int n2)
29      {
30          lato1 = n1;
31          lato2 = n2;
32      }
33
34      public override int Area()
35      {
36          return lato1 * lato2;
37      }
38  }
39
40  class test
41  {
42      public static void Main()
43      {
44          Quadrato q = new Quadrato(3);
45          Console.WriteLine(q.Area());
46          Rettangolo r = new Rettangolo (3, 5);
47          Console.WriteLine(r.Area());
48      }
49  }
```

Dalla riga 3 alla 6 definiamo la nostra classe astratta. Successivamente le classi derivate Quadrato e Rettangolo implementano il template in particolare dando corpo al metodo Area che obbligatoriamente, come detto, devono definire. Da notare l'uso, obbligatori,o anche questo della parola **override** (righe 17 e 34), in piena linea, se vogliamo, con la filosofia polimorfica. Per quanto banale salta subito all'occhio la differenza tra membri definiti **virtual** e quelli definiti **abstract**: un metodo virtual ha una sua implementazione, un metodo abstract non ce l'ha.
Come detto è anche possibile applicare il modificatore abstract anche ad altre entità come le proprietà, ad esempio a puro titolo formale:

```
public abstract int Area
{
  get;
  set;
}
```

A cui poi seguirà nella classe derivata la parte implementativa

```
public override int Area
{
 get
  {
    return area
   }
  set
  {
    area = value;
  }
}
```

Anche all'interno del framework .Net sono definite delle classi astratte, mi viene in mente una delle più usate, la classe Shape.
Delle classi astratte torneremo a parlare quando affronteremo le interfaccie.

L'ultimo modificatore è introdotto dalla keyword **partial**. Esso non è in realtà significativo quando si ha a che fare con piccoli programmi quali sono i miei esempi mentre è importante per grossi progetti effettuati a più mani. Questo modificatore si applica alle classi ma anche alle strutture, alle interfaccie ed ai metodi e permette che la definizione di queste entità sia suddivisa su più file o comunque in più parti. Questo facilita evidentemente il lavoro distribuito dal momento che una classe può sopravvivere su più parti. In generale valgono alcune semplici regole:

- Tutte le parti devono contenere la parola partial e devono essere disponibili in fase di compilazione
- Devono avere lo stesso livello di accessibilità
- Se una parte è dichiarata sealed o abstract tutta la classe diventa sealed o abstract. Stessa cosa per quanto riguarda il tipo eventualmente specificato in una parte.

Come detto per piccoli progetti la sua utilità è limitata. Di seguito un banale esempio puramente formale:

C# Esempio 6.1

```
1  using System;
2  public partial class A
3  {
4      int x = 0;
5  }
6
7  public partial class A
8  {
9      int y = 0;
10 }
11
12 class test
13 {
14     public static void Main()
15     {
16     }
17 }
```