

Capitolo 13

Le interfacce

La definizione migliore di cosa sia una interfaccia è quella maggiormente generica: si tratta di un insieme, con una propria denominazione, di membri astratti. L'astrazione, come sappiamo, fa sì che detti membri non dispongano di una propria implementazione, si tratta puramente di parti di una struttura in qualche modo protocollare e vedremo perché. Formalmente una interfaccia è invece così determinata:

[attributi] [modificatori] **interface** identificatore [parametri] [lista tipo dei parametri] [base dell'interfaccia] [clause] [corpo]

Tutto quanto si trova tra parentesi quadre è opzionale. Perciò scrivere:

```
interface Iintf {}
```

è già di sé sufficiente per dichiarare una interfaccia di nome `Iintf` completamente vuota. Alla fine fine una interfaccia non è definita in modo differente da una classe laddove esistono due fondamentali differenze; la prima è che banalmente la parola `class` è sostituita da `interface`, la seconda è che non esistono implementazioni nel corpo dell'interfaccia. Il nome pur se casuale, presenta l'applicazione di una delle linee guida proposte per la definizione delle interfacce, ovvero l'attribuzione del nome avente come prima lettera una "I" maiuscola. Nel framework esistono come vedremo molte interfacce e tutte hanno un nome che segue la regola esposta. Un'interfaccia oltre alla sua signature può contenere solo metodi, eventi e delegati. La sua natura protocollare, magari anche si potrebbe definire "contrattuale", si spiega perché **una classe che sia collegata ad una interfaccia si impegna ad implementare ogni elemento sia contenuto nell'interfaccia stessa**. Diversamente il compilatore segnala un errore. La differenza con una classe astratta è abbastanza sottile e, almeno in prima linea, può essere spiegato in maniera logica: una classe astratta dovrebbe essere considerata in senso gerarchico, ogni sua implementazione dovrebbe esserne un raffinamento concettuale mentre una interfaccia è come un modello che va completato. Inoltre, da un punto di vista tecnico, una classe astratta può anche definire al suo interno membri non astratti, costruttori mentre un'interfaccia ha solo membri astratti. Infine, e questo è particolarmente importante, è evidente che sotto un profilo eminentemente pratico, una classe astratta può venire raffinata solo da una classe derivata e quindi si presta molto meno delle interfacce alla costruzione di sistemi gerarchicamente complessi e correlati. Esiste anche una sorta di dualismo con i delegati e ne parleremo in coda al presente capitolo. Tuttavia non è mio scopo in questa sede scendere in dettagli architetturali che possono comunque essere approfonditi altrove.

Vediamo un primo esempio di uso di una interfaccia, che ricordiamo, è un reference type:

C#	Esempio 6.1
1	<code>using System;</code>
2	
3	<code>interface Ianimale</code>
4	<code>{</code>
5	<code> void verso();</code>
6	<code>}</code>
7	
8	<code>class Cane : Ianimale</code>
9	<code>{</code>
10	<code> public void verso()</code>
11	<code> {</code>
12	<code> Console.WriteLine("Bau");</code>
13	<code> }</code>
14	<code>}</code>
15	
16	<code>class Gatto : Ianimale</code>
17	<code>{</code>
18	<code> public void verso()</code>

```

19     {
20         Console.WriteLine("Miao");
21     }
22 }
23
24 class Program
25 {
26     public static void Main()
27     {
28     }
29 }

```

Questo programmino non fa praticamente nulla ma mostra una semplice implementazione da parte di due classi della interfaccia `Ianimale` definita tra la riga 3 e la 6. Questo tipo di implementazione, che fa uso di una completa definizione del metodo all'interno della classe, si definisce **implicita**. Una prima osservazione è che i metodi che implementano quelli definiti nell'interfaccia devono essere pubblici. Il compilatore è molto chiaro su questo aspetto che è determinato dal fatto che i membri di una interfaccia sono internamente automaticamente pubblici, il che, se ci si riflette un attimo è anche logico. Al contrario, anche in questo consequenzialmente, i membri di una interfaccia non possono avere modificatori a corredo. Il legame tra interfaccia ed oggetto che lo implementa può essere anche visto attraverso il seguente esempio:

```

C#      Esempio 6.1
1      using System;
2
3      interface Ianimale
4      {
5          void verso();
6      }
7
8      class Cane : Ianimale
9      {
10         void Ianimale.verso()
11         {
12             Console.WriteLine("Bau");
13         }
14     }
15
16     class Gatto : Ianimale
17     {
18         public void verso()
19         {
20             Console.WriteLine("Miao");
21         }
22     }
23
24
25     class Program
26     {
27         public static void Main()
28         {
29             Ianimale a = new Cane();
30             a.verso();
31         }
32     }

```

In questo caso il metodo viene implementato in forma esplicita (riga 10) in quanto vi è il richiamo diretto del nome dell'interfaccia mentre alla riga 29 viene creata, grossolanamente parlando, una istanza dell'interfaccia stessa specificando che l'implementazione dei metodi è quella della classe `Cane`.

Per verificare se una classe implementa una certa interfaccia è possibile ricorrere all'operatore **is**. Ad esempio, il seguente codice è adatto per l'esempio 13.1

```
public static void Main()
{
    Gatto g = new Gatto();
    if (g is Ianimale)
        Console.WriteLine("yes");
}
```

Molto semplice ma anche molto utile in quanto con una banale istruzione possiamo verificare se una classe dispone dei metodi definiti nell'interfaccia. Una via pratica alternativa e spesso preferibile in quanto più "leggera" fa invece uso dell'operatore **as**. Il fatto che **as** sia più leggero di **is** è dato dal fatto che quest'ultima istruzione prevede un confronto vero e proprio tra **g** e l'interfaccia **Ianimale** dopo di che se il confronto è andato a buon fine si procede. Al contrario **as** restituisce **true** o **false** il che in alcuni casi è molto vantaggioso. Si confrontino ad esempio i seguenti frammenti di codice:

<pre>using System; interface Ianimale { void verso(); } class Gatto : Ianimale { public void verso() { Console.WriteLine("Miao"); } } class Program { public static void Main() { Gatto g1 = new Gatto(); if (g1 is Ianimale) { Ianimale ian = (Ianimale)g1; ian.verso(); } } }</pre>	<pre>using System; interface Ianimale { void verso(); } class Gatto : Ianimale { public void verso() { Console.WriteLine("Miao"); } } class Program { public static void Main() { Gatto g1 = new Gatto(); Ianimale ian = g1 as Ianimale; if (ian != null) { ian.verso(); } } }</pre>
---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Come è evidente il codice contenuto nella colonna di sinistra, che uso di **is**, prevede un primo confronto e successivamente un cast. Quello nella colonna di destra invece effettua un solo confronto attraverso l'operatore **as**. In casi come questo pertanto **as** è preferibile.

Quando una classe implementa una interfaccia allora può essere a pieno titolo trattata come un elemento il cui tipo è l'interfaccia stessa.

Abbiamo visto due modalità di utilizzo dei metodi esposti da una interfaccia ovvero abbiamo visto come implementare in maniera esplicita ed in maniera implicita tali metodi. Un motivo per cui utilizzare il metodo esplicito invece di quello, forse più intuitivo, implicito, è il voler evitare di esporre in maniera forzosamente pubblica elementi della nostra classe. L'esempio 13.3 ci mostrerà un caso di uso pratico di richiamo di un metodo così implementato. Più avanti vedremo un altro motivo per cui è utile l'implementazione esplicita. Che si scelga la via implicita o quella esplicita comunque chi eredita da una interfaccia deve implementarne tutti i metodi, lo sottolineo ancora una volta.

Va comunque specificato che all'interno di una interfaccia non solo possono esservi membri come abbiamo visto finora ma anche eventi, indicizzatori e proprietà mentre sono esclusi costruttori e campi. A parte questo, come detto, non vi è poi gran che, ovviamente in prima istanza, da un punto di vista formale, a

C# - Capitolo 13 – Le interfacce

distinguere una classe da una interfaccia salvo quanto già indicato a inizio paragrafo. Peraltro non è possibile istanziare direttamente un'interfaccia come si fa per una classe ovvero scrivere:

```
interface Ianimale
{
.....
```

```
Ianimale an = new Ianimale(); // non si può!!!!
```

è scorretto e il compilatore è impietoso nel cassare questo codice.

Un'interfaccia può ereditare da una classe ed anche da una o più interfacce; proprio così, più di una e questo vale anche per le classi; infatti contrariamente a quanto avviene per l'ereditarietà definita tra classi classi è prevista la possibilità di utilizzare l'ereditarietà multipla quando si tratta di interfacce. Quindi è possibile scrivere qualche cosa del tipo:

```
interface I1 {}
interface I2 {}
class C : I1, I2
{}
```

o anche

```
interface I1 {}
interface I2 {}
interface I3 : I1, I2
```

Ovvero abbiamo nel primo caso definito la classe C che eredita dalle due interfacce I1 e I2 che devono essere separate tramite una virgola (abbiamo poi tralasciato ogni implementazione). Nel secondo esempio invece l'interfaccia I3 deriva da I1 e I2. Ci siamo limitati al caso in cui l'eredità multipla avviene a partire da due interfacce ma non esiste un limite teorico al numero di quelle da cui si può ereditare, ovviamente. Altrettanto naturale è la conseguenza che una classe che implementi più interfacce può essere oggetto di cast verso una qualunque di quelle interfacce. Formalmente come si può vedere non vi sono particolarità da segnalare. In questi casi il discorso si offre tuttavia a qualche non banale complicazione addizionale. Ad esempio, cosa accade se due (o più) interfacce espongono metodi aventi lo stesso nome? E' un problema, questo genere di conflittualità, ben noto a chi proviene dal C++ o da linguaggi che supportano l'ereditarietà multipla. Vediamo un primo caso in C# con le nostre interfacce nel prossimo esempio:

C#	Esempio 6.1
1	using System;
2	
3	interface In1
4	{
5	void saluta();
6	}
7	
8	interface In2
9	{
10	void saluta();
11	}
12	
13	class C1 : In1, In2
14	{
15	public void saluta()
16	{
17	Console.WriteLine("ciao");
18	}
19	}
20	
21	class Program
22	{
23	public static void Main()

```

24     {
25         C1 classe = new C1();
26         classe.saluta();
27     }
28 }

```

La classe C1 eredita dalle due interfacce I1 e I2 ed implementa il metodo saluta. Di quale delle due interfacce? Molto salomonicamente di entrambe. Questo è il caso più semplice e non offre difficoltà di sorta. E come potremmo fare se volessimo invece sviluppare i metodi in maniera indipendente l'uno dall'altro? La risposta è intuitiva, bisogna fare uso della implementazione esplicita, come nell'esempio che segue:

```

C#   Esempio 6.1
1   using System;
2
3   interface In1
4   {
5       void saluta();
6   }
7
8   interface In2
9   {
10      void saluta();
11  }
12
13  class c1 : In1, In2
14  {
15      void In1.saluta()
16      {
17          Console.WriteLine("io vengo da In1");
18      }
19
20      void In2.saluta()
21      {
22          Console.WriteLine("io vengo da In2");
23      }
24  }
25
26  class Program
27  {
28      public static void Main()
29      {
30          c1 classe = new c1();
31          ((In1)classe).saluta();
32      }
33  }

```

La riga 3 e la 8 definiscono le due interfacce entrambe presentano il metodo saluta. La classe che implementa tale metodo riprende entrambe le interfacce, riga 13 e l'implementazione viene sviluppata richiamando in forma esplicita il metodo stesso, ed è quanto viene svolto alle righe 15 e 20. In questo caso non è possibile utilizzare il modificatore public. E'interessante notare che per richiamare l'opportuno metodo è necessario, come alla riga 31, ricorrere ad un cast esplicito.

A questo punto complichiamo ancora le cose introducendo l'ereditarietà tra interfacce:

```

C#   Esempio 6.1
1   using System;
2
3   interface In1
4   {
5       void saluta();
6   }

```

```

7
8 interface In2: In1
9 {
10 }
11
12 interface In3: In1
13 {
14 }
15
16 class c1 : In2, In3
17 {
18     public void saluta()
19     {
20         Console.WriteLine("ciao");
21     }
22 }
23
24 class Program
25 {
26     public static void Main()
27     {
28         c1 classe = new c1();
29         classe.saluta();
30     }
31 }

```

Anche questo caso, in cui esiste la necessità di implementare un metodo presente in due interfacce ed avente quindi un unico nome in quanto le stesse lo ereditano da un modello archetipo (l'interfaccia In1), la soluzione è uguale al caso precedente. La cosa non finisce qui in quanto talora è possibile che si desideri effettuare un hiding nell'ambito di una interfaccia di un metodo contenuto ed ereditato da un'altra interfaccia. E' necessario allora usare la parola magica, si può davvero definire così, **new**. Ad esempio:

```

interface In2: In1
{
    new void saluta()
}

```

Se si omette tale keyword il compilatore si fa sentire, sia pure solo via warning:

warning CS0108: 'In2.saluta()' hides inherited member 'In1.saluta()'. Use the new keyword if hiding was intended.

La parte che ho sottolineato ci dice appunto come procedere. Va detto che comunque se non si procede ad una implementazione esplicita come vista in precedenza il comportamento del programma sarà comunque identico a quello osservato finora. Questo tipo di hiding tuttavia viene normalmente sconsigliato a livello di programmazione a oggetti (in effetti a me sembra un po' una porcheria... non fosse anche per le conseguenze possibili quando si reinventa e si modifica qualche cosa che esiste già, questo come linea di comportamento generale). Vedremo più avanti un comportamento migliore per questi casi.

Ritorniamo però a considerare i casi di ambiguità. Quale può essere la genesi di questi? E' evidente che un programmatore si può gestire come preferisce i nomi dei metodi delle interfacce da lui create evitando conflitti ma non così avviene quando ad esempio si adoperano interfacce sviluppate da altri. Questo può essere un caso tipico che origina ambiguità nel codice. Non è impossibile stando così le cose che si abbiano delle sovrapposizioni di nomi e che pure non si voglia che i metodi da questi rappresentati abbiano lo stesso comportamento. In tali situazioni tra l'altro generalmente non si ha accesso al codice delle interfacce quindi non è possibile modificare i nomi dei metodi, cosa che oggettivamente è anche parecchio pericolosa per lo sviluppo del resto del programma. La soluzione in questi casi comunque la conosciamo già, si tratta della implementazione esplicita e la vediamo nuovamente in pratica con un altro esempio, che non introduce nulla di eclatante, con cui chiudiamo la casistica:

```

C# Esempio 6.1
1  using System;
2
3  interface Iinsegnante
4  {
5      void sbraita();
6  }
7
8  interface Icapo
9  {
10     void sbraita();
11 }
12
13 class Stressante : Iinsegnante, Icapo
14 {
15     void Iinsegnante.sbraita()
16     {
17         Console.WriteLine("Studia!!!");
18     }
19
20     void Icapo.sbraita()
21     {
22         Console.WriteLine("Lavora!!!");
23     }
24 }
25
26 class Program
27 {
28     public static void Main()
29     {
30         Stressante interlocutore = new Stressante();
31         Iinsegnante i = (Iinsegnante)interlocutore;
32         i.sbraita();
33         Icapo c = (Icapo)interlocutore;
34         c.sbraita();
35     }
36 }

```

In questo caso abbiamo dato una implementazione distinta per i metodi, aventi lo stesso nome, presentati dalle due interfacce supportate dalla classe `Stressante`. In questo caso però esiste una ulteriore piccola complicazione: scoprite infatti cosa succede se aggiungete ad esempio alla riga 35 la seguente istruzione

```
interlocutore.sbraita();
```

la risposta è semplice: non `interlocutore` non può restituire nulla di nulla e il compilatore si arrabbia. La soluzione in questo caso è meno immediata della precedente in quanto, se si vuole mantenere il codice del main così come è, bisogna mettere mano alla classe `Stressante` cosa che si può fare in due modi diversi:

```

class Stressante : Iinsegnante, Icapo
{
    void Iinsegnante.sbraita()
    {
        Console.WriteLine("Studia!!!");
    }

    void Icapo.sbraita()
    {
        Console.WriteLine("Lavora!!!");
    }

    public void sbraita()
    {
        ((Iinsegnante)this).sbraita();
    }
}

```

C# - Capitolo 13 – Le interfacce

```
    }  
}
```

Oppure:

```
class Stressante : Iinsegnante, Icapo  
{  
    void Iinsegnante.sbraita()  
    {  
        Console.WriteLine("Studia!!!");  
    }  
  
    void Icapo.sbraita()  
    {  
        Console.WriteLine("Lavora!!!");  
    }  
  
    public void sbraita()  
    {  
        Console.WriteLine("Urlo");  
    }  
}
```

Nel secondo caso abbiamo definito un metodo diverso dalla implementazione proposta per le due interfacce. Anche in questo caso, ovviamente, tralasciamo gli aspetti “ingegneristici” relativi alla costruzione software 😊.

I membri che risultano implicitamente implementati sono sealed di default. Per cui devono essere opportunamente decorati nella classe base che li implementa per permettere successive operazioni di override. Vediamo un esempio:

C#	Esempio 6.1
1	using System;
2	
3	interface Isaluto
4	{
5	void hello();
6	}
7	
8	class saluto: Isaluto
9	{
10	public void hello()
11	{
12	Console.WriteLine("ciao");
13	}
14	}
15	
16	class saluto2: saluto
17	{
18	public override void hello()
19	{
20	Console.WriteLine("hola");
21	}
22	}
23	
24	class Program
25	{
26	public static void Main()
27	{
28	}
29	}

L'esempio qui sopra non compila e il perchè ci è spiegato chiaramente dal nostro fido amico csc.exe:

error CS0506: 'saluto2.hello()': cannot override inherited member 'saluto.hello()' because it is not marked virtual, abstract, or override

nella classe Saluto il metodo hello è sealed, come spiegato. Per cui è necessario modificare la segnatura del metodo alla riga 10 ad esempio come segue:

```
public virtual void hello()
```

così facendo il programma compila. Rivediamo pertanto il nostro esempio:

C#	Esempio 6.1
1	using System;
2	
3	interface Isaluto
4	{
5	void hello();
6	}
7	
8	class Saluto: Isaluto
9	{
10	public virtual void hello()
11	{
12	Console.WriteLine("ciao");
13	}
14	}
15	
16	class Saluto2: Saluto
17	{
18	public override void hello()
19	{
20	Console.WriteLine("hola");
21	}
22	}
23	
24	class Program
25	{
26	public static void Main()
27	{
28	Saluto s1 = new Saluto();
29	Saluto2 s2 = new Saluto2();
30	s1.hello();
31	s2.hello();
32	((Isaluto)s1).hello();
33	((Isaluto)s2).hello();
34	((Saluto)s2).hello();
35	}
36	}

La possibilità di effettuare in questo modo l'overriding ci risolve anche il problema visto in precedenza di usare new per ridefinire un metodo. Ovviamente esiste anche il caso in cui l'implementazione sia stata fatta in modo esplicito. Ad esempio il seguente frammento illustra tale situazione:

```
interface Isaluto
{
    void hello();
}

class Saluto: Isaluto
{
    void Isaluto.hello()
    {
        Console.WriteLine("ciao");
    }
}
```

In questo caso è evidente che il sistema dell'esempio precedente non può funzionare. Tuttavia esiste un sistema utilizzabile nel caso in cui si preveda che le classi che discendono da Saluto debbano effettuare un override del metodo:

```

C#   Esempio 6.1
1    using System;
2
3    interface Isaluto
4    {
5        void hello();
6    }
7
8    class Saluto: Isaluto
9    {
10       void Isaluto.hello()
11       {
12           hello();
13       }
14
15       protected virtual void hello()
16       {
17           Console.WriteLine("Ciao");
18       }
19   }
20
21   class Saluto2: Saluto
22   {
23       protected override void hello()
24       {
25           Console.WriteLine("hola");
26       }
27   }
28
29   class Program
30   {
31       public static void Main()
32       {
33       }
34   }

```

Questo programma compila e illustra appunto come operare in presenza di implementazione esplicita. Da notare che la classe derivata non può al suo interno modificare il modificatore `protected`.

Anche le `struct` possono ereditare dalle interfacce. Questa è una ulteriore differenza rispetto alle classi astratte dalle quali le strutture non possono invece importare nulla. Anche le `struct` devono mantenere fede all'impegno di implementare tutti i metodi dell'interfaccia da cui ereditano. Di seguito un esempio, poco più che la riprova che la cosa si può fare.

```

C#   Esempio 6.1
1    using System;
2
3    interface Isaluto
4    {
5        void hello();
6    }
7
8    struct Saluto: Isaluto
9    {
10       void Isaluto.hello()
11       {
12       }
13   }
14
15   class Program

```

```
16 {  
17     public static void Main()  
18     {  
19     }  
20 }
```

Delegati e interfacce.

Esiste un piccolo dualismo fra questi due potenti costrutti. In effetti entrambi perseguono la logica di separare l'implementazione di un metodo dalla sua specifica, dalla sua definizione. Per una interfaccia questo è immediato da anche un delegato se ci pensate in attimo si limita a richiamare un metodo esponendo una firma mentre l'implementazione ovviamente sta nel metodo richiamato. Sul web troverete molte descrizioni e alcune discussioni che vi torneranno utili a chiarirvi (forse...) le idee. Ad esempio cito questo link: <http://msdn.microsoft.com/en-us/library/ms173173.aspx>. A parer mio un modo per separare le cose è pensare a cosa si sta manipolando: se lavoro su metodi o gruppi di metodi i delegati possono essere preferibili, normalmente si incontra questa necessità per la soluzioni di problematiche specifiche. Quando si ragiona a livello di classi e si pensa che possano subentrare discorsi come eritarietà, overriding ecc... allora forse è meglio rivolgersi alle interfacce, in base alla mia esperienza siamo a livello di più ampio respiro.