Capitolo 10

Le classi – prima parte

Con le classi arriviamo ad un argomento già più volte invocato e che in effetti ricopre un ruolo centrale nell'ambito di C# e del framework .Net. Cosa sono innanzitutto le classi? Anche in questo caso direi che il concetto di "**contenitore**", in prima istanza, può rendere l'idea. All'interno di questo contenitore troviamo metodi e proprietà, (o, se vogliamo dati e funzioni che manipolano i dati stessi e svolgono altre azioni) variamente disponibili verso l'esterno come indicato nel capitolo dei modificatori. Su internet troverete molte spiegazioni approfondite di questo concetto e delle sue implicazioni ed uso. Le classi ricordano molto da vicino le strutture; diversamente da queste tuttavia, oltre a essere caricate nello heap (o meglio nel managed heap sotto il controllo del CLR) invece che nello stack esse sono protagoniste di complesse architetture in quanto possono discendere da altre classi ma anche avere una lunga sequenza di discendenti potendo dar forma a intricate dinastie, permettetemi questo termine improprio, il cui scopo è rappresentare come meglio possibile il pensiero e la rappresentazione delle varie entità così come pensata dal programmatore facilitando la soluzione al problema che lo stesso si trova ad affrontare. Le classi sono potenti ed espressive ma ci vuole un po' di esperienza per poterle sfruttare al meglio. Per capire meglio la loro natura basta pensare che esse rappresentano in realtà quegli "oggetti" che sono alla base di buona parte della programmazione odierna.

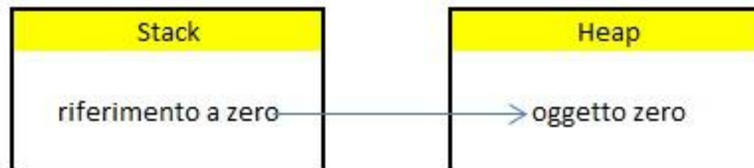
Come al solito un semplice esempio ci permetterà di prendere confidenza con l'argomento.

C# Esempio 10.1

```
1   using System;
2
3   class zero
4   {
5       public int i = 0;
6   }
7
8   class program
9   {
10      public static void Main()
12      {
13          zero z = new zero();
14      }
15  }
```

Finora avevamo usato la parola chiave **class** per definire un contenitore del metodo Main. In realtà essa è la keyword che definisce l'inizio di una classe. Per evidenziarle vediamo alla riga 3 definita una nuova classe. Al suo interno una proprietà pubblica quindi liberamente accessibile costituita da un intero avente valore 0 (diversamente da quanto avviene per le strutture nelle classi l'inizializzazione è possibile). La riga 13 invece, come avrete capito, crea una istanza della classe zero e tale istanza è identificata dal nome "z". A livello terminologico metodi e proprietà sono meglio definiti come membri della classe e in particolare distinguiamo

tra membri dati e membri funzione. I membri dati in particolare comprendono **campi**, **costanti** ed **eventi**. I primi due sono abbastanza intuitivi degli eventi, molto interessanti, riparleremo più avanti. Quindi, ad esempio, all'interno della classe zero il nostro intero "i" è un membro dato. Vedremo ovviamente vari esempi anche di membri funzione; i quali, in breve sono delle procedure che contengono codice finalizzato a manipolare, principalmente, i membri dato.
Da un punto di vista pratico come si presenta la situazione una volta istanzata la nostra classe zero?
Così:



Come d'altra parte era intuibile sulla base di quanto detto.

Vediamo ora un esempio di classe un po' più completo e il suo funzionamento.

C# Esempio 10.2

```
1   using System;
2
3   class sommatore
4   {
5       public int dato1 = 0;
6       public int dato2 = 0;
7       public int somma()
8       {
9           int s = this.dato1 + this.dato2;
10          return s;
12      }
13  }
14
15  class program
16  {
17      public static void Main()
18      {
19          sommatore s = new sommatore();
20          s.dato1 = 6;
21          s.dato2 = 7;
22          int totale = s.somma();
23          Console.WriteLine(totale);
24      }
25  }
```

I programmatori seri hanno il diritto di inorridire ☺ ma lo scopo di questi programmi è solo illustrativo. La riga 3 ci presenta una definizione di classe. Alla riga 5 e alla 6 dato1 e dato2 sono dei membri dato mentre alla 7 troviamo un membro funzione. Ci soffermeremo tra breve sulla parola this mentre per ora è importante capire cosa avviene dalla 19 alla 22. La 19 crea una istanza della classe sommatore tramite l'operatore **new** che si occupa dei corretti caricamenti nello heap, e in questo caso tale istanza si chiama s. Questa può essere vista, a tutti gli effetti, come una variabile che ha a disposizione tutto il corredo della classe alla quale

appartiene, proprio come un intero può usufruire di tutti i metodi e le proprietà di System.Int32 di cui in realtà, come abbiamo visto, è in pratica una istanza. In particolare in questo caso abbiamo definiti due membri dato e un membro funzione. L'accesso a questi membri è possibile tramite il solito operatore **. (punto)** come alla riga 20 ed alla 21 per i dati e alla 22 per la funzione. Quest'ultima riga in particolare è interessante in quanto viene richiamato il metodo il quale restituisce un valore intero, come si evince dalla firma alla riga 7, valore che è assegnato ad una variabile di tipo intero, "totale". In pratica effettuiamo un assegnamento ponendo sulla destra una funzione membro di una classe. Alla riga 9 viene presentato una keyword molto interessante, quel **this** al quale avevamo accennato. Essa permette in breve ad una istanza di usare in un certo senso se stessa ovvero, come in questo caso, membri in essa stessa definiti. Quindi, più formalmente, si tratta di un riferimento alla istanza corrente della classe. La funzione somma ad esempio adopera i valori dato1 e dato2 appena stabiliti.
Questo esempio ha svariate debolezze. La più evidente è quella di presentare tutti i membri come pubblici. In generale questa impostazione rende in un certo senso vulnerabile la classe e se questo non è particolarmente grave in un banale programma stand alone come l'esempio precedente può essere fonte di problemi in progetti più grossi dato che si possono avere accessi incontrollati da qualunque parte ai membri interni della classe. Vale la pena abituarsi subito ad altre tecniche che vedremo qui di seguito. Ovviamente, al punto di partenza al quale ci troviamo, è il sistema più immediato per mostrare i concetti base. E' possibile, a titolo di esercizio, applicare gli altri noti modificatori ai membri, quindi private, protected, internal e protected intenal per osservare cosa accade. Va aggiunto, per quanto dovrebbe essere sottointeso, che una classe al suo interno può contenere qualsiasi tipologia di dati, non necessariamente quindi interi o char ma anche array, stringhe e altre classi.
Facciamo comunque ora un piccolo passo in più senza rinnegare quanto fatto fino ad adesso.

C# Esempio 10.3

```
1   using System;
2
3   class PuntoSulPiano
4   {
5       public PuntoSulPiano(int x, int y)
6       {
7           this.x = x;
8           this.y = y;
9       }
10      public int x;
12      public int y;
13  }
14
15  class program
16  {
17      public static void Main()
18      {
19          PuntoSulPiano p1 = new PuntoSulPiano(1,2);
20      }
21  }
```

La riga 5 ci presenta un metodo speciale per le classi (applicabile anche alle strutture) che si chiama costruttore. **Tecnicamente questo altro non è che un membro che implementa le azioni necessarie alla inizializzazone di una istanza di classe**. Definizione canonica

usata anche da Microsoft. Un costruttore deve avere lo stesso nome della classe a cui appartiene e viene eseguito immediatamente in fase di creazione dell'oggetto. E' quello succede nel nostro esempio, alla riga 5 troviamo tale costruttore, il cui nome deve essere uguale a quello della classe cui appartiene, che attribuisce a x ed y interni i valori parametricamente passati in fase di istanziazione (riga 19). In questo caso i parametri sono passati derettamente ovviamente potrebbero essere ricavati da qualsivoglia altra fonte (standard input, lettura di un file o di un DB....). Un costruttore comunque non necessariamente inizializza delle variabili, è anche del tutto ammissibile costruire una classe come segue:

C#	Esempio 10.4
1	`using System;`
2	
3	`class Inutile`
4	`{`
5	`    public Inutile(int x)`
6	`    {`
7	`        Console.WriteLine(x);`
8	`    }`
9	`}`
10	
12	`class program`
13	`{`
14	`    public static void Main()`
15	`    {`
16	`        Inutile i1 = new Inutile(1);`
17	`    }`
18	`}`

Il costruttore alla riga 5 non va, come si vede, ad impattare su membri interni. Esso viene tuttavia come detto eseguito al momento della creazione della classe e quindi viene stampato a video il numero 1. Ovviamente, di norma, un costruttore viene utilizzato per funzioni di inizializzazione.
Va detto che è possibile anche non definire un costruttore nel qual caso sarà il compilatore a fornirne uno che inizializzerà i dati ai rispettivi valori di default. In ogni caso quindi un costruttore, definito dal'utente o di default, c'è sempre.

Ok, abbiamo messo un po' di carne al fuoco giusto per consentire a chi vuole di provare qualche semplice esempio sapendo, più o meno, cosa sta facendo. E' il momento di approfondire l'interessante discorso relativo ai costruttori.
Possiamo definire il seguente codice:

```
1)
using System;
class PuntoSulPiano
{
    public int x;
    public int y;
}
class program
{
```

```
    public static void Main()
    {
        PuntoSulPiano p1 = new PuntoSulPiano();
        Console.WriteLine(p1.x);
    }
}
```

La classe definita in questo caso non ha alcun costruttore per cui, come accennato, viene utilizzato quello di default che attribuirà il valore 0 a x e a y. Il compilatore si farà sentire segnalando che le due variabili, non essendo state assegnate, avranno valore 0.
Proseguiamo ora fornendo alla classe un costruttore:

2)
```
class PuntoSulPiano
{
    public PuntoSulPiano()
    {
        x = 1;
        y = 3;
    }
    public int x;
    public int y;
}
```

Quando viene creata l'istanza di classe entra in funzione il costruttore assegnando i valori che sono prefissati all'interno della classe. Non è un gran passo avanti ma ci permette di introdurre un successivo miglioramento evidente, che permette una maggior varietà per così dire operativa e che abbiamo già visto:

```
3)
class PuntoSulPiano
{
    public PuntoSulPiano(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    private int x;
    private int y;
}
```

In questo caso il costruttore accetta dei parametri dall'esterno per cui non necessariamente ogni istanza avrà membri con lo stesso valore iniziale come nel passaggio precedente. Questi 3 esempi ci parlano dei **costruttori di istanza**, come si vede il tutto è concettualmente molto semplice. Da notare nell'ultimo frammento di codice che x e y sono private quindi protette rispetto all'esterno.
Diversi sono invece i cosiddetti **costruttori statici**. Questi vengono usati di norma o per inizializzare dei dati statici o per eseguire operazioni che devono essere compiute una sola volta, è più o meno la definizione che si trova un po' ovunque. I costruttori statici sono soggetti ad una serie di regole precise in particolare:

- non possono essere invocati direttamente, quindi non se ne può controllare l'esatto attimo di entrata in gioco
- non accettano modificatori di accesso e nemmeno parametri.

Piuttosto rigidi, se vogliamo, comunque anche questi vengono chiamati in fase di start up della classe. Interessante notare che la presenza di un costruttore statico non impedisce che la classe ne abbia anche uno dinamico come nell'esempio che segue dove entrambi sono presenti rispettivamente alla riga 5 ed alla 9:

C# Esempio 10.5

```
1   using System;
2
3   class Saluto
4   {
5       static Saluto()
6       {
7           Console.WriteLine("Hello");
8       }
9       public Saluto()
10      {
12          Console.WriteLine("Ciao");
13      }
14  }
15
16  class program
17  {
18      public static void Main()
19      {
20          Saluto s1 = new Saluto();
21          Console.ReadLine();
22      }
23  }
```

Interessante è l'uso del modificatore private abbinato ad un costruttore. Di solito si utilizza in classi che presentano solo membri statici, come naturale. Al fine di rendere possibile l'istanziazione deve esserci anche un costruttore pubblico diversamente il compilatore si lamenterà. Provate ad esempio ad eliminare il costruttore pubblico nell'esempio precedente e ad definire private il costruttore alla riga 5. Ovviamente un possibile scopo per cui si usa un costruttore privato è proprio quello di impedire l'istanziazione della classe, in quanto essendovene già uno, sia pure blindato dal modificatore private, non può essere invocato quello di default.

Altra caratteristica da non sottovalutare, l'abbiamo già vista implicatamente nell'esempio 10.5 qui la rimarchiamo, è la possibilità di avere, pienamente disponibili, più di un costruttore, il che può portare a una più agevole e semplice scrittura del codice. E' essenziale che i costruttori abbiano delle signature diverse ovvero non abbiano la medesima parametrizzazione, diversamente non possono coesistere all'interno della stessa classe.

Ad esempio:

C#	Esempio 10.6

```
1   using System;
2   
3   class Punto
4   {
5       public Punto(int a, int b, int c)
6       {
7           this.a = a;
8           this.b = b;
9           this.c = c;
10      }
12      public Punto(int x, int y)
13      {
14          this.x = x;
15          this.y = y;
16      }
17      public int x;
18      public int y;
19      public int a;
20      public int b;
21      public int c;
22  }
23  
24  
25  public class Class1
26  {
27      public static void Main()
28      {
29          Punto p1 = new Punto(1,2,3);
30          Punto p2 = new Punto(1,2);
31          Console.WriteLine(p2.x);
32      }
33  }
```

I due costruttori sono alla riga 5 e alla 12 con un numero di parametri accettati diverso l'uno dall'altro. Se provate a mettere un terzo parametro di tipo intero nella firma del secondo costruttore il compilatore non lo accetterà.
Vediamo a questo punto un esempio molto semplice che rappresenta un po' il punto più avanzato a cui siamo arrivati sinora:

C#	Esempio 10.7

```
1   class Punto
2   {
3       public Punto(int x, int y)
4       {
5           this.x = x;
6           this.y = y;
7       }
8       public int x;
9       public int y;
10  }
12  public class Class1
13  {
14      public static void Main()
15      {
16          Console.Write("Inserisci l'ascissa: ");
17          int x = int.Parse(Console.ReadLine());
18          Console.Write("Inserisci l'ordinata: ");
```

```
19            int y = int.Parse(Console.ReadLine());
20            Punto p2 = new Punto(x, y);
21            Console.WriteLine(p2.x);
22            Console.ReadLine();
23        }
24    }
```

In questo modo possiamo scegliere i valori da passare parametricamente alla classe.

Questo è abbastanza efficiente ma esiste qualche cosa di ancora più elastico ed elegante che permette la manipolazione dei membri interni alle classi: stiamo parlando delle proprietà. Si tratta di qualche cosa molto importante da un punto di vista teorico in quanto permette di mettere in pratica uno dei concetti base della programmazione a oggetti ovvero l'**incapsulamento**. Esso si può definire in tanti modi a me, personalmente, piace vederlo come la possibilità da parte di un oggetto di contenere, manipolare e proteggere tutte le sue caratteristiche interne in piena autonomia evitando di esporre parti della propria implementazione non necessariamente condivisibili. Altrove troverete altre definizioni, più o meno simili, comunque, l'importante è comprendere che si tratta di uno dei cardini del pensiero della programmazione object-oriented.
Da un punto di vista formale una proprietà non è diversa da un campo pubblico. In pratica però esso non ha associato un certo ammontare di memoria per contenere dei dati e dispone di una propria implementazione. Vediamo un semplice esempio base:

C# Esempio 10.8

```
1   using System;
2   class Class1
3   {
4       private int x;
5       public int X
6       {
7           get
8           {
9               return x;
10          }
11          set
12          {
13              x = value;
14          }
15      }
16  }
17
18  class Test
19  {
20      public static void Main()
21      {
22          Class1 c = new Class1();
23          c.X = 7;
24          Console.WriteLine(c.X);
25      }
26  }
```

Le righe evidenziate costituiscono una definizione di proprietà per la classe Class1. Strettamente congiunta è l'istruzione alla riga 24 e quella alla 25. Come si evince proprio da queste due ultime X viene richiamata come fosse un membro qualunque. Esso agisce,

nell'ambito di class1 sul dato x che, essendo private, può essere modificato solo internamente alla classe stessa. Le istruzioni **get** e **set**, come intuibile restituiscono ed attribuiscono i valori di x. Evidentemente la riga 24 richiama set mentre la 25 utilizza get per esporre a video il valore. Ovviamente è possibile definire sequenze di istruzioni più complesse nell'ambito di set e get, per quanto sia buona pratica non appesantire eccessivamenti queste porzioni di codice infilandovi comandi non necessari e non pertinenti. Diversamente sarebbe opportuno rivedere un po' l'architettura di quanto si sta facendo. Da notare, come anticipato, che il costruttore viene richiamato come si trattasse di un normale campo pubblico.
Vediamo quindi un esempio:

C# Esempio 10.9

```
1   using System;
2
3   class class1
4   {
5       private int x;
6       private int y;
7       public class1(int a)
8       {
9           this.y = a;
10      }
11      public int X
12      {
13          get
14          {
15              return x;
16          }
17          set
18          {
19              x = value;
20              if (y % 2 == 0) x = 0;
21              else x = 1;
22          }
23      }
24  }
25
26  class Test
27  {
28      public static void Main()
29      {
30          Console.Write("Inserisci un numero: ");
31          int a = int.Parse(Console.ReadLine());
32          class1 c = new class1(a);
33          c.X = -1; ;
34          Console.WriteLine(c.X);
35      }
36  }
```

In questo caso è stata inserita un piccola elaborazione nell'ambito del set, come si vede dalla riga 19 alla 20.
Evidentemente la parte di codice che interessa il get viene invocata in fase di lettura mentre set viene utilizzata in fase di scrittura. E' possibile utilizzarne solo una delle due e pertanto, sulla base di quanto detto in presenza del solo get saremo davanti ad una classe read-only mentre in caso se è stato definito solo set la classe sarà write-only.

Le proprietà possono anche avere compiti di validazione dei dati prima che questi siano scritti e in questo senso risultano molto utili; ad es:

```
if (value > 10) x = value;
else x = 0;
```

Queste istruzioni, inserite nell'ambito di set, permettono di verificare ed eventualmente ignorare il valore del parametro di passaggio.
Avrete senza dubbio notato la presenza di quella parola **value**. Questa keyword rappresenta il valore manipolato da set, o meglio assegnato, dall'esterno al dato interno alla classe.

Molto utili, parlando di oggetti, sono anche gli indicizzatori. Questi, come il loro stesso nome ci suggerisce sono basati sull'uso di indici, proprio come gli array e infatti sono ben conosciuti nel mondo C# come "smart arrays". Tramite gli indicizzatori è permesso accedere agli oggetti con le medesime modalità degli array quindi tramite l'operatore []. Come viene spiegato un po' ovunque così come una proprietà sembra molto simile ad un campo così un indicizzatore appare simile ad un array. Concettualmente, per quanto in richiamino le proprietà sono un po' meno immediati di comprendere rispetto a queste e, come vedremo ci sono anche alcune importanti differenze.
Formalmente un indicizzatore è fatto come segue, usando sempre la keyword **this** :

```
[modificatore][tipo di ritorno] this [argomenti]
{
  get
  {
  .......
  }
  set
  {
  .....
  }
}
```

Dove il modificatore è ovviamente opzionale. Vediamo un semplice esempio di partenza:

C#	Esempio 10.10
1	`using System;`
2	
3	`class class1`
4	`{`
5	`    private int[] x = new int[]{3,4,7,10};`
6	`    public int this[int i]`
7	`    {`
8	`        get`
9	`        {`
10	`            return x[i];`
11	`        }`
12	`        set`
13	`        {`
14	`            x[i] = value;`
15	`        }`

```
16      }
17  }
18  
19  class Test
20  {
21      public static void Main()
22      {
23          class1 c1 = new class1();
24          c1[3] = 3;
25          Console.WriteLine(c1[3]);
26      }
27  }
```

Alla riga 6 abbiamo la piena applicazione della definizione vista in precedenza. La definizione dell'indicizzatore ci permette di accedere agli elementi del'array privato x, in modo semplice, come se si trattasse di accedere ad un campo pubblico. Ovviamente è necessario che si settino o si vogliano ottenere i valori di un tipo di dato indicizzabile. Ovvero se x fosse un semplice tipo int il programma precedente non compilerebbe.
La casistica che si può presentare è un po' più ampia rispetto al caso delle proprietà. Prendiamo ad esempio la situazione in cui siano due le strutture dati alle quali vogliamo accedere tramite un indicizzatore. Il problema è in realtà solo apparentemente complesso e può essere risolto nel modo più semplice:

C# Esempio 10.11

```
1   using System;
2   
3   class class1
4   {
5       private int[] x = new int[]{3,4,7,10};
6       private int[] y = new int[]{11,12,13,14};
7   
8       public int this[int i, int xy]
9       {
10          get
11          {
12              if (xy == 0) return x[i];
13              else return y[i];
14          }
15          set
16          {
17              if (xy == 0) x[i] = value;
18              else y[i] = value;
19          }
20      }
21  }
22  
23  class Test
24  {
25      public static void Main()
26      {
27          class1 c1 = new class1();
28          Console.WriteLine(c1[3,0]);
29          Console.WriteLine(c1[3,1]);
30      }
31  }
```

In questo caso abbiamo due elementi, definiti alle righe 6 e 7, ai quali accedere e, come si vede, non è necessario ricorrere a più indicizzatori ma si usa quello definito alla riga 9 senza alcun problema.
Una particolarità degli indicizzatori è che possiamo usare una parametrizzazione di qualsiasi natura, quindi non soltanto interi, per identificare gli elementi al suo interno, ovvero non è necessario che la signature dell'indicizzatore abbia un intero come parametro. L'esempio che segue utilizza una stringa come parametro nella firma dell'indicizzatore ma, ripeto, qualsiasi elemento, anche un generico oggetto può essere utilizzato se vi serve.

C# Esempio 10.12

```
1  using System;
2
3  class class1
4  {
5      private string[] x = new string[]{"ciao", " mondo", " crudele"};
6
7      public int this[string s]
8      {
9          get
10         {
11             for (int ind = 0; ind < 3; ind++)
12             {
13                 if (x[ind] == s)return ind;
14             }
15             return 0;
16         }
17     }
18 }
19
20 class Test
21 {
22     public static void Main()
23     {
24         class1 c1 = new class1();
25         Console.WriteLine(c1[" mondo"]);
26     }
27 }
```

La riga critica è la 7 dove si nota che all'indicizzatore è passata una stringa. Il programma restituisce il numero 1, indice di " mondo" nell'array di stringhe x.
L'esempio seguente mostra l'uso di una stringa al posto di un array e char come valore di ritorno:

C# Esempio 10.13

```
1  using System;
2
3  class class1
4  {
5      private string s = "abcdef";
6
7      public char this[int x]
8      {
9          get
10         {
11             return s[x];
```

```
12            }
13        }
14    }
15
16    class Test
17    {
18        public static void Main()
19        {
20            class1 c1 = new class1();
21            Console.WriteLine(c1[3]);
22        }
23    }
```

Per quanto banale si può notare che è possibile andare in modalità read-only implementando solo la parte relativa a get o in modalità write-only implementando solo il set, come abbiamo visto per le proprietà.
Se l'uso degli indicizzatori ricorda da vicino le proprietà esistono comunque, come abbiamo già detto, delle vistose differenze. La prima e più evidente è che le proprietà sono individuate attraverso i loro nomi mentre una differenziazione tra gli indicizzatori è possibile solo tramite le signature dal momento che tutte sono introdotte dalla parola this. In questo senso laddove vi siano più proprietà queste dovranno avere un nome diverso l'una dall'altra, mentre se sono presenti più indicizzatori questi dovranno essere caratterizzati da signature diverse. Inoltre gli indicizzatori fanno uso di parametri, cosa che non avviene quando lavoriamo con le proprietà. Queste ultime sono identificate da un nome mentre gli indicizzatori lavorano tramite indici.

Un altro interessante sistema di presentare i dati in una classe si ha utilizzando la parola chiave **readonly**. Un campo readonly differisce come avevamo anticipato da uno definito const per quanto di primo acchito possano sembrare sostanzialmente identici. La differenza più vistosa consiste nel fatto che tramite const noi imponiamo un riconoscimento del campo e del suo valore già a compile time:

```
public const x = 1;
```

e di qui non si scappa. Un campo readonly invece può comportarsi alla stessa maniera ma anche vedersi attribuito un valore in fase di esecuzione in particolare nell'ambito di un cotruttore. Il prossimo esempio è chiaro in questo senso:

C# Esempio 10.14

```
1     using System;
2
3     class class1
4     {
5         public readonly int r0 = 5;
6         public readonly int r1;
7         public const int c1 = 0;
8         public class1(int a)
9         {
10            r1 = a;
11        }
12    }
13
```

```
14  class Test
15  {
16      public static void Main()
17      {
18          Console.Write("Inserisci un numero: ");
19          int a = int.Parse(Console.ReadLine());
20          class1 c = new class1(a);
21          Console.WriteLine(c.r1);
22      }
23  }
```

La riga che più ci interessa è la 6 dove viene definito un campo r1 e ad esso non è attribuito alcun valore (o meglio gli viene assegnato 0 come valore di default). La riga 10 provvede ad inizializzarlo con un valore parametrico nell'ambito del costruttore di classe; r1 a quel punto non potrà più essere modificato, in particolare una istruzione del tipo:
`c.a = valore;`
nell'ambito del codice della classe Test non può funzionare.
L'operazione alla riga 10 è impossibile da applicare a c1, essendo questo campo blindato tramite la keyword const, mentre funzionarebbe su r0.
Dal momento che un campo readonly può essere inizializzato tramite un costruttore è evidente che costruttori diversi, come visto pienamente ammissibili, possono attribuire ad esso valori diversi.
La keyword readony applicata ad un campo è senza dubbio più duttile rispetto a const, laddove quest'ultima è più sicura e consente al compilatore un certo grado di ottimizzazione in più, grazie al fatto appunto che le informazioni sono note a compile time.
In effetti occorre fare attenzione nella scelta di questi due modificatori, la cosa può non essere banale ad esempio nel caso di programmi cooperanti.

Presentiamo infine un esempio di classe all'interno di un'altra classe, ovvero **innestata**. Il motivo che spinge a usare classi all'interno di altre è dato, oltre che da ragioni "stilistiche", in quanto in alcuni casi rendono più facile la lettura e la manutenzione del codice, anche da possibili questioni di sicurezza; se una classe è utilizzata solo da un'altra può non esservi motivo per esporla esternamente. In questo caso è <u>sufficiente dichiararla come private</u> all'interno della classe che la contiene. Ciò non toglie che sia possibile renderla pubblica perchè sia utilizzata dall'esterno, anche se, di norma, questo non si fa, o almeno è difficile, in base alla mia personale esperienza, trovarsi in situazioni tali per cui questo sia assolutamente il modo migliore per lavorare con le classi.
Piccolo quiz: perchè il codice seguente non compila? Come modificarlo affichè giri? Se non lo capite al volo lanciate il compilatore che si lamenterà in maniera molto esplicita...

```
C#  Esempio 10.15
1   using System;
2
3   class class1
4   {
5       private class class2
6       {
7           int x = 0;
8       }
9       class2 h = new class2();
10  }
```

```
11
12  class Test
13  {
14      public static void Main()
15      {
16          class1 c = new class1();
17          Console.WriteLine(c.h.x);
18      }
19  }
```

NOTA SULLE STRUCT.

Anche le struct, di cui abbiamo parlato rapidamente nel capitolo 4, dispongono di molti meccanismi propri delle classi. Ad esempio la teoria dei costruttori si applica pienamente alle strutture.

C# Esempio 10.16

```
1   using System;
2
3   struct Punto
4   {
5       public int x;
6       public int y;
7       public int z;
8
9       public Punto(int a, int b, int c)
10      {
11          this.x = a;
12          this.y = b;
13          this.z = c;
14      }
15  }
16
17  class test
18  {
19      public static void Main()
20      {
21          Punto p1 = new Punto(1,2,3);
22          Console.WriteLine(p1.x);
23      }
24  }
```

Discorso analogo si può fare per le proprietà e per gli indicizzatori.
Quello che manca alle struct e che invece è caratteristica delle classi.lo vedremo nel prossimo paragrafo.