Capitolo 6

Modificatori – metodi - operatori

Modificatori

Una breve spiegazione la meritano i cosiddetti modificatori. Li abbiamo già incontrati ora ne formalizziamo il ruolo così che anche esempi passati e futuri siano più chiari sotto questo aspetto. I modificatori riguardano le possibilità di accesso ai membri ai quali sono applicati siano quindi proprietà o metodi o altro. In particolare ne sono presenti 4 più uno composto da due precedenti:

- **private** – è il modificatore più restrittivo che consente l'accesso a un membro o al tipo soltanto **alla stessa classe o struct al quale esso appartiene**.
- **public** – è invece il più permissivo in quanto **consente accessibilità completa**.
- **protected** – consente l'accesso nell'ambito della **stessa classe o struttura o da una di quelle derivate.**
- **internal** – l'accesso è permesso **all'interno dell'assembly** cui il tipo o il membro appartiene.
- **protected internal** – permette l'accesso nell'ambito **dell'assembly o anche a un altro purchè ad accedere siano classi derivate da quella a cui il membro appartiene.**
- **static** – come già detto e senza andare troppo sul complicato, questa parola **permette l'accesso a classi o membri senza la necessità di creare istanze**.
- **readonly** – definisce un campo sul quale **sono possibili soltanto operazioni di lettura**. Tale campo andrà perciò inizializzato in fase di creazione.
- **volatile** – applicando volatile un dato **può essere modificato da più threads che lavorano contemporaneamente** ciascuno dei quali pertanto non blocca l'accesso al dato stesso.
- **new** – anche new può essere usato come modificatore e ha lo scopo di **nascondere relativamente ad un metodo per esempio, l'ereditarietà dello stesso da una classe base**.

Si possono applicare praticamente a qualunque tipo di dato e li incontreremo numerose volte nel prosieguo per cui non è il caso di insistere più di tanto. E' sufficiente, per ora, che ne sia chiaro il significato generale.

Metodi

Una questione estremamente importante quando si parla di C# è quella relativa ai metodi. Le struct, come abbiamo visto e le classi, come vedremo, hanno molto di frequente al loro interno dei metodi che definiscono le azioni permesse dalla classe o dalla struct stesse. Detto questo possiamo comunque prendere in considerazione i metodi da un punto di vista puramente formale anche senza addentrarci nell'universo delle classi.

Un metodo in fondo non è altro che un gruppo di istruzioni, si può considerare un sottoprogramma di dimensioni normalmente molto ridotte, che saranno eseguite quando

questo sarà richiamato esplicitamente da una qualche altra entità. Uno dei vantaggi dell'uso dei metodi è il fatto che è possibile utilizzarli quante volte si vuole quindi consentono una miglior organizzazione del codice e un consistente risparmio in termini di scrittura dello stesso. Parte importante per il concetto di metodo è quello di parametro, inteso come valore (spesso più di uno) che può essere passato al metodo e da questo utilizzato al suo interno. Iniziamo da un semplicissimo esempio basilare:

C# Esempio 6.1

```
1  using System;
2
3  class Program
4  {
5      public static void Saluta()
6      {
7          Console.WriteLine("Ciao");
8      }
9
10     public static void Main()
11     {
12         Saluta();
13     }
14 }
```

Qualche cosa relativo ai metodi, se lo ricordate, lo avevamo già visto nel capitolo "Hello World!". La parte che va dalla riga 5 alla 8 costituisce il nostro metodo, che si chiama "Saluta" sempre appartenente alla classe Program. Il metodo è dichiarato pubblico (**public**), è **static**, così da essere immediatamente utilizzabile ed è **void** in quanto non ha valori di ritorno. Dopo il nome del metodo stesso viene dichiarata tra parentesi la lista dei parametri, in questo caso non ce ne sono. Ed è proprio sui parametri che ci concentreremo in questo capitolo. L'esempio che segue è il caso più semplice di passaggio dei parametri:

C# Esempio 6.2

```
1  using System;
2  class Program
3  {
4      public static int Somma(int x, int y)
5      {
6          int somma = x + y;
7          return somma;
8      }
9
10     public static void Main()
11     {
12         Console.Write("Inserisci un numero: ");
13         int a = int.Parse(Console.ReadLine());
14         Console.Write("Inserisci un numero: ");
15         int b = int.Parse(Console.ReadLine());
16         int totale = Somma(a, b);
17         Console.WriteLine("Totale: " + totale);
18     }
19 }
```

In questo esempio il metodo restituisce un int come si nota dalla sua signature e come confermato dall'istruzione **return** che si occupa, appunto, di restituire al chiamante un valore coerente con la signature stessa. Il metodo viene chiamato alla riga 16 dove viene assegnato come valore finale di un intero, cosa che è compatibile col tipo di ritorno del metodo stesso. I parametri sono due, di tipo intero e devono essere passati nel numero richiesto da parte del metodo (che ne chiede appunto due x e y). Questo è il tipo di passaggio di parametri come detto più semplice, nella letteratura inglese si parla di in-parameters e i parametri sono passati "per valore". Ovviamente le regole di scope permettevano alle due variabili a e b di chiamarsi senza problemi x e y, detto per inciso.

Un'altra forma di parametrizzazione utilizza la keyword **ref** e si parla in questo caso di reference parameters o parametri per riferimento. In questo senso si può, in soldoni, affermare che quanto avviene nel metodo modifica anche il parametro del chiamante. Il termine riferimento fa infatti pensare al concetto di puntatore ed in fondo è proprio un puntatore al parametro originale che viene passato al metodo. La parola ref è stata pensata per rendere più chiaro lo scopo e l'uso di questo tipo di passaggio di parametri. Essa deve comparire sia nella signature del metodo sia di fianco al relativo parametro del chiamante. Da notare che, dal momento che passiamo un puntatore ad esso il parametro deve necessariamente essere inizializzato.

Vediamo ora un esempio:

C# Esempio 6.3

```
1  using System;
2  class Program
3  {
4      public static void Somma(int x, int y, ref int somma)
5      {
6           somma = x + y;
7      }
8
9      public static void Main()
10     {
11         Console.Write("Inserisci un numero: ");
12         int a = int.Parse(Console.ReadLine());
13         Console.Write("Inserisci un numero: ");
14         int b = int.Parse(Console.ReadLine());
15         int totale = 0;
16         Somma(a, b, ref totale);
17         Console.WriteLine("Totale: " + totale);
18     }
19 }
```

Il programma mi pare abbastanza chiaro, l'interesse è tutto nelle righe 4 e 16. Quest'ultima in particolare chiama il metodo Somma passando un parametro, che si chiama "*totale*", tramite un riferimento allo stesso. Il metodo somma accoglie il riferimento , il valore del riferimento viene modificato tramite l'operazione alla riga 6 e tale valore conclude il suo ciclo venendo memorizzato nella variabile *totale* di partenza. Come detto e ridetto ☺ la variabile che viene passata per riferimento deve essere inizializzata ciò che avviene alla riga 15 in quanto non è il solo valore che viene passato.

Il terzo sistema di parametrizzazione fa uso della keyword **out**. Anche in questo caso c'è la modifica del parametro corrispondente del chiamante il quale parametro non necessita tuttavia di alcuna inizializzazione. La modifica su di esso consiste proprio nell'attribuzione **obbligatoria** di un valore a tale parametro da parte del metodo chiamato. La keyword stessa "out" in fondo indica chiaramente che vi sarà un valore inviato verso l'esterno. Modifichiamo quindi l'esempio precedente:

C# Esempio 6.4

```
1  using System;
2  class Program
3  {
4      public static void Somma(int x, int y, out int somma)
5      {
6           somma = x + y;
7      }
8
9      public static void Main()
10     {
11         Console.Write("Inserisci un numero: ");
12         int a = int.Parse(Console.ReadLine());
13         Console.Write("Inserisci un numero: ");
14         int b = int.Parse(Console.ReadLine());
15         int totale;
16         Somma(a, b, out totale);
17         Console.WriteLine("Totale: " + totale);
18     }
19 }
```

Come si vede (riga 15) la variabile totale non è inizializzata, è passata come parametro al metodo Somma (riga 16) che in esso riverserà il risultato della elaborazione. Evidentemente out e ref sono molto simile ma, altrettanto ovviamente, funzionano in modo diverso. In particolare il compilatore si comporta in modo diverso, cercando l'esistenza di un valore. La differenza tra i due operatori può facilmente essere comprovata dal seguente esempio parallelo:

```
1  using System;
2  class Program
3  {
4      public static int fun(ref int y)
5      {
6           return 3;
7      }
8
9      public static void Main()
10     {
11         int x = 0;
12         int tot = fun(ref x);
13         Console.WriteLine(tot);
14     }
15 }
```

```
using System;
class Program
{
    public static int fun(out int y)
    {
         return 3;
    }

    public static void Main()
    {
        int y;
        int tot = fun(out x);
        Console.WriteLine(tot);
    }
}
```

La colonna di sinistra compila e stampa regolarmente il risultato, 3, a video mentre cercando di compilare il codice di destra si ottiene un eloquente:

error CS0177: The out parameter 'y' ***must be assigned to before control leaves the current method***

Pertanto, quando si ha a che fare con parametri out è necessario che il metodo restituisca un valore al parametro. Questo è il motivo per cui non vi è necessita di inizializzazione. Un'altra differenza operativa evidente è che non si possono usare parametri out per passare dei valori ad un metodo, nemmeno se i parametri fossero inizializzati. Ovvero:

C# Esempio 6.5

```
1  using System;
2  class Program
3  {
4      public static void fun(out int y)
5      {
6           int z = 3 + y;
7           Console.WriteLine(z);
8           y = 4;
9      }
10     public static void Main()
11     {
12         int x = 6;
13         fun(out x);
14     }
15 }
```

anche se x è inizializzato il programma (forse è troppo chiamarlo così) non compila segnalando che y è unassigned . Se mettessimo ref invece la compilazione andrebbe a buon fine

L'ultima keyword coinvolta nell'argomento parametri è, guarda caso, **params**. Questa è una istruzione molto interessante in quanto permette ad un metodo di accettare un numero variabile di parametri in ingresso. Proprio come abbiamo visto nel capitolo introduttivo con il nostro Console.WriteLine che accetta in input i parametri che vogliamo quindi non ncessariamente 1 o 2 o un numero prefissato. Vediamo subito un esempio e poi ne parleremo.

C# Esempio 6.6

```
1  using System;
2  class Program
3  {
4      public static int sum (params int[] valori)
5      {
6          int somma = 0;
7          foreach (int x in valori)
8          somma += x;
9          return somma;
10     }
11     public static void Main()
12     {
13         int ris1 = sum(2,3,6,100);
14         Console.WriteLine(ris1);
15         int[] temp = {2,3,6,100};
16         int ris2 = sum(temp);
17         Console.WriteLine(ris2);
18     }
19 }
```

La riga 4 è quella effettivamente critica. Ovvero dopo la keyword params abbiamo la definizione di un array di interi. (il prossimo paragrafo parla proprio degli array). L'uso di params è limitato da alcune regole:

- non ci devono essere ulteriori tipologie di parametri, quindi niente out, ref ecc...
- tutti i parametri devono essere dello stesso tipo, nel nostro primo esempio sono interi. C'è un piccolo trucco per aggirare questa limitazione.

Le righe 13 e 16 richiamano il metodo ed evidenziano i due modi per passare i parametri:

- attraverso un elenco i cui elementi sono separati tramite virgole (riga 13)
- attraverso un array usato come parametro (riga16).

In entrambi i casi il risultato è il medesimo, come è facile verificare.

Come detto i parametri devono essere tutti dello stesso tipo. L'esempio che segue insegna come aggirare questa limitazione. La cosa è intuitiva: è necessario usare un qualche cosa che sia adatto a contenere qualsiasi tipo di dato. Sappiamo che tutti i tipi derivano da object pertanto sarà tramite object che risolveremo il problema precedente. Ovviamente è una soluzione da usare quando serve perchè l'impatto sulle prestazioni può a volte disturbare. L'esempio che segue chiarisce quanto detto:

C# Esempio 6.7

```
1  using System;
2  class Program
3  {
4      public static void print (params object[] vari)
5      {
6          foreach (object obj in vari)
7          Console.WriteLine(obj.GetType());
8      }
9      public static void Main()
10     {
11         print(1, "aaa", 'a');
12     }
13 }
```

Operatori

Abbiamo già visto all'opera molti operatori ora è il momento per qualche formalizzazione almeno per i più usati. Sul web e nella letteratura dedicata troverete numerose tabelle devolute all'argomento. Personalmente apprezzo la seguente basata più che altro sull'uso:

Categoria	Operatori
Aritmetici	`+ - * / %`
Logici	`& | ^ ~ && || !`
Concatenazione	`+`
Incremento e decremento	`++ --`
Shift	`<< >>`
Comparazione	`< > <= >= == !=`
Accesso ai membri	`.`
Indicizzazione	`[]`
Cast	`()`
Condizionale (ternario)	`?:`

Delegati (concatenazione e rimozione)	**+ -**
Creazione di oggetti	**new**
Informazione sui tipi	**sizeof is typeof as**
Check sugli overflow	**= += -= *= /= %= &= |= ^= <<= >>=**
Indirezione e indice	**[]**
Coalescenza	**??**

Va detto che in alcuni casi schematizzazione del genere non sono poi così efficaci, almeno a mio modo di vedere anche se hanno il vantaggio del "colpo d'occhio". Alcuni operatori hanno comportanti polivalenti. Ad esempio il classico + è unario in quanto è il segno di un numero mentre è binario qualora impiegato in una somma o in una concatenzazione di stringhe. Particolare è l'operatore condizionale (conosciuto anche come "ternario") costituito dalla coppia **?** e **:** che viene usato come alternativo all'if. In pratica:

```
if (x = = 2) y = 0;
else y = 5;
```

può essere riscritto come:

```
y = x == 2 ? 0 : 5;
```

curiosamente in alcuni ambiti di sviluppo questo tipo di scrittura, compatta e tutto sommato non così criptica viene un po' messa da parte perchè "oscura" (sentito in prima persona...boh).

Di un certo interesse sono gli operatori **is** e **as**. Il primo è usato allo scopo di verificare se, a run-time, un certo oggetto sarà compatibile con un dato tipo. Il formato è molto semplice:

```
espressione is tipo.
```

Il valore di ritorno sarà, come intuibile, true o false. E' importante notare che in questo caso si parla di compatibilità ovvero si verifica se l'oggetto per così dire indagato appartiene proprio a quel tipo o è discendente da un qualche cosa che è compatibile. Esempio:

C# Esempio 6.8

```
1  using System;
2  class Program
3  {
4      public static void Main()
5      {
6          string s = "test";
7          int x = 8;
8          if (s is System.String) Console.WriteLine("ok");
9          if (s is System.Array) Console.WriteLine("ok");
10         else Console.WriteLine("Un array non è una stringa");
11         if (x is System.Int32) Console.WriteLine("ok");
12         if (x is object)Console.WriteLine("Si, x e' un oggetto");
13     }
14 }
```

Durante la compilazione appariranno dei warning che, in sostanza, si limitano a segnalare delle evidenti uguaglianze o disuguaglianze. La riga 12 comunque ci fa notare come un int sia compatibile con object il che è ovvio visto che da esso discende. Questa istruzione risulta senza

dubbio più utile in contesti più complessi con molte classi che costituiscono gerarchie multi livello.

L'operatore **as** invece effettua una conversione tra **reference type** compatibili. Si tratta, evidentemente, di una conversione esplicita. Si distingue da una conversione tramite cast per il fatto che nel caso in cui essa non sia possibile non viene sollevata alcuna eccezione ma semplicemente il ricevente conterrà un valore null. Anche qui vale la pena sottolineare che in alcuni casi è bene che sia così in altri è necessario avere un segnale chiaro che le cose non sono andate a buon fine. E' il programmatore che deve saper distinguere gli scenari. Vediamo un esempio di utilizzo di as:

C# Esempio 6.9

```
1  using System;
2  class Program
3  {
4      public static void Main()
5      {
6          object obj01 = 5;
7          object obj02 = "pippo";
8          string s1 = obj01 as string; //null!
9          string s2 = obj02 as string;
10         Console.WriteLine(s1);
11         Console.WriteLine(s2);
12     }
13 }
```

Se cercassimo di applicare il nostro as sui value types, per esempio scrivendo qualche cosa del tipo:

```
int x = 5;
float f = x as float;
```

il compilatore farebbe le dovute rimostranze. Come detto as si applica solo sui reference.

Un altro operatore utile soprattutto in usi che vedremo più avanti, è **typeof**, già incontrato in qualche esempio nei precedenti paragrafi e che verrà utile in modo particolare quando parleremo di Reflection e della gestione dei metadata. Esso non deve essere confuso con GetType, originario di Object, che permette di ottenere a run-time il tipo di una certa espressione. In effetti esistono vari modi per ricavare informazioni sui tipi.

Il primo, e più semplice, fa uso proprio di GetType:

```
int x = 0;
Type tipo = x.GetType();
```

La variabile tipo è una istanza della classe System.Type. L'istruzione di assegnamento è fattibile in quanto abbiamo conoscenza a compile time della variabile della quale estraiamo il tipo.

Un altro sistema si rifà alla seguente istruzione:

```
Type tipo = typeof(x);
```

Anche in questo caso è necessaria una conoscenza a compile time del tipo

Il terzo metodo, più debole nelle richieste in quanto non ha bisogno di conoscenze a compile time può essere impostato come segue:

```
Type tipo = Type.GetType(x);
```

Quando parleremo di Reflection torneremo sull'argomento, per ora basta sapere lo scopo dell'operatore, anche l'esempio portato, volutamente non in modo integrale, è solo una dimostrazione sintattica, nulla di più.

L'operatore **&** può essere usato in modo binario o unario. In quest'ultimo caso siamo in un contesto **unsafe** e ci viene restituito l'indirizzo dell'operando. Diversamente viene calcolato l'AND logico se applicato ai tipi interi e booleani.

C# Esempio 6.10

```
1  using System;
2  class Program
3  {
4      public static void Main()
5      {
6          int x = 9;
7          bool b = true;
8          if (b & x < 10)Console.WriteLine("ok");
9      }
10 }
```

Ovviamente la cosa è estensibile anche a più condzioni:

```
if (x == 9 & y == 7 & b == false);
```

Esiste una buona alternative a & ed è precisamente **&&**.Questo svolge funzioni del tutto analoghe a & nel senso che

`a & b` è perfettamente equivalente a
`a && b`

la differenza consiste nel fatto che il secondo valuta b solo se necessario. Ovvero se a è falso non valuta b in quanto a quel punto è perfettamente inutile visto che la tabella logica di AND

A	B	AND
Vero	Vero	Vero
Vero	Falso	Falso
Falso	Vero	Falso
Falso	Falso	Falso

indica che è true solo se lo sono tutte le componenti, diversamente è false. In questo senso risulta più efficiente.

Analogamente | valuta l'OR logico. Esso risulta true in tutti i casi tranne quando entrambi i membri oggetto di valutazione risultano false.

A	B	OR
Vero	Vero	Vero
Vero	Falso	Vero
Falso	Vero	Vero
Falso	Falso	Falso

Esempio:

C# Esempio 6.11

```
1  using System;
2  class Program
3  {
4      public static void Main()
5      {
6          int x = 9;
7          bool b = true;
8          if (b | x < 10)Console.WriteLine("ok");
9          if (!b | x == 9) Console.WriteLine("okok");
10     }
   }
```

Anche in questo caso la forma raddoppiata || effettua la seconda valutazione solo se necessario. Se la prima infatti è true allora tutta l'espressione la è e non vi è necessità di ulteriori controlli. Anche in questo caso la forma "short-circuit" come viene detta quella raddoppiata risulta più rapida.

L'operatore ^ invece computa lo XOR o eXclusive OR. Ricordo che la tabella logica per lo XOR è la seguente:

A	B	XOR
Vero	Vero	Falso
Vero	Falso	Vero
Falso	Vero	Vero
Falso	Falso	Falso

In pratica XOR è true quando solo una delle due espressioni lo è da qui la denominazione di OR esclusivo.
Vediamo l'esempio:

C# Esempio 6.12

```
1  using System;
2  class Program
3  {
4      public static void Main()
5      {
6          int x = 9;
7          bool b = true;
8          Console.WriteLine(b ^ x < 10);
9          Console.WriteLine(b ^ x > 10); // solo la prima è true
10     }
11 }
```

L'output è

False
True

Un altro interessante operatore è ~. Questo opera un complemento bit a bit dell'operando in pratica invertendo ognuno dei bit che lo compongono.

C# Esempio 6.13

```
1  using System;
2  class Program
3  {
4      public static void Main()
5      {
6          Console.WriteLine(~5);
7      }
8  }
```

Il risultato è -6. Il che è ovvio. Infatti:

```
 5 = 0000 0000 0000 0000 0000 0000 0000 0101
-6 = 1111 1111 1111 1111 1111 1111 1111 1010
```

L'operatore ~ può essere usato solo su interi, siano essi long, int32, uint ecc...

Diverso è invece un altro operatore, questo di uso molto comune, ovvero ! che rappresenta il NOT e si applica su variabili, valori ed espressioni booleane.

`!false` vale true

Definendo
`bool b = false;`
allora ! potrà essere applicato a b.

Anche `if(!(a == b))` è una scrittura corretta in quanto (a == b) è espressione booleana e quindi il tutto significa "se non è a uguale b" ovvero se a è diverso da b.

Invece scrivere `!2` non compila.

Forse un po' più complessi, anche concettualmente, sono gli operatori di shift ovvero >> e <<. Essi, che agiscono sui long, byte int e char, quindi non sui numeri in virgola, determinano uno spostamento in una direzione o nell'altra a livello binario di un numero di posizioni stabilito dall'operando di destra. In realtà la cosa è teoricamente parlando molto più semplice di quanto si pensi. Come vedremo invece le implicazioni pratiche non sono del tutto banali.
Vediamo l'esempio:

C# Esempio 6.14

```
1  using System;
2  class Program
3  {
4      public static void Main()
5      {
6          int x = 4;
7          int y = x << 3;
8          Console.WriteLine(y);
9      }
10 }
```

Il risultato è il numero 32. Il perchè è chiaro:

4 = 100 effettuando uno shift di 3 posizioni (è appunto il 3 che indica l'ampiezza dello shift) abbiamo 100000 che ò 32 decimale. Oppure

C# Esempio 6.15

```
1  using System;
2  class Program
3  {
4      public static void Main()
5      {
6          int x = 16;
7          int y = x >> 1;
8          Console.WriteLine(y);
9      }
10 }
```

E qui si passa da 10000 binario ovvero 16 a 1000 binario che è 8. Spostamenti eccessivi verso destra provocano l'azzeramento mentre se si va troppo a sinistra si rischiano risultati strani. Come al solito infatti i numeri regalano delle sorprese se vengono presi sotto gamba. Ad es:

```
int x = 1;
int y = x << 31;
```

Ovvero passiamo da

0000 0000 0000 0000 0000 0000 0000 0001 a
1000 0000 0000 0000 0000 0000 0000 0000 che è -2147483648

Oppure:

```
int x = 1;
int y = x << 34;
```

che fa 4. Quest'ultimo risultato può sorprendere assai. In realtà esso si spiega semplicemente col fatto che, nell'ambito del tipo Int32 non è in realtà possibile effettuare uno shift che sia superiore alle 32 posizioni. Questo perchè dell'operando di destra vengono prese in considerazione solo gli ultimi 5 bit se di tipo int e le ultime 6 se di tipo long. (quindi in questo caso lo shift permesso sarà al più di 64 posizioni). In breve se eseguo uno shift di 34 posizioni in realtà effettuerò tale manovra solo per due posizioni reali, come è evidente rappresentando il numero 34 in binario e considerando solo gli ultimi 5 bit, ragionando sugli int32. (Siete pigri? E allora ecco qua:)

0000 0000 0000 0000 0000 0000 0010 0010

Gli ultimi 5 bit rappresentano il numero due. Per cui shiftare 1 di 34 posizioni significa in realtà spostarlo di due quindi 0001 diventa 0100 che è 4. E se invece abbiamo:

```
int x = 4;
int y = x << 30;
```

Il risultato sarà 0. Così pure effettuando uno shift di 31 posizioni. Con l'operando di destra pari a 32 riavremo il nostro 4, come è ovvio.

Lo shift verso destra invece è relativamente più semplice.

```
int x = 8;
int y = x >> 27;
```

dà 0 come risultato, il che è ovvio. Se lo shift ritorna di 32 posizioni ricompare il nostro 8, mentre se l'operando di destra vale 33 il risultato sarà ovviamente 4 (abbiamo effettuato lo spostamento di 33 % 32 posizioni che fa 1).

In presenza di numeri anzichè con gli 0 abbiamo a che fare con gli 1. Il discorso per il resto non cambia gran che:

C# Esempio 6.16

```
1  using System;
2  class Program
3  {
4      public static void Main()
5      {
6          int x = -16;
7          int y = x << 1;
8          Console.WriteLine(y);
9      }
10 }
```

Qui il risultato sarà -32, come intuibile. Siamo passati da

1111 1111 1111 1111 1111 1111 1111 0000 a
1111 1111 1111 1111 1111 1111 1110 0000

Mentre ponendo l'operando di destra pari ad esempio a >> 10 avremmo come ovvio risultato -1 ovvero:
1111 1111 1111 1111 1111 1111 1111 1111
Le precedenti considerazioni restano valide.

Esiste un'altra particolarità: provate a compilare il seguente programma:

C# Esempio 6.17

```
1  using System;
2  class Program
3  {
4      public static void Main()
5      {
6          byte x = 4;
7          byte y = x << 2;
8          Console.WriteLine(y);
9      }
10 }
```

Il compilatore si lamenta nei soliti modi:

error CS0266: Cannot implicitly convert type 'int' to 'byte'. An explicit conversion exists (are you missing a cast?)

Evidentemente anche l'operatore di shift compie internamente una conversione a int che non poi possible riversare direttamente in un byte senza un cast. La soluzione anche in questo caso è quindi molto semplice, basta riscrivere la riga 7 come segue:

```
byte y = (byte)(x << 2);
```

Come si vede quando ci sono di mezzo i numeri bisogna andarci molto ma molto cauti.

Estremamente comuni sono poi tutti gli **operatori di confronto**, utilizzati nell'ambito, ovviamente, di espressioni booleane. Concettualmente non rappresentano nulla di particolarmente complesso e il problema maggiore sta nell'operare i confronti corretti tra entità effettivamente compatibili. Se i tipi numerici ammettono praticamente ogni tipo di confronto ad esempio per i booleani abbiamo solo == e != ovvero "uguale" e "diverso". Scrivere ad esempio:

```
if (true > false)
```

non compila e viene segnalato il fatto che quell'operatore non è applicabile sui booleano. Lo stesso dicasi per:

```
string a = "a";
int x = 0;
Console.WriteLine(a > x);
```

perchè non possono essere confrontati un intero ed una stringa, mentre se la variabile a fosse di tipo char allora la cosa sarebbe fattibile. Il significato dei simboli di confronto dovrebbe essere chiaro a chiunque abbia già programmato e sono comunque intuitivi, ma vediamoli comunque uno per uno:

Simbolo	Significato
>	Maggiore di
<	Minore di
>=	Maggiore o uguale
<=	Minore o uguale
==	Uguale a
!=	Diverso da

Piccolo, banale esempio di utilizzo:

C# Esempio 6.18

```
1  using System;
2  class Program
3  {
4      public static void Main()
5      {
6          Console.WriteLine("Inserisci un numero: ");
7          int x1 = Int32.Parse(Console.ReadLine());
8          Console.WriteLine("Inserisci un numero: ");
9          int x2 = Int32.Parse(Console.ReadLine());
10         if (x1 == x2) Console.WriteLine("I numeri sono uguali");
11         if (x1 > x2)
12         Console.WriteLine("Primo numero maggiore del secondo");
13         if (x1 < x2)
14         Console.WriteLine("Primo numero minore del secondo");
15     }
16 }
```

OVERLOAD

Tutti gli operatori, in C# come in altri linguaggi, hanno un loro comportamento predefinito. La possibilità di mettere in atto una azione di overload vuol dire attribuire definire una implementazione personalizzata di questo comportamento. In fondo overload riporta al concetto di "caricare" ed è esattamente questo lo scopo di questa funzionalità nei confronti di un operatore. Ciò permette ad esempio di mettere in atto soluzioni customizzate ai vari problemi che si possono presentare. Non manca chi storce il naso su questa possibilità, propria non solo di C# ma anche, tra gli altri, di C++; in effetti a volte ci possono essere problemi di lettura del codice e ancor di più a livello di semantica, ma, personalmente, non mi è mai capitato di incontrare difficoltà particolari determinate da questo meccanismo. Comunque sia esso esiste ed è ampiamente utilizzato. L'overload permesso in C# non stravolge in realtà la natura dell'operatore sul quale andiamo ad agire ma permette di gestire operandi user-defined che normalmente non sarebbero accettati

In C# è possibile effettuare l'overload di molti, non tutti, gli operatori. In particolare sono modificabili

```
False, true, + (unario e binario) – (unario e binario)! ~ ++ -- / * % & | ^ <<
>> == != >= > <= <
```

Ritorneremo sull'argomento dopo aver affrontato le classi.