AB...C#

Guida alla programmazione

Antonio Pelleriti

ANTONIO PELLERITI

AB...C# - Guida alla programmazione

a Caterina ed alla mia famiglia

Sommario

Prefazione

Il framework .NET introdotto da Microsoft nel luglio dell'anno 2000, costituisce forse una delle più grandi novità nel campo dello sviluppo del software da parecchi anni a questa parte, e C# ne rappresenta la punta di diamante, il linguaggio di programmazione principe.
A proposito, C# si pronuncia C sharp.
Con C# potremo sviluppare applicazioni desktop, pagine internet veramente e altamente dinamiche, applicazioni per l'accesso ai dati, o ancora per i telefoni e i palmari di ultima generazione, e chi più ne ha più ne metta, insomma ogni campo applicativo è coperto.
Questo manuale è rivolto in particolare a coloro che si avvicinano per la prima volta all'ambiente .NET ed al linguaggio di programmazione C#, fornendo una piccola infarinatura del primo e coprendo i concetti fondamentali di C#, dalla sua sintassi agli strumenti per compilare le applicazioni, dal paradigma orientato agli oggetti alle principali librerie di classi fornite da .NET. Non sarà naturalmente possibile affrontare argomenti più avanzati, come ad esempio i WebService, o l'interoperabilità COM, ma una volta poste le giuste basi sarete magari pronti per affrontare sfide più ardue.

• Organizzazione del libro

Il testo è strutturato nei seguenti capitoli:
Introduzione a C# e .NET. Espone l'architettura del framework .NET ed i suoi componenti e concetti fondamentali, come il Common Language Runtime, la compilazione JIT, la gestione della memoria.
Il primo programma in C#. E' un capitolo introduttivo con il quale comincerete immediatamente a mettere le mani in pasta, scrivendo, compilando, eseguendo il classico esempio introduttivo, con una veloce analisi del codice scritto.
Concetti di base. Il capitolo illustra i concetti fondamentali del linguaggio e della programmazione in C#, introducendo i suoi tipi fondamentali.
Controllo di flusso. Il quarto capitolo illustra i costrutti di C# per il controllo del flusso di un programma e gli operatori del linguaggio.
Programmazione ad Oggetti.. In questo capitolo vedremo cosa significa programmare ad oggetti e come C# supporta il paradigma di programmazione object oriented.
Classi fondamentali. Il capitolo espone ed illustra le classi fondamentali messe a disposizione da .NET, come le stringhe, e le collezioni di oggetti.
Concetti avanzati. E' un capitolo su argomenti leggermente più avanzati, ma indispensabili se si pensa di sviluppare seriamente in C# o in generale in un linguaggio .NET: eccezioni, delegati ed eventi.
Cenni di Windows Forms. Lo sviluppo di un'interfaccia grafica è spiegato in maniera indipendente da qualunque ambiente di sviluppo integrato, facendo uso semplicemente del puro linguaggio e delle classi .NET dedicate allo scopo.
Cenni di Input/Output. Nell'ultimo capitolo sono dati i concetti essenziali per potere salvare dati e leggerli da diversi tipi di file, cosa essenziale per applicazioni di una certa importanza.
Opzioni del compilatore csc. Nell'appendice è riportato un rapido riferimento delle opzioni da utilizzare con il compilatore csc da riga di comando.

• Guida alla lettura

Nel testo verranno utilizzati i seguenti stili e notazioni, per facilitare e rendere più chiara la lettura:
`In carattere a spaziatura fissa appariranno le istruzioni oppure`
`i`

```
blocchi
di codice c#
```

In grassetto le parole chiave di c# oppure altri elementi utilizzati nel linguaggio, come nomi di classi o metodi.

```
Per gli output testuali o i comandi su console verrà utilizzato questo carattere.
```

• Supporto

Antonio Pelleriti è ingegnere informatico, si occupa da anni di programmazione orientata agli oggetti e di metodologie di analisi e di sviluppo del software. Collabora come articolista con riviste del settore informatico, e con vari siti web. Per contattare l'autore, per critiche e suggerimenti, o qualsiasi altra cosa potete utilizzare l'indirizzo e-mail antonio.pelleriti@dotnetarchitects.it.
Il sito web ufficiale dove potete trovare gli esempi del libro, errata corrige, altri articoli e links su C# e .NET, o per dare i vostri commenti, critiche, suggerimenti è http://www.dotnetarchitects.it

• Ringraziamenti

Ringrazio e dedico il libro a Caterina, mia luce e mia guida, che mi sopporta , mi consiglia, e riesce a farmi raggiungere ogni mio obiettivo.

```
while(true) io.TiAdoro().
```

1 Introduzione a C# e .NET

Il framework .NET è l'infrastruttura che costituisce la nuova piattaforma creata da Microsoft per lo sviluppo di applicazioni component-based, n-tier, per internet, per l'accesso ai dati, per dispositivi mobili, o semplicemente per le classiche applicazioni desktop. La piattaforma .NET è composta da diverse tecnologie, strettamente accoppiate fra loro. Questo testo non vuole fornire una panoramica sull'intero universo .NET, ma focalizzerà l'attenzione sul linguaggio C#, linguaggio creato appositamente per il framework .NET.
In questo capitolo vedremo comunque di introdurre le nozioni fondamentali dell'architettura, con lo scopo di porre il lettore in grado di comprendere il contesto in cui si troverà sviluppando con il linguaggio C#, e di effettuare quindi le migliori scelte possibili di progetto e di implementazione.

1.1 L'architettura .NET

L'architettura del framework .NET è illustrata nella figura 1.1. Essa si appoggia direttamente al sistema operativo, nella figura viene indicato Windows, ma esistono e sono anche a buon punto progetti per portare .NET su ambienti diversi, ad esempio Mono su Linux..

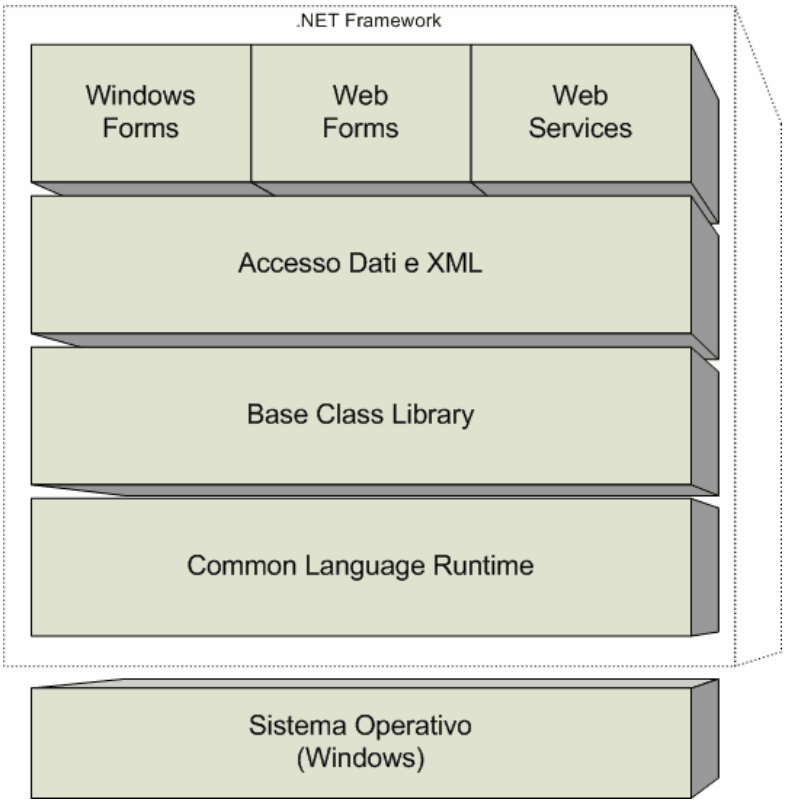


Figura 1.1 L'architettura del framework .NET

Il framework .NET consiste di cinque componenti fondamentali.
Il componente anima e cuore dell'intero framework, è il Common Language Runtime (CLR). Esso fornisce le funzionalità fondamentali per l'esecuzione di un'applicazione managed (gestita appunto dal CLR). Il CLR, a basso livello, si occupa inoltre dell'interfacciamento con il sistema operativo.
Lo strato immediatamente al di sopra del CLR, è costituito dalla Base Class Library (o .NET Framework Class Library) di .NET, cioè un insieme di classi fondamentali, utili e necessarie a tutte le

applicazioni ed a tutti gli sviluppatori. Ad esempio la BCL contiene i tipi primitivi, le classi per l'Input/Output, per il trattamento delle stringhe, per la connettività, o ancora per creare collezioni di oggetti. Dunque, per chi avesse esperienza con altre piattaforme, può essere pensato come un insieme di classi analogo a MFC, VCL, Java.
Naturalmente altre classi specializzate saranno sicuramente mancanti nella BCL. Al di sopra della BCL, vengono quindi fornite le classi per l'accesso alle basi di dati e per la manipolazione dei dati XML, che, come vedrete iniziando a fare qualche esperimento, sono semplici da utilizzare ma estremamente potenti e produttive.
Lo strato più alto del framework è costituito da quelle funzionalità che offrono un'interfacciamento con l'utente finale, ad esempio classi per la creazione di interfacce grafiche desktop, per applicazioni web, o per i sempre più diffusi web service.

1.1.1 CLR

Il componente più importante del framework .NET è come detto il CLR, Common Language Runtime, che gestisce l'esecuzione dei programmi scritti per la piattaforma .NET. Chi viene dal linguaggio Java non farà fatica a pensare al CLR come ad una macchina virtuale del tutto simile concettualmente alla Java Virtual Machine che esegue bytecode Java.
Il CLR si occupa dell'istanziazione degli oggetti, esegue dei controlli di sicurezza, ne segue tutto il ciclo di vita, ed al termine di esso esegue anche operazioni di pulizia e liberazione delle risorse.
In .NET ogni programma scritto in un linguaggio supportato dal framework viene tradotto in un linguaggio intermedio comune, detto CIL (Common Intermediate Language) o brevemente IL, ed a questo punto esso può essere tradotto ed assemblato in un eseguibile .NET, specifico per la piattaforma su cui dovrà essere eseguito. In effetti, a run-time, il CLR non conosce e non vuole conoscere in quale linguaggio lo sviluppatore ha scritto il codice, il risultato della compilazione, è un modulo managed, indipendente dal linguaggio utilizzato, addirittura è possibile scrivere le applicazioni direttamente in linguaggio IL.
La Figura 1.2 mostra il processo di compilazione del codice sorgente in moduli managed.

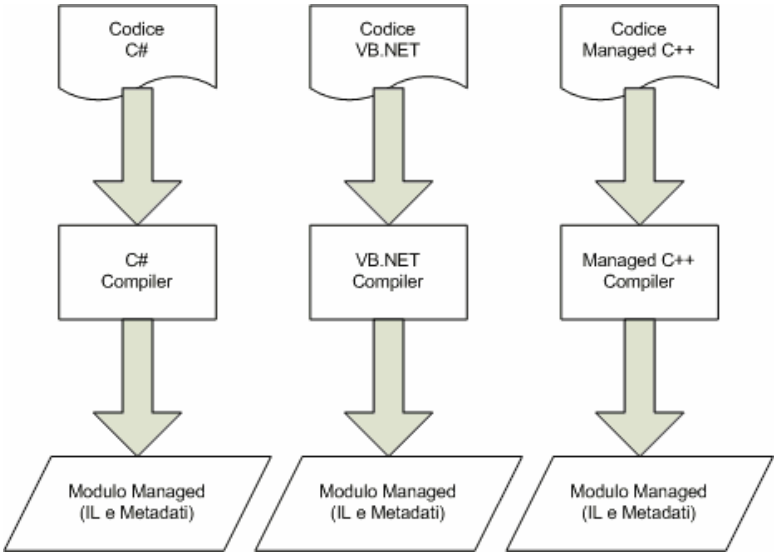


Figura 1.2 Compilazione del codice in moduli managed

Un modulo managed contiene sia il codice IL che dei metadati. I metadati non sono altro che delle tabelle che descrivono i tipi ed i loro membri definiti nel codice sorgente, oltre ai tipi e relativi membri esterni referenziati nel codice.

Il CLR però non esegue direttamente dei moduli, esso lavora con delle entità che sono chiamate assembly. Un assembly è un raggruppamento logico di moduli managed, e di altri file di risorse, ad esempio delle immagini utilizzate nell'applicazione, dei file html o altro ancora, ed in aggiunta a questi file, ogni assembly possiede un manifesto che descrive tutto il suo contenuto, ciò rende possibile il fatto che ogni assembly sia una unità autodescrittiva.
I compilatori .NET, ad esempio il compilatore C#, attualmente creano un assembly in maniera automatica a partire dai moduli, aggiungendo il file manifesto.
Un assembly può essere non solo un'eseguibile, ma anche una libreria DLL contenente una collezione di tipi utilizzabili eventuamente in altre applicazioni.

1.1.2 La compilazione JIT

Il codice IL non è eseguibile direttamente dal microprocessore, almeno da quelli attualmente in commercio, e nemmeno il CLR può farlo, in quanto esso non ha funzioni di interprete. Dunque esso deve essere tradotto in codice nativo in base alle informazioni contenute nei metadati. Questo compito viene svolto a tempo di esecuzione dal compilatore JIT (Just In Time), altrimenti detto JITter.
Il codice nativo viene prodotto on demand. Ad esempio la prima volta che viene invocato un metodo, esso viene compilato, e conservato in memoria. Alle successive chiamate il codice nativo sarà già disponibile in cache, risparmiando anche il tempo della compilazione just in time.
Il vantaggio della compilazione JIT è che essa può essere realizzata dinamicamente in modo da trarre vantaggio dalla caratteristica del sistema sottostante. Ad esempio lo stesso codice IL può essere compilato in una macchina con un solo processore, ma potrà essere compilato in maniera differente su una macchina biprocessore, sfruttando interamente la presenza dei due processori. Ciò implica anche il fatto che lo stesso codice IL potrà essere utilizzato su sistemi operativi diversi, a patto che esista un CLR per tali sistemi operativi.

1.1.3 CTS e CLS

Dati i differenti linguaggi che è possibile utilizzare per scrivere codice .NET compatibile, è necessario avere una serie di regole atte a garantirne l'interoperabilità, la compatibilità e l'integrazione dei linguaggi.
Una classe scritta in C# deve essere utilizzabile in ogni altro linguaggio .NET, ed il concetto stesso di classe deve essere uguale nei diversi linguaggi, cioè una classe come intesa da C#, deve essere equivalente al concetto che ne ha VB.NET oppure C++ managed, o un altro linguaggio.
Per permettere tutto questo, Microsoft ha sviluppato un insieme di tipi comuni, detto Common Type System (CTS), suddivisi in particolare, in due grandi categorie, tipi riferimento e tipi valore, ma come vedremo più in là nel testo, ogni tipo ha come primo antenato un tipo fondamentale, il tipo object, in tal modo tutto sarà considerabile come un oggetto.
Il Common Type System definisce come i tipi vengono creati, dichiarati, utilizzati e gestiti direttamente dal CLR, e dunque in maniera ancora indipendente dal linguaggio.
D'altra parte ogni linguaggio ha caratteristiche distintive, in più o in meno rispetto ad un altro. Se non fosse così, l'unica differenza sarebbe nella sintassi, cioè nel modo di scrivere i programmi.
Un esempio chiarificante può essere il fatto che C# è un linguaggio case sensitive, mentre non lo è VB.NET. Per garantire allora l'integrazione fra i linguaggi è necessario stabilire delle regole, e nel far ciò Microsoft ha creato una specifica a cui tali linguaggi devono sottostare. Tale specifica è chiamata Common Language Specification (CLS).
Naturalmente ogni linguaggio può anche utilizzare sue caratteristiche peculiari, e che non sono presenti in altri, in questo caso però il codice non sarà accessibile da un linguaggio che non possiede quella particolare caratteristica, nel caso contrario, cioè nel caso in cui, ad esempio, un componente è scritto facendo uso solo di caratteristiche definite dal CLS,. allora il componente stesso sarà detto CLS-compliant.

Lo stesso CTS contiene tipi che non sono CLS-compliant, ad esempio il tipo UInt32, che definisce un intero senza segno a 32 bit, non è CLS compliant, in quanto non tutti i linguaggi hanno il concetto di intero senza segno.

1.1.4 Gestione della memoria

In linguaggi come C e C++, lo sviluppatore si deve occupare im prima persona della gestione della memoria, cioè della sua allocazione prima di poter creare ed utilizzare un oggetto e della sua deallocazione una volta che si è certi di non dover più utilizzarlo.
Il CLR si occupa della gestione della memoria in maniera automatica, per mezzo del meccanismo di garbage collection. Tale meccanismo si occupa di tener traccia dei riferimenti ad ogni oggetto creato e che si trova in memoria, e quando l'oggetto non è più referenziato, cioè il suo ciclo di vita è terminato, il CLR si occupa di ripulirne le zone di memoria a questo punto non più utilizzate.
Tutto ciò libera il programmatore da buona parte delle proprie responsabilità di liberare memoria non più utilizzata, e d'altra parte dalla possibilità di effettuare operazioni pericolose nella stessa memoria, andando per esempio a danneggiare dati importanti per altri parti del codice.

Capitolo 2

2 Il primo programma in C#

In questo capitolo inizieremo a scrivere e compilare i nostri primi programmi in C#. Quindi cominciamo con l'esporre la modalità di scrittura e la struttura di ogni programma C#, proseguendo poi con l'introdurre il compilatore a riga di comando **csc**, fornito naturalmente insieme all'SDK del .NET framework.
Per scrivere un programma C# non abbiamo bisogno di nessuno strumento particolare o di potenti e costosi ambienti di sviluppo, anche se andando avanti con lo studio sentiremo l'esigenza, se non la necessità, di qualcosa che ci faciliti la vita di sviluppatore, di tester, di progettista di interfacce grafiche. Per il momento potrebbe addirittura bastarci il notepad di windows o il nostro editor di testo preferito per scrivere semplici programmi, e questa è sicuramente una buona strada da percorrere per imparare al meglio le fondamenta del linguaggio, non solo di C#.

2.1 Struttura di un programma

C# è un linguaggio totalmente orientato agli oggetti. Chi di voi non ha mai sentito parlare del paradigma di programmazione object oriented, troverà per il momento qualche concetto poco chiaro, ma potrà trovarne una efficace spiegazione nei successivi capitoli. Chi invece ha già esperienze di sviluppo in linguaggi come Java o C++ si troverà immediatamente a proprio agio.
Intanto cominciamo col dire che il cuore di un linguaggio di programmazione orientato agli oggetti è il concetto di classe. Dobbiamo pensare a tutto come ad un oggetto, e per scrivere un, pur semplice, programma in C# è necessario definire una classe, classe che conterrà il metodo di partenza dell'applicazione stessa.
Dopo questa breve premessa possiamo subito illustrare come è strutturato ogni programma C#. Cominciamo dicendo che i programmi possono naturalmente essere contenuti su più file di testo, e C# usa l'estensione .cs per questi.
Ogni file C# può contenere uno o più namespace (spiegati in maggior dettaglio nel paragrafo relativo, più avanto in questo capitolo), cioè spazi di nomi, all'interno dei quali sono contenute le definizioni di tipi, come classi, interfacce, strutture, enumerazioni, e delegati.

```
// struttura di un programma C#
using SpazioDiNomi;
namespace UnNamespace
{
   class MiaClasse
   {…}
   struct MiaStruct
   {…}
   interface IMiaInterfaccia
   {…}
   delegate tipo MioDelegate();
   enum MiaEnum
   {…}
   namespace UnAltroNamespace
   {
            …
   }
   class ClassePrincipale
   {
      public static void Main(string[] args)
      {
                  …
      }
```

```
    }
}
```

Il precedente blocco di codice costituisce lo scheletro generale di un programma C#.

2.2 Ciao Mondo

Chi si aspettava che qualunque libro di programmazione iniziasse con il classico esempio Hello World, è qui subito accontentato:

```
//primo programma C#
using System;

class CiaoMondo
{
      public static void Main()
      {
            Console.WriteLine("Ciao mondo");
      }
}
```

Adesso ne spiegheremo il funzionamento ed il significato.

2.2.1 Compilazione ed esecuzione

Dopo aver salvato il codice in un file di testo, con estensione .cs, ad esempio CiaoMondo.cs, per mandare in esecuzione il programma bisogna compilarlo, cioè creare il file eseguibile. Per far ciò utilizziamo il compilatore **`csc.exe`**, il cui modo di utilizzo è il seguente:

```
csc CiaoMondo.cs
```

Se tutto è filato liscio, cioè non abbiamo commesso errori nella scrittura del programma, otterremo un file che sarà eseguibile su un pc sul quale sia stato installato il .NET runtime. In questo caso specifico, verrà generato un file dal nome CiaoMondo.exe.
A questo punto possiamo eseguire il programma semplicemente lanciando il file CiaoMondo.exe ed otterremo sulla console la mitica frase:

```
Ciao Mondo!
```

Il compilatore csc permette di specificare svariate opzioni per la compilazione di un'applicazione. I dettagli per il suo utilizzo sono riportati nell'appendice del testo, ma mano a mano che procederemo nel libro ed ove si renda necessario utilizzare una specifica opzione, daremo le apposite istruzioni.

2.3 Un'occhiata al codice

Nel nostro primo esempio abbiamo definito una classe di nome CiaoMondo, utilizzando la parola chiave **`class`**. L'unico metodo che essa contiene è il metodo Main, e al suo interno non facciamo altro che richiamare il metodo per la scrittura di una linea di testo sulla console.

2.3.1 Input e output

La libreria di classi del framework .NET contiene, fra le altre, la classe System.Console.
Nell'esempio introduttivo abbiamo utilizzato il metodo statico **`WriteLine`**, che stampa una riga di testo sulla console andando poi a capo.
Un metodo analogo, **`ReadLine`**, che utilizzeremo più avanti nel testo, svolge l'operazione complementare, leggendo un testo inserito dall'utente al prompt dei comandi.

2.3.2 Il metodo Main

La classe CiaoMondo contiene il metodo principale dell'applicazione, cioè, come detto, il suo punto di ingresso, il punto dal quale l'applicazione comincia la sua esecuzione. Il comportamento di una classe è definito poi all'interno del blocco di testo delimitato da due parentesi graffe {}.
Il metodo **`Main`** può essere implementato in diverse maniere, prestando sempre attenzione all'iniziale maiuscola (C# è un linguaggio case-sensitive, dunque main è diverso da Main), ed esso può restituire un valore intero oppure nessun valore, inoltre può accettare dei parametri in ingresso.
Sebbene nell'esempio precedente abbiamo usato il modificatore `public` per il metodo Main, non ha nessuna importanza se esso sia presente o meno, anzi funzionerebbe tutto quanto anche mettendo `private`. Il metodo viene in ogni caso richiamato dall'esterno del programma per consentirne appunto l'avvio.
I possibili modi di scrivere il metodo sono, tralasciando il modificatore, i seguenti:

```
static void Main() {…}
static void Main(string[] args) {…}
static int Main() {…}
static int Main(string[] args) {…}
```

il primo e' il modo più semplice di scrivere il metodo, il quale in questo caso non restituisce nessun valore al termine dell'esecuzione del programma, come indicato dalla parola chiave void, e non prende nessun argomento in ingresso.
Nel secondo caso invece, è possibile avviare il programma usando degli argomenti., che saranno contenuti nella variabile args (vedremo più in là come trattarli).
Negli ultimi due casi il metodo, e dunque il programma, dovrà restituire al termine della sua esecuzione un valore intero, che in genere viene costituisce il codice di terminazione del programma stesso.
Al momento della compilazione, il compilatore ricerca per default il metodo Main dell'applicazione, e per default esso se ne aspetta uno ed uno solo.
E' però possibile scrivere più metodi Main in diverse classi, cosa molto utile ad esempio per la creazione di un test per ogni classe che implementiamo.
Nel caso in cui il compilatore trovi più di un metodo Main, esso restituirà un errore di compilazione specificando i molteplici Main trovati. Per evitare tale errore bisogna dunque specificare quale fra essi deve essere utilizzato come punto d'ingresso, utilizzando l'opzione /main del compilatore per specificare la classe che contiene il metodo, ad esempio nel seguente modo:

```
csc CiaoMondo.cs /main:CiaoMondo
```

Anche se ancora non abbiamo specificato il concetto di namespace, ma lo vedremo fra poco, sottolineiamo intanto che il nome della classe deve essere specificato in modo completo, includendo cioè anche i namespace che la contengono.

2.3.3 Argomenti del main

Come detto qualche riga fa, è possibile passare degli argomenti ad un programma dalla linea di comando.
Tali argomenti, vengono passati al metodo Main in un vettore di stringhe, eventualmente vuoto. Non abbiamo ancora trattato gli array in C#, ma per il momento basterà dare un occhiata ad un semplice esempio per capire come utilizzarli in questo caso:

```
//un altro programma C# con argomenti
//passati al main

using System;

class CiaoMondo
{
      public static void Main(string[] args)
      {
```

```
            if(args.Length>0)
            {
                  Console.WriteLine("Ciao {0}",args[0]);
            }
            else Console.WriteLine("Ciao, chi sei?");
      }
}
```

Dopo aver compilato il codice, ed aver ottenuto ad esempio il file esempio.exe eseguiamolo passando un argomento sulla linea di comando:

```
esempio Antonio
```

L'array args in questo caso conterrà la stringa `Antonio`, quindi la proprietà Length restituirà per l'array una lunghezza pari a 1.
Il metodo WriteLine stavolta stamperà la stringa Ciao seguita dall'elemento 0 dell'array args, cioè dal suo primo e unico elemento. E' bene sottolineare subito, infatti, che gli array, o meglio i loro elementi vengono numerati a partire da zero (vedi Array a pag. 28).

2.4 Commenti

I commenti all'interno del codice sono di importanza fondamentale, sia quando esso dovrà essere usato da altre persone, ma anche per lo sviluppatore stesso, che riprendendo un suo vecchio programma riuscirà ad orientarsi e a capire sicuramente meglio e con meno fatica, quello che aveva scritto e quello che voleva ottenere.
C# permette due modalità di inserimento di un commento all'interno del codice. Con la prima è possibile inserire un commento su una linea sola:

```
//un commento su una linea
```

mentre qualora sia necessario inserire commenti più lunghi, che si estendano cioè su più righe viene usata la seconda modalità, in cui il commento è delimitato fra le sequenze /* e */, ad esempio

```
/* questo è
 un commento su
 tre linee */
```

C# supporta inoltre la generazione automatica di documentazione, utilizzando dei particolari tag XML inseriti nei commenti.

2.5 Organizzare le classi

Per fornire una organizzazione gerarchica del codice, simile al modo in cui sono organizzate delle directory nel file system, C# utilizza il concetto di namespace.
A differenza del concetto di directory e file però, il concetto di namespace non è fisico ma solamente logico, non è necessaria cioè corrispondenza fra il nome dei namespace e il nome delle directory in cui sono contenuti.
Il namespace permette di specificare in modo completo il nome di una classe in esso contenuta, separando i nomi dei namespace con un punto (.) e finendo con il nome della classe stessa.
Per mezzo dei namespace ad esempio possiamo raggruppare classi che si riferiscono o che forniscono funzionalità correlate.

2.5.1 namespace

Per creare un namespace basta utilizzare la parola chiave **`namespace`**, seguita da un blocco {} all'interno del quale inserire le nostre classi o altri namespace. Ad esempio:

```
//esempio namespaces
using System;
namespace edmaster
{
      namespace ioprogrammo
```

```
	{
		namespace capitolo1
		{
			class CiaoMondo
			{
				//corpo della classe CiaoMondo
			}
		}
	}
}
```

In questo esempio abbiamo definito tre namespace annidati, che portano la nostra classe CiaoMondo ad avere il seguente fullname:

```
edmaster.ioprogrammo.capitolo1.CiaoMondo
```

Lo stesso risultato si ottiene in maniera più compatta usando direttamente la notazione seguente:

```
namespace edmaster.ioprogrammo.capitolo1
{
	class CiaoMondo
	{
		…
	}
}
```

2.5.2 using

La seconda linea di codice dell'esempio CiaoMondo fa uso di un'altra parola chiave di C#, **`using`**.
Tale parola chiave viene utilizzata per evitare l'utilizzo del nome completo della classe all'interno del codice, ad esempio la classe Console è contenuta nel namespace System, e senza l'utilizzo dell'istruzione

```
using System;
```

avremmo dovuto scrivere

```
System.Console.WriteLine("Ciao Mondo!");
```

e così per ogni classe contenuta nel namespace System. L'utilizzo del solo nome della classe si rende quasi necessario e senza dubbio comodo quando abbiamo a che fare con namespace molto annidati e quindi con nomi completi delle classi molto lunghi, ma bisogna ricordare che namespace e tipi devono avere nomi univoci, e che potrebbe accadere di trovare due classi con lo stesso nome all'interno di namespace diversi, ad esempio supponiamo di avere due namespace con i nomi seguenti ed usare la keyword **`using`** per riferirci al loro contenuto:

```
using edmaster.ioprogrammo.capitolo1.esempi
using edmaster.tag.capitolo2.esempi
```

Se all'interno dei due namespace sono presenti due classi con eguale nome **`Test`**, il compilatore non riuscirebbe a distinguere le due classi. In questo caso si rende necessario l'utilizzo del nome completamente qualificato:

```
edmaster.ioprogrammo.capitolo1.esempi.Test test1;
edmaster.tag.capitolo2.esempi.Test test2;
// la classe di test1 è diversa da quella di test2
```

Nello stesso ambito può essere utile l'altro modo di utilizzo della parola chiave **`using`**, che permette di abbreviare il nome di un namespace con un alias di nostra scelta, ad esempio scrivendo

```
using iop=edmaster.ioprogrammo.capitolo1.esempi
```

possiamo utilizzare l'alias **`iop`** in tutto il nostro codice, ad esempio possiamo chiamare ed utilizzare la classe

```
edmaster.ioprogrammo.capitolo1.esempi.Test
```

semplicemente con

```
iop.Test
```

A questo punto siete pronti per iniziare a programmare in C#, se è la prima volta che vi avvicinate al mondo della programmazione, potete cominciare con il ricopiare il codice dell'esempio CiaoMondo e compilarlo, poi magari fate qualche modifica e vedete ciò che succede.

3 Concetti di base

Per imparare seriamente un linguaggio è necessario porre le basi, e studiarne in maniera dettagliata ed approfondita la sintassi e le strutture di programmazione che esso mette a disposizione. In questo capitolo dunque iniziamo a porre le fondamenta del linguaggio C#.

3.1 Dichiarazioni

In C#, come in ogni linguaggio di programmazione, le dichiarazioni consentono la definizione degli elementi che costituiranno il programma.
Abbiamo già visto come avviene la dichiarazione di un namespace, ed abbiamo detto che all'interno dei namespace trovano posto le dichiarazioni dei tipi di quel dato namespace. Questi tipi possono essere ad esempio classi, struct, enum, delegati. All'interno dei tipi, ancora, verranno dichiarati altri elementi, come campi e metodi di una classe.
Le dichiarazioni sono necessarie, in quanto servono ad assegnare ad ogni elemento del programma un nome univoco, perlomeno nel proprio campo d'azione o scope.
Ad esempio scrivendo l'istruzione

```
UnTipo numero;
```

stiamo dichiarando che **`numero`** è un identificatore di un elemento di tipo **`UnTipo`**, abbiamo cioè dato un nome ad un elemento del programma.

3.2 Variabili

Una variabile è un identificatore di una zona di memoria, ed ogni variabile è associata ad un tipo che identifica ciò che la data porzione di memoria dovrà contenere.
Nel paragrafo precedente abbiamo già utilizzato una variabile, scrivendo

```
MioTipo numero;
```

abbiamo dichiarato una variabile di tipo *MioTipo*, cioè abbiamo prenotato e delimitato una zona di memoria adeguata a contenere un *MioTipo*. Diamo un'occhiata a degli esempi più reali, anche se ancora non abbiamo parlato dei tipi di C#.

```
int i;
float f;
string bt;
```

le tre istruzioni precedenti dichiarano tre variabili, i, f e bt, di tipo rispettivamente *int*, *float* e *string*. Queste dichiarazioni indicano inoltre al compilatore di allocare uno spazio di memoria adeguato a contenere tre dati dei relativi tipi. Ad esempio, come vedremo più avanti, scrivendo:

```
i=10;
```

Abbiamo inserito nell'area di memoria identificato da i il valore 10. Per mezzo delle variabili potremo in seguito accedere alle aree di memoria in cui sono contenuti i dati veri e propri:

```
int j=i;
```

In questa maniera abbiamo preso il contenuto della variabile i, cioè dell'area da essa identificata, e l'abbiamo inserito in quella identificata da j. A questo punto anche j conterrà il valore 10.

3.2.1 Identificatori

Un identificatore è un nome che assegniamo ad una variabile, ed esistono naturalmente delle regole che definiscono quali nomi sono leciti, quali invece non è possibile utilizzare.
Senza scendere nel dettaglio esponendo la grammatica del linguaggio, diciamo che in generale un identificatore è una sequenza di caratteri Unicode, sia lettere che numeri, ma che deve iniziare per lettera, oppure, come nella tradizione C, con il carattere _ (underscore), o ancora con @. Ad esempio sono validi come identificatori di variabile i seguenti:

```
_identificatore1
identificatore_2
straße;
überAlles;
```

Sottolineiamo inoltre che C# è un linguaggio case-sensitive, cioè esso distingue le lettere maiuscole dalle minuscole, quindi degli identificatori come i seguenti sono ben distinti e tutti leciti:

```
Identificatore
IDENTIFICATORE
IdEnTiFiCaToRe
```

All'interno di un identificatore è anche possibile utilizzare il codice Unicode di un carattere, nella forma \uNNNN, ad esempio il carattere underscore corrisponde a \u005f. Ad esempio:

```
int \u005fIdentificatore;//equivalente a _Identificatore
```

Un'altra specie di identificatori è quella dei cosiddetti identificatori verbatim, costituiti dal carattere @, seguito da un identificatore o anche da una keyword del linguaggio. E' infatti logico che non è possibile utilizzare una keyword come identificatore, ma con l'utilizzo della @ possiamo ad esempio implementare una classe di nome @class, con un campo di nome @if o ancora @then,

```
class @class
{
   private int @if;
}
```

Tutto ciò è possibile anche se non mi sento assolutamente di consigliarlo, anzi è decisamente meglio evitarlo.
Gli identificatori verbatim possono essere utili però in una circostanza: l'architettura .NET consente una completa interoperabilità fra linguaggi diversi, dunque potrebbe capitare che un semplice identificatore utilizzato nel nostro programma, potrebbe costituire una keyword in un altro linguaggio.
Se dunque volessimo interagire con il programma C#, da questo nuovo linguaggio dovremmo necessariamente utilizzare l'identificatore preceduto da @ in modo da non farlo interpretare come parola chiave del linguaggio.

3.2.2 Inizializzazione

Non basta dichiarare una variabile, prima di usarla dobbiamo assegnarle un valore, cioè dobbiamo riempire l'area di memoria che essa identifica.
Quando assegniamo il valore alla variabile per la prima volta, questo processo si chiama inizializzazione della variabile, mentre in genere viene detto assegnazione, se stiamo dando alla variabile un nuovo valore. L'inizializzazione o l'assegnazione, si effettuano con l'utilizzo dell'operatore di assegnazione =.

```
int nRuote;//dichiarazione
nRuote=4; //inizializzazione
float fTemp=36.2;//dichiarazione ed inizializzazione
fTemp=38.5; //assegnazione di un nuovo valore
```

La variabile nRuote viene inizializzata al valore 4, la variabile fTemp invece viene prima inizializzata a 36.2 e successivamente viene ad essa assegnato il nuovo valore 38.5.

3.3 Blocco

Ogni dichiarazione deve essere univoca, cioè un identificatore deve essere distinto da ogni altro. Ma come possiamo essere sicuri che un identificatore, cioè una variabile sia veramente unica? Immaginiamo ad esempio di mettere mano ad un programma scritto da qualcun altro, a questo punto sarebbe praticamente impossibile essere sicuri dell'univocità di un identificatore. Ed è a questo punto che entra in gioco il concetto fondamentale di **blocco** di codice e dei concetti di scope e durata di una variabile.
Un blocco è una parte di codice delimitata da { e }, ed è all'interno di ogni blocco che le variabili devono avere un nome univoco.
Il concetto di **scope**, o ambito di una variabile e strettamente collegato al concetto di blocco, in quanto una variabile ha raggio d'azione che inizia dalla sua dichiarazione e termina alla fine del blocco in cui è stata dichiarata.
E' necessario sottolineare il fatto che ogni blocco può contenere dei sottoblocchi, cioè blocchi di codice annidati, all'interno dei quali le variabili dichiarate nei blocchi più esterni sono ancora in azione, cioè sono visibili.

```
Class MainClass
{
      static void Main()
      {
            int a=10; //dichiarazione della variabile a

            {
                  int b=a; //dichiaro b ed a è ancora visibile
                  System.Console.WriteLine(b);
            }//fine del blocco annidato
            //qui b non è più visibile, la riga seguente darebbe errore
            //System.Console.WriteLine(b);

            System.Console.WriteLine(a); // a è ancora attiva
      }
}
```

In blocchi diversi è dunque lecito dichiarare variabili con lo stesso nome, in quanto lo scope delle variabili è diverso, ed in questa maniera si risolve il problema dell'unicità dei nomi delle variabili.
Il codice seguente è cioè perfettamente lecito:

```
{
      int var=10;
}  //qui termina l'ambito della prima var
{
      //in questo blocco dichiaro un'altra variabile var
      int var=5;
}
```

Ma anche un blocco annidato è un blocco diverso rispetto a quello che lo contiene, eppure non è possibile dichiarare una variabile con lo stesso nome di una esterna al blocco. Supponiamo di avere il seguente codice:

```
public class EsempioScope
{
      static void Main()
      {
            int intero=0;
            while(intero<5)
            {
                  System.Console.WriteLine(intero);
                  intero=intero+1;
            }
      }
}
```

La variabile `intero` è dichiarata all'interno del blocco delimitato dal metodo Main(), all'interno del quale abbiamo un altro blocco, quello definito dall'istruzione `while`. La variabile `intero` in questo caso è visibile all'interno del `while`, cioè lo scope della variabile si estende nel blocco più interno.
Infatti se provassimo a dichiarare all'interno del `while` un'altra variabile con nome `intero`, otterremmo un'errore di compilazione del tipo:

```
error CS0136: Una variabile locale denominata "intero" non può essere dichiarata in quest'ambito
perché darebbe un significato diverso a "intero", che è già utilizzato in un ambito "padre o
corrente" per identificare qualcos'altro.
```

In generale quindi due variabili con lo stesso nome non possono essere dichiarate all'interno dello stesso blocco di codice o in blocchi che hanno stesso scope, come l'esempio visto a riguardo del `while`.
Non avviene ad esempio quello che i programmatori C++ chiamano hiding, cioè la variabile del blocco interno non nasconde (hide) la variabile dichiarata nel blocco esterno.
L'unica eccezione si ha, come vedremo nel paragrafo seguente, quando abbiamo a che fare con variabili che siano campi di classe.
Abbiamo già visto come avviene la dichiarazione di un namespace, riprendiamola per un momento:

```
namespace edmaster
{
}
```

I namespaces definiscono anch'essi un blocco, cioè uno spazio di dichiarazione. Ma in questo caso il blocco stesso è come si suol dire **open ended**, questo perché un namespace può estendere il suo spazio su più file. Supponiamo ad esempio di avere due file che definiscono due classi, Classe1 e Classe2, appartenenti allo stesso namespace `pippo`. Il namespace quindi è uno solo, e definisce un solo blocco di dichiarazione, esteso però su file diversi:

```
// classe1.cs
namespace pippo
{
      Classe1
      {
            …
      }
}
```

...

```
// classe2.cs
namespace pippo
{
      Classe2
      {
            …
      }
}
```

Il namespace pippo, inoltre, non è contenuto in nessun altro namespace. Esso, così come ad esempio altri tipi non appartenenti a nessun namespace, si dice far parte di un **global namespace** (namespace globale).
Il concetto di **durata** di una variabile è simile a quello di scope, anzi per variabili locali ad un blocco, la durata coincide con il loro scope, in quanto alla chiusura del blocco, la variabile esce dal suo scope e contemporaneamente smette di vivere. Il discorso è diverso per le variabili di classe, la cui vita coincide con quella dell'oggetto che le contiene.

3.4 Variabili di classe

Una variabile dichiarata all'interno del corpo di una classe è una variabile di classe, chiamata anche campo o variabile membro. Torneremo su questo punto quando introdurremo il concetto di classe.

Per ora, parlando dello scope delle variabili, è sufficiente notare che un campo di una classe estende il suo scope su tutta la classe, possiamo quindi usarlo in qualunque punto della classe stessa, ed anzi, non importa il punto in cui viene dichiarato il campo, esso sarà comunque utilizzabile in tutta la classe anche se la sua dichiarazione avviene come ultima istruzione della classe stessa:

```
public class Uomo
{
      //il metodo utilizza il campo fTemperatura
   //dichiarato sotto
      public void StampaTemperatura()
      {
            System.Console.WriteLine("temperatura="+fTemperatura);
      }

      static void Main()
      {
            Uomo uomo=new Uomo();
            ex.StampaTemperatura();
      }

      float fTemperatura=36.5f;
}
```

Tornando sul concetto di durata di una variabile, e ora possibile distinguerlo da quello di scope: il secondo abbiamo detto essere il blocco delimitato dalla classe, ma la variabile ha una durata pari a quella dell'oggetto di cui fa parte.

3.5 Costanti

Come il nome suggerisce le costanti sono delle variabili che non sono variabili! Cioè il valore di una costante rimane immutato lungo tutta l'esecuzione di un programma e tale valore deve essere noto a tempo di compilazione. Ciò implica che una costante viene dichiarata ed inizializzata allo stesso tempo con un valore noto:

```
const double piGreco=3.1415;
```

Le costanti inoltre sono implicitamente **`static`**, il cui significato vedremo più avanti, ed in effetti non è permesso contraddistinguere una costante anche con la parola chiave static, cosa che provocherebbe il seguente messaggio di errore del compilatore:

```
La costante [nomeCostante] non può essere contrassegnata come static.
```

3.6 Tipi di dati

Il linguaggio C# possiede due categorie di tipi, il tipo **valore** ed il tipo **riferimento** (una terza categoria è costituita dal tipo **puntatore,** utilizzabile solo in codice *unsafe*, e che va aldilà degli scopi del testo).
I tipi valore sono così chiamati perché le variabili contengono direttamente i dati, cioè dei valori, mentre le variabili di tipi riferimento contengono appunto solo un riferimento a questi dati, le vedremo comunque più da vicino quando parleremo degli oggetti.
Con i tipi di riferimento, una variabile contiene l'indirizzo in memoria dell'oggetto, è quindi possibile che due differenti variabili facciano riferimento allo stesso oggetto, e di conseguenza operando su una delle due, i cambiamenti sull'oggetto saranno riflesse anche sull'altra variabile. Consideriamo ad esempio il codice seguente, che riprende la classe Uomo di qualche paragrafo fa:

```
Uomo drJekill=new Uomo();
Uomo mrHide=drJekill;
```

Non avendo ancora introdotto il concetto di classe e di oggetti, l'esempio potrebbe risultare oscuro per chi è alle prime armi con la programmazione orientata agli oggetti. Comunque, per il momento, basta sapere che una classe è un tipo riferimento.

La variabile `drJekill` è una variabile che fa riferimento ad un nuovo oggetto di tipo `Uomo`, la variabile `mrHide`, per come è inizializzata, fa riferimento alla stessa copia di `Uomo` referenziata dalla variabile `drJekill`, in parole povere abbiamo in memoria, e più precisamente in un'area detta **managed heap**, una sola copia di un oggetto di tipo, o meglio di classe, `Uomo`:

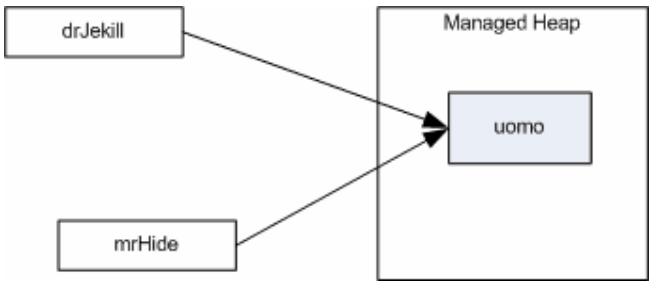


Figura 3.1 - Oggetti e riferimenti

`drJekill` e `mrHide` sono dunque lo stesso uomo!
Se una variabile di tipo riferimento è dichiarata ma non punta a nessun oggetto in memoria, si dice che essa ha valore **`null`**. Ad esempio è possibile assegnare:

```
Uomo pincoPallino=null;
```

per dire che uomo è una variabile di tipo Uomo, ma attualmente non referenzia in memoria nessun oggetto di tipo `Uomo`, come dire che pincoPallino non è nessuno!
La classe `Uomo` per come è stata implementata ha un campo `float` di nome `fTemperatura`, accessibile in questa maniera:

```
drJekill.fTemperatura=39.5;
```

Poiché anche `mrHide` referenzia lo stesso oggetto in memoria, con l'istruzione precedente, anche il campo `mrHide.fTemperaura` avrà valore 39.5. Possiamo verificarlo con il seguente programma:

```
using System;

class Uomo
{
      public float fTemperatura;
      public Uomo()
      {
            fTemperatura=36.5f;
      }
}

class TestRiferimenti
{
      static void Main()
      {
            Uomo drJekill=new Uomo();
            Uomo mrHide=drJekill;
            Console.WriteLine("temperatura di drJekill: "+drJekill.fTemperatura);
            Console.WriteLine("temperatura di mrHide: "+mrHide.fTemperatura);

            drJekill.fTemperatura=39.5f;

            Console.WriteLine("temperatura di drJekill: "+drJekill.fTemperatura);
            Console.WriteLine("temperatura di mrHide: "+mrHide.fTemperatura);
      }
}
```

Il quale, una volta eseguito, darà in output:

```
temperatura di drJekill: 36.5
temperatura di mrHide: 36.5
```

e dopo aver variato il valore di `drJekill.fTemperatura` a 39.5, anche `mrHide.fTemperatura` assumerà lo stesso valore:

```
temperatura di drJekill: 39.5
temperatura di mrHide: 39.5
```

Quanto appena detto non vale per due variabili di tipo valore, in quanto ognuna delle due variabili possiede una copia diversa dei dati, e dunque le azioni effettuate su una non influenzano i dati dell'altra variabile:

```
int a=0;
int b=a; //viene create una copia di a chiamata b, che ha valore 0
b=1; //il valore di a resta ancora 0
```

I tipi valore, o meglio, i valori delle variabili locali di un tipo valore, risiedono in un'area diversa di memoria, detta **stack**. Uno stack è una struttura usata per memorizzare dati in maniera LIFO (Last-in, First-out), vale a dire come una pila (in inglese stack) di oggetti, nella quale l'ultimo oggetto inserito sarà il primo ad essere estratto dalla pila.
Il linguaggio C# permette comunque di considerare il valore di qualsiasi tipo come un oggetto, cioè ogni tipo di C# deriva direttamente o indirettamente dalla classe `object`.

3.7 Tipi valore

I tipi valore sono così chiamati perché una variabile di un tale tipo contiene un valore del tipo stesso. Tutti i tipi valore sono discendenti direttamente dalla classe **`ValueType`**, che a sua volta deriva dalla classe **`Object`**, ma non è possibile derivare ulteriori tipi da un tipo valore. Nel linguaggio C# è possibile comunque creare nuovi tipi valore, e precisamente, di due diverse categorie:

- Tipi struct
- Tipi enumerazione

C# fornisce un insieme di tipi struct predefiniti, che è quello dei cosiddetti tipi semplici. I tipi semplici, sono identificati da keyword del linguaggio stesso, ma queste parole riservate sono solo degli alias per i tipi struct predefiniti contenuti nel namespace System, e definiti nel cosiddetto Common Type System (CTS), che, come dice il nome è il sistema di tipi comune a tutti i linguaggi che supportano e sono supportati dal .NET framework
Nella tabella seguente sono riportati i tipi semplici predefiniti del linguaggio C#, i nomi dei rispettivi tipi struct del Common Type System, ed il fatto se siano o meno tipi CLS Compliant, nel caso in cui sia necessario sfruttare l'integrazione con altri linguaggi.

Tipo semplice	Tipo CTS	CLS Compliant
sbyte	System.SByte	No
byte	System.Byte	Sì
short	System.Int16	Sì
ushort	System.UInt16	No
int	System.Int32	Sì
uint	System.UInt32	No
long	System.Int64	Sì
ulong	System.UInt64	No
char	System.Char	Sì
float	System.Single	Sì
double	System.Double	Sì
bool	System.Boolean	Sì
decimal	System.Decimal	Sì

Tabella 1 I tipi semplici predefiniti

Essendo, come detto, alias di tipi struct, ogni tipo semplice ha dei membri, ad esempio il tipo int possiede i membri del corrispettivo System.Int32 oltre a quelli della super classe System.Object, membri che vedremo più in dettaglio parlando del tipo object fra qualche paragrafo.
Se ad esempio volessimo ricavare il massimo ed il minimo valore che una variabile di un tipo numerico può assumere, potremmo utilizzare le proprietà `MaxValue` e `MinValue`, in questa maniera:

```
int maxIntero=int.MaxValue;
int minIntero=System.Int32.MinValue;
```

Come vedete è indifferente l'utilizzo dell'alias int o del nome completo System.Int32.

3.7.1 Tipi struct

Un tipo **`struct`** è molto simile ad una classe, in quanto esso può contenere campi, proprietà e metodi. La differenza sta nel fatto che le strutture sono tipi valore, quindi memorizzati sullo stack, mentre le istanze di una classe vengono creati nella memoria heap. Un'altra differenza è che le strutture non supportano il concetto di ereditarietà.
Quindi è conveniente creare strutture invece di classi quando dobbiamo definire tipi piccoli e non troppo complessi, e per i quali quindi vogliamo aumentare le prestazioni in termini di memoria. Ad esempio possiamo definire una struttura che rappresenti un punto del piano in questa maniera:

```
struct Punto
{
      public float x;
      public float y;
}
```

Sottolineo che questo è solo un esempio chiarificatore, il framework .NET contiene già un tipo Point, che è implementato proprio come struttura, ma le sue coordinate sono di tipo int.

3.7.2 Tipi enumerativi

Una enumerazione è un'insieme di valori aventi una qualche relazione, ad esempio l'enumerazione dei giorni della settimana, o dei mesi di un anno, o delle modalità di apertura di un file.
Quindi per definire un'enumerazione specifichiamo un insieme di valori che un'istanza del tipo enumerativo stesso può assumere. Ad esempio possiamo definire il tipo enumerativo dei giorni della settimana nella seguente maniera:

```
public enum Giorni
{
      Lunedì=1,
      Martedì=2,
      Mercoledì=3,
      Giovedì=4,
      Venerdì=5,
      Sabato=6,
      Domenica=7
}
```

Possiamo ora riferire i membri dell'enumerazione con l'operatore dot (cioè il punto .), la notazione `Giorni.Lunedì` ad esempio restituirà il membro di valore 1.
Possiamo inoltre ricavare anche la rappresentazione sotto forma di stringa dei membri dell'enumerazione:

```
Giorni g=Giorni.Lunedì;
string s=g.ToString(); //s vale ora "Lunedì"
```

La classe `System.Enum` fornisce dei metodi estremamente potenti ed utili per lavorare con i tipi enumerativi. Ad esempio è possibile ottenere tutti i membri dell'enumerazione, utilizzando il metodo `Enum.GetNames`, mentre per conoscerne i rispettivi valori è possibile utilizzare il metodo `Enum.GetValues`. Anticipando l'utilizzo dell'istruzione foreach ecco un piccolo esempio:

```
foreach(string s in Enum.GetNames(typeof(Giorni)))
            Console.WriteLine(s);
foreach(int i in Enum.GetValues(typeof(Giorni)))
            Console.WriteLine(i);
```

3.8 Literal

Un literal è un valore letterale di un qualunque tipo del linguaggio, ad esempio 10 è un literal di tipo intero. Solitamente il compilatore riconosce automaticamente il tipo di un literal incontrato nel programma, ma capita anche di dover risolvere delle ambiguità.
Ad esempio con 10 potremmo voler indicare il valore decimale 10, ma anche il valore esadecimale 10 corrispondente a 16 decimale, o ancora il numero double 10.0.
In tutti questi ed in altri casi è possibile utilizzare dei suffissi e dei prefissi per indicare al compilatore il tipo con cui vogliamo trattare il literal.
Per indicare un valore esadecimale si utilizza il prefisso 0x seguito dal valore esadecimale. Ad esempio 0xffff è un valore esadecimale, corrispondente a 65535 decimale, utilizzabile ad esempio nelle seguenti assegnazioni:

```
int i=0xffff;
long l=0xffff;
```

ma non nella seguente, perchè il valore 0xffff è troppo grande per essere contenuto in una variabile byte:

```
byte b=0xffff;
```

Un carattere che segue invece un valore numerico, serve ad indicare il tipo da utilizzare per trattare il valore stesso.
Una L o l indica un long, una F o f indica il tipo float, una D oppure d indica double, ed infine la U oppure u indicano un tipo unsigned. La U può essere usata sia da sola, nel qual caso indica un valore intero senza segno uint, oppure in combinazione con L, ad esempio UL ed LU indicano un valore ulong.

```
long l=10L; //il suffisso L o l indica un long
float f=10f; //f o F indica float
```

In alcuni casi il suffisso è superfluo, in altri come nel seguente è necessario:

```
float f1=10.0;//errato, impossibile convertire implicitamente il double 10.0 in float
float f2=10.0f;
//ok
```

Un numero reale può anche essere indicato utilizzando la classica notazione esponenziale:

```
double dExp=1.2e5;
double dExpNeg=120e-5;
```

il valore `dExp` è in questo caso pari a 120000, mentre `dExpNeg` sarà pari a 0.0012.
Per i tipi booleani sono naturalmente definiti i due literal true e false, che indicano rispettivamente il valore vero ed il valore falso.

3.9 Tipi riferimento

I tipi riferimento sono i tipi le cui istanze vengono create nell'area di memoria heap. Il linguaggio C# fornisce due tipi di riferimento primitivi, il tipo object ed il tipo string.

3.9.1 Tipo object

Il tipo **`object`** è il padre di tutti i tipi, primitivi e non. Tutti i tipi del framework .NET derivano dalla classe System.Object.
Come vedremo fra poco, dichiarare una variabile di tipo int, è perfettamente equivalente a dire che la variabile è di tipo System.Int32. Ogni tipo del framework .NET, e quindi di C# possiede un certo numero di metodi ereditati direttamente dalla classe System.Object, ad esempio i metodi ToString() o Equals(). Inoltre ogni variabile di qualsiasi tipo potrà essere trattato come oggetto, questo implica il notevole risultato che una variabile allocata sullo stack, potrà essere spostata nella memoria heap, con un'operazione detta di boxing, e viceversa, dallo heap allo stack, mediante l'operazione di unboxing. Torneremo alle operazioni di boxing e unboxing fra qualche pagina.

3.9.2 Tipo string

Il tipo **`string`** di C# fornisce allo sviluppatore potenti metodi per il trattamento di sequenze di caratteri. Il tipo string è un tipo riferimento, nonostante l'immediatezza nello dichiarare ed assegnare variabili stringa che farebbero pensare a prima vista ad un tipo valore. Ad esempio dichiarare ed assegnare una variabile di tipo string è fattibile in maniera perfettamente analoga al procedimento visto per una variabile intera, basta racchiudere il valore della stringa fra doppi apici ("…"):

```
string s="ciao";
```

Dietro le quinte di questa assegnazione viene creato un oggetto System.String, allocato sullo heap.
Provate ora ad immaginare cosa accade con le istruzioni seguenti, partendo dal presupposto che string è un tipo riferimento:

```
string s1="s1";
string s2=s1;
Console.WriteLine(s1);
Console.WriteLine(s2);
```

Come ci aspettiamo verranno naturalmente stampate due stringhe uguali, infatti le variabili s1 ed s2 puntano allo stesso oggetto nello heap.
Ma se a questo punto cambiamo ad esempio il valore di s2, cosa succede?

```
string s2="s2";
Console.WriteLine(s1);
Console.WriteLine(s2);
```

Essendo string un tipo riferimento ci aspetteremmo che, puntando s1 ed s2 allo stesso oggetto, il cambiamento sul contenuto di s2 si rifletta sulla stringa s1. Ed invece verranno stampati i due valori diversi "s1" ed "s2". Questo comportamento si spiega con il fatto che quando cambiamo il valore di una stringa, viene in realtà creato un oggetto completamente nuovo sullo heap, quindi la nostra variabile s1 punta ancora al valore "s1", mentre s2 punta ad un nuovo oggetto "s2", tutto ciò perché ogni stringa è una sequenza di caratteri immutabile, e dunque non è possibile modificarla, ma è necessario creare un nuovo oggetto.
La classe System.String fornisce un notevole numero di utilissimi metodi per il trattamento delle stringhe, tali metodi sono trattati in dettaglio nel capitolo a pag. 98.

3.10 Gli array

Un array è una sequenza di elementi, tutti dello stesso tipo, che può essere sia un tipo riferimento che un tipo valore. Tale sequenza è identificata da un nome, che è appunto il nome dell'array. Per dichiarare ed accedere gli elementi di un array, in C# si utilizza l'operatore di indicizzazione **[]** (parentesi quadre). Ad esempio la scrittura

```
int[] vettore;
```

identifica un array di numeri interi, ma non contiene ancora alcun elemento, nè dice quanti elementi l'array può contenere, cioè quale sia la sua lunghezza. Per inizializzare un array possono essere utilizzati diversi modi. Il primo consiste nel dichiarare e contemporanemamente assegnare gli elementi del vettore, scrivendo gli elementi dell'array racchiusi fra parentesi graffe e separati dalla virgola:

```
int[] vettore={1,2,3,4,5};
```

E' possibile inizializzare la dimensione di un array utilizzando la parola chiave new. Ad esempio se vogliamo che un array sia preparato per contenere 10 interi, ma non sappiamo ancora quali saranno i dieci valori che dovremo immagazzinare nell'array, possiamo scrivere

```
int[] vettore=new int[10];
```

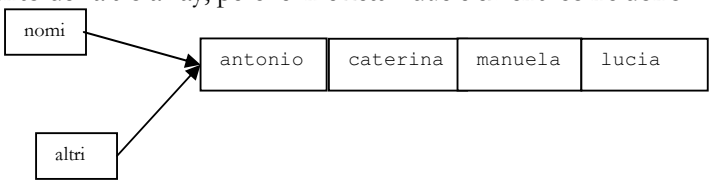
Gli elementi di un vettore sono numerati a partire da zero, così il precedente array avrà 10 elementi numerati da 0 a 9, che dunque potremo accedere singolarmente scrivendo l'indice dell'elemento da accedere al'interno delle parentesi quadre [ ], ad esempio per immagazzinare nella prima casella dell'array il valore 100, e nella seconda 101, basta scrivere:

```
vettore[0]=100;
vettore[1]=101;
```

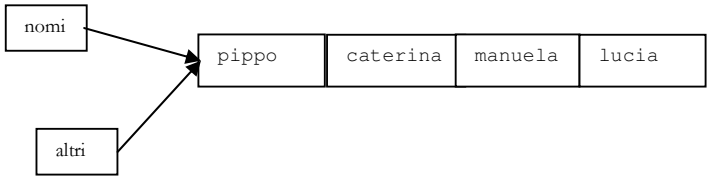
Gli array in C# sono istanze della classe System.Array. Quindi in realtà una variabile array, ad esempio la variabile vettore dell'esempio, è una istanza di un tipo riferimento. Se assegnassimo ad un'altra variabile un array,

```
int vettore2=vettore;
```

ciò che assegniamo è in realtà solo il riferimento all'array. Dunque se creiamo i due seguenti array, e poi ad esempio variamo il primo elemento di uno dei due array, la variazione si rifletterà anche sul primo elemento dell'altro array, perchè in effetti i due elementi coincidono.



Dopo l'assegnazione: Altri[0]="pippo";




```
string[] nomi={"antonio","caterina","manuela","lucia"};
string[] altri=nomi;
altri[0]= "pippo";//cambio il primo elemento dell'array altri;
Console.WriteLine(nomi[0]);//anche il primo elemento di nomi sarà uguale a "pippo";
```

La classe System.Array fornisce parecchi metodi per lavorare con gli array, per il momento anticipiamo che possiamo conoscere la lunghezza di un array usando la proprietà Length.

```
string[] stringhe={"Hello","World"};
int lunghezza=vettore.Length; // lunghezza=2
```

Notate che una volta inizializzato un array, non si può variare la sua lunghezza, di conseguenza anche la proprietà Length è di sola lettura. Qualora si avesse la necessità di utilizzare vettori di lunghezza e dimensione variabile la base class library .NET fornisce parecchie classi adatte a tale scopo, che vedremo nei prossimi capitoli.

3.10.1 Array multidimensionali

Nel paragrafo precedente abbiamo trattato solo array ad una dimensione, cioè sequenze semplici di elementi del tipo:

1	2	3	4	5

Ma è possibile definire anche array a due o più dimensioni, ad esempio per creare le cosiddette matrici:

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15

Le matrici non sono altro che più array monodimensionali disposti su più righe, dunque è necessario definire un array a due dimensioni, due perché dobbiamo tenere traccia sia del numero di riga, che del numero di colonna.
Analogamente è possibile estendere tale ragionamento e parlare di array a più di due dimensioni.

3.10.1.1 Array rettagolari

C# permette di definire due tipi di array multidimensionali. Il primo tipo è detto array **rettangolare**, ed è proprio quello che permette di definire una matrice, in quanto si tratta di più righe aventi lo stesso numero di colonne. La matrice è comunque solo l'esempio a due dimensioni di un array rettangolare.
Un array rettangolare si dichiara usando ancora le parentesi quadre dopo il tipo degli elementi, ma utilizzando una virgola per separare le dimensioni.

```
int[,] matrice=new int[3,4];
//matrice è un array a due dimensioni, la prima dimensione ha lunghezza 3, la seconda 4
int[,,] array3d;
//array3d ha tre dimensioni
array3d=new int[3,3,3];
```

Anche per accedere agli elementi dell'array si usa la stessa sintassi degli array monodimensionali, ad esempio, ricordando sempre che gli indici partono da zero, per impostare il valore del primo elemento in alto a sinistra, scriveremo:

```
matrice[0,0]=1;
array3d[0,0,0]=2;
```

E' possibile inoltre inizializzare anche gli array a più dimensioni usando le parentesi graffe, in modo annidato, per racchiudere i valori iniziali, ad esempio, la matrice di sopra a 3 righe e 5 colonne si può dichiarare ed inizializzare così

```
int[,] matrice2={{1, 2, 3,4,5},
                 {6, 7, 8,9,10},
                {11,12,13,14,15}};
```

3.10.1.2 Jagged array

Un secondo tipo di array a più dimensioni è detto **jagged array** oppure **ortogonale**. Facendo sempre l'esempio con due dimensioni, un array di tale tipo ha ancora più righe ma ognuna di esse può avere diverso numero di colonne. Supponiamo di voler implementare una struttura come quella rappresentata qui di seguito:

[1]
[2 3]
[4 5 6]

Per fare ciò non bisogna far altro che dichiarare un array di array, usando una coppia di parentesi quadre [ ] per ogni dimensione:

```
int[][] jagged;
jagged=new int[3][];//un array a due dimensioni
jagged[0]=new int[1]; // la prima riga ha lunghezza 1
jagged[1]=new int[2];//la seconda 2
jagged[2]=new int[3];//e la terza 3
```

L'accesso ad uno jagged array avviene naturalmente utilizzando la stessa sintassi vista prima, sia in lettura che in scrittura, cioè sia per ricavare il valore di un elemento che per settarlo:

```
jagged[0][0]=1;
Console.WriteLine("jagged[0][0]={0}",jagged[0][0]);
jagged[1][0]=2;
Console.WriteLine("jagged[1][0]="+jagged[1][0]);
```

Allo stesso modo si possono implementare array a più di due dimensioni, ad esempio per un array di strnghe a tre dimensioni, si può scrivere:

```
string[][][] jagStringhe;
```

3.11Conversioni di tipo

Spesso è necessario convertire una variabile da un tipo ad un altro. Ma esistono situazioni in cui ciò è possibile senza problemi e casi in cui bisogna prestare particolare attenzione, ed inoltre sono possibili diversi modi per effettuare conversioni di tipo.
La conversione viene effettuata in maniera implicita, cioè trasparente, senza nessun accorgimento particolare, quando è garantito che il valore da convertire non subirà cambiamenti, cioè perdita di precisione. Infatti abbiamo prima visto che ogni tipo è rappresentato da un certo numero di bit, quindi se ad esempio vogliamo assegnare il valore di una variabile int ad una variabile long, non corriamo alcun rischio, visto che il tipo long è sufficientemente capiente per contenere un int.
Ad esempio il codice seguente sarà compilato senza problemi:

```
int i=100;
long l=i;
```

mentre le seguenti linee non verranno compilate:

```
long l=100;
int i=l;
```

Ed il compilatore segnalerà che è

```
Impossibile convertire implicitamente il tipo "long" in "int".
```

Come caso particolare consideriamo ora il seguente:

```
const int costante=255;
byte b=costante; //ok
const int costanteBig=256;
```

```
byte b2=costanteBig; //no
```

In questo caso la prima conversione implicita è consentita perché il valore della costante è noto a tempo di esecuzione ed il suo valore è convertibile nel tipo byte. Nel secondo caso la variabile costanteBig ha valore 256, troppo grande per essere convertito in byte, dunque l'errore in fase di compilazione ci segnalerà giustamente questa situazione.

```
Impossibile convertire il valore costante "256" in "byte".
```

3.11.1 Conversioni implicite

Le conversioni implicite numeriche consentono la conversione da uno dei tipi numerici del linguaggio verso un altro tipo numerico.
La seguente tabella mostra quali sono le conversioni implicite numeriche consentite dal linguaggio C#, comunque basta notare che tali conversioni sono possibili se il tipo in cui convertire è rappresentato da un numero di bit maggiore del tipo originario.

Da	A
sbyte	short, int, long, float, double, decimal
byte	short, ushort, int, uint, long, ulong, float, double, decimal
short	int, long, float, double, decimal
char	ushort, int, uint, long, ulong, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long	float, double, decimal
ulong	float, double, decimal
float	double

Tabella 2 Le conversioni implicite

Da notare che le conversioni implicite da int, uint, long ed ulong verso il tipo float, e dai tipi long ed ulong verso double, possono provocare una perdita di precisione, ma non dell'ordine di grandezza.
Infine notiamo anche che non esiste nessuna conversione implicita verso il tipo char, ad esempio non è possibile convertire automaticamente un valore numerico in un carattere, come invece lo è ad esempio in C/C++.

3.11.2 Conversioni esplicite

Per eseguire delle conversioni non contemplate nei casi visti nel precedente paragrafo è necessario ricorrere alla cosiddetta operazione di **cast**. Con questo termine si intende la forzata conversione di un tipo in un altro. Per effettuare tale operazione si usa l'operatore di cast (vedi pag. 36), ad esempio:

```
long  l=1234567;
Console.WriteLine("l={0}",l);
int i=(int)l; //cast valido
Console.WriteLine("int i=(int)l; i={0}",i);
byte b=(byte)i; //cast non valido, il valore di i è troppo grande
Console.WriteLine("byte b=(byte)i; b={0}",b);
```

La conversione del valore long nell'int, in questo caso non darà problemi, dato il valore di l, mentre il secondo cast darà un risultato inaspettato.

3.11.3 boxing ed unboxing

Le operazioni di **boxing** ed **unboxing** consentono di convertire un qualsiasi tipo valore nel tipo object, quindi in un tipo riferimento, e viceversa. Esse consentono dunque di vedere un valore di qualsiasi tipo come un oggetto.

```
int i=123;
object box=i;
```

Nel codice sopra la variabile intera i, quindi un tipo valore, che si trova sullo stack, viene inscatolato nella variabile box, di tipo object, che quindi si troverà nell'area di memoria heap.
L'operazione automatica di boxing è quella che ci permette di considerare un tipo valore come derivato dalla classe object, e quindi di usare addirittura i suoi metodi come se fosse un oggetto.
Ad esempio è possibile ottenere la rappresentazione in formato stringa di un valore intero con il semplice utilizzo del metodo ToString della classe object:

```
int i=123;
string str=i.ToString();
```

in questo caso avviene automaticamente il boxing della variabile i in un object, del quale poi viene chiamato il metodo `ToString`.
Allo stesso modo può dunque avvenire il boxing di una struct, che è un tipo valore. Supponiamo di avere definito una struttura in questa maniera:

```
struct CartaDiCredito
{
      string numero;
      int meseScadenza;
      int annoScadenza
}
```

Il boxing di una variabile di tipo CartaDiCredito è semplicemente fattibile con l'assegnazione:

```
CartaDiCredito cc;
object carta=cc;
```

L'unboxing è l'operazione inversa e dunque permette di "scartare" da un oggetto un valore precedentemente in esso inscatolato. Ad esempio:

```
int i=123;
object box=i;//boxing
int n=(int)box;//unboxing
```

Naturalmente nell'oggetto box deve essere stato precedentemente memorizzato un intero, altrimenti verrebbe lanciata a runtime un eccezione di cast non valido.

```
int i=123;
object box=i;
float f=(float)box; //eccezione
```

L'ultimo esempio genererà un'eccezione, dato che abbiamo tentato di ottenere dall'oggetto box un float, mentre in realtà all'interno di box c'era un int.

3.11.4 La classe System.Convert

I metodi della classe **Convert**, contenuta nel namespace System ci permette di convertire un tipo di base in un altro tipo di base. In particolare i tipi di base supportati sono dai metodi della classe Convert sono i tipi Boolean, Char, SByte, Byte, Int16, Int32, Int64, UInt16, UInt32, UInt64, Single, Double, Decimal, DateTime e String.
La classe System.Convert fornisce metodi per convertire ognuno di essi in un altro, ad eccezione di alcuni casi, nei quali viene generata un'eccezione.
Ad esempio è possibile convertire una stringa lunga un carattere, come "a" nel corrispondente char 'a':

```
char a=System.Convert.ToChar("a");
```

ma se la stringa passata come argomento è più lunga di un carattere o è vuota, verrà lanciata l'eccezione di formato non valido.

```
char a=System.Convert.ToChar("aaa");
```

Grazie alla classe Convert è possibile ad esempio convertire un valore intero in un booleano, notizia positiva per i programmatori C/C++ che sono abituati a considerare automaticamente un intero nullo come false e uno diverso da zero come true.

```
int t=123;
int f=0;
Console.WriteLine("Convert.ToBoolean({0})={1}",t,Convert.ToBoolean(t));
Console.WriteLine("Convert.ToBoolean({0})={1}",f,Convert.ToBoolean(f));
```

La classe e' di grande utilità inoltre nel convertire una stringa in un valore numerico, operazione che risulta comune in molti programmi, ad esempio per leggere un valore numerico da una casella di testo, in cui è dunque contenuto come string e trasformarlo in qualche tipo numerico. Naturalmente è necessario prestare attenzione al fatto che una stringa potrebbe non essere convertibile in un qualche tipo numerico.

```
string str="12";
Console.WriteLine("str={0}",str);
short s=Convert.ToInt16(str);
int i=Convert.ToInt32(str);
long l=Convert.ToInt64(str);
double d=Convert.ToDouble(str);
byte b=Convert.ToByte(str);
float f=Convert.ToSingle(str);

Console.WriteLine("short {0}",s);
Console.WriteLine("int {0}",i);
Console.WriteLine("long {0}",l);
Console.WriteLine("double {0}",d);
Console.WriteLine("byte {0}",b);
Console.WriteLine("float {0}",f);
```

Una particolare versione del metodo Convert.ToInt32, accetta due parametri, con il primo ancora di tipo string ed il secondo un intero che specifica la base dalla quale si vuole convertire un valore. Ad esempio se vogliamo convertire il valore esadecimale "AB1" in decimale, basta chiamare il metodo così:

```
string strHex="AB1";
int dec=Convert.ToInt32(strHex,16);
Console.WriteLine(strHex+"="+dec);
```

4 Controllo di flusso

Per controllo di flusso di un programma si intende quell'insieme di funzionalità e costrutti che un linguaggio mette a disposizione per controllare l'ordine di esecuzione delle istruzioni del programma stesso. Se non potessimo controllare tale flusso, quindi, il programma sarebbe una sequenza di istruzioni, eseguite dalla prima all'ultima, nello stesso ordine in cui sono state scritte dal programmatore.

4.1 Gli operatori

Per controllare il flusso di un programma è necessario poter valutare delle espressioni. Le espressioni a loro volta sono scritte per mezzo di operatori applicati ad uno o più operandi.
Una volta valutate, le espressioni restituiscono un valore, di un determinato tipo, oppure, gli operatori possono variare il valore stesso di un operando, in questo caso si dice che essi hanno un side-effect, cioè effetto collaterale.
C# fornisce tre tipi di operatori:
-Unari: sono operatori che agiscono su un solo operando, o in notazione prefissa, cioè con l'operatore che precede l'operando, ad esempio come l'operatore di negazione -x, o in notazione postfissa, in cui l'operatore segue un operando, come ad esempio l'operatore di incremento x++.
-Binari: sono operatori che agiscono su due operandi in notazione infissa, cioè l'operatore è frapposto agli operandi, come ad esempio il classico operatore aritmetico di somma x+y.
-Ternari: sono operatori che agiscono su tre operandi. L'unico operatore ternario è ?:, anch'esso in notazione infissa, x ? y : z, e di cui vedremo più avanti nel capitolo l'utilizzo.
In un'espressione gli operatori vengono valutati utilizzando delle precise regole di precedenza e di associatività.

4.1.1 Precedenza

Quando un'espressione contiene più di un operatore è necessario ricorrere alle regole di precedenza e di associatività degli operatori che compaiono in essa.
Ad esempio nell'espressione x+y*z viene prima valutata la moltiplicazione y*z, ed il risultato di essa viene sommato al valore di x. Si dice che l'operatore * ha precedenza maggiore dell'operatore +.
Nella tabella seguente vengono mostrati gli operatori di C#, da quelli con precedenza più alta a quelli a precedenza minore, potete subito notare che gli operatori di moltiplicazione e divisione hanno precedenza maggiore rispetto alla somma ed alla sottrazione, come ci hanno insegnato alle scuole elementari.
Anche se molti di essi li incontreremo e li utilizzeremo solo più avanti nel proseguio del testo, la tabella sarà utile come veloce riferimento, quindi segnatevi pure questa pagina.

Categoria	Operatori
primari	`( )  .  [ ]  x++  x--  new  typeof  checked  unchecked`
unari	`+  -  !  ~  ++x  --x  (tipo)x`
moltiplicativi	`*  /  %`
additivi	`+  -`
shift	`>>  <<`
confronto e type-testing	`>  <  >=  <=  is  as`
uguaglianza	`==  !=`
AND logico	`&`
XOR logico	`^`

OR logico	|
AND condizionale	&&
OR condizionale	||
ternario	?:
assegnazione	= += -= *= /= %= <<= >>= &= ^= |=

Tabella 3 Precedenza degli operatori

4.1.2 Associatività

Quando nella stessa espressione ci sono più operatori con lo stesso livello di precedenza, bisogna ricorrere alle regole di associatività.
Tutti gli operatori binari, eccetto quelli di assegnazione, sono associativi a sinistra, cioè le espressioni vengono valutate da sinistra, ad esempio nell'espressione x+y+z viene prima valutata la somma x+y, ed il suo risultato è sommato a z:

x+y+z → (x+y)+z

Gli operatori di assegnazione e l'operatore ternario sono associativi a destra:

x=y=z → x=(y=z)

La precedenza è l'associatività possono essere controllate con l'uso delle parentesi, il che rende più chiaro anche l'intento del programmatore in espressioni particolarmente lunghe e complesse.

4.1.3 Operatori di assegnazione

Gli operatori di assegnazione, ad esempio il classico =, hanno lo scopo di assegnare ad una variabile posta alla sinistra dell'operatore, è quindi detta l-value (left-value), il valore della variabile o in genere dell'espressione posta alla destra dell'operatore, cioè l'r-value:

```
int n=2; //n è l'l-value, 2 è l'r-value
```

C# fornisce degli operatori di assegnazione composta, ad esempio l'istruzione

```
n=n+5;
```

che assegna ad n il suo stesso valore aumentato di 5, può essere brevemente scritta tramite l'operatore+=:

```
n+=5;
```

analogamente gli altri operatori della forma op= permettono di abbreviare le istruzioni nella seguente maniera:

```
x = x  op  y  →  x op= y
```

4.1.4 Operatori aritmetici

Gli operatori aritmetici di C# sono quelli che si incontrano nella stragrande maggioranza dei linguaggi di programmazione, vale a dire quelli di somma +, di sottrazione -, di moltiplicazione *, di divisione /, e di modulo %.
In particolare ricordiamo che l'operatore di modulo % restituisce il resto di una divisione fra interi, mentre con l'operatore / , applicato agli stessi operandi, si ottiene il valore troncato della divisione stessa:

```
int x=10;
```

```
int y=3;
int resto=x%y ; // restituisce 1
int quoz= x/y; //restituisce 3
```

4.1.5 Incremento e decremento

Un'altra scorciatoia offerta da C# è quella data dagli operatori di incremento ++ e decremento −−. Essi rispettivamente, incrementano di uno e decrementano di uno il valore dell'operando a cui vengono applicati, ad esempio le due istruzioni seguenti sono equivalenti:

```
x=x+1;
x++;
```

Ma la particolarità degli operatori di auto-incremento e auto-decremento è che essi possono essere usati sia in notazione prefissa, cioè con l'operatore che precede l'operando, sia in notazione postfissa come nell'esempio sopra.
Nel primo caso il valore dell'operando viene prima incrementato e poi viene restituito per essere utilizzato in un'espressione, nel secondo caso invece viene prima utilizzato il valore attuale, e solo dopo esso viene incrementato. Vediamo un esempio che renderà chiare le differenze nell'utilizzo delle due notazioni.

```
int i=0;
Console.WriteLine("i: {0}", i);
Console.WriteLine("++i: {0}", ++i); //Pre-incremento
Console.WriteLine("i++: {0}", i++); //Post-incremento
Console.WriteLine("i: {0}", i);
Console.WriteLine("--i: {0}", --i); //Pre-decremento
Console.WriteLine("i--: {0}", i--); //Post-decremento
Console.WriteLine("i: {0}", i);
```

Se provate ad eseguire il codice precedente, otterrete le seguenti righe di output:

```
i: 0
++i: 1
i++: 1
i: 2
--i: 1
i--: 1
i: 0
```

Nella prima riga di codice viene inizializzata la variabile i, ed il suo valore viene stampato nella seconda linea, nella terza la variabile i viene incrementata e solo dopo l'incremento, verrà usato il suo valore per stamparlo, che infatti è adesso pari a 1, nel caso del post-incremento invece viene prima stampato il valore attuale di i, ancora 1, e poi viene incrementato, infatti nella linea successiva il valore di i sarà 2. Allo stesso modo si spiegano le due successive operazioni di decremento.

4.1.6 Operatore di cast

L'operatore di cast (), si applica ad un'espressione per ottenere una conversione esplicita verso un dato tipo, ma solo quando la conversione è lecita, in caso contrario il compilatore solleverà un errore come abbiamo visto parlando delle conversioni di tipo. L'operazione di cast si effettua facendo precedere l'espressione di cui si vuol convertire il valore, dal tipo di destinazione racchiuso entro parentesi tonde, ad esempio:

```
int i=5;
byte b=(byte)i;
```

Notate che senza l'operazione di cast il compilatore ritornerebbe un errore, in quanto in genere una variabile intera non può essere contenuta in una variabile di tipo byte. Utilizzando il cast esplicito invece possiamo convertire in byte anche una variabile il cui valore è maggiore di quello normalmente ammissibile per un byte, cioè maggiore di 2^8-1=255:

```
int i=1000;
```

```
byte b=(byte)i;
```

in questo caso però il valore 1000 verrà troncato al numero di bit del tipo destinazione, cioè 8, scrivendo in binario il numero 1000 abbiamo:

bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	1	0	0	0

Quindi negli 8 bit del tipo byte avremo i primi 8 bit della variabile int, vale a dire 11101000 che in decimale corrisponde al valore 232, infatti se provate a compilare ed eseguire l'esempio seguente otterrete la conferma di quanto detto.

```
int i=1000;
byte b=(byte)i;
Console.WriteLine("i={0}, b={1}",i,b);
```

4.1.7 Operatori logici bitwise

Gli operatori logici **bitwise**, cioè bit a bit, vengono utilizzati per agire a livello di bit sui tipi interi predefiniti di C# oppure per effettuare le classiche operazioni booleane di AND, OR, e XOR fra operandi booleani.
Nel caso di operatori booleani è facile scrivere delle istruzioni che ci stampino una tabellina riassuntiva dell'algebra booleana:

```
public AlgebraBooleana()
{
      bool t=true;
      bool f=false;

      Console.WriteLine("operatore &");
      Console.WriteLine("f & f = {0}", f&f);
      Console.WriteLine("f & t = {0}", f&t);
      Console.WriteLine("t & f = {0}", t&f);
      Console.WriteLine("t & t = {0}", t&t);

      Console.WriteLine("operatore |");
      Console.WriteLine("f | f = {0}", f|f);
      Console.WriteLine("f | t = {0}", f|t);
      Console.WriteLine("t | f = {0}", t|f);
      Console.WriteLine("t | t = {0}", t|t);

      Console.WriteLine("operatore ^");
      Console.WriteLine("f ^ f = {0}", f^f);
      Console.WriteLine("f ^ t = {0}", f^t);
      Console.WriteLine("t ^ f = {0}", t^f);
      Console.WriteLine("t ^ t = {0}", t^t);
}
```

Le istruzioni precedenti stamperanno le tre tabelle delle operazioni di AND (&), OR (|) e XOR (^), secondo quanto esposto dalle seguenti:

operatore &

op1	op2	op1 & op2
true	true	true
false	true	false
true	false	false
false	false	false

operatore |

op1	op2	op1 | op2
true	true	true

false	true	true
true	false	true
false	false	false

operatore ^

op1	op2	op1 ^ op2
true	true	false
false	true	true
true	false	true
false	false	false

Nel caso di operandi integrali, gli operatori logici bitwise eseguono le operazioni booleane sui bit di pari posto dei due operandi, quindi per utilizzarli con profitto, come vedremo anche per gli operatori di Shift, bisogna conoscere un po' di numerazione binaria, oltre all'algebra booleana vista prima.
L'operatore binario & esegue un AND logico fra i bit di due operatori, vale a dire che per ogni coppia di bit restituisce un bit 1 se essi sono entrambi pari ad 1:

```
int x=5;//in binario 0000 0000 0000 0000 0000 0000 0000 0101
int y=9;//in binario 0000 0000 0000 0000 0000 0000 0000 1001
int z=x & y; // restituisce il valore 1
```

infatti l'unica coppia di bit pari a 1 è quella in posizione 0, dunque il risultato sarà un unico 1 in posizione 0 e tutti i restanti bit pari a zero, cioè tradotto in decimale, il valore 1.
Analogamente l'operatore | effettua l'OR logico degli operandi, quindi da un 1 se almeno un bit della coppia è pari ad 1:

```
z = x | y ; // restituisce 13
```

Mentre l'operatore ^ è l'operatore di OR esclusivo (o XOR) che da un 1 per ogni coppia in cui c'è uno ed un solo bit 1, cioè se entrambi I bit sono pari a 1 lo XOR darà il valore 0:

```
z = x ^ y; // restituisce 12
```

L'unico operatore bitwise unario è l'operatore ~ (tilde) di inversione, che inverte tutti i bit di un operando. Esso può essere applicato solo ad operandi numerici integrali, e nel caso in cui l'operando sia di tipo char,byte,short, esso verrà convertito in int. Facciamo un esempio con una variabile di tipo byte, quindi ad 8 bit:

```
byte b=10;//in binario 0000 1010
byte c=(byte)~b;//in 1111 1010 = 245
```

E' naturalmente necessario conoscere il sistema di numerazione binario per comprendere le conversioni da binario a decimale e viceversa degli esempi precedenti, ma con l'utilizzo degli operatori di shift possiamo anche scriverci un metodo che effettui tali conversioni per noi, oppure utilizzare la classe Convert per convertire da e in base 2.

4.1.8 Operatori di shift

Gli operatori di **shift** eseguono, come i precedenti, delle operazioni sui bit di un operando.
L'operatore << è l'operatore di shift (scorrimento) a sinistra, cioè scorre tutti i bit dell'operando a sinistra di tante posizioni quanto indicato dall'operando destro:

```
int x=1; //in binario 0000 0000 0000 0000 0000 0000 0000 0001
int y=x<<2 // 0000 0000 0000 0000 0000 0000 0100 = 4
```

L'operatore >> effettua invece lo shift a destra, inserendo a sinistra dei bit, con la cosiddetta estensione di segno: nel caso in cui il valore dell'operando da scorrere è positivo vengono inseriti dei bit 0, in caso contrario, valore negativo, vengono inseriti degli 1.

```
int x=15; //in binario 0000 0000 0000 0000 0000 0000 0000 1111
int y=x>>2; // 0000 0000 0000 0000 0000 0000 0000 0011 = 3
x=-4; // 1111 1111 1111 1111 1111 1111 1111 1100
y=x>>2; // 1111 1111 1111 1111 1111 1111 1111 1111 = -1
```

L'operando, agendo questi operatori a livello di bit, deve essere di un tipo primitivo integrale, cioè uno fra char, byte, short, int, long, e nel caso dei primi tre, cioè di tipi con un numero di bit minore del tipo int, l'operando viene convertito in int, mentre nel caso di operando long, il risultato dello shift è ancora un long.
Altri linguaggi, come Java, forniscono operatori di shift unsigned. In C# questa non costituisce una grave mancanza dato che gli stessi risultati si possono ottenere utilizzando i tipi unsigned: byte, ushort, uint, ulong.
Utilizzando gli operatori di shift e l'operatore & possiamo implementare dei metodi di conversione da binario a decimale e viceversa, in modo da poter riprovare i precedenti esempi ed avere la conferma delle operazioni a livello di bit che abbiamo esposto finora. Il funzionamento del metodo è dato come argomento di studio al lettore.

```
public string ToBinary(int n)
{
      string strBin="";
      for(int i=0;i<32;i++)
      {
            if( ((1<<i) & n) == 0)
                  strBin = "0" + strBin;
            else strBin = "1" + strBin;
      }
      return strBin;
}
```

I lettori più attenti si saranno posti almeno una domanda: se scorriamo i bit di un operando di una quantità maggiore della quantità di bit stessa? Ad esempio, se scorriamo di 33 bit, a destra o a sinistra, i 32 bit di un intero che valore assumerà tale intero?
La risposta è semplice, ma forse poco naturale: non è possibile uno scorrimento così ampio.
Infatti, ad esempio con il tipo int, dell'operando destro vengono considerati solo i 5 bit di ordine più basso, quindi al massimo è possibile scorrere 2^5=32 bit, con il tipo long invece vengono considerati i primi 6 bit, cioè si può avere uno scorrimento massimo di 64 bit.
Un altro modo di spiegare tali limiti delle operazioni di shift è il seguente. Se prendiamo il valore 33 e lo convertiamo in binario otterremo:

0000 0000 0000 0000 0000 0000 0010 0001
i primi 5 bit danno → 00001 = 1 decimale

In parole povere l'operando destro viene ridotto con un'operazione modulo 32 nel caso di int, e modulo 64 nel caso di long. Quindi volendo ad esempio scorrere un numero intero a sinistra di 33 bit, non otteniamo 0, come ci si aspetterebbe introducendo 33 bit 0, ma si ha solo uno scorrimento di 33%32=1 bit.

4.1.9 Operatori di confronto e di uguaglianza

Gli operatori di confronto sono fondamentali per controllare il flusso del programma. Essi restituiscono un valore booleano e dunque vengono utilizzati per confrontare i valori di due variabili o in generale di due espressioni. Gli operatori relazionali supportati da C# sono < (minore di), > (maggiore di), <= (minore o uguale), >= (maggiore o uguale), == (uguale a) e != (diverso da).

Il codice seguente illustra l'utilizzo di tutti gli operatori relazionali, che funzionano con tutti i tipi predefiniti del linguaggio, ad eccezione del tipo bool per il quale sono validi solo gli operatori di uguaglianza == e di disuguaglianza !=.

```
bool b;
int cento=100;
int zero=0;

b=(cento>zero);
Console.WriteLine("100 > 0 ? {0}",b);

b=(cento<zero);
Console.WriteLine("100 < 0 ? {0}",b);

b=(cento==zero);
Console.WriteLine(cento +"== " + zero +"?" + b);

b=(cento>=100);
Console.WriteLine(cento +">=100 ?" + b);

b=(cento<=100);
Console.WriteLine(cento +"<=100 ?" + b);

b=(cento!=100);
Console.WriteLine(cento +" != 100 ?" + b);

b=(cento!=0);
Console.WriteLine(cento +" != 0 ?" + b);
```

Il programma darà in output il risultato di tutti i confronti effettuati fra due variabili:

```
100 > 0 ? True
100 < 0 ? False
100== 0? False
100>=100 ? True
100<=100 ? True
100 != 100 ? False
100 != 0 ? True
```

Per il tipo predefinito string sono definiti gli operatori == e !=, che permettono di verificare l'uguaglianza o meno di due stringhe. Due stringhe sono considerate uguali se si verifica uno dei seguenti casi:

- Entrambe sono `null`
- Hanno uguale lunghezza (cioè stesso numero di caratteri) e i caratteri che le compongono sono uguali in ogni posizione.

In parole povere:

```
string uomo="antonio";
string donna="caterina";
bool b=(uomo==donna); // restituisce false
b=(uomo=="antonio"); //è true
```

Quando invece si ha a che fare con tipi riferimento in generale, quindi con le classi che noi stessi implementeremo, gli operatori di uguaglianza e disuguaglianza hanno un comportamento che potrebbe trarre in inganno chi è alle prime armi con la programmazione ad oggetti. Infatti ciò che tali operatori confrontano sono i riferimenti in memoria. Supponiamo di aver implementato una classe Automobile con un unico attributo targa.

```
class Automobile
{
public string targa;

      public Automobile(string str)
      {
            targa=str;
      }
```

```
}
```

La targa dovrebbe identificare univocamente un'automobile, quindi creando da qualche parte nel nostro codice due oggetti di classe Automobile, con lo stesso numero di targa, quello che ci aspetteremmo o quello che vorremo ottenere, confrontando i due oggetti è che i due oggetti sono uguali, cioè sono in realtà la stessa automobile.

```
Automobile auto1=new Automobile("AA 123 BB");
Automobile auto2=new Automobile("AA 123 BB");
if(auto1==auto2)
{
      Console.WriteLine("le automobili sono uguali");
}
else Console.WriteLine("le automobili sono diverse");
```

Ed invece eseguendo il codice precedente otteniamo che le due automobili sono diverse, questo perché in realtà abbiamo creato due oggetti diversi in memoria heap, e dunque con indirizzi diversi. Infatti se invece avessimo scritto

```
Automobile auto1=new Automobile("AA 123 BB");
Automobile auto3=auto1;
if(auto1==auto3)
{
      Console.WriteLine("le automobili sono uguali");
}
else Console.WriteLine("le automobili sono diverse");
```

Avremmo in questo caso due oggetti uguali o meglio due riferimenti allo stesso oggetto Automobile con targa "AA 123 BB".
Il comportamento degli operatori di uguaglianza può essere "sovraccaricato", cioè modificato in modo da restituire il risultato che lo sviluppatore vuole. Ne riparleremo comunque nel paragrafo dedicato proprio all'overloading degli operatori, a pag 70, intanto fate attenzione quando utilizzate gli operatori di uguaglianza con i tipi riferimento.

4.1.10 Assegnazione composta

Oltre ai già visti operatori di incremento e decremento, C# fornisce altri operatori che fungono da scorciatoia per delle operazioni calcolo e di assegnazione. Sono frequenti infatti istruzioni in cui ad un operando deve essere assegnato il suo stesso valore, combinato secondo qualche operazione con un secondo operando.
In generale la forma di tali assegnazioni composte (l'operatore = è detto di assegnazione semplice) è la seguente:

`x op = y` è equivalente a `x = x op y`

Dove l'operatore op può essere uno fra: `+ * / % << >> & | ^` .Quindi le istruzioni seguenti:

```
x=x+1;
x=x*10;
x=x/2;
x=x%3;
x=x<<1;
x=x>>2;
x=x&1;
x=x|2;
x=x^3;
```

possono essere riscritte nella seguente forma più compatta:

```
x+=1;
x*=10;
x/=2;
x%=3;
x<<=1;
x>>=2;
x&=1;
```

```
x|=2;
x^=3;
```

4.1.11 Operatori logici condizionali

Gli operatori && e || sono operatori detti logici condizionali o di **short circuit** (corto circuito). Essi sono la versione condizionale rispettivamente degli operatori logici bitwise & e |, con qualche differenza.
Gli operatori logici condizionali si applicano solo ad operandi che assumono valori di tipo booleano, siano essi variabili o espressioni più complesse.
L'operazione x && y corrisponde a x & y, ma nel primo caso il valore di y viene valutato solo se x è true, in quanto se esso assumesse valore false sarebbe inutile andare a valutare il valore dell'espressione y, dato che l'intera operazione di AND assumerebbe valore false in ogni caso.
Analogamente x || y è equivalente a x | y, ed in questo caso y è valutata se x è false, in quanto se x fosse true, l'intera operazione di OR restituirebbe sicuramente true, poiché è sufficiente che uno solo degli operandi dell'OR sia true.
Ecco perché si parla di operatori di corto circuito, la valutazione dell'operando destro può essere saltata, se il risultato complessivo è già definito dal primo operando, quindi l'intera operazione è "corto circuitata". Da ciò deriva anche il fatto che non può esistere un'operazione XOR short circuit, in quanto l'operazione stessa di XOR implica la necessità di valutare entrambi gli operandi.

4.1.12 Operatore ternario

L'operatore ?: è l'unico operatore ternario di C#. Esso permette di scrivere un'espressione if/else in una sola istruzione. La sintassi di utilizzo è la seguente:

```
expr1 ? expr2 : expr3
```

In base al valore assunto dall'espressione alla sinistra del ?, cioè expr1, esso restituirà uno solo dei valori degli altri due operandi, ed esattamente il valore di expr2 se expr1 è true, il valore di expr3 altrimenti.
Esso è come detto un modo di abbreviare il seguente costrutto if/else (vedi pag. 45):

```
if(expr1)
      return expr2;
else return expr3
```

Come esempio vediamo come sia possibile calcolare il massimo fra due numeri utilizzando solo l'operatore ternario:

```
int x=1;
int y=2;
int max=(x>y)?x:y;
Console.WriteLine(max);
```

Molti sviluppatori preferiscono evitare l'utilizzo dell'operatore ternario, così come molte aziende che sviluppano software ne proibiscono l'utilizzo nei propri documenti di stile a causa della presunta oscurità del codice che ne deriva. E' innegabile invece a mio modesto parere la sua eleganza ed espressività. Comunque, per tagliare la testa al toro, direi che è perlomeno necessario comprenderne l'uso e la semantica nell'eventualità di incontrarlo nel codice scritto da altri.

4.1.13 Checked ed unchecked

Con l'utilizzo degli operatori **`checked`** ed **`unchecked`** è possibile eseguire un'istruzione o un blocco di istruzioni che contengono operazioni aritmetiche, in un contesto controllato (checked) o non controllato (unchecked). Nel primo caso un overflow aritmetico genera un'eccezione, mentre nel secondo caso esso verrà ignorato.

Nell'esempio che segue l'operazione di somma viene eseguita in un contesto checked, e poiché a b1 viene assegnato il valore massimo che un byte può assumere, si avrà un'eccezione di overflow sommandovi il valore 1.

```
public void TestChecked()
{
      byte b1=Byte.MaxValue; //
      byte b2;
      try
      {
            b2=checked((byte)(b1+1));
      }
      catch(System.OverflowException oex)
      {
            Console.WriteLine(oex.Message);
      }
}
```

Se invece utilizziamo un contesto unchecked l'operazione di somma verrà eseguita ugualmente a b2 assumerà il valore 0, in quanto in tale contesto i bit che non entrano nel tipo destinazione, vengono troncati.

```
public void TestUnchecked()
{
      byte b1=Byte.MaxValue; //
      byte b2;
      b2=unchecked((byte)(b1+1));
      Console.WriteLine("b2={0}",b2);
}
```

Il controllo dell'overflow tramite gli operatori checked ed unchecked agisce sui cast tra tipi integrali e sui seguenti operatori applicati ad operandi di tipo integrale:

- Incremento ++
- Decremento --
- Negazione (unaria) -
- Somma +
- Differenza -
- Moltiplicazione *
- Divisione /

4.1.14 L'operatore Dot(.)

Negli esempi visti finora abbiamo utilizzato spesso l'operatore dot (.), soprattutto per richiamare un metodo, o per specificare il nome di una classe. Ad esempio nell'istruzione

```
System.Console.WriteLine("Hello");
```

L'operatore . è utilizzato per chiamare il metodo WriteLine della classe Console, e per specificare l'utilizzo della classe Console facente parte del namespace System. Quindi è chiaro che l'operatore . è in sintesi utilizzato per specificare quello che vogliamo trovare o richiamare, sia esso il membro di un tipo, sia esso il tipo di un dato namespace.

4.1.15 L'operatore new

L'operatore new è usato per creare nuove istanze di un dato tipo. Lo utilizzeremo spesso dunque quando avremo a che fare con le classi della BCL del framework .NET, ma anche con le classi definite da noi stessi. Per ora sottolineiamo solo i diversi modi di utilizzo dell'operatore new:

- Creazione di nuove istanze di classi o di tipi valore.

- Creazione di nuove istanze di array.
- Creazione di nuove istanze di tipi delegate.

4.1.16 Gli operatori typeof, is, as

C# fornisce degli operatori ad hoc per lavorare con i tipi. Il primo è l'operatore **typeof** che permette di ottenere un oggetto di classe System.Type per un dato tipo. La classe System.Type è la classe appunto utilizzata per rappresentare i tipi. Date un'occhiata all'esempio che segue:

```
public class TestTypeOf
{
      public TestTypeOf()
      {
            Type t1=typeof(string);
            Type t2=typeof(System.String);
            Console.WriteLine(t1.FullName);
            Console.WriteLine(t2.FullName);
      }

      static void Main()
      {
            new TestTypeOf();
      }
}
```

Eseguendo l'esempio noterete come il tipo string ed il tipo System.String sono in realtà, come già detto, lo stesso tipo, con nome completo System.String.
L'operatore **is** è utilizzato per determinare se il valore di un'espressione è compatibile in fase di esecuzione con un dato tipo. Infatti l'operatore restituisce un valore booleano, true se il valore dell'espressione può essere convertito nel tipo indicato, false altrimenti.

```
public class Test
{
public void TestIs(object o)
      {
            if(o is Uomo)
            {
                  Console.WriteLine("o è un uomo");
                  Uomo u=(Uomo)o;
            }
            else if(o is Donna)
            {
                  Console.WriteLine("o è una donna");
                  Donna d=(Donna)o;
            }
            else
            {
                  Console.WriteLine("o non è nè uomo nè donna");
                  Console.WriteLine("o è "+o.GetType());
            }
      }

      public static void Main()
      {
            Test test=new Test();

            Uomo antonio=new Uomo();
            Donna kate=new Donna();
            int n=0;
            test.TestIs(antonio);
            test.TestIs(kate);
            test.TestIs(n);
      }
}
```

Tornando a parlare di unboxing, ricorderete che se in tale operazione si tentava di estrarre da un object un valore di tipo diverso da quello effettivamente memorizzato in esso, veniva scatenata

un'eccezione di cast non valido. E' possibile tramite l'operatore is verificre che il tipo "inscatolato" in un oggetto è effettivamente del tipo voluto:

```
int i=123;
object box=i;
if(box is int)
{
      int n=(int)box;
}
```

L'operatore **as** è l'operatore di conversione incorporata, ed è utilizzato per convertire esplicitamente un valore in un dato tipo riferimento, ma a differenza di un'operazione di cast, non viene sollevata un'eccezione se la conversione non è possibile, viene invece restituito il valore null. Riprendendo l'esempio precedente, se tentassimo di convertire l'oggetto box, che sappiamo contenere un intero, in un oggetto string, otterremmo una stringa null:

```
System.String str=box as System.String;

if(str==null)
      Console.WriteLine("str è null");
```

L'operatore as fornisce dunque un modo di abbreviare il seguente blocco di codice, che fa uso di is per valutare se un oggetto è compatibile con un dato tipo, in caso positivo effettua la conversione, altrimenti restituisce null.

```
if(box is string)
      str=(string)box;
else str=null;
```

Ricordate sempre che l'operatore as è applicabile solo ai tipi riferimento, ad esempio il seguente codice provocherebbe un errore di compilazione:

```
int j=box as int;
```

in quanto int è un tipo valore.

4.2 Istruzioni di selezione

C# fornisce due costrutti di selezione, cioè due costrutti per controllare il flusso di un programma selezionando quale sarà la prossima istruzione da eseguire, selezione che si effettua in base al verificarsi di certe condizioni. Questi due costrutti, descritti nel seguito del paragrafo sono l'if/else e lo switch.

4.2.1 Il costrutto if/else

Il modo più comune per controllare il flusso di esecuzione di un programma, è quello di utilizzare il costrutto di selezione condizionale if/else. Il costrutto if/else può essere utilizzato in due maniere.
Nella prima, l'**if** è seguito da una condizione booleana racchiusa fra parentesi, e solo se essa è vera, quindi solo se assume valore true, viene eseguito il blocco successivo.

```
if( espressione_booleana )
   Blocco_istruzioni
```

Il blocco di istruzioni può essere sia una istruzione semplice seguita da un punto e virgola, oppure un blocco di istruzioni racchiuso fra parentesi graffe.
Il secondo modo è quello che prevede l'utilizzo della parola chiave **else**. Se l'espressione booleana dell'if non è vera, cioè non assume valore true, verrà eseguito il blocco o l'istruzione semplice che segue l'else.

```
if(espressione_booleana)
   ramo_if
else
   ramo_else
```

I costrutti if/else possono essere annidati, cioè all'interno di un ramo if o di un ramo else possono essere inseriti altri costrutti if o if/else. In tali casi è opportuno ricordare che l'else si riferisce sempre all'if immediatamente precedente, a meno che non si utilizzino le parentesi graffe per delimitare i blocchi. Ad esempio il seguente blocco di istruzioni, fra l'altro indentato male, potrebbe far credere che il ramo else sia riferito al primo if:

```
if(x==0)
      if(y==0) Console.WriteLine("ramo if");
else Console.WriteLine("ramo else");
```

invece tale codice è equivalente al seguente, in quanto l'else si riferisce come detto prima, all'if immediatamente precedente:

```
if(x==0){
      if(y==0){
            Console.WriteLine("ramo if");
      }
      else{
            Console.WriteLine("ramo else");
      }
}
```

In C++ è possibile utilizzare come condizione dell'if un'espressione di assegnazione, ad es. if(x=0), in quanto in tale linguaggio è possibile usare come condizione il fatto che una variabile intera sia nulla o meno. Ciò spesso porta ad errori di codice difficilmente individuabili, e questa possibilità è evitata in C#, in quanto la condizione dell'if deve per forza essere valutabile come valore booleano, cioè true o false. Se provaste infatti a compilare un'istruzione come

```
if(x=0)
{
   //…
}
```

otterreste l'errore di compilazione "Impossibile convertire int in bool", e ciò ci suggerisce che la condizione deve restituire necessariamente un valore di tipo bool.
Riprendendo il discorso dello short circuit visto con gli operatori condizionali, possiamo notare, tramite l'uso del costrutto if, cosa implica il cortocircuitarsi delle espressioni booleane contenute nella condizione dell'if stesso. Supponiamo di scrivere un metodo per testare se una variabile sia o meno positiva:

```
public bool IsPositive(int x)
{
      Console.WriteLine("Test di "+x);
      return x>0;
}
```

Utilizziamo tale metodo per valutare la condizione che tre interi siano tutti positivi, poniamo tale condizione all'interno di un if, possiamo verificare che nel caso che segue, cioè nel caso in cui utilizziamo l'operatore bitwise &, il metodo IsPositive viene chiamato tre volte, nonostante sia chiaro già dal primo test sulla x, che la condizione dell'if sarà falsa.

```
public void NoShortCircuit()
{
      int x=-1;
      int y=1;
      int z=2;
      if(IsPositive(x) & IsPositive(y) & IsPositive(z))
      {
            Console.WriteLine("Tutti positivi");
      }
```

```
}
```

Utilizzando invece l'operatore di short circuit &&, il metodo IsPositive() verrà chiamato solo una volta:

```
public void ShortCircuit()
{
      int x=-1;
      int y=1;
      int z=2;
      if(IsPositive(x) && IsPositive(y) && IsPositive(z))
      {
            Console.WriteLine("Tutti positivi");
      }
}
```

In quanto già la chiamata con parametro x darà risultato false e non è necessario valutare y e z.

4.2.2 Il costrutto switch

Se è necessario effettuare una scelta fra diversi alternativi percorsi di esecuzione, e tale scelta può essere basata sul valore assunto a runtime da una variabile di un tipo integrale o string, allora è possibile usare il costrutto **`switch`**.
La sintassi dello switch è la seguente:

```
switch(espressione)
{
   case costante1:
      blocco_istruzioni
      istruzione_salto;
   case costante2:
      blocco_istruzioni
      istruzione_salto;
   …
   [default:blocco_istruzioni]
}
```

L'espressione all'interno delle parentesi viene valutata, e se il suo valore corrisponde ad una delle costanti dei **`case`**, il flusso del programma salta al case corrispondente, del quale viene eseguito il blocco di istruzioni, fino a raggiungere una necessaria istruzione di salto. Se è presente una label **`default`**, e la valutazione dell'espressione produce un risultato non contemplato dalle costanti dei case, il flusso va alla label di default, se tale label invece non è presente il flusso va alla fine dello switch.
Tipicamente l'istruzione di salto che chiude un blocco case è l'istruzione **`break`**, che fa uscire il programma dal costrutto switch. Ma è possibile utilizzare anche una istruzione di salto qualunque (vedi pag 51).
In C#, a differenza di altri linguaggi come C++ e Java, non è possibile omettere l'istruzione di salto, cioè non deve essere possibile, in uno switch, passare da una sezione case alla seguente, questa è nota come regola "no fall through". In pratica tale regola impedisce di scrivere uno switch come il seguente:

```
switch(x)
{
      case 0:
            Console.WriteLine("x è nulla");
      case 1:
            Console.WriteLine("x è 0 oppure 1");
            break;
      default:
            Console.WriteLine("x è diverso da 0 e 1");
            break;
}
```

La regola "no fall through" evita la possibilità del verificarsi bug dovuti ad una dimenticanza di un break. Se però dovesse essere necessario ottenere il meccanismo vietato dalla regola, si può ricorrere

ad uno dei pochi usi sensati della keyword goto. L'esempio precedente compila correttamente se scritto nel seguente modo, in cui potete anche notare la posizione dell'etichetta default:

```
switch(x)
{
      default:
            Console.WriteLine("x è diverso da 0 e 1");
            break;
      case 0:
            Console.WriteLine("x è nulla");
            goto case 1;
      case 1:
            Console.WriteLine("x è 0 oppure 1");
            break;
      case 2:
            goto default;
}
```

Inoltre, se vogliamo raggruppare più etichette che devono avere lo stesso risultato possiamo scriverle una dopo l'altra, senza istruzioni fra esse, in tal maniera non viene violata la regola precedente:

```
switch(c)
{
      case 'a':
      case 'e':
      case 'i':
      case 'o':
      case 'u':
            Console.WriteLine("c è una vocale");
            break;
      default:
            Console.WriteLine("c non è una vocale");
            break;
}
```

Come già accennato, l'espressione da valutare in uno switch, può essere di un tipo a scelta fra byte, sbyte, short, ushort, int, uint, long, ulong, char oppure string. Tale tipo è detto tipo governante dello switch.
In ognuno di tali casi, le etichette dei case devono essere valori costanti e distinti di tipo uguale al tipo governante, o di un tipo convertibile in esso. Nel caso in cui il tipo governante sia invece string, è possibile utilizzare in un'etichetta case il valore null:

```
void GestioneComando(string cmd)
{
      case "apri":
            Console.WriteLine("Apri");
            Apri();
            brak:
      case "salva":
            Console.WriteLine("Salva"):
            Salva();
      case null:
            Console.WriteLine("comando non valido");
}
```

Con un metodo come quello dell'esempio precedente è possibile implementare facilmente un classico menù a riga di comando, cosa che potete svolgere come esercizio.

4.3 Istruzioni di iterazione

Le istruzioni di iterazione hanno lo scopo di eseguire ripetutamente un determinato blocco di istruzioni, formato eventualmente anche da una sola istruzione. Esse dunque permettono di eseguire più cicli attraverso uno stesso blocco di codice, un tipico esempio è quello in cui si esegue un ciclo per leggere o scrivere i diversi elementi di un vettore.

4.3.1 Il ciclo while

Il costrutto **while** permette di eseguire un blocco di istruzioni zero o più volte in base al valore di un'espressione booleana, secondo la seguente sintassi:

```
while(espressione_booleana)
   blocco_istruzioni
```

Fino a quando l'espressione booleana restituisce valore true, il blocco istruzioni verrà eseguito. E' dunque necessario, per uscire dal ciclo while, che l'espressione ad un certo punto diventi false, oppure che all'interno del ciclo while ci sia un'istruzione break, che fa saltare l'esecuzione alla fine del ciclo while, cioè all'istruzione immediatamente successiva, o comunque al di fuori di esso:

```
int i=0;
while(i<10)
{
      Console.WriteLine{"i={0}",i);
      i++;
}
```

L'esempio precedente esegue il blocco fra parentesi graffe fino a che il valore della variabile i non sia pari a 10, per tale valore infatti l'espressione i<10 diventa false.
Come detto prima è possibile anche terminare il ciclo con un break, dunque anche da un ciclo while la cui espressione di valutazione sia sempre true, come nel caso seguente, è possibile uscire.

```
int i=0;
while(true)
{
      Console.WriteLine{"i={0}",i);
      i++;
      if(i=10) break;
}
```

Non sono rari i casi in cui, a causa di un errore di programmazione, il programma entra in un ciclo infinito, e come conseguenza sembra bloccato nella sua esecuzione. Assicuratevi dunque che da un ciclo ad un certo punto si esca, e magari date qualche feedback all'utente nel caso di cicli particolarmente lunghi, ad esempio con una ProgressBar.
E' possibile utilizzare all'interno del ciclo while un'istruzione di salto **continue** (o un'altra istruzione di salto, vedi pag 51), con la quale il blocco rimanente di istruzioni viene saltato e si ricomincia un nuovo ciclo rivalutando l'espressione booleana.

```
int i=0;
while(i++<10)
{
      if(i%2==0)
            continue;
      Console.WriteLine("i={0}",i);
}
```

L'esempio precedente stampa il valore di i se esso è dispari, altrimenti incrementa il valore della variabile i e ricomincia il ciclo.
Notate che se l'espressione booleana è false in partenza, il blocco di istruzioni del ciclo while non verrà mai eseguito. Ciò costituisce la differenza principale con il ciclo do esposto nel successivo paragrafo.

4.3.2 Il ciclo do

Il ciclo **do** è analogo al precedente, con l'unica differenza che il blocco di istruzioni incorporato viene eseguito almeno una volta, in quanto l'espressione booleana viene valutata alla fine del primo ciclo. A questo punto, se l'espressione assume valore true, si esegue un nuovo ciclo, altrimenti si esce da esso.

```
int i=10;
do{
      Console.WriteLine("i={0}",i++);
```

```
}while(i<10);
```

Il valore di i è 10 in partenza , ciò nonostante il ciclo viene eseguito ugualmente e stampa il valore di i.

4.3.3 Il ciclo for

I due cicli precedenti, come visto d'altronde negli esempi, permettono di eseguire delle istruzioni incrementando di volta in volta il valore di una variabile. Tale situazione è così frequente in un programma, che la maggior parte dei linguaggi di programmazione, e fra questi naturalmente C#, fornisce una keyword dedicata allo scopo, l'istruzione **`for`**.
Il ciclo for ha la seguente sintassi generale:

```
for(inizializzazione; espressione_booleana; incremento)
   blocco_istruzioni
```

All'interno delle parentesi tonde che seguono la parola chiave for, vengono dunque eseguite tre diverse operazioni, tutte quante comunque facoltative.
Come prima operazione viene eseguita un'inizializzazione, in genere di una variabile contatore di ciclo, quindi viene valutata una condizione booleana, e se essa assume valore true viene eseguito il blocco di istruzioni. Alla fine di ogni iterazione viene eseguita la terza operazione, in genere un'incremento della variabile contatore. Quando l'espressione booleana diviene false, l'esecuzione del programma salta alla istruzione che segue il ciclo for.
Il ciclo for viene in genere impiegato per eseguire cicli che hanno un numero prestabilito di passi, ad esempio iterare lungo ogni elemento di un vettore la cui lunghezza è nota o comunque calcolabile prima del ciclo:

```
int[] vettore=new int[10];
for(int i=0;i<10;i++)
{
      vettore[i]=i;
      Console.WriteLine("vettore[{0}]={1}",i,vettore[i]);
}
```

Tramite l'operatore virgola (,) è possibile inoltre inizializzare ed incrementare diverse variabili all'interno di un ciclo for, con la limitazione che tutte abbiano uguale tipo. Il metodo seguente crea una matrice identica di dimensione pari al parametro dim, cioè una matrice con tutti gli elementi nulli, tranne quelli sulla diagonale che saranno pari a 1;

```
public int[,] CreaMatriceIdentita(int dim)
{
      int[,] matrice=new int[dim,dim];
      for(int i=0,j=0;i<dim && j<dim;i++,j++)
      {
            matrice[i,j]=(i==j)? 1: 0;
      }
      return matrice;
}
```

E' naturalmente possibile innestare più cicli for, ad esempio per trattare con le matrici è naturale innestarne due, ognuno dei quali itera uno dei due indici delle matrici. Il seguente esempio esegue la somma di due matrici identità, create con il precedente metodo:

```
int[,] matriceA=CreaMatriceIdentita(3);
int[,] matriceB=CreaMatriceIdentita(3)

int[,] matriceC=new int[3,3];
for(int i=0;i<3;i++)
{
      for(int j=0;j<3;j++)
      {
            matriceC[i,j]=matriceA[i,j]+matriceB[i,j];
      }
}
```

Come già detto le istruzioni fra le parentesi tonde del for sono tutte quante facoltative. Ad esempio è possibile implementare un ciclo infinito analogo ad un while(true){…}, scrivendo un ciclo for come il seguente:

```
for(;;)
{
   faQualcosa();
   break;
}
```

4.4 L'istruzione foreach

C# fornisce un'istruzione che permette di scorrere gli elementi di collezioni o array senza far uso di espressioni booleane, ma in maniera più naturale, scorrendo gli elementi in modo sequenziale. Tale istruzione è **`foreach`**. La sintassi generale dell'istruzione foreach è la seguente:

```
foreach(tipo elem in collezione)
   blocco_istruzioni
```

Come esempio scriviamo il codice per la stampa degli elementi di una matrice, utilizzando un for e poi con un foreach:

```
int[,] matrice=new int[10,10];

for(int i=0;i<10;i++)
{
      for(int j=0;j<10;j++)
      {
            Console.WriteLine(matrice[i,j]);
      }
}

foreach(int elemento in matrice)
{
      Console.WriteLine(elemento);
}
```

Come potete notare l'istruzione foreach fornisce maggiore chiarezza al codice, soprattutto quando si ha a che fare con Collection un po' più complesse rispetto ad un semplice array di interi.
Il rovescio della medaglia si ha nel fatto che l'istruzione foreach fornisce un accesso agli elementi della collezione in sola lettura, cioè nel suo blocco di istruzioni non è possibile modificare l'elemento a cui si sta accedendo. Ad esempio è errato il seguente ciclo foreach:

```
foreach(int elemento in matrice)
{
   elemento=0;
}
```

Se provaste a compilarlo otterreste il messaggio di errore "`Cannot assign to 'elemento' because it is read-only`".

4.5 Istruzioni di salto

Le istruzioni di salto permettono di trasferire il controllo dal punto in cui esse vengono eseguite ad un altro punto del programma. Il punto di arrivo è detto target del salto.

4.5.1 L'istruzione break

Abbiamo già incontrato l'istruzione **`break`** nei paragrafi precedenti. Il break serve a far uscire il programma da uno dei blocchi di istruzione dei costrutti switch, while, do, for o foreach, spostando l'esecuzione all'istruzione immediatamente successiva al costrutto.
Abbiamo già visto ad esempio un codice come il seguente:

```
int i=0;
while(true)
{
      if(i==10)
            break; //se i è 10 esce dal ciclo
      else Console.WriteLine(i++);
}
```

Nel caso in cui ci siano più blocchi innestati, ad esempio due cicli for, l'istruzione break si riferisce solo al ciclo in cui è presente, e fa dunque saltare l'esecuzione alla fine di tale ciclo. Per far uscire il programma ad un ciclo più esterno è possibile usare l'istruzione di salto incondizionato goto.

```
int i=0;
for(int i=0;i<10;i++)
{
      for(j=0;j<10;j++)
      {
            if(j==i)
                  break; //se i==j esce dal for interno e va al WriteLine seguente
      }
      Console.WriteLine("for1");
}
```

4.5.2 L'istruzione continue

L'istruzione **`continue`** piuttosto che terminare il ciclo in cui si trova, ciclo che può essere uno qualunque fra while, do, for e foreach, fa in modo di iniziare immediatamente una nuova iterazione, saltando la parte di ciclo che si trova dopo l'istruzione stessa. Riprendiamo l'esempio già visto nel paragrafo dedicato al while:

```
while(i++<10)
{
      if(i%2==0)
            continue;
      Console.WriteLine("i={0}",i);
}
```

Come per l'istruzione break, l'istruzione continue si riferisce sempre al ciclo in cui l'istruzione continue stessa è richiamata, con la possibilità ancora di usare il goto per saltare più di un livello di annidamento.

4.5.3 L'istruzione return

L'istruzione **`return`** è un'istruzione di salto che restituisce immediatamente il controllo al metodo che sta chiamando quello in cui si trova l'istruzione return stessa.

```
public void Primo()
{
      Secondo();
      //punto di ritorno
}
…

public void Secondo()
{
      //fa qualcosa
      if(condizione)
            return;
      //fa qualcos'altro
}
```

Nell'esempio il metodo `Primo` chiama il metodo `Secondo`, nel quale se si verifica una certa condizione, il controllo viene passato di nuovo a Primo, esattamente nel punto successivo alla chiamata `Secondo`.
La stessa cosa avviene nel caso in cui si raggiunga la fine del metodo `Secondo`.
Nel caso in cui il metodo restituisca un qualche valore è necessario utilizzare l'istruzione return seguita da una espressione che restituisce un valore del tipo dichiarato dal metodo stesso.

Ad esempio per restituire un valore intero 1 da un metodo come il seguente, basta utilizzare l'istruzione return 1:

```
public int RestituisciUno()
{
   return 1;
}
```

4.5.4 Goto

C# mette a disposizione la famigerata istruzione di salto incondizionato, l'istruzione **goto**!
Inizio subito col dire che esistono teoremi che provano il fatto che ogni programma è codificabile senza fare uso del goto o chi per lui (vedi Jacopini-Bohem), e che illustri autori ne deplorano l'utilizzo se non addirittura l'esistenza stessa (vedi Tanenbaum). Partendo da tali presupposti, con cui non posso che essere d'accordo, espongo comunque l'utilizzo dell'istruzione goto in una delle rare occasioni, se non l'unica, in cui può rivelarsi utile.
Nell'istruzione di selezione switch, a differenza di C++, in C# non è supportato il passaggio esplicito da un'etichetta case a un'altra. Se necessario, sarà possibile utilizzare goto per passare ad un'altra etichetta case dell'istruzione switch, utilizzando il goto con la sintassi goto case x oppure con goto default:

```
switch(x)
{
      case 1:
            Console.WriteLine("x>");
            goto case 0;
      case 0:
            Console.WriteLine("=0");
            break;
      default:
            Console.WriteLine("valore non valido");
            break;
}
```

C# non permette il salto verso un'etichetta posta in un blocco più interno di quello corrente, in quanto la leggibilità del codice ne risulterebbe senza dubbio compromessa.
Ad esempio non è possibile scrivere un codice come:

```
goto label1;
if(i==0)
{
      label1:
      Console.WriteLine("Label1");
}
```

E d'altronde la label1 non è nemmeno visibile all'istruzione goto in quanto si trova in un blocco più interno, cioè ha diverso scope.

5 Programmazione ad oggetti

Questo capitolo intende fornire una rapida introduzione al paradigma di programmazione Object Oriented, dando una rapida infarinatura dei concetti principali, e soffermandosi su come essi sono interpretati ed implementati nel linguaggio C#.

5.1 Oggetti e classi

Cos'è un oggetto? Sfogliando le prime pagine della quasi totalità dei testi dedicati alla OOP (Object Oriented Programming), una frase tipica che vi si legge è che tutto è un oggetto. Citando una definizione più seria e precisa data da uno dei padri del paradigma stesso, Grady Booch, possiamo dire che un oggetto è un qualcosa che ha un suo stato, una sua identità, ed un suo comportamento. Dal punto di vista dello sviluppatore software dunque, un oggetto è un qualcosa che mantiene dei dati interni, che fornisce dei metodi per manipolarli, e che risiede in una propria riservata zona di memoria, convivendo con altri oggetti, dello stesso tipo o di altro tipo.
Nel paradigma Object Oriented, in generale, il tipo di un oggetto è la sua classe, cioè ogni oggetto appartiene ad una determinata classe. Ad esempio un cane di nome Argo, che ha una propria identità, un proprio comportamento, ed uno stato (ad esempio il peso, l'altezza, la data di nascita), è un elemento di una classe, la classe Cane.
Il concetto di classe è sinonimo dunque di tipo. Ogni classe definisce le caratteristiche comuni di ogni oggetto che vi appartiene. Ad esempio tutti gli elementi della classe Cane, hanno quattro zampe, abbaiano, camminano. Ma ogni oggetto avrà poi delle caratteristiche che lo distinguono da ogni altro.
Ogni elemento di una classe, cioè ogni oggetto, si dice essere istanza di quella classe, e la creazione stessa di un oggetto viene anche detta, quindi, istanziazione dell'oggetto.
Tornando a parlare di sviluppo di software, quando programmiamo in maniera orientata agli oggetti, non dobbiamo fare altro che pensare ai concetti del dominio che stiamo affrontando, e ricavarne dunque gli oggetti e le funzionalità che essi devono avere, e fornire le modalità di comunicazione fra oggetti diversi della classe stessa o di classi differenti.
Inoltre ogni oggetto, nel mondo comune, non è un entità atomica, cioè ogni oggetto è formato da altri oggetti, il classico mouse ad esempio è formato da una pallina, da due o più tasti, magari da una rotellina, tutti oggetti che aggregati forniscono la funzionalità totale del mouse.

5.2 C# orientato agli oggetti

C# è un linguaggio totalmente orientato agli oggetti, cioè quando programmiamo in C# dobbiamo pensare in termini di classi e oggetti, tanto che anche per il più semplice dei programmi, come l'hello world visto qualche capitolo fa, è necessario creare una classe, che verrà istanziata dal CLR per eseguire il programma, invocando il metodo Main che sarà esposto dall'oggetto creato.

5.3 Le classi

In un linguaggio orientato agli oggetti, creare nuovi tipi di dati vuol dire creare nuove classi. Per far ciò C# fornisce la parola chiave **`class`**.
Secondo la definizione, o perlomeno una delle tante definizioni, una classe è una struttura di dati che può contenere membri dati (costanti e campi), membri funzione (metodi, proprietà, eventi, indicizzatori, operatori, costruttori di istanza, distruttori e costruttori statici) e tipi nidificati.
Supponiamo di voler creare la classe Veicolo, che rappresenti un veicolo terrestre qualunque. Quello che occorre fare innanzitutto, è scrivere il codice per definire la classe stessa, ed i suoi membri. Per membri della classe si intendono i suoi campi, che mantengono lo stato dell'oggetto, ed i suoi metodi, che definiscono il suo comportamento.

Per definire una classe in C# si utilizza questa sintassi generale:

```
[modificatore] class NomeClasse [:ClasseBase]
{
   [membri della classe]
}
```

Con tale sintassi viene creato un nuovo tipo o meglio la sua rappresentazione, mentre per creare un'istanza di esso, cioè un oggetto di classe NomeClasse si utilizza la keyword **new**:

```
NomeClasse oggetto=new NomeClasse();
```

I modificatori sono descritti nel prossimo paragrafo, per ora utilizzate pure il modificatore public per le vostre prime classi.
Il significato di classe base verrà invece illustrato parlando dell'ereditarietà.
La definizione dei membri della classe avviene all'interno delle parentesi graffe, che delimitano dunque il corpo della classe stessa.

5.3.1 Modificatori di accesso

Fra i modificatori applicabili alla dichiarazione di una classe, introduciamo per ora i modificatori d'accesso public ed internal, rinviando altri modificatori (new, sealed, abstract) al proseguio del testo.
Una dichiarazione di classe in generale può essere dichiarata con uno dei seguenti modificatori:

Modificatore d'accesso	Descrizione
`internal`	La classe è accessibile solo a classi dello stesso assembly, questo è il livello di visibilità predefinito.
`public`	Classe visibile anche al di fuori dell'assembly contentente la classe stessa.

La maggior parte delle classi viene dichiarata come public, quindi visibile anche in assembly diversi da quello di appartenenza. Se si omette il modificatore di accesso, verrà sottinteso che esso sia internal, dunque che la classe è visibile solo all'interno del suo stesso assembly.
Notate che non è possibile in genere utilizzare il modificatore `private` ad una classe o ad elementi che sono contenuti direttamente in un namespace, perchè non avrebbe senso in quanto in questa maniera la classe non sarebbe utilizzabile da nessuno, mentre è possibile definire una classe privata se innestata in un'altra come vedremo fra qualche riga.
Altri modificatori permettono di stabilire delle regole di accesso ai membri di una classe. Introduciamo per il momento solo i modificatori public e private, mentre parleremo degli altri modificatori possibili, quando avremo a che fare con il concetto di ereditarietà.
Il modificatore **private** applicato ad un membro di una classe, ad esempio ad un suo campo, indica che tale membro è visibile ed utilizzabile solo da istanze della classe stessa, mentre il suo accesso sarà impedito ad oggetti di classi diverse. Il modificatore **public** invece permette l'accesso da oggetti di qualunque altra classe. e scrivessimo una classe come la seguente

```
class MiaClasse
{
      private int unValore;
}
```

probabilmente non avrebbe nessun utilizzo, visto che il campo intValore è private, e quindi non accessibile dall'esterno. Per questo è necessario aggiungere ad esempio un metodo pubblico, richiamabile dall'esterno, che magari restituisca il valore del campo, sempre se esso debba essere utilizzato dall'esterno:

```
class MiaClasse
```

```
{
      private int unValore;

      public int GetValore()
      {
            return unValore;
      }
}
```

5.3.2 Classi nested

In C# è possibile innestare le dichiarazioni di classe, vale a dire dichiarare una classe internamente al corpo di un'altra, in tal caso, la classe innestata è in tutto e per tutto un membro della classe in cui è dichiarata, e dunque è possibile applicare ad essa i modificatori di membro.

```
class MiaClasse
{
      public MiaClasse()
      {

      }

      //classe innestata privata
      private class ClassePrivata
      {
            //…
      }

      //classe innestata pubblica
      public class ClassePubblica
      {
            …
      }
}

class TestClass
{
      public static void Main()
      {
            MiaClasse obj=new MiaClasse();
            MiaClasse.ClassePubblica obj2=new MiaClasse.ClassePubblica();
            //la seguente linea darebbe errore
            //MiaClasse.ClassePrivata obj2=new MiaClasse.ClassePrivata();
      }
}
```

Nell'esempio precedente la classe MiaClasse contiene al suo interno due dichiarazioni di classe, una private ed una public. La prima è utilizzabile solo all'interno della classe MiaClasse stessa, mentre la seconda è istanziabile anche all'esterno, come mostrato nel metodo Main della classe TestClass.

5.3.3 Campi di classe

I campi di un classe rappresentano i membri che contengono i dati di un'istanza della classe stessa. Un campo può essere di un tipo qualunque, può cioè essere un oggetto o anche un campo di tipo valore.

```
public class Veicolo
{
      public int ruote;
      public float velocita;
      public int direzione=90;
      private bool motoreAcceso=false;
      private AutoRadio radio;
}
```

La classe Veicolo così scritta possiede uno stato interno dato dal numero di ruote del veicolo, dalla velocità, dalla direzione di movimento e dallo stato del motore, che sono dunque dei dati rappresentati per mezzo di tipi valore, ed inoltre possiede un campo di classe AutoRadio, cioè un tipo riferimento che per ora supponiamo di aver implementato da qualche altra parte.

Come abbiamo visto prima, per istanziare un oggetto è necessario utilizzare la parola chiave new. Essa è seguita da un particolare metodo della classe, chiamato costruttore. Anche quando esso non viene esplicitamente definito, ogni classe ne fornisce uno standard, detto appunto costruttore di default, e che non accetta nessun parametro in ingresso. Quindi sebbene per la classe veicolo non abbiamo ancora pensato ad un costruttore, possiamo ugualmente istanziare un oggetto della classe Veicolo:

```
Veicolo auto=new Veicolo();
```

L'oggetto `auto` possiede i tre campi ruote, velocita e direzione, che possiamo utilizzare in lettura ed in scrittura perché li abbiamo dichiarati public, quindi visibili dall'esterno. Per accedere ai membri di una classe si usa l'operatore dot, cioè il punto, dunque se volessimo ad esempio impostare a 4 il numero di ruote dell'oggetto auto, scriveremmo:

```
auto.ruote=4;
```

Infatti i campi di una classe assumono determinati valori di default, se non vengono esplicitamente inizializzati. Questo avviene per assicurarsi che tali campi posseggano comunque un valore dopo la creazione di un oggetto. E' possibile comunque inizializzare i campi di una classe anche contemporaneamente alla dichiarazione, ad esempio il campo booleano `motoreAcceso`, viene inizializzato a `false` esplicitamente, mentre il campo `direzione` è stato inizializzato al valore 90.
La tabella seguente illustra quali valori vengono assunti dalle variabili di campo dei vari tipi primitivi di C#.

Tipo del campo	Valore di default
byte, sbyte	(byte)0
short, ushort	(short)0
int, uint	0
long, ulong	0L
float	0.0F
double	0.0D
char	'\u0000' (carattere null)
bool	false
decimal	0
string	"" (stringa vuota)
tipi riferimento	null

Possiamo ricavare una conferma dei valori di default scrivendo una classe che possegga campi di tutti i tipi e stampandone i valori, senza averli inizializzati esplicitamente:

```
class DefaultVal
{
      public short s;
      public ushort us;
      public byte b;
      public sbyte sb;
      public int i;
      public uint ui;
      public long l;
      public ulong ul;
      public float f;
      public double d;
      public char c;
      public bool bo;
      public decimal dec;
      public string str;
      public object obj;

      static void Main(string[] args)
      {
            DefaultVal test=new DefaultVal();
            Console.WriteLine("short={0}",test.s);
            Console.WriteLine("ushort={0}",test.us);
```

```
            Console.WriteLine("byte={0}",test.b);
            Console.WriteLine("sbyte={0}",test.sb);
            Console.WriteLine("int={0}",test.i);
            Console.WriteLine("uint={0}",test.ui);
            Console.WriteLine("long={0}",test.l);
            Console.WriteLine("ulong={0}",test.ul);
            Console.WriteLine("float={0}",test.f);
            Console.WriteLine("double={0}",test.d);
            if(test.c=='\u0000')
                  Console.WriteLine("char='\u0000'(nullo)");
            Console.WriteLine("bool={0}",test.bo);
            Console.WriteLine("decimal={0}",test.dec);
            Console.WriteLine("string=\"{0}\"",test.str);
            if(test.obj==null)
                  Console.WriteLine("object=null");
      }
}
```

E' bene sottolineare che i valori di default dei tipi, vengono applicati esclusivamente ai campi di una classe, mentre non valgono, come abbiamo già avuto modo di dire, per le variabili locali, le quali devono essere inizializzate prima del loro uso.

5.3.3.1 Campi costanti

E' possibile che il valore di un campo di una classe debba rimanere costante durante tutta l'esecuzione del programma, ad esempio è possibile fissare la lunghezza che deve assumere la targa del nostro veicolo. Per creare una costante si utilizza la parola chiave `const`:

```
class Veicolo
{
      private const LUNGHEZZA_TARGA=7;
      private string targa;
      //… resto della classe
}
```

in questa maniera il campo LUNGHEZZA_TARGA non può essere variato a run-time, inoltre esso deve essere inizializzato con un valore costante, non è possibile dunque assegnargli il valore di un'altra variabile, in quanto quest'ultimo appunto potrebbe variare da un'esecuzione all'altra.

5.3.3.2 Campi a sola lettura

A volte è molto più utile , piuttosto che assegnare un valore costante a tempo di compilazione, assegnare un valore durante l'esecuzione, come risultato di una espressione, ma non variarlo più, creare cioè un campo scrivibile solo una volta, ed a questo punto usarlo in sola lettura. Un simile campo si dichiara con la keyword `readonly` che precede il tipo.
Un tipico esempio potrebbe essere quello di fornire ad una classe un campo `OrarioDiCreazione`, il cui valore dovrebbe essere assegnato solo una volta, appunto alla creazione dell'oggetto, e da quel momento in poi non dovrebbe, e non può, essere variato.

```
class MiaClasse
{
      private readonly OrarioDiCreazione;
      //… resto della classe
}
```

Un campo `readonly` può essere assegnato solo all'interno dei costruttori della classe, cioè al momento della creazione di un oggetto. I costruttori di classe sono esposti in dettaglio a pag. 65.

5.3.4 Metodi e proprietà

Gli oggetti istanziati in un programma necessitano di comunicare fra di loro, ad esempio dopo aver istanziato un oggetto di classe Veicolo, come facciamo a metterlo in moto, ad accelerare e frenare? La classe Veicolo deve mettere a disposizione degli altri oggetti delle funzioni richiamabili dall'esterno, in

modo da poter spedire alle sue istanze dei messaggi. Tali funzioni vengono in genere chiamate metodi della classe.
Un metodo è definito nella seguente maniera, che rappresenta la cosiddetta signature o firma del metodo:

```
modificatori tipo_ritorno NomeMetodo(lista_parametri)
{
   // corpo del metodo
}
```

Il nome del metodo è quello con cui verrà richiamato da altre parti del codice. Il tipo di ritorno determina il tipo dell'eventuale valore che deve essere ritornato dal metodo con un'istruzione `return`. Fra le parentesi che seguono il nome possono essere specificati dei parametri con rispettivi tipi, che saranno in qualche modo utilizzati nel corpo del metodo stesso.
Riprendendo la classe Veicolo aggiungiamo dunque qualche metodo, ad esempio `AccendiMotore` e `SpegniMotore` per cambiare lo stato del motore, ad acceso e spento, rispettivamente.

```
public class Veicolo
{
      public int ruote;
      public float velocita;
      public int direzione;
      private bool motoreAcceso;

      public float GetVelocita()
      {
            return velocita;
      }

      public void AccendiMotore()
      {
            if(!motoreAcceso)
                  motoreAcceso=true;
      }

      public void SpegniMotore()
      {
            if(motoreAcceso)
                  motoreAcceso=false;
      }
}
```

Come avrete notato, i due metodi AccendiMotore e SpegniMotore sono dichiarati con valore di ritorno **`void`**. Tale tipo indica in realtà che il metodo non restituisce alcun valore al chiamante, e quindi non è necessario l'utilizzo dell'istruzione return, a meno che non si voglia terminare il metodo prima di raggiungere la fine del suo corpo.
Per utilizzare un metodo, è necessario chiamarlo tramite un oggetto della classe che lo definisce (a meno che il metodo non sia statico, vedremo tra poco cosa significa, o che il metodo non sia invocato all'interno di un altro metodo della classe stessa che lo espone), per mezzo della notazione:

```
nomeoggetto.NomeMetodo(argomenti);
```

ad esempio per chiamare il metodo AccendiMotore su un'istanza della classe Veicolo, prima bisogna costruire l'istanza e poi spedirgli il messaggio di accensione.

```
public class TestVeicolo
{
      static void Main()
      {
            Veicolo auto=new Veicolo(); //costruzione di auto
            auto.AccendiMotore(); //chiamata del metodo AccendiMotore();
      }
}
```

In C# i metodi devono necessariamente essere membri di una classe, non esistono dunque funzioni globali. Negli esempi che seguono, anche dove non vengono esplicitamente invocati con la notazione oggetto.metodo(), è sottinteso che i metodi appartengano ad una classe, e dunque l'invocazione con la notazione metodo(), cioè senza il nome dell'oggetto, presuppone che essi vengano invocati all'interno dell'oggetto stesso che li espone, cioè si agisce sull'oggetto stesso.
Per esplicitare l'azione sui membri dell'oggetto stesso, può essere utilizzata la parola chiave this, come vedremo più avanti.
Un tipo particolare di metodi sono le proprietà di una classe.
Le proprietà sono dei metodi, che permettono l'accesso in lettura e/o in scrittura agli attributi di una classe. In effetti il comportamento delle proprietà è ottenibile mediante la scrittura di metodi ad hoc, come saranno abituati a fare i programmatori java o C++, cioè i classici SetCampo o GetCampo.
Le proprietà permettono però di accedere ad un campo in maniera intuitiva, ad esempio con un'assegnazione, così come si farebbe con un campo pubblico, ma con la differenza che le proprietà non denotano delle locazioni di memorizzazione dei dati, come i campi, e permettono contemporaneamente di fare dei controlli sul valore che si vuole assegnare o leggere. Inoltre le proprietà mantengono il codice coerente e compatto, visto che la lettura e la scrittura sono implementate in blocchi adiacenti.
Come esempio aggiungeremo alla classe Veicolo una proprietà Direzione, per impostare la direzione di moto, supponendo che essa sia espressa in gradi, e dunque variabile da 0 a 359.

```
public int Direzione
{
      get
      {
            return direzione;
      }
      set
      {
            if (direzione<0)
                  direzione+=360;
            direzione=value%360;
      }
}
```

Abbiamo chiamato la proprietà `Direzione` (sfruttando il fatto che C# è case-sensitive) il che richiama immediatamente il fatto che essa si occupa dell'accesso al campo privato `direzione`. La lettura del campo si ottiene tramite il blocco **`get`**, che deve restituire un valore del tipo dichiarato dalla proprietà. La scrittura del campo avviene invece nel blocco **`set`**, al quale viene sempre passato un parametro in modo implicito, di nome **`value`**.
Sottolineiamo il fatto che non necessariamente i due rami get e set devono essere contemporaneamente presenti, ad esempio è semplice implementare una proprietà di sola lettura fornendo il solo ramo get:

```
public string Nome
{
      get
      {
            return nome;
      }
}
```

Analogamente, anche se capita meno frequentemente si può implementare una proprietà di sola scrittura con il solo ramo set, ad esempio:

```
public string Nome
{
      set
      {
            nome=value;
      }
}
```

5.3.4.1 Passaggio di parametri

Nel caso in cui il metodo abbia degli argomenti nella sua firma, è necessario specificare il corretto numero ed il corretto tipo degli argomenti stessi, al momento di richiamare il metodo, inoltre le variabili passate come argomenti ad un metodo devono necessariamente essere inizializzate. Ad esempio supponiamo di avere un metodo che effettua la somma di due numeri interi, e restituisce il valore di tale somma:

```
public int Somma(int a,int b)
{
      return a+b;
}
```

il metodo così definito deve essere invocato passando come parametri due variabili di tipo int, o di un tipo comunque convertibile, implicitamente o esplicitamente, in int:

```
int somma=Somma(1,2);
```

Gli argomenti possono in generale essere passati per riferimento e per valore. Nell'esempio precedente gli argomenti sono passati per valore, in quanto ciò che in effetti viene passato al metodo è una copia del valore degli argomenti, mentre le variabili originali non subiranno alcuna modifica, infatti se nel corpo del metodo modifichiamo i valori delle variabili argomento, e li stampiamo all'uscita dal metodo stesso, vedremo che i valori sono ancora quelli precedenti alla chiamata. Il metodo seguente calcola il quadrato di un intero, assegnandolo inizialmente alla variabile argomento:

```
public int Quadrato(int a)
{
      a=a*a;//assegno ad a il valore del suo quadrato
      return a;
}
```

Invocando tale metodo, passando come argomento una variabile n:

```
int n=3;
int q=Quadrato(n);
Console.WriteLine("n={0}, q={1}",n,q);
```

Il valore di n stampato all'uscita del metodo, rimane ancora quello assunto precedentemente alla chiamata del metodo stesso.
Nel passaggio per riferimento invece il valore della variabile viene modificato dal metodo, in quanto ciò che viene passato al metodo è appunto un riferimento al valore, e dunque per tipi di dato complessi è più efficiente tale modalità di passaggio dei parametri.
In C# il passaggio dei parametri avviene di default per valore, ciò però significa che nel caso di argomenti di tipo riferimento, cioè di oggetti, ciò che viene passato per valore è comunque il riferimento stesso. Dunque l'oggetto viene modificato dal metodo, ma non è possibile modificare il riferimento stesso. Vediamo un esempio per chiarire tale concetto fondamentale:

```
//Il metodo imposta il numero di ruote del veicolo
public class ImpostaRuote(Veicolo veicolo, int nRuote)
{
      veicolo.ruote=nRuote;
}
```

Il metodo precedente modifica il valore del campo Targa del Veicolo passato come argomento, infatti stampando il valore del campo dopo la chiamata del metodo, esso sarà uguale a quello passato come secondo paramentro al metodo precedente:

```
Veicolo auto=new Veicolo()
ImpostaRuote(auto,4);
Console.WriteLine("numero di ruote del veicolo: "+auto.ruote); //stampa il valore 4
```

Come detto non varia il riferimento dell'oggetto, quindi se all'interno di un metodo cerchiamo di assegnare alla variabile di tipo riferimento una nuova istanza, non otterremmo il risultato sperato:

```
//non funziona, il riferimento a veicolo è passato per valore
private void CreaNuovoVeicolo(Veicolo veicolo)
{
      veicolo=new Veicolo();
}
…

ImpostaRuote(auto,4);
CreaNuovoVeicolo(auto);
Console.WriteLine("numero di ruote passato per valore: "+auto.ruote);
//stampa il valore 4, e non zero
```

Per ottenere il comportamento del passaggio per riferimento degli argomenti, cioè se vogliamo che il valore degli argomento sia modificato all'interno di un metodo, è necessario utilizzare una nuova keyword, vale a dire **ref**, anteponendola al tipo dell'argomento da passare per riferimento:

```
//funziona, il veicolo è passato per riferimento
private void CreaNuovoVeicolo(ref Veicolo veicolo)
{
      veicolo=new Veicolo();
}
```

La parola chiave ref deve essere specificata anche invocando il metodo, anteponendola alla variabile da passare per riferimento:

```
ImpostaRuote(auto,4);
CreaNuovoVeicolo(ref auto);
Console.WriteLine("numero di ruote del veicolo passato per rif: "+auto.ruote); //stampa zero
```

La parola chiave ref può essere naturalmente utilizzata anche per gli parametri di tipo valore, ad esempio il metodo che calcola il quadrato di un numero, scritto così:

```
public void Quadrato(ref int a)
{
      a=a*a;
}
```

modifica il valore del parametro stesso, e ne imposta il valore al quadrato del valore originale:

```
int a=2;
Quadrato(a);
Console.WriteLine("a={0}",a); //stampa 4
```

A volte è utile poter far restituire ad un metodo più di un velore di ritorno. Tale meccanismo è implementabile utilizzando più argomenti ref, uno per ogni valore di uscita desiderato, ma esso, richiede l'inizializzazione delle variabili prima dell'invocazione, sebbene a dei valori qualsiasi, visto che saranno modificati all'interno del metodo, ma ciò può provocare confusione ed errori di interpretazione nei lettori del nostro codice.
Per rimediare a tale possibile confusione dunque, un'altra parola chiave introdotta dal linguaggio C# è **out**, che permette appunto di specificare che un dato argomento è un parametro di uscita di un metodo, che non necessita di inizializzazione prima dell'invocazione del metodo, e richiede dunque un'assegnazione all'interno del metodo stesso. Ad esempio il seguente metodo calcola le potenze del due, del tre e del quattro del valore passato come primo parametro:

```
private void CalcolaPotenze(int a,out int due,out int tre,out int quattro)
{
      due=a*a;
      tre=due*a;
      quattro=tre*a;
}
```

Anche per invocare i metodi con parametri di uscita, è necessario specificare la parola chiave out prima di ogni argomento, ma, come detto, a differenza della parola chiave ref, senza doverli inizializzare:

```
int a=2;
int due, tre, quattro; //tre variabili non inizializzate
CalcolaPotenze(a,due,tre,quattro);
Console.WriteLine("a^2={0}",due);//stampa a^2=4
Console.WriteLine("a^3={0}",tre);//stampa a^3=8
Console.WriteLine("a^4={0}",quattro);//stampa a^4=16;
```

L'uso della parola chiave out evita possibili bug introducibili con un uso distratto di ref, in quanto ad esempio non è possibile dimenticare di assegnare un valore alla variabile d'uscita, in quanto verrebbe segnalato l'errore già in fase di compilazione, come potete provare ad esempio compilando il codice seguente.

```
public void UnMetodo(out string str)
{
     //errore se non si inizializza str
        //str="stringa";
}
```

il metodo precedente non inizializza l'argomento d'uscita str, ed il compilatore darà in output un messaggio d'errore come il seguente:

```
Il parametro out "str" deve essere assegnato prima che il controllo lasci il metodo corrente.
```

Nell'esempio precedente abbiamo restituito tre valori. Ma supponiamo che volessimo calcolare le potenze di un numero fino ad un valore arbitrario dell'esponente, sarebbe necessario che il metodo presenti una lista molto lunga di parametri, e dunque scrivere un metodo per ogni possibile numero di valori d'uscita desiderati, cioè uno per restituire la potenza del due soltanto, uno per quella del due e del tre, e così via. In generale supponiamo dunque di volere un metodo che accetti un valore variabile di parametri. C# fornisce per tale eventualità la parola chiave **`params`**.
La parola chiave params specifica che il parametro di un metodo è in realtà un array di parametri, cioè il metodo può accettare un numero variabile di parametri. La sintassi per l'utilizzo della parola chiave params è la seguente:

```
modficicatore tiporitorno metodo(…, params tipo_array nome_parametro)
```

Ad esempio, per calcolare le potenze di un numero a, specificando quali potenze vogliamo ottenere in un array di interi, possiamo scrivere il metodo:

```
//Calcola le potenze di a specificate nell'array potenze
private void CalcolaPotenze(int a,params int[] potenze)
{
   foreach(int potenza in potenze)
      {
            int p=(int)Math.Pow(a,potenza);
            Console.WriteLine("a^{0}={1}",potenza,p);
      }
}
```

La keyword `params` può essere utilizzata una sola volta nella lista dei parametri, e deve essere applicata all'ultimo parametro del metodo. Inoltre non può essere combinata con `ref` o con `out`.

5.3.4.2 Overloading dei metodi

In C# una classe può avere più metodi con lo stesso nome, ma con numero e/o tipo di parametri differenti. Ad esempio il metodo per ottenere il quadrato di un numero intero può essere scritto:

```
public int Quadrato(int a)
{
      return a*a;
}
```

Tale metodo non può accettare un parametro a virgola mobile, ad esempio non è possibile invocare il metodo nella maniera seguente:

```
double d=Quadrato(2.1);
```

Sia perché il metodo quadrato non accetta come parametro un double, ed inoltre il tipo di ritorno è ancora int, mentre noi tentiamo di assegnare alla variabile d che è double. Magari si potrebbe risolvere questo semplice caso con una conversione di tipo, ma il concetto che vogliamo esporre è che è possibile implementare diversi metodi `Quadrato`, ognuno dei quali accetta diversi tipi di parametro, ad esempio:

```
public double Quadrato(double d)
{
      return d*d;
}
```

in tale maniera il compilatore sa che se invochiamo il metodo passando un parametro `int` deve utilizzare la versione `Quadrato(int a)`, mentre se invochiamo il metodo `Quadrato` con un parametro double deve utilizzare la seconda versione.
Questa tecnica viene chiamata overloading, cioè sovraccarico dei metodi, che è una tecnica fondamentale nella programmazione ad oggetti, ed è dunque assolutamente necessario imparare a maneggiarla.
Come detto è possibile avere più versioni di un metodo con lo stesso nome, ma con differenti parametri, nel numero o nel tipo, ma fate attenzione al fatto che non è possibile differenziare due metodi solo per il tipo di ritorno. Ad esempio non è possibile avere nella stessa classe che contiene il metodo `Quadrato(int)`, definito qualche riga sopra, il seguente:

```
double Quadrato(int a);
```

in quanto quest'ultimo metodo si differenzia solo per il tipo di ritorno. Infatti è possibile, anche se il metodo restituisce un valore, chiamarlo senza utilizzare tale valore di ritorno, ad esempio

```
Quadrato(5);
```

ma in questo caso sarebbe impossibile distinguere fra il metodo `Quadrato` che restituisce un `int` e quello che restituisce un `double`. Inoltre non è possibile differenziare due metodi solo per il fatto che in uno di essi un parametro è dichiarato `ref` mentre nell'altro è dichiarato `out`, ad esempio nella stessa classe non si possono avere entrambi i due metodi seguenti:

```
public void MioMetodo(ref string str)
{…}
public void MioMetodo(out string str)
{…}
```

Per mezzo dell'overload è possibile simulare dei parametri opzionali o di default, che sono caratteristiche supportate in alcuni altri linguaggi. Ad esempio supponiamo di avere un metodo:

```
public MioMetodo(string a,string b)
{
   //fa qualcosa con a e b
}
```

se vogliamo avere la facoltà di invocare il metodo `MioMetodo` specificando solo i parametri a e b, in modo che venga considerato per default uguale a `"hello"`, allora basta fornire l'overload seguente di `MioMetodo`:

```
public void MioMetodo(string a)
{
     MioMetodo(a, "hello");
}
```

che non fa altro che invocare la prima versione, passando appunto come secondo parametro il valore string `"hello"`.

5.3.5 La Parola chiave this

C# fornisce una parola chiave particolare, **`this`**, che viene utilizzata all'interno di un qualsiasi metodo di una classe per riferirsi all'istanza stessa in quel momento in esecuzione. E' cioè come se l'oggetto attivo voglia usare se stesso, specificandolo in modo esplicito per evitare fraintendimenti. Nella maggior parte dei casi ciò non è necessario, per esempio riprendiamo la classe Veicolo:

```
public class Veicolo
{
      …
      private bool motoreAcceso;


      public void AccendiMotore()
      {
            if(!motoreAcceso)
                  motoreAcceso=true;
      }
}
```

Il metodo `AccendiMotore` al suo interno fa riferimento al campo `motoreAcceso`, ed è logico che esso sia un campo della stessa classe. Potremmo scrivere, in maniera equivalente, il metodo in questa maniera:

```
public void AccendiMotore()
{
      if(!this.motoreAcceso)
            this.motoreAcceso=true;
}
```

La parola chiave `this` esplicita il fatto che stiamo usando un membro dello stesso oggetto, e quando la keyword `this` non è presente, tale fatto è sottinteso dal compilatore. Può capitare però che l'uso di `this` si renda necessario, oltre a contribuire ad una maggiore chiarezza del codice, ad esempio:

```
public void SetDirezione(int direzione)
{
      this.direzione=direzione;
}
```

Poiché il parametro ha lo stesso nome di un campo della classe, per riferirsi al campo è necessario usare la sintassi `this.direzione`, mentre scrivendo all'interno del metodo solo `direzione` ci si riferisce alla variabile passata come parametro al metodo.

5.3.6 Costruttori

Il costruttore di una classe è un metodo che permette di stabilire come un oggetto deve essere creato, cioè di controllare la sua inizializzazione. E' obbligatorio che il costruttore si chiami esattamente come la classe, e che non venga specificato alcun tipo di ritorno, neanche `void`, in quanto quello che restituisce un costruttore è proprio l'oggetto che esso deve costruire. Se per sbaglio o per distrazione scriviamo un costruttore con un tipo di ritorno `void` o un altro tipo, il compilatore segnalerà comunque l'errore, in quanto esso verrà visto come un membro qualsiasi della classe, ed un membro non può naturalmente avere lo stesso nome della classe che lo contiene. Un costruttore, come un qualsiasi altro metodo, può avere o meno dei parametri.
Supponiamo di volere costruire un oggetto della classe Veicolo specificando il numero di ruote, allora scriveremo un costruttore che ha come parametro un numero intero:

```
public class Veicolo
{
      private int nRuote;
      private string targa;
      private float velocita;
```

```
        public Veicolo(int ruote)
        {
                nRuote=ruote;
        }
}
```

Non è necessario comunque implementare un costruttore, in quanto, in tal caso, viene di default fornito quello senza parametri. Notate che se implementate almeno un costruttore con uno o più parametri nelle vostre classi, allora quello di default non sarà più disponibile, cioè se la classe `Veicolo` è stata definita come sopra non è più possibile istanziare un oggetto come abbiamo fatto finora con

```
Veicolo auto=new Veicolo(); //non esiste il costruttore
```

ma è necessario usare quello che abbiamo definito sopra e che prende in ingresso un intero:

```
Veicolo auto=new Veicolo(4);
```

Per ogni classe possiamo implementare tutti i costruttori che vogliamo, naturalmente ognuno con un numero o con un tipo differente di parametri, analogamente a come abbiamo visto parlando dell'overloading degli altri metodi della classe. Inoltre possiamo anche definire il costruttore come `private`, in tale maniera esso non sarà naturalmente visibile all'esterno, e quindi non potrà essere utilizzato per istanziare oggetti della classe mediante la keyword `new`. Vedremo più in là l'utilità di un costruttore privato.
In molti casi è comodo richiamare un costruttore da un altro costruttore della stessa classe, come abbiamo visto quando abbiamo richiamato una diversa versione di un metodo per simulare parametri opzionali o di default.
Per poter richiamare un costruttore non è possibile utilizzare il suo nome come fosse un metodo qualunque, a causa dei conflitti di nomi che si verrebbero a creare, ma è necessario utilizzare la parola chiave `this`, in modo da fornire un inizializzatore del costruttore, e ciò si ottiene facendo seguire al costruttore l'operatore : (due punti) e quindi la keyword `this` con i parametri del costruttore da invocare, come se `this` fosse il nome di un metodo.
Nell'esempio seguente, il costruttore senza parametri richiama il costruttore che accetta un intero, passandogli il valore 4. Ciò evita in genere la duplicazione inutile di codice che si avrebbe invece riscrivendo tutto ciò che è contenuto nell'altro costruttore.

```
public Veicolo(int n)
{
…
}

//se non fornisco il parametro, richiamo il costruttore
//l'argomento di default pari a 4
public Veicolo(): this(4)
{
…
}
```

E' bene sottolineare che usando la parola `this` come inizializzatore del costruttore, l'inizializzazione del relativo parametro avviene prima del successivo blocco contenuto nel costruttore, ed inoltre è possibile usare solo una volta this per chiamare un solo altro costruttore.

5.3.7 Distruttori

Il distruttore è, al contrario del costruttore, un metodo invocato quando un oggetto viene rimosso dalla memoria, ad esempio perché esce fuori dal suo scope. In .NET, però, la rimozione dalla memoria di un oggetto che esce fuori dallo scope, non è immediato, e ciò a causa del fatto che è il garbage collector che si occupa di tale rimozione, e la sua entrata in funzione non è deterministica. Tale non determinismo significa che non è prevedibile quando il distruttore verrà richiamato. L'implementazione di un distruttore segue comunque la seguente sintassi:

```
~NomeClasse()
```

```
{
      //effettuo la pulizia delle risorse
}
```

Cioè è un metodo con lo stesso nome della classe, preceduto dalla tilde ~, e senza parametri.
In teoria, la garbage collection di un oggetto dovrebbe avvenire quando non c'è più alcun riferimento all'oggetto, ma in genere è il Common Language Runtime che decide quando è necessario che il Garbage Collector venga azionato. Se è necessario pulire delle risorse che sappiamo di non dovere più utilizzare, visto che non possiamo prevedere quando un oggetto verrà distrutto, cioè rimosso dalla memoria, possiamo scrivere un metodo al cui interno liberiamo tali risorse, metodo che richiameremo esplicitamente quando si finisce di usare un oggetto. Inoltre inseriremo una chiamata allo stesso metodo anche all'interno del distruttore, in modo che anche se non lo invochiamo noi, esso verrà invocato alla distruzione dell'oggetto da parte del Garbage Collector.

```
public class MiaClasse()
{
      …

      ~NomeClasse()
      {
            Clean();
      }

      public void Clean()
      {
            //effettuo la pulizia delle risorse
      }

      public static void Main()
      {
            NomeClasse oggetto=new NomeClasse();
            //utilizzo oggetto
            …
            //alla fine libero le risorse
            oggetto.Clean();
      }

}
```

In questo esempio abbiamo implementato il metodo `Clean()`, ma naturalmente possiamo chiamare tale metodo come meglio ci pare. Vedremo più avanti che la liberazione delle risorse di una classe avviene in maniera standard implementando un metodo `Dispose()`.

5.3.8 Membri statici

In tutti gli esempi fatti finora e in quelli che avete scritto e compilato per conto vostro, avrete scritto centinaia di volte un metodo `Main`, facendolo precedere sempre dal modificatore **`static`**. Adesso è giunto il momento di spiegare il significato di membri statici di una classe, e dunque dell'utilizzo della keyword static.
Sarà ormai chiaro che quando creiamo un'istanza di una classe, cioè quando creiamo un oggetto, esso possiede uno stato proprio, cioè un proprio insieme dei campi, definiti dalla classe di appartenenza. Ad esempio, se la classe Persona viene definita nel seguente modo:

```
public class Persona
{
      private int eta;
   private string nome;

      public string Nome
      {
            get{
                  return nome;
            }
            set{
                  nome=value;
            }
      }
}
```

ogni oggetto della classe Persona possiede un proprio nome ed una propria età, ed anzi per accedere a tali campi è necessario creare delle istanze della classe:

```
Persona p1=new Persona();
Persona p2=new Persona();
p1.nome="antonio";
p2.nome="caterina";
```

Ci sono comunque delle situazioni in cui è necessario avere dei membri che non siano associati ad ogni singola istanza, ma che siano membri comuni a tutte le istanze di una classe. Tali membri sono i membri statici, detti anche membri di classe, al contrario dei precedenti che sono detti membri d'istanza.

5.3.8.1 Campi statici

Un campo contrassegnato con la parola chiave `static` è un campo statico, cioè un campo di classe. Ad esempio supponiamo di voler contare il numero di istanze della classe Persona del paragrafo precedente, create durante l'esecuzione del programma. Possiamo aggiungere un campo intero `contatore`, che appunto mantenga il numero di istanze. Tale campo deve essere appunto `static`, in quanto è un campo della classe `Persona`, ed è comune a tutte le istanze.

```
public class Persona
{
      public static int numeroPersone;

      // resto della classe
}
```

L'accesso ai membri `static` non avviene mediante un'istanza, in quanto come detto sono membri della classe, e dunque vengono acceduti tramite il nome della classe, ad esempio per accedere al campo `static numeroPersone` della classe persona, dobbiamo scrivere:

```
Persona.numeroPersone=1;
```

Qundi la notazione è semplicemente

```
NomeClasse.nomeMembroStatic
```

l'unica eccezione si può fare quando accediamo ad un membro `static` dall'interno di un'istanza della classe stessa, nel quale caso si può omettere il nome della classe.
Quindi possiamo automaticamente incrementare la varibiale che conta il numero di oggetti `Persona` creati, direttamente alla costruzione degli stessi, vale a dire nel costruttore, ed essendo all'interno di un'istanza della classe Persona potremmo anche omettere il nome della classe, ma per evitare confusione e mantenere chiaro il fatto che un membro è statico, è consigliabile utilizzare sempre la notazione NomeClasse.membroStatico.

```
public class Persona
{
      public static int numeroPersone;
public Persona()
{
      Persona.numeroPersone++;//una persona creata
}

~public Persona()
{
      Persona.numeroPersone--;//una persona distrutta
}

static void Main()
{
      Persona p1=new Persona();
      Console.WriteLine("1. Numero di persone viventi: {0}",Persona.numeroPersone);
      Persona p2=new Persona();
      Console.WriteLine("2. Numero di persone viventi: {0}",Persona.numeroPersone);
```

```
}
}
```

Non possiamo però verificare la distruzione di un oggetto con certezza, data che il processo di garbage collection non è deterministico.
I campo statici di una classe vengono inizializzati la prima volta che si crea un oggetto della classe stessa oppure la prima volta che viene usato un membro statico della classe, e se non inizializzati in modo esplicito, essi assumeranno i valori di default che abbiamo visto per campi non statici.

5.3.8.2 Metodi statici

Un metodo statico, analogamente a quanto visto per i campi, è un metodo che appartiene ad una intera classe, e non ad una singola istanza. Abbiamo già utilizzato dei metodi statici cioè metodi di classe, ad esempio il metodo `WriteLine` della classe `Console` è un metodo statico, infatti l'abbiamo sempre invocato con la notazione

```
Console.WriteLine();
```

Senza la necessità di istanziare un'istanza della classe `Console` e poi chiamare il metodo `WriteLine` sull'istanza creata, cosa d'altronde non permessa, visto che la classe `Console` non ha un costruttore pubblico.
Possiamo quindi scrivere un metodo statico per accedere ad un campo statico di una classe, ed all'interno di tale metodo possiamo solo accedere a membri statici della classe stessa o di altre classi.
Come esempio modifichiamo la classe Persona rendendo `private` il campo statico `numeroPersone`, ed aggiungiamo un metodo che restituisca il valore del campo. In tale maniera evitiamo che dall'esterno si possa dare un valore arbitrario al campo, con un'assegnazione tipo la seguente:

```
Persona.numeroPersone=100000;
```

Abbiamo fatto sì che l'accesso al campo possa avvenire solo dall'interno della classe, ad esempio solo nel costruttore per l'incremento e nel distruttore il decremento. Quindi aggiungiamo il metodo per la lettura del campo:

```
public static int GetNumeroPersone()
{
    return Persona.numeroPersone;
      //non possiamo accedere ad un membro non static,
}
```

Un metodo `static` è come detto un metodo di classe e dunque non esiste alcuna istanza utilizzabile dal suo interno. Ciò implica che non possiamo utilizzare all'interno di un metodo statico nessun membro d'istanza, tantomeno è possibile utilizzare la parola chiave `this`, visto che non c'è nessun this in questo caso:

```
public static int GetNumeroPersone()
{
      //non possiamo accedere ad un membro non static
      this.nome=""; //errore
}
```

5.3.9 Costruttori statici

Una delle novità introdotte da C# rispetto ad altri linguaggi classici, tipo C++ o Java, è la possibilità di implementare un costruttore statico. Un costruttore di tale tipo non può avere parametri e viene invocato solo una volta, prima dell'uso della classe. In genere un costruttore statico viene implementato quando si vogliono inizializzare dei campi statici della classe, ad esempio

```
public class Persona
{
   public static int numeroPersone;

      static Persona()
```

```
   {
      numeroPersone=0;
   }

   public Persona()
      {
      }
}
```

Notate che è possibile implementare contemporaneamente anche un costruttore non statico senza parametri, sebbene esista già un costruttore statico. In tale caso infatti non si ha conflitto di nomi perché il costruttore statico non può essere chiamato dal codice dello sviluppatore, ma viene invocato solo dal CLR. Se ci sono classi diverse che implementano i loro costruttori statici, non è determinato con certezza quale di essi verra chiamato per primo, quindi è bene non fare affidamento su tale ordine d'invocazione, l'unica garanzia è che il costruttore statico verrà prima o poi invocato, prima della costruzione di un'istanza della propria classe.

5.3.10 Overloading degli operatori

C# permette di aggiungere ad una classe un altro tipo di membri, cioè dei membri di overload degli operatori. L'operazione di overload degli operatori aggiunge la possibilità di utilizzare i classici operatori, ad esempio + o –, con le istanze di una classe.
Cio è di estrema utilità se implementiamo delle classi per le quali gli operatori sarebbero più naturali di una chiamata di metodo. Ad esempio supponiamo di avere una classe che rappresenti i numeri complessi della nota forma a+jb (oppure a +ib per i matematici).

```
Class NumeroComplesso
{
      private float re;
      private floate im;

      public float Re
      {
            get
            {
                  return re;
            }
            set
            {
                  re=value;
            }
      }

      public float Im
      {
            get
            {
                  return im;
            }
            set
            {
                  im=value;
            }
      }

      //formatta un numero complesso nella maniera classica a+jb
      public override string ToString()
      {
            return re+"+j"+im;
      }
}
```

Per i numeri complessi sono definite le operazioni di somma, sottrazione, moltiplicazione e divisione. Potremmo implementare ad esempio la prima scrivendo un metodo Somma:

```
public static NumeroComplesso Somma(NumeroComplesso z1, NumeroComplesso z2)
{
      NumeroComplesso s=new NumeroComplesso();
      s.Re=z1.Re+z2.Re;
```

```
      s.Im=z1.Im+z2.Im;
      return s;
}
```

Supponendo che z1 e z2 siano due istanze di NumeroComplesso, possiamo dunque calcolarne la somma così:

```
NumeroComplesso somma=NumeroComplesso.Somma(z1,z2);
```

L'overload degli operatori permette, come accennato, di utilizzare invece la più naturale sintassi:

```
NumeroComplesso somma=z1+z2;
```

Tale meccanismo è possibile in quanto un operatore non è altro che un metodo della classe, che ha come nome il simbolo dell'operatore stesso preceduto dalla keyword **`operator`**, che indica appunto che stiamo sovraccaricando un operatore, ed in più è necessario che il metodo sia statico, in quanto associato ad una classe o struct, e non ad una singola istanza di esse. Vediamo ad esempio come implementare l'operatore + per due numeri complessi:

```
public static NumeroComplesso operator +(NumeroComplesso z1, NumeroComplesso z2)
{
      NumeroComplesso s=new NumeroComplesso();
      s.Re=z1.Re+z2.Re;
      s.Im=z1.Im+z2.Im;
      return s;
}
```

E' possibile implementare tutti gli overload che si vogliono per un dato operatore, ad esempio supponiamo di voler effettuare la somma di un numero reale e di un complesso, in questa maniera:

5+ (3+j4)= 8+ j4

Il risultato è ancora un numero complesso, in quanto il numero reale si somma solo alla parte reale. Quello che bisogna fare è implementare un ulteriore overload dell'operatore + che prende come argomento un numero, ad esempio di tipo double:

```
public static NumeroComplesso operator +(double d, NumeroComplesso z)
{
      NumeroComplesso s=new NumeroComplesso();
      s.Re=d+z.Re;
      s.Im=z.Im;
      return s;
}
```

Ma questo non basterebbe se volessimo invece effettuare la somma nell'ordine inverso, cioè

(6+j)+7

dunque è necessario un secondo overload con i parametri scambiati di posto, e che in questo caso non fa altro che chiamare la precedente versione dell'operatore:

```
public static NumeroComplesso operator +(NumeroComplesso z,double d)
{
      //uso l'overload precedente!
      return d*z;
}
```

In questi esempi di overload dell'operatore +, il tipo di ritorno è ancora un `NumeroComplesso`, cioè la somma di due numeri complessi è ancora un numero complesso, così come lo è la somma di un double e di un complesso, ma possono anche esserci casi in cui il tipo di ritorno debba essere diverso. Ad esempio pensate al prodotto scalare di due vettori, che restituisce non un vettore, ma un numero scalare:

[a,b,c] * [d,e,f] = a*d+b*c+c*f

Dunque l'overload dell'operatore * per un'ipotetica classe Vettore, potrebbe essere implementato come nel modo seguente:

```
public static double operator * (Vettore x,Vettore y)
{
      return x.a*y.a + x.b*y.b + x.c* y.c;
}
```

Non sono esclusi da tali meccanismi altri operatori. Gli operatori unari sovraccaricabili sono:

```
+ - ! ~ ++ -- true false
```

`true` e `false` non sono naturalmente operatori nel senso classico, ma essendo spesso usati in espressioni booleane, è possibile eseguire overload come fossero operatori. Ad esempio supponiamo di voler valutare un numero complesso come booleano in un'espressione del tipo:

```
if(z1)
{
      Console.WriteLine("z1 è un numero complesso");
}
else Console.WriteLine("z1 è un numero reale");
```

in modo che un `NumeroComplesso` con la parte immaginaria nulla sia considerato semplicemente un numero reale, e che dunque, quando viene valutata l'espressione precedente, corrisponda al valore false. I seguenti overload sono adatti allo scopo:

```
public static bool operator true(NumeroComplesso z)
{
      return z.im!=0;
}

public static bool operator false(NumeroComplesso z)
{
      return z.im==0;
}
```

Gli operatori binari che supportano l'overload sono invece

```
+  -  /  *  %  &  |  ^  <<  >>  ==  !=  >=  >  <=  <
```

Implementando l'overload di un operatore binario si ottiene automaticamente l'overload del corrispondente operatore di assegnazione composta. Ad esempio avendo implementato l'overload dell'operatore +, sarà disponibile anche quello +=.
Una particolarità importante è invece data dagli operatori di confronto, i cui overload devono essere implementati a coppia, vale a dire che se implementiamo l'operatore di uguaglianza ==, deve essere fornito anche quello corrispondente di disuguaglianza !=, ed analogamente ciò vale per < e > e per <= e >=. Nel caso in cui non si forniscano tali overload, si otterrebbe un errore di compilazione.

```
public static bool operator ==(NumeroComplesso z1,NumeroComplesso z2)
{
      return (z1.Re==z2.Re)  && (z1.Im == z2.Im);
}

public static bool operator ==(NumeroComplesso z1,NumeroComplesso z2)
{
      //utilizza l'operatore ==
      return ! (z1==z2);
}
```

Un punto su cui torneremo parlando della classe `System.Object` e dell'ereditarietà è che effettuando l'overload degli operatori == e != bisogna anche eseguire l'override dei metodi `Equals` e `GetHashCode` ereditati appunto da `Object`.

5.3.11 Gli indicizzatori

Gli indicizzatori sono dei membri che permettono di accedere ad un oggetto in maniera analoga a quanto visto per gli array, cioè per mezzo dell'operatore di indicizzazione [ ] e di un indice racchiuso tra le parentesi stesse.
Se implementassimo ad esempio una nostra classe Matrice o Vettore, sarebbe naturale per un suo utilizzatore aspettarsi di avere a disposizione un meccanismo di indicizzazione dei suoi elementi, e sottolineiamo che sebbene esso sia molto potente e flessibile, è comunque bene utilizzarlo solo quando si abbia bisogno di un'indicizzazione tipo array, e quando essa è naturale per l'utilizzatore.
Gli indicizzatori in C# assomigliano molto a delle semplici proprietà con la differenza che i primi hanno anche dei parametri, appunto gli indici. Vediamo come aggiungere ad una classe un indicizzatore, ad esempio supponiamo di volere accedere alle righe di un documento di testo come se fossero gli elementi di un array:

```
class Documento
{
      private string[] righe;

      public Documento(int n)
      {
            righe=new string[n];
      }

      public string this[int i]
      {
            get
            {
                  if(i<righe.Length)
                  return righe[i];
                  else return "";
            }
            set
            {
                  if(i<righe.Length)
                        righe[i]=value;
            }
      }
}
```

con tale indicizzatore possiamo ricavare o impostare le righe di un oggetto `Documento` in questa maniera

```
Documento doc=new Documento(3);
doc[0]="prima";
doc[1]="seconda";
doc[2]="terza";
Console.WriteLine(doc[0]);
```

Fino a qui è tutto analogo all'indicizzazione degli array classici, ma possiamo anche utilizzare come indice un tipo non numerico, anzi uno qualsiasi dei tipi predefiniti di C#, compresi `object` e `string`. Ad esempio supponiamo di volere ricavare l'indice di una particolare riga di testo del documento, possiamo aggiungere un indicizzatore con tipo del parametro `string` e che restituisce un `int` . Il parametro stringa rappresenta la riga di testo da cercare nel documento, mentre il valore di ritorno rappresenta l'indice della riga trovata, se esiste, oppure il valore -1:

```
public int this[string str]
{
      get
      {
            for(int i=0;i<righe.Length;i++)
            {
                  if(righe[i]==str)
                        return i;
            }
            return -1;
      }
```

```
}
```

Notate che in questo caso l'indicizzatore possiede solo il ramo `get`, cioè è di sola lettura. Possiamo dunque ora usare l'indicizzatore:

```
int i=doc["seconda"];
Console.WriteLine("doc[\"seconda\"]={0}",i);// stampa 1
```

Analogamente è possibile implementare indicizzatori con più indici, come se avessimo a che fare con array multidimensionali. L'indicizzatore seguente permette di ricavare il carattere del documento ad una data riga ed una data colonna, oppure di impostarlo.

```
public char this[int riga,int colonna]
{
      get
      {
            if(riga<righe.Length)
            {
                  string str=righe[riga];
                  if(colonna<str.Length)
             return str[colonna];
            }
            return '\u0000';
      }
      set
      {
            if(riga<righe.Length)
            {
                  string str=righe[riga];
                  if(colonna<str.Length)
                  {
                        righe[riga]=str.Substring(0,colonna)+value;
                        if(colonna<str.Length-1)
                              righe[riga]+=str.Substring(colonna+1);
                  }
            }
      }
}
```

Notate che nell'implementazione abbiamo utilizzato l'indicizzatore di sola lettura della classe `System.String`, che consente di ottenere il carattere della stringa alla posizione specificata.

```
Console.WriteLine(doc[2]);     //stampa terza riga
doc[2,3]='r';//cambia il carattere di indice 3
Console.WriteLine(doc[0]);//stampa la terza riga modificata
```

5.4 L'incapsulamento

Quando programmiamo ad oggetti, e quindi implementiamo delle classi, incapsuliamo il codice in unità ben definite e delimitate, il cui funzionamento viene nascosto agli utilizzatori esterni, in modo da fornire solo un'interfaccia ben determinata. Ad esempio supponiamo di avere una classe Veicolo, che implementa un metodo `Accensione()`, quello che all'utilizzatore della classe interessa è solo sapere che tale metodo esiste, e quindi ad esempio invocarlo così:

```
Veicolo auto=new Veicolo();
auto.Accensione();
```

Quello che avviene per accendere l'automobile, all'utilizzatore esterno non interessa, ad esempio il corpo del metodo potrebbe essere semplicemente questo

```
public void Accensione()
{
      bMotoreAcceso=true;
}
```

oppure contenere più operazioni, semplici o complesse, necessarie ad accendere il motore

```
public void Accensione()
```

```
{
	AccendiQuadro();
	if(carburante>0)
	{
		IniezioneCarburante();
		//eccetera…
		bMotoreAcceso=true;
	}
}
```

La programmazione ad oggetti consente dunque la cosiddetta l'information hiding, cioè il nascondere le informazioni mediante l'incapsulamento in classi, e di accedere ad esse tramite metodi e proprietà. L'incapsulamento è assieme al polimorfismo ed all'ereditarietà, un concetto cardine del paradigma Object Oriented.
L'accesso alle informazioni è regolato per mezzo dei già citati modificatori di accesso (vedi pag 79).
Per default i membri di una classe sono `private`, ma è consigliabile comunque specificare il tipo di accesso in modo esplicito per evitare fraintendimenti.

5.5 Composizione di classi

Abbiamo visto che una classe può avere come campi, costituenti il suo stato, delle variabili di tipi primitivi. Ma in parecchi casi un oggetto è un agglomerato di diversi oggetti, ognuno di classe diversa. Ad esempio un Veicolo, oltre ad avere caratteristiche descrivibili mediante tipi numerici, come il numero di ruote, o stringhe per descrivere il modello, o altri tipi fondamentali (comunque ricordate che si tratta sempre di oggetti), è composto da altri oggetti più o meno complessi: ruote, motore, serbatoio, autoradio e chi più ne ha più ne metta.
Tali oggetti saranno dunque istanze di classi implementate da noi o da altri, ed inseriti come campi della classe contenitrice Veicolo. La sintassi è comunque identica anche in caso di classi composite, supponiamo di implementare una classe Motore, ed una classe Serbatoio, in questa maniera:

```
class Motore
{
	private	int cilindrata;
	public Motore(int cc)
	{
		cilindrata=cc;
	}

	public void Accendi()
	{
		Console.WriteLine("Vrooom!!");
	}
}

class Serbatoio
{
	float capacita;
	float carburante;
	public Serbatoio(float cap)
	{
		capacita=cap;
		carburante=0.0f;
	}

	public float Capacità
	{
		get
		{
			return this.capacita;
		}
	}

	public float Carburante
	{
		get
		{
			return carburante;
		}
```

```
            set
            {
                  if(value>0)
                  {
                        if(carburante+value>capacita)
                              carburante=capacita;
                        else carburante+=value;
                  }
            }
      }
}
```

Adesso creiamo una classe Auto che possiede un motore ed un serbatoio e ne utilizza le funzionalità, ad esempio per accedere il motore è necessario avere del carburante nel serbatoio;

```
class Auto
{
      private     int numeroRuote;
      private     string modello;
      private     Motore motore;
      private Serbatoio serbatoio;

      public Auto(string mod, int cilindrata)
      {
            numeroRuote=4;
            modello=mod;
            motore=new Motore(cilindrata);
            serbatoio=new Serbatoio(50.0f);
      }

      public bool Accensione()
      {
            if(serbatoio.Carburante>0)
            {
                  this.motore.Accendi();
                  return true;
            }
            else return false;
      }

      public void RiempiSerbatoio()
      {
            serbatoio.Carburante=serbatoio.Capacità;
      }
}
```

In questa maniera abbiamo creato una classe Auto, come composizione di oggetti di classi differenti.

5.5.1 Classi nidificate

E' possibile dichiarare ed implementare classi all'interno di altre dichiarazioni di classi, cioè nidificare le dichiarazioni di classe. Ad esempio potremmo riscrivere la classe `Auto` precedente dichiarando le classi `Motore` e `Serbatoio` come classi innestate (**nested**) all'interno della classe `Auto`.

```
class Auto
{
      class Motore
      {…}
      class Serbatoio
      {…}
      //corpo della classe Auto
}
```

Un possibile motivo per usare tale sintassi, oltre ad una maggior chiarezza sull'utilizzo che se ne vuol fare, è quella che le classi innestate o nested potrebbero essere utili solo alla classe che le contiene, e quindi non c'è motivo di dichiararle esternamente. Inoltre le classi nested hanno completa visibilità sulla classe che le contiene, anche dei membri privati, ad esempio il motore potrebbe sapere su quale auto è montato, senza necessità di aggiungere ulteriori metodi pubblici alla classe Motore.

E' comunque possibile anche dichiarare una classe nested come public, in tale maniera esse sarebbero utilizzabili anche al di fuori dalla classe ospitante, con la notazione `ClasseEsterna.ClasseInterna`, in questa maniera, ad esempio, potremmo creare un motore senza una auto che lo contenga:

```
Auto.Motore motore=new Auto.Motore();
```

E' possibile avere più livelli di classi innestate, con le stesse regole di quanto appena visto, ma lasciamo come esercizio al lettore il compito di creare una simile struttura di più livelli di nidificazione.

5.6 Ereditarietà e polimorfismo

Un altro dei concetti fondamentali della programmazione orientata agli oggetti è il concetto di ereditarietà. Fino ad ora abbiamo implementato una classe ogni volta che abbiamo dovuto scrivere un esempio, aggiungendo magari dei membri per migliorarne o raffinarne le funzionalità.
Il paradigma OOP permette di specificare con esattezza questa pratica di raffinamento o meglio di specializzazione delle funzionalità.
Supponiamo ad esempio di riprendere la classe `Veicolo` implementata qualche pagina fa, essa potrebbe rappresentare un qualsiasi veicolo generico, anzi al momento di costruire un istanza auto della classe Veicolo, ne abbiamo creata una specificando che il numero di ruote fosse pari a quattro.
L'ereditarietà consente di creare delle gerarchie di classi, da quelle più generiche, a quelle che ne specializzano, ereditandole se si vuole, le funzionalità. Ad esempio Veicolo potrebbe rappresentare una classe generica di oggetti che posseggono delle ruote e che si muovono su di esse, ma un'automobile potrebbe essere invece una classe che la specializza, fissando il numero di ruote a quattro e inglobando un motore, mentre una classe Bicicletta rappresenta ancora un Veicolo, ma essa ha due ruote e non ha motore, e si muove per mezzo di pedali.
Per rappresentare le relazioni di ereditarietà, è di notevole utilità il linguaggio di modellazione UML, in cui una classe viene rappresentata in generale come un rettangolo con il nome in alto, e diversi comparti per campi e metodi, ad esempio la classe Veicolo può essere rappresentata come nella Figura 5.1.

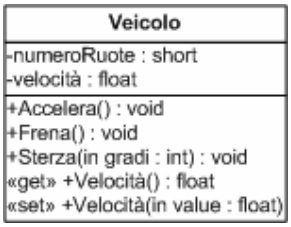


Figura 5.1: La classe Veicolo in UML

Osservando la classe `Veicolo` qui rappresentata, noterete che nel primo riquadro dall'alto appare il nome della classe, quindi in quello successivo i campi, ed infine i metodi e le proprietà. Non è scopo di questo testo illustrare il linguaggio UML, ma queste poche nozioni vi saranno utili nel proseguio e nella progettazione delle vostre prime classi.
Tornando a parlare di ereditarietà, essa si rappresenta in UML mediante una freccia che va dalla classe derivata alla classe madre, ad esempio le classi `Auto` e `Bicicletta` che derivano dalla classe `Veicolo`, possono essere rappresentate come nella Figura 5.2 .
Le classi `Auto` e `Bicicletta` condividono in questa maniera delle informazioni, come ad esempio `numeroRuote` e `velocità`, definite nella comune classe di base, o superclasse.

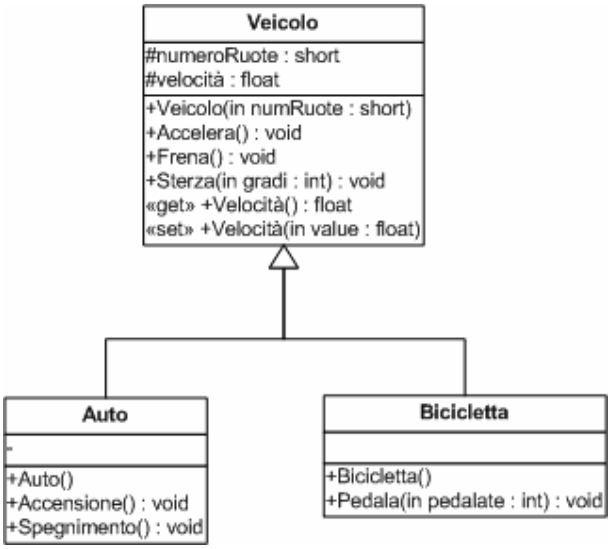


Figura 5.2 Ereditarietà

Ed ancora, continuando la nostra gerarchia, un' `Auto` potrebbe specializzarsi in una classe `AutoElettrica` che possiede un motore elettrico, mentre un `Tandem` può essere una classe che deriva molte delle sue caratteristiche da una `Bicicletta`. Se poi vogliamo complicarci le cose potremmo cercare di riunire i due rami creando una `BiciAMotore` o un `AutoAPedali`, ma per ora fermiamoci qui, esistono vari testi validi che illustrano nel dettaglio il paradigma di programmazione orientato agli oggetti.
In un linguaggio ad oggetti tutto ciò si traduce dicendo che una classe deriva da un'altra, ad esempio ogni classe della base class library, risalendo lungo la catena di ereditarietà ha come classe madre la classe `System.Object`, cioè alla fin fine tutto è un oggetto.

5.6.1 Implementare l'ereditarietà

Per derivare una nuova classe da una classe esistente, come ad esempio la classe `Auto` che deriva dalla classe `Veicolo`, basta definire la classe come visto finora, con in più il fatto che il nome stesso della classe viene seguito da due punti (:) e quindi dal nome della classe base.

```
class Auto: Veicolo
{
   …
}
```

E' possibile leggere la dichiarazione precedente come "Auto deriva da Veicolo". Con tale sintassi si informa il compilatore che la classe `Auto` è figlia della classe `Veicolo`, e quindi ogni membro della classe `Veicolo` è ereditato dalla classe `Auto`. Possiamo creare un'istanza della classe `Auto` ed utilizzare i membri ereditati da `Veicolo` semplicemente scrivendo

```
Auto auto=new Auto();
auto.Accelera();
auto.Frena();
```

In molti casi però è necessario regolare l'accesso ai membri della classe madre, ad esempio facendo in modo che alcuni di tali membri siano ereditati dalla classe figlia, mentre altri rimangano membri a disposizione solo della classe madre. Tali meccanismi vengono implementati con l'uso appropriato dei modificatori di accesso, dei quali abbiamo già visto i modifcatori `public` e `private`. La tabella seguente illustra tutti i modificatori di accesso applicabili ai membri di una classe per regolare appunto i meccanismi suddetti.

Modificatore	Descrizione

d'accesso	
public	I membri pubblici sono visibili a qualunque altra classe e relativi membri.
protected	I membri protected sono accessibili oltre che dalla classe stessa, anche dalle classi figlie derivate da essa.
private	I membri privati sono visibili solo all'interno della classe stessa.
internal	I membri internal sono visibili all'interno dello stesso assembly in cui è definita la classe.
protected internal	I mebri protected internal sono accessibili alle classi dello stesso assembly della classe, ed alle classi derivate da essa (anche definite in altri assembly).

Tabella 4 Modificatori ed ereditarietà

Riprendendo la nostra classe Veicolo, supponiamo che essa sia così dichiarata

```
class Veicolo
{
      private int numeroRuote;
      private float velocita;
}
```

In tale maniera, se deriviamo la classe `Auto` da quella `Veicolo`, i campi `numeroRuote` e `velocita` non sarebbero visibili dall'auto. In molti casi simili a questo è comodo poter dichiarare dei membri nascosti a tutto il mondo esterno, eccetto che alle classi derivate. Utilizzando il modificatore `protected` possiamo ottenere questo comportamento:

```
class Veicolo
{
      protected int numeroRuote;
      protected float velocita;
}
```

La classe `Auto`, derivata così da `Veicolo` avrà fra i suoi attributi anche i due campi di cui sopra e, ad esempio nel costruttore, possiamo inizializzarli come fossero due membri definiti in essa:

```
class Auto: Veicolo
{
      public Auto()
      {
            numeroRuote=4;
            velocita=0;
      }
}
```

I modificatori `internal` e `protected internal` permetteno di raffinare ulteriormente la gerarchia delle classi, così come esposto nella tabella precedente.

5.6.2 Upcasting e downcasting

Utilizzare l'ereditarietà permette di stabilire una relazione fra classi, in particolare una relazione "la classe figlia è del tipo della classe madre". Ad esempio `l'Auto` è un tipo di `Veicolo`. Ciò permette ad esempio di poter scrivere codice come il seguente:

```
Veicolo veicolo;
Auto auto=new Auto();
veicolo=auto;
veicolo.Accelera();
```

Possiamo trattare cioè un'auto come `Veicolo` qualsiasi. L'assegnazione della terza linea di codice converte un'istanza di una classe derivata nel tipo da cui deriva, in questo caso l'`Auto` in un `Veicolo`, tale operazione è chiamata **upcasting** ed è sempre lecita e permessa dal compilatore in quanto la

classe madre è naturalmente più generica della classe figlia, e quest'ultima, essendo derivata, conterrà almeno i metodi della classe madre ed eventualmente degli altri che ne specificano il comportamento. L'operazione inversa, cioè il **downcasting**, è quella di conversione da un oggetto della classe madre ad uno della classe figlia, e questo non è sempre lecito, in quanto non è possibile sapere se l'oggetto da convertire è compatibile con quello destinazione, ad esempio non possiamo con certezza sapere se un `Veicolo` è in realtà un `Auto` o una `Bicicletta`. Il downcasting è eseguibile per mezzo dell'operatore di cast oppure per mezzo dell'operatore **`as`**:

```
Veicolo nuovoVeicolo=new Veicolo();
Auto auto=(Auto)nuovoVeicolo;//downcasting non valido, genera un'eccezione

auto=nuovoVeicolo as Auto;//downcasting non valido, auto restituisce null
if(auto==null)
      Console.WriteLine("nuovoVeicolo non e un'auto");
```

Utilizzando l'operatore di cast, se il downcasting non è possibile, verrà generata un'eccezione `InvalidCastException`, mentre utilizzando l'operatore `as`, verrà restituito il valore `null`. Dunque è sempre bene, se non si è sicuri della possibilità di un downcasting, verificare il tipo degli oggetti, ad esempio con l'operatore **`is`**:

```
if(nuovoVeicolo is Auto)
{
      auto=(Auto)nuovoVeicolo;//il cast è possibile
      Console.WriteLine("nuovoVeicolo è un'auto");
}
```

5.6.3 Hiding ed overriding

I modificatori visti poc'anzi sono applicabili naturalmente anche ai metodi di una classe, ma in tal caso è necessario focalizzare l'attenzione su due aspetti in cui C# differisce da altri linguaggi. Supponiamo ad esempio di avere implementato all'interno della classe `Veicolo` un metodo `Accelera`, e di derivare dalla classe `Veicolo`, oltre alla classe `Auto` di prima, anche una classe Bicicletta. Naturalmente il metodo `Accelera` della `Bicicletta` sarà diverso nel funzionamento da quello implementato per la classe `Auto`, e magari da quello di ogni altro tipo di `Veicolo` che andremo ancora a derivare. In tal caso dovremo implementare più volte il metodo per ognuna delle classi. Ma se scriviamo semplicemente le classi `Veicolo` e `Auto` in questo modo

```
class Veicolo
{
      protected int numeroRuote;
      protected float velocita;
      protected float velocitaMax;

      public void Accelera()
      {
                  Console.WriteLine("Il veicolo accelera in qualche modo");
      }
}

class Auto:Veicolo
{
      public Auto()
      {
            numeroRuote=4;
            velocita=0;
            velocitaMax=180.0f;
      }

      public void Accelera()
      {
            if(velocita+5<velocitaMax)
                  velocita+=5;
            Console.WriteLine("velocità auto= "+velocita);
      }
}

public static void Main()
```

```
{
	Veicolo veicolo=new Auto();
	Auto auto=new Auto();

	veicolo.Accelera();
	auto.Accelera();
}
```

Il metodo `Accelera` della classe `Auto` nasconde (**hide**) il metodo omonimo della classe madre, quindi per un'oggetto `Auto` verrà invocato l'`Accelera` della relativa classe, al contrario avverrà per un `Veicolo`. Infatti l'output del codice precedente sarà questo:

```
Questo veicolo non parte!
velocità auto= 5
```

Il problema nasce nel metodo `Main` di sopra, in cui sia l'oggetto `veicolo` che `auto` vengono istanziati come `Auto`, ma se eseguite il codice vi accorgerete che con la prima chiamata ad `Accelera` verrà chiamato il metodo relativo della classe `Veicolo`, anche se l'oggetto è in effetti un'`Auto`, questo perché non abbiamo specificato meglio il comportamento dei metodi `Accelera` nelle due classi ed allora quello che governa quale versione del metodo chiamare è il tipo della variabile, `Veicolo` nel primo caso e `Auto` nel secondo, e non il tipo effettivo degli oggetti istanziati, che è `Auto` in entrambi i casi. Tale possibile sorgente di bug è segnalata
in fase di compilazione con un warning del tipo:

```
The keyword new is required on 'Auto.Accelera()' because it hides inherited member
'Veicolo.Accelera()'
```

Nei prossimi paragrafi vedremo come evitare tale comportamento.

5.6.4 Il polimorfismo

Il paradigma di programmazione orientato agli oggetti possiede un'altra caratteristica fondamentale, strettamente legata all'ereditarietà, il polimorfismo. Il polimorfismo indica la capacità degli oggetti di assumere comportamenti diversi a seconda della classe di appartenenza.
Ciò permette di ottenere un funzionamento più "safe" rispetto a quello visto nel paragrafo precedente, effettuando l'overriding dei metodi ereditati e prevedendo già nell'implementazione della classe madre quali dei suoi metodi saranno sottoposti a sovraccarico nelle classi derivate. Ciò si ottiene in due possibili maniere. La prima è quella di dichiarare il metodo della classe base con la parola chiave **`virtual`**, ed il metodo corrispondente delle classi figlie con la parola chiave **`override`**:

```
class Veicolo
{
	…
	public virtual void Accelera()
	{
		Console.WriteLine("Il veicolo accelera in qualche modo");
	}
}

class Auto:Veicolo
{
	…
	public override void Accelera()
	{
		if(velocita+5<velocitaMax)
			velocita+=5;
		Console.WriteLine("velocità auto= "+velocita);
	}
}

public static void Main()
{
	Veicolo veicolo=new Auto();
	Auto auto=new Auto();

	veicolo.Accelera();
	auto.Accelera();
```

```
}
```

In questo secondo esempio, verrà invocato in entrambi i casi il metodo `Accelera` della classe `Auto`, come è lecito aspettarsi.

5.6.5 Il versionamento

Ma siamo sicuri di sapere quali metodi saranno sottoposti ad overriding? Non certamente, soprattutto se qualcun altro eredita una classe da quelle che noi abbiamo implementato, quindi non è detto che i metodi saranno stati dichiarati `virtual` nella classe base.
Come ci avverte il warning visto prima nel paragrafo relativo all'hiding, dobbiamo dichiarare il nuovo metodo con la parola chiave **new**, in questa maniera dichiariamo esplicitamente che il nuovo metodo `Accelera` nasconde il metodo della classe madre, o in parole povere abbiamo creato una nuova versione del metodo:

```
class Auto:Veicolo
{
	…
	public new void Accelera()
	{
		if(velocita+5<velocitaMax)
			velocita+=5;
		Console.WriteLine("velocità auto= "+velocita);
	}
}
```

Lo stesso problema può nascere quando ad esempio siamo noi ad avere derivato una classe da una esistente, come nel caso della classe `Auto` derivata dalla classe `Veicolo`, e vogliamo specializzarla con un metodo completamente nuovo:

```
class Auto:Veicolo
{
	…
	public void Sterza(bool b)
	{
		if(b)
			Console.WriteLine("sterza a destra");
		else
			Console.WriteLine("sterza a sinistra");
	}
}
```

Abbiamo specializzato la classe `Auto` aggiungendo un nuovo metodo `Sterza`. Supponiamo che a questo punto l'autore originario della classe madre aggiunga anch'esso un nuovo metodo per sterzare con il veicolo, dandogli sfortunatamente proprio la stessa firma ma con un'implementazione diversa Ad esempio

```
classVeicolo
{
	…
	public void Sterza(bool b)
	{
		if(b)
			Console.WriteLine("sterza a sinistra");
		else
			Console.WriteLine("sterza a destra");
	}
}
```

Quando tenteremo di compilare nuovamente la nostra classe `Auto`, ci verrà ancora segnalato un warning, mentre l'autore della classe `Veicolo` andrà tranquillamente per la sua strada senza accorgersi di nulla.
In tal caso nessuno ci garantisce che tutto il codice che utilizza il metodo `Sterza` si comporterà come da noi previsto, ed i nostri oggetti auto potrebbero cominciare a sterzare al contrario! E non possiamo nemmeno rinominare il metodo perché questi oggetti con ogni probabilità non

funzionerebbero più se continuassero a cercare ancora il metodo `Sterza`, a meno di non correggere tutte le chiamate al metodo, ma potremmo avere rilasciato qualche libreria con tali chiamate e quindi anche questa soluzione è impraticabile. Anche questa volta però possiamo utilizzare la keyword `new` applicandola al nostro metodo `Sterza`, in questa maniera nascondiamo intenzionalmente il metodo `Sterza` della classe Veicolo ed ogni variabile di classe `Auto` utilizzerà la versione corretta del metodo.

```
class Auto:Veicolo
{
	…
	public new void Sterza(bool b)
	{
		…
	}
}
```

Possiamo utilizzare la parola chiave `new` anche sui campi, statici e non, di una classe, mentre non è possibile applicare ai campi la keyword `virtual`. E naturalmente non è possibile usare contemporaneamente sullo stesso metodo `override` e `new`.
La parola chiave `new` inoltre può essere anche applicata alla dichiarazione di una classe, in tal modo si intende esplicitamente che essa maschera i metodi ereditati dalla propria classe base, oppure può essere applicata ad una classe innestata con lo stesso nome di una classe innestata nella classe madre:

```
using System;
public class ClasseBase
{
   public class ClasseInnestata
   {
      public int x = 200;
   }
}

public class ClasseDerivata : ClasseBase
{
   new public class ClasseInnestata // la classe nested nasconde la classe della classe base
   {
      public int x = 100;
   }

   public static void Main()
   {
      ClasseInnestata S1  = new ClasseInnestata ();

      // Crea un oggetto della classe nascosta
      ClasseBase.ClasseInnestata S2 = new ClasseBase.ClasseInnestata ();

      Console.WriteLine(S1.x);
      Console.WriteLine(S2.x);
   }
}
```

In questo esempio viene creata un'istanza di ognuna delle classi nested, e l'output sarà in questo caso:

```
100
200
```

5.6.6 Chiamare i metodi della classe base

Per accedere a membri dichiarati nella classe madre è possibile utilizzare la parola chiave **`base`**, già utilizzata per invocarne i costruttori. Ad esempio supponiamo di avere le due seguenti classi:

```
class Prodotto
{
	protected decimal prezzo;

	public Prodotto(decimal prezzo)
	{
		this.prezzo=prezzo;
	}
```

```
      public virtual decimal CalcolaPrezzo()
      {
            return prezzo;
      }
}

class ProdottoScontato:Prodotto
{
      private decimal sconto=10;//sconto 10%

      public ProdottoScontato(decimal prezzo):base(prezzo)
      {
      }

      public override decimal CalcolaPrezzo()
      {
            return (1-sconto/100)*base.CalcolaPrezzo();
      }
}
```

Sia la classe `Prodotto` che la classe `ProdottoScontato` hanno un metodo per calcolare il prezzo, e nel caso della seconda il prezzo viene calcolato sulla base di quello restituito appunto dalla classe Prodotto, cioè con la chiamata:

```
base.CalcolaPrezzo();
```

otteniamo il prezzo della classe Prodotto e quindi vi applichiamo lo sconto. Quindi se istanziamo due prodotti così e ne visualizziamo il prezzo:

```
Prodotto p1=new Prodotto(100);
ProdottoScontato p2=new ProdottoScontato(100);
Console.WriteLine("Prezzo p1={0}",p1.CalcolaPrezzo());
Console.WriteLine("Prezzo p2={0}",p2.CalcolaPrezzo());
```

otterremo l'output seguente:

```
Prezzo p1=100
Prezzo p2=90.0
```

5.6.7 Classi astratte

Una classe viene detta astratta quando essa non può venire istanziata, cioè non è possibile creare un oggetto di tale classe. Una classe di tale genere serve quindi come template per altre classi, che saranno derivate da essa, per illustrare ad esempio un comportamento comune che esse dovranno avere, o, con una terminologia più appropriata, per descrivere un'interfaccia comune per le classi derivate.
Il linguaggio C# consente di creare classi astratte, utilizzando il modificatore **`abstract`** nella sua dichiarazione:

```
public abstract class Figura
{}
```

Una classe `abstract` può, ma non è obbligatorio, contenere metodi anch'essi `abstract`, dei quali non viene fornito il corpo, e quindi il funzionamento interno. Viceversa una classe che possiede dei metodi astratti deve obbligatoriamente essere dichiarata come astratta, altrimenti il compilatore ci avviserà con un errore.

```
public abstract class Figura
{
      protected float fArea;

      public abstract void CalcolaArea();

      public virtual string GetTipoFigura()
      {
            return "Figura generica";
      }
```

```
}
```

Una classe che deriva da una classe `abstract`, deve fornire una implementazione dei suoi metodi astratti, a meno che non sia anch'essa una classe astratta, ciò implica il fatto che tali metodi sono implicitamente dei metodi virtuali, e quindi non è necessario nè possibile dichiarare un metodo `abstract virtual`, mentre i metodi delle classi figlie che implementano i metodi astratti dovranno essere qualificati con il modificatore `override`:

```
public class Quadrato
{
	float lato;

	public override void CalcolaArea()
	{
		fArea= lato*lato;
	}

	public override string GetTipoFigura()
	{
		return "Quadrato";
	}

}
```

E' possibile anche dichiarare delle proprietà astratte, non fornendo l'implementazione `get` e/o `set`, ad esempio:

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

definisce una proprietà `Area` astratta, ed una classe che deriva dalla classe che la contiene deve fornire un'implementazione di entrambi i rami `get` e `set`:

```
public override float Area
{
	get
	{
		return fArea;
	}
	set
	{
		fArea=value;
	}
}
```

5.6.8 Classi sealed

Una classe **sealed** è una classe che non può essere derivata da una classe figlia, in tale maniera si blocca la catena di ereditarietà della classe. In genere il modificatore **`sealed`** si usa nei casi in cui non si vuole che la classe sia estesa in maniera accidentale, oppure non si vuole che qualcuno possa creare una propria classe derivandola da una esistente, ad esempio per motivi commerciali. Come esempio reale, notiamo che la classe `System.String` è una classe `sealed`, per evitare che si possa ereditare da essa una diversa implementazione delle stringhe.
Una classe `sealed` si crea semplicemente aggiungendo il modificatore `sealed` alla dichiarazione della classe:

```
class Triangolo
{
	//…
}

sealed class TriangoloScaleno:Triangolo
{

}
```

```
//errore di compilazione
class TriangoloScalenoIsoscele:TriangoloScaleno
{
}
```

La dichiarazione dell'ultima classe `TriangoloScalenoIsoscele` darà in compilazione un errore, in quanto si è tentato di derivarla dalla classe sealed `TriangoloScaleno`.
Una classe astratta non può essere naturalmente anche sealed, non avrebbe senso d'altronde non essendo a questo punto né istanziabile né estendibile da una classe figlia.
E' possibile anche applicare il modificatore `sealed` a singoli metodi, anche se è una pratica più rara. Un metodo `sealed` non può avere ulteriori override in classi derivate, ed ha senso quindi solo se è esso stesso un override di un metodo della classe base.

5.7 Interfacce

Un'interfaccia è un contratto applicabile ad una classe. Una classe che si impegna a rispettare tale contratto, cioè ad implementare un'interfaccia, promette che implementerà tutti i metodi, le proprietà, gli eventi, e gli indicizzatori, esposti dall'interfaccia. Il concetto è molto simile a quello di classe astratta, ma nel caso di una interfaccia non è possibile fornire alcuna implementazione dei membri di cui sopra, né dunque contenere dei campi, inoltre il tipo di relazione di una classe derivante da una classe astratta è sempre una relazione di generalizzazione, nel caso invece di una classe che implementa un'interfaccia, la relazione è appunto di implementazione. Ad esempio un `Triciclo` potrebbe derivare dalla classe astratta `VeicoloAPedali`, e potrebbe invece implementare un'interfaccia che espone i metodi per guidarlo, interfaccia implementabile anche da altri tipi, ad esempio dalla classe `Shuttle` che non è un `VeicoloAPedali`, ma deve essere guidabile, quindi è naturale pensare ad una interfaccia `IGuidabile`.
Un'interfaccia viene dichiarata utilizzando la parola chiave **`interface`**, in maniera perfettamente analoga a quanto visto per le classi, cioè con la seguente sintassi

```
[modificatore_accesso] interface InomeInterfaccia [: ListaInterfacceBase]
{
   //corpo dell'interfaccia
   .
   .
   .
}
```

Le linee guida .NET suggeriscono di chiamare le interfacce con un nome del tipo `INomeInterfaccia`, un esempio reale è l'interfaccia `IComparable` implementata dalle classi per cui si vuole fornire un metodo di comparazione e ordinamento.
Tutti i membri esposti da un'interfaccia sono implicitamente `public virtual`, in quanto servono appunto solo da segnaposto per quelle che saranno poi le loro implementazioni. Supponiamo di voler definire una interfaccia `IAtleta`. Un atleta deve poter correre e saltare. Quindi l'interfaccia potrebbe essere questa:

```
public interface IAtleta
{
      void Corre();
      void Salta();
}
```

Se vogliamo che una nostra classe implementi questa interfaccia, essa deve fornire un'implementazione public dei due metodi:

```
class Calciatore:IAtleta
{
      private string nome,cognome, squadra;
```

```
	public void Corre()
	{
		Console.WriteLine("Sto correndo");
	}

	public void Salta()
	{
		Console.WriteLine("Sto saltando");
	}
}
```

Una classe può anche implementare più interfacce contemporaneamente, in questo modo un oggetto è una sola cosa, ma si comporta in modi diversi, in maniera polimorfica.
Ad esempio se volessimo confrontare due calciatori, potremmo implementare contemporaneamente alla `IAtleta`, l'interfaccia `IComparable` che espone il seguende metodo `CompareTo`:

```
int CompareTo(object obj);
```

Il valore di ritorno può essere minore,uguale o maggiore di zero, ed è utilizzato per dire se l'istanza è minore, uguale o maggiore di obj, secondo un certo parametro di ordinamento.
Quindi possiamo confrontare due Calciatori, confrontando nome, cognome e squadra di appartenenza:

```
class Calciatore:IAtleta, IComparable
{
	private string nome, cognome, squadra;

	public void Corre()
	{
		Console.WriteLine("Sto correndo");
	}

	public void Salta()
	{
		Console.WriteLine("Sto saltando");
	}

	public int CompareTo(object obj)
	{
		if(obj is Calciatore)
		{
			Calciatore c=(Calciatore)obj;
			if(nome==c.nome && cognome==c.cognome && squadra==c.squadra)
				return 0;
		}
		return -1;
	}
}
```

C# consente l'ereditarietà singola delle classi, ma anche l'ereditarietà multipla delle interfacce, vale a dire che un'interfaccia può derivare da più interfacce contemporaneamente, ad esempio supponiamo di voler scrivere un'interfaccia `IAtletaUniversale` e che definisce il comportamento di un'atleta con più capacità atletiche, partendo da interfacce che definiscono le singole capacità:

```
interface INuotatore
{
	void Nuota();
}

interface ISciatore
{
	void Scia();
}

interface ITennista
{
	void Dritto();
	void Rovescio();
}
```

```
interface IAtletaUniversale: INuotatore, ISciatore,ITennista,IAtleta
{
      void Mangia();
      void Dormi();
}
```

Una classe che implementi l'interfaccia `IAtletaUniversale`, deve non solo fornire un corpo per i metodi `Mangia` e `Dormi`, ma anche tutti quelli delle interfacce da cui `IAtletaUniversale` deriva:

```
class AtletaCompleto:IAtletaUniversale
{
      public void Mangia()
      {
            Console.WriteLine("Mangio");
      }
      public void Dormi()
      {
            Console.WriteLine("Dormo");
      }
      public void Nuota()
      {
            Console.WriteLine("Nuoto");
      }
      public void Scia()
      {
            Console.WriteLine("Scio");
      }
      public void Dritto()
      {
            Console.WriteLine("Colpisco con il dritto");
      }
      public void Rovescio()
      {
            Console.WriteLine("Colpisco di rovescio");
      }
      public void Corre()
      {
            Console.WriteLine("Corro");
      }
      public void Salta()
      {
            Console.WriteLine("Salto");
      }
}
```

Se una classe deriva da una classe base ed implementa qualche interfaccia, è necessario che nella dichiarazione venga prima la classe, seguita dall'elenco delle interfacce.
Potrebbe naturalmente accadere che più interfacce espongano un metodo con la stessa firma, ad esempio supponiamo che le interfacce `ITennista` e `ISciatore` espongano un metodo `Esulta`, è necessario risolvere l'ambiguità nella classe che implementa le due interfacce, indicando esplicitamente quale metodo si sta implementando:

```
class AtletaCompleto:IAtletaUniversale
{
      .
      .
      .
      void ITennista.Esulta()
      {
            Console.WriteLine("Sono un grande tennista!");
      }

      void ISciatore.Esulta()
      {
            Console.WriteLine("Sono il miglior sciatore!");
      }
}
```

notate che l'implementazione dei metodi non può essere stavolta `public`, quindi se vogliamo utilizzare i due metodi, dobbiamo prima effettuare un cast dell'oggetto verso una delle due interfacce, in modo da risolvere il tipo da utilizzare, ad esempio:

```
AtletaCompleto atleta=new AtletaCompleto();
atleta.Esulta; //Errore, la classe non ha un metodo Esulta
((ISciatore)atleta).Esulta(); //OK esulta da sciatore
((ITennista)atleta).Esulta(); //OK esulta da tennista
```

6 Classi fondamentali

In questo capitolo verranno illustrate alcune delle classi fondamentali fornite dalla Base Class Library, ed il loro utilizzo in C# in situazioni valide in molte se non tutte le aree di programmazione.

6.1 La classe System.Object

Come già detto la classe System.Object è la classe da cui implicitamente ogni altra deriva se non viene specificata nessuna altra classe, ed abbiamo inoltre visto che la keyword `object` non è altro che un alias per `System.Object`, dunque perfettamente equivalente.
Se dunque scriviamo:

```
class Pippo
{
   …
}
```

è come scrivere:

```
class Pippo: System.Object
{
   …
}
```

La classe `System.Object` fornisce dei metodi `public` o `protected`, statici e di istanza, che dunque ogni altra classe eredita, e che può eventualmente ridefinire con un override se essi sono definiti anche come `virtual`.
La tabella seguente elenca tali metodi:

Metodo	Descrizione
`public virtual string ToString()`	Restituisce una stringa che rappresenta l'oggetto.
`public virtual int GetHashCode()`	Restituisce un intero, utilizzabile come valore di hash, ad per la ricerca dell'oggetto in un elenco di oggetti.
`public virtual bool Equals(object o)`	Effettua un test di uguaglianza con un'altra istanza della classe.
`public static bool Equals(object a,object b)`	Effettua un test di uguaglianza fra due istanze della classe.
`public static bool ReferenceEquals(object a,object b)`	Effettua un test di uguaglianza per verificare se due riferimenti si riferiscono alla stessa istanza della classe.
`public Type GetType()`	Restituisce un oggetto derivato da System.Type che rappresenta il tipo dell'istanza.
`protected object MemberwiseClone()`	Effettua una copia dei dati contenuti nell'oggetto, creando un'altra istanza.
`protected virtual void Finalize()`	Distruttore dell'istanza.

6.1.1 Il metodo ToString

Il metodo `ToString(…)` è probabilmente uno dei più utilizzati in qualsiasi tipo di applicazione, in quanto serve a fornire una rappresentazione testuale del contenuto di un oggetto. Esso è un metodo

`virtual` nella classe `System.Object`, dunque ogni classe può fornire un override di esso, in modo da restituire una stringa significativa e specifica dell'oggetto. Ad esempio i tipi numerici predefiniti di C# forniscono tale override in modo da restituire il valore numerico sotto forma di stringa.

```
int i=100;
string str=i.ToString(); // restituisce "100"
```

se non ridefiniamo il metodo nelle nostre classi, verrà invocato il metodo della classe System.Object, che restituirà una rappresentazione più generica.

```
namespace TestObject
{
      Class Studente
      {
            int matricola;
            string cognome;
            string nome;

            public Studente(int m, string n, string c)
            {
                  matricola=m;
                  cognome=c;
                  nome=n;
            }

            static void Main()
            {
                  Studente studente=new Studente(1234,"pinco", "pallino");
                  Console.WriteLine(studente);
            }
      }
}
```

La chiamata `Console.WriteLine(studente)` invoca al suo interno il metodo `studente.ToString()`, e in questo caso stamperà una stringa del tipo:

```
TestObject.Studente
```

Mentre magari ci saremmo aspettati o vorremmo, una stringa che contenga matricola, nome e cognome dello studente. Per far ciò, come abbiamo imparato nel precedente capitolo, scriviamo nella classe `Studente` un override del metodo `ToString()`:

```
public override string ToString()
{
      return "Studente "+matricola+" - "+cognome+" "+nome;
}
```

In questa maniera l'istruzione

```
Console.WriteLine(studente);
```

stamperà la stringa:

```
Studente 1234 - pallino pinco
```

6.1.2 I metodi Equals e ReferenceEquals

I metodi `Equals` effettuano il confronto di due istanze, e sono fondamentali in quanto vengono richiamati in diverse altre classi per testare l'uguaglianza di due oggetti, ad esempio nelle collezioni, che vedremo nel prossimo capitolo, il metodo di istanza `bool Equals(object o)` viene utilizzato per verificare se esse contengono o meno una certa istanza.
La classe System.Object tuttavia fornisce un'implementazione del metodo `Equals` che verifica semplicemente l'uguaglianza dei riferimenti.
Senza fornire un override del metodo, vediamo come si comporterebbe il seguente codice:

```
Studente s1=new Studente(1234,"pinco", "pallino");
Studente s2=s1;
Studente s3=new Studente(1234,"pinco", "pallino");
Console.WriteLine(s3.Equals(s1)); //stampa false
```

L'ultima riga stamperà il risultato `false`, dunque s3 e s1, pur rappresentando logicamente lo stesso Studente, vengono considerati diversi, in quanto si riferiscono a due istanze in memoria distinte della classe Studente.
Se invece scriviamo un metodo ad hoc nella classe `Studente`, facendo l'override del metodo `Equals` che ad esempio confronti la matricola:

```
public override bool Equals(object obj)
{
      if(obj is Studente)
      {
            return this.matricola==((Studente)obj).matricola;
      }
      return false;
}
```

otterremmo che il risultato del test di uguaglianza

```
s3.Equals(s1)
```

stavolta restituisca true.
Potrebbe essere necessario comunque dovere confrontare che siano uguali due riferimenti, cioè che in realtà due variabili referenzino la stessa istanza in memoria, a tal scopo la classe System.Object fornisce il metodo statico `ReferenceEquals`, che verifica appunto l'uguaglianza dei riferimenti di due oggetti. Ad esempio:

```
MiaClasse c1=new MiaClasse();
MiaClasse c2=c1;
Console.WriteLine("object.ReferenceEquals(c1,c2)= "+object.ReferenceEquals(c1,c2));//true
c2=new MiaClasse();
Console.WriteLine("object.ReferenceEquals(c1,c2)= "+object.ReferenceEquals(c1,c2));//false
```

La prima chiamata a `object.ReferenceEquals(c1,c2)` restituisce true in quanto c2 e c1 puntano allo stesso oggetto in memoria, mentre dopo avere costruito una nuova istanza ed assegnatala a c2, la stessa chiamata restituisce false, in quanto ora abbiamo appunto due istanze diverse della classe MiaClasse.
Per curiosità possiamo fare lo stesso esperimento, chiamando il metodo `ReferenceEquals` su una variabile di tipo valore, ad esempio un intero:

```
int i=0;
Console.WriteLine(object.ReferenceEquals(i,i));
```

In questo caso il confronto dei riferimenti restituirà sempre false, anche confrontando la variabile intera i con se stessa. Infatti gli argomenti del metodo ReferenceEquals devono essere di classe System.Object, quindi si ha il boxing automatico della variabile intera i in due diversi oggetti nella memoria heap.

6.1.3 Il metodo GetHashCode

Il metodo `GetHashCode` restituisce un valore intero, utilizzabile per lavorare con collezioni tipo tabelle hash, e memorizzare oggetti tipo chiave/valore. Una hash table è formata da tante locazioni, e la locazione in cui memorizzare o ricercare una coppia chiave/valore è determinata dal codice hash ricavato dalla chiave. Ad esempio qunado dobbiamo ricercare un dato oggetto, viene ricavato il codice hash dalla chiave, e nella posizione indicata da tale codice, verrà poi ricercata la chiave stessa, e quindi se essa viene trovata, può essere ricavato il valore corrispondente.
In particolare la Base Class Library fornisce una classe `Hashtable`, che utilizza il metodo `GetHashCode` per ricavare il codice hash, ed il metodo `Equals` per confrontare gli oggetti da memorizzare o già

memorizzati. Ed infatti se effettuiamo nella nostra classe un override del metodo Equals, il compilatore ci avviserà con un warning se non implementeremo anche l'override del metodo `GetHashCode`.
L'implementazione del metodo `GetHashCode` richiede il ritorno di un valore int, e dunque è necessario trovare un algoritmo che restituisca dei valori hash con una distribuzione casuale, con una certa velocità per questioni di performance, e naturalmente che restituisca valori hash uguali per oggetti equivalenti. Ad esempio due stringhe uguali dovrebbero restituire lo stesso codice hash.
La documentazione .NET contiene un esempio abbastanza significativo per una struct Point così fatta:

```
public struct Point
{
   public int x;
   public int y;

   public override int GetHashCode()
   {
      return x ^ y;
   }
}
```

Il metodo precedente restituisce lo XOR fra le coordinate x e y del punto.
Qualche sviluppatore preferisce richiamare direttamente il metodo della classe base Object, soprattutto se le funzionalità di hash non sono necessarie:

```
public override int GetHashCode()
{
   return base.GetHashCode();
}
```

6.1.4 Il metodo GetType

Il metodo `GetType` restituisce un'istanza della classe System.Type, che può essere utilizzata per ottenere una grande varietà di informazioni a run-time sul tipo corrente di un oggetto, ad esempio il namespace, il nome completo, i metodi di una classe, quindi è il punto di accesso alla cosiddetta tecnologia di Reflection, la quale consente di esplorare il contenuto di un tipo qualunque, ma ne riparleremo fra qualche capitolo.

```
System.Windows.Forms.Button b=new System.Windows.Forms.Button("Premi");
System.Type t=b.GetType();
Console.WriteLine("Il tipo di b è "+t.FullName);
```

naturalmente ciò vale anche per i tipi valore:

```
int i=12;
System.Type tInt=i.GetType();
Console.WriteLine("Il tipo di i è "+tInt.FullName);
Console.WriteLine("BaseType di i: "+tInt.BaseType.FullName);
```

6.1.5 Clonare un oggetto

La classe System.Object fornisce il metodo protected `MemberwiseClone` che consente di ottenere una copia dell'oggetto sul quale viene invocato. E' bene notare che verrà effettuata la copia bit a bit di tutti i campi non statici di un tipo valore, mentre per i campi di tipo riferimento ciò che viene copiato è solo il riferimento, e non l'oggetto a cui si riferisce il campo stesso. La copia dell'oggetto restituita, quindi avrà dei campi che si riferiscono agli oggetti a cui si riferiscono anche i campi dell'oggetto originale.
Scriviamo ad esempio una classe Cane, con due attributi di tipo valore ed uno di tipo Coda, che è un tipo riferimento implementato anche esso da noi. Forniamo inoltre alla classe Cane un metodo per effettuare la cosiddetta shallow copy di una sua istanza:

```
class Cane
```

```
{
      public string nome;
      public int età;
      public Coda coda;

      public Cane(string nome,int età)
      {
            coda=new Coda(10);
      }

      public Cane Copia()
      {
            return (Cane)this.MemberwiseClone();
      }

      public override string ToString()
      {
            return "Sono il cane "+nome+" di "+età+" anni ed una coda lunga "+coda.Length  ;
      }


}

class Coda
{
      private int length;

      public Coda(int l)
      {
            this.length=l;
      }

      public int Length
      {
            get
            {
                  return length;
            }
            set
            {
                  length=value;
            }
      }
}
```

Creiamo ora un istanza della classe Cane e la cloniamo:

```
public class TestObjectCloning
{
      public static void Main()
      {
            Cane fido=new Cane("Fido",2);
            Console.WriteLine(fido.ToString());
            Cane cloneDiFido=fido.Copia();
            Console.WriteLine(cloneDiFido.ToString());
            fido.età+=1;
            fido.coda.Length+=2;
            Console.WriteLine(fido.ToString());
            Console.WriteLine(cloneDiFido.ToString());
            Console.Read();
      }
}
```

L'output dell'esempio precedente è il seguente:

```
Sono il cane  di 0 anni ed una coda lunga 10
Sono il cane  di 0 anni ed una coda lunga 10
Sono il cane  di 1 anni ed una coda lunga 12
Sono il cane  di 0 anni ed una coda lunga 12
```

Quando variamo il campo età dell'oggetto fido, tale cambiamento non si riflette sull'istanza cloneDiFido, in quanto le due istanze possiedono due copie separate del campo, mentre l'assegnazione ad un campo di tipo riferimento come:

```
fido.coda.Length+=2;
```

si ripercuote su entrambe le istanze, perchè il metodo MemberwiseClone ha copiato solo il riferimento alla stessa unica istanza della classe Coda.
La modalità di copia realizzata tramite MemberwiseClone è detta **shallow copy**.
Nei casi in cui servisse una copia profonda dell'oggetto, cioè la **deep copy**, è necessario che la classe implementi l'interfaccia `ICloneable`, che espone l'unico metodo `object Clone`, in cui possiamo realizzare sia una deep copy che una shallow copy. Ad esempio potremmo fare ciò con la classe Cane:

```
public object Clone()
{
      Cane clone=new Cane(this.nome,this.età);
      clone.coda=new Coda(this.coda.Length);
      return clone;
}
```

In tal modo le due code sono due istanze distinte della classe Coda, e quindi i cambiamenti sulla coda di fido non influenzeranno la coda del clone2.

6.1.6 Distruzione di un oggetto: Finalize

Quando un oggetto termina il suo ciclo di vita, perchè ad esempio esce dal suo scope, esso può essere rimosso dalla memoria, per mezzo del garbage collector. Molti oggetti però potrebbero utilizzare delle risorse unmanaged, ad esempio aprire dei file, delle connessioni di rete, o verso un database, ed è necessario quindi prevedere un meccanismo che liberi anche tali risorse.
Questo meccanismo può essere chiamato finalizzazione di un oggetto, e dunque un oggetto che necessita di risorse unmanaged deve supportarlo. Quando il Garbage Collector stabilisce che un oggetto può essere rimosso dalla memoria, esso cerca ed eventualmente invoca il metodo Finalize ereditato dalla classe System.Object, in teoria dunque ogni classe potrebbe fornire un override del metodo:

```
class MiaClasse
{
      protected override void Finalize()
      {
            //libera le risorse unmanaged
      }
}
```

Tentando un simile override però è il compilatore che ci avverte che non è possibile fornire un override del metodo Finalize.
Lasciando perdere il motivo di ciò, il compilatore ci informa inoltre che invece di implementare l'override in modo "esplicito", C# usa una sintassi speciale, quella del distruttore che abbiamo già incontrato, il metodo sarà dunque scritto come:

```
class MiaClasse
{
      ~MiaClasse()
      {
            //libera le risorse unmanaged
      }
}
```

In questo caso il compilatore produrrà un codice analogo al precedente, con la sola aggiunta della gestione di eventuali eccezioni nella finalizzazione stessa. L'ultima cosa da sottolineare riguardo all'argomento è quella di usare il distruttore solo quando strettamente necessario, in quanto potrebbe notevolmente influire sulle prestazioni del garbage collector e perchè non si ha mai l'assoluta certezza che le risorse siano state effettivamente liberate a causa del non determinismo del meccanismo stesso di garbage collection. E' meglio allora liberare le risorse in modo esplicito, tramite un metodo ad hoc (il metodo Finalize è protected quindi non invocabile da oggetti di una classe differente), ad esempio utilizzando il cosiddetto pattern Dispose esposto nel prossimo paragrafo.

6.2 Il pattern Dispose

Il pattern **Dispose** permette di definire le funzionalità da fornire e da implementare in una classe per ottenere in maniera deterministica ed esplicita la distruzione delle sue istanze e la liberazione delle relative risorse occupate.
Nel far ciò ci viene in aiuto l'interfaccia **IDisposable** che espone un unico metodo:

```
public void IDisposable
{
      void Dispose();
}
```

Implementando in maniera efficace nella nostra classe un metodo pubblico Dispose(), esso potrà essere richiamato ogni volta che è necessario liberare le risorse associate all'oggetto. In genere il pattern Dispose è utilizzato in congiunzione al metodo Finalize, o meglio, come abbiamo visto nel precedente paragrafo, al distruttore della classe. Il seguente esempio illustra un esempio classico di implementazione del pattern Dispose:

```
public class Risorsa: IDisposable
{
      // handle di una risorsa unmanaged, ad esempio di un file
      private IntPtr handle;
      // una Icon è una risorsa managed.
      private Icon icona;

      private bool disposed = false;
      public Risorsa(IntPtr handle,string iconfile)
      {
            this.handle = handle;
            this.icona=new Icon(iconfile);
      }

      // Implementazione di IDisposable.
      public void Dispose()
      {
            Console.WriteLine("Dispose()");
            Dispose(true);

            // Informa il CLR di non invocare il distruttore
            // perchè le risorse sono state liberate con il Dispose precedente
            GC.SuppressFinalize(this);
      }

      // se disposing è true, vengono liberate sia risorse Managed che unmanaged.
      // con disposing = false, il metodo viene richiamato internamente dal CLR,
      // esattamente dal distruttore, che avrà già liberato le risorse managed,
      // dunque qui vengono ripulite solo quelle unmanaged.
      private void Dispose(bool disposing)
      {
            Console.WriteLine("Dispose({0})",disposing);
            if(!this.disposed)
            {
                  if(disposing)
                  {
                        // l'icona è una risorsa managed
                        icona.Dispose();
                  }

                  // libera opportunamente le risorse unmanaged
                  CloseHandle(handle);
                  handle = IntPtr.Zero;
            }

            //per evitare che Dispose venga eseguito più volte
            disposed = true;
      }

      [System.Runtime.InteropServices.DllImport("Kernel32")]
      private extern static Boolean CloseHandle(IntPtr handle);

      // Il distruttore viene invocato automaticamente
```

```
      // e solo se non è stato invocato il metodo Dispose
      ~Risorsa()
      {
            Console.WriteLine("~Risorsa()");
            Dispose(false);
      }
}
```

La classe Risorsa implementa l'interfaccia IDisposable, dunque contiene un metodo pubblico Dispose. Tale metodo può essere invocato direttamente da altri punti del codice.
All'interno del metodo viene invocato un ulteriore metodo Dispose(bool), nel quale possono essere liberate sia le risorse managed, come ad esempio un campo Icon, nel caso in cui il parametro disposing è true, o soltanto quelle unmanaged, ad esempio un handle di un file.
La classe possiede anche un distruttore che eventualmente viene richiamato in maniera automatica dal Garbage Collector, al cui interno viene ancora invocato il metodo Dispose con parametro false, in quanto in tal caso delle risorse managed se ne sarà occupato appunto il progesso di garbage collection.
Da notare infine che nel metodo Dispose() viene invocato anche il metodo GC.SuppressFinalize che informa il CLR che non è più necessaria l'invocazione del distruttore in quanto è appunto già stato invocato un metodo di pulitura delle risorse.
Ma perchè implementare un'interfaccia solo per aggiungere ad una classe un metodo public Dispose()? D'altronde potremmo inserire un metodo simile in qualsiasi classe e chiamarlo quando vogliamo, inoltre lo sviluppatore potrebbe anche dimenticare di invocare esplicitamente il metodo Dispose.
Il motivo c'è ed è molto semplice. Ricordate la parola chiave using? Esiste ancora un altro modo per utilizzarla, e questo modo garantisce ed automatizza la chiamata del metodo Dispose quando l'oggetto non serve più è può essere distrutto.

```
using(Risorsa res=new Risorsa())
{
      //uso l'oggetto res
}// <-- qui il CLR invoca automaticamente il metodo res.Dispose()
```

Se infatti proviamo ad utilizzare la classe Risorsa con il metodo precedente:

```
public static void Main()
{
      FileStream fs=File.OpenRead(@"C:\temp\tmp.txt");
      using(Risorsa res=new Risorsa(fs.Handle,@"C:\temp\App.ico"))
      {
            Console.WriteLine("Ho costruito una {0}",res.ToString());
      }
}
```

L'output del programma sarà come ci aspettiamo:

```
Ho costruito una SourceCode.Capitolo6.Risorsa
Dispose()
Dispose(True)
```

mentre se ad esempio non invochiamo esplicitamente il Dispose o non usiamo il blocco using è il CLR che si occupa di invocare il distruttore:

```
public static void Main()
{
      FileStream fs=File.OpenRead(@"C:\temp\tmp.txt");
      Risorsa res2=new Risorsa(fs.Handle,@"C:\temp\App.ico");
}
```

In questo caso infatti avremo un output differente:

```
~Risorsa()
Dispose(False)
```

6.3 La classe System.String

Senza dubbio il tipo stringa è uno dei più utilizzati in qualunque linguaggio di programmazione ed in qualunque tipologia di applicazione.
Una stringa rappresenta un insieme immutabile ed ordinato di caratteri, e precisamente è in .NET una sequenza di caratteri unicode, rappresentati da istanze della struttura System.Char.
In C# il tipo **System.String** è un tipo riferimento, derivato da System.Object, ma nonostante ciò non è possibile costruire una stringa utilizzando l'operatore new, e quindi la stringa è trattata come un tipo primitivo, cioè è sufficiente una semplice assegnazione di un literal ad una variabile di tipo string:

```
string str=new String("Hello"); //errore
string str2="Hello"; //OK
```

Le stringhe, come detto sono immutabili, non è possibile quindi modificare una stringa od uno dei suoi caratteri, a run-time. Facciamo un esempio, il metodo ToLower della classe String prende come parametro una stringa e la converte in minuscolo:

```
string str="HELLO";
Console.WriteLine(str.ToLower()); //stampa 'hello'
```

ciò che accade è che viene creata una nuova stringa "hello", che sarà velocemente distrutta dal garbage collector dopo essere stata stampata sullo schermo, mentre l'originale str resta uguale a "HELLO".
La classe System.String fornisce proprietà e metodi molto potenti ed allo stesso tempo semplici ed intuitivi per trattare le stringhe. Nei prossimi sottoparagrafi tratteremo i principali fra tali metodi.

6.3.1 Esaminare una stringa

Per ottenere la lunghezza di una stringa è possibile utilizzare la proprietà `Length`, mentre la proprietà `Chars` restituisce i caratteri che costituiscono la stringa stessa. In particolare quest'ultima proprietà è l'indicizzatore della classe String, dunque attraverso l'operatore [ ] si accede al carattere che si trova all'indice specificato.

```
string str="pippo";
int lung=str.Length;
for(int i=0;i<lung;i++)
{
      Console.WriteLine("Carattere {0}={1}", i, str[i]);
}
```

Se l'indice specificato è maggiore o uguale alla lunghezza della stringa, o è negativo, verrà generata un'eccezione IndexOutOfRangeException
Il metodo `ToCharArray` permette di ottenere tutti o parte dei caratteri che formano la stringa, sotto forma di un array Char[].

```
string str="hello";
char[] chars=str.ToCharArray();
foreach(char c in chars)
{
      Console.WriteLine(c);
}
```

Per sapere se una stringa contiene un carattere o una sottostringa, è possibile utilizzare i metodi `IndexOf` che restituisce l'indice della prima occorrenza, oppure `LastIndexOf` che restituisce invece l'indice dell'ultima occorrenza. Entrambi i metodi invece restituiscono il valore -1 se non viene trovato il valore ricercato.

```
string str="hello world";
Console.WriteLine(str.IndexOf("world"));// stampa 6
Console.WriteLine(str.IndexOf("l",5));//stampa 9
```

```
Console.WriteLine(str.IndexOf('o'));//stampa 4
Console.WriteLine(str.LastIndexOf('l'));//stampa 9
Console.WriteLine(str.LastIndexOf("or",5,3));//stampa -1
```

Come avrete notato, sono disponibili diversi overload dei metodi, che permettono di far partire la ricerca ad un dato indice, o di limitarla ad un certo numero di caratteri.
Analogamente, per mezzo dei metodi `IndexOfAny` e `LastIndexOfAny` è possibile ricavare la posizione all'interno di una stringa di uno o più caratteri contenuti in un array di char.

```
string str="ricerca nella stringa";
string seek="abc";
char[] chars=seek.ToCharArray();

int j=str.IndexOfAny(chars);
//restituisce 2, indice della prima c di 'ricerca'
Console.WriteLine("str.IndexOfAny({0})={1}",seek,j);

j=str.LastIndexOfAny(chars);
//restituisce 20, indice della ultima a della stringa str
Console.WriteLine("str.LastIndexOfAny({0})={1}",seek,j);
```

Esistono anche altri due overload dei metodi, il primo prende in ingresso un altro parametro intero, che indica l'indice di partenza della ricerca:

```
j=str.IndexOfAny(chars,10);
//restituisce 12, indice della a di 'nella'
Console.WriteLine("str.IndexOfAny({0},{1})={2}",seek,10,j);
```

Mentre il secondo overload prende un terzo parametro, ancora intero, che indica il numero di caratteri a partire dall'indice di partenza, in cui cercare uno dei caratteri:

```
j=str.IndexOfAny(chars,10,3);
//restituisce ancora 12, come nel precedente
Console.WriteLine("str.IndexOfAny({0},{1},{2})={3}",seek,10,3,j);
```

Stesso significato hanno gli overload del metodo LastIndexOfAny:

```
j=str.LastIndexOfAny(chars,10);
//restituisce 6, indice della a finale di 'ricerca', che è l'ultima occorrenza trovata nei primi
10 //caratteri
Console.WriteLine("str.LastIndexOfAny({0},{1})={2}",seek,10,j);

j=str.LastIndexOfAny(chars,10,3);
//restituisce -1, non trova nessuno dei caratteri 'abc' negli ultimi 3 caratteri dei primi 10
della //stringa
Console.WriteLine("str.LastIndexOfAny({0},{1},{2})={3}",seek,10,3,j);
```

6.3.2 Confronto fra stringhe

La classe String fornisce diversi metodi public, statici e non, per compiere una delle operazioni più comuni quando si ha a che fare con le stringhe, cioè il confronto.
Il primo fra questi è naturalmente il metodo `Equals`, fornito sia in una versione statica che in una versione d'istanza. Il metodo restituisce il valore true se due stringhe sono formate dallo stesso insieme di caratteri, e nel caso del metodo statico, esso prima verifica se due stringhe si riferiscono allo stesso oggetto, in tal caso restituisce immediatamente true, incrementando la velocità di esecuzione.

```
string s1="hello";
string s2="world";
string s3="he"+"llo";
Console.WriteLine(s1.Equals(s2));//false
Console.WriteLine(s1.Equals(s3));//true
Console.WriteLine(String.Equals(s1,s2));//false
Console.WriteLine(s1==s3);//true
```

Come si è visto, allo stesso scopo la classe System.String fornisce gli overload dei due operatori == e !=, che internamente non fanno altro che utilizzare la versione statica del metodo Equals.
Se lo scopo del confronto non è semplicemente quello di determinare l'uguaglianza di due stringhe, ma si vuole anche fornire un ordinamento alfabetico, ci vengono incontro i metodi `Compare`, `CompareTo` e `CompareOrdinal`.
Il primo è un metodo statico che determina come due stringhe devono essere ordinate una rispetto all'altra, restituendo un valore intero. Ad esempio confrontando due stringhe A e B il valore di ritorno potrà essere:

- Nullo se A è uguale a B
- Negativo se A è minore di B
- Positivo se A è maggiore di B

```
Console.WriteLine(String.Compare("A","B"));// -1, A è minore di B
Console.WriteLine(String.Compare("A","A")); // 0, A è uguale ad A
Console.WriteLine(String.Compare("B","A")); // 1, B è maggiore di A
Console.WriteLine(String.Compare("A",null)); // 1, A è maggiore di null
```

Il metodo Compare può essere utilizzato con un terzo parametro booleano, per specificare se il confronto debba essere fatto in modo case sensitive o meno.
Il metodo CompareOrdinal, anch'esso statico, è quasi identico al precedente, l'unica differenza è che effettua un confronto carattere per carattere, dunque, sebbene più veloce, in alcuni casi nei quali due stringhe sarebbero logicamente equivalenti, esso restituisce false se esse fossero scritte con caratteri diversi. Ciò accade ad esempio con parole straniere scritte in alfabeti che utilizzano particolari caratteri. Un esempio classico è dato dalle parole Strass e Straß:

```
bool b=String.CompareOrdinal("Strass", "Straß");//restituisce false
b=String.Compare("Strass", "Straß");//restituisce true
```

Il metodo Compare consente ad ogni modo di utilizzare un ulteriore parametro di tipo CultureInfo, per specificare secondo quale cultura, regione, nazione, o quant'altro, effettuare il confronto.
Il metodo CompareTo è, a differenza dei precedenti, un metodo d'istanza, che restituisce ancora un valore intero avente lo stesso significato visto nei casi precedenti.

```
string str="b";
Console.WriteLine(str.CompareTo("a"));//stampa 1
Console.WriteLine(str.CompareTo("c"));//stampa -1
Console.WriteLine(str.CompareTo("b"));//stampa 0
```

Di estrema utilità si rivelano anche i due metodi, complementari fra loro, `StartsWith` ed `EndsWith`, che restituiscono un valore booleano ad indicare rispettivamente se la prima o l'ultima parte di una stringa coincide con una data sottostringa passata come parametro:

```
str="hello world";
Console.WriteLine(str.StartsWith("hello"));//true
Console.WriteLine(str.EndsWith("world"));//true
```

6.3.3 Formattazione di un numero

Il metodo statico `String.Format` permette di formattare una stringa in maniera personalizzata. La formattazione di una stringa è il procedimento che implicitamente viene applicato anche quando ad esempio usiamo il metodo Console.WriteLine() per stampare una linea di testo sulla console.
Il codice:

```
Console.WriteLine("Il valore di pigreco è {0}",Math.PI);
```

produce l'output seguente (il separatore decimale dipende dalle impostazioni internazionali):

```
Il valore di pigreco è 3,14159265358979
```

Esso viene formattato sostituendo ai segnaposto del tipo {0},{1} e così via, contenuti nella stringa, i valori specificati nella lista separati dalla virgola. In questo caso l'unico parametro è Math.PI, corrispondente al segnaposto {0}.
Ciò che avviene dietro le quinte è l'invocazione del metodo String.Format, utilizzabile dunque per formattare stringhe in formati standard o personalizzati, e non solo per l'output su console, ma anche ad esempio per visualizzarle in una casella di testo, oppure salvarle su file di testo.
La formattazione è molto usata per visualizzare valori numerici in modo appropriato, ad esempio un double rappresentante una valuta deve essere rappresentato in modo diverso da un double che indica un valore ingegneristico in notazione esponenziale.
Nei segnaposti del tipo {n} incontrati spesso fino ad ora, possono essere aggiunte altre informazioni, e cioè l'allineamento e il formato da utilizzare.

```
{indice[,allineamento][:stringaDiFormato]}
```

La tabella seguente illustra le stringhe di formato predefinite per i tipi numerici, sono tutte utilizzabili indifferentemente in maiuscolo o minuscolo, tranne per il formato esponenziale.

Stringa di formato	Descrizione
C (Currency)	Il numero è convertito in una stringa che rappresenta una valuta nel formato locale.
D (Decimal)	Formato decimale, converte in base 10 e aggiunge degli zeri all'inizio se è specificata la precisione.
E (Exponential)	Formato scientifico o esponenziale. La precisione indica il numero di cifre decimali, 6 per default, mentre il simbolo esponenziale può essere maiuscolo o minuscolo.
F (Fixed-point)	Formato fixed-point, con la precisione che indica il numero di cifre decimali.
G (General)	Formato generale, il numero viene convertito in formato F oppure E, a seconda di quale dei due è il più compatto.
N (Number)	Il numero è convertito in una stringa con separatori delle migliaia e separatore decimale, ad es. 32,768.00.
P (Percent)	Formato percentuale.
X (Hexadecimal)	Il valore numerico è convertito in esadecimale, aggiungendo degli zeri se viene specificata la precisione.

Vediamo qualche esempio di formattazione:

```
double d=123456.789;
Console.WriteLine("{0:C2}",d);//€ 123.456,79
Console.WriteLine("{0:F1}",d);//123456,8
Console.WriteLine("{0:E}",d);//1,234568E+005

Console.WriteLine("{0:X}",7123);//1BD3

int i=10000;
int j=234;
Console.WriteLine("{0,10:D}+\n{1,10:D}=",i,j);
Console.WriteLine("{0,10:D}",i+j);
//     10000+
//       234=
//     10234
```

6.3.4 Altre operazioni con le stringhe

La classe String espone altri metodi ancora, che passiamo in rassegna in maniera veloce, ma con degli esempi molto esplicativi.
Il metodo `Substring` restituisce una sottostringa della stringa originale, a partire da una certa posizione e per una certo numero di caratteri, oppure fino alla fine della stringa se non viene specificata tale numero.

```
str=”hello world”;
Console.WriteLine(str.Substring(6));//stampa world
Console.WriteLine(str.Substring(6,2));//stampa wo
```

Se vogliamo invece suddividere una stringa in diverse sottostringhe, delimitate da uno o più caratteri, ed inserire tali sottostringhe in un array di string, possiamo utilizzare il metodo `Split`:

```
str="parole,separate,da;virgola;o;punto;e;virgola";
char[] sep=new char[]{',',';'};
string[] parole=str.Split(sep);

foreach(string parola in parole)
{
Console.WriteLine(parola);
}
```

L’esempio precedente suddivide la stringa str in un’array di stringhe, utilizzando come separatore i caratteri specificati nell’array di char sep.
Viceversa, è possibile concatenare due o più stringhe in una sola, utilizzando il metodo statico Concat, oppure il metodo Join se vogliamo anche inserire un elemento separatore fra le stringhe da unire.
Ad esempio dato l’array di stringhe ottenuto nel caso precedente, possiamo concatenarle in una sola stringa, senza spazi, in questa maniera:

```
string concat=String.Concat(parole);
Console.WriteLine(concat);//stampa paroleseparatedavirgolaopuntoevirgola
```

mentre con l’utilizzo del metodo Join possiamo ottenere una stringa come unione delle stringhe separate da un separatore, ad esempio uno spazio:

```
string join=String.Join(" ",parole);
Console.WriteLine(join);
// parole separate da virgola o punto e virgola
```

I metodi `Remove`, `Replace`, `Insert`, consentono di lavorare con una stringa, restituendo una nuova stringa in cui, come dicono i nomi, si è rimossa, rimpiazzata, o inserita una sottostringa o un insieme di caratteri.
L’esempio seguente riepiloga l’utilizzo dei tre metodi:

```
str="hello world";
Console.WriteLine(str.Remove(4,7));//stampa hell
Console.WriteLine(str.Replace("world","universe"));//stampa hello universe
Console.WriteLine(str.Replace('e','a'));//stampa hallo world
Console.WriteLine(str.Insert(6,"beautiful "));//stampa hello beautiful world
```

Per mezzo dei metodi `PadLeft` e `PadRight` possiamo formattare una stringa, aggiungendo un certo numero di caratteri ben specificati alla sinistra o alla destra di essa, utili dunque quando si vuole effettuare un allineamento a destra o a sinistra. Se non si specificasse il carattere viene utilizzato lo spazio come carattere di padding.

```
Console.WriteLine(str.PadLeft(15,'_'));//stampa ____hello world
Console.WriteLine(str.PadRight(15,'_'));//stampa hello world____
```

I metodi Trim, TrimLeft e TrimRight invece servono a rimuovere tutte le occorrenze di un dato insieme di caratteri, che si trovano a sinistra e a destra della stringa, nel primo caso, oppure solo dalla sinistra e solo dalla destra, rispettivamente con TrimLeft e TrimRight.

```
str="_-_-_hello world_-_-_";
char[] chars=new char[]{'_','-'};
Console.WriteLine(str.TrimStart(chars));//stampa  hello world_-_-_
Console.WriteLine(str.TrimEnd(chars));//stampa _-_-_hello world
Console.WriteLine(str.Trim(chars));//stampa hello world
```

Citiamo infine i metodi `ToUpper`, che data una stringa ne restituisce una con tutti i caratteri di quella originale in maiuscolo, o viceversa in minuscolo, con il metodo `ToLower`:

```
str="Hello World";
Console.WriteLine(str.ToUpper());//stampa HELLO WORLD
Console.WriteLine(str.ToLower());//stampa hello world
```

6.4 La classe StringBuilder

Le stringhe in C# sono immutabili. Quando si ha bisogno di una stringa mutevole di caratteri, viene in nostro soccorso la classe `StringBuilder` contenuta nel namespace System.Text.
La classe StringBuilder permette di effettuare dinamicamente operazioni di modifica sulle stringhe, nel senso che è possibile aggiungere, rimuovere, sostituire, inserire caratteri, anche dopo la creazione della sequenza di caratteri che rappresenta la stringa.
Internamente un oggetto StringBuilder mantiene un array di caratteri che rappresenta una stringa, ed è su questo array che i suoi metodi effettuano le manipolazioni necessarie. Ad esempio se è necessario concatenare una stringa ad una preceentemente costruita, il metodo Append semplicemente aggiunge i nuovi caratteri all'array, eventualmente aumentando la sua dimensione.

6.4.1 Costruire uno StringBuilder

Utilizzando il costruttore di default, viene riservato uno spazio di 16 caratteri. Il valore di tale capacità e quello della lunghezza effettiva della stringa contenuta nell'array, sono ottenibili mediante le proprietà `Capacity` e `Length`:

```
StringBuilder sb=new StringBuilder();
Console.WriteLine(sb.Capacity);//stampa il valore 16
Console.WriteLine(sb.Length);//stampa il valore 0
```

Oltre a queste due proprietà, ne è presente anche una terza di sola lettura, che indica il limite massimo fino al quale l'array di caratteri può crescere, ed è la proprietà `MaxCapacity`, che per default vale Int32.MaxValue (cioè 2^32 -1, più di 2 miliardi, una lunghezza più che sufficiente per stringhe di uso comune!).
Oltre al costruttore di default, la classe StringBuilder ne fornisce altri, che ad esempio permettono di impostare la capacità iniziale, oppure una stringa con cui inizializzare l'array di caratteri.

```
sb=new StringBuilder(100); //imposta la capacità iniziale a 100 caratteri
sb=new StringBuilder("hello"); //inizializza con i caratteri di 'hello'
sb=new StringBuilder(100,500);//capacità iniziale 100, che può crescere fino ad un massimo di 500
sb=new StringBuilder("hello",10); //inizializza con i caratteri di 'hello' e capacità iniziale 10
```

6.4.2 I metodi di StringBuilder

I caratteri che un oggetto StringBuilder conserva al suo interno, sono visualizzabili e modificabili singolarmente per mezzo della propietà `Chars`, che è anche l'indicizzatore della classe.

```
StringBuilder sb=new StringBuilder("hello");
Console.WriteLine("sb contiene i caratteri");
for(int i=0;i<sb.Length;i++)
{
```

```
      Console.WriteLine(sb[i]);
}
sb[0]='b';
Console.WriteLine(sb.ToString());
```

Per ottenere l'oggetto string costruito e gestito dall'oggetto StringBuilder viene utilizzato naturalmente il metodo ToString.
Il metodo ToString restituisce un riferimento alla stringa, e dopo la sua chiamata, se si modifica ancora la stringa mediante lo StringBuilder, viene in realtà creato un altro array di caratteri, e lo stesso accade se la modifica fa superare la capacità impostata.
La tabella seguente riporta altri metodi della classe StringBuilder per manipolare le stringhe.

Metodo	Descrizione
Append	Appende un oggetto convertito in stringa in coda all'array di caratteri, aumentando la sua capacità se necessario
Insert	Inserisce un oggetto convertito in stringa in una data posizione dell'array di caratteri, aumentando la sua capacità se necessario.
Replace	Sostituisce un carattere con un altro, oppure una stringa con un'altra all'interno del vettore di caratteri.
Remove	Rimuove un intervallo di caratteri dal vettore di caratteri.
AppendFormat	Appende gli oggetti specificate utilizzando una stringa di formato. E' uno dei metodi più utili della classe StringBuilder.

Tabella 5 I metodi di StringBuilder

Ecco qualche esempio per mostrare l'uso di ognuno di essi:

```
StringBuilder sb=new StringBuilder("hello");
sb.Append(" world");
sb.Insert(6,"beautiful ");
sb.Replace("world","universe");
sb.Remove(5,sb.Length-5);
sb.AppendFormat(" {0} {1}","da","Antonio");
Console.WriteLine(sb.ToString());
```

I metodi della classe StringBuilder restituiscono un riferimento alla stesso oggetto StringBuilder su cui sono stati chiamati. Questo permette di effettuare delle modifiche in sequenza, concatenando diverse chiamate:

```
StringBuilder sb2=new StringBuilder();
sb2.AppendFormat("\n\t{0}+{1}","Io","Programmo").Replace('+',' ').Append(" C#       ");
Console.WriteLine(sb2.ToString().TrimEnd());
```

6.5 Collezioni di oggetti

Le collezioni sono strutture dati fondamentali quando si tratta di memorizzare insiemi di oggetti, più o meno grandi, in memoria. Qualche capitolo fa abbiamo già visto come creare array di oggetti, ma gli array ordinari hanno delle limitazioni, ad esempio le loro dimensioni sono fissate in fase di istanziazione e non è più possibile variarle, inoltre l'unico modo che abbiamo utilizzato per leggere e scrivere i loro elementi, è attraverso l'indicizzazione.
Nel caso in cui si necessiti di maggiore flessibilità, ad esempio per la ricerca di un elemento, per l'ordinamento, per la rimozione di un elemento, o altro, il namespace **System.Collections** contiene classi adatte ad ogni scopo, permettendo ad esempio di creare e manipolare liste ordinate, code, pile, tabelle hash, dizionari.
Tali classi sono basate su interfacce che standardizzano i metodi per trattare tali insiemi di oggetti, ad esempio ogni collezione di oggetti implementa l'interfaccia `ICollection`, che espone la proprietà `Count` per ottenere il numero di elementi di una collezione ed il metodo `CopyTo` per copiare i suoi elementi in un array.

Rimarchiamo inoltre il fatto che l'interfaccia ICollection deriva a sua volta da `IEnumerable`, che espone il metodo `GetEnumerator`, ed è questo che permetterà ad esempio di iterare, in modo sequenziale, gli elementi di una collezione per mezzo della già vista istruzione foreach.
In questo paragrafo illustreremo le principali classi del namespace System.Collections ed i loro metodi e proprietà.

6.5.1 La classe System.Array

La classe **System.Array** fornisce metodi per utilizzare gli array, ad esempio crearli, manipolarli, ordinarli, ricercare un elemento al suo interno.
La classe Array è inoltre la classe base per tutti gli array, anche quelli costruiti utilizzando la sintassi base del linguaggio, vale a dire per mezzo delle parentesi quadre [ ]. Ciò significa che un array del tipo

```
int[] numeri=new int[10];
```

è semplicemente un'istanza la cui classe è derivata dalla classe System.Array, infatti se provate a visualizzare il nome della classe base della variabile numeri ne avrete la conferma immediata:

```
int[] numeri=new int[10];
Console.WriteLine(numeri.GetType().BaseType);//stampa System.Array
```

Possiamo ricavare delle informazioni su un array per mezzo delle proprietà e dei metodi, statici e d'istanza, messi a disposizione dalla classe System.Array.
Ad esempio la proprietà `Rank` restituisce il numero di dimensioni dell'Array, che naturalmente è maggiore di 1 nel caso di array multidimensionali.

```
int[] mono=new int[3];
int[,] bidim=new int[5,3];
int[,,] tridim=new int[3,4,5];
Console.WriteLine("mono ha dimensione {0}",mono.Rank);// 1 dimensione
Console.WriteLine("bidim ha dimensione {0}",bidim.Rank);// 2 dimensioni
Console.WriteLine("tridim ha dimensione {0}",tridim.Rank); // 3 dimensioni
```

Le tre istruzioni Console.WriteLine stamperanno in questo esempio le dimensioni 1, 2 e 3 rispettivamente.
La proprietà `Length` della classe System.Array consente di ottenere il numero totale di elementi in tutte le dimensioni di un array:

```
int[,] numeri=new int[5,3];
```

è una matrice 5x3, cioè ha in totale 15 elementi, infatti se eseguiamo l'istruzione:

```
int elementi=numeri.Length;
```

otteniamo il valore 15. Se invece vogliamo conoscere una per una le dimensioni di ogni riga, possiamo utilizzare il metodo `GetLength(int dim)` che prende come argomento l'indice della dimensione di cui vogliamo ricavare la lunghezza:

```
Console.WriteLine("La prima riga della matrice numeri ha {0} elementi",numeri.GetLength(0));
```

Possiamo quindi utilizzare il metodo GetLength per inizializzare gli elementi di un array con un ciclo for:

```
for(int i=0;i<numeri.GetLength(0);i++)
 for(int j=0;j<numeri.GetLength(1);j++)
  numeri[i,j]=i*j;
```

I metodi `GetLowerBound` e `GetUpperBound` restituiscono i limiti inferiore e superiore di una dimensione dell'array, ad esempio per un array bidimensionale di 3 per 5 elementi, le istruzioni seguenti

```
string[,] s=new string[3,5];

for(int i=0;i<numeri.Rank;i++)
{
Console.WriteLine("rank{0}:indice inf={1} e sup={2}", i, numeri.GetLowerBound(i),
numeri.GetUpperBound(i));
}
```

stamperanno per la prima dimensione i limiti [0,2] e per la seconda [0,4].
Oltre che con la sintassi già vista, si può creare un'istanza di un array, utilizzando il metodo statico `CreateInstance`. Tale metodo permette, con vari overload, di specificare ad esempio il tipo degli elementi che dovrà contenere l'array, la lunghezza, il numero di dimensioni e i limiti inferiori.

```
int[] lunghezze=new int[2]{3,4};
int[] limitiInf=new int[2]{1,2};
Array arr=Array.CreateInstance(typeof(string),lunghezze,limitiInf);
for(int i=0; i<arr.Rank;i++)
      Console.WriteLine("{0}\t{1}\t{2}",i,arr.GetLowerBound(i),arr.GetUpperBound(i));
//stampa l'output
//limiti  inf     sup
//0       1       3
//1       2       5
```

Fino ad ora abbiamo sempre utilizzato l'operatore di indicizzazione [ ] per accedere agli elementi di un array, in lettura o in scrittura. Creando però l'array come nell'esempio precedente, con il metodo CreateInstance, non sarà possibile utilizzare tale operatore, la classe Array fornisce dunque i metodi per svolgere le stesse operazioni, cioè i metodi `GetValue` e `SetValue`.

```
Array myArray=Array.CreateInstance( typeof(String), 2, 4 );
myArray.SetValue( "Un", 0, 0 );
myArray.SetValue( "array", 0, 1 );
myArray.SetValue( "di", 0, 2 );
myArray.SetValue( "stringhe", 0, 3 );
myArray.SetValue( "disposte", 1, 0 );
myArray.SetValue( "su", 1, 1 );
myArray.SetValue( "due", 1, 2 );
myArray.SetValue( "righe", 1, 3 );

// Stampa gli elementi dell'array.
for ( int i = myArray.GetLowerBound(0); i <= myArray.GetUpperBound(0); i++ )
   for ( int j = myArray.GetLowerBound(1); j <= myArray.GetUpperBound(1); j++ )
       Console.WriteLine( "\t[{0},{1}]:\t{2}", i, j, myArray.GetValue( i, j ) );
```

L'output sarà in questo caso:

```
L'array contiene:
[0,0]:  Un
[0,1]:  array
[0,2]:  di
[0,3]:  stringhe
[1,0]:  disposte
[1,1]:  su
[1,2]:  due
[1,3]:  righe
```

La ricerca di un elemento all'interno di un array è possibile per mezzo del metodo statico `BinarySearch` , che come fa intuire il nome implementa la ricerca binaria in un array: Uno degli overload del metodo ha la firma seguente:

```
int Array.BinarySearch(Array arr, Object obj)
```

Con tale metodo si ricerca all'interno dell'array arr l'elemento obj e viene restituito l'indice dell'elemento se esso esiste, altrimenti si otterrà un numero negativo, ad esempio:

```
int[] altriNumeri={ 1,2,3,4,5,6,7,8 };
int nIndex=Array.BinarySearch(altriNumeri,4);
Console.WriteLine("4 si trova all'indice {0}",nIndex);
nIndex=Array.BinarySearch(altriNumeri,10);
if(nIndex<0)
      Console.WriteLine("Il numero 10 non c'è");
```

Il metodo statico `Array.Clear` serve a cancellare gli elementi, tutti o un intervallo, di un array. Cancellare significa impostare a 0 dei numeri, a false dei boolean, o a null, altri tipi di oggetti.
Uno dei possibili overload è il seguente:

```
public static void Clear(Array array, int index, int length);
```

in cui il parametro array indica l'array da cancellare, index è l'indice a partire dal quale cancellare, e length il numero di elementi da cancellare, ad esempio se abbiamo un array di 5 numeri interi:

```
int[] numeri={ 1,2,3,4,5};
Array.Clear(numeri,2,2);
```

l'istruzione Clear in questo caso cancellerà gli elementi 3,4, cioè 2 elementi a partire dall'indice 2.
Il metodo statico `Copy`, ricopia gli elementi di un array sorgente in un array destinazione, con la possibilità di specificare il numero di elementi da copiare.

```
int[] altriNumeri={ 1,2,3,4,5};
int[] arrCopia=new int[altriNumeri.Length];

Array.Copy(altriNumeri,arrCopia,altriNumeri.Length);
for(int i=0;i<arrCopia.Length;i++)
      Console.Write("{0,3}",arrCopia[i]);
```

Questo esempio produrrà in output gli elementi dell'array arrCopia, che naturalmente saranno gli stessi elementi dell'array sorgente. Esiste anche il metodo di istanza `CopyTo(Array,int)` per effettuare la copia degli elementi di un array, ed inserirli in un array destinazione a partire da un dato indice:

```
Array myArray=Array.CreateInstance( typeof(String), 4 );
myArray.SetValue( "Un", 0 );
myArray.SetValue( "array", 1 );
myArray.SetValue( "di",  2 );
myArray.SetValue( "stringhe",  3 );

Array destArray=Array.CreateInstance(typeof(string),8);
myArray.CopyTo(destArray,myArray.Length);
Console.WriteLine("L'array destinazione contiene:");
for(int i=0;i<destArray.Length;i++)
{
      Console.WriteLine("destArray[{0}]={1}",i,destArray.GetValue(i));
}
```

Questo esempio copia gli elementi di myArray in destArray, inserendoli a partire dall'elemento con indice 4, mentre i precedenti elementi resteranno vuoti.
Per ordinare gli elementi di un array in modo semplice e veloce, possiamo utilizzare il metodo statico `Sort`, che prende come argomento l'array da ordinare.

```
string[] str={"a","c","f","e","d","b"};
Console.WriteLine("\nArray originale");
for(int i=0;i<str.Length;i++)
      Console.Write("{0,3}",str[i]);

Array.Sort(str);

Console.WriteLine("\nDopo l'ordinamento");
```

```
for(int i=0;i<str.Length;i++)
      Console.Write("{0,3}",str[i]);
```

Dopo la chiamata a Sort gli elementi verranno stampati in ordine alfabetico.
Se fosse necessario invertire l'ordine degli elementi di un array, è possibile utilizzare il metodo Reverse, passando come argomento l'array da invertire. Se ad esempio dopo aver ordinato l'array precedente, effettuiamo ora la chiamata:

```
Array.Reverse(str);
```

e stampiamo gli elementi, vedremo che l'ordine è adesso decrescente, cioè dalla f alla a. Infine vediamo come trovare la prima o l'ultima posizione in cui appare un elemento all'interno di un array. I metodi da utilizzare sono IndexOf e LastIndexOf:

```
byte[] myArray={0,1,0,0,1,0,1,0};
Console.WriteLine("\n\nmyArray contiene:");
for(int i=0;i<myArray.Length;i++)
      Console.Write("{0,3}",myArray[i]);

int index=Array.IndexOf(myArray,(byte)1);
Console.WriteLine("\nIl primo 1 si trova all'indice {0}",index);

index=Array.LastIndexOf(myArray,(byte)1);
Console.WriteLine("\nL'ultimo 1 si trova all'indice {0}",index);
```

6.5.2 La classe ArrayList

La classe **ArrayList** è molto simile ad un Array, la differenza fondamentale è che un ArrayList ha dimensioni che possono crescere nel caso in cui si sia raggiunta la massima capacità e si renda necessario aggiungere altri elementi. All'atto della creazione di un ArrayList viene specificata la capacità iniziale del vettore, o se non si specifica alcun numero, questa viene fissata al valore predefinito 16.

```
ArrayList al=new ArrayList(10);//capacità iniziale di 10 elementi
ArrayList al2=new ArrayList();//capacità iniziale predefinita di 16 elementi
```

E' essenziale distinguere la capacità di un ArrayList dal numero di elementi effettivamente presenti in esso. La prima può essere ricavata mediante la proprietà `Capacity`, mentre il secondo è dato dalla proprietà `Count`:

```
ArrayList al=new ArrayList(20);
int cap=al.Capacity; //restituisce 20;
int n=al.Count; //restituisce 0, nessun elemento è stato inserito
```

In genere la capacità sarà maggiore del numero di elementi effettivamente contenuti nell'ArrayList.
Per minimizzare lo spreco di memoria dovuto a posizioni dell'ArrayList non sfruttate, po' essere utilizzato il metodo `TrimToSize`, che imposta la capacità al numero di elementi, eliminando appunto le caselle vuote:

```
al.TrimToSize();
```

E' anche possibile inizializzare un ArrayList con un altro oggetto che implementi l'interfaccia ICollection, ad esempio se abbiamo un array ordinario di interi, possiamo creare un ArrayList che contenga tutti i suoi elementi:

```
int[] arr=new int[50];
for(int i=0;i<50;i++)
      arr[i]=i;
ArrayList vettore=new ArrayList(arr);
Console.WriteLine("L'arraylist vettore contiene");
```

```
foreach(int i in vettore)
      Console.Write(" "+vettore[i]);
```

L'ArrayList tratta gli elementi come riferimenti, quindi è possibile inserirvi una qualsiasi tipologia di oggetti e, anzi, nel caso in cui si voglia memorizzare un elemento di tipo valore, esso subirà il boxing automatico, prima di essere inserito nell'ArrayList. Leggendo un elemento bisogna invece effettuare l'unboxing in modo esplicito, naturalmente verso il tipo degli elementi contenuti nell'ArrayList:

```
int i=vettore[0]; //errore, non si può convertire implicitamente object in int
string s=(string)vettore[0]; //eccezione InvalidCastException
int i=(int)vettore[0]; //OK
```

Come visto è possibile accedere ai singoli elementi mediante l'operatore di indicizzazione, e per leggere un elemento il suo utilizzo è necessario. Ma la classe ArrayList fornisce anche dei metodi per aggiungere, rimuovere, inserire in una specifica posizione, uno o più oggetti alla volta.
Per aggiungere un elemento alla fine di un ArrayList è possibile utilizzare il metodo `Add`, che restituisce inoltre l'indice al quale l'oggetto è stato inserito.

```
int indice=vettore.Add("hello");
```

Il metodo `Insert` permette di inserire un elemento ad una specifica posizione:

```
Auto auto=new Auto();
vettore.Insert(1, auto);
```

Con i metodi `RemoveAt` e `Remove` si rimuovono elementi dall'ArrayList. Il primo rimuove l'oggetto alla posizione specificata, mentre con il secondo viene specificato direttamente l'oggetto da rimuovere, quindi verrà fatta una ricerca lineare all'interno dell'ArrayList fino a trovare l'oggetto.

```
vettore.RemoveAt(0);//rimuove il primo elemento
vettore.Remove(auto);//ricerca e rimuove l'oggetto auto
```

Il metodo Remove non restituisce alcun valore, nè lancia un'eccezione se l'oggetto non è stato trovato nell'ArrayList, può dunque essere utile verificare se un ArrayList contiene un dato oggetto. Il metodo `Contains` effettua questa ricerca e restituisce un booleano ad indicare se l'oggetto è presente o meno.

```
if(vettore.Contains(auto))
      vettore.Remove(auto);
```

Per ripulire un ArrayList da tutti gli elementi che esso contiene è sufficiente una chiamata al metodo Clear.

```
vettore.Clear(); //rimuove tutti gli elementi
```

La classe ArrayList permette di lavorare con più elementi contemporanemamente. Infatti, dato un insieme di oggetti contenuti in una collezione, cioè in un oggetto che implementa l'interfaccia Icollection, possiamo aggiungerli in un colpo solo, con il metodo `AddRange`, oppure inserirli in una data posizione con il metodo `InsertRange`, o ancora rimuovere un certo numero di elementi con il metodo `RemoveRange`:

```
vettore.AddRange(new string[]{"a","b","c"});//aggiunge un array di stringhe
vettore.InsertRange(vettore.Count-1, new int[]{1,2,3}); //aggiunge un array di interi in coda
vettore.RemoveRange(2,3);//rimuove 3 elementi a partire dall'indice 2
```

Il metodo `SetRange` permette invece di ricopiare su un certo intervallo dell'ArrayList un corrispondente numero di nuovi oggetti.

```
Vettore.SetRange(0,new int[]{3,4,5}); //sovrascrive i primi tre elementi con gli interi 3,4,5
```

6.5.3 Le tabelle Hash

Le tabelle hash sono strutture dati che conservano delle coppie chiave-valore, organizzate in base al codice hash della chiave. Il concetto è analogo a quello di un dizionario o ad una rubrica, in cui le voci sono organizzati in ordine alfabetico, ed a ognuna di esse corrisponde ad esempio una descrizione o un numero di telefono.
Per la loro struttura, le Hashtable sono utilizzate quando è necessario memorizzare e ricercare dei dati in maniera veloce ed efficiente, infatti quando si deve memorizzare una coppia chiave-valore, viene ricercata la locazione di memorizzazione valutando il codice hash della chiave. Naturalmente più oggetti possono avere uno stesso codice hash, dunque saranno memorizzati in una stessa locazione.
In fase di ricerca viene ancora valutato il codice hash, e ciò restringe il campo di ricerca dell'oggetto agli elementi memorizzati in una particolare locazione, dei quali viene letto e confrontato il valore della chiave con il valore ricercato.
La classe **Hashtable** della Base Class Library, rappresenta una simile struttura dati. La costruzione e popolazione di una Hashtable avviene in maniera molto semplice, basta istanziare la classe ed utilizzare il metodo `Add`:

```
Hashtable rubrica=new Hashtable();
rubrica.Add("Antonio","05 12345");
rubrica.Add("Caterina","09 41234");
rubrica.Add("Daniele","32 8765");
rubrica.Add("Rosita","09 09876");
```

In questa maniera ricercare un numero di telefono può essere fatto semplicemente con un'indicizzazione della Hashtable tramite il nome da cercare:

```
string numero=(string)rubrica["Caterina"];
```

Gli elementi possono essere aggiunti ad una tabella hash anche direttamente tramite l'indicizzatore:

```
rubrica["Pippo"]="0912354";
```

La differenza fra aggiungere un elemento con il metodo Add e l'aggiungerlo tramite l'indicizzatore, è che nel primo caso se l'elemento fosse già presente nella tabella viene generata un'eccezione ArgumentException, mentre nel secondo caso l'elemento viene rimpiazzato con quello nuovo.
Rimuovere un'elemento è altrettanto immediato per mezzo del metodo `Remove`, mentre la svuotatura di tutta la tabella può essere fatta tramite il metodo `Clear`;

```
rubrica.Remove("Pippo");
rubrica.Clear();
```

Per ricavare il numero di coppie chiave-valore conservate in una hashtable è sufficiente utilizzare la proprietà `Count`.

```
int count=rubrica.Count;
Console.WriteLine("La rubrica contiene {0} nomi",count);
```

La proprietà `Keys` contiene la collezione delle chiavi presenti nella hashtable, mentre con la proprietà `Values` si possono ottenere i valori:

```
Console.WriteLine("La rubrica contiene questi nomi:");
foreach(string str in rubrica.Keys)
{
      Console.WriteLine(str);
```

```
}
```

Le coppie nome valore vengono memorizzate in un'oggetto Hashtable come istanze della struttura `DictionaryEntry`, che possiede i campi `Key` e `Value`. Dunque per iterare tutti gli elementi con un ciclo foreach bisogna specificare come tipo degli elementi appunto DictionaryEntry:

```
Console.WriteLine("Contenuto della rubrica");
foreach(DictionaryEntry entry in rubrica)
{
      Console.WriteLine("{0}\t\t{1}",entry.Key,entry.Value);
}
```

6.5.4 Code e pile

Uno degli esercizi più frequenti in ogni libro di programmazione è quello che prevede l'implementazione di strutture dati come la coda o la pila, cioè di strutture FIFO (First In First Out) e LIFO (Last In Last Out). La Base Class Library ci viene incontro fornendoci due classi pronte all'uso: la classe **Queue** e la classe **Stack**, ed i relativi metodi.
La classe Queue rappresenta una struttura di tipo array, in cui però gli elementi vengono inseriti ad una estremità e vengono rimossi dall'altra. E' utile dunque per rappresentare processi di memorizzazione di messaggi in arrivo e del susseguente processamento nello stesso ordine.
L'inserimento in coda si effettua con il metodo `Enqueue`:

```
Queue codaMessaggi=new Queue();
codaMessaggi.Enqueue("Messaggio1");
codaMessaggi.Enqueue("Messaggio2");
codaMessaggi.Enqueue("Messaggio3");

foreach(string str in codaMessaggi)
{
      Console.Write("{0}\t",str);
}
```

Per estrarre invece l'elemento in testa alla coda, si utilizza il metodo `Dequeue`:

```
string msg=(string)codaMessaggi.Dequeue();
Console.WriteLine(msg);
```

Il metodo `Peek` consente di esaminare l'elemento in testa alla coda, senza però estrarlo.

```
Console.WriteLine("Peek");
msg=(string)codaMessaggi.Peek();
Console.WriteLine(msg);
```

La classe `Stack` rappresenta la classica pila di elementi, posizionati uno sull'altro al loro arrvo, e dunque in modo che l'ultimo arrivato sarà il primo ad essere estratto.
Il metodo per inserire un elemento in testa alla pila è il metodo `Push`:

```
Stack pila=new Stack();
pila.Push("primo");
pila.Push("secondo");
pila.Push("terzo");
```

stampiamo stavolta gli elementi della collezione, utilizzando un enumeratore, invece del solito foreach:

```
IEnumerator e=pila.GetEnumerator();
while(e.MoveNext())
{
      Console.WriteLine("{0}",e.Current);
}
```

Anche la classe Stack possiede il metodo Peek per visualizzare l'elemento in testa alla pila, senza però estrarlo.

6.5.5 Sequenze di bit

La classe **BitArray** rappresenta una struttura che permette di gestire un vettore di bit, ognuno dei quali è rappresentato da un valore true o false. In genere avendo a che fare con simili strutture, è prevedibile che si renderà necessario anche l'utilizzo di metodi prettamente rivolti alla manipolazione dei bit, come le operazioni di and, or, xor, e not, e la classe ci consente di eseguire, fra le altre, tutte queste operazioni.
Per costruire una BitArray abbiamo a disposizione diversi costruttori, ad esempio

```
BitArray ba1=new BitArray(8);
BitArray ba2=new BitArray(new bool[]{true,false,true,false});
BitArray ba3=new BitArray(8,true);
```

Il primo costruttore crea un array di bit di lunghezza 8, con tutti i bit a false per default, nel secondo caso i bit vengono inizializzati esplicitamente per mezzo di un array di valori bool, mentre del terzo BitArray è ancora specificata la lunghezza ma anche il valore di inizializzazione di tutti i suoi bit.
I bit di un BitArray possono essere modificati accedendo ad ognuno di essi per mezzo dell'operatore di indicizzazione, ad esempio:

```
ba1[0]=false;
ba1[1]=true;
```

Per effettuare un operazione di algebra booleana fra due vettori di bit, la classe BitArray fornisce dei metodi ad hoc, i quali si utilizzano nella maniera più logica possibile, ma bisogna prestare attenzione al fatto che oltre a ritornare un oggetto BitArray, lo stesso oggetto BitArray su cui si invocano i metodi viene modificato impostando i suoi bit a quelli ottenuti come risultato dell'operazione.
Ad esempio:

```
//ba1 0011
//ba2 1100
BitArray bOR=ba1.Or(ba3);
```

I bit del BitArray ba1 saranno stati modificati dopo l'invocazione del metodo `Or`, con il risultato dell'operazione di OR stessa, se infatti proviamo a stampare tali bit, otterremo:

```
Console.WriteLine("after ba1 OR ba2");
foreach(bool b in ba1)
{
      Console.Write(b?"1":"0");
}
//L'output sarà
//after ba1 OR ba2
//1111
```

Analogamente i metodi `And`, `Xor`, `Not` effettuano le relative operazioni booleane, modificando l'oggetto invocante.
Un array statico di interi naturalmente sarebbe più efficiente, in termini di prestazioni, ma la classe BitArray permette di gestire vettori di bit anche di lunghezza variabile, basta modificare la proprietà `Length` assegnando la nuova lunghezza, i bit eventualmente aggiunti saranno posti uguali a false:

```
BitArray ba3=new BitArray(4,true);
foreach(bool b in ba3)
      Console.Write(b?"1":"0");//stampa 1111
Console.WriteLine("raddoppio la lunghezza di ba3\n");
ba3.Length*=2;
foreach(bool b in ba3)
      Console.Write(b?"1":"0");//stampa 11110000
```

capitolo 7

7 Concetti avanzati

Dopo aver coperto le basi di C# ed il suo supporto al paradigma object oriented, in questo capitolo verranno illustrate alcune delle caratteristiche più avanzate del linguaggio, ed impareremo a sfruttarle per scrivere applicazioni più importanti e complesse.

7.1 Gestione delle eccezioni

Non esiste un programma perfetto, nonostante tutta l'attenzione che si può mettere nell'evitare gli errori, possono sempre sfuggire delle situazioni di possibile malfunzionamento, ed altre in cui il programma non si comporta come vorremmo. Non è inoltre sempre sufficiente prevedere tali situazioni e restituire ad esempio un codice di errore, soprattutto in metodi con diversi rami di esecuzione, e diverse cose che possono andare male. Supponiamo ad esempio di voler aprire una connessione di rete , scaricare un file, aprirlo e leggere delle stringhe rappresentanti dei numeri, convertirli in interi, fare delle divisioni, spedire il risultato sulla rete, e mille cose di altro genere. Dovremmo prevedere il fatto che il file non esista, o che sia già in uso, o che non abbiamo il permesso di leggerlo, o che se anche riuscissimo a leggerlo esso possa essere corrotto, che la connessione di rete non sia disponibile, che la divisione non sia eseguibile perchè i numeri sono nulli.
Per fortuna C# fornisce degli strumenti per gestire situazioni più o meno eccezionali, ed appunto queste funzionalità ricadono in quella che è detta gestione delle eccezioni.
Un'eccezione in C# è un oggetto, derivato, in modo diretto o indiretto, dalla classe System.Exception. Quando si verifica una eccezione, viene creato un tale oggetto contenente, tra le altre, informazioni sull'errore che si è verificato.
Eccezioni più specifiche possono inoltre derivare da classi particolari, ad esempio eccezioni che riguardano errori di Input/Output sono in genere derivate dalla classe IOException, da cui a loro volta derivano fra le altre le classi FileNotFoundException e FileLoadException, che riguardano ancora più specificatamente eccezioni nella gestione di file.

7.1.1 Catturare le eccezioni

Una volta che un'eccezione si è verificata, cioè è stata lanciata in un qualche punto del programma, è necessario catturarla e svolgere azioni di recupero.
Per far ciò, le parti di codice in cui è prevedibile che una o più eccezioni si possano verificare, vengono racchiuse in blocchi `try…catch…finally`. Un simile blocco è scritto in questa maniera:

```
try
{
      //codice che può generare un'eccezione
}
catch(TipoEccezione ex)
{
      //gestione dell'eccezione
}
finally
{
      //libera le risorse o svolge altre azioni
}
```

Se all'interno del blocco try si verifica un'eccezione del tipo previsto dall'istruzione catch, il controllo del programma passa appunto al blocco catch sottostante, in cui possono essere intraprese le azioni atte a risolvere il problema.

Se nel blocco try non si verifica invece alcun errore, o eventualmente si è raggiunta la fine del blocco catch, il flusso del programma prosegue nel blocco finally, in cui in genere vengono liberate delle risorse usate nel try precedente, o in genere vengono eseguite delle funzioni previste sia in caso di errore che in caso di esecuzione normale.
I blocchi catch possono essere anche più di uno in cascata, in maniera da gestire eccezioni di tipo differente, ad esempio supponiamo che nel blocco try tentiamo di aprire un file, può accadere che il file non esista, oppure che il file sia vuoto e si tenti di leggere una riga di testo:

```
public void TestTryCatchFinally()
{
      StreamReader sr=null;
      try
      {
            sr=File.OpenText(@"C:\temp\filevuoto.txt");
            string str=sr.ReadLine().ToLower;
            Console.WriteLine(str);
      }
      catch(FileNotFoundException fnfEx)
      {
            Console.WriteLine(fnfEx.Message);
      }
      catch(NullReferenceException flEx)
      {
            Console.WriteLine(flEx.Message);
      }
      finally
      {
            if(sr!=null)
                  sr.Close();
      }
}
```

Se il file non viene trovato l'esecuzione salterà al primo blocco catch, mentre nel caso in cui il file sia vuoto, al tentativo di leggere una riga con sr.ReadLine(), verrà restituito il valore null, e dunque nel convertire tale stringa nulla in minuscolo verrà generata un eccezione NullReferenceException, gestita dal secondo catch.
Alla fine, comunque vada, si passa dal blocco finally che eventualmente chiude l'oggetto StreamReader.
Nel gestire più tipi di eccezioni, con più blocchi catch, è necessario prestare attenzione all'ordine con cui appaiono i tipi di eccezione. Infatti è obbligatorio gestire per primi le eccezioni più specifiche, cioè che fanno parte di una catena di derivazione da System.Exception più lunga. Se non fosse così infatti l'eccezione potrebbe essere catturata da un catch precedente. Il compilatore comunque impedisce una simile eventualità, ad esempio:

```
try
{
      ...
}
catch(Exception ex)
{
      Console.WriteLine(ex.Message);
}
catch(FileNotFoundException fnfEx)
{
      Console.WriteLine(fnfEx.Message);
}
```

Il messaggio che il compilatore sarà in questo caso del tipo:

```
Un precedente catch cattura già tutte le eccezioni di questo tipo o di un tipo
superiore(System.Exception)
```

Nella gestione delle eccezioni è possibile omettere sia il blocco catch che il blocco finally, ma non entrambi contemporaneamente.
In particolare il solo blocco finally viene utilizzato per garantire che vengano eseguite certe istruzioni prima di uscire da un metodo, ad esempio il blocco try potrebbe contenere più istruzioni return, e

quindi più punti di uscita. Quando si incontra il primo dei return il programma prima di terminare il metodo passa dal blocco finally:

```
public int TestTryFinally()
{
	int i=2;
	try
	{
		switch(i)
		{
			case 2:
				return 4;
			case 3:
				return 9;
			default:
				return i*i;
		}
	}
	finally
	{
		Console.WriteLine("Prima di uscire passa dal finally");
	}
}
```

Il meccanismo per cui un'eccezione viene generata, viene chiamato throwing. Quando si verifica l'evento eccezionale, viene eseguita un'istruzione del tipo:

```
throw new TipoEccezione();
```

L'istruzione **`throw`** ha la funzione di generare e lanciare un'eccezione del tipo specificato, e se tale throw viene eseguito da una parte di codice eseguita all'interno di un blocco try, e se il tipo di eccezione è stato previsto in un blocco catch, entra in funzione la cattura dell'eccezione:

```
try
{
	throw new Exception("Genero un'eccezione");
}
catch(Exception ex)
{
	Console.WriteLine(ex.Message);
}
```

in questo caso l'eccezione verrà catturata e verrà stampato il messaggio "Genero un'eccezione" contenuto nell'istanza ex della classe System.Exception.
E' possibile utilizzare l'istuzione throw anche al di fuori di un blocco try, cioè in una parte qualunque di codice, ad esempio in un metodo. In questa maniera, al verificarsi di un'eccezione, non trovando il blocco catch per gestirla, il CLR risalirà lo stack delle chiamate fino a trovarne uno, o al limite fino a quando verrà restituito un messaggio di eccezione non gestita. Ad esempio supponiamo di scrivere un metodo che effettua la divisione fra due numeri:

```
public int Dividi(int a,int b)
{
	if(b==0)
		throw new DivideByZeroException("Eccezione generata da Dividi(a,b)");
	return a/b;
}
```

nel metodo l'eccezione può essere generata, ma non è gestita, allora utilizzando questo metodo da qualche altra parte nel nostro codice, dovremo gestire l'eventuale eccezione:

```
public void ChiamaDividi()
{
	try
	{
		Dividi(5,0);
	}
	catch(DivideByZeroException ex)
	{
		Console.WriteLine(ex.Message);
```

```
	}
}
```

L'eccezione generata nel metodo Dividi e propagata e gestita dal meto chiamante, cioè il metodo ChiamaDividi.
Come detto prima il throw può essere eseguito anche all'interno di un blocco catch, in questo caso si parla di eccezione rilanciata, ed esattamente viene rilanciata al metodo che ha chiamato quello in cui se è verificata, e così via risalendo lungo lo stack delle chiamate.

```
public void ChiamaDividi()
{
	try
	{
		Dividi(5,0);
	}
	catch(DivideByZeroException ex)
	{
		Console.WriteLine("Rilancio l'eccezione
"+ex.GetType()+Environment.NewLine+ex.Message);
		throw ex;
	}
}
```

Dopo aver stampato un messaggio con informazioni sull'eccezione, la stessa eccezione verrà rilanciata.
E' possibile scrivere blocchi catch anche senza specificare il tipo di eccezione fra parentesi, in questo caso verranno catturate in tale blocco tutti tipi di eccezione generati, ma non si possono così avere dettagli sull'eccezione verificatasi.

```
try
{
	Dividi(5,0);
}
catch
{
	Console.WriteLine("Si è verificata un'eccezione");
}
```

In un catch di tale tipo è possibile ancora rilanciare l'eccezione catturata, utilizzanda l'istruzione throw senza specificare nient'altro:

```
try
{
	Dividi(5,0);
}
catch
{
	throw;
}
```

Oltre a poter propagare l'eccezione ad un altro metodo, come negli esempi precedenti, è possibile scrivere dei blocchi annidati di try, in questa maniera un'eccezione generata all'interno di un blocco try, nel caso in cui il corrispondente catch non la gestisca, sarà propagata al blocco try esterno, ed eventualmente gestita dai blocchi catch di questo e così via.

```
int[] arr=new int[]{4,2,0};
int dividendo=100;
for(int i=0;i<4;i++)
{
	try
	{
		try
		{
			Console.WriteLine("{0}/{1}={2}",dividendo,arr[i],Dividi(dividendo,arr[i]));
		}
		catch(DivideByZeroException de)
		{
			Console.WriteLine(de.ToString());
			Console.WriteLine(de.Message);
```

```
            }
      }
      catch(IndexOutOfRangeException ie)
      {
            Console.WriteLine(ie.Message);
      }
}
```

Nel codice precedente abbiamo due try innestati, in quello più interno viene eseguita una divisione fra interi utilizzando il precedentemente visto metodo Dividi(int,int), l'eventuale eccezione DivideByZeroException viene gestita dal corrispondente catch. Ma gli operandi della divisione, in particolare il divisore viene preso da un array di tre interi, ciclando però con un for il cui indice varia da 0 a 3. All'ultimo accesso all'array, l'indice supererà i limiti, generando dunque una eccezione IndexOutOfRangeException, la quale non essendo gestita nel blocco try interno, verrà propagata a quello più esterno, in cui invece esiste un catch apposito. Provando a compilare il codice ed eseguendolo, avremo dunque due divisioni eseguite correttamente, e due eccezioni:

```
100/4=25
100/2=50
Eccezione generata in Dividi(a,b)
Index was outside the bounds of the array.
```

7.1.2 La classe System.Exception

La classe **System.Exception** costituisce la classe base per ogni altro tipo di eccezione in C#. La classe è in se stessa molto semplice, essendo costituita da una serie di proprietà pubbliche, utilizzabili per ricavare vari tipi di informazioni sull'eccezione stessa. Tali proprietà sono riassunte nella tabella seguente:

Proprietà	Descrizione
`public string Message`	Restituisce una stringa che descrive l'eccezione.
`public string Source`	Restituisce o imposta il nome dell'applicazione o dell'oggetto che ha generato l'eccezione.
`public string StackTrace`	Restituisce la rappresentazione dello stack di chiamate al momento dell'eccezione.
`public string HelpLink`	Restituisce o imposta il link alla documentazione sull'eccezione generata.
`public Exception InnerException`	Restituisce l'istanza di System.Exception che ha causato l'eccezione corrente, cioè se un'eccezione A è stata lanciata da una precedente eccezione B, allora la proprietà InnerException di A restituirà l'istanza B.
`public MethodBase TargetSite`	Restituisce il metodo in cui è stata generata l'eccezione.

Tabella 6 - Proprietà della classe System.Exception

Il seguente metodo mostra un esempio sul come utilizzare queste proprietà:

```
public void PrintExceptionInfo(Exception ex)
{
      Console.WriteLine("Message: {0}",ex.Message);
      Console.WriteLine("Source: {0}",ex.Source);
      Console.WriteLine("StackTrace: {0}",ex.StackTrace);
      Console.WriteLine("HelpLink: {0}",ex.HelpLink);
      Console.WriteLine("InnerException: {0}",ex.InnerException);
      Console.WriteLine("Method Name: {0}",ex.TargetSite.Name);
}
```

7.1.3 Eccezioni personalizzate

Ogni sviluppatore può creare le proprie classi di eccezioni per rappresentare meglio una situazione che può generarsi nella propria applicazione e quindi per rispondere meglio alle esigenze di gestione delle stessa. Avendo già visto come derivare una classe da una esistente, siamo già in grado di implementare le nostre eccezioni, derivando una classe dalla classe System.Exception, o da una eccezione il più vicino possibile a quella che dobbiamo implementare, ad esempio se si tratta di un'eccezione che riguarda l'input/output potremmo derivarla dalla IOException.

```
class EccezionePersonalizzata: Exception
{
}
```

già questo potrebbe bastare per lanciare un'EccezionePersonalizzata, in quanto il compilatore ci fornirà un costruttore di default senza parametri.

```
try
{
      throw new EccezionePersonalizzata();
}
catch(EccezionePersonalizzata ep)
{
      Console.WriteLine(ep.Message);
}
```

Il codice precedente stamperà un output del tipo:

```
Exception of type EccezionePersonalizzata was thrown.
```

Naturalmente è possibile fornire maggiori dettagli alle nostre eccezioni, partendo dall'implementare diversi costruttori, come nella seguente classe MiaException:

```
class MiaException:InvalidOperationException
{
      public MiaException():base()
      {
      }
      public MiaException(string msg):base(msg)
      {
      }
      public MiaException(string msg,Exception inner):base(msg,inner)
      {
      }
}
```

Testando questa nuova eccezione:

```
try
{
      try
      {
            int b=0;
            int a=1/b;
      }
      catch(DivideByZeroException ex)
      {
            throw new MiaException("Operazione impossibile",ex);
      }
}
catch(MiaException mp)
{
      Console.WriteLine("Message {0}",mp.Message);
      Console.WriteLine("InnerException: {0}",mp.InnerException.Message);
}
```

otterremo ad esempio questo messaggio:

```
Message: Operazione impossibile
InnerException: Attempted to divide by zero.
```

Nulla vieta, sviluppando le nostre classi di eccezioni personalizzate, di aggiungere campi e metodi ad hoc, ad esempio prevedere un codice numerico d'errore ed un metodo per ricavarlo da un'istanza dell'eccezione stessa, o ancora meglio dei metodi per recuperare dalla situazione d'errore nel modo più appropriato, in quanto, avendo progettato noi stessi l'eccezione, nessuno può meglio sapere come trattare i casi in cui si verifica.

7.2 Delegati

I delegate sono un semplice ed allo stesso tempo potente meccanismo per il reindirizzamento delle chiamate ad un metodo verso un altro metodo. Per chi conoscesse già i linguaggi C/C++ è un pò come avere a che fare con i puntatori a funzione o a metodi di una classe. Invocando dunque un delegate, si invoca in maniera indiretta il metodo o i metodi che il delegate maschera al suo interno, anche senza sapere quale metodo il delegate contenga.
La differenza fondamentale con i puntatori a funzione è che il meccanismo dei delegate è type-safe, è object oriented, ed è sicuro, perché come vedremo meglio fra qualche riga, i delegate non sono altro che delle istanze di normalissime classi C#, con dei metodi implementati per invocare altri metodi.
In parole povere è possibile dunque usare un metodo come se fosse un oggetto qualunque, e dunque passarlo come parametro ad un altro metodo.
Inoltre i delegate, come accennato prima, permettono di mascherare e quindi invocare con una sola chiamata più metodi in sequenza.
Per rendere più semplice e comprensibile l'argomento, introduciamo un esempio pratico, e supponiamo di voler realizzare un'applicazione bancaria, con una classe Banca, all'interno della quale sono contenuti i conti correnti dei clienti.

7.2.1 Dichiarazione di un delegate

La dichiarazione di un delegate è perfettamente analoga alla dichiarazione di una variabile, tale dichiarazione specifica la firma del metodo, cioè i parametri di ingresso e il tipo di ritorno, che il delegate stesso maschera. Tale metodo potrà poi essere sia un metodo statico che un metodo di istanza. Nel primo caso il delegate incapsula unicamente il metodo che dovrà invocare per conto di qualcun altro, nel secondo caso esso incapsula l'istanza ed un metodo di essa che dovrà invocare.
In generale la dichiarazione di un delegate avviene con la seguente sintassi:

```
[modificatore] delegate tipo_ritorno NomeDelegate([parametri]);
```

Dunque la dichiarazione è analoga a quella di un metodo qualunque con l'aggiunta della parola chiave **`delegate`**, e senza naturalmente fornirne il corpo, ma in realtà la dichiarazione nasconde quella di una classe derivata dalla classe **System.Delegate**.
Un metodo ed un delegato saranno compatibili, se essi hanno lo stesso tipo di ritorno e gli stessi parametri, sia come numero, sia come tipo, che come ordine di apparizione. Come modificatore d'accesso è possibile utilizzarne uno fra private, protected, public, internal e new. In particolare quest'ultimo viene usato per nascondere esplicitamente un delegato ereditato.
Riprendendo l'esempio, la seguente dichiarazione di delegate può essere utilizzata per riferirsi a metodi che prendono in ingresso un parametro di tipo String e non restituiscono un valore di ritorno:

```
public delegate void SaldoDelegate(ContoCorrente c);
```

Lo scopo è quello di fornire un metodo per la visualizzazione del saldo di un numero di conto corrente, tale saldo potrà essere fornito con un messaggio a video, con una stampa su file, con l'invio di un e-mail, e in mille altri modi.
La classe ContoCorrente ad esempio potrebbe avere due metodi diversi che rispettano la stessa firma specificata dal delegate, e, per completezza, uno dei due lo implementeremo statico:

```
public void VisualizzaSaldo(ContoCorrente conto)
{
      Console.WriteLine("saldo del conto {0}: {1}",conto.numeroConto,conto.saldoAttuale);
}

public static void SalvaSaldoSuFile(ContoCorrente conto)
{
      StreamWriter sw=null;
      try
      {
            sw=File.CreateText(@"c:\temp\saldo_"+conto.numeroConto+".txt");
            sw.WriteLine("saldo del conto {0}: {1}", conto.numeroConto,conto.saldoAttuale);
      }
      catch(IOException ioe)
      {
            Console.WriteLine(ioe.Message);
      }
      finally
      {
            if(sw!=null)
                  sw.Close();
      }
}
```

Il metodo VisualizzaSaldo mostrà sulla console il saldo del conto, mentre il metodo SalvaSaldoSuFile lo scriverà su un file di testo.

7.2.2 Istanziazione e invocazione di un delegate

Un delegate è dietro le quinte una classe derivata dalla classe System.Delegate, dunque l'istanziazione avviene per mezzo della parola chiave new, come per qualsiasi altro oggetto, e passando come argomento un metodo che rispetti la firma definita dal delegato stesso.
Ad esempio, per creare un'istanza del precedente `SaldoDelegate`, delegato del metodo `VisualizzaSaldo`, è sufficiente la chiamata

```
SaldoDelegate sd1=new SaldoDelegate(conto.VisualizzaSaldo);
```

Nessuna differenza con il secondo metodo che è statico:

```
SaldoDelegate sd2=new SaldoDelegate(ContoCorrente.SalvaSaldoSuFile);
```

Aggiungiamo allora alla classe ContoCorrente un metodo generico ElaboraSaldo che invocherà il delegate passato come argomento:

```
public void ElaboraSaldo(SaldoDelegate sd,ContoCorrente conto,string data)
{
      saldoAttuale=conto.CalcolaSaldo(data);
      sd(conto);
}
```

Quindi passando al metodo ElaboraSaldo l'istanza di delegate sd1, verrà indirettamente invocato il metodo VisualizzaSaldo, mentre con sd2 il metodo statico SalvaSaldoSuFile.
E' possibile che un delegate incapsuli più metodi contemporaneamente, in tal caso si parla di delegate multicast, i quali derivano appunto da una classe apposita, la classe **System.MulticastDelegate**.
La lista di invocazione di un delegate multicast viene creata utilizzando gli overload degli operatori + e += forniti dalla classe MulticastDelegate. Ad esempio, una volta creato il delegate sd1, è possibile aggiungere un delegate alla lista d'invocazione ed ottenere un delegate sd3 con la semplice istruzione:

```
sd3 = sd1 + new SaldoDelegate(ContoCorrente.SalvaSaldoSuFile);
```

oppure ottenere un risultato analogo con

```
sd1 += new SaldoDelegate(ContoCorrente.SalvaSaldoSuFile);
```

Naturalmente è possibile anche l'operazione inversa di rimozione dalla lista di invocazione, per mezzo degli operatori – e -=.
Notate però che è possibile concatenare più metodi in un delegate, cioè creare un MulticastDelegate, solo se la firma del delegate ha come tipo di ritorno void, d'altronde in caso contrario non si potrebbero utilizzare i diversi valori di ritorno. Il compilatore quindi osserverà proprio se il tipo restituito è void, ed in tal caso deriverà il delegate dalla classe MultiCastDelegate.
Il modo di utilizzo di un delegate multicast è analogo a quanto visto per i delegati semplici, infatti invocando il metodo ElaboraSaldo con il delegate multicast sd1, verranno eseguiti in sequenza i due metodi contenuti nella lista del delegate sd1.
La lista di invocazione, è ottenibile per mezzo del metodo `GetInvocationList` che restituisce un array di oggetti Delegate, i cui elementi sono disposti nell'ordine in cui verranno invocati.

```
Console.WriteLine("InvocationList di sd1");
foreach(Delegate d in sd1.GetInvocationList())
{
      Console.WriteLine("Delegate {0}",d.Method);
}
```

Il codice precedente stamperà i nomi dei metodi contenuti nel MulticastDelegate sd1, ad esempio:

```
InvocationList di sd1:
Delegate Void VisualizzaSaldo(ContoCorrente)
Delegate Void SalvaSaldoSuFile(ContoCorrente)
```

7.2.3 Delegati contro interfacce

Forse molti di voi avranno notato una similarità fra i delegate e le interfacce, ed in effetti entrambi permettono la separazione dell'implementazione dalla specifica.
Un'interfaccia specifica quali sono i membri che una classe deve esporre, membri che poi saranno implementati all'interno della classe stessa in maniera invisibile all'esterno.
Un delegate fa la stessa cosa, specificando la firma ed il tipo di ritorno di un metodo, metodo che poi sarà implementato in maniera differente ma compatibile con la firma.
Abbiamo già visto come dichiarare le interfacce, come implementarle in una classe, e come utilizzarle all'interno del nostro codice, ad esempio per usufruire della potenza e della flessibilità del polimorfismo.
I delegate sono utili tanto quanto le interfacce, ma si prestano molto meglio in situazioni forse più particolari, ad esempio, alcuni casi in cui è ideale utilizzare un delegate sono quelli in cui sappiamo già che un singolo metodo verrà chiamato, o se una classe vuole fornire più metodi per una stessa firma, o se tali metodi verranno implementati come statici, o ancora quando abbiamo a che fare con interfacce utente rispondenti a pattern di programmazione ad eventi.

7.3 Eventi

I messaggi sono il meccanismo principale utilizzato dalle applicazioni Windows, ma non solo, per spedire e/o ricevere notifiche di un qualcosa che è avvenuto e che interessa l'applicazione stessa. Ad esempio quando l'utente interagisce con una finestra cliccando su un pulsante, l'applicazione di cui fa parte la finestra verrà informata di ciò tramite un apposito messaggio.
Il framework .NET nasconde la complessità di basso livello dei messaggi, tramite il concetto di evento.
L'idea fondamentale da comprendere è simile al classico modello produttore-consumatore.

Un oggetto può generare degli eventi, un altro oggetto viene informato di essi ed intraprende delle azioni. Il mittente dell'evento non sa quale altro oggetto lo riceverà , dunque è necessario un meccanismo che funga da intermediario, che prescinda dalle tipologie di oggetti destinatari dell'evento. Tale meccanismo è quello illustrato nel precedente paragrafo, cioè quello dei delegate.

7.3.1 Generare un evento

Per generare un evento dobbiamo prima definire una classe che contenga le informazioni correlate ad esso. Il framework .NET fornisce una classe base da cui derivare i nostri eventi personalizzati, la classe **System.EventArgs**.
Supponiamo di avere a che fare con un'applicazione automobilistica, e di voler implementare un meccanismo che effettui il monitoraggio dei giri del motore, avvisando l'utente con un allarme quando il valore è al di sopra di una certa soglia, cioè quando il motore va fuori giri, e viceversa che avverta l'utenete che il motore si è spento perché il regime giri è sceso troppo.
Per il primo caso creiamo allora una classe come la seguente, che contiene il valore raggiunto dai giri del motore e formatta un messaggio di testo da restituire al consumatore dell'evento fuori giri.

```
public class MotoreFuoriGiriEventArgs: EventArgs
{
      private int rpm;

      public int Rpm
      {
            set
            {
                  rpm=value;
            }
            get
            {
                  return rpm;
            }
      }

      public string Message
      {
            get
            {
                  string msg=String.Format("Il numero dei giri motore è {0}/min",rpm);
                  return msg;
            }
      }
}
```

Bisogna poi dichiarare un delegate che serva come gestore dell'evento. Se l'evento non ha dati aggiuntivi è possibile utilizzare il delegate standard **EventHandler**. Nel nostro caso ne creiamo uno personalizzato in modo da mostrare la nomenclatura standard utilizzata per i gestori di eventi, ed uno standard che non necessità di ulteriori dati per l'evento MotoreSpento:

```
public delegate void MotoreFuoriGiriEventHandler(object sender, MotoreFuoriGiriEventArgs e);
public event EventHandler MotoreSpentoEvent;
```

Il framework .NET utilizza la convenzione di fornire un primo parametro sender, in modo che il gestore possa ricavare anche l'oggetto generatore dell'evento. Il secondo parametro conterrà invece l'evento con gli eventuali dati aggiuntivi che lo caratterizzano.
A questo punto introduciamo la parola chiave **event**. Con essa viene aggiunto un nuovo campo alla classe che genera l'evento, di tipo uguale al delegate precedentemente definito:

```
public event MotoreFuoriGiriEventHandler FuoriGiriEvent;
public event EventHandler MotoreSpentoEvent;
```

I due campi identificano i due eventi che possono essere generati dalla classe, nello stesso modo in cui, come ulteriore esempio, la classe Button possiede un campo Click, che indica l'evento di click su di esso.
Il passo successivo è l'aggiunta, sempre alla classe generatrice dell'evento, di un metodo con un nome che deve essere del tipo OnNomeEvento.
E' proprio in tale metodo che viene generato l'evento, invocando il delegate identificato dalla relativa keyword event, e creando un oggetto EventArgs contenente eventuali dati aggiuntivi. Nell'esempio dunque sarà:

```
//genera l'evento fuori giri
protected virtual void OnFuoriGiri(MotoreFuoriGiriEventArgs ev)
{
      FuoriGiriEvent(this,new MotoreFuoriGiriEventArgs(rpm));
}

//genera l'evento motore spento
protected virtual void OnMotoreSpento()
{
      MotoreSpentoEvent(this,new EventArgs());
}
```

Come in questo caso, in genere, il metodo è dichiarato protected virtual, in tale modo esso potrà essere ridefinito in una classe derivata, ed eventualmente, in questa verrà anche richiamato anche il metodo della classe base.
La classe Motore riportata qui di seguito riprende tutti i concetti esposti finora:

```
public class Motore
{
      private int rpm;
      private const int maxRpm=7000;
      private const int minRpm=700;
      private bool acceso;

      public delegate void MotoreFuoriGiriEventHandler(object sender, MotoreFuoriGiriEventArgs
e);

      public event MotoreFuoriGiriEventHandler FuoriGiriEvent;
      public event EventHandler MotoreSpentoEvent;

      public Motore()
      {
            rpm=0;
            acceso=false;
      }

      public int Rpm
      {
            get
            {
                  return rpm;
            }
      }

      public void Accendi()
      {
            acceso=true;
            this.rpm=800;
      }

      protected virtual void OnFuoriGiri(MotoreFuoriGiriEventArgs ev)
      {
            FuoriGiriEvent(this,new MotoreFuoriGiriEventArgs(rpm));
      }

      protected virtual void OnMotoreSpento()
      {
            MotoreSpentoEvent(this,new EventArgs());
      }
```

```
	public void AumentaRpm()
	{
		if(acceso)
			this.rpm+=100;
		if(rpm>maxRpm)
			OnFuoriGiri(new MotoreFuoriGiriEventArgs(rpm));
	}

	public void DiminuisciRpm()
	{
		if(acceso)
			this.rpm-=100;
		if(rpm<minRpm)
		{
			this.acceso=false;
			OnMotoreSpento();
		}
	}
}
```

La classe Motore fornisce due metodi, AumentaRpm e DiminuisciRpm che nel caso in cui i giri del motore sforino le soglie definite, richiamano i due metodi OnFuoriGiri ed OnMotoreSpento, e quindi generano gli eventi relativi.

7.3.2 Consumare un evento

Per consumare un evento bisogna prevedere un metodo che venga richiamato al verificarsi dell'evento, e registrare tale metodo come gestore dell'evento.
Scriviamo allora una classe Auto che contenga un campo di tipo Motore, il quale può eventualmente andare fuori giri o spegnersi, e quindi generare gli eventi relativi e che abbiamo definito nel paragrafo precedente.
I metodi che si occupano della gestione degli eventi devono avere la stessa firma dei delegate che definiscono gli evento stessi, cioè dei delegate MotoreFuoriGiriEventHandler e del delegate standard EventHandler. Dunque dei buoni gestori dell'evento FuoriGiriEvent e dell'evento MotoreSpentoEvent, potrebbero essere i due seguenti:

```
private void motore_FuoriGiriEvent(object sender, MotoreFuoriGiriEventArgs e)
{
	Console.WriteLine("Evento da {0}:\n{1}",sender.ToString(),e.Message);
	Frena();
}

private void motore_MotoreSpento(object sender, EventArgs e)
{
	Console.WriteLine("Evento da {0}:\nMotore Spento",sender.ToString());
}
```

Essi stampano semplicemente il tipo di evento generato, il messaggio contenuto nell'istanza MotoreFuoriGiriEventArgs nel primo caso, ed il tipo dell'oggetto che ha generato l'evento, che in questo esempio sarà sempre un'istanza della classe Motore.
Ora non ci resta che registrare i due gestori associandoli agli eventi relativi.
Per far ciò, C# permette di utilizzare gli operatori + e += per aggiungere un gestore all'evento, o anche al contrario – e -= per l'operazione di deregistrazione di un gestore, e ciò è giustificato dal fatto che abbiamo a che fare con delegate e vale quanto detto quando abbiamo parlato dei delegate multicast. Se la classe Auto contiene dunque un campo motore di classe Motore, sarà sufficiente, ad esempio nel costruttore, scrivere due istruzioni come le seguenti:

```
motore.FuoriGiriEvent+=new Motore.MotoreFuoriGiriEventHandler(motore_FuoriGiriEvent);
motore.MotoreSpentoEvent+=new EventHandler(motore_MotoreSpento);
```

Notate come nel primo caso, il delegate venga creato specificando anche il nome della classe Motore, visto che infatti esso era stato definito innestato nella classe Motore. Ciò non costituisce un obbligo,

il delegate poteva anche essere dichiarato al di fuori, ed in genere in qualunque parte di codice in cui è definibile una classe, l'unico vincolo è quello di assicurarne la visibilità.
Per completare la nostra classe Auto aggiungiamo due metodi che aumentano o diminuiscano i giri del motore:

```
public void Accelera()
{
      motore.AumentaRpm();
}

public void Frena()
{
      motore.DiminuisciRpm();
}
```

Per verificare come la generazione ed il consumo di eventi funzioni non ci resta che creare una sequenza di chiamate che portino il motore nelle condizioni di fuori giri o di spegnimento:

```
Auto auto=new Auto();
Console.WriteLine("rpm: {0}", auto.RpmMotore);
while(auto.RpmMotore<7000)
{
      auto.Accelera();
}
Console.WriteLine("rpm: {0}", auto.RpmMotore);
auto.Accelera();
```

In questo punto il motore supererà la soglia massima generando l'evento FuoriGiri, di cui la classe auto riceverà la notifica mediante la chiamata al gestore motore_FuoriGiriEvent.

```
while(auto.RpmMotore>700)
{
      auto.Frena();
}
Console.WriteLine("rpm: {0}", auto.RpmMotore);
auto.Frena();
```

Mentre qui il motore sarà al di sotto della soglia minima, e dunque si spegnerà generando il secondo evento ed informando ancora la classe Auto in cui verrà stavolta richiamato il gestore motore_MotoreSpento.
Il meccanismo di definizione e gestione degli eventi è fondamentale nella creazione delle applicazioni grafiche, quindi è fondamentale comprenderlo e assimilarlo per bene.

Capitolo 8

8 Cenni di Windows Forms

In questo capitolo daremo dei brevi cenni sul come avviene la creazione di un'applicazione Windows (o in generale a finestre) in C#, mediante l'uso delle classi messe a disposizioni dalla Base Class Library, ed in particolare nel namespace System.Windows.Forms. L'argomento richiederebbe più di un libro dedicato ad esso, e dunque non verrà approfondito più di tanto.
Esistono diversi ambienti integrati per lo sviluppo visuale delle applicazioni grafiche, da Visual Studio.NET a Borland C# Builder, passando per #Develop, ma in questo testo utilizzeremo esclusivamente codice scritto manualmente, in tale maniera il lettore prenderà confidenza con le classi, e potrà in seguito usufruire dei designer visuali in maniera più proficua e comprendendo perfettamente cosa avviene dietro le quinte.
In genere un'applicazione interattiva a finestre, ha almeno una finestra che si presenta all'utente all'avvio del programma. Tale finestra è creata in .NET mediante una classe ereditata dalla classe **System.Windows.Forms.Form**, popolata poi da controlli vari contenuti nello stesso namespace o implementabili in maniera personalizzata dallo sviluppatore.

```
using System.Windows.Forms;
class Form1: Form
{
      …
}
```

8.1 Applicazioni a finestre

Il ciclo di vita di un'applicazione parte con la creazione della form principale e la sua visualizzazione a schermo, e termina con l'adeguata pulitura delle risorse impegnate e la distruzione della form.
In genere lo startup dell'applicazione in C# avviene per mezzo del metodo statico `Run` della classe **Application**, che prende come unico parametro in ingresso un'istanza di una classe derivata da Form, quindi in genere un'istanza della form principale della nostra applicazione. Il metodo Run è invocato all'interno del metodo Main del programma:

```
static void Main()
{
      Application.Run(new Form1());//Form1 è la nostra classe Form principale
}
```

La chiamata al metodo Run provoca la visualizzazione dell'istanza della Form passata come argomento, ed a questo punto l'applicazione resta in attesa dell'interazione dell'utente o di altri eventi.
Il metodo Application.Run è invocabile anche senza alcun parametro, in tal caso non verrà mostrata alcuna finestra principale, che dovrà essere creata e visualizzata manualmente, come vedremo nel successivo paragrafo.
La classe Application fornisce un metodo complementare a Run, il metodo `Exit` che provoca lo shutdown immediato del programma e quindi la chiusura delle finestre dell'applicazione nel caso in cui abbiamo invocato il metodo Run passando come parametro una form.
E' comunque sconsigliabile utilizzare il metodo Exit, piuttosto conviene utilizzare il metodo `Close` sulla finestra principale, in modo da poter gestire anche il rilascio delle risorse occupate, dall'applicazione.

8.2 Compilazione dell'applicazione

La compilazione di un'applicazione Windows viene effettuata solitamente utilizzando il compilatore csc con l'opzione `/target:winexe`. In tale maniera all'avvio dell'applicazione non verrà visualizzata la console.

```
csc.exe /target:winexe nomeapp.cs
```

In fase di test e di debug dell'applicazione è comunque ancora utile la compilazione tramite il normale `/target:exe`, applicato di default, in quanto è così possibile stampare sulla console delle informazioni su quanto avviene nell'applicazione in esecuzione.

8.3 La classe Form

Una Form del framework .NET è quella che in genere siamo abituati a chiamare finestra (window). Tralasciando una descrizione di cosa sia una form, cosa che ormai è un concetto radicato in ogni utente di sistemi operativi più o meno moderni, vediamo come crearla con le classi messe a disposizione dalla BCL, ed in particolare con la classe **Form**, contenuta nel namespace System.Windows.Forms.
In genere la nostra applicazione avrà una form di avvio o form principale, che verrà creata come istanza di una classe derivata da Form:

```
using System.Windows.Forms;
class MainForm: Form
{
      static void Main()
      {
            MainForm f=new MainForm();
      }
}
```

L'istruzione new, come sappiamo, crea un'istanza di MainForm, ma ciò non è sufficiente a visualizzarla sullo schermo, ciò che è necessario fare per mostrare una finestra è invocarne il metodo Show

```
MainForm f=new MainForm();
f.Show();
```

un'alternativa è l'assegnazione del valore `true` alla proprietà `Visible`.
Il metodo opposto, cioè quello per nascondere (ma non distruggere) una finestra, è il metodo `Hide`:

```
f.Hide();
```

La finestra creata è mostrata nell'esempio precedente, naturalmente sarà una finestra senza alcun controllo, senza alcun titolo, e con dimensioni di default, ma probabilmente non farete neanche in tempo ad osservarla dato che subito dopo la chiamata al metodo Show, il flusso di esecuzione del codice ritornerà al metodo Main, e quindi l'applicazione termina uscendo dallo stesso metodo Main.
Ciò fornisce una motivazione del fatto per cui è utilizzato il metodo Application.Run per la visualizzazione della form principale del programma, mentre per le form che saranno generate dal programma in un secondo tempo, sarà sufficiente invocarne il metodo Show.
Creiamo ad esempio nel costruttore della form principale altre due form, e diamo un titolo a tutte e tre le finestre, utilizzando la proprietà Text, per distinguerle fra di loro.

```
class ShowForm:Form
{
      public ShowForm()
      {
            Form form2=new Form();
```

```
            form2.Text="Form2";
            form2.Show();
            Form form3=new Form();
            form3.Text="Form3";
            form3.Visible=true;
      }

      public static void Main()
      {
            ShowForm form = new ShowForm();
            form.Text="Main";
            Application.Run(form);
      }
}
```

Eseguendo il codice, appariranno le tre form, delle quali quelle con titolo "form2" e "form3" saranno state generate dalla prima form. Se infatti provate a chiudere la form principale, l'applicazione terminerà, chiudendo tutte le form ancora aperte. Le form secondarie sono invece indipendenti, e possono essere chiuse senza che naturalmente ne risenta l'intera applicazione. Questo comportamento è stato ottenuto grazie al fatto che abbiamo passato al metodo Application.Run() un parametro di classe Form, in caso contrario come già detto, è necessario invocare il metodo Application.Exit() per chiudere il programma.
Con l'esempio seguente la finestra verrà visualizzata, ma una volta chiusa questa, l'applicazione resterà appesa, ed infatti vi accorgerete che la linea Console.Write() non verrà mai eseguita.

```
public class ShowForm2
{
      public static void Main()
      {
            ShowForm2 form = new ShowForm2();
            form.Text="Main";
            Application.Run();
            Console.Write("Da qui non ci passo");
      }
}
```

Per provarlo compilate con target:exe ed eseguite l'applicazione dal prompt del dos, alla chiusura della form, la console rimarrà aperta e bloccata, costringendovi ad un CTRL+C per terminarla.
Comunque, anche se compilate con target:winexe, vi accorgerete dal task manager, che nonostante la form sia stata chiusa, e ancora in esecuzione l'applicazione.
Tale comportamento è un altro buon motivo per compilare le applicazioni ancora da testare e debuggare con target:exe, in tal modo infatti potrete chiudere con il CTRL+C nella console un'applicazione bloccata, senza dover ricorrere al Task Manager.

8.3.1 MessageBox

La **MessageBox** è un tipo di form particolare, utilizzata per mostrare la classica finestra di messaggio all'utente, ed eventualmente con l'aggiunta di uno o più pulsanti per ottenere una conferma o far effettuare una scelta durante l'esecuzione dell'applicazione.
Il seguente codice implementa un classico Hello World, stavolta però stampandolo all'interno di una MessageBox mostrata sullo schermo, come mostrato nella figura 8.1.

```
using System.Windows.Forms;
class MessageBoxHelloWorld
{
     public static void Main()
     {
          MessageBox.Show("Hello world in una MessageBox!");
     }
}
```

Figura 8.1 L'Hello World in una MessageBox

La classe MessageBox possiede un unico metodo, statico fra l'altro, il metodo `Show`.
Del metodo MessageBox.Show sono disponibili diversi overload, che permettono di personalizzare in vari modi la visualizzazione della MessageBox, i principali fra di essi sono riassunti nella tabella seguente:

MessageBox.Show
DialogResult MessageBox.Show(string text)
DialogResult MessageBox.Show(string text,string caption)
DialogResult MessageBox.Show(string text,string caption,MessageBoxButtons buttons)
DialogResult MessageBox.Show(string text,string caption,MessageBoxButtons buttons,MessageBoxIcon icon)
DialogResult MessageBox.Show(string text,string caption,MessageBoxButtons buttons,MessageBoxIcon icon,MessageBoxDefaultButton defaultButton)
DialogResult MessageBox.Show(string text,string caption,MessageBoxButtons buttons,MessageBoxIcon icon,MessageBoxDefaultButton defaultButton,MessageBoxOptions options)

Tabella 7 Gli overload di MessageBox.Show

Il primo esempio fatto utilizzava il primo metodo dell'elenco. Se volessimo far apparire un testo nella barra del titolo della MessageBox, sarebbe sufficiente utilizzare il metodo con i due parametri string, di cui il secondo specifica appunto il testo da visualizzare:

```
MessageBox.Show("Hello world","titolo della finestra");
```

Una MessageBox può contenere più pulsanti, fino ad un massimo di tre. Il tipo enumerativo `MessageBoxButtons` permette di specificarne il tipo ed il numero, utilizzando uno dei valori riportati nella tabella, il testo esatto dipende comunque dalla lingua del sistema operativo, ad esempio il pulsante Cancel sarebbe visualizzato con il testo Annulla nella versione italiana.

Membro	Descrizione
OK	Visualizza il pulsante Ok.
OKCancel	Visualizza i pulsanti Ok ed Annulla..
AbortRetryIgnore	Visualizza i tre pulsanti Interrompi, Riprova, Ignora.
RetryCancel	Visualizza i pulsanti Riprova e Annulla.
YesNo	Visualizza i pulsanti Sì e No.
YesNoCancel	Visualizza i pulsanti Sì, No e Annulla.

Tabella 8 L'enumerazione MessageBoxButtons

Come già visto, se non si utilizza nessuna di queste opzioni, verrà visualizzato solo il pulsante OK.
La scelta dell'utente, cioè il pulsante su cui esso cliccherà, verrà restituito come valore di ritorno del metodo, che è di tipo enumerativo `DialogResult` ed i cui membri sono appunto quelli visualizzabili come pulsanti, cioè OK, Cancel, Abort, Retry, Ignore, Yes, No, ed infine None, per indicare che nessun pulsante è stato premuto, e dunque non verrà mai restituito dal metodo MessageBox.Show:

```
DialogResult result=MessageBox.Show("Vuoi continuare?","Conferma",MessageBoxButtons.YesNo);
if(result==DialogResult.Yes)
{
      MessageBox.Show("L'utente ha scelto di continuare");
}
```

Il parametro `MessageBoxIcon` è utilizzato per visualizzare un'icona all'interno della MessageBox, un'icona indicante un particolare tipo di evento, ad esempio un errore irreversibile, o una semplice informazione all'utente, inoltre se il sistema operativo ha un particolare suono associato all'evento, esso verrà riprodotto automaticamente all'apparizione della MessageBox. L'enumerazione MessageBoxIcon ha i membri riportati nella tabella seguente, notate però che attualmente esistono solo quattro simboli diversi, dunque alcuni sono assegnati a più membri, come ad esempio nel caso di Warning ed Exclamation.

Membro	Descrizione
Asterisk	Visualizza una i blu minuscola in un fumetto bianco.
Error	Visualizza una croce bianca in un cerchio rosso.
Exclamation	Visualizza un punto esclamativo nero in un triangolo giallo.
Hand	Visualizza una croce bianca in un cerchio rosso.
Information	Visualizza una i blu minuscola in un fumetto bianco.
None	Non visualizza nessuna icona.
Question	Visualizza un punto interrogativo blu in un fumetto bianco.
Stop	Visualizza una croce bianca in un cerchio rosso.
Warning	Visualizza un punto esclamativo nero in un triangolo giallo.

Tabella 9 L'enumerazione MessageBoxIcon

Il seguente codice visualizza tante MessageBox in sequenza, una per ogni diverso possibile valore che può assumere il parametro MessageBoxIcon:

```
using System;
using System.Windows.Forms;
public class TestMessageBoxIcon
{
      public static void Main()
      {
            Array icons=Enum.GetValues(typeof(MessageBoxIcon));
            foreach(MessageBoxIcon icon in icons)
            {
                  MessageBox.Show("Visualizzo un'icona "+icon,
                              "MessageBoxIcon "+icon,
                              MessageBoxButtons.OK,
                              icon);
            }
      }
}
```

Se in una MessageBox vengono visualizzati due o tre pulsanti, è possibile specificare quale fra essi sarà il pulsante di default, cioè il pulsante che sarà evidenziato quando la MessageBox viene visualizzata. Tale possibilità viene data utilizzando il parametro di tipo MessageBoxDefaultButton, che è ancora un tipo enumerativo che può assumere i valori Button1, Button2, Button3.
Ad esempio invocando il metodo Show in questa maniera:

```
MessageBox.Show("Hello",
            "MessageBox",
            MessageBoxButtons.AbortRetryIgnore,
            MessageBoxIcon.Question,
            MessageBoxDefaultButton.Button2
            );
```

Il pulsante selezionato di default sarà il pulsante Retry (o Riprova per i sistemi in italiano).
L'ultimo possibile parametro è ancora di un tipo enumerativo, MessageBoxOptions, che comunque è usato molto raramente.

8.3.2 Finestre di dialogo

La classe Form può essere usata anche per creare finestre modali, cioè le cosiddette finestre di dialogo, degli esempi sono le dialog di apertura e salvataggio di un file o quelle per impostare la stampa. Una form visualizzata in maniera modale permette l'interazione dell'utente, tramite tastiera o mouse, solo sugli oggetti contenuti nella stessa form, e non su altre finestre della stessa applicazione.
L'unica differenza, rispetto a quanto visto finora, per mostrare una form in maniera modale, è di invocare il metodo `ShowDialog` invece che il già visto Show.
Alla chiusura di una finestra di dialogo deve essere restituito al metodo chiamante un valore `DialogResult`, illustrato nel paragrafo relativo alla classe MessageBox.
Tale valore deve dunque essere impostato mediante la proprietà Form.DialogResult, quindi deve essere resa invisibile la form mediante il metodo `Hide`, in modo che il chiamante può leggere il valore impostato, e sarà il metodo chiamante ad effettuare la chiusura della form.
Se la finestra di dialogo viene chiusa mediante l'icona con il pulsante di Close (la classica X in alto a destra), la proprietà DialogResult viene impostata automaticamente a Cancel.
Non tutte le finestre di dialogo hanno i due pulsanti OK e Annulla, ma il testo potrebbe essere diverso ad esempio se si tratta di una finestra per impostare delle proprietà è probabile che ci sarà un pulsante Imposta, oppure piuttosto che un Annulla ci sarà un Chiudi. Inoltre in molte finestre di dialogo è possibile confermare le scelte semplicemente premendo il tast Invio, piuttosto che annullare le scelte fatte premendo il tasto ESC.
Impostando le proprietà AcceptButton e CancelButton è possibile scegliere quali pulsanti verranno rispettivamente attivati alla pressione di Invio e di ESC.

```
public void CreateMyForm()
{
   Form form1 = new Form();
   // Crea 2 Button da usare per accept e cancel.
   Button button1 = new Button ();
   Button button2 = new Button ();

   button1.Text = "Conferma";
   button1.Location = new Point (10, 10);
   button2.Text = "Esci";
   button2.Location = new Point (button1.Left, button1.Height + button1.Top + 10);

   form1.AcceptButton = button1;
   form1.CancelButton = button2;

   form1.Controls.Add(button1);
   form1.Controls.Add(button2);

   form1.ShowDialog();
}
```

8.3.3 Le Common Dialog

Il framework .NET fornisce delle classi per utilizzare le comuni dialog delle applicazioni Windows, chiamate appunto Common Dialog. Queste classi, contenute nel namespace System.Windows.Forms e derivate dalla classe CommonDialog ad eccezione di PrintPreviewDialog, sono elencate nella tabella seguente, e permettono di visualizzare le finestre di dialogo relative.

Metodo	Descrizione
`OpenFileDialog`	Rappresenta la finestra di dialogo per la scelta di un file da aprire.
`SaveFileDialog`	Rappresenta la finestra di dialogo per il salvataggio di un file.
`PrintDialog`	Rappresenta la finestra di dialogo per l'impostazione delle proprietà di stampa.
`PrintPreviewDialog`	Rappresenta la finestra di dialogo per la

	visualizzazione dell'anteprima di stampa di un documento.
PageSetupDialog	Rappresenta la finestra di dialogo per l'impostazione del formato di stampa, come la dimensione e i margini delle pagine.
FontDialog	Visualizza e permette di scegliere uno dei font installati nel sistema operativo.
ColorDialog	Visualizza e permette la selezione di un colore standard del sistema e la definizione di colori personalizzati.

Anche la visualizzazione delle Common Dialog avviene per mezzo del metodo ShowDialog.
Per selezionare ed aprire uno o più file esistenti, viene utilizzata la classe `OpenFileDialog`. La proprietà `Title` imposta o ricava la stringa che appare sulla barra del titolo.
E' possibile impostare il filtro per la tipologia dei file da visualizzare, specificando una o più estensioni e la relativa descrizione per mezzo della proprietà `Filter`, la proprietà `FilterIndex` permette invece di selezionare il filtro predefinito, nel caso in cui la `Filter` specifichi più tipi di file.

```
OpenFileDialog dlg=new OpenFileDialog();
dlg.Title="Apri un file C#";
dlg.Filter="C# Files (*.cs)|*.cs|Text Files (*.txt)|*.txt|All Files (*.*)|*.*";
dlg.FilterIndex=2;
dlg.InitialDirectory=@"C:\temp";
dlg.ShowReadOnly=true;
dlg.ReadOnlyChecked=true;
dlg.CheckFileExists=false;

if(dlg.ShowDialog()==DialogResult.OK)
{
      MessageBox.Show("Apertura file "+dlg.FileNames[0]);
}
```

8.3.4 Proprietà e metodi delle form

La classe Form ha un grande numero di proprietà e metodi, sia propri che ereditati dalle classi base e soprattutto dalla classe Control, e quindi valide anche per i vari controlli posizionabili su una form che vedremo nei successivi paragrafi. E' impossibile fornire una trattazione completa dei membri della classe Form, ma cercheremo di fornire qualche esempio significativo, mostrando come definirne vari aspetti e comportamenti, grafici e non.
Abbiamo già incontrato la proprietà `Text` che permette di impostare il testo che apparirà sulla barra del titolo della finestra.

```
Form f=new Form();
f.Text="Titolo della finestra";
```

L'icona di una form è invece impostata o ricavata mediante la proprietà `Icon`. Ad esempio per caricare un'icona da un file possiamo scrivere:

```
f.Icon=new Icon("hello.ico");
```

Le dimensioni e la posizione di una Form sono configurabili in varie maniere, ad esempio è possibile usare la proprietà `Size` per impostare larghezza e altezza, o singolarmente `Width` per la prima e `Height` per la seconda:

```
f.Size=new Size(300,400);
```

è equivalente alle assegnazioni:

```
f.Width=300;
f.Height=400;
```

La dimensione dell'area client, cioè dell'area della form escludendo la barra del titolo ed i bordi è invece rappresentata dalla proprietà `ClientSize`.
La posizione sullo schermo è rappresentata dalla proprietà `Location` di classe `Point`, mentre la posizione rispetto al desktop, che tiene quindi conto ad esempio di barre degli strumenti o della barra delle applicazioni, è rappresentata dalla proprietà `DesktopLocation`:

```
f.Location=new Point(10,10);
```

La proprietà `Bounds` e, analogamente a sopra, la proprietà `DesktopBounds`, permettono di impostare in un colpo solo dimensione e posizione della form per mezzo di un oggetto Rectangle:

```
f.Bounds=new Rectangle(10,10, 300,400); //x, y, larghezza e altezza
Point p=new Point(10,10);
Size s=new Size(300,400);
f.DesktopBounds=new Rectangle(p,s);
```

Se si vuole dare una dimensione massima e una minima alla form basta assegnare i relativi valori alle proprietà `MaximumSize` e `MinimumSize`.

```
f.MinimumSize=new Size(200,150);
f.MaximumSize=new Size(800,600);
```

Una delle caratteristiche più implementate dagli sviluppatori è forse il centramento di una form sullo schermo, ad esempio all'avvio dell'applicazione, cosa che in C# è ottenibile semplicemente per mezzo della proprietà `StartPosition` che può assumere valori del tipo enumerativo `FormStartPosition`:

```
f.StartPosition=FormStartPosition.CenterScreen;//centra la form sullo schermo
f.StartPosition=FormStartPosition.CenterParent;//centra rispetto alla form madre
```

I bordi di una finestra possono assumere vari aspetti, ad esempio possono permettere di ridimensionarla con l'uso del mouse, o di renderla di dimensioni fisse, oppure di nascondere la barra del titolo, o ancora di eliminare completamente i bordi stessi. L'enumerazione `FormBorderStyle` contiene tutti i valori che permettono di definire tali aspetti, mediante l'assegnazione all'omonima proprietà Form.FormBorderStyle. Se il bordo è visibile, è possibile anche agire sui pulsanti di riduzione a icona, di massimizzazione, di help, o sul menù di sistema, assegnando dei valori booleani alle proprietà booleane `MinimizeBox`, `MaximizeBox`, `ControlBox`. Il seguente esempio crea una form adatta all'utilizzo come splash screen:

```
public SplashForm()
{
      this.FormBorderStyle = FormBorderStyle.None;
      this.MaximizeBox = false;
      this.MinimizeBox = false;
      this.StartPosition = FormStartPosition.CenterScreen;
      this.ControlBox = false;
}
```

Naturalmente una splash form è più accattivamente se usiamo qualche altro effetto grafico e magari un po' di colori.
Il colore di sfondo della form viene impostato agendo sulla proprietà `BackColor`:

```
this.BackColor=Color.Red;
```

Con l'uso della proprietà `Opacity` possiamo invece dare un effetto di trasparenza. Il suo valore può variare da 1.0, che indica una form non trasparente, mentre con 0.0 sarebbe completamente trasparente.
Potremmo allora fare apparire la form in modo graduale, con un ciclo che vari la Opacità, sarebbe ideale farlo in un thread separato per non bloccare l'applicazione:

```
public void ShowSplash()
{
      this.Opacity=0.0;
      this.Visible=true;
      for(double i=0;i<=1.0;i+=0.05)
      {
            this.Opacity=i;
            System.Threading.Thread.Sleep(100);//si ferma per 100 millisecondi
      }
      this.Opacity=1.0;
}
```

8.4 Aggiungere i controlli

Una form può contenere diversi tipi di controlli, in generale un controllo deriva dalla classe `System.Windows.Forms.Control`, che definisce le funzionalità di base per un componente che deve mostrare o ricavare informazioni dall'utente, come ad esempio una etichetta o una casella di testo con il quale esso può interagire, ad esempio pulsanti e barre di scorrimento.
La classe Form mantiene in una collezione i controlli in essa contenuti, la proprietà `Controls` permette di ricavare tale collezione ed utilizzarne i metodi per aggiungere e rimuovere ogni singolo controllo. Ad esempio per creare una casella di testo ed aggiungerla alla collezione di controlli della form è sufficiente scrivere il codice seguente:

```
Form f=new Form();
TextBox txt=new TextBox();
txt.Location=new Point(10,10);
txt.Size=new Size(100,20);
f.Controls.Add(txt);
```

mentre per rimuovere lo stesso controllo, nel caso sia stato precedentemente aggiunto alla collezione, basta invocare il metodo `Remove`:

```
f.Controls.Remove(txt);
```

Oltre ad aggiungere esplicitamente il controllo alla collezione, è possibile anche utilizzare la proprietà `Parent` della classe Control:

```
Button bt=new Button();
bt.Text="Clicca";
bt.Location=new Point(10,50);
bt.Parent=f;
bt.Parent=f;
```

Assegnare alla proprietà bt.Parent una form f, come nell'esempio, è equivalente all'utilizzo del metodo f.Controls.Add(bt), mentre assegnando il valore null alla proprietà Parent, si rimuove il controllo dalla collezione Controls della form Parent.
Creando una form come nell'esempio precedente, otterremo una finestra dall'utilità ancora molto limitata, se non quella di vedere il risultato dei nostri primi sforzi con le Windows Forms.

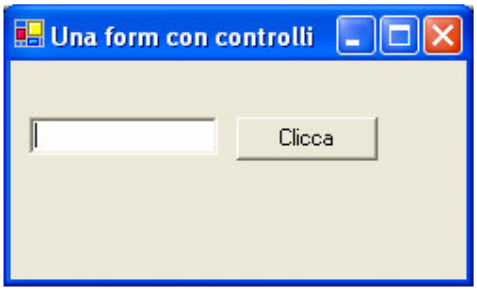


Figura 8.2 Una form con due controlli

E' possibile anche creare array di controlli ed aggiungerli in un colpo solo tramite il metodo `AddRange` della collection Controls, che prende come parametro appunto un array di oggetti Control:

```
TextBox[] caselle=new TextBox[4];
f.Controls.AddRange(caselle);
```

Oppure, in maniera forse ancor più comoda, una volta creati i vari controlli di vario tipo:

```
TextBox txt=new TextBox();
Label lab=new Label();
Button bt=new Button();
…
Controls.AddRange(new Control[]{txt,lab,bt});
```

8.4.1 Proprietà dei controlli

La Base Class Library fornisce le classi per creare e gestire tutti i classici componenti visuali delle applicazioni Windows. La classe principale da cui derivano tutti i controlli è la classe `Control`, pulsanti, label, check-box, barre di scorrimento, list-box e quant'altro sono tutti esempi di controlli derivati direttamente od indirettamente da Control.
La classe Control espone una serie di proprietà comuni alla maggior parte dei controlli, le principali di esse sono elencate nella Tabella 10. Molte proprietà sono impostabili e leggibili anche a runtime, mentre altre sono di sola lettura.

Proprietà	Descrizione
Anchor	Specifica quali lati del controllo sono ancorati al suo contenitore.
BackColor	Imposta o ricava il colore di sfondo del controllo.
BackgroundImage	Imposta o ricava l'immagine di sfondo mostrata sulla superficie del controllo.
Bottom	Ricava la distanza fra il lato inferiore del controllo e il lato superiore del contenitore.
Bounds	Imposta o ricava la posizione e le dimensioni del controllo.
Cursor	Imposta o ricava il puntatore del mouse da visualizzare quando esso si trova sulla superficie del controllo.
Dock	Specifica quale lato del controllo è legato al corrispondente lato del contenitore, ad esempio Imposta o ricavando il valore a DockStyle.Fill, il controllo riempie sempre l'area del contenitore.
Enabled	Specifica se il controllo è abilitato o meno.
Focused	Indica se il controllo ha il focus.
Font	Imposta o ricava il tipo di carattere per il testo mostrato dal controllo.
ForeColor	Imposta o ricava il colore del testo mostrato dal controllo.
Heigth	Imposta o ricava l'altezza del controllo
Left	Imposta o ricava la coordinata x del lato sinistro del controllo.
Location	Imposta o ricava la posizione dell'angolo superiore sinistro del controllo.
Name	Imposta o ricava il nome del controllo, per default esso è una stringa vuota.

Parent	Imposta o ricava il controllo contenitore del controllo corrente.
Right	Ricava la distanza fra il lato destro del controllo e il lato sinistro del suo contenitore.
Size	Imposta o ricava le dimensioni del controllo.
TabIndex	Imposta o ricava il valore di tab order del controllo. Il tab order determina l'ordine con cui i controlli ricevono il focus alla pressione del tasto TAB.
TabStop	Specifica se il controllo può ricevere il focus alla pressione del tasto TAB.
Tag	Imposta o ricava un oggetto associato al controllo e che può contenere dati su di esso.
Text	Imposta o ricava il testo associato al controllo. Ad esempio per una form è il testo sulla barra del titolo, per una TextBox è il testo contenuto in essa.
Top	Imposta o ricava la coordinata y del lato superiore del controllo.
Visible	Specifica se il controllo viene visualizzato sul contenitore.
Width	Imposta o ricava la larghezza del controllo.

Tabella 10: Le proprietà principali della classe Control

Le proprietà della classe Control, insieme ai suoi metodi, sono estese da eventuali altri membri specializzati delle classi derivate da Control. Nei prossimi paragrafi vedremo alcuni dei controlli più utili, e qualche esempio introduttivo su di essi, senza pretesa alcuna di essere esaustivi, ma con l'intento di stuzzicare il lettore ed ispirarne lo studio dell'argomento.

8.4.2 Controlli di testo

I controlli di testo permettono di visualizzare o ricavare il testo digitato dall'utente. Sicuramente fra i più utilizzati è il controllo `TextBox`, il classico campo di testo:

```
TextBox txt=new TextBox();
…
txt.Text="Clicca";
txt.MaxLength=20;//imposta la lunghezza massima del testo
```

Lavorare con il testo contenuto nella textBox è possibile anche con i tanti metodi della classe TextBox, ad esempio per selezionare una parte di testo contenuto in essa, o una parte di esso, è possibile utilizzare uno degli overload del metodo `Select` oppure `SelectAll` per selezionare tutto il testo.. Il seguente esempio ricerca una stringa all'interno di una casella, ed eventualmente la evidenzia:

```
string searchString = "hello";
int index = txt.Text.IndexOf(searchString, 0);
if (index != -1)
       txt.Select(index, searchString.Length);
```

La proprietà `SelectedText` conterrà a questo punto il testo selezionato, mentre `SelectionLength` è la lunghezza della stringa selezionata.
La classe TextBox può essere configurata per contenere più linee di testo per mezzo di una serie di altre proprietà, riassunte nel seguente frammento di codice:

```
txt.Multiline = true;
txt.ScrollBars = ScrollBars.Vertical;
txt.WordWrap = true;
txt.Lines = new String[]{"linea1","linea2","linea3"};
```

Un'altra proprietà utile è `PasswordChar`, che permette di impostare il carattere da visualizzare se la TextBox dovrà contenere una password:

```
txt.PasswordChar='*';
```

La classe TextBox deriva dalla classe `TextBoxBase`, insieme alla classe `RichTextBox`, utilizzabile quando è necessario un controllo con funzionalità più avanzate per la manipolazione del testo. Una RichTextBox è una casella di testo che può contenere testo in formato RTF, permettendo anche di caricare e salvare il suo contenuto su file, o ad esempio di formattare parti di testo con colori e caratteri differenti.

```
rtf = new RichTextBox();
rtf.Dock = DockStyle.Fill;
//carica il file, se non esiste genera un'eccezione
rtf.LoadFile("C:\\temp\\documento.rtf");

//cerca la stringa hello
rtf.Find("hello");

//colora la seleziona e cambia il font
rtf.SelectionFont = new Font("Verdana", 12, FontStyle.Bold);
rtf.SelectionColor = Color.Red;

//salva il file
rtf.SaveFile("C:\\temp\\documento.rtf", RichTextBoxStreamType.RichText);
```

La classe `Label` rappresenta la classica etichetta descrittiva, associata ad esempio ad un altro controllo:

```
Label lab=new Label();
lab.Text="etichetta";
```

Una label può anche contenere un'immagine, impostata per mezzo della proprietà `Image` o delle proprietà `ImageList` e `ImageIndex`, la prima contiene una lista di oggetti Image, ad esempio potrebbe contenere tutte le immagini da utilizzare in una form, mentre la seconda indica quale immagine della lista associare alla Label:

```
Label labImage=new Label();
ImageList immagini=new ImageList();
immagini.Images.Add(Image.FromFile("immagine.bmp");

labImage.ImageList=immagini;
labImage.ImageIndex=0;
labImage.Text="Icona";
```

Un nuovo controllo introdotto nella Base Class Library di .NET è `LinkLabel`, cioè una label il cui testo può però contenere uno o più collegamenti ipertestuali.
Per impostare un link è necessario utilizzare in particolare la proprietà `Links`, che contiene una collezione dei collegamenti, istanze di classe `LinkLabel.Link`. Mentre al click su uno di essi è necessario gestire l'evento `LinkClicked`. Nell'esempio seguente la LinkLabel link, contiene un testo con due diversi collegamenti, uno al sito www.hello.com, l'altro alla directory locale C:\windows, come in Figura 8.3

```
link.Text=@"Clicca per accedere al sito."+Environment.NewLine+@"Apri la directory C:\windows";
link.Links.Add(23,4,"www.hello.com");
link.Links.Add(48,10,"C:\\windows");
link.LinkClicked+=new LinkLabelLinkClickedEventHandler(link_LinkClicked);
```

Figura 8.3 Una LinkLabel con due collegamenti

Per mezzo del metodo `Add`, vengono aggiunti due collegamenti, impostando l'inizio del testo da visualizzare come link, e la sua lunghezza. Il gestore dell'evento LinkClicked potrebbe essere il seguente:

```
private void link_LinkClicked(object sender, LinkLabelLinkClickedEventArgs e)
{
      string target=e.Link.LinkData.ToString();
      LinkLabel label=(LinkLabel)sender;
      label.Links[label.Links.IndexOf(e.Link)].Visited=true;
      System.Diagnostics.Process.Start(target);
}
```

8.4.3 Controlli di comando

Abbiamo già visto nel precedente paragrafo come creare e posizionare su una form un pulsante, utilizzando la classe Button, ed il procedimento naturalmente sarà analogo per ogni altro controllo.
La classe `Button` deriva dalla classe astratta ButtonBase, la quale deriva a sua volta da Control.
La funzionalità tipica di un pulsante è naturalmente quella di eseguire una certa azione quando l'utente clicca su di esso, oppure quando il pulsante ha il focus e vengono premuti i tasti Enter o Spazio.
Per ottenere una notifica di tale evento bisogna associare al pulsante un gestore adeguato:

```
Class TestButton:Form
{
      public TestButton()
      {
            Button bt=new Button();
            bt.Text="Clicca";
            bt.Location=new System.Drawing.Point(10,10);
            bt.Parent=this;

            bt.Click+=new EventHandler(bt_Click);
      }

      private void bt_Click(object sender, EventArgs e)
      {
            Button button=(Button)sender;
            MessageBox.Show("Hai cliccato "+button.Text);
      }
}
```

L'istruzione:

```
bt.Click+=new EventHandler(bt_Click);
```

Installa un nuovo gestore per l'evento Click, di nome bt_Click e che rispetta la firma definita dal delegato EventHandler.
Il primo parametro sender, rappresenta l'oggetto che ha generato l'evento, in questo caso il pulsante bt. Essendo tale parametro di tipo object, per accedere alle proprietà di bt, è necessario un cast esplicito:

```
Button button=(Button)sender;
```

L'evento Click non è un'esclusiva della classe Button, ma è ereditato dalla classe Control, dunque ogni altro controllo può ricevere una notifica di tale evento, e la gestione è analoga a quanto visto per un Button.

La classe `NotifyIcon` permette di aggiungere un'icona all'area di notifica della barra delle applicazioni, quella cioè dove di norma appare l'orologio. Una NotifyIcon permette di visualizzare informazioni su un processo in esecuzione, e di gestirlo ad esempio aggiungendo un menù contestuale.
Creare una NotifyIcon con un tooltip associato, è semplice come creare un qualsiasi altro controllo:

```
notify=new NotifyIcon();
```

```
notify.Text="Esempio di NotifyIcon";
notify.Icon=new Icon(@"c:\temp\iconcsharp.ico");
notify.Visible=true;
```

Il codice precedente visualizza sull'area di notifica un'icona con un tooltip, che appare quando si posiziona su di essa il puntatore del mouse, come in Figura 8.4

Figura 8.4 Un'icona di notifica con un tooltip

E' possibile associare un menù contestuale, mediante la proprietà ContextMenu, ma impareremo nel paragrafo relativo ai menù come crearne uno, intanto vediamo come aggiungerne uno fittizio, con due voci che però non eseguono alcuna azione:

```
notify.ContextMenu=new ContextMenu(
      new MenuItem[]{new MenuItem("Ripristina"),new MenuItem("Esci")}
);
```

8.4.4 Controlli di selezione

I controlli di selezione permettono di specificare dei valori o effettuare delle scelte fra più valori.
Ad esempio se vogliamo dare all'utente la possibilità di impostare un valore del tipo Abilitato/Non abilitato, o in genere un valore booleano, possiamo utilizzare il controllo `CheckBox`, cioè la classica casellina all'interno della quale appare un segno di spunta al click del mouse, e sparisce al successivo click. La classe CheckBox deriva, come la classe Button, dalla classe ButtonBase.
La proprietà Text rappresenta per la CheckBox il testo che appare a lato della casella di controllo.

```
CheckBox chk=new CheckBox();
chk.Text="Disabilita";
```

Le proprietà peculiari della classe CheckBox sono le proprietà booleane Checked ed AutoCheck. La prima rappresenta lo stato della checkbox, cioè è true se la casella è spuntata, false altrimenti.
L'evento generato quando cambia lo stato della CheckBox, cioè il valore di Checked, è l'evento CheckedChanged. Nel caso in cui la proprietà AutoCheck è settata a true, come di default, quando un utente clicca sulla CheckBox, viene variato il valore di Checked e viene generato l'evento CheckedChanged.

```
chk.CheckedChanged+=new EventHandler(chk_CheckedChanged);
…
private void chk_CheckedChanged(object sender, EventArgs e)
{
      CheckBox c=(CheckBox)sender;
      c.Text=(c.Checked?"Checked":"Unchecked");
}
```

Mentre nel caso in cui AutoCheck è false, è necessario aggiungere anche un gestore dell'evento Click che modifichi in maniera manuale la proprietà Checked:

```
chk.Click+=new EventHandler(chk_Click);
…
private void chk_Click(object sender, EventArgs e)
{
      CheckBox c=(CheckBox)sender;
      c.Checked=!c.Checked;
}
```

La modifica della proprietà Checked, anche da codice, genera ancora un evento CheckedChanged, dunque è importante fare attenzione a non variarla all'interno del gestore di tale evento, altrimenti si rimarrebbe intrappolati in un ciclo infinito.
Una checkbox può anche assumere un terzo stato, indeterminato, che vuol dire una via di mezzo fra checked e unchecked, in tale stato la checkbox appare come una casella grigia. Per abilitare tale funzionamento è necessario porre pari a true la proprietà ThreeState ed utilizzare CheckState piuttosto che la proprietà Checked:

```
chk.ThreeState=true;
chk.CheckState=CheckState.Indeterminate;
```

Se si utilizza una CheckBox a tre stati bisogna installare un gestore dell'evento CheckStateChanged.
Se invece si vuole dare alla CheckBox l'apparenza di un pulsante classico, che però mantiene lo stato, che può essere premuto o meno, cioè una sorta di interruttore, si può utilizzare la proprietà Appearance e impostarla a Button, piuttosto che a Normal:

```
chk.Appearance=Appearance.Button;
```

Un ultimo tipo derivato da ButtonBase è la classe `RadioButton`, cioè il classico controllo che permette di scegliere una ed una sola opzione fra una serie di due o più. Se posizioniamo più RadioButton all'interno di un contenitore, ad esempio una Form, o in genere un controllo GroupBox, essi costituiscono un gruppo, e selezionando uno di essi, i rimanenti vengono automaticamente deselezionati.

```
radio1=new RadioButton();
radio1.Text="Scelta1";
```

Per determinare se uno dei RadioButton di un gruppo è selezionato o meno si utilizza ancora la proprietà Checked:

```
if(radio1.Checked)
      MessageBox.Show("Hai fatto la Scelta1");
```

Se si vuol simulare via codice il click di un utente su un RadioButton, può essere utilizzato il metodo PerformClick che genera un evento Click su di esso:

```
if(txt.Text=="scelta1")
      radio1.PerformClick();
else radio2.PerformClick();
```

Il controllo **`Listbox`** permette di visualizzare una lista di elementi, selezionabili dall'utente con un click del mouse.
La proprietà MultiColumn permette di disporre gli elementi della ListBox, su più colonne, mentre per mezzo della proprietà SelectionMode si può definire la proprietà di selezione, ad esempio se si vuol dare all'utente la possibilità di selezionare più elementi, con l'uso dei tasti CTRL e Shift, bisogna impostarla al valore MultiExtended, come nel seguente esempio:

```
ListBox lb = new ListBox();
lb.Size = new System.Drawing.Size(200, 100);
lb.Location = new System.Drawing.Point(10,10);
lb.MultiColumn = true;
lb.SelectionMode = SelectionMode.MultiExtended;
```

Gli elementi vengono aggiunti alla ListBox, per mezzo della proprietà Items, che è una collezione di oggetti qualsiasi.

```
public void FillList(ListBox lb)
{
      for(int i=0;i<10;i++)
            lb.Items.Add("Elemento "+i);
```

```
}
```

La proprietà SelectedItems invece conterrà la collezione degli elementi eventualmente selezionati, mentre per selezionare o deselezionare un elemento ad un dato indice si può utilizzare il metodo SetSelected:

```
lb.SetSelected(1, true);//seleziona l'elemento di indice 1
lb.SetSelected(3, false);//deseleziona l'elemento di indice 3
```

Spesso è necessario ricercare un elemento della ListBox, contenente una stringa. Il metodo FindString permette di ottenere l'indice del primo elemento che risponde a questo criterio:

```
string str="elemento3";
int x=lb.FindString(str);
if(x>-1)
{
      MessageBox.Show(str+" si trova all'indice "+x);
}
else MessageBox.Show(str+" non trovato");
```

La classe ComboBox rappresenta una casella combinata, cioè un campo di testo, eventualmente editabile, combinato con una ListBox, da cui scegliere fra una lista di possibili valori visualizzabili cliccando sulla freccia annessa al controllo.
Una ComboBox può assumere comunque tre diversi aspetti, impostabili per mezzo della proprietà DropDownStyle.
L'enumerazione `ComboBoxStyle`, fornisce i tre possibili valori per impostare la modalità della ComboBox, ad esempio si può scrivere:

```
ComboBox cbo=new ComboBox();
cbo.DropDownStyle=ComboBoxStyle.Simple;
```

Il valore Simple visualizza un campo TextBox e una ListBox degli elementi della Combo. Il valore DropDownList invece permette di selezionare un elemento da una lista che si apre cliccando sulla freccetta. Il terzo valore, DropDown, permette anche di inserire liberamente del testo nella casella della combo.
Come per una ListBox, gli elementi contenuti nella ComboBox sono conservati nella collezione restituita dalla prorpietà Items. Per aggiungere un elemento dunque basta chiamare il metodo Items.Add o anche AddRange per aggiungere un array di elementi:

```
cbo.Items.Add("nuovo elemento");
cbo.Items.AddRange(new string[]{"a","b","c"});
```

mentre per rimuovere un elemento dalla collezione Items, possiamo utilizzare il metodo Remove o RemoveAt. Il primo rimuove l'elemento specificato mentre il secondo rimuove l'elemento ad un dato indice.

```
c.Remove("nuovo elemento");
cbo.Items.RemoveAt(0);//rimuove il primo elemento
```

L'elemento selezionato oppure il suo indice sono restituiti dalle proprietà SelectedItem e SelectedIndex rispettivamente.

```
object selected=cbo.SelectedItem;
int indexSel=cbo.SelectedIndex;
```

SelectedItem ritorna null, se nessun elemento è selezionato, mentre SelectedIndex restituirà il valore -1.
Se si vuole gestire l'evento generato quando un utente cambia l'elemento selezionato in una ComboBox, bisogna fornire un gestore dell'evento SelectedIndexChanged:

```
private void cbo_SelectedIndexChanged(object sender, System.EventArgs e)
{
      MessageBox.Show("Selezionato l'elemento "+cbo.SelectedItem);
```

```
}
```

8.4.5 ListView

Il controllo ListView permette di rappresentare una lista di elementi, caratterizzati da un testo e da una eventuale icona. Ad esempio l'esplora risorse di Windows utilizza una vista simile per visualizzare liste di file.
La listview consente inoltre di scegliere fra quattro modi diversi di visualizzazione.
Nella modalità più complessa gli elementi sono visualizzati disposti per righe e colonne. La prima colonna contiene proprio il testo associato all'elemento, mentre nelle altre eventuali colonne vengono visualizzati dei subitems associati all'item, cioè all'elemento, della riga.
Nelle altre tre modalità invece una ListView può anche visualizzare solo una lista semplice degli elementi con il testo principale di ogni elemento, senza subitems, o ancora gli elementi disposti su più colonne con associata una icona piccola, o infine gli elementi su più colonne con icone grandi.
La modalità di visualizzazione viene impostata per mezzo della proprietà View.

```
ListView lv=new ListView();
lv.View=View.Details;
```

Gli altri valori dell'enumerazione View sono List, SmallIcon, LargeIcon.
Una ListView viene popolata con elementi rappresentati da oggetti ListViewItem, contenuti nella collezione Items. Ad esempio per creare un nuovo elemento possiamo scrivere:

```
ListViewItem lvi=new ListViewItem("Elemento 1");
```

Mentre per l'eventuale testo secondario, cioè per i sottoelementi, bisogna utilizzare la collezione SubItems associata ad ogni ListViewItem:

```
lvi.SubItems.Add("1");
lvi.SubItems.Add("2");
lvi.SubItems.Add("3");
```

l'elemento creato viene quindi aggiunto agli Items della ListView:

```
lv.Items.Add(lvi);
```

mentre per rimuoverlo basta utilizzare il metodo Remove:

```
lv.Items.Remove(lvi);
```

8.4.6 TreeView

Il controllo **`TreeView`** implementa la classica vista ad albero, comune a molte applicazioni per esplorare ad esempio la struttura a directory e sottodirectory del file system, o una qualunque altra struttura gerarchica.
L'albero è rappresentato come detto dalla classe TreeView, i suoi nodi sono invece degli oggetti **`TreeNode`**, immagazzinati in una TreeNodeCollection, e che a loro volta hanno un'altra collezione dei propri nodi figli.
Un nodo è caratterizzato da una stringa e da un'eventuale immagine.
Un TreeView possiede una proprietà Nodes, di tipo TreeNodeCollection, in cui inserire tutti i suoi nodi. Dopo aver costruito un albero:

```
TreeView tree=new TreeNode();
```

possiamo aggiungere ad esso dei nodi, utilizzando il metodo Add della sua TreeNodeCollection:

```
tree.Nodes.Add("Veicoli");
```

Il nodo Veicoli, è il primo nodo dell'albero, dunque di indice 0. I nodi sono accessibili infatti tramite indicizzazione della proprietà Nodes. Se ad esempio vogliamo aggiungere un nodo figlio al nodo radice, possiamo scrivere:

```
tree.Nodes[0].Nodes.Add("Automobili");
```

è anche possibile costruire prima gli elementi TreeNode ed aggiungerli in seguito:

```
TreeNode nodeMotocicli=new TreeNode("Motocicli");
tree.Nodes[0].Nodes.Add(nodeMotocicli);
```

O anche aggiungere contemporaneamente un array di nodi con il metodo AddRange:

```
nodeMotocicli.AddRange(new TreeNode[3]{ new TreeNode("Honda"), new TreeNode("Yamaha"), new
TreeNode("Aprilia")});
```

Per navigare in profondità un TreeView dobbiamo utilizzare le proprietà Nodes che ricaviamo indicizzando i singoli nodi a vari livelli. Ad esempio:

```
TreeNode automobili=tree.Nodes[0].Nodes[0];
automobili.Nodes.Add("Alfa");
automobili.Nodes.Add("Fiat");
automobili.Nodes.Add("Lancia");
automobili.Nodes[0].Nodes.Add("GT");
automobili.Nodes[1].Nodes.Add("Idea");
automobili.Nodes[2].Nodes.Add("Lybra");
TreeNode yamaha=tree.Nodes[0].Nodes[0].Nodes[2];
tree.SelectedNode=aprilia;
```

L'ultima riga del precedente codice seleziona un nodo dell'albero, dunque dopo aver aggiunto il controllo tree ad una form, il risultato apparirebbe come in Figura 8.5

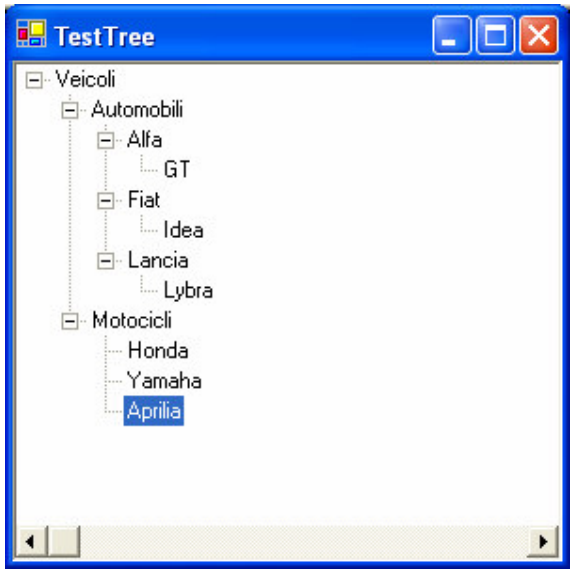


Figura 8.5 Un'albero creato con la classe TreeView

8.5 Menù

Ogni applicazione Windows che si rispetti ha un menù. I menù permettono infatti di dotare un'applicazione di diversi comandi in maniera non invasiva, cioè senza riempire una form di pulsanti e altri controlli per gestire ogni comando possibile. Ci sono due diversi tipi di menù: quello più familiare è forse quello che appare immediatamente sotto alla barra del titolo della form principale di un'applicazione, ad esempio quello contenente i classici elementi File, Modifica, Visualizza e così via. Molte applicazioni contengono inoltre dei menù contestuali, attivabili mediante il click del tasto destro su un particolare controllo dell'interfaccia grafica. Ad esempio cliccando sul foglio del editor di testi che sto utilizzando, appare un menù con i comandi taglia, copia, incolla, mentre se clicco sulla barra delle applicazioni di Windows posso impostarne una serie di proprietà.
La classe Menu, del namespaceSystem.Windows.Forms, costituisce la classe base di ogni tipo di menù.
Dalla classe Menu derivano la classe MainMenu, che rappresenta il menù classico associabile ad una form, mentre ContextMenu permette di realizzare i menù contestuali associati ai singoli controlli.
Ogni menù è costituito da una serie di oggetti MenuItem, la cui classe deriva anch'essa da Menu.

8.5.1 Il menù di un'applicazione

La classe MainMenu rappresenta il contenitore per la struttura dei menù di una form. Come detto un menù è costituito da un array di oggetti MenuItem. Dunque costruire un MainMenu può essere fatto con due diversi costruttori:

```
MainMenu();
MainMenu(MenuItem[]);
```

Per associare ad una form un menù, basta impostare la sua proprietà Menu:

```
MainMenu menuPrincipale=new MainMenu(menuItems);
miaForm.Menu= menuPrincipale;
```

I singoli MenuItem invece possono essere costruiti in diversi modi, ad esempio, per impostarne semplicemente il testo:

```
MenuItem mnuFileNuovo=new MenuItem();
mnuFileNuovo.Text="&Nuovo";
MenuItem mnuFileApri=new MenuItem("Apri…");
MenuItem mnuSeparatore=new MenuItem("-");
```

Il simbolo & prima di una lettera permette di visualizzare con una sottolineatura su tale lettera il testo, e quindi di avviare il commando con la pressione del tasto ALT+lettera, mentre un singolo trattino crea una linea di separazione.
Per creare invece uno dei menù di livello superiore, cioè quelli che appaiono sotto la barra del titolo, viene di solito usato il costruttore seguente:

```
MenuItem mnuFile=new MainMenu("&File",new MenuItem[]{mnuFileNuovo,mnuFileApri});
```

I menu finora creati non eseguono alcuna azione, in quanto non abbiamo aggiunto nessun gestore dell'evento Click:

```
mnuFileApri.Click+= new System.EventHandler(this. mnuFileApri_Click);
```

e naturalmente bisogna implementare il relativo metodo di gestione:

```
private void mnuFileApri_Click(object sender, System.EventArgs e)
{
```

```
        OpenFileDialog fd = new OpenFileDialog();
        fd.ShowDialog();
}
```

La classe MenuItem mette a disposizione diverse proprietà, per creare menù con funzionalità più avanzate, o ad esempio per impostarle se abbiamo instanziato il MenuItem con il costruttore di default. Fra queste citiamo Enabled per abilitare o disabilitare il MenuItem, Checked se si vuole posizionare un segno di spunta di fianco al MenuItem, o ancora RadioChecked se si vuole implementare un menù con MenuItem esclusivi.
La classe ContextMenu rappresenta un menù contestuale. Un menù contestuale viene costruito in maniera identica ad un MainMenu, l'unica differenza è che esso può essere attivato ciccando con il tasto destro su un controllo qualsiasi., impostando la proprietà ContextMenu, ad esempio, per una NotifyIcon:

```
notify.ContextMenu=new ContextMenu(
        new MenuItem[]{new MenuItem("Ripristina"),new MenuItem("Esci")}
);
```

Capitolo 9

9 Cenni di input/output

In questo capitolo vedemo come svolgere le fondamentali operazioni di input e output sul file system, e quindi come creare, modificare, cancellare i vari tipi di file e directory, e come accedere al registro di configurazione di Windows, operazioni per le quali la .NET Framework Class Library fornisce un vasto insieme di classi, organizzate nei namespace System.IO e Microsoft.Win32.

9.1 File e directory

Il namespace System.IO fornisce due coppie fondamentali di classi per la gestione di file e directory, in particolare per i primi abbiamo a disposizione le classi File e FileInfo, mentre per le directory le corrispondenti Directory e DirectoryInfo.
La classe File contiene solo dei metodi statici, utili per la creazione, lo spostamento, la copia, l'eliminazione, l'apertura di file o per ottenere generiche informazioni su di essi, come il tempo di creazione di un file o i suoi atrributi. La tabella seguente riassume i principali metodi della classe File.

Metodo	Descrizione
Open	Apre un file esistente e restituisce un oggetto FileStream.
Create	Crea un file con nome e percorso specificati..
Delete	Elimina il file specificato.
Copy	Copia un file esistente in un'altra posizione.
Move	Sposta un file esistente in un'altra posizione.
Exists	Verifica se un determinato file esiste.
OpenRead	Apre un file esistente in lettura.
OpenWrite	Apre un file esistente in scrittura.
CreateText	Crea un file di testo con nome e percorso specificati.
OpenText	Apre un file di testo in lettura.
AppendText	Apre un file di testo per appendere del testo a quello esistente.

Tali metodi hanno diversi overload, quindi è possibile utilizzarli in diversi modi, di cui vedremo i principali con qualche esempio, rinviando per ora i metodi che restituiscono un oggetto Stream..
Le seguenti righe mostrano come creare un file, copiarlo, spostarlo, ed eliminarlo:

```
File.Create(@"c:\temp\prova.txt");
File.Copy(@"c:\temp\prova.txt","c:\temp\prova2.txt");
File.Move(@"c:\temp\prova.txt","c:\temp\prova3.txt");
File.Delete(@"c:\temp\prova.txt");
```

E' bene prestare attenzione alle possibili eccezioni che possono essere generate da tali metodi. Ad esempio quando si tenta di eliminare un file inesistente o attualmente in uso da un altro processo sarà generata un'eccezione IOException:

```
try
{
      File.Delete(@"c:\temp\prova.txt");
```

```
}
catch(IOException ioe)
{
      //gestione dell'eccezione
}
```

Prima di eliminare il file sarebbe opportuno anche verificare che esso esista, con il metodo Exists:

```
if(!File.Exists(txtOrig.Text))
      File.Delete(txtOrig.Text);
```

Analogamente alla classe File, la classe Directory fornisce i metodi per manipolare una directory ed ottenere informazioni su di essa. La tabella seguente mostra i metodi principali della classe Directory:

Metodo	Descrizione
CreateDirectory	Crea una directory con nome e percorso specificati.
Delete	Elimina lsa directory specificata.
Exists	Copia una directory esistente in un'altra posizione.
Move	Sposta una directory esistente in un'altra posizione.
GetFileSystemEntries	Restituisce i file e le directory contenuti nella directory specificata
GetFiles, GetDirectories	Restituiscono i nomi dei file o delle sottodirectory di una directory specificata.
GetDirectoryRoot	Restituisce la radice del percorso specificato.
GetCurrentDirectory	Restituisce la directory di lavoro corrente dell'applicazione.
GetLogicalDrives	Restituisce i drive logici presenti nel sistema.
GetParent	Restituisce la directory madre del percorso specificato
AppendText	Apre un file di testo per appendere del testo a quello esistente.

Sia la classe File che la classe Directory permettono di ottenere e impostare i tempi di creazione o di accesso ad un dato elemento:

```
string creazione=File.GetCreationTime(@"c:\temp\pippo.txt").ToString();
File.SetLastWriteTime(@"c:\temp\pippo.txt",new DateTime(2000,8,2);
string path=@"c:\temp\nuova cartella";
Directory.Create(path);
Directory.SetLastAccessTime(path,new DateTime(1981,11,25));
```

Le due classi FileInfo e DirectoryInfo, come detto, forniscono anche metodi che permettono di ottenere altre informazioni sui file, come gli attributi, la data di creazione, la data di lettura e di modifica. A differenza delle classi File e Directory, le classi FileInfo e DirectoryInfo possono essere istanziate.
Il metodo statico per creare una directory:

```
Directory.CreateDirectory(@"c:\temp");
```

è equivalente al metodo Create della classe DirectoryInfo:

```
DirectoryInfo di=new DirectoryInfo(txtOrigine2.Text);
di.Create();
```

Il vantaggio del primo è quello di essere più veloce, in quanto non ha bisogno di istanziare un oggetto, ma nel secondo l'oggetto creato, resta disponibile per eseguire altre operazioni sulla stessa directory.
Le classi viste inoltre permettono di ricavare diverse informazioni su file e directory.
Il seguente esempio mostra come ricavare la data di creazione, l'ultimo accesso, l'ultima scrittura, ed il numero di file e sottodirectory di una data directory:

```
DirectoryInfo di=new DirectoryInfo(path);
StringBuilder sb=new StringBuilder();
sb.Append("CreationTime: "+di.CreationTime+"\n");
sb.Append("LastAccessTime: "+di.LastAccessTime+"\n");
sb.Append("LastWriteTime: "+di.LastWriteTime+"\n");
sb.AppendFormat("{0} contiene\n{1} file e {2} subdirectory",
            di.FullName,
            di.GetFiles().Length,
            di.GetDirectories().Length);
MessageBox.Show(sb.ToString());
```

Non abbiamo parlato in questo paragrafo della scrittura e lettura vera e propria di un file, in quanto tali operazioni vengono effettuate per mezzo di oggetti Stream che vedremo nel prossimo paragrafo.

9.2 Leggere e scrivere file

La scrittura e la lettura di un file vengono effettuate per mezzo di classi che rappresentano il concetto generico di Stream, cioè di un oggetto usato per trasferire dei dati.
Ad esempio quando dobbiamo leggere dei dati da una sorgente si parla di lettura di uno Stream, viceversa scrivendo dati su uno Stream si effettua il trasferimento di dati dal nostro programma ad una data destinazione.
Abbiamo parlato genericamente di sorgente e destinazione ma in questo paragrafo ci occuperemo di oggetti FileStream, usati per leggere e scrivere su un file dei flussi dati binari, e di oggetti StreamReader e StreamWriter che invece sono delle classi di aiuto nel trattare con file di testo.
Il namespace System.IO contiene molte altre classi che possono venirci incontro in particolari ambiti, ma naturalmente non possiamo essere esaustivi e trattarle in modo completo, ma ci limiteremo ad usarle in qualche esempio.

9.2.1 File binari

Un oggetto FileStream crea uno stream attorno ad un dato file. Con l'oggetto così creato è possibile effettuare le operazioni di apertura, scrittura e lettura, e di chiusura del file.
Per istanziare un FileStream possiamo utilizzare uno dei costruttori della classe:

```
string path=@"c:\temp\pippo.exe";
FileStream fs=new FileStream(path, FileMode.Open, FileAccess.Read);
```

oppure ricorrere ai metodi statici della classe File od utilizzare un oggetto FileInfo:

```
FileStream fs2=File.OpenRead(path);
FileInfo fi=new FileInfo(path);
Fi.OpenWrite();
```

Quando si crea un FileStream è possibile dunque specificare tre informazioni, oltre al nome del file, vale a dire la modalità FileMode, il tipo di accesso al file FileAccess, e la modalità di condivisione FileShare.
Questi parametri sono rappresentati per mezzo di tre tipi enumerativi, i cui valori si spiegano da soli.

Enumerazione	Valori	Descrizione
FileMode	Append, Create, CreateNew, Open,	Specifica la modalità di apertura o di creazione di un file.

	OpenOrCreate, Truncate	
FileAccess	Read, Write,ReadWrite	Specifica le modalità di accesso in lettura, scrittura o entrambe. Per default l'accesso è ReadWrite.
FileShare	Inheritable, None, Read, ReadWrite, Write	Specifica il tipo di accesso che altri FileStream possono avere sullo stesso file, per default è Read, cioè di sola lettura.

I valori delle enumerazioni FileAccess e FileShare sono dei flag combinabili per mezzo dell'operatore or, |.
Per leggere e scrivere dati per mezzo di un FileStream, si utilizzano due metodi. Il primo permette di leggere il successivo byte:

```
int nuovoByte=fs.ReadByte();
```

Se si è raggiunta la fine del file, il metodo ReadByte restituisce il valore -1.
Per leggere blocchi di byte di lunghezza predefinita, inserendoli in un array, è invece disponibile il metodo Read, i cui parametri sono l'array destinazione, l'offset dalla posizione corrente nel file, dal quale effettuare la lettura, ed il numero di byte da leggere.

```
byte[] bytesLetti=new byte[16];
int nBytesLetti=fs.Read(bytesLetti, 0 , 16);
```

La chiamata precedente tenta di leggere 16 byte, inserendoli nell'array bytesLetti, e restituisce il numero di byte effettivamente letti.
La scrittura avviene in maniera perfettamente speculare. Il metodo WriteByte scrive un byte sullo Stream:

```
byte unByte=255;
fs.WriteByte(unByte);
```

mentre per scrivere un blocco di byte, si utilizza il metodo Write:

```
byte[] bytesDaScrivere=new byte[16];
for(int i=0;i<16;i++)
{
      bytesDaScrivere[i]=i;
}
fs.Write(bytesDaScrivere, 0, 16);
```

Una volta terminato di utilizzare un FileStream è necessario chiuderlo:

```
fs.Close();
```

La chiusura libera le risorse e permette ad altri FileStream o ad altre applicazioni di accedere al file.
Altri metodi permettono di effettuare altre operazioni su un FileStream, ad esempio conoscerne la dimensione in byte, spostarsi al suo interno ad una data posizione, bloccarne e sbloccarne l'accesso ad altri processi:

```
int dim=fs.Length; //dimensione del file
fs.Seek(100); //posiziona al byte 100;
fs.Lock(); //blocca l'accesso …
fs.Unlock(); //e lo sblocca
```

Come esempio un po' più complesso mostriamo un metodo per stampare il contenuto di un file come fanno i classici editor esadecimali.

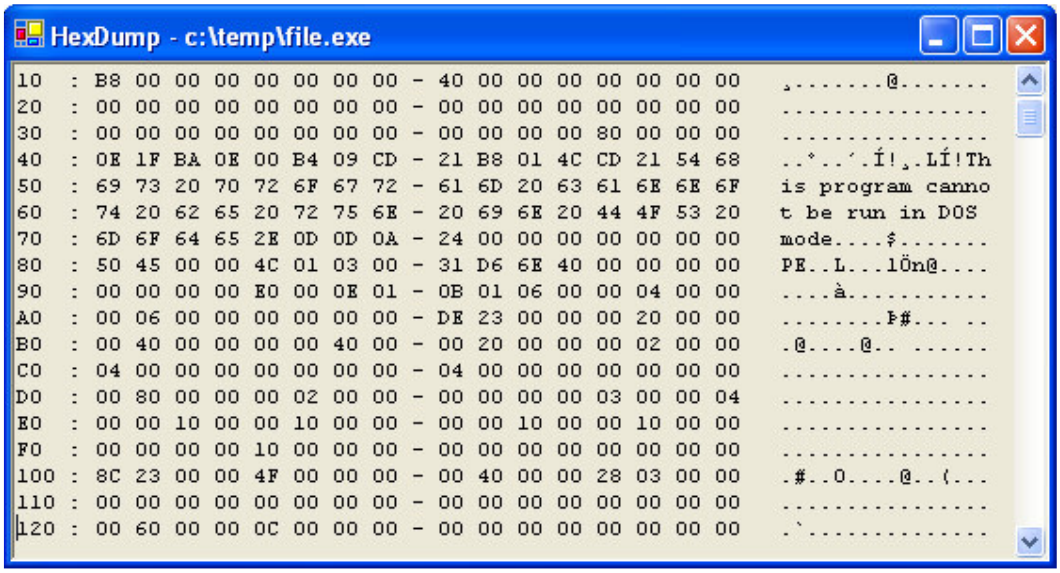


Figura 9.1 Un visualizzatore esadecimale

```
private string DumpFileStream(FileStream fs)
{
      byte[] bytesLetti = new byte[16];
      long   lPos  = 0;
      int    count;
      StringBuilder sb=new StringBuilder();

      while ((count = fs.Read(bytesLetti, 0, 16)) > 0)
      {
            sb.AppendFormat("{0,-4:X}: ",lPos);
            for (int i = 0; i < 16; i++)
            {
                  if(i < count)
                        sb.AppendFormat("{0:X2}", bytesLetti[i]);
                  else sb.AppendFormat(" ");
                  if(i == 7 && count > 7)
                        sb.Append(" - ");
                  else sb.Append(" ");
            }
            sb.Append("  ");
            for (int i = 0; i < 16; i++)
            {
                  char ch = (i < count) ? Convert.ToChar(bytesLetti[i]) : ' ';
                  sb.Append( Char.IsControl(ch) ? "." : ch.ToString() );
            }

            sb.Append(Environment.NewLine);
            lPos += 16;
      }
      return sb.ToString();
}
```

la Figura 9.1 mostra una semplice applicazione che costituita da una form con una sola TextBox. Per impostare il contenuto della TextBox viene inizialmente richiamato il metodo seguente che crea un FileStream e lo passa al metodo che abbiamo visto prima.

```
protected void Dump(string strFile)
{
      FileStream fs=null;
      try
      {
            fs = File.OpenRead(strFile);
            this.Text+=" - "+strFile;
            txtDump.Text=DumpFileStream(fs);
      }
      catch (Exception ex)
```

```
        {
                txtDump.Text=ex.Message;
        }
        finally
        {
                if(fs!=null)
                        fs.Close();
        }
}
```

9.2.2 Scrivere e leggere tipi primitivi

La classe FileStream permette di leggere e scrivere byte nudi e crudi. Se invece si vuol lavorare direttamente con i tipi di dati primitivi, la Base Class Library fornisce la classe BinaryReader per la lettura e BinaryWriter per la scrittura.
Per costruire un oggetto BinaryWriter, bisogna passare al suo costruttore un parametro FileStream, con il quale ad esempio si è creato un file:

```
FileStream fs=new FileStream("file.dat",FileMode.Create);
BinaryWriter bw=new BinaryWriter(fs);
```

Il metodo Write dispone di diversi overload che permettono di scrivere un valore di un tipo primitivo qualsiasi ed array di byte o di char:

```
bool bVal=true;
int nVal=-100;
uint unVal=uint.MaxValue;
byte byteVal=8;
sbyte sbyteVal=-8;
char chVal='a';
byte[] bytes=new byte[4]{20,30,40,50};
char[] chars=new char[5]{'h','e','l','l','o'};
string str="world";
double dVal=0.123d;
float fVal=3.14f;
long lVal=long.MaxValue;
ulong ulVal=ulong.MinValue;
decimal decVal=1000000000M;
short sVal=-32768;
ushort usVal=32767;

bw.Write(bVal);
bw.Write(nVal);
bw.Write(unVal);
bw.Write(byteVal);
bw.Write(sbyteVal);
bw.Write(bytes);
bw.Write(bytes,1,2);
bw.Write(chVal);
bw.Write(chars);
bw.Write(chars,0,3);
bw.Write(str);
bw.Write(dVal);
bw.Write(fVal);
bw.Write(lVal);
bw.Write(ulVal);
bw.Write(decVal);
bw.Write(sVal);
bw.Write(usVal);
```

Alla fine della scrittura su file è necessario chiudere l'oggetto BinaryWriter, il metodo Close chiude anche il FileStream sottostante:

```
bw.Close();
```

Supponiamo ora di voler leggere i dati salvati nel file di prima. Dopo aver aperto il solito FileStream, creiamo un BinaryReader:

```
FileStream fs=new FileStream(@"C:\temp\file.dat",FileMode.Open);
BinaryReader br=new BinaryReader(fs);
```

i dati devono quindi essere ricavati nello stesso ordine in cui sono stati scritti:

```
bool bVal=br.ReadBoolean();
int nVal=br.ReadInt32();
uint unVal=br.ReadUInt32();
byte byteVal=br.ReadByte();
sbyte sbyteVal=br.ReadSByte();
byte[] bytes=br.ReadBytes(4);
bytes=br.ReadBytes(2);
char chVal=br.ReadChar();
char[] chars=br.ReadChars(5);
chars=br.ReadChars(3);
string str=br.ReadString();
double dVal=br.ReadDouble();
float fVal=br.ReadSingle();
long lVal=br.ReadInt64();
ulong ulVal=br.ReadUInt64();
decimal decVal=br.ReadDecimal();
short sVal=br.ReadInt16();
ushort usVal=br.ReadUInt16();
```

e quindi chiudere l'oggetto BinaryReader:

```
br.Close();
```

9.2.3 File di testo

Nonostante sia possibile lavorare con file di testo continuando ad utilizzare la classe FileStream, o le classi BinaryReader e BinaryWriter, il framework .NET mette a nostra disposizione altre due classi che ci permettono di scrivere e leggere direttamente delle linee di testo.
Queste due classi sono StreamReader per la lettura e StreamWriter per la scrittura di testo.
Creare uno StreamReader è fattibile sia utilizzando uno dei costruttori della classe:

```
StreamReader sr=new StreamReader(@"c:\temp\pippo.txt");
```

Oppure mediante alcuni dei metodi delle classi File o FileInfo, ad esempio:

```
sr=File.OpenText(@"c:\temp\pippo.txt");
FileInfo fi=new FileInfo(@"c:\temp\pippo.txt");
sr=fi.OpenText();
```

Il modo più veloce di leggere un file di testo è quello di utilizzare il metodo ReadToEnd che restituisce una stringa con l'intero contenuto del file:

```
string strContent=sr.ReadToEnd();
```

Se si vuol invece leggere il file riga per riga, ad esempio per effettuare un parsing di ogni linea di testo è possibile utilizzare il metodo ReadLine:

```
string linea=sr.ReadLine();
```

se si è raggiunta la fine del file, il metodo ReadLine restituisce null.
E' possibile anche leggere il prossimo singolo carattere per mezzo del metodo Read o del metodo Peek. Quest'ultimo legge il carattere ma senza avanzare nel flusso del file.

```
char c;
int carattere=sr.Read();
if(carattere!=-1)
      c=(char)carattere;
```

il metodo restituisce un intero, in modo che se si raggiunge la fine del file, il valore restituito sarà -1. Se dunque si vuole ottenere un char bisogna effettuare un cast esplicito.
Il metodo Read ha un secondo overload che permette di leggere un dato numero di caratteri ed inserirli in un array:

```
int toRead=16;
char[] charsLetti=new char[toRead];
int nLetti=sr.Read(charsLetti,0, toRead);
```

Così come abbiamo visto fin'ora, anche uno StreamReader deve essere chiuso al termine del suo utilizzo.

```
sr.Close();
```

o ancora utilizzando la parola chiave using:

```
using (StreamReader sr = new StreamReader("file.txt"))
{
      String line;
      while ((line = sr.ReadLine()) != null)
            Console.WriteLine(line);
}
```

Per scrivere delle linee di testo si utilizza in maniera complementare la classe StreamWriter. Anche la creazione di uno StreamWriter può essere realizzata in vari modi:

```
StreamWriter sw=new StreamWriter("file.txt");
sw=File.CreateText("file.txt");
```

I metodi di StreamWriter sono analoghi a quelli visti con StreamReader, ad esempio per scrivere una linea di testo si utilizza il metodo Write con un parametro stringa. Il metodo WriteLine invece aggiungerà automaticamente anche il carattere di fine riga:

```
string str="linea di testo";
sw.Write(str);
sw.WriteLine(str);
```

I vari overload del metodo Write e del metodo WriteLine permettono di scrivere diversi tipi di dato, ad esempio se si vuole scrivere carattere per carattere o un array di caratteri, basta utilizzare il corrispondente overload::

```
char ch='a';
sw.Write(ch);
char[] charArray = {'a','b','c','d','e','f','g','h','i','j','k','l','m'};
sw.Write(charArray);
sw.Write(charArray,3,5);//scrive 5 caratteri a partire dall'indice 3
```

9.3 Accedere al registro

Il registro di configurazione di Windows è il punto di raccolta principale per tutte le informazioni riguardanti il sistema operativo, i dettagli per ogni singolo utente, le applicazioni installate, ed i dispositivi fisici. Il framework .NET ha praticamente azzerato l'importanza di conservare le impostazioni di un'applicazione nel registro, nel senso che gli assembly sono praticamente installabili e funzionanti con un semplice copia e incolla dei file necessari, ma nonostante ciò esso fornisce delle classi che permettono l'accesso completo al registro di sistema.

9.3.1 Le classi Registry e RegistryKey

L'accesso al registro, in lettura e scrittura è permesso da una coppia di classi contenute nel namespace Microsoft.Win32, esattamente la classe Registry e la classe RegistryKey.
Il registro è una struttura gerarchica formata da chiavi, che si diramano da un livello principale costituito da cosiddetti registry hive.

Se provate a lanciare l'utility regedit per la modifica del registro, vedrete infatti che sono presenti una sorta di chiavi radice, con nomi del tipo HKEY_CLASSES_ROOT, HKEY_CURRENT_USER HKEY_LOCAL_MACHINE, e così via.
Tali valori sono contenuti nell'enumerazione RegistryHive, di cui possiamo visualizzare i valori con un metodo come il seguente:

```
public void EnumerateHives()
{
      Array hives=System.Enum.GetValues(typeof(RegistryHive));
      foreach(object hive in hives)
            Console.WriteLine(hive.ToString());
}
```

La classe Registry permette di accedere alle chiavi principali del registro per mezzo di proprietà statiche che restituiscono un'istanza della classe RegistryKey, i cui nomi sono quelli restituiti dal metodo precedente.

```
RegistryKey hKeyLM=Registry.LocalMachine;
RegistryKey hKeyCR=Registry.ClassesRoot;
RegistryKey hKeyCU=Registry.CurrentUser
RegistryKey hKeyCC=Registry.CurrentConfig;
RegistryKey hKeyU=Registry.Users;
RegistryKey hKeyPD=Registry.PerformanceData;
RegistryKey hKeyDD=Registry.DynData;
```

A partire da queste chiavi possiamo esplorare l'intero registro, localizzando una qualunque sottochiave, ad esempio se volessimo leggere il contenuto della chiave Software contenuta in HKEY_LOCAL_MACHINE, possiamo utilizzare il metodo OpenSubKey della classe RegistryKey.

```
RegistryKey hKeyLM=Registry.LocalMachine;
RegistryKey hkSW=hKeyLM.OpenSubKey("Software");
```

Se tentassimo di aprire una sottochiave non esistente, il valore restituito sarebbe naturalmente null.
Per creare invece una chiave non esistente bisogna utilizzare il metodo CreateSubKey.

```
RegistryKey hkNuova=hkSW.CreateSubKey("NomeChiave");
```

Notate che se già esiste una chiave con il nome specificato, verrà restituita un'istanza di RegistryKEy corrispondente ad essa.
E' possibile anche eliminare una chiave dal registro con la chiamata al metodo DeleteSubKey, facendo attenzione naturalmente alla chiave che tentate di eliminare, per non rischiare di provocare danni irreparabili al sistema operativo. Magari è meglio provare ad eliminare la chiave che abbiamo creato prima:

```
hkSW.DeleteSubKey("NomeChiave");
```

Una volta localizzata e aperta la chiave di interesse, è possibile leggere, modificare, creare o cancellare i valori in essa contenuti.
Per leggere un valore il metodo RegistryKey.GetValue necessita di specificare una stringa rappresentante il valore da leggere. I nomi dei valori sono ricavabili mediante il metodo GetValueNames che restituisce un array di stringhe:

```
RegistryKey hkLM=Registry.LocalMachine;
RegistryKey hkCurrVer=hkLM.OpenSubKey(@"SOFTWARE\Microsoft\Windows\CurrentVersion");

Console.WriteLine("La chiave {0} contiene i valori",hkCurrVer.Name);
string[] values=hkCurrVer.GetValueNames();
foreach(string val in values)
{
      Console.WriteLine(val);
}
```

Ad esempio il sistema dell'autore , nella chiave SOFTWARE\Microsoft\Windows\CurrentVersion dell'hive HKEY_LOCAL_MACHINE, contiene un valore di tipo REG_SZ, cioè stringa, di nome ProgramFilesDir. Per leggerne il contenuto possiamo scrivere:

```
String strVal=hkCurrVer.GetValue("ProgramFilesDir");
```

che ad esempio potrebbe restituire il valore 'C:\Programmi'.

Appendice A

Opzioni del compilatore csc

Il compilatore C# può essere invocato dal prompt dei comandi, digitando il nome del suo eseguibile csc.exe.
In questa appendice vengono illustrate le opzioni che è possibile utilizzare con il compilatore csc, suddivise per categoria. Per ottenere un rapido riferimento ed un elenco di tali opzioni è possibile invocare il compilatore csc con l'opzione `/help` o in forma abbreviata `/?`.
Alcune opzioni hanno una forma abbreviata, in tal caso esse verranno visualizzate in parte fra parentesi quadre, ad esempio l'opzione indicata come `/t[arget]` indica che la forma completa è `/target`, mentre `/t` ne è la forma abbreviata.
Le opzioni che consentono di scegliere fra più valori, sono visualizzate con un | a separare i diversi valori possibili. Ad esempio `[+ | -]` indica che è possibile specificare un valore fra + e -.

Opzioni di output

Le opzioni di output permettono di specificare il tipo di output da produrre ed il nome degli eventuali file generati.

Opzione	Descrizione
`/t[arget]:exe`	Genera un applicazione console, il nome dell'eseguibile generato per default sarà quello del file contenente il metodo Main.
`/t[arget]:winexe`	Genera un'applicazione windows, cioè con interfaccia grafica. Il nome dell'eseguibile generato per default sarà quello del file contenente il metodo Main.
`/t[arget]:library`	Genera una libreria di classi DLL con estensione di default .dll, e con nome del file uguale a quello del primo file sorgente specificato.
`/t[arget]:module`	Genera un modulo managed con estensione di default .netmodule. Esso non conterrà dunque un manifest, non essendo un assembly. Vedi l'opzione /addmodule.
`/out:<nomefile>`	Specifica il nome del file da generare.
`/doc:<file.xml>`	Specifica il nome del file XML contenente la documentazione estratta dai commenti del codice. Vedi l'appendice B.

Opzioni per gli assembly .NET

Le opzioni di questa categoria permettono di specificare informazioni relative agli assembly da importare, o dei moduli da aggiungere ad assembly esistenti.

Opzione	Descrizione
`/addmodule:<file1>[;<file2>`	Aggiunge al file di output della compilazione i moduli (file con estensione .netmodule) specificati. I moduli aggiunti devono trovarsi comunque nella stessa directory dell'eseguibile a runtime.
`/lib:<dir1>[,<dir2>]`	Specifica una o più directory in cui ricercare eventuali assembly referenziati tramite l'opzione /reference.

/nostdlib	L'opzione viene utilizzata per non importare le librerie standard contenute nel file mscorlib.dll.
/r[reference]:<file>	Specifica uno o più file assembly contenenti tipi referenziati nel codice.

Opzioni di debugging e error checking

Le opzioni di questa categoria permettono di rendere più semplice il debug delle applicazioni e di specificare come ottenere delle informazioni sugli eventuali errori del codice.

Opzione	Descrizione
/bugreport:<file1>	Genera un file testuale contenente un report sul codice e sulla compilazione effettuata, ad esempio un elenco dei file compilati, la versione del CLR e del sistema operativo, ecc.
/debug[:full | pdbonly] /debug[+ | -]	Genera informazioni di debug nello stesso eseguibile del programma
/checked	Specifica di considerare o meno come eccezioni le operazioni che causano overflow aritmetici.
/fullpaths	Specifica il path assoluto del file output di compilazione.
/nowarn	Indica di non generare messaggi di warning.
/warn	Imposta il livello di warning.
/warnaserror	Considera i warning come errore.

Opzioni per i file di risorse

Le opzioni di questa categoria permettono di inserire nell'eseguibile generato dal compilatore delle risorse, ad esempio immagini o icone.

Opzione	Descrizione
/linkresource	Crea un collegamento ad una risorsa gestita.
/resource	Include una risorsa gestita nell'eseguibile generato dal compilatore.
/win32icon	Inserisce un file di icona (.ico) nell'eseguibile.
/win32res	Inserisce una risorsa Win32 nell'eseguibile.

Opzioni di preprocessore

L'unica opzione è la /define per definire dei simbolo di preprocessore da utilizzare nell'applicazione.

Opzione	Descrizione
/define	Definisce i simboli di preprocessore.

Opzioni varie

Opzione	Descrizione
/? /help	Mostra un help sulle opzioni del compilatore csc.
/@file.rsp	Specifica un file contenente le azioni e le opzioni da usare per la compilazione ed i file da compilare.
/main	Specifica la classe che contiene il metodo Main.
/incremental [+ | -]	Abilita o meno la compilazione incrementale, cioè compila solo le parti che hanno subito modifiche dall'ultima compilazione.
/noconfig	Lancia la compilazione senza utilizzare le opzioni contenute nel file

	csc.rsp contenuto nella stessa directory di csc.exe
`/nologo`	Non mostra le informazioni sul compilatore csc.exe utilizzato.
`/recurse:[\dir]file`	Effettua la compilazione cercando i file da compilare in tutte le sottodirectory delle directory specificate.
`/unsafe`	Compila il codice C# che utilizza la keyword unsafe.
`/utf8outpu`	Mostra le informazioni di output utilizzando la codifica UTF8.
`/baseaddress:indirizzo`	Specifica l'indirizzo di memoria al quale caricare una DLL.
`/codepage:id`	Specifica l'id di codepage da usare per i file sorgente. Se i sorgenti sono in Unicode o UTF8 l'opzione non è necessaria.