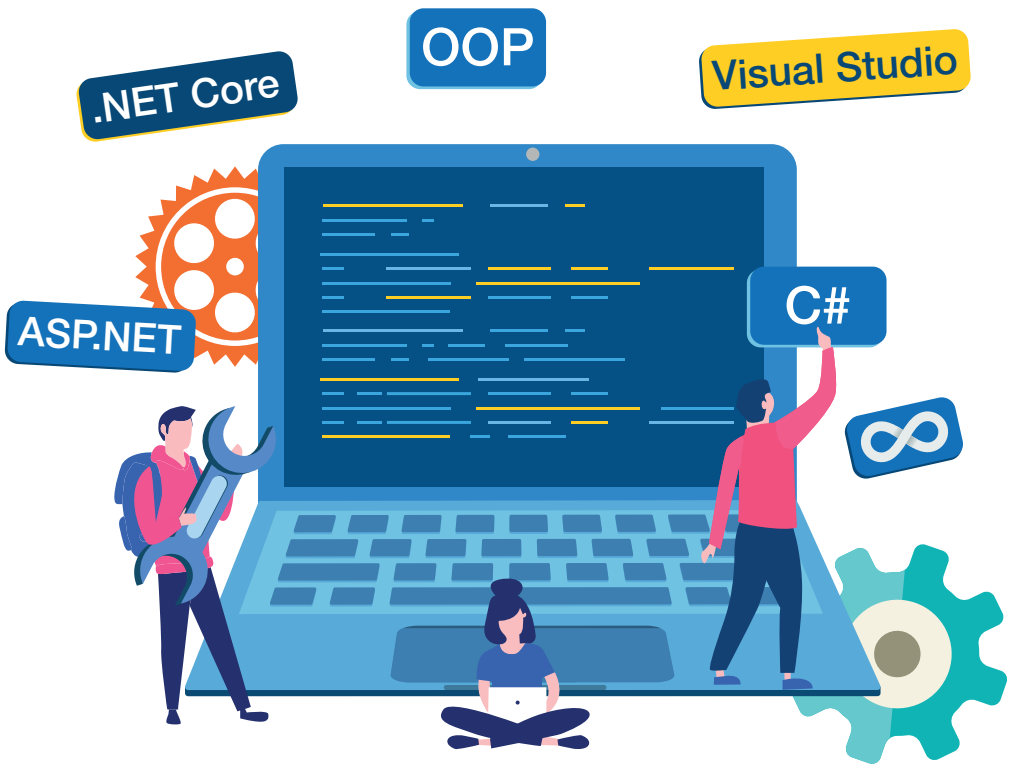
- Antonio Pelleriti -

PROGRAMMARECON

C# 8

Guida completa



Sviluppo per Windows, Linux, macOS ed esercizi pratici >>

La programmazione a oggetti, eventi, eccezioni, generics, LINQ >>

Ambiente di sviluppo Visual Studio 2019/.NET Core >>

La sintassi e i costrutti del linguaggio >>

*pro DigitalLifeStyle

*pro
DigitalLifeStyle

Programmare con C# 8

Guida completa

Antonio Pelleriti

EDIZIONI
LSWR

Programmare con C# 8 | Guida completa.

Autore: Antonio Pelleriti

Collana: *pro DigitalLifeStyle

Publisher: Marco Aleotti
Progetto grafico: Roberta Venturieri

© 2019 Edizioni Lswr* – Tutti i diritti riservati

ISBN: 978-88-6895-769-8
eISBN: 978-88-6895-770-4

I diritti di traduzione, di memorizzazione elettronica, di riproduzione e adattamento totale o parziale con qualsiasi mezzo (compresi i microfilm e le copie fotostatiche), sono riservati per tutti i Paesi. Le fotocopie per uso personale del lettore possono essere effettuate nei limiti del 15% di ciascun volume dietro pagamento alla SIAE del compenso previsto dall'art. 68, commi 4 e 5, della legge 22 aprile 1941 n. 633.

Le fotocopie effettuate per finalità di carattere professionale, economico o commerciale o comunque per uso diverso da quello personale possono essere effettuate a seguito di specifica autorizzazione rilasciata da CLEARedi, Centro Licenze e Autorizzazioni per le Riproduzioni Editoriali, Corso di Porta Romana 108, 20122 Milano, e-mail autorizzazioni@clearedi.org e sito web www.clearedi.org.

La presente pubblicazione contiene le opinioni dell'autore e ha lo scopo di fornire informazioni precise e accurate. L'elaborazione dei testi, anche se curata con scrupolosa attenzione, non può comportare specifiche responsabilità in capo all'autore e/o all'editore per eventuali errori o inesattezze.

L'Editore ha compiuto ogni sforzo per ottenere e citare le fonti esatte delle illustrazioni. Qualora in qualche caso non fosse riuscito a reperire gli aventi diritto è a disposizione per rimediare a eventuali involontarie omissioni o errori nei riferimenti citati.

Tutti i marchi registrati citati appartengono ai legittimi proprietari.

EDIZIONI LSWR
Via G. Spadolini, 7
20141 Milano (MI)
Tel. 02 881841
www.edizionilswr.it

Printed in Italy

Finito di stampare nel mese di settembre 2019 presso "Rotolito" S.p.A., Seggiano di Pioltello (MI)

(*) Edizioni Lswr è un marchio di La Tribuna Srl. La Tribuna Srl fa parte di LSWR GROUP.

Sommario

Introduzione

C# è un linguaggio di programmazione sviluppato da Microsoft, all'interno della piattaforma .NET, che da tempo è fra i protagonisti principali sul palcoscenico dello sviluppo software, avendo fatto il suo debutto nel 2000 e continuando a mantenersi sempre in linea con le tendenze, versione dopo versione, con l'aggiunta di caratteristiche e funzionalità sempre nuove, tanto da essere, oggi più che mai, costantemente ai primi posti nelle graduatorie dei linguaggi più scelti e utilizzati dalla comunità mondiale di sviluppatori, professionali, hobbisti, o da chi si affaccia per la prima volta a tale mondo.
In tale sua evoluzione, C# non è mai divenuto complesso o pesante da digerire, anzi ha mantenuto la semplicità senza trascurare la potenza che permette di affrontare e risolvere problemi legati ad ambiti di sviluppo eterogenei, sia dal punto di vista della piattaforma di esecuzione sia da quello prettamente pratico legato all'ambito applicativo.
C# e .NET rappresentano oggi le principali scelte per chi vuole creare software che girino sul desktop di un personal computer, applicazioni lato server, o anche applicazioni cloud e web eseguite all'interno di un browser internet, o ancora come app installate su uno smartphone o su un tablet, spaziando in qualunque caso lungo **scenari applicativi** anch'essi estremamente eterogenei: dal mondo dell'industria a quello dei software di produttività e gestione aziendale, passando per i videogame e per i sistemi di commercio elettronico.
L'introduzione di .NET ha costituito senza dubbio una delle principali rivoluzioni nel mondo dello sviluppo software, soprattutto per quanto riguarda l'ambiente che comprende i sistemi operativi di Microsoft, ossia le varie versioni di Windows.
Ma C# e .NET non sono assolutamente legati e limitati al mondo, un tempo chiuso, di **Windows**: l'iniziativa **.NET Core** rappresenta la visione multipiattaforma (supportando anche **MacOS** e **Linux**) e open source di Microsoft, che ora include anche lo sviluppo server, cloud, web, passando naturalmente per il mondo delle applicazioni per il desktop, fino ad arrivare al lato mobile con **Xamarin**, l'ambiente di esecuzione dedicato alle applicazioni per dispositivi **Android** e **iOS**.

Detto ciò, C# può quindi essere considerato a tutti gli effetti un linguaggio di programmazione Cross-Platform.
Se vi state chiedendo se vale la pena sviluppare per .NET, riflettete sulle seguenti statistiche: più del 90% dei personal computer nel mondo ha una versione di .NET installata, quindi si parla di oltre un miliardo di potenziali utenti. Su ognuna di queste macchine è possibile eseguire un'applicazione scritta in C#.
Il linguaggio C# è, fra quelli che è possibile utilizzare per lo sviluppo .NET, quello che riflette maggiormente le caratteristiche peculiari della piattaforma, in quanto nato con essa e per essa, e ne è quindi riconosciuto come il linguaggio principe. A oggi è giunto alla versione denominata **C# 8.0**, a cui questo libro è dedicato.
Non possono naturalmente mancare gli approfondimenti minimi su .NET, che è la piattaforma, oggi disponibile in varie implementazioni, su diversi sistemi operativi e architetture, all'interno della quale vengono sviluppate ed eseguite le applicazioni scritte in C#.
Il Framework .NET è la piattaforma originale, sviluppata fino a oggi praticamente di pari passo con C#, e la sua versione più recente, al momento della stampa, è la **4.8**.
L'implementazione più recente è **.NET Core**, che costituisce una delle più importanti innovazioni apportate all'intera piattaforma fin dal suo debutto. Al momento della scrittura del libro l'ultima versione rilasciata è la **3.0**.
I prossimi anni vedranno una piattaforma unica, **.NET 5**, per tutti i sistemi e gli ambiti di sviluppo, e C# continuerà ad essere il linguaggio principale di sviluppo.
Il libro che state iniziando a leggere esporrà le caratteristiche del linguaggio C#, aggiornate all'ultima versione disponibile al momento della sua stesura, che è come detto la 8.0, approfondirà le implementazioni .NET Core e .NET Framework, utilizzando principalmente come ambiente di sviluppo **Visual Studio 2019**, senza tralasciare strumenti disponibili anche per altri sistemi operativi, come **Visual Studio Code**. Anzi, se ne avete la possibilità, vi consiglio di provare a utilizzare quest'ultimo sui vari ambienti in cui è disponibile, e cioè Linux, MacOS e naturalmente lo stesso Windows.

A chi si rivolge il libro

Questo libro intende rivolgersi al lettore che si avvicina per la prima volta al mondo della programmazione, ma anche a chi invece possegga già qualche esperienza in tale ambito, magari con linguaggi differenti da C# e su piattaforme diverse da .NET.
Alcuni concetti infatti sono comuni a tutto il mondo della programmazione orientata agli oggetti e di conseguenza i capitoli iniziali che espongono tale paradigma di sviluppo possono essere letti in maniera rapida per arrivare al cuore della programmazione .NET in C#.

Lo scopo del libro è comunque quello di affrontare con la maggior precisione e il massimo approfondimento possibile i concetti trattati, in modo tale da diventare un riferimento completo del linguaggio C# anche per il programmatore già esperto che voglia avere a portata di mano una guida rapida per chiarire dubbi o per trovare risposte a domande e questioni più avanzate.
Inoltre, per chi, come il sottoscritto, vive e lavora nel mondo della programmazione da decenni, ho cercato di trovare e fornire spunti e curiosità legate a C# e .NET sottolineando l'evoluzione del linguaggio lungo le sue varie versioni e indicando in quale di esse ogni nuova funzionalità o caratteristica è stata introdotta.

Struttura del libro

Il libro è strutturato in maniera da permettere, anche a chi non ha mai programmato, di iniziare tale attività in maniera proficua, partendo quindi dalle basi del linguaggio **C#**, fino ad arrivare ai concetti più complessi, e coprendo ogni novità introdotta dalle varie versioni fino ad arrivare a **C# 8.0**, permettendo di padroneggiare così ogni argomento che riguardi la programmazione .NET, anche quelli non esplicitamente trattati in questo testo.
Ogni nuova funzionalità introdotta da C# 8 è evidenziata e riportata anche nell'indice analitico, in maniera che anche coloro che fossero già in possesso delle edizioni passate del libro possano avere un rapido e completo riferimento di tutte le novità del linguaggio.
La prima parte del libro, costituita dai primi sei capitoli, introdurrà la piattaforma .NET e le caratteristiche del linguaggio C# puro, iniziando quindi dalle parole chiave, dalla sintassi con cui si scrivono i programmi, introducendo i tipi fondamentali .NET, le espressioni e gli operatori, i costrutti per controllare il flusso di esecuzione dei programmi. Si passa poi alla programmazione orientata agli oggetti e a come essa viene supportata da C#. Nei capitoli finali si affronteranno argomenti più avanzati, ma anche più pratici.
Il **Capitolo 1** esegue una prima panoramica di .NET e della sua architettura, e delle sue implementazioni, in particolare di .NET Core, e del funzionamento dell'ambiente di esecuzione dei programmi.
Nel **Capitolo 2** viene fatta una panoramica degli strumenti e degli ambienti di sviluppo necessari a programmare in C#, quindi di .NET Core SDK, di Visual Studio 2019 e di Visual Studio Code.
Il **Capitolo 3** introduce i concetti e la sintassi di base del linguaggio C# e gli elementi fondamentali che costituiscono un programma.

Il **Capitolo 4** espone il sistema di tipi di .NET e le varie categorie di tali tipi creabili e utilizzabili in C#.
Nel **Capitolo 5** si vedrà come scrivere espressioni più o meno complesse all'interno di un programma, utilizzando anche gli operatori messi a disposizione dal linguaggio.
Il **Capitolo 6** invece mostra come controllare l'esecuzione di un programma, utilizzando gli appositi costrutti e istruzioni di controllo del flusso.
A partire dal **Capitolo 7** si entra nel mondo della programmazione ad oggetti in C#, e quindi si introducono concetti che permettono l'implementazione di classi personalizzate e strutture e dei vari membri che ne possono costituire il corpo.
Il **Capitolo 8** è la logica continuazione del precedente ed approfondisce altri concetti della programmazione orientata agli oggetti, in particolare quelli di ereditarietà e polimorfismo, e presenta quello di interfaccia, il tutto allo scopo di realizzare complesse gerarchie di classi.
Si passa poi a concetti sempre più avanzati e nel **Capitolo 9** viene introdotta la gestione delle cosiddette eccezioni, cioè delle situazioni di errore che si possono verificare durante l'esecuzione dei programmi.
Il **Capitolo 10** tratta le collezioni di oggetti e le tuple, oltre che il meccanismo dei cosiddetti tipi generici, che permette di implementare tipi parametrici.
Il **Capitolo 11** tratta un argomento fondamentale di C#, come quello della programmazione a eventi e dei metodi di gestione degli stessi, e argomenti strettamente legati a questi, come i delegate, i metodi anonimi e le espressioni lambda.
Quindi il **Capitolo 12** passa ad argomenti come LINQ, che permette l'interrogazione di varie forme di dati mediante una nuova sintassi e nuovi metodi e tipi, fondamentali nel lavoro quotidiano di uno sviluppatore C#.
Il **Capitolo 13** pone l'accento sulle prestazioni e affronta argomenti come il multithreading, la programmazione parallela e quella asincrona, per sfruttare i moderni processori dotati di più core.
Fra i formati di dati più utilizzati nelle applicazioni vi è senz'altro l'XML. Il **Capitolo 14** esplora le funzioni utilizzabili da C# per manipolare tale formato, partendo dal classico XML DOM, passando per XPath fino a LINQ to XML.
Il **Capitolo 15** scende in profondità nei meandri interni della composizione dei tipi. Infatti il meccanismo di Reflection permette di analizzare ogni aspetto di un oggetto, durante l'esecuzione di un programma. Nello stesso capitolo si vedranno anche gli attributi ed il loro utilizzo.
Il **Capitolo 16** è uno dei più lunghi, in quanto affronta un argomento importante in ambito pratico come l'accesso ai dati, esplorando l'input/output su file e le tecnologie ADO.NET e Entity Framework per l'accesso ai database relazionali.

Nel **Capitolo 17** viene affrontato uno degli argomenti chiave delle ultime versioni di C#, cioè la .NET Compiler Platform, o Roslyn, mostrando come utilizzare i servizi messi a disposizione del compilatore nelle applicazioni, o per scrivere estensioni di Visual Studio.
Nell'ultimo, il **Capitolo 18**, tramite lo sviluppo di una semplice applicazione pratica che copre vari ambiti, vengono messi in pratica i concetti appresi, e viene mostrata la versatilità di C# e .NET.
L'**Appendice A** è dedicata all'utilizzo delle classi necessarie per lavorare con stringhe e testi, e alle espressioni regolari, in quanto sono un argomento che trova parecchia utilità nella pratica di tutti i giorni.
L'**Appendice B** presenta alcuni cenni sulla programmazione con i puntatori, quindi in cosiddetto contesto *unsafe*, e per l'utilizzo da codice gestito di funzioni native, per mezzo dei servizi P/Invoke.
I capitoli si concludono con una sezione "Metti in pratica", che comprende domande di riepilogo a scelta multipla per valutare la propria comprensione e l'apprendimento dei concetti, e alcune esercitazioni per applicarli. L'**Appendice C** contiene le risposte alle domande.
In tal modo si sarà compiuto un completo viaggio all'interno di tutte le caratteristiche e potenzialità offerte dal linguaggio C# e dalla piattaforma .NET, senza naturalmente la pretesa di essere totalmente esaustivi, dati i limiti imposti dalla lunghezza del testo.

Le novità di C# 8

Il libro è aggiornato a tutte le nuove funzionalità introdotte con la versione 8 (o 8.0) del linguaggio C#. Per chi volesse un rapido riferimento di tali novità eccone un elenco:

- tipi riferimento nullable;
- membri predefiniti nelle interfacce;
- range e indici;
- miglioramenti nel pattern matching;
- flussi asincroni;
- espressioni switch;
- assegnazione null coalescing;
- funzioni locali statiche;
- dichiarazioni using;
- membri readonly di struct;
- sintassi alternativa interpolazione stringhe.

L'indice analitico alla fine del libro, alla voce **C# 8**, vi permetterà di trovare facilmente le pagine in cui ognuno di questi argomenti viene trattato.
Vale la pena notare, soprattutto per chi possiede l'edizione precedente "Programmare con C# 7", che il testo comprende anche le novità del linguaggio apportate nelle versioni intermedie 7.1, 7.2 e 7.3.

Esempi e contenuti extra

Ogni capitolo del libro contiene esempi pratici che potete scrivere e compilare autonomamente, utilizzando il compilatore a riga di comando fornito da .NET Core SDK 3.0, oppure all'interno di ambienti di sviluppo come **Visual Studio Code** e **Visual Studio**, in particolare la versione **2019** di quest'ultimo.
Le indicazioni e i link da cui scaricare gli esempi completi sono inoltre disponibili sul mio sito internet, alla pagina *http://www.antoniopelleriti.it/page/libro-csharp*, dotati di file di soluzione *.sln*, che potete quindi aprire direttamente in Visual Studio.
Sulla stessa pagina, troverete anche eventuali capitoli bonus o contenuti extra che, per limiti di spazio o per completare il testo con argomenti di cui si è avuta notizia magari dopo la stampa (per esempio dedicati alle future versioni C# 8.x), non fanno parte della versione finale del libro.

Errata corrige

Sebbene il libro sia stato letto e riletto, qualche errore può sempre sfuggire! Quindi eventuali correzioni potete trovarle sempre sul sito dedicato *http://www.antoniopelleriti.it/page/libro-csharp*.
Sullo stesso sito e sulla pagina Facebook ad esso dedicata, *https://www.facebook.com/programmare.con.csharp*, potete effettuare le vostre segnalazioni di errori e imprecisioni, e proporre magari suggerimenti per le prossime edizioni del libro.

L'autore

Antonio Pelleriti è ingegnere informatico, e si occupa da diversi anni di sviluppo software, su varie piattaforme.
In particolare il .NET Framework e le tecnologie ad esso correlate sono i suoi principali interessi fin dal rilascio della prima beta della piattaforma Microsoft, quindi da oltre 15 anni. In tale ambito un importante riconoscimento è stata la nomina a Microsoft MVP per .NET e in seguito per Visual Studio and Development Technologies.
Autore di numerose pubblicazioni per riviste di programmazione e di guide tascabili dedicate al mondo .NET, per Edizioni FAG ha già pubblicato nel 2011 il libro "Silverlight

4, guida alla programmazione", e per LSWR, la prima edizione di "Programmare con C# 5, guida completa" (2014), la seconda "Programmare con C# 6, guida completa" (2016) e la terza "Programmare con C# 7, guida completa" (2017).
Il suo sito personale su cui pubblica articoli e pillole di programmazione legate sempre a .NET e C# è *www.antoniopelleriti.it*.
Dopo aver girato in lungo ed in largo la penisola, partendo da **Braidi**, suo amato e ridente paesello abbarbicato sui monti Nebrodi, e lavorando per primarie aziende nazionali e multinazionali, ritorna in Sicilia, naturalmente continuando a seguire ogni aspetto di sviluppo legato alla piattaforma .NET, e lavorando a progetti software di ogni dimensione e tipo.
Tifoso, da sempre e per sempre, del **Milan**, con il privilegio di aver goduto degli anni in cui sui campi giocava e danzava il cigno di Utrecht, Marco Van Basten ("Il lutto in me per il suo precoce ritiro non si estingue ancora e mai si estinguerà." cit. C. Bene). Non disdegna una birra o un calice di vino in buona compagnia, e un buon sigaro, soprattutto quando finisce di scrivere un libro.
Marito di Caterina e papà di Matilda, a cui cerca di dare tutto e non togliere nulla.

Ringraziamenti

Scrivere un libro sul linguaggio che si studia e utilizza per lavoro fin dalla sua apparizione nel mondo dello sviluppo software può essere considerato un sogno che si realizza. Il successo delle prime edizioni è dunque una soddisfazione enorme, soprattutto per i feedback ricevuti dai lettori con le loro recensioni e tramite i loro messaggi ed email. In particolare un grazie enorme a tutti coloro che mi hanno fatto scovare errori e imprecisioni e che mi hanno donato i loro consigli per migliorare il testo in ogni suo aspetto. Il fatto che mi ritrovi anno dopo anno a scrivere le varie edizioni aggiornate, seguendo l'evoluzione di C#, è merito di tutti loro.
Grazie quindi a **te che stai leggendo**, chiunque tu sia, alla fine se non ci fosse qualcuno come te, questo libro non esisterebbe e non avrebbe senso di esistere.
Non posso che ringraziare poi ancora una volta Marco Aleotti e tutto il team di **Edizioni LSWR**, per la rinnovata fiducia e per la collaborazione.
Grazie alla mia amica **Francesca Scionti**, che mi ha dato più volte l'incoraggiamento necessario per portare a termine questa nuova "opera". E non solo, grazie anche per aver trovato il tempo e la voglia di rileggere, correggere qualche congiuntivo e qualche congiunzione: d'altronde se c'è qualcuno più precisino di me, forse è lei (l'unica cosa che non ha potuto correggere è questa dedica, perché nemmeno l'avrebbe voluta).
E come tutte le cose che faccio, alla mia famiglia e alle due persone che non solo arricchiscono, ma compongono la mia intera vita.

A **mia moglie Caterina**: vorrei essere il migliore al mondo, e anche se non lo sarò mai, vorrei che mi guardassi e mi pensassi come se lo fossi.
A **mia figlia Matilda**: prima che tu nascessi sognavo di poter diventare un giorno il tuo supereroe preferito; e tu mi fai sentire già il miglior supereroe del mondo.

```
while(true)
{
  Noi();
}
```

3

Concetti di base di C#

In questo capitolo verrà mostrato il funzionamento generale di qualsiasi programma scritto in linguaggio C#, quindi verranno introdotti gli elementi fondamentali della sua sintassi e le sue parole chiave.

Ogni linguaggio di programmazione è costituito da un insieme di *parole chiave* che costituisce una sorta di alfabeto o dizionario degli elementi consentiti e da un insieme di *regole* che devono essere rispettate per scrivere programmi validi nel linguaggio stesso.
Tramite questi elementi e queste regole sarà possibile esprimere una serie di istruzioni che indicano le operazioni e i calcoli da svolgere, magari basandosi anche su eventuali parametri inseriti esternamente dall'utente.
Se siete fra coloro che hanno già esperienza di programmazione in altri linguaggi, probabilmente potrete saltare o leggere in maniera molto rapida questo capitolo, che vi servirà comunque come riferimento rapido delle regole di base della sintassi o per rivedere quali siano le differenze rispetto al vostro linguaggio di programmazione di provenienza.
Nei prossimi paragrafi si analizzerà un primo semplice programma in C#, spiegando riga per riga e istruzione per istruzione la sua struttura di base e il suo funzionamento, per poi iniziare a introdurre i vari elementi del linguaggio C#, in modo tale che anche il lettore alle prime armi possa essere in grado di creare, fin da subito, i suoi primi semplici programmi.

Il primo programma

Per imparare un linguaggio di programmazione non basta esporne la teoria nuda e cruda, ma è necessario mettere in pratica ogni singolo esempio, provando e riprovando, sbagliando e trovando gli errori, ma anche il modo di correggerli. Per questo motivo si inizierà questa avventura nel meraviglioso mondo di C#, con originalità e fantasia, dal classico *Hello World*, cioè un programma che stampa su un prompt dei comandi proprio queste due parole.

> **NOTA**
>
> Se siete amanti delle curiosità, la prima apparizione documentata di un programma che stampi il testo *Hello World* sullo schermo, come primo esempio di programmazione, risale al 1972, in una guida introduttiva al linguaggio B scritta da Brian Kernighan, a uso interno dei laboratori Bell. Kernighan fu autore poi insieme a Dennis Ritchie del famoso libro *The C Programming Language*, che ripropone e rende l'esempio *Hello World* un classico nel mondo della programmazione e dei libri sull'argomento.

Nel capitolo precedente, nel paragrafo dedicato a .NET Core SDK e alla CLI in particolare, si è già visto il codice C# generato dal comando `dotnet` nella creazione di un'applicazione console. Tale programma è proprio un esempio di Hello World, che qui di seguito è riproposto per comodità e con qualche lieve modifica.
Riprendete quindi il codice sorgente del programma generato, oppure copiate le seguenti righe di codice all'interno di un file di testo vuoto, con il vostro editor di testo preferito (per esempio nel Blocco note di Windows):

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            //Stampa la stringa Hello World
            Console.WriteLine("Hello World");
        }
    }
}
```

Si noti solo la presenza di una riga aggiuntiva, quella che inizia con i simboli `//` seguita da un testo esplicativo (`Stampa…`).

Come già detto esso costituisce il sorgente C# del programma la cui azione, una volta eseguito, sarà quella di stampare sullo schermo la sequenza di caratteri `"Hello World"`.

> **NOTA**
>
> In informatica una sequenza di caratteri (cioè un testo) è chiamata *stringa*, e spesso, come in questo caso e come si continuerà a fare nel libro, è rappresentata racchiudendo i caratteri all'interno di doppi apici, come fatto appunto per la stringa `"Hello World"`.

Se non lo avete già fatto, salvate il file assegnandogli un nome a vostra scelta, utilizzando però l'estensione `.cs` (abbreviazione di C Sharp), per esempio **helloworld.cs**.

> **NOTA**
>
> Non è obbligatorio, a differenza di quanto accade, per esempio, con Java, nominare i file di codice C# con lo stesso nome della classe in esso contenuta. In C# è possibile utilizzare un nome a propria scelta, compresa l'estensione, anche se naturalmente sarà comodo dare un nome che corrisponda al contenuto e assegnare l'estensione `.cs` per riconoscere i file sorgenti e C#.

Per rendere eseguibile un programma, come quello appena scritto, bisogna procedere alla sua compilazione; come visto nel capitolo precedente esistono diverse possibilità per compilare il codice C#: dipende dal sistema operativo utilizzato e dagli strumenti installati. Per il momento, e per diversi capitoli a seguire, sarà sufficiente utilizzare semplicemente il comando `dotnet` oppure il compilatore `csc`, ma, se si preferisce e li si ha a disposizione, è naturalmente possibile compilare i programmi con gli ambienti di sviluppo Visual Studio o Visual Studio Code.
Per ulteriori informazioni sulla compilazione, se non avete ancora bene a mente il processo, potete tranquillamente tornare al capitolo precedente e ripetere i passaggi necessari.

Struttura di un'applicazione

Dopo aver scritto e compilato il primo programma C#, si passerà ora ad esaminarlo, per capire, riga per riga e istruzione per istruzione, il loro significato e funzionamento. Naturalmente non si pretende da chi non ha mai programmato che tutto sia chiaro alla prima lettura, ma non preoccupatevi, scopo di questa analisi dettagliata è quello di iniziare a familiarizzare con la struttura di ogni programma C#.

Ogni programma C# (ma il discorso vale anche per altri linguaggi di programmazione) è costituito da una sequenza di *istruzioni*, che verranno in qualche modo interpretate ed eseguite dall'ambiente di esecuzione, a partire da un preciso punto di partenza.
Dal punto di vista fisico, i programmi C# sono costituiti da uno o più file di codice sorgente.
C# è un linguaggio di programmazione *orientato agli oggetti*, e il concetto fondamentale di questo paradigma di programmazione (detto anche *object oriented*) è quello di **classe**.
Ogni programma deve contenere almeno una classe, la quale a sua volta contiene dei membri (per esempio campi, metodi, proprietà, eventi): ecco spiegato perché un pur semplice programma *Hello World*, che pretende semplicemente di stampare una stringa sullo schermo, sia costituito da tutte quelle istruzioni.
Per comodità di organizzazione del codice, più classi possono essere organizzate in contenitori logici, detti spazi dei nomi, ovvero **namespace**: un concetto simile a quello di directory e file, ma che esiste solo a livello logico e non fisico.
Un *namespace* è, in parole povere, un contenitore logico di tipi.
Tornando proprio al programma di esempio *Hello World*, esso inizia con la seguente istruzione:

```
using System;
```

Essa indica che vogliamo utilizzare nel nostro programma elementi contenuti nel namespace denominato `System`, che fa parte della libreria di base di ogni implementazione .NET e che quindi è uno dei più utilizzati.
L'istruzione `using System` sarà praticamente onnipresente in tutti i programmi che scriverete e incontrerete nella vostra vita di programmatori C#.
Nel nostro caso particolare abbiamo utilizzato l'istruzione `using`, perché all'interno del namespace indicato denominato `System` è presente la classe **`Console`** che verrà utilizzata qualche riga dopo.

NOTA

Alcune parole chiave, e fra queste la parola chiave `using`, possono assumere significati diversi all'interno di un programma C#, a seconda del contesto in cui vengono utilizzate. Più avanti nel libro si vedranno gli altri possibili significati e utilizzi di `using`.

L'istruzione `using System` ci permette di utilizzare la classe `Console` senza necessità di riferirsi a essa con il suo nome completo, cioè `System.Console`.

La riga seguente indica quindi che stiamo inserendo all'interno dello spazio dei nomi, o namespace, denominato `HelloWorld`, il blocco di codice che la segue e che è racchiuso fra parentesi graffe.

```
namespace HelloWorld
{
    …
}
```

All'interno di questo blocco, e quindi all'interno del namespace `HelloWorld`, viene poi definito un tipo, nel caso specifico una *classe*, denominata `Program` (chi è alle prime armi o non ha mai avuto a che fare con il paradigma di programmazione orientato agli oggetti, non deve preoccuparsi se il termine "classe" può risultare oscuro, ci torneremo su molto presto!). Per creare una classe si utilizza la parola `class` seguita dal nome che si vuole assegnare ad essa. In questo caso:

```
class Program
{
    …
}
```

Questa è l'intestazione della classe, ed è seguita da un blocco di istruzioni racchiuse ancora fra parentesi graffe. Tale blocco costituisce il *corpo* della classe stessa.
Al suo interno viene poi definito un unico metodo, cioè un'operazione eseguibile dalla classe, denominato `Main`. Ogni programma eseguibile C# deve contenere un metodo `Main` (si noti l'iniziale maiuscola, deve essere scritto proprio così) che costituirà il punto di ingresso dal quale il programma stesso inizierà la sua esecuzione.

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

Tralasciamo per ora le altre parole e simboli presenti prima (`public static void`) e dopo `Main` (le parentesi e relativo contenuto `(string[] args)`) passiamo al corpo dello stesso metodo, cioè il blocco ancora una volta delimitato e racchiuso fra parentesi graffe.
La prima è una riga di commento:

```
//Stampa la stringa Hello World
```

Ogni riga di testo che inizia con due barre o *slash* `//` indica un **commento** a quella parte di programma, che tornerà utile a chi leggerà il codice per comprenderne il significato e il funzionamento, ma che verrà ignorata dal compilatore. Dopo il commento, sempre all'interno del metodo `Main` vi è una sola istruzione, chiusa dal punto e virgola a indi-

carne la fine, il cui scopo è proprio quello di stampare a video le parole racchiuse fra i doppi apici.
Il linguaggio C# non possiede istruzioni standard per scrivere sullo schermo, ma utilizza le classi e i metodi messi a disposizione dalla *Base Class Library* (BCL) di .NET.
La BCL contiene, all'interno del namespace `System` (ecco il perché della prima istruzione `using`), la classe `Console`, che fornisce i metodi necessari, fra le altre cose, a stampare del testo sul prompt dei comandi, detto anche *console*.
In particolare il metodo `WriteLine` permette di scrivere la stringa indicata fra parentesi.

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

Dopo questa istruzione vi sono solo delle parentesi graffe chiuse, che servono ad indicare la fine dei relativi blocchi di codice: il blocco del metodo `Main`, il blocco della classe `Program`, e infine il blocco del namespace `HelloWorld`.
L'indentazione del testo, cioè la presenza di spazi vuoti prima delle istruzioni o delle parentesi non è obbligatoria, ma aiuta a identificare le parti del codice, e quindi a comprenderne più facilmente la struttura, il significato e quindi il funzionamento.

Ciclo di vita di un'applicazione

Il prodotto della compilazione .NET è in generale un *assembly*; se esso contiene un punto di ingresso (vedere il prossimo paragrafo) si dice che è un'*applicazione eseguibile* o semplicemente un *eseguibile*.
L'avvio di un'applicazione avviene quando l'ambiente di esecuzione, cioè il runtime .NET (CLR o CoreCLR, che per gli scopi della discussione possiamo considerare intercambiabili), invoca il metodo designato come punto di ingresso (cioè il metodo `Main`, decritto nel prossimo paragrafo).
Al termine dell'esecuzione di un'applicazione, il controllo viene restituito all'ambiente di esecuzione.

Il metodo Main

Nel paragrafo precedente abbiamo già avuto modo di apprendere che ogni programma C#, che debba poi essere compilato sotto forma di applicazione eseguibile, deve contenere un **punto di ingresso** (o *entry point*) che è costituito da un metodo denominato `Main`.
Chi non conosce il concetto di metodo, può pensarlo come a una procedura o operazione costituita da un elenco più o meno lungo di passi.
Il metodo `Main` è il punto dal quale l'applicazione inizierà la sua esecuzione, o meglio è il punto che il CLR di .NET cercherà per iniziare l'esecuzione dell'applicazione.

NOTA

C# è un linguaggio *case sensitive*, cioè che nella scrittura del codice sorgente è sensibile alla differenza fra lettere minuscole e maiuscole: si noti l'iniziale maiuscola del metodo `Main`, deve essere scritto proprio così.

Esistono diversi modi per implementare il metodo `Main`, a seconda che il programma debba ricevere in ingresso dei parametri (che possono essere indicati dall'utente dalla riga di comando) o meno, e debba restituire al termine dell'esecuzione un codice di uscita o meno.
Nell'esempio precedente, il metodo è preceduto da tre parole chiave del linguaggio C#:

```
public static void Main(string[] args)
```

La parola chiave **`public`** indica che un membro (in questo caso un metodo) è utilizzabile anche all'esterno della classe di cui fa parte.
La presenza della parola chiave `public` non è però in questo caso obbligatoria, in quanto il metodo `Main` è un metodo speciale, che deve essere utilizzato solo dal runtime, quindi in generale esso viene scritto senza essere preceduto da `public`.
La parola **static** è invece obbligatoria e deve essere sempre indicata per il metodo `Main`, ma, essendo un concetto un po' più complesso, dovrete pazientare un po' prima di averne una spiegazione: ci torneremo al momento opportuno.
L'ultima parola chiave prima del nome del metodo è **void**. In generale ogni operazione al termine della sua esecuzione può indicare o meno un risultato, restituendolo a chi ha avviato l'operazione stessa: esprimendo lo stesso concetto in termini di programmazione C#, un metodo può restituire o meno il risultato della propria esecuzione al chiamante del metodo stesso. In questo caso, `void` indica che nessun risultato sarà passato all'esterno.
Il metodo `Main` può essere scritto in quattro differenti modi (tralasciando la presenza o meno di `public` per i motivi appena esposti), naturalmente inserendolo sempre all'interno di una classe (dato che non esistono funzioni o metodi globali in C#):

```
static void Main()
static void Main(string[] args)
static int Main()
static int Main(string[] args)
```

La parola chiave `void` o `int` indica il tipo di risultato che può essere restituito all'esterno. All'interno delle parentesi che seguono il nome del metodo, invece, verranno indicati eventuali parametri che potranno essere utilizzati dal metodo stesso. Essi per esempio potranno essere inviati al programma da parte dell'utente, al suo avvio mediante il prompt dei comandi.

La prima forma di scrittura è quella più semplice, in quanto non restituisce nulla all'esterno e non riceve nessun parametro in ingresso.
Nel secondo caso, invece, è possibile avviare il programma usando degli argomenti. Per esempio se, una volta compilato e ottenuto l'eseguibile `helloworld.exe`, lanciamo l'esecuzione mediante il comando:

```
C:\> helloworld.exe param1
```

Il valore `param1` verrà inviato al metodo `Main`.
Utilizzando il comando `dotnet` per eseguire un progetto o direttamente un assembly già compilato, supponiamo `helloworld.dll`, il parametro sarà inviato in maniera analoga, scrivendolo alla fine del comando stesso, per esempio:

```
C:\> dotnet helloworld.dll param1
C:\> dotnet run HelloWorld param1
```

Vedremo in seguito come leggere, trattare e utilizzare questi parametri di ingresso all'interno del programma.
Nelle ultime due modalità di scrittura del metodo `Main`, è previsto che esso debba restituire all'esterno un codice numerico (numero intero per l'esattezza) che costituisce il codice di terminazione del programma. Tale opportunità viene utilizzata in genere quando il programma deve essere utilizzato per inviare informazioni sul proprio stato ad altri programmi o script che chiamano l'eseguibile. Il seguente è un esempio di metodo `Main` che restituisce il valore intero 0:

```
static int Main()
{
    return 0;
}
```

NOTA

Con C# 7.1 è stata introdotta la possibilità di scrivere il metodo `Main` come *asincrono*. Poiché l'argomento va ancora al di là degli scopi di questo capitolo, soprattutto per chi si avvicina adesso a C#, lo affronteremo con il dovuto dettaglio nel Capitolo 13, dedicato alla programmazione asincrona.

Main multipli

Sebbene un programma .NET possa avere un solo punto di ingresso, è possibile implementare più metodi `Main`, naturalmente in classi differenti. In questo caso però è obbligatorio indicare in fase di compilazione quale punto di ingresso utilizzare, me-

diante apposite opzioni del compilatore. Supponiamo, per esempio, di aver creato due diverse classi, ognuna contenente un metodo `Main`:

```
using System;
namespace MainMultipli
{
     class Program
     {
          public static void Main(string[] args)
          {
            Console.WriteLine("Hello World");
          }
     }

    class Program2
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello World 2");
        }
    }
}
```

Compilando l'applicazione con il comando `dotnet` nella .NET Core CLI, basta specificare l'opzione `/p:StartupObject`. Per esempio, per indicare il primo `Main` come punto di ingresso, si compilerà con il comando:

```
dotnet build /p:StartupObject=MainMultipli.Program
```

Un'altra possibilità per indicare il punto di ingresso da utilizzare è quella di farlo all'interno del file di progetto `.csproj`, mediante l'elemento `StartupObject`:

```
<PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.0</TargetFramework>
    <StartupObject>MainMultipli.Program</StartupObject>
  </PropertyGroup>
</Project>
```

Analogamente, compilando con il compilatore `csc` un eseguibile che usi il metodo `Main` della classe `Program`, si dovrà utilizzare il comando:

```
csc /main:MainMultipli.Program helloworld.cs
```

Compilando invece un programma con più metodi `Main`, senza specificare quale si vuol utilizzare tramite le precedenti opzioni, il compilatore restituirebbe un errore, indicando che sono stati definiti due o più punti di ingresso e che è necessario specificare appunto quale classe contiene il punto di ingresso desiderato.

Si noti che per compilare correttamente abbiamo dovuto indicare anche il nome completo della classe `Program`, cioè `MainMultipli.Program`, ottenuto con il namespace seguito dal punto e quindi dal nome della classe.
Avere più metodi `Main` potrebbe tornare utile per svariati motivi, per esempio per la scrittura di classi di test, per eseguire una particolare versione dell'applicazione con determinati parametri, per compilare separatamente una versione del programma che avvii l'interfaccia grafica, anziché invece quella a riga di comando e così via.

NOTA

Anche con Visual Studio è naturalmente possibile indicare il punto di ingresso da utilizzare. Se già sapete orientarvi all'interno delle sue funzionalità, aprite le proprietà di un progetto e andate alla schermata *Application*, dove sarà possibile impostare il campo *Startup Object*, scegliendo una delle classi contenenti un metodo `Main`.

Sintassi di base di C#

Ogni linguaggio di programmazione possiede una serie di parole utilizzabili per scrivere un programma e definisce una serie di regole di sintassi e semantica da osservare per scrivere programmi corretti.
In maniera simile a una lingua parlata, per imparare un nuovo linguaggio di programmazione si partirà dagli elementi semplici, che verranno utilizzati per creare elementi più complessi, allo stesso modo in cui, per creare una frase, si utilizzeranno parole, avverbi, preposizioni e così via.
In un linguaggio di programmazione, tali elementi semplici sono parole chiave, simboli, spazi, operatori e altri elementi che impareremo a conoscere nel resto del libro.

NOTA

C# appartiene alla famiglia dei linguaggi derivati dal linguaggio C, e in quanto tale la sintassi è basata su blocchi di codice delimitati da parentesi graffe e istruzioni terminate dal punto e virgola. In tal senso C# è quindi simile anche a linguaggi come C++ e Java.

Il codice di un programma C# è composto da una serie di *istruzioni*, ognuna delle quali terminata da un punto e virgola (`;`). Gli spazi presenti fra le istruzioni sono ignorati dal compilatore, quindi non ha importanza il modo di formattare e allineare il codice del programma: serve solo per permettere a sé stessi o ad altri sviluppatori di leggere e capire il significato del codice in maniera più immediata.

Per esempio, sarebbe possibile scrivere su una sola riga le istruzioni del programma *HelloWorld* visto in precedenza e compilarlo ugualmente senza problemi:

```
using System;namespace HelloWorld{public class Program { static void Main(string[] args) {
Console.WriteLine("Hello World");}
```

Si risparmierebbe certamente spazio, ma la leggibilità del codice sarebbe a dir poco compromessa.
Per prendere dimestichezza con il linguaggio e con il compilatore, vi consigliamo di provare a scrivere qualche esempio, utilizzando i vari elementi del linguaggio che presenteremo da qui alla fine del capitolo.
Potete utilizzare come schema di partenza l'esempio *Hello World* riproposto qui di seguito e inserire all'interno del metodo `Main` le vostre istruzioni:

```
using System;
class Program
{
    public static void Main(string[] args)
    {
        //inserite le vostre istruzioni e compilate!
    }
}
```

Commenti

I commenti al codice sono elementi fondamentali di qualunque programma, e servono a inserire testi utili a spiegare il significato di una parte di codice cui si riferiscono, ma che saranno completamente ignorati dal compilatore.
In C# è possibile inserire commenti in differenti maniere:

- commenti su una linea;
- commenti multiriga o delimitati;
- commenti di documentazione.

La prima modalità prevede linee singole di commento, anche consecutive, ed è realizzata facendo precedere il testo che costituisce il commento da due caratteri `//`, per esempio:

```
// questa è una linea di commento
// questa è una seconda linea
istruzione1; //questo commento inizia dopo l'istruzione1
```

Il commento che segue l'`istruzione1` è utile per commentare su una stessa riga una particolare istruzione, in quanto è immediatamente chiaro a che cosa si riferisce il commento stesso.

La seconda modalità per scrivere commenti in C# consente di scriverli anche su più righe di testo, racchiudendo il testo fra una sequenza di inizio commento `/*` e una di fine commento `*/`. Ecco un paio di esempi:

```
/* questo commento è scritto
   Su più linee
   Di testo */
istruzione1; /*questo è un commento delimitato*/
```

I commenti delimitati sono spesso utilizzati per commentare una parte di codice su più righe, magari temporaneamente. Per esempio, nel seguente codice le istruzioni 2 e 3 sono commentate e quindi non verranno compilate.

```
istruzione1;
/*
istruzione2;
istruzione3;
*/
Istruzione4;
```

Nel caso di commenti delimitati è necessario fare attenzione al caso particolare in cui all'interno del testo che costituisce il commento sia presente la sequenza `*/`, in quanto il commento si chiude appunto alla prima sua occorrenza. Nel seguente esempio:

```
/* questo commento contiene la sequenza "*/", questa parte è fuori dal commento */
```

il commento finisce alla prima occorrenza di `*/`, quindi la parte successiva non fa parte del commento e il compilatore tenterebbe di interpretarla come codice da eseguire (e naturalmente non riuscirebbe a compilare).
Un'altra tipologia di commenti permette di ottenere automaticamente della documentazione del proprio codice. Tali commenti sono creati utilizzando la sequenza `///` e usando al loro interno appositi tag xml per descrivere informazioni e contenuto.
Tali tag sono riconosciuti dal compilatore e possono essere usati per estrarre dai commenti gli elementi con cui creare un file XML di documentazione.
Ecco un esempio di metodo commentato con dei tag, che descrivono il funzionamento del metodo all'interno del tag `summary`, il significato del valore di ritorno e dei parametri:

```
///<summary>
/// Il metodo Somma esegue la somma di due numeri interi
///</summary>
///<returns>risultato della somma</returns>
///<param name="x">Primo numero</param>
///<param name="y">Secondo numero</param>
public int Add(int x, int y)
{
    return x + y;
}
```

Per indicare al compilatore di estrarre tali commenti e produrre la documentazione si deve utilizzare l'apposita opzione **/doc**, indicando il nome del file da generare:

```
csc /doc:help.xml Sorgente.cs
```

Il file XML ottenuto può essere poi elaborato mediante appositi strumenti per generare documentazione utilizzabile e distribuibile in vari formati: chm, html, word, pdf ecc.

Le variabili

Nell'esempio *Hello World* di qualche paragrafo fa non era presente alcuna forma di memorizzazione di dati. In generale in un programma che abbia un obiettivo pratico sarà necessario immagazzinare dati, elaborarli o usarli per qualche calcolo, permettere all'utente di inserirli da tastiera o leggerli sul monitor. Quindi in generale avremo bisogno di una sorta di contenitore in cui conservare dati, che viene chiamato **variabile**. La sintassi generale per dichiarare una variabile è la seguente:

```
<tipodati> <identificatore>;
```

Supponiamo per esempio di aver bisogno di conservare il valore di un numero intero, che magari rappresenta un anno di nascita; scriveremo allora:

```
int anno;
```

Questa istruzione dichiara una variabile di tipo `int` (cioè un numero intero) denominata `anno`. La variabile `anno` non ha ancora alcun valore assegnato e per utilizzarla è necessario prima farlo mediante un'istruzione di **assegnazione**, che utilizza l'operatore `=`, con la seguente sintassi:

```
anno=1975;
```

Se si tenta di utilizzare in un'istruzione qualsiasi (escluso quella di assegnazione naturalmente!) una variabile non inizializzata, il compilatore restituirà un errore abbastanza chiaro (del tipo "utilizzo di una variabile non assegnata"). Per esempio, scrivendo:

```
int età;
Console.WriteLine(età);
```

otterremo l'errore `"use of unassigned local variable età"`.
Una variabile può anche essere contemporaneamente dichiarata e inizializzata (cioè si assegna a essa un valore iniziale), in un'unica istruzione. Le due precedenti istruzioni si possono riunire cioè in una sola:

```
int anno=1975;
```

Se volessimo dichiarare più variabili di uno stesso tipo, sarebbe possibile farlo su una stessa riga, senza ripetere il tipo, per esempio:

```
int x = 1, y = 2;
```

L'istruzione precedente ha dichiarato due variabili di tipo `int`, chiamate `x` e `y`, e assegnato loro rispettivamente i valori `1` e `2`.
Per dichiarare, invece, variabili di tipo diverso, bisogna obbligatoriamente farlo su righe separate:

```
int x, y;
string nome, cognome;
double altezza;
```

La prima istruzione dichiara due variabili intere, `x` e `y`; la seconda due variabili `string` (cioè stringhe di testo) `nome` e `cognome`, e la terza una variabile di tipo `double` (un valore numerico con la virgola a doppia precisione) denominata `altezza`.
C# è un linguaggio fortemente tipizzato; ciò significa che il tipo di un qualunque elemento, per esempio di una variabile, è determinato già in fase di compilazione. Negli esempi precedenti, infatti, abbiamo dovuto stabilire al momento della dichiarazione della variabile il tipo di dati che essa potrà contenere e tale tipo non potrà essere modificato.

NOTA

Nella versione 4.0 di C# è stata introdotta la parola chiave `dynamic` che consente di creare variabili con tipo non assegnato a compile-time, e in generale quindi di rendere C# un linguaggio con caratteristiche dinamiche. Nel Capitolo 15 è spiegato nel dettaglio l'uso e il significato di `dynamic`.

Se si provasse per esempio a inizializzare una variabile di tipo `string`:

```
string nome;
```

e poi si tentasse di assegnarle un numero intero:

```
nome = 123;
```

si otterrebbe un errore in fase di compilazione che indica l'impossibilità di convertire un tipo `int` in `string`.

Identificatori

Nel precedente paragrafo abbiamo visto come sia possibile dichiarare una variabile assegnandole un nome detto identificatore.
In C# un **identificatore** è un nome che può essere utilizzato per diversi elementi in un programma: variabili, classi, metodi, namespace e così via. Un identificatore è una sequenza di caratteri Unicode, che inizia con una lettera o un underscore (_). I caratteri successivi possono poi essere lettere, underscore o anche numeri.

```
int numero; //ok
int _numero; //ok
int numero1; //ok
string straße; //ok
string état = "état"; //ok

int 1numero:// no, non può iniziare per numero
int num-ero; //no, contiene carattere - non valido
```

All'interno di un identificatore C# è anche possibile utilizzare il codice Unicode di un carattere, nella forma \uNNNN; per esempio, il codice del carattere _ corrisponde al valore \u005f, quindi è possibile scrivere una variabile in questo modo:

```
int \uOO5fIdentificatore;//equivalente a _Identificatore
```

Come già detto C# è un linguaggio case sensitive, quindi anche in questo caso bisogna rispettare maiuscole e minuscole. Per esempio, è possibile dichiarare tre variabili differenti nel seguente modo:

```
int anno;
int Anno;
int aNNo;
```

Naturalmente dichiarare e distinguere gli identificatori solo per mezzo delle maiuscole o minuscole è una pratica sconsigliata, in quanto porterebbe con ogni probabilità a errori dovuti a confusione o a uno scambio non voluto di variabili.

Nomenclatura delle variabili

La nomenclatura delle variabili, cioè la scelta di un nome da assegnare a esse e il formato utilizzato (maiuscole, minuscole), è fondamentale per scrivere un programma comprensibile anche a chi non l'ha scritto (o comunque anche a chi l'ha scritto, se è passato un certo periodo di tempo).
Esistono quindi regole o convenzioni, dette *naming convention*, che gli sviluppatori tendono più o meno a seguire, anche se poi influiscono sulla scelta il gusto personale o le scelte aziendali e di team.

In generale in .NET vengono utilizzati due tipi di convenzioni per la scrittura degli identificatori, soprattutto quando sono formati da più parole composte.
Tali convenzioni sono dette *PascalCase* e *camelCase*. La regola è molto semplice: in PascalCase ogni prima lettera delle parole che compongono l'identificatore è scritta in maiuscolo; con il camelCase, invece, la prima lettera dell'identificatore rimane minuscola.

```
string NomeCognomeIndirizzo; //pascal case
string nomeCognomeIndirizzo; //camel case
```

Per le variabili in C# è utilizzata, quasi all'unanimità, la convenzione camelCase, mentre in qualche caso, spesso per le variabili di classe, si fa iniziare l'identificatore con un underscore _, in maniera che ci si renda conto a prima vista che si tratta di un campo e non di una variabile locale.
Per altri elementi, come i nomi dei metodi o delle classi, si utilizza la convenzione Pascal.
Ripeto che spesso i gusti personali la fanno da padrone, con scelte anche discutibili come le seguenti:

```
string questa_e_una_variabile_con_underscore;
```

Un altro consiglio di buona programmazione è utilizzare sempre identificatori che abbiano un significato, in maniera da poter capire più facilmente il funzionamento del codice.
Per esempio, se volessimo calcolare l'area di un triangolo mediante la classica formula che si impara alle elementari, Area= Base x Altezza : 2, sarebbe opportuno dichiarare le variabili con nomi attinenti a ciò che andranno a memorizzare.
Quindi scrivere:

```
double base, altezza;
…
double area= base*altezza/2;
```

è certamente meglio che usare variabili "non parlanti", cioè con nomi che non indicano chiaramento il loro significato, come di seguito:

```
double a,b;
…
double c=a*b/2;
```

I tipi primitivi

I tipi di dati sono un concetto fondamentale per ogni programmatore, e non soltanto sulla piattaforma .NET.
Nei precedenti paragrafi abbiamo già visto, infatti, come uno dei componenti fondamentali del .NET Framework sia la Base Class Library, una libreria di tipi; abbiamo inoltre introdotto il Common Type System, che definisce lo standard seguito per la definizione e l'utilizzo dei tipi stessi, e abbiamo anche dovuto creare una classe (che è anch'essa un tipo), denominata `Program`, per scrivere il nostro primo programma.
Nel prossimo capitolo dedicheremo ampio spazio al concetto di tipo e nei successivi capitoli vedremo come un'applicazione C# sia composta da una collezione di tipi.
In questo e nei prossimi paragrafi elencheremo i tipi di dato primitivi e il loro utilizzo.
Abbiamo già visto come, al momento di definire una variabile, il primo passo sia quello di scegliere il tipo di dato da trattare. Quindi se si ha la necessità, per esempio, di memorizzare l'età di una persona in anni, si dovrà scegliere un tipo che consenta di lavorare con numeri interi, mentre se si sta sviluppando un programma di calcolo matematico sarà più opportuno utilizzare un tipo che consenta di gestire numeri con la virgola, e ancora, se il programma prevede la possibilità di memorizzare il nome della persona, una variabile dovrà poter contenere caratteri alfabetici.
Tutti i valori memorizzati in una variabile sono istanze di qualche tipo.
C# fornisce una serie di **tipi primitivi** che il compilatore supporta in maniera diretta mediante **alias** dedicati, che però sono più di semplici abbreviazioni o nomi.
In particolare abbiamo già accennato nel primo capitolo, parlando dell'architettura di .NET, al sistema comune di tipi, o **Common Type System** (CTS).
Quindi se utilizziamo, per esempio, l'alias `int` per definire una variabile di tipo intero, stiamo utilizzando in realtà un'istanza del tipo `System.Int32` del Common Type System di .NET.

> **NOTA** L'unico tipo primitivo di .NET che non possiede un alias è `System.IntPtr`, che rappresenta un puntatore o un handle.

Da ciò deriva che in .NET anche i valori di tipi semplici sono in realtà oggetti e, come vedremo parlando dei tipi complessi e delle classi, sono quindi dotati di metodi.
Per esempio, impareremo che ogni classe e quindi ogni oggetto è dotato di un metodo `ToString`, che permette di ottenere una rappresentazione testuale del suo valore:

```
int i=123;
string str = i.ToString(); //restituisce la sequenza di caratteri "123"
```

In C# esistono tredici differenti tipi semplici che possono essere suddivisi in categorie diverse a seconda della tipologia di dato che possono rappresentare; nei seguenti paragrafi li vedremo uno per uno.

Tipi interi

La tabella seguente mostra i tipi numerici interi (o integrali) predefiniti di C# e, fra le altre informazioni, la corrispondenza con il tipo del Common Type System.

Tabella 3.1 - Tipi interi predefiniti di C#.

Nome	Tipo CTS	Descrizione	Intervallo valori
`byte`	`System.Byte`	8 bit	da 0 a 255
`sbyte`	`System.SByte`	8 bit signed	da -128 a 127
`short`	`System.Int16`	16 bit	da -32.768 a 32.767
`ushort`	`System.UInt16`	16 bit unsigned	da 0 a 65.535
`int`	`System.Int32`	32 bit	da -2.147.483.648 a 2.147.483.647
`uint`	`System.UInt32`	32 bit unsigned	da 0 a 4.294.967.295
`long`	`System.Int64`	64 bit	da -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
`ulong`	`System.UInt64`	64 bit unsigned	da 0 a 18.446.744.073.709.551.615

La colonna con l'intervallo di valori possibili spiega il perché dell'esistenza di diversi tipi per rappresentare dei tipi numerici. Non basterebbe un semplice tipo `int` per rappresentare un qualsiasi numero? La risposta è data anche dalla colonna con la descrizione in cui è riportato il numero di bit necessari per rappresentare un valore del tipo corrispondente.

Ogni numero (così come ogni altro dato in realtà) è rappresentato da una sequenza di bit 0 e 1. Quindi per poter rappresentare un intero si utilizza il sistema numerico binario. Una sequenza di N bit può rappresentare un valore intero che va da 0 a 2^N-1.

Pertanto, se si utilizzano 8 bit, come nel caso del tipo byte, si potranno memorizzare solo valori che vanno da 0 a 2^8-1 che è pari a 255.

Se volessimo, quindi, memorizzare in una variabile intera un valore maggiore o uguale a 256, non sarebbero più sufficienti 8 bit.

Per chi non avesse dimestichezza con il sistema binario e le modalità di rappresentazione dei numeri, ecco un piccolo esempio. Supponiamo di avere a disposizione solo 2 bit, ognuno dei quali può assumere il valore 0 o 1. I diversi valori rappresentabili con 2

bit sono ottenuti combinando in ogni modo possibile i valori 0 e 1, ottenendo un totale di 4 combinazioni (in generale le combinazioni saranno pari a 2^N dove N rappresenta il numero di bit utilizzato):

```
00 = 0
01 = 1
10 = 2
11 = 3
```

Quindi in tal modo potrei rappresentare dei valori interi che vanno da 0 a 3 (o come detto prima da 0 a 2^N-1 = 2^2-1 = 4 -1 = 3). In generale quindi devo aggiungere dei bit e utilizzare sequenze più lunghe per poter creare più combinazioni e rappresentare valori più grandi.
Con 64 bit si riesce, come mostrato dalla tabella, a memorizzare valori fino a 18.446.744.073.709.551.615, un bel numero, ma che probabilmente non sarebbe sufficiente per applicazioni scientifiche nel campo dell'astronomia.
Nel caso dei tipi con segno (per esempio `sbyte`, `short`, `int`, `long`) uno dei bit non può essere utilizzato per il valore da memorizzare, ma sarà utilizzato per rappresentare il segno del numero. Per esempio, il tipo `sbyte` utilizza 1 bit per il segno + o -, mentre i rimanenti 7 bit saranno utilizzati per il valore. Con questi 7 bit posso rappresentare numeri con valore massimo pari a 2^7 = 128, quindi valori che vanno da -128 a +127 (e non 128, ricordiamo che c'è anche lo 0 da rappresentare).
La conclusione generale è che, con la rappresentazione binaria, più bit si hanno, più alti sono i valori rappresentabili.
D'altro canto se nella mia applicazione ho bisogno di rappresentare numeri più piccoli, per esempio l'età di una persona o il suo anno di nascita, è sufficiente utilizzare il tipo `byte` nel primo caso, `ushort` nel secondo, evitando di sprecare memoria per conservare 32 o 64 bit, la maggior parte dei quali rimarrebbe con valore 0.

NOTA

Sebbene molti dei nomi dei tipi semplici utilizzati in C# siano uguali a quelli di altri linguaggi, come C++ o Java, prestate attenzione alla loro definizione e lunghezza. Inoltre, notate che in .NET i tipi numerici sono indipendenti dalla piattaforma: un `int`, in quanto `Int32`, sarà sempre a 32 bit, mentre in C++ un `int` è a 32 bit se la piattaforma è Windows a 32 bit, ma è a 64 bit su un sistema a 64 bit.

I tipi interi con la presenza di un prefisso `u` nel nome sono tipi **unsigned**, cioè senza segno, quindi non permettono di memorizzare valori negativi. Per quanto riguarda il tipo

`byte`, esso è senza segno come comportamento predefinito, mentre il byte con segno si chiama **sbyte**, che sta per *signed byte*.
Naturalmente, per assegnare un valore numerico a una variabile, il tipo della variabile deve essere abbastanza capiente da contenerlo, altrimenti si verificherebbe un errore di compilazione. Per esempio:

```
byte a = 200; //ok
byte b = 300; //errore, il valore 300 è troppo grande per il tipo byte
```

Nel paragrafo "Valori letterali", poco più avanti in questo capitolo, verranno presentate varie modalità possibili per assegnare valori dei diversi tipi primitivi di C#.

Tipi a virgola mobile

In C# esistono due tipi a virgola mobile per rappresentare numeri non interi con la virgola. Anch'essi hanno un tipo CTS corrispondente (Tabella 3.2).

Tabella 3.2 - Tipi a virgola mobile predefiniti di C#.

Nome	Tipo CTS	Descrizione	Cifre decimali
`float`	`System.Single`	32 bit a singola precisione	7 cifre
`double`	`System.Double`	64 bit a doppia precisione	circa 15 cifre

Il tipo **`flat`** è un tipo a singola precisione e permette di rappresentare numeri con circa 7 cifre decimali, mentre il numero di tipo **`double`** può rappresentare circa 15 cifre decimali. Per assegnare un valore numerico con la virgola si usa una rappresentazione con il punto (`.`) per rappresentare il separatore dei decimali, per esempio:

```
double d = 1.234;
```

Il compilatore assume che un valore come il precedente sia da interpretare come `double`, quindi per assegnare un valore `float` bisogna aggiungere un suffisso `F` (oppure `f`) al numero:

```
float f = 0.1; //errore compilazione
float g = 0.1F; //ok
```

Nel primo caso si ha un errore in fase di compilazione perché il compilatore tenta di salvare un valore di tipo `double` in una variabile di tipo `float`.
Anche per il tipo `double` è possibile utilizzare esplicitamente il suffisso `D` (oppure `d`):

```
double d = 0.1; //ok
double e = 0.1D; //ok, equivalente
```

A causa della rappresentazione binaria, la precisione dei numeri a virgola mobile non è esatta, soprattutto nei risultati delle operazioni aritmetiche.
Provate per esempio a eseguire questa operazione:

```
float f = 0.1F * 9999999;
Console.WriteLine(f);
```

Matematicamente ci aspetteremmo la stampa a video del risultato 999999.9, invece si ottiene uno strano valore di 999999.938, con una perdita di precisione non indifferente.

Tipo decimal

Se si ha bisogno di un tipo a virgola mobile a maggior precisione, per esempio per effettuare calcoli su valute su cui non ci si può permettere una bassa precisione, .NET mette a disposizione un ulteriore tipo detto **`decimal`**.

Tabella 3.3 - Tipo a virgola mobile decimal di C#.

Nome	Tipo CTS	Descrizione	Cifre decimali
`decimal`	`System.Decimal`	128 bit alta precisione	28 cifre

Il tipo `decimal` usa 128 bit, quindi è più lento dei tipi `float` e `double`, e utilizza una notazione diversa per ottenere 28 cifre decimali di precisione. Per assegnare un valore `decimal` a una variabile si usa il suffisso `M` (oppure `m`):

```
decimal importo=100000.00M;
```

NOTA

In realtà il tipo `System.Decimal` è l'unico tipo valore a non essere un tipo primitivo di .NET. Viene riportato in questo elenco, come nella maggior parte dei testi, in quanto è un tipo numerico fondamentale, tant'è che è possibile usarlo per mezzo dell'alias `decimal`.

Tipo booleano

Il tipo **`bool`** serve a rappresentare uno dei due valori logici di verità, **`true`** oppure **`false`** (vero o falso).

NOTA

Il nome del tipo booleano deriva da George Boole, matematico considerato il fondatore della logica matematica.

Tabella 3.4 - Tipo booleano di C#.

Nome	Tipo CTS	Descrizione	Valori possibili
`bool`	`System.Boolean`	rappresenta un valore booleano	`true` oppure `false`

A differenza di altri linguaggi, in C# non è possibile convertire valori booleani in numeri interi o viceversa. Quindi, se una variabile è di tipo `bool` potrà assumere solo uno dei due valori `true` o `false`.
I valori di questo tipo sono utilizzati per indicare il verificarsi o meno di determinate condizioni, che possono quindi servire per intraprendere, o meno, determinate azioni. Per esempio, l'istruzione `if` permette di eseguire o meno un blocco di codice a seconda che la condizione fra parentesi assuma il valore `true` oppure `false`:

```
bool b;
…
if(b)
{
    //se b è true eseguo il blocco di codice
}
```

Nel Capitolo 6 verrà utilizzato estensivamente il tipo `bool` per valutare condizioni e controllare il flusso del programma mediante varie tipologie di istruzioni.

Tipo carattere

Per rappresentare singoli caratteri, C# mette a disposizione il tipo **char**.

Tabella 3.5 - Tipo char di C#.

Nome	Tipo CTS	Descrizione	Range
`char`	`System.Char`	singolo carattere Unicode a 16 bit	da U+0000 a U+ffff

I caratteri **Unicode** sono caratteri a 16 bit usati per rappresentare gran parte dei caratteri presenti nelle lingue di tutto il mondo: caratteri alfanumerici, accentati, segni di punteggiatura, caratteri speciali e così via.
I valori letterali del tipo `char` possono essere scritti come caratteri effettivi, sequenze di escape esadecimali o rappresentazioni Unicode, racchiudendoli fra singoli apici.
Tutte le istruzioni che seguono dichiarano per esempio una variabile `char` e la inizializzano con il carattere X:

```
char char1 = 'X';
char char2 = '\x0058';   // esadecimale
char char3 = '\u0058';   // rappresentazione Unicode
```

È inoltre possibile impostare una variabile `char` anche con valori interi, con una conversione di tipo, in quanto esiste una corrispondenza uno a uno fra un valore `char` e il tipo `int`.

```
char char4 = (char)88;   // conversione da int
```

Esistono poi *sequenze di escape* per rappresentare caratteri speciali. Per impostare un valore `char` mediante una delle sequenze elencate in tabella, bisogna utilizzare il carattere backslash (`\`) subito dopo il singolo apice.

Tabella 3.6 - Sequenze escape per il tipo char.

Sequenza escape	Nome carattere	Valore unicode
`\'`	Singolo apice	0x0027
`\"`	Doppio apice	0x0022
`\\`	Backslash	0x005C
`\0`	Null	0x0000
`\a`	Alert	0x0007
`\b`	Backspace	0x0008
`\f`	Form feed	0x000C
`\n`	New line	0x000A
`\r`	Ritorno a capo	0x000D
`\t`	Tab orizzontale	0x0009
`\v`	Tab verticale	0x000B

Il tipo string

Una stringa è una sequenza di caratteri di testo, che in C# può essere dichiarata e utilizzata per mezzo del tipo **`string`**. Ogni carattere di un oggetto `string` è di tipo `char`, quindi una stringa è composta da caratteri in formato Unicode.
Essendo fra i tipi più utilizzati, anche per le stringhe il compilatore C# può utilizzare un apposito alias `string`, in alternativa al nome completo del tipo che è `System.String`.
Per rappresentare una stringa basta racchiudere la sequenza di caratteri all'interno di doppi apici (a differenza del tipo `char`, in cui il singolo carattere è racchiuso fra apici singoli); per esempio `"hello world"` e `"Ciao"` sono due stringhe:

```
string str = "hello world";
string str2 = "Ciao";
```

Una stringa non ha limiti di lunghezza, la dimensione massima è limitata semplicemente dalla memoria a disposizione.

> **NOTA**
>
> La differenziazione principale dei vari tipi di .NET è quella fra la famiglia di tipi valore e la famiglia dei tipi riferimento. A differenza degli altri tipi semplici di C#, che fanno parte dei cosiddetti tipi valore, `string` è un tipo riferimento. Nel prossimo capitolo vedremo la differenza fra queste due categorie di tipi.

Una stringa in .NET è immutabile. Ciò significa che i caratteri che la compongono non possono essere modificati una volta creato l'oggetto `string`.
Potrebbe sembrare una stranezza e un grave limite: come facciamo a lavorare con un oggetto `string` se non possiamo nemmeno modificarne i singoli caratteri, per esempio convertendo i caratteri in maiuscolo o minuscolo, operazione del tutto normale?
In realtà la risposta è semplice: un oggetto `string` è immutabile, quindi eseguendo delle operazioni su di esso viene sempre restituita una nuova stringa con le modifiche richieste.
Per esempio, per unire due stringhe formandone una sola, è possibile semplicemente utilizzare l'operatore `+`:

```
string str1 = "hello";
string str2 = "world";
str1 = str1 + str2;
Console.WriteLine(str1); //il risultato è "helloworld"
```

Nella terza istruzione alla variabile `str1` viene assegnato un valore `"helloworld"` ottenuto dalla concatenazione di due altre stringhe. Dietro le quinte `str1` non viene modificata, ma viene creato un oggetto `string` totalmente nuovo contenente la stringa `"helloworld"`, che viene assegnata a `str1`, in sostituzione del precedente valore `"hello"`.
Ancora più chiaro se proviamo a utilizzare il metodo `ToUpper` della classe `String`:

```
string str3 = str1.ToUpper(); //converte in maiuscolo il valore di str1
Console.WriteLine(str3); //stampa "HELLOWORLD"
Console.WriteLine(str1); //stampa "helloworld"
```

Il metodo `ToUpper` consente di convertire in maiuscolo il valore della stringa su cui è invocato. Quindi, eseguendo la prima istruzione, la variabile `str3` conterrà il valore di `str1` convertito in maiuscolo, che viene stampato con la seconda istruzione.
La terza istruzione stampa il valore di `str1`, che sarà rimasto sempre uguale a `"hello"`; ciò significa che la nuova stringa in maiuscolo è stata inserita in un nuovo oggetto, senza modificare `str1`.

Come già avrete notato dagli esempi precedenti, il tipo `System.String` mette a disposizione una serie di metodi e proprietà per lavorare e interagire facilmente con oggetti di tipo `string`. Per esempio, per conoscerne la lunghezza basta utilizzare la proprietà `Length`:

```
int lunghezza = str.Length; //ricavo la lunghezza della stringa
```

Nell'Appendice A potete trovare una trattazione più approfondita sul tipo `System.String` e relativi metodi e proprietà; non è stato inserito qui perché ancora mancano diverse nozioni per capire fino in fondo il loro funzionamento e utilizzo.

Il tipo object

Per completezza, chiudiamo l'elenco dei tipi primitivi, con un secondo tipo riferimento. Tale tipo è una classe fondamentale, in quanto è la classe madre da cui ogni altro tipo che incontreremo discende o deriva, sia che si tratti di uno di quelli standard forniti dal framework .NET, sia che si tratti di un tipo personalizzato, per esempio fornito da terze parti oppure implementato da noi stessi.
Se non avete familiarità con il paradigma della programmazione orientata agli oggetti, espressioni come "classe madre" e "derivare da una classe" non avranno molto senso: portate pazienza, chiariremo tutte nel prossimo capitolo.
La superclasse da cui ogni altro tipo deriva è la classe `System.Object`, per la quale il compilatore C# riconosce l'alias `object`.
Il tipo `System.Object` definisce un tipo generico, che implementa metodi generali che ogni altro tipo avrà dunque a disposizione.

Valori letterali

Per rappresentare dei valori in C# e assegnarli a variabili o utilizzarli in espressioni o istruzioni è comodo utilizzare la loro rappresentazione testuale all'interno del codice C#.
Abbiamo infatti già utilizzato dei numeri per assegnarli a variabili intere, oppure dei suffissi per i numeri a virgola mobile, o ancora abbiamo visto come un singolo carattere sia rappresentato racchiudendolo fra apici singoli e così via.
Tali valori letterali servono a rappresentare quindi un valore sotto forma di codice C#. Ogni tipo semplice visto nei paragrafi precedenti ha dei formati per rappresentare i propri valori letterali.
Per esempio, per assegnare le seguenti variabili numeriche si utilizzano dei valori letterali:

```
float f = 0.1F;
double d = 0.1D;
char ch = 'a';
```

Già negli esempi precedenti abbiamo utilizzato qualcuno di essi, ma in questo paragrafo faremo un rapido riepilogo, che poi sarà possibile adoperare come riferimento dei valori letterali utilizzabili con i vari tipi semplici di C#; indicheremo anche le regole che vengono utilizzate dal compilatore per usare i valori con più tipi (per esempio 123 è un valore letterale che può rappresentare uno qualunque dei tipi interi predefiniti).

Valori letterali booleani

I valori possibili per un tipo `bool` sono `true` e `false`, che rispettivamente hanno il significato di vero e falso. Non c'è molto altro da dire.

```
bool bt = true;
bool bf = false;
```

Valori letterali interi

Un valore letterale intero serve a rappresentare valori dei tipi `int`, `uint`, `long`, `ulong`, e può assumere due forme, quella intera e quella esadecimale.
Nella forma intera si può aggiungere al valore letterale un suffisso. Il tipo di un valore intero viene determinato seguendo queste regole.

- Se non è presente un suffisso, il valore viene considerato del primo tipo fra i seguenti in cui esso può essere rappresentato: `int`, `uint`, `long`, `ulong`.
- Se il valore ha il suffisso `U` oppure `u`, esso è del primo fra i seguenti tipi in cui può essere rappresentato: `uint`, `ulong`.
- Se il suffisso è `L` o `l`, il tipo che il valore assume è il primo fra i seguenti: `long`, `ulong`.
- Se il suffisso invece è `UL` (oppure un'altra delle possibili combinazioni `Ul`, `uL`, `ul`, `LU`, `Lu`, `lU`, `lu`) il valore è di tipo `ulong`.

In ogni caso il valore letterale intero deve essere all'interno del range massimo permesso dal tipo `ulong` (18.446.744.073.709.551.615); in caso contrario verrebbe generato un errore di compilazione. In generale, per quanto riguarda il suffisso `L`, esso è da preferire alla l minuscola, per evitare di confonderla con il numero 1.
I seguenti sono esempi di assegnazione di un numero intero alle rispettive variabili:

```
int i = -456;
uint ui = 1000;
long l =  -123456789L;
ulong ul = 123456789UL;
```

Oltre che con un valore letterale intero, come negli esempi precedenti, è possibile utilizzare anche la notazione esadecimale, cioè `0x` seguito dal valore in base 16 del numero da rappresentare:

```
int i = 0x1C8; // è il valore esadecimale di 456
```

Valori letterali reali

Per rappresentare numeri reali e utilizzarli come `float`, `double` o `decimal`, è possibile utilizzare un numero composto da parte intera e da un'eventuale parte decimale, separandoli con il punto e aggiungendo un eventuale suffisso.
Il suffisso `F` (o `f`) si utilizza per il tipo float, `D` (o `d`) per i `double`, `M` (o m) per i `decimal`. Se il suffisso viene omesso, si sottintende un numero di tipo `double`.

```
float f = 10.0F;
double d = 123.45D;
decimal dec = 78.9M;
float f2 = 1.2; //errore di compilazione, il numero senza suffisso è double.
```

La seconda modalità per rappresentare un numero reale è quella che viene comunemente chiamata **notazione scientifica**, e utilizza una parte intera e un esponente, preceduto quest'ultimo da una lettera `e` oppure `E`.
L'esponente può essere sia positivo sia negativo e rappresenta un fattore in potenza di 10. Per esempio:

```
double d = 1.23e5; // è pari a 1.23 x 10^5
double d = 4.5e-2; // è pari a 4.5 x 10^-2
```

Si noti che la lettera `E` non può confondersi con il valore `E` in numerazione esadecimale, perché essa non può essere utilizzata con numeri reali.

Separatori di cifre

Quando si ha a che fare con numeri composti da molte cifre, potrebbe sorgere qualche difficoltà nel leggerli e comprendere il valore. Quindi, analogamente ai separatori delle migliaia utilizzati per separare i numeri in gruppi, C# (a partire dalla versione 7.0) permette di separare le cifre utilizzando il carattere _ (underscore).
Supponiamo per esempio di avere la dichiarazione della seguente variabile:

```
int numero = 1234567890;
```

La variabile `numero` è composta da 10 cifre, perciò, per aumentarne la leggibilità, si possono raggruppare le cifre a tre a tre a partire da destra, scrivendo:

```
int numerosep = 1_234_567_890;
```

Il separatore è utilizzabile in tutti i tipi numerici (meglio naturalmente se sono tipi che permettono valori molto grandi), per esempio `long`, `double` e così via.

```
long longNum = 1_234_567_890;
double longDoub = 1_234_567.890_9999;
```

Anche con numeri in notazione esadecimale è possibile separare le cifre, per esempio separando i byte rappresentati con due cifre ciascuno in uno stesso numero:

```
long hex = 0x01_23_45_67_89_AB_CD_EF;
```

Il separatore _ può essere inserito (a partire dalla versione C# 7.2) anche subito dopo il prefisso di base 0x:

```
long hex = 0x_01_FF; //errore
long hex = 0x01_FF; //ok
```

Valori letterali binari

Per rappresentare valori numerici direttamente in formato *binario*, mediante cioè sequenze di bit 0 e 1, anziché scrivere i numeri in valore decimale oppure esadecimale, è possibile utilizzare il prefisso di base `0b`.
Per esempio per dichiarare un intero di valore `8`, che in binario è rappresentato come `1000`, basta scrivere:

```
var b = 0b1000;
```

Naturalmente anche in questo caso possiamo separare i bit con il carattere _, utilissimo per rappresentare numeri molto lunghi, per esempio raggruppandoli a blocchi di 4 bit:

```
b = 0b1010_0100_1111_1100; // = 42236 in formato decimale
```

Anche in questo caso, il separatore _ può essere inserito subito dopo il prefisso `0b`:

```
b = 0b_1010_0100_1111_1100;
```

Valori letterali per i caratteri

Nel paragrafo sul tipo carattere abbiamo già esposto le possibili modalità di rappresentazione letterale di un carattere.
In generale basta racchiudere il carattere da rappresentare fra apici singoli, mentre altre due possibilità sono la rappresentazione mediante il valore esadecimale e quella mediante il valore Unicode.

```
char ch = 'a';
char char2 = '\x0058'; // esadecimale
char char3 = '\u0058'; // rappresentazione Unicode
```

L'ultima possibilità è utilizzare il carattere `\` per formare le sequenze di escape (vedi la Tabella 3.6).