- Antonio Pelleriti -

PROGRAMMARECON

C# 6

Guida completa

2ª EDIZIONE BESTSELLER

Compilatore e ambiente di sviluppo Visual Studio 2015 >>

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

Sviluppo per Windows, DB, LINQ, XML, Compiler API >>

Sintassi e costrutti del linguaggio >>

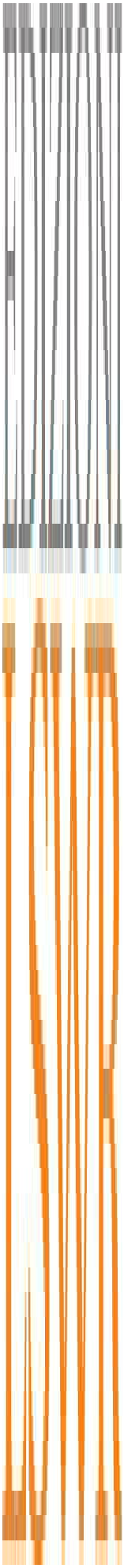
*pro DigitalLifeStyle

PROGRAMREON

C# 6

Guida completa

Antonio Pelleriti



Programmare con C# 6 | Guida completa
Autore: Antonio Pelleriti

Collana: *pro DigitalLifeStyle

Editor in Chief: Marco Aleotti
Progetto grafico: Roberta Venturieri
Immagine di copertina: © djvstock | Thinkstock
Realizzazione editoriale e impaginazione: Studio Dedita di Davide Gianetti

ISBN: 978-88-6895-308-9

EDIZIONI LSWR

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

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

Sommario

Introduzione

Il linguaggio di programmazione C# è ormai un attore protagonista maturo ed esperto sul palcoscenico dello sviluppo software, non solo in ambito Microsoft. Esso è infatti sulla scena da quasi quindici anni e ha continuato a evolversi costantemente dal 2000 a oggi, introducendo, di volta in volta, in ognuna delle versioni rilasciate, caratteristiche e funzionalità nuove, volte a migliorare la produttività dello sviluppatore che lo utilizza nel proprio lavoro quotidiano.

In tale evoluzione il linguaggio non è mai divenuto pesante e complesso e, anzi, ha mantenuto la semplicità e la freschezza che gli hanno permesso, dividendo naturalmente il merito con la piattaforma .NET Framework per cui e con cui è nato, di poter affrontare e risolvere problemi legati ad ambiti di sviluppo appartenenti a mondi differenti, sia dal punto di vista della piattaforma di esecuzione sia da quello prettamente pratico e legato all'ambito applicativo.

Per tale motivo C# e .NET costituiscono oggi una delle principali scelte per chi vuole creare un software che giri sul desktop di un personal computer, come applicazione web all'interno di un browser internet o, ancora, come app installata su uno smartphone o su un tablet, spaziando in qualunque caso in degli scenari applicativi estremamente eterogenei: dal mondo dell'industria a quello dei software di produttività e gestione aziendale, passando per i videogame e i sistemi di commercio elettronico.

La piattaforma .NET ha costituito una delle principali rivoluzioni nel mondo dello sviluppo software, soprattutto per quanto riguarda l'ambiente costituito dai sistemi operativi di Microsoft, cioè delle varie versioni di Windows, che oggi abbracciano anche il mondo mobile.

Le ultime release, a partire da quelle della famiglia Windows 8.x e oggi Windows 10, permettono di sviluppare app che gli utenti possono acquistare e scaricare direttamente dal Windows Store, un negozio digitale analogo all'App Store di Apple o al Play Store di Google, e che quindi costituisce una vetrina che si affaccia su un mercato immenso di potenziali clienti.

Ma C# e .NET non sono assolutamente legati e limitati al mondo Windows e di Microsoft.

I componenti fondamentali di .NET, oltre al linguaggio C#, sono divenuti degli standard open e quindi liberamente implementabili anche su sistemi differenti, opportunità che, come dimostra

l'implementazione open source Mono che può girare anche su sistemi Linux o Mac, non è rimasta solo una fantasia o un'opportunità mai sfruttata.

Le recenti dichiarazioni di Microsoft hanno inoltre aperto strade un tempo raramente percorse, abbracciando il mondo open source. La nuova visione di Microsoft ha quindi dato vita a un nuovo filone di .NET, denominato nella sua interezza .NET 2015, all'interno del quale trova posto .NET Core, un framework open source, modulare, portabile e quindi dedicato principalmente allo sviluppo cross platform. In tal modo esso potrà facilmente essere portato su sistemi e architetture diverse, come Linux e MacOS-X.

Fra i componenti principali di .NET Core sono da citare i nuovi compilatori (fra cui quello di C#), riscritti completamente in .NET e resi anch'essi disponibili alla comunità di sviluppatori come progetti open source. Visual Studio 2015 stesso è basato su questa nuova piattaforma, chiamata .NET Compiler Platform, che a lungo è stata conosciuta come Roslyn, e che è stata messa anch'essa a disposizione degli sviluppatori per sfruttare le stesse API e librerie che costituiscono il cuore pulsante dell'ambiente di sviluppo di Microsoft.

Il parco mondiale di macchine dotate della piattaforma .NET supera il miliardo, secondo recenti conteggi fatti per mezzo di Microsoft Update, e quindi praticamente esatti. Su ognuna di queste macchine è possibile eseguire un'applicazione scritta in C#.

Volgendo lo sguardo verso un altro dei mercati che gli sviluppatori non possono più ignorare, quello degli smartphone e dei tablet, il sistema operativo Windows Phone ha riscosso un successo sempre crescente e l'Italia è una delle nazioni in cui tale successo è addirittura più marcato. Ancora meglio, con Windows 10 il sistema operativo si è mosso verso un'unificazione sempre più spinta, consentendo l'esecuzione di applicazioni scritte una volta sola, su piattaforme hardware estremamente eterogenee e che, a ragion veduta, sono state denominate applicazioni universali.

Con C# potrete sviluppare applicazioni di questo genere e, se volete abbracciare un mercato ancora più ampio, potrete addirittura mantenere lo stesso linguaggio per portare le vostre app nei territori un tempo ostili di Apple e di Android, sfruttando strumenti come quelli forniti da Xamarin, basati su C#.

Il linguaggio C# è, fra quelli che è possibile utilizzare per lo sviluppo .NET, quello che riflette maggiormente le caratteristiche distintive della piattaforma, in quanto nato con essa e per essa, e ne è quindi riconosciuto come il linguaggio principe. A oggi esso è giunto alla versione denominata C# 6.

Il Framework .NET, sviluppato praticamente di pari passo, segue una numerazione differente e la sua versione più recente, al momento della stampa, è la 4.6.1.

Microsoft inoltre, che già forniva agli sviluppatori Windows l'ambiente di sviluppo integrato denominato Visual Studio, ha rilasciato da subito la versione Visual Studio .NET per adeguarlo al nuovo mondo, aggiornando costantemente anche tale ambiente e affiancandolo alle versioni di .NET susseguitesi nel tempo.

Il libro che state iniziando a leggere esporrà quindi le caratteristiche del linguaggio C#, aggiornate all'ultima versione disponibile al momento della sua stesura, che è come detto la 6, e di .NET Framework 4.6.1, utilizzando come ambiente di sviluppo la versione Visual Studio 2015.

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 quello che invece possiede già una qualsivoglia esperienza in tale ambito, magari con linguaggi differenti da C# e su piattaforme diverse da .NET.

Alcuni concetti sono infatti comuni al mondo della programmazione orientata agli oggetti e quindi i capitoli iniziali che espongono tale paradigma di sviluppo possono essere letti in maniera rapida per arrivare al cuore della programmazione .NET in C#.

Scopo del libro è comunque quello di affrontare con la maggior precisione e approfondimento possibile i concetti trattati e costituire quindi un riferimento completo del linguaggio C#, anche per il programmatore già esperto che vuole avere a portata di pagina una guida rapida, a cui fare riferimento per chiarire dubbi o trovare risposte a domande e questioni più avanzate.

Inoltre, per chi, come il sottoscritto, vive e lavora nel mondo della programmazione da decenni e che quindi affronta l'argomento con ancora maggiore passione, ho cercato di trovare e fornire degli spunti e delle curiosità legate a C# e .NET nel diramarsi delle pagine e dei capitoli, facendo notare qual è stata l'evoluzione del linguaggio lungo le sue varie versioni e indicando in quale di esse è stata introdotta ogni nuova funzionalità o caratteristica.

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# 6 fino ad arrivare ai concetti più complessi, permettendo di padroneggiare così ogni argomento che riguardi la programmazione .NET, anche quelli non trattati in questo testo.

La prima parte del libro, costituita dai primi cinque capitoli, introduce il framework .NET e le caratteristiche del linguaggio C# puro, iniziando dalle parole chiavi, dalla sintassi con cui si scrivono i programmi, introducendo i tipi fondamentali del framework .NET, le espressioni e gli operatori, e i costrutti per controllare il flusso di esecuzione dei programmi.

Il Capitolo 1 esegue una prima panoramica di .NET e dei suoi componenti, come il CLR e la Framework Class Library, mostrando anche i concetti di base della compilazione ed esecuzione dei programmi scritti in C#, ed elencando poi gli strumenti di programmazione che saranno utilizzati nel resto del libro e in ogni attività di sviluppo fatta come sviluppatori C#.

Il Capitolo 2 mostra un primo programma C# e ne analizza il funzionamento dopo averlo compilato mediante strumenti come il compilatore a riga di comando oppure l'ambiente integrato Visual Studio. Lo stesso capitolo introduce la sintassi di base del linguaggio e i suoi elementi.

Il Capitolo 3 espone il sistema di tipi di .NET e le varie categorie di tali tipi creabili e utilizzabili in C#.

Nel Capitolo 4 si vedrà come scrivere espressioni più o meno complesse all'interno di un programma, utilizzando i vari operatori messi a disposizione dal linguaggio.

Il Capitolo 5 invece mostra come controllare l'esecuzione di un programma, utilizzando gli appositi costrutti e istruzioni di controllo del flusso.

A partire dal Capitolo 6 si entra nel mondo della programmazione a oggetti in C#, perciò esso introduce concetti che permettono l'implementazione di classi personalizzate e struct e, quindi, dei vari membri che ne possono costituire la struttura.

Il Capitolo 7 è la logica continuazione del precedente e approfondisce altri concetti della programmazione a 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 8, viene introdotta la gestione delle cosiddette eccezioni, cioè delle situazioni di errore che si possono verificare durante

l’esecuzione dei programmi.

Il Capitolo 9 tratta le collezioni di oggetti e la gestione di tipi parametrici mediante il meccanismo dei cosiddetti generics.

Il Capitolo 10 tratta una questione fondamentale di C#, cioè quella della programmazione a eventi e dei metodi di gestione degli stessi, e di argomenti strettamente legati, come quello dei delegate, dei metodi anonimi e delle espressioni lambda.

Avanzando lungo i capitoli si copriranno tutte le sfaccettature del linguaggio C#, introdotte nelle sue varie versioni. Il Capitolo 11 include quindi anche materie come LINQ, che permette l’interrogazione di varie forme di dati mediante una nuova sintassi e nuovi metodi e tipi, introdotti in C# 3.0.

Il Capitolo 12 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 13 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 14 scende in profondità nei meandri della composizione dei tipi. Infatti il cosiddetto meccanismo di reflection permette di analizzare ogni aspetto di un oggetto, a tempo di esecuzione. Nello stesso capitolo si vedranno anche gli attributi e il loro utilizzo.

Il Capitolo 15 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, LINQ to SQL e Entity Framework per l’accesso ai database relazionali.

Nel Capitolo 16, viene fatta una panoramica della .NET Compiler Platform, o Roslyn, mostrando come utilizzare i servizi messi a disposizione del compilatore nelle applicazioni o scrivere estensioni di Visual Studio.

Nell’ultimo, il Capitolo 17, tramite lo sviluppo di semplici esempi in diversi ambiti, viene mostrata infine 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 mostra dei rapidi cenni sulla programmazione con i puntatori e sull’utilizzo da codice gestito di funzioni native, per mezzo dei servizi P/Invoke.

L'Appendice C contiene le soluzioni alle domande di riepilogo poste alla fine di ogni capitolo.

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.

Esempi pratici e capitoli bonus

Ogni capitolo del libro contiene esempi pratici che potete scrivere e compilare autonomamente, sia utilizzando il compilatore a riga di comando csc, sia utilizzando l'ambiente di sviluppo Visual Studio, in particolare la versione 2015; anche le versioni precedenti sono adeguate se non avete intenzione di utilizzare le caratteristiche del linguaggio introdotte con C# 6 (ma se state leggendo questo libro probabilmente vorrete farlo!).

Per chi non fosse provvisto di una licenza professionale di Visual Studio 2015 non c'è alcun problema: come avremo modo di vedere già dai primi capitoli, è disponibile una versione denominata Community liberamente scaricabile da Internet, che permette di sviluppare e distribuire applicazioni scritte in C#, per ogni ambiente, per esempio Desktop, Web, o Mobile.

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 che, per limiti di spazio o per aggiornamenti al testo con argomenti usciti dopo la stampa, 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 a 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, spero numerose, 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 a esso correlate sono i suoi principali interessi fin dal rilascio della prima beta della piattaforma Microsoft, quindi da quasi 15 anni. In tale ambito il riconoscimento forse più importante è la nomina a Microsoft MVP per .NET da gennaio 2015 e MVP per Visual Studio e Tecnologie di Sviluppo da gennaio 2016.

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, nel 2014, la prima edizione di "Programmare con C# 5, guida completa".

Dopo aver girato in lungo e 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 diventando consulente e freelance su progetti software di ogni dimensione e tipo.

Fa parte dello user group siciliano Orange Dot Net (http://www.orangedotnet.org) e partecipa spesso a conferenze ed eventi in qualità di speaker, trattando sempre argomenti legati allo sviluppo software.

Il suo sito personale, su cui pubblica articoli e pillole di programmazione legate sempre a .NET e C#, è www.antoniopelleriti.it.

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 della prima edizione è dunque una soddisfazione enorme, soprattutto per i feedback ricevuti dai lettori con le loro recensioni, i messaggi e le 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 dopo poco più di un anno mi sia ritrovato a scrivere l'edizione aggiornata è merito di tutti loro.

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.

E naturalmente, come sempre in maniera anticipata, ringrazio te, lettore, che stai tenendo in mano questo libro cartaceo o lo stai sfogliando virtualmente su un pc o un altro dispositivo e che quindi hai già riposto fiducia nei miei confronti (magari per la seconda volta, se hai già letto la prima edizione!).

Grazie Nonna Concetta, che mi protegge sulla stella più bella che c'è.

Infine, e come sempre, grazie a mia moglie Caterina, che mi sopporta, mi incoraggia e mi sostiene istante per istante e che, durante gli ultimi tempi di lavoro e revisione, ha portato, spesso faticosamente, in grembo la nostra prima bimba Matilda, che adesso è senza dubbio la cosa più importante della nostra vita. while(true) {ViAdoro();}

C# e la piattaforma .NET

Il .NET Framework è un ambiente di sviluppo ed esecuzione di applicazioni, introdotto da Microsoft all'inizio degli anni 2000, insieme a C#, linguaggio di programmazione multiparadigma e principalmente orientato agli oggetti.

La piattaforma .NET nasce alla fine degli anni Novanta, quando Microsoft inizia a lavorare a un nuovo e rivoluzionario modello di sviluppo e programmazione dei propri (ma non solo) sistemi operativi, più semplice e al contempo più potente rispetto a quello fino ad allora utilizzato dai programmatori del mondo Windows.

A quei tempi il nome con cui Microsoft si riferiva a tale nuova piattaforma, ancora in fase di sviluppo, era *NGWS*, acronimo di *Next Generation Windows Services,* che stava proprio a indicare le intenzioni di creare una nuova generazione di strumenti con cui si sarebbero sviluppati software e servizi per il sistema operativo Windows.

Insieme al nuovo framework, che assume il nome definitivo di .NET Framework quando viene ufficialmente annunciata la sua prima beta pubblica (il 13 novembre 2000), viene progettato un nuovo linguaggio di programmazione, anch'esso denominato con un nome provvisorio, COOL (*C-Like Object Oriented Language*), per poi essere battezzato con l'ormai noto nome di C# (C Sharp) e che nasce per diventare il linguaggio principe per lo sviluppo di software sulla nuova piattaforma.

Sebbene il nome .NET possa essere fuorviante per chi si sta approcciando per la prima volta al framework, in quanto potrebbe richiamare concetti legati esclusivamente al mondo Internet, esso deriva dalla volontà, dalla convinzione e quindi dal conseguente impegno che Microsoft intende profondere nel mondo delle applicazioni, sempre più distribuite e interconnesse, appartenenti al mondo desktop, ma anche al Web e, negli ultimi anni, al mondo dei dispositivi mobili come smartphone e tablet.

Essendo quindi C# così fortemente e quasi indissolubilmente legato alla piattaforma .NET, è necessario che chiunque intenda iniziare a sviluppare in tale linguaggio, o che aspiri comunque a passare a un livello di sviluppo avanzato, comprenda che la conoscenza dei vari aspetti del .NET Framework costituisce un prerequisito fondamentale della programmazione in C#.

Non è infatti minimamente immaginabile pensare a C# come a un linguaggio di programmazione a sé stante e slegato da .NET, anche se sia il linguaggio sia l'ambiente di esecuzione dei programmi, seppur inventati da Microsoft, sono divenuti degli standard ECMA e ISO/IEC, che chiunque può liberamente implementare su qualunque piattaforma; Microsoft ha inoltre reso open source importanti parti sia del framework .NET e delle librerie di base, sia il compilatore di C# e relativi servizi.

Prima di .NET

Perché a un certo punto della storia dello sviluppo software è sorta la necessità di inventarsi una nuova piattaforma di sviluppo e di esecuzione delle applicazioni e un nuovo linguaggio di programmazione? Come spesso accade, la storia stessa può aiutarci a fornire la risposta.

Prima dell'avvento di .NET, il mondo degli sviluppatori Windows doveva barcamenarsi all'interno di una giungla di diverse tecnologie, tecniche e linguaggi.

Si poteva scegliere di programmare servendosi delle API *Win32*, che costituivano l'interfaccia di programmazione del sistema operativo in linguaggio C, oppure fare uso delle Microsoft Foundation Classes (MFC) in C++; altri ancora sceglievano *COM*, la modalità di sviluppo indipendente dal linguaggio scelto per creare oggetti e componenti, che forse può essere considerata il predecessore di .NET, almeno negli intenti e negli scopi.

Dalla prima versione di Windows, la 1.0 rilasciata nel 1985, per semplificare lo sviluppo di applicazioni e quindi aumentare la produttività (si pensi che un Hello World che mostri una finestra nella prima versione di Windows era lungo all'incirca 150 linee di codice), Microsoft introduce diverse novità: nuove versioni di API a 32 bit, nuove funzioni, librerie, strumenti, tecnologie e anche la possibilità di programmare utilizzando linguaggi diversi dal C, al fine di nascondere (o almeno per cercare di farlo!) le complessità di più basso livello.

Vedono così la luce, in ordine sparso e senza pretese di esaustività, sigle e nomi come Win32, COM, ATL, DDE, ActiveX, MFC, Visual Basic ecc.

Arrivati alla fine degli anni Novanta, le necessità e i possibili ambiti applicativi su cui gli sviluppatori possono e devono cimentarsi sono diventati parecchi e si apre inoltre un nuovo orizzonte a cui affacciarsi in maniera sempre più frequente, il Web.

L'avvento di .NET

Quando in Microsoft si inizia a progettare .NET, l'obiettivo da raggiungere è ben chiaro: .NET vuole essere innanzitutto un ambiente unificato che supporti lo sviluppo e l'esecuzione di applicazioni.

Esso non è legato ad alcun sistema operativo in particolare, anche se Microsoft ha sempre rilasciato le proprie versioni per l'esecuzione su sistemi Windows e Windows Mobile/Windows Phone. Oggi, la visione strategica più recente e aggiornata di Microsoft ha dato vita, sotto il nome di .NET 2015, a un framework all'insegna del cross platform e dell'open source, che consentirà il funzionamento ottimale anche su sistemi Linux e Mac.

Un'altra implementazione non Microsoft, molto nota e anch'essa open source, è quella denominata Mono (vedere http://www.mono-project.com/ per approfondire), che fornisce una versione di .NET che, oltre a Windows, gira su sistemi Linux e MacOS X e che comprende anche il compilatore C# e strumenti di sviluppo come *MonoDevelop*.

Ulteriori implementazioni consentono inoltre di sviluppare in C# applicazioni per le piattaforme mobile iOS e Android (rispettivamente chiamate *Xamarin.iOS* e *Xamarin.Android*, anch'esse nate dal suddetto Mono, infatti le prime versioni erano note con i nomi di MonoTouch e MonoAndroid).

Oltre a non essere limitato ad alcun sistema operativo, .NET non ha vincoli di sorta nemmeno per il tipo di applicazioni che è possibile sviluppare; infatti consente di programmare ed eseguire applicazioni per il mondo desktop, per il Web, per il mobile e, in generale, per ogni ambiente su cui sarà possibile implementare l'ambiente di esecuzione .NET. Infine, come già detto, è anche possibile scegliere il proprio linguaggio preferito per lo sviluppo, naturalmente fra quelli che consentono una compilazione per l'ambiente .NET (ve ne sono a decine oltre a C#). Una delle potenzialità più rilevanti di tale caratteristica multilinguaggio è che sarà possibile utilizzare, da un'applicazione scritta in C#, anche librerie o componenti scritti in altri linguaggi.

NOTA	Un elenco dei linguaggi di programmazione utilizzabili per lo sviluppo in .NET è disponibile al seguente link: https://en.wikipedia.org/wiki/List_of_CLI_languages.

Definizione di .NET

.NET Framework è una piattaforma per lo sviluppo e un ambiente di esecuzione che fornisce vari servizi alle applicazioni in esecuzione al suo interno.

Esso è costituito da due componenti principali: il Common Language Runtime (CLR), ovvero il motore di esecuzione che gestisce l'intero ciclo di esecuzione delle applicazioni, e la libreria di classi di base del .NET Framework, che fornisce una raccolta di codice testato e riutilizzabile dagli sviluppatori nelle proprie applicazioni.

Rispettando gli obiettivi che il team .NET si era posto, il framework possiede le seguenti caratteristiche di base:

- fornire una piattaforma di sviluppo moderna e orientata agli oggetti;
- interoperabilità fra diversi linguaggi di programmazione; i compilatori dei linguaggi utilizzabili in .NET generano un codice intermedio, denominato Common Intermediate Language (CIL), che, a sua volta, viene compilato in fase di esecuzione dal Common Language Runtime. Con questa funzionalità, le funzioni scritte in un linguaggio sono accessibili ad altri linguaggi e i programmatori possono concentrarsi sulla creazione di applicazioni nei propri linguaggi preferiti;
- multitarget in quanto è possibile creare delle librerie portabili che funzionano su diverse piattaforme come Windows, Windows Phone, Xbox;
- un sistema di tipi comune, infatti nei linguaggi tradizionali i tipi supportati sono definiti dal compilatore, il che rende difficile l'interoperabilità fra linguaggi diversi. .NET definisce un sistema di tipi (Common Type System) accessibili e utilizzabili da qualunque linguaggio che supporti il .NET Framework;
- un'estesa libreria di classi di base che i programmatori hanno a disposizione; possono quindi utilizzare una libreria di tipi e relativi membri facilmente accessibili dalla libreria di classi .NET Framework, anziché dover scrivere grandi quantità di codice per gestire operazioni comuni di programmazione di basso livello. Inoltre .NET Framework include delle librerie specifiche per particolari aree applicative, per esempio ASP.NET per il Web, ADO.NET o Entity Framework per l'accesso a database, WCF per lo sviluppo di applicazioni orientate ai servizi e così via;
- miglioramento e semplificazione della fase di distribuzione e versionamento delle applicazioni;
- un ambiente di esecuzione dalle performance elevate e ottimizzate per ogni piattaforma su cui esso è supportato;
- compatibilità di versione; infatti, in generale, un'applicazione sviluppata per una data versione del .NET Framework continuerà a funzionare su tutte le versioni successive;
- esecuzione side-by-side, cioè la possibile coesistenza di diverse versioni dell'ambiente di esecuzione di .NET sullo stesso computer, in maniera che ogni applicazione o ogni sua versione particolare possa essere eseguita nell'ambiente per il quale era stata sviluppata;

- gestione semplificata della memoria, grazie all'esecuzione automatica di un Garbage Collector; il programmatore è esentato dal doversi preoccupare di operazioni come allocare e rilasciare la memoria manualmente, è l'ambiente di esecuzione che si occuperà di tutto.

C# nasce assieme a .NET e riflette dunque ognuna delle caratteristiche progettuali del framework.

Ogni programma scritto in C# è eseguibile solo ed esclusivamente all'interno dell'ambiente di esecuzione .NET. Per tale motivo una comprensione approfondita di .NET e dei suoi singoli componenti (esposti nei prossimi paragrafi) è fondamentale per iniziare a programmare in C# e per riuscire a ottenere un livello elevato di conoscenza e messa in pratica efficace delle sue caratteristiche.

Il linguaggio C#

C# si pronuncia *C sharp* (oppure in inglese come se si leggessero le parole *see sharp*) ed è un linguaggio di programmazione orientato agli oggetti, ma con caratteristiche che gli permettono di essere definito multiparadigma, semplice e moderno, type-safe e dall'utilizzo non limitato o vincolato ad alcun ambito applicativo.

Esso affonda le sue radici nella famiglia di linguaggi che fanno capo al C e la sua sintassi lo rende molto simile a C++ e Java, tanto che originariamente il creatore di Java, *James Gosling*, dichiarò apertamente che C# era una semplice imitazione del suo linguaggio, anzi che C# era Java privato di affidabilità, produttività e sicurezza; in effetti non era possibile negare, all'epoca, che la sintassi dei due linguaggi fosse molto simile e che il .NET Framework (o come vedremo meglio a breve il CLR) fosse un concetto analogo alla *JVM* (*Java Virtual Machine*), l'ambiente di esecuzione dei programmi Java.

NOTA	Perché si chiama C#? In quanto linguaggio derivato dal C, il nome C# deriva dalla combinazione di C con il simbolo diesis preso dalla notazione musicale, che indica che la nota a cui si riferisce va alzata di un semitono, a intendere un linguaggio che è un passo superiore a C. Secondo altre interpretazioni il simbolo # è invece la combinazione di quattro +, quindi un modo per scrivere C++++. Per gli amanti delle curiosità: il simbolo corretto del diesis è ♯, che in inglese si legge *sharp* ed è diverso dal simbolo del cancelletto #. Ricordo ancora con un certo divertimento qualcuno che si ostinava a chiamare il linguaggio *C cancelletto* o *C diesis* (se siete fra questi ora non avete più motivo per farlo)!

Dal momento della sua nascita e quindi dalla sua prima versione, esso si è però costantemente evoluto, introducendo importanti novità in ognuna delle successive revisioni, che hanno aggiunto a C# anche caratteristiche prese da linguaggi funzionali e dinamici.

NOTA	C# è stato standardizzato dall'ente ECMA International come ECMA-334 standard e da ISO/IEC con lo standard ISO/IEC 23270:2006. Il compilatore di C# per il .NET Framework realizzato da Microsoft è un'implementazione che rispetta i due standard. Il file pdf dello standard ECMA è liberamente scaricabile all'indirizzo http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf. La versione C# 6 sfrutta invece la più recente implementazione open source del compilatore, chiamata in codice Roslyn, e che fa parte della .NET Compiler Platform.

Il creatore del linguaggio C# è *Anders Hejlsberg*, ingegnere software danese, leggenda vivente del mondo della programmazione. Hejlsberg vanta un'esperienza trentennale in Borland, dove ha creato il compilatore Turbo Pascal e il suo successore, Borland Delphi. Nel 1996 approda in Microsoft, dove inizialmente si occupa del *J*++ (versione Microsoft del linguaggio Java) e delle Windows Foundation Classes, per poi divenire l'architetto capo di C#.

Storia di C# e .NET

Fornire una sorta di cronistoria delle varie versioni di .NET e C# non è una questione di pura curiosità, ma serve a comprendere le correnti evolutive che hanno portato alla versione attuale di .NET e C#. Quindi verrà fatto ora un breve riepilogo delle release che si sono susseguite in questi anni.

C# viene annunciato al pubblico per la prima volta nel luglio del 2000 alla Professional Developer Conference, presentato insieme al .NET Framework e alla prima versione .NET di Visual Studio. In quell'occasione una versione beta viene rilasciata anche ai partecipanti alla conferenza.

La prima beta pubblica viene poi rilasciata il 13 novembre dello stesso anno, mentre la versione RC (Release Candidate) è annunciata da Bill Gates nell'ottobre del 2001.

La prima versione definitiva del .NET Framework, la 1.0, vede quindi la luce nel gennaio del 2002, seguita dopo poche settimane da Visual Studio .NET: è la prima versione dell'IDE che permette di sviluppare applicazioni per il nuovo ambiente di esecuzione.

Nell'aprile del 2003 il .NET Framework viene aggiornato alla versione 1.1, mentre C# passa direttamente alla versione 1.2, l'unica versione minore fino adesso rilasciata. Nella stessa occasione anche l'ambiente di sviluppo viene aggiornato, assumendo la denominazione di Visual Studio .NET 2003.

Per la versione 2.0, sia di .NET sia di C#, bisogna invece attendere il 2005. Tale versione introduce numerosi e importanti cambiamenti e segna anche la direzione di una rotta evolutiva totalmente differente da quella del concorrente Java. Fra le novità troviamo i *generics*, le *classi parziali* e i tipi *nullable*.

Sempre nello stesso anno viene rilasciato Visual Studio 2005, che perde il suffisso .NET in quanto ormai implicitamente presente.

.NET 3.0 (nome in codice *WinFX*), rilasciato nel 2006, è una versione incrementale rispetto alla precedente, che non apporta cambiamenti alla struttura di base, rimasta uguale alla 2.0, ma introduce dei nuovi componenti di contorno (*WPF*, *WCF*, *WF* e *Cardspace*).

Alla fine del 2007, il .NET Framework passa alla versione 3.5 mentre le specifiche di C# sono aggiornate alla versione 3.0, che introduce per la prima volta delle caratteristiche prettamente funzionali, come le espressioni *lambda* e *LINQ*. Visual Studio invece viene rilasciato nella versione 2008.

Nell'aprile 2010, .NET e C# passano alle rispettive versioni 4.0, mentre viene rilasciato anche Visual Studio 2010. Gli aggiornamenti principali di C# 4.0 sono costituiti dall'introduzione di caratteristiche tipiche di un linguaggio dinamico.

Nel febbraio 2012 il .NET Framework arriva alla versione 4.5 mentre Visual Studio viene aggiornato alla versione 2012. Nell'agosto dello stesso anno vengono pubblicate le specifiche di C# 5.0, che includono come caratteristica fondamentale il paradigma di programmazione *asincrona*.

La Tabella 1.1 riepiloga in maniera concisa tale evoluzione lungo gli anni.

Tabella 1.1 **- Storia delle versioni di .NET, C# e Visual Studio.**

Anno	.NET Framework	C#	Visual Studio
2002	1.0	1.0	Visual Studio .NET 2002
2003	1.1	1.2	Visual Studio .NET 2003
2005	2.0	2.0	Visual Studio 2005
2006	3.0	2.0	Visual Studio 2005
2007	3.5	3.0	Visual Studio 2008
2010	4.0	4.0	Visual Studio 2010
2012	4.5	5.0	Visual Studio 2012
2013	4.5.1 4.5.2	5.0	Visual Studio 2013
2015	4.6	6.0	Visual Studio 2015

La versione 4.6 del .NET Framework e l'edizione 2015 di Visual Studio, rilasciate il 20 luglio 2015, hanno apportato diverse novità in termini di innovazione, di produttività e di nuove funzionalità messe a disposizione degli sviluppatori di qualunque tipologia di software: desktop, Web, mobile, cloud e così via.

Contemporaneamente il linguaggio C# arriva alla versione 6.0 e, naturalmente, l'IDE e il framework la supportano pienamente.

Questo testo è quindi perfettamente aggiornato e allineato e descrive tutte le ultime caratteristiche di C#, utilizzando l'ambiente di sviluppo Visual Studio 2015 per la stesura ed esecuzione degli esempi pratici. Notate che per l'utilizzo della versione 6.0 di C# è necessario utilizzare proprio Visual Studio 2015, perché integra appunto il nuovo compilatore del linguaggio.

Quando fra qualche anno verranno apportate novità al linguaggio in se stesso, nessuna paura, basterà ripartire da qui e studiare solo le nuove caratteristiche (magari acquistando l'edizione aggiornata di questo libro!).

NOTA	Se siete curiosi di conoscere cosa bolle in pentola e quali sono le caratteristiche oggetto di studio e di valutazione per una futura inclusione nelle specifiche del linguaggio C# (e magari se desiderate anche contribuire) potete seguire le pagine e le discussioni ufficiali su GitHub: https://github.com/dotnet/roslyn/wiki.

Versioni e dipendenza

Un componente fondamentale di .NET è il CLR (Common Language Runtime), cioè il suo ambiente di esecuzione, che vedremo più in dettaglio nei prossimi paragrafi.

Ogni versione di .NET Framework consiste quindi di una versione del CLR (oltre che di un insieme di librerie di base), ma il suo numero di versione non è mai andato di pari passo con la versione del framework.

Per esempio il CLR 2.0, una volta introdotto con la versione 2.0 del .NET Framework, è rimasto immutato anche per le versioni 3.0 e 3.5 del framework stesso (non esiste alcun CLR 3.x); ciò perché queste ultime due versioni costituiscono dei rilasci incrementali o aggiornamenti, che hanno aggiunto nuove librerie e funzionalità di alto livello messe a disposizione degli sviluppatori, mantenendo lo stesso ambiente di esecuzione precedente.

.NET Framework 4.0 introduce invece il CLR 4.0, mentre con il rilascio delle versioni seguenti di .NET, 4.5 e successive, il CLR non ha subìto aggiornamenti (vale a dire che il numero di versione principale è sempre 4.0).

NOTA	Il numero di versione di C# non coincide con quello del .NET Framework (vedere la Tabella 1.1), ma ciò non deve sorprendere: C# è un linguaggio, .NET è un framework, mentre il CLR è un suo componente fondamentale; tuttavia non si può negare che la politica di numerazione delle versioni di .NET, del CLR e del linguaggio C# è stata spesso molto confusionaria.

Ogni versione di .NET è in generale compatibile con la precedente, in maniera da poter continuare a eseguire applicazioni scritte per una data versione di .NET anche su ogni versione successiva; quando non è così, le versioni di .NET sono comunque installabili side-by-side.

Installazione *side-by-side* significa che sullo stesso sistema operativo è possibile installare diverse versioni di .NET una affiancata all'altra. In questo modo ogni applicazione verrà eseguita sull'ambiente .NET appropriato e con le librerie adeguate.

Nelle più recenti versioni del sistema operativo Windows, il .NET Framework è inoltre già presente senza necessità di installazioni aggiuntive. Per esempio, su Windows 10 come impostazione predefinita è già presente il .NET Framework 4.6 e 3.5, su Windows 8.1 è presente la versione 4.5.1, mentre il 3.5 può essere abilitato direttamente dal pannello di controllo.

Potete verificare le versioni di .NET installate sul vostro sistema in differenti modi, per esempio esplorando la directory di installazione predefinita, in genere C:\Windows\Microsoft.NET\Framework.

La Figura 1.1 mostra il contenuto della cartella su un sistema con diverse versioni installate fino alla 4.5 (non è un errore, non c'è alcuna cartella 4.5.xxx perché, come già detto, il .NET 4.5 è installato come aggiornamento sul posto).

Framework

File | Home | Condividi | Visualizza

Aggiungi ad Accesso rapido | Copia | Incolla | Taglia | Copia percorso | Incolla collegamento | Appunti

Sposta in | Copia in | Elimina | Rinomina | Organizza

Nuova cartella | Nuovo elemento | Accesso facilitato | Nuovo

Questo PC › Disco locale (C:) › Windows › Microsoft.NET › Framework

Accesso rapido
Desktop
Download
Dropbox
Foto iCloud
Documenti
OneDrive
Immagini

Nome	Ultima modifica	Tipo
1040	31/07/2015 12:34	Cartella di file
URTInstallPath_GAC	23/07/2015 00:17	Cartella di file
v1.0.3705	30/07/2015 17:22	Cartella di file
v1.1.4322	30/07/2015 17:22	Cartella di file
v2.0.50727	15/09/2015 09:54	Cartella di file
v3.0	31/07/2015 12:34	Cartella di file
v3.5	31/07/2015 12:34	Cartella di file
v4.0.30319	17/09/2015 15:32	Cartella di file

Figura 1.1 – Directory di installazione di .NET Framework.

Una modalità più precisa per conoscere le versioni di .NET presenti su un computer, o per stabilire la versione esatta, è quella di interrogare il registro. Se siete utenti abbastanza esperti potete quindi aprire l'editor di registro (regedit.exe) e verificare il valore Release della chiave HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP\v4\Full.

La Figura 1.2 mostra il contenuto di tale chiave su una macchina che esegue Windows 10, e in tale caso il valore Release è pari a 393295 (393297 su altri sistemi operativi). Per verificare invece il numero di versioni del CLR presenti sul sistema, potete lanciare il comando clrver da un prompt dei comandi di Visual Studio (più avanti vedremo anche come aprire tale prompt). Il risultato stampato sulla stessa macchina è il seguente:

Editor del Registro di sistema

File Modifica Visualizza Preferiti ?

- MSOSOAP
- MSSearch36
- MSSQLServer
- MSXML60
- MTF
- Multimedia
- Multivariant
- NET Framework Setup
  - NDP
    - CDF
    - v2.0.50727
    - v3.0
    - v3.5
    - v4
      - Client
      - Full

Nome	Tipo	Dati
(Predefinito)	REG_SZ	(valore non impostato)
CBS	REG_DWORD	0x00000001 (1)
Install	REG_DWORD	0x00000001 (1)
InstallPath	REG_SZ	C:\Windows\Microsoft.NET\Framework64\v4.0.30319\
Release	REG_DWORD	0x0006004f (393295)
Servicing	REG_DWORD	0x00000000 (0)
TargetVersion	REG_SZ	4.0.0
Version	REG_SZ	4.6.00079

Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP\v4\Full

Figura 1.2 – Chiavi del registro di Windows con le versioni di .NET Framework.

C:\Program Files (x86)\Microsoft Visual Studio 14.0> **clrver**

Microsoft (R) .NET CLR Version Tool Version 4.6.81.0
Copyright (c) Microsoft Corporation. All rights reserved.

Versions installed on the machine:
v2.0.50727
v4.0.30319

Quindi sono presenti in questo caso sia il CLR 2.0 che il CLR 4.0.

NOTA	Se siete abbastanza esperti da sapere scrivere e compilare un programma in C#, potete invece utilizzare il metodo RuntimeEnvironment.GetSystemVersion per ottenere la versione del CLR in cui è in esecuzione il programma stesso.

.NET Framework 4.6 è compatibile con le applicazioni precedenti compilate con le versioni di .NET Framework 1.1, 2.0, 3.0, 3.5, 4 e 4.5.1/4.5.2. In altre parole, le applicazioni e i componenti compilati con le versioni precedenti di .NET Framework funzioneranno su .NET Framework 4.6.

NOTA	Non è necessario installare versioni precedenti di .NET Framework o di CLR prima di installare la versione più recente.

La Figura 1.3 mostra una visione d'insieme degli aggiornamenti del CLR, con i vari rilasci del framework e le principali novità introdotte. Al momento della revisione finale di questo libro, la versione più recente del .NET Framework è la 4.6, che introduce vari miglioramenti alla piattaforma, nuove API e correzioni di bug.

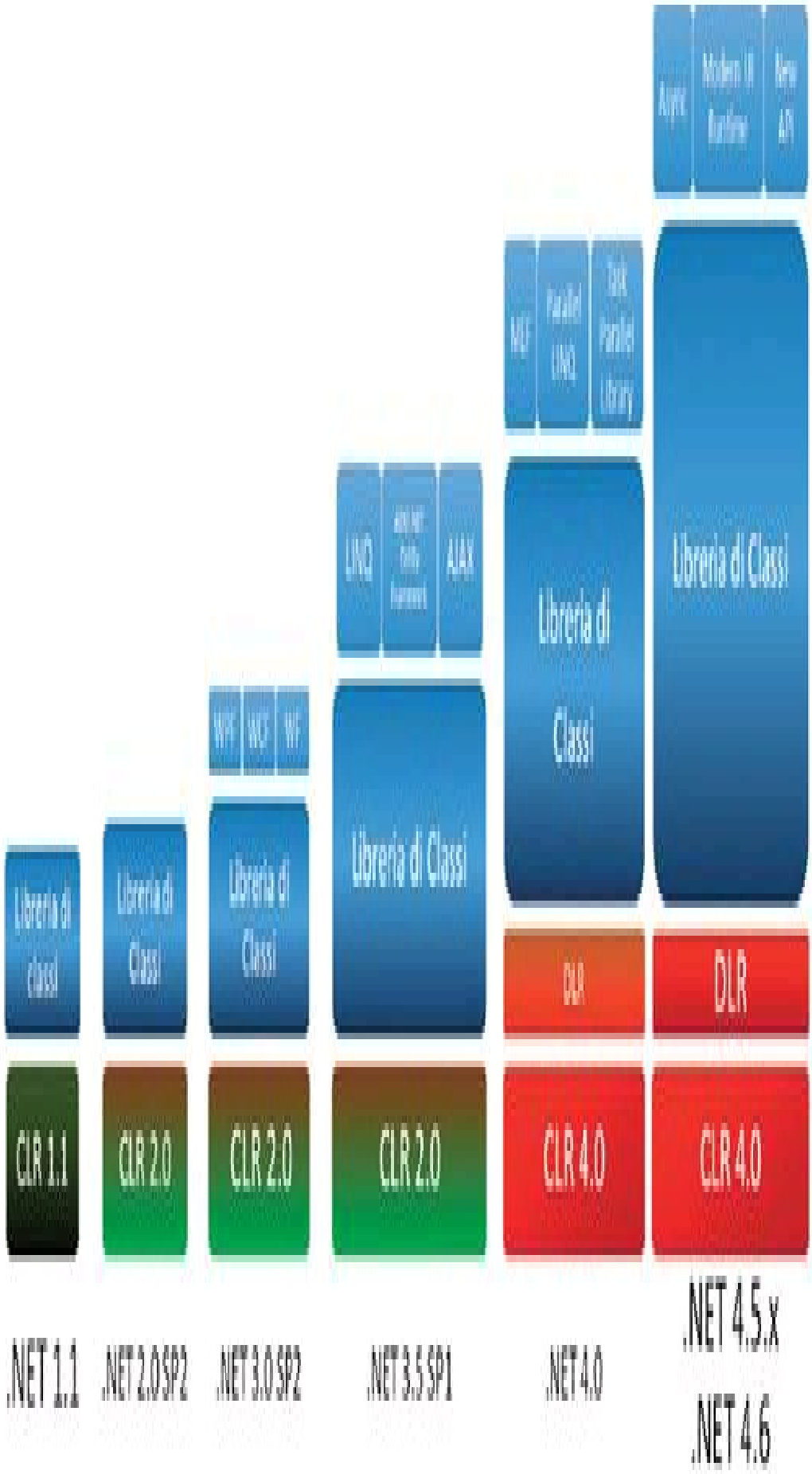

Libreria di Classi
Libreria di Classi
WPF
WCF
WF
Libreria di Classi
LINQ
AJAX
Libreria di Classi
MEF
Parallel LINQ
Task Parallel Library
Libreria di Classi
Async
Modern IO Runtime
Libreria di Classi
DLR
DLR
CLR 1.1
CLR 2.0
CLR 2.0
CLR 2.0
CLR 4.0
CLR 4.0
.NET 1.1
.NET 2.0 SP2
.NET 3.0 SP2
.NET 3.5 SP1
.NET 4.0
.NET 4.5.x
.NET 4.6

Figura 1.3 – Versioni del .NET Framework e novità introdotte.

.NET 2015

.NET è naturalmente un componente strategico per il mondo dello sviluppo software visto da Microsoft. Il 2015 è stato un anno di particolare innovazione, che ha portato una ventata di novità in questo mondo. Secondo questa nuova visione, l'intera piattaforma ha assunto il nome di .NET 2015 ed essa non riguarda solo le tecnologie di sviluppo vere e proprie, ma può essere considerata anche una nuova filosofia.

Il .NET Framework 4.6 fa parte di questa piattaforma, insieme a un nuovo framework denominato .NET Core 5.

La Figura 1.4 riassume quanto è oggi racchiuso sotto il termine .NET 2015.

.NET Framework è ancora naturalmente una parte fondamentale e verrà esaminato in maniera più approfondita nei prossimi paragrafi. Esso non è cross platform e può essere installato solo su Windows (anche se, come già detto, esistono altre implementazioni non Microsoft che permettono di eseguire applicazioni .NET su altri sistemi). Al di sopra del .NET Framework, i suoi utilizzatori sono le applicazioni WPF e Windows Forms per il desktop, e ASP.NET per il Web.

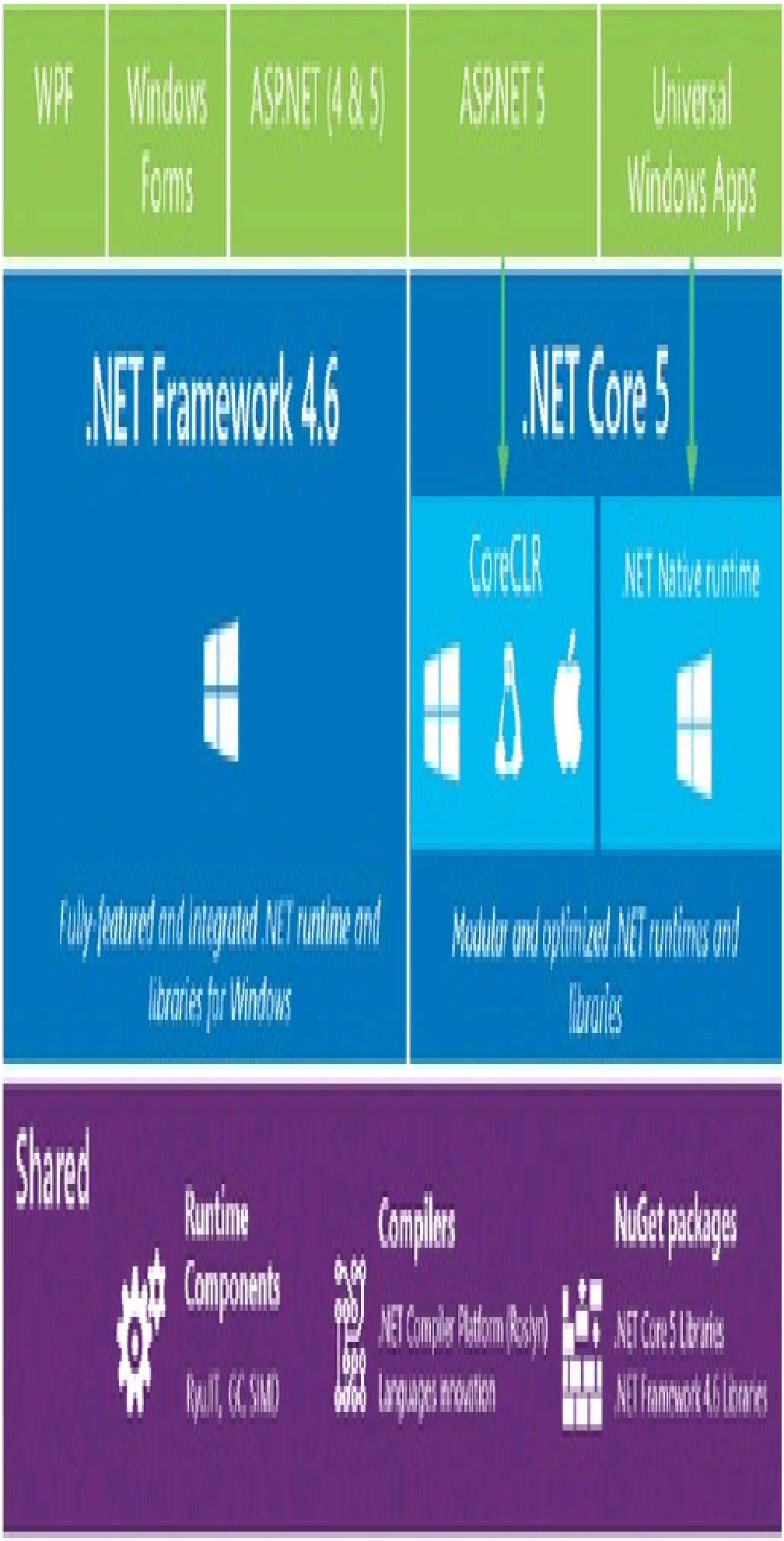

WPF
Windows Forms
ASP.NET (4 & 5)
ASP.NET 5
Universal Windows Apps
.NET Framework 4.6
.NET Core 5
CoreCLR
.NET Native runtime
Fully featured and integrated .NET runtime and libraries for Windows
Modular and optimized .NET runtimes and libraries
Shared
Runtime Components
RyuJIT, CC SIMD
Compilers
.NET Compiler Platform (Roslyn)
Languages innovation
NuGet packages
.NET Core 5 Libraries
.NET Framework 4.6 Libraries

Figura 1.4 – .NET 2015.

.NET Core

.NET Core è una versione modulare del .NET Framework, di cui costituisce le fondamenta, e può essere utilizzato per lo sviluppo di applicazioni in un'ampia varietà di ambiti specifici.

La differenza principale con il fratello maggiore è che esso è completamente costituito da componenti che possono essere installati localmente come pacchetti insieme alle proprie applicazioni, utilizzando solo quelli necessari.

.NET Core consiste di CoreCLR, cioè il runtime o motore di esecuzione e le librerie di base, di CoreFX, cioè le librerie con le classi fondamentali per lo sviluppo, e del compilatore.

Infine, proprio per rendere e mantenere .NET Core un framework cross platform, quindi disponibile su diversi sistemi operativi (in particolare Microsoft supporta Windows, Linux e Mac OSX), esso è open source.

NOTA	I componenti di .NET Core (in particolare di CoreCLR e CoreFX) sono open source su GitHub: le pagine ufficiali e il codice sorgente sono raggiungibili agli url https://github.com/dotnet/coreclr e https://github.com/dotnet/corefx.

Un altro runtime, appositamente pensato per eseguire le applicazioni Universal Windows (vale a dire quelle distribuite tramite Windows Store e che girano sul sistema operativo Windows 10), è denominato .NET Native Runtime. In questo caso le applicazioni sono automaticamente compilate in codice nativo e quindi ottimizzate per un'esecuzione ottimale sul sistema operativo Microsoft.

CoreCLR e .NET Native Runtime possono essere utilizzati direttamente dalle applicazioni, oppure costituire uno strato per degli utilizzatori di più alto livello. Per esempio la Figura 1.4 mostra come ASP.NET 5 utilizzi il CoreCLR (oppure il .NET Framework completo visto prima) mentre le Universal Windows Apps si basano sul .NET Native Runtime.

.NET Compiler Platform

La principale novità di C# 6 può essere considerata l'introduzione di un nuovo compilatore. La .NET Compiler Platform, nota anche con il nome in codice di Roslyn, è la piattaforma open source di compilazione e di strumenti di analisi dedicata ai linguaggi C# e VB.

L'idea alla base di questa piattaforma è quella di consentire l'accesso alle funzioni di compilazione come se fossero un insieme di servizi e non solo una scatola nera a cui dare in pasto del codice e ottenerne un programma eseguibile o una libreria compilata. Secondo

questa filosofia, nota anche come *Compiler as a Service*, il compilatore stesso, o meglio la sua API (Application Programming Interface), potrà essere utilizzata nelle proprie applicazioni, per esempio per creare delle funzioni di analisi, diagnostica, generazione ed elaborazione del codice, come quelle fornite da Visual Studio.

NOTA	La pagina ufficiale e il codice sorgente della .NET Compiler Platform (o Roslyn) si trovano su GitHub al seguente url: https://github.com/dotnet/roslyn/.

Architettura di .NET Framework

Il .NET Framework è una piattaforma complessa e, al tempo stesso, una tecnologia che fornisce tutto il necessario per lo sviluppo e l'esecuzione di applicazioni eterogenee: desktop, mobile, Web, servizi e così via.

Il framework consiste di due componenti principali: il suo ambiente di esecuzione, detto CLR, e la libreria di classi, detta *Framework Class Library*. Il nucleo di questa libreria è un insieme di altre classi di base, detto per questo *Base Class Library* (BCL).

La Figura 1.5 mostra una visione di insieme dell'architettura a livelli di .NET.

Al di sotto del CLR vi è il sistema operativo, che può essere uno qualunque dei sistemi per i quali esiste un'implementazione del CLR stesso.

Il .NET Framework, a sua volta, è utilizzato da ogni applicazione o tecnologia che sfrutta la sua libreria di classi.

In generale infatti le applicazioni sfruttano diverse tecnologie, costituite da librerie di livello ancora più alto, che implementano particolari funzionalità utili in un determinato ambito di sviluppo. Per esempio WPF e Windows Forms forniscono le classi per scrivere applicazioni per il desktop dotate di interfaccia grafica, mentre ASP.NET è la tecnologia per lo sviluppo web.

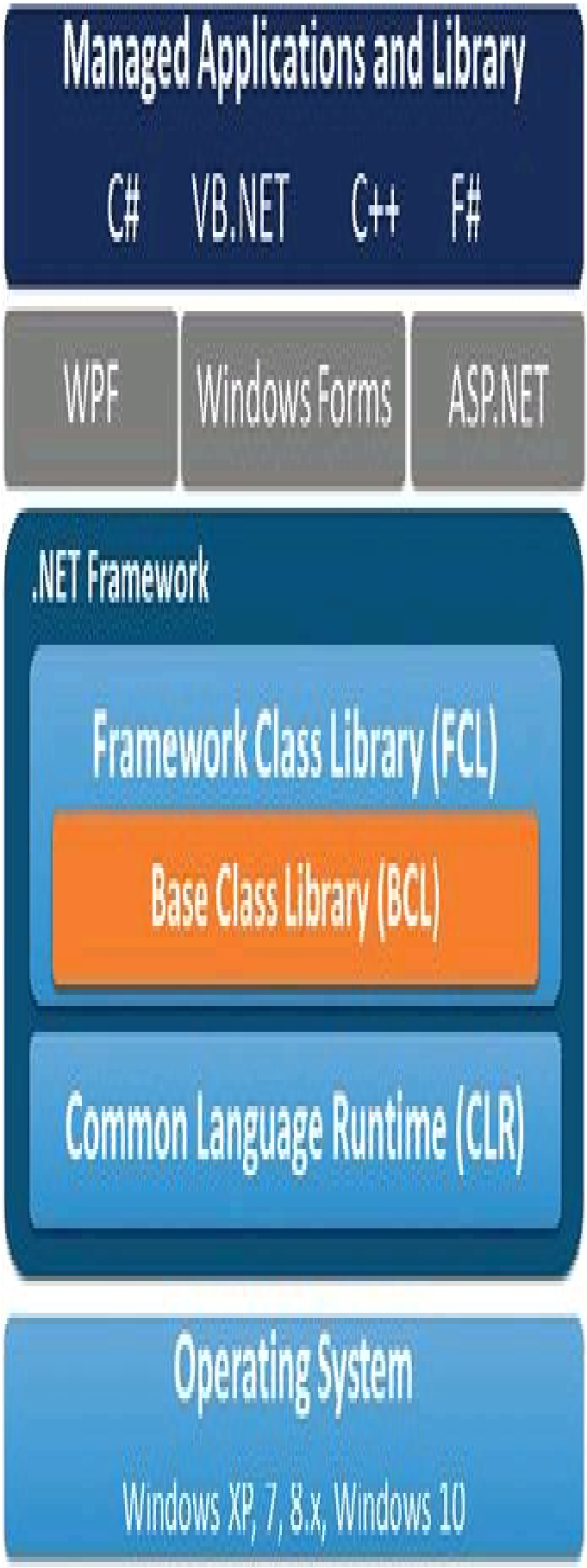

Managed Applications and Library
C# VB.NET C++ F#
WPF
Windows Forms
ASP.NET
.NET Framework
Framework Class Library (FCL)
Base Class Library (BCL)
Common Language Runtime (CLR)
Operating System
Windows XP, 7, 8.x, Windows 10

Figura 1.5 – Architettura a livelli del .NET Framework.

Per la scrittura del codice sorgente delle applicazioni può essere utilizzato uno qualunque dei linguaggi cosiddetti *.NET Enabled* (nella Figura 1.5 sono elencati solo i linguaggi supportati in maniera predefinita da Visual Studio 2015).

Common Language Runtime

Il componente fondamentale di .NET è il suo ambiente di esecuzione, il cosiddetto Common Language Runtime (CLR), che costituisce una sorta di macchina virtuale all'interno della quale i programmi scritti in C# (o in uno dei linguaggi .NET Enabled, cioè supportati dalla piattaforma) vengono eseguiti.

Il codice eseguito dal CLR viene detto *managed code*, o *codice gestito* (dal CLR appunto). Al contrario, il codice che non passa dal CLR viene detto *unmanaged code*, cioè *codice non gestito*, con il quale si intende dunque tutto il codice macchina, per esempio il codice nativo scritto sfruttando le API Win32 di Windows.

Compilazione ed esecuzione

Un programma scritto in C# prima di poter essere eseguito deve essere convertito in un linguaggio intermedio, chiamato CIL (*Common Intermediate Language*) o IL (*Intermediate Language*), indipendente dalla CPU e dal linguaggio (anche Visual Basic verrà convertito nello stesso IL), che è comprensibile dal CLR.

L'Intermediate Language è una sorta di linguaggio macchina, ma di livello molto più alto, in quanto possiede anche caratteristiche di linguaggio orientato agli oggetti, funzioni per l'utilizzo di array e per la gestione degli errori.

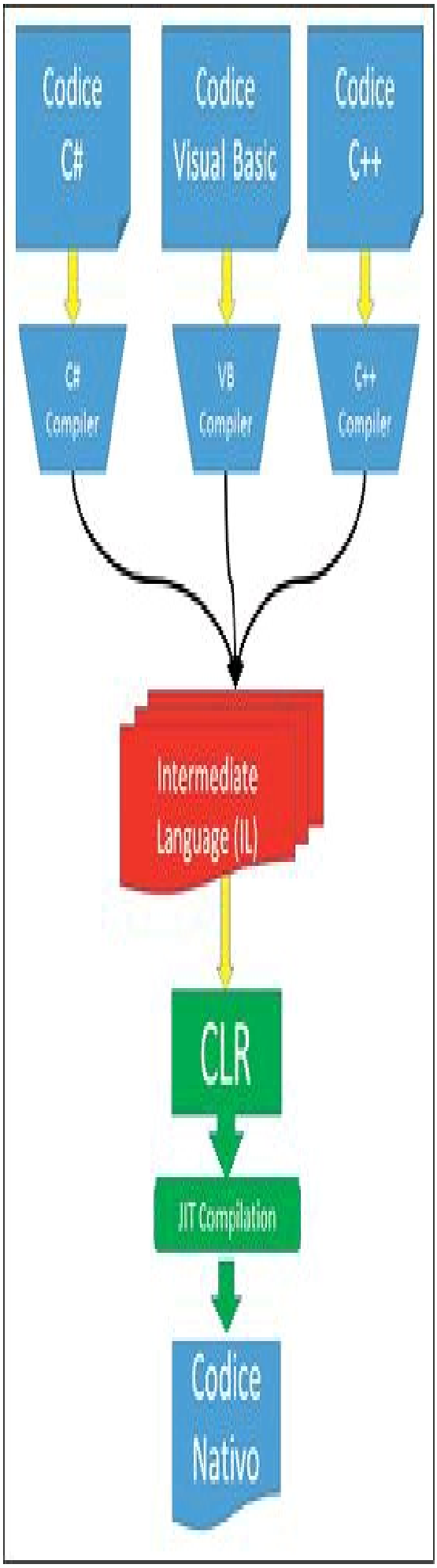

Codice
C#
Codice
Visual Basic
Codice
C++
C#
Compiler
VB
Compiler
C++
Compiler
Intermediate
Language (IL)
CLR
JIT Compilation
Codice
Nativo

Figura 1.6 – Compilazione ed esecuzione in .NET.

Il CLR non ha idea di quale linguaggio sia stato utilizzato per produrre il codice intermedio IL, e non gli interessa nemmeno. Il codice IL, però, non è ancora eseguibile direttamente dal sistema operativo.

Quindi, in un secondo processo di compilazione, il programma, ora in codice IL, viene trasformato nel linguaggio macchina specifico della piattaforma su cui esegue il CLR stesso. Tale processo viene chiamato *compilazione JIT* (*Just-In-Time*), ed è eseguita dal *JITCompiler* o *Jitter*. Il Jitter si occupa di produrre il codice specifico della piattaforma: per esempio se esso è in esecuzione su una versione x86 di Windows esso produrrà istruzioni x86. A questo punto il sistema operativo sarà in grado di eseguire l'applicazione.

La modalità di compilazione *Just-in-Time* permette di ottenere performance superiori rispetto alla modalità di esecuzione di un linguaggio interpretato.

Il compito del CLR però non termina qui: esso gestirà l'esecuzione delle applicazioni in una sorta di macchina virtuale, occupandosi di funzioni fondamentali come la gestione della memoria e degli eventuali errori durante l'esecuzione, la sicurezza e l'interoperabilità.

Sarebbe possibile anche scrivere un programma direttamente in codice IL e poi darlo in pasto al JIT Compiler, ma linguaggi di alto livello come C# hanno proprio il compito di rendere più semplice lo sviluppo.

I vantaggi del codice IL sono l'interoperabilità fra i linguaggi .NET e la possibilità di essere eseguito su diverse piattaforme, in quanto basterà avere un'implementazione specifica del CLR che converta lo stesso IL (uguale per tutte le piattaforme) nel codice macchina nativo della piattaforma su cui è in esecuzione.

Microsoft fornisce con la sua implementazione della piattaforma .NET anche un assembler di linguaggio IL, chiamato ILAsm, e il corrispondente disassemblatore, ILDasm, che permette di ottenere il codice IL a partire da un programma eseguibile.

Assembly

Quando un programma C# viene compilato, il codice IL ottenuto viene conservato all'interno di uno o più file detti *assembly*.

Un assembly è un'unità logica che può essere direttamente eseguita dal sistema operativo (file .exe) oppure che può essere utilizzata da altri programmi come libreria di codice (file .dll). La struttura logica è identica in entrambi i casi, l'unica differenza è che un assembly eseguibile contiene anche un cosiddetto entry point, cioè un punto di ingresso che indica al CLR da dove iniziare l'esecuzione dell'applicazione.

Nonostante le estensioni utilizzate siano ancora .exe e .dll, il contenuto dei rispettivi file è differente dal formato nativo dei file eseguibili e delle librerie di Windows.

Ogni assembly, infatti, contiene oltre al codice intermedio anche un *manifest* che descrive l'assembly stesso, dei *metadati* che descrivono i tipi contenuti dell'assembly e delle *risorse* opzionali (per esempio immagini, audio ecc.).

Grazie a tali metadati ogni assembly è completamente autodescrittivo e non sono necessarie altre informazioni esterne per poterlo utilizzare. In parole povere sarà sufficiente copiare uno o più assembly che costituiscono un'applicazione su un computer per poterla eseguire (naturalmente sarà necessaria la presenza del .NET Framework su tale computer).


Assembly Manifest
Metadati dei tipi
Codice IL
Risorse (immagini, ecc.)

Figura 1.7 – Formato e contenuto di un assembly .NET.

La Figura 1.7 mostra il formato di un assembly .NET e il suo contenuto.

In realtà il contenuto di un assembly può anche essere suddiviso in diversi file, per esempio separando le risorse in un altro assembly, in maniera da poter eventualmente sostituire solo la parte con il codice IL, senza dover necessariamente ridistribuire un corposo insieme di altre risorse non modificate.

DLL Hell

Il concetto di assembly ha cercato di mettere fine al cosiddetto problema dell'inferno delle DLL (*DLL Hell*), rendendo anche notevolmente più semplice la distribuzione delle applicazioni.

Prima dell'arrivo di .NET, infatti, il problema di versionamento delle DLL e della loro condivisione provocava spesso malfunzionamenti nelle applicazioni installate, dovuti, per esempio, all'installazione di nuove versioni della stessa applicazione o, peggio ancora, di altre applicazioni che utilizzavano la stessa DLL in una versione differente.

Si immagini, per esempio, un'applicazione A che utilizza la DLL versione 1.0; successivamente viene installata l'applicazione B che utilizza la stessa DLL ma in versione 1.1 e sovrascrive la precedente. A questo punto A smette di funzionare per qualche strano motivo (probabilmente utilizzava una funzione che è stata modificata nella versione 1.1). Per sistemare le cose l'utente reinstalla l'applicazione A ripristinando la versione DLL 1.0. La conseguenza è che smette di funzionare l'applicazione B che richiede la DLL 1.1.

Immaginate ora questo scenario per decine di applicazioni: ecco il cosiddetto inferno delle DLL.

Uno degli obiettivi progettuali del team del .NET Framework era quello di trovare una soluzione definitiva alla problematica descritta. Per farlo .NET fornisce le seguenti funzionalità e caratteristiche:

- le applicazioni sono auto-descrittive, rimuovono la dipendenza dal registro, non impattano sul funzionamento di altre applicazioni installate e consentono una semplice e indolore disinstallazione;
- le informazioni relative alle versioni devono essere registrate e ricordate, in maniera che ogni applicazione conosca per esempio quale insieme di assembly e quale versione deve utilizzare;
- supporto per consentire l'esistenza di molteplici versioni dello stesso assembly side-by-side (cioè fianco a fianco) in maniera che ogni applicazione possa utilizzare la versione

più appropriata. Il .NET Framework porta il concetto di installazione side-by-side ancora più avanti, consentendo l'installazione di più versioni di se stesso sullo stesso sistema, in maniera che più applicazioni che richiedono versioni diverse di .NET siano comunque eseguibili;
- isolamento delle applicazioni cioè la possibilità di rendere possibile l'esecuzione delle applicazioni in maniera indipendente dal sistema operativo su cui sono installate; per esempio è possibile distribuire tutti gli assembly richiesti da un'applicazione nella stessa directory e quindi senza complesse procedure di installazione e registrazione di componenti e DLL. Allo stesso tempo l'installazione o disinstallazione di altre applicazioni non influisce su tale applicazione isolata.

La Framework Class Library

Il .NET Framework fornisce una gigantesca libreria di assembly, contenente migliaia di oggetti (chiamati tipi) utilizzabili all'interno delle proprie applicazioni. Tale libreria è chiamata Framework Class Library (FCL).

È praticamente impossibile conoscere tutti i tipi messi a disposizione ed è anzi compito del programmatore imparare a cercare al suo interno le funzionalità di cui ha bisogno nello sviluppo di ogni particolare applicazione.

La libreria mette infatti a disposizione tipi per diverse tipologie di applicazioni (desktop, Web, web service, mobile e così via) e per diversi ambiti applicativi (input/output, database, XML ecc.), suddividendoli in una struttura gerarchica mediante il concetto di spazio dei nomi o namespace (vedere il Capitolo 2).

Un sottoinsieme contenente i tipi di base e non correlato ad alcuna particolare tecnologia è detto anche Base Class Library (BCL).

Un *tipo* è una rappresentazione di dati, quindi la BCL contiene, fra gli altri, i tipi che rappresentano dati fondamentali come i numeri.

La libreria è utilizzabile naturalmente da qualsiasi linguaggio .NET visto che, come abbiamo già accennato, gli assembly sono prodotti a partire da uno stesso linguaggio intermedio IL.

In realtà, a partire dalla versione 3.0 di .NET, è stata modificata la modalità di distribuzione e di raggruppamento dei tipi messi a disposizione dal framework.

Tale versione infatti ha introdotto nuovi insiemi di tipi, raggruppandoli in nuove librerie, ognuna dedicata a una particolare area di sviluppo:

- WPF (*Windows Presentation Foundation*) – contiene tipi per lo sviluppo di interfacce grafiche di nuova generazione, con un nuovo paradigma di sviluppo e disegno e un nuovo

linguaggio di definizione dell'interfaccia, chiamato *XAML*;

- WCF (*Windows Communication Foundation*) – un insieme di tipi per realizzare servizi web e applicazioni che si scambiano dati in rete in maniera indipendente dal protocollo (http, tcp ecc.);
- WF (*Workflow Foundation*) – un framework per lo sviluppo di applicazioni basate su flussi di attività.

Con la versione 4.6 i tipi contenuti nella Framework Class Library e utilizzabili dagli sviluppatori sono circa quindicimila!

Common Type System

Data l'importanza fondamentale dei tipi all'interno della piattaforma .NET, Microsoft ha creato una specifica formale, detta Common Type System (CTS), nella quale viene descritto come sono definiti i tipi e come funzionano.

Tale specifica è comune ed è condivisa fra i vari linguaggi di programmazione supportati da .NET, quindi permette la piena interoperabilità fra applicazioni e librerie scritte in linguaggi diversi.

Prima dell'era .NET un problema abbastanza comune nello sviluppo di applicazioni scritte in tecnologie o linguaggi differenti, in particolare allo scopo di ottenerne l'interoperabilità, era lo scambio di dati.

Basta pensare che un tipo che rappresenta una sequenza di caratteri di testo, la stringa, era rappresentato e chiamato in maniera differente in quasi ognuno dei linguaggi utilizzati: char* in C, CString in MFC, BSTR in COM.

In .NET il tipo stringa è implementato all'interno della libreria di base, è denominato System.String, ed è lo stesso per ogni linguaggio, sia esso C#, Visual Basic, C++ o un altro linguaggio .NET.

Un altro esempio è quello del tipo che definisce i numeri interi che, per esempio, in alcuni linguaggi è dipendente dalla piattaforma e dal compilatore, quindi utilizzerà 16, 32, 64 bit e così via.

In .NET invece il tipo int ha sempre 32 bit, su qualunque sistema operativo, in qualunque linguaggio.

Per ottenere questo obiettivo il CTS stabilisce le regole del gioco sulla creazione dei tipi e la prima di queste regole è che ogni tipo ha una madre comune dalla quale eredita varie caratteristiche. La madre di tutti i tipi è la classe System.Object.

Common Language Infrastructure

Come parte della strategia di Microsoft, la Common Language Infrastructure (CLI) è una specifica che consente a un programma scritto in un qualsiasi linguaggio supportato da .NET di essere eseguito su un qualunque sistema operativo, utilizzando un ambiente di esecuzione comune (cioè il già citato CLR).

CLI è stato approvato come standard dall'ECMA (ECMA-335) e comprende e specifica:

- un linguaggio comune (CLS, Common Language Specification) e le regole per le interoperabilità fra linguaggi;
- un insieme comune di tipi (CTS, Common Type System);
- una serie di informazioni sulla struttura di un programma, indipendenti dal linguaggio, per permettere a programmi scritti in linguaggi diversi di comunicare fra loro;
- un ambiente di esecuzione virtuale comune, che esegue i programmi.

Si faccia attenzione al fatto che il CLI non è un componente o un'implementazione, ma solo una specifica, che può essere implementata in modi diversi.

Il .NET Framework è l'implementazione proprietaria di Microsoft della specifica CLI, che comprende il CLR e un'ampia collezione di librerie, risorse e strumenti di sviluppo.

.NET Core è un'implementazione open source che implementa lo standard ECMA 335.

Il progetto Mono è invece un'iniziativa promossa e sponsorizzata da Novell per creare una versione open source per i sistemi UNIX della piattaforma .NET.

Strumenti di programmazione

Per scrivere un qualunque programma in C# è necessaria una dotazione minima costituita da un editor di testo, con il quale si scriverà e salverà il cosiddetto codice sorgente.

Per trasformare poi il codice sorgente in un programma eseguibile sarà necessario darlo in pasto a un compilatore del linguaggio, nel nostro caso un compilatore C#.

Il compilatore standard fornito insieme al .NET Framework è chiamato *C# compiler* ed è utilizzabile dal prompt dei comandi lanciando l'eseguibile csc.exe (vedere Paragrafo "Developer Command Prompt" più avanti).

Così però la produttività non sarebbe di certo il massimo e quindi, per costruire applicazioni complesse in maniera molto più efficiente, sarà necessario abbandonare il blocco note e il prompt dei comandi e utilizzare appositi ambienti di sviluppo integrato, detti anche IDE (Integrated Development Environment), il cui rappresentante più noto nel mondo .NET è senza dubbio *Visual Studio* (del quale abbiamo già avuto modo di elencare le varie versioni, rilasciate in parallelo a quelle di .NET Framework e C#).

Nel prossimo capitolo si vedrà come compilare i primi esempi di codice utilizzando il compilatore a riga di comando csc.exe, in maniera da comprendere più a fondo il processo che porta a ottenere degli assembly .NET, ma nel resto del libro utilizzeremo quasi esclusivamente Visual Studio come ambiente di sviluppo e compilazione.

Essendo però Visual Studio uno strumento fin troppo grande e potente, soprattutto nella prima parte del libro e per chi è alle prime armi, il suo utilizzo sarebbe paragonabile allo sparare a una mosca con un cannone; infatti, fra le altre possibilità rapide per testare delle semplici istruzioni o programmi C#, vi è anche quella di utilizzare qualche semplice programmino che ci eviti sia di dover salvare un file di testo per poi darlo in pasto a csc.exe, sia di dover creare appunto un progetto Visual Studio.

Fra questi programmi, vi suggerisco *Linqpad*, disponibile anche in una versione gratuita sul sito www.linqpad.net, che supporta anche C# 6 e permette di eseguire programmi o istruzioni e osservarne rapidamente il risultato.

Un'altra possibilità, che sfrutta la nuova piattaforma di compilazione ed è disponibile con l'Update 1 di Visual Studio 2015, è quella di utilizzare la finestra C# Interactive, attivabile dal menu View -> Other Windows. Questo strumento è un editor REPL (read-eval-print-loop), all'interno del quale è possibile scrivere una qualunque espressione o istruzione C# ed

eseguirla premendo Invio. Ulteriori informazioni e documentazioni sono disponibili a questo link, https://github.com/dotnet/roslyn/wiki/Interactive-Window.

NOTA	L'editor interattivo della finestra C# Interactive è disponibile anche da linea di comando, fuori da Visual Studio. Basta aprire un prompt dei comandi di sviluppo (vedi più avanti nel capitolo, "Developer Command Prompt") e digitare il comando csi (C# interactive)" (CSharp Interactive) per iniziare una nuova sessione.

.NET Framework SDK

Per poter sviluppare in C# o sulla piattaforma .NET in generale non è necessario acquistare Visual Studio o altre licenze Microsoft, come molti penseranno.

Il .NET Framework SDK (Software Development Kit) è incluso all'interno del Windows SDK, contiene tutto ciò che è necessario per lo sviluppo in .NET, ed è scaricabile gratuitamente insieme a decine di altre risorse, librerie, componenti e strumenti dal sito ufficiale http://msdn.microsoft.com/netframework.

Installando Visual Studio 2015, comunque, non sarà necessario nient'altro per poter sviluppare in C#, in quanto la procedura di setup dell'ambiente di sviluppo provvederà anche a installare l'SDK del .NET Framework.

Visual Studio 2015

Per sviluppare in maniera professionale ed efficiente un IDE come Microsoft Visual Studio costituisce una scelta quasi obbligata (anche se non l'unica).

Oltre alle varie versioni a pagamento, Visual Studio 2015 è disponibile anche nella versione Community, che è gratuita per singoli sviluppatori o piccoli team.

Per scrivere gli esempi di questo libro è stato utilizzato Visual Studio Professional 2015, ma se non avete la possibilità di acquistarlo, potete tranquillamente installare Visual Studio 2015 Community per iniziare a sviluppare in C#, ma anche per poter sviluppare applicazioni per il desktop, per il Web, per servizi cloud, per le Universal App di Windows 10, per Windows Phone, o anche per Android e iOS.

Tutto quello che imparerete a eseguire all'interno di Visual Studio 2015 Community sarà utilizzabile anche all'interno delle versioni più avanzate.

Nel seguito del libro quindi, quando ci riferiremo all'ambiente Visual Studio 2015 Community, lo chiameremo semplicemente Visual Studio 2015 o anche solo Visual Studio.

Visual Studio 2015 supporta lo sviluppo di applicazioni in uno dei linguaggi C#, Visual Basic, C++ e F#.

Nel prossimo capitolo verrà fornita un'ampia panoramica dell'ambiente di sviluppo Visual Studio 2015, a partire dalla composizione della sua interfaccia grafica, e saranno esposte le varie funzionalità messe a disposizione dello sviluppatore.

Developer Command Prompt

Dopo aver installato il .NET Framework SDK, oppure Visual Studio 2015, anche in versione Community, per aprire un prompt dei comandi pronto alla compilazione, cioè con la variabile di ambiente PATH contenente tutti i percorsi necessari, basta avviare il cosiddetto Developer Command Prompt.

Su Windows 7 è avviabile, in genere, dal menu Start -> Programmi -> Microsoft Visual Studio 201x -> Visual Studio Tools, mentre su Windows 8.x o Windows 10 basta avviare la ricerca (Tasto Windows+Q) e digitare Developer Command Prompt for VS201x.

Nel prossimo capitolo si tornerà sull'argomento, fornendo qualche dettaglio in più.

Da un prompt dei comandi per sviluppatori così avviato è possibile lanciare la compilazione di un programma C# semplicemente digitando il comando csc seguito dal percorso (o solo il nome se ci si trova già nella cartella) del file contenente il programma da compilare (csc è l'acronimo di *CSharp Compiler*).

Se volete conoscere le opzioni del compilatore basta digitare il seguente comando nello stesso prompt:

```
csc -?
```

Se tutto va bene e se il .NET Framework SDK è installato correttamente, la prima riga stampata dopo aver premuto il tasto Invio riporterà delle informazioni sulla versione del compilatore, per esempio:

```
Compilatore Microsoft (R) Visual C# versione 1.0.0.50618
```

Nel prossimo capitolo si utilizzerà il compilatore csc per creare i primi esempi di programmi in C#.

Riepilogo

Per riuscire a diventare sviluppatori esperti di un linguaggio basato su .NET come C# è praticamente un obbligo capire l'architettura e il funzionamento del .NET Framework.

In questo primo capitolo quindi è stata esposta una panoramica di .NET e dei suoi componenti, come il CLR e la Framework Class Library, mostrando anche i concetti di base della compilazione ed esecuzione dei programmi scritti in C#. Infine abbiamo elencato gli strumenti di programmazione che utilizzeremo nel corso del libro e in ogni attività di sviluppo che dovremo affrontare come sviluppatori C#. Nei prossimi capitoli, alla fine di ognuno troverete delle domande di riepilogo, per mettervi alla prova e misurare il vostro grado di comprensione.

Concetti di base di C#

In questo capitolo si vedranno gli elementi fondamentali della sintassi del linguaggio C#, le operazioni necessarie per creare programmi eseguibili utilizzando il compilatore C# a riga di comando e l'ambiente di sviluppo Visual Studio.

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 operazioni e calcoli da svolgere, magari basandosi anche su istruzioni inserite 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 sono le differenze con il vostro linguaggio di programmazione di provenienza.

Per iniziare a mettere subito le mani in pasta, obiettivo di questo capitolo è quello di utilizzare il compilatore di C# a riga di comando, richiamabile digitando il nome dell'eseguibile *csc.exe*, al fine di compilare dei programmi scritti con un qualsiasi editor di testo; anche il blocco note di Windows va bene a tale scopo, ma se avete un vostro editor di testo preferito (io per esempio utilizzo Notepad++) che faccia anche un minimo di syntax highlighting, cioè di evidenziazione delle parole chiave o degli altri elementi del linguaggio, ben venga: sarà più semplice comprendere la struttura e la correttezza del codice scritto.

Avendo però a disposizione un ambiente di sviluppo integrato come Visual Studio, il compito del programmatore risulterà notevolmente facilitato e la sua produttività aumenterà esponenzialmente.

Nei prossimi paragrafi si scriverà il primo programma in C#, spiegando riga per riga e istruzione per istruzione il suo funzionamento, per poi iniziare a introdurre i vari elementi del linguaggio C#, in maniera da mettere da subito in grado di creare semplici programmi anche il lettore alle prime armi.

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 e il modo di correggerli. Per questo motivo inizieremo la nostra avventura nel meraviglioso mondo di C#, con originalità e fantasia, scrivendo e compilando il classico Hello World, cioè un programma che stampi su un prompt dei comandi queste due parole.

Prima di spiegarvene il significato, copiate le seguenti righe di codice all'interno di un file di testo vuoto, con il vostro editor preferito:

```
using System;

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

Questo testo costituisce il codice sorgente, in linguaggio 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".

Salvate il file assegnandogli un nome a vostra scelta, ma con 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 più comodo dare un nome che corrisponda al contenuto e assegnare l'estensione .cs per riconoscere i file sorgenti C#.

Per rendere eseguibile un programma, come quello appena scritto, bisogna procedere alla sua compilazione; in questo primo esempio utilizzeremo il compilatore a riga di comando csc.

Se avete installato Visual Studio 2015 (come si vedrà più avanti in questo capitolo) o anche una versione precedente, è possibile avviare in Windows un prompt dei comandi di Visual Studio. In base al vostro sistema operativo basta avviarlo utilizzando l'apposita procedura: su Windows 7 per esempio basta avviarlo dal menu Start all'interno del gruppo Visual Studio 2015.

Su Windows 8.x e 10, invece, è possibile anche ricercare mediante l'apposita funzione di ricerca (scorciatoia da tastiera Windows+Q) il comando Developer Command Prompt for VS2015.

Il comando csc è disponibile inoltre all'interno di un normale prompt dei comandi di Windows (avviabile mediante la scorciatoia da tastiera Windows+R, digitando cmd e premendo Invio) solo se sono state impostate delle apposite variabili di ambiente, in particolare aggiungendo alla variabile PATH il percorso in cui si trova l'eseguibile csc.exe.

Le variabili di ambiente si impostano in Windows dal Pannello di controllo, in cui aprire Sistema, e quindi facendo clic su Impostazioni di Sistema Avanzate. A questo punto nella finestra delle proprietà, nella sezione Avanzate, fare clic sul pulsante Variabili d'ambiente. Non resta che aggiungere alla variabile d'ambiente PATH, visualizzata nell'elenco delle variabili di sistema, il percorso dell'eseguibile csc.exe.

Il percorso predefinito del compilatore csc.exe, per Visual Studio 2015, e quindi per C# 6, è:

C:\Program Files (x86)\MSBuild\14.0\bin\csc.exe

Per impostare più facilmente le variabili di ambiente necessarie, è possibile eseguire il file batch vsvars32.bat, che si trova all'interno della sotto-directory Tools, presente nella cartella di installazione di Visual Studio 2015; se avete installato l'IDE all'interno della directory predefinita troverete il file suddetto nella cartella:

C:\Program Files (x86)\Microsoft Visual Studio 14.0\Common7\Tools

Se non avete installato invece Visual Studio, ma lavorate su un computer dotato solo del Windows Software Development Kit (SDK), potete utilizzare il compilatore C# lanciandolo dal prompt dei comandi dell'SDK stesso, che si trova all'interno del menu Microsoft .NET Framework SDK. Ripetiamo comunque che è consigliabile dotarsi da subito di Visual Studio 2015, anche nella versione Community.

A questo punto, per verificare se il compilatore funziona, provate a digitare il comando csc e premete Invio.

Se il comando viene riconosciuto verranno visualizzate alcune righe con le informazioni sulla versione del compilatore, come le seguenti (naturalmente l'output esatto dipende dalla

versione installata):

```
C:\Program Files (x86)\Microsoft Visual Studio 14.0>csc
Compilatore Microsoft (R) Visual C# versione 1.0.0.50618
Copyright (C) Microsoft Corporation. Tutti i diritti riservati.
```

In questo caso, la versione 1.0.0.50618 corrisponde al compilatore C# della .NET Compiler Platform, noto anche con il nome in codice Roslyn.

Ora basta posizionarsi nella directory in cui avete salvato in precedenza il file helloworld.cs e lanciare il comando di compilazione, che nel formato più semplice permette di compilare il file sorgente utilizzando come unico parametro il suo nome:

```
C:\> csc helloworld.cs
```

Se non ci sono errori nel codice sorgente, e non dovrebbero essercene se avete copiato correttamente le istruzioni presenti nell'esempio precedente, la compilazione andrà a buon fine in pochi secondi, generando nella stessa directory un file eseguibile (o più esattamente un *assembly* eseguibile di .NET) con lo stesso nome del file .cs, ma con estensione .exe.

Per eseguire il primo programma C# appena compilato basta digitare nello stesso prompt dei comandi il suo nome (anche senza specificare l'estensione .exe), come un qualsiasi altro eseguibile:

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

Il risultato sarà in questo caso, e come ci si aspettava, semplicemente la visualizzazione della frase "Hello World" all'interno della finestra del prompt.

Per la distribuzione ed esecuzione delle applicazioni .NET basta copiare i file prodotti dalla compilazione (ed eventuali librerie necessarie all'esecuzione) in un qualsiasi computer in cui sia presente il .NET CLR. Nelle ultime versioni di Windows lo trovate già installato e potete facilmente distribuire i vostri programmi copiandoli in una qualunque cartella.

Anatomia di un'applicazione

Dopo essere riusciti a compilare il primo programma C#, si passerà ora a esaminarlo, per capirne, riga dopo riga e istruzione dopo istruzione, il 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 anche in altri linguaggi) 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.

C# è un linguaggio di programmazione *orientato agli oggetti* e il cuore del paradigma di programmazione object oriented è il concetto di classe.

Per comodità di organizzazione del codice, più classi possono essere organizzate in dei 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.

Dato che C# è un linguaggio orientato agli oggetti, ogni programma deve contenere almeno una classe, la quale a sua volta contiene dei membri: ecco spiegato perché un pur semplice programma Hello World, che pretende semplicemente di stampare una stringa sullo schermo, sia costituito da tutte quelle istruzioni.

Tornando proprio al programma di esempio Hello World, esso inizia con la seguente istruzione:

```
using System;
```

Essa indica che vogliamo utilizzare nel nostro programma degli elementi contenuti nel namespace denominato System, che fa parte della libreria di base di .NET e che quindi è uno dei più utilizzati. L'istruzione using System sarà quindi praticamente onnipresente in tutti i programmi che scriverete e incontrerete nella vostra vita di programmatori C#.

Un *namespace* è in parole povere un contenitore di tipi.

Nel nostro caso particolare abbiamo utilizzato l'istruzione using, perché all'interno del namespace System è presente la classe Console, che verrà utilizzata qualche riga dopo.

NOTA	La parola chiave using può assumere significati diversi all'interno di un programma C#, a seconda del contesto in cui viene utilizzata. Più avanti nel libro si vedranno gli altri possibili

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 invece 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 nostro 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 si preoccupi se il termine classe è oscuro, ci torneremo su molto presto!).

Per creare una classe si utilizza la parola class seguita dal nome che si vuole assegnare a essa:

```
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. All'interno della classe viene definito un unico metodo denominato Main. Ogni programma eseguibile C# deve contenere un metodo Main, che costituirà il punto di ingresso dal quale il programma stesso inizierà la sua esecuzione:

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

Tralasciamo per ora le altre parole e simboli presenti prima e dopo Main e passiamo al corpo dello stesso metodo, 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, che tornerà utile a chi leggerà il codice per comprenderne il significato e il funzionamento, ma che verrà ignorato dal compilatore.

Dopo il commento, sempre all'interno del metodo Main, vi è una sola istruzione, chiusa dal punto e virgola a indicarne 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 le parentesi:

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

Dopo questa istruzione vi sono solo delle parentesi graffe chiuse, che servono a 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 e 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) è chiamato *applicazione eseguibile*.

Quando viene eseguita un'applicazione .NET, viene creato un nuovo application domain.

In tal modo una stessa applicazione può essere eseguita contemporaneamente in più copie nello stesso sistema, ognuna all'interno del proprio dominio applicativo.

Un application domain consente così di isolare un'applicazione, mantenendone lo stato. I tipi caricati all'interno del dominio di un'applicazione, così come gli oggetti che verranno creati durante il suo ciclo di vita, sono totalmente distinti da quelli creati da una seconda copia della stessa applicazione in un altro application domain.

L'avvio di un'applicazione avviene quando l'ambiente di esecuzione, cioè il CLR di .NET, invoca il metodo designato come punto di ingresso (cioè il metodo Main, descritto 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 essere compilato sotto forma di applicazione eseguibile, deve contenere un punto di ingresso (o entry point), che è costituito da un metodo denominato Main.

C# è un linguaggio *case sensitive*, cioè che fa distinzione fra lettere minuscole e maiuscole, quindi il metodo Main deve essere scritto proprio così, con l'iniziale M in maiuscolo.

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.

Ci sono 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) e debba restituire al termine dell'esecuzione un codice di uscita.

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 è utilizzabile 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 CLR, quindi in generale esso viene scritto senza essere preceduto da public.

La parola static è invece obbligatoria e deve essere sempre indicata per il 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 metodo può restituire o meno il risultato della propria esecuzione al chiamante. 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):

```
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 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 mediante il prompt dei comandi.

La prima forma è quella più semplice, in quanto non restituisce nulla all'esterno e non prende 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. Vedremo nel seguito come trattare e utilizzare questi parametri di ingresso.

Negli ultimi due casi il metodo Main deve restituire all'esterno un codice numerico (numerico 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. Ecco un esempio di metodo Main che restituisce il valore intero 0:

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

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, mediante l'opzione /main del compilatore csc. 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");
}
}
}
```

Per compilare un eseguibile che usi il metodo Main di Program dovremo quindi utilizzare il comando:

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

Compilando infatti normalmente e senza specificare l'opzione main un programma con più metodi Main, il compilatore restituirebbe un errore, indicando che sono stati specificati due o più punti di ingresso e che è necessario utilizzare l'opzione main.

Si noti che per compilare correttamente abbiamo dovuto anche indicare il nome completo della classe Program, cioè MainMultipli.Program, ottenuto indicando 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 quella a riga di comando e così via.

NOTA	Anche Visual Studio permette di indicare, mediante un'apposita finestra, quale punto di ingresso 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.

Il compilatore csc

Per la scrittura di applicazioni complesse e che abbiano un'utilità per il mondo reale non basterà, o meglio non converrà, utilizzare un editor di testo e il *compilatore csc*. Nonostante questo è bene comprendere a fondo il processo di compilazione del codice e imparare a produrre un programma eseguibile o delle librerie, anche se non si ha a disposizione un ambiente di sviluppo integrato (detto anche IDE) come Visual Studio.

Inoltre, anche per grandi progetti, potrebbe essere necessario comprendere le opzioni del compilatore a riga di comando, per esempio per utilizzare dei sistemi di compilazione automatica del codice, come msbuild, che è lo strumento utilizzato da Visual Studio per la compilazione di soluzioni e progetti e che quindi potrebbe tornare utile se si ha una soluzione da compilare su un computer che però non ha Visual Studio installato.

La conoscenza delle varie opzioni di csc consente infine di capire ancora più a fondo C# ed il .NET Framework in generale.

Per conoscere tutte le opzioni utilizzabili con csc è possibile intanto eseguire il seguente comando:

```
C:\> csc /help
```

Oppure in forma abbreviata, csc /?. La risposta, se il comando è disponibile (cioè se il .NET Framework è installato correttamente e se la variabile d'ambiente PATH comprende il percorso di csc.exe) sarà simile alla seguente:

```
C:\Program Files (x86)\Microsoft Visual Studio 14.0>csc -?
Compilatore Microsoft (R) Visual C# versione 1.0.0.50618
Copyright (C) Microsoft Corporation. Tutti i diritti riservati.

Opzioni del compilatore Visual C#
```

Di seguito sarà visualizzato un elenco di tutte le opzioni disponibili e utilizzabili con il comando csc e la relativa spiegazione.

Specifica di input/output

La compilazione di un programma C# coinvolge uno o più file sorgente e ha come output un assembly che può essere di diversi tipi e di cui è possibile specificare il nome.

Se non si utilizzano opzioni particolari, il nome dell'assembly prodotto dalla compilazione sarà uguale al nome del file .cs contenente il metodo di ingresso Main ed estensione .exe.

È possibile specificare il nome del file eseguibile, o in generale dell'assembly, da produrre specificandolo mediante l'opzione output o out.

Per esempio, per compilare il file helloworld.cs producendo l'assembly hello.exe (anziché helloworld.exe) basterà scrivere:

```
C:\> csc /out:hello.exe helloworld.cs
```

Un programma può essere costituito da diversi file sorgenti. Per compilarli in un unico assembly basterà indicarli tutti uno dopo l'altro:

```
C:\> csc /out:hello.exe helloworld.cs hello2.cs hello3.cs
```

Il compilatore consente di ottenere dal codice C# anche delle librerie dll, utilizzabili in altre applicazioni o da altre librerie.

Un assembly di tipo dll non ha un punto di ingresso Main, in quanto dovrà essere utilizzato da altri programmi, che poi costituiranno l'eseguibile vero e proprio e forniranno quindi il punto di ingresso.

Per specificare il tipo di assembly da produrre si utilizza l'opzione target (abbreviabile in t), i cui valori principali e più comuni sono quelli indicati nella tabella seguente:

Tabella 2.1 **- Opzioni target del compilatore C#.**

Opzione	Descrizione
/target:exe	Crea un'applicazione eseguibile da console; è l'opzione di default, se non specificato altrimenti.
/target:winexe	Crea un'applicazione eseguibile con interfaccia grafica. Nulla vieta di utilizzare l'opzione exe, ma con winexe si evita di far apparire il prompt dei comandi quando l'applicazione viene eseguita con un doppio clic.
/target:library	Crea una libreria dll.

Per esempio, per compilare un file hellolib.cs e ottenere un assembly hellolib.dll, basterà utilizzare il seguente comando:

```
C:\> csc /target:library hellolib.cs
```

Riferimenti esterni

All'interno dei file di codice sorgente C# possono esserci dei riferimenti a namespace e relativi tipi definiti in qualche assembly esterno rispetto ai file da compilare. Per esempio, nel caso precedente abbiamo visto che per stampare una riga di testo si è utilizzato il metodo WriteLine della classe Console, definita nel namespace System.

In generale ciò avviene per ogni tipo referenziato all'interno del nostro codice. Il compilatore ricerca se ogni elemento utilizzato è implementato nel codice C#, in caso contrario deve essere in grado di trovarlo in un assembly esterno.

È quindi necessario poter informare il compilatore su quali assembly esterni debba utilizzare e ciò avviene per mezzo dell'opzione reference (abbreviabile in r).

Per esempio, se volessimo utilizzare dei tipi presenti all'interno di un assembly chiamato HelloLib.dll, scriveremmo il comando di compilazione nel seguente modo:

```
C:\> csc /target:exe /reference:hellolib.dll helloworld.cs
```

I più attenti fra i lettori si saranno posti una domanda: perché allora il comando che abbiamo utilizzato per compilare il primo esempio helloworld.cs è andato a buon fine, nonostante utilizzi il tipo System.Console definito certamente in un assembly esterno? La domanda è legittima.

Il namespace System è contenuto in un assembly standard del .NET Framework, chiamato MSCorLib.dll, che il compilatore deve essere in grado di trovare e utilizzare per leggere

l'implementazione della classe Console. Quindi il comando corretto e completo sarebbe il seguente:

```
C:\> csc /out:hello.exe /r:mscorlib.dll helloworld.cs
```

Come si può verificare, funziona correttamente e va a buon fine compilando il file helloworld.cs passato in ingresso.

MSCorLib.dll è un assembly che contiene tutti i tipi standard (come Byte, Int32, String e molti altri che impareremo presto a utilizzare) contenuti nel namespace System; sono utilizzati così frequentemente che il compilatore aggiunge automaticamente un riferimento a esso.

Se per qualche motivo si volesse utilizzare il compilatore senza aggiungere tale riferimento automatico a MSCorLib.dll, bisognerebbe utilizzare un'altra opzione: nostdlib. Infatti, provando a compilare l'esempio helloworld.cs con il seguente comando:

```
C:\> csc /nostdlib hell oworld.cs
```

Si otterrebbe un errore perché il compilatore non riuscirebbe a risolvere il riferimento alla classe Console contenuta nel namespace System.

NOTA	Se vi state chiedendo a cosa possa servire l'opzione /nostdlib, la risposta è che può essere utilizzata per compilare il proprio namespace System; infatti è utilizzata da Microsoft per compilare l'assembly MSCorLib.dll stesso. Un altro utilizzo possibile è per la compilazione di programmi che non verranno eseguiti dalla versione standard del .NET Framework e che quindi non faranno uso dell'assembly MSCorLib.dll.

Opzioni predefinite

Quando si lavora a un programma C# è logico che capiti di doverlo compilare più volte. Anziché dover digitare continuamente le stesse opzioni e un elenco magari molto lungo di file sorgente .cs da passare al comando csc, è possibile salvarle in uno o più file appositi, detti *response file*, che utilizzano l'estensione .rsp, e poi utilizzarli con csc riferendosi a essi con il carattere @.

Per esempio, supponiamo di dover utilizzare il comando seguente per compilare la nostra applicazione:

```
C:\> csc /out:hello.dll /target:library /r:assembly123.dll libreria2.cs libreria1.cs libreriahelloworld.cs
```

Possiamo salvare le opzioni:

```
/out:hello.dll /target:library /r:assembly123.dll libreria2.cs libreria1.cs libreriahelloworld.cs
```

in un file opzioni.rsp e utilizzarlo così:

```
C:\> csc @opzioni.rsp
```

È possibile utilizzare più file rsp e combinarli anche con opzioni passate direttamente sulla riga di comando.

Quando viene eseguito il comando csc, in aggiunta alle opzioni specificate da riga di comando, il compilatore ricerca all'interno della stessa directory di csc.exe un response file csc.rsp, contenente le opzioni di compilazione predefinite.

Per esempio, all'interno della cartella contenente il file csc.exe (verificate in C:\Windows\Microsoft.NET\Framework\v4.0.30319\ oppure, se avete installato Visual Studio 2015, in C:\Program Files (x86)\MSBuild\14.0\Bin\) è possibile trovare anche il file csc.rsp, con un contenuto simile al seguente (è un riassunto, per questioni di spazio):

```
# This file contains command-line options that the C#
# command line compiler (CSC) will process as part
# of every compilation, unless the "/noconfig" option
# is specified.

# Reference the common Framework libraries
/r:Accessibility.dll
/r:Microsoft.CSharp.dll
/r:System.Configuration.dll
/r:System.Configuration.Install.dll
/r:System.Core.dll
/r:System.Data.dll
/r:System.Data.Linq.dll
/r:System.dll
/r:System.Drawing.Design.dll
/r:System.Drawing.dll
…
```

Quindi, per compilare un'applicazione che utilizza uno degli assembly standard elencati non sarà necessario specificarlo mediante l'opzione reference.

Se comunque si specifica un'opzione già presente in un file rsp, il conflitto verrà risolto dando precedenza all'opzione specificata dall'utente.

Per non includere invece il file csc.rsp in compilazione, si può utilizzare l'opzione /noconfig.

Visual Studio 2015

Come abbiamo già accennato, per lo sviluppo professionale di applicazioni non è sicuramente possibile limitarsi all'utilizzo di un editor di testo e del compilatore a riga di comando.

Esistono diversi ambienti di sviluppo integrato, detti anche *IDE* (Integrated Development Environment), che aiutano lo sviluppatore nel suo compito e velocizzano di molto lo sviluppo software. Per C#, citiamo fra gli altri *SharpDevelop* e *MonoDevelop*, entrambi open source. Il secondo è anche multipiattaforma e fornisce gli appositi installer, oltre che per Windows, anche per MacOS X e per varie distribuzioni Linux.

Visual Studio, invece, è l'IDE di casa Microsoft e, senza timore di smentite, è certamente l'ambiente più completo e potente per lo sviluppo in .NET e quindi C#.

Per chi non potesse o volesse acquistare una versione a pagamento, esiste anche la versione *Community*, di Visual Studio 2015 che è totalmente gratuita e scaricabile dalla pagina https://www.visualstudio.com/downloads/download-visual-studio-vs.

Con Visual Studio 2015 Community, è possibile sviluppare in C# praticamente qualunque tipo di progetto, dalle applicazioni desktop per Windows a quelle web in ASP.NET e naturalmente anche app distribuibili sul Windows Store per Windows 8/WinRT, app per smartphone dotati di Windows Phone 7.x/8.x o ancora Windows Universal App per Windows 10, sia desktop che mobile.

L'elenco non si ferma qui: Visual Studio 2015 consente di sviluppare applicazioni in C# anche per dispositivi dotati di altri sistemi operativi, come Android o iOS, utilizzando l'integrazione con la piattaforma Xamarin.

NOTA	Nel seguito del capitolo e lungo il resto del libro, si utilizzerà la versione inglese di Visual Studio 2015, quindi la descrizione di comandi e menu potrebbe essere diversa se utilizzate una versione in italiano. Il mio consiglio è comunque di utilizzare una versione inglese perché, qualora ve ne fosse la necessità, sarà più semplice trovare su Internet consigli e soluzioni per particolari problemi e difficoltà ricercandoli in inglese.

La fase di installazione di Visual Studio 2015 non è differente da quella di una qualsiasi altra applicazione Windows e consente di selezionare delle funzionalità opzionali, in particolare per il supporto allo sviluppo di Windows Universal Apps e di app multipiattaforma, per esempio Android e iOS. L'approccio più semplice, se non avete problemi di spazio, è quello di scegliere l'installazione completa.

Per mostrare ed esporre tutte le funzionalità messe a disposizione da Visual Studio non basterebbe in effetti un intero libro, quindi nei paragrafi successivi faremo una rapida panoramica delle principali funzioni in esso presenti, in maniera da permettervi di iniziare da subito a creare i vostri esempi di programmi C#, compilarli, eseguirli e risolvere bug e difetti mediante debug.

Molti aspetti saranno più chiari e comprensibili una volta approfonditi i relativi concetti del linguaggio. Lascio a voi il compito di approfondire l'utilizzo dell'IDE e scoprirne ogni suo aspetto man mano che andrete avanti con la lettura del libro: vi suggerisco di provare a sperimentare, prendere confidenza con ogni suo comando e imparare a conoscere ogni sua finestra; non ve ne pentirete, anzi ogni giorno scoprirete qualcosa di nuovo che ve lo farà apprezzare ancora di più.

Impostazioni dell'ambiente

Quando si lancia Visual Studio 2015 per la prima volta dopo l'installazione, si potrà effettuare l'accesso con il proprio Microsoft Account, oppure decidere di farlo in seguito. La possibilità di eseguire l'accesso con un account Microsoft, oltre alla verifica della licenza, porta con sé diversi vantaggi: in particolare la sincronizzazione delle proprie impostazioni preferite fra diverse postazioni di sviluppo, per esempio scorciatoie da tastiera personalizzate, il layout delle finestre, il tema grafico e, inoltre, effettuato la prima volta l'accesso, si manterrà anche il login a servizi collegati come Azure e Visual Studio online.

Il secondo passo è la selezione della configurazione più adatta per il linguaggio che si ha intenzione di utilizzare.

La scelta fra le diverse opzioni deriva dal fatto che Visual Studio è stato ed è continuamente aggiornato da Microsoft, per riflettere le ultime tecnologie e gli strumenti di sviluppo che vengono introdotti, compresi quindi anche i diversi linguaggi.

La Figura 2.1 mostra la finestra di selezione. La scelta suggerita è naturalmente quella Visual C# Development Settings in quanto, se state leggendo questo libro, probabilmente siete interessati a ottimizzare il vostro ambiente per il linguaggio C#.

Import and Export Settings Wizard ? X

Choose a Default Collection of Settings

Which collection of settings do you want to reset to?

- General
- JavaScript
- Visual Basic
- Visual C#
- Visual C++
- Visual F#
- Web Development
- Web Development (Code Only)

Description:

Customizes the environment to maximize code editor screen space and improve the visibility of commands specific to C#. Increases productivity with keyboard shortcuts that are designed to be easy to learn and use.

< Previous | Next > | Finish | Cancel

Figura 2.1 – Selezione delle impostazioni dell'ambiente di sviluppo in Visual Studio 2015.

Tale opzione ottimizzerà lo spazio di scrittura del codice, le scorciatoie da tastiera e la visibilità dei comandi di Visual Studio 2015 per lo sviluppo in C#.

Naturalmente si è liberi di personalizzare interamente ogni aspetto dell'IDE e, volendo, è possibile anche ripristinare una delle impostazioni elencate nella finestra precedente, mediante il comando Tools -> Import and Export Settings, che avvierà una procedura guidata per l'importazione ed esportazione delle configurazioni (utile se, per esempio, si utilizzano più macchine), ma anche per il loro ripristino totale.

Sempre all'interno del menu Tools sono presenti le voci Customize e Options, che permettono di apportare delle personalizzazioni per ogni singolo aspetto dell'IDE.

Per esempio, scegliendo il comando Options e poi selezionando la sezione Environment -> General è possibile modificare il tema di Visual Studio, scegliendolo fra Blue, Dark e Light (vedi Figura 2.2), oppure con Fonts and Colors è possibile modificare i colori e i caratteri utilizzati per l'editor di codice. Una volta avviato l'ambiente, vi si presenterà una finestra con una serie di menu e toolbar, la schermata di partenza (vedi Figura 2.3), che permette di visualizzare dei video in streaming per imparare a sviluppare ogni tipologia di progetto, la sezione di news e la sezione Getting Started, che contiene link a risorse utili per lo sviluppo software in .NET. Sulla sinistra, inoltre, è presente la sezione Start che permette di avviare la creazione di un nuovo progetto, come vedremo nel prossimo paragrafo, e l'elenco dei progetti recenti, per poterli riaprire in maniera rapida.

Options

Search Options (Ctrl+E)

- Environment
  - General
  - AutoRecover
  - Documents
  - Extensions and Updates
  - Find and Replace
  - Fonts and Colors
  - Import and Export Settings
  - International Settings
  - Keyboard
  - Notifications
  - Quick Launch
  - Startup
  - Synchronized Settings
  - Tabs and Windows
  - Task List
  - Web Browser

Visual experience

Color theme: Light

☐ Apply title case styling to menu bar

☑ Automatically adjust visual experience based on client performance

☑ Enable rich client visual experience

☐ Use hardware graphics acceleration if available

Visual Studio is currently using software rendering. The visual experience settings automatically change based on system capabilities.

10 items shown in Window menu

10 items shown in recently used lists

☑ Show status bar

☑ Close button affects active tool window only

☐ Auto Hide button affects active tool window only

Manage File Associations

OK Cancel

Figura 2.2 – Opzioni di configurazione di Visual Studio 2015.

Start Page - Microsoft Visual Studio

Quick Launch (Ctrl+Q)

FILE EDIT VIEW BUILD DEBUG TEAM TOOLS TEST ANALYZE WINDOW HELP

Antonio Pelleriti

Attach...

Toolbox

Start Page

Solution Explorer

Team Explorer

Professional 2013

Start

New Project...

Open Project...

Open from Source Control...

Recent

AppendiceA-Strings

WindowsStoreApp1

HelloPhoneApp

WebApplication1

MVCApplicationStart

MVCApplication

WebPagesApplication

WingtipToys

WebApplication1

CorollaPhoto

Discover what's new in Professional 2013

You can find information about new features and enhancements in Professional 2013 by reviewing the following sections.

Learn about new features in Professional 2013

See what's new in .NET Framework 4.5.1

Explore what's new in Team Foundation Service

Relocate the What's New information

What's new on Microsoft Platforms

Windows

Windows Azure

ASP.NET vNext and Web

Windows Phone

Microsoft Office

SharePoint Development

Announcements

Visual Studio 2013.1 RC is Available

mercoledì, dicembre 11, 2013

Read Brian Harry's note on the first Visual Studio 2013 Update.

Rename Refactor Helper for Visual Studio 2013

martedì, dicembre 3, 2013

Check out this new DevLabs prototype, a "rename" refactoring tool for Visual C++.

MVPs Explore Visual Studio 2013

martedì, dicembre 3, 2013

Read MVP posts diving into the features of Visual Studio 2013.

Watch the Visual Studio 2013 Virtual Launch

martedì, dicembre 3, 2013

Ready

Figura 2.3 – Schermata principale di Visual Studio 2015.

Creare un nuovo progetto

Dopo aver installato e configurato Visual Studio si può iniziare a creare il primo progetto C#.

Raramente si inizia a sviluppare un'applicazione da un file di testo vuoto, come si farebbe invece utilizzando il compilatore a riga di comando.

Un progetto Visual Studio consente di raggruppare tutti i file e le risorse necessarie per compilare un'applicazione eseguibile, un assembly, un'applicazione web scritta in ASP.NET e così via.

Visual Studio consente inoltre di selezionare la tipologia di progetto mediante il comando New -> Project, presente nel menu principale File, oppure utilizzando il link rapido New Project, presente nella schermata di avvio.

Il primo passo nella creazione di un nuovo progetto è la scelta del template. Per esempio, se si vuol creare un'applicazione Windows da eseguire sul desktop, quindi dotata di interfaccia grafica a finestre, si potrà scegliere una fra le tipologie Windows Forms Application e WPF Application, all'interno della sezione Windows/Classic Desktop, mentre per creare una Windows Universal Apps per Windows 10, basterà scegliere uno dei template presenti nella sezione Windows/Universal. Le sezioni sulla sinistra permettono di esplorare agevolmente le varie tipologie di progetto gestibili da Visual Studio 2015, anche scegliendo il linguaggio che si vuol utilizzare (non siete obbligati a utilizzare C#, ma in questo libro è dato per scontato che lo facciate).

I template sono moltissimi e si possono eventualmente crearne o installarne di nuovi: per esempio, selezionando la sezione Online sulla sinistra, sarà possibile esplorare e scaricare nuovi template o esempi di progetti dal Web.

Supponiamo ora che abbiate scelto il template Console Application, che serve appunto per creare un'applicazione testuale da eseguire nel prompt dei comandi; il passo successivo è impostare il nome del progetto e il percorso in cui crearlo.

Visual Studio permette anche di scegliere la versione del .NET Framework che si vuol utilizzare per il progetto, basta selezionarla dalla casella combinata in alto, sempre nella finestra di creazione del progetto (vedi Figura 2.4).

Una volta confermata la creazione del progetto cliccando sul tasto OK, Visual Studio creerà automaticamente i file di base necessari per iniziare lo sviluppo.

Nel caso del template Console Application verrà generato un file Program.cs, contenente una classe con il metodo Main.

Tale file viene automaticamente aperto al centro, nell'editor di testo, e potete quindi da subito iniziare a scrivere il vostro codice, per esempio ricreando l'esempio Hello World visto in precedenza.

New Project

Recent

Installed

Templates

Visual C#

Windows

Web

Android

Cloud

Extensibility

iOS

LightSwitch

Office/SharePoint

Reporting

Silverlight

Test

WCF

Workflow

Other Languages

Other Project Types

Modeling Projects

Samples

Online

.NET Framework 4.5.2 | Sort by: Default | Search Installed Templates (Ctrl+E)

.NET Framework 2.0
.NET Framework 3.0
.NET Framework 3.5
.NET Framework 4
.NET Framework 4.5
.NET Framework 4.5.1
.NET Framework 4.5.2
.NET Framework 4.6
<More Frameworks...>

al Windows)	Visual C#
plication	Visual C#
	Visual C#
n	Visual C#
ASP.NET Web Application	Visual C#
Shared Project	Visual C#
Class Library (Package)	Visual C#
Console Application (Package)	Visual C#
Class Library	Visual C#
Class Library (Portable)	Visual C#

Type: Visual C#

A project for creating a command-line application

Click here to go online and find templates.

Name: ConsoleApplication2

Location: c:\users\antonio\documents\visual studio 2015\Projects

Browse...

Solution: Create new solution

Solution name: ConsoleApplication2

☑ Create directory for solution

☐ Add to source control

OK | Cancel

Figura 2.4 – Creazione di un nuovo progetto in Visual Studio 2015.

Esplorare le soluzioni

Visual Studio consente di raggruppare all'interno di un unico contenitore, detto *Solution* (o *Soluzione* in italiano), diversi progetti.

Lo sviluppo di un'applicazione reale coinvolge spesso diversi progetti; per esempio, l'interfaccia grafica sarà costituita da un progetto Windows Forms o WPF, che conterrà la classe con il metodo Main e costituirà l'eseguibile vero e proprio. Tale progetto a sua volta potrà utilizzare altri assembly, per esempio contenenti il codice necessario per l'accesso a un database, o contenere delle classi di utilità, magari riprese e riutilizzate da progetti precedenti.

Ogni progetto che fa parte di una soluzione verrà compilato in un unico assembly e quindi potrà essere utilizzato da un altro progetto, impostandolo fra i suoi riferimenti (come già visto anche utilizzando il compilatore csc).

Una volta creato il progetto, o aperto uno esistente, avrete notato che sulla destra (ma ciò dipende anche dalla configurazione dell'IDE e siete liberi di spostarlo dove più vi aggrada) è presente una finestra intitolata Solution Explorer (vedi Figura 2.5). Il Solution Explorer consente di navigare ed esaminare i progetti che fanno parte della soluzione e i singoli file che a loro volta costituiscono i singoli progetti.

Facendo doppio clic su un file di codice C#, esso verrà aperto al centro della finestra dell'IDE, in un editor di testo, dove a questo punto potrà essere modificato.

In realtà esistono diversi editor di codice o testo, dedicati ognuno a un particolare tipo di file. Per esempio oltre ai file .cs di codice C#, si potrebbe avere la necessità di modificare file di codice XAML, XML, HTML e così via.

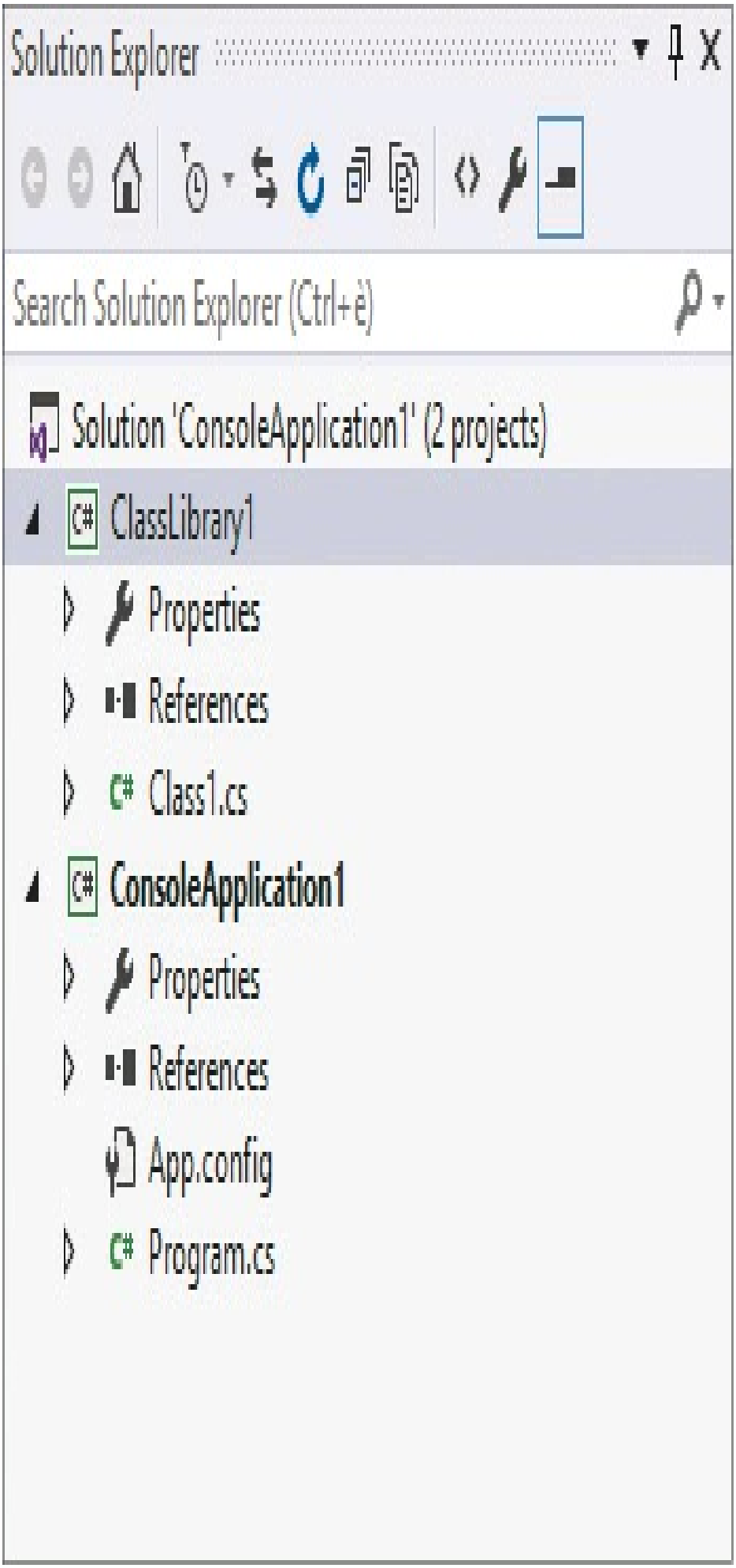

Solution Explorer
Search Solution Explorer (Ctrl+è)
Solution 'ConsoleApplication1' (2 projects)
ClassLibrary1
Properties
References
Class1.cs
ConsoleApplication1
Properties
References
App.config
Program.cs

Figura 2.5 – Il Solution Explorer di Visual Studio 2015.

Il progetto Console Application precedentemente creato contiene il file Program.cs, ma anche un file app.config, che è un file in formato XML contenente impostazioni per l'applicazione (per esempio il .NET runtime supportato) e, all'interno di una cartella Properties, un ulteriore file di codice C# denominato AssemblyInfo.cs, che contiene informazioni descrittive e di versione dell'assembly.

Fisicamente Visual Studio 2015 memorizza la soluzione all'interno di un file di testo con estensione *.sln*, contenente tutti i riferimenti ai progetti che essa comprende.

Ogni progetto a sua volta sarà memorizzato in un file in formato XML, la cui estensione dipende dalla tipologia di progetto. Per esempio un progetto C# sarà memorizzato in un file con estensione *.csproj*.

Il Solution Explorer è stato notevolmente potenziato in ogni edizione di Visual Studio. Da Visual Studio 2012 esso consente di ottenere una vista ad albero non solo dei file che costituiscono i singoli progetti, ma anche delle classi e dei membri di queste ultime. Per esempio, il file Program.cs, può essere a sua volta espanso, visualizzando il nome della classe Program e, ancora più giù, i suoi metodi, in questo caso l'unico metodo Main (vedi Figura 2.6).

Il Solution Explorer consente di ricercare agevolmente i file che costituiscono l'intera soluzione, digitando all'interno della casella di testo in alto. La ricerca avviene sia per nome di file sia all'interno del contenuto.

Se provate, per esempio, a digitare Main, il contenuto della soluzione verrà filtrato mostrando solo il file Program.cs.

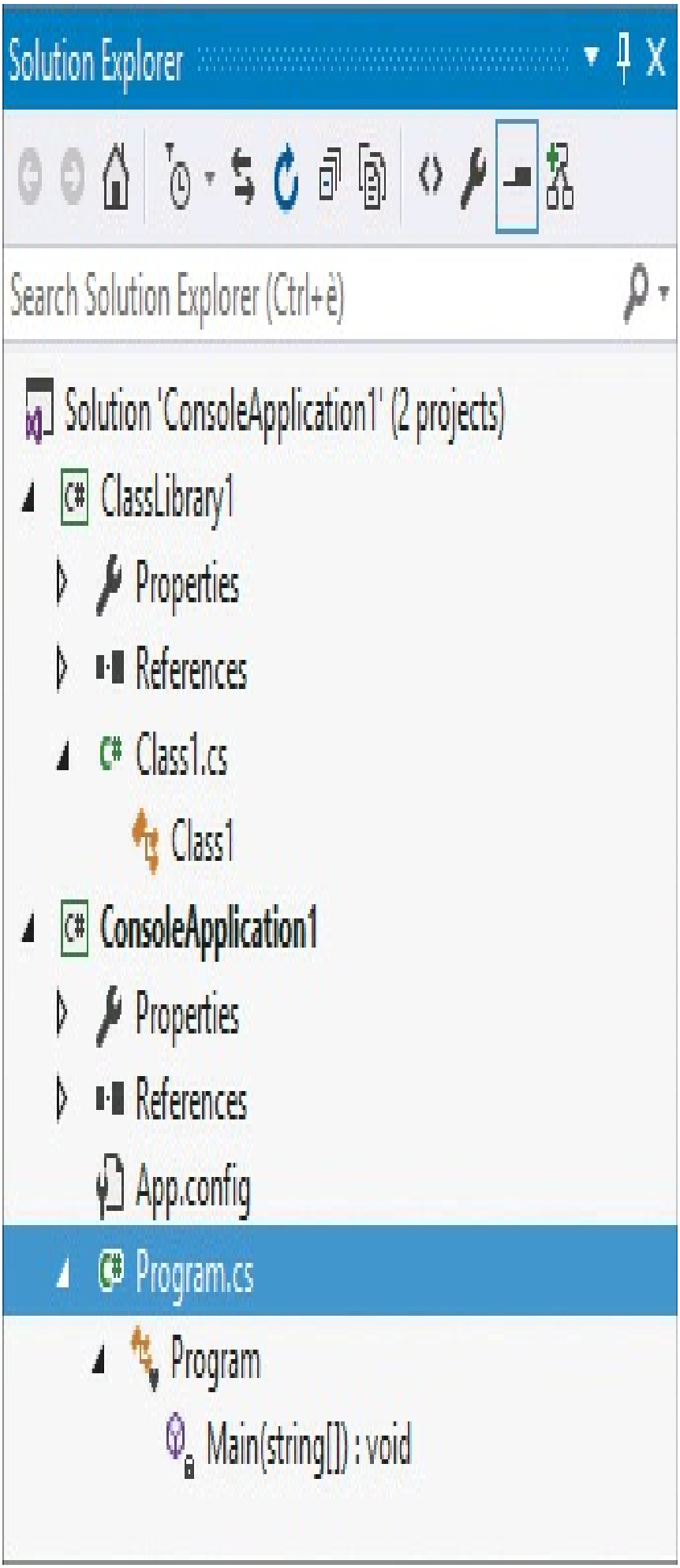

Solution Explorer
Search Solution Explorer (Ctrl+è)
Solution 'ConsoleApplication1' (2 projects)
ClassLibrary1
Properties
References
Class1.cs
Class1
ConsoleApplication1
Properties
References
App.config
Program.cs
Program
Main(string[]) : void

Figura 2.6 – Struttura ad albero dei progetti nel Solution Explorer.

Proprietà di progetto

Ogni progetto facente parte di una soluzione Visual Studio possiede delle proprietà, specifiche del tipo di progetto in esame.

Per visualizzare tali proprietà, è possibile fare clic con il tasto destro del mouse sul nome del progetto all'interno del Solution Explorer e selezionare poi il comando Properties.

Si aprirà così una finestra con diverse schede selezionabili. Per il progetto Console Application si potrà, per esempio, modificare la versione del .NET Framework da utilizzare come target, il nome della classe contenente il metodo `Main` e altri aspetti di configurazione che, utilizzando il compilatore csc da riga di comando, si dovrebbero impostare mediante apposite opzioni.

Nel caso di C#, all'interno della sezione Build, facendo clic sul pulsante Advanced, è possibile, fra le altre opzioni, personalizzare anche la versione del linguaggio che si vuole utilizzare per compilare l'applicazione (che in Visual Studio 2015 è naturalmente C# 6.0).

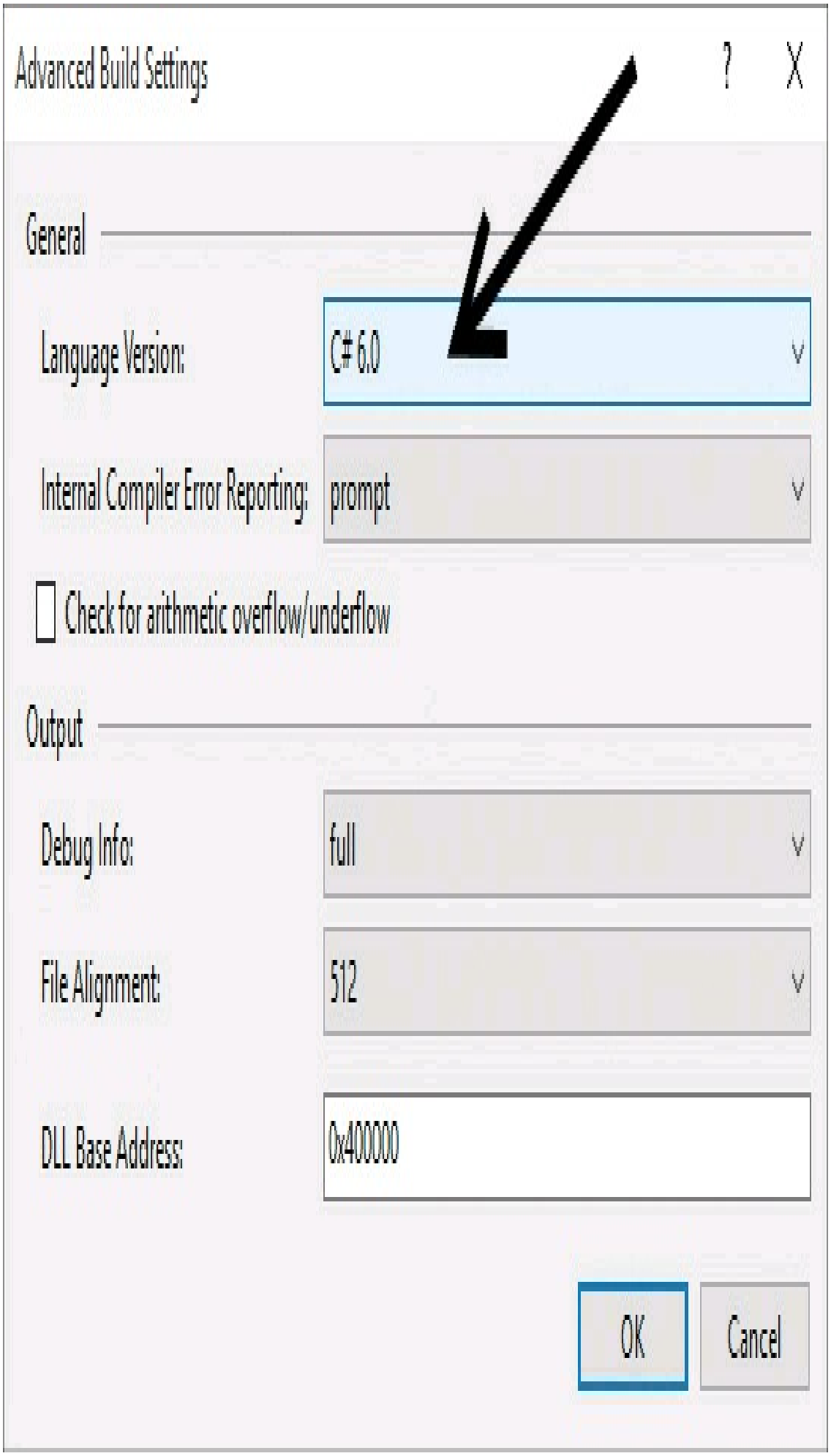

Advanced Build Settings
?
X
General
Language Version:
C# 6.0
Internal Compiler Error Reporting:
prompt
Check for arithmetic overflow/underflow
Output
Debug Info:
full
File Alignment:
512
DLL Base Address:
0x400000
OK
Cancel

Figura 2.7 – Impostazione della versione del linguaggio C#.

Scrittura del codice

L'editor di codice di Visual Studio è naturalmente la parte dell'IDE che al momento ci interessa di più, in quanto ci permetterà di scrivere e modificare i nostri programmi e, in quanto sviluppatori, sarà naturalmente anche la parte dell'ambiente che utilizzeremo maggiormente.

Per aprire un file di codice già presente nella soluzione, come già detto, basta fare doppio clic su di esso nel Solution Explorer ed esso verrà visualizzato nella parte centrale della finestra di Visual Studio, pronto per ricevere le nostre modifiche.

Visual Studio consente di lavorare contemporaneamente su più file di codice, aprendoli in una comoda interfaccia che permette di passare da uno all'altro semplicemente cliccando sull'etichetta con il nome del file. In tal modo è possibile lavorare anche su file di codice appartenenti a progetti diversi.

L'editor di codice di Visual Studio non è un semplice elaboratore di testi. Esso fornisce funzionalità per semplificare e ottimizzare la scrittura di codice e per visualizzare informazioni su di esso in maniera rapida ed efficiente.

Per esempio, le parti di codice vengono colorate ed evidenziate per indicare che si tratta di parole chiave del linguaggio, oppure di stringhe, di nomi di tipi e così via.

La Figura 2.8 mostra il file Program.cs dell'applicazione Console che abbiamo appena creato.

Si noti come tutte le parole chiave presenti sono colorate diversamente (using, namespace, class, public) così come il nome della classe Program appare in un grigio differente da quello del nome del metodo Main.

Sempre nella Figura 2.8, si noti come l'IDE attualmente contenga due etichette in alto, con il nome dei file di codice aperti, Program.cs e Class1.cs. Il file Program.cs è evidenziato con un asterisco, per indicare che esso contiene delle modifiche non salvate.

ConsoleApplication1 - Microsoft Visual Studio

FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS ANTS ARCHITECTURE TEST

Debug | Any CPU | Start

Toolbox

Program.cs* | Class1.cs

ConsoleApplication1 | ConsoleApplication1.Program | Main(string[] args)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {

        }
    }
}
```

Figura 2.8 – Editor di codice di Visual Studio.

L'editor permette di scrivere codice formattato correttamente, in maniera da renderlo più leggibile e chiaro mediante innumerevoli funzionalità: esso aggiunge, per esempio, le corrette indentazioni, indica le corrispondenze di parentesi aperte e chiuse (basta posizionarsi su una di esse), evidenzia gli errori di sintassi (provate, per esempio, a scrivere una qualunque parola che non fa parte delle keyword di C# ed essa verrà sottolineata mediante una linea ondulata, mentre un tooltip indicherà il tipo di errore, vedi Figura 2.9).

Program.cs* ✎ X Class1.cs

ConsoleApplication1 | ConsoleApplication1.Program | Main(string[] args)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
    class Program
    {
        statib void Main(string[] args)
        {
            The name 'statib' does not exist in the current context.
        }
    }
}
```

Figura 2.9 – Errori di sintassi nell'editor di codice.

Inoltre, noterete che sulla parte sinistra esistono dei piccoli pulsanti con un simbolo -, che permettono di chiudere una determinata regione di codice: per esempio, facendo clic sul simbolo - presente in corrispondenza della riga in cui si trova il metodo Main, il metodo stesso verrà collassato, sulla destra appariranno dei puntini di sospensione per indicare la presenza di codice nascosto e il pulsante si trasformerà ora in un simbolo +, che consente di espandere nuovamente il codice.

In tal modo è possibile concentrarsi su particolari porzioni di codice in lavorazione e collassare le parti già finite e testate.

Intellisense

L'editor facilita la scrittura del codice mediante la funzionalità intellisense, di cui non potrete fare a meno già dopo il primo utilizzo.

Basta iniziare a scrivere del testo nell'editor e Visual Studio suggerirà possibili completamenti del testo in maniera contestuale, cioè a seconda del punto in cui i nuovi caratteri vengono digitati.

Supponiamo di voler completare il metodo Main, ricreando l'esempio Hello World: necessitiamo quindi di utilizzare il metodo WriteLine della classe Console. Se iniziate a scrivere all'interno del metodo Main la lettera C, immediatamente verranno suggeriti nomi di classi e metodi che iniziano per C e, man mano che si aggiungono lettere, tale elenco viene filtrato.

NOTA	Per forzare l'esecuzione dell'intellisense e l'apparizione dei suggerimenti, utilizzate la combinazione da tastiera CTRL+Spazio.

Dall'elenco è possibile selezionare un elemento, completando la parola e quindi risparmiando tempo nella digitazione. Per esempio, completate la digitazione scegliendo la classe Console e poi digitate il punto per poter completare l'istruzione. L'intellisense vi mostrerà a questo punto l'elenco di tutti i membri della classe Console; digitate una W, per filtrare solo i membri che iniziano per tale lettera, e poi scegliete il metodo WriteLine, completando la digitazione con l'apertura di una parentesi tonda. All'apertura di tale parentesi, intellisense suggerirà anche i possibili parametri che il metodo può accettare (vedi Figura 2.10).

I tooltip, a partire dalla versione 2015, sono ora totalmente a colori; inoltre, grazie all'integrazione della .NET Compiler Platform (Roslyn) sono fornite molte più possibilità di analisi e di azioni. Quando l'IDE ha un suggerimento da fornire, per esempio un intervento di

correzione da apportare, apparirà un'icona a forma di lampadina, detta *light bulb,* sul margine sinistro dell'editor oppure in un particolare punto del codice.

Una volta fatto clic su tale icona, apparirà una o più Quick Action, selezionabili ed eseguibili con un clic. Le light bulb sono anche attivabili manualmente con la scorciatoia (CTRL+.).

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

- ActivationContext
- AppContext
- BitConverter
- Console
- ConsoleApplication1
- ConsoleCancelEventArgs
- ConsoleCancelEventHandler
- ConsoleColor
- ConsoleKey

```
static void Main(string[] args)
{
    Console.Wr
}
```

- Write
- WriteLine

```
static void Main(string[] args)
{
    Console.WriteLine();
}
```

▲ 11 of 19 ▼ void Console.WriteLine(string value)

Writes the specified string value, followed by the current line term

value: The value to write.

Figura 2.10 – Intellisense di Visual Studio in azione.

Per esempio, provate a togliere dal codice della classe tutte le istruzioni using in testa al file. Scrivendo ora all'interno del Main la parola Console, essa non sarà più riconosciuta. Apparirà come preannunciato una light bulb (vedi Figura 2.11), che permetterà di aggiungere automaticamente l'istruzione mancante.

```
static void Main(string[] args)
{
    Console
```

using System;

Change 'Console' to 'System.Console'.

Generate property 'Program.Console'

Generate field 'Console' in 'Program'

Generate read-only field 'Program.Console'

Generate local 'Console'

CS0103 The name 'Console' does not exist in the curre

```
using System;

namespace ConsoleApplication1
...
```

Preview changes

Figura 2.11 – Light bulb di Visual Studio in azione.

Selezionando una delle azioni suggerite sarà possibile poi avere una rapida anteprima di quanto accadrà eseguendola e quindi attivare il suo completamento.

Aggiungere nuovi elementi

Una volta creato un progetto, esso conterrà solo i pochi file definiti dal template scelto.

Naturalmente sarà presto necessario aggiungere nuovi file di codice o altri elementi a seconda della tipologia di applicazione. Per esempio, se si vuol creare una nuova classe e aggiungerla a un progetto, basta fare clic con il tasto destro sul nome del progetto stesso e poi, dal menu contestuale, selezionare il comando Add -> New Item. A questo punto sarà possibile scegliere uno dei possibili elementi utilizzabili nel particolare tipo di progetto, scegliere un nome e far creare a Visual Studio i file necessari.

Se volete includere nel progetto un file già esistente, magari creato con un editor di testo esterno o da un altro sviluppatore, potete selezionare il comando Add -> Existing Item e poi andare a sfogliare il file system ricercando il file da aggiungere.

Inoltre, sempre dal menu contestuale del progetto, è possibile organizzare meglio i vari elementi, creando delle sottocartelle.

Anche una soluzione può contenere delle cartelle per organizzare meglio i progetti in essa contenuti. In particolare Visual Studio consente di creare delle Solution Folder, che non hanno però una corrispondenza reale con delle directory fisiche, ma sono solo un concetto logico di raggruppamento di progetti o altri elementi come file di documentazione o di progetto.

Finestra di design

Se si sta sviluppando un'applicazione dotata di interfaccia grafica, per esempio Windows Forms, WPF o un'applicazione web con ASP.NET, è possibile utilizzare la finestra di *Design* e una *Casella degli strumenti* per creare le interfacce in maniera visuale, risparmiandosi la scrittura manuale di codice.

Provate, per esempio, a creare un nuovo progetto e scegliere il template Windows Forms Application. Visual Studio creerà automaticamente una soluzione con un progetto contenente, oltre al file Program.cs, anche un nuovo elemento denominato Form1.

Una finestra di un progetto Windows Forms è rappresentata in .NET mediante la classe Form e in Visual Studio è possibile progettarla e modificarla semplicemente trascinandovi sopra dei controlli, spostandoli, ridimensionandoli e, in genere, impostandone le proprietà.

Aprendo il file Form1, con un doppio clic dal Solution Explorer, si aprirà al centro dell'IDE una finestra di design, come mostrato nella Figura 2.12.

Toolbox

Quando è aperta una Form, o comunque un elemento di interfaccia grafica, Visual Studio consente di utilizzare una Casella degli strumenti o *Toolbox*, ancorata in genere alla sinistra dell'ambiente (vedi Figura 2.12), dalla quale è possibile selezionare controlli e componenti da aggiungere all'interfaccia che si sta costruendo.

Provate a trascinare un controllo Button nell'area della Form1 e spostarlo o ridimensionarlo.

Se si installano strumenti e suite di controllo di terze parti, essi verranno visualizzati all'interno della Toolbox, in una categoria apposita.

È inoltre possibile personalizzare la Toolbox, rimuovendo o aggiungendo manualmente altri elementi.

ConsoleApplication1 - Microsoft Visual Studio

Quick Launch (Ctrl+Q)

FILE EDIT VIEW PROJECT BUILD DEBUG TEAM FORMAT TOOLS ANTS ARCHITECTURE TEST .NET REFLECTOR ANALYZE WINDOW HELP

Antonio Pelleriti

Debug | Any CPU | ConsoleApplication1 | Start

Data Sources

Toolbox

Search Toolbox

- All Windows Forms
- Common Controls
  - Pointer
  - Button
  - CheckBox
  - CheckedListBox
  - ComboBox
  - DateTimePicker
  - Label
  - LinkLabel
  - ListBox
  - ListView
  - MaskedTextBox
  - MonthCalendar
  - NotifyIcon
  - NumericUpDown
  - PictureBox
  - ProgressBar
  - RadioButton
  - RichTextBox
  - TextBox
  - ToolTip
  - TreeView
  - WebBrowser
- Containers
- Menus & Toolbars
- Data
- Components
- Printing
- Dialogs
- Reporting

Form1.cs [Design]* | Program.cs* | Class1.cs

Form1

button1

Error List

Entire Solution | 0 Errors | 0 Warnings

Search Error List

Code	Description	Project	File

Solution Explorer

Search Solution Explorer (Ctrl+è)

- Solution 'ConsoleApplication1' (3 projects)
  - ClassLibrary1
  - ConsoleApplication1
  - WindowsFormsApplication1
    - Properties
    - References
    - App.config
    - Form1.cs
      - Form1.Designer.cs
      - Form1.resx
      - Form1
    - Program.cs

Solution Explorer | Team Explorer

Diagnostic Tools

Properties

button1 System.Windows.Forms.Button

MaximumSize	0; 0
MinimumSize	0; 0
Modifiers	Private
Padding	0; 0; 0; 0
RightToLeft	No
Size	75; 23
TabIndex	0
TabStop	True
Tag	
Text	button1
TextAlign	MiddleCenter

Text

The text associated with the control.

Ready

107, 85

75 x 23

Figura 2.12 – Design di interfacce grafiche Windows.

Finestra delle proprietà

In modalità Design è possibile selezionare i singoli componenti e controlli dell'interfaccia grafica e impostarne le proprietà, mediante un'apposita *Finestra delle proprietà*, che appare, in genere, alla destra dell'ambiente, subito sotto il Solution Explorer.

Per esempio, nella Figura 2.12 sono mostrate le proprietà del controllo Button, aggiunto prima sulla superficie della Form1.

Dalla finestra delle proprietà potete osservare i valori delle varie proprietà del controllo e modificarle direttamente; per esempio, provate a cambiare il testo visualizzato sul pulsante, impostando la proprietà Text.

Aggiungere progetti alla soluzione

Abbiamo ormai capito che in definitiva un progetto è l'insieme di tutti i file di codice e di eventuali risorse, che concorreranno a generare un singolo assembly (per esempio, l'assembly eseguibile o una libreria dll).

Una soluzione invece è l'insieme di tutti i progetti che costituiscono un'applicazione.

Per aggiungere un nuovo progetto alla soluzione attualmente aperta in Visual Studio, basta fare clic con il tasto destro sul nome della soluzione e dal menu contestuale selezionare il comando New Project presente nel sottomenu Add.

A questo punto si aprirà la finestra di creazione di un nuovo progetto vista in precedenza, che però stavolta non creerà una nuova soluzione, ma aggiungerà il progetto stesso a quella attualmente aperta.

Gestione dei riferimenti

Compilando un'applicazione da riga di comando con il compilatore csc, per indicare dei riferimenti ad assembly esterni, si è visto che è necessario utilizzare l'opzione /reference: seguita dai nomi di tali assembly.

In Visual Studio tale procedura è notevolmente semplificata e, inoltre, potendo essere una soluzione costituita da più progetti, deve essere possibile indicare che un progetto a sua volta ha fra i suoi riferimenti uno o più progetti della stessa soluzione, che quindi devono essere compilati al volo.

Avrete già notato che nel Solution Explorer, all'interno di ogni progetto è presente un ramo References, contenente appunto i riferimenti ad assembly esterni al progetto in esame.

Per modificare o aggiungere i riferimenti di un progetto, basta fare clic con il tasto destro su References e quindi selezionare il comando Add Reference, che aprirà una finestra denominata Reference Manager.

Reference Manager - ConsoleApplication1

Assemblies

Framework

Extensions

Projects

Shared Projects

COM

Browse

Targeting: .NET Framework 4.6

Search Assemblies (Ctrl+E)

	Name	Version
	PresentationFramework.Classic	4.0.0.0
	PresentationFramework.Luna	4.0.0.0
	PresentationFramework.Royale	4.0.0.0
	ReachFramework	4.0.0.0
	sysglobl	4.0.0.0
☑	System	4.0.0.0
	System.Activities	4.0.0.0
	System.Activities.Core.Presentation	4.0.0.0
	System.Activities.DurableInstancing	4.0.0.0
	System.Activities.Presentation	4.0.0.0
	System.AddIn	4.0.0.0
	System.AddIn.Contract	4.0.0.0
	System.ComponentModel.Composition	4.0.0.0
	System.ComponentModel.Composition.Regist...	4.0.0.0
	System.ComponentModel.DataAnnotations	4.0.0.0
	System.Configuration	4.0.0.0
	System.Configuration.Install	4.0.0.0
☑	System.Core	4.0.0.0
☑	System.Data	4.0.0.0
☑	System.Data.DataSetExtensions	4.0.0.0
	System.Data.Entity	4.0.0.0
	System.Data.Entity.Design	4.0.0.0
	System.Data.Linq	4.0.0.0
	System.Data.OracleClient	4.0.0.0
	System.Data.Services	4.0.0.0
	System.Data.Services.Client	4.0.0.0

Browse... OK Cancel

Figura 2.13 – Il gestore dei riferimenti di Visual Studio 2015.

Il Reference Manager consente di esplorare e selezionare i riferimenti da aggiungere al progetto. Tali riferimenti possono essere scelti fra gli assembly che fanno parte del .NET Framework, estensioni di terze parti installabili in Visual Studio, esplorando e selezionando un file dal PC, fra le librerie COM registrate sul sistema, oppure indicando altri progetti della stessa soluzione.

Compilazione di soluzione e progetti

Oltre a permettere la scrittura di codice e la progettazione di interfacce grafiche, l'ambiente di sviluppo di Visual Studio consente di compilare e produrre gli assembly che saranno distribuiti con le nostre applicazioni.

Una volta scritto il codice che costituisce il programma mediante l'editor di codice e disegnata l'interfaccia grafica per mezzo della finestra di design, bisognerà provarlo per verificare che tutto funzioni come ci si aspetta (o come si aspetta il vostro cliente!).

Per compilare uno o più progetti che costituiscono una soluzione, Visual Studio mette a disposizione diversi comandi. La modalità più semplice è quella di utilizzare il comando Build Solution, presente all'interno del menu Build (oppure la scorciatoia da tastiera F6). La procedura di build della soluzione procederà a compilare tutti i progetti presenti nella soluzione e, se non ci sono errori che impediscono la compilazione, produrrà gli assembly di ogni progetto; per esempio, nel caso del progetto di applicazione Console si otterrà un file eseguibile dal CLR.

In particolare, gli assembly risultanti dal processo di build verranno salvati all'interno di una cartella bin, posizionata o eventualmente creata da Visual Studio all'interno della directory contenente il progetto o, più esattamente, all'interno di una sottocartella Debug o Release, a seconda della configurazione utilizzata per la compilazione.

In realtà il processo di build può anche avvenire indirettamente, avviando l'esecuzione del programma da Visual Studio stesso per avviarne il test e il debug, come mostrato più avanti nel Paragrafo "Testing e debugging".

Configurazioni di build

Ogni progetto può essere compilato utilizzando diverse opzioni di configurazione, che possono essere salvate in maniera da riutilizzarle agevolmente.

Infatti, sia sulla toolbar principale di Visual Studio sia all'interno della finestra di proprietà di ogni progetto nella scheda Build, potrete notare la presenza di una casella combinata da cui scegliere una particolare configurazione.

In particolare, creando un progetto mediante i template predefiniti di Visual Studio, vengono automaticamente create due configurazioni denominate Debug e Release.

Come indica il nome, la configurazione Debug viene utilizzata durante la fase di test e debugging del codice e produce un codice meno ottimizzato che occupa più memoria, ma che permetterà di eseguire un debugging più efficace, riga per riga, del codice dell'applicazione, perché il file eseguibile o l'assembly contengono le informazioni aggiuntive utili per il debug.

La modalità Release invece è prevista per compilare la versione da rilasciare all'utente, ottimizzando quindi gli assembly generati sia in termini di prestazioni sia di memoria o di spazio occupato.

Il modo più semplice per passare da una modalità di build all'altra è quella di modificarla dalla toolbar di Visual Studio, come mostrato nella Figura 2.14.

La voce Configuration Manager presente nella stessa casella permette di modificare tali configurazioni o crearne di nuove. Per ogni configurazione, nel caso di soluzioni multi-progetto, è possibile indicare quali dei progetti presenti compilare e in che modalità.


Studio
DEBUG
TEAM
TOOLS
ANTS
ARCHITECTURE
TEST
Debug
Any CPU
ConsoleApplication1
Debug
Release
Configuration Manager...
Program.cs*
Class1.cs
ConsoleApp

Figura 2.14 – Scelta della configurazione di build della soluzione.

Non è obbligatorio denominare le configurazioni Debug e Release. In certi casi può essere necessario creare delle ulteriori configurazioni, assegnando loro quindi nuovi nomi. Per esempio, se una soluzione contiene dei progetti che rappresentano versioni diverse di una stessa applicazione, per esempio una per Windows Store e una per Windows Phone, si potrebbero creare configurazioni apposite per eseguire il build della soluzione solo per la versione Store oppure solo per la versione Phone.

Testing e debugging

Durante lo sviluppo di una qualunque applicazione si verificheranno diversi tipi di malfunzionamenti e problemi, alcuni dovuti a errori logici dello sviluppatore, altri a semplici errori di sintassi. Gli errori di sintassi vengono evidenziati al termine della procedura di compilazione in quanto essa non andrà a buon fine.

Finestra degli errori

Gli errori di compilazione, insieme a situazioni meno gravi dette *warning*, verranno elencati all'interno della finestra degli errori, visualizzata in basso, sotto all'editor di codice.

La Figura 2.15 mostra i risultati relativi alla compilazione di una soluzione che produce due errori, indicati dall'icona di errore con la croce, e un avviso, o warning, contrassegnato da un punto esclamativo. La finestra degli errori mostra delle informazioni sull'errore che si è verificato e indica il punto del programma in cui è avvenuto (nome del file, riga e colonna al suo interno). Per correggere tali situazioni bisogna esaminare il punto di origine dell'errore. Facendo doppio clic sulla riga di un errore, Visual Studio aprirà l'editor di codice, posizionando il cursore esattamente nel punto che provoca l'errore in esame. Più esperienza possiede il programmatore, più facile sarà naturalmente la correzione di tali errori. A un occhio esperto basta una frazione di secondo per capire un messaggio di errore e comprendere il motivo che l'ha provocato, quindi sarà in grado rapidamente di apportare le giuste correzioni.

Una volta corretti tutti gli errori, sarà possibile avviare l'esecuzione del programma.

ConsoleApplication1 - Microsoft Visual Studio

FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS ANTS ARCHITECTURE TEST .NET REFLECTOR ANALYZE WINDOW HELP

Debug | Any CPU | ConsoleApplication1 | Start

Form1.cs [Design]* | Program.cs* | Class1.cs

ConsoleApplication1 | ConsoleApplication1.Program | Main(string[] args)

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(arg);

            int b = 0;
            int a = 1 * 10.0;
        }
    }
}
```

100 %

Solution Explorer

Search Solution Explorer (Ctrl+è)

- Solution 'ConsoleApplication1' (3 projects)
  - ClassLibrary1
  - ConsoleApplication1
    - Properties
    - References
    - App.config
    - Program.cs
  - WindowsFormsApplication1
    - Properties
    - References
    - App.config
    - Form1.cs
      - Form1.Designer.cs

Solution Explorer | Team Explorer

Properties

Error List

Entire Solution | 2 Errors | 1 Warning | 0 Messages | Build + IntelliSense

Search Error List

Code	Description	Project	File	Line
CS0103	The name 'arg' does not exist in the current context	ConsoleApplication1	Program.cs	10
CS0219	The variable 'b' is assigned but its value is never used	ConsoleApplication1	Program.cs	12
CS0266	Cannot implicitly convert type 'double' to 'int'. An explicit conversion exists (are you missing a cast?)	ConsoleApplication1	Program.cs	13

Ready | Ln 13 | Col 22 | Ch 22 | INS

Figura 2.15 – Finestra degli errori e avvisi di compilazione.

Esecuzione del programma

È possibile, nel caso di un progetto che produce un'applicazione eseguibile, avviarne l'esecuzione mediante il comando Start Debugging presente nel menu Debug, oppure in maniera ancor più immediata, con un clic sul pulsante con l'icona Play presente nella toolbar principale di Visual Studio o ancora con la scorciatoia da tastiera F5.

L'avvio dell'esecuzione, in genere, avvia il processo di build, visto precedentemente, e quindi, se la compilazione va a buon fine, procede con l'esecuzione vera e propria del programma, partendo dal suo punto di ingresso, cioè dal metodo Main.

La modalità di esecuzione Start Debugging fa in modo che Visual Studio avvii il programma in un nuovo processo, ma esso rimane collegato all'IDE per permetterne il debug (vedere il prossimo paragrafo).

Per uscire da un programma avviato in modalità Debugging, da Visual Studio, è necessario fare clic sul pulsante Stop Debugging (icona con il quadrato simbolo di stop), oppure utilizzare lo stesso comando Stop Debugging che apparirà nel menu Debug.

Naturalmente il programma potrà essere interrotto in maniera autonoma: per esempio, se si fosse avviata un'applicazione a finestra, basterà chiudere quella principale per interromperne il debug e scollegare Visual Studio da questa modalità.

Breakpoint

Il vantaggio di avviare un programma in modalità Debug risulterà evidente quando sarà necessario esaminare errori di logica, che non provocano errori di compilazione ma situazioni anomale durante l'esecuzione del programma stesso. Per esempio errori di calcolo, di visualizzazione, o comunque derivanti da cosiddetti bug.

In tal caso è utile poter seguire l'esecuzione del programma passo passo, cioè istruzione dopo istruzione, impostando dei *breakpoint*, o punti di interruzione, in cui poter esaminare il valore assunto da variabili o da oggetti in quel preciso istante.

Per impostare tali punti di interruzione è possibile utilizzare ancora una volta più modalità. La più semplice è quella di fare un clic con il mouse sulla barra verticale subito a sinistra dell'editor di codice, in corrispondenza dell'istruzione prima della quale si vuol interrompere l'esecuzione (vedi Figura 2.16). Una volta impostato un punto di interruzione apparirà un pallino. Per rimuovere un breakpoint basta fare clic nuovamente sullo stesso.

Una seconda possibilità per aggiungere e rimuovere i breakpoint è usare i comandi nel menu Debug oppure la scorciatoia da tastiera F9. Se a questo punto si avvia il programma in

modalità Debug, una volta raggiunto un punto di interruzione, Visual Studio mette in pausa l'esecuzione, evidenziando in giallo nell'editor di codice la successiva istruzione che verrà eseguita.

In questo modo è possibile esaminare cosa sta accadendo nel programma, dal vivo durante la sua esecuzione. In particolare è possibile visualizzare i valori delle variabili semplicemente posizionandosi su di esse con il mouse, oppure utilizzando le finestre di ispezione, che permettono di seguire l'andamento delle variabili nel tempo o, meglio, eseguendo le istruzioni una dopo l'altra.

Program.cs*

ConsoleApplication1.Program | Main(string

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string hello="hello world";
            Console.WriteLine(hello);
        }
    }
}
```

BREAKPOINT

Figura 2.16 – Impostazione di un breakpoint.

Infatti, una volta interrotto a un dato breakpoint, il programma potrà essere riavviato, oppure eseguito istruzione dopo istruzione, mediante i comandi Step Into (F11), Step Over (F10) e Step Out (Shift+F11).

Il primo comando permette di eseguire la successiva istruzione e, nel caso in cui essa è l'invocazione di un metodo, di entrare al suo interno per eseguire le relative istruzioni passo passo.

La seconda modalità, invece, esegue la successiva istruzione, ma non entra all'interno di un metodo, trattandolo come se fosse una singola istruzione.

L'ultima, infine, permette, nel caso in cui si stia eseguendo il debug di un metodo, di saltare il resto delle sue istruzioni e passare alla prima istruzione dopo il suo completamento.

Visual Studio permette di apportare modifiche al codice in diretta, anche mentre è in esecuzione il debug. Tale possibilità consente di ispezionare variabili e di correggere eventuali errori senza dover rieseguire il build e riavviare l'esecuzione del programma. Infatti, sempre mentre ci si trova in modalità Debug, è possibile spostare il punto di esecuzione e indicare quale sarà la successiva istruzione che si vuole eseguire, eventualmente indicandone anche una più indietro, già eseguita.

Per fare ciò basta utilizzare la freccia che si trova sulla colonna in grigio alla sinistra dell'editor e trascinarla con il mouse verso la riga che si vuol indicare come successiva istruzione.

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string hello = "hello world";
            Console.WriteLine(hello);
        }
    }
}
```

100 %

Locals

Name	Value	Type
args	{string[0]}	string[]
hello	"hello world"	string

Figura 2.17 – Interruzione di un programma in corrispondenza di un breakpoint.

Un'altra possibilità per variare il flusso di esecuzione del programma è utilizzare i diversi comandi disponibili nel menu contestuale, che appare cliccando con il tasto destro sull'editor di codice quando l'esecuzione è stata interrotta da un breakpoint.

In tale menu, fra i comandi più utili è possibile, per esempio, selezionare la voce Run To Cursor, che farà continuare l'esecuzione fino al punto in cui si trova il cursore, oppure Show Next Statement, che mostrerà l'istruzione successiva che verrebbe eseguita riprendendo l'esecuzione oppure eseguendone uno step.

Finestre di ispezione

Durante l'esecuzione in modalità Debug, una volta raggiunto un punto di interruzione, è possibile monitorare il valore delle variabili o delle proprietà di un oggetto utilizzando delle apposite finestre di ispezione. Esse sono in particolare di tre tipologie differenti:

- Local – la finestra che mostra il valore delle variabili locali, accessibili dal punto di esecuzione attuale del programma;
- Auto – una finestra automatica che mostra il valore delle ultime variabili e oggetti utilizzati durante l'esecuzione;
- Watch – una o più finestre che permettono di inserire esplicitamente il nome delle variabili che si vogliono analizzare. Per aggiungere una finestra di Watch basta fare clic con il pulsante destro su una delle finestre di ispezione e selezionare il comando Add Watch.

Le finestre di ispezione sono visibili solo quando il programma è in esecuzione in modalità Debug.

```
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string hello="hello world";
            object o = hello;
            Console.WriteLine(hello);
        }
    }
}
```

100 %

Locals

Name	Value	Type
args	{string[0]}	string[]
hello	"hello world"	string
o	"hello world"	object {s
$a	0	int

Call Stack

Name	Lang
ConsoleApplication1.exe!ConsoleApplication1.Prog	C#
[External Code]	

Autos | Locals | Watch 1 | Find Symbol Results | Call Sta... | Breakp... | Comm... | Immed... | Output | Error List

Figura 2.18 – Finestre di ispezione delle variabili e di debug.

Altre finestre utili per il debugging di un'applicazione sono visualizzate sulla destra, in genere affiancate a quelle di ispezione appena elencate. Esse consentono, per esempio, di visualizzare lo stack delle chiamate, l'elenco dei breakpoint attivi, inserire delle istruzioni C# oppure visualizzare delle variabili digitandone il nome in una finestra *Immediate*.

Progetto di avvio

Una soluzione, come detto, contiene in genere diversi progetti, quindi potrebbe anche contenere diversi progetti eseguibili; per esempio, supponiamo di voler sviluppare un'applicazione in due versioni, una per Windows 8 e una per Windows Phone, che sfruttino magari una libreria comune. Entrambe le applicazioni possono essere debuggate ed eseguite, anche contemporaneamente.

Se si vuol avviare uno solo dei due progetti, è possibile fare clic con il tasto destro sul progetto da avviare e selezionare la voce Start New Instance, all'interno del sottomenu Debug.

Un'altra possibilità per impostare il progetto di avvio da utilizzare quando si eseguirà il debug, per esempio con il tasto F5, è di selezionare invece il comando Set As Startup Project.

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 nomi, verbi, 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, per questo motivo, la sintassi è basata sui blocchi di codice delimitati da parentesi graffe e istruzioni terminate dal punto e virgola. In tal senso C# è simile 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: è utile solo per permettere a se stessi o ad altri sviluppatori di leggere e capire il significato del codice stesso in maniera più immediata.

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

```
using System;namespace Capitolo2_ConcettiDiBase{ public class Program { static void Main(string[] args) {
Console.WriteLine("Hello World");}
```

Si risparmierebbe certamente spazio, ma la leggibilità del codice sarebbe come minimo 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:

```
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, che servono a inserire dei 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:

```
// questo 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 cosa si riferisce il commento stesso.

La seconda modalità per scrivere commenti in C# invece 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 alla virgola 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 elementi di 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 così ottenuto può essere poi elaborato mediante appositi strumenti per generare della documentazione utilizzabile e distribuibile in svariati formati: CHM, HTML, word, pdf e così via.

Le variabili

Nell'esempio Hello World che abbiamo scritto qualche paragrafo fa non era presente alcuna forma di memorizzazione di dati. In generale in un qualunque programma che abbia un obiettivo pratico sarà necessario immagazzinare dei dati, elaborarli o usarli per dei calcoli, permettere all'utente di inserirli mediante la tastiera o leggerli sul monitor. Quindi avremo bisogno di una sorta di contenitore in cui conservare dei dati, che viene chiamato *variabile*.

C# è un linguaggio fortemente tipizzato, quindi per dichiarare una variabile è necessario definire il tipo di dati che essa può memorizzare e assegnarle un identificatore, cioè un nome con il quale successivamente si potrà leggere o scriverne il contenuto.

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 darglielo mediante un'istruzione di *assegnazione*, che avviene usando l'operatore =, con la seguente sintassi:

```
anno=1975;
```

Se si tentasse di utilizzare in un'istruzione qualsiasi (esclusa quella di assegnazione naturalmente!) una variabile non inizializzata, il compilatore restituirebbe 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.

Come già accennato, 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 rende C# un linguaggio con caratteristiche dinamiche. Si vedrà più in dettaglio nel Capitolo 14 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 di 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 \u005fIdentificatore;//equivalente a _Identificatore
```

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

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

```
int aNNo;
```

Naturalmente è una pratica sconsigliata quella di dichiarare e distinguere degli identificatori solo per mezzo delle maiuscole o minuscole, 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 creato (o comunque anche in questo caso, se è passato un certo periodo di tempo).

Esistono quindi delle 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 camelCase, invece, la prima lettera dell'identificatore rimane minuscola:

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

Per le variabili in C# è quasi unanimemente utilizzata la convenzione camelCase, mentre in qualche caso, spesso per le variabili di classe, si fa iniziare l'identificatore con un _, 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 di utilizzare degli 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 dei nomi attinenti a ciò che andranno a memorizzare. Quindi scrivere:

```
double base, altezza;
…
```

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

è certamente meglio che usare delle variabili senza 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 solo su 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 nel resto dei capitoli vedremo come un'applicazione C# sia composta da una collezione di tipi. In questo e nei prossimi paragrafi elencheremo i tipi di dati 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 dei 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 dei 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 degli alias dedicati, che però sono più di semplici abbreviazioni o nomi. In particolare abbiamo già accennato nel Capitolo 1, 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 degli 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 2.2 - **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 necessario 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, potrò memorizzare 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 quattro 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 riescono, 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 che effettuano calcoli astronomici.

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 la mia applicazione necessita 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, in cui la maggior parte di essi rimarrebbero 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, come da tabella seguente.

Tabella 2.3 - **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 float 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 esso 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 dei numeri, la precisione dei numeri a virgola mobile non è esatta, soprattutto effettuando operazioni aritmetiche su di essi. 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 necessita 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 2.4 - **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 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, e la maggior parte dei testi lo fanno, 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 2.5 - **Tipo booleano di C#.**

Nome	Tipo CTS	Descrizione	Valori possibili
bool	System.Booleano	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. Esso è utilizzato 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 5 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 2.6 **- 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 utilizzati 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 delle *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 2.7 - **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 sicuramente 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 dei 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. Questo potrebbe sembrare una stranezza e un grosso 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 "HELLO"
Console.WriteLine(str1); //stampa "hello"
```

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 panoramica relativa al tipo System.String e ai suoi metodi e proprietà; non l'abbiamo inserito qui perché ancora ci mancano diverse nozioni per capire 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.

Naturalmente chi non conosce il paradigma di programmazione orientata agli oggetti non capirà il senso di classe madre e del termine "derivare da essa"; per il momento non resta che fidarsi aspettando di leggere il 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 dei metodi generali che ogni altro tipo avrà dunque a disposizione.

Valori letterali

Per rappresentare dei valori in C# e assegnarli a delle 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 è 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';
```

Avrete notato che già negli esempi precedenti abbiamo utilizzato qualcuno di essi, ma in questo paragrafo si farà un rapido riepilogo, in maniera da poterlo adoperare come riferimento dei valori letterali utilizzabili con i vari tipi semplici di C#, indicando 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 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 dei numeri reali e utilizzarli come float, double o decimal, è possibile utilizzare un numero composto da una 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, posizionando quest'ultimo dopo 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 1.23×10^5
double d= 4.5e-2; // è pari a 4.5×10^{-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.

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 prevedono la rappresentazione mediante il valore esadecimale oppure 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 (vedere la Tabella 2.6).

I namespace

Un *namespace*, o spazio dei nomi, è un contenitore logico per raggruppare dei tipi e organizzarli in maniera da creare una struttura ordinata.

Il .NET Framework contiene nella sua libreria standard migliaia di classi, ognuna delle quali appartiene a un determinato settore di utilizzo: manipolazione di stringhe, input/output, networking, sicurezza, XML, collezioni, e in questo caso è proprio il caso di dire chi più ne ha più ne metta.

Questa abbondanza però può provocare due diversi problemi: il primo è quello di rintracciare la classe che può fare al caso nostro per risolvere un determinato compito, il secondo è trovare un nome univoco. Inserendo invece ogni classe in un proprio namespace si risolvono entrambe le questioni.

Il concetto è analogo a quello delle directory del file system. In ogni cartella possiamo inserire un insieme di file e sotto-directory, a patto che ognuno di essi abbia un nome univoco all'interno della cartella che lo contiene.

Tutte le classi standard della Base Class Library sono organizzate a partire dal namespace System e nei suoi sotto-namespace. Per esempio la classe Console è contenuta direttamente nel namespace System, mentre la classe File è contenuta nel namespace IO, contenuto a sua volta in System.

NOTA	Non è obbligatorio che una classe sia contenuta in un namespace, anche se è estremamente consigliato proprio per i motivi appena esposti e, soprattutto, in progetti medio-grandi.

Il *nome completo* di una classe è ottenuto facendo precedere il nome della classe stessa dall'intero percorso dei namespace in cui si trova, usando il punto come separatore; le classi Console e File avranno quindi i seguenti nomi completi:

```
System.Console
System.IO.File
```

In tal modo è possibile avere due classi con lo stesso nome all'interno di namespace diversi. Potremmo per esempio definire una nostra classe Console, ma all'interno di un namespace diverso da System.

Per definire un namespace basta dichiararlo utilizzando la parola chiave namespace e inserire poi i tipi che esso dovrà contenere all'interno del blocco delimitato dalle parentesi graffe che seguono la dichiarazione:

```
namespace MioNamespace
{
//contenuto del namespace
class MiaClasse
{}
}
```

A questo punto il nome completo della classe MiaClasse sarà MioNamespace.MiaClasse.

È anche possibile innestare i namespace uno dentro l'altro, creando così una struttura gerarchica:

```
namespace MioNamespace
{
namespace SottoNamespace
{
class MiaClasse
{}
}
}
```

Lo stesso risultato può comunque essere ottenuto scrivendo direttamente il nome completo del namespace, utilizzando il punto (.) come separatore:

```
namespace MioNamespace.SottoNamespace
{
class MiaClasse
{}
}
```

Ora il nome completo della classe è MioNamespace.SottoNamespace.MiaClasse.

NOTA	Esistono delle convenzioni per denominare i namespace in cui inserire i propri tipi, in maniera da renderli univoci e riconoscibili come appartenenti a un dato progetto o assembly. Innanzitutto viene in genere utilizzato il Pascal Casing. Un formato spesso utilizzato e anche consigliato dal team di sviluppo di .NET prevede la seguente struttura: <Company>.(<Product>|<Technology>)[.<Feature>][.<Subnamespace>]. Per esempio Microsoft ha creato un namespace Microsoft.Office.Tools.Excel che contiene classi utili a interagire con il modello a oggetti di Excel.

Utilizzare i namespace

Per evitare di dover utilizzare il nome completo per riferirsi a uno o più tipi, è possibile utilizzare l'istruzione using seguita dal nome del namespace. Per esempio, dato che in quasi ogni programma C# si farà uso delle classi del namespace System, all'inizio di ogni file sorgente è presente l'istruzione:

```
using System;
```

L'istruzione using indica al compilatore che si vogliono "importare" le classi presenti nel namespace indicato; in tal modo, anziché utilizzare il nome System.Console.WriteLine per scrivere sulla console, basterà scrivere solo Console.WriteLine:

```
using System;
class Program
{
static void Main()
{
Console.WriteLine("hello world");
}
}
```

Includendo l'istruzione using prima di ogni altra istruzione, tutti i tipi importati saranno utilizzabili nell'intero file.

NOTA	L'istruzione using è concettualmente analoga all'istruzione include presente per esempio in C/C++, ma a differenza di questa non importa fisicamente i namespace indicati. Quindi non è necessario preoccuparsi se ci sono delle istruzioni using ridondanti, magari non utilizzate nel codice del file corrente, perché in tal caso saranno semplicemente ignorate dal compilatore.

Una seconda modalità permette di annidare le istruzioni using, scrivendole subito dopo l'inizio di un altro namespace. In questo modo, l'importazione dei tipi riguarderà solo il namespace in oggetto:

```
namespace A
{
using System;
//qui si possono usare i tipi di System
}
namespace B
{
//qui non è attiva l'istruzione using System
}
```

L'istruzione using static, introdotta con C# 6, permette invece di abbreviare le istruzioni, consentendo l'omissione non solo di un namespace, ma anche del nome di un determinato tipo. Per esempio se si vuole utilizzare il metodo WriteLine senza specificare il nome della classe System.Console, si può scrivere:

```
using static System.Console;
class Program
{
static void Main()
{
WriteLine("hello world");
}
}
```

NOTA	L'istruzione using static può essere utilizzata solo per importare metodi e proprietà statiche, di cui vedremo più avanti il significato esatto.

All'interno del blocco di un namespace è possibile utilizzare qualsiasi altro tipo definito nello stesso namespace, senza necessità di istruzioni using, perché il namespace corrente è implicitamente importato.

Namespace, file e assembly

I namespace non sono assolutamente correlati ai file e alle directory sul file system, anzi differenti namespace sono spesso contenuti in uno stesso assembly, oppure uno stesso namespace potrebbe essere presente in diversi assembly (naturalmente con tipi diversi in questo secondo caso).

Questa seconda affermazione deriva dal concetto stesso di namespace e lo spazio di dichiarazione di un namespace è *open ended*, cioè non chiuso; in parole povere per aggiungere nuovi tipi a un namespace basta semplicemente utilizzare nuovamente lo stesso namespace.

Due blocchi di namespace con lo stesso nome contribuiscono a creare lo stesso spazio di dichiarazione di un namespace, per esempio:

```
namespace MioNamespace
{
class MiaClasse1
{}
}
namespace MioNamespace
{
class MiaClasse2
{}
}
```

Le due dichiarazioni di namespace contengono due classi differenti all'interno dello stesso namespace MioNamespace. Esse possono trovarsi all'interno dello stesso file, in file differenti o anche in assembly differenti, ma dal punto di vista logico le classi MiaClasse1 e MiaClasse2 si trovano all'interno dello stesso namespace.

NOTA	Per compilare un programma C# che utilizza dei namespace, oltre a inserire le opportune istruzioni using, è necessario che il compilatore sappia anche all'interno di quali assembly ritrovare namespace e classi utilizzate nel codice, e ciò viene indicato per mezzo dell'opzione /reference:nomeassembly del compilatore csc oppure aggiungendo i riferimenti opportuni al progetto in Visual Studio.

In generale esiste una correlazione fra il nome di un namespace e il nome dell'assembly che lo contiene. Per esempio, il namespace System.Xml e i sotto-namespace si trovano all'interno dell'assembly System.Xml.dll, ma lo stesso assembly contiene anche tipi facenti parte di System e di System.Configuration.

Alias di namespace

Anche facendo uso dei namespace per organizzare il proprio codice potrebbero comunque capitare delle ambiguità.

Se, per esempio, all'interno dello stesso file di codice sorgente avessimo bisogno di utilizzare due classi con lo stesso nome, ma facenti parte di namespace differenti, la prima soluzione sarebbe utilizzare il nome completo delle due classi, o perlomeno utilizzare la parola chiave using per la classe di un primo namespace e poi utilizzare il nome completo per la classe contenuta nel secondo namespace.

Facciamo un esempio pratico: il .NET Framework contiene almeno due classi denominate Image, una contenuta nel namespace System.Drawing, che rappresenta i dati relativi ai pixel di un'immagine, e una seconda nel namespace System.Windows.Controls, che invece rappresenta un controllo grafico da utilizzare per visualizzare un'immagine all'interno di una finestra di un'applicazione Windows.

Capita abbastanza frequentemente che all'interno dello stesso codice ci si debba riferire a entrambi i namespace, utilizzando l'istruzione using:

```
using System.Drawing;
using System.Windows.Controls;
```

A questo punto, però, utilizzando il nome Image per creare un oggetto, il compilatore non riuscirebbe a distinguere fra le due classi suddette. Dovremo quindi utilizzare il nome completo, per esempio System.Drawing.Image, oppure utilizzare l'istruzione using per creare un alias del namespace:

```
using Draw=System.Drawing;
using WinControls=System.Windows.Controls;
```

In tal modo possiamo abbreviare il nome completo delle classi e scrivere per esempio:

```
new Draw.Image(); //crea un oggetto System.Drawing.Image
```

Le keyword

Esistono delle parole riservate che non possono essere utilizzate come identificatori, in quanto costituiscono delle parole chiave, o *keyword*, utilizzate dal linguaggio C#. Esse costituiscono il vocabolario del linguaggio stesso e sono di seguito elencate come riferimento.

Tabella 2.8 - **Keyword del linguaggio C#.**

abstract	as	base	bool	break
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	do	double	else	enum
event	explicit	extern	false	finally
fixed	float	for	foreach	goto
if	implicit	in	int	interface
internal	is	lock	long	namespace
new	null	object	operator	out
override	params	private	protected	public
readonly	ref	return	sbyte	sealed
short	sizeof	stackalloc	static	string
struct	switch	this	throw	true
try	typeof	uint	ulong	unchecked
unsafe	ushort	using	virtual	void
volatile	while			

Se, per qualche motivo, si rendesse necessario l'utilizzo di una delle parole chiave come identificatore è possibile evitare il conflitto facendo precedere l'identificatore dal carattere @. Per esempio, è possibile dichiarare una variabile così:

```
int @class;
```

Il carattere @ non fa parte del nome della variabile, quindi se ho una variabile denominata:

```
int miavariabile;
```

essa è perfettamente equivalente e interscambiabile con:

```
int @miavariabile;
```

NOTA	L'impiego del carattere @ per utilizzare una keyword come identificatore può essere utile qualora, grazie alle capacità multilinguaggio di C#, si dovesse rendere necessario l'utilizzo di una libreria scritta in un altro linguaggio, in cui l'identificatore non è invece una keyword.

Un identificatore preceduto dal prefisso @ è anche detto identificatore *verbatim*. Altre parole sono dette keyword contestuali perché appunto sono riservate a specifici contesti.

NOTA	Tutte le parole chiave di C# sono state definite nella versione 1.0 del linguaggio. Altre keyword aggiunte in seguito sono solo di tipo contestuale, per evitare che programmi scritti con versioni vecchie di C#, in cui magari un identificatore non era una parola chiave, cominciassero a dare errori di compilazione con versioni più recenti del compilatore.

Al di fuori di tali contesti esse possono essere utilizzate come identificatori, anche senza utilizzare il carattere @.

Tabella 2.9 - **Keyword contestuali del linguaggio C#.**

add	ascending	async	await	by
descending	dynamic	equals	from	get
global	group	in	into	join
let	nameof	on	orderby	partial
remove	select	set	value	var
when	where	yield		

Per esempio, incontreremo spesso le parole chiave get e set nella dichiarazione delle proprietà di una classe (se non sapete cos'è una proprietà, fidatevi!), ma al di fuori del contesto di proprietà si può anche utilizzarle per definire il nome di una variabile:

```
int get=1;
int set=2;
```

Alcune parole chiave possono essere utilizzate in diversi contesti, per esempio partial può essere utilizzata per definire un tipo parziale oppure un metodo.

Istruzioni

Un'istruzione è un pezzo di codice sorgente che permette di definire un'azione da eseguire nel programma.

In generale, un'istruzione è una porzione di codice che termina con un punto e virgola (;) ma ciò vale solo per istruzioni semplici, come possono esserlo quelle di dichiarazione delle variabili:

```
int i; //istruzione di dichiarazione di una variabile
```

oppure l'istruzione di invocazione di un metodo come quello di scrittura del testo per mezzo del metodo WriteLine:

```
Console.WriteLine("hello world");
```

Le istruzioni di un programma vengono eseguite in generale in sequenza, quindi se ho tre righe con le seguenti tre istruzioni:

```
int a=1;
int b=2;
int somma=a+b; //espressione per il calcolo dell'addizione
```

esse verranno eseguite una dopo l'altra, assegnando prima il valore 1 alla variabile a, poi il valore 2 a b, e infine calcolando la somma per mezzo della terza e ultima istruzione.

Ogni istruzione che usa delle variabili ha bisogno quindi che le stesse variabili siano state definite in un'istruzione precedente.

Scrivendo le tre righe precedenti in ordine inverso si avrebbe un errore di compilazione perché, per eseguire l'operazione di somma, le variabili a e b devono già essere conosciute e quindi definite in un'istruzione eseguita in precedenza.

In C# esistono però svariati tipi di istruzioni, più o meno complesse. Per esempio, è possibile scrivere delle istruzioni che consentono di variare il flusso di esecuzione del programma, effettuando delle scelte per saltare una parte di codice, oppure per eseguire ciclicamente le stesse istruzioni un certo numero di volte, per valutare delle condizioni e così via. Torneremo su tali concetti parlando di controllo di flusso nel Capitolo 5.

Blocchi di codice

Oltre a essere costituito da istruzioni separate dal punto e virgola, un programma C# è strutturabile in blocchi di codice. Ogni blocco è delimitato da una parentesi graffa di apertura { e una di chiusura } e al suo interno è possibile inserire una serie di istruzioni oppure altri blocchi.

Un blocco costituisce esso stesso un'istruzione, ma non è necessario che un blocco sia poi chiuso da un punto e virgola per separarlo da istruzioni o blocchi che lo seguono.

NOTA	Se un blocco è chiuso da un ; non si verifica nessun errore di compilazione, ma questo solo perché il ; viene considerato come un'istruzione vuota, senza alcun effetto. Anche per questione di stile è però sconsigliato mettere un punto e virgola alla fine di un blocco.

Il seguente è un esempio di blocco di codice C#, contenente un'istruzione e un altro sottoblocco con altre due istruzioni:

```
{
Istruzione1;
{
Istruzione2;
Istruzione3;
}
}
```

Si noti come in questo caso, sebbene gli spazi e le tabulazioni non concorrano a determinare o modificare la logica di esecuzione del programma, l'indentazione di istruzioni e blocchi serva a strutturare graficamente il codice sorgente per renderlo più leggibile.

Ambito delle variabili

In C# l'ambito o scope di una variabile rappresenta la zona del programma in cui essa è visibile e può essere utilizzata. Il concetto di ambito o scope è strettamente correlato a quello di blocco; infatti, in generale, l'ambito di una variabile è determinato dalle seguenti regole:

- l'ambito di una variabile locale coincide con il blocco in cui essa è dichiarata, quindi si estende dalla parentesi graffa di apertura fino alla prima parentesi graffa che chiude il blocco, per esempio la chiusura di un metodo;
- una variabile definita in un ciclo è invece visibile all'interno del blocco definito dal ciclo stesso (vedremo come, per esempio, un ciclo `for` o un ciclo `while` siano definiti anch'essi come blocchi delimitati da `{` e `}`);
- una variabile di classe, detta anche campo della classe, è invece visibile all'interno della classe stessa.

All'interno di ogni blocco di codice il nome assegnato a una variabile deve essere univoco, cioè non è possibile dichiarare due variabili con lo stesso nome all'interno di uno stesso blocco:

```
{
int variabile=0;
string variabile=""; //errore, il nome variabile è già usato in questo blocco
}
```

Se invece avessimo due blocchi differenti potremmo utilizzare lo stesso nome, perché non ci sarebbe alcun conflitto di ambito, ogni variabile vive nel suo mondo:

```
{
int variabile=0;
}
{
string variabile=""; //ok, il nome variabile è usato in un altro blocco
}
```

Una variabile definita all'interno di un blocco estende il suo ambito anche all'interno di blocchi annidati all'interno del blocco precedente. Per esempio:

```
{
int a=0;
{
int b=a+2; //la variabile a è visibile
}
b=1; //errore, lo scope di b termina con la parentesi graffa precedente
}
```

La variabile `a` nell'esempio precedente viene definita nel primo blocco. Con la seconda parentesi graffa inizia un nuovo blocco, innestato nel precedente. La variabile `a` è quindi visibile all'interno di questo secondo blocco e può essere utilizzata per scrivere l'espressione `a+2`.

Con la prima parentesi graffa chiusa termina il blocco innestato, quindi termina lo scope della variabile `b`, che non può più essere utilizzata nel blocco esterno.

Se invece a questo punto si provasse a dichiarare una nuova variabile `a` all'interno del blocco annidato, come nel seguente esempio:

```
{
int a = 0;
{
string a = ""; //errore la variabile a è già dichiarata
}
}
```

il compilatore restituirebbe un errore, indicando che non è possibile dichiarare una variabile denominata `a` perché darebbe un altro significato alla variabile `a` denominata in un ambito padre.

NOTA	Il comportamento in C# è differente da quello di C++, in cui avviene il cosiddetto *hiding* dei nomi; in C++ infatti è lecito dichiarare una variabile in un blocco interno: in questo caso si dice che la variabile interna *nasconde* quella esterna.

Costituisce un'eccezione invece il caso di un campo di classe, in quanto in un metodo è possibile definire una nuova variabile con il nome uguale a quello del campo, e in questo caso la variabile locale nasconderebbe quella di classe.

Come ultimo esempio consideriamo ora una classe in cui potremmo aver definito due metodi come i seguenti:

```
static void Metodo1()
{
int a = 1;
}
static void Metodo2()
{
Console.WriteLine(a);
}
```

La variabile a è locale al metodo Metodo1, quindi il suo ambito è confinato fra le parentesi graffe che costituiscono il corpo del metodo. Essa non è visibile all'interno del metodo Metodo2, che costituisce un altro ambito.

Variabili di tipo implicito

A partire da C# 3.0 è possibile evitare di dichiarare il tipo di una variabile locale in maniera esplicita.

Per mezzo della keyword contestuale var si può dichiarare una variabile e inizializzarla mediante un'istruzione come la seguente:

```
var str="hello";
```

Anche così C# continua a rispettare le regole di linguaggio fortemente tipizzato, in quanto il compilatore si occupa di determinare quale tipo di dato viene assegnato alla variabile. Quindi l'istruzione precedente è perfettamente equivalente a:

```
string str="hello";
```

A seguito dell'assegnazione del valore, il tipo della variabile non può essere più modificato. Per esempio se a questo punto si scrivesse una nuova istruzione:

```
str=1;
```

si otterrebbe un errore di compilazione.

L'utilizzo di var è permesso solo sotto determinate condizioni. La prima è che la variabile deve essere inizializzata, non può essere solo dichiarata, perché il tipo deve essere chiaramente determinato:

```
var i=0; //ok
var i; //errore
```

Il valore assegnato non può essere null (null rappresenta un riferimento a nessun oggetto, vedere il Capitolo 3):

```
var oggetto = null; //errore
```

Infine l'inizializzatore deve essere un'espressione (vedere il prossimo paragrafo), sempre perché deve essere possibile determinare con esattezza il tipo del valore assegnato alla variabile.

Il vantaggio di utilizzare la parola chiave var è innanzitutto quello di poter dichiarare una variabile senza dover ripetere il tipo:

```
System.Int32 i=new System.Int32();
var i=new System.Int32();
```

In altri casi il vantaggio è ancora maggiore, in quanto il tipo restituito da una certa operazione non è sempre facilmente determinabile a occhio nudo.

Direttive di preprocessore

Oltre alle parole chiave del linguaggio, è possibile utilizzare in C# dei comandi conosciuti come direttive di preprocessore. Esse non definiscono il comportamento del programma in se stesso, ma istruiscono il compilatore su come eseguire il suo compito.

Per esempio è possibile indicare che una certa porzione di codice debba essere compilata o meno in base alla definizione di un simbolo, anch'esso definito per mezzo di un'apposita direttiva. Questi comandi o direttive sono distinguibili dal resto delle istruzioni standard di C# perché iniziano con il carattere #.

Tabella 2.10 - **Direttive di preprocessore.**

Direttiva	Descrizione
#define sym	definisce il simbolo sym
#undef sym	elimina la definizione del simbolo sym
#if sym	verifica se il simbolo sym è definito
#else	definisce un ramo alternativo all'#if
#endif	identifica la fine di un blocco #if
#elif sym	combina #else con un ulteriore #if
#warning msg	genera un avviso in fase di compilazione
#error msg	genera un errore di compilazione
#pragma warning	sopprime o ripristina avvisi di compilazione
#line	permette di ridefinire numero di linea o nome del file utilizzato da avvisi ed errori di compilazione
#region nome	identifica l'inizio di una regione
#endregion	identifica la fine della regione iniziata con la precedente direttiva #region

Nei prossimi paragrafi sono descritte le direttive più utilizzate.

Le direttive #define e #undef

La direttiva #define permette di definire l'esistenza di un simbolo di preprocessore. Per esempio scrivendo:

```
#define DEBUG
```

si indica al compilatore che un simbolo con nome DEBUG è stato definito.

Al contrario, per rimuovere la definizione di un simbolo definito in precedenza, si utilizza la direttiva #undef:

```
#undef DEBUG
```

Le direttive #if, #else, #elif, #endif

Queste direttive servono per indicare al compilatore quali parte del codice C# compilare, in base all'esistenza o meno di un simbolo definito con l'istruzione #define. Per esempio, se si vuol compilare un'istruzione solo in modalità Debug si può definire un simbolo DEBUG come nel precedente paragrafo e quindi verificare la sua esistenza mediante la direttiva #if:

```
#if DEBUG
Console.WriteLine("modo debug");
#endif
```

Se il simbolo non è definito, il compilatore semplicemente ignora tutto il codice contenuto fra #if e #endif.

La sintassi completa di queste direttive è la seguente:

```
#if DEBUG
Console.WriteLine("modo debug");
#elif RELEASE
Console.WriteLine("modo release");
#else
Console.WriteLine("debug e release non definite");
#endif
```

Questa tecnica di includere o escludere codice dalla compilazione, in base alla definizione di determinati simboli, è detta *compilazione condizionale.*

È possibile anche combinare e verificare più condizioni all'interno di un'unica direttiva. Per esempio, per verificare se entrambi i simboli SYM1 e SYM2 sono definiti possiamo usare una direttiva come la seguente:

```
#if SYM1 && SYM2
```

L'operatore && nella precedente direttiva indica un AND logico.

Se invece si vuol verificare se almeno uno dei due simboli è definito, si può utilizzare l'operatore ||, che indica l'OR logico:

```
#if SYM1 || SYM2
```

Un simbolo è considerato avere valore true se esso è stato definito, altrimenti false. Quindi è possibile, per esempio, scrivere una direttiva:

```
#if SYM1 || (SYM2 == false)
```

Le direttive #region e #endregion

Queste direttive non influenzano la compilazione, ma servono semplicemente a raggruppare delle porzioni di codice all'interno di una regione, assegnando a essa un nome. Per esempio:

```
#region variabili
int a;
int b;
#endregion
```

In tal modo si è definita una regione denominata variabili, all'interno della quale verranno inserite le dichiarazioni delle variabili.

Queste direttive sono particolarmente utili e utilizzate in ambienti di sviluppo come Visual Studio, che evidenzia le regioni di codice e permette di espanderle e comprimerle utilizzando il mouse.

Le direttive #warning e #error

La direttiva #warning consente di generare un avviso, con un determinato messaggio di testo, in una precisa riga del codice, magari in base a una condizione definita con #if. Per esempio:

```
#if DEBUG
#warning questo file non deve essere utilizzato in DEBUG
#endif
```

La direttiva #error invece genera un errore, interrompendo il processo stesso di compilazione:

```
#if DEBUG
#error questo file non deve essere utilizzato in DEBUG
#endif
```

Input e output da riga di comando

Per poter iniziare a sviluppare in C# e assimilare i concetti esposti, è necessario metterli in pratica scrivendo più codice possibile, compilando e imparando dagli errori.

Il programma Hello World realizzato all'inizio del capitolo ha come solo scopo quello di stampare una stringa di testo sulla console.

Prima di passare ad argomenti più complessi, vedremo quindi in questo paragrafo come interagire con l'utente in un'applicazione di tipo Console (eseguita cioè all'interno del prompt dei comandi), stampando i risultati e leggendo l'input inserito mediante la tastiera.

Ciò sarà utile soprattutto nei prossimi capitoli del libro, ma anche nel momento in cui vorrete sfruttare, ai fini del debug, il buon vecchio metodo di stampare i valori delle variabili oppure visualizzare dei messaggi per riconoscere il punto di esecuzione del programma che state testando.

La classe Console

La classe Console presente nella Base Class Library, nel namespace System, fornisce tutti i metodi necessari per l'input di dati e per l'output dei risultati su una finestra della console.

La classe Console è una classe statica; per ora vi basti sapere che ciò implica che non è possibile creare un'istanza della classe su cui invocare i metodi e le proprietà. In questo caso, cioè, basta usare il nome della classe e poi il nome del metodo da invocare o della proprietà, separandoli con un punto (.).

Per esempio, è possibile impostare il colore di sfondo della finestra e quello con cui verrà stampato il testo, utilizzando le due proprietà BackgroundColor e ForegroundColor:

```
Console.BackgroundColor = ConsoleColor.White; //imposta il colore bianco come sfondo
Console.Clear();
Console.ForegroundColor = ConsoleColor.Blue; //stampa il testo in blu
```

Impostando il colore di sfondo viene modificato solo il colore dietro a ogni carattere, mentre la finestra intera resterebbe di colore nero. Per colorare di bianco tutto lo sfondo viene utilizzato nell'esempio precedente il metodo Clear.

NOTA	I valori possibili utilizzabili come colori sono ricavati dal tipo ConsoleColor, che è un tipo enum. Esso cioè fornisce un'enumerazione di possibili valori. Si tornerà sul concetto di enum nel Capitolo 3.

Output su console

Per scrivere del testo sulla console, il primo metodo è quello denominato Write, che altro non fa che stampare il valore passato al metodo come parametro. Per esempio, invocando il metodo nel seguente modo:

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

verrà stampata sulla console il testo “Hello World” passato come argomento.

Il metodo Write invocato più volte consecutivamente non fa andare a capo il testo, ma continua a stampare a partire dal punto precedente. Prendiamo il seguente esempio:

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

Esso stamperà le tre stringhe, una per ogni chiamata, una dopo l’altra, formando le famigerate parole Hello World separate da uno spazio.

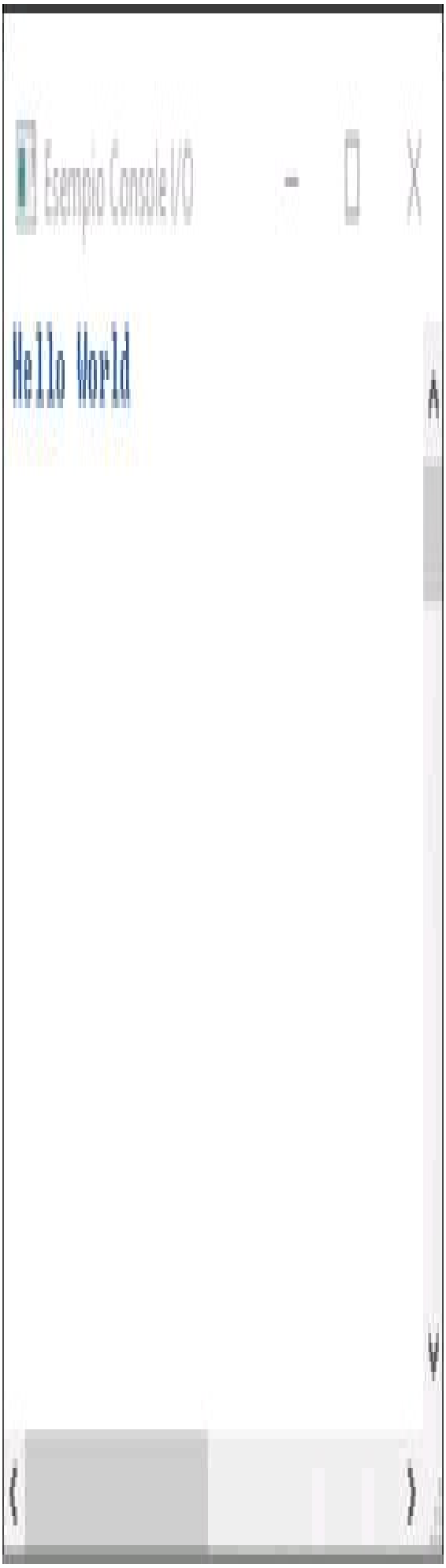

Esempio Console I/O
Hello World

Figura 2.19 – Invio di testo alla Console.

Il metodo WriteLine invece fa andare il testo a capo al termine del testo stampato. Sostituendo nel precedente esempio il metodo Write con WriteLine il risultato sarebbe invece di questo tipo:

```
Hello
World
```

Si noti che anche alla seconda chiamata, dopo aver stampato uno spazio, il testo va a capo.

I metodi di scrittura non servono solo a stampare stringhe, ma permettono di passare come parametro un oggetto qualunque, sia esso di tipo primitivo o di qualunque altro tipo (infatti fra i tipi accettati dai metodi Write e WriteLine di Console vi è anche il tipo object):

```
int i = 123;
bool b = true;
Console.WriteLine(i); //Stampa 123 e va a capo
Console.WriteLine(b); //Stampa True e va a capo
```

Nell'esempio precedente il metodo WriteLine stampa due variabili, o per l'esattezza il loro valore, rispettivamente di tipo int e di tipo bool.

In generale i metodi Write e WriteLine accettano più parametri in ingresso. Se ci sono più parametri essi dovranno essere separati con la virgola.

Stringhe di formato

Nel caso in cui i metodi Write o WriteLine debbano stampare più valori, visualizzandoli in una determinata maniera, bisognerà indicare ai metodi il formato con cui stamparli, quindi si utilizzerà una cosiddetta stringa di formato, che dovrà sempre essere inviata al metodo come primo parametro.

La sintassi di Write o WriteLine avrà in questo caso questo formato generico:

```
Console.WriteLine(string format,object arg1,object arg2,object arg3, …)
```

La stringa di formato stabilisce come stampare gli altri parametri passati al metodo, definendo degli appositi segnaposto, numerati a partire da 0, e racchiusi fra parentesi graffe. Questi segnaposto saranno sostituiti dal parametro corrispondente alla posizione indicata.

Supponiamo per esempio di avere la seguente stringa di formato:

```
string formato="Ciao {0}, ecco il numero {1}";
```

La stringa contiene due segnaposto, indicati con {0} e {1}. Per utilizzarla all'interno del metodo WriteLine bisognerà quindi inviare anche due parametri, per esempio:

```
Console.WriteLine(formato, "Matilda", 123);
```

I parametri utilizzati sono in questo caso la stringa "Matilda" e il numero intero 123, che rispettivamente andranno a sostituire il segnaposto {0} e il segnaposto {1}.

Il codice precedente produce sulla finestra di console il seguente risultato:

```
Ciao Matilda, ecco il numero 123
```

Il numero utilizzato nei segnaposto non deve superare il numero di parametri inviati al metodo. Per esempio, il seguente codice restituirebbe un errore in fase di esecuzione, perché sono stati indicati solo i parametri 1 e 2 e non vi è alcun parametro per il segnaposto {2}:

```
Console.WriteLine("stampo tre numeri: {0}, {1}, {2}", 1, 2);
```

Nella stringa di formato può essere utilizzato un numero qualunque di segnaposti, messi in un ordine qualunque, e possono essere ripetuti più volte. Per esempio, il seguente codice:

```
Console.WriteLine("stampo i numeri {1}, {0}, {0}", 1, 2);
```

invierà alla console il testo:

```
Stampo i numeri 2, 1, 1
```

Spesso capita che il testo da stampare debba essere presentato con un certo formato, che dipende dal tipo di dato, aggiungendo informazioni sulla precisione dei valori numerici oppure allineando il testo a destra o sinistra.

In generale abbiamo visto che i segnaposto sono scritti utilizzando un indice numerico all'interno delle parentesi graffe. Il formato generico di ogni elemento segnaposto è il seguente:

```
{ indice[,allineamento][:StringaFormato]}
```

La componente *indice* identifica il parametro come abbiamo visto finora.

La seconda componente, facoltativa, serve a impostare la larghezza del campo in cui dovrà essere visualizzato il parametro corrispondente e l'*allineamento* all'interno di tale campo.

Si aggiunge quindi dopo l'indice una virgola come separatore, quindi un valore intero con segno che indica la larghezza preferita del campo formattato. Per esempio:

```
Console.WriteLine("{0, 5}", 123);
```

In questo caso l'elemento di formato ha indice 0 e contiene anche il componente di allineamento.

Se il valore di allineamento è inferiore alla lunghezza della stringa formattata, il componente allineamento verrà ignorato e come larghezza del campo verrà utilizzata la lunghezza della stringa.

Se il valore di allineamento è invece maggiore della lunghezza della stringa formattata, per la spaziatura eventualmente necessaria verranno utilizzati spazi vuoti.

I dati formattati verranno allineati a destra se il valore di allineamento è positivo, a sinistra se il valore di allineamento è negativo. Ecco qualche esempio che chiarisce il tutto:

```
float num = 1.2F;
Console.WriteLine("|{0, 5}|", num); //allinea a destra
Console.WriteLine("|{0, -5}|", num); //allinea a sinistra
```

Il primo WriteLine utilizza un valore di 5 per impostare la larghezza da occupare e allinea a destra il valore del float num. Nel secondo caso, invece, la stringa viene allineata a sinistra. All'interno della stringa di formato sono stati inseriti due caratteri | per visualizzare meglio l'allineamento e lo spazio occupato:

```
| 1,2|
|1,2 |
```

Se il valore di num occupa più dei 5 caratteri indicati come spaziatura, il valore 5 viene ignorato. Per esempio eseguendo il codice:

```
num = 1.2345678F;
Console.WriteLine("|{0, 5}|", num);
```

Il risultato sarà:

```
|1,234568|
```

La terza componente è un campo facoltativo che permette di indicare il *formato* con cui rappresentare i parametri e che in genere è utilizzato per valori numerici o altri oggetti come date e orari.

Per esempio, se il num del precedente esempio rappresentasse un valore monetario, potremmo utilizzare tale campo per impostare una precisione di due cifre decimali e il simbolo della valuta.

Nel caso di tipi numerici, tale campo utilizza la seguente sintassi generale:

```
:Axx
```

Il carattere : è il separatore da inserire per separarlo dalle precedenti componenti. A è un singolo carattere di formato che si può scegliere fra nove possibili, xx specifica invece la precisione da utilizzare (che può andare da 0 a 99).

La Tabella 2.11 mostra i valori possibili per l'identificatore di formato numerico A.

Tabella 2.11 **- Identificatori di formato numerico standard.**

Identificatore	Nome	Descrizione	Esempio

C	valuta	Valuta monetaria; la precisione indica il numero di cifre decimali.	123.456 rappresentato come valuta diviene:€ 123,46
D	decimal	Numero intero, positivo o negativo. Se il valore di precisione è maggiore della lunghezza del numero inserisce degli 0 a sinistra.	1234 usando sei cifre di precisione diventa:001234
E	esponenziale	Notazione esponenziale o scientifica con mantissa ed esponente. La precisione indica le cifre decimali.	123.456789 con due cifre decimali: 123E+002
F	virgola fissa	Numero a virgola fissa.La precisione indica le cifre decimali.	123.45678 con tre cifre decimali viene arrotondato a 123.457
G	generale	Formato generale per rappresentare un numero; utilizza la notazione più compatta fra quella F e quella E.La precisione indica il numero di cifre significative.	123.456 oppure 123E+002
N	numero	Formato numerico con separatore dei decimali e separatore delle migliaia (usa la virgola o il punto in base alle impostazioni di sistema).La precisione indica le cifre decimali.	123456.789 viene rappresentato come 123.456,789
P	percentuale	Rappresentazione percentuale; moltiplica il numero per 100. La precisione indica le cifre decimali.	0.12345 viene rappresentato come 123,45%
R	riconversione	Formato della stringa in maniera da poter riconvertire nel corrispondente numero originale.La precisione viene ignorata.	123,456789
X	esadecimale	Converte in base esadecimale un valore intero.Se il valore di precisione è maggiore della lunghezza del numero inserisce degli 0 a sinistra.	123 in esadecimale è 7B

NOTA	Gli identificatori non sono case sensitive, quindi è possibile specificarli sia in maiuscolo sia in minuscolo, tranne nel caso dell'identificatore esponenziale, che stamperà E oppure e nella stringa risultato, oppure nel caso dell'identificatore esadecimale, che stamperà le cifre A, B, C, D, E, F in maiuscolo o minuscolo a seconda dell'utilizzo di X o x.

Il seguente esempio stampa lo stesso numero utilizzando i vari identificatori, per mostrare la differente formattazione prodotta:

```
double val = 123.456789D;
Console.WriteLine("currency: {0:C2}", val);
Console.WriteLine("decimal: {0:D5}", 123);
Console.WriteLine("esponenziale: {0:E2}", val);
Console.WriteLine("virgola fissa: {0:F3}", val);
Console.WriteLine("generale: {0:G}", val);
Console.WriteLine("numerico: {0:N4}", 123456789.123456789);
Console.WriteLine("percent: {0:P2}", 0.123);
Console.WriteLine("round trip: {0:R}", val);
Console.WriteLine("esadecimale: {0:X}", 123);
```

Il risultato del codice precedente sarà il seguente (il simbolo di valuta visualizzato dipende dalla lingua del sistema operativo):

```
currency: € 123,46
decimal: 00123
esponenziale: 1,23E+002
virgola fissa: 123,457
generale: 123,456789
numerico: 123.456.789,1235
percent: 12,30%
round trip: 123,456789
esadecimale: 7B
```

Infine mostriamo un identificatore di formato personalizzato, utile per rappresentare valori numerici.

Il carattere # in una stringa di formato rappresenta un carattere segnaposto per cifre.

Se nel valore da formattare è presente una cifra nella posizione del carattere #, tale cifra viene copiata nella stringa di risultato, altrimenti sarà ignorata. Ecco qualche esempio:

```
Console.WriteLine("{0:#.###}", 1.23456); //stampa 1234
Console.WriteLine("{0:##,##}", 123.456);
Console.WriteLine("{0:(+##)###-###-###}", 39123456789);
```

Interpolazione di stringhe

A partire da C# 6, è possibile un ulteriore approccio alla formattazione del testo, chiamato *interpolazione di stringhe*, introdotto per semplificare l'uso delle stringhe di formato.

Supponiamo di voler creare una stringa componendo i valori delle due variabili seguenti:

```
string nome="Caterina";
int età=33;
```

Usando le stringhe di formato come fatto finora, per esempio con il metodo WriteLine, scriveremmo:

```
Console.WriteLine("{0} ha {1} anni", nome, età); //Caterina ha 33 anni
```

L'interpolazione di stringhe consente di evitare l'uso dei segnaposto numerici e creare la stringa finale utilizzando direttamente i nomi delle variabili o delle espressioni fra parentesi

graffe. Innanzitutto, bisogna far precedere la stringa dal carattere $:

```
Console.WriteLine($"{nome} ha {età} anni");
```

L'espressione della stringa interpolata assomiglia a una sorta di template, che mostra esattamente il formato finale che verrà assunto dalla stringa prodotta, in maniera quindi molto più leggibile.

Una stringa interpolata può essere utilizzata ovunque possa essere usata una normale stringa.

Naturalmente, anche nel definire una stringa interpolata possono essere utilizzati gli identificatori di formato visti in precedenza o altri che incontreremo in seguito, indicandoli subito dopo l'espressione da interpolare, separati dai due punti. Per esempio, per formattare una data in maniera da visualizzare l'orario corrente, si può ottenere una stringa come segue:

```
DateTime dt=DateTime.Now; //ottiene data e ora attuale
Console.WriteLine($"sono le ore {dt:HH:mm:ss}"); //formatta come ore:minuti:secondi
```

La variabile dt, che contiene la data e l'ora, viene usata all'interno della stringa interpolata, indicando il formato HH:mm:ss, cioè ore:minuti:secondi.

Per includere le parentesi graffe in una stringa interpolata bisogna raddoppiarle:

```
String str=$"contiene le parentesi graffe {{}}";
```

Input da Console

Per leggere il testo digitato dall'utente sulla console e utilizzarlo nei nostri programmi è possibile usare il metodo ReadLine della classe Console. Il metodo restituisce una stringa composta dai caratteri inseriti dall'utente fino alla pressione del tasto Invio:

```
string str=Console.ReadLine();
Console.WriteLine("Ciao {0}", str);
```

Il codice precedente legge il testo inserito salvandolo nella variabile str. Questa poi viene utilizzata come parametro per il metodo WriteLine e, ormai, siamo in grado di capire che il risultato sarà la stampa di una stringa formata da Ciao seguita dal testo contenuto nel primo parametro.

Parametri dell'applicazione

Abbiamo già detto, nel paragrafo dedicato al metodo Main, che eseguendo un'applicazione dal prompt dei comandi è possibile inviare a essa dei parametri, digitandoli direttamente subito dopo il nome dell'eseguibile.

Per esempio, se l'applicazione compilata ha prodotto il file eseguibile helloworld.exe possiamo eseguirla inviandole un parametro direttamente così:

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

Se si vuol inviare più di un parametro, basta separarlo con lo spazio dai precedenti:

```
C:\> helloworld.exe param1 param2
```

I parametri indicati nella riga di comando saranno inviati all'applicazione come argomenti del metodo Main, che abbiamo visto in generale poter essere scritto così:

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

L'oggetto args contiene gli eventuali parametri passati all'applicazione. L'oggetto è in particolare un array, cioè un vettore contenente una sequenza di elementi di un dato tipo.

Vedremo più in dettaglio nel Capitolo 3 cosa sono e come funzionano gli array, ma per il momento si mostreranno le basi necessarie per gestire i parametri.

In questo caso esso può contenere zero o più elementi di tipo string, ognuno dei quali rappresenta un parametro scritto sul prompt dei comandi. Per esempio, nel caso precedente:

```
C:\> helloworld.exe param1 param2
```

l'array args conterrà due stringhe, rispettivamente param1 e param2, che possiamo utilizzare all'interno del metodo Main.

Per accedere a ognuno dei parametri si utilizza un operatore [], indicando il numero o l'indice del parametro che si vuol leggere fra tali parentesi quadre, partendo da zero. Per esempio se si vuol leggere e assegnare i due parametri a due variabili di tipo string, basterà scrivere le seguenti due istruzioni:

```
string arg1= args[0];
string arg2= args[1];
```

Se si tentasse di leggere un parametro con un indice non esistente, si verificherebbe un errore di accesso all'array e sulla console verrebbe scritto qualcosa del tipo:

```
Unhandled Exception: System.IndexOutOfRangeException: Index was outside the bounds of the array.
```

È possibile quindi verificare se l'array args contiene un certo numero di elementi, prima di tentare di accedere alla particolare posizione. Imparando più avanti a usare gli array e gli altri operatori di C#, tale procedura sarà semplicissima, per il momento prendetelo come un suggerimento:

```
if(args.Length>=2)
{
string arg1= args[0];
string arg2= args[1];
}
```

Il codice precedente verifica se il parametro args contiene due o più elementi e solo in questo caso tenta di leggerli e assegnarli alle due variabili.

Se si utilizza Visual Studio per eseguire programmi console, è sempre possibile inviare parametri da riga di comando modificando le proprietà del progetto: per farlo fate clic con il tasto destro sul nome del progetto stesso, all'interno del Solution Explorer, e poi selezionate la scheda Debug.

Per indicare i parametri da inviare all'applicazione bisogna inserirli nella casella di testo Command Line Arguments, presente nella sezione Start Options, come mostrato per esempio in Figura 2.20.

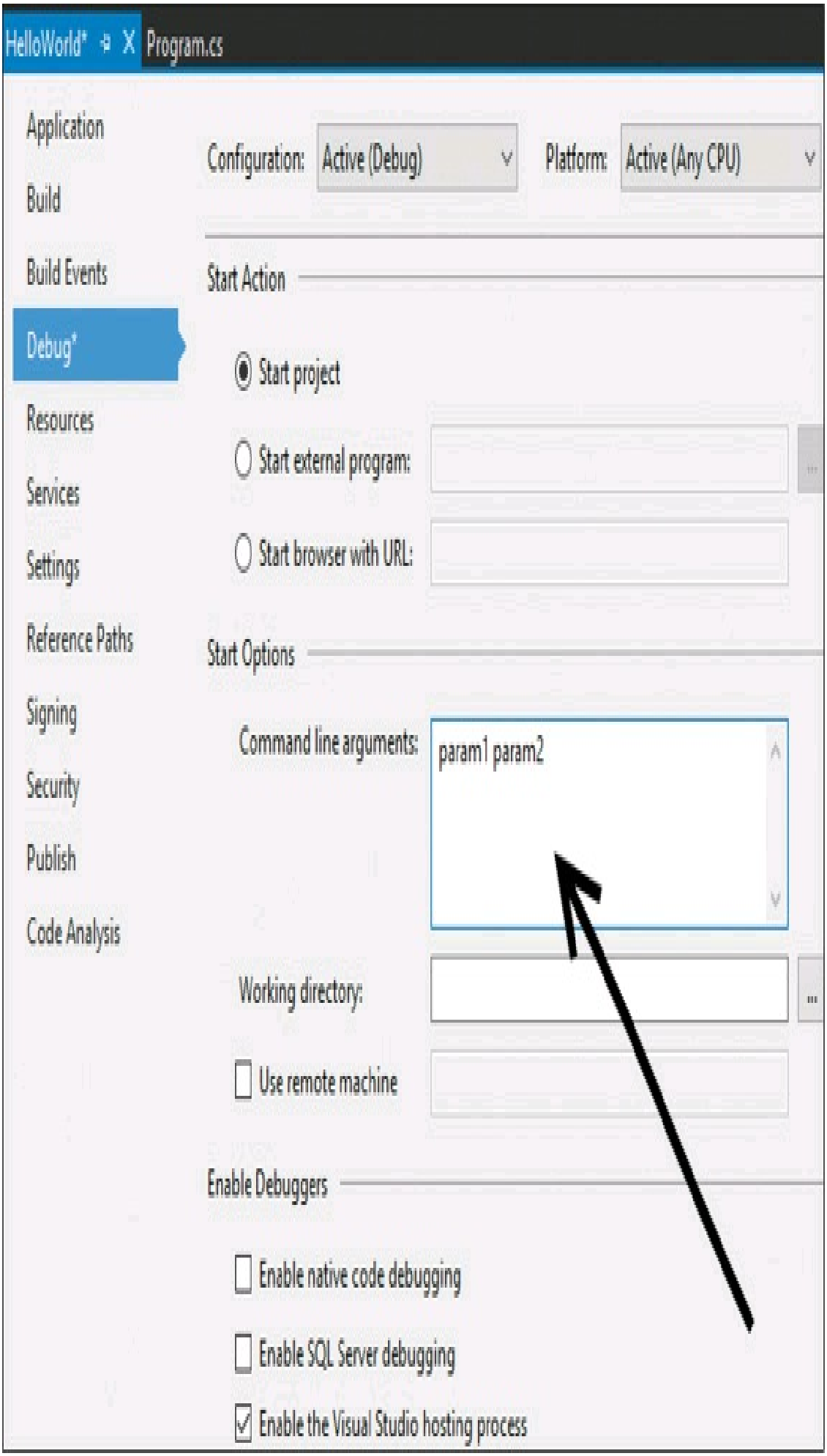

HelloWorld*
Program.cs
Application
Build
Build Events
Debug*
Resources
Services
Settings
Reference Paths
Signing
Security
Publish
Code Analysis
Configuration: Active (Debug)
Platform: Active (Any CPU)
Start Action
Start project
Start external program:
Start browser with URL:
Start Options
Command line arguments:
param1 param2
Working directory:
Use remote machine
Enable Debuggers
Enable native code debugging
Enable SQL Server debugging
Enable the Visual Studio hosting process

Figura 2.20 – Invio di parametri da riga di comando in Visual Studio.

Domande di riepilogo

1) Dichiara una variabile per memorizzare un numero intero, chiamandola myInt e assegna il valore iniziale 1:

a. myInt=1;
b. variable myInt=1;
c. int myInt=1;
d. int myInt= new int(1);

2) Qual è il comando per compilare il codice presente nel file hello.cs e ottenere un assembly in formato libreria chiamato helloworld.dll?

a. csc /target:dll /out:helloworld hello.cs
b. csc /target:library /out:helloworld.dll hello.cs
c. compile hello.cs /out:helloworld.dll
d. csharpcompiler /target:lib /out:helloworld.dll hello.cs

3) Per calcolare la radice quadrata di un metodo tramite il metodo Math.Sqrt senza specificare il nome della classe Math è necessario usare l'istruzione:

a. using System.Math;
b. using class System.Math;
c. using static System.Math;
d. using static class System.Class

4) Per stampare sulla Console una variabile val nel formato valuta con due cifre decimali si utilizza:

a. Console.WriteCurrency("{0:2}", val);
b. Console.Write("{0:€2}", val);
c. Console.Write("{0:$2}", val);
d. Console.Write("{0:C2}", val);

5) Quale di questi identificatori di variabile è utilizzabile e corretto in C#?

a. \u005fnumero
b. 1numero
c. num-ero
d. ~numero

Tipi e oggetti

Il .NET Framework si basa sul Common Type System, che distingue tipi valore e riferimento, e poi classi, strutture, enumerazioni e array. In questo capitolo si vedrà la creazione di oggetti dei diversi tipi e la conversione di oggetti da un tipo a un altro.

Ogni programma C# è basato sull'interazione reciproca di oggetti di vario tipo: numeri, stringhe, date oppure oggetti più complessi. Il .NET Framework infatti fornisce centinaia di tipi già pronti all'uso e messi a disposizione degli sviluppatori.

Il tipo di dati è quindi un concetto fondamentale, tanto che, come mostrato nel Capitolo 1, Microsoft insieme al .NET Framework ha introdotto una specifica, detta *Common Type System*, che descrive come i tipi vengono dichiarati, creati, utilizzati e gestiti dal CLR. Il termine Common (comune) indica che i tipi in .NET possiedono, e devono possedere, caratteristiche tali da essere utilizzabili in ogni linguaggio di programmazione e su ogni sistema operativo e hardware sottostante.

In tal modo, per esempio, un numero intero sarà rappresentato e funzionerà allo stesso modo in C#, in Visual Basic, in C++ e così via, e anche su ogni sistema operativo, in maniera indipendente dalla sua architettura e da quella della CPU che eseguirà le istruzioni. Per esempio un numero intero può essere dichiarato per mezzo della parola chiave `int`, che corrisponderà a un oggetto del tipo `System.Int32`, e che su ogni piattaforma verrà sempre rappresentato con 32 bit.

Per rendere possibile tale funzionamento il Common Type System introduce anche un tipo base, la classe `System.Object`, e la definisce come la madre di tutti gli altri tipi, sia quelli forniti dal .NET Framework, sia quelli che implementerà lo sviluppatore.

In questo capitolo, quindi, vedremo innanzitutto come .NET suddivide i tipi in due sottofamiglie, i tipi valore e quelli riferimento.

Nel seguito cominceremo a distinguere le varie categorie di tipi, introducendo i concetti di classi, strutture ed enumerazioni, in maniera da permetterci da subito di creare e utilizzare oggetti e i loro membri.

Il capitolo servirà come introduzione al mondo della programmazione a oggetti, che verrà poi approfondito nel dettaglio nel Capitolo 6: alcuni argomenti potranno sembrare affrontati in modo superficiale, ma ci basterà al momento darne un'infarinatura per poter da subito iniziare a creare i nostri esempi.

Fra i tipi di .NET si vedranno più in dettaglio gli array, che permettono la creazione di insiemi di elementi di uno stesso tipo.

Tipi di dati e oggetti

Un programma C# contiene come minimo un tipo di dati. Anche l'esempio Hello World, creato all'inizio del capitolo precedente, contiene una classe denominata Program, che altro non è che un tipo.

Ciò significa che ogni programma C# sarà costituito da un insieme di tipi, ognuno con una sua struttura, che servirà a rappresentare un particolare dato da manipolare nel programma stesso.

Un tipo di dati è una sorta di schema per creare strutture di dati più o meno complesse. Quindi ogni tipo avrà un nome, una struttura interna nella quale conterrà i dati e una serie di altre caratteristiche che ne definiscono il comportamento e le possibilità di comunicazione con altri tipi.

Per esempio, per memorizzare un dato "numero intero" abbiamo utilizzato nel Capitolo 2 delle variabili di tipo int, o meglio di tipo System.Int32.

Il tipo suddetto ha quindi nome "Int32", è contenuto nel namespace System, ha una struttura interna che permette di memorizzare 32 bit e possiede dei metodi e delle proprietà per scrivere, leggere e manipolare i bit stessi che rappresentano il numero intero.

Se il tipo è uno schema di rappresentazione dei dati, i dati veri e propri rappresentano invece un'istanza di tale tipo. Per esempio, Int32 è il tipo che rappresenta gli interi, mentre il numero 123 è un'istanza del tipo Int32.

Ogni istanza di un dato tipo è anche detto oggetto. Da ciò deriva il termine di programmazione orientata agli oggetti: ogni dato in questo paradigma di sviluppo è un'istanza di un ben determinato tipo o, appunto, un oggetto.

Ogni oggetto mantiene il proprio stato, all'interno di uno schema di memorizzazione definito dal tipo, e può comunicare e interagire con altri oggetti, secondo ben determinate regole, anch'esse indicate dal proprio tipo.

Riassumendo in una sola frase, in .NET e C# qualsiasi cosa è un oggetto.

Tipi valore e tipi riferimento

Poiché tutto è un oggetto, il .NET Framework e in particolare il Common Type System, definiscono un tipo speciale detto System.Object o, utilizzando l'alias del compilatore C#, semplicemente object.

Tutti gli altri tipi, sia quelli esistenti sia quelli che ogni sviluppatore implementerà nelle proprie applicazioni, sono organizzati secondo una gerarchia che parte dalla radice comune rappresentata da System.Object. In tal modo viene garantito che ogni tipo abbia come minimo un insieme di funzionalità ereditate da System.Object.

La prima distinzione fondamentale che viene fatta all'interno di questa gerarchia di tipi è fra tipi valore e tipi riferimento. Tale distinzione deriva principalmente dalla modalità di conservazione dei dati in memoria.

NOTA	Una terza categoria di tipi, quella dei puntatori, è utilizzabile solo in un contesto di codice unsafe, all'interno del quale sono esclusi i meccanismi di controllo e gestione automatica della memoria del CLR. L'utilizzo di codice unsafe è comunque un argomento che va al di là degli scopi del testo e che è fra i meno utilizzati dalla stragrande maggioranza degli sviluppatori C#. Nell'Appendice B potete comunque trovare una breve introduzione e degli esempi sul contesto unsafe e sui puntatori.

Una volta in esecuzione, i dati manipolati dal programma devono essere mantenuti in memoria; in particolare in .NET ogni programma in esecuzione utilizza due regioni di memoria, dette *stack* e *heap* (che spesso è anche detta *managed heap*, avendo a che fare con memoria gestita dal CLR).

Uno stack è una struttura usata per memorizzare i dati in modalità *LIFO* (*Last-In-First-Out*), vale a dire come una pila di oggetti (in inglese stack), nella quale l'ultimo oggetto inserito sarà necessariamente il primo a poter essere estratto dalla pila.

I tipi valore sono tipi che conservano direttamente il valore che essi definiscono; infatti sono i tipi più semplici, come quelli primitivi che abbiamo visto nel Capitolo 2, perché occupano una piccola e ben determinata quantità di memoria. Per questo motivo gli oggetti di tipo valore vengono memorizzati in genere nello stack.

	Non è corretto affermare che le variabili di tipo valore sono sempre memorizzate nello stack, in quanto ci sono diversi casi e possibilità da considerare (per esempio, un tipo

NOTA

riferimento potrebbe contenere un campo di tipo valore), ma per il momento tali dettagli vanno al di là dei nostri scopi.
Intanto possiamo correggere l'affermazione dicendo che i tipi valore possono essere memorizzati nello stack, i tipi riferimento invece sono memorizzati sempre nell'heap.
Un'altra possibile distinzione fra tipi valore e riferimento deriva da come gli oggetti vengono copiati: se copio un dato di tipo valore, viene copiato direttamente il suo valore, se copio un dato di tipo riferimento viene copiato solo il riferimento, quindi i tipi valore sono in genere tipi che richiedono una piccola area di memoria.

Nella Figura 3.1, per esempio, viene mostrato uno schema del funzionamento dell'area di memoria stack: a sinistra, l'area di memoria stack contiene già degli oggetti numerici, la freccia Top indica la posizione alla quale verrà inserito il prossimo oggetto.

Eseguendo un'istruzione di assegnazione della variabile `int i=180475`, il valore viene memorizzato nello stack occupando una nuova locazione di memoria, spostando la freccia Top più in alto.

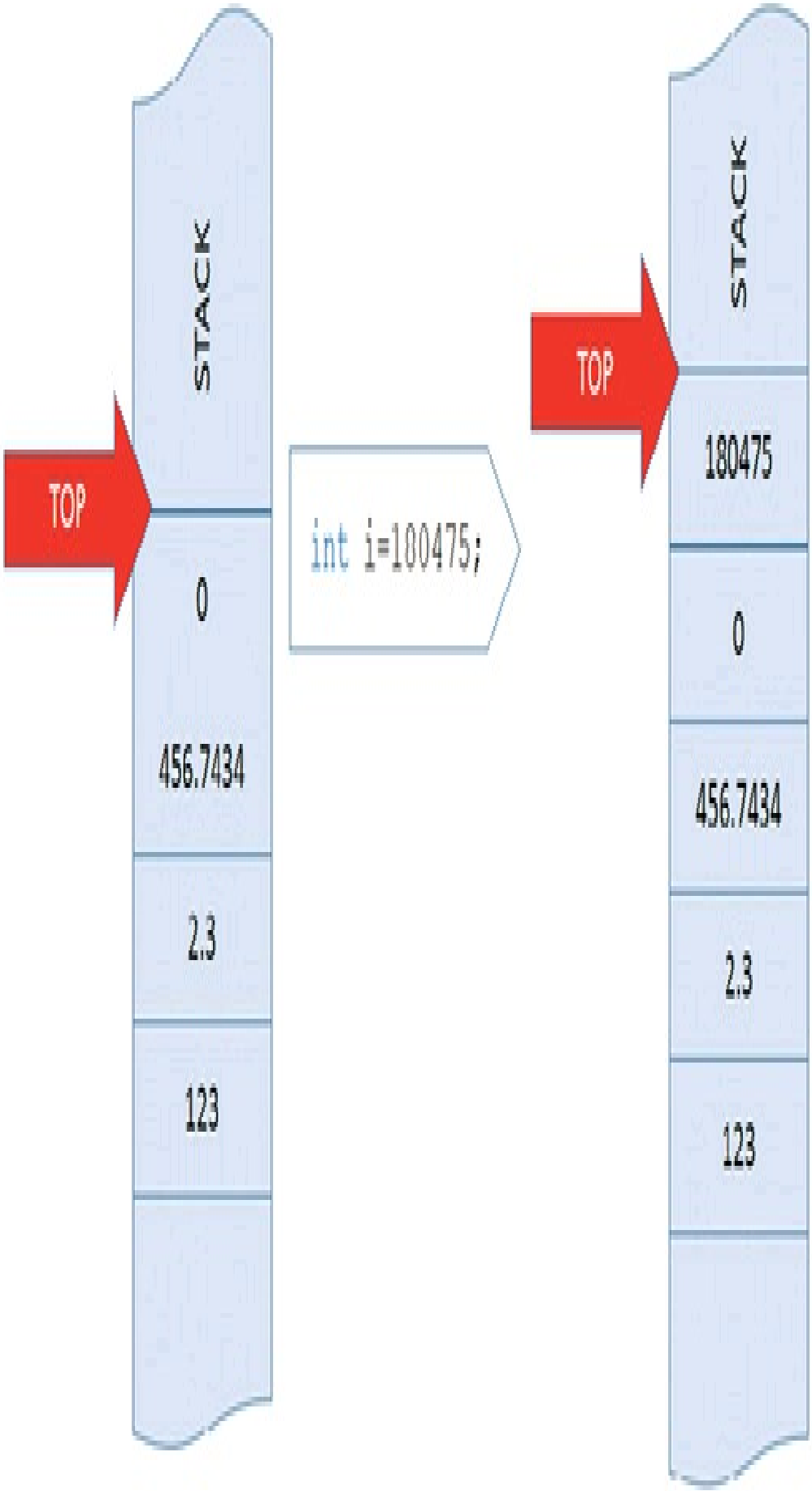

STACK
TOP
0
456.7434
2.3
123
int i=100475;
STACK
TOP
180475
0
456.7434
2.3
123

Figura 3.1 – Funzionamento della memoria stack.

Al contrario dei tipi valore, gli oggetti di tipo riferimento conservano al loro interno soltanto un indirizzo, cioè appunto un riferimento che punta alla locazione di memoria nel quale si troveranno i dati veri e propri (che possono essere molto complessi e occupare molta memoria). Ciò significa che anche un'altra variabile può puntare allo stesso oggetto, mediante un altro riferimento.

Per questo motivo per conservare un oggetto di tipo riferimento occorrono due aree di memoria: all'interno dello stack verrà memorizzato soltanto il riferimento, che punta a una precisa area di memoria heap, nel quale verrà conservato l'oggetto vero e proprio con i dati che esso contiene (come mostrato nella Figura 3.2).

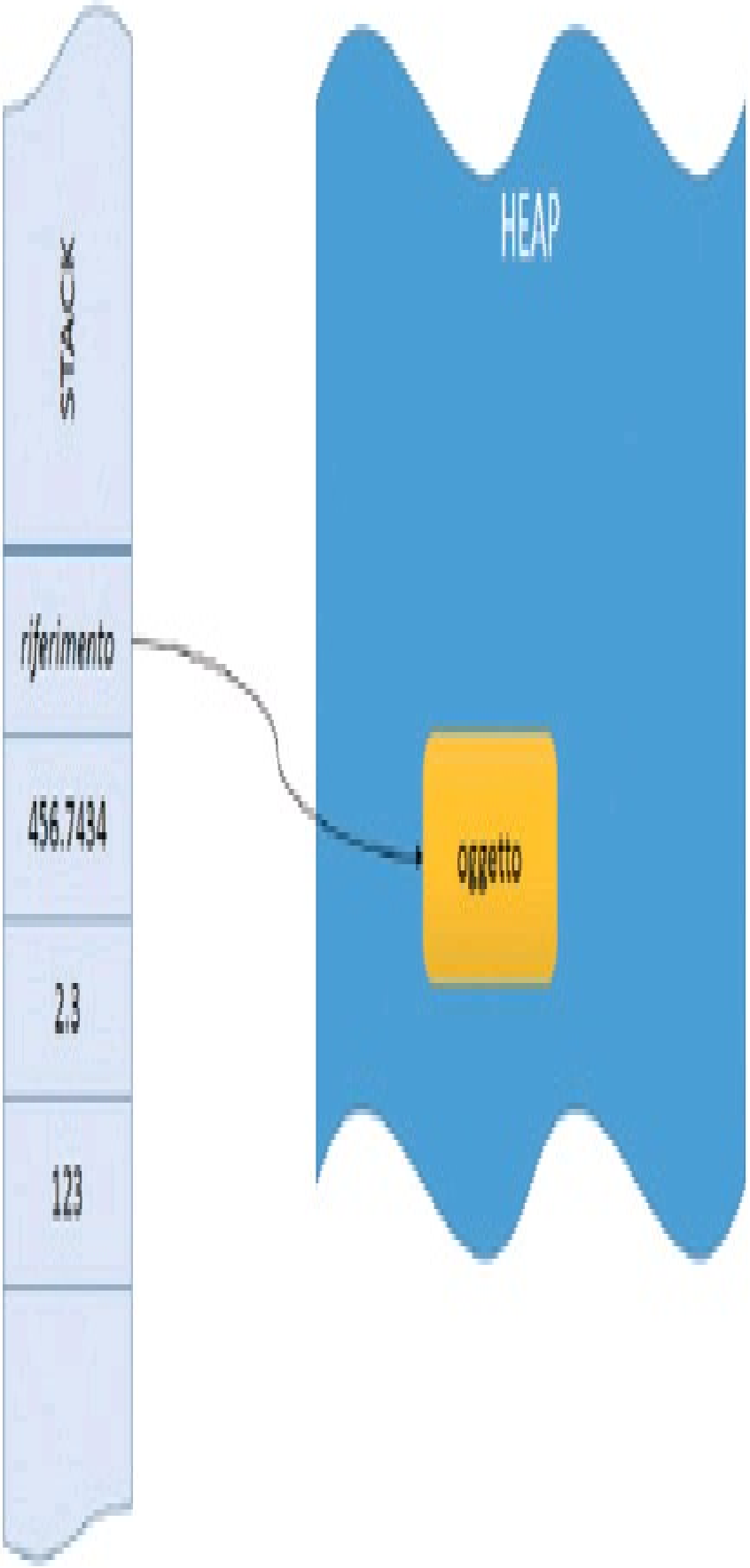

STACK
riferimento
456.7434
2.3
123
HEAP
oggetto

Figura 3.2 – Funzionamento della memoria heap.

Gli oggetti in memoria heap possono quindi essere rimossi in un ordine qualsiasi, ma non è il programmatore che se ne occupa direttamente. Il CLR, mediante il processo di Garbage Collection, verifica quando l'oggetto in memoria heap non ha più nessun riferimento proveniente dallo stack e quindi sa che può liberare quell'area di memoria.

NOTA	I tipi valore occupano esattamente il quantitativo di memoria necessario a conservare il valore stesso. Per esempio un Int32 occuperà esattamente 4 byte (32 bit) di memoria stack. Invece per un oggetto riferimento, che conterrà al suo interno più oggetti, a loro volta di tipo valore o riferimento, bisognerà sommare la memoria occupata da ogni oggetto, più un overhead di memoria per la gestione del riferimento. Per esempio una classe con quattro campi di tipo int occuperà perlomeno 4 x 4 = 16 byte in memoria heap.

Gerarchia dei tipi

Avendo chiarito che ogni tipo deriva dalla classe madre System.Object e che tutti i rimanenti tipi sono suddivisi in tipi valore e tipi riferimento, possiamo ora dare uno sguardo alla gerarchia dei tipi di .NET.

I tipi valore ricadono in due categorie:

- struct
- enum

I tipi primitivi predefiniti di C#, che sono quelli già visti nel Capitolo 2, sono tutti di tipo struct:

- sbyte
- byte
- short
- ushort
- int
- uint
- long
- ulong
- bool
- float
- double
- decimal
- char

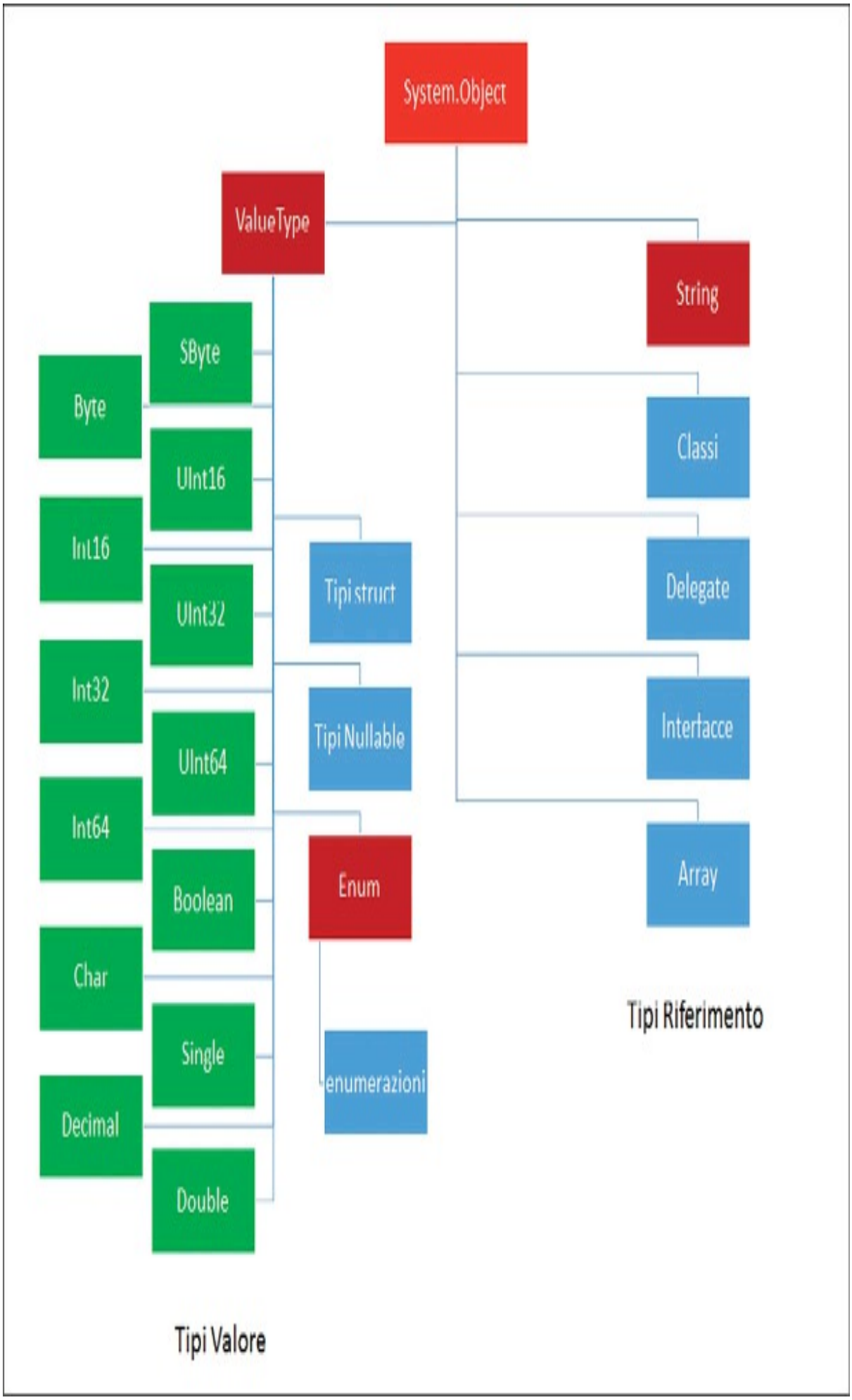

System.Object
ValueType
String
SByte
Byte
UInt16
Int16
UInt32
Int32
UInt64
Int64
Boolean
Char
Single
Decimal
Double
Tipi struct
Tipi Nullable
Enum
enumerazioni
Classi
Delegate
Interfacce
Array
Tipi Riferimento
Tipi Valore

Figura 3.3 – Gerarchia di tipi valore e riferimento.

I tipi valore non sono discendenti direttamente da object, ma in realtà derivano dal tipo ValueType, che a sua volta discende naturalmente da System.Object.

I tipi riferimento a loro volta sono suddivisibili in una delle seguenti categorie:

- class
- interface
- delegate
- array

I tipi riferimento predefiniti di C# sono i seguenti:

- object
- string
- dynamic

Nei prossimi paragrafi e capitoli vedremo più nel dettaglio queste categorie, sia per comprenderne l'utilizzo sia per imparare a creare i propri tipi personalizzati.

Utilizzo dei tipi

In questo paragrafo daremo qualche cenno sull'utilizzo dei tipi, in particolare sulla sintassi utilizzata per creare oggetti e per accedere alle loro funzionalità. In tal modo saremo subito in grado di utilizzare quelli messi a disposizione dal .NET Framework e sfruttarli nelle nostre applicazioni.

Il primo passo per utilizzare un tipo di dati è quello di creare un oggetto o, con gergo più tecnico, istanziarlo (cioè creare un'istanza).

Molti concetti potrebbero sembrare oscuri al lettore alle prime armi, ma verranno chiariti quando introdurremo in maniera più approfondita i concetti della programmazione a oggetti in C#. Per il momento cercate di portare pazienza e cominciate a mettere in pratica quanto esposto, scrivendo e compilando i vari esempi.

In generale, mentre la creazione di un tipo valore può avvenire semplicemente assegnando il valore stesso a una variabile, quella di un tipo riferimento richiede la costruzione di un'istanza per mezzo dell'operatore new (solo il tipo string permette di assegnare dei valori letterali a una variabile). Nulla vieta però di utilizzare l'operatore new per creare oggetti di tipo valore.

Le seguenti tre istruzioni sono identiche:

```
int i=0;
int i=new int();
System.Int32 i=new System.Int32();
```

Il loro effetto è quello di creare un nuovo oggetto di tipo int, alias di System.Int32.

Subito dopo l'operatore new viene indicato il nome di un metodo speciale che ha lo stesso nome del tipo: questo metodo speciale viene detto *costruttore*.

Una volta costruito l'oggetto, esso può essere utilizzato tramite i membri, per esempio metodi e proprietà, del tipo.

La sintassi generale per interagire con un oggetto prevede l'utilizzo dell'operatore dot (.), che segue il nome dell'oggetto. Per esempio, ogni tipo possiede (in quanto derivato da Object) il metodo per ottenere una rappresentazione testuale dell'oggetto stesso, denominato ToString. Per invocare tale metodo sulla variabile i di tipo int, scriveremo:

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

NOTA

Il metodo ToString viene implicitamente invocato ogni volta che si ha bisogno di ottenere una rappresentazione testuale di un oggetto. Per esempio è possibile invocare il metodo

Console.WriteLine passandogli come parametro un oggetto di un tipo qualsiasi: in questo caso dietro le quinte viene invocato il metodo ToString dell'oggetto.

Altri membri, detti statici, non hanno bisogno di un'istanza in quanto agiscono direttamente sul tipo. Per esempio, i tipi numerici predefiniti hanno tutti due proprietà statiche MinValue e MaxValue, che restituiscono rispettivamente il valore minimo e massimo che può assumere una variabile del tipo stesso.

Per il tipo Byte, per esempio, essi restituiranno i valori 0 e 255:

```
byte min=Byte.MinValue; // = 0
byte max=Byte.MaxValue; // = 255
```

Il tipo **System.Object**

Abbiamo già detto che il tipo che svolge il ruolo di genitore di tutti gli altri tipi, e quindi anche il nodo radice della gerarchia di tipi .NET, è il tipo System.Object.

Nel paradigma di programmazione orientato agli oggetti il significato della precedente affermazione indica innanzitutto che ogni tipo eredita da System.Object sia il funzionamento sia i dati.

NOTA	L'ereditarietà nel mondo software è diversa da quella fra umani. Gli oggetti che derivano da un oggetto padre ne ereditano comportamento e dati, ma l'oggetto padre non perde i dati che ha passato ai figli, in realtà li condivide.

Un'altra conseguenza del fatto che System.Object costituisca la radice dell'intera gerarchia è che sia i tipi valore sia i tipi riferimento derivano da un tipo comune. Tramite esso quindi le due categorie di tipi vengono unificate sotto un solo padre (la Figura 3.3 vista in precedenza mostra la gerarchia di tipi derivata da System.Object). L'effetto principale di tale unificazione è che anche una variabile di tipo valore, derivando da Object, potrà essere trattata come oggetto e questo implica a sua volta che una variabile il cui valore è allocato in memoria stack potrà essere spostata nella memoria heap e viceversa (vedere i concetti di boxing e unboxing più avanti).

Dal punto di vista pratico, cioè per lo sviluppatore, la discendenza dalla classe System.Object implica che ogni oggetto di un qualunque altro tipo può utilizzare dei metodi che eredita da tale classe comune.

I metodi principali definiti nella classe Object sono i seguenti:

- Equals – confronta l'oggetto con un altro per verificarne l'uguaglianza;
- GetHashCode – calcola un codice hash numerico per l'oggetto, utilizzabile per memorizzare e riconoscere l'oggetto in maniera più efficiente all'interno di certe collezioni definite hashtable;
- GetType – restituisce il tipo dell'oggetto;
- ToString – restituisce una rappresentazione testuale dell'oggetto.

Proviamo a utilizzare i metodi su una variabile di tipo valore, per esempio un int:

```
int i=123;
int j=456;
Console.WriteLine(i.ToString());
Console.WriteLine(i.Equals(j));
```

```
Console.WriteLine(i.GetType());
```

Il risultato stampato dalle istruzioni Console.WriteLine sarà il seguente:

```
123
False
System.Int32
```

Le istanze dei tipi valore e quelle string possono essere costruite direttamente mediante assegnazione dei corrispondenti valori letterali (vedere il Capitolo 2). Quindi anche dei valori letterali come i seguenti:

```
1
"stringa"
true
'a'
```

possono essere trattati (perché lo sono) come oggetti, che derivano dalla classe System.Object, e su di essi è possibile invocare i metodi di cui sopra.

Per esempio possiamo provare a invocare il metodo GetType su una serie di oggetti per ricavarne il nome del tipo CTS:

```
Console.WriteLine("{0}", new byte().GetType());
Console.WriteLine("{0}", new sbyte().GetType());
Console.WriteLine("{0}", new short().GetType());
Console.WriteLine("{0}", new ushort().GetType());
Console.WriteLine("{0}", 1.GetType());
Console.WriteLine("{0}", 1U.GetType());
Console.WriteLine("{0}", 1L.GetType());
Console.WriteLine("{0}", 1UL.GetType());
Console.WriteLine("{0}", 1.0F.GetType());
Console.WriteLine("{0}", 1.0D.GetType());
Console.WriteLine("{0}", 1M.GetType());
Console.WriteLine("{0}", true.GetType());
Console.WriteLine("{0}", 'a'.GetType());
Console.WriteLine("{0}", "string".GetType());
```

Il metodo GetType restituisce un oggetto di tipo Type.

Sebbene possa sembrare una ripetizione, la classe Type rappresenta proprio il tipo di un tipo e fra i suoi membri è possibile ricavarne anche il nome.

In questo caso, utilizzandolo all'interno di un'istruzione WriteLine, viene implicitamente invocato il metodo ToString dell'oggetto Type ottenuto. Il risultato, per ogni istruzione, sarà la stampa del nome del tipo ricavato da ogni valore:

```
System.Byte
System.SByte
System.Int16
System.UInt16
System.Int32
```

System.UInt32
System.Int64
System.UInt64
System.Single
System.Double
System.Decimal
System.Boolean
System.Char
System.String

I metodi della classe Object sono personalizzabili all'interno di ogni tipo che deriva da essa, in maniera da funzionare in maniera più idonea al tipo stesso.

L'implementazione del metodo GetHashCode fornito dalla classe String, per esempio, restituisce codici hash identici per valori stringa identici. Di conseguenza, due oggetti String restituiscono lo stesso codice hash se rappresentano lo stesso valore stringa.

Vedremo in dettaglio come personalizzare metodi e proprietà derivati da una classe madre quando parleremo di ereditarietà e di metodi virtuali, nel Capitolo 7.

Boxing e unboxing

Si definisce con il termine boxing l'azione di convertire un'istanza di un tipo valore in un oggetto di tipo riferimento. Il termine stesso indica infatti l'atto di "inscatolare", in questo caso un valore all'interno di un oggetto contenitore:

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

Nell'esempio precedente la variabile intera i, quindi un tipo valore, che si trova nella memoria stack, viene inscatolata nella variabile box, di tipo object, che quindi verrà memorizzata nell'area heap.

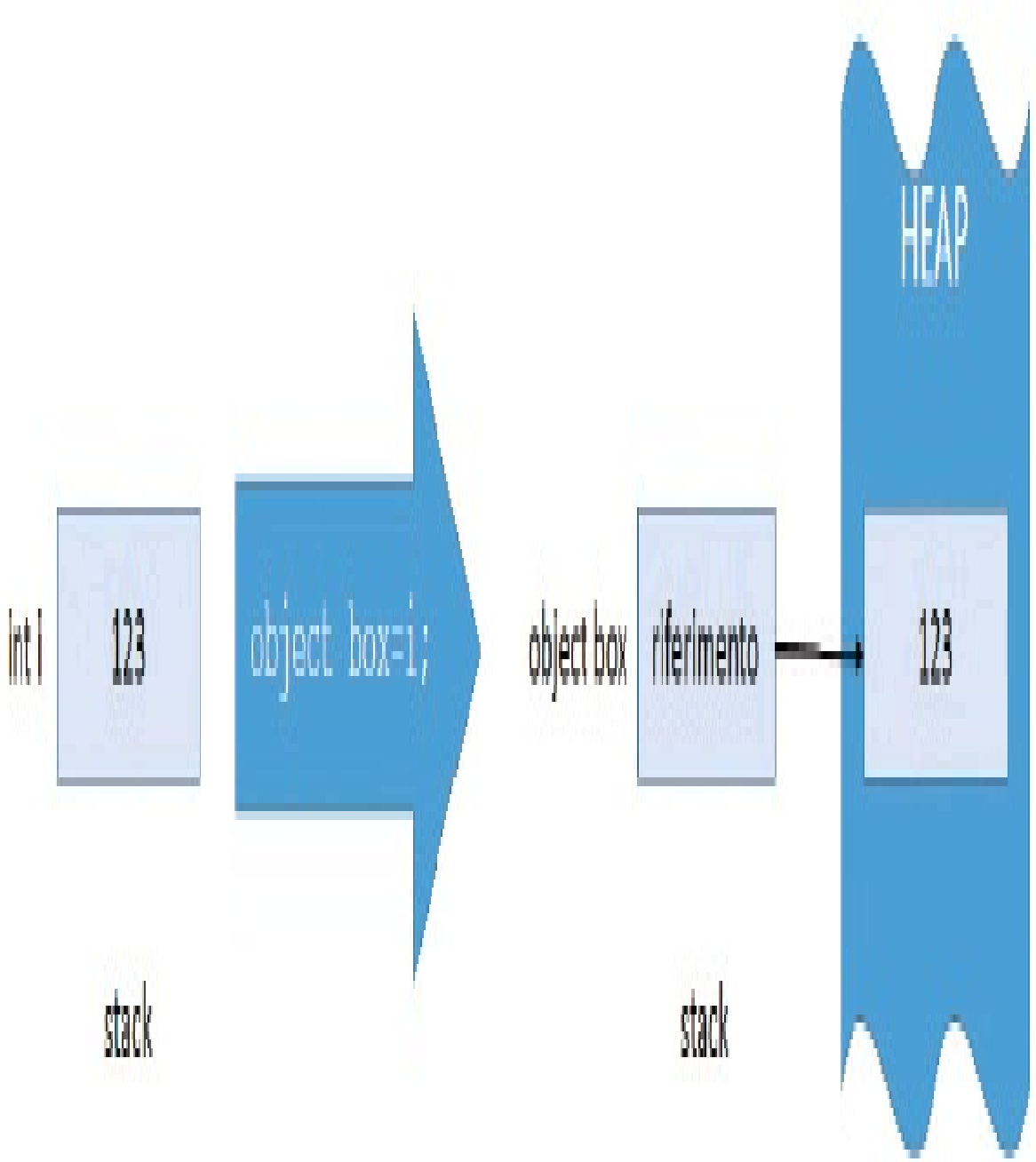

int i
123
stack
object box=i;
object box
riferimento
stack
HEAP
123

Figura 3.4 – Gerarchia di tipi valore e riferimento.

Con l'operazione di boxing viene eseguita una copia del valore in un nuovo oggetto. Quindi se dopo il boxing si modifica il valore della variabile i, il contenuto della variabile box rimane immutato:

```
int i = 123;
object box=i;
i= 456;
Console.WriteLine(box); //box contiene 123
```

L'operazione inversa è l'unboxing, che permette di "scartare" da un oggetto un valore precedentemente inscatolato. Per esempio:

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

La terza istruzione, per mezzo di un'operazione di conversione (vedremo fra poco cosa significa), estrae dall'oggetto box un valore int (indicato fra parentesi prima dell'oggetto box).

Naturalmente, perché l'istruzione vada a buon fine, dobbiamo essere sicuri che nell'oggetto box vi sia stato precedentemente inserito un valore del tipo che si vuol estrarre.

Per esempio se avessimo scritto:

```
double d=(double)box;
```

durante l'esecuzione del programma si sarebbe verificato un errore (o meglio un'eccezione) perché abbiamo tentato di estrarre dall'oggetto box un valore double, mentre in realtà al suo interno c'era un valore int.

Per verificare quindi il tipo effettivo di un oggetto, per esempio il tipo contenuto nella variabile box dell'esempio precedente, è possibile utilizzare il metodo GetType messo a disposizione dalla classe System.Object:

```
Console.WriteLine(box.GetType()); //stampa System.Int32
```

Le classi

La categoria più comune di tipi riferimento è quella delle classi. In questo paragrafo introdurremo il concetto di classe, dando una definizione dei vari aspetti utili per poterne creare di semplici e soprattutto iniziare a utilizzare quelle messe a disposizione dal .NET Framework.

Per creare le nostre classi personalizzate (che forse è l'attività principale nello sviluppo di una nuova applicazione) e approfondire i vari aspetti di ogni classe dovremo attendere fino al Capitolo 6, dedicato alla programmazione a oggetti.

Una classe è un tipo di riferimento che è possibile derivare direttamente da un'altra classe e che viene derivato in modo implicito da System.Object.

La classe definisce innanzitutto la struttura dei dati che un oggetto (detto anche un'istanza della classe) può contenere (campi della classe) e le operazioni che l'oggetto può eseguire (metodi, eventi, proprietà e altro).

Supponiamo, per esempio, di dover rappresentare nella nostra applicazione dei clienti e per ognuno di essi supponiamo di dover manipolare informazioni come nome, cognome e ogni altra caratteristica che dovrà essere memorizzata o elaborata.

Definizione di una classe

Il primo passo sarà la definizione di una classe, per la quale sceglieremo un identificatore, cioè un nome, per esempio Customer (ma anche Cliente andrebbe bene per i più nazionalisti!). Per creare una classe si utilizza la parola chiave class, seguita dal nome scelto per il tipo:

```
class Customer
{
//campi
//proprietà
//metodi
//eventi
// …
} //fine della classe
```

Subito dopo la dichiarazione della classe, inizia il blocco che definisce il corpo della classe e che sarà racchiuso fra parentesi graffe. All'interno del corpo della classe verranno definiti campi, proprietà, metodi ed eventi, a cui collettivamente ci si riferisce con il termine di membri della classe.

NOTA	Ogni classe, come più volte detto, deriva in modo implicito dalla classe System.Object, quindi non è necessario esplicitarlo in alcun punto; infatti nella dichiarazione della classe Customer non vi è alcun riferimento a essa.

Creazione di un oggetto

Spesso i termini di classe e oggetto vengono confusi e scambiati, ma sono concetti differenti ed è bene comprenderne il significato, a rischio di essere ripetitivi. Abbiamo infatti già spiegato la differenza dicendo per esempio che Int32 è il tipo che definisce un numero intero, mentre il numero 123 è un oggetto di tipo Int32.

La classe Customer, quindi, definisce la struttura del tipo di dati che conterrà le informazioni di ogni cliente, mentre un oggetto di classe Customer sarà una sua istanza concreta, per esempio il cliente specifico con nome Paolo e cognome Rossi.

Per creare un oggetto (o, se preferite, istanziare una classe) si utilizza l'operatore new, che si occupa di allocare la memoria heap necessaria a invocare il metodo costruttore della classe.

NOTA	Se non ci fosse sufficiente memoria per creare l'oggetto desiderato verrebbe scatenata una cosiddetta eccezione OutOfMemoryException, cioè un errore di memoria insufficiente. Vedremo più avanti che cos'è un'eccezione, per il momento pensate a esse come errori durante l'esecuzione del programma.

Se per esempio fra i nostri clienti ci fosse il Dr. Jekyll potremmo creare la sua istanza in questo modo:

```
Customer drJekyll=new Customer();
```

Invocando un metodo con lo stesso nome della classe, che è presente implicitamente anche se non l'abbiamo definito, è stata creata un'istanza della classe Customer, della quale viene restituito un riferimento salvato appunto nella variabile drJekill.

Il metodo precedente è detto costruttore della classe e una classe può anche definirne più di uno, con diversi insiemi di parametri che è possibile inviargli. Il costruttore senza parametri è detto costruttore di default.

Essendo un tipo riferimento, la variabile drJekill non contiene alcun dato al suo interno e, infatti, è anche possibile solo dichiarare una variabile di classe Customer, senza costruire alcuna istanza:

```
Customer mrHide;
```

In questo caso, la variabile mrHide non ha alcun riferimento a nessun oggetto e quindi è inutilizzabile. Infatti, se provassimo a scrivere subito dopo un'istruzione che tenti di utilizzare

l'oggetto, riceveremmo un errore di compilazione perché si è fatto uso di una variabile non assegnata:

```
Customer mrHide;
Console.WriteLine(mrHide); // errore, uso di variabile non assegnata
```

L'unico modo, a questo punto, per utilizzare una variabile così dichiarata è quello di assegnarle un valore, che naturalmente deve essere di tipo Customer:

```
mrHide = drJekyll;
```

In tal modo abbiamo assegnato alla variabile mrHide un riferimento all'oggetto a cui si riferisce la variabile drJekyll.

La Figura 3.5 mostra cosa avviene in memoria, con la variabile drJekyll che è un riferimento che punta all'oggetto Customer, creato e memorizzato nel managed heap.

Dopo l'assegnazione della seconda variabile, il risultato è che essa si riferisce allo stesso oggetto.

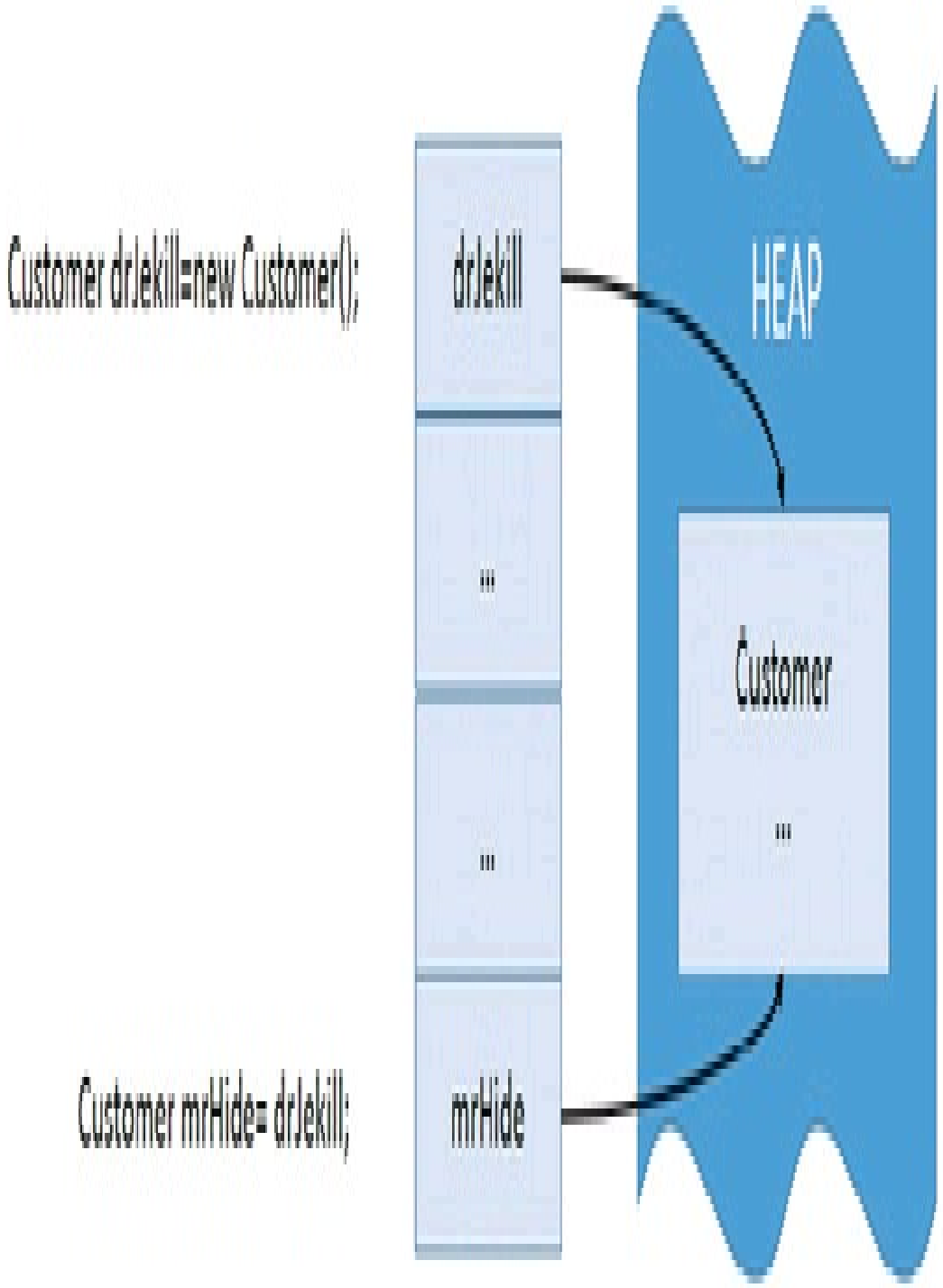

Customer drJekill=new Customer();
drJekill
...
...
Customer mrHide= drJekill;
mrHide
HEAP
Customer
...

Figura 3.5 – Riferimenti e oggetti.

In parole povere, nella memoria heap abbiamo una sola copia di un oggetto di classe Customer (drJekyll e mrHide sono quindi la stessa persona!).

Il fatto che le due variabili si riferiscano allo stesso oggetto in memoria fa sì che qualsiasi delle due variabili si utilizzi per riferirsi a esso avrà effetto sullo stesso oggetto.

Supponiamo per esempio che la classe Customer abbia anche una proprietà Name, per mantenere il nome del cliente (non abbiamo ancora visto come definire una proprietà, ma l'esempio che segue è abbastanza chiaro ugualmente), e che si assegni a tale proprietà un valore per mezzo della variabile drJekyll:

```
drJekyll.Name="Henry";
```

Poiché anche mrHyde referenzia lo stesso oggetto, stampando il valore della sua proprietà Name, si otterrà esattamente lo stesso nome assegnato alla prima variabile:

```
Console.WriteLine("{0} Jekyll", drJekyll.Name);
Console.WriteLine("{0} Hyde", mrHyde.Name);

//risultato
//Henry Jekyll
//Henry Hyde
```

Per chiarire eventuali dubbi, confermiamo che ciò non vale per variabili di tipo valore, perché esse memorizzano il loro valore direttamente nella memoria stack. Quando si assegna a una variabile di un tipo valore una seconda variabile, viene copiato direttamente il valore di quest'ultima:

```
int a=123;
int b=a; //viene eseguita una copia del valore di a, quindi assegnato a b lo stesso valore 123
b=0; //cambia solo il valore di b, a rimane uguale a 123
```

Accesso ai membri

I membri di una classe sono tutti gli elementi che contengono i dati delle istanze della classe stessa, campi ed eventi, o che permettono di accedere alle sue funzionalità per manipolarne i dati, cioè metodi, proprietà e così via.

Per interagire con un membro della classe basta utilizzare su un oggetto l'operatore punto (.).

La classe String, per esempio, mette a disposizione dello sviluppatore decine di membri, fra cui la proprietà Length, che serve a ricavare la lunghezza o numero di caratteri che la compongono, e il metodo Contains per verificare se all'interno di una stringa è contenuta una sequenza di caratteri specifica:

```
string str="hello world";
int lunghezza=str.Length; //restituisce 11
```

```
bool b=str.Contains("world"); //restituisce true
```

La seconda istruzione accede alla proprietà Length che restituisce un valore di tipo int, la terza invece accede al metodo Contains e utilizza la sintassi di invocazione, che prevede l'utilizzo delle parentesi tonde subito dopo il nome, all'interno delle quali vengono indicati gli eventuali parametri da inviare al metodo.

Nell'esempio, il metodo Contains viene invocato passando a esso un solo parametro, di tipo string, che indica la sequenza di caratteri da ricercare nella stringa originale. Il metodo restituisce un valore booleano a indicare se la stringa str contiene il parametro.

Per utilizzare i membri di una classe che non abbiamo definito noi, e di cui quindi non conosciamo il contenuto e il funzionamento, è necessario in generale avere a disposizione la documentazione.

Per esempio, la documentazione della classe String e di tutte le classi del .NET Framework è disponibile e ricercabile online sul sito http://msdn.microsoft.com (date un'occhiata in particolare alla classe String all'indirizzo http://msdn.microsoft.com/it-it/library/system.string.aspx e provate a cercare i membri Length e Contains).

Nel caso in cui comunque abbiate già installato Visual Studio 2015, potete notare come esso, per mezzo della funzione di intellisense, renda la vita dello sviluppatore molto più semplice. Se infatti provate ad accedere ai membri di un qualunque oggetto digitando il punto subito dopo l'identificatore di una variabile, l'IDE vi mostrerà un elenco dei membri, con delle icone che ne indicano la tipologia, e che è possibile ricercare digitando le prime lettere che ne compongono il nome.

La Figura 3.6 mostra per esempio l'intellisense attivato su una variabile di tipo String.

```
string str = "hello";

str.L
```

- IndexOf
- IndexOfAny
- Insert
- IsNormalized
- LastIndexOf
- LastIndexOfAny
- Length
- Normalize
- PadLeft

int string.Length { get; }

Gets the number of characters in the current string object.

Figura 3.6 – Intellisense di Visual Studio 2015 che visualizza i membri di un oggetto.

La parola chiave **null**

La parola chiave null è un valore letterale che rappresenta un riferimento nullo, cioè nessun riferimento a un oggetto. null quindi è il valore predefinito di ogni variabile di un tipo riferimento.

È possibile assegnare esplicitamente il valore null, quindi a una variabile: in tal modo l'oggetto precedentemente referenziato dalla variabile non sarà più utilizzato e può essere cancellato dalla memoria:

```
string str=null;
```

indica che la variabile str non punta a nessun oggetto.

NOTA	Il valore null non può essere assegnato a una variabile con tipo implicito, perché in questo caso deve essere conosciuto il tipo del valore assegnato e null non ha alcun tipo.

Naturalmente se più oggetti si riferiscono allo stesso oggetto, assegnando il valore null a uno di essi, i rimanenti continuano a puntare all'oggetto in memoria.

Riprendendo l'esempio con la classe Customer:

```
Customer drJekyll=new Customer();
Customer mrHyde=drJekyll;
drJekyll.Name="Henry";
Console.WriteLine("{0} Jekyll", drJekyll.Name);
Console.WriteLine("{0} Hyde", mrHyde.Name);
```

entrambe le variabili puntano allo stesso oggetto in memoria heap.

Se a questo punto assegno il valore null alla prima variabile, mrHyde continuerà a referenziare l'oggetto:

```
Console.WriteLine("{0} Hyde", mrHyde.Name);
Console.WriteLine("{0} Jekyll", drJekyll.Name); //errore
```

la seconda istruzione stavolta restituirebbe un errore.

Infatti, quando un oggetto viene utilizzato invocando un metodo o leggendo una proprietà per mezzo di una variabile che punta all'oggetto, è necessario che l'oggetto stesso sia diverso da null.

In caso contrario si verifica un errore noto come "riferimento nullo". Per esempio:

```
string str="hello world";
int lunghezza=str.Length; //restituisce 11
str=null;
```

```
lunghezza=str.Length; //errore!
```

La variabile lunghezza viene valorizzata con il numero intero che rappresenta la lunghezza della stringa contenuta nell'oggetto str (o meglio l'oggetto a cui si riferisce la variabile str).

Se la variabile str viene impostata al valore null, essa non punta più ad alcun oggetto. Il successivo utilizzo di Length non va quindi a buon fine e genera una cosiddetta eccezione NullReferenceException, uno degli errori più frequenti.

È compito del programmatore verificare quali sono i casi in cui un oggetto potrebbe essere null e quindi evitare di utilizzarlo per accedere ai suoi membri.

La parola chiave **void**

Per esprimere il fatto che un metodo non restituisce alcun valore, e quindi nessun tipo, C# utilizza la parola chiave void: esso non è un tipo speciale ma indica proprio la mancanza di tipo.

Abbiamo già incontrato void nella definizione del metodo Main, appunto nel caso in cui non si voglia che il programma restituisca alcun valore.

Valori predefiniti

Nel capitolo precedente abbiamo visto che ogni variabile prima di essere utilizzata deve avere un valore assegnato.

A differenza delle variabili locali, le variabili definite all'interno di un tipo (per esempio i campi di una classe o di una struct) vengono automaticamente inizializzate al loro valore di default.

Tutti i tipi semplici dichiarano un costruttore di default che si occupa di inizializzare i campi, assegnando a essi un valore corrispondente a una sequenza di bit 0, che corrisponde ai valori predefiniti riportati nella seguente tabella.

Tabella 3.1 **- Valori predefiniti.**

Tipo	Valore
sbyte	0
byte	0
short	0
ushort	0
int	0
uint	0
long	0L
ulong	0UL
float	0.0F
double	0.0D
decimal	0.0M
bool	False
char	'\0'
tipi riferimento	null
tipi enum	0 (vedere enumerazioni)

È abbastanza semplice ricordarsi il valore di default, che è 0 per i tipi numerici, il carattere nullo '\0' per i char, false per il tipo bool e null per i tipi riferimento.

In pratica per ogni campo viene invocato il rispettivo costruttore, quindi un campo int ha valore predefinito 0, perché l'istruzione seguente:

```
int i=new int();
```

ha lo stesso effetto di:

```
int i = 0;
```

NOTA	Per ricavare il valore di default di ogni tipo si può utilizzare la parola chiave `default` (utile quando avremo a che fare con i generics). Il valore predefinito di un `int` si può ricavare per esempio con l'istruzione `default(int)`.

Le struct

Un tipo struct, o struttura, è un tipo valore che deriva da ValueType, che a sua volta deriva da System.Object.

Nella libreria di classi .NET Framework tutti i tipi di dati primitivi (Boolean, Byte, Char, DateTime, Decimal, Double, Int16, Int32, Int64, SByte, Single, UInt16, UInt32 e UInt64) sono definiti come strutture.

Una struct è simile a una classe, con le differenze fondamentali che una struct è un tipo valore e che essa non supporta l'ereditarietà (anche se derivano come detto in modo implicito da ValueType). Ciò implica che dai tipi predefiniti suddetti non possono essere derivati altri tipi.

Una struct si utilizza al posto di una classe quando si necessita di un tipo "leggero", dalla struttura semplice, che occupa poca memoria.

Per esempio, un tipo numerico personalizzato potrebbe essere un ottimo pretesto per creare un nuovo tipo valore utilizzando una struct.

Un altro tipico utilizzo potrebbe essere quello in cui si prevede di aver a che fare con un insieme di oggetti di tale tipo, per esempio array, che quindi in memoria potrebbero essere memorizzati in segmenti contigui (gli oggetti nell'area heap devono essere referenziati mediante riferimenti e accedere alla loro posizione in memoria potrebbe richiedere più tempo rispetto a un accesso diretto in memoria stack).

In genere viene implementata una struct personalizzata quando la struttura dati da realizzare è rappresentabile come un insieme di tipi semplici da raggruppare in un solo tipo.

La sintassi per implementare una struct è praticamente identica a quella vista per le classi:

```
struct <NomeStruct>
{
//membri struct
}
```

La sezione con i membri contiene i componenti che rappresentano la struttura del tipo dati, nel formato:

```
modificatoreAccesso tipo nome;
```

Il modificatore di accesso indicato prima del nome, che è utilizzabile come vedremo anche per le classi, serve a stabilire se il codice esterno alla struct può accedere e utilizzare i suoi membri.

Nel caso delle struct si utilizza il modificatore public per ogni campo, a indicare appunto che tali campi sono pubblicamente accessibili.

Per esempio supponiamo di voler rappresentare un punto nello spazio tridimensionale, composto da tre coordinate x, y, z. In tal caso basterebbe utilizzare tre valori, magari di tipo double, ma è comodo trattarli come un unico oggetto e, inoltre, sarebbe utile avere a disposizione dei metodi per manipolarli come unico oggetto.

Un'implementazione di questa struttura potrebbe essere la seguente:

```
struct Point3D
{
public double X;
public double Y;
public double Z;
}
```

Una volta definita tale struct possiamo definire una variabile del tipo Point3D come fatto per ogni altro tipo:

```
Point3D punto;
```

Essendo un tipo valore non è necessario utilizzare l'operatore new, ma possiamo direttamente utilizzare i suoi campi:

```
Point3D p;
p.X = 1;
p.Y = 2;
p.Z = 3;
Console.WriteLine("({0},{1},{2})",p.X,p.Y,p.Z);
```

Anche qui però i campi devono essere esplicitamente inizializzati, altrimenti l'istruzione WriteLine restituirebbe un errore di compilazione, indicante un possibile uso di campi non assegnati. Al contrario, se costruissimo l'oggetto p utilizzando il costruttore, i campi sarebbero automaticamente inizializzati al valore di default del proprio tipo (vedere il paragrafo precedente).

Ogni struct, così come avviene per le classi, ha un proprio costruttore (il metodo con lo stesso nome del tipo) predefinito, quello senza parametri, quindi non è necessario implementarne uno; anzi, a differenza delle classi, non è consentito personalizzare il costruttore predefinito e facendolo si avrebbe un errore di compilazione.

Quindi costruendo un oggetto Point3D invocando il costruttore della classe, il valore dei tre campi sarebbe per tutti uguale a 0:

```
Point3D p2 = new Point3D();
Console.WriteLine("p2 ({0},{1},{2})", p2.X, p2.Y, p2.Z); //stampa p2 (0,0,0)
```

Infatti i campi delle struct vengono inizializzati automaticamente ai valori predefiniti dei rispettivi tipi e, a differenza delle classi, non è possibile fornire dei valori di inizializzazione nella loro dichiarazione.

La seguente dichiarazione di struct è quindi errata:

```
struct Point3D
{
public double X=1; //errore
public double Y;
public double Z;
}
```

Il compilatore restituirebbe un errore, a indicare che non si possono avere inizializzatori dei campi e delle proprietà di una struct.

Le enumerazioni

Un'enumerazione è un tipo valore, utile quando si necessita di un tipo che raggruppi dei valori con una qualche relazione fra di loro.

Per esempio si può definire un'enumerazione dei giorni della settimana, dei mesi dell'anno, delle modalità di apertura di un file e così via. Sono tutti casi in cui i valori possibili sono ben determinati e limitati numericamente.

Un tipo enumerazione si crea utilizzando la parola chiave enum e dichiarando l'elenco dei suoi possibili valori, separati da una virgola:

```
enum <NomeTipo>
{
<elemento1>,
<elemento2>,
…
}
```

Internamente un tipo enum viene implementato come una struct derivata dalla classe System.Enum (vedere il Capitolo 6 per ulteriori dettagli sulle struct).

Immaginiamo per esempio di dover conservare in una variabile uno fra i sette possibili giorni della settimana. Possiamo quindi definire un tipo enum, chiamandolo GiorniSettimana, nel seguente modo:

```
enum GiorniSettimana
{
Lunedì,
Martedì,
Mercoledì,
Giovedì,
Venerdì,
Sabato,
Domenica,
}
```

L'ultimo membro dell'enumerazione può anche essere seguito da una virgola, come in C++. Ciò fornisce maggiore flessibilità nell'aggiungere o rimuovere valori dall'enumerazione: nell'esempio precedente basta commentare la riga con Domenica e l'enumerazione potrà essere compilata ugualmente, senza necessità di rimuovere la virgola dopo Sabato.

Per utilizzare un valore dell'enumerazione in un'istruzione del programma, basta accedere a esso come se fosse un membro di una classe, cioè con l'operatore punto:

```
GiorniSettimana giorno=GiorniSettimana.Lunedì;
```

A ogni enumerazione è anche associato un tipo intero sottostante che, se non altrimenti specificato, è il tipo int, in maniera che a ogni nome dell'elenco corrisponda anche un valore intero.

Non specificando esplicitamente il valore, esso parte da 0 con il primo elemento e poi viene incrementato di 1 per i seguenti.

Nell'esempio di enumerazione precedente, l'elemento Lunedì corrisponderà a 0, Martedì a 1 e così via.

Possiamo verificare che ogni elemento ha un valore int, convertendolo esplicitamente:

```
int i=(int)GiorniSettimana.Lunedì; //i=0
```

Al contrario, è possibile convertire un numero in un tipo enum, sempre che il valore numerico ricada nell'intervallo dei valori possibili:

```
GiorniSettimana giorno=(GiorniSettimana) 1; // = GiorniSettimana.Martedì
giorno=(GiorniSettimana)8; //errore
```

Se invece si vuole ottenere una stringa con il nome del membro di un'enumerazione basta utilizzare il metodo ToString:

```
string nomeGiorno=GiorniSettimana.Lunedì.ToString();
```

Per modificare il tipo da usare per gli elementi, basta indicarlo con i due punti (:) subito dopo il nome dell'enumerazione. Per esempio:

```
enum Colori : uint
{
Rosso = 0xFF0000,
Verde = 0x00FF00,
Blu = 0x0000FF
}
```

In tal modo ogni elemento avrà un valore di tipo uint.

Se invece si vuol assegnare esplicitamente un valore a ogni nome dell'enumerazione, basta inizializzare i membri come se fossero variabili del tipo sottostante:

```
enum Colori
{
Rosso = 1,
Giallo= 2,
Verde = 3,
Marrone=4,
Red = 1,
}
```

Notate che nell'esempio precedente sia il membro Rosso che quello Red hanno stesso valore 1. Non ha probabilmente un senso pratico, ma non è un errore. Infatti è possibile ripetere più

volte lo stesso valore, anche assegnando il valore mediante un membro precedente dell'enum:

```
enum Colori
{
Rosso = 1,
Giallo= 2,
Verde = 3,
Marrone=4,
Red=Rosso,
}
```

Inoltre, non è necessario assegnare un valore a tutti i membri; in tal caso sarà il compilatore ad assegnare i valori mancanti secondo il seguente algoritmo:

- se il primo membro non ha un valore esplicito gli viene assegnato 0;
- se a un membro non viene assegnato un valore esplicito, gli si assegna un valore pari a quello precedente aumentato di 1.

Quindi, se avessimo la seguente dichiarazione di enumerazione:

```
enum Colori
{
Rosso, // 0
Giallo=4
Verde, // 5
Marrone=5,
Nero //6
}
```

al primo membro dell'enumerazione, Rosso, viene assegnato il valore 0. Il secondo, Giallo, ha un valore esplicito uguale a 4. Il terzo non ha un valore assegnato esplicitamente, quindi assumerà valore pari al precedente aumentato di 1, cioè Verde avrà valore 5. Anche Marrone ha un valore uguale a 5, ma non è un problema, come detto precedentemente. Infine, Nero, assumerà valore pari al precedente più 1, cioè 6.

Flag di bit

Negli esempi precedenti abbiamo definito i membri di un'enumerazione come mutuamente esclusivi, per esempio un colore può assumere uno solo dei valori dell'enumerazione Colori.

Un'altra modalità per definire e utilizzare le enumerazioni è quello di assegnare dei valori che siano combinabili fra di loro, in maniera che ogni variabile del tipo enum rappresenti non un singolo valore ma appunto una combinazione dei valori possibili.

Il primo passo per creare un simile tipo enum è utilizzare un attributo Flags (vedremo nel seguito del libro cos'è un attributo) indicandolo fra parentesi quadre:

```
[Flags]
enum GiorniSettimana
{
```

```
…
}
```

Supponiamo quindi di voler impostare una variabile di tipo GiorniSettimana in maniera da poter rappresentare contemporaneamente più di un giorno. In questo caso è opportuno assegnare i valori dei singoli membri dell'enumerazione in maniera che ognuno di essi sia una potenza del 2:

```
[Flags]
enum GiorniSettimana
{
Lunedì=1,
Martedì=2,
Mercoledì=4,
…
Domenica = 128,
}
```

Chi ha dimestichezza con la numerazione binaria avrà già intuito che scrivendo i suddetti valori in base 2, in ognuno di essi un solo bit sarà pari a 1:

```
[Flags]
enum GiorniSettimana
{
Lunedì=1, //00000001
Martedì=2, //00000010
Mercoledì=4, //00000100
…
Domenica=128,//10000000
}
```

Ciò rende molto semplice combinare due o più valori in una singola variabile, in particolare con un'operazione di OR (chi non ha mai avuto a che fare con operazioni bit a bit, può tornare a questo paragrafo dopo aver affrontato l'argomento nel prossimo capitolo).

Infatti, combinando in OR per esempio i valori Lunedì e Martedì, otterremmo un valore numerico di 3 perché i due bit più a destra sono pari a 1:

```
GiorniSettimana giorniChiusura=GiorniSettimana.Lunedì | GiorniSettimana.Martedì; // = 00000011 = 3
```

Nella precedente istruzione l'operatore | indica l'operazione di OR bit a bit.

NOTA	Non è obbligatorio utilizzare l'attributo Flags, ma esso serve soprattutto per rappresentare il valore combinato come stringa. Infatti, anziché visualizzare il valore 3 nell'esempio precedente, verrebbe visualizzato il nome dei valori combinati Lunedì,Martedì.

Per verificare se una variabile contiene un determinato valore dell'enumerazione, eventualmente combinato con altri, è necessario verificare se il corrispondente bit sia pari a 1, quindi combineremo in AND il valore con quello da verificare:

```
if ((giorniChiusura & GiorniSettimana.Lunedì) == GiorniSettimana.Lunedì)
{
Console.WriteLine(giorniChiusura);
}
```

L'istruzione if verifica se la combinazione in AND (mediante l'operatore &, vedere il prossimo capitolo) della variabile giorniChiusura e del valore Lunedì dell'enumerazione ha il bit corrispondente a Lunedì (che è il primo bit a destra) impostato al valore 1.

La combinazione in AND quindi restituisce 1 nella posizione in cui entrambi i bit delle variabili confrontate sono 1:

```
00000011 Lunedì|Martedì
00000001 Lunedì
------------- AND &
00000001
```

A partire da .NET 4.0 un altro modo per verificare la condizione è quello di utilizzare l'apposito metodo HasFlag della classe System.Enum, e quindi disponibile in ogni enumerazione:

```
if (giorniChiusura.HasFlag(GiorniSettimana.Lunedì))
{
Console.WriteLine(giorniChiusura);
}
```

Tipi nullable

I tipi riferimento possono rappresentare un valore non esistente mediante un riferimento null. I tipi valore, per definizione, invece devono necessariamente avere un valore.

In alcuni casi sorge però l'esigenza di dover trattare una variabile di tipo valore come se non avesse ancora alcun valore assegnato: in particolare avendo a che fare con i database può capitare spesso tale situazione.

In C# 2.0 è stato introdotto il supporto a questa eventualità mediante i cosiddetti tipi nullable.

Per fare in modo che un qualsiasi tipo valore possa assumere anche valore null si utilizza l'operatore ? come suffisso della dichiarazione del tipo:

```
int? numNullable=null;
numNullable=1;
```

Con questa dichiarazione il compilatore produce un tipo che può comportarsi come il normale tipo valore int, e infatti è possibile assegnargli un valore intero, oppure come tipo che può anche non avere alcun valore definito, mediante l'assegnazione di null.

Torneremo in maniera approfondita sul concetto di nullable type nel Capitolo 9, quando parleremo di generics, in quanto la sintassi qui presentata è semplicemente una sorta di abbreviazione che nasconde un'implementazione più complessa.

Tipi anonimi

Nel Capitolo 2 abbiamo discusso la parola chiave var per la definizione delle cosiddette *variabili di tipo implicito*.

Utilizzandola in congiunzione con la parola chiave new è possibile creare delle variabili di tipo anonimo.

I *tipi anonimi* forniscono un modo per definire una classe dichiarandone le proprietà, ma senza la necessità di creare una definizione di classe vera e propria.

In parole povere un tipo anonimo è una classe senza nome, derivata naturalmente da System.Object, la cui struttura viene ricavata in maniera implicita dal compilatore, allo stesso modo in cui esso ricava e assegna il tipo a una variabile implicita var. Per esempio:

```
var cliente=new {Nome="mario", Cognome="rossi", Età=50};
```

L'utilizzo di var in questo caso è obbligatorio perché non abbiamo nessun tipo da dichiarare.

La parola chiave new è seguita da un blocco chiamato *inizializzatore* dell'oggetto, all'interno del quale vengono definite e assegnate direttamente delle proprietà, il cui tipo viene anch'esso definito implicitamente dal compilatore.

Dall'esempio precedente il compilatore genererà una classe con due proprietà Nome e Cognome di tipo string e una proprietà Età di tipo int.

In questo modo è possibile ora leggere le proprietà dell'oggetto come per un'istanza di qualsiasi altro tipo:

```
string nome=cliente.Nome;
Console.WriteLine("{0} {1}, anni {2}", cliente.Nome, cliente.Cognome, cliente.Età);
```

NOTA	Non è permesso scrivere le proprietà di un'istanza di un tipo anonimo, in quanto esse sono di sola lettura. Questa opportunità invece è consentita in VB.NET.

Un tipo anonimo è utile per creare un contenitore da utilizzare temporaneamente in un blocco di codice, per esempio all'interno di un metodo; infatti, non avendo un tipo dichiarato, non è possibile utilizzarli come parametri di altri metodi o come valori di ritorno.

Essi comunque non vanno in conflitto con la caratteristica di linguaggio fortemente tipizzato di C#, in quanto il compilatore in ogni caso crea un tipo ben definito.

NOTA I tipi anonimi sono stati introdotti in C# 3.0, insieme a LINQ, in quanto è proprio in questo ambito che trovano l'utilizzo primario; li vedremo quindi in maggior dettaglio proprio nel Capitolo 11 dedicato a LINQ.

Operatore **typeof**

Anticipando di qualche pagina il capitolo dedicato agli operatori di C#, introduciamo ora l'operatore typeof che permette di ottenere informazioni su un particolare tipo. Esso restituisce un oggetto System.Type per un tipo al quale viene applicato. La sintassi di utilizzo è la seguente:

```
Type type=typeof(int);
```

A differenza del metodo GetType che ogni tipo possiede in quanto derivato da Object, e che si può invocare su un'istanza, l'operatore typeof agisce su un tipo. Inoltre il primo viene valutato a runtime, mentre typeof viene valutato staticamente a tempo di compilazione:

```
int i=0;
Type t1= i.GetType();
Type t2= typeof(int);
Console.WriteLine(t1.FullName);// System.Int32
Console.WriteLine(t2.FullName);// System.Int32
```

La sua utilità consiste principalmente nel poter ottenere quindi informazioni su un tipo senza dover necessariamente creare un'istanza.

La classe Type possiede proprietà per ricavare informazioni sul nome, sull'assembly, sul tipo da cui deriva e altre caratteristiche che fanno parte dell'argomento reflection, che verrà anch'esso approfondito nel Capitolo 14.

Conversioni di tipo

Una *conversione di tipo* consente di trattare un particolare oggetto o espressione come se fosse di un tipo differente.

Le conversioni in C# si possono distinguere in *implicite* ed *esplicite* e ciò determina se è necessario un apposito operatore fornito dallo sviluppatore per eseguire una conversione.

NOTA	Il prossimo capitolo presenterà in dettaglio, fra gli altri, gli operatori utili per la conversione esplicita di un tipo verso un altro.

La distinzione fra implicite ed esplicite indica innanzitutto che nella prima forma una conversione da un tipo all'altro è sempre possibile, in quanto può avvenire fra tipi simili fra di loro, e il compilatore conosce abbastanza di essi per procedere implicitamente alla conversione.

Nel caso di una conversione esplicita, invece, la struttura dei tipi è più complessa ed è quindi possibile solo in determinate circostanze, tanto che lo sviluppatore deve farne esplicita richiesta nel programma.

Consideriamo per esempio il seguente codice C#:

```
byte b=123
int i=b;
```

Alla variabile `i` viene assegnato il valore di tipo `byte` b, e quindi viene effettuata una copia del valore 123 creando un valore di tipo `int`. Tutto ciò avviene dietro le quinte senza problemi, in maniera implicita.

Se provassimo a fare una cosa analoga, in senso opposto:

```
int i=123;
byte b=i;
```

il compilatore, nonostante il valore 123 sia perfettamente compatibile con il tipo `byte`, segnalerebbe che è impossibile convertire implicitamente il tipo `int` in `byte` perché non sa se a tempo di esecuzione il valore di `i` sarebbe tale da sforare il valore massimo di un tipo `byte`.

Inoltre nel messaggio di errore, se provate a compilare, verrebbe dato anche un suggerimento del tipo: "Esiste una conversione esplicita. Hai dimenticato un cast?".

Conversioni implicite

Le *conversioni implicite* consentono di convertire il valore di un'espressione di un certo tipo verso un altro tipo numerico in maniera automatica, senza nessuna sintassi particolare, in quanto se ne occupa direttamente il compilatore.

In generale ciò avviene semplicemente mediante un'assegnazione oppure l'utilizzo di un valore dov'era previsto un altro tipo.

Nell'esempio precedente, il valore di una variabile byte è stato convertito in un valore int:

```
byte b=123
int i=b;
```

Perché la conversione sia possibile il tipo di destinazione deve essere tale da poter contenere il valore di partenza. Quindi, avendo già conosciuto i vari tipi numerici primitivi di C#, si può dire che una conversione avviene implicitamente se il tipo destinazione è formato da un numero di bit maggiore del tipo originale.

Per esempio, nel caso precedente, il tipo int può sicuramente contenere un tipo byte in quanto int è formato da 32 bit (cioè 4 byte).

Nel caso contrario, invece, i 4 byte del tipo int non possono essere convertiti esplicitamente in una variabile formata da un solo byte.

I casi più complessi sono quelli che coinvolgono delle espressioni, in quanto utilizzando degli operatori il tipo del risultato potrebbe essere diverso da quello degli operandi.

Per esempio sommando 2 byte, molto probabilmente il risultato finale non sarà più contenibile in un solo byte, e quindi l'operatore somma restituisce un int:

```
byte b1=150;
byte b2=200;
byte somma=b1+b2; //errore
```

Nell'ultima istruzione si avrebbe un errore di compilazione perché il tipo int restituito dalla somma non è convertibile implicitamente in byte.

La seguente tabella mostra le conversioni implicite consentite in C# per i tipi numerici.

Tabella 3.2 **- Conversioni implicite.**

Tipo partenza	Tipi destinazione
sbyte	short, int, long, float, double, decimal
byte	short, ushort, int, uint, long, ulong, float, double, decimal
short	int, long, 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
char	ushort, int, uint, long, ulong, float, double, decimal

Da notare che è possibile convertire implicitamente un valore char in uno numerico, ma non è possibile il contrario:

```
char ch = 'a';
int valChar = ch;
Console.WriteLine("{0} = {1}", ch, valChar); //a=97

ch=97; //errore, necessita cast esplicito
```

Oltre alle assegnazioni, le conversioni implicite possono avere luogo quando è necessario passare a un metodo degli argomenti di un certo tipo, non coincidenti con quelli previsti dal metodo stesso.

Per esempio, supponiamo di dover utilizzare il seguente metodo, che esegue la somma fra due numeri di tipo int:

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

Se avessimo due variabili da sommare, di tipo diverso da int, esse potrebbero essere utilizzate senza nessun accorgimento particolare nel caso in cui fosse possibile una conversione implicita di tipo.

Per esempio, se i due operandi da sommare sono due short, il metodo Somma è invocabile senza conversioni:

```
short s1=1, s2=2;
Somma(s1,s2);
```

Una conversione implicita può avvenire inoltre anche fra un tipo derivato e il suo tipo padre:

```
ClasseDerivata d = new ClasseDerivata();
ClasseBase b = d; // conversione implicita.
```

Per esempio, dato che tutti i tipi derivano da object, è possibile utilizzare una stringa assegnandola implicitamente a una variabile object:

```
Object obj="stringa"; //la stringa viene convertita implicitamente in object
```

Conversioni esplicite

Le *conversioni esplicite* sono conversioni di tipo che non possono avvenire in maniera automatica come quelle implicite, perché potrebbero fallire a runtime.

Come suggerisce il nome, per effettuare una conversione esplicita bisogna indicare al compilatore quale conversione eseguire, cioè indicare il tipo di destinazione.

Abbiamo già visto dei casi di conversione in cui si verifica un errore perché non è possibile convertire un valore verso un altro:

```
int i=123;
byte b= i; //errore
byte b1=150;
byte b2=200;
byte somma=b1+b2; //errore
```

Per consentire al compilatore di convertire il risultato dell'espressione di somma in un tipo byte, bisogna aggiungere un apposito operatore, detto operatore di cast, che consiste nell'indicare fra parentesi il tipo destinazione desiderato, posizionandolo prima del valore da convertire:

```
int i=123;
byte b= (byte) i;
```

In questo modo il compilatore viene forzato a effettuare la conversione della variabile i in byte.

La conversione esplicita non è da prendere sottogamba. Nel caso precedente tutto fila liscio, perché il valore 123 è tranquillamente rappresentabile con un solo byte.

In altri casi, o comunque con valori non predeterminabili a tempo di compilazione, ma che derivano dall'utilizzo e dalle operazioni eseguite nel programma, il valore potrebbe superare quello massimo consentito:

```
byte b1=150;
byte b2=200;
byte somma=(byte)(b1+b2);
```

In questo caso il compilatore procede tranquillamente come gli abbiamo indicato, ma il risultato assegnato alla variabile somma non sarà quello atteso di 350. Provate a stampare la variabile somma:

```
Console.WriteLine("{0} + {1} = {2}",by1, by2, somma); //stampa 150+200=94.
```

Da dove viene fuori il valore 94? Dando un'occhiata alla rappresentazione binaria delle variabili entrate in gioco nell'esempio si ha che:

```
b1 = 150 => 10010110
b2 = 200 => 11001000
b1+b2 = 350 => 101011110
```

Il valore di 350 non è più rappresentabile con 8 bit, quindi il bit più a sinistra verrà scartato; il risultato è il valore 01011110 che, in decimale, è appunto pari a 94.

NOTA	Nel capitolo seguente dedicato a espressioni e operatori vedremo come verificare se l'azione di cast andrà a buon fine o meno, per mezzo dell'operatore checked.

La classe Convert

La libreria di base di .NET contiene una classe denominata Convert, che fornisce dei metodi di conversione esplicita da un tipo primitivo a un altro, e da e verso il tipo DateTime, che serve a rappresentare data e orario.

La classe Convert è statica (non è necessario né possibile creare un'istanza della classe Convert) e ogni metodo ha un nome del tipo Convert.To*NomeDelTipoDestinazione*(*TipoOriginale*).

Per esempio, per convertire esplicitamente come nell'esempio del paragrafo precedente un int in un byte, si utilizza il metodo Convert.ToByte:

```
int i=123;
byte b=Convert.ToByte(i);
```

Naturalmente anche la classe Convert ha dei vincoli di funzionamento: nemmeno con i suoi metodi è possibile convertire per esempio in byte un valore maggiore del massimo consentito di 255. In questo caso però viene generata un'eccezione OverflowException:

```
byte b=Convert.ToByte(300); //overflow, valore troppo grande per essere convertito in byte
```

Fra i suoi metodi, quelli più utili e utilizzati sono quelli che convertono una stringa in un tipo numerico, sempre che la stringa sia in un formato corretto e convertibile:

```
string str = "123";
int i1 = Convert.ToInt32(str);
double d1 = Convert.ToDouble(str);
```

Se la stringa contiene un testo non interpretabile si avrebbe un'eccezione di formato.

Un'altra versione del metodo ToInt32, che accetta un secondo parametro, è utile quando si vuole convertire una stringa rappresentante un numero in una base diversa dalla decimale:

```
int esa=Convert.ToInt32("1AB", 16); //converte dall'esadecimale
int bin=Convert.ToInt32("10010111", 2); //converte dal binario
```

Il secondo parametro indica la base da utilizzare.

Infine segnaliamo un altro metodo, ChangeType, che permette di convertire genericamente un tipo in un altro di destinazione, che è possibile indicare come secondo parametro:

```
string str="123";
int i = (int)Convert.ChangeType(str, typeof(int));
```

Tale metodo potrebbe servire quando non si conosce a tempo di compilazione il tipo di destinazione verso cui convertire un valore, oppure per realizzare una classe che lavori con tipi diversi.

Gli array

Tutti i tipi visti finora consentono di memorizzare un singolo valore o un riferimento a un singolo oggetto.

Spesso è necessario, invece, trattare più valori di uno stesso tipo allo stesso tempo, senza dover dichiarare una variabile per ognuno di essi.

Un tipo utile per gestire queste situazioni è quello degli array, che fa parte della famiglia dei tipi riferimento, qualunque sia il tipo di valori che esso conterrà. Un array rappresenta quindi un numero fisso di elementi di uno stesso tipo. Gli elementi sono memorizzati in uno spazio di memoria contiguo, consentendo di ottenere la massima efficienza nell'accesso agli stessi.

Per dichiarare un array si deve indicare il suo nome e il tipo degli elementi che saranno contenuti al suo interno, secondo la seguente sintassi:

```
<nometipo>[] nomeVariabile;
```

Per esempio, il seguente è un array di numeri interi:

```
int[] vettore;
```

La variabile vettore identifica un array di numeri interi, ma non contiene ancora nessun elemento, né indica quanti numeri potrà contenere, cioè la sua lunghezza.

Per creare un array è necessario utilizzare l'operatore new e indicare il numero di elementi che esso potrà contenere. Ciò implica che è necessario conoscere a priori il numero di elementi della sequenza:

```
int[] vettore=new int[3];
```

La precedente istruzione alloca la memoria heap (essendo un array un tipo riferimento) necessaria a contenere tre oggetti di tipo int.

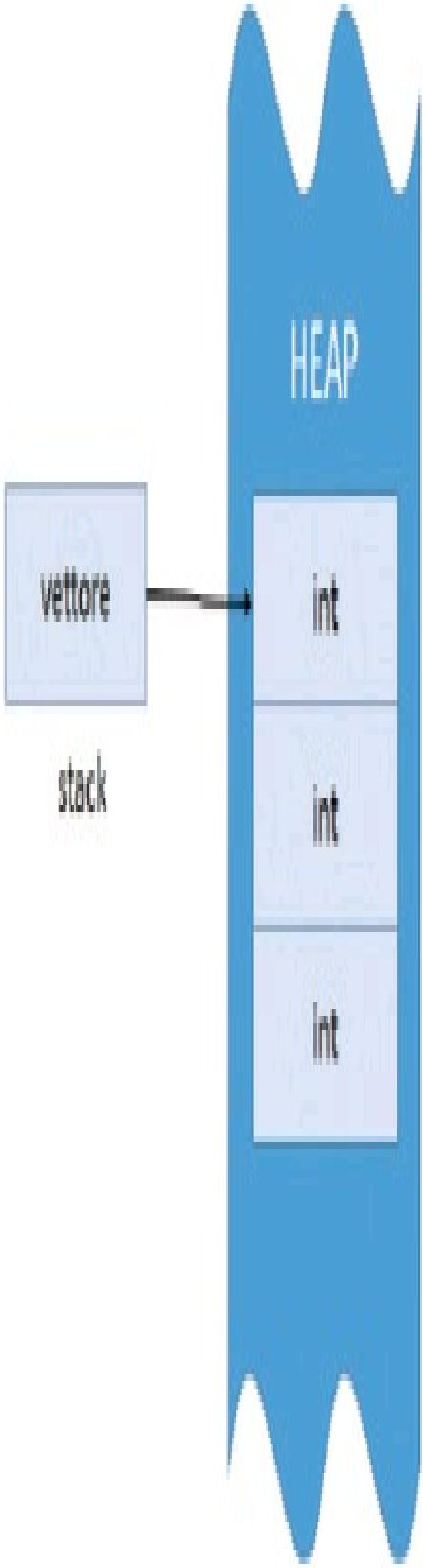

HEAP
vettore
int
int
int
stack

Figura 3.7 – Inizializzazione di un array in memoria heap.

Un array non può essere ridimensionato una volta definita la sua lunghezza. L'unico modo per farlo è creare un nuovo array, copiando eventualmente gli elementi esistenti in un altro array.

È possibile anche assegnare direttamente i valori degli elementi dell'array specificandoli mediante un inizializzatore di array, che ha come sintassi:

```
int[] vettore=new int[3]{ 4, 6, 8};
```

Il precedente array conterrà i tre numeri interi 4, 6, 8.

Se si utilizza l'inizializzatore è anche possibile omettere la dimensione dell'array, visto che il numero di elementi la indica implicitamente.

Infine un'ultima forma di inizializzazione è ancora più compatta:

```
int[] vettore= {4, 6, 8};
```

Accesso agli elementi

Per accedere ai singoli elementi di un array si utilizzano le parentesi quadre, indicando fra di esse l'indice dell'elemento, cioè la sua posizione, a partire da 0. Per esempio, per ottenere il primo elemento dell'array vettore:

```
int primo=vettore[0];
```

mentre per assegnare poi un nuovo elemento alla prima locazione:

```
vettore[0]=123;
```

Se si tentasse di accedere a un indice che va al di là della lunghezza dell'array si avrebbe un'eccezione di tipo IndexOutOfRangeException.

Essendo ogni variabile di tipo array un riferimento al vero e proprio array in memoria, se assegniamo una seconda variabile nel seguente modo:

```
int[] vettore2=vettore;
```

in realtà abbiamo assegnato solo il riferimento, quindi sia vettore sia vettore2 sono due variabili che referenziano lo stesso array di elementi.

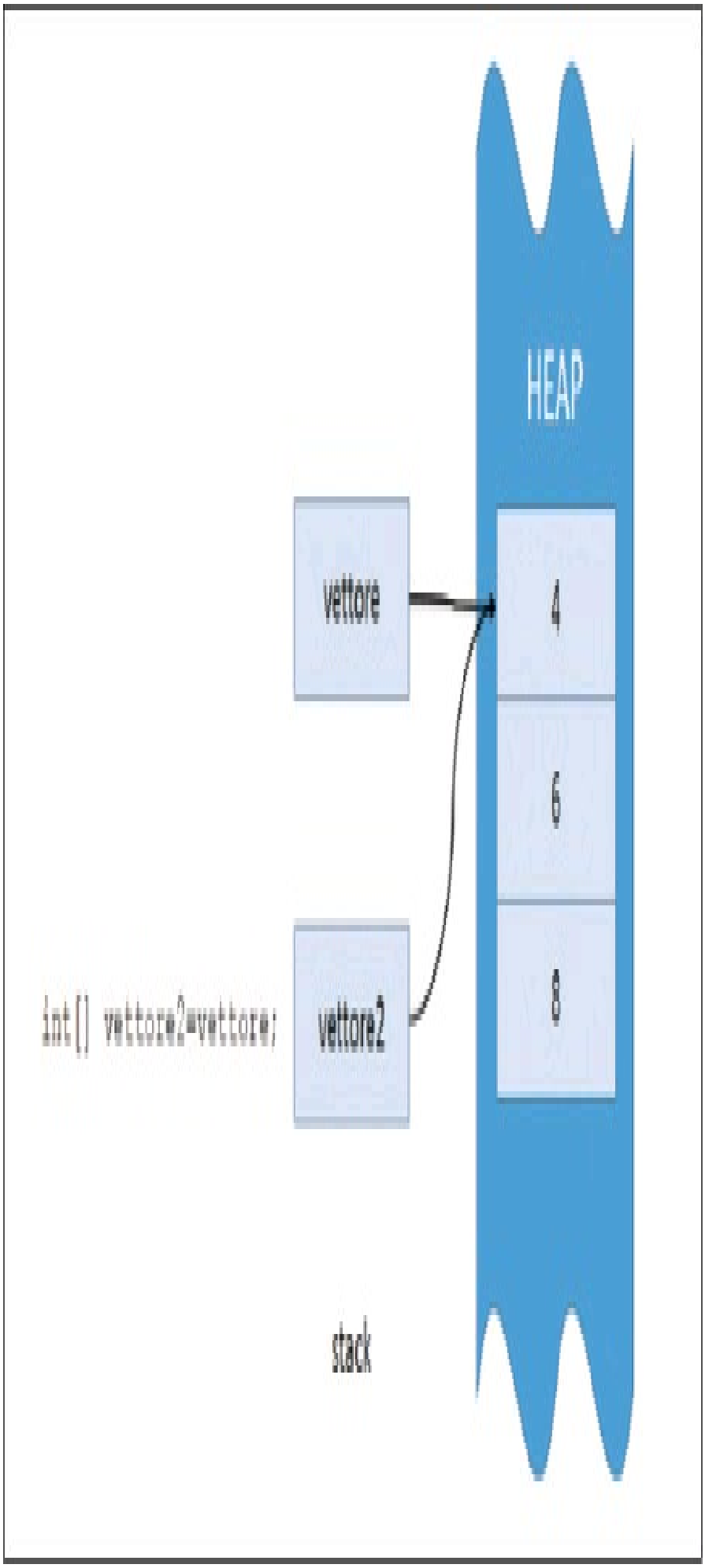

HEAP
vettore
4
6
8
int[] vettore2=vettore;
vettore2
stack

Figura 3.8 – Un array è un tipo riferimento.

Infatti, accedendo per esempio al secondo array per modificare il valore del primo elemento e poi andando a leggere tale elemento per mezzo della prima variabile, si otterrà lo stesso risultato:

```
vettore2[0]=123;
Console.WriteLine("{0}", vettore[0]);
Console.WriteLine("{0}", vettore2[0]); //stampa lo stesso valore
```

Nella pratica capiterà spesso di dover accedere in sequenza a tutti o parte degli elementi di un array, effettuando un ciclo su di essi. In tali casi si utilizzeranno apposite istruzioni, come per esempio il ciclo for (vedere il Capitolo 5) e le proprietà e i metodi della classe Array, per ricavare le dimensioni a runtime.

Array multidimensionali

Gli array normalmente hanno una singola dimensione, nel senso che sono una sequenza di elementi cui si può accedere con un solo indice.

C# consente di utilizzare array multidimensionali, accessibili quindi con più indici. Esistono due tipologie di array multidimensionali, quelli rettangolari e quelli jagged.

Array rettangolari

Gli array *rettangolari* consentono di realizzare strutture a più dimensioni, specificando la lunghezza di ognuna di esse. Basta definire le diverse dimensioni all'interno delle parentesi quadre e separarle con la virgola. Il caso più semplice è quello a due dimensioni, che crea una tabella o matrice:

```
int[,] matrice=new int[3,4];
```

Il risultato sarebbe una struttura con tre righe e quattro colonne, come quella nella Tabella 3.3.

Tabella 3.3 **- Un array rettangolare a due dimensioni.**

int	int	int	int
int	int	int	int
int	int	int	int

Per accedere agli elementi dell'array bisogna utilizzare la stessa sintassi, specificando però tutti gli indici necessari. Nel caso bidimensionale, per esempio, per accedere agli elementi sulla prima riga bisognerà scrivere:

```
matrice[0,0]=1;
matrice[0,1]=2;
matrice[0,2]=3;
matrice[0,3]=4;
```

È possibile utilizzare un inizializzatore di array anche in questo caso, se si conoscono i valori in anticipo.

Per esempio, sempre nel caso a due dimensioni dovremo utilizzare un altro blocco di parentesi graffe a racchiudere i singoli array da quattro elementi ciascuno:

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

Quindi, in generale, il numero di virgole per creare un array rettangolare è pari al numero di dimensioni diminuito di uno. Un array tridimensionale di stringhe, per esempio, sarà dichiarato come segue:

```
String[,,] tridimensionale;
```

Array irregolari

Il secondo tipo di array multidimensionale è detto *irregolare* o *jagged* e consente di costruire array a più dimensioni ma con più flessibilità rispetto a quelli rettangolari.

Per esempio, mentre un array rettangolare bidimensionale avrà per ogni riga sempre lo stesso numero di colonne, uno jagged array a due dimensioni avrà una struttura con più righe, ma ognuna contenente un numero di colonne differente.

Supponiamo per esempio di voler realizzare una struttura dati come la seguente:

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

In pratica bisogna realizzare un array di array. Quindi la creazione di un simile tipo di array prevede la seguente sintassi, indicando innanzitutto il numero di righe:

```
int[][] jagged=new int[3][];
```

Non è possibile dichiarare direttamente anche la seconda dimensione, in quanto appunto ogni riga avrà una lunghezza differente. Il seguente codice darebbe un errore di compilazione:

```
int[][] jagged=new int[3][3]; //errore
```

Per inizializzare le tre righe quindi si procederà assegnando a ognuna di esse gli ulteriori tre array che ormai sappiamo bene come realizzare:

```
jagged[0]=new int[1] {1};
jagged[1]=new int[] {2,3};
int[] terza = { 4, 5, 6 };
jagged[2] = terza;
```

Nell'esempio precedente, per ogni riga dell'array abbiamo usato delle possibili differenti sintassi. Per accedere ai singoli elementi si utilizzeranno ora le parentesi quadre, specificando i due indici:

```
int elemento=jagged[0][0]; // = 1
```

In maniera analoga a quanto visto, sarà possibile dichiarare e creare array di tipo jagged a più di due dimensioni. Per esempio, il seguente codice serve a dichiarare un array di stringhe a tre dimensioni:

```
string[][][] arrayStringhe;
```

NOTA	L'utilizzo di array nella pratica avverrà abbastanza raramente, a meno di esigenze specifiche. Nel Capitolo 9, dedicato ai tipi di collezioni messe a disposizione da .NET, vedremo una nutrita serie di classi che consentiranno di lavorare con collezioni di elementi in maniera molto più produttiva e semplice. Ciò non toglie che conoscere la struttura e il funzionamento degli array sia fondamentale.

Domande di riepilogo

1) Una struct può contenere un costruttore senza parametri esplicito: vero o falso?

2) In che modo si accede alla proprietà Nome di un oggetto persona?

a. string nome = persona->Nome;
b. string nome = persona:Nome;
c. string nome = persona(Nome);
d. string nome = persona.Nome;

3) Nell'enumerazione enum Giorni { Lun=1, Mar, Mer, Gio, Ven, Sab, Dom}, qual è il valore numerico di Dom ?

a. 1
b. 6
c. 7
d. null

4) Il termine *boxing* indica:

a. conversione di tipo valore in tipo riferimento
b. conversione di tipo riferimento in tipo valore
c. costruzione di un'istanza di una classe
d. lettura del tipo di una classe

5) Se f è di tipo float, quale delle seguenti conversioni è errata?

a. int i=f;
b. double j=f;
c. int k=Convert.ToInt32(f);
d. decimal d = (decimal)f;

6) Per ricavare il tipo di un'istanza obj, quale istruzione è quella corretta?

a. Type t = obj.Type;
b. Type t = obj.GetType();
c. Type t = typeof(obj);
d. Type t = Type.GetType(obj);

7) Per impostare il valore 1 come quinto elemento dell'array di interi int[] array=new int[5]; bisogna scrivere:

a. array[5]=1;
b. array[0]=1;
c. array[4]=1
d. array.Last=1;

8) L'istruzione var cliente= {Nome="mario", Cognome="rossi"} crea:

a. una variabile di tipo anonimo
b. una variabile di tipo cliente
c. una variabile di tipo var
d. errore di compilazione

Espressioni e operatori

C# permette di manipolare valori e variabili e combinarli assieme per formare delle espressioni più o meno complesse, mediante l'uso di varie tipologie di operatori.

Nei due precedenti capitoli abbiamo posto le basi del linguaggio C# e dello sviluppo di un programma, imparando a dichiarare e inizializzare le variabili. In questo capitolo continueremo questo percorso e vedremo come manipolare e combinare variabili e valori, combinandoli in espressioni più complesse per mezzo degli operatori.

Un'espressione è, in C#, una sequenza di operandi e operatori.

Il concetto di espressione è analogo a quello che abbiamo imparato alle scuole elementari, combinando fra loro degli operandi mediante le classiche operazioni di somma, sottrazione, moltiplicazione e divisione.

Proviamo, per esempio, a scrivere una semplice espressione matematica che effettua la somma di due numeri:

```
somma=2+3
```

In questa espressione 2 e 3 sono operandi, combinati mediante un operatore `+`, che indica l'operazione di addizione.

In un linguaggio di programmazione le cose si svolgono in maniera analoga. Potremmo assegnare i valori 2 e 3 a due variabili di tipo `int` e poi dichiarare un'ulteriore variabile chiamandola `somma`, nella quale verrà memorizzato il risultato dell'espressione, scritta combinando le due variabili `int` per mezzo di un operatore che, non sorprendetevi, è anche in C# scritto per mezzo del simbolo `+`.

In questo capitolo vedremo quindi quali sono le tipologie di operatori utilizzabili in C#, qual è il loro funzionamento e come utilizzarli per scrivere delle espressioni.

Le espressioni saranno poi fondamentali nel costruire istruzioni più o meno complesse per controllare il flusso di esecuzione di un programma, ma questo lo vedremo nel prossimo

capitolo.

Gli operatori

Le espressioni sono costruite per mezzo di operandi e operatori. Gli operatori indicano quali operazioni eseguire sugli operandi.

In C# gli operatori sono classificabili in tre categorie a seconda della loro *arietà*, cioè del numero di operandi su cui un operatore agisce.

Gli operatori, quindi, possono appartenere a una di queste tre categorie:

- unari – vengono applicati a un singolo operando e possono essere scritti in notazione prefissa, cioè prima dell'operando a cui si riferiscono (per es. `-x`), o postfissa, cioè subito dopo l'operando (per es. `x++`);
- binari – agiscono su due operandi in notazione infissa, cioè frapposti agli operandi, per esempio quasi tutti gli operatori matematici sono binari (`x+y`);
- ternari – esiste un solo operatore ternario, che è un operatore logico, e agisce appunto su tre operandi, in notazione infissa.

Le espressioni

Per effettuare dei calcoli e assegnare i risultati a una variabile è necessario scrivere delle istruzioni contenenti delle espressioni.

L'*espressione* più semplice è proprio quella di assegnazione del valore di una variabile:

```
int i=0; //espressione di assegnazione semplice
```

Le specifiche di C# indicano che un'espressione è una sequenza di operandi e operatori. Per esempio, il valore assegnato alla variabile può derivare da un calcolo eseguito da un'espressione matematica:

```
int i=1+2; //espressione
int a=i+1; //espressione
```

La seconda istruzione utilizza la variabile `i` e l'operatore `+` (che, come già detto, anche in C# indica l'operazione di somma) per eseguire un'addizione. L'operatore `=` invece serve ad assegnare il valore risultante alla variabile alla sua sinistra. Nella prima istruzione il risultato della somma 1+2 è assegnato alla variabile `i`, nella seconda, la somma del valore di `i` e di 1 viene assegnato ad `a`.

Precedenza e associatività degli operatori

Gli operandi di un'espressione possono a loro volta essere altre espressioni, che potremmo chiamare sotto-espressioni, e in tal modo si potranno comporre espressioni più o meno complesse. In tal caso l'espressione totale generale conterrà più di un operatore, per esempio:

```
int result=a + b * c;
```

La regola principale stabilisce che gli operandi vengono valutati da sinistra a destra. Quindi, nel caso sopra viene prima valutata a, poi b e infine c (a, b, c potrebbero essere espressioni più complesse, oppure chiamate ad altri metodi).

A questo punto bisogna applicare gli operatori ai rispettivi operandi, quindi per ottenere il risultato di un'espressione come questa è necessario stabilire quali operatori hanno la precedenza.

Per esempio, nell'espressione precedente ha maggior precedenza l'operatore di moltiplicazione * rispetto a quello di addizione + (questo l'abbiamo imparato alle elementari). Quindi l'espressione sarebbe valutata come se fosse scritta così:

```
int result=a + (b * c);
```

nella quale viene naturalmente valutata per prima la sotto-espressione raggruppata fra parentesi.

La Tabella 4.1 mostra gli operatori di C#, suddivisi in categorie e in ordine di precedenza: in alto quelli con precedenza maggiore e poi, scendendo lungo le righe, quelli con precedenza più bassa.

Per esempio gli operatori moltiplicativi, come l'operatore * di moltiplicazione e / di divisione, hanno precedenza maggiore rispetto a quelli additivi (addizione + e sottrazione -).

Le indicazioni x o y servono come variabili segnaposto, per mostrare il modo di utilizzo di un operatore. Per esempio, x++ indica che l'operatore ++ viene usato in modalità postfissa, posizionandolo subito dopo l'operando.

Tabella 4.1 **- Operatori e precedenza.**

Categoria	Operatori
primari	x.y x?.y f(x) a[x] a?[x] x++ x-- new typeof default checked unchecked delegate sizeof ->
unari	+ - ! ~ ++x --x (T)x await &x *x
moltiplicativi	* / %
additivi	+ -

shift	<< >>
confronto e verifica tipi	< > <= >= is as
uguaglianza	== !=
AND logico	&
XOR logico	^
OR logico	|
AND condizionale	&&
OR condizionale	||
null coalescing	??
ternario	?:
assegnazione ed espressioni lambda	= *= /= %= += -= <<= >>= &= ^= |= =>

NOTA	Nei prossimi paragrafi di questo capitolo e poi nel prossimo capitolo, dedicato al controllo di flusso, verrà mostrato in maggior dettaglio il significato e l'utilizzo di quasi ogni operatore (i pochi rimanenti verranno affrontati mano a mano nel resto del libro); per il momento soffermiamoci sulla modalità di valutazione delle espressioni.

Quando due operandi hanno la stessa precedenza e sono entrambi utilizzati in un'espressione, intervengono le regole di associatività.

A eccezione dell'operatore di assegnazione, tutti gli operatori binari sono associativi a sinistra, cioè l'espressione viene valutata applicando gli operatori da sinistra verso destra. Per esempio:

```
int result= 1 + 2 + 3;
```

viene valutata come:

```
int result = (1 + 2) + 3;
```

L'operatore di assegnazione e l'operatore ternario sono associativi a destra, quindi le espressioni vengono valutate da destra verso sinistra. Per esempio:

```
x = y = z;
```

viene raggruppata implicitamente come se fosse:

```
x = (y = z);
```

È sempre possibile modificare la precedenza e l'associatività, esplicitandola con l'uso delle parentesi e raggruppando le espressioni che devono essere valutate per prime.

Se, per esempio, volessimo prima far eseguire la somma e poi moltiplicare il risultato per 3, bisognerebbe scrivere l'espressione nel seguente modo:

```
int result = (5 + 4) * 3;
```

L'uso delle parentesi è consigliato nella scrittura di espressioni particolarmente complesse, perché si fa prima a valutare visivamente i gruppi di espressioni raggruppate che ricordarsi le regole di precedenza e associatività. Pensiamo per esempio a un'espressione come la seguente:

```
bool b = x< 10 && y > 20 || x < y-10 && y > 100;
```

Sicuramente sarebbe più immediato capirne il significato e valutarne il risultato se la scrivessimo così:

```
bool b = ((x<10) && (y>20) || (x < y - 10) && (y > 100));
```

In alternativa è possibile suddividere le espressioni complesse in più istruzioni:

```
bool b1 = (x<10) && (y>20);
bool b2 = (x < y - 10);
bool b3 = y>100;
bool result = b1 || b2 && b3;
```

Nell'ultima espressione basta ricordarsi, oppure verificare nella tabella di precedenza, che l'AND condizionale (&&) ha precedenza rispetto all'OR (||) e quindi l'espressione finale sarebbe valutata come:

```
bool result= b1 || (b2 && b3);
```

Promozioni numeriche

Utilizzando gli operatori binari con operandi numerici, capita spesso che questi ultimi siano di tipo differente. Per esempio, si potrebbero voler sommare un int e uno short, oppure dividere un int per un double e così via.

In generale tutti gli operatori hanno diverse forme di utilizzo possibili, per esempio l'operatore di moltiplicazione può essere utilizzato per agire su due int, su due float e così via.

Quando il compilatore C# incontra un'espressione che utilizza uno degli operatori binari +, –, *, /, %, &, |, ^, ==, != , >, <, >= e <=, esso converte implicitamente gli operandi verso uno stesso tipo che, tranne nel caso degli operatori di confronto, diviene anche il tipo del risultato dell'espressione. Tale conversione implicita verso un tipo comune è detta *promozione numerica* e utilizza le seguenti regole, nell'ordine in cui sono elencate:

- se uno degli operandi è di tipo decimal, anche il secondo operando è convertito in decimal (a meno che il secondo sia un double o float, nel qual caso si avrà un errore di compilazione e bisogna esplicitamente eseguire una conversione);
- altrimenti, se uno dei due operandi è double, l'altro operando viene convertito in double;
- altrimenti, se uno dei due operandi è float, l'altro operando viene convertito in float;
- altrimenti, se uno dei due operandi è ulong, l'altro operando viene convertito in ulong (a meno che non sia di tipo sbyte, short, int oppure long, nel qual caso si verifica un errore di compilazione);
- altrimenti, se uno dei due operandi è long, l'altro operando viene convertito in long;
- altrimenti, se uno dei due operandi è uint e l'altro operando è di tipo sbyte, short oppure int, entrambi vengono convertiti in long;
- altrimenti, se uno dei due operandi è uint, l'altro operando viene convertito in uint;
- altrimenti entrambi gli operandi sono convertiti nel tipo int.

Per esempio, se si esegue la somma fra un int e un double, entrambi gli operatori sono convertiti in double e quindi il risultato sarà anch'esso di tipo double.

Per verificare queste regole potete provare a scrivere un programmino che esegue, per esempio, la somma di due operandi di vario tipo e stampare il tipo della variabile risultato, a cui viene assegnato il valore in maniera implicita mediante la parola chiave var:

```
decimal dec1=2.0M;
int i=1;
var result=dec1+i;
Console.WriteLine("result è di tipo {0}", result.GetType().Name);
```

In questo caso il risultato è di tipo decimal, per la prima delle regole elencate sopra.

Operatori aritmetici

Gli *operatori aritmetici* sono quelli che permettono di eseguire le classiche quattro operazioni aritmetiche, più l'operatore di modulo o resto. Essi sono operatori binari e quindi associativi a sinistra, ma due di essi, + e -, possono essere anche utilizzati come operatori unari.

La Tabella 4.2 elenca i cinque operatori aritmetici disponibili in C#.

Il risultato di un'espressione che utilizza un operatore aritmetico dipende dal tipo degli operandi, ed eventualmente il compilatore dovrà effettuare una promozione numerica degli operandi secondo le regole esposte nel precedente paragrafo.

Tabella 4.2 **- Gli operatori aritmetici.**

Operatore	Nome	Descrizione
+	addizione	esegue l'addizione dei due operandi
-	sottrazione	sottrae il secondo operando dal primo
*	moltiplicazione	moltiplica i due operandi
/	divisione	divide il primo operando per il secondo
%	modulo	divide il primo operando per il secondo e restituisce il resto della divisione

C#, per le operazioni aritmetiche, permette di utilizzare gli operatori su operandi di tipo int, uint, long, ulong, float, double e decimal. Negli altri casi bisognerà implicitamente o esplicitamente effettuare una conversione di tipi.

I seguenti esempi mostrano l'utilizzo degli operatori aritmetici:

```
int a = 10;
int b = 4;
int somma= a + b; //14
int diff=a - b; //6
int molt= a * b; //40
```

Divisione fra interi

Il caso della divisione fra due numeri espone bene il funzionamento degli operatori aritmetici con operandi di diverso tipo. Quando si dividono due numeri interi, il risultato sarà un intero:

```
int quoz= a/b; //2
```

In questo caso l'espressione restituisce il valore 2 intero, cioè la parte decimale del risultato viene ignorata.

Se invece si volesse ottenere anche il valore dei decimali, uno dei due operandi dovrebbe essere espresso come float o double, oppure bisognerebbe convertire esplicitamente uno dei due:

```
double quoz= (double)a/b; //2.5
```

Quando si effettua una divisione di una variabile intera per un numero e si vuole un risultato con le cifre decimali, il numero dovrà essere espresso mediante un valore letterale double o float:

```
int num=10;
double quoz=num / 2.0d;
```

Se si tentasse di dividere una variabile numerica o un valore numerico per il valore 0, si avrebbe un'eccezione a runtime di tipo DivideByZeroException.

Resto della divisione

L'operatore *modulo* % restituisce il resto della divisione intera se gli operandi sono interi, altrimenti restituisce il resto con i decimali. Ecco qualche esempio di utilizzo dell'operatore modulo con operandi di diverso tipo e valore:

```
int resto= 0%123; // = 0, perché 0 diviso 123 fa 0 con resto di 0.
resto = 1%2; // =1 perché 1 diviso 2 fa 0 con resto di 1
resto = 3%3; // =0 perché 3 diviso 3 fa 1 con resto di 0
```

Dividendo due numeri reali si ottiene invece il resto con i decimali:

```
double resto = 2.0 % 1.5; // = 0.5, perché 2/1.5 fa 1 con resto di 0.5
resto = 2.5 % 1.5; //=1 perché 2.5/1.5 fa 1 con resto di 1
```

Operatori unari

Abbiamo già detto che gli operatori + e - possono essere utilizzati anche come operatori *unari* sebbene il + sia usato raramente, esso restituisce infatti lo stesso valore dell'operando, quindi è superfluo.

Per esempio, per cambiare di segno un valore e assegnarlo a una variabile possiamo usare l'operatore -:

```
int pos= 5;
int neg= -pos; //risultato -5
```

Concatenazione di stringhe

L'operatore + può essere utilizzato anche con due operandi di tipo string e in questo caso il suo effetto sarà quello di concatenare le due stringhe, restituendone una nuova composta dalla loro unione:

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

Tale funzionamento è reso possibile dal cosiddetto overloading dell'operatore +, che fornisce una versione particolare dell'operatore + per la classe String.

Nel seguito del libro vedremo come è possibile implementare l'overload di alcuni operatori per le classi definite dall'utente.

Incremento e decremento

Gli operatori di *incremento* ++ e *decremento* -- permettono rispettivamente di incrementare e decrementare di 1 il valore di una variabile numerica. Sono operatori unari e possono essere utilizzati sia in notazione prefissa, cioè indicando l'operatore prima dell'operando, sia postfissa, cioè posizionando l'operatore dopo l'operando. La prima forma ha cioè la seguente sintassi:

```
++x;
--x;
```

mentre nel secondo caso la sintassi prevede l'operatore subito dopo l'operando:

```
x++;
x--;
```

La differenza di comportamento delle due modalità si noterebbe solo se gli operatori venissero utilizzati in un'espressione il cui risultato dovrà essere assegnato a una variabile.

Nella notazione prefissa, quindi, l'operando viene prima incrementato o decrementato e poi il suo valore verrà utilizzato.

Al contrario, con la notazione postfissa, viene prima restituito il vecchio valore dell'operando e poi esso verrà decrementato o incrementato.

Vediamo qualche esempio per chiarire tale differenza di comportamento e utilizzo:

```
int x=1;
int y=x++; //restituisce x e poi incrementa
Console.WriteLine("x={0} ; y={1}", x,y); //stampa x= 2, y=1

x=1;
y=++x; //incrementa x e poi restituisce il valore
Console.WriteLine("x={0} ; y={1}", x,y); //stampa x= 2, y=2
```

Analogamente per l'operatore di decremento:

```
int x=1;
int y=x--; //restituisce x e poi decrementa
Console.WriteLine("x={0} ; y={1}", x,y); //stampa x= 0, y=1

x=1;
y= --x; (decrementa x e poi restituisce il valore
Console.WriteLine("x={0} ; y={1}", x,y); //stampa x= 0, y=0
```

Controllo di overflow

A tempo di esecuzione può capitare che un'espressione su tipi integrali produca un valore al di fuori dell'intervallo consentito per il tipo stesso.

Come mostrato nel Capitolo 2, una variabile di tipo byte può assumere un valore massimo pari a 255, poiché formata da 8 bit.

Quindi se si tenta, per esempio, di eseguire il seguente blocco di istruzioni:

```
byte b1 = 200;
byte b2 = 150;
byte somma =(byte)(b1 + b2);
```

la variabile somma, che è di tipo byte, non può contenere il valore teorico risultante dalla somma, pari a 350, ma, a causa del cosiddetto *overflow* (straripamento), risulterà pari a 94.

NOTA	Nel paragrafo sulle conversioni esplicite, nel Capitolo 3, abbiamo già dato una spiegazione del perché il risultato di questa somma sia pari a 94.

In pratica una volta raggiunto il valore massimo di 255, la variabile ricomincia dal valore minimo del tipo utilizzato, che per il byte è 0.

Ogni tipo numerico fornisce un'apposita proprietà per ricavare il minimo e il massimo valore consentito:

```
byte min=Byte.MinValue;
byte max=Byte.MaxValue;
```

In teoria, provando ora a incrementare il valore max (che è, come detto, 255) anche solo di uno, il risultato non può essere 256, che non sarebbe più un valore valido del tipo byte, ma sarà pari a 0:

```
byte val=max++;
Console.WriteLine(val); //0
```

Al contrario, se al valore della variabile min (che sarà pari a 0 nel caso del byte) venisse sottratto il valore 1, otterremmo un valore finale di 255:

```
byte min = Byte.MinValue;
byte underflow = (byte)(min-1); //255
```

Questo fenomeno di overflow/underflow può quindi in generale portare a risultati inattesi ed errati di un programma.

Sarà necessario, da parte dello sviluppatore, prestare attenzione ai valori ottenuti nei calcoli oppure utilizzare due appositi operatori, mediante i quali il compilatore viene informato che

dovrà controllare le operazioni aritmetiche e le conversioni che agiscono sui tipi interi.

L'operatore checked permette di tenere sotto controllo tali operazioni e informare quindi il CLR che si è verificato un errore di tipo OverflowException, anziché consentire l'overflow in maniera silenziosa.

Per esempio, possiamo riscrivere lo stesso esempio precedente con l'operatore checked per verificare che l'operazione di incremento sia valida:

```
byte val= checked(max++);
```

In questo modo l'istruzione provocherà la suddetta OverflowException, a informarci che il calcolo ha superato l'intervallo di valori consentito. Il messaggio di errore restituito a tempo di esecuzione da tale eccezione sarà del tipo:

```
Arithmetic operation resulted in an overflow.
```

L'operatore checked si può utilizzare sia per un'espressione sia per un intero blocco di istruzioni.

Nel caso precedente esso agisce solo sull'espressione di incremento, mentre se invece volessimo che tutte le espressioni di un blocco vengano valutate in un contesto controllato (checked) scriveremmo:

```
checked
{
//espressioni
byte b=123+456;
}
```

Il controllo del contesto di esecuzione mediante gli operatori checked e unchecked agisce sulle operazioni di cast fra un tipo intero verso un altro tipo intero, oppure da float e double verso tipi interi e sui seguenti operatori applicati a operandi di tipi interi:

- gli operatori unari ++, -- e -;
- gli operatori binari +, -, *, /, quando entrambi gli operandi sono interi.

Se stiamo sviluppando un'applicazione in cui eventuali overflow siano sempre e comunque inaccettabili, per esempio un programma di calcolo, sarebbe come minimo noioso dover aggiungere l'operatore all'interno di espressioni o attorno a blocchi di codice per verificare ogni operazione pericolosa.

Quindi è possibile indicare che ogni operazione aritmetica debba essere eseguita in contesto checked, mediante un'apposita opzione /checked del compilatore csc.

```
C:/> csc /checked nomeprogramma.cs
```

Abilitando tale opzione, ogni volta che un overflow si verificherà nel programma, verrà scatenata un'eccezione OverflowException.

Naturalmente la stessa opportunità ci viene fornita dall'interfaccia grafica di Visual Studio.

Per abilitare tale opzione basta aprire la pagina delle proprietà del progetto e cliccare sul pulsante Advanced nella scheda Build. Nella finestra di dialogo che si aprirà sarà ora possibile selezionare la casella Check for arithmetic overflow/underflow (vedi Figura 4.1).

t

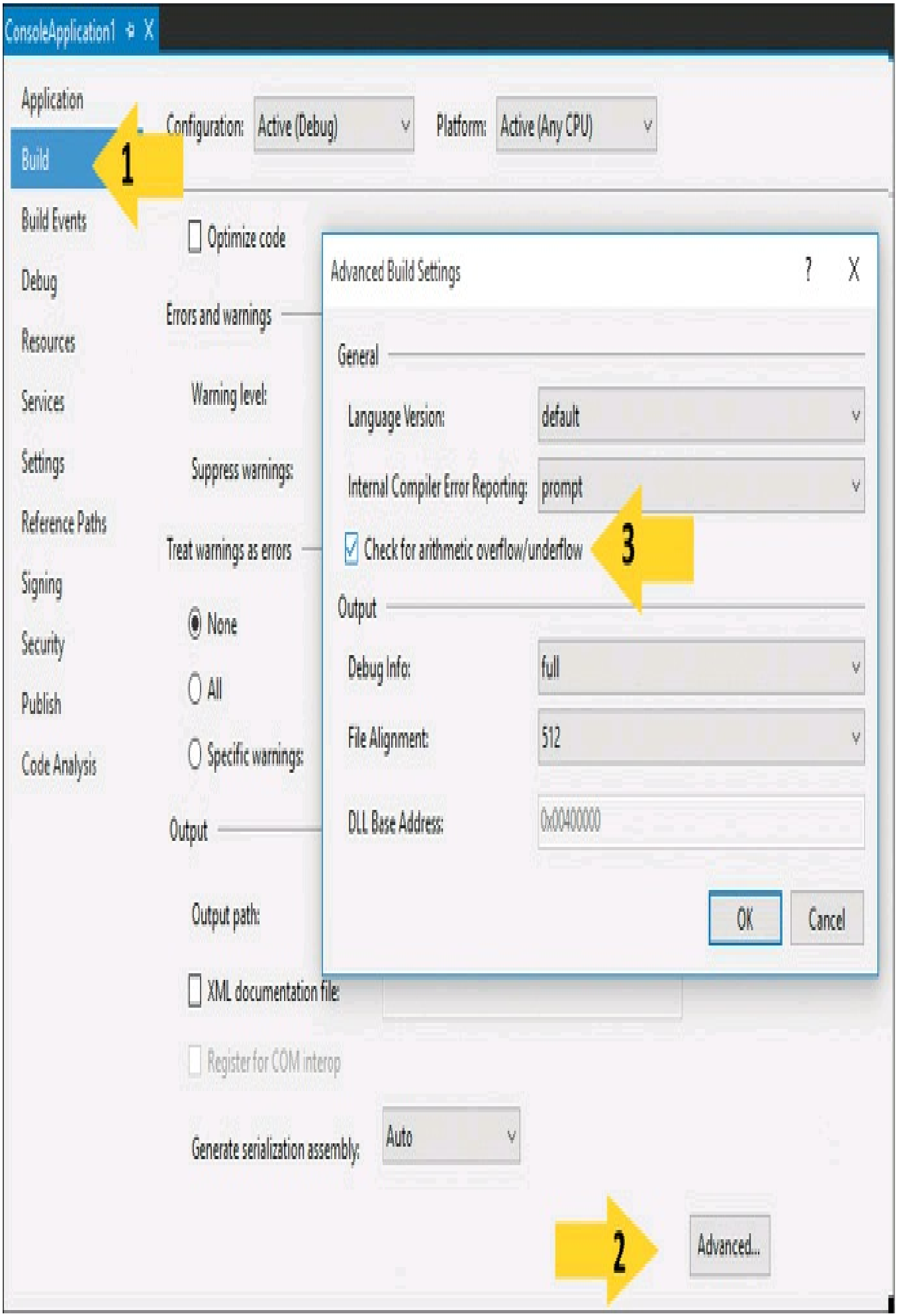

ConsoleApplication1
Application
Build
Build Events
Debug
Resources
Services
Settings
Reference Paths
Signing
Security
Publish
Code Analysis
Configuration: Active (Debug)
Platform: Active (Any CPU)
1
Optimize code
Errors and warnings
Warning level:
Suppress warnings:
Treat warnings as errors
None
All
Specific warnings:
Output
Output path:
XML documentation file:
Register for COM interop
Generate serialization assembly: Auto
2
Advanced...
Advanced Build Settings
General
Language Version: default
Internal Compiler Error Reporting: prompt
Check for arithmetic overflow/underflow
3
Output
Debug Info: full
File Alignment: 512
DLL Base Address: 0x00400000
OK
Cancel

Figura 4.1 – Abilitazione del controllo degli overflow in Visual Studio.

Supponiamo ora di aver abilitato a livello di compilatore, oppure nelle proprietà del progetto Visual Studio, il controllo di overflow. Come fare ad accettare la possibilità che si verifichi tale evento in qualche espressione o blocco particolare?

L'operatore unchecked funziona proprio come il corrispondente checked, disabilitando le eccezioni Overflow per particolari casi.

Assumendo che il seguente codice venga compilato con l'opzione /checked:

```
unchecked
{
byte b1 = 200;
byte b2 = 150;
byte somma =(byte)(b1 + b2);
}
```

il CLR ignorerà l'overflow che si verificherà all'interno del blocco unchecked.

Operatori di confronto

Gli *operatori di confronto* sono degli operatori binari che restituiscono un valore di tipo bool, risultato di un confronto fra i due operandi. Essi saranno di importanza fondamentale nei costrutti di flusso che vedremo nel prossimo capitolo.

Tabella 4.3 **- Tipi a virgola mobile predefiniti di C#.**

Operatore	Nome	Descrizione
<	minore di	restituisce true se il primo operando è minore del secondo
>	maggiore di	restituisce true se il primo operando è maggiore del secondo
<=	minore o uguale a	restituisce true se il primo operando è minore o uguale al secondo
>=	maggiore o uguale a	restituisce true se il primo operando è maggiore o uguale al secondo
==	uguale a	restituisce true se il primo operando è uguale al secondo
!=	diverso da	restituisce true se il primo operando è diverso dal secondo

```
int x=1;
int y=2;
bool b;
b= (x<y); //true
b = (x>y); //false
b = (x>=y); //false
b = (x<=y); //true
b = (x==y); //false
b = (x!=y); //true
```

È possibile confrontare tutti i tipi predefiniti di C#, a esclusione del tipo bool, per il quale valgono solo gli operatori == e !=:

```
bool f=false;
bool t=true;
Console.WriteLine(f==t); // false
Console.WriteLine(f!=t); //true
```

In generale, per operandi di tipo riferimento, gli operatori di confronto agiscono solo sui riferimenti.

Quindi se costruiamo due istanze diverse di una stessa classe, i riferimenti saranno diversi (perché in memoria heap si avranno due diversi oggetti):

```
Customer cliente1=new Customer("pippo");
Customer cliente2=new Customer("pippo");
Console.WriteLine(cliente1==cliente2); // false, due riferimenti differenti

cliente2=cliente1; //cliente2 e cliente1 referenziano ora lo stesso oggetto
Console.WriteLine(cliente1==cliente2); //true
```

Per le stringhe, anche se string è un tipo riferimento, il funzionamento degli operatori di uguaglianza e disuguaglianza è differente.

NOTA	Il tipo String implementa l'overload degli operatori == e !=, cioè ridefinisce il loro comportamento.

Il confronto mediante gli operatori == e != verifica, infatti, se le sequenze di caratteri contenuti nelle due stringhe confrontate sono uguali o diverse. In particolare due stringhe sono considerate uguali se:

- entrambe sono null;
- hanno uguale lunghezza e i caratteri che le compongono sono uguali per ogni posizione.

Per esempio:

```
string str1="hello";
string str2="ciao";
Console.WriteLine(str1==str2); //false
Console.WriteLine(str1=="hello"); // true
```

Gli operatori == e != possono anche essere utilizzati per confrontare un operando di un tipo nullable con il valore null:

```
int? x;
```

Si può quindi verificare se x ha un valore assegnato o meno, mediante una delle seguenti espressioni booleane:

```
x == null
null == x
x != null
null != x
```

Se non esiste nessun overload dei due operatori applicabili al tipo, il risultato delle espressioni viene calcolato utilizzando le proprietà bool HasValue, che x possiede in quanto tipo nullable e che indica se una variabile nullable possiede un valore.

In particolare le prime due espressioni equivalgono a verificare l'espressione !x.HasValue, mentre le altre due equivalgono a verificare l'espressione x.HasValue.

Nel Capitolo 9, dedicato ai tipi generici, vedremo più nel dettaglio i tipi nullable.

Operatori bit a bit

Tabella 4.4 **- Gli operatori bit a bit di C#.**

Operatore	Nome
&	AND
|	OR
^	XOR
~	NOT

C# supporta una serie di operatori che permettono di agire direttamente sui bit di un operando numerico, oppure di effettuare le classiche operazioni dell'algebra booleana su operandi di tipo bool. Essi sono detti operatori *bit a bit* (o *bitwise*) e sono elencati nella Tabella 4.4.

Gli operatori &, |, ^ sono binari, mentre l'operatore ~ è unario.

Nel caso di operandi di tipo booleano, è facile scrivere una serie di istruzioni che ci stampino una tabellina riassuntiva dell'algebra booleana. Basta combinare i valori di true e false con i tre operatori logici:

```
bool t = true;
bool f = false;

Console.WriteLine("operatore AND (&)");
Console.WriteLine("{0} & {1} = {2}", f, f, f & f);
Console.WriteLine("{0} & {1} = {2}", f, t, f & t);
Console.WriteLine("{0} & {1} = {2}", t, f, t & f);
Console.WriteLine("{0} & {1} = {2}", t, t, t & t);
Console.WriteLine();

Console.WriteLine("operatore OR (|)");
Console.WriteLine("{0} | {1} = {2}", f, f, f | f);
Console.WriteLine("{0} | {1} = {2}", f, t, f | t);
Console.WriteLine("{0} | {1} = {2}", t, f, t | f);
Console.WriteLine("{0} | {1} = {2}", t, t, t | t);
Console.WriteLine();

Console.WriteLine("operatore XOR (^)");
Console.WriteLine("{0} ^ {1} = {2}", f, f, f ^ f);
Console.WriteLine("{0} ^ {1} = {2}", f, t, f ^ t);
Console.WriteLine("{0} ^ {1} = {2}", t, f, t ^ f);
Console.WriteLine("{0} ^ {1} = {2}", t, t, t ^ t);
```

Compilando il codice precedente otterremo il risultato seguente:

```
operatore AND (&)
False & False = False
False & True = False
```

True & False = False
True & True = True

operatore OR (|)
False | False = False
False | True = True
True | False = True
True | True = True

operatore XOR (^)
False ^ False = False
False ^ True = True
True ^ False = True
True ^ True = False

Nel caso di operandi integrali, invece, gli operatori eseguono le operazioni logiche direttamente sui bit corrispondenti dei due operandi. L'operatore &, quindi, eseguirà un'operazione di AND fra i bit dei due operandi, cioè per ogni coppia di bit esso restituirà un bit di valore 1 se entrambi i bit sono pari a 1. Per esempio, se effettuiamo un'operazione di & fra due operandi di tipo byte, ecco cosa si otterrebbe:

```
byte x = 5; // 0000 0101
byte y = 9; // 0000 1001
byte z = (byte)(x & y); // 0000 0001
Console.WriteLine(z); // =1
```

L'unica coppia di bit pari a 1 è quella in prima posizione (da destra a sinistra), quindi il risultato dell'intera operazione di & sarà pari a 00000001, che corrisponde al valore 1 in decimale.

NOTA	Si noti come il risultato dell'operazione & sia stato convertito esplicitamente in tipo byte, perché l'operatore &, secondo le regole già viste di promozione numerica, esegue l'operazione convertendo i due byte in due int.

In maniera analoga l'operatore | esegue l'operazione di OR logico degli operandi, quindi restituisce 1 per ogni coppia di bit in cui c'è almeno un bit pari a 1. Nel nostro caso:

```
z=(byte)(x|y); // 0000 1101
Console.WriteLine(z); // = 13
```

Il terzo operatore logico ^ esegue l'operazione di OR esclusivo, che restituisce un bit 1 per ogni coppia in cui c'è un solo bit pari a 1. Quindi:

```
z=(byte)(x|y); // 0000 1100
Console.WriteLine(z); // = 12
```

L'unico operatore bit a bit unario è l'operatore ~ (tilde) di NOT, che inverte i bit dell'operando a cui viene applicato (tale operazione è detta anche complemento a 1). Quindi, se applichiamo l'operatore NOT alla variabile x dell'esempio precedente, otterremo:

```
byte x=5; // 0000 0101
byte notx=~x; // 1111 1010
Console.WriteLine(notx); // = 250
```

Anche nel caso dell'operatore NOT, il valore risultante deve essere convertito esplicitamente verso il tipo byte perché l'operatore converte implicitamente il suo operando, e quindi il risultato, al tipo int.

Operatori di shift

Gli *operatori di shift* eseguono delle operazioni di scorrimento dei bit di un operando integrale, spostandoli a destra o sinistra.

Tabella 4.5 **- Gli operatori di shift di C#.**

Operatore	Nome
>>	scorrimento a destra
<<	scorrimento a sinistra

L'operatore << scorre i bit a sinistra del numero di posizioni indicato mediante l'operando destro, inserendo tanti bit uguali a 0 sulla destra:

```
byte i =1; //0000 0001
byte shift= i<<3; //0000 1000
Console.WriteLine(shift);//=8
```

Al contrario, l'operatore >> scorre i bit verso destra, inserendo a sinistra dei bit pari a 0 oppure a 1 con la regola di estensione del segno: se il valore dell'operando da scorrere è positivo vengono inseriti dei bit 0, se è di valore negativo vengono inseriti dei bit 1:

```
int x = 15; // 0000 0000 0000 0000 0000 0000 0000 1111
int y = x>>2; // 0000 0000 0000 0000 0000 0000 0000 0011
Console.WriteLine(y); // =3
X = -4; // 1111 1111 1111 1111 1111 1111 1111 1100
y = x>>2; // 1111 1111 1111 1111 1111 1111 1111 1111
Console.WriteLine(x); //=-1
```

Sebbene agire a livello di bit, come permettono gli operatori di shift e quelli logici visti nel paragrafo precedente, non sia molto comune a meno di applicazioni specifiche, tali operatori possono tornare molto utili.

L'operazione di scorrimento, data la natura della numerazione binaria, corrisponde a una moltiplicazione o divisione per due per ogni posizione, rispettivamente nel caso di scorrimento a sinistra o a destra.

Proviamo infatti a rivedere in dettaglio gli esempi precedenti:

```
int i=1;
int shift= i<<3; //=8
```

Lo scorrimento verso sinistra di tre posizioni ha avuto l'effetto di moltiplicare tre volte per due l'operando:

```
i*2*2*2=1*2*2*2=1*8=8
```

Vista la questione in termini matematici, l'operando è stato moltiplicato per una potenza del due con esponente pari al numero di posizioni spostate: $1*2^3$.

Al contrario, se si scorre a destra, l'operando viene diviso per la potenza di due con esponente pari al numero di posizioni:

```
int i=128; //1000 0000
int shift= i >> 6; // //0000 0010
Console.WriteLine(shift); //=2
```

In questo caso il risultato finale è 2, come se il numero 128 fosse stato diviso per 2^6 , che è pari a 64.

Quindi, se fosse necessario moltiplicare o dividere degli interi per delle potenze del due, un modo molto efficiente di farlo è lo shift a destra o sinistra.

Una possibile applicazione pratica di quanto appena detto consente di realizzare un convertitore di unità di misura fra kilobyte, megabyte, gigabyte e così via. Ognuno di essi è infatti dato dall'unità precedente moltiplicata per 1024, per esempio 1 megabyte è pari a 1024 kbyte, mentre 1 gigabyte a sua volta è pari a 1024 megabyte. Quindi, essendo 1024 pari a 2^{10}, per convertire un valore nell'unità di misura immediatamente superiore basterà dividerlo per 1024, quindi scorrere i suoi bit di 10 posizioni mediante l'operatore di shift a destra; al contrario, per ottenere il valore nell'unità di misura immediatamente inferiore basterà moltiplicarlo per 1024, quindi utilizzare lo shift a sinistra di 10 posizioni:

```
int gb=1;
int mb= gb << 10;
Console.WriteLine("{0} gb = {1} mb", gb,mb);
int kb= mb<< 10;
Console.WriteLine("{0} gb = {1} kb", gb,kb);
int bytes=kb<<10;
Console.WriteLine("{0} gb = {1} bytes", gb,bytes);

bytes=1048576;
int mb=bytes>>10;
Console.WriteLine("{0} bytes={1}", bytes, mb);
int gb= mb>>10;
Console.WriteLine("{0} bytes={1}", bytes, gb);
```

Operatori di assegnazione

Gli *operatori di assegnazione* hanno lo scopo di assegnare un valore o in generale il risultato di un'espressione a una variabile (o come vedremo più avanti a una proprietà o a un indicizzatore). In particolare, la variabile destinazione dell'assegnazione è posta alla sinistra dell'operatore, mentre l'espressione da valutare viene posta alla sua destra.

Abbiamo più volte utilizzato l'operatore di assegnazione semplice `=`, soprattutto per inizializzare una variabile:

```
int x=5: //assegnamento di 5 alla variabile x
double d= x/2.0D; //assegna il risultato della divisione alla variabile d
```

L'operatore può anche essere utilizzato in cascata, per eseguire più di un'assegnazione in una singola istruzione:

```
a = b = c;
```

Poiché l'operatore di assegnazione è associativo a destra, l'espressione precedente è equivalente a:

```
a = (b = c);
```

quindi prima viene assegnato il valore di `c` a `b` e quindi `b` viene assegnato ad `a`. In questo esempio sia `a` sia `b` avranno lo stesso valore di `c`.

Assegnazione composta

Altri operatori di assegnazione, detti di *assegnazione composta*, permettono di assegnare a una variabile un nuovo valore basato sul suo valore attuale, al quale viene applicato un ulteriore operatore aritmetico.

Gli operatori di assegnazione composta funzionano in generale secondo la seguente sintassi:

x **op=** y;

il cui effetto sarà equivalente a:

x = x **op** y;

Vale a dire che l'operando `op` viene applicato agli operandi `x` e `y` come un normale operatore binario, e il risultato dell'espressione viene poi assegnato a `x`. Per esempio:

```
int x=1;
x+=2; //equivalente a x=x+2 quindi il risultato finale è x =3
```

La Tabella 4.6 mostra gli operatori di assegnazione composta disponibili in C#.

Tabella 4.6 **- Gli operatori di assegnazione composta.**

+=	addizione

-=	sottrazione
*=	moltiplicazione
/=	divisione
%=	modulo
&=	AND bitwise
|=	OR bitwise
^=	XOR bitwise
<<=	shift a sinistra
>>=	shift a destra

Operatori logici condizionali

Gli *operatori logici condizionali* sono utilizzati per confrontare o negare i valori logici dei propri operandi booleani e restituire il corrispondente risultato. Essi sono detti anche operatori di corto circuito e costituiscono la versione logica degli operatori bit a bit &, | e !.

C# possiede gli operatori logici condizionali elencati nella Tabella 4.7.

Tabella 4.7 **- Gli operatori logici condizionali di C#.**

Operatore	Descrizione
&&	AND logico
‖	OR logico
!	NOT logico

Gli operatori logici condizionali si applicano solo a operandi di tipo bool, siano essi delle variabili oppure espressioni più complesse, il cui valore finale è naturalmente un bool. La sintassi di utilizzo di tali operatori è la seguente:

```
x && y; // true se x==true E y==true
x || y; //true se x==true OPPURE y ==true
!x; // restituisce il valore negato di x, cioè true se x==false, false se x==true
```

L'operatore logico ! (si chiama e si legge *NOT*) è unario, restituendo il valore booleano opposto dell'espressione a cui viene applicato.

L'operazione x && y corrisponde a x & y, ma y viene valutata solo se x è true in quanto, se x fosse false, l'intera operazione AND sarebbe false, qualunque sia il valore di y.

L'operazione x || y corrisponde a x | y, ma y è valutata solo se x è false in quanto, se x fosse true, l'intera operazione di OR restituirebbe in ogni caso true, poiché per l'OR è sufficiente che un solo operando sia true.

Per i due motivi precedenti gli operatori logici condizionali sono detti operatori di corto circuito: la valutazione dell'operando destro può essere saltata se il risultato complessivo dell'operazione condizionale si può già desumere dalla valutazione del primo operando.

Da ciò deriva anche il fatto che non esiste un operando XOR logico condizionale, in quanto per definizione stessa di XOR, bisogna valutare necessariamente entrambi gli operandi.

Operatore ternario

L'unico *operatore ternario* di C# è l'operatore ?:, detto anche *operatore condizionale*, che permette di restituire una fra due espressioni, a seconda che una condizione risulti vera o falsa.

NOTA	L'operatore ternario consente di abbreviare un costrutto if/else (vedere il prossimo capitolo) utilizzando una sola istruzione.

La sintassi di utilizzo dell'operatore ternario è la seguente:

```
op1 ? op2 : op3
```

In base al valore assunto dal primo operando, quello a sinistra del ?, l'operatore ternario restituirà uno degli altri due operandi. L'operando op1 deve quindi essere un'espressione che ritorna un valore bool. Se il suo valore è true, l'intera espressione sarà pari al valore di op2; se invece op1 è false, l'intera espressione restituisce il valore op3. Esso è un modo di abbreviare un costrutto di questo tipo:

```
if(op1)
return op2;
else
return op3;
```

In parole povere, se op1 è verificata restituisce il valore di op2, altrimenti restituisce il valore di op3.

Il seguente esempio mostra come valutare il massimo fra due numeri utilizzando una sola istruzione che si avvale dell'operatore ternario:

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

Se x>y, la variabile max conterrà il valore della variabile x, altrimenti quello di y.

NOTA	Molti sviluppatori preferiscono evitare l'utilizzo dell'operatore ternario, così come mi è capitato di vedere in qualche azienda dei documenti di stile che proibiscono il suo utilizzo, a causa della poca chiarezza del codice che ne deriva. A mio parere invece esso è particolarmente elegante ed espressivo. Se proprio non avete voglia di usarlo, vi consiglio comunque di cercare di comprenderlo, nell'eventualità di incontrarlo all'interno di codice scritto da altri.

Controllo di riferimenti nulli

Nella programmazione a oggetti una delle operazioni di controllo più frequenti e forse per questo più noiose e ripetitive consiste nel verificare che un oggetto non sia nullo. C# fornisce degli appositi operatori anche per semplificare questi controlli ed evitare errori inaspettati.

Operatore null coalescing

L'operatore *null coalescing* ?? serve a definire un valore predefinito per un oggetto di un tipo nullable o di un tipo riferimento. Esso restituisce un oggetto o valore indicato come primo operando a sinistra dell'operatore ?? nel caso in cui esso sia diverso da null, altrimenti restituisce il secondo operando, quello posto alla destra dell'operatore:

```
object nullobj=null;
object obj=123;

var v1=nullobj ?? 2; // restituisce 2 perché nullobj è uguale a null
var v2=obj ?? 2; //restituisce 123, perché obj è diverso da null;
```

Anche l'operatore di null coalescing può essere utilizzato al posto di un costrutto if/else o di un operatore ternario. Per esempio, supponiamo di avere le seguenti istruzioni che, a seconda del valore di una stringa, assegnano a una variabile name il suo valore oppure un valore predefinito:

```
string name;
…
if(str!=null)
name=str;
else name="senza nome";
```

oppure, utilizzando l'operatore condizionale ternario:

```
string name= (str!=null) ? str: "senza nome";
```

Utilizzando l'operatore di null coalescing ??, possiamo semplificare ancora di più con la seguente istruzione:

```
string name= str ?? "senza nome";
```

Sebbene non sembri molto più semplice di un operatore ternario, l'operatore ?? rende più agevole la scrittura di espressioni lambda (che affronteremo in un apposito capitolo) oppure di istruzioni composte che prevedono la verifica in cascata di più oggetti o espressioni. Per esempio la seguente istruzione:

```
var result= expr1 ?? expr2 ?? valPredefinito;
```

consente di assegnare alla variabile result il valore di expr1, se esso è diverso da null, altrimenti di expr2 se questo è diverso da null, o infine, se expr1 e expr2 sono entrambe null, assegna un valore

valPredefinito.

Operatore null conditional

L'operatore *null conditional* è stato introdotto in C# 6. Esso permette di controllare se un oggetto è null, evitando in questo caso di interagire con esso e riducendo la quantità di codice ripetitivo da scrivere.

L'operatore null conditional è rappresentato con i caratteri ?..

NOTA	L'operatore null conditional è chiamato anche operatore *Elvis*, per la somiglianza del simbolo ? con il ciuffo di Elvis Presley.

Si supponga, per esempio, di voler ottenere la lunghezza di una stringa, mediante la proprietà Length, ma tenendo presente che la stringa può anche essere null:

```
int len=str.Length;
```

Se str non è null, nessun problema. La proprietà Length restituirà la sua lunghezza. Ma se str fosse null si verificherebbe un'eccezione NullReferenceException. Facendo precedere all'operatore . quello null conditional ?., si può scrivere:

```
int? len=str?.Length;
```

L'operatore null conditional fa in modo che solo se str è diverso da null, venga letta la proprietà Length. Altrimenti viene restituito direttamente il valore null (quindi il tipo della variabile len deve essere un nullable).

La sintassi precedente è equivalente a quella dell'operatore ternario:

```
int? len=str!=null ? str.Length: (int?)null;
```

L'operatore null conditional è utilizzabile anche con membri indicizzatori o con array, per esempio supponiamo di voler ricavare il primo carattere della stringa str: se str è null, sappiamo che l'operatore restituirà null, in caso contrario potrà accedere al carattere a un dato indice:

```
char? primo=str?[0];
```

Più avanti nel libro si vedranno altre modalità d'uso dell'operatore, in particolare utilizzandolo con oggetti più complessi.

Operatore **nameof**

L'operatore nameof è stato introdotto in C# 6 per ottenere in maniera semplice il nome di un qualsiasi elemento di codice, per esempio variabili, tipi, membri di un tipo e così via.

Per esempio:

```
void Main()
{
Console.WriteLine($"esecuzione di {nameof(Main)}");
string str="hello";
string name=nameof(str);
}
```

Nel primo caso viene usato per stampare una sorta di log del metodo in esecuzione. Il secondo ricava il nome di una variabile.

L'utilizzo di nameof evita di dover scrivere manualmente tali nomi, con il rischio di commettere errori di battitura, di doverli ricontrollare e correggere manualmente. Inoltre in strumenti come Visual Studio, rinominando un membro, possono essere aggiornati automaticamente anche gli operandi di nameof, cosa che non sarebbe possibile utilizzando direttamente delle stringhe.

Nel caso di un membro di un tipo, esso restituisce solo il nome del membro stesso, senza riferimenti al suo contenitore. Per esempio, data la seguente classe:

```
class MiaClasse
{
public void Metodo1(int a) {}
}
```

Se obj è un'istanza di MiaClasse, con la sintassi seguente si può ottenere il nome del metodo Method:

```
Console.WriteLine(nameof(obj.Metodo1)); //stampa Metodo1
```

Il nome di un membro di un tipo si può ottenere anche utilizzando il nome del tipo, per esempio:

```
nameof(MiaClasse.Metodo1) //restituisce Metodo1
nameof(Console.Title.Length)//restituisce Length
```

L'operatore tornerà utile in diversi ambiti, che mostreremo al momento opportuno nel seguito del libro.

Operatori di tipo

Finora gli operatori visti hanno consentito di valutare e agire sul valore degli operandi. Gli *operatori di tipo*, invece, permettono di operare sul tipo delle espressioni a cui essi vengono applicati. Nel Capitolo 3 abbiamo già visto l'operatore typeof, che permette di ottenere un oggetto System.Type a tempo di compilazione e senza necessità di avere un'istanza del tipo stesso.

Operatore di cast

Nel capitolo precedente abbiamo già affrontato l'argomento delle conversioni esplicite fra tipi, introducendo l'operatore di *cast*. Esso è un operatore che indica il tipo verso cui si vuol convertire un'espressione. L'operazione di cast viene eseguita inserendo il tipo di destinazione fra due parentesi tonde, immediatamente prima dell'espressione da convertire. Per esempio:

```
int i=123;
short sh=(short)i;
```

Nel caso precedente il valore int viene convertito in short.

Naturalmente non sempre il cast è possibile e, in tal caso, si potrebbe avere un errore a tempo di compilazione oppure un'eccezione a runtime. Per esempio, il seguente codice:

```
String str = "stringa";
int i = (int)str;
```

non può essere compilato perché il tipo string non può essere convertito in int neanche in modo esplicito.

L'operatore di cast può anche essere utilizzato per le conversioni fra tipi riferimento oppure per l'operazione di unboxing, che permette di ricavare il valore contenuto in un object:

```
int i=1;
object obj=i;
int j=(int)obj; //obj viene riconvertito in int
```

L'operatore is

All'interno di un programma può capitare abbastanza frequentemente di dover verificare se un tipo riferimento è compatibile con un altro, in particolare quando si utilizzerà l'ereditarietà per creare gerarchie di classi.

L'operatore is permette di verificare se il tipo di un oggetto è compatibile con un altro, restituendo il valore booleano true in caso positivo:

```
string obj="";
```

```
bool b= obj is string;
```

Nel caso precedente l'operatore is restituisce true perché l'oggetto obj è proprio del tipo desiderato, cioè string.

Ma l'operatore is verifica non solo che il tipo sia proprio quello indicato, ma anche se esso è compatibile per mezzo di una conversione oppure per mezzo di boxing/unboxing.

Supponiamo, per esempio, di avere una classe Cliente derivata da una classe Persona. Come vedremo nel Capitolo 6, dedicato alla programmazione a oggetti, in questo caso la classe Cliente è di un tipo compatibile con la classe Persona, cioè in parole povere un Cliente è anche una Persona. L'operatore is può quindi essere utilizzato come segue:

```
Cliente c=new Cliente();
Console.WriteLine(c is Persona); //stampa True
```

Nel seguente esempio, invece, l'operatore is permette di verificare se il valore incapsulato nell'oggetto mediante boxing è di tipo int:

```
object obj = 1;
if (obj is int)
Console.WriteLine("j is int");
```

Nell'esempio ancora una volta l'operatore restituisce true.

L'operatore as

Nel paragrafo precedente, per mezzo dell'operatore is, abbiamo visto come verificare se un determinato tipo riferimento è compatibile con un altro, in genere perché derivato da esso.

Il passo successivo, in caso positivo, consiste nel convertire il tipo in quello compatibile.

La prima possibilità è di eseguire un cast, per mezzo dell'operatore ():

```
if(obj is Tipo)
Tipo t=(Tipo)obj;
```

Se però l'oggetto obj non è di un tipo compatibile con il tipo indicato fra parentesi, nell'esempio Tipo, verrà sollevata un'eccezione InvalidCastException.

L'operatore as rende la conversione più semplice, con la particolarità che in caso di conversione non possibile non si verifica alcuna eccezione, ma restituisce un riferimento null. Si utilizza così:

```
Tipo t=obj as Tipo;
```

L'operatore restituisce un oggetto del tipo indicato oppure null, quindi, come nell'esempio, viene quasi sempre utilizzato in un'espressione di assegnazione. Il risultato dell'istruzione precedente è equivalente a:

```
Tipo t;
if(obj is Tipo)
{
t=(Tipo)obj;
}
else t=null;
```

Come l'operatore `is`, anche `as` può essere utilizzato solo fra tipi riferimento oppure per operazioni di boxing e unboxing.

Domande di riepilogo

1) Date due variabili int a=5 e int b=2, il risultato dell'espressione int c=a / b è:

a. c = 2;
b. c = 2.5;
c. c = 3;
d. errore a runtime

2) Date due variabili int a=5 e int b=2, il risultato dell'espressione int c=a & b è:

a. c = 0
b. c = 2
c. c = 5
d. errore a runtime

3) Data la variabile int a=5, il risultato dell'espressione a*=2 è:

a. a = 2
b. a = 7
c. a = 10
d. errore a runtime

4) Date due variabili bool a=true e bool b=false, il risultato dell'espressione bool c = !(a && b) è:

a. c = false
b. c = true
c. errore di compilazione
d. errore a runtime

5) Data la variabile byte max=Byte.MaxValue; quanto vale b dopo l'istruzione seguente:

```
checked
{
byte b = (byte)(max+1);
}
```

a. 256
b. 1
c. si verifica un'eccezione
d. 255

6) Se A è un metodo che restituisce true e B restituisce false, valutando la condizione if(A() && B()) quale di queste affermazioni è vera?

a. Viene eseguito solo A
b. Viene eseguito solo B
c. Non viene eseguito nessuno dei due
d. Vengono eseguiti entrambi

7) Data la variabile string str="abc", che valore assume la variabile len dopo l'istruzione int? len=str?.Length ?? 0?

a. null
b. 0
c. 3
d. Si verifica un'eccezione

8) Quale fra questi operatori è stato introdotto in C# 6?

a. nameof
b. typeof
c. sizeof
d. nullof

Controllo di flusso

Il flusso di esecuzione di un programma C# può essere controllato, al verificarsi di determinate condizioni, mediante istruzioni di selezione, iterazione e salto, che permettono di impostare condizioni e regole di esecuzione.

Ogni programma richiede delle funzionalità per controllarne il flusso di esecuzione, in base a determinate condizioni, sia dettate dal funzionamento interno del programma, sia derivanti da scelte dell'utente che interagisce con esso.

Si pensi per esempio a un'applicazione che permetta di inserire le proprie credenziali per accedere a un'area riservata. In base al successo o meno dell'autenticazione, l'utente verrà indirizzato a un'apposita sezione dell'applicazione, oppure riceverà un messaggio di errore.

È dunque necessario poter controllare il flusso di esecuzione dell'applicazione, in maniera da controllare innanzitutto il verificarsi o meno di una condizione booleana, cioè che le credenziali di accesso siano state inserite correttamente, e quindi eseguire in base a essa una serie di istruzioni oppure un'altra, che equivale a portare l'utente in una parte dell'applicazione oppure in un'altra.

Si noti che non abbiamo parlato di alcuna tipologia specifica di applicazione. Essa potrebbe essere costituita da un classico programma a finestra, che all'avvio visualizza una schermata di inserimento di username e password, oppure da un'applicazione web ASP.NET con una sezione riservata del sito protetta da una pagina di login o ancora da un'applicazione per il proprio telefonino nel quale inserire un codice ricevuto via SMS per autenticarsi correttamente.

In tutti i casi, il linguaggio di programmazione dovrà permettere la scrittura di espressioni condizionali, la verifica per mezzo di istruzioni o metodi, la selezione o il salto verso un'apposita sezione del codice.

Espressioni condizionali

Nel precedente capitolo abbiamo già avuto modo di osservare il funzionamento di operatori che permettono la verifica di certe espressioni booleane, e in particolare abbiamo trattato i cosiddetti operatori di confronto, quelli logici e l'operatore condizionale ternario, oltre a quelli di null coalescing e null conditional.

Le *espressioni condizionali* vengono scritte utilizzando tali operatori per ottenere un valore booleano, in base al quale il programma potrà poi decidere se e quali ulteriori azioni intraprendere. Per esempio, una semplice espressione per verificare se un numero corrisponde a uno specifico valore utilizzerà l'operatore di uguaglianza ==:

```
bool b = (x == 0); //
```

La condizione da valutare può diventare più complessa se fosse necessario verificare più espressioni combinandole in qualche maniera. Per esempio, se si vuole verificare che il numero x sia compreso in un certo intervallo numerico, dovremo utilizzare due espressioni con gli operatori di confronto (maggiore e minore) e un ulteriore operatore logico && per combinarle:

```
bool b=(x>=0 && x<=10); //b è true se 0 <= x <= 10
```

Se nelle espressioni condizionali è presente una variabile booleana, l'espressione stessa può essere semplificata, in particolare nei casi in cui l'espressione si basi sugli operatori di uguaglianza == o disuguaglianza !=. Infatti, dato che l'espressione deve essere valutata come valore bool, e una variabile bool lo è per definizione, le seguenti espressioni in cui b è una variabile di tipo bool:

```
b==true
b==false
```

possono essere scritte rispettivamente come:

```
b
!b
```

Costrutti di selezione

Il linguaggio C# consente agli sviluppatori di far prendere decisioni a un programma per mezzo di alcuni *costrutti di selezione*.

Il termine selezione deriva dalla possibilità di selezionare quale parte del programma eseguire a seguito del verificarsi di una determinata condizione booleana.

NOTA	In C# si avrà sempre a che fare con espressioni di controllo o condizioni booleane: esso infatti richiede che l'espressione da valutare restituisca sempre uno dei valori del tipo bool, cioè true o false. Non è possibile interpretare un numero in un'espressione condizionale, a differenza di C/C++ (per esempio, zero per indicare false e diverso da zero per il valore true).

L'istruzione if

Il costrutto di selezione più semplice è l'istruzione **if**, che consente di eseguire o meno un blocco di codice al verificarsi o meno di una condizione. La sua sintassi generale è la seguente:

```
if(expr)
blocco_istruzioni
```

Se expr assume valore true verrà eseguito il blocco di istruzioni successivo, che può essere una singola istruzione oppure un blocco racchiuso fra parentesi graffe. Altrimenti, se expr è false, il blocco viene saltato.

NOTA	Personalmente con le istruzioni if utilizzo un blocco sempre delimitato da parentesi graffe, anche se esso è costituito da una singola istruzione, sia per mantenere uno stile uniforme, sia perché sarà più semplice e veloce aggiungere o togliere altre istruzioni dal blocco: non sarà necessario aggiungere le parentesi graffe e sistemare l'indentazione.

Ecco un paio di esempi di utilizzo dell'istruzione if:

```
int x=0;

//singola istruzione
if(x==0)
Console.WriteLine("x è nullo");

//blocco di istruzioni
if(x>0)
{
Console.WriteLine("x è maggiore di zero");
}
```

Ripetiamo che l'espressione fra parentesi deve necessariamente essere valutata come bool, altrimenti si verificherebbe un errore di compilazione, come nel caso seguente:

```
if(x=0) //errore, non è un'espressione booleana
{
…
}
```

L'esempio precedente restituisce un errore di compilazione del tipo "Impossibile convertire int in bool". Infatti l'istruzione x=0 è un'espressione di assegnazione, quindi x è un valore di tipo **int**.

Poiché il compilatore si aspetta come condizione di controllo dell'if un valore bool, esso tenta di convertire il tipo int di x in booleano, che non è implicitamente possibile.

NOTA	L'istruzione if(x=0) è invece utilizzabile in C/C++ in quanto è possibile utilizzare come condizione dell'if un valore numerico. Ciò potrebbe portare a bug del codice difficilmente individuabili, perché l'assegnazione con il singolo = potrebbe essere stata una semplice dimenticanza, mentre l'intenzione era quella di eseguire un confronto di uguaglianza.

Come accennato nel precedente paragrafo, se un'espressione condizionale prevede il test di una variabile booleana, possiamo utilizzare direttamente il nome della variabile per testare se essa è true o false. Partiamo dall'esempio seguente:

```
bool b;
if(b==true)
{
}
if(b!=true) // oppure if(b==false)
{}
```

Essendo b una variabile bool, il cui valore quindi sarà sempre uno fra true o false, le due istruzioni if precedenti possono essere semplicemente scritte come:

```
if(b) //se b è true…
{
}
if(!b)//se b è false
{
}
```

Il codice è così molto più leggibile e semplice.

Operatori logici di corto circuito

Riprendiamo ora il concetto di *operatori logici di corto circuito*, visto nel precedente capitolo parlando degli operatori logici condizionali, per far notare, tramite l'uso di un'istruzione if, cosa implica il cortocircuitarsi delle espressioni booleane utilizzate come condizione dell'if stesso.

Per esempio, se si vuol eseguire un blocco di codice solo se tre interi sono tutti pari, si può implementare un semplice metodo che esegua questa verifica (non abbiamo ancora spiegato come implementare un metodo, ma non è nulla di complicato, quindi riportiamo il codice completo della classe Program). Prestate particolare attenzione alle due istruzioni if:

```
class Program
{
public static bool NumeroPari(int x)
{
Console.WriteLine("Verifico se {0} è pari", x);
return x % 2 == 0;
}

static void Main(string[] args)
{
int a = 2;
int b = 3;
int c = 4;

//operatore & bitwise, esegue tutti e tre i metodi
Console.WriteLine("verifica con &");
if (NumeroPari(a) & NumeroPari(b) & NumeroPari(c))
{
Console.WriteLine("Tutti pari");
}
else Console.WriteLine("Non sono tutti pari");
Console.WriteLine();

//operatore &&, non esegue la verifica di c, perché ha trovato b dispari
Console.WriteLine("verifica con &&");
if (NumeroPari(a) && NumeroPari(b) && NumeroPari(c))
{
Console.WriteLine("Tutti pari");
}
else Console.WriteLine("Non sono tutti pari");
}
}
```

Il metodo NumeroPari verifica se un numero è pari calcolando il modulo due del numero, cioè verificando se il resto della divisione per due è zero. Inoltre stampa una stringa in maniera da poter verificare quando viene eseguito e per quale numero in ingresso.

All'interno del metodo Main vengono dichiarate tre variabili a, b, c di tipo int e con la prima istruzione if viene usata una condizione che utilizza l'operatore & bit a bit per verificare che tutte e tre le variabili siano pari.

Nel secondo if viene invece utilizzato l'operatore &&. La differenza fra i due if è che nel secondo la verifica per la variabile c non viene mai eseguita perché, appena viene verificato che b è dispari, è inutile procedere con la verifica successiva, in quanto la condizione

assumerà in ogni caso valore false. Provando a compilare ed eseguire il codice si otterrà infatti un output simile al seguente:

```
verifica con &
Verifico se 2 è pari
Verifico se 3 è pari
Verifico se 4 è pari
Non sono tutti pari

verifica con &&
Verifico se 2 è pari
Verifico se 3 è pari
Non sono tutti pari
```

Nel secondo if il metodo NumeroPari è stato eseguito solo due volte. È quindi utile e importante conoscere la modalità di funzionamento degli operatori logici condizionali, per comprendere quando e se verranno valutate le condizioni booleane che li utilizzano.

L'istruzione if/else

Se l'istruzione if consente di implementare la scelta di eseguire o meno un singolo ramo di codice, mediante l'istruzione **if/else** è possibile implementare una decisione a due rami:

```
if(expr)
blocco1
else
blocco2
```

Se l'espressione condizionale expr assume valore true, verrà eseguito il blocco di istruzioni blocco1, mentre se essa è false, si eseguirà solo il blocco di istruzioni blocco2. Ecco qualche esempio che utilizza per il test della condizione il metodo IsNullOrEmpty della classe String (che restituisce true se una data stringa è null oppure è vuota):

```
string str=Console.ReadLine();
if(String.IsNullOrEmpty(str))
{
Console.WriteLine("inserisci una stringa");
}
else
{
Console.WriteLine("hai inserito {0}", str);
}
```

La stringa str viene letta dal prompt dei comandi mediante il già noto metodo ReadLine della classe Console; se l'utente non ha inserito nulla, verrà quindi valutata come true l'espressione di controllo dell'if, altrimenti verrà eseguito il ramo else.

Poiché entrambi i rami possono essere costituiti da una qualunque istruzione o blocco di istruzioni, è possibile annidare altre istruzioni if/else ed eseguire quindi dei test condizionali in cascata, per esempio in questo modo:

```
if(condizione1)
{
Blocco1
}
else if(condizione2)
{
Blocco2
}
else
{
Blocco3
}
```

L'esempio precedente verifica una condizione1; se essa è true viene eseguito il primo blocco di istruzioni, altrimenti viene valutata la condizione2 e così via. Se nessuna delle condizioni viene valutata come true, alla fine verrà eseguito il ramo di codice dell'ultimo else, cioè il Blocco3.

Da notare che a loro volta anche i blocchi di istruzioni inseriti all'interno dei rami di ogni if o else possono contenere altri costrutti if/else:

```
if(condizione1)
{
if(condizione3)
{
Blocco1
}
}
else if(condizione2)
{
Blocco2
}
```

In questo esempio, se la condizione1 è true viene poi verificato il valore della condizione3, e in caso positivo viene eseguito il Blocco1, mentre se la condizione3 è false viene saltato sia il Blocco1 sia il Blocco2 (che viene saltato perché si era già verificata la condizione1).

Il risultato finale dell'innestare i due if con condizione1 e condizione3 è equivalente all'utilizzo di un operatore && per combinarle in una sola espressione di controllo:

```
if(condizione1 && condizione3)
{
Blocco1
}
else if(condizione2)
{
Blocco2
}
```

Nel caso di if/else *innestati* è opportuno prestare attenzione alla composizione dei vari rami di codice: un'istruzione else si riferisce sempre all'if immediatamente precedente, a meno che non si utilizzino delle parentesi graffe per delimitare esplicitamente i vari blocchi.

Nel seguente codice, per esempio, l'istruzione else non si riferisce al primo if, anche se l'indentazione (fatta volutamente male) può far credere ciò:

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

L'esempio precedente è equivalente al seguente, in cui con l'utilizzo delle parentesi graffe si è esplicitato che l'else si riferisce all'if che lo precede:

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

Quindi solo se il valore di x è uguale a 0 viene eseguito il blocco interno e si valuterà il valore di y per entrare nell'if o nel ramo else.

L'istruzione switch

Nel precedente paragrafo abbiamo scritto un esempio di istruzioni if/else innestate per valutare diverse condizioni in cascata, una dopo l'altra. Un'altra possibilità per selezionare un blocco di codice da eseguire fra tanti, in base al valore assunto da un'espressione, è l'istruzione **switch**. Uno switch permette di valutare un'espressione di controllo e poi eseguire una specifica sezione di codice, a seconda del valore assunto dall'espressione precedente.

Ogni sezione viene indicata mediante un'etichetta **case** seguita dal valore da confrontare con l'espressione di controllo.

Il valore deve essere necessariamente una costante dello stesso tipo dell'espressione di controllo, o comunque convertibile implicitamente nello stesso tipo.

Il valore che segue l'etichetta case non può quindi essere un'ulteriore espressione da valutare a runtime. Inoltre tale costante deve essere univoca, cioè ogni sezione deve poter essere selezionata escludendo le altre.

È poi possibile indicare una sezione di codice predefinita, identificata dall'etichetta default, che verrà eseguita se il valore dell'espressione non corrisponde a nessun altro case. La sintassi generale del costrutto switch è la seguente:

```
switch(espressione)
{
```

```
case <costante1>:
}
```

La seguente istruzione switch verifica per esempio il valore assunto dalla variabile intera num ed esegue il blocco di codice contenuto nel corrispondente case:

```
int num;
…
switch(num)
{
case 1:
//blocco1
break;
case 2:
//blocco2
break;
default:
break;
}
```

Il valore assunto dall'espressione, che in questo caso è una semplice variabile, viene valutato, quindi viene eseguito il blocco della sezione corrispondente a tale valore. Nell'esempio, i case possibili sono quelli che verranno eseguiti per il valore 1 e per il valore 2 di num. Se num assumesse un valore diverso da 1 e 2, verrebbe eseguito il blocco all'interno della sezione default.

Quando si raggiunge l'istruzione break, il programma esce dallo switch e prosegue l'esecuzione con la prima istruzione che si trova dopo lo stesso switch.

Non è obbligatorio includere un'etichetta default in uno switch; in questo caso l'esecuzione continuerebbe saltando l'intero switch. Inoltre, si noti che, anche se l'etichetta default è indicata come ultima, anch'essa deve essere chiusa da un **break**.

NOTA	L'istruzione break è quella più utilizzata con lo switch, ma è possibile anche utilizzare un'istruzione return per uscire direttamente dal metodo attuale, oppure un'istruzione di salto goto (vedere più avanti).

Questo implica che non esiste nessuna regola o obbligo per l'ordine delle etichette: esse possono essere scritte in una sequenza qualunque, anche se nel caso di tipi numerici è consigliabile scriverle in ordine crescente e lasciare l'etichetta default sempre per ultima.

Il tipo dell'espressione di controllo dello switch è detto *tipo governante* e può essere uno fra i seguenti: sbyte, byte, ushort, int, uint, long, ulong, bool, char, string, un tipo enum, oppure un tipo nullable corrispondente a uno dei precedenti.

NOTA	È possibile utilizzare come tipo governante anche una classe personalizzata, ma solo se essa implementa l'overload di un operatore di conversione implicita (vedremo il significato nel Capitolo 6, nel Paragrafo "Overload degli operatori").

Nel caso in cui si utilizzi una stringa, il confronto con i valori delle costanti avviene in maniera case sensitive:

```
string name;
…
switch(name)
{
case "a":
//blocco a
break;
case "A":
//blocco A
break;
}
```

Nell'esempio precedente sono presenti due case, uno identificato dalla stringa "a", uno dalla stringa in maiuscolo "A", il che è perfettamente lecito.

Se il tipo governante è String oppure un tipo nullable, è possibile utilizzare null come costante di un'etichetta case:

```
switch(name)
{
case null:
Console.WriteLine("non hai inserito una stringa");
break;
…
}
```

L'utilizzo di tipi enumerativi con gli switch è abbastanza frequente e molto comodo perché ogni etichetta può essere facilmente identificabile e associabile ai membri dell'enumerazione stessa.

Ecco un esempio che utilizza l'enumerazione DayOfWeek, che contiene i giorni della settimana, basando l'espressione di controllo dello switch sul valore della data corrente:

```
DayOfWeek day = DateTime.Today.DayOfWeek;

switch(day)
{
case DayOfWeek.Monday:
…
break;
case DayOfWeek.Tuesday:
…
break;
case DayOfWeek.Wednesday:
```

```
….
break;
case DayOfWeek.Tuesday:
…
break;
case DayOfWeek.Wednesday:
…
break;
case DayOfWeek.Thursday:
…
break;
case DayOfWeek.Friday:
…
break;
}
```

L'obbligo di chiudere una sezione con l'istruzione break implica che in C#, a differenza, per esempio, di C/C++, non è possibile eseguire più sezioni una dopo l'altra. Questa regola è detta *no-fall through*; infatti se si provasse a compilare uno switch come il seguente:

```
switch(num)
{
case 1:
//blocco1
case 2:
//blocco2
break;
}
```

si otterrebbe un errore di compilazione con un messaggio del tipo:

```
Control cannot fall through from one case label ('case 1:') to another
```

L'unica eccezione possibile a questa regola è quando il blocco contenuto in una sezione case rimane vuoto; in tal modo è possibile raggruppare più sezioni che debbano eseguire lo stesso blocco di codice:

```
Console.WriteLine("digita un tasto");
char c = Console.ReadKey().KeyChar;
switch(c)
{
case 'a':
case 'e':
case 'i':
case 'o':
case 'u':
Console.WriteLine("hai digitato la vocale {0}", c);
break;
default:
Console.WriteLine("non hai inserito una vocale");
break;
}
```

Istruzioni di iterazione

Le *istruzioni di iterazione* permettono di eseguire ripetutamente un determinato blocco di istruzioni. Il numero di cicli da eseguire può essere predeterminato, per esempio per analizzare tutti gli elementi di un vettore o tutti i caratteri di una stringa, oppure può essere inizialmente indeterminato e fermarsi in base al verificarsi o meno di una determinata condizione.

C# fornisce quattro tipologie di costrutti iterativi, analizzati nei prossimi paragrafi.

L'istruzione while

Il ciclo **while** è un semplice costrutto iterativo che permette di testare una condizione booleana ed eseguire un blocco di istruzioni solo se e fino a quando essa rimane verificata (cioè uguale a true). La sintassi dell'istruzione while è la seguente:

```
while(espressione)
bloccoIstruzioni
```

Il blocco di istruzioni verrà eseguito fino a quando l'espressione di test avrà valore true, quindi potrebbe anche non essere mai eseguita se già al primo test essa venisse valutata come false:

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

L'esempio precedente stampa il valore della variabile i fino a quando il suo valore non sarà pari a 10. All'interno del blocco istruzioni, la variabile i viene incrementata a ogni ciclo, quindi dopo dieci volte, cioè alla decima iterazione, i avrà valore pari a 10 e l'espressione condizione del while diverrà false.

A questo punto il programma proseguirà la sua esecuzione con l'istruzione immediatamente successiva al costrutto while, cioè stampando la stringa "fine while".

Un'altra possibilità per terminare il ciclo è utilizzare un'istruzione di salto (vedere il Paragrafo "Istruzioni di salto" più avanti in questo capitolo), per esempio con un break:

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

```
break;
}
```

In questo esempio la condizione del ciclo while è sempre true, quindi si avrebbe un cosiddetto *ciclo infinito* se non ci fosse un'istruzione all'interno del ciclo stesso che lo interrompa. L'istruzione if verifica se la variabile i ha raggiunto il valore 10 e quando ciò sarà vero eseguirà un'istruzione break che interrompe le iterazioni.

```
while(espressione)
{
        .
        .
        .
        break;
        .
}
//fine while
```

Figura 5.1 - L'istruzione break in un ciclo while.

NOTA	Non sono rari i casi in cui, per qualche errore di programmazione, un ciclo while prosegue le sue iterazioni all'infinito, facendo sembrare il programma interamente bloccato. Assicuratevi quindi che vi sia sempre una condizione di uscita da un ciclo.

C# permette anche di utilizzare un'istruzione continue all'interno di un ciclo while. L'istruzione continue, come quella break, interrompe il ciclo attuale, però, a differenza della precedente, essa non termina l'intero ciclo ma ritorna all'inizio dello stesso, rivalutando l'espressione condizionale:

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

Nell'esempio precedente la prima istruzione all'interno del ciclo verifica se la variabile i è pari (mediante un'operazione modulo 2) e in tal caso la incrementa di 1 e salta il resto del blocco con l'istruzione continue.

In ogni iterazione in cui essa assume invece un valore dispari, verrà stampato il suo valore.

```
while (espressione)
{
    ;
    ;
    ;
    continue;
    ;
}
//fine while
```

Figura 5.2 - L'istruzione continue in un ciclo while.

Le istruzioni break e continue sono utilizzabili in tutti i costrutti iterativi di C# con lo stesso effetto.

L'istruzione do

Il ciclo while del precedente paragrafo esegue come prima operazione la verifica dell'espressione booleana, quindi può anche capitare che non venga mai eseguita alcuna iterazione perché l'espressione assume subito valore false.

Un'altra istruzione di ciclo simile al while, l'istruzione **do/while**, permette invece di eseguire almeno una volta il blocco di istruzioni e quindi verificare la condizione solo alla fine del ciclo. La sua sintassi è la seguente:

```
do
bloccoIstruzioni
while(espressione);
```

Si noti che a differenza del precedente while, l'istruzione do richiede un punto e virgola (;) dopo le parentesi che chiudono l'espressione condizionale:

```
char c;
do
{
Console.WriteLine("premi q per uscire");
c = Console.ReadKey().KeyChar;
}
while(c!='q');
```

Il blocco viene eseguito almeno una volta, quindi legge il tasto premuto dall'utente e, fino a quando esso non corrisponde al carattere q, continua a eseguire il ciclo.

L'istruzione for

Un'altra tipologia di istruzioni iterative di C# permette di definire una sequenza di espressioni di inizializzazione e quindi eseguire un blocco di istruzioni fino a quando una condizione resta vera, infine valutare un'altra sequenza di istruzioni prima di proseguire l'iterazione.

L'istruzione o ciclo **for** definisce quindi tre sezioni:

```
for(inizializzazione; condizione; iterazione)
bloccoIstruzioni
```

L'inizializzazione viene eseguita prima dell'esecuzione del ciclo e permette di definire un'espressione con cui inizializzare delle variabili di iterazione (per esempio assegnando il valore iniziale di un contatore).

La successiva espressione permette di valutare una condizione booleana, che se è true darà luogo alla successiva iterazione altrimenti terminerà il ciclo (per esempio valutando se il contatore ha raggiunto un determinato numero).

L'espressione di iterazione, invece, viene valutata al termine di ogni ciclo e di solito permette di aggiornare le variabili di iterazione (per esempio incrementando un contatore).

Il ciclo for viene generalmente utilizzato per eseguire un numero predeterminato di iterazioni, contandole per mezzo di un'apposita variabile. Per esempio, il seguente ciclo for stampa 10 volte la stringa "hello world":

```
for(int i=0; i<10; i++)
{
Console.WriteLine("hello world");
}
```

La sezione di inizializzazione dichiara una variabile intera i, assegnando il valore iniziale 0. A ogni ciclo viene verificato che i sia minore di 10 e, in caso affermativo, si esegue il blocco di istruzioni. A questo punto nella sezione di iterazione viene incrementato il valore della variabile contatore i.

La variabile i è locale al ciclo for stesso, quindi è utilizzabile al suo interno come se fosse una normale variabile:

```
for(int i=1; i<=10; i++)
{
Console.WriteLine("i^2={1}", i, i*i);
}
```

L'esempio precedente stampa tutti i quadrati dei numeri da 1 a 10 (si noti che la variabile i stavolta parte da 1 e che anche il valore 10 rispetta la condizione i<=10).

Il ciclo for viene spesso utilizzato per stampare o accedere a tutti gli elementi di un array o di una collezione.

Per ottenere il numero di elementi di un array monodimensionale è possibile, per esempio, leggere la proprietà Length, esposta dalla classe Array da cui deriva ogni array:

```
int[] array=new int[10];
for (int index = 0; index<array.Length; index++)
{
array[index] = index;
}
```

L'esempio precedente esegue un ciclo per inizializzare tutti gli elementi di un array di int, impostando per ognuno di essi il valore dell'indice stesso, quindi alla fine l'array conterrà i valori 0, 1, 2, e così via fino a 9.

Il ciclo for può contenere altri cicli for innestati. Tale pratica è utile per accedere per esempio a strutture di dati con due o più indici, come una matrice o array rettangolare:

```
int[,] matrice = new int[10, 10];
for (int riga = 0; riga < matrice.GetLength(0); riga++)
{
for (int colonna = 0; colonna < matrice.GetLength(1); colonna++)
{
matrice[riga, colonna] = riga * colonna;
Console.Write("{0,5}", matrice[riga, colonna]);
}
Console.WriteLine();
}
```

L'esempio precedente stampa una specie di tabellina delle moltiplicazioni, dallo 0 al 9, producendo un output simile al seguente:

```
0 0 0 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7 8 9
0 2 4 6 8 10 12 14 16 18
0 3 6 9 12 15 18 21 24 27
0 4 8 12 16 20 24 28 32 36
0 5 10 15 20 25 30 35 40 45
0 6 12 18 24 30 36 42 48 54
0 7 14 21 28 35 42 49 56 63
0 8 16 24 32 40 48 56 64 72
0 9 18 27 36 45 54 63 72 81
```

Si è detto all'inizio che le tre sezioni del ciclo for possono contenere più espressioni, infatti basta delimitarle con la virgola. Potremmo per esempio creare una cosiddetta matrice identità, cioè una tabella formata da numeri 0 e con dei numeri 1 solo sulla diagonale, utilizzando un singolo for, con le due variabili riga e colonna inizializzate e incrementate nel seguente modo:

```
matrice = new int[10, 10];
for (int riga = 0, colonna = 0; riga < 10 && colonna < 10; riga++, colonna++)
{
matrice[riga,colonna] = 1;
}
```

Sebbene la modalità di uso più frequente del ciclo for sia quella appena esposta, cioè con delle variabili contatore da inizializzare e incrementare, è possibile utilizzare e valutare condizioni diverse.

È inoltre possibile omettere una o più sezioni. Il seguente esempio replica lo stesso funzionamento di un ciclo while(true), in quanto non inizializza nessuna variabile e non testa alcuna condizione:

```
for(;;)
{
//istruzioni }
```

Vale a dire che tutte e tre le sezioni sono formate da istruzioni vuote. In questo caso, come nel ciclo while, sarà necessario utilizzare un'istruzione break per uscire dal ciclo infinito che ne deriva:

```
for(;;)
{
//istruzioni
if(condizione)
break;}
```

Analogamente a quanto visto per il ciclo while, l'istruzione continue permette di terminare l'esecuzione del blocco e passare alla successiva iterazione, naturalmente eseguendo prima la sequenza di istruzioni di iterazione.

L'istruzione foreach

C# possiede un'istruzione che permette di scorrere tutti gli elementi di una collezione enumerabile di oggetti o di un array, senza l'utilizzo di indici e condizioni booleane.

Tutte le collezioni in .NET, come vedremo nel capitolo dedicato, sono enumerabili; tecnicamente significa che esse implementano un'interfaccia IEnumerable. Se ancora non sapete cosa voglia dire interfaccia e che cosa implichi, non preoccupatevi; per il momento fidatevi se vi dico che, per esempio, sia gli array sia le stringhe sono tipi enumerabili.

L'istruzione **foreach** serve a iterare tutti gli elementi di un tipo enumerabile e utilizza questa sintassi:

```
foreach(Tipo identificatore in collezione)
bloccoIstruzioni
```

Il tipo e l'identificatore dichiarano una variabile locale di iterazione, che a ogni ciclo assume il valore del successivo elemento della collezione.

Il seguente esempio stampa tutti gli elementi di un array:

```
int[] array=new int[]{1,2,3,4,5,6,7,8,9,10};
foreach(int i in array)
{
Console.WriteLine("{0}", i);
}
```

Il tipo della variabile di iterazione può essere assegnato implicitamente per mezzo della parola chiave var.

Per esempio si può scrivere il precedente ciclo foreach così:

```
foreach(var i in array)
{
Console.WriteLine("{0}", i);
}
```

Sarà il compilatore a determinare che i è di tipo int, recuperandolo dal tipo degli elementi dell'array.

Come detto, anche una stringa rappresenta un oggetto enumerabile e gli elementi di questa collezione sono i caratteri che formano la stringa:

```
string str="hello world";
foreach(char c in str)
{
Console.WriteLine("{0}", c);
}
```

Un'importante osservazione da fare riguardo all'istruzione foreach è che all'interno del suo blocco di istruzioni non può essere modificata la variabile di iterazione locale, come se essa fosse di sola lettura. Il seguente esempio non compilerà correttamente:

```
foreach(int i in array)
{
i=0; //errore
}
```

La descrizione dell'errore di compilazione indicherebbe che non è possibile assegnare un valore a i perché è una variabile di iterazione del foreach.

```
Cannot assign to 'i' because it is a 'foreach iteration variable'
```

Anche il blocco dell'istruzione foreach può essere interrotto o saltato per mezzo delle istruzioni break o continue.

Istruzioni di salto

Le *istruzioni di salto* servono a trasferire l'esecuzione del programma a un altro punto del programma stesso, in maniera incondizionata. Abbiamo già visto e utilizzato le istruzioni break e continue, all'interno di cicli e switch.

L'istruzione break

L'istruzione **break** serve a uscire dal blocco di codice dei costrutti switch, while, do, for e foreach e quando il programma la incontra trasferisce il punto di esecuzione all'istruzione immediatamente successiva al blocco stesso.

Se ci sono più costrutti innestati, l'istruzione break si riferisce a quello che la contiene:

```
for(int i=0;i<10;i++)
{
for(int j=0;j<20;j+=2)
{
…
break; //esce dal ciclo interno
}
}
```

L'istruzione continue

L'istruzione **continue** permette di iniziare una nuova iterazione di un ciclo while, do, for oppure foreach.

Analogamente al break, nel caso di più cicli innestati, l'istruzione continue si riferirà al rispettivo ciclo in cui essa viene utilizzata.

L'istruzione goto

L'istruzione **goto** è certamente una delle più discusse della storia dei linguaggi di programmazione, nel bene e nel male. Essa permette di effettuare un salto incondizionato verso un altro punto del programma identificato da un'etichetta.

La sua cattiva fama deriva proprio dal fatto che essa rende il flusso del codice poco controllabile e imprevedibile. L'uso indiscriminato di istruzioni goto produrrebbe infatti dei programmi che spesso vengono soprannominati con il termine dispregiativo di *Spaghetti Code*, nome che deriva dal fatto che il flusso di esecuzione sarebbe appunto aggrovigliato come un piatto di spaghetti.

Un famoso articolo contro l'uso dell'istruzione goto è stato pubblicato nel marzo 1968 da Edsger Dijkstra, celebre informatico olandese, morto nel 2002. Esso è intitolato "Go To

NOTA

Statement Considered Harmful" e contiene una forte critica all'uso eccessivo di istruzioni goto nei linguaggi di programmazione dell'epoca, sostenendo invece la programmazione strutturata. Secondo Dijkstra la qualità dei programmatori è una funzione decrescente della quantità di istruzioni goto presenti nei loro programmi.

A questo punto vi chiederete perché l'istruzione goto è stata introdotta e mantenuta anche in C#, un linguaggio nato decenni dopo e in un epoca ben diversa.

Mostriamo intanto il suo funzionamento e sintassi e poi passeremo al perché in alcuni casi essa può tornare utile. La sintassi è semplice:

```
goto etichetta;
```

dove etichetta è una stringa di testo definita da qualche altra parte nel codice e indica da dove riprendere l'esecuzione in seguito al salto dall'istruzione goto. L'etichetta sarà sempre seguita dai due punti:

```
etichetta:
bloccoIstruzioni
```

Per esempio, il seguente codice scrive 10 volte la stringa hello world, tornando indietro all'etichetta inizio per mezzo di un'istruzione goto, fino a quando un contatore ha valore minore di 10:

```
int i=0;
inizio:
Console.WriteLine("hello world");
i++;
if(i<10)
goto inizio;
```

Il codice precedente, come ormai ben sappiamo, può facilmente essere scritto senza goto, per esempio con un'istruzione for o con un while.

NOTA

Esiste anche un teorema, addirittura precedente all'articolo di Dijkstra, enunciato nel 1966 dagli informatici Corrado Böhm e Giuseppe Jacopini, in base al quale qualunque algoritmo può essere implementato utilizzando tre sole strutture: la sequenza, la selezione e il ciclo, quindi senza istruzioni di salto.

Ci sono un paio di casi pratici in cui l'istruzione goto in C# può tornare utile.

Abbiamo visto come, nel caso di due cicli for innestati, sia possibile uscire dal ciclo for più interno utilizzando un'istruzione break. Per uscire anche dal ciclo for esterno si può utilizzare un goto.

Supponiamo di voler ricercare all'interno di un array bidimensionale un certo numero e quindi di voler uscire dal ciclo for appena la ricerca ha successo, per evitare di scorrere ancora tutti gli elementi rimanenti:

```
int[,] array = {
{1,2,3,4,5},
{6,7,8,9,10}
};

int cercato = 3;
for (int i = 0; i < array.GetLength(0); i++)
{
for (int j = 0; j < array.GetLength(1); j++)
{
if (array[i, j] == cercato)
{
goto trovato;
}
}
}

Console.WriteLine("Il numero non è stato trovato");
goto nontrovato;

trovato:
Console.WriteLine("Il numero è stato trovato");
nontrovato:
Console.WriteLine("fine ricerca");
```

Durante il ciclo se il numero cercato corrisponde a un elemento dell'array in esame, il ciclo si interrompe e l'esecuzione salta all'etichetta trovato. Utilizzando un break in questo caso si sarebbe interrotto solo il ciclo for interno, passando alla successiva iterazione del for esterno.

NOTA	Ci sono delle regole e limitazioni all'utilizzo del goto: non è possibile utilizzare il goto per saltare verso un'etichetta posta in un blocco di codice più interno di quello corrente, per esempio dal for esterno a quello innestato, ed è una fortuna perché la leggibilità del codice ne risulterebbe fortemente compromessa! Questo deriva dal fatto che l'etichetta cui si riferisce un goto deve essere in generale nello scope dell'istruzione goto stessa. Inoltre non è possibile utilizzare un goto per uscire da un blocco finally (l'utilizzo della parola chiave finally verrà introdotto nel Capitolo 8 sulla gestione delle eccezioni).

L'altro caso di utilizzo motivato del goto è all'interno di un costrutto switch, che in effetti già utilizza delle etichette per definire le sezioni.

Abbiamo inoltre già detto che per lo switch vale la regola no-fall through e quindi ogni sezione deve essere chiusa da un'istruzione break.

Per mezzo dell'istruzione goto si può trasferire il controllo a un'etichetta case specifica di un'istruzione switch o all'etichetta default di un'istruzione switch, utilizzando rispettivamente una delle due sintassi seguenti:

```
goto case etichetta;
goto default;
```

Ecco un esempio con i due utilizzi di goto in uno switch:

```
decimal prezzo = 5;
decimal sconto = 2;
decimal prevendita = 0.5m;
DayOfWeek day = DateTime.Today.DayOfWeek;
switch(day)
{
case DayOfWeek.Monday:
prezzo += 5;
break;

case DayOfWeek.Tuesday:
prezzo-=sconto;
goto case DayOfWeek.Monday;
case DayOfWeek.Sunday:
prezzo+=2;
goto default;

default:
prezzo += prevendita;
break;
}
Console.WriteLine(prezzo);
```

Nel caso DayOfWeek.Tuesday, dalla variabile prezzo viene decrementato un importo di sconto e poi il flusso dell'esecuzione salta al case Monday.

Nel caso invece di DayOfWeek.Sunday, il prezzo viene incrementato e poi salta all'etichetta default.

L'istruzione return

L'istruzione **return** serve a uscire da un metodo di una classe, restituendo il controllo al chiamante del metodo stesso.

Se il metodo ha dichiarato un tipo di ritorno, l'istruzione return serve a indicare il valore restituito dal metodo. Per esempio, il metodo Main può essere scritto in maniera da restituire un numero int, che potrà indicare un valore da restituire all'esterno, per esempio a uno script che ha lanciato il programma. Per restituire tale valore si utilizza quindi un'istruzione return seguita dall'int:

```
static int Main(string[] args)
{
if(args.Length==0)
{
```

```
Console.WriteLine("Non è stato inviato alcun parametro");
return -1;
}
else
{
…
return 0;
}
}
```

Nel caso in cui un metodo non debba restituire alcun valore al chiamante, esso dichiarerà come tipo di ritorno void e quindi per uscire dal metodo basterà un'istruzione return senza specificare alcun valore:

```
static void Main(string[] args)
{
if(args.Length==0)
{
Console.WriteLine("Non è stato inviato alcun parametro");
return;
}
else
{
…
return;
}
}
```

Nel caso di void il metodo può anche terminare la sua esecuzione senza return, in quanto esso termina implicitamente appena raggiunta la parentesi graffa di chiusura del suo blocco:

```
static void Main(string[] args)
{
if(args.Length==0)
{
Console.WriteLine("Non è stato inviato alcun parametro");
return;
}
…
//non è necessaria l'istruzione return
}
```

L'istruzione **throw**

Anticipiamo il capitolo sulla gestione delle eccezioni per introdurre l'ultima delle istruzioni di salto.

L'istruzione **throw** permette di segnalare una situazione anomala (che appunto viene chiamata *eccezione*) avvenuta durante la normale esecuzione di un programma. L'eccezione da lanciare è, come vedremo, un oggetto di un tipo derivato dalla classe System.Exception:

```
public class ThrowTest
{
```

```
static void Main(string[] args)
{
if (args.Length == 0)
{
throw new ArgumentNullException();
}
Console.Write("{0} argomenti", args.Length); // se l'eccezione viene lanciata non si arriverà mai a questa linea
}
}
```

Nell'esempio precedente, se non viene passato alcun argomento al programma, viene lanciata, mediante l'istruzione throw, un'eccezione di tipo ArgumentNullException. In tal caso il programma termina la sua esecuzione, non arrivando mai all'istruzione Console.Write.

In generale l'istruzione throw verrà sempre utilizzata in congiunzione con un costrutto try/catch che permette di gestire appunto il verificarsi di un'eccezione e intraprendere eventuali azioni correttive.

Il Capitolo 8 affronterà in modo approfondito la gestione delle eccezioni in C#.

Domande di riepilogo

1) Che tipo può essere utilizzato nella condizione di un'istruzione if?

a. int oppure bool
b. bool
c. string
d. Un tipo qualsiasi

2) Usando l'istruzione break in un blocco switch:

a. si esce dal programma
b. si passa al successivo case
c. si esce dallo switch
d. si valuta nuovamente l'espressione dello switch

3) Quali sono le istruzioni di iterazione utilizzabili in C#?

a. for, while, loop, goto;
b. for, do while, foreach, forever
c. for, when, while, branch
d. for, foreach, while, do while

4) Nel linguaggio C# esiste l'istruzione goto. Vero o falso?

5) Quale istruzione si utilizza per restituire un valore da un metodo?

a. return
b. throw
c. void
d. result

6) L'istruzione continue può essere utilizzata in un qualunque tipo di iterazione for, foreach, while, do while. Vero o falso?

7) Qual è il valore di i dopo l'esecuzione del seguente codice:

```
int i = 10;
do
{
i++;
} while(i<10);
```

a. 10
b. 11

c. si verifica un ciclo infinito
d. 0

Programmazione a oggetti in C#

C# fornisce il pieno supporto a tutti i concetti della programmazione orientata agli oggetti, per la creazione e configurazione di classi e struct e dei relativi membri con cui definirne struttura e comportamento.

Il linguaggio C#, come Java e C++, è un linguaggio che è stato progettato principalmente per l'impiego del paradigma orientato agli oggetti, ma che permette anche di utilizzare parecchi aspetti procedurali.

C#, inoltre, si è evoluto in maniera tale che, allo stato attuale, viene definito un linguaggio multiparadigma in quanto possiede anche caratteristiche tali da renderlo, a seconda dei casi, utilizzabile come linguaggio dichiarativo, imperativo e funzionale.

Nei prossimi paragrafi vedremo innanzitutto i concetti fondamentali del paradigma di programmazione orientata agli oggetti, per poi passare alla loro interpretazione da parte del linguaggio C#, il che ci permetterà di creare le nostre gerarchie di classi e utilizzarle per scrivere applicazioni sempre più complesse.

La programmazione orientata agli oggetti

La programmazione *orientata agli oggetti*, o semplicemente programmazione a oggetti, è una filosofia di progettazione e sviluppo software in cui la struttura di un programma è basata su un insieme di oggetti che interagiscono e collaborano fra di loro per eseguire un compito.

Il concetto è quindi notevolmente differente da altri modelli di programmazione in cui, per esempio, un programma consiste in una serie sequenziale di istruzioni, da eseguire una dopo l'altra.

Nella *OOP* (*Object Oriented Programming*) ogni oggetto è capace di inviare e ricevere messaggi da altri oggetti e ognuno di essi possiede uno stato e può compiere determinate azioni, strettamente correlati con il tipo di oggetto.

I concetti della programmazione orientata agli oggetti nascono dal mondo reale. In fondo, quando dobbiamo eseguire una qualsiasi azione possiamo pensare a noi stessi come se stessimo interagendo con un oggetto. Per esempio, per scrivere una lettera al computer dobbiamo innanzitutto interagire con il computer, che è un oggetto. L'oggetto computer possiede un pulsante, che dobbiamo premere per accenderlo. L'azione di accensione corrisponde a inviare all'oggetto computer un preciso messaggio.

Per utilizzare poi il programma di elaborazione testi avviamo l'applicazione interagendo, per esempio, con il mouse mediante un clic su un'icona. Non abbiamo fatto altro che inviare un altro messaggio al sistema operativo, indicandogli il programma da avviare.

Se infine vogliamo stampare il documento, basterà cliccare su un apposito pulsante ed eventualmente scegliere la stampante da utilizzare.

Ancora una volta possiamo pensare a queste azioni come a precisi messaggi inviati al computer, con determinati oggetti che interagiscono a loro volta fra di loro scambiandosi altri messaggi (il documento da stampare, la stampante da utilizzare e così via).

In tutto ciò l'utente normale non sa, e probabilmente non gli interessa nemmeno sapere, cosa succede dentro al computer. Il funzionamento interno è incapsulato e nascosto; ciò che l'utente può fare è interagire mediante un'interfaccia esterna (il pulsante di accensione, la tastiera, le icone da cliccare).

La programmazione orientata agli oggetti porta questa tipologia di pensiero all'interno del mondo dello sviluppo software. In tal modo, anziché pensare a un programma come a una

sequenza di istruzioni e calcoli da eseguire uno dopo l'altro, bisogna iniziare a pensare agli oggetti che partecipano e collaborano per realizzare un determinato compito.

Per esempio, un programma dotato di interfaccia grafica potrà essere pensato e sviluppato come un insieme di oggetti finestra, all'interno dei quali avremo altri oggetti caselle di testo e pulsanti, per mezzo dei quali sarà possibile inviare messaggi al programma stesso per indicargli le azioni da eseguire.

Storia della OOP

La programmazione orientata agli oggetti è una tecnica di programmazione che muove i primi passi negli anni Cinquanta e Sessanta del secolo scorso, secondo la quale ogni concetto può essere rappresentato come un oggetto dotato di uno stato.

Il linguaggio che generalmente viene indicato come primo esempio di linguaggio dotato di funzionalità object oriented è *Simula* (1967), mentre l'esempio canonico di linguaggio totalmente orientato agli oggetti è *Smalltalk* (1972), con il quale sono stati sviluppati quasi tutti i concetti fondamentali di tale paradigma.

Sebbene inizialmente la maggioranza dei programmatori in tali epoche abbia ignorato il nuovo paradigma, continuando a preferire linguaggi ancora più storici e affermati (e soprattutto basati sul paradigma procedurale) per sviluppare i propri programmi (*C*, *Basic*, *Pascal*, *Fortran*), i concetti della programmazione a oggetti tornano prepotentemente ad affacciarsi nel mondo dello sviluppo software negli anni Ottanta, con il linguaggio *C++*, poi negli anni Novanta con *Java*, mentre C#, come già ampiamente detto, vede la luce e inizia a diffondersi nel 2000.

Oggi il paradigma orientato agli oggetti continua a essere ampiamente utilizzato in svariati settori dello sviluppo software.

Vantaggi della programmazione orientata agli oggetti

Il paradigma di programmazione orientato agli oggetti nasce per cercare di risolvere alcune delle problematiche tipiche dei linguaggi di programmazione procedurale. I programmi scritti in maniera procedurale, infatti, risultavano dei blocchi monolitici contenenti tutte le funzionalità in uno o al massimo pochi moduli.

Con il crescere della complessità dei programmi e quindi della dimensione del codice sorgente si arrivava a una notevole difficoltà di manutenzione e debugging, e spesso capitava che ogni sviluppatore dovesse conoscere l'intero funzionamento del programma per poter correggere errori o apportare modifiche. Da ciò derivano i primi concetti di programmazione strutturata, con la possibilità di suddividere il codice sorgente in diversi file e incapsulare le

funzionalità in moduli o funzioni più piccoli, ognuno magari mantenuto e sviluppato da programmatori diversi.

Inoltre, avere applicazioni basate su tali caratteristiche portava spesso a introdurre errori in maniera involontaria, perché ogni modifica su una parte di codice poteva influire sul funzionamento del resto del programma.

L'isolamento e l'incapsulamento di funzionalità all'interno di oggetti separati permette invece di mantenere il codice in maniera molto più semplice, separandone le responsabilità anche in diversi team.

Pensare a oggetti permette poi un'analisi del modello di business più veloce, traducendolo facilmente nel modello da sviluppare in un software, anche perché è il cliente stesso che riesce a esporre più facilmente e chiaramente le proprie esigenze.

Nel mondo attuale, in cui quasi ogni applicazione deve poter essere integrata in sistemi esistenti e lavorare in ambiente distribuito, il modello di programmazione lineare procedurale è diventato insufficiente e praticamente impossibile da implementare.

È sempre più fondamentale, se non obbligatorio, poter adottare modelli di programmazione che permettano di sviluppare e distribuire il software in maniera da rendere possibile la distribuzione di componenti, l'accesso a database remoti, l'uso di interfacce utente separate dalla logica, la migrazione su dispositivi e sistemi remoti, il tutto in maniera da poter assegnare a diversi team di specialisti singole parti dell'intero progetto.

Le caratteristiche della OOP che vedremo nel resto del capitolo si adeguano perfettamente a tali nuove metodologie ed esigenze di sviluppo software.

Non preoccupatevi intanto di capire come tali caratteristiche si leghino con C#, ci arriveremo più avanti.

Se comunque avete già esperienza di sviluppo orientato agli oggetti e ne conoscete i concetti fondamentali, potete saltare questa sezione teorica.

Che cos'è un oggetto

L'oggetto è il concetto fondamentale del paradigma orientato agli oggetti. In tutti i testi, siti, articoli che parlano di programmazione a oggetti leggerete che tutto può e deve essere pensato come se fosse un oggetto.

Utilizzando una definizione più precisa, data da uno dei padri del paradigma OOP stesso, *Grady Booch*, un oggetto è un qualcosa che possiede un suo stato, un suo comportamento e una sua identità.

Dal punto di vista di un programmatore, quindi, un oggetto sarà una struttura che mantiene dei dati al suo interno, che fornisce delle funzioni per manipolarli e che risiede in una propria area di memoria riservata, convivendo e comunicando con altri oggetti, dello stesso tipo o di altro tipo.

Per esempio, se dovessimo occuparci di progettare e sviluppare un'applicazione dedicata al mondo della telefonia, un oggetto Smartphone potrebbe essere caratterizzato, fra le altre cose, da una marca e un modello, e mettere a disposizione dell'utente un'interfaccia mediante la quale avviare una chiamata, leggere messaggi e così via.

La struttura e il comportamento di oggetti simili sono definiti in una classe comune, quindi in termini OOP un oggetto è un'*istanza* di una classe (vedremo fra pochi paragrafi come implementare in C# una classe), anzi i termini istanza e oggetto sono intercambiabili e sinonimi e la creazione di un oggetto viene detta anche istanziazione dell'oggetto.

Riprendendo l'esempio precedente, la classe Smartphone è una struttura dati o template che permette di rappresentare un tipo di telefonino. Un'istanza della classe Smartphone è invece un oggetto specifico, quindi potranno esserci diversi oggetti di tipo Smartphone, ognuno con una propria identità (per esempio ognuno ha un codice IMEI univoco), anche se il funzionamento poi sarà identico.

La differenza fra classe e oggetto è sottile, ma fondamentale da comprendere. La classe è qualcosa di astratto, che non esiste, è ciò che definisce come un oggetto sarà costruito, una sorta di documento progettuale.

L'oggetto è concreto, esiste, è costruito in base al progetto definito dalla sua classe, ha uno stato costituito da dati e valori e può eseguire delle funzionalità o scambiare messaggi con altri oggetti.

Quando dunque si inizia a progettare un'applicazione, bisogna innanzitutto pensare ai concetti del dominio che si sta affrontando e ricavarne gli oggetti e le funzionalità che essi dovranno avere, fornendo magari le modalità di comunicazione fra essi o con oggetti di classi diverse.

I principi della OOP

Nei prossimi paragrafi verranno esposti dei concetti fondamentali del paradigma orientato agli oggetti, validi per qualsiasi linguaggio di programmazione che ricada in questa categoria. Non preoccupatevi quindi di come essi vengono implementati in C#, lo vedremo più avanti, nel resto del capitolo.

Astrazione

L'*astrazione* (o abstraction) non è un concetto peculiare del paradigma a oggetti, esso è presente in qualunque linguaggio che permetta di definire delle strutture dati, denominate anche ADT (Abstract Data Type), con cui il programma dovrà interagire.

Quando si pensa a come creare tali strutture dati bisogna pensare a quali dettagli dovranno essere gestiti dall'applicazione ed eliminare tutto ciò che non serve o comunque non interessa ai fini dell'applicazione stessa, cercando di ridurre al minimo tutte le complessità superflue.

Uno smartphone, per esempio, potrebbe essere rappresentato mediante una struttura dati che permetta di memorizzare la sua marca, il suo modello e magari altre caratteristiche peculiari come la quantità di memoria interna e la dimensione dello schermo, mentre dal punto di vista dell'utilizzatore potremmo definire delle funzioni come l'accensione, lo spegnimento, l'invio di un SMS e così via.

Tali caratteristiche e comportamenti dipendono però dall'applicazione stessa e dalla prospettiva con cui viene vista dai suoi utenti. Se, per esempio, il programma si rivolge a utenti più tecnici ed esperti, allora potrebbe essere necessario memorizzare e gestire dati come il codice IMEI, il tipo di batteria utilizzata, il suo voltaggio e altri dettagli ancora più complessi e specifici, fornendo di conseguenza anche i metodi e le funzioni che permettano di interagire con essi.

Il processo di astrazione è, dunque, quello con cui si descrive in maniera più o meno essenziale un oggetto, nascondendo i dettagli implementativi interni e riducendo le complessità, in maniera che lo sviluppatore possa concentrarsi su pochi aspetti per volta.

Come vedremo nel prossimo paragrafo, l'astrazione è un concetto strettamente correlato con quello di incapsulamento.

Incapsulamento

Quando si interagisce o si utilizza un oggetto nel mondo reale non ci interessa come esso sia stato costruito internamente e come sta funzionando.

Chi utilizza un telefonino, per esempio, non sa, e probabilmente non ci pensa nemmeno, cosa accade al suo interno quando digita numeri e lettere sulla tastiera e poi preme Invio per inviare un SMS.

L'*incapsulamento* (detto anche *encapsulation* o *information hiding*) è il processo mediante il quale si nascondono al mondo esterno i dettagli implementativi e si proteggono i dati interni degli oggetti (incapsulandoli appunto).

In termini di programmazione C# vedremo come tutto ciò si realizzi mediante i concetti di proprietà e metodi e i relativi modificatori di accesso, meccanismi che forniscono al mondo

esterno la possibilità di interagire con l'oggetto e modificarne lo stato senza dover conoscere e modificare direttamente i dettagli interni.

Riprendendo l'esempio dello smartphone, a un utilizzatore non interessa come viene rappresentato internamente un contatto salvato in rubrica, ma può utilizzare delle funzioni per ricercare, creare, modificare ed eliminare tali contatti.

L'incapsulamento rende quindi i dati degli oggetti più sicuri e affidabili, perché esistono e si conoscono quali sono i modi per accedere a essi e quali sono le operazioni ammesse per modificarli.

Inoltre, ciò permette di modificare i dettagli interni senza dover avvisare e quindi costringere gli oggetti esterni ad alterare le modalità con cui interagiscono con un dato oggetto.

Per fare un esempio, è come se in uno smartphone cambiassimo modalità di gestione della rubrica contatti, magari salvandola in formato XML anziché binario: ciò resterebbe totalmente trasparente all'utilizzatore esterno, che continuerebbe a usare le solite funzioni senza accorgersi del cambiamento interno. Rendere tutto visibile all'esterno sarebbe come fornire all'utilizzatore dello smartphone la possibilità di aprire l'oggetto e interagire mediante un cacciavite sui suoi componenti!

I meccanismi di incapsulamento vengono realizzati da C# mediante il concetto di classe, che fornisce un'interfaccia pubblica utilizzabile dall'esterno, nascondendo i dettagli implementativi interni.

Ereditarietà

L'*ereditarietà* è un meccanismo che permette e facilita il riuso del codice esistente e la manutenzione dello stesso. Infatti, se una classe possiede determinate caratteristiche possiamo riutilizzarle creando una nuova classe derivata da essa e che erediti alcune di tali caratteristiche, evitando di duplicare codice già scritto.

In natura molti oggetti possono essere classificati mediante delle gerarchie, in accordo a determinate caratteristiche comuni, o ad altre caratteristiche che vengono invece estese a oggetti più specializzati.

La *specializzazione* è infatti il meccanismo più comune di ereditarietà, che può essere descritto mediante una relazione di tipo "è un".

Per esempio, quando diciamo che un cane è un animale, stiamo affermando che il cane è una versione specializzata del tipo animale. Il cane è un animale con certe caratteristiche, per esempio ha una coda, quattro zampe e abbaia. Anche il gatto è una versione speciale di un

animale, ha delle caratteristiche comuni con il cane, ma tante altre invece peculiari della sua razza, per esempio non abbaia ma miagola.

In termini OOP il meccanismo di ereditarietà permette di definire nuove classi, ereditandole da quelle esistenti e quindi estendendone le caratteristiche mediante nuovi metodi e proprietà, oppure ridefinendone alcune.

Ciò permette di semplificare il modo di lavorare con gli oggetti, in quanto molte caratteristiche verranno inglobate in un'unica classe ed esse potranno poi essere riutilizzate in classi da essa derivate.

In C# è consentita la cosiddetta *ereditarietà singola*, cioè una classe può ereditare le caratteristiche di una e una sola classe madre, mentre con il concetto di interfacce vedremo come sia possibile creare una sorta di *ereditarietà multipla*.

L'ereditarietà, in .NET, è consentita solo nella famiglia dei tipi riferimento.

Negli esempi precedenti abbiamo già utilizzato termini specifici riguardanti il concetto di ereditarietà.

Quando si definiscono e si usano delle gerarchie di classi vengono utilizzate diverse terminologie, con molti termini che indicano lo stesso concetto, cioè sinonimi. Per esempio, una classe che rappresenta la *classe madre* di una gerarchia viene anche detta *classe base* o *superclasse*. Al contrario, la *classe figlia* viene anche detta *classe derivata*, o ancora classe che estende un'altra classe, perché alla versione base essa aggiunge altre caratteristiche.

Polimorfismo

Un altro dei pilastri fondamentali del paradigma orientato agli oggetti è il *polimorfismo*. Il termine polimorfismo indica letteralmente la possibilità di assumere molte forme e, nell'ambito della programmazione, rappresenta la possibilità per differenti oggetti di eseguire una stessa azione o rispondere a uno stesso messaggio in maniera diversa. Esso è quindi strettamente legato all'ereditarietà.

Riprendendo l'esempio di cane e gatto, come classi derivate da una stessa classe animale, essi eseguiranno l'azione di emettere un verso in maniera specifica: il gatto miagola, il cane abbaia.

In un linguaggio orientato agli oggetti, come C#, tale concetto verrà implementato mediante le operazioni di *overloading* dei metodi, che permette di implementare in una stessa classe più metodi con lo stesso nome e parametri differenti, oppure di *overriding*, che invece permette di ridefinire un metodo presente in una classe base in una classe derivata, quindi con stesso nome e stessi parametri.

Per tali motivi l'overloading è anche detto *polimorfismo a tempo di compilazione*, mentre l'overriding è un *polimorfismo a tempo di esecuzione*.

Le classi

In un linguaggio orientato agli oggetti, creare nuovi tipi di dati vuol dire creare nuove classi. In .NET e quindi C#, come visto nel Capitolo 2 e nel Capitolo 3, sono presenti due differenti categorie di tipi: valore e riferimento.

Mediante i tipi riferimento, C# implementa tutte le caratteristiche del paradigma di programmazione orientato agli oggetti viste nei precedenti paragrafi.

Il tipo più comune di tipi riferimento è la classe, cioè una struttura dati che definisce e mantiene lo stato e il comportamento in una singola unità.

La creazione di applicazioni di una certa complessità implica l'attività di astrazione degli oggetti che fanno parte del dominio applicativo, l'incapsulamento delle loro caratteristiche in classi, la creazione di gerarchie di classi comuni da cui derivare classi specializzate, l'implementazione di metodi che svolgano compiti specifici per ogni classe.

La creazione di una classe in C# avviene mediante l'utilizzo della parola chiave class, la cui sintassi più semplice prevede l'indicazione del nome della classe e quindi del blocco che conterrà i suoi membri:

```
class SmartPhone
{
//membri di classe
}
```

NOTA	In C# non è obbligatorio chiamare la classe con lo stesso nome del file in cui essa sarà contenuta o viceversa, anche se per convenzione o comodità ciò si verifica spesso. Inoltre uno stesso file può contenere più classi.

La sintassi completa per definire una classe può anche essere più complessa e prevedere altri elementi. La sintassi completa è:

```
[attributi] [modificatori] [partial] class NomeClasse [:ClasseBase, Interfacce, ParametroTipoGenerico]
{
[membri]
}
```

Quindi una definizione di classe può anche essere parecchio complessa, con diversi attributi, modificatori, il nome della classe da cui deriva e così via.

NOTA	Per utilizzare, per esempio, l'ereditarietà in C# si usano i due punti (:) subito dopo il nome della classe che si sta definendo, seguiti dal nome della classe madre. Poiché in .NET ogni

classe deriva dalla classe System.Object, se non è indicato esplicitamente un altro tipo, la classe che stiamo definendo sarà direttamente e implicitamente figlia di System.Object.

Nei prossimi paragrafi vedremo come utilizzare tali elementi per creare le nostre gerarchie di classi e controllarne i vari dettagli.

Modificatori di accesso

Prima di andare avanti con gli altri elementi che permettono di definire la struttura completa di una classe è necessario comprendere come controllare l'accesso alle classi e ai loro membri da parte di altro codice, vale a dire come implementare e controllare il livello di incapsulamento.

È possibile infatti specificare una sorta di permesso di visibilità per mezzo dei cosiddetti *modificatori di accesso*, che possono essere indicati nella definizione di una classe (prima della parola chiave class) o precedendo la dichiarazione dei membri della classe.

Tabella 6.1 **- I modificatori di accesso.**

Modificatore	A che cosa si applica	Descrizione
public	tipi e membri	I tipi e membri public non hanno limiti di accesso: sono visibili da qualunque tipo, da tipi derivati e da altri assembly.
protected	membri e tipi innestati	Accesso consentito solo all'interno della classe che definisce l'elemento e da classi derivate.
private	membri e tipi innestati	Accesso consentito solo all'interno della classe che definisce l'elemento.
internal	tipi e membri	Gli elementi internal sono accessibili solo all'interno dell'assembly in cui sono definiti.
protected internal	membri e tipi innestati	Gli elementi che combinano protected e internal sono accessibili da qualunque tipo nello stesso assembly e in classi derivate anche di altri assembly.

Il modificatore predefinito di una classe, se non specificato, è internal, il che indica che una classe è visibile solo all'interno dell'assembly che la contiene. Il modificatore internal si può naturalmente anche specificare in maniera esplicita.

Esso si utilizza in genere all'interno di librerie di classi contenenti tipi (che saranno indicati come public) esposti per l'uso da parte di progetti differenti, e in cui si indicano con internal le

classi che saranno a solo uso interno (dell'assembly stesso).

Per esempio, possiamo esplicitamente dichiarare come internal la classe SmartPhone così:

```
internal class SmartPhone
{
}
```

NOTA	È possibile rendere i tipi e i membri internal di un assembly visibile anche a degli assembly definiti come "amici", mediante un apposito attributo chiamato InternalsVisibleTo. Nel Capitolo 14, parlando di attributi globali, vedremo come utilizzarlo per mettere in pratica tale possibilità.

Se una classe è indicata come public essa sarà invece utilizzabile da qualunque altra classe, anche in altri assembly, e infatti tale modificatore viene scelto spesso per le classi definite all'interno di librerie di classi da referenziare anche in altri progetti.

```
public class SmartPhone
{
}
```

I modificatori private, protected e protected internal, utilizzati nella dichiarazione di una classe, hanno un uso specifico, in quanto essi in genere indicano che un tipo o membro non può essere utilizzato in classi differenti o che esso è visibile solo in una classe derivata da quella attuale.

Per esempio, una classe marcata come private non potrebbe nemmeno essere istanziata, quindi i modificatori private, protected e protected internal sono utilizzabili solo nel caso di tipi innestati (vedi più avanti in questo capitolo), in maniera da poter controllare dove una classe innestata è istanziabile:

```
internal class SmartPhone
{
private class Battery
{
}
}
```

I membri di una classe invece sono implicitamente private, quindi, se non altrimenti specificato mediante altri modificatori, essi sono visibili e utilizzabili solo all'interno della classe stessa.

Membri di una classe

Dopo aver definito la classe bisognerà poi definirne i relativi membri. Una classe può contenere membri dati (campi e costanti), che mantengono lo stato di un oggetto, membri funzione (metodi, proprietà, eventi, indicizzatori, operatori, costruttori e distruttori), che

invece ne definiscono il comportamento, ed eventuali altri definizioni di tipi innestati al suo interno.

La Tabella 6.2 elenca le diverse tipologie di membri che possono appartenere a una classe.

Non esiste una regola obbligatoria per l'ordine in cui i membri sono definiti in una classe, ma spesso, per convenzione, i campi e le costanti vengono definiti per primi.

Tabella 6.2 - **Tipi di membri di una classe.**

Membro	Tipo di membro	Descrizione
Campo	dati	Variabili utilizzate per contenere dati associati alla classe.
Costante	dati	Valori costanti associati alla classe.
Evento	dati	Membri che consentono di notificare ad altri oggetti un evento accaduto all'interno della classe.
Metodo	funzione	Azioni eseguibili dalla classe.
Proprietà	funzione	Punti di accesso ai dati della classe, sia in lettura sia in scrittura.
Costruttore	funzione	Azioni eseguite alla costruzione di un'istanza della classe.
Distruttore o Finalizzatore	funzione	Azioni eseguite quando il CLR determina che un oggetto non è più utilizzato e quindi può essere rimosso dalla memoria.
Operatore	funzione	C# consente di ridefinire degli operatori (come + e -) in maniera da poter essere utilizzati direttamente con le istanze della classe.
Indicizzatore	funzione	Tramite gli indicizzatori è possibile accedere agli oggetti mediante degli indici, come fossero array o collezioni.
Tipo innestato	dati	Tipi definiti all'interno di un altro tipo (classe o struct).

A eccezione dei costruttori statici, è possibile specificare un modificatore di accesso per ognuno di tali membri, per controllare la visibilità dei membri stessi.

Per esempio, un membro definito come public sarà utilizzabile ovunque, mentre i membri private saranno utilizzabili solo all'interno della classe a cui appartengono.

I membri di una classe possono essere membri associati a ogni particolare istanza della classe, oppure membri statici che, come vedremo fra qualche paragrafo, sono dei membri associati alla classe intera, e non a dei particolari oggetti.

Come abbiamo già avuto modo di vedere nei capitoli precedenti, per accedere ai membri di una classe, ove le regole di visibilità (eventualmente controllate mediante i modificatori) lo permettano, si utilizza l'operatore punto (.), per esempio:

```
string str="hello";
int lunghezza=str.Length;
```

Nel codice precedente mediante l'operatore . si accede alla proprietà denominata Length della classe String, assegnando il suo valore alla variabile lunghezza.

Un membro di una classe, avendo un proprio tipo, può poi essere utilizzato ripetendo l'uso dell'operatore punto, in cascata.

Per esempio, la classe Console, ha una proprietà Title che rappresenta il titolo della finestra console, di tipo string, e quest'ultimo ha una proprietà Length per ottenere la lunghezza della stringa in questione:

```
int len = Console.Title.Length;
```

NOTA	Nel Capitolo 4, si è visto il funzionamento dell'operatore null conditional. Esso permette di accedere ai vari membri di un oggetto, anche in cascata, verificando che ognuno di essi sia diverso da null.

Nei prossimi paragrafi introdurremo i vari tipi di membri definibili in una classe, a esclusione di eventi e operatori che vedremo invece nei prossimi capitoli.

Campi e costanti

Un *campo* è una variabile che appartiene a una classe, quindi il suo ambito si estende al solo corpo della classe in cui viene definito.

I campi di una classe contengono e rappresentano lo stato di ogni istanza della classe stessa. Ciò significa che ogni oggetto, istanza di una determinata classe, ha una propria copia dei valori contenuti nei campi, ben distinti da quelli di un'altra istanza della classe.

La sintassi più semplice per definire i campi di una classe è analoga a quella vista per la definizione di una qualunque variabile, basta quindi dichiararne tipo e identificatore. Per esempio, possiamo aggiungere alla classe Smartphone definita in precedenza un campo string per memorizzarne la marca e uno per il modello:

```
class Smartphone
{
string marca;
string modello;
}
```

NOTA	A differenza di C++, in C# non è possibile definire variabili globali, cioè esterne a qualunque definizione di classe.

I membri di una classe sono implicitamente private (vedere il paragrafo sui modificatori di accesso), quindi nel precedente esempio i campi e le costanti della classe sono entrambi privati, cioè utilizzabili solo all'interno della classe Smartphone.

Specificando il modificatore private si ha solo maggior chiarezza nel codice; la seguente dichiarazione di classe è perfettamente equivalente alla precedente:

```
class Smartphone
{
private string marca;
private string modello;
}
```

Essendo i campi privati, cercare di utilizzarli direttamente con l'operatore . provocherebbe un errore di compilazione:

```
static void Main()
{
Smartphone phone=new Smartphone();
phone.marca="sony"; //errore, il campo è private
}
```

In genere i campi di una classe sono dichiarati private oppure protected, come vedremo parlando di classi derivate, nel caso in cui essi debbano essere ereditati da una classe che estende quella in cui sono definiti.

Se modifichiamo quindi la visibilità dei campi in public per poterli utilizzare direttamente:

```
class Smartphone
{
public string marca;
public string modello;
}
```

possiamo a questo punto accedere direttamente ai campi marca e modello da una qualunque altra classe del programma, per esempio scrivendo direttamente i nuovi valori:

```
phone.marca="sony";
phone.modello="t28";
```

Il modificatore public romperebbe le regole di incapsulamento e quindi, anziché dichiarare dei campi come public, è meglio controllarne l'accesso o permetterne la modifica mediante proprietà o metodi.

Per accedere a un membro privato, all'interno della stessa classe naturalmente, basta utilizzare il nome del membro stesso. Nel seguente esempio il metodo GetDescrizione (vedremo a breve

cosa sono e come dichiarare i metodi di una classe) restituisce una stringa risultante dalla concatenazione dei due campi privati:

```
class Smartphone
{
private string marca;
private string modello;

public string GetDescrizione()
{
return marca + " " + modello; }
}
```

In tal modo, costruendo un oggetto di classe Smartphone sarà possibile accedere al metodo GetDescrizione per ottenerne appunto una descrizione, ma non accedere direttamente ai campi privati. Per esempio, se nel metodo Main definito nella classe Program si costruisce un simile oggetto potremmo scrivere:

```
class Program
{
static void Main()
{
Smartphone sp=new Smartphone();
string descrizione=sp.GetDescrizione(); //ok
sp.marca="windows phone"; //errore se il campo marca è private
}
}
```

Come per le variabili locali, è possibile dichiarare più campi che condividano modificatori e tipo, separandoli con la virgola:

```
class Smartphone
{
private string marca, modello;
}
```

L'inizializzazione dei campi è opzionale; essi possono essere dichiarati e inizializzati, altrimenti assumeranno il valore di default previsto per il proprio tipo (vedere il Paragrafo "Valori Predefiniti" nel Capitolo 3). Possiamo verificare tali valori scrivendo una classe che contenga campi di tutti i tipi e stampandoli subito dopo averne creato un oggetto, quindi per comodità dichiareremo i campi public:

```
class ValoriPredefiniti
{
public byte b;
public sbyte sb;
public short s;
public ushort us;
public int i;
public uint ui;
public long l;
public ulong ul;
```

```
public float f;
public double d;
public char ch;
public bool bo;
public decimal dec;
public string str;

static void Main()
{
ValoriPredefiniti dv=new ValoriPredefiniti ();
Console.WriteLine("byte={0}", dv.b);
Console.WriteLine("sbyte={0}", dv.sb);
Console.WriteLine("short={0}", dv.s);
Console.WriteLine("ushort={0}", dv.us);
Console.WriteLine("int={0}", dv.i);
Console.WriteLine("uint={0}", dv.ui);
Console.WriteLine("long={0}", dv.l);
Console.WriteLine("ulong={0}", dv.ul);
Console.WriteLine("float={0}", dv.f);
Console.WriteLine("double={0}", dv.d);
Console.WriteLine("char={0}", dv.ch);
Console.WriteLine("bool={0}", dv.bo);
Console.WriteLine("decimal={0}", dv.dec);
Console.WriteLine("string={0}", dv.str);}
```

Un campo può naturalmente essere anche di un tipo riferimento, in tal modo si potranno creare classi che al loro interno contengono riferimenti ad altri tipi complessi, sfruttando un meccanismo di *composizione* di oggetti, detto anche *aggregazione*.

Per esempio, un oggetto Smartphone, potrà essere pensato come composto da diversi elementi: una batteria, una scheda SIM, diversi pulsanti e così via.

Se è necessario gestire ognuno degli elementi nel nostro programma, possiamo pensarli come possibili classi e quindi implementarle per esempio nel seguente modo:

```
class Battery
{
private double percentualeCarica;
}

class Sim
{
private ushort prefisso;
private uint numero;
}
```

A questo punto, la classe Smartphone potrà contenere e aggregare al suo interno degli oggetti di tipo Battery e Sim, semplicemente definendo gli appositi campi:

```
class Smartphone
{
private Battery battery;
private Sim sim;
}
```

Spesso capiterà anche di dover gestire nelle nostre classi collezioni di elementi diversi, per esempio uno smartphone potrà avere una tastiera composta da una collezione di oggetti, che magari possiamo pensare come un array di Button:

```
class Button
{
private string Simbolo;
}

class Smartphone
{
private Battery battery;
private Sim sim;
private Button[] pulsanti;
}
```

Una costante di classe è invece un particolare tipo di campo il cui valore è predeterminato e assegnato a tempo di compilazione. Per dichiarare un campo costante si fa precedere alla dichiarazione del campo il modificatore const:

```
class Cerchio
{
private const double PI_GRECO=3.1415D;
}
```

In questo caso il campo, oltre a essere private, è anche dichiarato come costante.

Il valore di una costante deve essere obbligatoriamente indicato contemporaneamente alla sua assegnazione e quindi a tempo di compilazione.

Ci sono però casi in cui il valore di un campo deve rimanere costante durante tutto il tempo di esecuzione di un programma, ma può capitare che il suo valore non sia conosciuto a priori. In questo caso è possibile utilizzare dei campi a sola lettura, indicando mediante il modificatore readonly che il valore sarà assegnato al campo all'interno di un costruttore della classe:

```
public class Document
{
private readonly DateTime creationTime;

public Document()
{
creationTime = DateTime.Now;
}
}
```

Un campo readonly non deve essere obbligatoriamente inizializzato nel costruttore, ma è possibile farlo contemporaneamente alla sua dichiarazione:

```
public class Document
{
private readonly DateTime creationTime= DateTime.Now;

}
```

NOTA	Una curiosità da notare è che un campo readonly può essere assegnato più volte nei contesti in cui è permesso, cioè è possibile avere un'assegnazione in linea con la dichiarazione del campo stesso e poi un'altra assegnazione nel costruttore della classe.

Metodi

I membri funzione principali di una classe sono i *metodi*, che permettono di definire il comportamento di un oggetto, eseguendo delle azioni o inviando e ricevendo dei messaggi. Un metodo può ricevere dei dati di ingresso, chiamati parametri, e restituire dei dati di output, se è stato definito il tipo di ritorno di tali dati. La sintassi generale per dichiarare un metodo è quindi la seguente:

```
[modificatori] TipoRitorno NomeDelMetodo([Parametri separati da virgola])
{
//istruzioni }
```

Anche in questo caso i modificatori di accesso permettono di controllare la visibilità del metodo, quindi un metodo a uso interno utilizzerà, per esempio, il modificatore private, mentre uno che dovrà essere utilizzato anche da parte di altri oggetti dovrà essere necessariamente dichiarato come public.

Uno smartphone, per esempio, potrebbe avere un metodo per accenderlo:

```
public class Smartphone
{
public void Accendi()
{
//implementare logica di accensione
}
}
```

I metodi possono accedere agli altri membri della classe. Per esempio il metodo GetDescrizione restituisce, per mezzo dell'istruzione return, un oggetto string formato dalla concatenazione dei campi della classe:

```
public class Smartphone
{
private string marca, modello;

public string GetDescrizione()
{
return marca + " " + modello;
}
}
```

Si noti come all'interno del metodo possiamo utilizzare i campi della classe come se fossero variabili locali; in effetti l'ambito dei campi si estende anche all'interno dei metodi della loro classe.

Nel caso in cui il metodo non debba restituire alcun dato, ma semplicemente eseguire una sequenza di istruzioni, è possibile indicare **void** come tipo di ritorno. Per esempio, potremmo utilizzare un metodo per impostare marca e modello dello Smartphone, senza necessità di avere un valore di ritorno:

```
public void ImpostaDati(string ma, string mod)
{
marca=ma;
modello=mod;
}
```

Chiamata di metodi

Per invocare o chiamare un metodo, cioè utilizzarlo all'interno della stessa classe che lo definisce, o all'interno di altre classi se i modificatori di accesso lo consentono, si utilizza l'operatore di invocazione, costituito dalle parentesi tonde () al cui interno si indicheranno eventuali argomenti separati da virgole.

Per esempio, per chiamare il metodo ImpostaDati all'interno della stessa classe Smartphone possiamo scrivere semplicemente:

```
ImpostaDati("mia marca", "mio modello");
```

Mentre se volessimo utilizzare lo stesso metodo, il che è consentito in quanto public, su un'istanza della classe Smartphone, creata magari nel metodo Main, dovremmo specificare anche il nome dell'istanza e quindi accedere al metodo con l'operatore . in quanto membro della classe Smartphone:

```
static void Main()
{
Smartphone phone=new Smartphone();
phone.ImpostaDati("mia marca", "mio modello");
}
```

Il membro dal quale viene invocato un metodo è in generale detto *chiamante*. Per esempio nel caso precedente il metodo Main è il chiamante del metodo ImpostaDati.

Overload dei metodi

La firma di un metodo, cioè la sequenza composta dal nome del metodo e dai parametri di ingresso, deve essere univoca. Quindi all'interno di una classe potremo avere due o più metodi con uno stesso nome, a patto che i parametri di ingresso siano di numero o tipo differente. Si parla in questo caso di sovraccarico o overload dei metodi. Per esempio, potremmo aggiungere un altro metodo ImpostaDati con un solo parametro, che imposti entrambi i campi allo stesso valore:

```
public void ImpostaDati(string m)
{
marca=m;
```

```
modello=m;
}
```

Si presti attenzione al fatto che la firma non include il tipo di ritorno, quindi non è possibile avere due metodi che si differenziano solo per il tipo restituito:

```
public int ImpostaDati(string m)
{…}

public void ImpostaDati(string m) //errore di compilazione
{…}
```

Valore di ritorno

Come visto nei precedenti esempi, un metodo può restituire un valore al chiamante e quindi è necessario indicare un tipo di ritorno, inserendolo prima del nome del metodo e dopo gli eventuali modificatori:

```
public string GetDescrizione()
{
//corpo del metodo
return marca+" "+modello;
}
```

Se il tipo di ritorno non è **void**, il metodo deve restituire un valore del tipo indicato, prima della fine del metodo, mediante l'istruzione **return** seguita dal valore restituito. La parola chiave return interrompe anche l'esecuzione del metodo.

Un metodo può anche contenere più punti di uscita, cioè più istruzioni return, che per esempio restituiranno valori diversi a seconda del percorso seguito dal codice all'interno del metodo.

Supponiamo che il metodo debba restituire una stringa solo se entrambi i campi marca e modello sono valorizzati, oppure un valore di default nel caso in cui uno dei due campi almeno non sia stato valorizzato:

```
public string GetDescrizione()
{
if(String.IsNullOrEmpty(marca) || String.IsNullOrEmpty(modello))
return "marca o modello non inizializzato";
return marca+" "+modello;
}
```

Il valore restituito da un metodo viene utilizzato nel punto in cui il metodo è stato invocato. Per esempio, la seguente istruzione esegue il metodo GetDescrizione e assegna il valore di ritorno alla variabile desc:

```
string desc=GetDescrizione();
```

La variabile desc quindi conterrà il valore di tipo string restituito dal metodo.

Se un metodo esegue delle istruzioni, ma non deve restituire al chiamante nessun valore, bisogna indicare come tipo di ritorno void:

```
public void Accendi()
{
//non restituisce nessun valore
}
```

In questo caso non è necessario utilizzare l'istruzione return e il metodo terminerà la sua esecuzione quando raggiungerà la parentesi graffa di chiusura. Se si vuol concludere in anticipo il metodo, per esempio perché si sono verificate determinate condizioni, è sempre possibile utilizzare un'istruzione return non seguita da alcun valore:

```
public void Accendi()
{
if(batteriaScarica)
return;
else
{
//istruzioni di accensione
}
}
```

Questa forma dell'istruzione return è consentita solo se il tipo di ritorno del metodo è void.

Parametri

Un metodo può accettare dei *parametri* di ingresso, che rappresentano degli argomenti di input da utilizzare all'interno del metodo stesso.

La lista dei parametri viene indicata fra parentesi, utilizzando il tipo e un nome del parametro, come fossero variabili. Tale elenco viene detto lista dei parametri formali del metodo:

```
class Calcolatore
{
public double Potenza(double numero, int esponente)
{
double risultato = numero;
for (int i = 1; i < esponente; i++)
{
risultato *= numero;
}
return risultato;
}
}
```

Il metodo Potenza dichiara due parametri di ingresso formali, uno di tipo double chiamato numero e l'altro di tipo int chiamato esponente. All'interno del metodo è stata implementata una semplice elevazione del parametro numero all'esponente indicato, mediante un ciclo for.

Il corpo del metodo contiene anche un'altra variabile di tipo double, denominata risultato, utilizzata assieme ai due parametri formali, come se questi fossero normali variabili.

I valori dei parametri vengono assegnati al di fuori del metodo, da parte del codice che invoca il metodo, detto anche codice o metodo chiamante. Per esempio, se vogliamo elevare alla potenza 4 il numero 2, invocheremo il metodo così:

```
Calcolatore calc=new Calcolatore();
double risultato=calc.Potenza(2,4);
```

Le variabili o espressioni utilizzate come parametri da inviare al metodo vengono anche dette *parametri attuali* o *argomenti*. Ogni parametro attuale utilizzato nell'invocazione del metodo deve essere dello stesso tipo o di un tipo compatibile con il parametro formale, indicato nella firma del metodo alla stessa posizione.

All'interno del metodo, quindi, la variabile numero assumerà il valore 2, mentre la variabile esponente sarà inizializzata con il valore 4.

Un metodo che non prende alcun parametro di ingresso sarà dichiarato con una lista di parametri vuota, cioè con due parentesi vuote:

```
public int MetodoSenzaParametri()
{
return 0;
}
```

Passaggio dei parametri

Il comportamento predefinito per passare i parametri a un metodo è quello chiamato *passaggio per valore*. Esso avviene copiando il valore dei parametri e utilizzandoli all'interno del metodo come fossero delle nuove variabili che non influenzano quelle originali. Per esempio, se utilizziamo due variabili per invocare il metodo Potenza della classe Calcolatrice:

```
double numero=2;
int esponente=4;
Calcolatore calc=new Calcolatore();
double risultato=calc.Potenza(numero,esponente);
```

i valori delle variabili numero ed esponente vengono copiati, quindi le variabili originali non vengono influenzate dall'esecuzione del metodo.

Proviamo a implementare un metodo in maniera che modifichi i parametri di ingresso:

```
public void PassByValue(int i)
{
i++;
Console.WriteLine("i={0}", i);
}
```

All'interno del metodo PassByValue il valore della variabile i viene stampato e poi incrementato di 1.

Invochiamo ora il metodo utilizzando una variabile i contenente un valore 1:

```
int i=1;
PassByValue(i); //stampa 2
Console.WriteLine("i={0}", i); //stampa 1
```

Subito dopo l'esecuzione del metodo, viene ristampato il valore della variabile i, che fuori dal metodo ha mantenuto il valore originario pari a 1.

Parametri riferimento

Una seconda modalità di passaggio dei parametri è detta *passaggio per riferimento* e consente di passare al metodo un riferimento al valore originario e non una copia di esso.

Per indicare che un parametro è passato per riferimento a un metodo si utilizza il modificatore **ref**:

```
void PassByRef(ref int i)
{
i++;
Console.WriteLine("i={0}", i);
}
```

La parola chiave ref deve essere utilizzata anche prima dell'argomento utilizzato nell'invocare il metodo e quindi dovrà essere una variabile dichiarata in precedenza:

```
int i=1;
PassByRef(ref i); //stampa 2
Console.WriteLine("i={0}", i); //stampa 2
```

Questa volta il parametro rappresenta un alias della variabile originale, quindi si riferisce alla stessa locazione di memoria. La modifica che avviene all'interno del metodo si riflette pertanto sulla variabile esterna al metodo utilizzata come parametro.

Il modificatore ref contribuisce alla firma del metodo, quindi è possibile avere nella stessa classe due metodi con lo stesso nome e che differiscano solo per il tipo di passaggio dei parametri:

```
void Metodo(int i)
{}

void Metodo(ref int i)
{
}
```

I due metodi sono perfettamente distinguibili perché, se si invoca Metodo senza usare il modificatore ref, verrà utilizzata la prima versione del metodo, mentre, per usare la seconda versione, la variabile i deve essere necessariamente preceduta da ref.

Non è possibile invocare un metodo con passaggio per riferimento usando direttamente un valore come parametro:

```
Metodo(ref 1); //errore, devo utilizzare una variabile
```

Inoltre, la variabile passata per riferimento deve essere necessariamente inizializzata:

```
int i;
Metodo(ref i); //errore, devo utilizzare una variabile inizializzata
```

Ciò implica che una variabile ref all'interno di un metodo è da considerarsi sicuramente inizializzata.

Negli esempi precedenti abbiamo utilizzato il passaggio per riferimenti di variabili di tipo valore. Se il parametro di un metodo è di un tipo di riferimento, il parametro attuale del metodo sarà un oggetto, quindi le modifiche sull'oggetto all'interno del metodo si rifletteranno sull'oggetto che è stato utilizzato come parametro anche all'uscita dal metodo stesso, sia che esso venga passato come parametro valore sia come parametro riferimento.

Supponiamo, per esempio, di avere una semplice classe contenente un campo public di tipo int:

```
class Uomo
{
public int altezza=170;
}
```

Ora implementiamo un metodo che cambi l'altezza dell'oggetto Uomo passato come argomento:

```
void CambiaAltezza(Uomo uomo)
{
Console.WriteLine("prima dell'assegnazione: {0}", uomo.altezza);
uomo.altezza = 180;
}
```

Proviamo ora a stampare il valore dell'altezza prima e dopo l'invocazione del metodo:

```
Uomo uomo = new Uomo();
Console.WriteLine("prima del metodo: {0}", uomo.altezza); //stampa 170
CambiaAltezza(uomo); //cambia altezza in 180
Console.WriteLine("dopo metodo: {0}", uomo.altezza); //stampa 180
```

Come ci si aspettava, il valore del campo altezza dell'istanza uomo, passato come parametro, viene modificato all'interno del metodo.

Esiste però una precisazione da fare: nell'esempio precedente abbiamo usato un campo dell'oggetto passato come argomento, ma non gli abbiamo assegnato un nuovo valore.

Cosa succede se all'interno del metodo assegniamo al parametro di un tipo riferimento un nuovo oggetto? In questo caso entra in gioco la distinzione di passaggio per valore o per riferimento.

Nel primo caso, nel momento in cui al parametro viene assegnato un nuovo oggetto, viene persa la relazione fra il parametro e l'oggetto originale, quindi avremo due oggetti diversi in memoria:

```
static void CambiaAltezza2(Uomo uomo)
{
Console.WriteLine("prima dell'assegnazione: {0}", uomo.altezza);
uomo = new Uomo();
uomo.altezza = 180;
Console.WriteLine("dopo l'assegnazione: {0}", uomo.altezza);
}
```

Nel nuovo metodo CambiaAltezza al parametro uomo viene assegnato un nuovo oggetto Uomo. Quindi la variabile uomo costruita esternamente al metodo non viene influenzata dal cambio del valore effettuato all'interno del metodo:

```
Uomo uomo = new Uomo();
Console.WriteLine("prima del metodo: {0}", uomo.altezza);
CambiaAltezza(uomo);
Console.WriteLine("dopo metodo: {0}", uomo.altezza);
```

Il risultato sarà quindi il seguente:

```
prima del metodo: 170
prima dell'assegnazione: 170
dopo l'assegnazione: 180
dopo metodo: 170
```

Proviamo ora a implementare un'ulteriore nuova versione del metodo CambiaAltezza, modificando solo il passaggio del parametro per riferimento:

```
void CambiaAltezza(ref Uomo uomo)
{
Console.WriteLine("prima dell'assegnazione: {0}", uomo.altezza);
uomo.altezza = 180;
}
```

Eseguendo lo stesso codice e passando ora l'argomento uomo con il modificatore ref:

```
Uomo uomo = new Uomo();
Console.WriteLine("prima del metodo: {0}", uomo.altezza);
CambiaAltezza(ref uomo);
Console.WriteLine("dopo metodo: {0}", uomo.altezza);
```

il risultato sarà che l'oggetto originale, costruito all'esterno del metodo, viene sostituito da un nuovo oggetto, costruito all'interno del metodo:

```
prima del metodo: 170
prima dell'assegnazione: 170
dopo l'assegnazione: 180
dopo metodo: 180
```

NOTA Il passaggio di parametri in C# mediante il modificatore ref o out (vedere il prossimo paragrafo) è molto raro; in genere è possibile ottenere il medesimo risultato ripensando il codice e implementando una soluzione alternativa.

Parametri di uscita

In generale un metodo può avere un solo valore di uscita, il cui tipo è indicato nella firma del metodo, che viene restituito mediante l'istruzione return.

Utilizzando il modificatore **out** è possibile indicare che un parametro di un metodo è un parametro di uscita: il suo valore sarà assegnato all'interno del metodo e quindi potrà essere restituito al chiamante del metodo stesso.

Il concetto di parametro di uscita è strettamente correlato a quello di parametri per riferimento visti nel precedente paragrafo.

Nella libreria di classi di base di .NET, la struct Int32 fornisce un metodo statico per convertire una stringa in un numero intero, ove la stringa rappresenti effettivamente un numero. Tale metodo, denominato TryParse, ha la seguente firma:

```
bool TryParse(string str, out int result)
```

Esso restituisce quindi un valore booleano, a indicare se la conversione è possibile e, in caso positivo, assegna al parametro result, che utilizza un modificatore out, il valore ottenuto dalla conversione del primo parametro string in un numero intero.

Per utilizzare il metodo bisogna passare, in corrispondenza del parametro di uscita, una variabile anche non assegnata in precedenza, preceduta dal modificatore out:

```
int result;
string str="123";
if(int.TryParse(str, out result))
{
Console.WriteLine(result); //la variabile result ha valore 123
}
else Console.WriteLine("la stringa non rappresenta un numero intero");
```

Per implementare un metodo con uno o più parametri di uscita, basta assegnare il valore di tutti loro prima della terminazione del metodo stesso:

```
void PotenzeMultiple(int val, out int potenza2, out int potenza3)
{
potenza2 = val * val;
potenza3 = potenza2 * val;
}
```

I due parametri potenza2 e potenza3, alla fine del metodo, conterranno la potenza del 2 e del 3 del primo parametro int.

Per utilizzare tale metodo bisogna necessariamente indicare il modificatore out in tutti i parametri che lo prevedono:

```
int i=2;
int p2,p3;
PotenzeMultiple(i, out p2, out p3);
```

Internamente i metodi con parametri out sono implementati come quelli ref. L'unica differenza si nota nell'utilizzo, in quanto nel caso di parametri di uscita possono anche essere utilizzate delle variabili non inizializzate, come nell'esempio precedente, perché in ogni caso esse dovranno essere assegnate all'interno del metodo.

NOTA	Come già detto per i parametri ref, anche i parametri di uscita out sono raramente utilizzati, tranne nel caso di metodi di parsing come quello TryParse visto nei precedenti esempi. Se vi trovaste a implementare un metodo con più di uno o due parametri di uscita, vi consiglio di pensare a una soluzione diversa, magari creando un'apposita classe con più campi e passando al metodo un singolo oggetto di tale classe. Un'altra soluzione è l'utilizzo della classe System.Tuple, introdotta con .NET 4.0, che vedremo nel Capitolo 9 dedicato alle collezioni.

Array di parametri

Molti metodi possono richiedere il passaggio di un numero di parametri non determinato. Fra questi abbiamo più volte utilizzato il metodo Console.WriteLine, a cui è possibile inviare un numero di parametri indefinito semplicemente separandoli con la virgola, subito dopo il primo parametro rappresentante la stringa di formato:

```
Console.WriteLine("{0} {1} {2}", 1,2,3);
```

In questo caso il metodo accetta tre valori interi.

Per utilizzare un array di parametri bisogna utilizzare il modificatore **params**, come ultimo parametro del metodo, seguito dalla dichiarazione di un array. La presenza del modificatore params indica che possiamo invocare il metodo con un numero qualsiasi di valori separati da virgola (anche nessuno: in questo caso l'array sarà vuoto) oppure direttamente con un array dello stesso tipo:

```
double CalcolaMedia(params double[] array)
{
double media = 0;
for (int i = 0; i < array.Length; i++)
{
media += array[i];
}
return media / array.Length;
}
```

Il metodo CalcolaMedia può quindi essere utilizzato indicando un numero qualunque di valori di tipo double:

```
CalcolaMedia(1.2, 2.0, 3, 4.3);
CalcolaMedia(1, 2 ,3.1 ,4.2, 5, 6);
```

oppure direttamente con un array di double:

```
int[] array=new double[] {1,2,3,4};
CalcolaMedia(array);
```

Naturalmente, la prima modalità è molto più comoda e immediata, ma l'effetto è identico. Nel primo caso è il compilatore che crea per noi un array di valori a partire dai vari parametri separati da virgola.

Parametri con nome

Finora abbiamo utilizzato i parametri in maniera posizionale, cioè ci siamo affidati esclusivamente alla posizione per invocare un metodo con i parametri appropriati: ogni parametro attuale corrisponde al parametro formale indicato nella firma del metodo alla stessa posizione. Per esempio, se un metodo ha la firma seguente:

```
void MioMetodo(int a, int b, int c);
```

quando lo si invoca con:

```
MioMetodo(1,2,3);
```

i valori 1, 2, 3, corrisponderanno rispettivamente ai parametri formali a, b e c.

Una seconda modalità, introdotta con C# 4.0 e che non si basa sulla posizione, è invece quella dei cosiddetti *parametri con nome* o *denominati*, che consente di specificare anche in fase di invocazione del metodo il nome del parametro che si vuol inviare al metodo.

Per quanto riguarda l'implementazione del metodo non cambia nulla, dato che i parametri hanno già un identificatore. In fase di invocazione, invece, basta far precedere il valore utilizzato come argomento dal nome del parametro, seguito dai due punti. Per esempio, possiamo riscrivere la precedente istruzione così:

```
MioMetodo(c:3, b:2, a:1);
```

È possibile utilizzare anche contemporaneamente parametri posizionali e parametri con nome, a patto che i parametri posizionali vengano indicati per primi:

```
MioMetodo(1, 2, c:3);
```

In questo caso, il primo valore sarà utilizzato come argomento del parametro a, il secondo valore come parametro b, mentre il terzo è esplicitamente nominato come c.

Il vantaggio principale dei parametri con nome è quello di chiarire a quale parametro corrisponde un determinato argomento, rendendo così il codice più leggibile e auto-documentato.

Parametri opzionali

A partire dalla versione 4.0, C# permette di definire dei valori predefiniti per i parametri, rendendo quindi opzionale il passaggio di tali parametri nell'invocazione di un metodo. In altre parole, definendo i valori predefiniti che i parametri assumeranno si può evitare di passare tutti gli argomenti di un metodo.

Per assegnare i valori predefiniti basta indicarli direttamente nella firma del metodo, con una semplice assegnazione:

```
public void MioMetodo(int a=0, int b=1)
{}
```

In questo caso il metodo MioMetodo ha due parametri, di tipo int, con valori predefiniti pari a 0 e 1.

Quindi si può invocare il metodo sia indicando dei parametri personalizzati, sia lasciando i valori predefiniti:

```
MioMetodo(1,2);
MioMetodo(); //utilizza i valori 0 e 1
```

Non tutti i tipi di parametri possono essere utilizzati come parametri opzionali. In particolare, per i tipi riferimento, essi possono essere opzionali solo utilizzando null come valore predefinito. Inoltre i parametri possono essere opzionali solo se non utilizzano i modificatori out e ref.

Per quanto riguarda la modalità di scrittura della firma di un metodo, i parametri opzionali devono essere posizionati tutti quanti dopo i parametri obbligatori ed eventualmente devono precedere gli array params visti nel precedente paragrafo.

Esistono anche regole di assegnazione dei valori ai rispettivi parametri opzionali. Per evitare ambiguità, infatti, i parametri opzionali possono essere omessi a partire dall'ultimo di essi.

Per esempio, se invochiamo il precedente metodo MioMetodo che ha due parametri opzionali a e b, esplicitandone solo uno, il parametro omesso sarà il secondo, cioè il parametro b:

```
MioMetodo(1); //il parametro a è pari a 1, il parametro b utilizza il valore di default 1
```

Se si vogliono invece omettere determinati parametri, è possibile esplicitare i nomi dei parametri come visto nel precedente paragrafo; per esempio se vogliamo invocare il metodo

MioMetodo utilizzando il valore di default di a e indicando esplicitamente quello di b, possiamo scrivere:

```
MioMetodo(b:1); //il parametro a viene omesso e utilizzerà il valore predefinito 0
```

Espressioni corpo

Capita spesso di avere dei metodi che restituiscono semplicemente un valore, oppure metodi composti da una singola istruzione. A partire da C# 6 è possibile utilizzare in tal caso una sintassi abbreviata, indicando l'espressione che costituisce il corpo del metodo subito dopo l'operatore *lambda* =>:

```
public int Somma(int a, int b) => a+b;
public Stampa(string str) => Console.WriteLine(str);
```

Questa modalità, chiamata anche *Expression-Bodied Members*, può essere utilizzata, come vedremo più avanti nel capitolo, anche per le proprietà e gli indicizzatori. Le espressioni lambda saranno invece trattate nel Capitolo 10.

Purtroppo ci sono anche delle limitazioni, quindi non tutti i membri sono sostituibili con espressioni corpo: non può esserci naturalmente un blocco di più istruzioni (se fosse necessario, basterebbe utilizzare un metodo normale!), e di conseguenza non si possono utilizzare istruzioni un po' più complesse come if e switch, ma in molti casi basteranno l'operatore ternario, il null coalescing o il null conditional.

Ricorsione

All'interno di un metodo è possibile invocare uno o più metodi differenti, con un livello di profondità qualunque.

Per esempio, supponiamo di avere una classe MiaClasse con i tre metodi seguenti, cioè A, B e C:

```
class MiaClasse
{
A()
{
Console.WriteLine("A");
B();
}

B()
{
Console.WriteLine("B");
C();
}

C()
{
Console.WriteLine("C");
}
}
```

All'interno del metodo A viene invocato il metodo B e all'interno di questo viene invocato C.

Sappiamo ormai che al termine dell'esecuzione di un metodo viene restituito il controllo al chiamante, nel punto immediatamente successivo all'invocazione del metodo. Per esempio, quando termina l'esecuzione del metodo C, il controllo torna all'interno del metodo B, che termina a sua volta e restituisce il controllo al metodo A.

Un metodo può anche invocare se stesso come metodo da eseguire. In questo caso si parla di *ricorsione*. La ricorsione può essere utilizzata per risolvere particolari problemi in maniera molto efficiente ed elegante. Il classico esempio di ricorsione è il calcolo del fattoriale di un numero. Il fattoriale di un numero è definito come segue:

- se n è un numero intero >1, il suo fattoriale è dato dal valore n*(n-1)*(n-2) * … * 1;
- se n è pari a 0 oppure a 1, il fattoriale di n è pari al valore di n.

Per implementare tale calcolo, quindi, bisogna moltiplicare il numero n per il numero n decrementato di 1 fino a quando il valore decrementato è pari a 1.

Quindi, utilizzando la ricorsione, possiamo scrivere il seguente metodo di calcolo:

```
int Fattoriale(int n)
{
if(n<=1)
return n;
return n*Fattoriale(n-1); //moltiplica n per il risultato dello stesso metodo Fattoriale di n-1
}
```

Il meccanismo è identico a quello descritto per l'invocazione di un altro metodo. Al termine dell'esecuzione del metodo, il risultato viene restituito al chiamante.

Costruttori

Un particolare tipo di metodo, eseguito per allocare la memoria necessaria e inizializzare un'istanza di una classe, è detto *costruttore*.

A differenza dei metodi non si indica alcun tipo di ritorno e il nome del costruttore deve coincidere con il nome della classe.

Per invocare un costruttore si deve utilizzare la parola chiave new. Infatti abbiamo già avuto modo di istanziare un oggetto semplicemente scrivendo qualcosa del genere:

```
Smartphone mioTel=new Smartphone();
```

In questo caso abbiamo utilizzato un costruttore che non accetta parametri in ingresso.

Sebbene non sia stato esplicitamente implementato alcun costruttore per la classe Smartphone, è stato possibile costruirne un'istanza. Ogni classe, infatti, fornisce un *costruttore predefinito* senza parametri, detto anche *costruttore di default*, che si può ridefinire, se necessario,

includendo al suo interno le istruzioni che si vogliono eseguire in fase di creazione di un'istanza della classe.

NOTA	Non è necessario occuparsi dell'allocazione della memoria necessaria a contenere l'istanza, è il Common Language Runtime che se ne occupa per conto nostro quando viene invocato il costruttore della classe mediante l'operatore new.

Come visto nel paragrafo sui campi, istanziando un oggetto i campi vengono inizializzati ai valori predefiniti dei rispettivi tipi.

Se fosse necessario, invece, fornire dei valori di inizializzazione diversi o eseguire altre operazioni quando si costruisce un oggetto, bisognerebbe implementare uno o più costruttori personalizzati.

Per esempio, se vogliamo ridefinire il comportamento del costruttore predefinito, basta dichiararlo e implementarlo esplicitamente all'interno della classe:

```
class Smartphone
{
private string marca;
private string modello;
public Smartphone()
{
marca="senza marca";
modello="non inizializzato"; }
}
```

In questo caso, il costruttore inizializza il valore dei campi marca e modello con due stringhe diverse.

Si noti che il costruttore utilizza in questo caso il modificatore public, per permettere la creazione di un oggetto della classe da un qualunque punto del nostro programma.

Nulla vieta di usare un modificatore diverso, proprio per controllare chi e come può istanziare un oggetto. Spesso si utilizza un costruttore privato per evitare che la classe venga istanziata esplicitamente e consentire magari a un'altra classe apposita di occuparsi della costruzione di tali oggetti (per esempio per controllare il numero di oggetti istanziabili, consentendo l'esistenza di una sola istanza):

```
class Smartphone
{
private string marca;
private string modello;
private Smartphone()
{
marca="senza marca";
```

```
modello="non inizializzato"; }
}
```

NOTA	Il pattern di programmazione che ha lo scopo di garantire che di una classe sia possibile la creazione di una e una sola istanza, fornendo un punto di accesso globale a tale istanza, è detto *singleton*. La classe precedente non è un esempio completo di singleton, in quanto manca del metodo di creazione dell'istanza e di un punto di accesso a essa.

Se fosse necessario passare dei parametri in fase di creazione di un oggetto, è possibile definire uno o più costruttori con diversi argomenti (analogamente ai metodi, anche per i costruttori è possibile l'overload). Per esempio, potremmo voler istanziare un oggetto Smartphone indicando al costruttore i valori con cui inizializzare i campi privati marca e modello:

```
class Smartphone
{
private string marca;
private string modello;

public Smartphone(string m, string mod)
{
marca=m;
modello=mod; }
}
```

Tornando a parlare di costruttore predefinito e senza parametri, si noti che esso viene fornito dal compilatore se e solo se non abbiamo implementato altri costruttori personalizzati.

Nell'implementazione precedente della classe Smartphone, che ha un costruttore personalizzato con due parametri, il costruttore di default non viene generato, quindi se si desidera mantenere la possibilità di invocare un costruttore senza parametri bisogna esplicitamente implementarlo:

```
class Smartphone
{
private string marca;
private string modello;

public Smartphone()
{
}

public Smartphone(string m, string mod)
{
marca=m;
modello=mod; }
}
```

Il costruttore di default, in questo caso, non esegue alcuna operazione particolare, oltre a quella di inizializzare i campi con i valori predefiniti.

Per evitare l'inutile duplicazione di codice, è possibile dall'interno di un costruttore invocarne un altro mediante l'utilizzo della parola chiave this, nel seguente modo:

```
public Smartphone(string m, string mod)
{
marca=m;
modello=mod; }

public Smartphone(string m): this(m,m)
{}

public Smartphone() : this("non inizializzato")
{
//altre inizializzazioni
}
```

Il secondo costruttore, per esempio, invoca il costruttore con due parametri, passando a esso la stessa stringa. Il terzo invece invoca il secondo (che a sua volta invocherà il primo) passando una stringa "non inizializzato" che sarà utilizzata per valorizzare i due campi.

Abbiamo detto in precedenza che eseguendo il costruttore i campi vengono inizializzati ai loro valori predefiniti.

L'ordine di inizializzazione dipende dall'ordine in cui essi appaiono all'interno della classe e i campi sono inizializzati prima dell'esecuzione del resto di istruzioni presenti eventualmente nei costruttori:

```
class Valori
{
int primo=1; //assegnato per primo
int secondo=2; //assegnato per secondo
int terzo; //assegnato per terzo il valore predefinito 0

public Valori()
{
//i campi sono già assegnati
terzo=primo+secondo;
}}
```

Finalizzatori o distruttori

Complementare a quello dei costruttori è il concetto dei *finalizzatori* o *distruttori*, che sono dei metodi da eseguire quando un oggetto non è più utilizzato, per esempio perché si esce dal suo ambito di utilizzo e quindi la sua memoria può essere liberata.

Scrivendo un finalizzatore, cosa che non accade molto frequentemente in C# perché è il CLR che si occupa della gestione della memoria (vedere il prossimo paragrafo), si possono eseguire delle azioni personalizzate di pulizia, in particolare quando è necessario gestire la rimozione di oggetti non gestiti dal CLR.

Per implementare un finalizzatore si usa una sintassi simile a quella dei costruttori, facendo precedere però al nome della classe il carattere ~:

```
~ class MiaClasse
{
~MiaClasse()
{
// logica del finalizzatore
}
}
```

Un finalizzatore, a differenza dei costruttori, non può essere dichiarato public né static, inoltre non può avere parametri (perché vengono eseguiti automaticamente dal CLR) e non può invocare la classe madre.

Si ricordi inoltre che non è predicibile l'istante in cui il finalizzatore verrà invocato dal CLR, quindi è bene non fare affidamento su un particolare momento o ordine di esecuzione.

La presenza di un finalizzatore rallenta l'esecuzione della procedura di Garbage Collection (ossia della pulizia di oggetti non più referenziati), in quanto gli oggetti senza finalizzatori vengono eliminati direttamente dalla memoria, mentre quelli con finalizzatore necessiteranno di un ulteriore passaggio.

Garbage Collection

Ogni oggetto istanziato in C# viene allocato e quindi utilizza una parte di memoria.

Il Common Language Runtime ha un servizio chiamato *Garbage Collector* (GC) che si occupa di gestire automaticamente la memoria occupata dagli oggetti istanziati.

Il Garbage Collector permette allo sviluppatore di non doversi preoccupare di tale gestione, cioè di doverla allocare al momento della creazione, né, al contrario, di doverla liberare quando un oggetto non è più utilizzato.

Il funzionamento del Garbage Collector si basa su un algoritmo che verifica quali oggetti non sono più referenziati da altri oggetti e quindi possono essere considerati inutilizzati. Per esempio, supponiamo di avere un metodo al cui interno viene creato e utilizzato un oggetto della classe Smartphone:

```
public void CreateVehicle()
{
Smartphone mioTel=new Smartphone();
//utilizzo dell'istanza mioTel
...
}
```

Quando si esegue il metodo, un'istanza della classe viene allocata in memoria heap.

Al termine del metodo, l'istanza esce dal suo scope, quindi essa non è più utilizzata e non vi è più alcun riferimento a essa. La memoria occupata dall'oggetto può a questo punto essere liberata e riutilizzata.

La procedura di *Garbage Collection* non è deterministica, cioè non avviene in istanti certi, perché altrimenti diverrebbe troppo pesante e controproducente, in quanto l'applicazione viene totalmente bloccata durante il suo intervento.

Essa invece viene lanciata periodicamente dal CLR, basandosi su una serie di fattori, per esempio sulla quantità libera di memoria, su quella necessaria per istanziare e allocare un nuovo oggetto. Quando parte il processo di GC, viene costruito una sorta di grafo degli oggetti in memoria, partendo dai riferimenti radice e procedendo lungo tutti gli oggetti creati a partire da essi. Alla fine di questo processo si ha un elenco degli oggetti non più referenziati che saranno sottoposti alla Garbage Collection vero e proprio.

Il Garbage Collector e il suo algoritmo hanno diverse ottimizzazioni che vanno al di là dello scopo di questo libro, per cui, se siete interessati all'argomento, si consiglia di approfondire su testi o siti specializzati.

Per il momento vi basti sapere che gli oggetti non più referenziati non vengono rimossi immediatamente dalla memoria, perché in genere in brevi lassi di tempo vengono create molte istanze di breve utilizzo. In tal modo, se il processo di Garbage Collection dovesse intervenire in ognuna di tali occasioni, le prestazioni peggiorerebbero sensibilmente. Le istanze vengono allora suddivise in diverse generazioni e più esattamente in tre: *Gen0*, che rappresenta la generazione degli oggetti appena creati, *Gen1*, che sono gli oggetti che hanno superato una prima procedura di GC, e *Gen2*, che contiene tutti gli altri.

In tal modo il Garbage Collector può riconoscere gli oggetti vecchi, che presumibilmente continueranno a essere utilizzati anche in futuro, da quelli nuovi, di cui la maggior parte è invece di tipo usa e getta (come le variabili locali).

Forzare il Garbage Collector

Sebbene, come detto nel precedente paragrafo, il Garbage Collector di .NET sia un meccanismo automatico controllato dal CLR, in alcuni casi può essere utile controllarne il comportamento.

Il metodo System.GC.Collect può essere invocato per forzare l'avvio del GC, ma per farlo devono esserci buoni motivi, in maniera da non peggiorare la situazione bloccando il programma.

Lo sviluppatore deve essere perfettamente a conoscenza del comportamento della propria applicazione, prima di usare tale metodo e, in ogni caso, non esagerare con il suo utilizzo.

Un esempio in cui potrebbe essere il caso di usare GC.Collect, è quello in cui si sono dovuti creare un sacco di nuovi oggetti e si sa che, al termine del loro utilizzo, potrà essere liberata una buona quantità di memoria.

Il riferimento this

Per riferirsi esplicitamente all'istanza corrente, all'interno di un oggetto, è possibile utilizzare la parola chiave **this**.

Abbiamo utilizzato this per invocare una particolare versione di un costruttore nella dichiarazione di un altro costruttore.

In generale this viene omesso, perché, per esempio, quando si utilizza un campo o un metodo è implicitamente sottinteso che ci si riferisca a un campo o metodo della stessa classe (d'altronde se non fosse così si avrebbe un errore di compilazione, visto che i campi e i metodi di altre classi non sarebbero visibili e non esistono variabili globali). Le seguenti versioni del costruttore di default sono quindi perfettamente equivalenti:

```
public Smartphone(string m, string mod)
{
marca=m;
modello=mod; }
```

```
public Smartphone(string m, string mod)
{
this.marca=m;
this.modello=mod; }
```

Un'altra possibilità di utilizzo è invece quella che ci permette di risolvere le ambiguità che possono verificarsi fra parametri e campi. Per esempio, è possibile chiamare i parametri del costruttore con gli stessi nomi dei campi:

```
public Smartphone(string marca, string modello)
{
this.marca=marca;
this.modello=modello; }
```

In tal modo è più facile assegnare i nomi dei parametri e utilizzarli per inizializzare i campi. L'identificatore a sinistra dell'operatore di assegnazione si riferisce ai campi dell'istanza, mentre quello a destra si riferisce al parametro del costruttore.

Infine, vediamo come utilizzare la parola chiave this per passare l'istanza corrente a un altro metodo o proprietà. Supponiamo che la classe Sim possegga un campo public di tipo Smartphone, a cui vogliamo assegnare il telefono che la contiene; potremmo in tal caso scrivere:

```
class Sim
{
public Smartphone phone;
}

class Smartphone
{
private Sim sim;
public Smartphone()
{
sim=new Sim();
sim.Phone=this;
}
}
```

Nel costruttore della classe Smartphone viene inizializzata un'istanza della classe Sim, al cui campo Phone viene assegnata l'istanza corrente di Smartphone.

Il riferimento this non può essere utilizzato all'interno di membri statici, perché essi non fanno parte di alcuna istanza.

In C# this si utilizza quindi in almeno quattro diverse occasioni. Oltre a quello visto in questo paragrafo, per riferirsi all'istanza corrente del metodo, lo abbiamo già utilizzato per concatenare l'esecuzione di diversi costruttori della stessa classe, e lo vedremo in seguito come modificatore per creare indicizzatori e metodi di estensione.

Membri statici

Come visto in precedenza, i membri di una classe sono implicitamente associati a una particolare istanza di una classe. Per esempio, un campo a cui è associato un dato valore appartiene a uno specifico oggetto, mentre un secondo oggetto della stessa classe potrà mantenere nello stesso campo un valore diverso.

Se costruiamo due diversi oggetti Smartphone, a ognuno di essi potremo associare un particolare modello e marca:

```
Smartphone tel1=new Smartphone("soni", "t28")
Smartphone tel2=new Smartphone(smasung", "4s");
```

In tal modo ogni oggetto potrà avere un suo stato e sarà possibile interagire con tale particolare istanza mediante altri membri della classe.

Ci sono casi in cui, invece, una classe deve condividere dei membri fra tutte le sue istanze. La dichiarazione di un membro statico o di classe avviene mediante l'utilizzo del modificatore static. Un tipico esempio è quello di un campo di tipo intero che mantenga il numero di oggetti costruiti:

```
class Smartphone
{
```

```
public static int Contatore;
Smartphone()
{
Contatore++;
}
}
```

In questo modo ogni volta che si costruisce un'istanza della classe, il campo statico Contatore verrà incrementato di uno. Tale valore è condiviso fra tutte le istanze, quindi per accedere a esso non è necessario utilizzare l'operatore . su una particolare istanza, ma bisognerà utilizzare il nome della classe:

```
Smartphone tel1=new Smartphone()
Console.WriteLine(Smartphone.Contatore); //stampa 1
Smartphone tel2=new Smartphone();
Console.WriteLine(Smartphone.Contatore); //stampa 2
```

Il modificatore static può essere utilizzato anche per metodi e costruttori. Analogamente ai campi, i metodi saranno così associati alla classe e non a una particolare istanza. Quindi anch'essi dovranno essere invocati utilizzando il nome della classe.

Inoltre, non essendo appunto collegati a un'istanza, non possono interagire con lo stato di un oggetto; ciò significa che al loro interno potranno essere utilizzati solo altri membri statici.

Per esempio, potremmo implementare un metodo di reset del contatore precedente nel seguente modo:

```
public static void ResetCounter()
{
Contatore=0;
//marca=null; //errore, non è un campo statico
}
```

In questo caso non è stato necessario nemmeno utilizzare il nome della classe, perché il metodo ResetCounter appartiene alla stessa classe che definisce il campo Contatore. Se però volessimo resettare il contatore da un'altra classe, dovremmo invocarlo indicando esplicitamente il nome della classe:

```
Smartphone.ResetCounter();
Console.WriteLine(Smartphone.Contatore); //stampa 0
```

Costruttori statici

Anche un costruttore può essere dichiarato come statico.

Mentre un costruttore di istanza inizializza un particolare oggetto di una determinata classe, un costruttore statico effettua inizializzazioni a livello di classe. Un tipico utilizzo dei costruttori statici è quello di inizializzare i campi statici di una classe.

La dichiarazione di un *costruttore statico* è identica a quella di un costruttore di istanza, nel senso che il suo nome deve coincidere con il nome della classe e non deve indicare alcun tipo di ritorno.

Le differenze sono le seguenti: può esistere al massimo un costruttore statico, a esso non può essere applicato alcun modificatore di accesso, non può essere utilizzato alcun parametro e si utilizza naturalmente il modificatore static:

```
static Smartphone()
{
Contatore=0;}
```

Il costruttore statico viene invocato automaticamente dal compilatore (ecco perché non si possono utilizzare modificatori di accesso e parametri) subito prima della costruzione della prima istanza della classe oppure appena si tenta di accedere per la prima volta a un membro statico della classe.

Le proprietà

Le *proprietà* permettono di implementare il meccanismo di incapsulamento, controllando le modalità di accesso ai campi privati di una classe.

La sintassi principale prevede la dichiarazione del modificatore, del tipo e del nome della proprietà, mentre all'interno del suo blocco possono essere indicati due diversi ambiti di accesso, uno eseguito in lettura, implementato con un blocco **get**, e uno eseguito in scrittura, implementato mediante un blocco **set**:

```
class Smartphone
{
private string modello;
public string Modello
{
get
{
return modello;
}
set
{
modello=value;
}
}
}
```

All'interno del blocco set è possibile utilizzare la parola chiave value, che contiene il valore assegnato alla proprietà. Per leggere e scrivere una proprietà, infatti, si utilizza l'operatore = di assegnazione:

```
Smartphone phone=new Smartphone();
```

```
phone.Modello="abc123";
```

Assegnando un valore abc123 alla proprietà, all'interno del blocco set, la parola chiave value conterrà tale valore. Al contrario, per utilizzare il valore di una proprietà, basta accedere a essa mediante l'operatore . come per un qualsiasi membro:

```
Console.WriteLine(phone.Modello);
```

Il risultato potrebbe sembrare molto simile a quello che si otterrebbe creando dei campi con modificatore public.

La differenza è che mediante una proprietà è possibile eseguire del codice per controllare come ottenere dei dati privati della classe o per controllare l'impostazione di tali dati, senza dare un accesso diretto a essi.

Per esempio, se un campo numerico deve essere limitato in maniera da assumere solo valori appartenenti a un particolare intervallo, oppure se per una stringa devono essere consentiti solo dei valori alfanumerici ma non caratteri speciali, o, ancora, se si vuol controllare la lunghezza della stringa inserita, è possibile, utilizzando una proprietà, verificare ed eventualmente correggere i dati che si vogliono assegnare a un campo, scrivendo dell'apposito codice nel ramo set.

Il seguente esempio mostra come assegnare una stringa a un campo, solo se il valore passato alla proprietà ha una lunghezza minore di 10 caratteri:

```
public string Modello
{
get
{
return modello;
}
set
{
if(value.Length<10)
modello=value;
}
}
```

Inoltre non è necessario avere un campo da restituire mediante una proprietà. Essa potrebbe, per esempio, restituire un valore basato su un calcolo o su una combinazione di altri campi della classe.

Per esempio, per ottenere una concatenazione di modello e marca dello Smartphone, anziché implementare un metodo, potremmo scrivere una proprietà come la seguente:

```
public string Descrizione
{
get
```

```
{
return marca+" "+modello;
}
}
```

Come si vede nel caso precedente, non è obbligatorio implementare entrambi i blocchi get e set. La proprietà Descrizione ha il solo blocco get, quindi è una proprietà a sola lettura. Per esempio, se vogliamo creare una *proprietà di sola lettura* che incapsula un campo private, accessibile dall'esterno, possiamo evitare di implementare il blocco set:

```
public string Modello
{
get
{
return modello;
}
}
```

In questo modo non è possibile assegnare un valore alla proprietà e quindi al campo.

Al contrario, implementando solo il blocco set, avremmo una *proprietà di sola scrittura*, anche se questa possibilità è poco utilizzata.

Come comportamento predefinito le funzioni di accesso get e set hanno lo stesso livello di visibilità associato alla proprietà. Quindi, se la proprietà è, per esempio, public, lo saranno anche le funzioni get e set.

Esiste però la possibilità di impostare un livello di accessibilità differente per una delle due funzioni. Nel seguente esempio la proprietà è public, ma il metodo di accesso set è impostato come private:

```
public string Modello
{
get
{
return modello;
}
private set
{
modello=value;
}
}
```

Il livello di accesso impostato per la funzione di accesso deve essere più restrittivo di quello impostato per la proprietà, cioè se per esempio la proprietà è protected, non è possibile impostare una funzione di accesso come public.

Proprietà automatiche

L'utilizzo più comune di proprietà in una classe C# rimane sempre quello di incapsulare dei campi private, al fine di permetterne la lettura e la scrittura da parte di classi differenti. Tale eventualità è così comune che in C# 3.0 sono state introdotte le proprietà automatiche, che evitano di dover aggiungere a una classe dei campi privati e poi scrivere il codice per accedere a essi mediante una proprietà.

Le proprietà automatiche permettono di implementare una proprietà in maniera concisa nel seguente modo:

```
public string Marca {get;set;}
```

Con le proprietà automatiche non vi è più necessità di dover aggiungere il campo sottostante con accesso private, da leggere e scrivere con una proprietà, in quanto sarà il compilatore a generarlo automaticamente dietro le quinte.

NOTA	In Visual Studio è possibile implementare rapidamente una proprietà automatica, utilizzando lo snippet di codice prop, seguito dalla pressione del tasto TAB, il che permetterà a questo punto di assegnare il nome alla proprietà, per la quale ci penserà l'IDE a implementare i due rami get e set.

Con C# 6, è possibile inizializzare le proprietà automatiche direttamente nella loro definizione, in maniera simile ai campi, utilizzando una semplice assegnazione:

```
public string Marca {get;set;} = "abc";
```

L'inizializzazione avviene dietro le quinte, agendo direttamente su un campo nascosto e non utilizzando i rami set. Questo rende possibile anche l'inizializzazione di proprietà con modificatori differenziati per i rami get e set:

```
public string Marca {get; private set;} = "abc";
```

Fino a C# 5 per le proprietà automatiche era necessario indicare entrambi i rami get e set. Con C# 6 è invece possibile anche dichiarare proprietà automatiche di sola lettura, cioè con il solo ramo get, in quanto esse sono inizializzabili come visto sopra:

```
public string Marca {get; } = "abc";
```

Oppure all'interno di un costruttore:

```
public string Marca {get; }

public Smartphone(string marca)
{
Marca = marca;
}
```

Espressioni corpo

Un'altra possibilità, introdotta da C# 6, per inizializzare le proprietà è quella di scrivere delle espressioni, come già visto per i metodi, utilizzando l'operatore lambda. In tal caso, non è possibile indicare il ramo get, in quanto implicitamente si tratta di proprietà a sola lettura.

La seguente classe definisce la proprietà NomeCompleto, il cui valore è definito da un'espressione che concatena due campi privati:

```
public class Cliente
{
private string nome;
private string cognome;

public string NomeCompleto => nome + " " + cognome;
public Cliente(string n, string c)
{
nome = n;
cognome = c;
}
}
```

Mediante l'uso di un'espressione corpo è così possibile evitare la scrittura di un blocco di codice get e di un'espressione return.

Proprietà statiche

Come per gli altri membri è possibile utilizzare la parola chiave **static** per definire delle proprietà statiche. Esse sono particolarmente utilizzate quando servono a incapsulare dei campi privati della classe a loro volta statici. D'altronde non è possibile accedere a membri di istanza all'interno di una proprietà static:

```
private static int campo;
public static int MyProp
{
get
{
return campo;
}
}
```

Overload degli operatori

Fra i membri definibili per un dato tipo, fra i più potenti e versatili vi sono gli operatori.

Abbiamo già visto nel Capitolo 4 che cosa e quali sono gli operatori utilizzabili in C#, accennando anche al fatto che essi sono in qualche modo personalizzabili, per i tipi definiti dallo sviluppatore, mediante un meccanismo detto di *overload* o *sovraccarico* degli operatori.

Potrebbe essere utile utilizzare operatori standard come +, -, * e così via con istanze di tipi personalizzati ed evitare quindi di scrivere metodi appositi per eseguire tali operazioni. Per esempio, supponiamo di aver implementato una classe ComplexNumber, che rappresenta un

numero complesso: con l'espressione numero complesso si intende un numero formato da una parte immaginaria e da una parte reale; può essere perciò rappresentato dalla somma di un numero reale e di un numero immaginario, per esempio 2+3i):

```
class ComplexNumber
{
private int real;
private int imaginary;

public ComplexNumber(int r, int i)
{
real=r;
imaginary=i;
}

public override string ToString()
{
if (imaginary > 0)
return String.Format("{0}+{1}i", this.real, this.imaginary);
else return String.Format("{0}-{1}i", this.real, -this.imaginary);
}
}
```

NOTA	La classe ComplexNumber contiene l'implementazione di un metodo ToString con il modificatore override. Nel prossimo capitolo vedremo che tale modificatore consente di ridefinire un metodo ereditato da una classe base, in questo caso da System.Object.

Sui numeri complessi, è possibile eseguire le quattro normali operazioni aritmetiche, che però hanno una loro definizione particolare. Per esempio, la somma fra due numeri complessi Z_1 e Z_2 è definita come segue:

$Z_1 = a+ib$
$Z_2 = c+id$
$Z_1 + Z_2 = (a+c) + i(b+d)$

Quindi, per eseguire la somma fra due istanze della classe ComplexNumber potremmo implementare un metodo Add:

```
public ComplexNumber Add(ComplexNumber z2)
{
return new ComplexNumber(this.real + z2.real, this.imaginary + z2.imaginary);
}
```

Per sommare due istanze, z1 e z2, e ottenere un altro oggetto che rappresenti la somma, bisognerebbe utilizzare il metodo Add nel seguente modo:

```
ComplexNumber somma=z1.Add(z2);
```

L'overload o sovraccarico degli operatori consentirebbe invece di scrivere l'operazione di somma utilizzando il naturale operatore +, cioè così:

```
ComplexNumber somma=z1 + z2;
```

Per consentire questa sintassi è necessario fornire, alla classe ComplexNumber, un'implementazione speciale dell'operatore + o degli altri operatori che si vogliono utilizzare con istanze della classe, che altro non è che l'implementazione di un overload di un metodo statico:

```
public static ComplexNumber operator + (ComplexNumber z1, ComplexNumber z2)
{
return new ComplexNumber(z1.real + z2.real, z1.imaginary + z2.imaginary);
}
```

La particolarità di questo metodo consiste nell'aggiunta della parola chiave operator, che indica al compilatore che si sta definendo appunto l'overload di un operatore ed esattamente dell'operatore che segue tale parola chiave (nell'esempio precedente l'operatore +). Il tipo di ritorno dipende naturalmente dal valore restituito dall'operatore; nel caso della somma fra numeri complessi, esso è ancora un numero complesso, ma ciò non è necessario (per esempio più avanti vedremo l'overload degli operatori logici di confronto che restituiscono un bool).

I parametri del metodo di overload dell'operatore sono gli oggetti su cui l'operatore agisce e, nel caso degli operatori binari (come +), essi rappresentano l'operando a sinistra e a destra dell'operatore.

In C# è obbligatorio definire tale metodo con i modificatori public e static, il che significa che l'operatore potrà essere utilizzato ovunque e che non è associato a una particolare istanza, ma alla classe in cui il metodo è definito.

Uno stesso operatore può agire su diversi tipi di dati, per esempio un numero complesso può essere addizionato a un intero, il cui valore si sommerà alla parte reale:

```
public static ComplexNumber operator + (ComplexNumber z1, int i)
{
return new ComplexNumber(z1.real + i, z1.imaginary);
}
```

In questo caso, quindi, l'operatore + può agire su un operando sinistro di tipo Complex-Number e un int:

```
ComplexNumber z;
ComplexNumber sommaint=z+10;
```

Se si provasse a questo punto a eseguire la somma scambiando gli operandi, nel seguente modo:

```
ComplexNumber sommaint=10+z;
```

si otterrebbe un errore di compilazione, che segnala l'impossibilità di applicare l'operatore + a operandi di tipo int e ComplexNumber. Quindi è necessario implementare un nuovo overload

dell'operatore, con gli operandi invertiti:

```
public static ComplexNumber operator + (int i, ComplexNumber z1)
{
return z1 + i;
}
```

Questo nuovo overload, visto che ce l'abbiamo già a disposizione, utilizza la versione precedente dell'operatore + con gli operandi scambiati.

Espressioni corpo

Anche per l'overload degli operatori, C# 6 consente l'utilizzo di espressioni per definire il loro corpo. Il precedente esempio, con l'overload dell'operatore + per la classe ComplexNumber, può essere riscritto come segue:

```
public static ComplexNumber operator +(ComplexNumber z1, int i) => new ComplexNumber(z1.real + i, z1.imaginary);
```

Operatori sovraccaricabili

Non per tutti gli operatori di C# è possibile implementare un *overload*. Gli operatori unari per i quali è possibile farlo sono i seguenti:

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

Ecco, per esempio, come effettuare l'overload dell'operatore unario di negazione -, per ottenere da un numero complesso (a+ib) il suo numero opposto (-a-ib):

```
public static ComplexNumber operator -(ComplexNumber z)
{
return new ComplexNumber (-z.real, -z.imaginary);
}
```

Rivedendo l'elenco precedente, forse vi starete chiedendo se vi è sfuggito qualcosa: true e false sono operatori? No, non lo sono, ma poiché essi sono utilizzati in espressioni booleane e condizionali (per esempio in un'istruzione if è possibile testare se un bool b è true semplicemente con if(b)), l'overload di true e false può essere utile per poter utilizzare direttamente un oggetto in tali espressioni.

In genere l'overload di true e false viene eseguito per tipi che possono assumere due soli valori oppure per tipi assimilabili al booleano (per esempio il tipo DbNull utilizzato per rappresentare un valore nullo nei database).

Gli operatori true e false, inoltre, devono essere sempre sovraccaricati in coppia (non è possibile implementare l'overload di uno solo dei due). Ecco un esempio di overload, implementato con espressioni corpo, di true e false per la classe ComplexNumber, supponendo che un numero complesso sia equiparabile a true se almeno una fra le parti reale e immaginaria è diversa da 0, e false se entrambe sono 0:

```
public static bool operator true(ComplexNumber z) => z.real != 0 || z.imaginary != 0;
public static bool operator false(ComplexNumber z) => z.real == 0 && z.imaginary == 0;
```

Con questi overload si potrà utilizzare un'istanza di ComplexNumber in questo modo:

```
ComplexNumber z;
…
if(z)
{
//parte reale o immaginaria sono diverse da zero
}
else //numero complesso nullo
```

Nell'esempio precedente, non è possibile ancora utilizzare l'operatore ! per la negazione booleana, anche se è stato sovraccaricato il false. Sarà necessario eseguire l'overload anche dell'operatore unario !.

Gli operatori binari sovraccaricabili sono invece i seguenti:

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

Quando un operatore binario è sottoposto a overload, anche il corrispondente operatore di assegnazione composta (se esiste) implicitamente lo è. Per esempio, se si implementa l'operatore +, avremo a disposizione anche quello +=.

Oltre alle operazioni aritmetiche, si possono sovraccaricare anche operatori logici, condizionali, di shift. Gli overload degli operatori di confronto devono essere eventualmente implementati in coppia: == e !=, <= e >=, < e > (d'altronde non avrebbe senso poterne utilizzare solo uno dei due); inoltre essi devono restituire un valore bool.

Ecco per esempio come implementare l'overload degli operatori < e > per un numero complesso, considerando che ogni numero a+ib ha un modulo definito come la radice quadrata di (a^2+b^2) e che viene implementato come proprietà della classe:

```
public double Modulo => Math.Sqrt(Math.Pow(this.real, 2) + Math.Pow(this.imaginary, 2));
public static bool operator <(ComplexNumber z1, ComplexNumber z2)
{
return z1.Modulo < z2.Modulo;
}
public static bool operator >(ComplexNumber z1, ComplexNumber z2)
{
return z2.Modulo < z1.Modulo;
}
```

Nel caso precedente l'operatore > utilizza l'overload dell'operatore <, scambiando gli operandi z1 e z2, per eseguire il confronto al contrario. In tal modo possiamo confrontare due numeri complessi semplicemente scrivendo:

```
if(z1>z2)
{…}
```

L'operatore di indicizzazione [] non è fra quelli elencati negli operatori sovraccaricabili, però fra qualche paragrafo vedremo come, mediante gli indicizzatori di una classe, si ottenga un risultato analogo all'overload.

Nell'elenco degli operatori binari che possono essere sottoposti a overload mancano anche gli operatori logici condizionali && e ||, ma essi vengono valutati tramite & e |, che invece è possibile sottoporre a overload.

Conversioni personalizzate

L'operatore di cast () non può essere sottoposto a overload, ma è possibile definire delle conversioni personalizzate fra tipi, creando particolari overload di operatori.

Nei capitoli precedenti abbiamo visto come la conversione da un tipo all'altro può essere effettuata in maniera implicita, oppure esplicita mediante l'operatore di cast.

Per implementare tali conversioni personalizzate bisogna utilizzare le keyword **explicit** oppure **implicit**, rispettivamente per le conversioni esplicite (o cast) e per quelle implicite, come se fossero degli operatori. Tali particolari overload sono necessari se si vuole convertire un tipo personalizzato, per esempio una classe, in un altro tipo che non ha alcuna relazione di ereditarietà o interfacce comuni.

La dichiarazione di una conversione avviene mediante la definizione di un operatore, usando come nome dell'operatore quello del tipo verso il quale convertire e, come unico parametro, il tipo di origine della conversione.

È necessario che il tipo di origine oppure il tipo del risultato della conversione, ma non entrambi, coincida con il tipo che contiene l'operatore.

Ecco per esempio come implementare, all'interno della classe ComplexNumber, la *conversione esplicita* di un numero intero verso un numero complesso, semplicemente creando un complesso con parte reale pari all'intero e parte immaginaria nulla:

```
public static explicit operator ComplexNumber (int i)
{
return new ComplexNumber(i,0);
}
```

Oppure equivalentemente con un'espressione lambda:

```
public static explicit operator ComplexNumber(int i) => new ComplexNumber(i, 0);
```

In tal modo potremmo scrivere un'espressione del genere:

```
int i=1;
ComplexNumber z=(ComplexNumber) i;
```

Il compilatore tenta di trovare una conversione possibile anche quando non ne esiste una diretta fra il tipo origine e il tipo destinazione. Per esempio, anche la conversione fra double e ComplexNumber sarebbe possibile:

```
double d=1.0;
ComplexNumber z=(ComplexNumber) d;
```

Questa conversione avverrà correttamente, perché abbiamo precedentemente definito una conversione personalizzata esplicita da int a ComplexNumber e il compilatore C# sa come convertire un double in int con un altro cast. Quindi, combinando le conversioni, esso riesce a convertire un numero double in ComplexNumber.

La procedura per definire una conversione personalizzata implicita è analoga alla precedente, basta utilizzare la parola chiave implicit. Per esempio, se volessimo convertire implicitamente un ComplexNumber in un array di due numeri int, potremmo implementare il seguente metodo:

```
public static implicit operator int[](ComplexNumber z)
{
return new int[]{z.real, z.imaginary};
}
```

Esso crea un array a partire dalle due componenti reale e immaginaria del numero complesso. Potremmo quindi assegnare direttamente un oggetto ComplexNumber a un array di int:

```
ComplexNumber z=new ComplexNumber(1,2);
int[] array=z; //si ottiene [1,2]
```

Tipi innestati

Una classe o struct definita all'interno di un'altra classe o struct è detta tipo innestato, annidato, o nidificato. Il concetto di tipi innestati consente di realizzare un incapsulamento di tipo "ha un", controllando il livello di visibilità nel dettaglio. Per esempio uno Smartphone possiede una batteria, che può essere rappresentata mediante una classe Battery. Se tale oggetto non deve essere direttamente utilizzato all'esterno della classe Smartphone, si può definire la classe Battery come classe innestata in Smartphone:

```
public class Smartphone
{
private string modello;
class Battery
{
private double percentualeCarica;
}
}
```

I tipi innestati hanno come visibilità predefinita private, quindi la classe Battery dell'esempio precedente non è utilizzabile all'esterno della classe Smartphone. Ma è possibile modificare il modificatore di accesso, applicandone uno qualunque fra public, internal, protected, protected internal.

Per esempio, se volessimo rendere visibile la classe Battery all'interno dell'intero assembly, potremmo impostare il suo modificatore di accesso a internal:

```
public class Smartphone
{
private string modello;
internal class Battery
{
private double percentualeCarica;
}
}
```

Ora è possibile istanziare Battery all'interno dello stesso assembly, utilizzando il nome interamente qualificato, cioè con la seguente sintassi:

```
Smartphone.Battery battery=new Smartphone.Battery();
```

I tipi innestati, in quanto essi stessi membri, hanno accesso diretto a tutti i membri del tipo contenitore. Basta avere naturalmente un'istanza del tipo esterno da utilizzare, per esempio passandola come argomento al costruttore del tipo innestato:

```
public class Smartphone
{
private string modello;
internal class Battery
{
private Smartphone phone;
internal Battery(Smartphone phone)
{
this.phone=phone;
}
private double percentualeCarica;
}
}
```

In questo modo, all'interno della classe interna Battery si può accedere al campo private della classe Smartphone.

Il livello di tipi innestati non ha limite, per esempio all'interno della classe Battery è possibile definire un ulteriore tipo innestato, per esempio un'enumerazione dei livelli di carica:

```
public class Smartphone
{
private string modello;
internal class Battery
{
private Smartphone phone;
internal Battery(Smartphone phone)
{
this.phone=phone;
}
private double percentualeCarica;
```

```
private enum LivelloBatteria {esaurita, basso, medio, alto, totale};
}
}
```

In questo caso l'enumerazione LivelloBatteria è private, quindi è utilizzabile solo all'interno della classe Battery.

Classi statiche

Abbiamo trattato in precedenza cosa sono i membri statici e come, per esempio, è possibile definire dei costruttori statici per una classe. Se una classe contiene solo membri statici, per esempio proprietà e metodi, essa può essere definita come statica:

```
public static class MiaClasse
{
//membri statici
}
```

Una classe statica non può mai essere istanziata, quindi può essere utilizzata solo per invocare metodi e proprietà statiche. Un esempio di classe statica contenuta nella libreria di classi di base di .NET è la classe Console, i cui metodi, per esempio WriteLine, vengono utilizzati senza necessità di istanziare un oggetto della classe.

Inizializzatori di oggetti

La creazione di un'istanza di una determinata classe avviene in genere mediante l'utilizzo dell'operatore new, seguito dal nome del costruttore che si vuol utilizzare, a cui vengono eventualmente passati i parametri adeguati e previsti dal costruttore stesso.

Un'altra possibilità consente di utilizzare un *inizializzatore* di oggetti, indicando i campi e le proprietà pubbliche che si vogliono inizializzare, subito dopo l'invocazione del costruttore, impostandone i relativi valori.

Per esempio, data la classe Smartphone seguente, che possiede due proprietà pubbliche e due costruttori:

```
class Smartphone
{
public string Marca{get;set;}
public string Modello {get;set;}
private DateTime creationTime;

public Smartphone(): this(DateTime.Now)
{
}

public Smartphone(DateTime dt)
{
creationTime=dt;
}
}
```

è possibile inizializzarne un'istanza, impostando direttamente le proprietà Marca e Modello così:

```
Smartphone phone=new Smartphone { Marca="aple", Modello="ifone 5" };
```

Se si vuole utilizzare un altro costruttore con altri parametri, è possibile in ogni caso indicare sia i parametri del costruttore sia l'inizializzatore dell'oggetto:

```
Smartphone phone=new Smartphone(DateTime.Now) { Marca="aple", Modello="ifone 5" };
```

Con gli inizializzatori di oggetto si deve prestare attenzione al fatto che l'inizializzazione dei membri indicati avviene subito dopo l'esecuzione del costruttore, quindi i valori potrebbero anche già essere stati impostati nel costruttore stesso e poi sovrascritti dall'inizializzatore.

NOTA	Gli inizializzatori di oggetto sono stati introdotti con C# 3.0 e saranno particolarmente utili con LINQ, come si vedrà nel Capitolo 11 dedicato a tale argomento.

Indicizzatori

Gli indicizzatori sono dei membri analoghi alle proprietà che consentono di accedere agli oggetti come se fossero degli array, cioè mediante un indice. Essi sono quindi particolarmente utilizzati in classi che contengono liste di elementi.

Fra le classi standard di .NET, System.String possiede un indicizzatore che consente di accedere ai caratteri che formano la stringa stessa. È infatti possibile ottenere i vari caratteri utilizzando l'operatore di indicizzazione [] e l'indice numerico, con base zero, del carattere che si vuol ottenere:

```
string str="hello";
char h=str[0];
char o=str[4];
```

La sintassi quindi è identica a quella di accesso agli elementi di un array, con la differenza, come vedremo fra poco, che l'indice non deve essere necessariamente numerico.

Per implementare un indicizzatore in una classe personalizzata, bisogna creare una proprietà chiamata **this**, specificando il tipo di ritorno e il parametro da utilizzare come indice:

```
public Tipo this[TipoIndice indice]
{
get
{
…
}
set
{
…
}
}
```

Per esempio, se avessimo una classe TelephoneNumber che rappresenta i numeri telefonici memorizzati in un telefono, potremmo innanzitutto aggiungere alla classe Smartphone un array di numeri telefonici:

```
class TelephoneNumber
{
public string Nome { get; set; }
public int Number { get; set; }
}
class Smartphone
{
private TelephoneNumber[] numeriTelefonici;
public TelephoneNumber this[int index]
{
get
{
return numeriTelefonici[index];
}
set
{
numeriTelefonici[index]=value;
}
}
}
```

A questo punto per accedere alla proprietà indicizzata basta utilizzare l'operatore [] su un'istanza della classe:

```
Smartphone sp=new Smartphone();
TelephomeNumber primo= sp[0];
```

Una classe può contenere diversi indicizzatori, basta che il tipo utilizzato come indice all'interno delle parentesi quadre sia univoco per ognuno di essi. Per esempio, potremmo accedere a un numero telefonico anche per nome, aggiungendo un nuovo indicizzatore this, con un indice di tipo string anziché int:

```
public TelephoneNumber this[string nome]
{
get
{
foreach (TelephoneNumber number in numeri)
{
if (number.Nome == nome)
return number;
}
return null;
}
}
```

In questo esempio, il ciclo foreach verifica se esiste un numero con il nome corrispondente a quello passato come indice e in caso affermativo lo restituisce. Se si raggiunge la fine del

ciclo senza risultati, l'indicizzatore restituisce invece il riferimento null.

Naturalmente non è l'implementazione più efficiente per farlo, ma ancora non abbiamo a nostra disposizione strumenti e classi come le collezioni.

Inizializzatori con indice

A partire da C# 6, gli oggetti di classi con indicizzatori possono essere inizializzati mediante una sintassi semplificata.

All'interno delle parentesi graffe di un inizializzatore (vedi Paragrafo "Inizializzatori di oggetti") è possibile infatti assegnare i valori direttamente a un dato indice dell'oggetto.

Supponiamo per esempio di avere la seguente classe:

```
class Smartphone
{
Dictionary<string,string> numbers=new Dictionary<string, string>();

public string this[string name]
{
get
{
if (numbers.ContainsKey(name))
return numbers[name];
return null;
}
set
{
numbers[name] = value;
}
}
}
```

Tralasciando il fatto che utilizzi un tipo Dictionary, su cui torneremo nel Capitolo 9, l'indicizzatore permette di leggere e scrivere i numeri di telefono, accedendo al dizionario numbers, mediante un indice che rappresenta il nome del contatto.

Per esempio, data un'istanza della classe Smartphone, è possibile scrivere e leggere un numero di un contatto nel seguente modo:

```
Smartphone phone = new Smartphone()
phone["matilda"]="1234567"; //scrive numero
Console.WriteLine(phone["matilda"]); //legge e stampa numero
```

L'inizializzazione dell'oggetto può sfruttare la sintassi introdotta da C# 6, utilizzando appunto gli indicizzatori, nel seguente modo:

```
Smartphone phone = new Smartphone()
{
["antonio"] = "3341234",
["caterina"] = "3204445",
```

```
};
```

Tale sintassi sarà utilizzata anche per l'inizializzazione dei dizionari, come si vedrà nel Capitolo 9 nella parte dedicata alle collezioni.

Espressioni corpo

Il corpo di un indicizzatore, con C# 6, può essere abbreviato utilizzando un'espressione lambda, come visto già per metodi, proprietà e overload di operatori.

Naturalmente in questo caso, come per le proprietà, l'indicizzatore è necessariamente di sola lettura.

Riprendendo la classe ComplexNumber di qualche paragrafo fa, ecco un paio di indicizzatori che permettono di accedere alla parte reale o immaginaria utilizzando un carattere oppure una stringa come indice:

```
public int this[char name] => name=="r"? real : imaginary;
public int this[string name] => name=="real"? real: imaginary;
```

Se il carattere passato come indice al primo è uguale a "r", oppure la stringa passata al secondo è "real", viene restituita la parte reale del numero, altrimenti quella immaginaria. In entrambi gli esempi è stato utilizzato l'operatore ternario per gestire la valutazione della condizione.

Metodi di estensione

Ci sono diversi modi per estendere una classe esistente; naturalmente quello che vi sarà venuto in mente come principale è l'ereditarietà (ed era anche l'unico prima di .NET 3.5), che abbiamo già discusso in teoria e che affronteremo nella pratica nel prossimo capitolo.

Un'altra possibilità per aggiungere funzionalità a una classe esistente, sia implementata da noi che da terze parti, per esempio presa dalla Base Class Library, è quella di utilizzare i *metodi di estensione*. Con i metodi di estensione si può dotare una classe esistente di nuovi metodi, per esempio per eseguire azioni particolari non originariamente previste dal creatore della classe.

La classe System.String, come visto, possiede già molti metodi utili per affrontare varie problematiche, ma magari manca ancora di qualcosa, che potrebbe servire in particolari ambiti.

NOTA	Fra le altre cose la classe String è una classe sealed, il che, come vedremo nel prossimo capitolo, significa che non è possibile derivare da essa un'altra classe mediante ereditarietà. Quindi, per aggiungere a essa delle funzionalità non esistenti o personalizzate, dobbiamo ricorrere necessariamente ai metodi di estensione.

Supponiamo, per esempio, di voler verificare se una stringa rappresenta un numero.

La prima possibilità è utilizzare il metodo TryParse di System.Double:

```
string str="123";
double res;
bool isNumeric=(double.TryParse(str, out res));
```

Quindi abbiamo già a disposizione quello che serve. A questo punto potremmo anche creare una classe di utility, in cui inserire i nostri metodi statici da utilizzare come e quando vogliamo, semplificando il codice precedente in maniera da evitare di dover dichiarare una variabile da passare come parametro di output:

```
public class StringHelper
{
public static bool IsNumeric(string str)
{
double res;
return double.TryParse(str, out res));

}
}
```

In questo modo, ogni volta che abbiamo necessità di verificare se una stringa rappresenta un double, possiamo utilizzare il metodo statico IsNumeric:

```
bool isNumeric=StringHelper.IsNumeric(str);
```

Ma se dovessimo ripetere spesso questa operazione, oppure se dovessimo utilizzare il metodo concatenandolo con altre operazioni, incontreremmo qualche difficoltà. Per esempio, se la stringa da verificare deve essere prima ripulita da eventuali spazi bianchi, dovremmo scrivere:

```
string str=" 123 "; //notare gli spazi prima e dopo
bool isNumeric=StringHelper.IsNumeric(str.Trim());
```

Il metodo Trim si occupa di rimuovere gli spazi bianchi; sulla stringa risultante si utilizza poi il metodo IsNumeric della classe StringHelper.

L'obiettivo è ora quello di estendere la classe String in maniera da avere un metodo che si utilizzi semplicemente così:

```
bool isNumeric=str.IsNumeric();
```

I metodi di estensione ci consentono di farlo. Per creare dei metodi di estensione di una particolare classe, bisogna definire una classe statica, alla quale aggiungere i metodi con cui vogliamo estendere un particolare tipo.

Il primo parametro del metodo deve essere del tipo che si vuol estendere, per esempio String, preceduto da this. La parola chiave **this** indica al compilatore che il metodo farà parte del tipo

String. All'interno del metodo naturalmente bisogna implementare la logica desiderata, per esempio:

```
namespace Extensions
{
public static class StringExtensions
{
public static bool IsNumeric(this string str)
{
double res;
return double.TryParse(str, out res);
}
}
}
```

In tal modo ora la classe String possiede un nuovo metodo, basta ricordarsi di referenziare il namespace in cui abbiamo creato la classe con i metodi di estensione:

```
using Extensions;
…
bool isNumeric=str.IsNumeric();
```

La Figura 6.1 mostra come a questo punto anche in Visual Studio il metodo sia visibile e visualizzato dall'intellisense, compresa la sua documentazione.

```
public static class StringExtensions
{
    /// <summary>
    /// Verifica se la string contiene un numero double
    /// </summary>
    /// <param name="str">stringa da verificare</param>
    /// <returns>true se la stringa rappresenta un double</returns>
    public static bool IsNumeric(this string str)
    {
        double d;
        return double.TryParse(str, out d);
    }

}
class Program
{
    static void Main(string[] args)
    {
        string str = "123.45";
        bool isnumeric = str.IsNu
    }
```

IsNumeric

(extension) bool string.IsNumeric()

Verifica se la string contiene un numero double

```
}
```

Figura 6.1 – Intellisense di Visual Studio per un metodo di estensione.

I metodi di estensione, facendo ora parte della stessa classe String, possono essere invocati concatenandoli a invocazione di altri metodi, per esempio:

```
bool isNumeric= str.Trim().IsNumeric();
```

NOTA	Tale possibilità tornerà particolarmente utile con LINQ e, infatti, i metodi di estensione principali implementati nella libreria di classi di .NET sono quelli che aggiungono le funzionalità di query LINQ a determinate classi (vedi Capitolo 11).

Si presti attenzione ai nomi assegnati ai metodi di estensione: se la classe da estendere possiede già un metodo con un nome scelto per estenderla, il compilatore utilizzerà sempre il metodo originale.

Inoltre si faccia attenzione anche alla gerarchia della classe estesa con metodi: se per esempio aggiungiamo un metodo di estensione alla classe Object, esso sarà utilizzabile per qualunque classe, in quanto tutte derivano da Object.

Utilizzo dell'operatore null conditional

Nel Capitolo 4 abbiamo già visto il funzionamento dell'operatore null conditional (o *Elvis*) introdotto da C# 6 per accedere ai membri di un oggetto, verificando che ognuno di essi sia diverso da null. Esso può essere utilizzato in cascata, nel caso in cui il membro a cui accedere possa assumere valore null. Supponiamo per esempio di avere una struttura di classi come la seguente:

```
public class Cliente
{
public Indirizzo IndirizzoCliente { get; set;}
}
public class Indirizzo
{
public string Via { get; set;}
}
```

La classe Cliente ha una proprietà di tipo Indirizzo che è un'altra classe con una proprietà Via, di tipo string.

Per ricavare la lunghezza della stringa che ricava la via di un oggetto Cliente basterebbe scrivere:

```
int len=cliente.IndirizzoCliente.Via.Length;
```

Fino a quando i vari membri sono diversi da null, nessun problema. Ma potrebbe verificarsi il caso in cui uno qualunque sia appunto null e quindi l'accesso a esso provochi un'eccezione.

L'operatore ?. gestisce questi casi, propagando la verifica a ogni membro:

```
int? len=cliente?.IndirizzoCliente?.Via?.Length;
```

In questo modo se cliente, oppure IndirizzoCliente, oppure ancora Via hanno valore null, il risultato finale ottenuto sarebbe proprio null, senza necessità di scrivere, per esempio, delle condizioni if, per controllare uno per uno i vari membri.

Nell'esempio, la variabile len è necessariamente di tipo nullable, cioè int?, perché essa deve essere capace di memorizzare anche il valore null.

NOTA	L'operatore null conditional è chiamato anche operatore di null propagation, proprio per questa sua caratteristica di poter essere applicato ai membri in cascata, propagando appunto la verifica.

L'operatore null conditional può essere usato anche per invocare un metodo, su un'istanza possibilmente null, evitando lo scatenarsi di una NullReferenceException:

```
Cliente cliente=null;
cliente?.ToString();
```

In questo caso la chiamata del metodo ToString() non ha alcun effetto, è come se non ci fosse proprio stata.

L'operatore null conditional può essere utilizzato in congiunzione con l'operatore ?? di null coalescing.

Se volessimo infatti assegnare un valore diverso da null alla variabile len, potremo scrivere:

```
int len=str?.Length ?? 0; //se str è null restituisce 0
```

L'ultima istruzione equivale a utilizzare l'operatore ternario:

```
int len = (str != null) ? str.Length : 0;
```

Struct

Le struct sono molto simili alle classi in quanto permettono di definire membri dati e membri funzione.

Conosciamo già la differenza principale fra i due tipi di dati: le struct sono tipi valore mentre le classi sono tipi riferimento.

NOTA	Il fatto che una variabile sia di un tipo valore non implica che essa risiederà sempre nella memoria stack. Per esempio, se una classe contiene un campo di un tipo struct, esso sarà allocato insieme all'oggetto che lo contiene, in memoria heap.

Se è necessario definire dei tipi che rappresentino semplici strutture dati, è possibile utilizzare le struct. Esempi di tipi adeguati a essere implementati come strutture possono essere dei punti in uno spazio bidimensionale o tridimensionale, un numero complesso, un indirizzo IPV4 formato da 4 byte.

In generale una struct ha pochi membri dati, non richiede l'uso dell'ereditarietà e ha un comportamento di tipo valore, per esempio una copia di una variabile struct, mediante assegnazione, copia i valori dei membri dati, anziché i riferimenti.

Per definire una nuova struttura si utilizza la parola chiave struct in maniera analoga alla creazione di una classe, per esempio:

```
struct Point
{
public double X;
public double Y;
}
```

Il fatto che una struct sia un tipo valore non impedisce che essa possa contenere dei campi di tipo riferimento.

Come nell'esempio precedente, spesso per una struct si utilizzano campi public; ciò sembrerebbe contraddire quanto abbiamo detto parlando di incapsulamento, ma in questo caso, essendo le struct utilizzate per creare semplici tipi che raggruppano campi di tipi anch'essi semplici, essa non è considerata una pratica errata.

Una struct, come una classe, ha sempre un costruttore di default predefinito, che non accetta parametri, però, a differenza delle classi, esso non può essere implementato esplicitamente:

```
struct Point
{
```

```
public double X;
public double Y;

public Point() //errore, non può essere definito esplicitamente un costruttore di default
{
}
}
```

Inoltre, non è possibile aggirare tale limitazione; inizializzando i campi direttamente, come nel seguente esempio, si ottiene un errore di compilazione:

```
struct Point
{
public double X=1; //errore
public double Y=2; //errore
}
```

Per creare una variabile di un tipo struct, è possibile utilizzare come per le classi l'operatore **new**:

```
Point p=new Point();
p.X=1;
p.Y=2;
```

La differenza, rispetto all'utilizzo di new con una classe, è che in questo caso non viene allocato lo spazio in memoria heap, ma semplicemente viene invocato il costruttore della struct e inizializzati i campi ai rispettivi valori di default: prima dell'assegnazione dei valori a p.X e p.Y, essi assumeranno quindi valore 0.

Senza utilizzare l'operatore new, è lecito e corretto anche scrivere il codice così:

```
Point p;
p.X=1;
p.Y=2;
```

Il codice precedente funziona perché il compilatore alloca memoria nello stack sufficiente per tutti i campi della struct, mentre, se si fosse trattato di una classe, l'istruzione di assegnazione avrebbe dato un errore, perché la variabile p avrebbe rappresentato un riferimento null.

L'errore di compilazione, però, si verificherebbe scrivendo qualcosa del genere:

```
Point p;
Console.WriteLine("{0},{1}", p.X, p.Y);
```

In questo caso p.X e p.Y sono due variabili non inizializzate, quindi se non si assegna loro un valore, come nell'esempio precedente, non è consentito utilizzarle.

Costruttori

Come detto sopra, non è possibile scrivere esplicitamente il costruttore di default di una struct, ma è possibile aggiungere altri *costruttori* personalizzati con parametri.

NOTA	Il costruttore predefinito senza parametri non è implementabile in modo esplicito per evitare errori e comportamenti inaspettati; infatti, se fosse possibile sostituirlo, si potrebbero inizializzare al suo interno i campi di una struct con valori diversi da quelli predefiniti previsti per i rispettivi tipi.

Per esempio, se volessimo inizializzare i valori delle coordinate X e Y in fase di creazione di una variabile Point, potremmo implementare un costruttore come il seguente:

```
public Point(double x, double y)
{
this.X=x;
this.Y=y;
}
```

In questo caso, è obbligatorio inizializzare i valori di tutti i campi della struttura.

Come nelle classi, in una struct è possibile utilizzare la parola chiave this per riferirsi all'oggetto corrente, anche se in questo caso non si tratta di un oggetto di tipo riferimento, quindi è come se this fosse una variabile del tipo della struct corrente.

Valori o riferimenti

Quando si chiama l'operatore new su una classe, una sua istanza viene allocata nell'area di memoria heap. Al contrario, se invece si crea una variabile di un tipo struct, essa viene creata nello stack. Questo comporta dei vantaggi in termini di prestazioni.

L'assegnazione di una variabile di tipo struct a un'altra variabile crea una copia del valore, a differenza dell'assegnazione di tipi riferimento, in cui appunto viene copiato solo il riferimento e non l'oggetto puntato dal riferimento. Quindi nel seguente esempio, dopo l'assegnazione, i valori delle coordinate dei punti saranno identici:

```
Point pt1=new Point(1,1)
Point pt2=pt1; //copia del valore
Console.WriteLine("{0} {1}", pt.X, pt.Y); //stampa 1,1
Console.WriteLine("{0} {1}", pt2.X, pt2.Y); //stampa 1,1
```

Se ora si modificano i campi di pt2, i cambiamenti non si rifletteranno sulla variabile pt1:

```
pt2.X=11;
pt2.Y=22;

Console.WriteLine("{0} {1}", pt.X, pt.Y); //stampa 1,1
Console.WriteLine("{0} {1}", pt2.X, pt2.Y); //stampa 11,22
```

Di conseguenza, in maniera simile all'assegnazione, se una variabile struct viene passata come argomento di un metodo, oppure restituita da un metodo con return, essa viene passata per valore e quindi viene eseguita una copia della struct.

Il seguente metodo, per esempio, somma a una variabile Point le coordinate di un altro Point:

```
struct Point
{
public double X;
public double Y;

public void AddToPoint(Point pt)
{
pt.X += this.X;
pt.Y += this.Y;
}
}
```

Se ora proviamo a utilizzarlo nel seguente modo:

```
Point pt=new Point(1,1);
Point pt2=new Point(3,4);
pt2.AddToPoint(pt);
Console.WriteLine("{0} {1}", pt.X, pt.Y); //stampa 1,1
```

la variabile pt, uscendo dal metodo, possiede ancora i valori (1,1), perché quella utilizzata all'interno del metodo AddToPoint è una copia di quella originale.

Una struct può essere comunque passata per riferimento a un metodo utilizzando i modificatori ref e out.

Utilizzando, infatti, il modificatore ref nel metodo AddToPoint:

```
public void AddToPoint(ref Point pt)
{
pt.X += this.X;
pt.Y += this.Y;
}
…
Point pt=new Point(1,1);
Point pt2=new Point(3,4);
pt2.AddToPoint(ref pt);
Console.WriteLine("{0} {1}", pt.X, pt.Y); //stampa 4,5
```

stavolta la variabile pt passata per riferimento vedrà i suoi campi modificati, quindi i valori di X e Y dopo l'esecuzione della somma e l'uscita dal metodo saranno pari a (4,5).

Differenze con le classi

Una struct può contenere tutti i tipi di membri già visti per le classi, quindi membri per memorizzare dati (campi, costanti ed eventi) e membri di tipo funzione (proprietà, metodi, costruttori, indicizzatori, operatori e tipi innestati) a eccezione di distruttori e costruttori senza parametri.

Quindi la struct Point può, per esempio, contenere campi, proprietà e metodi, come nel seguente esempio:

```
public struct Point
```

```
{
private double x;
private double y;

public double X
{
get
{
return x;
}
set
{
x=value;
}
}
public double Y
{
get
{
return y;
}
set
{
y=value;
}
}
public string DisplayPoint() => $"{X},{Y}";
}
```

Le struct derivano implicitamente dalla classe ValueType, ma non supportano l'ereditarietà. Non è cioè possibile indicare in una dichiarazione di struct il nome di un'altra struct da cui essa deriva. Di conseguenza, una struct non può essere dichiarata né con il modificatore abstract né con sealed.

Una struct può implementare invece delle interfacce (vedere il prossimo capitolo).

Tipi parziali

I *tipi parziali*, introdotti con C# 3.0, consentono di suddividere il codice di implementazione di un tipo in più file. Tale eventualità è utile quando il codice è talmente lungo da essere più facilmente gestibile in diversi file, magari separandone diverse sezioni per funzionalità, oppure quando si utilizzano strumenti di generazione del codice che creano parte dei tipi in maniera automatica, e che non devono essere modificati manualmente dallo sviluppatore.

Un tipico esempio si ha sviluppando applicazioni Windows Forms in Visual Studio. L'IDE genera automaticamente il codice che costruisce la finestra e i suoi controlli, mentre i metodi di logica dovranno essere implementati dallo sviluppatore. In questo caso il codice di definizione dell'interfaccia viene generato e mantenuto in un file .cs separato, evitando che lo sviluppatore per errore lo modifichi provocando malfunzionamenti o errori.

Ciò sarebbe tranquillamente permesso dal fatto che non è obbligatorio salvare una classe in un file con il suo stesso nome. Per esempio, Visual Studio utilizza un file del tipo MyForm.designer.cs per generare il codice automatico della classe MyForm, mentre nel file MyForm.cs il programmatore potrà inserire i propri metodi facenti parte della stessa classe.

Per creare tipi parziali (parliamo di tipi, perché classi, struct e interfacce possono tutte essere suddivise in più file di codice) basta utilizzare il modificatore partial, prima della definizione della classe:

```
//File class1.cs
partial class Class1
{
//membri
}

//secondo_File.cs
partial class Class1
{
//altri membri
}
```

Ogni parte del file deve comprendere il modificatore partial; il seguente codice, per esempio, sarebbe errato:

```
//File class1.cs
partial class Class1
{
}

//secondo_File.cs
class Class1 //errore manca partial
{
```

```
}
```

Inoltre, i modificatori di accesso delle diverse parti non devono essere in conflitto. Per esempio, non è possibile dichiarare una parte della classe come public e in un altro file una parte come private:

```
//File class1.cs
public partial class Class1
{
}

//secondo_File.cs
private partial class Class1 //errore conflitto dei modificatori di accesso
{
}
```

Invece è possibile specificare una classe base o interfacce indipendenti da implementare per ognuna delle parti.

Vedremo nel prossimo capitolo come utilizzare ereditarietà e interfacce, per il momento notate che il seguente codice è corretto e legittimo:

```
//File class1.cs
partial class Class1: ClasseBase
{
//membri
}

//secondo_File.cs
partial class Class1: Interfaccia1
{
//altri membri
}
```

Naturalmente perché tutte le parti della classe siano compilate correttamente, ogni file che concorre alla definizione del tipo deve essere dato in pasto al compilatore.

I metodi parziali

Un tipo parziale può contenere *metodi parziali*, cioè metodi la cui implementazione è suddivisa in una parte di definizione, che indica solo la firma del metodo, e una parte con l'implementazione vera e propria del corpo del metodo.

Anche in questo caso è necessario utilizzare il modificatore partial prima della definizione del metodo, in entrambe le parti che lo compongono:

```
//File class1.cs
partial class Class1
{
partial void MetodoParziale();
}
```

```
//secondo_File.cs
partial class Class1
{
partial void MetodoParziale()
{
//implementazione del metodo
}
}
```

NOTA	Visual Studio aiuta nell'implementazione dei metodi parziali, suggerendo il nome di un metodo di cui è presente solo la parte di definizione, subito dopo aver digitato la parola chiave partial e premuto spazio.

Nel caso in cui sia presente solo la parte di definizione del metodo, il compilatore tralascia di compilarli, ma non segnalerà nessun errore. Ciò è consentito in maniera da poter scrivere, magari da parte di strumenti automatici, delle firme di metodi che poi potranno essere implementati o meno.

Il fatto che possa mancare l'implementazione del metodo implica che i metodi parziali devono restituire void e sono implicitamente private, quindi non possono avere altri modificatori di accesso (per completezza, anche se non sappiamo ancora a cosa servono, i metodi parziali non possono utilizzare nemmeno modificatori di ereditarietà come virtual, abstract, new, override, sealed, extern).

Infatti, se mancasse la parte di implementazione del metodo, non sarebbe possibile ottenere un valore di ritorno che potrebbe essere utilizzato nel seguito; inoltre, una classe esterna, se fosse public, anche in un altro assembly, non potrebbe sapere se l'implementazione è effettivamente presente o meno.

Tipi anonimi

Nel Capitolo 2 abbiamo visto come utilizzare la parola chiave var per creare variabili di tipo implicito, ricavandolo cioè direttamente dall'espressione assegnata alla variabile stessa.

Combinando l'utilizzo di **var** con l'operatore new è possibile creare oggetti di tipo anonimo. Un *tipo anonimo* non è altro che una classe senza nome che deriva direttamente da System.Object.

La definizione della struttura dati della classe viene ricavata direttamente dalla sua inizializzazione (come accade per le già citate variabili implicite). Per esempio, se volessimo definire un tipo che ci permetta di memorizzare nome, cognome e data di nascita di un cliente, potremmo semplicemente scrivere:

```
var cliente1 = new {Nome="bill", Cognome="gates", DataNascita=new DateTime(1955, 10, 28)};
```

Il compilatore genera automaticamente una classe, il cui nome non è disponibile allo sviluppatore, e dotata di tre proprietà, Nome, Cognome e DataNascita, le prime due di tipo string e la terza di tipo DateTime, ma tutte a sola lettura. Quindi, se costruiamo un altro oggetto nella stessa maniera, cioè con dati dello stesso tipo:

```
var cliente2 = new {Nome="steve", Cognome="jobs", DataNascita=new DateTime(1955, 2, 24);
```

i tipi dei due oggetti cliente1 e cliente2 sono coincidenti, quindi si può assegnare un oggetto all'altro:

```
cliente2=cliente1;
```

Se per costruire un tipo anonimo si utilizzano direttamente le proprietà di un oggetto esistente, allora le proprietà del tipo anonimo prenderanno lo stesso nome di quelle assegnate. Per esempio, se a partire da un'istanza cliente della classe seguente:

```
class Cliente
{
public string Nome;
public string Cognome;
public DateTime DataNascita;
}
```

costruiamo un oggetto di tipo anonimo come segue:

```
var cliente4=new {cliente.Nome, cliente.Cognome, cliente.DataNascita};
```

il tipo dell'oggetto cliente4 sarà un tipo anonimo con tre proprietà denominate Nome, Cognome, e DataNascita (cioè un tipo ancora una volta identico a quello degli esempi sopra). Questo naturalmente è vero se non si utilizzano nuovi nomi per il tipo anonimo costruito a partire da un altro oggetto:

```
var cliente5=new {NomeCliente=cliente.Nome, cliente.Cognome, cliente.DataNascita};
```

Nell'ultimo esempio il tipo anonimo risultante ha una proprietà denominata NomeCliente e non Nome.

I tipi anonimi, come vedremo nel Capitolo 11, vengono solitamente utilizzati nella costruzione di query LINQ, in particolare per restituire un sottoinsieme delle proprietà di un oggetto.

Domande di riepilogo

1) Quali fra questi non è uno dei principi della programmazione orientata agli oggetti?

a. Polimorfismo
b. Ereditarietà
c. Sovrascrittura
d. Incapsulamento

2) In C# è consentita l'ereditarietà multipla fra classi. Vero o falso?

3) Per creare una classe A che deriva dalla classe B si scrive:

a. class A -> B
b. class A extends B
c. class A - B
d. class A : B

4) Come si definiscono dei parametri opzionali in un metodo?

a. public void MioMetodo(int a=0, int b=1)
b. public void MioMetodo(out int a, out int b)
c. public void MioMetodo(params int a=0, int b=1)
d. public void MioMetodo(int? a, int? b)

5) Quale fra questi metodi può accettare un numero di argomenti non determinato?

a. Calcola(int[] array)
b. Calcola(int[] params)
c. Calcola(params int[] a)
d. Calcola(params int array)

6) Quale fra queste proprietà è scritta in modo errato?

a. public string Nome1 => "nome";
b. public string Nome2 { get; } = "nome";
c. public string Nome3 { get; set;} = "nome";
d. public string Nome4 => return "nome";

7) Quale parola chiave si utilizza per l'overload di un operatore?

a. operator
b. overload

c. define

d. override

8) Qual è un modo corretto per scrivere un metodo di estensione Ext del tipo object?

a. public extern void Ext(this object obj)

b. public static void Ext(this object obj)

c. public static void Ext(object obj)

d. public static void Ext(extern object obj)

9) Una struct può derivare da un'altra struct. Vero o falso?

10) Quale fra questi è la firma corretta di un metodo partial?

a. partial void Method(int a);

b. private partial void Method();

c. partial int Method();

d. public partial void Method(int a);

Ereditarietà e polimorfismo

C# supporta i concetti di programmazione orientata agli oggetti, come l'ereditarietà e il polimorfismo, che permettono di creare gerarchie di tipi, sfruttando anche i costrutti di classi astratte e interfacce per definirne e implementarne le funzionalità e il comportamento.

Il linguaggio C#, in quanto linguaggio orientato agli oggetti, supporta tutti i concetti del paradigma OOP. L'ereditarietà e il polimorfismo permettono di implementare una classe a partire da una esistente, per riutilizzarne e ampliarne le funzionalità.

Le classi in C# possono derivare da una singola classe madre, quindi è supportata l'ereditarietà singola, ma mediante l'utilizzo delle interfacce, che permettono di definire un insieme di funzionalità, è possibile implementare anche una particolare forma di ereditarietà multipla. I tipi valore come le struct, invece, non possono essere derivati da altri tipi, ma possono implementare anch'essi delle interfacce.

Un altro modo di definire le funzionalità comuni di una classe base, condivisibili da più classi derivate, risiede nel concetto di classe astratta.

Ereditarietà

Una classe può derivare da una classe esistente ereditando da essa tutti i membri dati e i membri funzione non privati. La classe figlia avrà così a disposizione le versioni base di tali membri, implementate nella classe madre, ma con la possibilità di personalizzarli, fornendo in particolare una propria implementazione di metodi e proprietà o aggiungendo nuovi membri.

Questo tipo di ereditarietà è detta *ereditarietà di implementazione* e si utilizza quando una serie di funzionalità comuni a diverse classi possono essere raggruppate in una singola classe madre, oppure quando si vuole creare una nuova classe aggiungendo nuove funzionalità a una classe esistente.

Per implementare in C# una classe *derivata* da una classe madre, si indica il nome della classe madre facendolo seguire al nome della classe che si sta implementando e li si separa con l'operatore due punti (:). In generale quindi la sintassi è la seguente:

```
public class A
{
//membri di A
}

class B: A
{
}
```

Si dice in questo caso che la classe B *deriva* da A, oppure B *eredita* da A.

Per visualizzare le gerarchie di ereditarietà è molto utile ricorrere a *UML*, un linguaggio di modellazione e di specifica basato sul paradigma orientato agli oggetti.

In particolare UML consente di disegnare dei *diagrammi delle classi*, in cui le classi sono raffigurate mediante rettangoli con dei compartimenti per il nome della classe stessa, per i campi, per i metodi e così via, anche se non è obbligatorio inserire e indicare tutti i membri.

Per indicare che una classe A deriva da una classe B, si disegna una freccia continua che parte dalla classe derivata e termina sulla classe madre (vedi Figura 7.1).

Come detto, non è obbligatorio indicare tutti i membri: spesso i diagrammi di classe vengono utilizzati per rappresentare uno schema semplificato delle classi che compongono il progetto, indicando solo relazioni di ereditarietà.

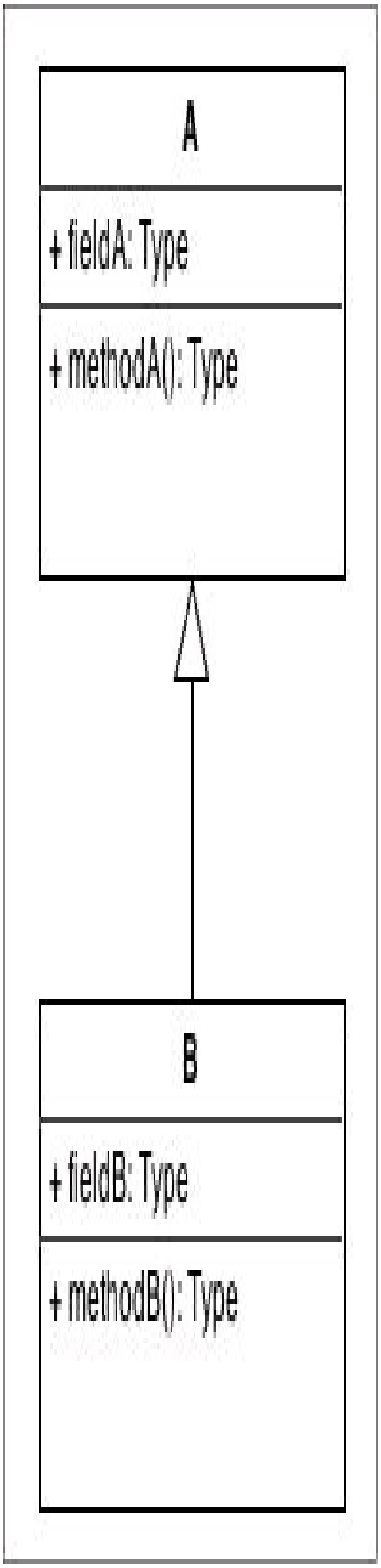

A
+ fieldA: Type
+ methodA(): Type
B
+ fieldB: Type
+ methodB(): Type

Figura 7.1 - Diagramma delle classi UML per indicare classe madre e classe derivata.

NOTA	Visual Studio 2015 consente di creare dei diagrammi di classe a partire dalle classi contenute in un progetto, semplicemente trascinandole sull'area di disegno, oppure di creare nuove classi direttamente dal diagramma, generandone poi il codice C#. Per creare un nuovo diagramma, basta aggiungere un nuovo elemento al progetto e quindi selezionare il template Class Diagram.

Ogni classe ha come madre predefinita la classe System.Object. Ciò significa che, per esempio, la dichiarazione della classe A vista sopra, in cui non è indicata esplicitamente nessuna classe madre, è equivalente alla seguente:

```
public class A: System.Object
{
…
}
```

Poiché l'ereditarietà di .NET è singola, è un errore indicare più classi madre:

```
class A { … }
class B { … }
class C: A, B //Errore
{
}
```

NOTA	La possibilità di ereditare una classe da più classi contemporaneamente è consentita nel linguaggio C++. La scelta di .NET di implementare solo l'ereditarietà singola fra classi consente di semplificare i meccanismi di ereditarietà; inoltre, grazie alla presenza della super classe System.Object, ogni classe ha sempre almeno una classe madre. Una sorta di ereditarietà multipla è consentita, come vedremo più avanti, mediante il meccanismo di implementazione di più interfacce.

Aggiungiamo ora alla classe A un campo e un metodo:

```
public class A
{
private int i;
public void Display()
{
//..
}
}

class B: A
{
}
```

Come detto prima, una classe figlia eredita dalla sua classe madre sia i membri dati sia i membri funzione, a patto che non siano privati. Quindi, nell'esempio sopra, la classe B non eredita il campo intero i, perché privato, mentre eredita il metodo pubblico Display.

Per capire meglio i concetti di ereditarietà e comprendere come applicarli al mondo reale, nei prossimi paragrafi utilizzeremo degli esempi basati sul mondo dei mezzi di trasporto.

In generale possiamo creare una classe Vehicle, da cui derivare ogni altro tipo di mezzo di trasporto. Innanzitutto partiamo col raggruppare all'interno della classe Vehicle delle caratteristiche comuni, per esempio l'anno di costruzione, la velocità attuale, la velocità massima, il numero di passeggeri che può trasportare, e dei metodi che determinano il comportamento delle sue istanze, per esempio quelli per accelerare e frenare.

Naturalmente tali caratteristiche derivano anche dall'ambito in cui dovrà funzionare l'applicazione in cui verrà utilizzata la classe: se, per esempio, si trattasse di un programma per gestire la registrazione dei veicoli presso la motorizzazione civile, allora implementeremmo anche classi per gestire il proprietario, la data di immatricolazione e cose del genere.

Per il momento, i nostri esempi saranno focalizzati a mostrare i vari aspetti relativi all'ereditarietà e al polimorfismo in C#. Iniziamo intanto con implementare una prima versione della classe Vehicle, che è la seguente:

```
class Vehicle
{
private short yearBuilt;
private string numberOfPassengers;
private double maxSpeed;
private double currentSpeed;

public string ModelName{ get; set;}
public double MaxSpeed
{
get
{
return maxSpeed;
}
set
{
maxSpeed = value;
}
}
public void Accelerate()
{
if(currentSpeed < maxSpeed)
currentSpeed += 1.0;
}
public void Brake()
```

```
{
if(currentSpeed > 0)
currentSpeed -= 1.0;
}

public void PrintInfo()
{
Console.WriteLine("{0}: current speed: {1}", ModelName, currentSpeed);
}
}
```

Fin qui niente di nuovo, la classe Vehicle può essere istanziata e utilizzata come già visto nel precedente capitolo:

```
Vehicle v=new Vehicle();
v.ModelName="modello 123";
v.Accelerate();
v.PrintInfo();
v.Brake();
v.PrintInfo();
```

I metodi Accelerate e Brake consentono di variare la velocità del veicolo, senza dover accedere direttamente al campo private currentSpeed; inoltre essi controllano al loro interno che la velocità non superi quella massima prevista per il veicolo e che non scenda al di sotto dello zero frenando. Questo è un tipico esempio di *incapsulamento*.

Una prima specializzazione della classe Vehicle, mediante *ereditarietà*, la si può pensare creando due classi derivate, che rappresentino rispettivamente i veicoli su strada e i veicoli che invece possono volare, chiamandole per esempio StreetVehicle e FlyingVehicle. Esse hanno in comune tutto ciò che è stato implementato nella classe Vehicle e ognuna di esse ha poi delle caratteristiche speciali tipiche della categoria di veicolo che rappresentano. La classe StreetVehicle, per esempio, può specializzare la classe Vehicle aggiungendo un campo che memorizza il numero di ruote, mentre un veicolo che vola avrà fra i suoi campi l'altitudine in volo e la direzione o rotta in gradi. Per implementare le due nuove classi si scriverà quindi:

```
public class StreetVehicle: Vehicle
{
private byte wheels; //numero di ruote
public byte Wheels
{
get{ return wheels;}
set{ wheels=value;}
}
}

public class FlyingVehicle: Vehicle
{
public double Altitude {get;set;} //altitudine in metri
public short DirectionDegrees {get;set;} //rotta in gradi
}
```

Sia la classe StreetVehicle sia FlyingVehicle ereditano dalla loro classe madre i membri non privati, cioè in questo caso i metodi pubblici.

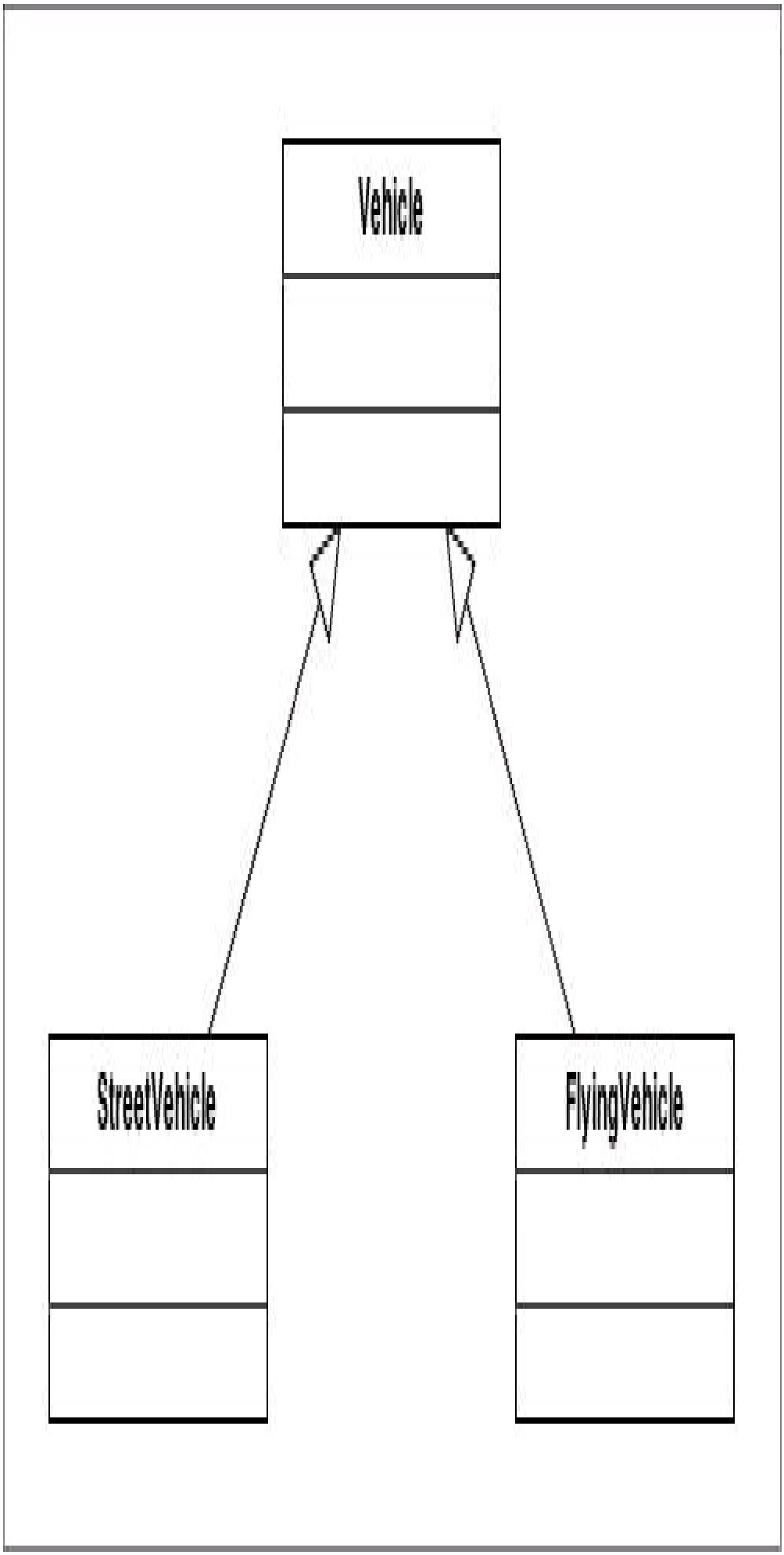

Vehicle
StreetVehicle
FlyingVehicle

Figura 7.2 – Diagramma delle classi derivate da Vehicle.

Quindi possiamo per esempio istanziarle e utilizzarle nel seguente modo:

```
StreetVehicle car=new StreetVehicle();
car.ModelName="Fiat Uno";
car.Wheels=4;
car.Accelerate();
car.PrintInfo();
```

La classe StreetVehicle eredita dalla classe Vehicle tutti i membri pubblici, come la proprietà ModelName e i metodi Accelerate e PrintInfo usati nell'esempio, che quindi sono utilizzabili anche per mezzo dell'istanza car.

L'approccio tramite ereditarietà permette quindi di riutilizzare il codice già scritto per la classe base Vehicle ed evitare di duplicarlo per aggiungere le stesse funzionalità a StreetVehicle e FlyingVehicle.

Il livello di profondità delle gerarchie di classi non è limitato e può estendersi a proprio piacimento, anche se naturalmente deve avere un senso e portare a una qualche utilità e non essere un semplice esercizio a chi riesce ad arrivare più in fondo.

Come esempio di ulteriore estensione, possiamo usare a loro volta la classe StreetVehicle per derivare una classe Car, che rappresenta l'automobile, e la classe FlyingVehicle con una Airplane, che rappresenta invece un aeroplano:

```
class Car: StreetVehicle
{
private double livelloCarburante;
private bool motoreAcceso;
public void Accendi()
{
if(!motoreAcceso && livelloCarburante>0)
motoreAcceso=true;
}
public void Spegni()
{
if(motoreAcceso)
motoreAcceso=false;
}

public Car()
{
Wheels=4;
}
}

class Airplane: FlyingVehicle
{
public void Decolla() {…}
public void Atterra() {…}
```

}

La classe Car estende la classe base aggiungendo due nuovi campi e due metodi, per gestire l'accensione e lo spegnimento dell'automobile. La classe Airplane, invece, aggiunge i metodi per il decollo e l'atterraggio.

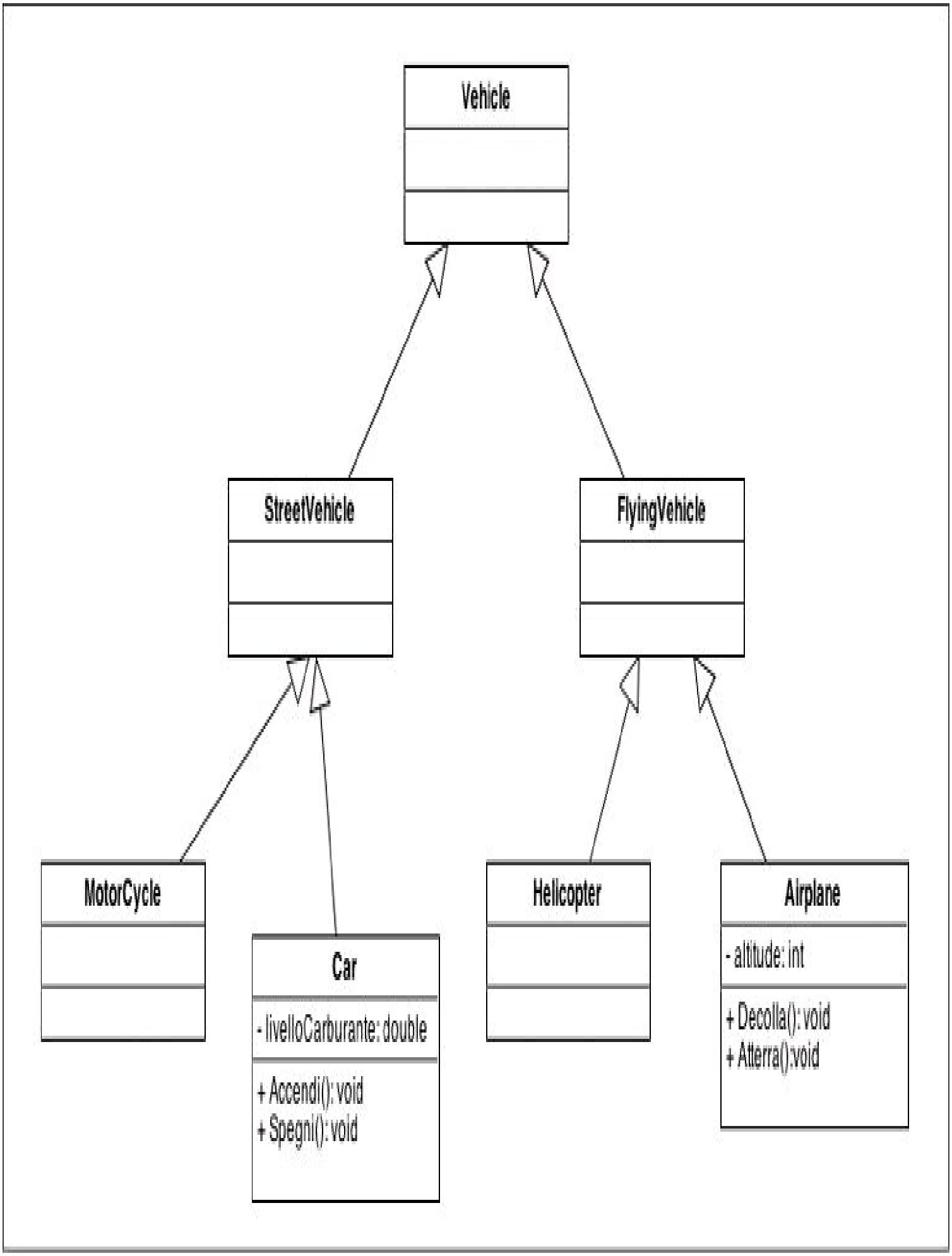

Vehicle
StreetVehicle
FlyingVehicle
MotorCycle
Car
- livelloCarburante: double
+ Accendi(): void
+ Spegni(): void
Helicopter
Airplane
- altitude: int
+ Decolla(): void
+ Atterra(): void

Figura 7.3 – Diagramma delle classi Airplane e Car.

Istanziando le due classi, possiamo ancora sfruttare il codice delle classi madre, sia di quelle direttamente correlate sia delle classi antenate più lontane:

```
Car car=new Car();
car.Accendi();
car.Accelerate(); //ereditato da Vehicle
car.Brake(); //ereditato da Vehicle
```

Come fatto nell'esempio, l'istanza di Car può usare sia la proprietà Wheels ereditata da StreetVehicle, impostandone a 4 il valore predefinito, sia i metodi Accelerate e Brake ereditati invece dall'antenata Vehicle, oltre naturalmente a quelli direttamente implementati.

Accessibilità dei membri

L'ereditarietà non altera il livello di visibilità dei membri delle classi coinvolte e quindi mantiene l'incapsulamento implementato nella classe base. Per esempio, la classe StreetVehicle, derivata da Vehicle, non può accedere ai campi privati come currentSpeed e numberOfPassengers:

```
StreetVehicle sv=new StreetVehicle();
sv.currentSpeed=10; //errore, campo privato di Vehicle non accessibile
```

Il codice precedente genera un errore di compilazione, in quanto il campo currentSpeed è private e quindi, come detto nel precedente capitolo, accessibile solo all'interno della classe in cui è definito. Non è possibile accedere ai membri private nemmeno direttamente all'interno della classe derivata. Provando, per esempio, a impostare il valore di currentSpeed all'interno del costruttore di StreetVehicle:

```
public StreetVehicle()
{
this.currentSpeed=0; //errore
}
```

l'errore di compilazione sarebbe il medesimo dell'esempio precedente, causato dal fatto che currentSpeed è inaccessibile, a causa del livello di protezione. Ciò conferma che gli unici membri utilizzabili all'interno delle classi derivate sono quindi al momento quelli public.

Se si vuol controllare in maniera più fine la visibilità dei membri da ereditare, è possibile utilizzare il modificatore di accesso protected. Il modificatore protected, infatti, limita la visibilità di un membro alle classi appartenenti a una catena di ereditarietà: se un membro di una classe è protected, esso sarà utilizzabile solo all'interno delle classi che derivano da essa.

L'accesso protected internal significa protected OPPURE internal, non protected E internal. In altre parole, un membro protected internal è accessibile da qualsiasi classe all'interno dello stesso

NOTA assembly, e dalle classi derivate, anche in altri assembly. Ecco perché spesso ci si riferisce a questa combinazione di modificatori chiamandola *proternal* o *intected*.
Per limitare l'accesso solo alle classi derivate presenti nello stesso assembly, si deve dichiarare la classe stessa internal e i relativi membri come protetti.

Modificando, per esempio, il modificatore di accesso del campo currentSpeed da private a protected, esso diverrebbe accessibile anche all'interno delle classi derivate:

```
class Vehicle
{
…
protected double currentSpeed;
…
}
```

Quindi, se ora si provasse a inizializzarlo all'interno della classe derivata StreetVehicle, tutto funzionerebbe alla perfezione:

```
class StreetVehicle: Vehicle
{
public StreetVehicle()
{
this.currentSpeed=0; //ok
}
}
```

Naturalmente protected limita l'accesso all'interno della gerarchia: esso non permette di utilizzare un membro all'infuori della classe che lo definisce o di quelle che lo ereditano. Utilizzando quindi correttamente i modificatori di accesso in una gerarchia di classi, si può controllare perfettamente l'incapsulamento e la composizione delle classi.

Classi sealed

Il livello di profondità della gerarchia può essere limitato, evitando che una classe possa essere ulteriormente specializzata. Se una classe utilizza il modificatore sealed, infatti, essa non può essere utilizzata come classe madre da nessun'altra classe. Per esempio:

```
sealed class Car: StreetVehicle
{
…
}
```

La classe Car deriva da StreetVehicle, ma con la presenza di sealed non può essere utilizzata per scrivere una classe derivandola da essa come la seguente:

```
class Supercar: Car //errore, Car è sealed
{
…
}
```

Il modificatore sealed evita che una classe possa essere specializzata e magari utilizzata per scopi non previsti. In genere si indica una classe come sealed quando si vuole limitare il suo utilizzo all'interno di una libreria, magari per motivi commerciali.

La libreria di classi di .NET Framework contiene molte classi sealed. Un esempio classico è la classe System.String: non è infatti possibile creare una classe di stringhe speciali che ereditino le funzionalità implementate dalla classe standard String.

Polimorfismo

All'interno di classi appartenenti a una gerarchia di ereditarietà possono essere definiti membri con lo stesso nome, in maniera che una classe derivata possa, per esempio, implementare lo stesso membro in maniera differente rispetto all'implementazione della classe madre. Un oggetto di tipo Car, per esempio, è anche uno StreetVehicle e ancora un Vehicle, quindi può assumere comportamenti differenti utilizzando i metodi delle due classi madre.

Tale possibilità di comportarsi in maniera differente è la caratteristica che abbiamo chiamato *polimorfismo* e che costituisce uno dei pilastri della programmazione orientata agli oggetti. Infatti, è possibile assegnare un oggetto di una sottoclasse a una variabile di una classe madre:

```
Vehicle v=new Car(); //un oggetto car è sempre un Vehicle
```

Quindi un oggetto di una classe figlia può sempre essere convertito implicitamente nel tipo di una classe madre; questa conversione è detta anche *upcasting*:

```
Car car=new Car();
Vehicle v=car;
```

Non è invece sempre vero il contrario: un veicolo non è detto che sia un'automobile, potrebbe anche essere un aeroplano. Tale conversione è anche detta *downcasting*:

```
Car car=new Car();
Vehicle v=car; //upcasting
Car car2=(Car)v; //downcasting
```

Un oggetto può essere convertito nel tipo di una sua sottoclasse solo se l'istanza rappresenta un oggetto della sottoclasse stessa, per questo è richiesto un cast esplicito:

```
Vehicle v=new Car(); //il veicolo è un oggetto Car
Airplane plane=(Airplane)v; //errore, non si può convertire Car in Airplane
```

Se, come nell'esempio precedente, la conversione non va a buon fine, viene generata un'eccezione di tipo InvalidCastException.

Nel Capitolo 4 abbiamo visto come convertire un oggetto di tipo riferimento in un altro tipo, per mezzo dell'operatore as. Utilizzandolo nel caso precedente, per eseguire un downcast, cioè una conversione da una classe madre verso una sottoclasse, esso restituirà un riferimento null nel caso in cui la conversione non sia possibile:

```
Vehicle v=new Car(); //il veicolo è un oggetto Car
Airplane plane=v as Airplane; //null, non si può convertire Car in Airplane
```

L'operatore is, invece, verifica se una conversione fra tipi riferimento è possibile. Con l'operatore is è quindi possibile verificare se un oggetto deriva da una classe madre (oppure,

come vedremo in seguito, se implementa un'interfaccia):

```
Car car=new Car();
if(car is Vehicle)
{
//true, perché Car deriva da Vehicle
}
```

Mediante tale operatore è possibile verificare anche il contrario, in maniera da essere certi di poter eseguire un downcast, per esempio con l'uso successivo dell'operatore as:

```
Vehicle v=new Car();
if(v is Car)
{
Car c=v as Car; //ok
}
if(v is Airplane)
{
Airplane ap=v as Airplane; //non entra in questo if
}
```

Nascondere i membri

Una sottoclasse può utilizzare i membri pubblici o protected definiti in una classe madre, oppure aggiungerne di nuovi per definire nuove funzionalità. Non può eliminare in alcun modo invece dei membri ereditati. Per esempio, all'interno della classe Vehicle è stato definito un metodo Accelerate, utilizzabile da tutte le sottoclassi, quindi anche da un'istanza Car o da una Airplane, che avranno il medesimo funzionamento:

```
Airplane airplane=new Airplane();
airplane.Accelerate();
Car car=new Car();
car.Accelerate();
```

È normale però che in ogni classe figlia si possa modificare il comportamento ereditato, in modo da specializzarlo in maniera adeguata. Un'automobile, per esempio, accelera in maniera diversa rispetto a un aereo, quindi il metodo Accelerate dovrebbe in teoria avere un'implementazione differente.

La prima cosa che vi sarà venuta in mente a questo punto è di implementare all'interno della classe figlia un metodo già presente nella classe madre, in maniera da adeguarne il comportamento alla classe stessa. Aggiungiamo quindi all'interno di Airplane un nuovo metodo Accelerate, con la stessa firma di quello presente nella classe madre Vehicle e con un'implementazione differente, per esempio che aumenti la velocità in maniera più rapida:

```
class Airplane: FlyingVehicle
{
public void Decolla() {…}
public void Atterra() {…}
```

```
public void Accelerate()
{
this.currentSpeed+=5;
if(currentSpeed>maxSpeed)
currentSpeed=maxSpeed;
}
}
```

Cosa succede ora invocando tale metodo su un'istanza di Airplane? Succede quello che ci si aspetta, cioè che l'istanza utilizzerà il proprio metodo Accelerate, nascondendo quello ereditato dalla classe madre; tale pratica, infatti, viene anche detta *name hiding* dei membri.

Nascondere dei membri ereditati non è un errore, l'unica particolarità vi viene segnalata dal compilatore mediante un warning: provando a compilare la classe con il metodo che nasconde quello ereditato, si riceve un avviso del tipo "il metodo Accelerate nasconde il metodo ereditato".

Quindi per indicare al compilatore che siamo al corrente di quello che stiamo facendo e che abbiamo volontariamente implementato un metodo con la stessa firma, è possibile utilizzare il modificatore new con il nuovo metodo:

```
class Airplane: FlyingVehicle
{
public void Decolla() {…}
public void Atterra() {…}

public new void Accelerate()
{
this.currentSpeed+=5;
if(currentSpeed>maxSpeed)
currentSpeed=maxSpeed;
}
}
```

NOTA	Nascondere un membro senza utilizzare il modificatore new è permesso e provoca solo un warning, perché potrebbe anche accadere che, al momento in cui alla classe figlia è stato aggiunto il nuovo membro, la classe madre utilizzata non possedeva ancora un membro con lo stesso nome. Esso potrebbe essere aggiunto in una nuova versione, successivamente all'implementazione della classe figlia.

In generale, la procedura di nascondere i membri di una classe con lo stesso nome segue delle regole ben precise, che dipendono dal tipo di membro:

- un membro costante, campo, proprietà, evento, o tipo innestato inserito in una classe o struct nasconde tutti i membri delle classi base con lo stesso nome;
- un metodo introdotto in una classe o struct nasconde tutti i membri diversi dai metodi con lo stesso nome e i metodi con la stessa firma (nome del metodo, numero e tipi di

parametri);

- un indicizzatore aggiunto a una classe o struct nasconde tutti gli indicizzatori ereditati con la stessa firma (cioè con stesso numero e tipi di parametri).

Quindi una classe figlia può implementare, per esempio, un metodo con nome maxSpeed, anziché un metodo che nasconde un campo della classe madre.

NOTA	Se un membro utilizza il modificatore new, ma non c'è nessun membro ereditato da nascondere, il compilatore restituisce un altro warning. Per risolverlo basta eliminare il modificatore new che in questo caso non è necessario.

Proviamo a fare qualche esperimento per capire cosa accade nascondendo dei membri e utilizzandoli in una classe derivata. La seguente classe ClasseBase implementa un metodo Test e una sua classe figlia ClasseDerivata lo nasconde mediante il modificatore new:

```
class ClasseBase
{
public void Test()
{
Console.WriteLine("ClasseBase.Test");
}
}

class ClasseDerivata
{
public new void Test()
{
Console.WriteLine("ClasseDerivata.Test");
}
}
```

Creando ora un'istanza di ognuna delle due classi, proviamo a invocare su di esse il metodo Test che entrambe espongono:

```
ClasseDerivata cd=new ClasseDerivata();
cd.Test();//Stampa ClasseDerivata.Test
ClasseBase cb=cd;
cb.Test(); //Stampa ClasseBase.Test
```

Il metodo Test che verrà utilizzato nei due casi dipende dal tipo dell'istanza a tempo di compilazione. Infatti, l'istanza cb, anche se le è stato assegnato un oggetto ClasseDerivata a runtime, utilizza il metodo Test della classe base, perché il tipo della variabile assegnato a tempo di compilazione è proprio ClasseBase.

NOTA	L'ordine di utilizzo dei modificatori con i membri è una questione di gusti o convenzioni; in generale il modificatore di accesso viene sempre mantenuto come primo, seguito dai modificatori di ereditarietà (new, sealed, override) e poi dal tipo.

Accesso ai membri base

Se si vuole utilizzare esplicitamente un membro della classe base, per esempio all'interno di un metodo della classe derivata che ne nasconde uno ereditato, è possibile utilizzare la parola chiave base. Il suo utilizzo è simile alla parola chiave this, solo che, anziché restituire l'istanza corrente, essa restituisce l'istanza convertita nel tipo della sua classe madre.

La parola chiave base può essere utilizzata all'interno di un costruttore, per invocare il costruttore ereditato (vedere il prossimo paragrafo) o all'interno di un metodo o proprietà di un'istanza, per invocare un qualsiasi membro della classe base.

Riprendendo il metodo Accelerate, che è definito nella classe Vehicle e poi nascosto dall'omonimo metodo della classe Airplane, all'interno di quest'ultimo possiamo per esempio accedere al metodo ereditato:

```
class Airplane: FlyingVehicle
{
…
public new void Accelerate()
{
while(currentSpeed<100)
base.Accelerate();
}
}
```

Naturalmente la parola chiave base può essere usata per accedere ai membri della classe madre da qualsiasi metodo della classe derivata, non solo dall'interno dello stesso metodo che nasconde quello ereditato. Per esempio, se all'interno del metodo Takeoff implementato qui di seguito si vuol utilizzare il metodo Accelerate della classe base, e non quello implementato all'interno della stessa classe Airplane, basta utilizzare esplicitamente la parola base:

```
class Airplane: FlyingVehicle
{
…
public new void Accelerate()
{
while(currentSpeed<100)
base.Accelerate();
}

public void TakeOff()
{
base.Accelerate(); //utilizza il metodo della classe base FlyingVehicle
Accelerate(); //utilizza il metodo della classe Airplane
}
}
```

In tal modo l'utilizzo di base è utile quando bisogna eseguire delle azioni comuni, quindi raggruppate in un metodo della classe madre, seguito da azioni aggiuntive e peculiari di ogni

classe figlia.

Costruttori della classe base

Una classe derivata può implementare i propri costruttori, mentre quelli della classe madre non sono automaticamente ereditati. Se, per esempio, la classe StreetVehicle possedesse un costruttore per inizializzare il numero di ruote, nel seguente modo:

```
public class StreetVehicle
{
protected byte wheels;

public StreetVehicle(byte wheels)
{
this.wheels=wheels;
}
}
```

la classe Car derivata da StreetVehicle non erediterebbe il costruttore appena definito, cioè non sarebbe possibile inizializzare un oggetto Car così:

```
Car car=new Car(4);
```

La parola chiave base è quindi spesso utilizzata all'interno di un costruttore della classe derivata, per invocare quelli della classe madre.

Si può dunque personalizzare il costruttore di default della classe Car per invocare il costruttore con il parametro byte della classe StreetVehicle:

```
public class Car:StreetVehicle
{
private int livelloCarburante;
public Car(): base(4)
{
livelloCarburante=100;
}
}
```

Il costruttore, prima di eseguire le proprie azioni, in questo caso l'inizializzazione del campo livelloCarburante, invoca il costruttore di StreetVehicle, passandogli il parametro 4.

Il funzionamento della parola base per invocare il costruttore è analogo a quello visto con this, solo che, anziché invocare un altro metodo della stessa classe, ne invoca uno della classe base.

Un'osservazione da fare riguarda il metodo predefinito senza parametri. Se una classe derivata definisce un costruttore personalizzato e non utilizza la parola chiave base per indicare il costruttore della classe madre da invocare, automaticamente verrà invocato il costruttore predefinito.

Quindi, nel caso in cui il costruttore senza parametri della classe madre non fosse accessibile, per esempio perché si è utilizzato il modificatore private o perché non esiste, avendo implementato un nuovo costruttore con parametri, sarà necessario utilizzare base per indicare quale costruttore della classe base adoperare:

```
public class StreetVehicle
{
protected byte wheels;

public StreetVehicle(byte wheels)
{
this.wheels=wheels;
}
}
```

La classe StreetVehicle ha un solo costruttore in questo caso. Quindi, se derivassimo da essa la classe Car con un costruttore:

```
public class Car: StreetVehicle
{
private int livelloCarburante;
public Car() //errore di compilazione
{
livelloCarburante=100;
}
}
```

si avrebbe un errore di compilazione, che ci avvisa del fatto che la classe base non ha un costruttore con zero parametri. In questo caso, infatti, il compilatore tenta di invocare implicitamente un costruttore di StreetVehicle senza parametri. Sarà dunque necessario indicare con base quale costruttore di StreetVehicle invocare:

```
public class Car: StreetVehicle
{
private int livelloCarburante;
public Car(): base(4)
{
livelloCarburante = 100;
}
}
```

Membri virtuali

Quando si implementa una gerarchia di classi è spesso possibile prevedere quali membri potranno essere ridefiniti nelle classi derivate, per fornire un'implementazione specializzata. Tali membri dovranno essere indicati con il modificatore virtual nella classe madre e sono quindi chiamati virtuali. Nelle classi derivate essi potranno essere ridefiniti utilizzando il modificatore override. Possono essere dichiarati come virtuali e quindi sottoposti a override, oltre ai metodi, anche le proprietà, gli eventi e gli indicizzatori di una classe madre.

Riprendiamo l'esempio della classe Vehicle con il metodo Accelerate. Dato che diverse classi potranno derivare da essa, ognuna potrà implementare un proprio metodo Accelerate, dal comportamento differente e specifico.

Un oggetto automobile, rappresentato da Car, dovrà implementare un metodo Accelerate differente rispetto a quello implementato per un aeroplano, rappresentato da Airplane. Poiché tale possibilità è prevedibile, all'interno della classe Vehicle possiamo indicare il metodo Accelerate come virtuale:

```
public class Vehicle
{
public virtual void Accelerate()
{
Console.WriteLine("Vehicle acceleration");
if(currentSpeed < maxSpeed)
currentSpeed += 1.0;
}
}
```

Nella classe derivata Car, se si vuol specializzare il metodo Accelerate, è necessario utilizzare il modificatore override:

```
public class Vehicle
{
private double livelloCarburante=100;
public override void Accelerate()
{
Console.WriteLine("Car acceleration");
if(livelloCarburante>0 && currentSpeed < maxSpeed)
{
currentSpeed += 2.0;
livelloCarburante-= currentSpeed/10;
}
}
}
```

A differenza del caso in cui il metodo della classe madre veniva nascosto mediante il modificatore new, qui il metodo utilizzato dipende dal tipo dell'istanza a runtime. Per esempio:

```
Car ferrari=new Car();
ferrari.Accelerate();
Vehicle sv=ferrari;
sv.Accelerate();
```

In entrambi i casi in cui viene invocato Accelerate, viene utilizzato il metodo implementato dalla classe Car, in quanto a runtime sia la variabile ferrari sia la variabile sv sono un riferimento a un'istanza di Car.

Quindi nell'invocazione di un metodo virtual, il tipo a runtime dell'istanza sulla quale avviene l'invocazione determina l'implementazione del metodo da utilizzare.

Invece, nel caso di un'invocazione di metodo non virtuale, il fattore determinante è il tipo al tempo di compilazione, cioè il tipo dichiarato della variabile.

Anche nei metodi di override è possibile invocare dei metodi della classe madre utilizzando la parola chiave base. Anzi, se si utilizza Visual Studio e si inizia a implementare l'override di un metodo (scrivendo la parola override e premendo spazio, l'IDE suggerisce un elenco di metodi virtuali), alla selezione del metodo da implementare con il mouse o con la pressione del tasto Invio verrà fornita un'implementazione predefinita, che invoca il metodo virtual corrispondente mediante base (vedi Figura 7.4).

```
class Supercar : Car
{
    public override
}
```


```
class Supercar : Car
{
    public override void Accelerate()
    {
        base.Accelerate();
    }
}
```

Figura 7.4 – Implementazione dell'override di un metodo in Visual Studio.

L'implementazione di *membri virtuali* e dei corrispondenti *override* è analoga per altri tipi di membri. Per esempio, per le proprietà:

```
public class Vehicle
{
public virtual double ConsumoKm
{
return 0;
}
}

public class Car: Vehicle
{
public override double ConsumoKm
{
return 10;
}
}
```

Altra osservazione da fare riguardo all'implementazione di metodi virtual/override è che in entrambi i casi bisogna mantenere lo stesso modificatore di visibilità:

```
public class Vehicle
{
public virtual void Accelerate()
{}
}

public class Car
{
protected override void Accelerate() //errore
{}
}
```

Il codice precedente non compilerebbe perché il metodo override è dichiarato come protected, mentre quello virtual è public.

Metodi sealed

L'override di un metodo virtual può avvenire a qualsiasi livello di ereditarietà. Se per esempio la classe Car, derivata da StreetVehicle, ha una sottoclasse, chiamata Supercar, quest'ultima può a sua volta eseguire l'override di un metodo virtual della classe StreetVehicle.

Riprendendo il caso precedente del metodo Accelerate, che in Car è un override del metodo virtual di StreetVehicle, è possibile implementare un ulteriore metodo di override nella classe Supercar:

```
class Supercar: Car
{
public override void Accelerate()
{
```

```
Console.WriteLine("Car acceleration");
if(livelloCarburante>0 && currentSpeed < maxSpeed)
{
currentSpeed += 4.0;
livelloCarburante-= currentSpeed/5;
}

}
}
```

Può capitare in alcuni casi che si voglia bloccare il livello a cui è possibile eseguire l'override di un metodo.

Qualche paragrafo fa abbiamo già visto che una classe può essere dichiarata come sealed, se si vuole evitare che da essa vengano derivate ulteriori classi figlie.

Se, invece di bloccare l'ereditarietà di un'intera classe, si vuol bloccare la possibilità di eseguire l'override di alcuni metodi virtuali in essa definiti, è possibile utilizzare il modificatore sealed per evitare che ulteriori override possano avvenire nelle classi derivate.

Quindi potremmo evitare che la classe Supercar implementi un suo override di Accelerate aggiungendo il modificatore sealed al metodo Accelerate di Car:

```
public class Car
{
public sealed override void Accelerate()
{}
}
```

Classi astratte

Lavorando con l'ereditarietà, capita spesso di implementare delle classi che non rappresentano oggetti del mondo reale ma che servono a raggruppare funzionalità comuni che poi saranno utilizzate dalle classi derivate. Nel mondo reale, per esempio, non esiste un oggetto veicolo volante, ma esisteranno elicotteri e aeroplani.

Quindi, in realtà, la classe FlyingVehicle, implementata negli esempi precedenti e utilizzata come classe madre per le classi derivate Airplane, rappresenta un concetto astratto e non deve poter essere istanziata come oggetto concreto.

Questi concetti sono presenti anche in molti linguaggi orientati agli oggetti e fra questi vi è C#.

Una classe che serve a raggruppare delle funzionalità da fornire alle sue classi figlie e che non dovrà essere istanziata mediante l'operatore new è chiamata una classe astratta.

Per definire tale tipologia di classi si utilizza il modificatore abstract prima della parola chiave class. Modifichiamo, per esempio, la definizione della classe FlyingVehicle:

```
public abstract class FlyingVehicle
```

```
{
…
}
```

In tal modo, tentando di creare un oggetto di classe FlyingVehicle mediante new:

```
FlyingVehicle fv=new FlyingVehicle(); //errore
```

il compilatore ci avviserebbe che non è possibile creare un'istanza di una classe astratta. Essa quindi potrà essere solo utilizzata come classe da cui derivare nuove classi, a loro volta astratte o concrete.

NOTA	Una classe astratta può anche derivare da una classe non astratta.

Metodi astratti

Una classe astratta può contenere normali metodi e costruttori, anche se non verrà mai istanziata direttamente (infatti, come visto in precedenza, un costruttore può essere invocato da una classe derivata), in maniera da fornire delle implementazioni pronte all'uso alle sue classi figlie.

Inoltre, una classe astratta può anche contenere dei metodi a loro volta astratti, che non possiedono un corpo con l'implementazione vera e propria, ma che indicano solo la firma:

```
public abstract class SpaceVehicle: FlyingVehicle
{
public abstract void Launch();
}
```

La classe SpaceVehicle rappresenta un oggetto volante spaziale astratto, praticamente un oggetto volante non identificato!

Il fatto che un metodo astratto, come l'unico metodo Launch di SpaceVehicle, non abbia un'implementazione, implica che essa dovrà essere necessariamente fornita da una classe derivata, quindi un metodo astratto è virtual per definizione:

```
public class SpaceShip: SpaceVehicle
{
public override void Launch()
{
Console.WriteLine("3...2...1...lift off!");
}
}
```

NOTA	Il modificatore virtual non può e non deve essere utilizzato insieme al modificatore abstract.

Una classe che deriva da una classe abstract deve quindi fornire un override dei metodi astratti della classe madre, a meno che essa stessa non rimanga abstract.

Anche le proprietà possono essere definite come astratte; in questo caso saranno i metodi di accesso get e set a non possedere alcuna implementazione, per esempio:

```
public abstract string AbstractProperty
{
get;
set;
}
```

Altri membri dichiarabili come abstract sono, oltre a metodi e proprietà, gli eventi e gli indicizzatori.

Una classe che contiene almeno un membro astratto deve necessariamente essere essa stessa una classe astratta. Non è vero il contrario, una classe astratta può anche non contenere metodi astratti: essa sarà semplicemente non istanziabile.

I metodi astratti o virtuali possono avere qualunque modificatore di accesso, escluso private (in tal caso non sarebbero infatti visibili alle classi figlie e quindi non si potrebbe implementarne il relativo override).

Interfacce

Un'*interfaccia* è un tipo riferimento che permette di definire una sorta di contratto. Una classe o struct che implementa un'interfaccia deve rispettare tale contratto. Le interfacce vengono quindi utilizzate per indicare le funzionalità che si vogliono implementare, lasciando alla classe o alla struct il compito di fornire l'implementazione vera e propria.

Per definire un'interfaccia si utilizza la parola chiave interface, al posto di class o struct, mentre i suoi membri non devono avere nessuna implementazione (similmente ai membri astratti).

Per convenzione il nome delle interfacce inizia per I maiuscola, seguito in genere da un sostantivo o da un aggettivo che indica un comportamento, scritto in Pascal Case:

```
public interface INomeInterfaccia
{
void Metodo();
int Prop{get;set;}
…
}
```

Una definizione di interfaccia può contenere zero o più membri, che possono essere metodi, proprietà, eventi e indicizzatori. Tali membri hanno implicitamente accesso public (in quanto interfaccia essi devono poter essere utilizzati esternamente) e sono astratti, quindi non si può e non si deve utilizzare alcun altro modificatore di accesso. Inoltre i membri di un'interfaccia non possono essere statici.

NOTA	Un'interfaccia senza membri viene spesso utilizzata per marcare una classe con una sorta di metadato ottenibile a runtime. Dato che in C# è possibile ottenere lo stesso risultato utilizzando gli attributi, è consigliabile evitare l'utilizzo delle interfacce senza membri.

Supponiamo, per esempio, di voler dotare gli oggetti Vehicle visti nei paragrafi precedenti di funzionalità per il controllo automatico della velocità. Possiamo definire un'interfaccia come la seguente:

```
public interface ICruiseControl
{
int? CruiseSpeed {get;};
void ResetCruise();
void SetCruise(int speed);
void StartControl();
}
```

L'interfaccia ICruiseControl permetterà di indicare una velocità di crociera da mantenere, mediante la proprietà CruiseSpeed, un metodo SetCruise, al quale passare la velocità da mantenere,

un metodo ResetCruise, che invece disattiva il controllo automatico, e un metodo StartControl per attivare la funzione di regolazione.

Implementazione di un'interfaccia

Per indicare che una classe implementa un'interfaccia si utilizza la stessa sintassi usata per derivare una classe da una classe madre. Quindi, se volessimo dotare la classe Car delle funzioni di ICruiseControl, dovremmo indicare che essa implementa l'interfaccia in questo modo:

```
class Car: ICruiseControl
{
//implementazione
}
```

La classe che implementa un'interfaccia deve necessariamente fornire l'implementazione di tutti i membri.

Una classe può derivare da un'altra e implementare delle interfacce. Per esempio la classe Car può, come visto in precedenza, estendere la classe StreetVehicle e implementare l'interfaccia ICruiseControl; basta indicare sempre per primo il nome della classe madre e poi le interfacce con la virgola come separatore:

```
class Car: StreetVehicle, ICruiseControl
{
…
private int? cruiseSpeed;
public int? CruiseSpeed
{
get { return cruiseSpeed; }
}
public void ResetCruise()
{
cruiseSpeed=null;
}
public void SetCruise(int speed)
{
cruiseSpeed=speed;
}
public void StartControl()
{
if (currentSpeed < cruiseSpeed)
{
while (currentSpeed < cruiseSpeed)
Accelerate();
}
else
{
while (currentSpeed > cruiseSpeed)
Brake();
}
}
```

```
}
```

Si noti che, per poter implementare i vari metodi, può essere necessario far ricorso alle altre funzionalità della classe o aggiungerne di nuove. Nell'esempio precedente, è stato aggiunto alla classe Car un campo cruiseSpeed, che viene incapsulato mediante la proprietà definita dall'interfaccia, mentre gli altri metodi si servono di Accelerate o Brake, già implementati nella classe.

Essendo un'interfaccia un tipo riferimento, è possibile ricavare un riferimento di tale tipo mediante un cast o una conversione con l'operatore as. Per esempio, se abbiamo una classe Car come nell'esempio precedente, che implementa un'interfaccia ICruiseControl, possiamo ottenere un riferimento all'interfaccia semplicemente scrivendo:

```
Car car=new Car();
ICruiseControl icc=car as ICruiseControl;
ICruiseControl icc2=(ICruiseControl)car;
```

È possibile quindi utilizzare i membri dell'interfaccia direttamente sui riferimenti all'interfaccia:

```
icc.SetCruise(90);
icc.StartControl();
```

Poiché una classe che implementa un'interfaccia può, a sua volta, essere derivata da una classe madre, l'implementazione dell'interfaccia può avvenire anche mediante dei membri ereditati. Per esempio, se una classe possiede già dei membri che corrispondono a quelli dell'interfaccia da implementare e questi sono visibili dalla classe figlia, quest'ultima non deve aggiungere nessun altro membro.

Ecco un esempio, con la classe Report che implementa già l'interfaccia IDocument; avendo un metodo Print, e quindi la sua classe derivata SubReport, per implementare a sua volta IDocument non deve aggiungere nient'altro:

```
interface IDocument
{
void Print();
}

class Report
{
public void Print()
{
Console.WriteLine("Print");
}
}

class SubReport : Report, IDocument
{
}
```

Interfacce multiple

Mentre una classe può derivare solo da una classe (perché l'ereditarietà fra classi è singola), essa può implementare più interfacce contemporaneamente, fornendo quindi le implementazioni dei metodi di tutte le interfacce.

NOTA	Poiché il meccanismo di rappresentare delle funzionalità che poi verranno implementate da una classe assomiglia molto al concetto di ereditarietà multipla, l'implementazione di interfacce da parte di una classe viene spesso chiamata *ereditarietà di interfaccia*, per distinguerla dall'ereditarietà fra classi.

```
public interface ISearchable
{
int Find(string text);
void Replace(string str, string strRep);
}
public interface IPrintable
{
void Print();
}
```

Se una classe vuole implementare entrambe le interfacce dovrà fornire l'implementazione dei due metodi di ISearchable e del metodo di IPrintable:

```
public class Document: ISearchable, IPrintable
{
public int Find(string text)
{
…
}
public void Replace(string str, string strRep)
{
…
}
public void Print()
{
…
}
}
```

Un'interfaccia può contenere decine di membri, quindi per implementarla in una classe possono essere richiesti molto tempo e attenzione (perché come detto una classe deve implementare tutti i membri definiti dall'interfaccia).

Più interfacce possono definire uno stesso membro, o meglio, un membro con lo stesso nome. Per esempio, supponiamo di avere due interfacce come le seguenti:

```
public interface IDocument
{
void Print();
```

```
}
public interface IPrintable
{
void Print();
}
```

Sia IDocument sia IPrintable definiscono un metodo Print e, se una classe vuole implementarle entrambe, il modo più semplice per farlo è di definire un unico metodo Start, che soddisferà entrambe le interfacce:

```
public class Document: IDocument, IPrintable
{
public void Print()
{
//implementazione Print
}
}
```

Ma cosa succede se, nonostante lo stesso nome, i metodi si riferiscono a due implementazioni completamente differenti? L'unico modo per gestire questa eventualità è quella di poter implementare due metodi distinti, indicando esplicitamente a quale interfaccia si riferisce ognuno di essi, come mostrato nel seguente paragrafo.

Implementazione implicita ed esplicita

Un'interfaccia può essere implementata in maniera *implicita* o *esplicita*. La prima modalità è quella già vista, che consiste semplicemente nell'aggiungere alla classe che implementa l'interfaccia tutti i membri pubblici indicati dall'interfaccia stessa.

La modalità esplicita, invece, indica esplicitamente anche il nome dell'interfaccia prima del membro.

Riprendendo l'esempio precedente, la classe Document può, per esempio, implementare due metodi Print esplicitando il nome dell'interfaccia e risolvendo così il conflitto di nomi:

```
public class Document: IDocument, IPrintable
{
void IDocument.Print()
{
…
}

void IPrintable.Print()
{
//implementazione Print
}
}
```

In tal modo, il modificatore di accesso non è più public e quindi, per poter accedere ai membri delle interfacce, bisogna utilizzare un riferimento all'interfaccia stessa:

```
Document doc=new Document();
```

```
doc.Print(); //errore non visibile
(doc as IDocument).Print(); //utilizza Print di IDocument
(doc as IPrintable).Print(); //utilizza Print di IPrintable
```

L'implementazione può anche essere *mista*, cioè implicita per alcuni membri ed esplicita per altri:

```
public class Document: ISearchable, IPrintable
{
int ISearchable.Find(string text) //implementazione esplicita
{
…
}
void ISearchable.Replace(string str, string strRep) //implementazione esplicita
{
…
}
public void Print() //implementazione implicita
{
…
}
}
```

In questo caso, solo il metodo Print dell'interfaccia IDocument sarebbe utilizzabile direttamente tramite un'istanza di Document, mentre per utilizzare gli altri due occorre un riferimento all'interfaccia ISearchable.

```
class Car : Vehicle, ICruiseControl
{
    protected double[] speed;
    private bool[] isCruiseOn;

    public void Accelerate()
    {
        base.Accelerate();
        speed[this.GetIndex()] += 5;
        if (isCruiseOn[this.GetIndex()])
        {
            Console.WriteLine("Car {0} is cruising at {1} km/h",
                this.GetIndex(), speed[this.GetIndex()]);
        }
        else
        {
            Console.WriteLine("Car {0} is accelerating to {1} km/h",
                this.GetIndex(), speed[this.GetIndex()]);
        }
    }
```

```
    public bool[] IsCruiseOn
    {
        get
        {
            return isCruiseOn;
        }
        set
        {
            isCruiseOn = value;
        }
    }
```

Figura 7.5 – Implementazione implicita o esplicita delle interfacce in Visual Studio.

Visual Studio fornisce un fondamentale aiuto per l'implementazione delle interfacce in quanto, dopo aver indicato il nome dell'interfaccia da implementare, basta utilizzare il solito menu contestuale e selezionare se implementare l'interfaccia implicitamente o esplicitamente.

Ereditarietà delle interfacce

Un'interfaccia supporta l'ereditarietà da altre interfacce ma, a differenza delle classi, dove l'ereditarietà è singola, possono anche essere specificate più interfacce di base.

L'interfaccia derivata eredita tutti i membri definiti dalle interfacce da cui deriva. Per esempio, se volessimo definire una nuova interfaccia, che specializzi la precedente ICruiseControl aggiungendo nuovi membri, potremmo scrivere:

```
public interface IAutoPilot: ICruiseControl
{
Coordinate Destination { get; set; }
void StartNavigation();
void StopNavigation();
}
```

L'interfaccia IAutoPilot aggiunge all'interfaccia ICruiseControl tre nuovi membri, quindi una classe che vorrà implementare l'interfaccia IAutoPilot deve implementare anche i membri di ICruiseControl.

Derivare un'interfaccia da più interfacce di base serve spesso a ottenere un'interfaccia con più membri, anche senza aggiungerne di nuovi e quindi, anche se in generale la pratica di scrivere interfacce senza membri non è comune, in questo caso è una scelta ragionevole:

```
public interface ISearchablePrintable: ISearchable, IPrintable
{
//eredita i membri di ISearchable ed IPrintable
}
```

Una classe eredita tutti i membri di un'interfaccia implementati da una sua classe madre:

```
interface IDocument
{
void Print();
}

class Report: IDocument
{
public void Print()
{
Console.WriteLine("Report.Print");
}
}

class SubReport : Report
{
```

```
}
```

La classe SubReport, per esempio, eredita il metodo Print, che implementa l'interfaccia IDocument.

Se SubReport volesse modificare il comportamento dei suoi membri, non essendo i membri di un'interfaccia marcabili come virtual, deve farlo mediante il modificatore new:

```
class SubReport : Report
{
public new void Print()
{
Console.WriteLine("SubReport.Print");
}
}
```

Il metodo Print in SubReport nasconde il metodo Print di Report, ma non altera il mapping del metodo Print. Se si invoca il metodo Print per mezzo di un riferimento all'interfaccia IDocument esso sarà ancora mappato sull'implementazione fornita da Report, anche se l'istanza a runtime è un SubReport:

```
Report report = new Report();
report.Print(); //stampa Report.Print

SubReport subReport = new SubReport();
subReport.Print();// stampa SubReport.Print

IDocument irep = report;
irep.Print(); //stampa Report.Print

IDocument isub = subReport;
isub.Print(); //stampa Report.Print
```

È però possibile implementare il metodo nella classe madre, in questo caso Report, come virtual e quindi eseguirne l'override nella classe derivata:

```
class Report: IDocument
{
public virtual void Print()
{
Console.WriteLine("Report.Print");
}
}
class SubReport : Report
{
public override void Print()
{
Console.WriteLine("SubReport.Print");
}
}
```

Questa volta ogni invocazione del metodo Print sarà mappata sull'implementazione del tipo assunto dall'oggetto a runtime e quindi, rieseguendo lo stesso esempio, l'effetto sarà il seguente:

```
Report report = new Report();
```

```
report.Print(); //stampa Report.Print
SubReport subReport = new SubReport();
subReport.Print();// stampa SubReport.Print

IDocument irep = report;
irep.Print(); //stampa Report.Print
IDocument isub = subReport;
isub.Print(); //stampa SubReport.Print
```

Invece i membri di interfaccia implementati in maniera esplicita non possono essere dichiarati come virtual, quindi non è possibile eseguirne l'override in una classe derivata.

Un possibile workaround è in questo caso quello di implementare un nuovo metodo virtual, che potrà essere invocato dal metodo che implementa l'interfaccia esplicitamente e che può essere sottoposto a override nella classe derivata:

```
class Report: IDocument
{
void IDocument.Print()
{
Console.Write("IDocument.Print ->");
this.Print();
}

protected virtual void Print()
{
Console.WriteLine("Report.Print");
}
}
class SubReport : Report
{
protected override void Print()
{
Console.WriteLine("SubReport.Print");
}
}
```

Invocando quindi il metodo Print tramite un riferimento all'interfaccia, verrà innanzitutto eseguito il metodo IDocument.Print della classe Report e poi il metodo implementato dal tipo reale ricavato a runtime:

```
IDocument irep = report;
irep.Print(); //stampa IDocument.Print -> Report.Print
IDocument isub = subReport;
isub.Print(); //stampa IDocument.Print -> SubReport.Print
```

Reimplementazione di interfacce

Una classe che eredita l'implementazione di un'interfaccia dalla classe madre può reimplementare l'interfaccia stessa, ridichiarandola nuovamente all'interno dell'elenco di classi e interfacce nella sua definizione.

Naturalmente valgono tutte le regole di ereditarietà viste nel paragrafo precedente. Per esempio, se ripartiamo dall'interfaccia IDocument e dalla classe Report già viste:

```
interface IDocument
{
void Print();
}

class Report: IDocument
{
void IDocument.Print()
{
Console.WriteLine("IDocument.Print");
}
}
```

possiamo derivare da Report una nuova classe SpecialSubReport, reimplementando l'interfaccia IDocument e quindi fornendo un nuovo metodo Print:

```
class SpecialSubReport : Report, IDocument
{
public void Print()
{
Console.WriteLine("SpecialSubReport.Print");
}
}
```

Anche in questo caso il metodo IDocument.Print viene correttamente mappato sul metodo Print di SpecialSubReport:

```
SpecialSubReport specsub=new SpecialSubReport();
IDocument ispecsub=specsub;
ispecsub.Print(); //stampa SpecialSubReport.Print
```

Classi astratte e interfacce

Sebbene classi *astratte* e *interfacce* condividano degli aspetti e si assomiglino nel fatto che entrambe costituiscono tipi non istanziabili direttamente, nulla vieta la scelta di utilizzarle contemporaneamente.

La differenza principale consiste nel fatto che una classe astratta, in quanto classe, può far uso solo di ereditarietà singola, quindi può derivare da una sola classe alla volta. Una classe astratta può implementare però una o più interfacce, come potrebbe fare una qualsiasi altra classe concreta, con la particolarità che può anche mappare alcuni o tutti i membri dell'interfaccia in membri astratti. Per esempio, se IDocument è l'interfaccia definita come segue:

```
interface IDocument
{
void Print();
void Save();
```

```
}
```

una classe abstract che la implementa può essere la seguente:

```
abstract class AbstractDocument: IDocument
{
public abstract void Print();
public virtual void Save()
{
//implementazione
}
}
```

Una classe concreta che deriva da AbstractDocument, infatti, dovrà necessariamente eseguire l'override del metodo Print e quindi implementare concretamente e interamente l'interfaccia.

L'altro metodo Save, invece, è implementato già nella classe astratta, ma lo si potrà eventualmente specializzare nelle classi derivate.

Si noti però che i membri che implementano in maniera esplicita l'interfaccia non possono essere abstract, ma tali membri possono in ogni caso invocarne altri che possono invece essere abstract:

```
abstract class AbstractDocument: IDocument
{
void IDocument.Print()
{
this.AbstractPrint();
}
public abstract void AbstractPrint();
}
```

Un'interfaccia definisce solo la firma dei membri, senza poter fornire alcuna implementazione. Una classe astratta, invece, in mezzo ai metodi astratti può anche implementarne altri non astratti, che le classi derivate potranno utilizzare o specializzare.

Una classe astratta, inoltre, può già prevedere dei campi per mantenere lo stato di un oggetto, mentre un'interfaccia può al massimo definire delle proprietà e, ancora una volta, senza nessuna implementazione concreta.

La scelta fra classi astratte e interfacce (una non esclude l'altra) si baserà quindi su tutta questa serie di vantaggi e svantaggi che una tipologia fornisce rispetto all'altra.

Utilizzo delle interfacce

La libreria di classi base di .NET fornisce diverse interfacce che sono implementate dai tipi della libreria stessa o che possono essere implementate da classi e struct personalizzate. Sapere come utilizzare le interfacce, sia dal punto di vista dell'implementazione sia da quello dell'utilizzo, è fondamentale per sfruttare al massimo tanto le librerie di base quanto quelle di

terze parti, oltre che per avvantaggiarsi realmente del paradigma orientato agli oggetti, in questo caso del polimorfismo.

Una delle interfacce più utilizzate della libreria standard è per esempio IComparable, che definisce un unico metodo CompareTo, che serve a confrontare l'istanza corrente con un altro oggetto dello stesso tipo e restituisce un numero intero che indica se l'istanza corrente precede, segue, oppure si trova nella stessa posizione di un altro oggetto.

Come ormai ben sappiamo, per implementare l'interfaccia basta indicarla nell'elenco delle classi di base e poi implementare i membri. Possiamo per esempio creare una classe ComparableCar che aggiunge alla classe base Car una proprietà Targa, che utilizzeremo per il confronto nel metodo CompareTo dell'interfaccia IComparable:

```
class ComparableCar: Car, IComparable
{
public string Targa{get;set;}
public int CompareTo(object obj)
{
if (obj is ComparableCar)
{
ComparableCar other = obj as ComparableCar;
return this.Targa.CompareTo(other.Targa);
}
return -1;
}
}
```

In questo caso, per implementare il metodo CompareTo, si è fatto uso del metodo CompareTo fornito da String, che infatti implementa l'interfaccia IComparable.

Le interfacce possono essere utilizzate come parametri di metodi, evitando di specificare una classe o struct particolare, in quanto, per esempio, il metodo indica solamente che vuole come proprio argomento un qualunque oggetto con dei particolari membri e non di un tipo specifico. Per esempio, potremmo definire un metodo:

```
public int ConfrontaOggetti(IComparable comp1, IComparable comp2)
{
int ord=comp1.CompareTo(comp2);
…}
```

ed esso potrà utilizzare come parametro qualunque oggetto che implementi l'interfaccia IComparable:

```
ComparableCar car1=new ComparableCar();
ComparableCar car2=new ComparableCar();
ConfrontaOggetti(car1,car2);
```

Altre classi utilizzano poi l'interfaccia IComparable internamente, per esempio la classe ArrayList può contenere una lista di elementi, che può essere ordinata mediante il suo metodo Sort:

```
ArrayList list = new ArrayList();
list.Add(new ComparableCar (){Targa = "CP787MC" });
list.Add(new ComparableCar (){Targa = "AB234GF" });
list.Add(new ComparableCar (){Targa = "ME123234" });
list.Sort();
```

Il metodo Sort utilizza internamente il metodo CompareTo per ordinare gli elementi della lista, quindi tali oggetti dovranno essere istanze di classi che implementano l'interfaccia IComparable.

Utilizzando un approccio basato sulle interfacce, il codice diventa più flessibile ed estendibile, in quanto non è necessario conoscere tutti i possibili tipi che implementano un'interfaccia, in nessun momento.

Inoltre esso è certamente più robusto, in quanto i metodi che utilizzano come parametri delle interfacce interagiranno solo con i membri dell'interfaccia stessa, che sono ben noti e definiti, senza aver necessità di conoscere gli altri membri eventualmente forniti dalla classe dell'oggetto.

Domande di riepilogo

1) Un metodo astratto come può essere utilizzato in una classe derivata?

a. La classe derivata deve implementare l'overload del metodo
b. Nella classe derivata non è visibile il metodo astratto
c. La classe derivata deve implementare l'override del metodo
d. Il metodo deve essere riscritto mantenendo la parola chiave abstract

2) Perché un membro di una classe base sia visibile nelle classi derivate, esso deve essere:

a. private
b. protected
c. public
d. nested

3) Dato un metodo definito in una classe base, per implementarne l'override in una classe derivata, si useranno rispettivamente:

a. sealed e override
b. override e override
c. virtual e override
d. virtual e overload

4) Quale affermazione è falsa?

a. Una classe può derivare da un'altra classe
b. Una classe può implementare più interfacce
c. Un'interfaccia può derivare da più interfacce
d. Una struct può derivare da una struct

5) Se una classe Dog deriva da una classe Animal, in che modo la prima può chiamare il costruttore della classe madre?

a. public Dog(): base()
b. public Dog(): Animal()
c. public Dog(): this()
d. public Dog()

{ base(); }

6) Per rendere un metodo M della classe C visibile solo alle classi derivate da C e presenti nello stesso assembly di C, si deve rendere:

a. la classe C internal e il metodo M public
b. la classe C protected e il metodo M public
c. la classe C protected e il metodo M protected
d. la classe C internal e il metodo M protected

7) Se una classe implementa due interfacce Interfaccia1 e Interfaccia2, con lo stesso metodo, per fare in modo che l'implementazione predefinita sia quella di Interfaccia1 si deve implementare:

a. Interfaccia1 e Interfaccia2 esplicitamente
b. Interfaccia1 esplicitamente e Interfaccia2 implicitamente
c. Interfaccia1 implicitamente e Interfaccia2 esplicitamente
d. non è possibile implementare due interfacce con uno stesso metodo

8) Una struct può implementare un'interfaccia. Vero o falso?

Gestione delle eccezioni

La gestione degli errori che possono verificarsi durante l'esecuzione di un programma è un aspetto fondamentale dello sviluppo di software. In .NET gli errori sono chiamati eccezioni e anch'essi sono istanze di particolari classi.

Durante il normale funzionamento e utilizzo di ogni applicazione possono verificarsi diverse condizioni di errore, imputabili a varie cause. Un'applicazione, per esempio, può smettere di funzionare o non riuscire a eseguire un determinato compito, a causa di un errore esterno o di sistema, come la caduta di una connessione o la mancanza di permessi per l'accesso a un file, oppure può anche capitare che l'errore sia un problema di programmazione vero e proprio, volontario o meno, come una divisione di un numero per zero o un ciclo while che non incontra mai una condizione di uscita.

Infine non sottovalutate i vostri utenti, troveranno sempre un modo di mandare in crash la vostra applicazione se non glielo impedite: se un campo di testo prevede l'inserimento di soli numeri, fate in modo che le lettere non siano accettate, altrimenti ci sarà sempre qualcuno che proverà a digitarle!

La gestione di tali errori e l'adozione delle opportune azioni preventive per evitare il loro verificarsi, oppure azioni di recupero una volta che l'errore si è manifestato, per esempio mediante messaggi di avviso, sono attività fondamentali da cui nessun programmatore deve esimersi.

Nella storia dei linguaggi di programmazione sono state adottate diverse modalità per gestire gli errori, più o meno produttive ed efficaci. Molti per esempio utilizzano dei *codici di errore*, restituiti dalle funzioni, per indicare se esse sono andate a buon fine o al contrario per far capire che cosa è andato storto.

In tal modo tocca allo sviluppatore esaminare ogni possibile valore o codice e intraprendere di conseguenza eventuali azioni correttive, eventualmente anche con una serie di if e di goto.

Ma è pur sempre libero di ignorarli, anzi spesso quello che succede è proprio questo, il programma va avanti finché può, nonostante gli errori, portando a funzionamenti inaspettati.

Il .NET Framework e il linguaggio C# forniscono un meccanismo di intervento strutturato, uniforme e, ancora una volta, basato su tipi, che è quello delle eccezioni.

Questo meccanismo permette di separare la parte di codice che identifica gli errori da quella che invece li gestisce. Inoltre, in generale, un'eccezione avvisa sempre del verificarsi di un errore e porterà all'interruzione del programma se non viene gestita, mentre deve essere esplicitamente ignorata se proprio si vuole farlo.

NOTA	La gestione strutturata delle eccezioni, come quella utilizzata in .NET, viene spesso detta *SEH*, che altro non è che l'acronimo di Structured Exception Handling.

Che cosa sono le eccezioni

In generale, un programmatore dovrebbe cercare di prevedere quali sono le porzioni di codice che sono sottoposte a possibili condizioni di errore, anche se è quasi impossibile poter pensare a tutti gli errori che possono verificarsi, quando possono avvenire e in quali circostanze.

In quanto evento non prevedibile, quindi, un'eccezione è una particolare condizione di errore che si verifica in casi straordinari.

In C# anche un'*eccezione* è un oggetto, cioè un'istanza di una classe, che contiene informazioni sui motivi che hanno portato all'errore e consente di risalire al punto e allo stato dell'applicazione in cui esso si è verificato.

Gli esempi scritti e provati finora hanno semplicemente ignorato la possibilità del verificarsi di un errore, anche se spesso abbiamo anche detto che una certa istruzione o metodo può provocare un certo tipo di eccezione.

Per provare a capire intanto cos'è e come si verifica un'eccezione, scriviamo un semplice metodo che esegue la divisione fra due interi:

```
public static int Divide(int x, int y) => x/y;
```

All'apparenza niente di complicato, il metodo deve semplicemente eseguire la divisione di un intero `x` per un intero `y`. Ma cosa succede se il numero intero `y` assume valore 0?

```
public static void Main()
{
int a=10;
int b=0;
int risultato= Divide(a,b);
Console.WriteLine("{0}",risultato);
}
```

Se provate a compilare ed eseguire il programma precedente mediante il compilatore a riga di comando, vi verrà restituito un errore, come mostrato nella Figura 8.1, e inoltre il programma smetterà di funzionare in maniera piuttosto brusca.

C:\temp\Eccezioni.exe

```
Eccezione non gestita: System.DivideByZeroException: Tentativo di divisione per
zero.
   in Eccezioni.Program.Divide(Int32 x, Int32 y)
   in Eccezioni.Program.Main()
```

Eccezioni

Eccezioni ha smesso di funzionare

Windows si è verificato un problema che impedisce il funzionamento corretto del programma. Se è disponibile una soluzione, verrà chiuso il programma e inviata una notifica automatica.

Debug | Chiudi programmi

Figura 8.1 – Eccezione non gestita dovuta a una divisione per zero.

Dal messaggio di errore si nota immediatamente che l'eccezione è un oggetto di tipo System.DivideByZeroException con un messaggio aggiuntivo altrettanto chiaro che indica che la causa dell'errore è stata "Attempted to divide by zero" (cioè un tentativo di divisione per zero).

NOTA	Si noti che in C# i tipi numerici a virgola mobile, float e double, permettono la divisione per zero, restituendo un risultato che rappresenta particolari valori come NaN, acronimo di Not-A-Number, oppure +Infinito e -Infinito.

Se invece provaste a eseguire lo stesso programma in modalità Debug in Visual

Studio, appena giunti alla riga di codice incriminata, cioè quella che tenta di eseguire la divisione per zero, l'IDE interromperebbe l'esecuzione e mostrerebbe una finestra con informazioni sull'eccezione appena verificatasi.

Inoltre viene evidenziata la riga che ha provocato l'errore ed è possibile, fra le altre cose, ottenere informazioni aggiuntive sull'oggetto DivideByZeroException: cliccando per esempio sul link View Detail, si aprirebbe un'altra finestra con i dettagli sull'oggetto e la possibilità di esaminarne le proprietà.

```
namespace Eccezioni
{
    class Program
    {
        private static int Divide(int x, int y) => x / y;

        public static void Main()
        {
            int a = 10;
            int b = 0;
            int risultato = Divide(a, b);
            Console.WriteLine("{0}",risultato);
        }

    }
}
```

DivideByZeroException was unhandled

An unhandled exception of type 'System.DivideByZeroException' occurred in Eccezioni.exe

Additional information: Tentativo di divisione per zero.

Troubleshooting tips:

Make sure the value of the denominator is not zero before performing a division operation.

Get general help for this exception.

Search for more Help Online...

Exception settings:

Break when this exception type is thrown

Actions:

View Detail...

Copy exception detail to the clipboard

Open exception settings

Figura 8.2 – Eccezione non gestita in Visual Studio.

L'esempio serve solo a far capire che cosa accadrebbe in caso di eccezione, dato che in questo caso la situazione è perfettamente gestibile a priori; per esempio, con un istruzione if potremmo semplicemente verificare se il numero da inviare come parametro y è diverso da zero e in caso contrario avvisare l'utente:

```
if(b!=0)
int risultato= Divide(a,b);
else Console.WriteLine("Impossibile dividere per zero");
```

Questa filosofia è proprio quella giusta: lo sviluppatore deve sempre evitare di incappare in situazioni pericolose e gestirle innanzitutto in maniera da evitare il verificarsi di eccezioni.

Ove possibile bisogna fare di tutto per evitare il verificarsi di tali errori; per esempio, se un'applicazione ha un'interfaccia grafica e prevede l'inserimento di un numero in una casella di testo, è bene controllare l'inserimento dei caratteri da parte dell'utente, in maniera da non permettergli di inserire lettere o altro e quindi risparmiandosi ulteriori controlli o gestione di errori in una fase successiva.

Ma se l'esempio precedente è molto semplice, proviamo un po' a complicarlo, per mostrare come dietro l'angolo vi siano sempre possibili fonti di errore. Anziché inizializzare direttamente le due variabili intere, cosa abbastanza inutile, supponiamo che la nostra applicazione di divisione debba ricevere i due numeri da riga di comando e quindi restituire il risultato della divisione, ove possibile, oppure un messaggio di errore. Mantenendo sempre il metodo Divide visto sopra, potremmo ora scrivere il metodo Main del programma in questo modo:

```
static void Main(string[] args)
{
int a = int.Parse(args[0]);
int b = int.Parse(args[1]);
if (b != 0)
{
int risultato = Divide(a, b);
Console.WriteLine(risultato);
}
else Console.WriteLine("impossibile dividere per zero");
}
```

Le variabili a e b vengono inizializzate convertendo i parametri inviati da riga di comando, quindi se la variabile b è diversa da zero si procede alla divisione, altrimenti viene stampato un avviso. Ma che cosa succede se avviamo il programma senza indicare i parametri da riga di comando o indicandone uno solo? Accedendo all'array per leggere i due elementi args[0] e args[1] si verificherebbe un'eccezione di tipo System.IndexOutOfRangeException, cioè accesso

agli elementi dell'array al di fuori del numero di elementi. Bisognerebbe quindi controllare che il numero di parametri sia almeno due con un ulteriore if:

```
if (args.Length == 2)
{
int a = int.Parse(args[0]);
int b = int.Parse(args[1]);
…
}
```

E a questo punto cosa succede se i parametri inviati da riga di comando non rappresentano due numeri interi? Anziché un errore di accesso all'array, o all'interno del metodo Divide, verrebbe generata un'eccezione FormatException al momento di invocare il metodo int.Parse, quindi ci toccherebbe verificare il formato dei parametri oppure utilizzare il metodo int.TryParse.

Possiamo pure fermarci qui. Si è certamente capito che l'errore o la situazione eccezionale è sempre dietro l'angolo, quindi sono necessari costrutti e meccanismi per poter in ogni caso gestire ed evitare di interrompere improvvisamente l'esecuzione dell'applicazione o di fornire all'utente risultati errati o impredicibili.

Gestire le eccezioni

La gestione strutturata delle eccezioni prevede e necessita di una serie di oggetti e attività.

Un'eccezione è un'istanza di una classe derivata da System.Exception, che viene creata quando una certa situazione eccezionale di errore si verifica. Ogni errore sarà quindi rappresentato da un oggetto di tali classi, siano essi errori a livello di sistema, oppure errori a livello applicativo che dipendono dalla particolare applicazione.

Esempi di eccezioni di sistema sono condizioni di errore generiche, come la divisione per zero vista prima, una condizione di memoria esaurita, oppure il tentativo di utilizzo di un oggetto il cui valore è null.

Invece, un errore specifico di un'applicazione potrebbe essere il tentativo di scrivere o leggere un particolare file di configurazione, il cui percorso non è valido, oppure il tentativo di aprire un url con connessione di rete assente e chi più ne ha più ne metta (purtroppo!).

Un'applicazione esegue ogni sua parte di codice normalmente, con la possibilità però di stare in guardia e controllare che un dato tipo di eccezione si possa verificare. Nel caso in cui tale tipo di errore si presenti nella porzione di codice controllata, l'applicazione può eseguire un altro blocco di codice, dedicato appositamente a gestire il tipo di eccezione incriminato. Per eseguire queste attività, C# mette a disposizione delle apposite parole chiave (vedi Tabella 8.1) che consentono di catturare, ma anche di generare, delle eccezioni.

Tabella 8.1 **- Parole chiave per la gestione delle eccezioni.**

Parola chiave	Descrizione
try	identifica un blocco di codice che può generare un'eccezione
catch	identifica un blocco di codice dedicato alla gestione di tutte o di un particolare tipo di eccezione
finally	identifica un blocco di codice da eseguire a prescindere dal verificarsi o meno di un'eccezione
throw	istruzione utilizzata per generare una nuova eccezione, o per rilanciare un'eccezione già verificatasi rimandando la sua gestione

Le istruzioni try e catch

Se una parte di codice può causare un errore, e quindi scatenare una particolare eccezione, si può utilizzare un'istruzione **try** per indicare che tale blocco deve essere monitorato. In

corrispondenza del blocco try dovrà esserci un'ulteriore parte di codice, indicata dall'istruzione **catch**, in cui verranno inserite le istruzioni necessarie a gestire le eventuali eccezioni che si dovessero verificare.

NOTA	Sebbene la gestione delle eccezioni, mediante istruzioni try e catch, sia del tutto analoga a quella di Java, C# non supporta le cosiddette eccezioni verificate (o checked), presenti in Java e altri linguaggi. In Java ogni metodo che può generare un particolare tipo di eccezione lo dichiara nella sua firma, quindi ogni parte di codice che utilizza tale metodo deve necessariamente utilizzare un blocco try/catch per gestire tale tipo di eccezione.

In generale, quindi, la sintassi di utilizzo delle istruzioni try e catch è:

```
try
{
//esecuzione di codice che può scatenare un'eccezione
}
catch
{
//codice da eseguire se si verifica un'eccezione
}
```

Il flusso del codice è il seguente: il codice nel blocco try viene eseguito normalmente e, se non si verifica nessuna eccezione, il blocco con la clausola catch viene completamente saltato. Se invece all'interno del blocco try si verifica qualche eccezione, allora l'esecuzione salta direttamente alla prima istruzione del blocco catch.

Vediamo un possibile programma che può generare delle eccezioni e controlliamolo con un blocco try/catch:

```
static void Main(string[] args)
{
try
{
Console.WriteLine("Inserisci il divisore a");
int a = int.Parse(Console.ReadLine());
Console.WriteLine("Inserisci il dividendo b");
int b = int.Parse(Console.ReadLine());
int ris = a / b;
Console.WriteLine("{0}/{1}={2}", a,b, ris);
}
catch
{
Console.WriteLine("si è verificato un errore");
}
}
```

In questo caso, le variabili a e b vengono inizializzate leggendo i valori dal prompt dei comandi. Possono verificarsi come prima diverse condizioni di errore, per esempio b

potrebbe essere zero, oppure i valori inseriti non essere convertibili in int.

In ogni caso il codice verrebbe interrotto al momento dell'errore e il controllo passerebbe al blocco catch, all'interno del quale viene stampato un generico messaggio.

Il blocco catch può però essere scritto in tre modalità differenti. Quello appena visto è il modo più generico, che permette di catturare qualunque tipo di eccezione si possa verificare nel try corrispondente.

Prestate comunque attenzione all'uso di catch molto generici (cioè che catturano generiche eccezioni di classe Exception) e che hanno in più il corpo di istruzioni vuoto, cioè che non eseguano alcuna azione, a meno che ciò non sia strettamente necessario e voluto:

```
try
{
//codice che può causare diverse eccezioni
}
catch
{
}
```

Tale pratica potrebbe portare all'introduzione di bug difficili da scovare, perché ogni eccezione verrebbe catturata e inoltre, non eseguendo alcuna istruzione di recupero, probabilmente non vi accorgereste nemmeno del fatto che si sia verificata.

Nella nota precedente abbiamo affermato che la gestione delle eccezioni di .NET non è stringente come quella di altri linguaggi che utilizzano un approccio più rigido, detto delle eccezioni verificate.

Ogni metodo può probabilmente provocare il verificarsi di un'eccezione, ma come facciamo a sapere quali tipi di eccezioni? La risposta è una delle più frequenti nel mondo della programmazione in generale: consultate la documentazione, in questo caso quella dell'SDK di .NET, che trovate anche online partendo dall'url msdn.microsoft.com.

Se un metodo può lanciare delle eccezioni, all'interno della sua pagina di documentazione sarà presente una sezione che elenca e descrive le eccezioni che possono verificarsi e le relative cause. Per esempio, il metodo Parse della classe int riporta un elenco con tre eccezioni (vedi Figura 8.3).

Int32.Parse Method (String)

.NET Framework 4.6 and 4.5 | Other Versions

Converts the string representation of a number to its 32-bit signed integer equivalent.

Namespace: System
Assembly: mscorlib (in mscorlib.dll)

Syntax

C# | C++ | F# | VB

```
public static int Parse(
        string s
)
```

Parameters

s

Type: System.String
A string containing a number to convert.

Return Value
Type: System.Int32
A 32-bit signed integer equivalent to the number contained in s.

Exceptions

Exception	Condition
ArgumentNullException	s is **null**.
FormatException	s is not in the correct format.
OverflowException	s represents a number less than MinValue or greater than MaxValue.

Figura 8.3 - Eccezioni generabili da un metodo nella documentazione del .NET Framework.

Catch specifici

L'istruzione catch può anche specificare il tipo di eccezione che si occuperà di catturare. In generale, quindi, la sua sintassi è la seguente:

```
catch(TipoEccezione [variabile])
{
//codice per gestire il tipo TipoEccezione
}
```

Le parentesi quadre messe attorno a variabile indicano che il nome della variabile stessa è opzionale e può essere anche specificato solo il tipo. Quindi, se per esempio volessimo eseguire azioni di recupero specifiche, come stampare un messaggio specifico, per il tipo di eccezione DivideByZeroException possiamo modificare l'esempio precedente così:

```
static void Main(string[] args)
{
try
{
Console.WriteLine("Inserisci il divisore a");
int a = int.Parse(Console.ReadLine());
Console.WriteLine("Inserisci il dividendo b");
int b = int.Parse(Console.ReadLine());
int ris = a / b;
Console.WriteLine("{0}/{1}={2}", a,b, ris);
}
catch(DivideByZeroException)
{
Console.WriteLine("non è possibile dividere per zero");
}
}
```

Nel caso però in cui l'eccezione generata fosse un'altra, con l'esempio precedente essa non verrebbe gestita da nessun blocco catch presente e, quindi, l'effetto sarebbe lo stesso che si otterrebbe se non ci fosse alcun try/catch.

Il blocco catch senza tipo di eccezione visto nel paragrafo precedente è quindi equivalente a scriverne uno per il tipo System.Exception, in quanto ogni altra classe di eccezioni è derivata da questa.

In quanto oggetti, le eccezioni hanno proprietà che possono essere utilizzate per ricavare maggiori informazioni su di esse. Quindi, una terza modalità di scrittura dell'istruzione prevede anche di ottenere l'istanza dell'eccezione generata e utilizzarne i membri:

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

```
try
{
Console.WriteLine("Inserisci il divisore a");
int a = int.Parse(Console.ReadLine());
Console.WriteLine("Inserisci il dividendo b");
int b = int.Parse(Console.ReadLine());
int ris = a / b;
Console.WriteLine("{0}/{1}={2}", a, b, ris);
}
catch(DivideByZeroException dex)
{
Console.WriteLine(dex.Message);
}
}
```

Nell'esempio precedente, se viene scatenata un'eccezione DivideByZeroException, all'interno del catch che si occupa di gestire tale tipo di eccezione, si utilizza anche una variabile che conterrà l'eccezione stessa. Quindi viene utilizzato l'oggetto memorizzato nella variabile dex per stamparne la proprietà Message, che contiene informazioni di tipo testuale sull'eccezione stessa. In questo caso viene stampato il testo "Attempted to divide by zero" (naturalmente se il sistema è in inglese).

La proprietà Message è molto importante ai fini del debug, perché contiene informazioni di tipo testuale che lo sviluppatore può utilizzare per rendersi conto dei motivi che hanno portato all'errore.

Nel paragrafo dedicato alla classe System.Exception sono esposte altre proprietà utili a capire il motivo e il punto del codice in cui una determinata eccezione si è verificata.

Catch multipli

Negli esempi precedenti si è vista la possibilità di catturare un particolare tipo di eccezione, indicandolo come parametro della clausola catch.

Come detto, però, all'interno di un blocco try, soprattutto molto complesso, sono molteplici le condizioni di errore che possono verificarsi e ognuna di esse magari richiede un trattamento diverso. È quindi possibile utilizzare più blocchi catch, specificando differenti tipi di eccezione per ognuno di essi (con o senza oggetto da utilizzare nel blocco per esaminare l'istanza dell'eccezione):

```
try
{
…
}
catch(DivideByZeroException dex)
{
Console.WriteLine(dex.Message);
}
catch(FormatException ex)
```

```
{
Console.WriteLine("formato inserito non valido");
}
```

Poiché non sempre è possibile conoscere quali tipi di eccezione si possono verificare in un dato blocco di codice, soprattutto se al suo interno si invocano anche altri metodi che a loro volta possono scatenare eccezioni, non è certamente pratico realizzare una catena di blocchi catch che catturino tutti i tipi di eccezioni possibili.

In pratica i blocchi catch vengono aggiunti solo per i tipi di eccezioni che si vogliono e si possono gestire localmente, ma se si vuole in ogni caso catturare un qualsiasi altro tipo non presente fra i catch specifici, si può aggiungere come blocco finale un blocco catch generico:

```
try
{
}
catch(DivideByZeroException ex)
{
}
catch(FormatException)
{
}
catch
{
//qualsiasi altro tipo di eccezione
}
```

L'importante, in questo caso, è posizionare il blocco generico alla fine dell'elenco: infatti, quando si verifica un'eccezione, l'elenco dei catch presenti viene passato in rassegna nell'ordine in cui sono scritti, fino a incontrarne uno che soddisfa il suo tipo.

In generale, quindi, vanno prima inseriti i blocchi catch specifici e a seguire quelli più generici (grazie all'ereditarietà, infatti, è anche possibile che una classe di eccezioni derivi a sua volta da un altro tipo), fino ad arrivare al blocco catch senza specifica di alcun tipo.

Per esempio, la classe System.IO.FileFormatException, che rappresenta un'eccezione dovuta al fatto che un file non rispetta il formato previsto per un certo tipo di dati, deriva da System.FormatException, che a sua volta deriva da System.SystemException e quest'ultima ancora da System.Exception.

Supponiamo quindi di avere il seguente blocco try/catch:

```
try
{
}
catch(SystemException ex)
{
}
catch(FormatException ex)
```

```
{
Console.WriteLine("il formato non è valido");
}
catch(FileFormatException ex)
{
Console.WriteLine("il formato del file non è valido");
}
```

I messaggi indicati nei catch del tipo FormatException e in quello di FileFormatException non verrebbero mai stampati, in quanto un'eccezione di tali tipi sarebbe gestita subito dal primo catch. L'esempio precedente, quindi, non verrebbe compilato correttamente e il compilatore ci informerebbe con un errore del tipo:

A previous catch clause already catches all exceptions of this or a super type ('System.SystemException')

Si noti inoltre come le variabili che rappresentano le eccezioni hanno scope limitato al relativo blocco catch; per questo è possibile utilizzare lo stesso nome ex in ogni istruzione catch.

Filtri di eccezioni

C# 6 ha introdotto la possibilità di aggiungere ai catch delle eccezioni dei filtri condizionali, utilizzando la parola chiave when, così da catturare l'eccezione solo se una certa condizione è verificata:

```
catch(TipoEccezione ex) when(CondizioneBooleana)
{
}
```

Per esempio, supponiamo di voler leggere il contenuto di una pagina web per mezzo del metodo DownloadString della classe WebClient. Questo può provocare una situazione imprevista, come pagina non trovata o un altro errore di rete, scatenando una

WebException:

```
try
{
string url="http://antoniopelleriti.it/file.txt";
using (WebClient client = new WebClient())
{
var str= client.DownloadString(url);
}
}
catch (WebException ex) when (ex.Status == WebExceptionStatus.ProtocolError)
{
Console.WriteLine(ex.ToString());
}
catch
{
Console.WriteLine("errore");
}
```

Il blocco catch sarà eseguito solo se la proprietà Status di ex assume il valore ProtocolError. In caso contrario sarebbe completamente ignorato e verrebbe eseguito il catch generico, se presente.

Con i filtri di eccezioni è possibile scrivere catch sullo stesso tipo di eccezione, specificando condizioni diverse, in maniera da eseguire un blocco catch piuttosto che un altro:

```
try
{
}
catch(System.Net.WebException ex) when(ex.Status== WebExceptionStatus.Timeout)
{
…
}
catch(System.Net.WebException ex) when(ex.Status == WebExceptionStatus.NameResolutionFailure)
{
…
}
```

Naturalmente, se più condizioni fossero verificate solo il blocco corrispondente alla prima sarebbe eseguito.

La condizione booleana può essere anche valutata con un metodo o una proprietà. Un utilizzo tipico, per esempio, è l'implementazione di un metodo Log che restituisca un valore bool, all'interno del quale analizzare il contenuto dell'eccezione e salvarlo magari su un database o inviarlo per email:

```
public bool Log(Exception ex)
{
//memorizza eccezione per esempio in un db
...
return false; //evita di gestirla nel catch
}
```

Il metodo Log può ora essere invocato in un filtro when:

```
catch (WebException ex) when (Log(ex))
{
throw; //se Log torna true l'eccezione viene rilanciata
}
```

In tal modo l'eccezione non verrà gestita dal catch, visto che il metodo Log restituisce false.

Da notare che i filtri sulle eccezioni non sono solo una scorciatoia sintattica, ma permettono, come nel caso del Log precedente di ricavare informazioni su un'eccezione che si è verificata, senza interferire con il normale flusso di esecuzione. Il metodo Log, infatti, può analizzare il contenuto dell'oggetto Exception e poi, restituendo false, fa in modo che l'eccezione continui a propagarsi come se il filtro non ci fosse.

In caso di gestione nel catch potrebbe accadere invece di alterare lo stacktrace, per esempio con un'istruzione throw ex.

L'istruzione finally

Il modo più completo di scrivere un blocco di gestione e cattura delle eccezioni consiste nell'aggiungere un ulteriore blocco mediante l'istruzione **finally**:

```
try
{
}
catch
{
}
finally
{
}
```

Il blocco finally viene eseguito in qualunque caso. Se all'interno del try non si verifica alcuna eccezione, il codice in esso contenuto viene completato, dopodiché il controllo passa alla prima istruzione del blocco finally.

Se invece si verificasse un'eccezione, l'esecuzione prima passerebbe all'interno di un blocco catch, che gestisce l'eccezione in oggetto, e poi al blocco finally.

Inoltre il blocco finally verrebbe eseguito anche se all'interno del try vi fosse un'istruzione di salto, per esempio un goto, un return o qualcosa del genere.

Nel caso in cui sia presente il blocco finally, è possibile anche omettere del tutto il blocco catch. Si può cioè scrivere un costrutto try/finally come il seguente:

```
try
{
}
finally
{
}
```

Il blocco finally viene utilizzato quando si vuol essere sicuri che una data porzione di codice venga eseguita in ogni caso. Per esempio, se nel try si utilizzano certe risorse che consumano molta memoria o che necessitano una sorta di chiusura, all'interno del finally si potrebbero eseguire le istruzioni necessarie; esempi classici possono essere il rilascio di un file oppure la chiusura di una connessione a un database. Per esempio, per leggere il contenuto di un file testuale è possibile utilizzare un oggetto StreamReader in questo modo:

```
StreamReader sr;
try
{
sr=File.OpenText(path);
string content = sr.ReadToEnd();
}
finally
{
```

```
if(sr!=null)
sr.Dispose();
}
```

Poiché per leggere un file si usano risorse non gestite, è necessario utilizzare il metodo Dispose per rilasciare tali risorse. Con un costrutto try/finally è possibile quindi assicurarsi che il metodo Dispose venga invocato alla fine dell'utilizzo dell'oggetto StreamReader. Tale costrutto è perfettamente equivalente all'utilizzo dell'istruzione using, che è possibile usare con oggetti che implementano l'interfaccia IDisposable per invocarne automaticamente il metodo Dispose al termine del blocco definito dall'istruzione stessa. Il precedente esempio è perfettamente identico al seguente:

```
using(StreamReader sr=File.OpenText(path))
{
string content=sr.ReadToEnd();
}
```

Una volta raggiunta la parentesi di chiusura del blocco, viene invocato implicitamente il metodo Dispose dell'oggetto istanziato con l'istruzione using.

Ricerca del catch

Un blocco di codice può contenere chiamate a metodi che a loro volta possono diramare il flusso di esecuzione in profondità.

Quando si verifica un'eccezione, il CLR cerca di trovare il blocco catch più vicino e adeguato a gestirla, che potrebbe anche non essere nello stesso blocco di esecuzione in cui l'eccezione si è originariamente verificata.

Supponiamo, per esempio, di eseguire la divisione dei due interi mediante un apposito metodo Divide, come fatto in precedenza:

```
int Divide(int a,int b)
{
return a/b;
}
```

Se il parametro intero b fosse pari a zero, si verificherebbe un'eccezione DivideBy-ZeroException all'interno del metodo Divide, che però non contiene alcun try/catch. In questo caso bisogna risalire lo *stack delle chiamate*, fino a trovare un catch adatto a gestire l'eccezione.

Se non si trova alcun catch adeguato, l'eccezione non viene gestita e il programma terminerebbe in modo anomalo, come visto in precedenza. Per esempio, supponiamo che il metodo Divide venga invocato a sua volta da un metodo Esegui, come il seguente:

```
void Esegui(int a, int b)
{
try
{
```

```
Divide(a,b);
}
catch(DivideByZeroException dex)
{
Console.WriteLine(dex.Message);
}
}
```

In questo caso l'eccezione verificatasi all'interno del metodo Divide verrebbe gestita dal catch presente nel metodo Esegui, che ha invocato il metodo Divide.

Try/catch/finally innestati

I blocchi try/catch/finally possono essere innestati anche all'interno di altri blocchi. Tale pratica può risultare utile quando si dovesse avere un ciclo già all'interno di un try e quindi, per evitare di interrompere l'intero ciclo a causa di un'eccezione, si inserisce un ulteriore costrutto try/catch all'interno del ciclo stesso.

Nel seguente esempio vengono eseguite una serie di divisioni per mezzo di un for e, se si verifica una eccezione DivideByZeroException, essa viene catturata dal catch interno al ciclo stesso. Eventuali altri tipi di eccezioni, invece, interromperebbero il for perché sarebbero gestite dal catch esterno:

```
int[] array = {2, 0, 32, 12, 7, 0, 8};
try
{
for(int i = 0;i<array.Length; i++)
{
try
{
Console.WriteLine("{0}/{1}={2}", i*i, array[i], i/array[i]);
}
catch(DivideByZeroException ex)
{
Console.WriteLine("Impossibile eseguire divisione, passo alla prossima");}
}
}
}
catch(Exception ex)
{
Console.WriteLine(ex.Message);
}
```

Anche un blocco catch o finally può contenere un ulteriore blocco try/catch/finally. Per esempio, si supponga di voler realizzare una sorta di log delle eccezioni, facendo inviare all'applicazione un'email o salvare un record in un database per ogni eccezione catturata. L'operazione di invio email o salvataggio potrebbe causare essa stessa un'altra eccezione, magari perché non c'è connessione di rete oppure perché il database non è raggiungibile, quindi in questo caso si dovrebbe semplicemente ignorare l'eccezione con un catch vuoto:

```
try
{
//codice
}
catch(Exception ex)
{
Console.WriteLine(ex.Message);
try
{
//invio email delle informazioni sull'eccezione
}
catch
{
//impossibile inviare email
}
}
```

Eccezioni non gestite

Se un'eccezione non viene gestita da nessun blocco catch adeguato, essa risale lo stack fino ad arrivare al metodo Main, quindi, se nemmeno qui c'è una clausola catch per gestirne il tipo, l'applicazione verrà terminata mostrando un messaggio di errore (come visto nella Figura 8.1).

Il .NET Framework fornisce però un modo per verificare se un'eccezione ha raggiunto il limite dello stack delle chiamate, senza essere gestita. In particolare la classe AppDomain fornisce l'evento UnhandledException, che il CLR utilizza per notificare l'applicazione quando appunto un'eccezione non viene gestita. Poiché ancora non è stato affrontato l'argomento eventi in C# (lo vedremo nel Capitolo 10), il codice seguente potrà risultare un po' criptico, ma non è niente di complesso:

```
static void Main(string[] args)
{
AppDomain.CurrentDomain.UnhandledException += OnUnhandledException;
//genero un'eccezione non gestita da nessun try/catch
throw new Exception("Eccezione non gestita");
}
private static void OnUnhandledException(object sender, UnhandledExceptionEventArgs e)
{
Console.WriteLine("Si è verificata un'eccezione non gestita: {0}",
(e.ExceptionObject as Exception).Message);
}
```

L'operatore += utilizzato nel codice precedente permette di associare al verificarsi dell'evento UnhandledException un metodo che si occuperà di gestirlo e che verrà invocato quando verrà lanciata l'eccezione della riga seguente nel metodo Main.

Il metodo OnUnhandledException riceve come argomento un oggetto la cui proprietà ExceptionObject rappresenta l'eccezione non gestita. Tale meccanismo consente di eseguire

eventuali azioni di pulizia e di log degli errori, perché in ogni caso la situazione non è più recuperabile: all'uscita di tale metodo l'applicazione verrà terminata.

La classe **System.Exception**

La classe madre di tutti i tipi di eccezioni è la classe System.Exception. È una classe molto semplice, che fornisce delle proprietà pubbliche, utili per ricavare informazioni e dettagli sull'eccezione che si è verificata.

Inoltre la classe può essere utilizzata come classe base per creare dei tipi di eccezioni personalizzati, come verrà mostrato a breve nel Paragrafo "Creare nuove eccezioni".

NOTA	Le classi principali derivate da Exception sono SystemException e ApplicationException, che rispettivamente rappresentano errori di sistema ed errori applicativi. La classe ApplicationException doveva servire come base per creare le proprie applicazioni personalizzate, ma dato che non forniva alcun valore aggiunto, tale pratica non è più utilizzata e anzi è sconsigliata.

Proprietà di Exception

Quando si utilizza un blocco catch specificando il tipo di eccezione e una variabile per ottenere l'istanza dell'eccezione che si è verificata, è possibile leggere le proprietà dell'oggetto ereditate dalla classe System.Exception. Le proprietà più importanti sono descritte nella tabella seguente.

Tabella 8.2 **- Le proprietà pubbliche della classe System.Exception.**

Proprietà	Descrizione
Message	messaggio di testo che descrive l'eccezione
StackTrace	rappresentazione in forma testuale dello stack delle chiamate, utile per capire dove si è verificata l'eccezione
InnerException	rappresenta l'eventuale eccezione interna che ha causato quella attuale
TargetSite	il metodo che ha causato l'eccezione, se disponibile
Source	ottiene o imposta il nome dell'oggetto o applicazione che ha causato l'eccezione
HelpLink	un collegamento al file guida che descrive l'eccezione
Data	dizionario di coppie chiave-valore con ulteriori informazioni sull'eccezione
HResult	un valore numerico codificato che viene assegnato a ogni specifica eccezione

Ecco un esempio che stampa alcune di queste proprietà in un blocco catch:

```
try
{
…
}
```

```
catch(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);
}
```

Le proprietà Message e InnerException, in particolare, possono essere specificate all'interno del costruttore di Exception, quando si costruisce una nuova istanza di un'eccezione.

Tipi di eccezioni comuni

La libreria di classi di base del .NET Framework contiene una serie di classi di eccezioni che vengono utilizzate (cioè si possono verificare) più frequentemente durante l'esecuzione di un programma, in quanto gestiscono le condizioni di errore più comuni. Esse possono inoltre essere utilizzate anche come classi da cui derivare delle eccezioni personalizzate.

La seguente tabella fornisce un elenco e descrizione di tali eccezioni.

Tabella 8.3 **- Le classi di eccezioni più comuni.**

Classe	Descrizione
ArgumentException	Uno degli argomenti di un metodo non è valido. Viene utilizzata come classe base per eccezioni di tale genere.
ArgumentNullException	Un argomento di un metodo è null, ma non è consentito per il metodo in esame.
ArgumentOutOfRangeException	Un valore utilizzato come argomento non è fra quelli consentiti o nell'intervallo consentito.
ArithmeticException	Classe utilizzata come base per le eccezioni di tipo aritmetico, di cast o di conversioni.
DivideByZeroException	Derivata da ArithmeticException, causata da una divisione intera per zero.
OverflowException	Un'operazione ha dato come risultato un overflow, per esempio si è superato il valore massimo consentito.
NotFiniteNumberException	Eccezione generata quando un valore a virgola mobile è un numero infinito o Not-a-Number (NaN).
FormatException	Un argomento non ha il formato corretto previsto.
ArrayTypeMismatchException	Un oggetto inserito in un array non è del tipo corretto.
IndexOutOfRangeException	Un indice utilizzato per accedere a una collezione è al di là della sua dimensione.

InvalidCastException	Indica un errore di conversione esplicita da un tipo all'altro, in genere perché i tipi non sono compatibili.
NullReferenceException	Fra le più comuni, si verifica quando si tenta di accedere a un membro di un oggetto, ma quest'ultimo è null.
OutOfMemoryException	Indica memoria esaurita quando si tenta di istanziare un nuovo oggetto con new.
StackOverflowException	Si verifica se lo stack è esaurito a causa di troppe chiamate di metodi in profondità, spesso a causa di ricorsioni.
TypeInitializationException	Si verifica se c'è un'eccezione nel costruttore statico di una classe ed essa non è gestita al suo interno.
InvalidOperationException	Un'operazione viene eseguita su un oggetto ma il suo stato attuale non lo consente.
IO.IOException	Classe base delle eccezioni di input/output.
IO.FileNotFoundException	Eccezione lanciata quando un file da utilizzare non esiste.
NotImplementedException	Il metodo utilizzato non è stato implementato.

L'istruzione **throw**

Un'eccezione, oltre a poter essere generata dal CLR a causa di un errore che si può verificare nel codice della libreria di base o di terze parti, può essere creata e lanciata anche da parte dello sviluppatore nel proprio codice. Per generare un'eccezione si utilizza l'istruzione throw seguita da un'istanza dell'eccezione:

```
throw OggettoEccezione;
```

Naturalmente l'istanza può anche essere creata al volo al momento di lanciarla. Per esempio, spesso viene utilizzata l'eccezione NotImplementedException all'interno di un metodo che deve ancora essere implementato:

```
public void Metodo()
{
throw new NotImplementedException ("devo ancora lavorarci!");
}
```

Il costruttore di Exception e delle classi da essa derivate in particolare permette di impostare le proprietà Message e InnerException.

Ogni eccezione personalizzata, come vedremo, potrebbe anche implementare nuove proprietà e quindi anche nuovi costruttori.

Rilanciare le eccezioni

Poiché l'istruzione throw usa come parametro un'eccezione, essa può anche essere utilizzata all'interno del blocco catch per rilanciare un'eccezione appena verificatasi e che si sta gestendo.

Gli scenari di utilizzo dell'istruzione throw possono essere quindi differenti. In genere, se l'eccezione si verifica in un metodo che non interagisce direttamente con l'utente, essa non viene gestita localmente e quindi viene rilanciata al metodo chiamante. Scrivendo, per esempio, una libreria di classi, è più che logico rilanciare eccezioni all'esterno, in maniera che sia lo sviluppatore che usa la libreria a intraprendere eventuali azioni di gestione delle eccezioni.

Per rilanciare l'eccezione corrente all'interno di una clausola catch non è necessario specificare alcun oggetto:

```
try
{
//codice che genera eccezione
}
catch(NullReferenceException ex)
{
```

```
throw;
}
catch
{
throw; //viene rilanciata l'eccezione corrente
}
```

Naturalmente, dato che si necessita di un'eccezione da considerare come quella corrente, l'istruzione throw senza parametri può essere utilizzata solo all'interno di un blocco catch.

Se il blocco catch contiene anche il nome della variabile, è possibile utilizzare tale nome per rilanciare l'eccezione a cui si riferisce:

```
try
{
//codice che genera eccezione
}
catch(Exception ex)
{
throw ex;
}
```

La differenza con l'istruzione throw senza parametro, vista prima, è che in questo caso viene perso lo stacktrace, cioè la descrizione del percorso che ha portato all'eccezione e all'eventuale blocco catch che si occuperà di gestire l'eccezione così rilanciata: essa apparirà come se fosse stata generata proprio nel punto in cui si trova l'istruzione throw. In generale è quindi consigliabile omettere la variabile e mantenere intatto lo stacktrace.

Un altro possibile scenario è quello in cui l'eccezione da gestire possiede anche delle proprietà aggiuntive, che permettono di distinguere le cause dell'errore e quindi trattare ogni caso in maniera differente.

Per esempio, il tipo WebException rappresenta gli errori che si verificano durante l'accesso alla rete, quindi la sua proprietà Status indica l'eventuale motivo di errore:

```
try
{
Console.WriteLine("metodo WebEx");
WebClient client = new WebClient();
client.DownloadFile("http://www.sitononesistente.org/nomedir/nomefile.txt", "C:\\temp\\test.txt");
}
catch (WebException ex)
{
if (ex.Status == WebExceptionStatus.NameResolutionFailure)
Console.WriteLine("file non esistente");
else throw;
}
```

La proprietà Status contiene un valore dell'enumerazione WebExceptionStatus che, nel caso di un indirizzo non valido (come il percorso usato nell'esempio), sarà pari a

NameResolutionFailure. In questo caso viene stampato un messaggio di testo, mentre gli altri casi vengono rilanciati.

Il rilancio dell'eccezione può essere utile anche quando non si vuol gestire l'eccezione ma si vuole, per esempio, creare una sorta di log dell'errore e procedere poi con la gestione vera e propria in un catch più esterno:

```
try
{
//codice che genera eccezione
}
catch(Exception ex)
{
Log(ex.Message);
throw;
}
```

Per mezzo di throw, l'eccezione rilanciata può anche essere di un tipo differente dall'eccezione originale, in genere più specifico e che dia maggiori dettagli sull'errore che si è verificato. Tale possibilità è utilizzata maggiormente quando si implementano le proprie classi di eccezioni personalizzate, che vedremo nel prossimo paragrafo:

```
try
{
//codice che genera eccezione DivideByZeroException
}
catch(DivideByZeroException ex)
{
throw new NotImplementedException("La divisione per zero non è stata implementata", ex);
}
```

Cambiando il tipo di eccezione, è sempre consigliato inglobare l'eccezione originale all'interno della proprietà InnerException; per farlo basta utilizzare il costruttore che accetta come parametro anche un oggetto eccezione, come nell'esempio precedente.

Creare nuove eccezioni

Ogni eccezione è un'istanza di una classe che deriva direttamente o indirettamente da System.Exception.

Sfruttando l'ereditarietà è quindi possibile creare le proprie classi di eccezioni personalizzate, che serviranno a generare e gestire delle particolari condizioni di errore che si possono verificare all'interno delle proprie applicazioni.

Per esempio, dato che la classe ArgumentException è molto generica e indica un generico errore che si è verificato perché una data operazione ha tentato di utilizzare un argomento non valido, sarebbe comodo avere delle indicazioni più precise sul perché si è verificato tale evento.

Potremmo quindi derivare una classe personalizzata da ArgumentException, in maniera da poterla catturare con un catch specifico e intraprendere azioni di recupero appropriate.

Naturalmente è possibile utilizzare una classe di eccezione qualunque come classe madre per le eccezioni personalizzate. Le linee guida di Microsoft, però, sconsigliano la creazione di gerarchie di eccezioni troppo profonde e consigliano al massimo di derivare le proprie eccezioni da una delle classi più comuni, come quelle mostrate nella Tabella 8.3 del paragrafo precedente. Per esempio, se si tratta di un tipo di eccezione che ha a che fare con l'input/output la si potrebbe derivare dalla classe IOException.

NOTA	La convenzione utilizzata per creare classi di eccezioni personalizzate suggerisce di terminare il nome con il suffisso Exception.

Riprendendo gli esempi sulla gerarchia di classi per la gestione di veicoli, supponiamo di voler implementare una nostra eccezione personalizzata, che si verificherà quando un oggetto veicolo non è utilizzabile perché rimasto senza benzina. Creare una classe di eccezioni personalizzata è molto semplice, basta derivarla da un altro tipo di eccezione:

```
class NoFuelException: Exception
{
}
```

Il codice precedente potrebbe essere sufficiente, dato che la classe base fornisce il costruttore predefinito per istanziarla e le proprietà necessarie per poterla esaminare in un blocco catch:

```
try
{
throw new NoFuelException();
```

```
…
}
catch(NoFuelException)
{
…
}
```

NOTA	Il CLR consente di lanciare come eccezione un qualunque oggetto di una qualunque classe, anche un Int32 o una string. Microsoft ha però deciso che, all'interno di un linguaggio di programmazione che rispetti le regole CLS (Common Language Specification) di interoperabilità, ogni eccezione venga derivata da System.Exception.

È consigliabile dotare le classi di eccezioni personalizzate almeno dei tre costruttori seguenti, che sono i più comunemente utilizzati per costruire un'istanza:

```
class NoFuelException : InvalidOperationException
{
public NoFuelException():
{}

public NoFuelException(string message): base(message)
{}

public NoFuelException(string message, Exception inner): base(message, inner)
{}
}
```

In questo caso, l'eccezione deriva da InvalidOperationException (ma non cambierebbe nulla nel suo funzionamento se la derivassimo da Exception). Ogni costruttore invoca poi il costruttore della classe madre, mediante l'istruzione base.

Riprendendo la classe Car, derivata da MotorVehicle, si può ora, per esempio, lanciare un'eccezione NoFuelException quando si tenta di invocare il metodo Accelerate, ma il veicolo ha terminato la benzina, cosa che, in codice C#, sarà indicata da una proprietà LivelloCarburante che ha raggiunto il valore 0:

```
class MotorVehicle
{
public string Targa { get; set; }
public double LivelloCarburante { get; set; }
}
class Car: MotorVehicle
{
public Car(string targa)
{
Targa = targa;
}
public void Accelera()
{
if (LivelloCarburante == 0)
```

```
throw new NoFuelException("Il livello carburante è 0");
else
{
Console.WriteLine("vroooom!!!");
LivelloCarburante-=1.0;
}
}
public void Spegni()
{
Console.WriteLine("Stop!");
}
}
```

Nulla vieta di raffinare le proprie eccezioni personalizzate aggiungendo altre proprietà, metodi, costruttori. Per esempio si potrebbe aggiungere alla classe NoFuelException una proprietà in cui memorizzare il veicolo che ha scatenato l'eccezione e magari un costruttore per valorizzare direttamente la suddetta proprietà:

```
class NoFuelException : InvalidOperationException
{
//altri costruttori
//…
public MotorVehicle Vehicle{get;set;}
public NoFuelException(MotorVehicle vehicle): base("non c'è più benzina")
{
Vehicle = vehicle;
}
}
```

In tal modo, il metodo Accelera potrebbe lanciare l'eccezione passando al suo costruttore l'istanza del veicolo rimasto a secco:

```
public void Accelera()
{
if (LivelloCarburante == 0)
throw new NoFuelException(this);
else
{
Console.WriteLine("vroooom!!!");
LivelloCarburante-=1.0;
}
}
```

In un blocco catch è quindi ora possibile anche recuperare il veicolo che ha causato l'eccezione e usarne le relative proprietà:

```
static void Main(string[] args)
{
Car car = new Car("CP787MC") {LivelloCarburante=10.0};
try
{
car.Accelera();
}
```

```
catch (NoFuelException ex)
{
Console.WriteLine("Il veicolo targato {0} non può accelerare: {1}", ex.Vehicle.Targa, ex.Message);
}
finally
{
car.Spegni();
}
}
```

Prestazioni ed eccezioni

Una volta compreso il meccanismo delle eccezioni e la loro perfetta integrazione nel mondo Object Oriented di C#, si potrebbe essere tentati di utilizzare blocchi try/catch e implementare le proprie gerarchie di eccezioni anche per casistiche che non lo necessiterebbero.

In realtà, una volta fatta un po' di esperienza, ci si accorge che il lancio di un'eccezione e il relativo catch ha un impatto negativo sulle prestazioni di un'applicazione.

Quindi cercate di evitare l'uso esagerato di eccezioni e, soprattutto, progettate e sviluppate il vostro codice in maniera da trattarle appunto come eventi assolutamente eccezionali. Ove è possibile, controllate in anticipo se un blocco di codice è eseguibile senza errori. L'esempio della divisione per zero o della conversione di una stringa in numero calza alla perfezione.

Evitate di usare un blocco try/catch per gestire queste eventualità come fossero casi d'uso normali. È molto meglio verificare, per esempio, che una stringa rappresenti un numero prima di tentare di convertirla, o che una connessione a un database sia aperta prima di tentare di eseguire delle query su di esso. Nel primo caso, per esempio, Microsoft ha pensato bene di implementare appositi metodi del tipo int.TryParse, double.TryParse e così via. Anziché scrivere del codice che tenta di convertire una stringa in intero e ne gestisce eventuali eccezioni, come quello di seguito:

```
string str=Console.ReadLine();
int i;

try
{
i=int.Parse(str);
//esegui un'operazione con i
}
catch(FormatException)
{
//gestione eccezione
}
```

è possibile evitare i problemi di performance che si verificherebbero in caso di eccezioni mediante l'uso del seguente codice, dal punto di vista funzionale equivalente al precedente:

```
string str=Console.ReadLine();
int i;
if(int.TryParse(str, out i)
{
//esegui un'operazione con i
}
```

Naturalmente, l'eccezione FormatException non è l'unica che potrebbe verificarsi in un blocco di codice come il precedente, quindi un try/catch potrebbe in ogni caso essere necessario per gestire altre condizioni di errore.

Domande di riepilogo

1) Quale affermazione è vera?

a. Un blocco try/catch/finally può essere innestato in un altro blocco try/catch/finally
b. Il blocco finally viene eseguito se non si verifica un'eccezione
c. Possono esistere più blocchi finally specificando diversi tipi di eccezioni
d. Viene sempre eseguito il blocco catch che specifica il tipo di eccezione più specifico

2) Per impostare una condizione booleana che selezioni il blocco catch da eseguire, si scrive:

a. catch(Exception) if (condizione)
b. catch(Exception) when (condizione)
c. if(condizione) catch(Exception)
d. when(condizione) catch(Exception)

3) Rilanciando un'eccezione ex con l'istruzione throw ex:

a. lo stacktrace viene azzerato e riparte proprio dall'istruzione throw
b. lo stacktrace viene mantenuto, a differenza di un'istruzione throw senza ex
c. viene generata e lanciata una nuova istanza dell'eccezione
d. il messaggio dell'eccezione viene azzerato e impostato a quello predefinito del tipo di eccezione

4) Il codice di un blocco finally:

a. viene eseguito in caso di eccezione
b. viene eseguito solo se non si verifica un'eccezione
c. viene eseguito solo se non c'è un catch specifico dell'eccezione che si è verificata
d. viene eseguito in qualunque caso

5) Se si usano dei filtri when per le eccezioni, lo stesso tipo di eccezione può apparire su più blocchi catch. Vero o falso?

6) Quale evento può essere utilizzato per catturare eccezioni non gestite con blocchi try/catch?

a. UnexpectedException della classe AppDomain
b. UnhandledException della classe AppDomain
c. UnhandledException della classe Application
d. SystemException della classe Application

7) Quali costruttori vengono in genere implementati per una classe di eccezione MyEx personalizzata?

a. MyEx()
b. MyEx(string message)
c. MyEx(string message, InnerException inner)
d. MyEx(InnerException inner)

8) Il verificarsi di un'eccezione ha un impatto negativo sulle prestazioni di un programma. Vero o falso?

Tipi generici e collezioni

La maggior parte delle applicazioni ha bisogno di manipolare collezioni di oggetti in memoria. Il .NET Framework fornisce una varietà di classi progettate per tale obiettivo, mentre l'utilizzo di tipi generici consente di parametrizzare il codice su collezioni e tipi differenti.

La manipolazione di liste o elenchi di elementi è un'attività che ogni sviluppatore si troverà presto ad affrontare. Molti programmi necessitano di interagire con molteplici elementi, scorrendo per esempio dati e informazioni per effettuare su di essi calcoli o altre operazioni. Basti pensare a un qualunque social network, dove si ha a che fare con liste di amici, liste di messaggi, gruppi di pagine e così via.

Prima dell'avvento dei tipi generici, introdotti con .NET 2.0, la gestione di queste liste di oggetti doveva necessariamente utilizzare delle classi che potessero contenere un tipo qualunque di elementi, quindi la scelta di System.Object era imprescindibile come tipo base. Le classi e le interfacce del namespace System.Collections possono infatti trattare varie collezioni di oggetti dalla struttura diversa, come liste, code, dizionari ecc., ma tutte quante memorizzano elementi considerandoli di tipo object. Questo modo di agire, però, causa una serie di problematiche (per esempio la necessità di convertire gli oggetti di una collezione da e verso System.Object). Si è pertanto arrivati all'introduzione di tipi, detti *generici*, che possono utilizzare a loro volta un altro tipo come parametro.

In tale ambito, com'è logico, l'implementazione di una collezione che può lavorare con un qualunque tipo di elementi, fissandolo a priori, è collegata al concetto di tipi generici, tanto che il .NET Framework ha introdotto un altro namespace dedicato proprio alle nuove collezioni: System.Collections.Generic.

Per tale motivo gli argomenti “generics” e “collections” sono da considerarsi legati fortemente l’uno all’altro e sono trattati entrambi in questo capitolo.

Che cosa sono i generics

C# ha due differenti modi per creare nuovi tipi riutilizzando del codice già scritto. Uno l'abbiamo sviscerato a fondo parlando di ereditarietà: una classe che deriva da un'altra, per esempio, può ereditarne e utilizzarne i membri.

Il secondo meccanismo, introdotto con la versione 2.0 del linguaggio C# e di .NET, è quello dei *generics*.

Essi introducono in .NET Framework il concetto di parametri di tipo, cioè la capacità di creare classi e metodi all'interno dei quali la specifica di uno o più tipi da essi utilizzati viene rinviata fino a quando non sarà necessario creare un'istanza della classe o invocare un metodo. In parole povere, con i generics, C# fornisce la potenzialità di creare dei *template* di tipi, che potranno essere parametrizzati con tipi differenti.

Per comprendere l'utilità e la potenzialità di tali concetti, supponiamo di dover trattare liste di oggetti.

Prima dell'avvento dei generics, per creare tali liste era necessario utilizzare una classe come ArrayList che può contenere elementi di un qualunque tipo di oggetto, in quanto ereditato da System.Object. Vi era però una problematica possibile: per caso o per errore, poteva anche accadere che all'interno di un'unica collezione ci si ritrovasse con istanze di tipi differenti. E non solo: una volta ricavato un oggetto dalla lista, per utilizzarlo sarebbe stato sempre necessario effettuare un cast verso un particolare tipo, con la possibilità che tale conversione non andasse a buon fine, o comunque con l'obbligo di verificare se la conversione stessa fosse possibile, con conseguente degrado delle prestazioni.

Una seconda possibilità, che fra l'altro era una pratica abbastanza comune, consisteva nel creare una collezione fortemente tipizzata, implementando una nuova classe derivata per esempio da una classe astratta CollectionBase. Essa fornisce le funzionalità necessarie a lavorare con liste di oggetti, ma non evita il dover duplicare il codice ogniqualvolta si lavori con collezioni di oggetti di un nuovo tipo.

Per esempio, dovendo all'interno di un'applicazione gestire elenchi di clienti e fornitori, liste di ordini e fatture, insiemi di articoli e prodotti, ci si sarebbe presto ritrovati a implementare una nuova classe per ogni tipo di oggetto da collezionare.

I tipi generici permettono, fra le altre cose, di semplificare la gestione di tali collezioni di oggetti, consentendo, come vedremo, di utilizzare collezioni di un tipo qualunque che chiameremo T e che è sufficiente specificare solo al momento della creazione della collezione

stessa, come se fosse un parametro (e in effetti lo è; come abbiamo visto, anche un tipo è rappresentato alla fin fine da un'istanza della classe Type). I generics però non si limitano alle collezioni: anche una classe, una struct o un'interfaccia possono essere implementate come tipi generici, creando quindi un tipo che può essere istanziato e utilizzato in maniera differente a seconda di uno o più parametri, costituiti da altri tipi.

Anziché creare classi e metodi che funzionano solo con particolari tipi, è possibile ottenere la stessa funzionalità su tipi differenti e senza la necessità di implementare versioni di classi e overload di metodi differenti per ogni tipo di dato da trattare.

I generics permettono di definire tipi e metodi generici, per i quali è possibile specificare dei parametri di tipo. Tali parametri sono dei segnaposto, che verranno sostituiti dai nomi dei tipi esatti da utilizzare al momento della creazione di un oggetto di tale tipo generico o al momento dell'invocazione di un metodo generico. In questo modo, all'interno del tipo o del metodo generico, potranno essere implementati algoritmi che funzionano su diversi tipi di oggetti.

C# permette di creare cinque categorie di generics: classi, struct, interfacce, delegate e metodi.

NOTA	Non abbiamo ancora affrontato i delegate (li vedremo nel prossimo capitolo), ma intanto pensate a loro come a un'altra categoria di tipi implementabili in C#.

Senza i generics

Per mostrare alcuni dei vantaggi dei generics, e i problemi che essi permettono di evitare, vedremo ora un paio di esempi del loro utilizzo.

Supponiamo di voler creare una lista di interi, utilizzando la classe ArrayList che, come vedremo alla fine di questo stesso capitolo, rappresenta una raccolta di oggetti di tipo qualunque.

```
ArrayList lista=new ArrayList();
lista.Add(1);
lista.Add(2);
```

La classe ArrayList può memorizzare al suo interno qualunque tipo di oggetto utilizzando la classe base System.Object. Quindi, ogni volta che si utilizza il metodo Add con un elemento di tipo int, come nel caso precedente, viene eseguito un boxing del valore numerico. Essa, inoltre, permette, naturalmente, di aggiungere all'elenco un qualsiasi altro tipo, quindi nella stessa istanza lista potremmo inserire contemporaneamente anche delle stringhe:

```
lista.Add("stringa");
```

A questo punto si ha una raccolta di elementi eterogenei, che da una certa prospettiva potrebbe anche sembrare un vantaggio: con una sola classe si possono memorizzare oggetti di qualsiasi tipo.

In realtà l'esempio nasconde già diverse insidie o problemi. Il primo è di carattere prestazionale: a ogni inserimento della lista deve essere eseguito un boxing di un tipo valore, oppure un cast verso object nel caso di istanze di tipi riferimento.

Diversamente, volendo utilizzare i valori presenti nella lista, estraendoli da essa si dovrà eseguire il processo contrario di unboxing o di cast verso il tipo originale. Questo d'altra parte provoca altri problemi di performance, che saranno evidenti soprattutto quando si avrà a che fare con liste contenenti parecchi elementi.

L'altra limitazione, sebbene appunto prima l'abbiamo quasi vista come un vantaggio, è la mancanza di controllo a livello di sicurezza dei tipi. Potendo inserire un qualunque oggetto in una classe come ArrayList, nulla garantisce sul tipo degli elementi presenti in una raccolta. Supponiamo, per esempio, di voler calcolare la somma degli elementi presenti nella lista precedentemente istanziata, con un ciclo foreach:

```
int sum=0;
foreach(int i in lista)
{
sum += i;
}
```

Il codice precedente, a causa della presenza di un oggetto string come terzo elemento, provoca a un certo punto un'eccezione di tipo InvalidCastException. Quindi, per impedire situazioni del genere, lo sviluppatore deve eseguire uno sforzo maggiore, effettuando controlli sul tipo di elementi oppure gestendo ancora più situazioni fonti di possibili eccezioni.

Una pratica comune per evitare tali possibili situazioni, prima di C# 2.0 (o comunque senza utilizzare i generics), era quella di implementare classi apposite per contenere solo specifici tipi di elementi, spesso dette *collezioni fortemente tipizzate*.

L'approccio in tal caso fa perdere molto in generalizzazione e costringe lo sviluppatore a riscrivere lo stesso codice ripetitivo per ogni tipo di oggetti da trattare.

Come vedremo a breve, l'introduzione dei generics, e quindi, in questo caso particolare, di collezioni generiche, risolve questo e altri problemi, oltre ad aggiunge all'arsenale dei programmatori .NET una nuova formidabile arma.

Parametri di tipo

Nella definizione di un tipo o di un metodo generico si utilizzano i *parametri di tipo*, che sono dei segnaposto per i tipi specificati poi nella creazione dell'istanza del tipo o dell'invocazione di un metodo.

Per indicare i parametri di tipo si utilizzano le parentesi angolari e, per convenzione, come nome di tali parametri si usa la lettera T (che sta per "Tipo"):

```
class List<T>
{
}
```

La precedente è una definizione di una classe List che usa un parametro di tipo T.

Se si hanno più parametri, o in genere se si vogliono usare nomi più descrittivi, si usa la lettera T come prefisso:

```
class Dictionary<TKey, TValue>
{
}
```

La precedente classe Dictionary utilizza due parametri di tipo, uno che sarà usato come tipo delle chiavi, il secondo come tipo dei valori, quindi in questo caso è fondamentale far capire mediante i nomi quale sarà il ruolo dei due parametri.

I parametri di tipo possono essere naturalmente utilizzati in tutte le entità definibili come generiche: classi, interfacce, struct, delegate, metodi.

Altri costrutti non possono dichiarare parametri di tipo, ma possono utilizzare quelli presenti. Per esempio, una proprietà non può dichiarare un parametro di tipo, ma può utilizzare quelli definiti dalla propria classe.

Nel seguente esempio si è implementato l'indicizzatore per la classe generica Lista<T>, che restituisce quindi un oggetto di tipo T presente nell'array elementi all'indice specificato:

```
class Lista<T>
{
public T this[int index]
{
get
{
return elementi[index];
}
}
}
```

Classi generiche

Per creare un tipo generico, è necessario specificare uno o più parametri di tipo nella sua definizione, racchiudendoli all'interno di due parentesi angolari. Quindi non esiste nessuna parola chiave speciale per definire un tipo generico.

I parametri di tipo indicati saranno poi utilizzabili all'interno del corpo della classe. Per esempio, supponendo di voler implementare una classe Lista, che può contenere un determinato tipo di oggetti dichiarato solo al momento della creazione di un nuovo oggetto Lista, si scriverà la seguente definizione di classe:

```
public class Lista<T>
{
private T[] array;
public Lista(int len)
{
array=new T[len];
}
}
```

Il parametro di tipo T indica un generico tipo degli elementi. All'interno della classe viene dichiarato un campo array, che usa T come tipo degli elementi che saranno in esso contenuti.

In questo momento la classe Lista<T> si dice essere un tipo *aperto*, in quanto la sua struttura dati non è ancora completamente definita. Lo sarà al momento di istanziare un oggetto.

Si noti che il costruttore della classe non deve specificare il parametro di tipo nella sua firma, ma eredita naturalmente quello della classe stessa, infatti lo utilizza per creare l'array di oggetti di tipo T.

La stessa regola vale per l'eventuale distruttore. Sarebbe quindi errato scriverlo così:

```
public Lista<T>(int len) //errore
{
array=new T[len];
}
```

NOTA	Per leggere o pronunciare il nome di un tipo generico si usa il nome del tipo non generico seguito da "di T" o dal nome del tipo utilizzato come argomento. Per esempio, per Lista<T> si può dire "Lista di T", oppure per Lista<int> si dirà "Lista di int".

Tipi costruiti

A partire dalla definizione di una classe generica, come Lista<T>, si può indicare quale tipo reale utilizzare al posto dei segnaposto (in questo caso il solo T) per costruire una classe

istanziabile. Per esempio, se ci si vuole servire del tipo int, si otterrà la classe costruita Lista<int>, quindi la seguente potrebbe essere la creazione di una lista di cinque elementi interi:

```
Lista<int> lista=new Lista<int>(5);
```

Il tipo int, che sostituisce il parametro di tipo T, viene anche detto *argomento di tipo*.

Il tipo Lista<int> è detto a questo punto un tipo *chiuso*, o *costruito*. A partire dal tipo generico Lista<T>, si possono costruire quanti tipi si vogliono.

Se, per esempio, fosse necessario trattare oggetti di tipo string, si dichiarerà un oggetto come segue:

```
Lista<string> listaStringhe=new Lista<string>(10);
```

Nel caso in cui la definizione del tipo utilizzi più parametri di tipo, essi vengono separati mediante la virgola. Per esempio, una classe dizionario che fa corrispondere a un insieme di chiavi un insieme dei valori potrebbe essere dichiarata come segue:

```
public class Dizionario<T, U>
{
private T[] keys;
private U[] values;
}
```

Naturalmente, ogni classe costruita a partire dalla precedente definizione può anche utilizzare tipi differenti come argomenti:

```
Dizionario<int, string> diz=new Dizionario<int, string>();
```

NOTA	Sebbene i generics di C# ricordano, per la loro somiglianza, i template di C++, si tratta di due tecniche diverse. La prima differenza sta nel fatto che i template sono una tecnica che definisce totalmente i tipi a compile-time, mentre i generics vengono costruiti a runtime. La seconda peculiarità dei generics è rappresentata dal fatto che essi sono una caratteristica del framework .NET e non del solo linguaggio C#, quindi devono supportare l'interoperabilità fra linguaggi diversi, concetto non presente naturalmente in C++ nativo.

I tipi generici possono essere innestati, nel senso che i tipi utilizzati come parametri per costruire un generic, possono a loro volta essere dei tipi generici.

Per esempio, una List<T> può essere costruita utilizzando come tipo T dei suoi elementi un Dictionary<int,string>:

```
List<Dictionary<int,string>> listaDict;
```

Per semplificare la nomenclatura di tipi generici particolarmente complessi, è possibile utilizzare l'istruzione using, e creare degli alias:

```
using ListDict=System.Collections.Generic.List<System.Collections.Generic.Dictionary<int, string>>;
```

In tal modo il tipo generico è utilizzabile direttamente mediante l'alias ListDict, come fosse un normale tipo:

```
ListDict lista=new ListDict();
```

Classi generiche ereditate

Una classe generica può essere ereditata come una classe qualunque.

La classe derivata è libera di mantenere aperti i parametri di tipo:

```
class ListaFiglia<T>: Lista<T>
{
}
```

oppure di derivare da un tipo costruito chiuso, cioè ottenuto specificando i tipi al posto dei parametri nella classe generica:

```
class ListaString: Lista<string>
{
}
```

In questo caso, la classe ListaString non è più una classe generica, in quanto ha sostituito al parametro T l'argomento string.

Una classe derivata da una generica può inoltre aggiungere nuovi parametri di tipo:

```
class ListaSpeciale<T, U>: Lista<T>
{
}
```

Tipi generici innestati

Come parametro di tipo, in quanto essi stessi normali tipi definibili in .NET, possono essere utilizzati altri tipi generici. In tal modo, si ha la possibilità di definire dei tipi generici innestati. Per esempio, come argomento di tipo della classe Lista<T> potremmo utilizzare il tipo generico Nullable<int> (vedi più avanti per i tipi Nullable), ottenendo così una Lista di Nullable di int:

```
Lista<Nullable<int>> lista= new Lista<Nullable<int>>();
```

Valori predefiniti

Poiché a un parametro di tipo può, in generale, essere sostituito un qualunque tipo argomento, può verificarsi spesso un problema all'interno dei nostri tipi generici: quello di assegnare un valore predefinito a una variabile.

All'interno di un tipo costruito, infatti, un parametro di tipo può essere sostituito sia da tipi valore, sia da tipi riferimento, senza averne alcuna conoscenza a priori.

Quindi, se il tipo argomento fosse uno di tipo riferimento, potremmo utilizzare null come valore predefinito, ma ciò non è valido per i tipi valore.

Inoltre, anche se fossimo sicuri che il tipo argomento fosse uno dei tipi valore, l'assegnazione di un valore numerico non andrebbe bene perché il tipo valore potrebbe anche essere una struct.

La soluzione fornita da C# è l'utilizzo dell'operatore default, che restituisce il valore predefinito di un tipo qualunque, secondo le seguenti regole:

- per un tipo riferimento T, default(T) restituisce null;
- per un tipo valore numerico, default(T) restituisce il valore zero;
- per un tipo valore struct, default(T) inizializza i membri con null o zero, a seconda che essi siano rispettivamente di tipo riferimento o valore.

Supponiamo di avere la seguente classe generica, che dichiara un campo di tipo T e uno di tipo U, i quali rappresentano i parametri di tipo generici:

```
class Generica<T,U>
{
private T campo1;
private U campo2;
public Generica()
{
campo1=default(T);
campo2=default(U);
Console.WriteLine(campo1);
Console.WriteLine(campo2);
}
}
```

I due campi vengono inizializzati nel costruttore della classe.

Costruendo ora un tipo concreto a partire dalla classe generica:

```
Generica<int, string> gen=new Generica<int, string>();
```

i valori di campo1 e campo2 sarebbero in questo caso pari rispettivamente a 0 e null, come potete verificare da quello che stampa l'istruzione WriteLine, eseguendo l'esempio.

Membri statici

I membri statici di un tipo generico sono condivisi solo con un particolare tipo costruito. Per esempio, se si considera la classe seguente, che possiede un campo statico:

```
public class Lista<T>
{
public static int Numero;}
```

ogni tipo costruito, a partire da Lista<T>, avrà una propria versione del campo statico:

```
Lista<int>.Numero=1;
Lista<string>.Numero=2;

Console.WriteLine(Lista<int>.Numero); //stampa 1
Console.WriteLine(Lista<string>.Numero); //stampa 2
```

Vincoli

I parametri di tipo possono essere sostituiti da un qualunque tipo argomento. È possibile però applicare dei *vincoli*, in maniera da richiedere specifiche caratteristiche ai tipi utilizzabili.

Per indicare un vincolo si usa la parola chiave where, che segue il nome della classe. Essa permette di definire le regole che i tipi devono rispettare. Tali regole possono essere di vario tipo, come elencato qui di seguito:

```
where T: <NomeClasse> //indica che il tipo deve derivare dalla classe base specificata
where T: <Interfaccia> //indica che il tipo deve implementare l'interfaccia specificata
where T: class //indica che dev'essere un tipo riferimento
where T: struct //indica che dev'essere di tipo valore
where T: new() //indica che il tipo deve possedere un costruttore pubblico senza parametri
where T: U //indica che il tipo deve corrispondere o derivare dal tipo argomento fornito per U
```

Per esempio, se volessimo fare in modo che la classe Lista<T> vista precedentemente possa contenere solo elementi di un tipo riferimento, potremmo specificare il vincolo nel seguente modo:

```
class Lista<T> where T:class
{
}
```

Se poi tale tipo riferimento dovesse, per esempio, implementare una determinata interfaccia (anche perché all'interno della classe generica verranno utilizzati i membri di quest'ultima), potremmo specificare il nome dell'interfaccia.

La seguente definizione costringerà ogni argomento di tipo a essere uno di quelli che implementano l'interfaccia IComparable:

```
class Lista<T> where T: IComparable
{}
```

Ogni parametro di tipo può avere una propria clausola where, senza utilizzare alcun separatore per le clausole where di parametri differenti:

```
class Generica<T, U> where T: class
where U: T
```

L'esempio vincola T a essere di tipo riferimento e U a derivare a sua volta da T. Naturalmente, il fatto di disporre le clausole where su più righe è una pura questione di stile e di comodità di lettura. In questo caso, se provassimo a costruire un tipo nel seguente modo:

```
Generica<int, string> gen;
```

si avrebbe un errore di compilazione dovuto al fatto che int non è di tipo riferimento.

Se invece si vuole specificare che uno stesso parametro di tipo deve rispettare più vincoli, essi vengono indicati nella stessa clausola where, separandoli con la virgola:

```
class Generica<T, U> where T: class, new()
where U: T, IComparable
```

In questo caso, il parametro T deve essere di tipo riferimento e deve possedere un costruttore pubblico predefinito, mentre U deve derivare da T e implementare l'interfaccia IComparable.

Alcuni vincoli non sono compatibili fra loro e, inoltre, ci sono regole di ordine da rispettare.

Suddividendo i tipi di vincoli in tre categorie, primari (NomeClasse, class, struct), secondari (Interfaccia) e vincolo di costruttore, può essere applicato al massimo un vincolo primario, zero o più vincoli Interfaccia, e al massimo un vincolo di costruttore. Inoltre, l'ordine in cui essi devono apparire è proprio questo indicato.

Per esempio, non è logicamente possibile utilizzare per uno stesso parametro il vincolo class e quello struct. Oppure, se si utilizza il vincolo new(), esso deve essere posto come ultimo nell'elenco, se usato assieme ad altri vincoli.

Vincoli ereditati

Se una classe viene derivata da una con vincoli, anche i vincoli stessi vengono ereditati, ma devono essere esplicitamente indicati anche nella classe figlia.

```
class Lista<T> where T: class
{
}
class ListaFiglia<T>: Lista<T>
where T:class
{
}
```

Se non si indica la clausola where anche nella seconda classe, si ottiene un errore di compilazione, a indicare che T deve essere di tipo riferimento. Sebbene possa sembrare una ripetizione inutile, tale pratica aumenta la chiarezza del codice: altrimenti potrebbe aggiungere confusione vedere all'interno della classe derivata l'uso di un tipo di cui non si conosce la provenienza.

Inoltre, vi è anche un ragione tecnica: in una classe derivata da una generica, tutti i parametri di tipo sono nuovi, tanto che è anche possibile assegnare nomi diversi o vincoli diversi ma compatibili con quelli ereditati:

```
class AltraClasse
{
}

class ListaFiglia<TFiglio>: Lista<TFiglio>
where TFiglio: AltraClasse
```

```
{
}
```

La ListaFiglia eredita in questo caso il parametro di tipo, ma lo denomina con il nuovo nome TFiglio. Inoltre, il parametro viene ulteriormente specializzato, indicando che il tipo TFiglio deve essere di tipo AltraClasse oppure derivare da essa.

Il nuovo vincolo è compatibile con quello della classe Lista<T> in quanto più stringente: AltraClasse è infatti un tipo riferimento.

Metodi generici

I *metodi generici* possono utilizzare uno o più parametri di tipo nella propria firma e tornano utili quando si vuol rendere generico un algoritmo anziché un tipo.

Un metodo generico è un metodo che ha uno o più parametri di tipo, che possono essere utilizzati sia come parametri di ingresso sia come tipi di ritorno.

I parametri di tipo vengono indicati anche qui fra parentesi angolari, dopo il nome del metodo, ma prima della lista di parametri formali. Per esempio, il seguente metodo usa il parametro di tipo T sia come parametro di ingresso, sia come tipo di ritorno in una classe:

```
public T GetDefaultValue<T>(T param1)
{
return default(T);
}
```

In tal modo, il metodo GetDefaultValue può funzionare su diversi tipi di parametri.

Per utilizzare il metodo, deve quindi essere specificato il tipo concreto da utilizzare al momento della sua invocazione:

```
int i= GetDefaultValue<int>(1);
```

Il compilatore è in grado di ricavare autonomamente il tipo di parametri utilizzati per invocare il metodo, quindi è anche possibile omettere i tipi da utilizzare come argomenti. Per esempio, l'invocazione del metodo precedente può essere semplicemente scritta come:

```
int val=1;
int i= GetDefaultValue(val);
```

Tale processo, detto *inferenza dei tipi*, permette di dedurre i tipi in base agli argomenti passati al metodo. In questo caso, quindi, il compilatore sa che dovrà utilizzare l'implementazione del metodo generico che ha int come parametro.

I metodi generici possono essere dichiarati sia all'interno di tipi a loro volta generici, sia in tipi concreti.

All'interno di una classe non generica, abbiamo visto che i metodi non generici possono utilizzare i parametri di tipo. Un classico esempio è il metodo Swap, che scambia due oggetti di un dato tipo T:

```
class SwapClass<T>
{
void Swap(ref T left, ref T right)
{
T temp= left;
```

```
left = right;
right=temp;
}
}
```

Se il metodo generico fa parte di una classe generica, esso può anche utilizzare i parametri di tipo della classe stessa, al suo interno. Tuttavia, nel caso in cui definisca esso stesso dei parametri di tipo ed essi siano denominati allo stesso modo di quelli utilizzati dalla classe, il compilatore genererà un messaggio di avviso:

```
public class SwapClass<T>
{
public void Swap<T>(ref T left, ref T right)
{
T temp= left;
left = right;
right=temp;
}
}
```

In questo esempio, poiché sia la classe sia il metodo `Swap` dichiarano un parametro di tipo `T`, quello del metodo nasconderà il tipo della classe (e il compilatore ci avviserà del conflitto).

Per evitare confusioni fra i tipi, e nel caso in cui quindi un metodo generico debba definire parametri di tipi differenti o aggiuntivi, è bene utilizzare identificatori nuovi:

```
public class SwapClass<T>
{
public void Swap<U>(ref U left, ref U right)
{
T temp= left;
left = right;
right=temp;
}
}
```

Con la stessa sintassi vista nel paragrafo precedente, indicando le clausole `where` dopo la parentesi di chiusura dei parametri formali, è possibile definire dei vincoli anche per i metodi generici. Il seguente metodo, per esempio, può implementare l'ordinamento degli elementi di un array confrontandoli, quindi il tipo di tali elementi dovrà implementare l'interfaccia `IComparable`:

```
public void Sort<T>(T[] elements) where T: IComparable
{
//implementazione per esercizio!
}
```

Metodi di estensione

Anche i *metodi di estensione*, visti nel Capitolo 6, possono essere implementati come metodi generici.

Come già visto, la parola chiave this, che precede un parametro del metodo, indica il tipo di oggetto che potrà invocare il metodo. Per esempio, se volessimo eseguire l'ordinamento di un oggetto Lista contenitore di istanze di un tipo qualunque, basterebbe implementare all'interno del metodo di estensione seguente la logica necessaria:

```
public static class UtilityClass
{
public static Lista<T> Ordina<T>(this Lista<T> obj)
{
//ordina elementi e restituisce la lista ordinata
//implementazione per esercizio
return obj;
}
}
```

In questo modo, per ottenere la versione ordinata, basterebbe invocare il metodo su un qualunque oggetto Lista, così:

```
Lista<string> list=new Lista<string>();
list.Ordina<string>();
```

Anche in questo caso non sarebbe necessario esplicitare il tipo generico, in quanto il compilatore lo ricaverebbe dal tipo dell'istanza list. Basta quindi scrivere:

```
list.Ordina();
```

NOTA	I metodi di estensione generici sono utilizzati abbondantemente all'interno di LINQ per aggiungere le operazioni di query standard ai tipi che implementano IEnumerable e IEnumerable<T>. Per esempio, il metodo di estensione per l'ordinamento è già fornito pronto all'uso da LINQ.

Interfacce generiche

L'implementazione di *interfacce generiche* consente di scrivere interfacce in cui le firme dei metodi che esse definiscono contengono parametri di tipo, sia come parametri formali sia come tipi di ritorno.

Per dichiarare un'interfaccia generica, si utilizza la stessa sintassi vista per le classi, indicando i parametri di tipo fra parentesi angolari e poi utilizzandoli come se fossero tipi qualunque:

```
public interface ITransformer<T,S>
{
S Transform(T param1);
}
```

Una classe che implementa l'interfaccia può essere a sua volta generica:

```
class Transformer<T,S>: ITransformer<T,S>
{
public S Transform(T param1)
{
return default(S);
}
}
```

Oppure, dato che l'interfaccia generica può poi essere ereditata da diverse interfacce costruite specificando i parametri, esse possono essere implementate da classi non generiche:

```
class TransformerIntString: ITransformer<int,string>
{
public string Transform(int param1)
{
return param1.ToString();
}
}
```

In questo caso, l'interfaccia sostituisce i parametri di tipo rispettivamente con i tipi `int` e `string`.

L'interfaccia deve quindi essere implementata dalla classe `TransformerIntString` utilizzando i tipi specificati.

Dato che l'impiego di altri tipi al posto di `T` e `S` definisce una classe completamente nuova e differente, una classe potrebbe anche implementare diverse interfacce costruite a partire da quella generica:

```
class TransformerIntStringInt: ITransformer<int,string>, ITransformer<string, int>
{
public string Transform(int param1)
{
```

```
return param1.ToString();
}

public int Transform(string param1)
{
int val;
if(int.TryParse(param1, out val))
return val;
return default(int);
}
}
```

Delegate generici

L'ultimo tipo che può essere implementato come generico è quello dei delegate.

Il concetto di *delegate* verrà introdotto e approfondito nel prossimo capitolo, quindi per il momento potrebbe apparire leggermente oscuro. Pensate a esso come a una sorta di oggetto che rappresenta una famiglia di metodi dalla firma ben specificata:

```
public delegate int MioDelegate(string str);
```

La dichiarazione precedente definisce un delegate.

Essa assomiglia alla dichiarazione di un metodo, con l'aggiunta della parola chiave delegate. Quindi, i metodi compatibili con il delegate MioDelegate sono quelli che restituiscono un int e che accettano in ingresso un unico parametro string.

Impiegando i generics, è possibile utilizzare parametri di tipo nella dichiarazione di un delegate:

```
public delegate T MioDelegate<T, U>(U val);
```

In questo modo un metodo come il seguente:

```
int MioMetodo(string val)
{
return Convert.ToInt32(val);
}
```

rispetta la firma del delegate MioDelegate e può essere considerato un oggetto assegnabile a una variabile di tipo MioDelegate:

```
MioDelegate<int,string> del=MioMetodo;
```

L'oggetto del può essere poi utilizzato per invocare uno o più metodi che esso rappresenta:

```
int i = del("123");
```

oppure:

```
int i=del.Invoke("123");
```

Anche i delegate generici possono definire dei vincoli, mediante clausole where, per restringere i tipi di parametri utilizzabili.

Nel prossimo capitolo vedremo a cosa servono nella pratica e come utilizzare i delegate in un programma C#.

Conversioni dei parametri di tipo

I tipi utilizzati come parametri all'interno di un qualunque tipo generico non sono noti a tempo di compilazione, quindi portano a un interessante scenario in caso di conversioni da eseguirsi a runtime.

Supponiamo di voler esaminare il tipo di un parametro e quindi eseguire un'appropriata conversione prima di utilizzarlo:

```
public class Generica<T>
{
void Metodo(T val)
{
if(val is string)
{
string str=(string)val; //errore
}
}
}
```

Sebbene l'operatore is permetta di verificare che l'argomento val sia di tipo string, la successiva conversione esplicita verso il tipo string non piace al compilatore, che restituirà un errore con un messaggio "Impossibile convertire T in string". Il compilatore, infatti, non conosce il tipo T, quindi tenta di effettuare una conversione personalizzata che naturalmente non esiste.

Le soluzioni in questo caso sono almeno un paio. La prima prevede l'utilizzo dell'operatore as:

```
void Metodo(T val)
{
if(val is string)
{
string str=val as string;
}
}
```

L'operatore as però può agire solo su oggetti di tipo riferimento. Quindi, se si volesse convertire verso un tipo valore, per esempio int, la precedente strategia non è applicabile.

Un'altra soluzione, che è utilizzabile in maniera molto più generale (vale anche nel caso precedente), è quella di effettuare una doppia conversione, la prima verso object (ogni tipo è convertibile in object) e poi verso il tipo finale di destinazione:

```
void Metodo(T val)
{
if(val is string)
{
string str=(string)(object)val;
```

```
}
}
```

Come detto la stessa strategia può essere utilizzata con un tipo valore:

```
void Metodo(T val)
{
if(val is int)
{
int i=(int)(object)val;
}
}
```

Nel caso precedente, si ha un'operazione di boxing del parametro val, seguita da una di unboxing verso int.

Struct generiche

In modo analogo a quanto visto per classi e interfacce, anche le struct possono essere implementate in maniera generica. Esse possono avere quindi parametri di tipo e vincoli, seguendo le stesse regole.

La seguente struct, per esempio, definisce un punto le cui coordinate possono essere di un generico tipo T, vincolato a essere esso stesso una struct (per esempio, un tipo numerico):

```
struct Point<T> where T:struct
{
public T x;
public T y;
}
```

In questo modo, possiamo creare delle struct costruite con tipi come le seguenti:

```
Point<int> ptInt;
Point<double> ptDouble;
```

ma non come segue:

```
Point<string> ptStr; //error, string non è un tipo valore
```

Tipi nullable

Fra le strutture generiche contenute e messe a disposizione da .NET, introdotte anch'esse con C# 2.0 e che abbiamo già avuto modo di utilizzare (senza sapere nulla di generics!) nel Capitolo 3, vi sono i cosiddetti tipi *nullable*.

La sintassi seguente:

```
int? variabile;
```

definisce una variabile intera che, pur essendo un tipo valore, può così assumere anche il valore null.

Essa utilizza l'operatore ?, ma si tratta di una sorta di abbreviazione, o "zucchero sintattico", come si suol dire, per il tipo generico Nullable<T>.

La stessa dichiarazione precedente è infatti equivalente alla seguente:

```
Nullable<int> variabile;
```

Il tipo Nullable<T> è una struct generica con un parametro di tipo T che è vincolato a essere anch'esso una struct.

In generale, quindi, una variabile di un tipo nullable può essere dichiarata in uno dei due seguenti modi equivalenti:

```
Nullable<T> variabile;
T? variabile;
```

Una variabile nullable permette di memorizzare un qualsiasi valore del tipo argomento, oppure il riferimento null.

Quindi, una variabile come la precedente può essere per esempio utilizzata come segue:

```
Variabile = 1;
variabile = null;
```

Ecco la dichiarazione del tipo Nullable<T> nella libreria standard:

```
public struct Nullable<T> where T:struct
{
private T @value;
…
}
```

All'interno del campo privato viene memorizzato un eventuale valore del tipo nullable.

Il tipo definisce delle proprietà per poter lavorare e trattare più agevolmente variabili di tipi nullable. Per esempio, le proprietà in sola lettura HasValue e Value permettono di verificare se il valore assegnato sia non nullo e recuperarlo:

```
if(variabile.HasValue)
{
int i=variabile.Value;
}
```

La proprietà HasValue restituisce true se la variabile contiene un valore, altrimenti false. Di conseguenza la proprietà Value restituisce tale valore se esso è presente; invece, in caso contrario, genererebbe un'eccezione InvalidOperationException.

È tuttavia possibile anche utilizzare i classici operatori di uguaglianza e disuguaglianza per eseguire la stessa verifica, e l'operatore di cast per ricavarne il valore contenuto. L'esempio precedente è equivalente a:

```
if(variabile!=null)
{
int i=(int)variabile;
}
```

Si può poi utilizzare il metodo GetValueOrDefault per ottenere il valore assegnato a una variabile nullable, oppure il valore predefinito del tipo sottostante se quello assegnato è null:

```
int predefinito=variabile.GetValueOrDefault();
```

Oppure è possibile, se il valore assegnato è null, restituire un valore predefinito a scelta, passato come argomento al metodo:

```
int predefinito=variabile.GetValueOrDefault(5);
```

In questo caso, se il valore assegnato alla variabile è null, verrà restituito il valore intero 5.

Abbiamo già avuto modo di vedere l'utilizzo dell'operatore di null coalescing ??, che funziona anche con i tipi nullable.

Esso permette di ottenere lo stesso effetto del metodo GetValueOrDefault, utilizzando una sintassi più immediata e breve:

```
int predefinito= variabile ?? 5; //se variabile è null, assegna il valore 5
```

È inoltre possibile utilizzare i classici operatori aritmetici per eseguire operazioni su variabili nullable:

```
int? x=1;
int? y=2;
int? somma=x+y; //=3
```

Gli operandi vengono convertiti nei rispettivi tipi sottostanti, quindi poi vengono usati gli operatori aritmetici standard. Se però uno dei due operandi nullable non ha un valore assegnato, anche il risultato di un'operazione aritmetica sarà pari a null. Per esempio:

```
int? x=null;
int? result= x * 2; //restituisce null
```

Se uno dei valori che partecipano alla somma o ad altra operazione non è nullable, sarà necessario eseguire una conversione esplicita:

```
int? x=2;
int z=3;
somma= x+z; //errore

somma=(int)x+z;
```

Ovviamente, l'ultima istruzione va a buon fine solo se x ha un valore assegnato.

L'operazione di conversione opposta, da un tipo non nullable verso il corrispondente nullable, avviene invece in maniera implicita, senza necessità di operatori di cast:

```
int x=1;
int? nx= x;
```

Covarianza e controvarianza

Covarianza e *controvarianza* sono due concetti introdotti in C# 4.0, ai quali spesso ci si riferisce anche con il termine collettivo di *varianza*. Essi descrivono se è possibile e come vengono effettuate conversioni di tipo, da un tipo figlio a un tipo padre o viceversa.

Bene, vi chiederete, cosa c'entra tutto questo con i generics e soprattutto: "Ma non abbiamo già affrontato l'argomento 'conversioni di tipo'?".

In effetti sì, ma nel caso di tipi generici c'è qualcosa da considerare nel dettaglio per comprenderne a fondo il funzionamento.

Vediamo il seguente esempio, in cui la classe Car deriva dalla classe madre Vehicle:

```
class Vehicle
{
}
class Car: Vehicle
{
}
```

Se proviamo a istanziare un oggetto Car e assegnarlo a un riferimento Vehicle, tutto funziona, in quanto l'oggetto Car è derivato da Vehicle e quindi ogni Car è anche un Vehicle:

```
Car ferrari=new Car();
Vehicle vehicle=ferrari;
```

Questo meccanismo è detto *compatibilità di assegnamento*, e il suo funzionamento è dovuto alla natura object oriented di C#, in particolare alla caratteristica che abbiamo chiamato polimorfismo.

In C#, fin dalla versione 1.0, è perfettamente lecito anche assegnare array di un tipo derivato a uno di un tipo padre:

```
Car[] carArray=new Car[10];
carArray[0] = ferrari;
Vehicle[] vehicleArray=carArray;
```

Si dice che tale operazione di creazione e assegnazione di un array è *covariante* perché essa conserva la relazione di ereditarietà presente fra la classe Vehicle e la classe derivata Car. Ma il supporto di tale caratteristica per gli array è considerato da molti controverso, in quanto non type-safe, e può provocare problemi come quello esposto di seguito.

Supponiamo, infatti, di avere ora un'altra classe Moto, anch'essa figlia di Vehicle. Tentiamo quindi di inserire nell'array vehicleArray un elemento di tipo Moto:

```
class Moto:Vehicle
```

```
{}
vehicleArray[0]=new Moto();
```

Il compilatore ce lo permette tranquillamente come prima, perché Moto deriva da Vehicle, ma stavolta, a tempo di esecuzione, l'assegnazione dell'elemento provocherà un'eccezione di tipo ArrayTypeMismatchException (vedi Figura 9.1), affermando che il tipo dell'elemento non è compatibile con quello dell'array!

NOTA	Il supporto della covarianza per gli array è stato introdotto per permettere l'interoperabilità con linguaggi in cui essa è supportata: come J#, l'implementazione .NET di Java in cui tale caratteristica era presente.

In effetti, pensandoci bene, ce lo potremmo anche aspettare: l'assegnazione dell'array carArray alla variabile vehicleArray ha semplicemente assegnato un riferimento, quindi anche vehicleArray punta a un array di Car, e di conseguenza può contenere solo oggetti di tale tipo.

```
class Vehicle
{
}

class Car : Vehicle
{
}

class Moto : Vehicle
{
}

class Program
{
  static void Main(string[] args)
  {
    Car ferrari = new Car();
    Vehicle vehicle = ferrari; //assignment compat

    Car[] carArray=new Car[10];
    carArray[0] = ferrari;
    Vehicle[] vehicleArray=carArray; //array covariance

    vehicleArray[1] = new Moto();
    Null
```

ArrayTypeMismatchException was unhandled

Attempted to access an element as a type incompatible with the array.

Troubleshooting tips:

Make sure the object type is convertible to the array type.

Get general help for this exception.

Search for more Help Online...

Exception settings:

Break when this exception type is thrown

Actions:

View Detail...

Copy exception detail to the clipboard

Open exception settings

Figura 9.1 - Eccezione dovuta all'assegnazione di oggetto non compatibile con array.

Aggiungiamo ora l'ingrediente generics e proviamo, per esempio, a creare una lista di Car (utilizzando la classe List<T> che vedremo nel paragrafo dedicato alle collezioni):

```
List<Car> macchine=new List<Car>();
```

e quindi ad assegnarla a una lista di Vehicle:

```
List<Vehicle> veicoli=macchine;
```

Ci si aspetterebbe probabilmente che anche qui non si verifichi alcun problema, come per gli array, e invece il compilatore ci avviserà che è impossibile una conversione implicita dal tipo List<Car> a quello List<Vehicle>.

Inoltre, non è consentita nemmeno una conversione esplicita:

```
List<Vehicle> veicoli=(List<Vehicle>)macchine;
```

In altre parole, le classi generiche sono invarianti per assicurare maggior sicurezza rispetto ai tipi.

Ciò potrebbe tuttavia compromettere la flessibilità, impedendo di scrivere del codice generale come il seguente:

```
public void StartRun(List<Vehicle> veicoli)
{
foreach(Vehicle v in veicoli)
v.Accelerate();
}
```

e utilizzarlo così, con una List di Car anziché di Vehicle:

```
List<Car> cars=new List< Car >();
cars.Add(new Car());
cars.Add(new Car());
StartRun(cars); //errore
```

Una prima soluzione è implementare il metodo come generico e aggiungere un vincolo al parametro di tipo:

```
static void StartRun<T>(List<T> veicoli) where T: Vehicle
{
foreach (Vehicle v in veicoli)
v.Accelerate();
}
```

L'introduzione in C# 4.0 del supporto di covarianza e contravarianza per le interfacce generiche e per i delegate generici rende il lavoro con questi tipi molto più semplice e, probabilmente, più vicino a come il buon senso potrebbe suggerire.

Varianza e generics

La covarianza e la controvarianza sono implementabili per i parametri di tipo nelle interfacce e nei delegate generici, utilizzando apposite parole chiave.

Un'interfaccia generica o un delegate generico saranno quindi detti *varianti* se il parametro di tipo è dichiarato covariante o controvariante.

In generale, se la classe Derivata è convertibile in Base, allora il tipo Generico è covariante se Generico<Derivata> è convertibile in Generico<Base>. Per esempio, IEnumerable<T> è un'interfaccia covariante, il che significa che un IEnumerable<string> è implicitamente convertibile in IEnumerable<object> perché string è derivato da object:

```
IEnumerable<string> listaStr=new List<string>();
IEnumerable<object> listaObj=listaStr;
```

In questo modo, la covarianza appare molto intuitiva perché somigliante al polimorfismo, che permette la compatibilità di un tipo derivato con un suo tipo base.

Un tipo Generico è invece controvariante se, assumendo sempre che Derivata sia convertibile in Base, Generico<Base> è convertibile in Generico<Derivata>.

L'interfaccia IComparer<T> è per esempio controvariante. O meglio: lo è il parametro di tipo T. Ciò significa che è possibile utilizzare come argomento un tipo T qualsiasi oppure uno meno derivato. Quindi, supponendo che per esempio Vehicle implementi l'interfaccia IComparer<Vehicle>:

```
class Vehicle: IComparer<Vehicle>
{
public int Compare(Vehicle x, Vehicle y)
{…}
}
```

Sarà in tal modo possibile assegnare un IComparer di Vehicle a un riferimento IComparer di una classe figlia:

```
IComparer<Vehicle> v= new Vehicle();
IComparer<Car> c = v;
```

Interfacce e delegate varianti di .NET

Le interfacce IEnumerable<T> e IComparer<T> utilizzate negli esempi del precedente paragrafo sono due tipi facenti parte della libreria standard di .NET Framework 4.

Nella libreria di classi di .NET esistono altre interfacce, elencate di seguito come riferimento, che hanno parametri di tipo covarianti (out) e/o controvarianti (in):

- IComparable<in T>;
- IComparer<in T>;
- IEnumerable<out T>;

- IEnumerator<out T>;
- IEqualityComparer<in T>;
- IGrouping<out TKey, out TValue>;
- IOrderedEnumerable<out TElement>;
- IOrderedQueryable<out T>;
- IQueryable<out T>.

Inoltre, i seguenti delegate generici, anch'essi varianti, fanno sempre parte della libreria di base:

- Action<in T>,;
- Action<in T1, in T2, … in T16>;
- Comparison<in T>;
- Converter<in TInput, out TOutput>;
- Func<out TResult>;
- Func<in T, out TResult>;
- Func<in T1, in T2, …, in T16, out TResult>;
- Predicate<in T>.

Tali interfacce e i delegate generici sono utilizzati da parecchie API del .NET Framework, oltre a poter essere impiegati nel proprio codice da parte degli sviluppatori.

Una volta affrontati argomenti come eventi, delegate e LINQ, molti aspetti ora forse oscuri si dimostreranno più chiari e riveleranno le proprie potenzialità.

Implementare interfacce varianti

A partire da .NET 4 e quindi da C# 4.0, è possibile marcare i parametri di tipo di interfacce e di delegate generici come covarianti e controvarianti.

Un parametro di tipo *covariante* è marcato con la parola chiave out. Lo si può utilizzare come tipo di ritorno di un metodo definito da un'interfaccia, oppure come tipo di ritorno di un delegate. Non si può invece utilizzarne uno come vincolo di tipo per i metodi di interfaccia.

Ecco un esempio di parametro di tipo indicato come covariante nella definizione di un'interfaccia:

```
interface IBuilder<out T>
{
T Build();
}
```

Facciamo ora implementare l'interfaccia IBuilder a una classe dedicata a costruire istanze di classe Car:

```
class CarBuilder: IBuilder<Car>
{
public Car Build()
{
return new Car(); }
}
```

Ora è possibile assegnare un oggetto che implementi, per esempio, l'interfaccia IBuilder<Car> a una variabile di tipo IBuilder<Vehicle>.

```
CarBuilder cb=new CarBuilder();
IBuilder<Vehicle> vb=cb;
```

Senza l'annotazione out aggiunta al parametro di tipo dell'interfaccia IBuilder, l'ultima assegnazione non sarebbe accettata dal compilatore, perché il tipo CarBuilder non sarebbe convertibile in un oggetto IBuilder<Vehicle>.

Un assaggio dell'utilità di avere interfacce con parametri di tipo covarianti è dato dall'interfaccia IEnumerable. Riprendendo l'esempio di qualche paragrafo fa, il metodo StartRun può ora sfruttare tale interfaccia:

```
static void StartRun(IEnumerable<Vehicle> veicoli)
{
foreach (Vehicle v in veicoli)
v.Accelerate();
}
```

così potrà essere invocato anche passando un parametro IEnumerable di un tipo derivato da Vehicle:

```
List<Car> cars=new List<Car>(); //List<T> implementa IEnumerable di T
StartRun(cars);
```

Se un parametro di tipo viene marcato con la parola chiave in, esso sarà invece *controvariante*. La controvarianza è supportata nelle interfacce generiche solo se il parametro di tipo viene utilizzato come parametro di input dei metodi, come nel caso seguente:

```
public interface ICruiseControl<in T>
{
void SetSpeed(T obj, double speed);
}
```

Creiamo ora una classe che implementi tale interfaccia, per esempio per agire su oggetti di tipo Vehicle, supponendo che la classe Vehicle abbia una proprietà pubblica Speed:

```
class VehicleCruiseControl: ICruiseControl<Vehicle>
{
public void SetSpeed(Vehicle v, double speed)
{
V.Speed=speed;
}
}
```

Poiché l'interfaccia ICruiseControl è controvariante, è possibile assegnare un'istanza di VehicleCruiseControl a un riferimento ICruiseControl<Car>:

```
VehicleCruiseControl ccVehicle=new VehicleCruiseControl();
ccVehicle.SetSpeed(new Vehicle(), 100);
ICruiseControl<Car> ccCar = ccVehicle;
```

L'ultima istruzione restituirebbe un errore se l'interfaccia non avesse il parametro di tipo controvariante.

Riepilogando, se un parametro di tipo è marcato con out, esso viene detto covariante. Se invece un parametro è marcato con in, viene detto controvariante. Se, infine, non vi è alcuna annotazione, il parametro si dice essere invariante.

Nel seguente esempio di interfaccia, X è un parametro covariante, Y è controvariante, Z è invariante:

```
interface MyInterface<out X, in Y, Z>
{
X Metodo(Y y);
Z Proprieta { get; set; }
}
```

Implementare delegate varianti

Nel prossimo capitolo si approfondirà il concetto di delegate, accennato qualche paragrafo fa.

In .NET 4, è possibile convertire implicitamente un tipo di delegate generico in un altro utilizzando i meccanismi di varianza fin qui visti.

Per abilitare tale conversione, è necessario marcare i parametri di tipo come covarianti o controvarianti utilizzando rispettivamente le parole chiave out e in.

Creiamo ora un delegate generico, come visto nel Paragrafo "Delegate generici" in maniera che esso rappresenti, per esempio, un metodo per creare un oggetto di un dato tipo T:

```
public delegate T CreationDelegate<T>();
```

Esso può essere reso covariante in questo modo:

```
public delegate T CreationDelegate<out T>();
```

Così, entrambi i metodi seguenti sono compatibili con il delegate:

```
public string CreateString() { return "";}
public object CreateObject() { return null;}

CreationDelegate<string> delStr= CreateString;
CreationDelegate<object> delObj=delStr;
```

L'ultima assegnazione è permessa dal fatto che i delegate CreationDelegate<string> e CreationDelegate<object> sono stati resi compatibili dall'utilizzo della parola chiave out, che ha

marcato il parametro di tipo T come covariante.

Mediante la parola chiave in, invece, è possibile sfruttare la controvarianza per i parametri di tipo passati in ingresso a un delegate.

```
public delegate int ConvertToIntDelegate<in T>(T obj);
```

NOTA	Parametri che usano modificatori ref e out non possono essere marcati come parametri varianti. Nel caso precedente, quindi, se ref o out apparissero come modificatori del parametro obj, il tipo T non potrebbe essere marcato come controvariante.

Il seguente metodo è compatibile con il delegate:

```
public int ConvertObject(object obj)
{
return Convert.ToInt32(obj);
}
```

Pertanto, si può assegnare a un oggetto delegate nel seguente modo:

```
ConvertToIntDelegate<object> delConvObj = ConvertObject;
```

Grazie alla controvarianza del parametro, l'oggetto delConvObj è a sua volta compatibile con un delegate che usa string come corrispondente argomento di tipo, nonostante string sia più specifico di object:

```
ConvertToIntDelegate<string> delConvStr = delConvObj;
```

E infatti, ora si può anche invocare il delegate, utilizzando un valore string come argomento di ingresso:

```
delConvStr.Invoke("123");
```

È infine possibile supportare contemporaneamente covarianza e controvarianza nello stesso delegate, ma per parametri di tipo differenti. Per esempio:

```
public delegate U ConvertToIntDelegate<in T, out U>(T obj);
```

Collezioni in .NET

Nel Capitolo 3 abbiamo visto come C# offra un semplice tipo, l'array, per creare sequenze di elementi di un qualunque tipo riferimento o valore. Essendo un tipo supportato intrinsecamente dal Common Type System e dal CLR, gli array sono particolarmente efficienti.

Essi soffrono però anche di una serie di limitazioni. La principale è, naturalmente, che hanno una lunghezza fissa, da stabilire a priori al momento della loro creazione, e ciò non consente di aggiungere nuovi elementi al vettore, se si è raggiunto il limite stabilito. Il .NET Framework definisce quindi altri tipi di collezioni, che rappresentano varie tipologie di strutture dati utili a lavorare con insiemi di oggetti.

Il namespace System.Collections di .NET, in particolare, contiene classi pronte all'uso, interfacce che definiscono le varie operazioni possibili su di esse, e classi astratte da utilizzare come base per le proprie collezioni personalizzate.

Ogni classe di questo namespace implementa l'interfaccia ICollection, che a sua volta implementa IEnumerable, aggiungendo poi, eventualmente tramite l'implementazione di altre interfacce, funzionalità specifiche per la particolare struttura dati che essa rappresenta.

La Figura 9.2 mostra le classi e le interfacce principali del namespace System.Collections.

Un altro namespace, System.Collections.Specialized, contiene altri tipi di raccolte fortemente tipizzate e specializzate nella gestione di particolari elementi: per esempio, StringCollection per trattare collezioni di stringhe anziché di object, e StringDictionary che invece implementa una tabella in cui chiave e valore sono di tipo string.

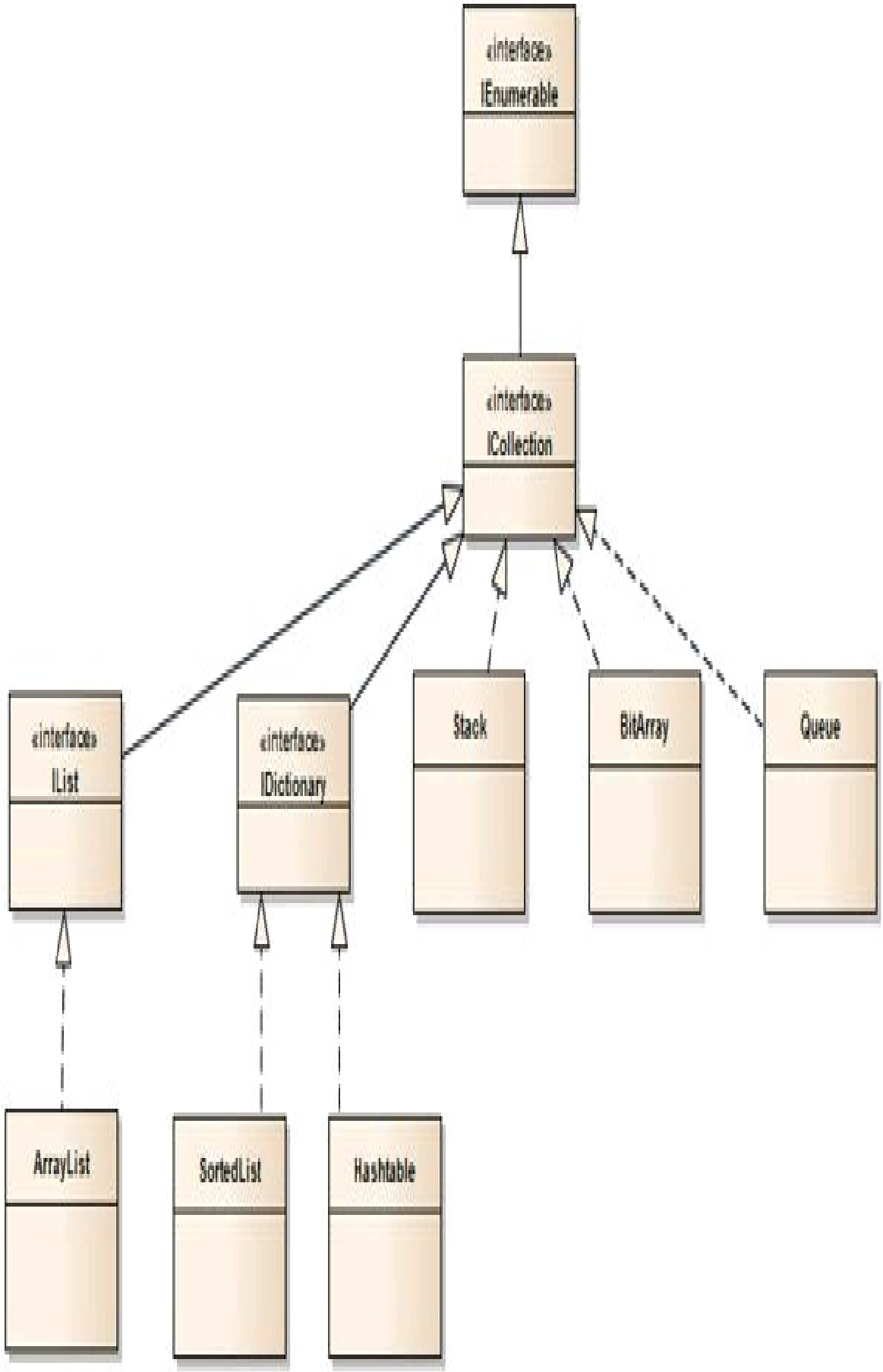

«interface»
IEnumerable
«interface»
ICollection
«interface»
IList
«interface»
IDictionary
Stack
BitArray
Queue
ArrayList
SortedList
Hashtable

Figura 9.2 - Diagramma delle classi del namespace System.Collections.

Dall'introduzione dei generics, la maggior parte delle classi standard messe a disposizione in tali namespace sono generiche, come vedremo a breve. Permettono quindi di memorizzare e manipolare qualunque tipo di oggetto, senza dover ricorrere alla superclasse System.Object per renderle utilizzabili in maniera generale, ed evitando quasi sempre anche la necessità di implementare nuove classi specializzate.

I tipi per lavorare con collezioni generiche sono contenuti nel namespace System.Collections.Generic.

Interfacce per collezioni

Tutte le collezioni condividono una serie di operazioni da eseguire sui loro elementi: per aggiungerne, inserirne, ottenere quello che si trova in una particolare posizione, rimuoverne alcuni, ordinarli, ricercarli e così via.

L'approccio di .NET Framework è pertanto quello di definire delle interfacce comuni, presenti sia in formato generico che non, e che sono implementate dai tipi di collezioni della libreria di base, oppure implementabili anche da classi di collezioni personalizzate.

Ogni classe che rappresenta un dato tipo di raccolta di oggetti implementa una o più di tali interfacce per fornire i metodi necessari al suo funzionamento.

Le interfacce sono disponibili sia in versione generica sia non generica, anche se queste ultime possono considerarsi mantenute per compatibilità con il passato.

Le principali interfacce messe a disposizione da .NET, e implementate dalle classi che poi esamineremo nel dettaglio, sono le seguenti:

- IEnumerable<T> e IEnumerable – forniscono le funzionalità per enumerare gli elementi di una collezione;
- ICollection<T> e ICollection – forniscono funzionalità per manipolare le collezioni, aggiungendo e rimuovendo elementi, e proprietà per ricavarne il numero;
- IList<T> e IList – forniscono funzionalità per l'accesso diretto agli elementi mediante indice;
- IReadOnlyList<T> – è la versione a sola lettura dell'interfaccia IList, in quanto permette di ottenere il numero di elementi e di leggerli mediante indice;
- IDictionary<TKey,TValue> e IDictionary – permettono di implementare collezioni di coppie chiave/valore;
- IReadOnlyDictionary<K,V> – è la versione a sola lettura della precedente;
- ISet<T> – rappresenta insiemi che possono contenere oggetti distinti.

Le classi che implementano le interfacce, combinando le varie funzionalità, saranno esposte nel seguito del capitolo.

Enumerare elementi

Sebbene le classi di collezioni possano avere una struttura dati diversa, uno dei requisiti fondamentali è quello di poter enumerare tutti gli elementi che compongono una raccolta di oggetti. Tale requisito è soddisfatto mediante l'implementazione delle interfacce IEnumerable e IEnumerator, e delle corrispondenti versioni generiche IEnumerable<T> e IEnumerator<T>.

L'ormai nota istruzione foreach, che ripete un blocco di istruzioni per ogni elemento di un insieme, funziona su collezioni che implementano l'interfaccia IEnumerable o l'interfaccia IEnumerable<T>. Queste espongono a loro volta un unico metodo, GetEnumerator, che restituisce un oggetto enumeratore di tipo IEnumerator o IEnumerator<T>:

```
public interface IEnumerable
{
IEnumerator GetEnumerator();
}
```

Per mezzo di tale enumeratore, l'istruzione foreach riesce a scorrere lungo tutti gli elementi della raccolta. Infatti, un IEnumerator e la sua controparte generica mettono a disposizione una proprietà Current, che restituisce l'elemento corrente, e un metodo MoveNext, che consente di spostarsi a quello successivo.

Un ulteriore metodo Reset consente infine di ritornare alla posizione iniziale per ricominciare l'enumerazione da capo.

In pratica, un'istruzione foreach come la seguente:

```
string[] nomi=new string[] {"antonio", "paolo", "mario", "gino", "luigi"};
foreach(string nome in nomi)
{
Console.WriteLine(nome);
}
```

è trasformata dal compilatore in un blocco di istruzioni simile al seguente:

```
IEnumerator enumerator= ((IEnumerable)nomi).GetEnumerator();
string nome;
while (enumerator.MoveNext())
{
nome = (string)enumerator.Current;
Console.WriteLine(nome);
}
```

Per fortuna, grazie all'istruzione foreach, tale complessità è nascosta allo sviluppatore.

Le versioni generiche delle interfacce IEnumerable e IEnumerator derivano dalle loro controparti non generiche:

```
interface IEnumerator<T>: IEnumerator, IDisposable
{
T Current {get;}
}

interface IEnumerable<out T>: IEnumerable
{
IEnumerator<T> GetEnumerator();
}
```

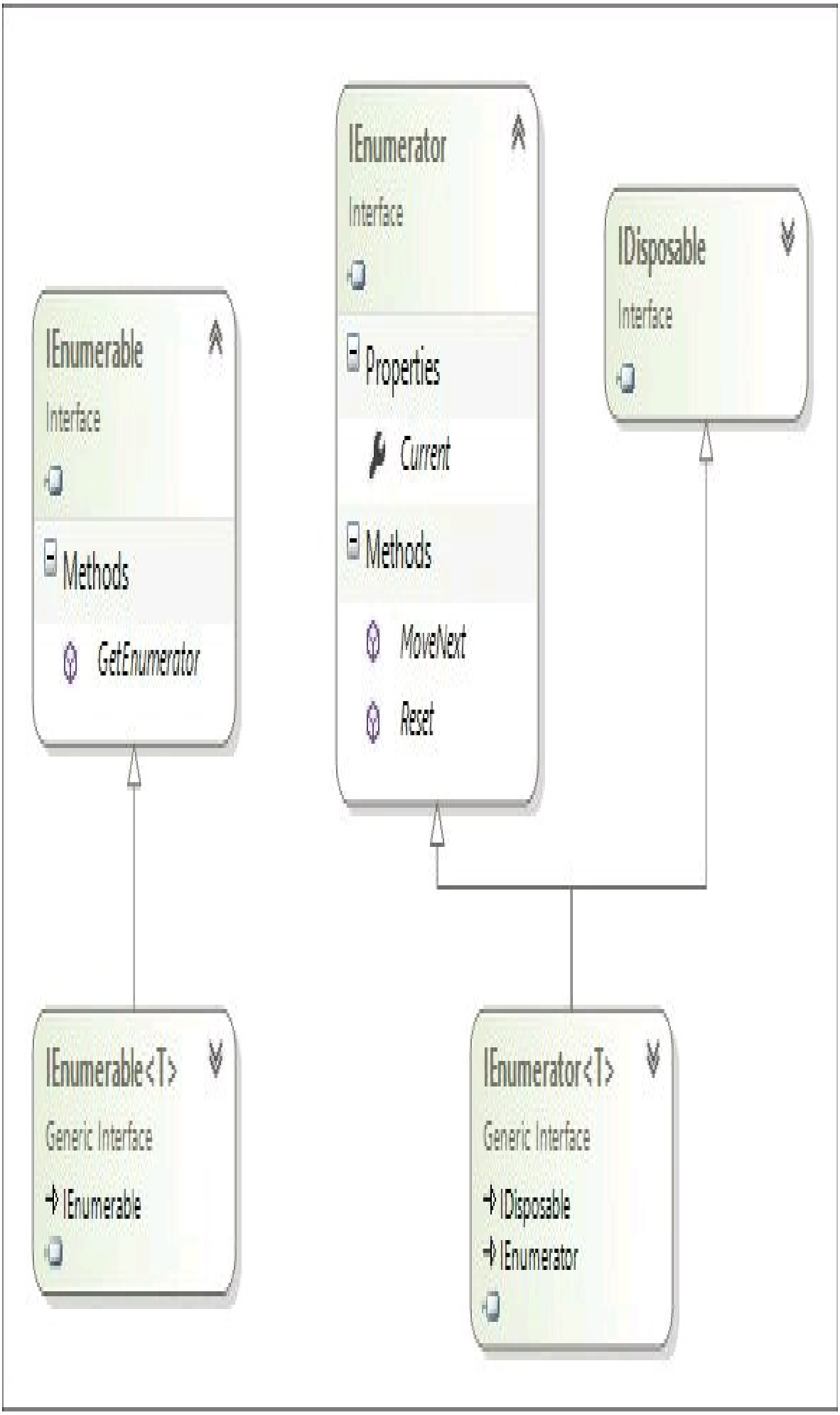

IEnumerable
Interface
Methods
GetEnumerator
IEnumerator
Interface
Properties
Current
Methods
MoveNext
Reset
IDisposable
Interface
IEnumerable<T>
Generic Interface
IEnumerable
IEnumerator<T>
Generic Interface
IDisposable
IEnumerator

Figura 9.3 - Diagramma delle interfacce IEnumerable e IEnumerator.

Grazie all'introduzione dei generics, viene garantita maggiore sicurezza rispetto ai tipi. Gli elementi contenuti nelle collezioni, infatti, dovranno essere necessariamente di un ben determinato tipo e non più trattati come object.

NOTA	Si noti la presenza della parola chiave out per il parametro di tipo dell'interfaccia IEnumerable<T>: come abbiamo visto qualche paragrafo fa, essa indica che l'interfaccia è controvariante.

L'interfaccia IEnumerable<T> è alla base di *LINQ to Objects*, che vedremo nel Capitolo 11.

Anche la classe System.Array implementa, a partire da .NET 2.0, l'interfaccia generica IEnumerable<T>. Quindi, un array di stringhe è per esempio utilizzabile come IEnumerable<string> e tramite questa si potrà ottenere un IEnumerator<string>:

```
IEnumerator<string> enumeratorArray= ((IEnumerable<string>)nomi).GetEnumerator();
```

Come visto sopra, inoltre, l'interfaccia IEnumerator<T> implementa anche l'interfaccia IDisposable. In tal modo, gli enumeratori possono scorrere oggetti che necessitano di essere rilasciati una volta terminata l'enumerazione.

In questo caso, l'istruzione foreach utilizzerà implicitamente un'istruzione using per chiamare il metodo Dispose al termine del proprio blocco:

```
using(IEnumerator enumerator= ((IEnumerable)nomi).GetEnumerator()
{
…
}
//invoca enumerator.Dispose
```

Implementazione di enumerazioni

Quando si ha necessità di aggiungere alle proprie classi delle funzionalità da collezione, o perlomeno di scorrere un insieme di elementi, è necessario implementare le interfacce IEnumerable e IEnumerator, che esistono in versione generica e non.

In C# 1.0, non esistendo ancora i tipi generici, per implementare l'interfaccia IEnumerable era necessario scrivere un po' di codice a mano, implementando come di consueto il metodo esposto dall'interfaccia stessa.

Il caso più semplice è quello in cui la classe che debba essere enumerata incapsuli al suo interno un oggetto rappresentante esso stesso una collezione, in modo da poterne invocare il metodo GetEnumerator. La seguente classe, per esempio, rappresenta una collezione di stringhe, memorizzate all'interno del campo stringhe:

```
class StringColl: IEnumerable
{
private string[] vettore;
public StringColl()
{
vettore = new string[] { "a", "b", "c" };
}

public IEnumerator GetEnumerator()
{
return vettore.GetEnumerator();
}
}
```

In tal modo è possibile utilizzare un oggetto della classe StringColl con un'istruzione foreach:

```
StringColl myColl=new StringColl();
foreach(var s in myColl)
{
Console.WriteLine(s);
}
```

Invece, quando non si ha a che fare con una collezione, è anche necessario creare una classe che implementi l'interfaccia IEnumerator, e quindi creare e fare restituire una sua istanza al metodo GetEnumerator. Ciò implica la scrittura di ancora più codice e la gestione di una macchina di stato, per esempio in maniera che a ogni invocazione del MoveNext si tenga conto dello spostamento di una sorta di cursore e si tenga traccia dell'oggetto da restituire per mezzo della proprietà Current.

La seguente classe implementa l'interfaccia IEnumerator:

```
class StringEnumerator : IEnumerator
{
private string[] strings;
int position;
public StringEnumerator(string[] strings)
{
this.strings = new string[strings.Length];
strings.CopyTo(this.strings, 0);
this.position = -1;
}

public object Current
{
get
{
if(position>-1 && position<strings.Length)
return strings[position];
throw new InvalidOperationException();
}
}
public bool MoveNext()
{
if (position < strings.Length - 1)
```

```
{
position++;
return true;
}
return false;
}
public void Reset()
{
position = -1;
}
}
```

Il campo position mantiene l'indice dell'elemento da restituire. La classe StringEnumerator viene poi utilizzata dal metodo GetEnumerator della seguente classe:

```
class StringColl: IEnumerable
{
private string[] vettore;
StringEnumerator enumerator;
public StringColl()
{
vettore = new string[] { "a", "b", "c" };
enumerator=new StringEnumerator(vettore);
}

public IEnumerator GetEnumerator()
{
return enumerator;
}
}
```

Un approccio più flessibile, e che non richiede necessariamente l'utilizzo di una collezione interna su cui effettuare un ciclo, è quello realizzato per mezzo di un iteratore, come verrà mostrato nel prossimo paragrafo.

Iteratori

Un *iteratore* è una funzionalità introdotta in C# che consente di implementare classi enumerabili ed enumeratori in maniera molto più semplice e flessibile di quanto mostrato nel paragrafo precedente.

Mediante l'utilizzo di un'apposita istruzione, yield, sarà il compilatore a occuparsi di tutto, evitando allo sviluppatore la scrittura manuale del codice necessario all'implementazione dell'interfaccia IEnumerator.

Il costrutto da implementare in questi casi viene appropriatamente chiamato *iteratore*.

Un iteratore è un blocco di codice che contiene una o più istruzioni yield. Tale blocco può essere un metodo, una proprietà di sola lettura o anche il corpo di un operatore. Per capire il

funzionamento dell'istruzione yield, e quindi degli iteratori, vediamo intanto un metodo che restituisce una sequenza di due stringhe:

```
public IEnumerable<string> GetNames()
{
yield return "antonio";
yield return "caterina";
}
```

Il modo con cui il metodo GetNames restituisce le due stringhe è differente da quello visto finora.

Il tipo di ritorno dichiarato dal metodo è un IEnumerable<string>, oggetto che non appare restituito in nessun punto del metodo: anzi, non è presente alcuna istruzione return. Esso, invece, contiene due istruzioni yield return, ognuna delle quali indica che c'è un altro elemento nell'enumerazione e quindi causa l'emissione di una stringa.

Infatti, utilizzando ora un'istruzione foreach per stampare le stringhe restituite dal metodo GetNames:

```
foreach (string str in iter.GetNames())
{
Console.WriteLine(str);
}
```

verranno stampate le due stringhe restituite per mezzo delle due istruzioni yield return.

Un iteratore implementato e invocato tramite il nome di un metodo è detto iteratore *named* o *denominato*, e può naturalmente accettare anche dei parametri in ingresso e controllare quindi il modo di restituire gli elementi al chiamante.

Una stessa classe può anche implementare più iteratori per mezzo di metodi e/o di proprietà. Naturalmente, nel caso di proprietà, l'iteratore dovrà essere implementato all'interno del blocco di accesso get:

```
public IEnumerable<string> Strings
{
get
{
yield return "hello";
yield return "world";
}
}
```

Un iteratore può restituire un IEnumerable, come nei casi precedenti di iteratori denominati, oppure un IEnumerator (o le loro versioni generiche, se necessario), che potrà essere utilizzato all'interno di una classe che implementa l'interfaccia IEnumerable, ed esattamente all'interno del metodo GetEnumerator: sarà il compilatore a gestire automaticamente il tutto.

```
class StringColl: IEnumerable
{
private string[] vettore;
public StringColl()
{
vettore = new string[] { "a", "b", "c" };
}
public IEnumerator GetEnumerator()
{
foreach(string str in vettore)
yield return str;
}
}
```

Come nell'esempio precedente, nel metodo GetEnumerator non appare nessuna istruzione return.

È il compilatore che dietro le quinte realizzerà una classe IEnumerator innestata, la istanzierà e la restituirà all'interno del metodo.

Utilizzando lo stesso approccio, è possibile implementare la versione generica di IEnumerable<T>:

```
class StringColl : IEnumerable<string>
{
private string[] vettore;
public StringColl()
{
vettore = new string[] { "a", "b", "c" };
}
IEnumerator IEnumerable.GetEnumerator()
{
return GetEnumerator();
}
public IEnumerator<string> GetEnumerator()
{
foreach (string str in vettore)
yield return str;
}
}
```

Si noti come in questo caso, dato che IEnumerable<T> a sua volta deriva da IEnumerable, sia necessario implementare non solo la versione generica, ma anche quella non generica di GetEnumerator. La versione non generica invoca a sua volta la versione generica, all'interno della quale l'enumerazione è implementata per mezzo di un blocco iteratore.

Un blocco iteratore può essere anche infinito, nel senso che non deve necessariamente restituire elementi prendendoli da un insieme. In questo caso, sarà necessario poter interrompere l'enumerazione: per esempio, al verificarsi di una data condizione. Se si vuol terminare prematuramente un iteratore, si utilizza un'istruzione yield break.

```
Random rand = new Random();
public IEnumerable<double> GetRandomNumbers(double soglia)
{
while (true)
{
double d = rand.NextDouble() * 100;
if (d < soglia)
yield break;
yield return d;
}
}
```

Il metodo implementa un *named iterator*, al quale è possibile passare un parametro soglia, che controlla quando terminare il ciclo di enumerazione. All'interno di tale ciclo viene restituito un nuovo numero casuale, generato dalla classe Random.

```
foreach(double numero in GetRandomNumbers(10))
{
Console.WriteLine(numero);
}
```

Il ciclo foreach precedente stampa tutti i numeri casuali ottenuti, fino a quando non ne venga generato uno inferiore alla soglia 10, passata come argomento. Ciò dimostra che dietro gli iteratori non ci sono semplici liste di elementi. Di conseguenza, un iteratore permette di ottimizzare le prestazioni senza dover necessariamente mantenere in memoria l'intero insieme di elementi, ma creandoli e restituendoli quando necessario.

La classe System.Array

La classe Array nel namespace System è la classe madre di ogni array in C#.

Anche la classe System.Array, pur facendo parte di un altro namespace, implementa le interfacce definite nel namespace System.Collections: esattamente, ICollection, IEnumerable e IList (oltre a qualche altra).

NOTA	Come già detto nei paragrafi precedenti, a partire da .NET 2.0, la classe Array implementa a runtime le versioni generiche delle interfacce suddette: ICollection<T>, IList<T> e IEnumerable<T>.

Derivando dalla classe Array, ogni vettore ha a disposizione un insieme di metodi comuni, a prescindere dal tipo di elementi che esso conterrà.

Come già visto, ed essendo un tipo fondamentale, C# fornisce una sintassi speciale per dichiarare, inizializzare e accedere agli elementi di un array:

```
int[] vettore=new int[10];
int[] vettore2= {1,2,3,4};
vettore[0]=1;
int primo=vettore2[0];
```

Quando si dichiara un array come il precedente, il CLR crea implicitamente una classe derivata da Array, la quale a sua volta implementa delle interfacce generiche per trattare il tipo di elementi indicato.

Per esempio: nel caso precedente, l'array di int implementa l'interfaccia IList<int>. In alternativa alla sintassi precedente, è possibile utilizzare direttamente la classe Array e i suoi metodi. In particolare, il metodo CreateInstance permette di ottenere un array monodimensionale di elementi dell'oggetto Type specificato e della lunghezza indicata:

```
Array vettore=Array.CreateInstance(typeof(int), 10); //equivalente a new int[10]
```

Altri overload permettono di creare array multidimensionali:

```
vettore.SetValue(1,0);
int val=(int)vettore.GetValue(0);
```

I metodi GetValue e SetValue nel codice di cui sopra permettono, rispettivamente, di ottenere un elemento e impostare un oggetto alla posizione indicata.

Inoltre, a partire dall'oggetto Array, è possibile eseguire un cast esplicito per ottenere un array C# di un tipo compatibile: rispettando, naturalmente, le regole di covarianza viste nel Paragrafo "Covarianza e controvarianza".

```
int[] array=(int[])vettore;
```

Fra le proprietà definite dalla classe Array, e utili a trattare con essa, vi sono le proprietà Length e LongLength che rappresentano il numero totale di elementi dell'array in tutte le sue dimensioni, rispettivamente in formato int e long. Per esempio, se volessimo utilizzare un ciclo for per scorrere e inizializzare ogni elemento presente nel vettore precedente, scriveremmo:

```
//crea un array: 0,1,2,3,4,5,6,7,8,9
for(int i=0; i< vettore.Length; i++)
{
vettore[i]=i;
}
```

Oppure, per accedere all'ultimo elemento dell'array, essendo l'indice a base zero, scriveremmo:

```
int ultimo=vettore[vetore.Length-1]
```

Se si desidera conoscere il numero di dimensioni dell'array, è a nostra disposizione la proprietà Rank:

```
int dimensioni=vettore.Rank;
```

Per esempio, un array normale restituirà 1, una matrice bidimensionale 2 e così via.

Per enumerare gli elementi, è anche possibile utilizzare l'istruzione foreach, senza quindi dover necessariamente ricavare il numero di elementi:

```
foreach(var elemento in vettore)
{
Console.WriteLine(elemento);
}
```

Un'altra opzione è quella di utilizzare il metodo statico della classe Array stessa, ForEach, che ha questa firma:

```
Array.ForEach(T[] array, Action<T> action);
```

Action<T> è un delegate che incapsula un metodo che esegue un'azione sull'oggetto che gli è stato passato; quindi, tramite il metodo ForEach, tale azione sarà eseguita su tutti gli elementi. Per mezzo di esso, possiamo implementare rapidamente un'istruzione che stampi tutti gli elementi dell'array:

```
Array.ForEach(new int[]{1,2,3,4,5}, Console.Write);
```

La classe Array fornisce poi una serie di metodi utili per ricercare, ordinare, invertire, copiare, e in generale manipolare gli elementi di un vettore.

Il metodo statico Clear permette di impostare a zero (per array di tipo valore) oppure a null (tipo riferimento) il valore di tutti gli elementi di un array o di un loro intervallo. Per esempio, per azzerare tutti gli elementi di un array di int, si può scrivere:

```
int[] vettore={1,2,3,4,5,6,7,8,9};
Array.Clear(vettore, 0 , vettore.Length);
```

Per ordinare gli elementi dell'array, si può utilizzare il metodo Sort o uno dei suoi overload. Il metodo Sort utilizza un algoritmo *QuickSort* ed è necessario che il tipo degli elementi implementi l'interfaccia IComparable o IComparable<T>. Ogni elemento deve essere infatti confrontato con tutti gli altri per poterne decidere il corretto ordinamento. Il modo più semplice di utilizzarlo è il seguente:

```
var vettoreOrdinato=Array.Sort(vettore);
```

Una versione generica del metodo, Sort<T>, permette di ordinare Array di qualunque tipo utilizzando l'interfaccia IComparable<T>.

Molti dei tipi forniti dalla libreria .NET implementano l'interfaccia IComparable/IComparable<T>, quindi possono essere confrontati e ordinati in una collezione.

Per ordinare gli elementi di un array secondo un ordine personalizzato, si può invece utilizzare un altro overload di Sort, che usi un oggetto IComparer oppure IComparer<T>.

La seguente classe generica implementa un algoritmo di confronto in maniera da ordinare gli elementi in maniera decrescente.

```
class ReverseComparer<T> : IComparer<T> where T:IComparable
{
public int Compare(T x, T y)
```

```
{
return -(x.CompareTo(y));
}
}
```

Poiché il vincolo implica che il tipo degli elementi implementi l'interfaccia IComparable, il metodo Compare non fa altro che invocare il metodo CompareTo su ogni oggetto e restituirne il risultato negato.

A questo punto, per ordinare il vettore precedente basta scrivere il seguente metodo:

```
Array.Sort(vettore, new ReverseComparer<int>());
```

NOTA	Un'altra possibilità è utilizzare un metodo che rispetti la firma del delegate IComparison. Tuttavia, considerato che i delegate li affronteremo nel prossimo capitolo, rinviamo questa opzione.

Un altro metodo utile per modificare l'ordine degli elementi è il metodo Reverse:

```
Array.Reverse(vettore);
```

Per riprendere il discorso di ordinamento alla rovescia, si sarebbe potuto anche utilizzare il metodo Reverse, subito dopo la chiamata a Sort:

```
vettore=new int[] {2, 1, 3};
Array.Sort(vettore); //ordina in maniera crescente
//vettore è ora 1,2,3
Array.Reverse(vettore); //inverte gli elementi;
//vettore è ordinato al contrario 3,2,1
```

Per ricercare elementi particolari all'interno di un Array, si possono utilizzare diversi metodi a seconda del risultato che si è interessati a ottenere.

Per trovare un particolare elemento, usando un efficiente algoritmo di ricerca binaria, ci si può servire del metodo BinarySearch. Esso, però, funziona solo su array ordinati.

Supponendo che x sia l'intero da ricercare all'interno dell'array vettore, si può quindi scrivere:

```
index=Array.BinarySearch(vettore, x);
```

I metodi IndexOf e LastIndexOf funzionano anche con array non ordinati:

```
index = Array.IndexOf(vettore, x);
```

Per trovare l'ultima posizione in cui si trova l'elemento x, si può utilizzare invece il metodo analogo LastIndexOf:

```
index = Array.LastIndexOf(vettore, x);
```

Nel caso in cui i metodi precedenti non trovino alcun elemento, non sarà generata alcuna eccezione, ma verrà semplicemente restituito come risultato il valore -1. Questi metodi,

inoltre, effettuano una ricerca per uguaglianza, eseguendo una sorta di enumerazione degli elementi fino a trovare quello cercato.

Una serie di altri metodi di ricerca utilizza invece i delegate, per i quali ancora una volta vi rinviamo al prossimo capitolo. Intanto, mostreremo qui qualche semplice esempio utile per capire come utilizzare i metodi di Array.

NOTA	Come si vedrà nel prossimo capitolo, a partire da C# 2.0 è possibile utilizzare anche un metodo anonimo al posto dei delegate, che evita la necessità di creare un metodo apposito. Con C# 3.0, invece, un'opzione ancora più concisa è quella delle espressioni lambda.

In particolare, il delegate da utilizzare è del tipo Predicate<T>. Praticamente, esso rappresenta qualunque metodo accetti come argomento un oggetto e restituisca un booleano, cioè un valore true o false a seconda di determinate condizioni da verificare sull'oggetto stesso:

```
public delegate bool Predicate<T>(T obj);
```

Un possibile metodo che può quindi essere utilizzato come predicato è il seguente, che restituisce true se l'intero passato come argomento è pari:

```
public static void IsPari(int x)
{
return x%2==0;
}
```

Nel seguente esempio, viene invocato il metodo statico Find della classe Array per trovare il primo numero pari, e come secondo argomento del metodo viene utilizzato appunto il Predicate costruito mediante il metodo IsPari:

```
int primoPari= Array.Find(vettore, new Predicate<int>(IsPari));
```

Come impareremo, per usare un Predicate, o in generale un delegate, è possibile utilizzare una versione abbreviata, indicando solo il nome del metodo. Per esempio, per trovare l'ultimo elemento dell'array che rispetta il predicate IsPari, basta scrivere:

```
int ultimoPari= Array.FindLast(vettore, IsPari);
```

In maniera analoga, il metodo FindAll restituisce tutti gli elementi che soddisfano una condizione. Quindi, il tipo di ritorno è a sua volta un array dello stesso tipo di quello su cui si effettua la ricerca:

```
int[] numeriPari=Array.FindAll(vettore, IsPari);
```

Altri metodi di ricerca utili forniti dalla classe Array, e che utilizzano un predicate come quelli appena visti, sono Exists e TrueForAll. Il primo verifica se esiste un elemento che soddisfa il predicate, il secondo verifica se tutti gli elementi soddisfano la condizione:

```
bool esistePari=Array.Exists(vettore, IsPari);
bool tuttiPari=Array.TrueForAll(vettore, IsPari);
```

La classe System.Array fornisce poi dei metodi per eseguire la copia degli elementi di un array.

Due di questi metodi sono statici, Copy e ConstrainedCopy, mentre i metodi CopyTo e Clone sono metodi di istanza. Tutti quanti effettuano una copia cosiddetta *shallow*, superficiale: nel caso di elementi di tipo riferimento, solo i riferimenti sono copiati e non gli oggetti veri e propri.

Il metodo Copy esegue la copia di tutti gli elementi o di un sottoinsieme di essi:

```
int[] origine={1,2,3,4,5,};
int[] destinazione=new int[origine.Length];
Array.Copy(origine, destinazione, origine.Length);
```

Se si vuole invece copiare un sottoinsieme, come i primi tre elementi, nelle ultime tre posizioni di un array destinazione, si utilizzerà l'overload seguente:

```
int[] destinazione=new int[origine.Length];
Array.Copy(origine, 0, destinazione, 3, 3); //destinazione contiene ora 0,0,0,1,2,3
```

Il metodo ConstrainedCopy agisce in maniera atomica: se, per qualsiasi motivo, la copia anche solo di un elemento non va a buon fine, tutti i cambiamenti sono annullati.

La firma del metodo prevede l'indicazione dell'array di origine e indice di partenza, dell'array destinazione e indice di partenza, e del numero di elementi da copiare:

```
Array.ConstrainedCopy(origine, 0, destinazione, 0, origine.Length);
```

Invece, il metodo di istanza CopyTo, a partire da un determinato indice, copia su un array destinazione gli elementi dell'array sul quale viene invocato il metodo:

```
origine.CopyTo(destinazione, 0);
```

Il metodo Clone, infine, esegue una clonazione e restituisce una nuova istanza di array dello stesso tipo:

```
destinazione=(int[]) origine.Clone();
```

Per una trattazione più approfondita dei vari metodi e delle diverse proprietà della classe Array, e del loro modo di utilizzo, è come sempre consigliabile un'analisi della relativa documentazione.

Tuple

Gli array e le collezioni generiche permettono di manipolare raccolte di oggetti dello stesso tipo.

Se si ha la necessità di creare e trattare collezioni di oggetti di tipo differente, in maniera fortemente tipizzata (altrimenti potremmo sempre creare array di object, o utilizzare le

collezioni non generiche), a partire dalla versione 4.0 di .NET è possibile utilizzare le cosiddette *tuple*.

Una tupla permette quindi di creare oggetti composti da più elementi, senza dover necessariamente creare una classe apposita, con i campi necessari per contenerli. Possono essere quindi utili in differenti frangenti: per rappresentare il record di un database composto da differenti campi di vario tipo, per esempio, oppure per far restituire a un metodo più valori, senza necessità di utilizzare parametri con modificatore out.

Esistono otto differenti versioni generiche della classe Tuple e una classe statica che permette di istanziare le precedenti:

```
Tuple<T1>
Tuple<T1, T2>
…
Tuple<T1, T2, T3, T4, T5, T6, T7, TRest>
```

Ogni differente versione supporta un numero diverso di oggetti. Per esempio, Tuple<T1> contiene un elemento, Tuple<T1, T2> due diversi elementi di tipo differente e così via.

La versione con otto elementi, che indica l'ultimo parametro di tipo con TRest, permette di creare tuple con più di sette elementi, innestando altre tuple come ultimo elemento.

NOTA	Sebbene la versione con un singolo parametro di tipo possa sembrare inutile, visto che conterrà un singolo valore, essa è necessaria per creare una 8-tupla, usando un parametro di tipo Tuple<T1> come ultimo parametro TRest.

Per istanziare una tupla, è possibile utilizzare il costruttore che ogni classe generica mette a disposizione. Per esempio, supponiamo di dover rappresentare un record di un database che contenga la popolazione di una città in un dato anno. Possiamo utilizzare una 3-tupla (per indicare una tupla con un certo numero di elementi si usa spesso la notazione *n-tupla*, dove n è il numero di elementi) in cui il primo tipo è string, per contenere il nome della città, il secondo è un int per la popolazione, il terzo è uno short che conterrà l'anno:

```
Tuple<string, int, short> record=new Tuple<string, int, short>("Roma", 2760000, 2010);
```

Un'altra modalità per creare tuple è usare la classe non generica Tuple, la quale possiede tanti overload del metodo statico Create quante sono le diverse tuple istanziabili.

Per esempio, per creare la tupla precedente possiamo usare la seguente sintassi:

```
Tuple<string, int, short> record=Tuple.Create<string, int, short>("Roma", 2760000, 2010);
```

Il metodo Create può avvantaggiarsi dell'inferenza dei tipi e quindi riconoscere automaticamente i parametri di tipo da utilizzare a partire dai valori usati come argomenti. La

precedente istruzione, per esempio, può anche essere scritta come segue:

```
var record=Tuple.Create("Roma", 2760000, (short)2010);
```

Ogni versione generica della classe Tuple mette poi a disposizione tante proprietà quanti sono i parametri utilizzati per ricavare i vari elementi che compongono la tupla.

Nel caso di 3-tupla o tripletta come la precedente, si può quindi ricavare ogni elemento così:

```
string nome=record.Item1;
int popolazione=record.Item2;
short anno=record.Item3;
```

Si noti che ognuna delle proprietà ItemX è tipizzata staticamente.

Come già accennato, le tuple possono risultare utili per consentire a un metodo di restituire più di un valore, senza utilizzare parametri out.

Supponiamo che un metodo della nostra applicazione legga da un database i dati sulla popolazione di una città e li voglia restituire a un metodo chiamante. Una possibile firma del metodo potrebbe quindi essere la seguente:

```
public Tuple<string,int,short> LoadCityData()
{
string nome="Roma";
int pop=2760000;
short anno=2010;
var record=Tuple.Create(nome, pop, anno);
return record;
}
```

Essendo un tipo riferimento, per confrontare due tuple è necessario utilizzare il metodo Equals, in quanto l'operatore di uguaglianza confronta solo i riferimenti:

```
var record1 = Tuple.Create<string, int, short>("Roma", 2870000, (short)2011);
var record2 = Tuple.Create<string, int, short>("Roma", 2870000, (short)2011);
if (record1 != record2)
Console.WriteLine("riferimenti a tuple diverse");

if (record1.Equals(record2))
Console.WriteLine("Le tuple sono uguali");
```

Liste di elementi

La classe List<T> e la non generica ArrayList rappresentano liste di oggetti a dimensione variabile. Esse hanno funzionalità simili a quelle di un array, con la differenza che la loro dimensione è dinamica, cioè essa si espande quando è necessario inserire nuovi elementi.

Nel seguito mostreremo esempi di codice utilizzando in genere la classe fortemente tipizzata List<T>, ma la corrispondente classe non generica ArrayList ha un utilizzo analogo, potendo contenere elementi di tipo object.

Una volta istanziata una lista, per aggiungere oggetti a essa si utilizza il metodo Add:

```
List<string> lista =new List<string>();
lista.Add("hello");
lista.Add("world");
```

Essendo una lista generica, essa è fortemente tipizzata. Quindi, una volta istanziata la classe, essa potrà contenere solo elementi del tipo corrispondente al parametro indicato: nel caso precedente, di tipo string.

L'ArrayList, invece, potrà contenere come già detto anche elementi eterogenei:

```
ArrayList lista=new ArrayList()M
lista.Add("hello");
lista.Add(1);
```

Se si conoscono in anticipo alcuni o tutti gli elementi che dovranno far parte della lista, possono essere passati direttamente al costruttore della stessa, sotto forma di un IEnumerable, per esempio da un array:

```
int[] array=new int[] {1,2,3,4};
List<int> listaInteri=new List<int>(numeri);
```

Il metodo Add inserisce gli elementi nell'ordine indicato, cioè uno dopo l'altro. Se invece si desidera inserire un elemento in una precisa posizione, è possibile utilizzare il metodo Insert:

```
lista.Insert(indice, "stringa");
```

È possibile inserire contemporaneamente più elementi utilizzando il metodo AddRange oppure il metodo InsertRange. Il primo inserisce gli oggetti contenuti in un IEnumerable in coda alla lista, mentre il secondo accetta anche un parametro indicante l'indice al quale inserire i nuovi elementi:

```
string[] nuovi=new string[] {"a", "b", "c"};
lista.InsertRange(2, nuovi);
```

Naturalmente, l'indice non deve assumere un valore al di là della dimensione attuale della lista.

Come per gli array, è possibile accedere a un elemento di una lista mediante l'indice:

```
string primo=lista[0];
```

In maniera analoga, è possibile modificare un elemento, assegnandolo sempre mediante indice:

```
lista[0]="ciao";
```

Ciò permette di utilizzare un ciclo for per accedere a tutti gli elementi, ma naturalmente, poiché IList<T> deriva da IEnumerable, per scorrere tutti gli elementi presenti in lista si può anche ricorrere all'istruzione foreach:

```
foreach(var element in lista)
{
Console.WriteLine(element);
}
```

Per la rimozione di tutti gli elementi è disponibile il metodo Clear, mentre è possibile rimuoverne anche solo uno per volta utilizzando il metodo Remove o il metodo RemoveAt:

```
lista.Remove("hello");
lista.RemoveAt(indice);
```

Entrambe le classi mantengono gli elementi in un array interno e, quando si raggiunge la capacità massima, questo viene sostituito dietro le quinte da uno più grande.

Infatti, è possibile utilizzare una proprietà Count per ottenere il numero effettivo di elementi presenti nella lista, mentre la proprietà Capacity rappresenta la dimensione della struttura interna, che in genere sarà maggiore del numero di elementi effettivo e che verrà aumentata quando necessario.

Per mezzo del metodo TrimExcess, è eventualmente possibile rimuovere le locazioni della struttura interna non necessarie, in maniera da risparmiare memoria.

```
lista.TrimExcess();
```

In tal modo, Count e Capacity saranno a questo punto coincidenti.

Le liste possono essere ordinate per mezzo del metodo Sort, che funziona in maniera analoga a quello visto per gli array. Diversi metodi permettono invece di effettuare ricerche di particolari elementi all'interno della lista:

```
int primo=lista.IndexOf("hello");
int ultimo=lista.LastIndexOf("hello");
bool contiene=lista.Contains("ciao");
```

Altri metodi permettono di ricercare tramite delegate, come Predicate, elementi che soddisfano particolari condizioni, oppure di agire sugli elementi delle liste utilizzando un delegate Action<T>.

Il delegate Predicate<T>, già incontrato in qualche esempio sulla classe Array, può essere utilizzato per verificare quali elementi di una lista soddisfino una condizione. Per esempio, per ricavare da una lista di nomi tutti quelli più lunghi di un certo numero di caratteri, possiamo innanzitutto scrivere un metodo che effettui tale verifica su una stringa generica:

```
public bool IsLongName(string str)
{
return str.Length>7;
}
```

Quindi possiamo utilizzare il metodo IsLongName come Predicate da passare al metodo Find, che restituisce la prima occorrenza trovata, o al metodo FindAll, che restituisce a sua volta una List di tutti gli oggetti che rispettano il Predicate:

```
string first=lista.Find(IsLongName);
string all=lista.FindAll(IsLongName);
```

Il delegate Action<T> è un metodo che esegue una determinata azione sull'oggetto passato come argomento, quindi gli elementi di List<T> sono passati individualmente a esso.

Per esempio, il metodo ForEach, che prende un delegate Action<T> come parametro, può essere utilizzato per stampare tutti gli elementi di una lista con una singola istruzione:

```
lista.ForEach(Console.WriteLine);
```

Dando un'occhiata alla documentazione, troverete numerosi altri metodi che permettono di trattare in maniera molto efficiente e flessibile liste di oggetti. Vi consigliamo quindi di provare a scrivere qualche esempio e verificare il funzionamento dei vari membri delle classi List<T> e ArrayList e delle relative interfacce.

Inizializzatori di collezione

A partire da C# 3.0, per inizializzare una collezione è possibile utilizzare una sintassi, già vista per gli array, che permette di indicare tutti gli elementi che fanno parte della collezione stessa in una singola istruzione. Nello specifico, tale funzionalità è permessa per tutte le classi che implementano l'interfaccia ICollection<T>.

Un inizializzatore di collezione permette di istanziare l'oggetto e di aggiungere i suoi elementi in un sol colpo:

```
List<string> lista=new List<string>() {"a", "b", "c";};
```

Si eviterà così di scrivere una sequenza di istruzioni Add, come nel caso seguente:

```
List<string> lista=new List<string>();
lista.Add("a");
lista.Add("b");
lista.Add("c");
```

Dietro le quinte, un inizializzatore di collezione come quello visto sopra non fa altro che invocare il metodo Add per ognuno degli elementi indicati fra parentesi graffe. Lo si può facilmente provare scrivendo una propria classe collezione. Per esempio, la seguente StringColl implementa l'interfaccia IEnumerable ed è dedicata a contenere una collezione di stringhe (naturalmente non ha senso pratico, visto che basterebbe usare una List<string>):

```
class StringColl : IEnumerable
{
public List<string> Stringhe {get;}
public StringColl()
```

```
{
Stringhe = new List<string>();
}

public IEnumerator GetEnumerator()
{
foreach (string str in Stringhe)
yield return str;
}
}
```

A questo punto, si potrebbe provare a inizializzare un'istanza della classe StringColl passando gli elementi fra parentesi graffe, nel seguente modo:

```
var coll = new StringColl() { "a", "b", "c"};
```

Il risultato sarebbe un errore di compilazione, che indicherebbe proprio la mancanza di un metodo Add per il tipo StringColl. Errore che, comunque, si risolverebbe implementando il metodo all'interno della classe StringColl stessa. Per esempio:

```
public void Add(string s)
{
stringhe.Add(s);
}
```

Con C# 6 è possibile anche utilizzare un metodo di estensione della classe, dando maggior flessibilità e possibilità agli sviluppatori:

```
public void Add(this StringColl coll, string s)
{
coll.Stringhe.Add(s);
}
```

Liste a sola lettura

La classe List<T> espone un metodo AsReadOnly che restituisce un oggetto di tipo ReadOnlyCollection<T>, il quale permette di accedere agli elementi della lista, ma non di modificarli.

Poiché l'oggetto restituito dal metodo AsReadOnly è un *wrapper*, esso riflette i cambiamenti eventualmente apportati alla lista sottostante, anche se non permette la modifica diretta degli elementi.

NOTA	In ambito software un wrapper è un modulo o un oggetto che ne riveste un altro, ovvero che funziona da tramite fra l'utilizzatore esterno e l'oggetto interno.

Per esempio, se da una lista ottengo una versione a sola lettura:

```
List<string> lista=new List<string>() {"antonio", "mario", "paolo", "franco"};
var listaReadOnly=lista.AsReadOnly();
```

l'oggetto listaReadOnly contiene tutti gli elementi di lista e non permette di modificarli.

Modificando ora un elemento della lista originale, tale variazione si riflette anche sull'oggetto listaReadOnly:

```
lista[2]="pippo";
listaReadOnly.ToList().ForEach(Console.WriteLine);
```

Si noti che, per stampare tutti gli elementi della seconda lista, si utilizza il metodo ForEach di List<T>, quindi è necessario prima ottenere tale oggetto per mezzo del metodo ToList.

Le interfacce generiche IReadOnlyList, IReadOnlyCollection e IReadOnlyDictionary sono state introdotte da .NET 4.5. A eccezione dell'ultima, esse sono covarianti, quindi possono essere utilizzate come argomenti di metodi che accettano parametri di un tipo base. Per esempio, se avessimo un metodo che accetta in ingresso una lista di Vehicle, potremmo passargli come argomento una lista di Car (considerando Car figlia della classe Vehicle).

La classe LinkedList

Una *lista concatenata* è una struttura dati che consiste in una sequenza di elementi o nodi, ognuno con uno o due riferimenti verso il nodo successivo e/o precedente. In particolare, l'implementazione di .NET è permessa dalla classe generica LinkedList<T>, che rappresenta una lista doppiamente concatenata o bidirezionale (perché ogni nodo ha due riferimenti, uno verso il precedente e uno verso il successivo, eventualmente null per il nodo finale).

NOTA	A differenza di altri tipi di collezioni, non esiste una versione non generica della classe LinkedList.

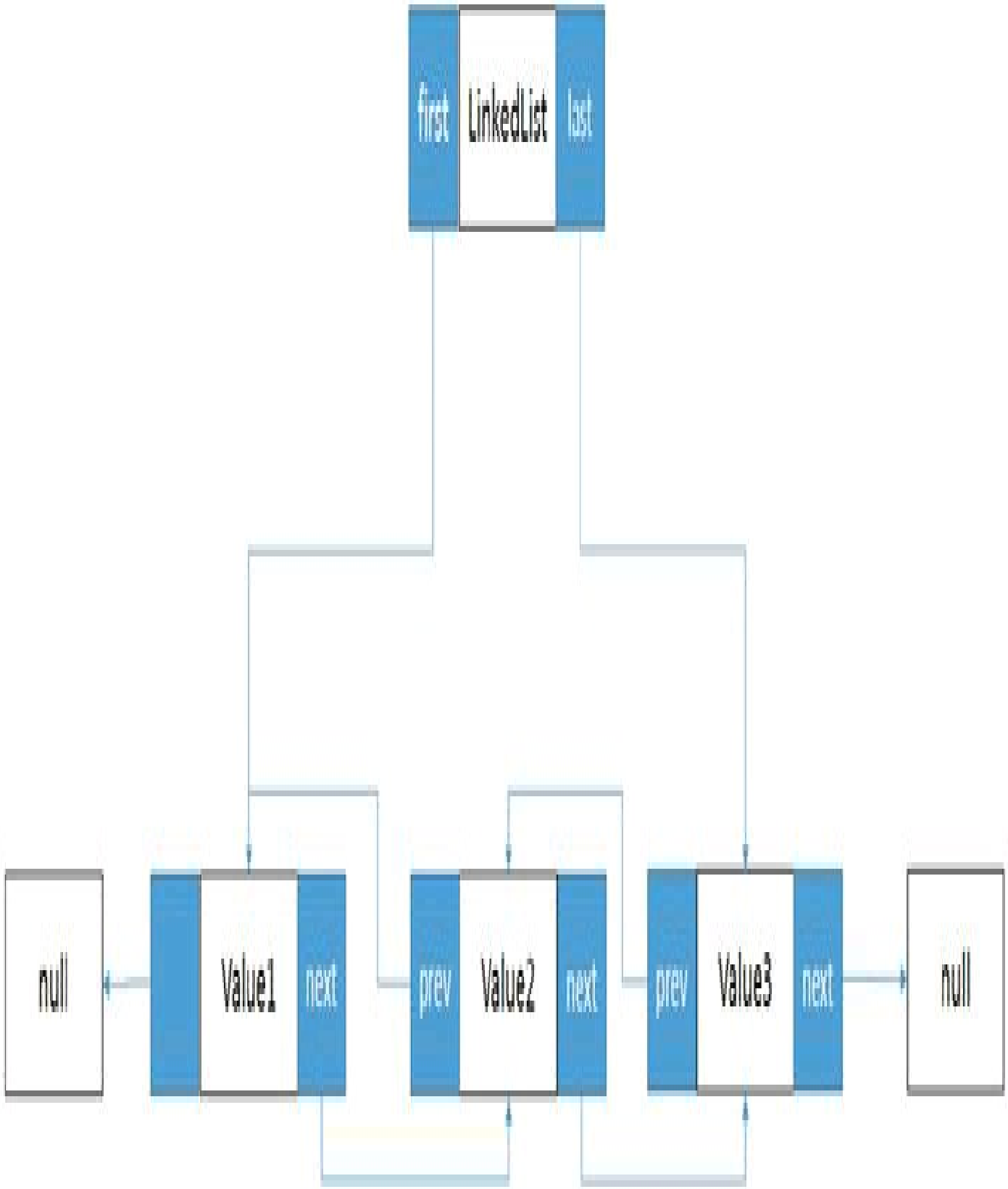

first
LinkedList
last
null
Value1
next
prev
Value2
next
prev
Value3
next
null

Figura 9.4 - Esempio di lista doppiamente concatenata.

La Figura 9.4 mostra una lista doppiamente concatenata, in cui ogni elemento contiene un oggetto ma anche i riferimenti agli elementi precedenti e successivi.

Il principale vantaggio di una struttura dati come la lista doppiamente concatenata è che un elemento può essere inserito in un punto qualsiasi della lista in maniera estremamente efficiente. Tale operazione, infatti, richiede solo la creazione di un nuovo nodo e l'impostazione dei riferimenti, non la gestione di un array interno, e quindi l'eventuale spostamento di un elemento al suo interno e la creazione di nuovi vettori.

Lo svantaggio, invece, è dato dall'impossibilità dell'accesso diretto a un elemento tramite indice: è consentito solo un accesso sequenziale, quindi anche la ricerca deve avvenire in tal senso. La classe LinkedList<T>, infatti, implementa le interfacce IEnumerable<T> e ICollection<T>, ma non IList<T>.

La classe LinkedList espone tutti i metodi e le proprietà necessarie a gestire tale struttura dati. Ogni nodo della lista è rappresentato da un'istanza della classe LinkedListNode<T>, che fornisce una proprietà Value per impostare e ottenere l'elemento contenuto nel nodo, e le proprietà Next e Previous per navigare la lista passando rispettivamente al nodo successivo o a quello precedente.

Nel seguito vedremo qualche esempio sull'utilizzo della classe LinkedList<T> e dei suoi membri.

Il modo più semplice per inizializzare una lista con dei nodi conosciuti è quello di utilizzare il costruttore che prende un IEnumerable come parametro; per esempio, a partire da una List<T> o da un array, potremmo creare una lista concatenata così:

```
List<string> list = new List<string>() { "elementi", "della", "lista" };
LinkedList<string> frasi = new LinkedList<string>(list);
```

Altrimenti si parte da una lista vuota:

```
LinkedList<string> frasi=new LinkedList<string>(); //lista vuota
```

A questo punto, per aggiungere nuovi elementi alla lista bisogna utilizzare il metodo AddFirst o AddLast, mentre con AddBefore e AddAfter si può aggiungerne uno rispettivamente prima e dopo un determinato nodo:

```
LinkedListNode<string> primo= frasi.AddFirst("questo"); //Aggiunge primo nodo
var last= frasi.AddLast("esempio"); //aggiunge come ultimo nodo
frasi.AddAfter(primo, "è"); //aggiunge dopo il primo
frasi.AddBefore(last, "un"); //aggiunge prima dell'ultimo
```

La classe espone anche una proprietà Count per ottenere il numero di nodi, mentre per elencarli tutti è naturalmente possibile usare un foreach:

```
Console.WriteLine("La lista ha {0} nodi ", frasi.Count);
foreach (var node in frasi)
{
Console.WriteLine(node);
}
```

Analogamente, sono presenti metodi per rimuovere elementi specifici dalla lista, oppure il primo, o l'ultimo:

```
frasi.Remove("questo");
frasi.RemoveFirst();
frasi.RemoveLast();
```

Ogni nodo possiede le proprietà Previous e Next per ottenere rispettivamente il predecessore e il successore, le quali saranno eventualmente null se essi non esistono: è questo il caso del primo nodo della lista, che non ha un Previous, e dell'ultimo, che non ha un Next.

La classe Queue

Una *coda* è una struttura dati di tipo *First-In-First-Out* (*FIFO*): il primo elemento che viene inserito in tale collezione sarà anche il primo a uscire, proprio come avviene alla coda di un casello autostradale o in cassa al supermercato.

In .NET una coda è implementata per mezzo della classe Queue<T> o della corrispondente non generica Queue.

I metodi principali di tali classi sono quelli che permettono di accodare un elemento, Enqueue, e di far uscire il successivo elemento, Dequeue.

```
Queue<int> coda=new Queue<int>();
coda.Enqueue(1);
coda.Enqueue(2);
coda.Enqueue(3);
```

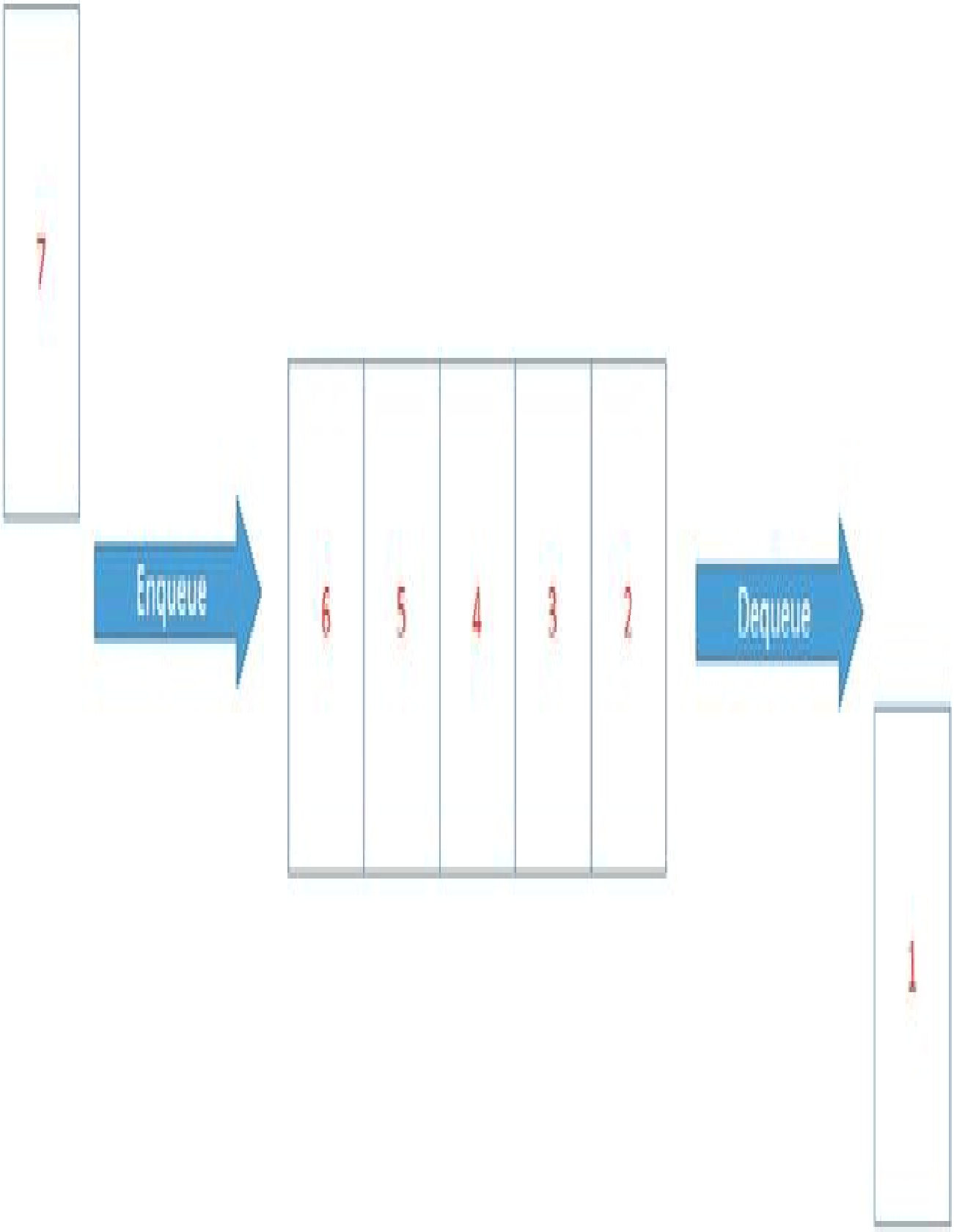

7
Enqueue
6
5
4
3
2
Dequeue
1

Figura 9.5 - Esempio di coda e operazioni Enqueue e Dequeue.

In quanto implementazione di IEnumerable<T>, una coda è enumerabile, senza naturalmente disturbarne il contenuto:

```
foreach(var elemento in coda)
{
Console.WriteLine(elemento);
}
//stampa 1,2,3
```

Se a questo punto rimuovo un elemento:

```
int prossimo=coda.Dequeue();
```

il contenuto della coda sarà ora 2,3.

Il metodo Peek, invece, consente di ottenere l'elemento in testa senza rimuoverlo:

```
int prossimo=coda.Peek();
```

Anche la classe Queue ha una proprietà Count che rappresenta il numero totale di elementi in coda:

```
int inCoda=coda.Count;
```

Da un oggetto Queue può essere ottenuto un array degli elementi in esso contenuti mediante il metodo ToArray:

```
int[] array=coda.ToArray();
```

Un array o un altro oggetto IEnumerable possono essere usati per inizializzare una coda:

```
Queue<int> copia = new Queue<int>(array);
Queue copiaNonGen = new Queue(array);
```

Internamente le code sono implementate mediante un array, che viene automaticamente ridimensionato e modificato quando richiesto dall'inserimento o dalla rimozione di elementi.

Le prestazioni delle operazioni di Enqueue e Dequeue sono naturalmente ottimizzate mantenendo un riferimento al primo e all'ultimo elemento della coda.

La classe Stack

Una *pila* è una struttura dati di tipo *Last-In-First-Out* (*LIFO*) che permette appunto di impilare elementi uno sull'altro, in maniera che l'ultimo a essere inserito sia sempre il primo a poter essere estratto.

L'implementazione è fornita sia in versione generica dalla classe Stack<T>, sia non generica con la classe Stack. In maniera simile alle code del precedente paragrafo, esse forniscono i metodi per impilare un elemento, Push, e per estrarne uno, Pop:

```
Stack<int> stack=new Stack<int>();
stack.Push(1);
stack.Push(2);
stack.Push(3);
int top=stack.Pop(); // è l'ultimo inserito, 3
```

Il metodo Peek, invece, consente di ispezionare la testa della pila senza estrarre l'elemento:

```
int top=stack.Peek();
```

Anche di uno Stack è possibile conoscere il numero di elementi ed enumerarlo:

```
int count=stak.Count;
foreach(int i in stack)
{
Console.WriteLine(i);
}
```

Il metodo ToArray restituisce un vettore degli elementi presenti nello stack.

Da un array, o altro oggetto IEnumerable, è viceversa possibile inizializzare una pila, passandolo al costruttore apposito:

```
int[] array=stack.ToArray();
Stack<int> copia=new Stack<int>(array);
```

L'oggetto BitArray

Un oggetto BitArray consente di mantenere e manipolare una sequenza di bit, rappresentati da valori booleani, true per l'1 (on) e false per lo 0 (off), in maniera più efficiente di un'implementazione manuale tramite array o liste generiche di bool.

La classe fornisce diversi costruttori per inizializzare array di bit di una certa lunghezza e impostando il valore iniziale dei bit:

```
BitArray ba=new BitArray(8, true); // 8 bit true
```

I metodi Set e SetAll permettono di impostare i bit, individualmente o tutti insieme, a un valore bool:

```
ba.SetAll(false);
ba.Set(0, true);
```

Il metodo Get, invece, permette di leggere ogni singolo bit:

```
bool bit=ba.Get(0);
```

La classe possiede un indicizzatore per cui è possibile accedere ai singoli bit mediante indice, in lettura e scrittura:

```
bit=ba[0];
ba[0]=false;
```

Naturalmente, la classe è enumerabile. Per esempio, per stampare il valore di tutti i bit in formato numerico potremmo utilizzare un foreach e un operatore ternario:

```
foreach(var bit in ba)
{
Console.WriteLine(bit?1:0);
}
```

La classe fornisce anche i metodi per eseguire le consuete operazioni logiche (AND, OR, XOR) su due istanze della classe stessa e ottenere un BitArray con i bit impostati secondo l'operazione eseguita:

```
BitArray ba1=new BitArray(8);
BitArray ba2=new BitArray(8, true);
BitArray bAnd=ba1.And(ba2);
```

Insiemi

Un *insieme* contiene elementi senza permettere la presenza di duplicati. Un elemento può appartenere o meno a un dato insieme.

Il .NET Framework all'interno del namespace System.Collections.Generic definisce un'interfaccia generica ISet<T> che espone i metodi per aggiungere e rimuovere elementi da un insieme. Per esempio, il metodo Add verifica se un elemento fa già parte di un insieme, quindi agisce in modo da avere al massimo una copia di ogni elemento.

Altri metodi permettono inoltre di eseguire le classiche operazioni fra insiemi diversi, che combinano un ISet<T> con una qualsiasi raccolta che implementi l'interfaccia IEnumerable<T>. Per esempio, l'unione fra un insieme e un altro insieme o IEnumerable è implementata dal metodo UnionWith. Il metodo ExceptWith rimuove dall'insieme gli elementi contenuti in un'altra collezione, implementando la differenza fra insiemi. Il metodo IntersectWith, invece, modifica l'insieme corrente in maniera che esso contenga solo gli elementi contenuti anche in un'altra collezione. Il metodo SymmetricExceptWith rimuove da un insieme tutti gli elementi che fanno parte di un'altra collezione, ma aggiunge a esso quelli che invece fanno parte di questa collezione ma non dell'insieme.

Poi esistono altri metodi che servono a confrontare due insiemi, e anche questi prendono come parametro una collezione IEnumerable per rappresentare il secondo insieme con cui confrontare quello corrente. Il metodo SetEquals verifica se due insiemi contengono gli stessi elementi. I metodi IsSubsetOf e IsProperSubsetOf determinano se l'insieme corrente è un sottoinsieme di un altro; in particolare, il secondo metodo verifica se si tratta di un sottoinsieme proprio, cioè se il secondo insieme contiene anche altri elementi non appartenenti al primo.

Analogamente, i metodi IsSupersetOf e IsProperSupersetOf effettuano il confronto precedente in direzione opposta, cioè verificando se un insieme è sovrainsieme di un altro. Infine, il

metodo Overlaps verifica se due insiemi hanno almeno un elemento in comune.

Due classi implementano le interfacce ISet<T>, HashSet e SortedSet, ognuna in maniera leggermente differente, come esposto nel paragrafo che segue. In particolare, come si evince dal nome, la seconda mantiene gli elementi ordinati.

Le classi HashSet *e* SortedSet

La classe HashSet<T> implementa l'interfaccia ISet<T> e rappresenta quindi un insieme di elementi di tipo qualunque.

La costruzione di un HashSet può avvenire mediante un costruttore che prevede già il passaggio di una raccolta IEnumerable dalla quale ottenere gli elementi da inserire, oppure mediante l'uso del metodo Add, il quale restituisce un bool a indicare se l'elemento è stato inserito o meno. In particolare, se l'elemento fa già parte dell'insieme, il metodo restituirà false.

Ecco qualche esempio di utilizzo dei metodi di ISet implementati dalla classe HashSet:

```
HashSet<int> hset = new HashSet<int>();
hset.Add(2);
hset.Add(1);
if (hset.Add(1))
{
Console.WriteLine("Elemento aggiunto");
}
else Console.WriteLine("Il numero 1 fa già parte dell'insieme");
hset.Add(4);
hset.Add(3);
```

Una HashSet può anche essere inizializzata indicando un oggetto IEnumerable, come un array, direttamente nel costruttore:

```
HashSet<int> hset = new HashSet<int>(new int[] { 2, 4, 3, 1 });
HashSet<int> hset2 = new HashSet<int>(new int[] { 5, 4, 8, 2, 7, 6 });
```

I metodi che implementano le operazioni fra insiemi agiscono direttamente modificando l'insieme su cui vengono invocati:

```
hset.UnionWith(hset2); //hset = 1,2,3,4,5,6,7,8

int[] array={1,2,3,4};
hset = new HashSet<int>(array);
hset.IntersectWith(hset2); //hset = 2,4

hset = new HashSet<int>(array);
hset.ExceptWith(hset2); //hset = 1,3

hset = new HashSet<int>(array);
if (hset.Overlaps(hset2))
{
Console.WriteLine("i due insiemi condividono degli elementi");
hset.IntersectWith(hset2);
}
```

La classe SortedSet<T> ha un funzionamento praticamente identico alla appena vista HashSet, con l'ulteriore caratteristica che gli elementi aggiunti a essa vengono mantenuti in maniera ordinata. Infatti, se proviamo ad aggiungere casualmente qualche elemento, o a eseguire qualche operazione di unione con altre collezioni, e quindi a stampare gli elementi di un SortedSet, essi appariranno in ordine.

In questo caso, l'ordinamento avviene in maniera crescente, ma la classe SortedSet permette di specificare un oggetto IComparer per implementare un ordinamento personalizzato.

La seguente classe consente di ordinare in maniera decrescente delle stringhe:

```
class ReverseStringComparer: IComparer<string>
{
public int Compare(string x, string y)
{
return -x.CompareTo(y);
}
}
```

Possiamo quindi usarla per l'ordinamento di un SortedSet, passandone un'istanza al costruttore:

```
SortedSet<string> ss = new SortedSet<string>(
new string[]{"a", "d", "c", "b"},
new ReverseStringComparer());
PrintEnumerable(ss);
```

Dizionari

Un *dizionario* è una collezione in cui ogni elemento è formato da una coppia di tipo chiave/valore.

La chiave serve ad accedere a un particolare elemento della collezione, allo stesso modo in cui, per esempio, il termine ricercato in un vocabolario consentirà di leggere la sua definizione completa.

Il framework .NET definisce due interfacce per l'implementazione di classi di tipo dizionario: IDictionary e la generica IDictionary<TKey, TValue>. Entrambe espongono metodi e proprietà che permettono di aggiungere e rimuovere elementi, elencare le chiavi e i valori presenti, verificare se nella collezione è contenuta una chiave o un valore.

Le classi Dictionary *e* Hashtable

La classe generica Dictionary<K,V> è fra le classi di collezioni più utilizzate e permette di definire dizionari con chiavi e valore di un qualunque tipo:

```
Dictionary<int, string> dict=new Dictionary<int,string>();
dict.Add(1, "a");
dict.Add(2, "b");
```

Un'altra sintassi di inizializzazione possibile già a partire da C# 3 è quella che permette di indicare il contenuto del dizionario mediante un inizializzatore di collezione, nel seguente modo:

```
Dictionary<int, string> dict = new Dictionary<int, string>()
{
{ 1, "a" },
{ 2, "b" }
};
```

È quindi necessaria un'altra coppia di parentesi graffe per ogni elemento del dizionario. E anzi, se gli elementi della collezione fossero oggetti più complessi, magari inizializzati anch'essi, le parentesi graffe sarebbero ancora di più:

```
class Persona
{
public string Nome { get; set; }
public int ID { get; set; }
}

Dictionary<int, Persona> persone = new Dictionary<int, Persona>()
{
{ 111, new Persona { Nome="Antonio", ID=123 } },
{ 222, new Persona { Nome="Caterina", ID=456 } },
};
```

A partire da C# 6, è stata introdotta ancora un'altra modalità di inizializzazione, la quale rende più semplici la stesura e la lettura del codice, in quanto utilizza la sintassi degli indicizzatori. Il dizionario precedente può essere per esempio scritto come segue:

```
Dictionary<int, Persona> persone = new Dictionary<int, Persona>()
{
[111] = new Persona {Nome="Antonio", ID=123},
[222] = new Persona {Nome="Caterina", ID=456},
};
```

Quindi, all'interno delle parentesi graffe, basta assegnare gli elementi ai valori chiave indicati fra parentesi quadre.

La chiave degli elementi del dizionario deve essere univoca, quindi non può essere aggiunto un altro elemento con una chiave già esistente:

```
dict.Add(1, "b"); //genera un'eccezione se 1 è già utilizzata
```

L'indicizzatore permette di accedere ai valori del dizionario specificandone la chiave:

```
string val = dict[1];
```

Se la chiave specificata non esiste, viene scatenata un'eccezione KeyNotFound-Exception. In scrittura, invece, se la chiave non esiste, l'elemento viene aggiunto al dizionario, altrimenti viene sovrascritto il valore esistente:

```
dict[1] = "a"; //se la chiave 1 esiste viene sovrascritto il valore
dict[2] = "b"; //se la chiave 2 non esiste viene aggiunto un nuovo elemento
```

Riprendendo la sintassi di inizializzazione di C# 6, mediante indicizzatori, sebbene sia stata introdotta per semplificare la scrittura, essa può portare a bug difficili da scovare. Infatti, l'assegnazione mediante indice di uno stesso elemento non provoca errore, ma come appena visto sovrascrive l'elemento. Il seguente esempio crea quindi un dizionario con un solo elemento:

```
Dictionary<int, string> dict = new Dictionary<int, string>()
{
[1] = "a",
[1] = "b",
};
```

La chiave 1 è infatti usata, per errore, due volte. Quindi, la seconda assegnazione sovrascrive la prima e l'unico elemento presente alla fine sarà [1, "b"]. La situazione non si verificherebbe usando la sintassi alternativa:

```
Dictionary<int, string> dict = new Dictionary<int, string>()
{
{ 1, "a" },
{ 1, "b" }
};
```

In questo caso, verrebbe lanciata un'eccezione, permettendo di trovare subito il problema.

I metodi ContainsKey e ContainsValue consentono di verificare se all'interno del dizionario esiste rispettivamente un elemento con la chiave indicata o un elemento dal valore indicato:

```
if(dict.ContainsKey(1))
{
string value=dict[1];
}
```

È poi possibile tentare di recuperare un valore corrispondente a una chiave, anche se non si è sicuri della sua esistenza, utilizzando il metodo TryGetValue:

```
string val;
if(dict.TryGetValue(1, out val)
{
Console.WriteLine(val);
}
```

Si presti attenzione al fatto che, per verificare l'esistenza di una chiave o recuperare un valore corrispondente a essa, viene utilizzato il metodo Equals del tipo usato per le chiavi.

Quindi si può modificare tale comportamento effettuando l'override di Equals ove possibile, oppure utilizzando un IEqualityComparer, che può essere passato al costruttore del Dictionary. L'interfaccia IEqualityComparer definisce infatti i metodi Equals e GetHashCode, all'interno dei quali implementare i meccanismi di confronto di due oggetti.

Se, per esempio, le chiavi utilizzate sono istanze di una classe complessa, è possibile sia effettuare l'override di tali metodi ereditati da System.Object, sia scrivere una classe di confronto:

```
class MyKey
{
public string KeyProp1 { get; set; }
public string KeyProp2 { get; set; }
]

public class MyKeyComparer: IEqualityComparer
{
public bool Equals(MyKey x, MyKey y)
{
return x.KeyProp1 == y.KeyProp1 &&
x.KeyProp2 == y.KeyProp2;
}

public int GetHashCode(MyKey obj)
{
return obj.KeyProp1.GetHashCode()+obj.KeyProp2.GetHashCode();
}
}

Dictionary<MyKey, string> dict=new Dictionary<MyKey, string(new MyKeyComparer());
```

Per enumerare chiavi e valori presenti nel dizionario è possibile utilizzare diverse strategie.

Utilizzando un ciclo foreach su un oggetto di classe Dictionary<K,V>, verranno enumerati degli elementi di tipo generico KeyValuePair<K,V>, dal quale ricavare poi chiave e valore per mezzo delle proprietà Key e Value:

```
foreach (KeyValuePair<int, string> pair in dict)
{
Console.WriteLine("{0} : {1}", pair.Key, pair.Value);
}
```

La proprietà Keys restituisce invece una collezione delle chiavi presenti nel dizionario, quindi queste possono essere utilizzate per accedere a ogni elemento e ottenerne il valore:

```
foreach (int key in dict.Keys)
{
Console.WriteLine("{0} : {1}", key, dict[key]);
}
```

Inoltre, è possibile enumerare direttamente i valori:

```
foreach (string valore in dict.Values)
{
Console.WriteLine(valore);
}
```

La classe non generica che implementa un dizionario si chiama invece Hashtable, che fornisce le stesse funzionalità appena viste per la classe Dictionary<K,V>, ma con qualche differenza cui

prestare attenzione, sempre che si abbia necessità di ricorrere a tale versione non generica. La classe Hashtable, infatti, implementa l'interfaccia non generica IDictionary che, a differenza della IDictionary<K,V>, non genera un'eccezione, ma restituisce null se si tenta di accedere a un elemento con una chiave non esistente:

```
Hashtable ht=new Hashtable();
object obj=ht[0]; // = null
```

Inoltre, essa espone un metodo Contains che verifica l'esistenza di una chiave (praticamente replica il funzionamento di ContainsKey).

Enumerando un oggetto Hashtable si ottiene un'enumerazione di oggetti Dictionary-Entry, dei quali è possibile leggere le proprietà Key e Value:

```
foreach (DictionaryEntry entry in ht)
{
Console.WriteLine("{0} : {1}", entry.Key, entry.Value);
}
```

Le classi SortedDictionary e SortedList

Se si necessita di una collezione i cui valori siano ordinati secondo una chiave, è possibile utilizzare una delle due classi generiche SortedDictionary<K,V> e SortedList<K,V>. Il nome della seconda classe è un po' fuorviante, in quanto anch'essa implementa l'interfaccia IDictionary<K,V>.

NOTA	Il framework .NET contiene anche una versione non generica della classe SortedList, in cui chiavi e valori sono di tipo object.

Le due classi hanno un modello di programmazione identico e la differenza risiede essenzialmente nell'implementazione interna, che si riflette su prestazioni differenti.

Leggendo la documentazione delle due classi si possono ricavare maggiori dettagli.

La classe SortedDictionary è più efficiente e veloce nell'inserimento di elementi in maniera casuale, mentre la SortedList è più efficiente nella ricerca di elementi al suo interno perché già al momento dell'inserimento gestisce il loro ordinamento.

Infatti le proprietà Keys e Values, fornite dalla classe SortedList, contengono le collezioni delle chiavi e dei valori e, a differenza di quelle della classe SortedDictionary, implementano IList<T>.

Sono quindi accessibili tramite indice numerico, il che dimostra che queste collezioni sono internamente ordinate.

Ecco qualche esempio di utilizzo della classe SortedList (che vale anche per SortedDictionary, a meno dei dettagli implementativi appena esposti):

```
SortedList<int, string> slist = new SortedList<int, string>();
slist.Add(1, "Antonio");
slist.Add(41, "Caterina");
slist.Add(26, "Daniele");
slist.Add(81, "Rosita");
```

Provando a enumerare gli elementi mediante un foreach, potete verificare che quelli appena aggiunti sono già ordinati per chiave. Quest'ultima deve essere univoca, quindi, se si tenta di inserire un secondo elemento con una chiave già esistente, verrà scatenata un'eccezione:

```
try
{
slist.Add(1, "Pippo");
}
catch
{
Console.WriteLine("La chiave esiste già");
}
```

Mediante il metodo Remove, è possibile rimuovere l'elemento identificato da una particolare chiave:

```
slist.Remove(26);
```

È possibile accedere a ogni elemento, in lettura o in scrittura, mediante l'indicizzatore, a cui passare una chiave esistente:

```
slist[1] = "Pippo";
```

Il metodo ContainsKey, in ogni caso, permette di verificare se la chiave specificata è utilizzata:

```
if (slist.ContainsKey(41))
{
string nome = slist[41];
}
```

Invece, il metodo ContainsValue può verificare l'esistenza di un valore, mentre con IndexOfValue si può ottenere l'indice numerico a cui si trova il valore specificato:

```
if (slist.ContainsValue("Pippo"))
{
int index=slist.IndexOfValue("Pippo");
int key=slist.Keys[index];
}
```

Il precedente blocco di codice non funziona invece con la classe SortedDictionary. In particolare, non è possibile accedere alla collezione Keys per indice, come già detto evidenziando le differenze fra le due classi.

Domande di riepilogo

1) Che cosa indica T in una definizione del tipo Lista<T>?

a. Il generico numero degli elementi che faranno parte della lista
b. Il generico tipo degli elementi che faranno parte della lista
c. Il generico nome che può assumere la classe Lista
d. Il generico valore che assumerà ogni elemento della lista

2) Quale affermazione sui generics è falsa?

a. Una classe generica può essere ereditata come una classe qualunque
b. Una classe derivata da una generica è libera di mantenere aperti i parametri di tipo
c. Come parametro di tipo può essere utilizzato solo un tipo non generico
d. Una classe generica derivata da un'altra generica può aggiungere nuovi parametri di tipo

3) L'operatore default applicato a un tipo riferimento restituisce:

a. un'istanza del tipo
b. il valore null
c. eccezione a runtime
d. errore di compilazione

4) Per indicare che il parametro di tipo T di un generic deve disporre di un costruttore pubblico senza parametri, si scrive:

a. where T: public()
b. where T()
c. where T: class()
d. where T: new()

5) La dichiarazione di variabile int? variabile; indica che:

a. variabile può assumere il valore null
b. variabile non può assumere il valore null
c. il tipo di variabile non è conosciuto a compile-time
d. il valore di variabile è assegnato in maniera casuale

6) Quale istruzione è utilizzata per l'implementazione di un iteratore?

a. yield
b. enumerator

c. foreach
d. iterator

7) Per creare oggetti composti da un numero determinato di altri elementi di vario tipo, si utilizza:

a. la classe Array
b. la classe Tuple
c. la classe Vector
d. la classe Generic

8) Uno Stack è una struttura di dati Last-In-First-Out (LIFO). Vero o falso?

9) Quale affermazione è corretta?

a. Un dizionario è una collezione di stringhe
b. Un dizionario è una collezione in cui ogni elemento è formato da una coppia chiave/valore
c. Un dizionario è una collezione di oggetti ad accesso casuale
d. Un dizionario è un insieme non ordinato di oggetti

Delegate ed eventi

Il CLR di .NET usa meccanismi di notifica per consentire a oggetti di classi differenti di comunicare fra di loro e scambiare dei messaggi al verificarsi di particolari eventi. L'implementazione di tale modello si basa sui delegate.

Gli argomenti che saranno oggetto di questo capitolo vengono spesso trattati come avanzati in molti testi sul linguaggio C#.

Delegate ed eventi sono però parte fondamentale delle specifiche di C# e quindi di ogni programma scritto in tale linguaggio, sin dalla versione 1.0, e costituiscono una parte integrante dell'infrastruttura basata su messaggi e notifiche scambiati fra oggetti.

Finora si è visto come implementare delle classi e istanziare degli oggetti, utilizzando i membri che essi mettono a disposizione, per esempio metodi e proprietà, in maniera praticamente sequenziale.

All'interno del metodo Main si creano degli oggetti, su questi si impostano delle proprietà e si invocano dei metodi che eventualmente restituiscono dei valori, che poi verranno utilizzati da altri oggetti per svolgere nuove azioni. In un programma complesso, però, tali oggetti possono svolgere le proprie azioni e nel frattempo restare in ascolto e rispondere a eventi esterni, anche attendendo una risposta o il completamento di un compito da un oggetto differente.

Tali eventi si possono verificare a loro volta per cause differenti, per esempio per l'interazione utente, come il clic su un pulsante o la selezione di un elemento da una griglia, oppure per fattori esterni: si pensi per esempio alla ricezione di un'email oppure allo scadere di un timer. Ogni applicazione può monitorare il verificarsi di tali avvenimenti e riceverne delle notifiche, ed eventualmente rispondere intraprendendo opportune azioni.

I delegate costituiscono la base per gli eventi e consentono il funzionamento dei meccanismi di notifica e della loro gestione utilizzati nell'intero .NET Framework.

In questo capitolo si vedrà come creare e manipolare i delegate, utilizzandoli poi con la parola chiave event di C# per implementare e assegnare dei metodi di gestione degli eventi. In alternativa, le espressioni lambda e i metodi anonimi potranno essere utilizzati per scrivere dei blocchi di codice che i delegate eseguiranno, semplificando l'utilizzo di questi ultimi.

I delegate

I *delegate* sono uno dei tipi riferimento di .NET, ma un oggetto di tale tipo, anziché referenziare per esempio un'istanza di una classe, rappresenta un metodo. In questo modo, è possibile trattare i metodi come se fossero dati, assegnandoli a una variabile, passandoli come argomenti a un altro metodo e così via.

Per definire un delegate bisogna specificare la firma di un metodo (cioè i suoi parametri) e il suo tipo di ritorno. All'interno di un delegate si può a questo punto incapsulare un qualsiasi metodo di qualunque classe che rispetti sia la firma sia il tipo di ritorno dei metodi che esso rappresenta. Una volta creato un delegate, quindi, esso potrà invocare il metodo o i metodi che sono stati incapsulati al suo interno.

La parola *delegate* utilizzata per indicare questo tipo di oggetto calza a pennello: scegliendo un delegato e assegnandogli un compito, si garantisce che esso si comporterà in un modo ben preciso e con determinati oggetti da consegnare o ricevere.

NOTA	Il concetto di delegate è simile a quello dei puntatori a funzione utilizzati in linguaggi come C/C++, ma quello dei delegate è un meccanismo orientato agli oggetti e type-safe: in C si può utilizzare l'indirizzo di una funzione e usare questa, per esempio, come parametro di un'altra funzione, ma non è possibile verificare che tipo di funzione si sta passando. Un delegate, invece, potendo contenere solo l'indirizzo di metodi che rispettano la sua specifica, garantisce che essi potranno essere utilizzati solo in un preciso modo.

Definire un delegate

La specifica di un delegate definisce la firma del metodo che esso può incapsulare. Per tale specifica si utilizza la parola chiave delegate, seguita dal tipo di ritorno, dal nome del delegate e dai parametri:

[modificatore] **delegate** tipo_ritorno NomeDelegate([parametri]);

Tale dichiarazione, quindi, è perfettamente analoga a quella di un metodo, con l'eccezione che non ha un corpo ed è preceduta dalla parola chiave delegate.

Prendiamo questo possibile esempio di definizione di delegate:

```
public delegate int MioDelegate(string str, double d);
```

In questo modo, il compilatore sa che il delegate MioDelegate può incapsulare tutti i metodi che restituiscono un int e che prendono come parametri di ingresso un oggetto string e un double. Ogni delegate quindi è type-safe, cioè sicuro rispetto ai tipi che può trattare, per definizione.

Come visto sopra, la dichiarazione di un delegate può anche essere preceduta da uno dei modificatori di accesso, a seconda del punto in cui essa viene effettuata, in maniera da poter controllare i possibili passaggi del programma in cui potrà essere utilizzato. Un delegate, per esempio, può essere dichiarato direttamente in un namespace, come tipo a sé stante, e quindi come public o internal:

```
namespace Test
{
public delegate void MioDelegate(string p);
public class MiaClasse
{
…
}
}
```

In questo modo si potrà creare e utilizzare un oggetto delegate, come vedremo a breve, come fosse un qualunque tipo del namespace Test, eventualmente anche riferendosi a esso con un'istruzione using o con il nome completo Test.MioDelegate.

La dichiarazione del delegate può inoltre essere innestata all'interno di una classe e quindi, per essere utilizzabile anche all'esterno, dovrà avere un modificatore di accesso public e si dovrà utilizzare il suo nome completo per usarlo appunto all'esterno delle classe:

```
public class MiaClasse
{
public delegate int MioDelegate()
}
```

In questo caso, il nome completo del delegate sarà MiaClasse.MioDelegate.

Dietro le quinte, mediante l'utilizzo della parola chiave delegate, il compilatore C# crea una classe, derivata da System.MulticastDelegate, che a sua volta è figlia della classe System.Delegate.

La classe MulticastDelegate, e quindi le sue figlie, permettono di accedere a una lista di metodi referenziati dal delegate, detta *lista di invocazione* o *invocation list*, oltre che a vari metodi, proprietà e operatori utili a gestire tale lista.

Creare un delegate

Per capire il funzionamento dei delegate, sicuramente un argomento abbastanza ostico per chi lo affronta la prima volta, non c'è niente di meglio che scrivere qualche esempio che mostri come crearli e utilizzarli.

Supponiamo di aver definito un delegate che prenda come parametro un valore int e restituisca una stringa:

```
public delegate string Int2StringDelegate(int i);
```

Questo delegate sarà compatibile con qualunque metodo, statico o di istanza, che abbia come tipo di ritorno string e che accetti un parametro int.

Per istanziare un delegate bisogna quindi indicare il metodo che esso incapsula. Supponiamo di avere quindi una classe con un metodo che rispetti la firma del delegate precedente:

```
public delegate string Int2StringDelegate(int i);

class Converter
{
public string ConvertToString(int i)
{
return i.ToString();
}
}
```

Una prima possibilità per istanziare un delegate è quella che prevede l'uso di new, indicando il nome del metodo come se si trattasse del parametro di un normale costruttore:

```
Int2StringDelegate isdel =new Int2StringDelegate(ConvertToString);
```

Si noti che viene indicato solo il nome del metodo, senza parentesi tonde, visto che non si tratta di una chiamata del metodo ConvertToString. Naturalmente, perché il codice precedente funzioni, il metodo ConvertToString deve essere nello stesso ambito del delegate: quindi, nel caso specifico, dovremmo trovarci all'interno della stessa classe Converter. Altrimenti bisognerebbe prima avere a disposizione un'istanza della classe Converter, quindi utilizzare il nome completo del metodo:

```
static void Main()
{
Converter converter=new Converter();
Int2StringDelegate isdel=new Int2StringDelegate(converter.ConvertToString);
}
```

Anche in questo caso, l'intellisense di Visual Studio viene in aiuto allo sviluppatore. Come potete notare dalla Figura 10.1, esso suggerisce la firma del metodo che accetta il delegate, in questo caso indicando come esso si aspetti un metodo di target del tipo string (int), cioè un metodo che restituisca un tipo string e prenda un parametro int.

```
public delegate string Int2StringDelegate(int i);

class Converter
{
    public string ConvertToString(int i)
    {
        return i.ToString();
    }
}

class Program
{
    static void Main()
    {
        Converter converter = new Converter();
        Int2StringDelegate del = new Int2StringDelegate();
    }
}
```

Int2StringDelegate(string (int) target)

Figura 10.1 - Intellisense di Visual Studio per i delegate.

Un delegate può anche essere costruito con un metodo statico; in questo caso non sarà necessaria alcuna istanza di classe.

Per esempio, se la classe Converter avesse anche un metodo statico StaticConvertToString, potremmo istanziare un delegate utilizzandolo nel seguente modo:

```
Int2StringDelegate isdel=new Int2StringDelegate(Converter.StaticConvertToString);
```

Un delegate può essere poi assegnato a una variabile tipizzata implicitamente:

```
var isdel =new Int2StringDelegate(converter.Convert.ToString);
```

Nella pratica, difficilmente si utilizza l'operatore new, in quanto è possibile costruire un delegate in maniera implicita e cioè assegnando direttamente il nome del metodo a una variabile, come nel seguente caso:

```
Int2StringDelegate isdel = converter.ConvertToString;
```

oppure, nel caso di metodo statico:

```
Int2StringDelegate isdel = Converter.StaticConvertToString;
```

Il compilatore riconosce dalla dichiarazione della variabile il tipo del delegate, quindi dietro le quinte creerà una sua istanza assegnandola alla variabile isdel.

Riferendoci al nome del metodo ConvertToString, senza parentesi e argomenti, stiamo tecnicamente utilizzando un *method group*, cioè l'insieme dei possibili metodi (che può anche essere uno solo) con uno stesso nome ma con parametri differenti (cioè con diversi overload). Sarà il compilatore a scegliere quello giusto in base alla firma del delegate.

NOTA	L'assegnazione o conversione di un method group in delegate, che si traduce nella creazione implicita di quest'ultimo, è stata introdotta in C# 2.0.

Per completezza mostriamo un'ulteriore possibilità di creazione di delegate: per mezzo del metodo statico Create della classe Delegate, di cui sono disponibili diversi overload.

Quello seguente, per esempio, prende come argomento il tipo del delegate da creare, l'istanza della classe che espone il metodo e il nome del metodo da assegnare al delegate:

```
Int2StringDelegate del3= (Int2StringDelegate)Delegate.CreateDelegate(
typeof(Int2StringDelegate),
converter,
"ConvertToString");
```

Fra gli altri overload vi è anche quello che permette di omettere l'istanza della classe, utilizzabile con i metodi statici.

Essendo ogni delegate l'istanza di una classe MulticastDelegate, e indirettamente di Delegate, possiamo utilizzarne i metodi e le proprietà. Fra queste ultime vi segnalo le proprietà Method e Target, mediante le quali è possibile ricavare il metodo assegnato al delegate e il nome del tipo a cui appartiene, nel caso di metodo di istanza:

```
var method=isdel.Method;
Console.WriteLine(method.Name);
var target = isdel.Target;
Console.WriteLine(target.ToString());
```

L'oggetto restituito dalla proprietà Method è un oggetto MethodInfo, il quale consente di ottenere varie informazioni su un metodo; nell'esempio sopra ne viene letto semplicemente il nome.

Da notare che, nel caso di un metodo statico, la proprietà Target sarà null.

Invocare un delegate

Riprendendo l'esempio precedente, a questo punto la variabile isdel di tipo Int2StringDelegate contiene un riferimento al metodo ConvertToString della classe Converter.

La sintassi per invocare il metodo che il delegate contiene è identica all'invocazione del metodo stesso:

```
string str= isdel(123);
```

Tale istruzione è un'abbreviazione della chiamata esplicita al metodo Invoke dell'oggetto delegate:

```
string str=isdel.Invoke(123);
```

La variabile isdel, istanza di un delegate, può essere usata come parametro di un metodo. Grazie a questo, possiamo scrivere del codice che può funzionare in diversi modi a seconda del metodo assegnato a runtime al delegate. Supponiamo per esempio di avere implementato un metodo come il seguente:

```
public void UseDelegate(Int2StringDelegate myDel, params int[] values)
{
foreach(int i in values)
{
myDel(i);
}
}
```

Esso ha come parametri un delegate di tipo Int2StringDelegate e un numero variabile di argomenti interi. Al suo interno il delegate invocherà il metodo a cui esso si riferisce, quindi sarà possibile utilizzare il metodo UseDelegate con un qualsiasi metodo che rispetti la firma del delegate, assegnandolo a runtime:

```
Int2StringDelegate isdel=converter.ConvertToString;
UseDelegate(isdel, 1,2,3,4);
```

Delegate multicast

I delegate risultano particolarmente utili in quanto essi possono mantenere al loro interno più riferimenti a diversi metodi; naturalmente tutti quanti devono rispettare la sua firma. In questo caso si parla di delegate *multicast*, e infatti, come già detto in precedenza, ogni istanza di un delegate è un'istanza della classe MulticastDelegate. Tale possibilità è particolarmente utile, come vedremo parlando di eventi, per scrivere del codice che possa notificare più oggetti invocandone di ognuno un metodo.

Per mostrare tale possibilità supponiamo, per esempio, di avere un semplice delegate del seguente tipo:

```
public delegate void EmptyDelegate();
```

Abbiamo inoltre due metodi statici, con esso compatibili, ognuno dei quali stampa il proprio nome sulla console:

```
public void Metodo1()
{
Console.WriteLine("Metodo 1");
}
public void Metodo2()
{
Console.WriteLine("Metodo 2");
}
```

Per combinare più delegate si utilizzano gli operatori + oppure +=:

```
EmptyDelegate multicast =Metodo1;
multicast +=Metodo2;
```

La prima istruzione crea un'istanza del delegate EmptyDelegate utilizzando il Metodo1.

La seconda riga crea un nuovo delegate e lo aggiunge al primo; tale istruzione è equivalente a:

```
multicast = multicast + MioMetodo2;
```

L'operatore + o += utilizzerà internamente il metodo Combine della classe Delegate.

Il metodo Combine si può anche invocare direttamente, ma utilizzando gli operatori il codice risulta notevolmente più semplice da scrivere e da leggere. Esso può invece tornare utile se si vogliono combinare più di due delegate in una sola istruzione, in quanto uno dei possibili overload prende come parametro un array di Delegate:

```
EmptyDelegate multicast=(EmptyDelegate)Delegate.Combine(
new EmptyDelegate(Metodo1),
new EmptyDelegate(MioMetodo2),
new EmptyDelegate(MioMetodo3));
```

Un Delegate è immutabile: l'operatore + crea internamente un nuovo delegate, con una lista di metodi risultante dalla combinazione dei delegate combinati. Il nuovo delegate viene quindi

assegnato alla variabile multicast.

NOTA	Uno stesso metodo può essere aggiunto più volte alla lista di invocazione di un delegate, quindi, invocando quest'ultimo, il metodo verrà eseguito tante volte quante è presente in lista.

Invocando ora il delegate, i due metodi che esso referenzia verranno eseguiti uno dopo l'altro, nell'ordine di inserimento, quindi la seguente istruzione:

```
multicast(); //invoco il delegate multicast
```

eseguirà i due metodi e stamperà in sequenza le due stringhe:

```
Metodo1
Metodo2
```

La lista dei metodi di un delegate è detta lista di invocazione. È possibile ottenerla utilizzando il metodo GetInvocationList della classe MulticastDelegate, che restituisce un array di Delegate. Per esempio, per ottenere la lista dei delegate dalla precedente istanza multicast e invocarli manualmente in un ciclo foreach possiamo scrivere:

```
Delegate[] list= multicast.GetInvocationList();
foreach(Delegate del in list)
{
Console.WriteLine("invoco {0}", del.Method);
((EmptyDelegate)del).Invoke();
}
```

Per invocare ogni delegate mediante il metodo Invoke è necessario effettuare un cast al tipo corretto del delegate.

Una seconda possibilità, se non si conosce il tipo esatto del delegate e non è possibile eseguire una conversione, è utilizzare il metodo DynamicInvoke fornito dalla classe Delegate, il quale ottiene le informazioni necessarie risolvendo a runtime il tipo del delegate e il metodo da invocare. In questo caso, però, le prestazioni sono inferiori rispetto all'utilizzo di Invoke.

Per rimuovere un delegate dalla lista di invocazione si utilizzano gli operatori - e -=:

```
multicast -= Metodo1;
```

Come nel caso della combinazione di delegate con l'operatore +, anche l'utilizzo di - crea un nuovo delegate: il nuovo delegate sarà la copia di quello originale, ma la sua invocation list non conterrà il metodo rimosso. Se un metodo è presente più volte all'interno della lista di invocazione, verrà rimossa la prima istanza trovata; se esso invece non è presente nella lista, l'operatore non ha alcun effetto.

Si noti che, se la lista di invocazione di un delegate è vuota, il tentativo di invocarlo scatenerà un'eccezione, quindi è sempre bene verificare tale eventualità confrontando l'istanza del delegate con il riferimento null:

```
EmptyDelegate ed=Metodo1;// invocation list contiene 1 metodo
ed -= Metodo1; // list vuota
if(ed!=null)
{
ed();
}
```

Negli esempi visti finora, il delegate EmptyDelegate non accetta alcun parametro e non ha nessun tipo di ritorno. I più attenti si saranno chiesti: ma cosa succede invocando un delegate multicast che restituisce dei valori?

Riprendiamo il delegate Int2StringDelegate e creiamo un delegate multicast combinando il già visto ConvertToString e il seguente, che raddoppia un int e lo restituisce sotto forma di string:

```
public string RaddoppiaNumero(int i)
{
return (i*2).ToString();
}
```

Il delegate multicast conterrà i due metodi nella propria invocation list, nell'ordine di inserimento:

```
Int2StringDelegate multicast= converter.ConvertToString;
multicast += converter.RaddoppiaNumero;
```

Il parametro 10 passato al delegate verrà a sua volta utilizzato come argomento per l'esecuzione dei due metodi, ma il risultato di tipo string sarà restituito solo dall'ultimo metodo invocato, mentre i valori restituiti in precedenza verranno semplicemente ignorati.

```
Console.WriteLine(multicast(10)); //stampa 20
```

Un altro caso particolare è quello in cui i parametri passati a un metodo hanno il modificatore ref, per esempio:

```
delegate void RefDelegate(ref int x);
```

In questo modo, nel caso di delegate multicast, il valore passato come argomento verrà modificato dai vari metodi combinati. Quindi, invocando il metodo successivo, tale valore sarà quello risultante dal metodo precedente, non quello iniziale. Supponiamo, per esempio, di avere i seguenti metodi:

```
public static void Raddoppia(ref int x)
{
x*=2;
}
public static void Triplica(ref int x)
{
```

```
x*=3;
}
```

Combinandoli in un delegate multicast come segue, e invocandolo con un parametro di partenza x, questo sarà prima raddoppiato e poi triplicato:

```
int x=1;
RefDelegate multicast= Raddoppia;
multicast+= Triplica;
multicast(ref x);
Console.WriteLine(x);//stampa 6
```

Infatti, la sequenza di invocazione sarà la seguente:

```
int x=1;
Raddoppia(ref x);
//x = 1* 2
Triplica(ref x);
//x = 2* 3
Console.WriteLine(x);//x vale 6
```

Delegate e interfacce

Probabilmente, chi non ha molta dimestichezza con i delegate noterà delle analogie con il concetto di interfaccia: entrambi definiscono una sorta di contratto, ma, mentre le interfacce definiscono un insieme di membri che un tipo deve implementare, un delegate si occupa di specificare un'unica tipologia di metodo.

L'altra differenza fondamentale è che le interfacce sono specificate a tempo di compilazione e, perché un tipo implementi un'interfaccia, il tipo stesso deve definire tutti i membri che essa indica.

Un delegate, invece, viene creato a runtime e può essere utilizzato per eseguire metodi diversi a seconda delle necessità, fornendo quindi un canale anche fra oggetti per i quali in fase di specifica non era stata probabilmente prevista alcuna comunicazione.

I delegate generici

Nel capitolo precedente abbiamo sviscerato il concetto di tipo generico, introducendo rapidamente i *delegate generici*. Ora che abbiamo approfondito anche la nozione di delegate, la soluzione del puzzle dovrebbe essere più completa e fornire quindi un quadro più chiaro. Utilizzando i generics è possibile adoperare parametri di tipo nella dichiarazione di un delegate:

```
public delegate TDest ConvertOriginToDest<TDest, UOrig>(UOrig val);
```

In questo modo, esso può essere impiegato in maniera generale con metodi che usano parametri di tipo differente, evitando di dover invece definire delegate che probabilmente saranno utili solo in casi particolarmente specifici.

Il delegate generico precedente è, per esempio, compatibile con entrambi i seguenti metodi:

```
public string IntToString(int i)
{
return i.ToString();
}

public int StringToInt(string s)
{
return int.Parse(s);
}
```

Quindi potremmo istanziare e utilizzare il delegate generico nelle due diverse casistiche, specificando i parametri di tipo necessari in questo modo:

```
ConvertOriginToDest<int, string> isconvert= IntToString;
string s= isconvert (123);
```

Oppure, cambiando i parametri di tipo:

```
ConvertOriginToDest<string, int> siconvert = StringToInt;
int i=siconvert ("123");
```

Anche nei delegate generici si possono inserire dei vincoli ai parametri di tipo. Per esempio, volendo fare in modo che il delegate `ConvertOriginToDest` permetta di incapsulare solo metodi che lavorano su tipi valore, si potrebbe modificare la sua definizione nel seguente modo:

```
public delegate TDest ConvertOriginToDest<TDest, UOrig>(UOrig val) where TDest:struct, UOrig:struct
```

Così, nessuno dei due metodi `IntToString` e `StringToInt` sarebbe compatibile con esso, perché `string` è un tipo riferimento. Lo sarebbe invece un metodo che convertisse un tipo valore in un altro:

```
//il metodo più inutile del mondo
public int IntToDouble(double d)
{
return (int)d;
```

}

L'utilizzo di delegate generici è particolarmente utile nella definizione di eventi, argomento che affronteremo più avanti in questo capitolo.

Il .NET Framework, inoltre, utilizza estensivamente i delegate generici, definendone parecchi di utilizzo generale, come Func<T> e Action<T> che vedremo nel prossimo paragrafo.

I delegate generici **Func** e **Action**

Il framework .NET, nella versione 3.5, ha introdotto due delegate generici che spesso evitano di dover definire delegate personalizzati perché sono utilizzabili in molti casi. Questi due tipi di delegate, definiti nel namespace System, sono denominati Func<> e Action<> (non sono stati indicati i parametri di tipo perché sono disponibili diverse versioni, a seconda del numero di parametri che essi permettono di utilizzare).

NOTA	I delegate generici Func e Action risulteranno di notevole utilità all'interno di LINQ, come si vedrà nel Capitolo 11.

Il delegate Func<> incapsula i metodi che accettano da zero a 16 parametri di ingresso di vario tipo e restituiscono un risultato di un altro tipo altrettanto generico.

```
delegate TResult Func<out TResult>();
delegate TResult Func<in T, out TResult>(T arg);
delegate TResult Func<in T1, in T2, out TResult>(T arg);
// fino alla versione con 16 parametri T1, T2,…T16
```

Come già detto nel Capitolo 9, e come rivedremo nel prossimo paragrafo, i modificatori in e out indicano la varianza dei delegate. Il nome Func deriva dal concetto di funzione che in base a degli ingressi restituisce un valore di uscita.

Supponiamo di avere definito un nostro delegate del tipo seguente:

```
public delegate int CalcolaFunzioneDelegate(int i);
```

e il metodo per il calcolo del fattoriale, compatibile con lo stesso delegate:

```
public int Fattoriale(int n)
{
if(n==1)
return 1;
return n*Fattoriale(n-1);
}
```

Il delegate generico Func<T,T> evita di dover creare questo nuovo delegate, in quanto potremo usarlo con i parametri corretti Func<int,int>:

```
Func<int,int> func=Fattoriale;
```

Il delegate Func è così utilizzabile, per esempio, come parametro di un generico metodo di calcolo:

```
public T Evaluate(Func func<T,T>, T val)
{
return func(val);
```

```
}
```

Il precedente metodo invoca al suo interno quello incapsulato in Func, quindi è ora utilizzabile per avviare la valutazione della funzione con i parametri indicati. Per esempio:

```
int result=Evaluate(Fattoriale, 5);
```

In questo modo, il metodo Evaluate potrà essere utilizzato per valutare ed eseguire una qualunque funzione, matematica o non, che prenda un valore di un dato tipo e restituisca un risultato dello stesso o di un altro tipo.

Anziché definire un nostro metodo compatibile con il delegate Func, vediamo come utilizzare il metodo statico fornito dalla classe Math del .NET Framework, che calcola la radice di un numero double, restituendo il risultato sotto forma di altro double:

```
Func<double, double> funcRadice=Math.Sqrt;
Double radice=Evaluate<double>(funcRadice, 144);
```

Un metodo che rispetti la firma del delegate può anche essere utilizzato senza la necessità di creare esplicitamente un delegate (ciò vale in generale, non solo per Func<>). Per esempio, se volessimo ora valutare il logaritmo in base 10 di 100, potremmo invocare il metodo EvaluateFunction come segue:

```
double log= Evaluate(Math.Log10, 100.0d); //log = 2 (perché 10^2 fa 100)
```

Si noti come, nel caso precedente, non sia stato necessario esplicitare il parametro di tipo double: il compilatore è in grado di ricavare dai parametri il tipo corretto da utilizzare.

Il delegate Action<>, invece, incapsula i metodi che accettano dei parametri di ingresso ed eseguono un'azione senza restituire alcun valore di ritorno.

Supponiamo di voler implementare un metodo che può generare un'eccezione, e di voler eseguire delle azioni di recupero di quest'ultima in maniera configurabile e determinabile dal metodo chiamante:

```
public void OperazionePericolosa(Action<Exception> errorAction)
{
//esegue le operazioni e in caso di errore esegue l'azione indicata
try
{
throw new Exception("Eccezione provocata");
}
catch(Exception ex)
{
errorAction(ex);
}
}
```

In tal modo è possibile utilizzare il metodo OperazionePericolosa e indicare l'azione da eseguire in caso di eccezione, che potrebbe essere semplicemente la stampa del messaggio di errore:

```
public void PrintError(Exception ex)
{
Console.WriteLine(ex.Message);
}
```

Oppure il rethrow dell'eccezione:

```
public void Rethrow(Exception ex)
{
throw ex;
}
```

Adesso si può invocare il metodo OperazionePericolosa, indicando una delle possibili azioni da eseguire:

```
OperazionePericolosa(PrintError); //stampa l'error per mezzo del metodo PrintError
OperazionePericolosa(Rethrow); //rilancia l'eccezione
```

Nel capitolo precedente, parlando di collezioni di tipo List<T>, abbiamo già avuto modo di utilizzare il delegate Action<T> per stampare tutti gli elementi di una lista con una singola istruzione. Il metodo ForEach di List<T>, infatti, permette di eseguire una determinata azione su ogni elemento della lista; tale azione è espressa da un delegate Action<T>. Il metodo Console.WriteLine, per esempio, è compatibile con questo delegate in quanto restituisce void e prende dei parametri di ingresso da stampare. Quindi, per stampare una List<T>, possiamo scrivere semplicemente:

```
List<string> lista=new List<string>( new string[] {"questa", "è", "una", "collezione", "di", "stringhe"});
lista.ForEach(Console.WriteLine);
```

Covarianza e controvarianza di Func e Action

Come già anticipato, affrontando l'argomento nel Capitolo 9, grazie al supporto di covarianza e controvarianza introdotto con C# 4.0, due delegati possono essere convertiti uno nell'altro in maniera molto naturale.

Per indicare che un parametro di tipo utilizzato come tipo di ritorno è covariante, si impiega il modificatore out. Applicando invece il modificatore in ai parametri di tipo degli argomenti di ingresso, essi saranno controvarianti. I delegate generici Func e Action supportano quindi covarianza e controvarianza dei parametri di tipo. Fra le varie versioni del delegate Func, per esempio, è disponibile la seguente:

```
delegate T Func<out T>();
```

La presenza del modificatore out indica che il tipo T è covariante. Consente cioè l'assegnazione di un delegate che usa un dato parametro di tipo a un delegate con un parametro di tipo che a sua volta è padre del precedente.

Scriviamo un esempio per chiarire meglio il concetto. Dato il metodo:

```
public string Hello()
{
return "hello";
}
```

possiamo effettuare le seguenti assegnazioni:

```
Func<string> fString= Hello;
Func<object> fObject= fString;
```

Ciò è possibile in quanto il tipo `Func<string>` è compatibile con `Func<object>`, perché `string` è una classe derivata da `object`.

Analogamente, possiamo verificare la controvarianza dei delegate `Action<>`; prendiamone per esempio la versione più semplice:

```
delegate Action<in T>(T arg);
```

Il parametro di input `T` è controvariante poiché marcato con il modificatore `in`, quindi è possibile assegnare un delegate `Action` a un secondo delegate `Action` con parametro di una classe derivata.

Dato il metodo:

```
public void UseObject(object obj)
{
Console.WriteLine(obj);
}
```

possiamo quindi effettuare l'assegnazione che segue:

```
Action<object> actionobj= UseObject;
Action<string> actionstr=actionobj;
```

Il delegate **Predicate<T>**

Fra i delegate definiti dalla libreria .NET e più utilizzati nella pratica, specialmente con le collezioni generiche e con LINQ, vi è il delegate Predicate<T> così definito:

```
delegate bool Predicate<T>(in T obj);
```

Esso rappresenta i metodi che valutano dei criteri sull'oggetto passato come argomento e restituiscono un valore booleano, quindi è molto indicato per implementare dei filtri su collezioni.

Infatti, svariati metodi delle collezioni standard, come List<T>, utilizzano un Predicate<T> per indicare le condizioni da usare nei metodi di ricerca Find, FindAll e così via.

Il parametro di questo tipo è controvariante. Ciò significa che è possibile utilizzare il tipo specificato o qualsiasi tipo meno derivato. Supponiamo, per esempio, di avere una List<int> e di volerne ricavare solo quelli pari. Un metodo che esamina un intero e restituisce true se tale intero è pari potrebbe essere il seguente:

```
bool IsPari(int i)
{
return i%2==0;
}
```

Quindi, per filtrare la lista possiamo utilizzare il metodo IsPari, così da istanziare un delegate di tipo Predicate<int> da passare al metodo FindAll di List<int>:

```
List<int> lista=new List<int>(new int[] {1,2,3,4,5,6,7,8});
List<int> listaPari= lista.FindAll(IsPari);
```

Ogni elemento della lista originale verrà passato come argomento al metodo rappresentato dal delegate, e quelli che faranno restituire true a tale metodo saranno restituiti infine dal metodo FindAll.

Metodi anonimi

I *metodi anonimi* sono una funzionalità introdotta da C# 2.0, il cui utilizzo però è stato comunque quasi soppiantato dall'introduzione delle espressioni lambda in C# 3.0 (vedere il prossimo paragrafo).

Un metodo anonimo è un blocco di codice utilizzato per istanziare un delegate senza la necessità di scrivere un metodo separato. Spesso capita, infatti, che tale metodo venga implementato esclusivamente a tale scopo, e quindi non utilizzato al di fuori della creazione del delegate. In parole povere, quindi, i metodi anonimi consentono di scrivere un metodo in linea con la creazione del delegate utilizzando una particolare sintassi:

```
delegate([ListaParametri])
{
//blocco codice
}
```

Supponiamo, per esempio, di avere ancora a che fare con il delegate definito qualche paragrafo fa:

```
delegate string ConvertToString(int);
```

Per istanziarlo, senza creare un metodo come visto fino a poco fa, possiamo scrivere:

```
ConvertToString convDel=delegate(int i)
{
return i.ToString();
}
```

I metodi anonimi, naturalmente, possono essere utilizzati con delegate di qualunque tipo, generici e non, con tipi di ritorno o meno.

Riprendendo l'esempio del paragrafo precedente sul delegate Predicate<T>, si può usare un metodo anonimo da passare al metodo FindAll di una List<T>:

```
List<int lista=new List<int>(new int[] {1,2,3,4,5,6,7,8});
List<int> listaPari= lista.FindAll(delegate(int i){ return i%2==0 } );
```

Espressioni lambda

Il termine *lambda* deriva da un sistema matematico chiamato *lambda calcolo*, sviluppato per analizzare e calcolare funzioni, quindi utilizzato in molti linguaggi funzionali.

Introdotte in C# 3.0, le *espressioni lambda* consentono di esprimere funzioni in maniera molto concisa e sono utilizzabili per istanziare un delegate, o al posto dei metodi anonimi.

Un'espressione lambda ha la seguente forma:

```
(parametri) => espressione
```

È cioè una lista di parametri seguita dall'operatore lambda => e da un'espressione che utilizza i parametri stessi.

Se i parametri sono più di uno, saranno separati con la virgola; invece, se c'è un solo parametro e il suo tipo è determinabile mediante inferenza, può essere eliminato sia il nome del tipo sia la parentesi.

Il seguente metodo, ripreso dal paragrafo precedente, verifica se un numero è pari:

```
public bool IsPari(int x)
{
return x%2==0;
}
```

Come espressione lambda possiamo scriverlo in una delle forme seguenti:

```
(int x) => { return x%2==0; }
```

E poiché abbiamo un solo parametro, se il suo tipo è determinabile dal contesto, esso si può omettere insieme alle parentesi:

```
x => { return x%2==0; }
```

Essendo l'espressione costituita da una sola istruzione, il blocco non necessita nemmeno delle parentesi graffe:

```
x => return x%2==0;
```

Infine, quando l'espressione contiene una singola istruzione `return`, anche questa può essere omessa:

```
x => x%2==0;
```

Nel caso in cui il blocco dell'espressione lambda sia costituito da più istruzioni, sarà necessario invece utilizzare le parentesi graffe per delimitarlo.

NOTA	Per distinguere espressioni lambda formate da una singola espressione da quelle che invece sono costituite da un blocco più lungo di istruzioni (in genere comunque non più di tre) racchiuso da parentesi graffe, si usano i termini *espressione lambda* per le prime e *dichiarazione lambda* per le seconde.

La semplice espressione precedente, che restituisce un bool a partire da un parametro int, può quindi essere utilizzata in qualunque punto in cui viene richiesto un delegate con gli stessi parametri e lo stesso tipo di ritorno. Per esempio, l'espressione lambda precedente è compatibile con il delegate Predicate<int>:

```
delegate bool Predicate<int>(int i);
```

Quindi si può anche istanziare quest'ultimo con l'espressione:

```
Predicate<int> pred = x => x%2==0;
```

Naturalmente nulla vieta (anzi è spesso consigliabile) di utilizzare le parentesi per rendere più leggibile il codice. L'assegnazione precedente è senz'altro più chiara se scritta come segue:

```
Predicate<int> pred = ( (x) => x%2==0 );
```

A questo punto, il delegate può essere invocato normalmente, passandogli un parametro di tipo int:

```
bool b= pred(25431); //false
```

L'invocazione del delegate non farà altro che valutare l'espressione e restituirne il risultato. Il vantaggio è quindi quello di poter usare un'espressione molto concisa al posto di delegate e metodi anonimi.

Lo stesso esempio che utilizza un metodo anonimo per ricercare i numeri pari in una lista di interi può essere semplicemente scritto usando l'espressione lambda precedente:

```
List<int> lista=new List<int>(new int[] {1,2,3,4,5,6,7,8});
List<int> listaPari= lista.FindAll( x =>return x%2==0 );
```

Le espressioni lambda, oltre che con i Predicate<T>, sono spesso utilizzate in accoppiata con i delegate Func<> e Action<>. Per esempio, una funzione che calcola la media fra due interi e restituisce il risultato come double può essere scritta come:

```
Func<int, int, double> mediaFunc= (x,y) => (x+y)/2.0;
double average=mediaFunc(3,7);
```

Mentre una Action che prende due stringhe, le concatena e le stampa in maiuscolo, può essere così implementata e utilizzata:

```
Action<string,string> upperAction = (s1,s2) => {
string s=s1+" "+s2;
Console.WriteLine(s.ToUpper());
```

```
};
upperAction("Hello", "World");
```

Variabili catturate

All'interno di un'espressione lambda è possibile utilizzare variabili definite all'esterno del blocco dell'espressione stessa. Per esempio, dato il seguente codice:

```
int factor=5;
Func<int,int> multiply= x => x*factor;
```

l'espressione utilizza la variabile esterna factor, che verrà detta *variabile catturata*. Un'espressione lambda che cattura variabilai è invece detta *closure*.

Le variabili catturate non sono valutate direttamente al momento della cattura, ma solo nel momento in cui l'espressione stessa verrà invocata.

```
int factor=5;
Func<int,int> multiply= x => x*factor;
factor=10;
int result=multiply(5); //restituisce 50 perché usa il valore factor = 10;
```

Una variabile catturata dall'espressione mantiene la sua visibilità anche al di fuori del suo scope originario. Cioè se, per esempio, per creare una closure si è utilizzata una variabile passata a un metodo, e poi la closure è stata usata per creare un delegate Func, tale delegate può essere invocato anche se la variabile catturata originariamente non si trova più nello scope attuale:

```
public Func<int,int> CreaFunc()
{
int factor=5;
Func<int,int> multiply= x => x*factor;
return multiply;
}

public static void Main()
{
Func<int,int> func=CreaFunc();
int result = Func(5);
}
```

NOTA	Il funzionamento delle closure è permesso dall'implementazione interna creata dal compilatore, che trasforma l'espressione in una classe e le variabili catturate in campi privati di tale classe.

Un'espressione lambda può anche modificare una variabile catturata:

```
int x=0;
Func<int> modificaVariabile = () => x = 10;
modificaVariabile();
Console.WriteLine(x); //x=10;
```

Alberi di espressioni

Un'espressione lambda può essere interpretata in maniera da creare *alberi di espressioni*.

Un albero di espressioni fornisce l'astrazione necessaria a rappresentare del codice sotto forma di un albero di oggetti, in cui ogni nodo è un'espressione. Tali concetti, abbastanza avanzati, sono stati introdotti in .NET 3.5 e sono cruciali per il funzionamento di LINQ.

Il namespace System.Linq.Expressions contiene i tipi necessari a trattare con alberi di espressioni, in particolare una classe Expression<T> che è utilizzabile per incapsulare un'espressione lambda. In questo modo si può ottenere una rappresentazione astratta dell'espressione lambda che non può essere eseguita direttamente, ma che potrà essere analizzata in modo programmatico e svolgere azioni in risposta all'espressione lambda.

Il codice rappresentato da un albero di espressioni può essere compilato ed eseguito, ed essendo astratto può essere utilizzato in diversi contesti. Per esempio, rimanendo in un programma C#, esso verrà compilato in codice IL, mentre avendo a che fare con un database relazionale potrà essere convertito in una query SQL.

La seguente istruzione assegna un'espressione lambda a un oggetto Expression<T>, creando un albero di espressioni:

```
Expression<Func<int, bool>> exprPari = num => num % 2==0;
```

L'albero di espressioni exprPari può essere compilato per ottenere un delegate invocando il metodo Compile:

```
Func<int, bool> isPari = exprPari.Compile();
```

Il delegate così ottenuto può essere ora invocato:

```
Console.WriteLine(isPari(4));
```

La seguente, invece, è una sintassi semplificata per ottenere lo stesso risultato senza istanziare un oggetto Func:

```
Console.WriteLine(exprPari.Compile()(5));
```

Per il momento non entriamo nel dettaglio, aspettando il prossimo capitolo su LINQ, ma potete pensare che l'albero di espressioni creato dall'espressione lambda ha scomposto quest'ultima creando dei nodi di tipo adeguato per ogni componente: uno per il parametro num, uno per la costante 2, uno per l'operatore binario % e così via.

Eventi

Ora che abbiamo appreso il concetto di delegate, è possibile affrontare l'argomento che ne costituisce una delle principali applicazioni pratiche, vale a dire gli *eventi*.

Gli eventi consentono a un oggetto di avvisare altri oggetti che si è verificato qualcosa che potrebbe interessarli. Gli eventi sono uno dei principali meccanismi utilizzati all'interno di applicazioni con interfaccia grafica. Per esempio, supponendo di avere a che fare con un'applicazione Windows Forms, l'azione di clic su un pulsante genera un evento di cui l'applicazione stessa sarà notificata, e quindi potrà reagire intraprendendo apposite azioni, come aprire un'altra finestra.

L'oggetto che genera l'evento è detto *publisher* (oppure *produttore*), mentre quelli che restano in ascolto dell'evento specifico, ed eventualmente lo gestiscono, sono detti *subscriber* (oppure *consumatore*). Questo pattern di programmazione è quindi spesso chiamato publish-subscribe, oppure *produttore-consumatore*.

Il produttore di un evento non sa (e non deve sapere) quali oggetti saranno i consumatori dell'evento stesso. È quindi necessario un meccanismo che funga da intermediario fra le due parti, una sorta di ambasciatore, ed esso, come forse avrete già immaginato, è il meccanismo dei delegate. Essi, infatti, forniscono l'infrastruttura necessaria a invocare dei metodi di callback dall'interno di un altro metodo.

.NET, per mezzo della tipologia di membro di una classe detta *evento*, e C#, per mezzo della parola chiave event, forniscono il resto degli strumenti che permettono l'implementazione del pattern di programmazione a eventi.

Pubblicare eventi

La parola chiave event di C# consente di definire un particolare tipo di membro di una classe, detto *evento*.

Quindi, innanzitutto, è necessario dichiarare una classe che conterrà o genererà le informazioni relative all'evento stesso. Riprendendo l'utilizzo della classe `Car` che rappresenta un'automobile, immaginiamo di voler implementare un meccanismo che effettui il monitoraggio dei giri del motore, avvisando l'utente quando il valore raggiunge una certa soglia, cioè quando il motore va fuori giri, e viceversa che avverta l'utente quando il motore si è spento perché il regime di giri è sceso troppo. Ogni evento dovrà dunque poter mantenere delle informazioni utilizzabili dal suo consumatore per capire cosa è successo ed, eventualmente, per sapere come reagire. Il tipo dell'evento può essere un qualunque delegate,

in questo modo ogni consumatore potrà utilizzare un metodo compatibile con esso per ricevere una notifica dell'evento sottoscritto.

In generale, dunque, la sintassi per definire un membro evento è la seguente:

[modificatore] **event** TipoDelegate NomeEvento;

Come detto, il delegate può essere uno qualsiasi, definito eventualmente all'interno della classe stessa (in genere, in casi come questo, il delegate viene definito subito prima della dichiarazione dell'evento), oppure fornito dal .NET Framework, come l'Action<> visto qualche paragrafo fa, oppure, come vedrete spesso nel seguito o nel codice di terze parti, il delegate EventHandler (generico o meno).

Un evento, in quanto membro di una classe, può avere un modificatore di accesso qualunque. Inoltre, esso può anche essere dichiarato come abstract, sealed e static.

Riprendiamo ora la classe Car per mettere in pratica quanto detto, aggiungendo due membri di tipo event, uno per la gestione dell'evento FuoriGiri, il secondo per l'evento MotoreSpento. Come tipo di tali eventi, utilizziamo il delegate EventHandler, che rappresenta un metodo con la seguente firma:

```
public delegate void EventHandler(
Object sender,
EventArgs e
)
```

Il framework .NET utilizza spesso questa convenzione suggerita da Microsoft per i delegate gestori degli eventi: il primo parametro sender rappresenta l'oggetto che ha scatenato l'evento, mentre il secondo parametro contiene dati relativi all'evento ed è in genere un oggetto di una classe derivata da EventArgs, che è quella usata per gli eventi che non forniscono altri dati.

```
class Car
{
public event EventHandler MotoreFuoriGiri;
public event EventHandler MotoreSpento;

private int numeroGiri;
public bool EngineOn {get;set;}

public void Start()
{
numeroGiri=800;
EngineOn=true;
}

public void Accelerate()
{
if(EngineOn)
numeroGiri+=100;
}
```

```
public void Decelerate()
{
if(EngineOn)
numeroGiri-=100;
}
}
```

A questo punto bisogna fare in modo che, quando il numeroGiri superi una certa soglia, o scenda sotto una minima, la classe Car crei l'evento relativo e avvisi eventuali altri oggetti interessati all'evento. Modifichiamo quindi il metodo Decelerate. Al suo interno verrà invocato il delegate relativo al motore, se esso è diverso da null (cioè se la sua invocation list contiene qualche metodo).

Il fatto che il delegate sia diverso da null indica che qualche oggetto ha sottoscritto l'evento e intende esserne notificato.

```
public void Decelerate()
{
if(!EngineOn)
return;
numeroGiri-=100;
if(numeroGiri<=300)
{
numeroGiri=0;
EngineOn=false;
OnMotoreSpento();
}
}

private void OnMotoreSpento()
{
if(MotoreSpento!=null)
{
MotoreSpento(this, EventArgs.Empty);
}
}
```

Il metodo Decelerate verifica se il numeroGiri è sceso sotto la soglia minima; in caso affermativo, invoca un metodo OnMotoreSpento (anche questa è una convenzione, la forma OnNomeEvento si utilizza per denominare un metodo che si occupa di notificare gli oggetti in ascolto). Poiché il delegate EventHandler non prevede di inviare dati aggiuntivi sull'evento e quindi, come detto, usa EventArgs come tipo del secondo parametro, in questo caso viene utilizzato il valore EventArgs.Empty per rappresentare la mancanza di dati.

Gestire eventi

Passiamo ora dal lato del sottoscrittore, vedendo come un oggetto può dirsi interessato agli eventi di un'altra classe e, una volta notificato di tali avvenimenti, intraprendere eventuali azioni.

Incontrando la parola chiave event, il compilatore C# genererà tutta l'infrastruttura necessaria a permettere la sottoscrizione dell'evento ai suoi consumatori e a cancellare tale sottoscrizione quando non sarà più necessario gestirlo.

Un oggetto sottoscrittore, quindi, potrà registrarsi in maniera da essere notificato del verificarsi di un determinato evento e indicare quale metodo (che deve essere rispettato dal delegate tipo dell'evento stesso) si occuperà della sua gestione.

La sintassi generale per sottoscrivere un evento e indicare il metodo gestore è la seguente:

```
publisher.NomeEvento += new NomeDelegate(NomeGestore);
```

In tal modo, il metodo chiamato NomeGestore verrà registrato come gestore dell'evento NomeEvento dell'oggetto publisher.

Una sintassi alternativa e abbreviata, oltre che preferibile, è quella già vista nella creazione di un delegate, con la quale è possibile evitare di istanziare quest'ultimo, indicando direttamente il nome del metodo gestore:

```
publisher.NomeEvento += NomeGestore;
```

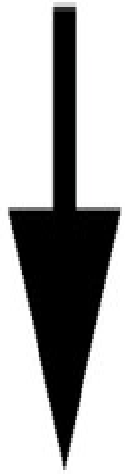
Riprendendo l'esempio, è quindi necessario partire da una classe che utilizza istanze della classe Car. Chiamiamola CarMonitor:

```
class CarMonitor
{
public CarMonitor(Car car)
{
car.MotoreSpento+= GestisciMotoreSpento;
}

public void GestisciMotoreSpento(object sender, EventArgs args)
{
//gestione evento
Car c=sender as Car;
Console.WriteLine("Il motore si è spento");
}
}
```

```
public class CarMonitor
{
    public CarMonitor(Car car)
    {
        car.MotoreSpento+=
    }
}
```

car_MotoreSpento; (Press TAB to insert)



```
public class CarMonitor
{
    public CarMonitor(Car car)
    {
        car.MotoreSpento += car_MotoreSpento;
    }

    void car_MotoreSpento(object sender, EventArgs e)
    {
        throw new NotImplementedException();
    }
}
```

Figura 10.2 - Intellisense e completamento automatico dei gestori di evento.

All'interno del costruttore, al quale viene passata un'istanza della classe Car per mezzo dell'operatore += come visto prima, si registra quale gestore dell'evento MotoreSpento il metodo GestisciMotoreSpento.

Visual Studio aiuta a scrivere i gestori di un qualsiasi evento mediante una funzione di autocompletamento. Nel caso dell'esempio di cui sopra, basterà iniziare a scrivere car.MotoreSpento += e l'editor di codice visualizzerà un tooltip mostrato in Figura 10.2, che suggerirà di premere TAB per inserire il gestore.Istanziamo ora una classe Car, mettiamo in moto invocando il metodo Start e creiamo un'istanza della classe CarMonitor per gestire gli eventi dell'oggetto Car appena costruito:

```
static void Main()
{
Car car=new Car();
car.Start();
CarMonitor monitor=new CarMonitor(car);
while(car.EngineOn)
car.Decelerate();
}
```

Provando a eseguire il programma così implementato, l'istanza monitor attenderà il verificarsi dell'evento MotoreSpento, che avverrà dopo un po' di cicli dell'ultima istruzione while.

Facciamo notare che un gestore dell'evento non deve necessariamente essere un metodo dell'oggetto subscriber dell'evento. In generale, un evento può essere gestito da:

- un metodo di istanza;
- un metodo statico;
- un metodo anonimo;
- un'espressione lambda.

Dati dell'evento

Supponiamo ora di voler passare al metodo di gestione di un evento anche delle informazioni relative all'evento stesso. Per esempio, nel caso dell'evento FuoriGiri, potremmo passare al metodo gestore anche il valore raggiunto, incapsulando in qualche modo la variabile numeroGiri in un oggetto da passare al metodo. In effetti, è proprio quello che si fa implementando una classe derivata da EventArgs e utilizzandola come secondo parametro di un gestore dell'evento.

La nostra classe potrebbe, per esempio, essere la seguente:

```
public class GiriMotoreEventArgs: EventArgs
{
```

```
public int NumeroGiriRaggiunto{get;set;}
}
```

Ora che abbiamo una classe che conterrà il numero di giri raggiunto in caso di evento FuoriGiri, è necessario anche un appropriato delegate che accetti questa classe come secondo parametro.

NOTA	È possibile continuare a usare il delegate EventHandler, perché una classe derivata da EventArgs sarebbe accettata come secondo parametro, ma così facendo all'interno del metodo gestore dovremmo essere a conoscenza del tipo esatto per eseguire una eventuale conversione.

La prima possibilità è utilizzare il delegate generico EventHandler<T>, usando come parametro di tipo la classe GiriMotoreEventArgs appena implementata. In tal caso, la dichiarazione dell'evento all'interno della classe publisher Car andrebbe scritta nel seguente modo:

```
public event EventHandler<GiriMotoreEventArgs> MotoreFuoriGiri;
```

Mentre il metodo Accelerate, che scatena l'evento in caso di superamento della soglia massima di giri, potrebbe essere modificato nel seguente modo:

```
public void Accelerate()
{
if (EngineOn)
{
numeroGiri += 100;
if (numeroGiri > 8000)
{
OnFuoriGiri();
}
}
}
```

Il metodo OnFuoriGiri è quello che notificherà i sottoscrittori dell'evento MotoreFuoriGiri, utilizzando il delegate generico e passando come parametro un'istanza di GiriMotoreEventArgs:

```
private void OnFuoriGiri()
{
if (MotoreFuoriGiri != null)
{
MotoreFuoriGiri(this, new GiriMotoreEventArgs() { NumeroGiriRaggiunto = this.numeroGiri });
}
}
```

Dal punto di vista del sottoscrittore dell'evento, bisognerà implementare il metodo gestore dell'evento stesso, rispettando la firma del delegate EventHandler<GiriMotoreEventArgs>:

```
//sottoscrizione evento
car.MotoreFuoriGiri += car_MotoreFuoriGiri;
```

All'interno del gestore è ora possibile utilizzare il secondo parametro per ricavare i dati sull'evento:

```
//metodo gestore
void car_MotoreFuoriGiri(object sender, GiriMotoreEventArgs e)
{
Console.WriteLine("Fuori giri: " + e.NumeroGiriRaggiunto);
}
```

Una seconda possibilità, anziché utilizzare il delegate generico EventHandler come appena fatto, è quella di definire un proprio delegate e quindi servirsene per la dichiarazione dell'evento. Per esempio, la classe Car potrebbe essere modificata come segue:

```
public class Car
{
delegate void GiriMotoreEventHandler(Car car, GiriMotoreEventArgs args);
public event GiriMotoreEventHandler MotoreFuoriGiri;
}
```

Di conseguenza, andrebbe leggermente modificato anche il metodo gestore dell'evento, nella classe CarMonitor, utilizzando la nuova firma:

```
//metodo gestore modificato
void car_MotoreFuoriGiri(Car car, GiriMotoreEventArgs e)
{
Console.WriteLine("Fuori giri: " + e.NumeroGiriRaggiunto);
}
```

Rimuovere gestori degli eventi

Una volta che si è certi che un metodo di gestore di un evento non debba più essere utilizzato, è possibile rimuoverlo dalla lista di sottoscrittori dell'evento stesso utilizzando l'operatore -= con il nome del metodo da rimuovere:

```
car.MotoreFuoriGiri -= car_MotoreFuoriGiri;
```

Eventi e classi figlie

Se una classe che definisce e pubblica un evento può essere derivata, la classe figlia potrebbe avere la necessità di scatenare lo stesso evento. Poiché un evento è un particolare tipo di delegate, esso non è invocabile in una classe derivata; per esempio, creando una classe ElectricCar figlia di Car, l'implementazione del metodo OnMotoreSpento seguente non sarebbe valida e provocherebbe un errore di compilazione, essendo l'evento MotoreSpento definito nella classe Car:

```
public class ElectricCar : Car
{
private void OnMotoreSpento()
{
if (MotoreSpento != null)
{
MotoreSpento(this, EventArgs.Empty);
}
}
```

```
}
```

Per aggirare il problema, bisogna consentire alla classe derivata di invocare il metodo OnMotoreSpento della classe base, quindi in quest'ultima si deve modificare tale evento rendendolo protected e virtual:

```
public class Car
{
protected virtual void OnMotoreSpento()
{
if (MotoreSpento != null)
{
MotoreSpento(this, EventArgs.Empty);
}
}
}
```

A questo punto, la classe ElectricCar può effettuare l'override invocando il metodo base:

```
public class ElectricCar : Car
{
protected override void OnMotoreSpento()
{
//eventuale codice personalizzato di ElectricCar

base.OnMotoreSpento();
}
}
```

Prima dell'invocazione del metodo base è sempre possibile eseguire del codice specifico della classe figlia.

NOTA	Sebbene in teoria un evento possa essere marcato con il modificatore virtual, e quindi se ne possa dichiarare poi un override in una classe derivata, tale pratica è sconsigliata dalla documentazione di Microsoft.

Funzioni personalizzate di accesso a eventi

Una funzione di accesso a un evento, o *event accessor*, è l'implementazione dei suoi operatori += e -=. Quando si utilizzano tali operatori per aggiungere e rimuovere metodi di gestione dell'evento, è in genere il compilatore che si occupa di creare implicitamente il codice necessario dietro le quinte.

Quando, per esempio, abbiamo scritto che la classe Car espone l'evento MotoreSpento:

```
public event EventHandler MotoreSpento;
```

e poi si è utilizzato l'operatore += per aggiungere un gestore di tale evento:

```
car.MotoreSpento += car_MotoreSpento;
```

il compilatore ha fatto in modo di creare un campo privato delegate del tipo utilizzato dall'evento e una coppia di funzioni di accesso.

In alcuni casi è però necessario fornire esplicitamente tali funzioni di accesso personalizzate, che andranno scritte replicando in pratica ciò che il compilatore in genere fa per noi, cioè creando del codice simile a quello di una proprietà, a eccezione del fatto che i due rami di accesso sono denominati add e remove, anziché get e set:

```
private EventHandler _motoreSpento;

public event EventHandler MotoreSpento
{
add
{
_motoreSpento+= value;
}
remove
{
_motoreSpento-=value;
}
}
```

Le funzioni di accesso personalizzate sono in genere utili in tre scenari:

- se la classe espone un numero elevato di eventi e non si vuole utilizzare un campo per ciascuno, in quanto si prevede che il numero di sottoscrittori degli stessi sarà basso. Tale scenario è tipico delle applicazioni Windows Forms, in cui per esempio ogni controllo di interfaccia espone decine di eventi, ma solo pochi di essi saranno gestiti;
- se la classe implementa esplicitamente delle interfacce che espongono eventi.

Nel primo scenario, la classe può definire un dizionario che conterrà i metodi sottoscrittori degli eventi:

```
private Dictionary<string, System.Delegate> eventTable;
```

Il dizionario eventTable dovrà essere inizializzato e necessiterà dell'aggiunta di un elemento per ogni evento della classe, con chiave corrispondente al nome dell'evento stesso. Per esempio, potremmo eseguire nel costruttore il seguente codice:

```
eventTable=new Dictionary<string,
```

Le funzioni di accesso personalizzate dei vari eventi faranno in modo di utilizzare tale dizionario, aggiungendo o rimuovendo lo specifico metodo di gestione dalla lista di invocazione del delegate corrispondente a ogni evento (memorizzato usando il nome come chiave):

```
public event EventHandler Event1
{
add
```

```
{
lock(eventTable)
{
eventTable["Event1"] = (EventHandler)eventTable["Event1"] + value;
}
}
remove
{
lock(eventTable)
{
eventTable["Event1"] = (EventHandler)eventTable["Event1"] - value;
}
}
}
```

Interfacce ed eventi

Un'interfaccia può definire anche degli eventi come membri da implementare.

Le regole per implementare un'interfaccia con eventi sono essenzialmente le stesse da seguire per metodi e proprietà:

```
public interface IVehicle
{
event EventHandler MotoreSpento;
}

public class Moto : IVehicle
{
public event EventHandler MotoreSpento;
}
```

Nel caso di interfacce multiple che definiscono eventi con lo stesso nome, o in generale implementando in maniera esplicita un'interfaccia, è necessario definire funzioni personalizzate di accesso agli eventi come visto nel precedente paragrafo.

Per esempio, se l'interfaccia IVehicle di cui sopra fosse implementata in maniera esplicita, dovranno essere necessariamente implementati i blocchi di accesso personalizzati add e remove per l'evento MotoreSpento, come mostrato di seguito:

```
public class Moto: IVehicle
{
private EventHandler _motoreSpento;

public event EventHandler MotoreSpento
{
add
{
_motoreSpento+=value;
}
remove
{
_motoreSpento-=value;
}
```

}
}

Eventi e interfaccia grafica

Nello sviluppo di applicazioni con interfaccia grafica la gestione degli eventi assume un ruolo fondamentale. Pensate, per esempio, alle decine di azioni e sorgenti che possono scatenare un evento: l'utente che clicca il mouse o digita sulla tastiera, una finestra che viene aperta o chiusa da codice, il sistema operativo che invia qualche messaggio all'applicazione.

Per mostrare come utilizzare il modello a eventi di .NET, vedremo ora come esso funziona all'interno di una applicazione Windows Forms, sfruttando il supporto che Visual Studio fornisce nella documentazione degli eventi disponibili per ogni controllo grafico e nella creazione di gestori di tali eventi.

NOTA	La gestione corretta degli eventi di un'applicazione Windows Forms richiede una conoscenza più approfondita dei vari tipi di eventi che ogni controllo espone, oltre che del ciclo di vita di un'applicazione a finestre. Già l'avvio di una tale applicazione implica il verificarsi di una serie di eventi, che è impossibile esaurire in poche righe: creazione dell'handle della finestra, caricamento della finestra, attivazione, visualizzazione sullo schermo e così via.

Il primo passo nella creazione di un'applicazione Windows Forms è la scelta del template di Visual Studio corretto. Avviando quindi la procedura di creazione di un nuovo progetto (per esempio, dal menu File scegliete New Project), all'interno della categoria Visual C#/Windows troverete, fra gli altri, il template Windows Forms Application (vedi Figura 10.3).

Add New Project

Recent

Installed

Visual C#

Windows

Universal

Windows 8

Classic Desktop

Web

Android

Cloud

Extensibility

iOS

LightSwitch

Office/SharePoint

Reporting

Silverlight

Test

WCF

Workflow

Telerik

Other Languages

Other Project Types

Modeling Projects

Online

.NET Framework 4.6 · Sort by: Default

Search Installed Templates (Ctrl+E)

Blank App (Universal Windows)	Visual C#
Windows Forms Application	Visual C#
WPF Application	Visual C#
Console Application	Visual C#
Shared Project	Visual C#
Class Library	Visual C#
Class Library (Portable)	Visual C#
Class Library (Universal Windows)	Visual C#
Windows Runtime Component (Universal Windows)	Visual C#
Telerik WPF Application	Visual C#
Unit Test App (Universal Windows)	Visual C#
Coded UI Test Project (Windows Phone)	Visual C#

Type: Visual C#

A project for creating an application with a Windows Forms user interface

Click here to go online and find templates.

Name: HelloWindowsForms

Location: c:\users\antonio\documents\visual studio 2015\Projects

Browse...

OK

Cancel

Figura 10.3 - Creazione di un progetto Windows Forms Application.

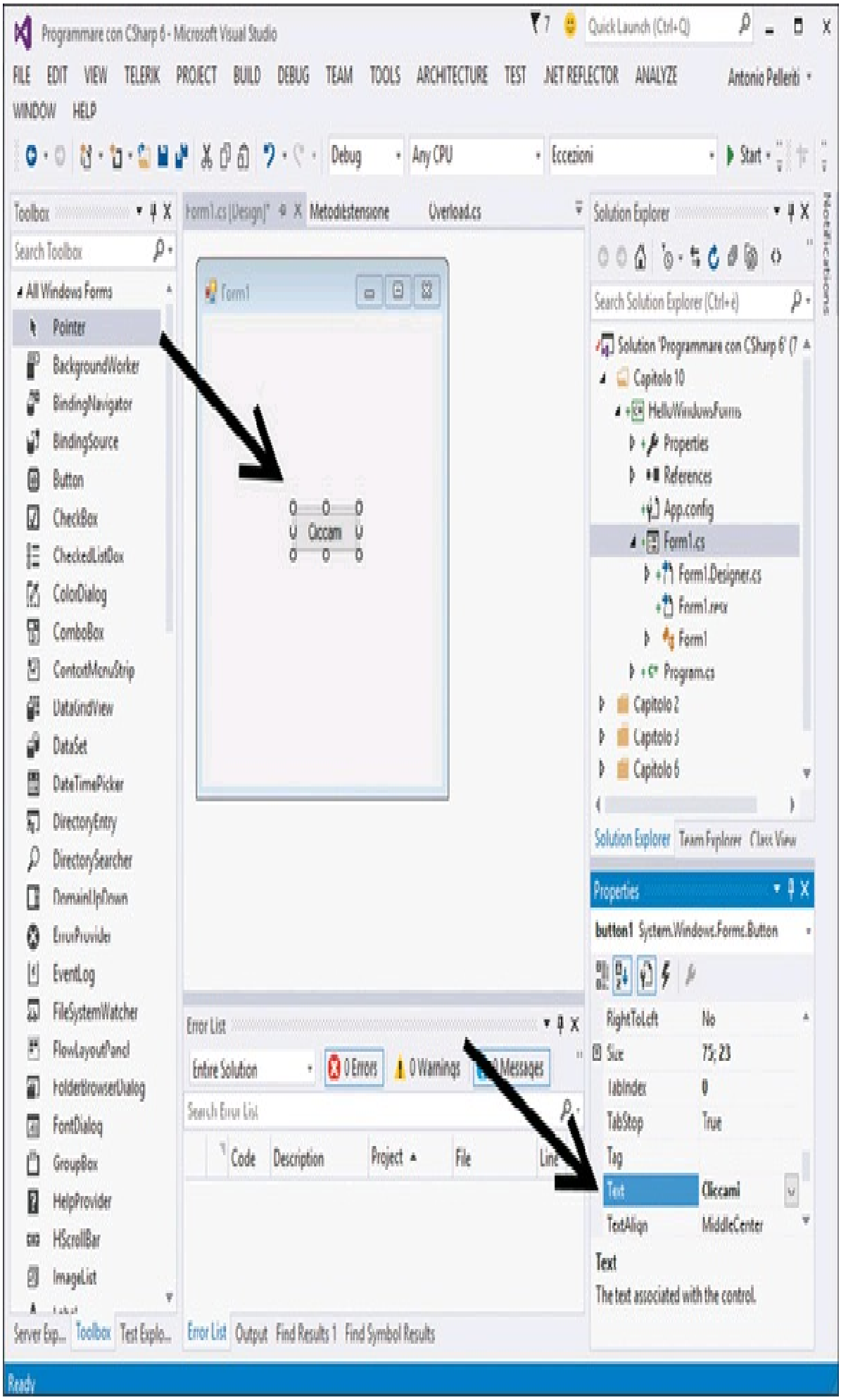

Programmare con CSharp 6 - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
FILE EDIT VIEW TELERIK PROJECT BUILD DEBUG TEAM TOOLS ARCHITECTURE TEST .NET REFLECTOR ANALYZE
Antonio Pelleriti
WINDOW HELP
Debug
Any CPU
Eccezioni
Start
Toolbox
Search Toolbox
All Windows Forms
Pointer
BackgroundWorker
BindingNavigator
BindingSource
Button
CheckBox
CheckedListBox
ColorDialog
ComboBox
ContextMenuStrip
DataGridView
DataSet
DateTimePicker
DirectoryEntry
DirectorySearcher
DomainUpDown
ErrorProvider
EventLog
FileSystemWatcher
FlowLayoutPanel
FolderBrowserDialog
FontDialog
GroupBox
HelpProvider
HScrollBar
ImageList
Form1.cs [Design]*
MetodiEstensione
Overload.cs
Form1
Cliccami
Error List
Entire Solution
0 Errors
0 Warnings
Messages
Search Error List
Code
Description
Project
File
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'Programmare con CSharp 6' (7
Capitolo 10
HelloWindowsForms
Properties
References
App.config
Form1.cs
Form1.Designer.cs
Form1.resx
Form1
Program.cs
Capitolo 2
Capitolo 3
Capitolo 6
Solution Explorer
Team Explorer
Class View
Properties
button1 System.Windows.Forms.Button
RightToLeft
No
Size
75; 23
TabIndex
0
TabStop
True
Tag
Text
Cliccami
TextAlign
MiddleCenter
Text
The text associated with the control.
Server Exp...
Toolbox
Test Explo...
Error List
Output
Find Results 1
Find Symbol Results
Ready

Figura 10.4 - Designer di Visual Studio per una Windows Forms Application.

Date un nome al vostro progetto, per esempio HelloWindowsForms, e fate clic su OK. Visual Studio si occuperà di creare una prima finestra, istanza della classe Form, e la mostrerà subito nel designer, al centro dell'ambiente di sviluppo. Selezionando la Form con un clic del mouse, la finestra delle proprietà in basso a destra mostrerà i vari attributi della finestra e permetterà di variarli a proprio piacimento. La casella degli strumenti, o *toolbox*, che si trova in genere ancorata a sinistra, permetterà di aggiungere nuovi controlli, trascinandoli direttamente sulla superficie della finestra.Supponiamo per esempio di voler creare un pulsante, istanza del controllo Button, in maniera che poi potremo gestire il clic del mouse su di esso, per eseguire una qualunque azione. Trascinando allora un oggetto Button sul designer, potete tornare alla finestra delle proprietà e modificare eventualmente la proprietà Text digitando direttamente il nuovo testo che volete assegnare al pulsante.

Programmare con CSharp 6 - Microsoft Visual Studio

FILE EDIT VIEW TELERIK PROJECT BUILD DEBUG TEAM TOOLS ARCHITECTURE TEST .NET REFLECTOR ANALYZE WINDOW HELP

Antonio Pelleriti

Debug | Any CPU | Eccezioni | Start

Toolbox

Search Toolbox

All Windows Forms

- Pointer
- BackgroundWorker
- BindingNavigator
- BindingSource
- Button
- CheckBox
- CheckedListBox
- ColorDialog
- ComboBox
- ContextMenuStrip
- DataGridView
- DataSet
- DateTimePicker
- DirectoryEntry
- DirectorySearcher
- DomainUpDown
- ErrorProvider
- EventLog
- FileSystemWatcher
- FlowLayoutPanel
- FolderBrowserDialog
- FontDialog
- GroupBox
- HelpProvider
- HScrollBar
- ImageList

Server Exp... Toolbox Test Explo...

Form1.cs [Design]* MetodiEstensione Overload.cs

Form1

Cliccami

Error List

Entire Solution | 0 Errors | 0 Warnings | 0 Messages

Search Error List

Code	Description	Project	File	Line

Error List Output Find Results 1 Find Symbol Results

Solution Explorer

Search Solution Explorer (Ctrl+è)

- Solution 'Programmare con CSharp 6' (7
  - Capitolo 10
    - HelloWindowsForms
      - Properties
      - References
      - App.config
      - Form1.cs
        - Form1.Designer.cs
        - Form1.resx
        - Form1
      - Program.cs
  - Capitolo 2
  - Capitolo 3
  - Capitolo 6

Solution Explorer Team Explorer Class View

Properties

button1 System.Windows.Forms.Button

RightToLeft	No
Size	75; 23
TabIndex	0
TabStop	True
Tag	
Text	Cliccami
TextAlign	MiddleCenter

Text

The text associated with the control.

Notifications

Ready

Figura 10.5 - Proprietà di un controllo Button.

Dopodiché, essendo interessati agli eventi gestibili da un controllo Button (o, per essere precisi, esposti dalla classe Button), fate clic sul pulsante Eventi, quello con il simbolo di un fulmine (vedi Figura 10.6).

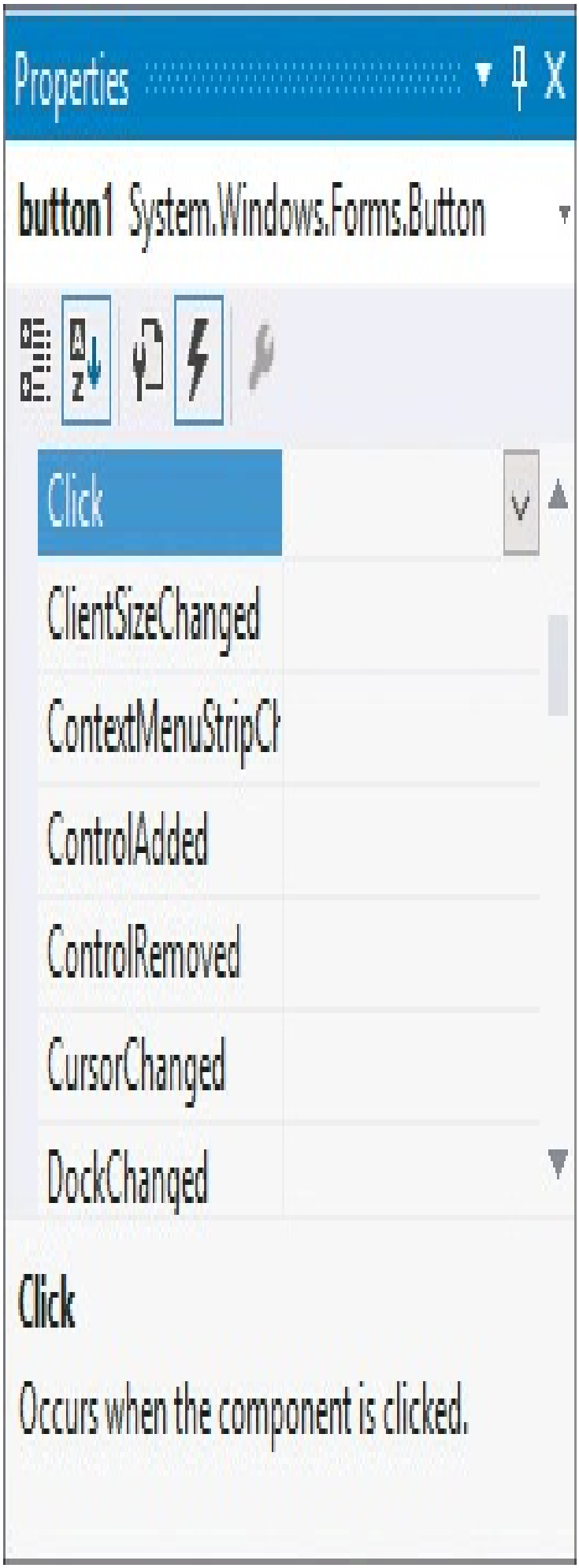

Properties
button1 System.Windows.Forms.Button
Click
ClientSizeChanged
ContextMenuStripCh
ControlAdded
ControlRemoved
CursorChanged
DockChanged
Click
Occurs when the component is clicked.

Figura 10.6 - Eventi di un controllo Button.

In tal modo, verranno elencati tutti gli eventi disponibili per l'oggetto selezionato nella finestra di design, che deve essere il pulsante appena creato; posizionatevi su quello da gestire, per esempio Click. La finestra delle proprietà mostrerà nella propria barra di stato una descrizione dell'evento selezionato, per esempio per Click apparirà la dicitura Occurs when the component is clicked. Non ci resta ora che digitare, nella casella di testo corrispondente all'evento, il nome del metodo gestore.

NOTA	Per velocizzare la creazione di gestione degli eventi, è possibile fare doppio clic sulla casella di testo vuota. Visual Studio utilizzerà in questo caso una convenzione del tipo nomeControllo_NomeEvento. Inoltre, il che vale anche in caso di gestore di evento già esistente, aprirà l'editor di codice, posizionandosi alla riga esatta del metodo.

Utilizzando la convezione standard (vedere la nota precedente), digitiamo per esempio button1_Click e premiamo il tasto Invio. Visual Studio creerà un gestore dell'evento vuoto, e aprirà l'editor di codice, visualizzando quindi un metodo come il seguente:

```
private void button1_Click(object sender, EventArgs e)
{
}
```

Non rimane che scrivere il codice che costituirà il corpo del metodo, cioè le azioni da eseguire al clic sul pulsante.

Supponiamo, per esempio, di voler visualizzare una classica finestrella con un messaggio di testo; per farlo basta utilizzare la classe MessageBox e uno degli overload del metodo Show:

```
private void button1_Click(object sender, EventArgs e)
{
MessageBox.Show("Hello World");
}
```

NOTA	Un'altra funzionalità di Visual Studio consente di creare il gestore di un evento predefinito in maniera ancora più rapida. Per il controllo Button tale evento è proprio Click, quindi facendo doppio clic su di esso all'interno del designer, l'IDE creerà il gestore dell'evento come fatto in precedenza.

A questo punto basta avviare il debug o l'esecuzione dell'applicazione, fare clic sul pulsante e, se tutto va bene, dovrebbe apparire la MessageBox come in Figura 10.7.

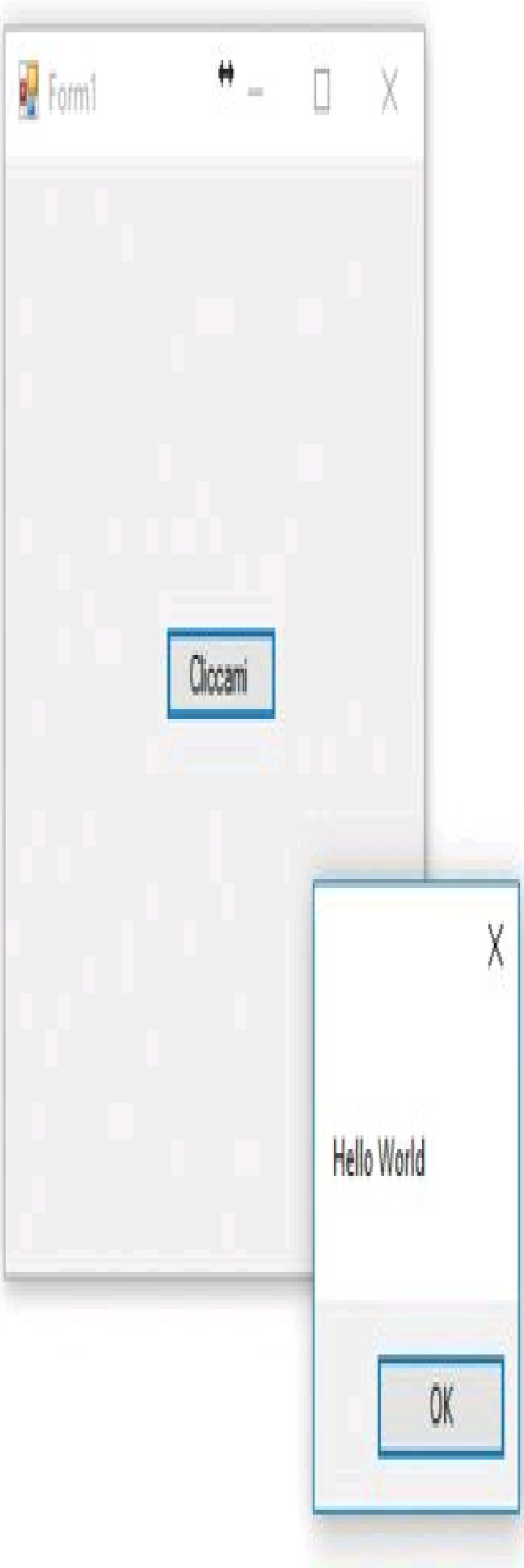

Form1
Cliccami
Hello World
OK

Figura 10.7 - MessageBox visualizzata al clic su un controllo Button.

Analizzando la classe Form1 creata per noi da Visual Studio, noterete che non appare alcuna istruzione in cui all'evento Click viene aggiunto il gestore button1_Click. Ricordatevi, però, che Visual Studio divide l'implementazione della classe Form1 in due file, e infatti la dichiarazione della classe utilizza il modificatore partial.

All'interno del file Form1.designer.cs, come mostrato nella Figura 10.8, ritroverete tutto il codice necessario, e dandoci un'occhiata potrete iniziare a capire come avviene la costruzione dell'interfaccia grafica direttamente via codice.

Form1.Designer.cs | Form1.cs | Form1.cs [Design]

HelloWindowsForms | HelloWindowsForms.Form1 | InitializeComponent()

```
this.button1.Name = "butt
this.button1.Size = new S
this.button1.TabIndex = 0
this.button1.Text = "Cliccami";
this.button1.UseVisualStyleBackColor = true;
this.button1.Click += new System.EventHandler(this.button1_Click);
//
// Form1
//
this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
this.ClientSize = new System.Drawing.Size(284, 261);
this.Controls.Add(this.button1);
this.Name = "Form1";
this.Text = "Form1";
this.ResumeLayout(false);

}

#endregion

private System.Windows.Forms.Button button1;
}
```

System.Drawing.Point.Point(int x, int y) (+ 3 overloads)

Initializes a new instance of the System.Drawing.Point class with the specified coordinates.

Solution Explorer

on Explorer (Ctrl+è)

n 'Programmare con CShar

- Capitolo 10
  - HelloWindowsForms
    - Properties
    - References
    - App.config
    - Form1.cs
      - Form1.Designer.cs
      - Form1.resx
      - Form1
    - Program.cs
- Capitolo 2
- Capitolo 3
- Capitolo 6
- Capitolo 8

Figura 10.8 - Codice di creazione dell'interfaccia grafica generato da Visual Studio.

Nulla vieta di creare il codice di gestione di un evento direttamente con le proprie mani, d'altronde conosciamo ormai la sintassi da utilizzare. Per esempio, se volessimo gestire l'evento FormClosing della finestra, che si verifica quando l'utente o qualche altra azione tenta di chiudere la finestra stessa, basterebbe scrivere all'interno del costruttore di Form1 la seguente istruzione:

```
this.FormClosing += Form1_FormClosing;
```

L'evento FormClosing è di tipo FormClosingEventHandler e un gestore di evento compatibile con esso potrebbe essere il seguente:

```
void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
MessageBox.Show("Bye Bye");
}
```

In questo modo, poco prima della chiusura della finestra, apparirà un'altra MessageBox.

Come esercizio, vi consiglio di provare a implementare la richiesta di una conferma di chiusura all'utente, mostrando una MessageBox con la scelta Sì/No. Il mio suggerimento è di utilizzare il parametro FormClosingEventArgs, che espone una proprietà Cancel.

Domande di riepilogo

1) Qual è il modo corretto di definire un delegate?

a. public delegate int MioDelegate(string str);
b. public int delegate MioDelegate(string str);
c. public int MioDelegate(delegate string str);
d. public delegate int MioDelegate(str);

2) Nella dichiarazione della variabile Func<int,int> f, f rappresenta:

a. un metodo con due argomenti di tipo int
b. un metodo con un argomento di tipo int e che restituisce un valore int
c. un metodo che restituisce una coppia di valori int
d. un metodo che restituisce lo stesso valore di ingresso

3) Qual è il modo corretto di inizializzare fx usando un'espressione lambda?

a. Func<int, int> fx = (int)x => x * x;
b. Func<int, int> fx = x => return x * x;
c. Func<int, int> fx = x => x * x;
d. Func<int, int> fx = return (x => x * x);

4) Qual è il modo corretto di dichiarare un evento MyEvent?

a. public void event EventHandler MyEvent
b. public EventHandler MyEvent
c. public event MyEvent(EventHandler)
d. public event EventHandler MyEvent

5) In che modo si gestisce l'evento Click di un controllo button1 con il metodo button1_Click?

a. button1_Click += Click
b. button1.Click += button1_Click
c. button1.Click(button1_Click)
d. button1_Click = button1.Click

6) Un'interfaccia può definire anche degli eventi come membri da implementare. Vero o falso?

7) Il delegate Action<int> rappresenta:

a. un metodo senza argomenti che restituisce un int
b. un metodo con un argomento int e che restituisce void

c. un metodo con un argomento `int` e che restituisce un valore `int`
d. un metodo che restituisce una collezione di valori `int`

LINQ

LINQ è un potente insieme di strumenti e funzionalità introdotte in C# 3.0 per scrivere con un'apposita nuova sintassi delle query di interrogazione di varie sorgenti di dati: collezioni di oggetti locali, dati XML letti da file oppure ottenuti da servizi web, database relazionali locali o remoti.

Questo capitolo affronta le funzionalità, introdotte in C# 3.0 e .NET 3.5, conosciute con l'acronimo di *LINQ*, che sta per *Language Integrated Query* (si pronuncia come *link*), e che permettono, per mezzo di una nuova apposita sintassi e di nuovi tipi aggiunti alla libreria di classi del framework, di effettuare interrogazioni su diversi formati di sorgenti di dati, utilizzando delle query simili a quelle utilizzate con database relazionali basate su SQL.

In particolare vedremo come utilizzare la nuova sintassi, gli operatori LINQ e l'insieme dei metodi di estensione aggiunti al .NET Framework per effettuare query su collezioni di tipo Enumerable<T>.

LINQ sfrutta, inoltre, una serie di funzionalità di C# che abbiamo già affrontato nei precedenti capitoli, spesso facendo notare che sarebbero tornate utili proprio con LINQ. Variabili implicite, tipi anonimi, metodi di estensione, inizializzatori di oggetti, espressioni lambda: saranno alcuni dei mattoni fondamentali da utilizzare per creare ed eseguire delle query LINQ.

Che cos'è LINQ

LINQ è un modello di programmazione, oltre che un insieme di funzionalità e tecnologie integrate direttamente nel linguaggio C# e nel .NET Framework (quindi valide anche per tutti gli altri linguaggi supportati da .NET).

La prima apparizione di LINQ risale al 2005, come estensione di Visual Studio 2005, mentre a partire da .NET 3.5, quindi dal 2007, esso ne fa parte come cittadino di prim'ordine e perfettamente integrato nel resto della piattaforma.

LINQ fornisce uno strato di astrazione basato su una sintassi per la scrittura delle query comune e su un insieme di metodi di estensione, chiamati anche *standard query operators*, che rendono possibile l'accesso a diverse sorgenti di dati, senza preoccuparsi dei dettagli implementativi di ognuna di esse. Inoltre, delle apposite regole di traduzione consentono di creare delle query LINQ a partire da espressioni lambda e tipi anonimi. A seconda della sorgente di dati cui si vuol accedere, sarà un'apposita e specifica implementazione di LINQ a tradurre ed eseguire le query. Tali implementazioni sono dette *LINQ Provider*.

Un LINQ Provider analizza la query e la traduce nel linguaggio e nelle istruzioni specifiche della sorgente dati. Per esempio, se si vuol utilizzare LINQ per eseguire una interrogazione su un database SQL Server, il provider *LINQ to SQL* creerà appunto delle query SQL che verranno poi inviate al database server.

.NET include gli assembly dei LINQ Provider seguenti:

- *LINQ to Objects* – non è un vero e proprio provider, in quanto esegue le query direttamente su collezioni di oggetti in memoria, come array e liste;
- *LINQ to SQL* – esegue le query su database SQL Server (in origine chiamato DLINQ);
- *LINQ to Entities* – basato sull'Entity Framework, che praticamente è uno strumento ORM (Object Relational Mapping) che crea degli oggetti .NET a partire dagli oggetti di un database (per esempio dalle tabelle);
- *LINQ to XML* – converte un documento XML in una collezione di oggetti XElement su cui esegue le query (in origine chiamato XLINQ);
- *LINQ to Datasets* – poiché LINQ to SQL è dedicato specificatamente a SQL Server, per supportare un generico database è possibile eseguire query su dei Dataset ricavati da qualunque database.

Esistono implementazioni di LINQ Provider non ufficiali dedicate alle più svariate sorgenti di dati: database di vari produttori, file Excel, JSON, o addirittura siti specifici come Google

NOTA o il social network Twitter. Basta provare a cercare su Internet se esiste il LINQ Provider per una particolare sorgente dati. Inoltre, se non esiste, potete cimentarvi nella sua implementazione.

La novità più visibile introdotta da LINQ è costituita senza dubbio dalle espressioni di query, scritte con una specifica *sintassi di query*. Una query viene formulata per mezzo di un'espressione ed è quindi utilizzabile come un qualsiasi altro costrutto del linguaggio. Ciò significa che le query sono fortemente tipizzate, in maniera che il compilatore C# possa già capire a tempo di compilazione se esse sono sintatticamente corrette (infatti le specifiche del linguaggio hanno introdotto anche la specifica delle espressioni di query LINQ).

Nei prossimi paragrafi utilizzeremo l'implementazione LINQ to Objects per scrivere degli esempi di query LINQ che agiscono su collezioni di oggetti, in particolare su array e su List<T>.

Espressioni di query

Le nuove funzionalità introdotte da LINQ e integrate direttamente in C# consentono di scrivere delle espressioni di query che possono eseguire interrogazioni su qualsiasi collezione di oggetti implementi le interfacce IEnumerable/IEnumerable<T> o IQueryable/IQueryable<T>.

NOTA	L'interfaccia IEnumerable<T> è stata affrontata nel Capitolo 9, parlando di collezioni. IQueryable<T> è una interfaccia derivata IEnumerable<T>, introdotta con LINQ, che consente di valutare query su sorgenti dati in maniera ottimizzata, utilizzando gli alberi di espressioni (introdotti nel Capitolo 10). In tal modo, è per esempio possibile inviare una query a un database server remoto, eseguirla e ricavare i risultati.

Le espressioni di query sono scritte con un'apposita *sintassi di query* dichiarativa (o *query syntax*) introdotta da C# 3.0, oppure mediante una serie di metodi di estensione o *operatori di query standard*, e in questo caso si parla di *sintassi di metodi* (o *method syntax*). Come vedremo, le due sintassi sono anche combinabili in una singola query.

La sintassi di query, per chi avesse un minimo di dimestichezza con i database relazionali, è molto simile a quella del linguaggio *SQL* (Structured Query Language): permette di dichiarare delle query, ma non specifica come esse verranno eseguite.

La sintassi di metodi, invece, funziona per mezzo della classica invocazione di una serie di metodi di estensione, quindi è una forma imperativa, in quanto evidenzia la sequenza di esecuzione dei vari operatori di query utilizzati. In particolare, i metodi di estensione estendono il tipo IEnumerable<T>.

Dal punto di vista delle prestazioni non vi è alcuna differenza, in quanto la query syntax è costituita da alias che vengono poi in ogni caso tradotti dal compilatore in chiamate ai metodi e quindi in method syntax.

La sintassi di query è senza dubbio più leggibile e chiara, quindi da preferire ove possibile. In alcuni casi è però necessario utilizzare i metodi di estensione perché alcuni operatori non sono disponibili nella sintassi di query. Le espressioni scritte con la sintassi di query, oltre a eseguire ricerche applicando filtri o criteri, permettono anche di eseguire ordinamenti e raggruppamenti dei risultati con una minima quantità di codice e hanno una leggibilità notevolmente alta.

Esempio con sintassi di query

Iniziamo a mostrare una prima espressione di query, che utilizza la query syntax, per comprenderne da subito le potenzialità e allo stesso tempo la semplicità.

Il primo passo fondamentale è ricordarsi di aggiungere la seguente direttiva using ai propri file .cs che utilizzano LINQ:

```
using System.Linq;
```

Naturalmente, sarà anche necessario aggiungere i riferimenti agli assembly contenenti i tipi e i metodi di estensione LINQ (ancora una volta, utilizzando Visual Studio non dobbiamo preoccuparci di tali aspetti: basta usare come versione target di .NET Framework la 3.5 o una superiore).

Creiamo ora un array di interi che, come ben sappiamo, implementa l'interfaccia IEnumerable<T> ed è quindi interrogabile con una query:

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

Una qualunque collezione di dati è chiamata in LINQ una *sequenza di elementi*. Quindi l'array precedente è una sequenza i cui elementi sono dei numeri interi.

Definiamo una query che estragga dalla sequenza tutti gli elementi pari e ne restituisca il quadrato:

```
IEnumerable<int> query= from n in array
where n%2 == 0
select n*n;
```

La clausola where applica un filtro, costituito da un'espressione lambda, all'array originale, mentre quella select restituisce il risultato del calcolo del quadrato. In grassetto è evidenziata la query scritta con una sintassi finora mai incontrata, ma che è puro C#. Infatti, viene riconosciuta sia da Visual Studio, che evidenzia le parole chiave from, in, where, select, sia soprattutto dal compilatore, che appunto la compilerà senza problemi.

NOTA	Un ottimo strumento complementare a Visual Studio, che consente di scrivere ed eseguire query LINQ collegandosi eventualmente a varie sorgenti dati, è il già citato LINQPad. Potete utilizzarlo per testare il funzionamento corretto delle query prima di integrarle nelle vostre applicazioni. Fra l'altro, ne esiste anche una versione completamente gratuita e con le funzionalità necessarie a tale scopo.

La precedente è però solo una dichiarazione della query che, per restituire dei risultati, deve ora essere eseguita. Per farlo è sufficiente utilizzarla come una qualunque enumerazione, ottenendo gli elementi che fanno parte del risultato.

```
foreach(int result in query)
```

```
{
Console.WriteLine(result);
}
```

Esempio con sintassi di metodi

La stessa query precedente può essere scritta con la sintassi di *method syntax*, cioè utilizzando i metodi di estensione:

```
IEnumerable<int> query = array.Where(n => n %2==0).Select(n=>n*n);
```

La query viene costruita combinando prima un metodo Where e poi un metodo Select. Si noti come le espressioni lambda utilizzate come parametro siano rimaste identiche.

Anche in questo caso la query deve essere ora eseguita, enumerando i risultati:

```
foreach(int result in query)
{
Console.WriteLine(result);
}
```

Esempio con sintassi mista

Le due sintassi possono essere combinate: il che è spesso una scelta obbligata, soprattutto quando un certo operatore di query è disponibile solo come metodo di estensione.

In questo caso, per esempio, è possibile dichiarare una query con la query syntax e poi applicare ulteriori operatori di query, invocando uno o più metodi di estensione:

```
int numeroRisultati= (from n in array
where n%2 == 0
select n*n).Count();
```

L'esempio precedente conta il numero di risultati ottenuti dalla query utilizzata in precedenza, applicando a essa l'operatore Count.

Variabili di query

Sebbene negli esempi precedenti abbiamo dichiarato il tipo restituito da una query come IEnumerable<int>, dato che l'array su cui eseguire la ricerca era un array di int, in genere non è facilmente intuibile il tipo di risultato restituito da una query.

Infatti, tale tipo può anche non essere ovvio né addirittura accessibile direttamente da codice perché, come vedremo fra qualche paragrafo, il risultato può essere generato anche utilizzando dei tipi anonimi. Quindi è comodo dichiarare la variabile di query, cioè la variabile che conterrà la query da eseguire, e non il risultato, come variabile di tipo implicito. Una delle query precedenti, per esempio, può essere dichiarata come segue, utilizzando la parola chiave var:

```
var query= from n in array
where n%2 == 0
select n*n;
```

A questo punto, per l'esecuzione non cambia nulla: si può tranquillamente eseguire un foreach sulla variabile query.

Esecuzione differita

Come già si sarà capito dagli esempi di interrogazioni appena scritte, l'esecuzione di una query, cioè la sua valutazione, avviene in maniera differita rispetto alla sua dichiarazione. Riprendiamo il primo degli esempi precedenti:

```
//dichiarazione della query
var query= from n in array
where n%2 == 0
select n*n;
```

In esso, l'assegnazione alla variabile query costituisce solo una dichiarazione di ciò che la query dovrà fare. I risultati veri e propri si otterranno solo eseguendo il successivo metodo foreach:

```
//esecuzione differita
foreach(int result in query)
Console.WriteLine(result);
```

Questa viene chiamata *esecuzione differita* o *esecuzione pigra* (*lazy*) della query ed è valida se l'espressione della query restituisce un IEnumerable<T>. In tal modo, se i dati su cui la query deve agire cambiano, anche i risultati cambieranno, perché la query verrà eseguita sui nuovi dati:

```
int[] array= {1,2,3,4};
var query= from n in array
where n%2 == 0
select n*n;

//prima esecuzione
foreach(int result in query)
Console.WriteLine(result); //risultato 4, 16

//modifica dei dati
array[1]=4;
array[3]=6;

//nuova esecuzione
foreach(int result in query)
Console.WriteLine(result); //risultato 4, 16
```

Se la query restituisce però un numero scalare, come nel caso dell'esempio che utilizza l'operatore Count, l'esecuzione è immediata:

```
//esecuzione immediata
var count= (from n in array
where n%2 == 0
select n*n).Count();
```

Il risultato del conteggio viene immediatamente assegnato alla variabile count.

Operatori LINQ

Le potenzialità di LINQ derivano dall'insieme di operatori utilizzabili per scrivere delle query, che di base vengono espresse tramite la sintassi di query, di cui ci si può servire con qualsiasi LINQ Provider e che quindi può agire su qualsiasi tipo di sorgente dati.

Combinando la sintassi di query con uno o più operatori di query standard, le potenzialità di LINQ ne risultano notevolmente aumentate, fornendo un insieme di strumenti ancora più potente e che può eseguire query con un controllo ancora maggiore sui dati da ottenere. Gli operatori di query sono implementati tramite metodi di estensione delle classi statiche Enumerable e Queryable, che si trovano nell'assembly System.Core.dll, e che possono essere utilizzate su collezioni di oggetti che implementano le interfacce IEnumerable<T> e IQueryable<T>.

Tipi di operatori

Esistono decine di operatori LINQ, appartenenti a diverse tipologie, il cui utilizzo verrà mostrato in dettaglio nei paragrafi che seguono.

NOTA	Tutti i metodi di estensione implementati nelle classi Enumerable e Queryable possono essere elencati dal Visualizzatore oggetti di Visual Studio. Dal menu View, selezionate la voce Object Browser, poi navigate all'interno dell'assembly System.Core, nel namespace System.Linq, per trovare le due classi. Selezionandole, sulla sinistra apparirà l'elenco dei metodi disponibili.

Object Browser | Formula1Data.cs | Program.cs* | Program.cs

Browse: My Solution

<Search>

- System.Diagnostics
- System.Diagnostics.Eventing
- System.Diagnostics.Eventing.Reader
- System.Diagnostics.PerformanceData
- System.Dynamic
- System.IO
- System.IO.MemoryMappedFiles
- System.IO.Pipes
- System.Linq
  - Enumerable
  - EnumerableExecutor
  - EnumerableExecutor<T>
  - EnumerableQuery
  - EnumerableQuery<T>
  - IGrouping<out TKey,out TElement>
  - ILookup<TKey,TElement>
  - IOrderedEnumerable<TElement>
  - IOrderedQueryable
  - IOrderedQueryable<out T>
  - IQueryable
  - IQueryable<out T>
  - IQueryProvider
  - Lookup<TKey,TElement>
  - OrderedParallelQuery<TSource>
  - ParallelEnumerable
  - ParallelExecutionMode
  - ParallelMergeOptions
  - ParallelQuery
  - ParallelQuery<TSource>
  - Queryable
- System.Linq.Expressions
- System.ManagementInstrumentation
- System.Runtime.CompilerServices
- System.RuntimeInteropServices

Take<TSource>(this System.Collections.Generic.IEnumerable<TSource>, int)
TakeWhile<TSource>(this System.Collections.Generic.IEnumerable<TSource>, System.Func<TSource,int,bool>)
TakeWhile<TSource>(this System.Collections.Generic.IEnumerable<TSource>, System.Func<TSource,bool>)
ThenBy<TSource, TKey>(this System.Linq.IOrderedEnumerable<TSource>, System.Func<TSource,TKey>, System.C
ThenBy<TSource, TKey>(this System.Linq.IOrderedEnumerable<TSource>, System.Func<TSource,TKey>)
ThenByDescending<TSource, TKey>(this System.Linq.IOrderedEnumerable<TSource>, System.Func<TSource,TKey
ThenByDescending<TSource, TKey>(this System.Linq.IOrderedEnumerable<TSource>, System.Func<TSource,TKey
ToArray<TSource>(this System.Collections.Generic.IEnumerable<TSource>)
ToDictionary<TSource, TKey, TElement>(this System.Collections.Generic.IEnumerable<TSource>, System.Func<TS
ToDictionary<TSource, TKey, TElement>(this System.Collections.Generic.IEnumerable<TSource>, System.Func<TS
ToDictionary<TSource, TKey>(this System.Collections.Generic.IEnumerable<TSource>, System.Func<TSource,TKe
ToDictionary<TSource, TKey>(this System.Collections.Generic.IEnumerable<TSource>, System.Func<TSource,TKe
ToList<TSource>(this System.Collections.Generic.IEnumerable<TSource>)
ToLookup<TSource, TKey, TElement>(this System.Collections.Generic.IEnumerable<TSource>, System.Func<TSou
ToLookup<TSource, TKey, TElement>(this System.Collections.Generic.IEnumerable<TSource>, System.Func<TSou
ToLookup<TSource, TKey>(this System.Collections.Generic.IEnumerable<TSource>, System.Func<TSource,TKey>,
ToLookup<TSource, TKey>(this System.Collections.Generic.IEnumerable<TSource>, System.Func<TSource,TKey>
Union<TSource>(this System.Collections.Generic.IEnumerable<TSource>, System.Collections.Generic.IEnumerable
Union<TSource>(this System.Collections.Generic.IEnumerable<TSource>, System.Collections.Generic.IEnumerable
Where<TSource>(this System.Collections.Generic.IEnumerable<TSource>, System.Func<TSource,int,bool>)
Where<TSource>(this System.Collections.Generic.IEnumerable<TSource>, System.Func<TSource,bool>)
Zip<TFirst, TSecond, TResult>(this System.Collections.Generic.IEnumerable<TFirst>, System.Collections.Generic.IE

public static System.Collections.Generic.IEnumerable<TSource> **Where<TSource>**(this System.Collections.Generic.IEnumerable<TSource> source, System.Func<TSource,int,bool> *predicate*)
Member of System.Linq.Enumerable

Summary:
Filters a sequence of values based on a predicate. Each element's index is used in the logic of the predicate function.

Type Parameters:
TSource: The type of the elements of source.

Figura 11.1 - I metodi di estensione LINQ nel Visualizzatore oggetti di Visual Studio.

Ogni operatore è disponibile in diversi overload. Ciascuno di essi, in quanto metodo di estensione, ha come primo parametro un riferimento a un oggetto IEnumerable<T> o IQueryable<T>. Molti di questi metodi richiedono poi di specificare altri parametri che indichino all'operatore come eseguire la propria operazione. Tali parametri sono nella maggior parte dei casi dei delegate generici, che sono stati anch'essi introdotti da LINQ e che abbiamo già incontrato nel Capitolo 10: i delegate Func<> e Action<>.

Come esempio, ecco la firma di uno degli overload dell'operatore Count implementato dal seguente metodo di estensione:

```
public static int Count<TSource>(this IEnumerable<TSource> source, Func<TSource,bool> predicate);
```

In genere, i delegate Func e Action vengono creati mediante l'assegnazione di una espressione lambda, che permette in maniera rapida e semplice di esprimere delle condizioni. Per esempio, l'operatore Count precedente, che serve a contare il numero di elementi che fanno parte di una sequenza, può prendere come parametro un delegate Func<T, bool> in maniera da restringere gli elementi da contare. Il delegate restituirà un bool per ogni elemento della sequenza, a indicare se l'elemento debba essere incluso nei risultati, e quindi nel conteggio, o meno.

Possiamo per esempio contare solo gli elementi positivi di un array di interi in questo modo:

```
var query=(from n in array select n);

int count= query.Count(n=> n>0);
```

Come noterete, l'utilizzo delle espressioni lambda è veramente immediato e comodo.

Tuttavia, un delegate Func o Action può anche essere costruito mediante la classica sintassi dei delegate, ove necessario. Per esempio, supponiamo di dover utilizzare un metodo esistente che restituisca un bool per ogni oggetto passato a esso come parametro:

```
public bool IsPositive(int n)
{
return n>0;
}
```

Un delegate Func<int,bool> può essere istanziato utilizzando il metodo precedente:

```
Func<int, bool> func=new Func<int, bool> (IsPositive);
```

E a questo punto passato come parametro di un operatore LINQ, per esempio il già visto Count:

```
int count=query.Count(func);
```

Tale modalità di utilizzo degli operatori standard di LINQ sarà praticamente la norma nel seguito, specialmente per quelli che non hanno una parola chiave nella sintassi di query.

La seguente tabella contiene un elenco di tutti gli operatori standard per le query LINQ, suddivisi per tipologia e accompagnati da eventuali parole chiave C# utilizzabili per esprimerli mediante sintassi di query.

Tabella 11.1 **- Operatori standard di LINQ.**

Tipologia	Operatore	Descrizione	Keyword
Filtro	Where	specifica gli elementi da escludere dalla sequenza di origine	where
Proiezione	Select	proietta ogni elemento di una sequenza	select
	SelectMany	proietta ogni elemento di una sequenza uno a molti	
Ordinamento	OrderBy	ordina una sequenza in maniera crescente	orderby
	OrderByDescending	ordina una sequenza in maniera decrescente	orderby ascending descending
	ThenBy	ordina una sequenza già ordinata in maniera crescente	
	ThenByDescending	ordina una sequenza già ordinata in maniera decrescente	
	Reverse	inverte l'ordinamento di una sequenza	
Raggruppamento	GroupBy	raggruppa gli elementi di una sequenza in base a un criterio	group … by
Unione	Join	esegue un'unione interna (*inner join*) di due sequenze usando proprietà comuni	join … in … on … equals …
	GroupJoin	esegue un'unione di due sequenze raggruppando i risultati	join … in … on … equals … into …
Insiemi	Distinct	elimina i duplicati da una sequenza	
	Except	restituisce la differenza fra due sequenze	
	Intersect	restituisce le intersezioni fra due sequenze	
	Union	restituisce l'unione fra due sequenze	
	Zip	unisce due sequenze applicando una funzione a ogni coppia di elementi	
Partizionamento	Skip	salta un dato numero di elementi della sequenza	
	SkipWhile	salta gli elementi di una sequenza fino a quando una condizione è vera	

	Take	restituisce un dato numero di elementi della sequenza	
	TakeWhile	restituisce gli elementi di una sequenza fino a quando una condizione è vera	
Concatenazione	Concat	concatena due sequenze in un'unica	
Conversione	AsEnumerable	restituisce una sequenza come IEnumerable<T>	
	Cast	converte gli elementi di una sequenza verso un dato tipo	from tipo in
	OfType	restituisce gli elementi di una sequenza compatibili con un dato tipo	
	ToArray	converte una sequenza in array	
	ToDictionary	converte una sequenza in un dizionario	
	ToList	converte una sequenza in lista	
	ToLookup	converte una sequenza in un oggetto Lookup	
Uguaglianza	SequenceEqual	verifica se due sequenze sono uguali confrontando i rispettivi elementi	
Elementi	ElementAt	restituisce l'elemento all'indice specificato	
	ElementAtOrDefault	restituisce l'elemento all'indice specificato, o un valore predefinito se non esiste l'indice	
	First	restituisce il primo elemento di una sequenza	
	FirstOrDefault	restituisce il primo elemento di una sequenza, o un valore predefinito se non viene trovato nessun elemento	
	Last	restituisce l'ultimo elemento di una sequenza	
	LastOrDefault	restituisce l'ultimo elemento di una sequenza, o un valore predefinito se non viene trovato nessun elemento	
	Single	restituisce l'unico elemento di una sequenza	
	SingleOrDefault	restituisce l'unico elemento di una sequenza, o un valore predefinito se non viene trovato nessun elemento	
Generazione	Empty	restituisce una sequenza vuota	
	DefaultIfEmpty	restituisce un elemento predefinito se la sequenza è vuota	
	Range	restituisce una sequenza di numeri interi	
	Repeat		

		restituisce una sequenza di elementi ripetuti	
Quantificatori	All	restituisce true se tutti gli elementi di una sequenza soddisfano una condizione	
	Any	restituisce true se almeno un elemento soddisfa una condizione	
	Contains	restituisce true se la sequenza contiene un dato elemento	
Aggregazione	Aggregate	esegue una funzione personalizzata sugli elementi di una sequenza	
	Average	calcola la media di una sequenza numerica	
	Count	conta gli elementi di una sequenza, eventualmente quelli che soddisfano una condizione	
	LongCount	come Count, ma per sequenze molto grandi, restituisce un valore di tipo long	
	Max	restituisce il valore massimo di una sequenza	
	Min	restituisce il valore minimo di una sequenza	
	Sum	restituisce la somma dei valori di una sequenza	

Nei prossimi paragrafi si vedranno diversi esempi di utilizzo dei vari operatori standard di LINQ.

Sintassi delle query

Nei precedenti paragrafi abbiamo visto dei semplici esempi scritti mediante la *query syntax* di LINQ. La sintassi delle query è però più completa di quanto finora visto e comprende altre parole chiave per dichiarare query più articolate e complesse. Vedremo ora una spiegazione completa della sintassi, mentre il significato e l'utilizzo di ogni possibile clausola verrà esposto nei paragrafi seguenti più nel dettaglio.

Ogni espressione di query inizia sempre con una clausola from, seguita dal corpo della query, e termina sempre con la clausola select o con la clausola group.

NOTA	Chi conosce il linguaggio SQL avrà notato che le query LINQ seguono un ordine differente da quelle SQL. In particolare, una query SQL inizia con l'istruzione select e termina con from, oppure con where se esiste un filtro. Fra le ragioni per cui LINQ inizia con select vi è anche l'esigenza da parte di Visual Studio di conoscere la variabile da utilizzare nel seguito della query, così da poter fornire per essa funzionalità di intellisense e autocompletamento.

La clausola select esegue la proiezione del risultato di un'espressione in un'enumerazione, mentre group lo proietta in un insieme di gruppi basati su delle condizioni di raggruppamento (ogni gruppo è un'enumerazione). Altre clausole, come vedremo a breve, consentono di creare query più articolate.

La sintassi completa di una query LINQ ha quindi il seguente formato:

```
from [tipo] identificatore in espressione
[clausole-query-body]
clausola select | clausola group
[continuazione-query]
```

Le parti fra parentesi quadre indicano parti opzionali, il carattere | indica invece la scelta fra una delle due clausole: select oppure group.

NOTA	Le specifiche del linguaggio C# contengono anche la specifica completa della sintassi per scrivere espressioni di query LINQ, a dimostrare che esse fanno parte del linguaggio C# e vengono compilate come qualsiasi altro costrutto.

Ogni clausola from genera una variabile di range locale, che assumerà a ogni iterazione il valore di uno degli elementi di una sequenza.

La clausola iniziale from può essere seguita da una o più clausole from, let, where, join o orderby, che a loro volta hanno questo formato:

let identificatore = espressione
where espressione-booleana
join [tipo] identificatore **in** espressione **on** espressione **equals** espressione
join [tipo] identificatore **in** espressione **on** espressione **equals** espressione **into** identificatore
orderby tipi-ordinamento

Ogni clausola let introduce una nuova variabile che permette di salvare temporaneamente il risultato di una sotto-query, per poterlo riutilizzare nella query principale.

Ogni clausola where rappresenta un filtro di ricerca degli elementi di una sequenza.

Ogni join confronta specifiche chiavi di una sequenza sorgente con quelle di un'altra sequenza, in maniera da combinare delle coppie.

Ogni clausola orderby esegue l'ordinamento degli elementi in base a dei criteri, in maniera ascendente o discendente.

La clausola finale, che può essere come già detto una fra select oppure group, specifica il formato del risultato, in termini della variabile di range.

Infine, una clausola opzionale into può essere utilizzata per concatenare le query, passando i risultati di una a quella successiva.

Nei seguenti paragrafi tali clausole verranno utilizzate per scrivere delle query, quindi il loro funzionamento sarà spiegato più nel dettaglio.

Modello dati di esempio

Nei prossimi paragrafi vedremo vari esempi di query più o meno articolate, che utilizzano le varie clausole della sintassi di query e poi gli operatori mediante metodi di estensione. Prima di andare avanti, però, è necessario costruirci una sorgente dati: in questo caso una collezione di oggetti, da utilizzare nei vari esempi.

Supponiamo dunque di voler implementare un'applicazione per la gestione del campionato di Formula 1. In particolare si gestiranno le squadre e i relativi piloti, e le due classi seguenti sono lo stretto necessario per farlo:

```
class F1Team
{
public string TeamName {get; set;}
public Pilot[] Pilots {get; set;}
public int Wins {get; set;}
public override string ToString()
{
return String.Format("Team {0}, Wins: {1}", TeamName, Wins);
}
}

class Pilot
```

```
{
public F1Team Team {get; set;}
public string FirstName {get; set;}
public string LastName {get; set;}
public int IDCountry {get; set;}
public int Points {get; set;}
}

class Country
{
public int IDCountry {get; set;}
public string Name {get; set;}

public static List<Country> All = new List<Country>(
new Country[]
{
new Country(){IDCountry=1, Name="Spagna"},
new Country(){IDCountry=2, Name="Brasile"},
new Country(){IDCountry=3, Name="Germania"},
new Country(){IDCountry=4, Name="Australia"},
new Country(){IDCountry=5, Name="Regno Unito"},
new Country(){IDCountry=6, Name="Messico"},
new Country(){IDCountry=7, Name="Finlandia"},
new Country(){IDCountry=8, Name="Svizzera"},
new Country(){IDCountry=9, Name="Italia"},
new Country(){IDCountry=10, Name="USA"},
}
);
}
```

Utilizzando le due classi F1Team e Pilot, istanziamo e popoliamo la seguente collezione di squadre e di relativi piloti. La classe Country espone invece una proprietà statica All che restituisce un elenco delle nazioni, il cui valore IDCountry è utilizzato da ogni istanza di Pilot. La collezione teams, inizializzata di seguito, verrà utilizzata in tutti i prossimi esempi di query LINQ come collezione di partenza:

```
List<F1Team> teams= new List<F1Team>();
teams .Add(
new F1Team() { TeamName = "Ferrari", Wins=7, Pilots = new Pilot[]
{
new Pilot() { FirstName="Fernando", LastName="Alonso", IDCountry=1, Points=120 },
new Pilot() { FirstName="Felipe", LastName="Massa", IDCountry=2, Points=42 }
}
});
teams.Add(
new F1Team()
{
TeamName = "Red Bull", Wins=10 Pilots = new Pilot[2]
{
new Pilot() { FirstName="Sebastian", LastName="Vettel", IDCountry=3, Points=112 },
new Pilot() { FirstName="Mark", LastName="Webber", IDCountry=4, Points=77 }
}
});
```

```
teams.Add(
new F1Team()
{
TeamName = "McLaren",Wins=2, Pilots = new Pilot[2]
{
new Pilot() { FirstName="Jenson", LastName="Button", IDCountry=5, Points=88 },
new Pilot() { FirstName="Sergio", LastName="Perez", IDCountry=6,Points=30 }
}
});
```

```
teams.Add(
new F1Team()
{
TeamName = "Lotus", Wins=3, Pilots = new Pilot[2]
{
new Pilot() { FirstName="Kimi", LastName="Raikkonen", IDCountry=7,Points=90 },
new Pilot() { FirstName="Romain", LastName="Grosjean", IDCountry=8,Points=51 }
}
});
```

```
teams.Add(
new F1Team()
{
TeamName = "Mercedes Petronas", Wins=1, Pilots = new Pilot[2]
{
new Pilot() { FirstName="Nico", LastName="Rosberg", IDCountry=3, Points=63 },
new Pilot() { FirstName="Lewis", LastName="Hamilton", IDCountry=5,Points=98 }
}
});
```

Inserite il codice precedente in un metodo o in una proprietà statica di una classe qualunque, e fatevi restituire la lista teams in maniera da poterla riutilizzare facilmente. Per esempio:

```
class F1Data
{
public static List<F1Team> Teams
{
//inserire qui codice precedente
return teams;
}
}
```

Inoltre, per comodità e per evitare di scrivere sempre lo stesso ciclo foreach, vi consiglio anche di implementare il seguente metodo Dump per stampare i risultati ottenuti da una query LINQ (nel caso di risultati enumerabili e non scalari):

```
public static void Dump(IEnumerable query)
{
foreach (var q in query)
Console.WriteLine(q);
}
```

Selezione di dati

La query LINQ più semplice è quella che permette di ottenere dei risultati da una collezione di dati senza applicare alcun filtro ed è costruita, con la sintassi di query, mediante le keyword from, in e select.

```
List<F1Team> teams=GetTeams();

var query=from team in teams
select team;
```

Tramite questa query si ottengono tutti gli oggetti F1Team contenuti nella lista teams. In effetti, non è di molta utilità, dato che restituisce una collezione di oggetti che già avevamo nella sorgente di dati iniziale.

L'identificatore che segue la parola chiave from è la variabile di iterazione o di range, la quale verrà utilizzata nel seguito della query: in particolare, in questo caso come variabile della clausola select, dove conterrà ogni elemento da restituire fra i risultati.

Per il momento, ciò che vogliamo far notare è che il tipo dei risultati può essere ricavato dal compilatore mediante inferenza, se il tipo della sorgente dati è conosciuto. In questo esempio la collezione è una List<F1Team>, quindi il risultato di una query su tale sorgente sarà un IEnumerable<F1Team>. Invece, nel caso in cui il tipo di dati non è specificato, per esempio perché la query viene eseguita su un oggetto IEnumerable non tipizzato, come un ArrayList, la clausola from permette di specificare mediante la sintassi il tipo di dati da ricavare:

from **tipo** identificatore in collezione

Per esempio:

```
ArrayList list=new ArrayList();
list.AddRange(F1Data.Teams);

var query=from F1Team team in list
select team;
```

Una query termina con una clausola select o una clausola group.

Nei due esempi precedenti si è utilizzata una select semplice che restituisce una sequenza dello stesso tipo degli oggetti contenuti nella sorgente dati: il che, come abbiamo scritto, è abbastanza inutile perché la sequenza ottenuta è praticamente la stessa collezione contenuta nella sorgente dati originaria.

La clausola select può tuttavia essere utilizzata per trasformare gli elementi nella sorgente dati in una sequenza di nuovi tipi, oppure per ottenere solo determinate proprietà da tali elementi. Questa procedura, che approfondiremo anche nel seguito, si chiama *proiezione*.

Per esempio, se ci interessa ricavare il nome di tutte le squadre di Formula 1, possiamo specificare il nome della proprietà TeamName della classe F1Team:

```
var query = from team in teams
select team.TeamName;
```

In questo caso, il risultato della query sarà un'enumerazione di stringhe.

Come già abbiamo avuto modo di dire, ogni espressione LINQ scritta mediante la sintassi di query C# viene poi trasformata in una sequenza di invocazioni ai metodi di estensione LINQ. L'ultimo esempio, quindi, può essere scritto come segue, ed è in qualcosa di simile che il compilatore tradurrà l'espressione:

```
var query = teams.Select(team => team.TeamName);
```

Sotto-query

La sintassi generale delle espressioni LINQ consente di specificare all'interno del corpo di una query ulteriori query, innestando quindi più clausole from. In molti casi, infatti, l'elemento ricavato da una sequenza può essere a sua volta una sequenza o contenerne una. Riprendendo il modello dati di esempio, ogni oggetto F1Team espone appunto una proprietà Pilots, che restituisce un array di oggetti Pilot. Per dichiarare una query che acceda all'array dei piloti di ogni team è necessario utilizzare una sotto-query, creando di fatto una clausola from composta:

```
var query = from team in teams
from pilot in team.Pilots
select pilot.LastName;
```

La query così scritta consente di ricavare tutti i cognomi dei piloti di tutti i team presenti nella sorgente dati.

Naturalmente, ogni clausola from potrà specificare le proprie clausole where e/o group by, per restringere o raggruppare i risultati. Per esempio, se si vuole restringere l'insieme dei risultati con due clausole where, una che prenda solo i team che hanno almeno un pilota (tutti!) e una che invece restringa l'insieme dei piloti a quelli il cui cognome sia più lungo di sette caratteri, la query può essere modificata come segue:

```
var query = from team in teams
from pilot in team.Pilots
where pilot.LastName.Length>8
where team.Pilots.Length>1
select pilot.LastName;
```

L'indentazione utilizzata è facoltativa, ma sicuramente usarla rende più leggibili le query, soprattutto se molto lunghe e articolate.

La parola chiave let

In un'espressione di query è spesso utile, e molte volte necessario, salvare il risultato di una sotto-espressione in una nuova variabile, in maniera da poterla utilizzare poi con le clausole successive.

La parola chiave let permette di creare una nuova variabile di range e di inizializzarla con il risultato di un'espressione. Una volta inizializzata, a essa non può essere assegnato un nuovo valore, ma può essere utilizzata per leggerne i risultati che contiene.

Nella seguente query, la clausola let memorizza il valore della proprietà Wins di ogni team, lo utilizza per ordinarli e se ne serve per filtrare la squadra con più di tre vittorie.

```
var query = from team in teams
let wins = team.Wins
orderby wins descending
where wins > 3
select team.TeamName + ": " + wins;
```

Una query può contenere quante clausole let si vuole, prima o dopo la clausola where; inoltre una let può referenziare un'altra variabile di range inizializzata con una let precedente. Una variabile definita con let non deve necessariamente contenere un valore scalare come nel caso precedente. Per esempio, può essere utile per contenere un'altra sequenza di elementi:

```
Var query = from team in teams
let wins = team.Wins
let pilots= team.Pilots
from pilot in pilots
where pilot.Points>80
orderby wins descending
select team.TeamName +" wins: "+wins+ ", leader " + pilot.LastName+" "+pilot.Points ;
```

Nell'esempio precedente, la seconda clausola let contiene la collezione di piloti di ogni team, ma solo quelli con un punteggio superiore a 80. In tal modo, come risultato finale stampa solo i nomi dei team con un pilota del genere e il relativo cognome.

Tipi di proiezioni

La *proiezione* degli elementi di una sequenza consente di enumerare gli elementi e trasformarli o mapparli in una sequenza di altri elementi, eventualmente di tipo differente.

Nel paragrafo precedente, per esempio, abbiamo già visto come ricavare una sequenza formata da stringhe, ricavate da una proprietà degli elementi, utilizzando l'operatore Select o la corrispondente parola chiave select. L'operazione di proiezione permette però di creare nuovi tipi di dati, oppure di trasformare quelli originali, direttamente in una query, basandone le proprietà sugli elementi di una sequenza di origine.

La seguente query consente di ricavare il nome di tutti i team della sorgente dati di esempio, trasformando la proprietà TeamName in maniera che essi appaiano in maiuscolo:

```
var query = from team in teams
select team.TeamName.ToUpper();
```

Oppure usando il metodo Select nel seguente modo:

```
query = teams.Select(team => team.TeamName.ToUpper());
```

Il metodo Select proietta quindi ogni elemento in un altro di formato differente, applicandovi in questo caso il metodo ToUpper di String.

In LINQ, però, le proiezioni consentono di creare al volo nuovi oggetti per memorizzare all'interno del loro stato i risultati della query stessa.

La seguente query enumera gli elementi e per ognuno di essi crea un oggetto Tuple<string,int> formato dal nome della squadra e dal numero di vittorie:

```
var query = from team in teams
select new Tuple<string, int>(team.TeamName, team.Wins);
```

Il risultato dell'esecuzione della query sarà quindi un'enumerazione di tuple, le quali possono essere utilizzate in un normale foreach, ricavandone le proprietà:

```
foreach (Tuple<string, int> t in query)
Console.WriteLine("{0}: {1}", t.Item1, t.Item2);
```

Inizializzatori di oggetto

Una seconda strategia di proiezione sfrutta gli *inizializzatori di oggetto*, che permettono di istanziare un oggetto impostando campi e proprietà pubbliche (vedere il Capitolo 6), senza necessità di invocare un costruttore:

```
var query3 = from team in teams
select new F1Team { TeamName= team.TeamName, Wins= team.Wins };
```

Gli oggetti F1Team che faranno parte della sequenza di risultati vengono inizializzati mediante i valori ricavati dalla variabile team.

Tipi anonimi

L'utilizzo di un tipo esatto e noto a priori non è però necessario. LINQ consente di sfruttare il concetto di *tipo anonimo*. Infatti, la clausola select consente di creare una query che restituisce oggetti anonimi, formandoli dinamicamente:

```
var query = from team in teams
select new {team.TeamName, team.Wins};

foreach (var item in query)
Console.WriteLine(("{0}: {1}", item.TeamName, item.Wins);
```

In questo caso, è obbligatorio la dichiarazione della query come variabile di tipo implicita, in quanto non vi è modo di conoscere il tipo anonimo a tempo di compilazione. Si noti, inoltre, come nel foreach la variabile di iterazione possa utilizzare direttamente le proprietà del tipo anonimo. Il vantaggio principale nell'utilizzo dei tipi anonimi nelle proiezioni è quello di poter ricavare solo le proprietà che saranno necessarie nelle successive elaborazioni, che nel caso di servizi o database remoti possono ottimizzare il traffico dati.

Com'è naturale, la query precedente è realizzabile in termini di metodi di estensione, utilizzando la sintassi di creazione del tipo anonimo all'interno di un delegate o espressione lambda:

```
var query = teams.Select(team=> new { team.TeamName, team.Wins });
```

Un ulteriore operatore di proiezione consente di replicare le query contenenti delle sotto-query, ottenute qualche paragrafo fa mediante clausole from composte.

Il compilatore C# traduce infatti le query con clausole from composte in chiamate all'operatore SelectMany:

```
var query5 = teams.SelectMany(team => team.Pilots.Where(pilot=>pilot.LastName.Length>7),
(t,pilot)=> pilot.LastName);
```

Il metodo SelectMany consente di enumerare una sequenza a partire da ogni elemento di un'altra sequenza, e quindi di ottenere un'unica collezione dall'unione delle due. Inoltre, è possibile invocare una funzione di selezione o trasformazione da applicare a ogni elemento della sequenza ottenuta.

L'esempio di cui sopra utilizza una prima espressione per ricavare per ogni F1Team la relativa collezione Pilots, filtrata per ottenere solo gli oggetti che hanno LastName di lunghezza almeno pari a 8. La seconda espressione, invece, seleziona la proprietà LastName, in modo tale che la sequenza finale sia un'enumerazione di stringhe contenente i cognomi dei piloti.

Filtrare i dati

Lo scopo principale di un'interrogazione su una sorgente dati è principalmente quello di ricavare dei risultati restringendo il numero di elementi della sorgente in base a dei criteri di ricerca o filtri.

Per mezzo della clausola where si possono costruire delle espressioni condizionali di tipo booleano, che verranno usate per filtrare i dati della sequenza originale. Per esempio, per ottenere solo team di Formula 1 che hanno ottenuto almeno cinque vittorie, e con il nome che inizia con la lettera M, possiamo scrivere la seguente query:

```
var query = from team in teams
where team.Wins > 0 &&
team.TeamName.ToLower().StartsWith("m")
select team;
```

La clausola where è detta un *operatore di restrizione*, o *filtro*, in LINQ perché restringe i risultati della query.

Non tutti i filtri possono essere espressi mediante sintassi di query, quindi in casi più avanzati bisognerà necessariamente ricorrere ai metodi di estensione che permettono di usare delegate

ed espressioni lambda per creare filtri complessi. La query precedente, utilizzando l'operatore standard Where, può essere scritta così:

```
query = teams.Where(team => team.Wins > 0 &&
team.TeamName.ToLower().StartsWith("m")
);
```

Un tipo di filtro impossibile usando la sintassi semplificata di C# è quello che prevede un secondo parametro che rappresenta l'indice dell'elemento della sequenza. In questo caso, è possibile utilizzare il secondo overload del metodo Where. Per esempio, per applicare una condizione solo agli elementi di indice pari di una sequenza, è possibile riscrivere la query precedente così:

```
query = teams.Where( (team, index) => team.Wins > 0 &&
team.TeamName.ToLower().StartsWith("m") &&
index%2==0
);
```

Ordinare i dati

In generale, i risultati ottenuti da una query LINQ vengono restituiti senza un ordine particolare. Tuttavia, utilizzando la clausola orderby, eventualmente con un operatore che specifichi se l'ordinamento debba essere crescente (valore predefinito) o decrescente, è possibile ottenere una sequenza di risultati ordinata:

```
var query = from team in teams
orderby team.TeamName
select team.TeamName;
```

La clausola orderby viene seguita dall'elemento sul quale si vuol basare l'ordinamento: in questo caso, i team saranno ordinati per nome, in maniera alfabetica crescente.

Naturalmente, è possibile applicare la clausola orderby in combinazione con la clausola where, in maniera da ordinare i risultati filtrati:

```
//ottiene i team con almeno 3 vittore, ordinandoli per nome
var query = from team in teams
where team.Wins>=3
orderby team.TeamName
select team.TeamName;
```

Il comportamento predefinito è quello crescente, ma è possibile specificare il tipo di ordinamento come parametro della clausola orderby, indicando uno dei due valori ascending e descending, rispettivamente per l'ordinamento crescente e per quello decrescente:

```
orderby team.TeamName descending
```

La chiave su cui basare l'ordinamento può essere personalizzata a seconda delle proprie esigenze: non si è obbligati a utilizzare una proprietà degli elementi, ma una qualunque

funzione per estrarre una chiave ordinabile. Per esempio, per ordinare i team in maniera decrescente a seconda della lunghezza del nome si può utilizzare la clausola seguente, che usa come chiave la proprietà Length della stringa TeamName:

```
orderby team.TeamName.Length descending
```

È inoltre possibile specificare più chiavi di ordinamento, separandole con la virgola, in maniera da ottenere un risultato del tipo "ordina per… e quindi per…". Per esempio, per ordinare i nostri team per numero di vittorie in maniera decrescente e poi per nome in maniera crescente, ottenendo una sorta di classifica, possiamo scrivere:

```
var query2 = from team in teams
orderby team.Wins descending, team.TeamName
select new { team.Wins, team.TeamName };
```

La clausola orderby viene risolta dal compilatore in una chiamata all'operatore di ordinamento OrderBy, mentre la orderby descending in una chiamata all'operatore OrderByDescending:

```
var query3 = teams.OrderBy(team => team.TeamName);
```

Se è necessario specificare più chiavi di ordinamento, le chiamate ai metodi possono essere concatenate con quelle a ThenBy e ThenByDescending:

```
var query4 = teams.OrderByDescending(team => team.Wins).ThenBy(team => team.TeamName);
```

Fra i metodi di estensione che non hanno un corrispondente nella sintassi di query, vi è il metodo Reverse, il quale consente molto semplicemente di ottenere i risultati di una query in ordine invertito rispetto a quello originale. Per esempio, i risultati contenuti nella precedente query4:

```
var reverse= (from team in teams
where team.Wins>0
select team.TeamName).Reverse();
```

Raggruppare i dati

La clausola group… by serve a raggruppare gli elementi di una sequenza secondo una particolare chiave:

```
var query = from team in teams
from pilot in team.Pilots
group pilot by team;
```

La query inizia con due clausole from per ottenere tutti i piloti di tutte le squadre; quindi, per mezzo di una clausola group, essi vengono raggruppati per squadra.

```
foreach (var gr in query)
{
Console.WriteLine(gr.Key);
foreach(var pilot in gr)
{
```

```
Console.WriteLine(" - {0} {1}", pilot.FirstName, pilot.LastName);
}
}
```

Ogni gruppo è un oggetto che implementa l'interfaccia IGrouping<K,V>, con il parametro di tipo K che rappresenta la chiave, mentre il secondo parametro è il tipo degli elementi contenuti nella collezione di oggetti che hanno in comune tale chiave. Pertanto, nel nostro caso, la query restituisce un oggetto IGrouping<F1Team, Pilot> e il foreach potrebbe essere riscritto esplicitando il tipo così:

```
foreach (IGrouping<F1Team,Pilot> group in query)
{
F1Team team = group.Key;
Console.WriteLine(team);
foreach (Pilot pilot in group)
{
Console.WriteLine(" - {0} {1}", pilot.FirstName, pilot.LastName);
}
}
```

Per ogni gruppo, quindi, prima viene ottenuta la chiave di tipo F1Team mediante la proprietà Key, e poi viene enumerata la collezione di piloti contenuta in ogni gruppo.

Continuazione di query

Una query deve terminare con una clausola select o con una group, ma il risultato di una query può anche essere passato a una successiva mediante la clausola into. Quest'ultima segnala infatti la continuazione di una query, salvando il risultato ottenuto mediante una select o una group (o join). Sul risultato ottenuto possono a questo punto essere applicate nuove where, orderby, select, group, creando praticamente una nuova query. Per esempio:

```
var query = from team in teams
where team.Wins>2
select team.Pilots
into topTeamPilots
from tp in topTeamPilots
where tp.Points>50
orderby tp.Points descending
select tp.LastName+" "+tp.FirstName+": "+tp.Points;
```

La prima parte dell'espressione ricava i piloti dei team con almeno due vittorie. Da tale risultato, memorizzato in topTeamPilots, viene creata una nuova espressione che filtra i piloti con almeno 50 punti, li ordina in maniera decrescente e infine crea una proiezione di cognome, nome e punteggio.

Operazioni di join

Un'unione di sequenze in LINQ crea una sequenza sola a partire da due collezioni, accoppiando un elemento della prima con uno della seconda in base a una proprietà comune.

Per creare un'unione si utilizza, in syntax query, la clausola join.

NOTA	L'operazione join di LINQ è concettualmente simile a quella di SQL, in cui essa permette di unire i dati di una tabella con quelli di un'altra in base a dei campi in comune.

La sintassi generale della clausola join è la seguente:

join oggettoInterno **in** collezioneInterna **on** chiaveEsterna **equals** chiaveInterna

Come vedremo dagli esempi, la posizione degli operandi è importante; spiega anche perché non viene utilizzato un operatore == ma un apposito equals per accoppiare la chiaveEsterna con la chiaveInterna.

Con equals la chiaveEsterna, alla sua sinistra, si riferisce e consuma una sequenza esterna, mentre la chiave alla sua destra si riferisce alla collezioneInterna. Fra l'altro, provando a invertire le due chiavi, si avrebbe un errore di compilazione, perché il loro ambito è limitato proprio alle posizioni suddette.

Nel nostro modello dati di esempio, ogni istanza di Pilot è correlata a una istanza di Country mediante una proprietà IDCountry presente in entrambe le classi. Quindi, possiamo per esempio scrivere una query di join per ricavare per ogni pilota il nome della relativa nazione di appartenenza:

```
var query = from team in teams
from pilot in team.Pilots
join country in Country.All on pilot.IDCountry equals country.IDCountry
select new {pilot.LastName, CountryName=country.Name);
```

È da notare che, rispettando la sintassi generale, la chiave pilot.IDCountry si riferisce alla collezione esterna, mentre la proprietà country.IDCountry appartiene agli oggetti della collezione Country.All.

Stampiamo il risultato, con i dati di esempio finora usati, mediante un foreach:

```
foreach (var p in query)
{
Console.WriteLine("{0} ({1})", p.LastName, p.CountryName);
}
```

Si otterrà qualcosa del genere:

```
Alonso (Spagna)
Massa (Brasile)
Vettel (Germania)
Webber (Australia)
Button (Regno Unito)
Perez (Messico)
Raikkonen (Finlandia)
```

Grosjean (Svizzera)
Rosberg (Germania)
Hamilton (Regno Unito)

Si noti che le nazioni Italia e USA non appaiono, in quanto nessun pilota è associato a esse.

NOTA	In termini di database, il risultato che si ottiene con una clausola join corrisponde a un'istruzione *inner join*, con la quale vengono restituiti solo gli oggetti di una collezione che hanno un corrispondente nella seconda.

Con gli operatori standard, la query precedente può essere scritta per mezzo del metodo Join:

```
var query2 = teams.SelectMany(t => t.Pilots).Join( //collezioneEsterna
Country.All, //collezioneInterna
pilot => pilot.IDCountry, //chiaveEsterna
country => country.IDCountry, //chiaveInterna
(p, c) => new { p.LastName, CountryName = c.Name } //risultato
);
```

I commenti indicano le corrispondenze con le collezioni e le chiavi come definite per la clausola join.

Il metodo GroupJoin

Un secondo tipo di join può essere effettuato mediante il secondo utilizzo previsto per la clausola into, estendendo la sintassi della clausola join così come segue:

join oggettoInterno **in** collezioneInterna **on** chiaveEsterna **equals** chiaveInterna **into**

Stavolta, per comodità, la query di esempio sarà spezzata in due, in maniera da avere un elenco di piloti restituito dalla prima, da utilizzare nella vera e propria join:

```
var pilots = from team in teams
from pilot in team.Pilots
select pilot;

var query3 = from country in Country.All
join pilot in pilots on country.IDCountry equals pilot.IDCountry
into pilotsxCountry
select new { CountryName=country.Name, Pilots = pilotsxCountry };
```

Tale operazione emette una sequenza di risultati gerarchica, a differenza della join vista prima che ha restituito un risultato piatto. In questo caso, invece, si ottiene un risultato simile a quello visto con la clausola group by, quindi una collezione di nomi di nazioni, a ogni elemento della quale corrisponde una collezione di piloti. La si può stampare come segue:

```
foreach (var group in query3)
{
Console.WriteLine(group.CountryName);
foreach(Pilot pilot in group.pilots)
{
Console.WriteLine(" {0} {1}", pilot.LastName, pilot.FirstName);
```

```
}
}
```

NOTA	In termini di database, stavolta, non c'è una vera e propria corrispondenza: questo tipo di join, infatti, restituisce i risultati di una *left outer join*, sovrapposti a quelli della inner join.

In questo caso, la collezione delle nazioni verrà enumerata anche se non vi sono elementi delle collezioni di piloti corrispondenti. E infatti, fra i risultati appariranno anche Italia e USA, che nel modello dati di esempio non hanno piloti corrispondenti. Naturalmente, sarebbe possibile modificare la query per escludere le nazioni senza piloti, ma questo ve lo lasciamo per esercizio (basta una clausola where!).

Questa seconda join verrà tradotta dal compilatore per mezzo del metodo GroupJoin, che possiamo usare per replicare l'espressione di query precedente nel seguente modo:

```
IEnumerable<Pilot> pilots=teams.SelectMany(t=>t.Pilots);

var query4 = Country.All.GroupJoin(
pilots, //collezione interna
country => country.IDCountry, //chiave esterna
pilot => pilot.IDCountry, //chiave interna
(c, p) => new { CountryName=c.Name, Pilots = p }); //risultato
```

La sequenza esterna è costituita dalla collezione di Country sulla quale viene invocato il metodo GroupJoin.

Chiavi complesse

Negli esempi di join precedenti, come chiavi di confronto per accoppiare gli elementi delle collezioni sono state utilizzate sempre semplici proprietà. Se si avesse però la necessità di eseguire confronti su chiavi multiple, per esempio su due proprietà di uno stesso oggetto, sarebbe possibile ancora una volta servirsi dei tipi anonimi.

Supponiamo che una classe Classe1 abbia le proprietà ID1 e Name1, mentre una classe Classe2 esponga le proprietà ID2 e Name2. Per eseguire una join in maniera da accoppiare gli elementi che abbiano ID1=ID2 e Name1=Name2, si potrà scrivere una query così:

```
var query = from a in collezione1
join b in collezione2 on
new {a.ID1, a.Name1}
equals
new {b.ID2, b.Name2}
…
```

Naturalmente, perché ciò sia possibile, i tipi delle proprietà corrispondenti devono essere identici.

Operazioni sugli insiemi

LINQ fornisce degli operatori che permettono di trattare le sequenze come fossero insiemi di elementi, in maniera da restituire dei risultati basati sull'assenza o sulla presenza di elementi uguali in due sequenze diverse. Essi non hanno clausole corrispondenti nella sintassi di query.

Per tali operatori useremo due array numerici come dati di esempio, in maniera da rendere più semplice la comprensione dei meccanismi di funzionamento:

```
int[] array1={1,2,3,4,5,3,4,5};
int[] array2={1,3,4,6,7,8};
```

L'operatore Distinct lavora su un solo insieme e permette di selezionare occorrenze uniche di elementi, restituendo quindi una sola volta elementi eventualmente presenti in più copie:

```
var queryDistinct = array1.Distinct(); //risultato 1,2,3,4,5
```

Distinct risulta comodo per ottenere risultati unici da altre query più complesse che possono contenere duplicati:

```
var pilots = (from team in teams
from pilot in team.Pilots
select pilot.FirstName).Distinct();
```

L'operatore di unione Union unisce le due sequenze, eliminando eventuali elementi duplicati.

```
var union = array1.Union(array2); //1,2,3,4,5,6,7,8
```

L'operatore Except esegue la differenza fra due insiemi, restituendo quindi gli elementi che fanno parte del primo insieme ma non del secondo. Di conseguenza, il risultato di array1.Except(array2) sarà differente da quello di array2.Except(Array1):

```
var except=array1.Except(Array2); // 2,5
```

L'operatore Intersect, infine, esegue l'intersezione fra i due insiemi, restituendo solo gli elementi che fanno parte di entrambi:

```
var except=array1.Intersect(Array2); // 2,5
```

NOTA	Tutti gli operatori sugli insiemi utilizzano i metodi GetHashCode e Equals per stabilire se due elementi sono uguali o meno. Ogni operatore ha però degli overload che consentono di indicare come parametro un oggetto IEqualityComparer da usare per confrontare gli oggetti.

L'operatore Zip non esegue una vera e propria operazione fra insiemi. Esso restituisce un nuovo insieme applicando una funzione a ogni coppia di elementi presi dai due insiemi originali. Per esempio, a partire dagli array1 e array2 si può ottenere un arraySomma, in cui ogni elemento è la somma dei due elementi:

```
var zip=array1.Zip(array2, (a,b) => a+b);
```

Se le sequenze non dispongono dello stesso numero di elementi, il metodo unisce le sequenze finché non raggiunge la fine di una. Nell'esempio, array1 dispone di otto elementi, mentre array2 di sei: la sequenza risultante disporrà quindi di soli sei elementi, saltando gli ultimi due di array1.

Quantificatori

Gli operatori di questa categoria servono a verificare l'esistenza di elementi all'interno di una sequenza, eventualmente che soddisfano una particolare condizione, quindi restituiscono un valore booleano true in caso di successo.

L'operatore Contains verifica se la sequenza contiene un particolare elemento:

```
int[] array={1,2,3,4};
bool b=array.Contains(1); //true
```

L'operatore All verifica se tutti gli elementi di una sequenza soddisfano una condizione:

```
bool b=array.All(i=>i>0); //true se tutti positivi
```

L'uso di All, così come degli altri operatori, è particolarmente vantaggioso combinandolo con gli altri metodi per creare delle query più articolate. Per esempio, la seguente espressione Where utilizza un operatore All per verificare se tutti i piloti hanno un punteggio positivo, quindi nel complesso la query restituirà solo i team che hanno tutti i piloti a punti:

```
var query = teams.Where(t => t.Pilots.All(p => p.Points > 0));
```

L'operatore Any ha un funzionamento identico, ma verifica che almeno un elemento soddisfi la condizione.

```
var query = teams.Where(t => t.Pilots.Any(p => p.Points > 100));
```

La precedente query restituisce i team che hanno almeno un pilota con più di 100 punti. Si potrebbe anche utilizzare il metodo Count, verificando che esso restituisca un numero maggiore di zero, ma in questi casi è consigliato l'uso di Any: si ferma al primo elemento che incontra, senza dover verificare tutta la sequenza.

Partizionare dei dati

Il termine partizionamento in LINQ si riferisce alla possibilità di restituire una sotto-sequenza a partire da una iniziale, selezionando o saltando determinati elementi.

Anche qui utilizzeremo un array per mostrare il funzionamento degli operatori:

```
int[] array= {1,1,2,3,1,2, 1,3};
```

L'operatore Skip salta un certo numero di elementi specificato e restituisce una sequenza costituita da quelli rimanenti:

```
var query= array.Skip(4); //1,2,1,3
```

SkipWhile, invece, salta degli elementi fino a quando una determinata condizione rimane vera e restituisce il resto della sequenza:

```
var query= array.SkipWhile(n=>n<3); //3, 1,2,1,3
```

La precedente istruzione salta gli elementi dell'array fino a quando essi sono minori di tre, quindi restituisce la sotto-sequenza rimanente.

L'operatore Take, al contrario, restituisce i primi n elementi di una sequenza:

```
var query= array.Take(4); //1,1,2,3
```

TakeWhile, invece, restituisce i primi elementi che soddisfano una determinata condizione e salta tutto il resto:

```
var query= array.TakeWhile(n=> n<3); //1,1,2
```

Operazioni sugli elementi

Una serie di operatori consente di recuperare uno specifico elemento da una sequenza. Se si conosce l'indice dell'elemento da recuperare si può intanto utilizzare il metodo ElementAt:

```
F1Team team=teams.ElementAt[0];
```

Gli operatori First e Last consentono di specificare una condizione opzionale per recuperare, rispettivamente, il primo e l'ultimo elemento della sequenza.

L'operatore Single, invece, restituisce l'unico elemento della sequenza se non si specifica alcun parametro, oppure l'unico elemento che soddisfi la condizione specificata. Quindi, se si prevede di dover utilizzare Single, bisogna essere certi che la sequenza contenga uno e un solo elemento da restituire.

I tre operatori precedenti, nel caso in cui la sequenza non contenga alcun elemento oppure non ne restituisca nessuno che soddisfi la condizione, generano un'eccezione. Se invece si vuol restituire in tali casi un valore di default, sono disponibili degli operatori analoghi il cui nome termina con "OrDefault": ElementAtOrDefault, FirstOrDefault, LastOrDefault, SingleOrDefault.

Il valore di default restituito è default(T), cioè il valore predefinito del tipo di elementi della collezione. Nel caso di elementi di tipo riferimento sarà quindi null, false per il bool, e 0 per i tipi numerici.

Operatori di aggregazione

In molti casi pratici, può essere utile eseguire dei calcoli su sequenze di oggetti, e LINQ fornisce gli operatori per le più comuni operazioni di aggregazione.

La maggior parte di tali operatori sono immediati da utilizzare. Abbiamo già incontrato l'operatore Count, che non fa altro che effettuare un conteggio degli elementi di una sequenza: tutti se non si usa alcun parametro, altrimenti solo quelli che soddisfano una condizione. Per esempio, per ottenere il numero di squadre con almeno una vittoria, si può invocare il metodo Count come segue:

```
int countTeam= teams.Count(t => t.Wins > 0);
```

La variante LongCount, è utilizzabile su sequenze che contengono, e possono quindi restituire, un numero di elementi molto grande; infatti restituisce un long anziché un int.

Tutti gli operatori di aggregazione risultano di notevole utilità in combinazione con altri operatori o all'interno di altre query per eseguire operazioni di aggregazione sul risultato delle query stesse.

L'operatore Sum, per esempio, che serve a effettuare la somma dei valori di una sequenza, ha diverse varianti, distinte intanto per il tipo numerico su cui agiscono. Se abbiamo un array di interi, è possibile eseguirne la somma semplicemente così:

```
int[] array={1,2,3,4,5};
int somma=array.Sum();
```

Questa implementazione esegue la somma assumendo che tutti gli elementi di una collezione siano dello stesso tipo numerico.

Se la sequenza non è invece composta da elementi di un tipo numerico, bisogna estrarre da essi un valore da poter sommare. Per farlo, si usa un altro overload del metodo Sum che prevede un selettore come secondo parametro. Il seguente esempio mostra l'uso di questa versione di Sum:

```
var query = from team in teams
let points= team.Pilots.Sum(p => p.Points)
orderby points descending
select new { team.TeamName, points};
```

La query restituisce, per ogni team, la somma dei punti dei relativi piloti, utilizzando la proprietà Points come selettore numerico, e salvandola nella variabile points creata con la parola chiave let; inoltre gli elementi sono ordinati in maniera decrescente in base a tali somme. Quindi, ciò che si ottiene è in pratica la classifica costruttori del campionato di Formula 1:

```
TeamName = Red Bull, points = 189
TeamName = Ferrari, points = 162
TeamName = Mercedes Petronas, points = 161
TeamName = Lotus, points = 141
TeamName = McLaren, points = 118
```

Gli altri operatori di aggregazione hanno una modalità di utilizzo analoga.

L'operatore Average calcola la media dei valori numerici che costituiscono una sequenza, oppure estratti da una sequenza:

```
var queryAverage = from team in teams
select new { team.TeamName, AvgPoints = team.Pilots.Average(p => p.Points) };
```

Gli operatori Min e Max restituiscono, rispettivamente, il minimo e il massimo di una sequenza:

```
int max = (from team in teams
from pilot in team.Pilots
select pilot.Points).Max();
```

Anche Min e Max possono utilizzare un secondo parametro per indicare un selettore:

```
int min = (from team in teams
from pilot in team.Pilots
select pilot).Min(p=>p.Points);
```

Il valore minimo o quello massimo vengono calcolati confrontando gli elementi che formano la sequenza e che per essere dunque confrontati devono implementare l'interfaccia IComparable o IComparable<T>.

L'ultimo operatore di aggregazione è Aggregate, che consente di specificare un'operazione personalizzata di accumulazione. Anch'esso possiede diversi overload: per l'esattezza tre.

Il seguente esempio mostra come unire le iniziali delle stringhe contenute in un array, formando un acronimo a partire da esse (naturalmente non è necessario usare Aggregate, ci sono modi molto più semplici per ottenere lo stesso risultato):

```
string frase = "senatus populus que romanus";
var acronimo = frase.Split(' ').Aggregate("", (result, word) => result + word.ToUpper()[0] + ".");
```

Il metodo Aggregate, in questo overload, accetta un valore iniziale (la stringa vuota, in questo caso) e accumula, concatenandoli al valore precedente, il primo carattere (convertito in maiuscolo) di ogni parola contenuta nella frase e un carattere punto. Il risultato finale, ottenuto eseguendo il codice di cui sopra, sarà quindi la stringa "S.P.Q.R".

Un altro esempio di Aggregate mostra come ottenere per ogni team la somma dei punti di ogni pilota di un team (per lo stesso risultato sarebbe sufficiente l'operatore Sum):

```
var queryAggregate = from team in teams
select new { Points = team.Pilots.Aggregate(0, (total, p) => total + p.Points) };
```

Conversioni di tipo

Sebbene LINQ lavori su sequenze generiche, di tipo IEnumerable/IEnumerable<T> o IQueryable/IQueryable<T>, in alcuni casi è possibile utilizzare dei metodi per convertire da e verso altri tipi. Più che per cambiare tipo, gli operatori di tale tipologia servono per gestire

problemi ed esigenze legate all'esecuzione differita delle query LINQ. ToArray, ToList, ToDictionary, ToLookup causano infatti l'esecuzione immediata della query restituendo il risultato nel tipo di collezione intuibile dal nome dei metodi.

Per esempio, se si vuol ottenere una List<F1Team> contenente i team il cui nome inizia per M, si può applicare l'operatore ToList a questa query:

```
List<F1Team> result= teams.Where(t=>t.TeamName.ToUpper().StartsWith("M")).ToList();
```

Analogamente, il metodo ToArray permette di ottenere un array a partire dagli elementi di una sequenza:

```
Pilot[] array = teams.SelectMany(t => t.Pilots).ToArray();
```

L'operatore ToDictionary crea un'istanza della classe Dictionary<K,V> e permette di specificare mediante un primo delegate la chiave del dizionario, e mediante un secondo, opzionale, il valore corrispondente alla chiave prima ricavata. Il seguente esempio crea un dizionario che usa i nomi dei team come chiave e un array di piloti della squadra come valore corrispondente alla chiave stessa:

```
Dictionary<string, Pilot[]> dict = (from team in teams
where team.Wins > 0
select team).ToDictionary(t => t.TeamName, t => t.Pilots);

foreach (string key in dict.Keys)
{
Console.WriteLine(key);
foreach (Pilot p in dict[key])
{
Console.WriteLine("- {0}: {1}",p.LastName, p.Points);
}
}
```

In modo analogo, il metodo ToLookup crea un'enumerazione di tipo Lookup<K,V>: una sorta di dizionario costituito da una sequenza di elementi chiave, a ognuno dei quali corrispondono altre sequenze di elementi. L'esempio precedente può essere facilmente replicato con il metodo ToLookup come segue:

```
ILookup<F1Team, Pilot[]> lookup = (from team in teams
where team.Wins > 0
select team).ToLookup(t => t, t => t.Pilots);
```

L'operatore AsEnumerable restituisce semplicemente la sequenza iniziale sotto forma di un oggetto IEnumerable<T>. In tal modo viene eseguita una conversione al volo, consentendo per esempio di utilizzare i metodi di estensione di IEnumerable<T> anche se la sequenza originale implementa dei metodi personalizzati.

Gli ultimi due operatori di questa categoria sono OfType e Cast.

Il primo, OfType, filtra la sequenza cui è applicato restituendo solo gli elementi di un dato tipo, il che è utile se si hanno sequenze di oggetti di tipo differente:

```
List<object> lista = new List<object>() { "a", 1, "b", 2, "c" };
var stringhe= lista.OfType<string>();
```

Se nessun elemento è compatibile con il tipo ricercato, verrà restituita un'enumerazione vuota.

L'operatore Cast, invece, permette di convertire un'enumerazione di elementi di un dato tipo in uno differente, sempre che la conversione dal primo al secondo sia possibile.

Nel seguente esempio, la lista di stringhe viene convertita in un'enumerazione di object:

```
List<string> lista=new List<string> {"a", "b", "c" };
var objects = lista.Cast<object>();
```

NOTA	Gli operatori Cast e OfType restituiscono sequenze di oggetti che sono riferimenti a quelli delle sequenze originali e non delle copie. Quindi essi valutano la sequenza originale ogni volta che si enumera il loro risultato, a differenza degli altri operatori di conversione che causano un'esecuzione immediata della query.

Operatori di generazione

Gli operatori appartenenti a questa categoria permettono di creare nuove sequenze, generandone gli elementi, che potranno essere utilizzate in altre query applicandovi altri operatori o algoritmi di calcolo.

Il metodo Range della classe Enumerable consente di creare un determinato intervallo di numeri interi, indicando il valore iniziale e il numero di elementi.

Il seguente esempio genera e stampa il nome dei dodici mesi dell'anno:

```
var months = Enumerable.Range(1, 12).Select(n => CultureInfo.CurrentCulture.DateTimeFormat.GetMonthName(n));
months.ToList().ForEach(Console.WriteLine);
```

L'enumerazione di numeri dall'1 al 12 viene usata per invocare il metodo di proiezione Select all'interno del quale, per ogni numero, viene ricavato il nome del mese.

Il metodo Repeat crea un'enumerazione di un certo numero di copie dello stesso elemento. Nel caso di un tipo riferimento, ogni elemento è un riferimento allo stesso oggetto:

```
int[] array= Enumerable.Repeat<int>(1, 10).ToArray();
```

L'esempio precedente crea un'enumerazione di 10 interi il cui valore è impostato pari a 1. Dato che il tipo di oggetti da ripetere può essere uno qualunque, tale operatore è utile per ripetere un dato numero di volte la stessa query, passata come oggetto da replicare:

```
var expr=Enumerable.Repeat(from team in teams
select team.TeamName), 10);
```

In questo caso, il metodo genera 10 sequenze contenenti tutte i nomi delle squadre, cioè una sequenza di sequenze.

Il metodo Empty restituisce una sequenza di oggetti di un dato tipo vuota, operazione utile per inizializzare delle sequenze:

```
IEnumerable<Pilot> pilots=Enumerable.Empty<Pilot>();
```

Infine, il metodo DefaultIfEmpty restituisce gli elementi di una sequenza esistente oppure, se essa è vuota, un valore predefinito.

```
foreach(var pilot in pilots.DefaultIfEmpty(new Pilot(){ LastName="Sconosciuto"}))
{
Console.WriteLine(pilot.LastName);
}
```

Nell'esempio precedente, se la collezione pilots è vuota, il ciclo viene eseguito su un unico oggetto Pilot predefinito, costruito all'interno del metodo DefaultIfEmpty.

Altri operatori

Concludiamo la panoramica degli operatori standard con l'operatore di concatenazione e quello di uguaglianza.

L'operatore Concat permette di ottenere un'unica sequenza a partire da due sequenze originali, concatenandole una all'altra:

```
var s1 = Enumerable.Range(1, 5);
var s2 = Enumerable.Range(6, 5);
var concat = s1.Concat(s2); //1 … 10
```

L'operatore SequenceEqual permette di verificare che due sequenze siano uguali, confrontando ogni coppia di elementi corrispondenti:

```
bool b= s1.SequenceEqual(s2);
```

Se due sequenze hanno lo stesso numero di elementi e gli elementi in una stessa posizione sono uguali, allora le sequenze sono considerate uguali.

Anche qui, si tenga presente che il confronto viene fatto utilizzando i metodi GetHashCode e Equals. Oppure ci si può servire di un secondo overload del metodo SequenceEqual, il quale usa un parametro di tipo IEqualityComparer.

Domande di riepilogo

1) Quale namespace è necessario perché possa essere eseguita una query LINQ, per esempio Where, su una collezione?

a. System.Data
b. System
c. System.Linq
d. Nessuno, Where è un metodo di ICollection

2) Dato un array di interi, la seguente istruzione con query LINQ:

```
var query= from n in array
where n > 0
select n-1;
```

è equivalente a:

a. var query = array.Select(n => n-1);
b. var query = array.ToList(n => if(n>0) n-1);
c. var query = array.Where(n => n - 1).ToList();
d. var query = array.Where(n => n>0).Select(i => i-1);

3) La query syntax e la method syntax di LINQ non possono essere usate contemporaneamente, in modo misto. Vero o falso?

4) Dato l'array di stringhe string[] giorni={"lun","mar","mer","gio","ven","sab","dom"}, qual è il valore di count dopo aver eseguito il codice seguente?

```
var query=giorni.Where(g=>g.Contains("m"));
query=query.Where(g=>g.StartsWith("m"));
int count=query.Count();
```

a. 1
b. 2
c. 7
d. Eccezione a runtime

5) Dato l'array di stringhe string[] giorni={"lun","mar","mer","gio","ven","sab",

"dom"}, e il codice seguente, che tipo assume query?

```
query=from g in giorni
where g.Contains("e")
select g;
```

a. string

b. object
c. IQueryable<string>
d. IEnumerable<string>

6) Quante istruzioni let può contenere una query LINQ?

a. Una
b. Una per ogni istruzione select
c. Una per ogni istruzione where
d. Non ci sono limiti

7) Per restituire dei dati complessi da una query LINQ, si utilizza in genere:

a. un tipo anonimo
b. un tipo object
c. un tipo generic
d. LINQ lavora solo su tipi semplici

8) Dato un array di stringhe string[] giorni, e il codice seguente, quante volte viene eseguita l'istruzione Max?

```
var query=from g in giorni
where g.Length== giorni.Max(g.Length)
select g;
```

a. Una sola volta
b. Una per ogni elemento dell'array
c. Dipende dalla lunghezza di ogni elemento
d. Nessuna

Multithreading, programmazione asincrona e parallela

Le attività da gestire in concorrenza possono essere eseguite su thread diversi, con un processo detto di multi-threading. Per sfruttare la potenza dell'hardware, C# consente di eseguire più thread in parallelo e nascondere le complessità della scrittura di codice asincrono.

Ogni applicazione di una certa complessità necessita di gestire ed eseguire più compiti in contemporanea.

Un'applicazione lenta e che si blocca ad aspettare il completamento di un'operazione finisce per stancare l'utente in breve tempo. Se poi si tratta di un'applicazione con interfaccia grafica, che al clic su un pulsante avvia un compito particolarmente gravoso e non permette di fare nient'altro nell'attesa, ci si ritrova ben presto a ricorrere alla pressione della fatidica combinazione di tasti CTRL-ALT-CANC, nel tentativo di capire perché si è bloccato tutto.

Questo capitolo descriverà come utilizzare diverse tecnologie e pattern di programmazione per migliorare le performance di un'applicazione.

Innanzitutto, si affronterà l'argomento "thread": cosa sono, come vengono creati e come avviarli per eseguire più azioni in parallelo.

L'uso di più thread implica la possibilità di accedere a una stessa risorsa o sezione di codice da più parti contemporaneamente, quindi sarà necessario servirsi di meccanismi di sincronizzazione di tali accessi.

A partire da .NET Framework 4.0, lo sviluppatore ha a disposizione delle astrazioni di livello più alto rispetto ai thread e che sono utilizzabili in maniera più semplice ed efficace: la *Task*

Parallel Library e, in particolare, la classe Task e la classe Parallel saranno oggetto di questo capitolo.

Infine vedremo come, con la principale novità di C# 5.0, cioè le parole chiave async e await, la *programmazione asincrona* sia praticamente integrata nella sintassi del linguaggio: passo fondamentale per le moderne applicazioni mobile e web, che sfruttano risorse sul Web non sempre raggiungibili e dispositivi dalla potenza di calcolo spesso limitata.

Threading

Windows consente a ogni processo di contenere più thread.

Un *thread* è un percorso di esecuzione di un processo all'interno di una stessa applicazione eseguibile. Un'applicazione può anche funzionare perfettamente con un singolo thread, quello primario creato dal sistema operativo all'avvio dell'applicazione stessa, cioè quando si entra nel Main. Tale thread può anche creare e avviare a sua volta thread secondari per eseguire più compiti in parallelo.

Threads

Search: Search Call Stack Group by: Process ID Columns

	ID	Managed ID	Category	Name	Location	Priority
Process ID: 4984 (6 threads)						
	9020	0	Worker Thread	<No Name>	<not available>	Highest
	7392	6	Worker Thread	<No Name>	<not available>	Normal
	6180	7	Worker Thread	vshost.RunParkingWindow	[Managed to Native Transition]	Normal
	9248	9	Worker Thread	.NET SystemEvents	[Managed to Native Transition]	Normal
	4140	10	Main Thread	Primary	Multithreading.Program.Main	Normal
	7836	11	Worker Thread	t1	Multithreading.Program.Main.AnonymousMethod_0	Normal

Program.cs

Multithreading.Program Main(string[] args)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

namespace Multithreading
{
    class Program
    {
        static void Main(string[] args)
        {
            Thread.CurrentThread.Name="Primary";
            Thread t1 = new Thread(new ThreadStart(() =>
            {
                Console.WriteLine(Thread.CurrentThread.Name+" executing");
                                  "t1 executing"
            }
```

Figura 12.1 – Debug di applicazioni multithread in Visual Studio.

Supponiamo, per esempio, di dover leggere il contenuto di un file per poi analizzarlo. Se il file è molto grosso, il processo di lettura potrebbe bloccare la nostra applicazione fino a quando essa non ha terminato di leggerlo. L'ideale sarebbe quindi eseguire la lettura del file in un percorso di esecuzione indipendente, cioè in un thread secondario, in maniera da consentire all'applicazione di continuare a operare. È il CLR, in collaborazione con il sistema operativo, a permettere tale possibilità, cioè il cosiddetto *multithreading*.

Un thread è rappresentato dalla classe Thread del namespace System.Threading. Visto che deve esistere almeno un thread di esecuzione (quello primario), si può ottenere l'istanza del thread corrente per mezzo della proprietà statica di Thread, CurrentThread.

Inoltre, fra le proprietà di Thread, è utile ai fini del debug la proprietà Name, che consente di impostare o ricavare il nome del thread:

```
static void Main()
{
Thread.CurrentThread.Name="Primary";
…
}
```

Tale nome è per esempio visualizzato in Visual Studio all'interno della finestra di debug Threads (se non è visibile si può renderla tale dal menu Debug -> Other Windows Threads).

Per creare un nuovo thread, è necessario creare una nuova istanza della classe Thread e avviarla utilizzando il metodo Start. Per indicare il codice che dovrà essere eseguito da parte del nuovo thread, si usa un delegate ThreadStart in maniera che esso punti al metodo da eseguire all'avvio del thread suddetto.

La firma del delegate ThreadStart è la seguente:

```
public delegate void ThreadStart();
```

Quindi, il codice da eseguire all'interno del nuovo thread sarà un metodo con tipo di ritorno void e senza parametri di input:

```
public void CodiceSecondario()
{
Console.WriteLine(Thread.CurrentThread.Name);
for(int i=0;i<10;i++)
{
Console.WriteLine(i);
Thread.Sleep(100);
}
}
```

Il metodo statico Sleep consente di interrompere il thread attuale per il numero di millisecondi indicato, oppure per l'intervallo di tempo che viene determinato da un oggetto TimeSpan. Nel caso precedente, il thread che esegue il metodo si interromperà per 100 millisecondi a ogni ciclo del for.

Per far partire il nuovo thread, basta invocare il metodo Start sulla nuova istanza di Thread:

```
Thread th1=new Thread(new ThreadStart(CodiceSecondario));
th1.Start();
```

Naturalmente, per evitare di dover dichiarare appositamente un metodo, è possibile utilizzarne uno anonimo:

```
Thread th2 = new Thread(delegate()
{
Console.WriteLine(Thread.CurrentThread.Name);
for(int i=0;i<10;i++)
{
Console.WriteLine(i);
Thread.Sleep(100);
}
});
th2.Start();
```

O ancora un'espressione lambda, la quale consente una sintassi ancora più semplificata per creare un delegate anonimo:

```
Thread th3 = new Thread( () =>
{
Console.WriteLine(Thread.CurrentThread.Name);
for(int i=0;i<10;i++)
{
Console.WriteLine(i);
}
});
th3.Start();
```

Subito dopo l'avvio di un thread, la sua proprietà IsAlive sarà true.

Se si vogliono ottenere maggiori informazioni sullo stato di un thread, è anche possibile verificarlo utilizzando la proprietà ThreadState, che può assumere i valori dell'enumerazione omonima. Un thread in esecuzione avrà per esempio valore Running per tale proprietà.

Sebbene esista un metodo Suspend, esso è deprecato, insieme al corrispondente Resume. Quindi, il controllo di un thread e la sua eventuale terminazione vanno fatti in maniera diversa, per esempio con una variabile booleana che indichi quando terminare un ciclo for come quello del precedente thread:

```
Thread th3 = new Thread( () =>
{
```

```
Console.WriteLine(Thread.CurrentThread.Name);
for(int i=0;i<10;i++)
{
if(mustExit)
break;
Console.WriteLine(i);
}
});
```

La variabile mustExit potrà così essere impostata esternamente al thread, causando l'uscita dal for e quindi la terminazione del thread stesso.

```
th4.Start(hello);
```

È possibile attendere che un thread secondario finisca la sua esecuzione prima di continuare quella del thread che lo ha creato, utilizzando il metodo Join:

```
Thread th=new Thread(CodiceSecondario);
th.Start();
th.Join();
Console.WriteLine("th terminato");
```

Il metodo Join può anche accettare un parametro come valore di timeout in maniera che, qualora il thread non termini prima di tale valore, l'esecuzione del thread prima rimasto bloccato in attesa possa riprendere ugualmente.

Priorità dei thread

La proprietà Priority della classe Thread permette di impostare un livello di priorità assegnabile al thread, che determini quanto tempo di esecuzione il sistema operativo dedicherà a esso, relativamente ad altri thread.

Il tipo della proprietà Priority è un'enumerazione, ThreadPriority, che può assumere uno dei seguenti valori:

- Lowest;
- BelowNormal;
- Normal;
- AboveNormal;
- Highest.

Si faccia attenzione a non assegnare a un thread una priorità molto alta senza motivo, perché ciò avverrebbe a discapito degli altri:

```
Thread th=new Thread(()=>Console.Write("alta priorità"));
th.Priority=ThreadPriority.AboveNormal;
```

Si noti, inoltre, che l'assegnazione di un livello di priorità non è per nulla deterministico: esso influenza semplicemente la quantità di tempo di CPU che sarà dedicata dal sistema operativo

al thread in oggetto, ma la schedulazione vera e propria spetta in ogni caso a quest'ultimo, secondo i propri meccanismi.

Thread di background e foreground

Supponiamo di creare uno o più thread all'interno del nostro metodo Main (che avvia, come ormai ben sappiamo, il thread primario dell'applicazione) e di farli eseguire immediatamente:

```
static void Main()
{
Thread tcount1 = new Thread(()=>
{
for (int j = 0; j < 10; j++)
{
Console.WriteLine("{0}: {1}", Thread.CurrentThread.Name, j);
Thread.Sleep(1000);
}
});
tcount1.Start();
}
```

Sebbene il metodo Main termini subito dopo l'avvio del thread, l'applicazione non conclude la sua esecuzione come di solito. Il thread secondario, infatti, è ancora in esecuzione per terminare il suo compito (che durerà almeno 10 secondi, dovendo fare dieci cicli con pausa di un secondo a ognuno di essi).

In realtà, tale comportamento è vero se i thread secondari sono dei thread di *foreground*. Di default, tutti i thread creati esplicitamente in un'applicazione sono di foreground e la mantengono viva finché non siano tutti quanti terminati.

Per fare in modo che i thread creati siano, al contrario, dei thread da eseguire in background, è necessario impostare esplicitamente la proprietà IsBackground a true:

```
thread.IsBackground=true;
```

A qualunque punto sia giunto nel suo compito, un thread di background viene terminato quando anche il thread primario dell'applicazione termina.

Passaggio di parametri ai thread

In molti casi potrebbe essere necessario dover passare dei parametri avviando un nuovo thread, così da poterli utilizzare all'interno del codice secondario eseguito.

Un secondo costruttore di Thread prevede l'utilizzo di un delegate Parameterized-ThreadStart che, a differenza di ThreadStart, consente anche di passare un object come argomento del metodo eseguito nel thread, che quindi esso potrà utilizzare al suo interno. Per esempio, supponiamo di avere un metodo come il seguente:

```
public static void PrintObject(object obj)
```

```
{
Console.WriteLine(obj);
}
```

Il Thread può essere istanziato quindi così:

```
Thread th4=new Thread(PrintObject);
```

Il parametro, invece, verrà inviato a esso tramite il solito Start:

```
string hello = "hello";
th4.Start(hello);
```

La stessa identica procedura può essere implementata tramite un metodo anonimo:

```
string hello = "hello";
Thread th5=new Thread(new ParameterizedThreadStart( (object parameter)=>
{
Console.WriteLine(parameter);
for(int j=0;j<10;j++)
{
Console.Write(".");
Thread.Sleep(100);
}
}));
th5.Start(hello);
```

L'approccio appena visto è quello presente in C# fin dalla prima versione e soffre principalmente della limitazione di accettare un unico parametro possibile.

Grazie alle espressioni lambda introdotte in C# 3.0, è possibile rendere il tutto molto più semplice e, soprattutto, passare un numero di parametri qualunque e di qualsiasi tipo. Il concetto chiave è quello delle variabili catturate dalle espressioni:

```
string hello = "hello";
int n=5;
int interval=1000;
Thread th6=new Thread( ()=> {
for(int i=0;i<n;i++)
{
Console.WriteLine(hello);
Thread.Sleep(interval);
}
});
```

L'esempio precedente utilizza tre variabili dichiarate fuori dal thread e catturate dall'espressione lambda che costituisce il corpo del thread secondario.

Concorrenza e sincronizzazione

La possibilità di avere più thread di esecuzione, oltre a portare ai vantaggi fino a qui esposti, può purtroppo provocare anche qualche problema; si pensi, per esempio, all'eventualità di più thread che accedano a una risorsa condivisa, anche una semplice variabile. Pertanto, quando si ha a che fare con un'applicazione multithread, è necessario assicurarsi che i dati utilizzati siano protetti dal possibile accesso da parte di thread differenti.

Cerchiamo di illustrare il problema con un semplice esempio. Supponiamo di avere un campo intero all'interno della nostra classe:

```
private int n=0;
```

Quindi creiamo ed eseguiamo cinque semplici thread che non fanno altro che leggere il campo, incrementarlo e stamparne il valore:

```
for (int i = 0; i<5; i++)
{
Thread th = new Thread(()=>{
n = n+1;
Thread.Sleep(1);
Console.WriteLine("n={0}",n);
});
th.Start();
}
```

L'invocazione dello `Sleep` consente di simulare il comportamento del sistema operativo, che potrebbe interrompere l'esecuzione di un thread A proprio dopo avere incrementato il campo `n` e prima di stamparlo.

In tal modo, uno o più thread concorrenti potrebbero nel frattempo aver letto e incrementato la stessa variabile così che, quando il thread A riprenderà la sua esecuzione, il valore di `n` non sarà più quello che aveva lasciato, perché sarà stato modificato dai suoi concorrenti.

Tale condizione, in cui due o più thread tentano di accedere alla stessa risorsa, viene detta *race condition*.

Un'altra condizione che, invece, può portare al blocco dell'applicazione perché diversi thread restano in attesa l'uno dell'altro, in maniera indefinita, è detta *deadlock*.

Provando a eseguire il codice precedente, si otterranno quindi valori non predicibili a priori. Per esempio, fra le varie prove ho ottenuto spesso la stampa di questi valori:

```
n=3
n=3
n=4
```

```
n=5
n=5
```

Il risultato finale sarà un valore corretto, perché n ha raggiunto il valore di 5, ma i valori intermedi utilizzati dai thread sono invece soggetti a possibili disturbi.

Ciò, intanto, fa capire l'esigenza di progettare in maniera corretta le applicazioni che sfruttano più thread di esecuzione. Suggerisce anche la necessità di utilizzare degli accorgimenti che evitino le race condition, i deadlock o altre problematiche: nei prossimi paragrafi mostreremo alcune strategie di sincronizzazione degli accessi per rendere il nostro codice thread-safe, cioè sicuro rispetto ai thread che tentano di accedere risorse condivise.

L'istruzione lock

La prima tecnica che permette di sincronizzare gli accessi alle risorse condivise da parte di thread molteplici è l'uso della parola chiave lock.

Essa permette di definire un blocco di istruzioni, detto *sezione critica*, che deve essere sincronizzato, in maniera che un thread che inizi a eseguire tale blocco non possa essere interrotto da un altro prima di completarlo.

L'utilizzo di lock si basa su un oggetto di un tipo riferimento (perché altrimenti verrebbe bloccata solo una copia), che viene utilizzato come token da acquisire e da bloccare per poter entrare all'interno della sezione critica di istruzioni:

```
lock(obj)
{
//sezione critica
}
```

In tal modo, un secondo thread non potrà acquisire il token, se già utilizzato da un altro thread, e dovrà attenderne il rilascio. Riprendendo l'esempio problematico del paragrafo precedente, ecco come utilizzare lock per controllare l'accesso alla variabile n:

```
private object myObj=new object();

for (int i = 0; i<5; i++)
{
Thread th = new Thread(()=>
{
lock(myObj)
{
n=n+1;
Thread.Sleep(1);
Console.WriteLine("n={0}",n);
}
});
th.Start();
}
```

Nel caso precedente, viene istanziata una variabile myObj da usare con la keyword lock.

Sebbene un qualunque oggetto di tipo riferimento sia utilizzabile come token, le best practice suggeriscono di definire un campo privato, o privato e statico, per proteggere l'accesso a dati comuni a più istanze.

NOTA	È importante utilizzare un oggetto privato con l'istruzione lock, altrimenti anche un oggetto esterno potrebbe acquisire un lock sullo stesso oggetto.

Eseguendo ora il codice, i cinque thread istanziati accederanno uno alla volta alla sezione critica. Quindi, non vi è più pericolo che uno di essi incrementi la variabile n e poi venga interrotto. Infatti, un secondo thread, appena giunto al blocco lock, verrà arrestato e rimarrà fermo in attesa che il precedente thread rilasci l'uso di myObj. Il rilascio dell'oggetto avverrà quando un thread uscirà dalla sezione critica, cioè dal blocco racchiuso nel costrutto lock.

La classe Interlocked

Per semplici problematiche di sincronizzazione, come l'incremento di una variabile dell'esempio precedente, anziché utilizzare l'istruzione lock, con relativa creazione di un oggetto, è possibile sfruttare la classe Interlocked.

L'utilizzo di Interlocked è molto più efficiente. Purtroppo, però, è possibile solo per casistiche in teoria molto semplici ma che, nonostante questa apparente semplicità, non sono operazioni atomiche e quindi non sono thread-safe.

L'incremento di una variabile, per esempio, non è un'operazione atomica, in quanto essa richiede la lettura del valore attuale, l'incremento del valore e la memorizzazione del nuovo valore. Un thread potrebbe quindi essere interrotto proprio in mezzo a tali fasi.

Anziché controllare l'incremento di una variabile tramite lock, per esempio così:

```
lock(myObj)
{
n++;
}
```

si può utilizzare il metodo **Increment** di Interlocked:

```
Interlocked.Increment(ref n);
```

che incrementa il valore di n in maniera thread-safe.

La classe Interlocked fornisce altri metodi per implementare operazioni su variabili condivise fra molteplici thread in maniera atomica: decremento, somma di due variabili, confronto di variabili.

La classe Monitor

L'istruzione lock vista in precedenza viene interpretata dal compilatore trasformandola nell'uso della classe Monitor, che, utilizzata esplicitamente da parte dello sviluppatore, consente un controllo più fine della sincronizzazione.

Un'istruzione del tipo:

```
lock(locker)
{
//sezione critica
}
```

è equivalente all'uso dei metodi di Monitor nel seguente modo:

```
Monitor.Enter(locker);
try
{
//sezione critica
}
finally
{
Monitor.Exit(locker);
}
```

Tuttavia, la classe Monitor, a differenza di lock, permette anche di definire un tempo massimo che un thread dovrà attendere prima di entrare nella sezione critica. Tale gestione di timeout si implementa utilizzando il metodo TryEnter:

```
bool lockTaken=false;
Monitor.TryEnter(locker, 1000, ref lockTaken);
if (lockTaken)
{
try
{
//sezione critica
n++;
}
finally
{
Monitor.Exit(locker);
}
}
else
{
//lock non acquisito
}
```

Il metodo TryEnter imposta un timeout in millisecondi, o tramite un TimeSpan, e utilizza una variabile booleana passata con il modificatore ref. Se il timeout scade prima di riuscire ad acquisire il lock, tale variabile rimarrà impostata a false, indicando che non è stato possibile ottenere l'accesso alla sezione critica.

Il metodo Exit, infine, rilascia il blocco e quindi rende possibile a un altro thread l'accesso alla sezione critica.

La classe Mutex

La classe Mutex è un'altra di quelle che permettono in .NET la sincronizzazione degli accessi a risorse condivise, in questo caso anche da parte di processi differenti.

Il concetto è molto simile a quello dei Monitor: il nome Mutex deriva infatti da *Mutual Exclusion*, cioè *mutua esclusione*, in quanto consente l'accesso a un data risorsa a un solo thread per volta, escludendo gli altri.

A differenza dei Monitor però, un Mutex può essere conosciuto a livello di sistema operativo, quindi se un processo ha già acquisito l'uso esclusivo di un Mutex, identificato dal suo nome, non potrà essere utilizzato da nessun altro thread, anche di processi differenti.

NOTA	Se il Mutex viene creato senza assegnargli alcun nome, o con una stringa vuota, esso sarà locale, cioè non condiviso con altri processi a livello di sistema operativo.

Per creare un Mutex, quindi, bisogna indicare se il thread corrente voglia acquisire da subito il Mutex (con il primo parametro), assegnare un eventuale nome (il secondo parametro) e farsi restituire come parametro d'uscita un valore booleano che indichi se l'oggetto è stato creato correttamente. In tal modo è possibile, per esempio, creare un'applicazione di cui sia eseguibile una sola istanza alla volta:

```
bool createdNew;
Mutex mutex = new Mutex(false, "MyMutex123", out createdNew);

if (!createdNew)
{
Console.WriteLine("È possibile eseguire solo un'istanza dell'applicazione");
}
```

Se un'altra istanza dell'applicazione è riuscita a creare e ad acquisire il Mutex di nome MyMutex123, una seconda istanza che proverà ad avviarsi otterrà false come valore di createdNew.

Per utilizzare invece i Mutex come controllori di accesso a una sezione critica, si utilizza il metodo WaitOne:

```
if (mutex.WaitOne())
{
try
{
//sezione critica
}
finally
{
```

```
mutex.ReleaseMutex();
}
}
```

Per rilasciare un Mutex è necessario invocare ReleaseMutex all’uscita dal blocco condiviso.

Pool di thread

La creazione diretta di istanze di Thread non è l'unica opzione per eseguire thread secondari. Tale procedura, infatti, è abbastanza pesante e costosa dal punto di vista delle performance, tanto che spesso, soprattutto per codice molto semplice da eseguire nel thread secondario, si impiegherebbe più tempo nel suo avvio e nella sua interruzione, anche se si tratta di pochi millisecondi, che nell'esecuzione del codice vera e proprio.

Dato che in molte applicazioni vengono creati thread il cui stato è quasi sempre sospeso, in attesa che si verifichi un evento che, per esempio, potrebbe essere una richiesta periodica di ricerca informazioni o aggiornamenti, è consigliabile utilizzare un pool di thread, gestito dal CLR.

Un *pool di thread* è un gestore che crea e mantiene un insieme di thread pronti all'uso, riutilizzando quando possibile quelli creati in precedenza.

Sin dalla prima versione di .NET, la libreria delle classi di base contiene una classe ThreadPool che implementa direttamente un pool di thread, da utilizzare per accodare attività da eseguire, mediante il metodo QueueUserWorkItem:

```
public static void Main()
{
ThreadPool.QueueUserWorkItem(new WaitCallback(LongOperation));
}

// esegue un'attività.
static void LongOperation (Object stateInfo)
{
Console.WriteLine("Hello world from thread pool.");
}
```

Il metodo LongOperation verrà avviato quando un thread diventerà disponibile.

Il seguente frammento mostra come utilizzare la classe ThreadPool per gestire diversi thread, simulandoli con un ciclo for:

```
static void Main(string[] args)
{
for (int i = 1; i <= 5; i++)
{
ThreadPool.QueueUserWorkItem((obj) =>
{
for (int j = 0; j < 10; j++)
{
Console.WriteLine("{0}{1}", obj, new string('.', j));
Thread.Sleep(100);
}
```

```
}, i);
}
Console.ReadLine();
}
```

In questo caso, al metodo QueueUserWorkItem viene passato come parametro anche il valore della variabile contatore del ciclo for. Eseguendo l'esempio noterete, però, come non vi sia nessun determinismo nell'ordine di esecuzione dei vari thread e, ciclo dopo ciclo, sarà sempre un thread diverso a essere eseguito per primo. Ecco un esempio di risultato dei primi due cicli di esecuzione:

```
1
3
5
4
2
3.
1.
4.
5.
2.
```

Inoltre, la classe ThreadPool non fornisce alcun modo per essere notificati del completamento di uno dei thread; si è costretti a scrivere manualmente il proprio codice per ottenere tale gestione, per esempio utilizzando degli eventi.

A partire da .NET 4.0, però, è stata introdotta la classe Task, che fa parte della Task Parallel Library e che è un metodo preferibile per avvantaggiarsi del pool di thread.

I task

L'uso diretto ed esplicito della classe Thread per implementare applicazioni multithreading, oltre a essere soggetto a diverse limitazioni riguardanti il passaggio di parametri, la gestione di valori di ritorno, la gestione a basso livello di diverse situazioni potenzialmente pericolose come la sincronizzazione, è anche un compito molto gravoso dal punto di vista delle performance. Soprattutto quando il numero di thread da gestire comincia a essere elevato.

Per venire incontro alle esigenze dello sviluppatore di applicazioni multithread, è stato introdotto il concetto di *task*, che è un'astrazione di livello più alto rispetto ai thread: rappresenta un'attività o operazione da eseguire, che può essere svolta sia in un thread secondario sia in maniera sincronizzata, cioè in modo che il chiamante attenda il completamento di tale attività prima di proseguire.

Task Parallel Library

In .NET 4.0 è stata introdotta la *Task Parallel Library* (TPL): un insieme di tipi e API il cui scopo è facilitare lo sviluppo di applicazioni che fanno uso di attività parallele e concorrenti, e che costituisce inoltre le fondamenta delle nuove funzionalità di programmazione asincrona introdotte in C# 5.0, su cui torneremo più avanti.

La Task Parallel Library ha introdotto, in particolare, la nuova classe Task, all'interno del namespace System.Threading.Tasks, che rappresenta il concetto di task presentato nel paragrafo precedente. Essa consente di avviare un'operazione e di gestire anche eventuali attività aggiuntive da iniziare al completamento di quelle precedenti. Inoltre, è possibile farsi restituire un valore di ritorno quando tale operazione è stata completata.

La classe Task può avvantaggiarsi o meno di un pool di thread, ma non è limitata solo al paradigma di programmazione multithread.

Avviare un task

Per avviare un nuovo Task in un thread secondario, servendosi di un pool di thread, si può utilizzare il metodo seguente:

```
Task.Factory.StartNew(() => Console.WriteLine("Hello tasks"));
```

La proprietà Factory di Task restituisce un oggetto TaskFactory, che fornisce i metodi per la creazione di nuovi Task.

Il metodo StartNew crea un nuovo Task e lo accoda per l'esecuzione. Esso è disponibile in sedici diversi overload e quello utilizzato qui, in particolare, prende come parametro un

delegate Action.

Dalla versione 4.5 di .NET, la classe Task espone anche un metodo Run, che semplifica ancora di più la creazione e l'esecuzione del task:

```
Task.Run(()=>Console.WriteLine("Hello tasks 2"));
```

NOTA	Per testare in Visual Studio gli esempi sui task di questo capitolo, è consigliabile inserire come ultima istruzione del metodo Main una chiamata a Console.ReadLine, in maniera che i task creati abbiano il tempo di completare e mostrare il loro output prima dell'uscita dal programma.

Entrambi i metodi, oltre ad avviarne l'esecuzione, restituiscono l'oggetto Task che rappresenta l'attività da svolgere e che è utilizzabile per monitorare l'avanzamento dell'attività e il relativo completamento.

NOTA	Visual Studio fornisce un'apposita finestra di monitoraggio dei Task, simile a quella vista per i Threads. La si può visualizzare mediante il menu Debug -> Windows -> Tasks (vedi Figura 12.2).

Tasks

ID	Status	Start Tim...	Duration (sec)	Location	Task	Pro
4	Blocked	0,000	15,712	Tasks.Program.Main	Tasks.Program.Main.AnonymousMeth	115
7	Active	1,012	14,700	Tasks.Program.Main	Tasks.Program.Main.AnonymousMeth	115
6	Active	1,012	14,700	System.Threading.Ta	Task: <Main>b_5	115
5	Active	1,011	14,701	System.Threading.Ta	Task: <Main>b_0_4	115
10	Scheduled	1,013	14,698	[Scheduled and wait	Task: <Main>b_5	115
9	Scheduled	1,012	14,700	[Scheduled and wait	Task: <Main>b_5	115
8	Scheduled	1,012	14,700	[Scheduled and wait	Task: <Main>b_5	115

```
List<Task> tasks = new List<Task>();
for (int i = 1; i <= 5; i++)
{
    int number = i;
    tasks.Add(Task.Run(() =>
    {
        for (int j = 0; j < 10; j++)
        {
            Console.WriteLine("{0} {1}", number, new string('.', j));
            Thread.Sleep(500);
        }
    }));
}
```

Figura 12.2 – Finestra di monitoraggio dei task in Visual Studio.

Se si vuole creare un Task senza avviarlo (pratica non molto comune), è possibile utilizzare uno dei costruttori di Task e invocarne il metodo Start in seguito:

```
Task mytask=new Task(TaskOperation);
mytask.Start();
```

Per conoscere lo stato di un Task, è possibile esaminare la sua proprietà Status, che restituisce uno dei valori dell'enumerazione TaskStatus. Per esempio, un task ancora in esecuzione avrà valore Running per la proprietà Status, oppure RanToCompletion se è stato completato con successo.

Altre proprietà booleane consentono invece di verificare direttamente se un task è stato completato correttamente (IsCompleted), se si è verificata un'eccezione e quindi è stato terminato (IsFaulted), oppure se è stato annullato (IsCanceled). Su quest'ultima eventualità in particolare torneremo più avanti.

Se si vuole attendere ed essere certi che un Task sia stato completato, e solo dopo questo evento proseguire con il resto del codice, si utilizza il metodo Wait:

```
Task.Run(()=>
{
OperazioneLunga();
});
task.Wait();
```

Il metodo Wait blocca l'esecuzione del codice fino a quando il task in esame non è terminato.

Un altro overload permette altresì di specificare un timeout, trascorso il quale l'attesa viene interrotta anche se il task non ha ancora completato il suo compito. Per mostrare tale casistica, supponiamo di avere un task di durata pari a tre secondi, che simuliamo con uno Sleep, e di impostare un timeout pari a due secondi con il metodo Wait:

```
Task longTask=Task.Run( ()=>{
Console.WriteLine("Start task...");
Thread.Sleep(3000);
Console.WriteLine("End task");
});
longTask.Wait(2000);
Console.WriteLine("after wait..." );
```

Poiché il timeout scatterà prima del completamento del task, il risultato sarà il seguente:

```
Start task...
after wait...
End task
```

Il comportamento predefinito del CLR è quello di eseguire i task sfruttando un pool di thread, ma in casi particolari ciò può essere evitato. Per esempio, per task molto lunghi è preferibile

evitare l'utilizzo di un pool di thread, soprattutto se il numero di task è elevato. Per farlo basta utilizzare un overload che accetti un parametro TaskCreationOptions, indicando il valore LongRunning:

```
Task longTask2 = Task.Factory.StartNew(() =>
{
for (int i = 0; i < 100; i++)
{
Thread.Sleep(100);
}
},
TaskCreationOptions.LongRunning);
```

Ora supponiamo di voler simulare l'esecuzione di più Task, in maniera analoga a quanto già fatto con la classe ThreadPool:

```
for (int i = 1; i <= 5; i++)
{
int number = i;
Task.Run(() =>
{
for (int j = 0; j < 10; j++)
{
Console.WriteLine("{0} {1}", number, new string('.', j));
Thread.Sleep(500);
}
});
}
```

Si noti come la variabile contatore i venga assegnata a una variabile temporanea number, così che ogni task riceva un parametro dal valore univoco.

Per fare in modo che il codice si fermi finché tutti e cinque i task non siano stati completati, è possibile utilizzare il metodo WaitAll, che prende come parametro l'array popolato dai task creati:

```
List<Task> tasks = new List<Task>();
for (int i = 1; i <= 5; i++)
{
int number = i;
tasks.Add(Task.Run(() =>
{
for (int j = 0; j < 10; j++)
{
Console.WriteLine("{0} {1}", number, new string('.', j));
Thread.Sleep(500);
}
})
);

Task.WaitAll(tasks.ToArray());
Console.WriteLine("all tasks completed");
```

Per creare un array da una List basta utilizzare il metodo ToArray.

Il metodo WaitAll attenderà il completamento di tutti i task passati come argomento.

Un metodo analogo, WaitAny, permette invece di attendere il completamento del primo dei task avviati e contenuti nell'array.

Restituire valori

Se un task al termine del suo compito deve restituire un valore, è possibile utilizzare la versione generica Task<TResult>, figlia di Task, che come parametro di tipo specifica il tipo del valore di ritorno.

Per ottenere un'istanza di Task<TResult>, basta ricorrere al metodo Run utilizzando un delegate Func<TResult>.

Supponiamo, per esempio, di voler effettuare il download di una pagina web (che può essere un compito molto impegnativo in basa alla raggiungibilità, alla lentezza della connessione, alla dimensione della pagina) e di restituire quest'ultima sotto forma di stringa HTML:

```
Task<string> task = Task<string>.Run(() =>
{
String url="http://www.microsoft.com";
WebClient wc=new WebClient();
return wc.DownloadString(url);
});

string result = task.Result;
```

La lettura della proprietà Result, se il task non è ancora stato completato, bloccherà l'esecuzione del thread corrente fino al completamento.

Se si vuole utilizzare il TaskFactory, o se non si può compilare per .NET 4.5, è ancora possibile ottenere un Task<T> utilizzando la proprietà Factory come visto in precedenza:

```
Task<string> ts=Task<string>.Factory.StartNew( ()=> {
String url=("http://www.microsoft.com";
WebClient wc=new WebClient();
return wc.DownloadString(url);

});
```

Interruzione di un task

Un Task o un Task<T> possono anche essere annullati, prima del loro completamento, utilizzando un *token* di annullamento o cancellazione. L'utilità di tale opportunità è evidente in applicazioni con interfaccia grafica, in cui magari un pulsante Start avvia una procedura mediante un Task, ma un pulsante Annulla può essere utilizzato per annullare il task suddetto.

Un task, per essere annullabile, deve essere avviato con un parametro di tipo CancellationToken. Un oggetto di tale tipo viene creato tramite un'istanza della classe CancellationTokenSource:

```
var cancelTokenSource = new CancellationTokenSource();
CancellationToken token = cancelTokenSource.Token;
```

Quindi si crea il task, passando il token come argomento al metodo Run:

```
Task taskCancellable = Task.Run(() =>
{
token.ThrowIfCancellationRequested();

for(int i=0;i<50;i++)
{
Thread.Sleep(100);
token.ThrowIfCancellationRequested();
}
}, token);
```

All'interno del codice eseguito dal Task, è possibile ora verificare se è stata inviata una richiesta di cancellazione mediante il metodo ThrowIfCancellationRequest, che implicitamente, in caso affermativo, genera un'eccezione TaskCanceledException che è una sottoclasse di OperationCanceledException.

Per generare una richiesta di cancellazione, è possibile utilizzare il metodo Cancel dell'oggetto CancellationTokenSource, che comunica una richiesta immediata, oppure il metodo CancelAfter, che invece schedula la richiesta dopo un determinato intervallo di tempo:

```
try
{
cancelTokenSource.CancelAfter(1000);
taskCancellable.Wait();
}
catch (AggregateException aggrEx)
{
if(Ex.InnerException is OperationCanceledException)
{
//true
}
if (taskCancellable.IsCanceled && taskCancellable.Status == TaskStatus.Canceled)
{
Console.WriteLine(aggrEx.InnerException.ToString());
}
}
```

Un task annullato assumerà lo stato Canceled, restituito dalla proprietà Status, mentre la proprietà bool IsCanceled sarà pari a true.

Se si vogliono o si devono effettuare altre operazioni, prima di generare l'eccezione di cancellazione, è possibile verificare se c'è una richiesta di annullamento in sospeso mediante

la proprietà IsCancellationRequested:

```
if (token.IsCancellationRequested)
{
//esegue altre operazioni di pulizia
token.ThrowIfCancellationRequested();
}
```

Un CancellationToken può anche essere passato ai metodi di attesa di completamento del task. Wait, WaitAll e WaitAny possono utilizzare un token di cancellazione come parametro, in maniera da terminare prematuramente l'attesa (non il task).

Continuazione di task

In molti casi è già noto a priori quale dovrà essere la sequenza di esecuzione di diversi task: quindi, una volta completato un task, bisogna avviarne un altro, e poi magari un altro ancora. I task possono essere concatenati in una procedura detta di *continuazione*, mediante un metodo ContinueWith.

Simuliamo un'operazione da eseguire in un task mediante un metodo, al quale passeremo un valore intero a indicare la durata:

```
public static void Operation(int time)
{
Console.WriteLine("Start operation");
Thread.Sleep(time);
Console.WriteLine("End operation");
}
```

Se vogliamo eseguire due task in maniera che il secondo inizi solo al completamento del primo, basta utilizzare il metodo ContinueWith sulla prima istanza:

```
Task ts = Task.Run(() => Operation(1000));
ts.ContinueWith(taskPrecedente => Operation(2000));
```

Il metodo ContinueWith, che ha diversi overload, prende come argomento l'oggetto Task precedente e restituisce un altro Task, che quindi può essere utilizzato per aggiungere ulteriori continuazioni con altre chiamate a ContinueWith.

È possibile ritardare l'avvio di un Task combinando la chiamata al metodo statico Delay con il metodo ContinueWith:

```
Task taskRitardato = Task.Delay(5000).ContinueWith( Console.WriteLine("dopo 5 secondi...";)));
```

Se un task restituisce un risultato che si vuol passare a quello successivo, e quindi si utilizza la classe Task<T>, il metodo di continuazione sarà anch'esso generico, ContinueWith<T>.

Il seguente esempio prevede un primo Task che scarica una pagina web; il risultato sotto forma di string sarà letto dal secondo Task, che restituirà il numero di vocali trovate. Quindi, il metodo da utilizzare in questo caso sarà ContinueWith<int>:

```
Task<string> webtask = Task<string>.Run(() =>
{
return DownloadHtml("http://www.microsoft.com");
});
Task<int> taskVocali = webtask.ContinueWith<int>(downloadTask =>
{
int count=0;
string vocali="aeiou";
string result=downloadTask.Result;
foreach(char ch in result)
{
if(vocali.IndexOf(ch)>-1)
count++;
}
return count;
});
Console.WriteLine(taskVocali.Result);
```

Il risultato finale sarà ottenuto dalla proprietà Result del secondo task.

Si può anche gestire la situazione in cui si hanno diverse attività, al completamento delle quali continuare con un altro Task. In tal caso, si utilizza il metodo ContinueWhenAll della classe TaskFactory:

```
Task[] taskArray = new Task[2];
taskArray[0]=new Task(()=> Operation(1000));
taskArray[1] = new Task(() => Operation(2000));
taskArray[0].Start();
taskArray[1].Start();
Task taskCont = Task.Factory.ContinueWhenAll(taskArray, (tasksPrecedenti) => Console.WriteLine("{0} Tasks completati",
tasksPrecedenti.Count()));
taskCont.Wait();
```

Nel caso in cui il task finale debba restituire un valore, è disponibile anche il generico ContinueWhenAll<TResult>. Al contrario, se si vuole avviare il task finale quando almeno uno di quelli antecedenti ha completato la sua esecuzione, è possibile utilizzare il metodo ContinueWhenAny o il generico ContinueWhenAny<TResult>, il cui utilizzo è analogo al precedente:

```
taskArray[0] = new Task(() => Operation(1000));
taskArray[1] = new Task(() => Operation(2000));
taskArray[0].Start();
taskArray[1].Start();
Task<string> taskCont = Task.Factory.ContinueWhenAny<string>(taskArray, taskPrec => "uno dei task è terminato");
Console.WriteLine(taskCont.Result);
```

In questo caso, appena uno dei due task nell'array termina, il taskCont inizia la sua esecuzione. In particolare, quest'ultimo ritorna un valore di tipo string.

La classe Task mette poi a disposizione dei metodi statici per creare dei Task che rappresentano l'azione di attesa del completamento di un insieme di altri task o di uno qualunque di tali task.

Per esempio, il metodo WhenAll crea un Task che verrà completato quando tutti i task passati in un array o in una collezione IEnumerable saranno completati:

```
Task taskAll = Task.WhenAll(new Task[] { t0, t1, t2, t3, t4 });
taskAll.Wait();
```

Al contrario, WhenAny crea un Task che sarà completato quando uno dei task sarà stato completato:

```
Task taskAny = Task.WhenAny(new Task[] { t0, t1, t2, t3, t4 });
taskAny.Wait();
```

Task figli o innestati

Un Task figlio o innestato è un'istanza di Task che viene creata durante l'esecuzione di un altro Task, che sarà detto task padre. I task figli possono comportarsi in due modi differenti. Se un task esegue indipendentemente dal padre, esso viene detto *detached*, ed è il comportamento predefinito:

```
var parent = Task.Factory.StartNew(() =>
{
Console.WriteLine("start task parent.");
var childTask = Task.Factory.StartNew(() =>
{
Console.WriteLine("child task start.");
Thread.Sleep(5000);
Console.WriteLine("child task complete.");
});
});

parent.Wait();
Console.WriteLine("end parent.");
```

L'output stampato sulla console mostra che il task padre termina l'esecuzione anche se il figlio non ha ancora completato la sua:

```
start task partent.
end parent.
child task start.
child task complete.
```

A differenza dei task figli detached, i figli *attached* sono sincronizzati con l'esecuzione del padre. In tal caso, il padre verrà considerato come completato solo quando tutti i task figli saranno a loro volta completati. Per ottenere questo effetto, la creazione dei task figli deve essere fatta utilizzando l'opzione TaskCreationOptions.AttachedToParent, come nel seguente esempio:

```
var parentAttached = Task.Factory.StartNew(() =>
{
Console.WriteLine("start task parent.");
var childTask = Task.Factory.StartNew(() =>
{
Console.WriteLine("attached child task start.");
```

```
Thread.Sleep(2000);
Console.WriteLine("attached child task complete.");
}, TaskCreationOptions.AttachedToParent);
});

parentAttached.Wait();
Console.WriteLine("end parent.");
```

In questo caso, invece, l'output sarà:

```
start task parent.
attached child task start.
attached child task complete.
end parent.
```

Al contrario dei task figli detached, i task figli creati come attached hanno queste peculiarità:

- il padre attende il completamento dei task figli;
- le eccezioni vengono propagate al padre (la gestione delle eccezioni nei task è mostrata nel prossimo paragrafo);
- lo stato del task padre dipende dallo stato dei task figli.

Gestione delle eccezioni

A differenza di quella effettuata nella programmazione multithread che usa la classe Thread o ThreadPool, la gestione delle eccezioni con i Task è molto semplice, ma merita senz'altro qualche considerazione.

Se il codice eseguito all'interno di un Task scatena un'eccezione (e quindi si dice che "il task fallisce"), l'eccezione stessa viene propagata al punto in cui si attende il completamento del Task stesso, per esempio con Wait o simili, o in cui si accede alla proprietà Result nel caso di Task<TResult>.

Supponiamo di volere eseguire in un Task il seguente metodo di divisione:

```
public static int Dividi(int x, int y)
{
return x / y;
}
```

Basta creare un Task<int> e passare gli argomenti al metodo Dividi. Se però il parametro y è nullo, si verificherà un'eccezione DivideByZeroException.

Per gestire le eccezioni bisogna utilizzare un blocco try/catch che contiene il tentativo di lettura di Task.Result:

```
Task<int> taskDiv = Task.Run<int>(() => Dividi(1, 0));
try
{
int risultato = taskDiv.Result;
}
```

```
catch(AggregateException aggrEx)
{
Console.WriteLine(aggrEx.InnerException.ToString());
}
```

L'eccezione generata dal Task viene inglobata in una AggregateException.

La classe AggregateException contiene tutte le eccezioni che hanno causato il fallimento dei task coinvolti e che sono accessibili tramite la proprietà InnerExceptions. Tramite la proprietà InnerException di quest'ultima è possibile risalire all'eccezione originaria.

Un'altra possibilità di esaminare le eccezioni che hanno causato il fallimento del Task è quindi verificare se la proprietà IsFaulted è true e ricavare le eccezioni:

```
try
{
...
}
catch
{
if (taskDiv.IsFaulted)
{
Console.WriteLine("il task è fallito");
foreach(var ex in taskDiv.InnerExceptions)
Console.WriteLine(ex.Message);
}
}
```

Programmazione asincrona in C# 5.0

Un metodo *sincrono* svolge il suo lavoro e al termine ritorna il controllo al chiamante. Viene quindi anche detto metodo *bloccante* perché l'esecuzione del programma resta bloccata fino a quando il metodo non termina la sua esecuzione.

I normali metodi di una classe sono metodi sincroni, per esempio Console.WriteLine viene invocato, eseguito e solo al suo completamento si prosegue con l'istruzione successiva.

Un metodo asincrono, al contrario, una volta invocato restituisce il controllo al chiamante immediatamente, nel frattempo inizia a svolgere il suo lavoro. Quando si istanzia e avvia un Task, mediante il metodo Run, il chiamante non si blocca ad attendere il completamento del task creato, ma lo esegue in maniera asincrona, continuando la sua esecuzione, a meno che non si voglia attenderne poi il completamento, per esempio utilizzando il metodo Wait.

L'importanza del paradigma di *programmazione asincrona* sta principalmente nel fatto che esso permette di evitare colli di bottiglia e di migliorare la risposta dell'applicazione, soprattutto se si tratta di applicazioni con interfaccia grafica che accedono a risorse la cui disponibilità e il tempo di risposta sono limitati e non predicibili.

Si pensi che oggi la stragrande maggioranza delle applicazioni è implementata in maniera distribuita, accede a reti locali o al Web, gira su dispositivi smartphone o tablet con una potenza di calcolo limitata rispetto ai desktop e con larghezza di banda spesso anch'essa limitata o instabile.

Se una risorsa sul Web non è momentaneamente disponibile, oppure se l'attività di accesso è lenta, l'applicazione dovrà necessariamente attendere. In un processo sincrono, l'intera applicazione rimarrebbe bloccata; in uno asincrono, invece, è possibile nel frattempo continuare a eseguire altri compiti, e l'utente può continuare a utilizzare l'applicazione.

Date queste premesse, è facile comprendere che avere un linguaggio che supporta nativamente delle primitive di programmazione asincrona è un vantaggio fondamentale. In C# 5.0, la novità principale è costituita proprio dalle parole chiave async e await, che rendono più immediato, e soprattutto integrato direttamente nel linguaggio, l'utilizzo di funzionalità introdotte nella Task Parallel Library e viste nei paragrafi precedenti.

In parole povere, con async e await la scrittura di codice asincrono, funzionalmente pari a codice che sfrutta la classe Task e i concetti di continuazione, mantiene la stessa struttura e semplicità del codice sincrono.

Sarà il compilatore C# a tradurlo, generando per conto dello sviluppatore del codice che sfrutta i tipi e i membri contenuti nei namespace System.Threading e System.Threading.Tasks.

Nei prossimi paragrafi mostreremo l'utilizzo di async e await in C# per la scrittura di codice asincrono.

Le parole chiave async e await

La parola chiave async serve a marcare un metodo come asincrono. Essa può essere utilizzata come modificatore di un metodo classico ma, come vedremo, anche di metodi anonimi e di espressioni lambda.

La presenza di async indica al compilatore che all'interno del metodo verranno utilizzate funzionalità asincrone, per l'esattezza la parola await.

Se non è presente il modificatore async nella dichiarazione del metodo, non sarà possibile utilizzare await.

Non è vero invece il contrario: un metodo può anche contenere async nella sua firma ma non utilizzare await nel corpo, anche se in tal caso il compilatore ci avviserebbe con un warning.

NOTA	La presenza della parola chiave async potrebbe sembrare superflua perché, se all'interno di un metodo si usa await, ciò implica che il metodo è necessariamente asincrono. In realtà, poiché la parola chiave await è stata introdotta solo con C# 5.0, essa può essere anche usata come normale identificatore in codice sincrono. Quindi essa è una parola chiave contestuale, cioè lo è solo in contesto asincrono, che è segnalato dalla presenza del modificatore async.

La struttura di un metodo asincrono è simile alla seguente:

```
public async void MetodoAsincrono()
{
await Metodo();
}
```

La parola chiave await è usata per sospendere l'esecuzione di un metodo fino a che il metodo da essa preceduto non completi la sua esecuzione. Quindi essa indica che l'esecuzione dell'espressione deve avvenire in maniera asincrona.

L'espressione usata con await deve essere o restituire un oggetto *awaitable*, in genere un Task o un Task<T>.

Ma, generalizzando il discorso, l'attività che deve essere eseguita in asincrono non è necessariamente un'istanza della classe Task, bensì un oggetto di un tipo che espone un metodo

GetAwaiter, che non ha parametri di ingresso e restituisce un oggetto cosiddetto *awaiter*, con i suoi specifici membri utilizzati dall'infrastruttura di programmazione asincrona.

In realtà, però, raramente è necessario implementare il proprio tipo di awaiter e si utilizzeranno direttamente oggetti Task.

NOTA	La classe Task è un tipo *awaitable* proprio perché espone un metodo GetAwaiter, che restituisce un'istanza del tipo TaskAwaiter.

Ripensando al concetto di continuazione dei task di qualche paragrafo fa, await permette quindi di implementare la gestione della continuazione in maniera molto semplice.

Esecuzione di metodi asincroni

Quando si incontra l'espressione await, l'esecuzione ritorna al chiamante del metodo asincrono, ma il CLR attacca una continuazione al task con await in maniera che, quando esso termina, l'esecuzione ritorna nuovamente all'interno del metodo e continua dall'istruzione successiva all'espressione await.

Proviamo a scrivere un esempio più completo per comprendere meglio il funzionamento e il flusso di esecuzione che ne deriva. Il metodo seguente è indicato come asincrono mediante la parola chiave async:

```
public static async void MetodoAsincrono()
{
Console.WriteLine("before await");
await Task.Run(() =>
{
Console.WriteLine("start task");
Thread.Sleep(3000);
Console.WriteLine("end task");
});
Console.WriteLine("after await");
}
```

Al suo interno viene creato e avviato un Task, ma poi si attende il suo completamento, mediante l'uso di await, prima di continuare con l'istruzione successiva.

Per utilizzare il MetodoAsincrono non serve nulla di speciale:

```
static void Main()
{
Console.WriteLine("before");
MetodoAsincrono();
Console.WriteLine("after");}
```

Compilando ed eseguendo il codice precedente, provate a intuire quale sarà l'ordine di esecuzione dei vari metodi WriteLine e quindi l'output prodotto. Altrimenti, se non avete voglia

o pazienza di provare, ecco il risultato:

```
before
before await
start task
after
end task
after await
```

Quindi, dopo l'invocazione del MetodoAsincrono, il suo metodo chiamante, che qui è il Main, continua la sua esecuzione. Infatti, il secondo WriteLine del Main, quello che stampa after, viene eseguito prima che il task interno venga completato.

Terminato il task all'interno del metodo asincrono, quello atteso con await, il controllo ritorna all'interno di MetodoAsincrono, all'istruzione successiva, cioè alla Console.WriteLine("after await").

La parola chiave await può essere utilizzata in qualunque contesto del codice, a esclusione di codice nativo unsafe, di blocchi catch/finally e lock, o nel metodo Main (perché non è possibile usare il modificatore async con quest'ultimo).

In pratica, await può apparire nel corpo di qualunque metodo, metodo anonimo ed espressione lambda marcati con async.

Il seguente metodo implementa la copia di uno stream sorgente su uno destinazione, in maniera asincrona:

```
public static **async** void CopyToAsync(Stream source, Stream destination)
{
byte[] buffer = new byte[0x1000];
int numRead;
while((numRead = **await** source.ReadAsync(buffer, 0, buffer.Length)) > 0)
{
**await** destination.WriteAsync(buffer, 0, numRead);
}
}
```

L'espressione await appare sia all'interno della condizione del while, per leggere i byte dalla sorgente, sia all'interno del corpo dello stesso ciclo, in fase di scrittura.

Scrittura di metodi asincroni

Un metodo marcato come async può essere invocato anch'esso con un'espressione await, se restituisce un Task o Task<T>. La presenza di await "trasformerà" il tipo di ritorno da Task<TResult> in TResult oppure, se il tipo di ritorno è Task, in void. Per esempio, si può scrivere un metodo asincrono come il seguente, che simula un Task della durata di un secondo e poi restituisce una stringa, indicando come tipo di ritorno Task<string>:

```
public static async Task<string> GetStringAsync()
{
```

```
await Task.Delay(1000);
return "completed";
}
```

All'interno del metodo bisogna restituire direttamente un oggetto string (e non Task<string>).

Il metodo GetStringAsync può essere utilizzato per ottenere una stringa, e aspettare il suo completamento, per mezzo della parola chiave await.

```
string str=await GetStringAsync();
Console.WriteLine(Str);
```

Il valore della variabile str viene impostato al completamento di GetStringAsync, dopodiché l'esecuzione prosegue con l'istruzione successiva che stampa la stringa ottenuta.

Come detto prima l'await di un Task non generico restituisce invece un void. Quindi, per scrivere un metodo asincrono, che non restituisce valori, basta utilizzare al posto di void il tipo non generico Task:

```
public async Task OperazioneAsync()
{
await Task.Delay(5000);
Console.WriteLine("task completato");
}
```

All'interno del metodo non è necessario restituire un oggetto Task, perché sarà il compilatore a gestire la continuazione restituendo il controllo al metodo asincrono (all'istruzione Console.WriteLine, in questo caso).

Il metodo OperazioneAsync ora è semplicemente utilizzabile come se fosse un normale metodo sincrono, basta farlo precedere da await:

```
await OperazioneAsync();
```

NOTA	La convenzione suggerita da Microsoft per i nomi dei metodi asincroni, cioè quelli che restituiscono un Task, prevede di aggiungere il suffisso Async. In tal modo, sarà facile riconoscere il metodo e capire che è necessario utilizzarlo con await.

Utilizzo di await in catch/finally

Prima di C# 6 non era consentito utilizzare l'istruzione await per invocare metodi asincroni in blocchi catch o finally, a causa di limitazioni del compilatore.

Questo costringeva spesso gli sviluppatori a scrivere codice più articolato per chiamare poi il metodo asincrono subito dopo il catch/finally in base a una condizione. C# 6 permette invece di usare await direttamente al loro interno:

```
try
{
```

```
//codice che genera eccezioni
}
catch (Exception ex)
{
await SaveExceptionAsync(ex); //invoca metodo asincrono
}
finally
{
await LogAsync(ex); //invoca metodo asincrono
}
```

Delegate ed espressioni lambda asincrone

Come detto in precedenza, l'utilizzo di async e await può avvenire quasi in ogni contesto di codice. Le espressioni await, infatti, non sono permesse all'interno di codice unsafe, in blocchi lock, o nel metodo Main. Per il resto, possiamo scrivere del codice asincrono praticamente ovunque.

Supponiamo di aver implementato un metodo asincrono per la lettura di un file e simuliamolo con il seguente esempio:

```
public async Task ReadFileAsync(string path)
{
//simulazione
await Task.Delay(1000);
Console.WriteLine(path);
}
List<string> files = new List<string>(){"file1.txt", "file2.txt", "file3.txt"};
```

Potremmo quindi usare il metodo ReadFileAsync su ogni elemento della lista con un delegate asincrono passato al metodo ForEach:

```
files.ForEach(async delegate(string file)
{
await ReadFileAsync(file);
});
```

Lo stesso risultato si può ottenere con una espressione lambda:

```
files.ForEach(async file => await ReadFileAsync(file));
```

Notate dove appaiono in entrambi i casi le parole chiave async e await.

Espressioni lambda e delegate asincroni possono essere usati anche come gestori di evento:

```
button1.Click += async (sender, args) =>
{
await ReadFileAsync(…);
}
```

Un'espressione lambda asincrona può anche restituire un risultato:

```
Func<string, Task<string>> Lambda = async (file) => await GetNumbersFromStringAsync(file);
```

```
string result=await Lambda("129mns93");
```

In questo caso, il tipo di ritorno del delegate Func deve naturalmente essere un Task<TResult>.

Esecuzione parallela

Un metodo asincrono, indicato dal modificatore async, può anche essere invocato senza await. Supponendo quindi di invocare due o più metodi asincroni in questo modo, uno dopo l'altro, essi saranno eseguiti in parallelo.

Utilizziamo il metodo seguente, che ricava in maniera asincrona da una stringa i caratteri di formato numerico:

```
public static async Task<string> GetNumbersFromStringAsync(string str)
{
StringBuilder sb = new StringBuilder();
await Task.Run(()=>
{
foreach(char ch in str)
{
if(Char.IsNumber(ch))
sb.Append(ch);
}
});
return sb.ToString();
}
```

Come detto in precedenza, la firma del metodo dichiara il tipo di ritorno Task<string>, ma all'interno del metodo stesso avremo un'istruzione return che restituirà direttamente un oggetto string.

Possiamo ora utilizzarlo con await:

```
string str1 = await GetNumbersFromStringAsync("a1sd32jklfs89'03jmfws");
Console.WriteLine(str1);
string str2 = await GetNumbersFromStringAsync("fsdf03,v01gr52");
Console.WriteLine(str2);
```

Il metodo viene invocato due volte e tramite await si attende il completamento in entrambi i casi prima di stampare il risultato.

Se invece non si utilizza await, le due invocazioni del metodo saranno eseguite in parallelo:

```
var task1 = GetNumbersFromStringAsync("a1sd32jklfs8912gs13'03jmfws");
var task2 = GetNumbersFromStringAsync("03,gr5lasd023las vsd 2");
```

In questo caso, poiché manca await, il valore di ritorno è di tipo Task<string>. A questo punto, è possibile attendere il completamento di uno o di entrambi i task ottenuti, sia con await:

```
await task1;
await task2;
Console.WriteLine(task1.Result);
```

```
Console.WriteLine(task2.Result);
```

sia con i metodi della classe Task visti in precedenza, per esempio Wait, WaitAll e WaitAny:

```
Task.WaitAll(task1, task2);
Console.WriteLine(task1.Result);
Console.WriteLine(task2.Result);
```

Metodi asincroni e interfaccia grafica

L'utilizzo di async e await, o in generale di funzioni asincrone, è di enorme utilità nello sviluppo di interfacce grafiche, le quali devono rimanere *responsive* anche quando eseguono operazioni pesanti o lunghe.

Supponiamo, per esempio, di avere in un'applicazione Windows Forms un pulsante che avvia un'operazione di calcolo, la quale impiegherà diversi secondi a restituire il risultato. Tale operazione può essere simulata dal seguente metodo, che impiegherà circa 10 secondi a terminare la sua esecuzione:

```
private void ExecuteLongOp()
{
for (int i = 0; i < 10; i++)
Thread.Sleep(1000);
}
```

Il gestore dell'evento Click di un pulsante presente nella nostra interfaccia grafica (chiamiamolo button1) può quindi avviare l'operazione così:

```
private void button1_Click(object sender, EventArgs e)
{
button1.Enabled = false;
ExecuteLongOp();
button1.Enabled = true;
MessageBox.Show("Operazione sincrona completata");
}
```

La prima istruzione disabilita il pulsante per evitare di renderlo cliccabile nuovamente. Ma tale accorgimento è inutile perché, una volta avviata l'esecuzione del metodo ExecuteLongOp, l'intera interfaccia resterà bloccata e non sensibile a una qualunque azione dell'utente. Dopo 10 secondi, sempre che l'utente non l'abbia terminata forzatamente, l'applicazione si sveglierà e mostrerà una MessageBox.

L'esperienza utente in questo caso non è stata sicuramente il massimo. Immaginate che cosa accadrebbe per operazioni ancora più lunghe e pesanti.

Sfruttando ora i concetti di programmazione asincrona e le parole chiave async e await di C#, vediamo come realizzare lo stesso esempio in maniera che l'interfaccia e l'utilizzo dell'intera applicazione non ne risultino compromessi.

Innanzitutto bisogna implementare una versione asincrona del metodo precedente:

```
private async Task ExecuteLongOpAsync()
{
await Task.Run(() =>
{
for (int i = 0; i < 10; i++)
Thread.Sleep(1000);
});
}
```

Le stesse istruzioni vengono eseguite per mezzo di un Task.

Adesso, supponendo di avere un secondo pulsante button2, non ci resta che utilizzare il nuovo metodo in maniera asincrona, con un'espressione await:

```
private async void button2_Click(object sender, EventArgs e)
{
button2.Enabled = false;
await ExecuteLongOpAsync();
button2.Enabled = true
MessageBox.Show("Operazione async completata");
}
```

Si noti che anche il metodo gestore dell'evento deve utilizzare il modificatore async, dato che al suo interno deve invocare il metodo ExecuteLongOpAsync con await. Eseguendo il nuovo esempio, con un clic sul pulsante button2 l'applicazione rimane perfettamente attiva e l'interfaccia risponde ancora all'input utente, che potrebbe per esempio fare clic su un pulsante Stop per fermare l'esecuzione del metodo, cosa non possibile nel primo caso.

Questo esempio mostra con quanta rapidità e con quanta semplicità sia possibile utilizzare le funzionalità asincrone di C# senza dover creare ed eseguire Thread o Task, e soprattutto con una sintassi praticamente identica a quella utilizzata in ambito sincrono.

Programmazione parallela

Negli ultimi anni, i produttori di CPU hanno immesso sul mercato nuovi prodotti dalle prestazioni sempre più elevate, sfruttando la possibilità di integrare più core all'interno di una singolo processore. La presenza di due, quattro, otto core è ormai la norma nei personal computer odierni e non è difficile attendersi da qui a breve l'uscita di CPU con un ancora più alto numero di core.

Per sfruttare questa potenza di calcolo e quella futura, in particolare la possibilità di far svolgere più compiti contemporaneamente alle CPU multicore senza dover scrivere codice multithreading di basso livello, la Task Parallel Library, oltre a fornire le classi per la gestione di Task viste finora, include anche una classe Parallel, che è un'ulteriore astrazione dei thread, dedicata all'esecuzione di codice in parallelo.

Questo insieme di API, che comprende la già vista Task Parallel Library, la classe Parallel e *PLINQ* (che sta per *Parallel LINQ*, un'implementazione in parallelo di LINQ), è anche noto come Parallel Framework, abbreviato in PFX.

La Figura 12.3 mostra l'architettura di .NET 4.5 dedicata alla programmazione parallela.

La classe Parallel

Il parallelismo sui dati permette di eseguire una stessa operazione in parallelo sugli elementi di un array o di una collezione. I dati di origine vengono suddivisi in partizioni in modo che più thread, eseguiti su core del processore differenti, possano occuparsi contemporaneamente di segmenti di dati differenti. Tale parallelismo è supportato da .NET con la Task Parallel Library, e in particolare tramite la classe Parallel, inclusa nel namespace System.Threding.Tasks.

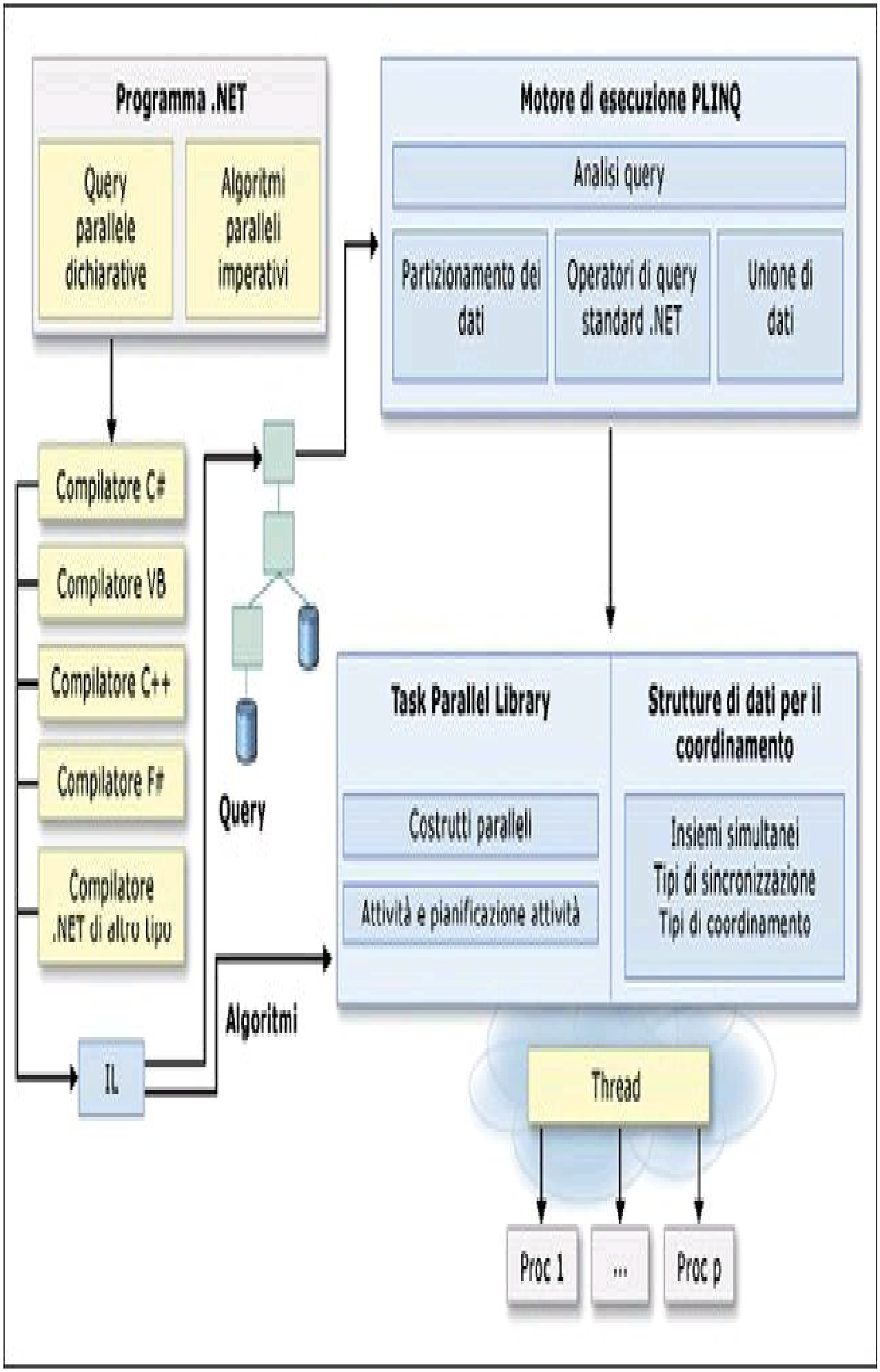

Programma .NET
Query parallele dichiarative
Algoritmi paralleli imperativi
Motore di esecuzione PLINQ
Analisi query
Partizionamento dei dati
Operatori di query standard .NET
Unione di dati
Compilatore C#
Compilatore VB
Compilatore C++
Compilatore F#
Compilatore .NET di altro tipo
IL
Query
Algoritmi
Task Parallel Library
Costrutti paralleli
Attività e pianificazione attività
Strutture di dati per il coordinamento
Insiemi simultanei
Tipi di sincronizzazione
Tipi di coordinamento
Thread
Proc 1
...
Proc p

Figura 12.3 – Architettura di programmazione parallela in .NET 4.5.

Parallel è una classe di alto livello che espone tre metodi statici, i quali implementano la versione parallela dei cicli for e foreach. Questi metodi sono elencati e descritti nella Tabella 12.1.

Tabella 12.1 - **I metodi della classe Parallel.**

Metodo	Descrizione
Parallel.For	esegue un ciclo equivalente a for in parallelo
Parallel.ForEach	esegue un ciclo equivalente a foreach in parallelo
Parallel.Invoke	esegue un array di delegate in parallelo

NOTA	L'esecuzione in parallelo dipende da vari fattori, come il carico del sistema e la quantità di dati da processare. Ciò significa che l'utilizzo di un metodo della classe Parallel non assicura tale tipo di esecuzione.

Il metodo Parallel.For

Il metodo Parallel.For consente di eseguire un ciclo for in parallelo; in tal modo, anziché eseguire le varie iterazioni una dopo l'altra, in sequenza, esso le eseguirà utilizzando thread differenti. Il numero di thread e la loro creazione sarà automaticamente ottimizzata, per ottenere il massimo parallelismo possibile.

Il metodo ha dodici diversi overload, di cui il più semplice ha la seguente firma:

```
public static ParallelLoopResult For(
int fromInclusive,
int toExclusive,
Action<int> body
)
```

Il primo e il secondo parametro rappresentano l'indice di partenza (incluso) e quello finale (escluso) delle iterazioni da eseguire, mentre l'ultimo è un delegate di tipo Action<int> che rappresenta l'azione da iterare.

Per esempio, per stampare i numeri da 1 a 10, si può utilizzare il ciclo Parallel.For in questo modo:

```
Parallel.For(1, 11, i => Console.WriteLine("Parallel.For {0}", i));
```

L'ordine con cui saranno eseguite le iterazioni non è predicibile a priori. Provando a eseguire il precedente esempio, il risultato sulla mia macchina è il seguente:

```
Parallel.For 0
Parallel.For 3
```

```
Parallel.For 1
Parallel.For 2
Parallel.For 7
Parallel.For 9
Parallel.For 8
Parallel.For 4
Parallel.For 5
Parallel.For 6
```

Quindi, il metodo Parallel.For deve essere usato solo se le iterazioni sono indipendenti l'una dall'altra. Per esempio, se un'iterazione x dipende dai risultati della (x-1)esima, allora non è possibile utilizzare la tale versione parallela perché si otterrebbero risultati finali imprevedibili.

Gli altri overload del metodo accettano altri parametri, utili in particolari casi. Per esempio, è possibile impostare delle opzioni per controllare il grado di parallelismo, oppure monitorare e manipolare lo stato del ciclo.

Il seguente esempio utilizza un delegate Action<int, ParallelLoopState>:

```
Parallel.For(0, 50, (i, loopState) =>
{
Console.WriteLine(i);
if (i > 10)
{
loopState.Break();
}
});
```

L'oggetto ParallelLoopState viene utilizzato per fermare il ciclo prematuramente. In particolare, tale classe ha due metodi: Break e Stop.

Nel precedente esempio viene invocato il metodo Break quando l'indice di iterazione i assume un valore maggiore di 10; in tal modo si informa il ciclo che le iterazioni successive a quella corrente devono essere fermate. Si noti che l'ordine di esecuzione delle iterazioni non è sequenziale, quindi prima di giungere alla numero 10 potrebbero anche essere state avviate iterazioni con i superiore a 10.

Il metodo Stop, invece, forza tutti i thread avviati a terminare la loro esecuzione il prima possibile, quindi le iterazioni rimanenti non completate devono essere abbandonate, anche se hanno un numero di iterazione inferiore a quella che ha causato lo Stop.

Nel caso in cui fosse necessario conoscere lo stato di esecuzione delle iterazioni, per esempio per sapere se l'intero ciclo è stato completato, il metodo For restituisce un oggetto ParallelLoopResult:

```
ParallelLoopResult result=Parallel.For(0, 50, (i, loopState) =>
{
```

```
Console.WriteLine(i);
if (i > 10)
{
loopState.Break();
}
});
if(!result.IsCompleted)
{
int i=result.LowestBreakIteration;
}
```

La proprietà IsCompleted indica se le iterazioni sono state completate. LowestBreak-Iteration, nel caso in cui il For sia stato interrotto con un Break, restituisce l'indice dell'iterazione più bassa che ha chiamato il Break.

Il metodo Parallel.ForEach

Il metodo Parallel.ForEach è l'analogo parallelizzato del costrutto foreach, quindi permette di eseguire un ciclo sugli elementi di una IEnumerable, ma in maniera asincrona. Per esempio, volendo eseguire un ciclo su tutti gli elementi di una lista di stringhe, il metodo ForEach stampa tutte le parole della lista e la relativa lunghezza:

```
List<string> list=new List<string>(){"questa", "è", "una", "frase", "di", "poche", "parole"};
Parallel.ForEach(list, word => Console.WriteLine("{0}: {1}", word, word.Length));
```

Anche in questo caso, l'ordine delle iterazioni non segue necessariamente quello delle stringhe nella lista originale. Per esempio, una delle tante esecuzioni sulla mia macchina ha dato questo risultato:

```
questa: 6
è: 1
poche: 5
frase: 5
parole: 6
una: 3
di: 2
```

Anche ForEach possiede venti overload differenti, simili a quelli di For, e restituisce un ParallelLoopResult.

Il metodo Parallel.Invoke

Il metodo Invoke consente di eseguire in parallelo un array di delegate Action. I metodi da eseguire in parallelo possono essere contenuti in un vero array, oppure passati come numero variabile di argomenti.

Supponiamo di voler eseguire in parallelo i metodi M1, M2, M3 (definiti da qualche parte); la prima modalità di invocarli in tal modo è quindi:

```
Parallel.Invoke(M1, M2, M3);
```

Mentre, se essi fossero già contenuti in un array, basterebbe passare quest'ultimo:

```
Action[] methods = new Action[] { M1, M2, M3 };
Parallel.Invoke(methods);
```

L'altro overload di Invoke permette di impostare delle opzioni di esecuzione del metodo, con un oggetto ParallelOptions. In particolare, è possibile impostare la proprietà MaxDegreeOfParallelism che indica il numero massimo di thread da usare, un TaskScheduler personalizzato per decidere l'ordine di esecuzione dei vari delegate, o impostare un token di annullamento, per interrompere l'esecuzione.

Ecco un esempio di utilizzo:

```
ParallelOptions op = new ParallelOptions();
op.MaxDegreeOfParallelism = 2;
op.CancellationToken = new CancellationTokenSource().Token;
try
{
Parallel.Invoke(op, M1, M2, M3);
}
catch (OperationCanceledException)
{
Console.WriteLine("Cancelled!");
}
```

Dopo che il token di annullamento è stato impostato, chiamando il metodo Cancel o CancelAfter dell'oggetto CancellationTokenSource, viene perciò scatenata un'eccezione OperationCanceledException interrompendo l'esecuzione di ulteriori delegate. Quelli già in esecuzione verrebbero completati in ogni caso.

PLINQ

Parallel LINQ è un'implementazione in parallelo di *LINQ to Objects*.

Tutti i metodi di estensione di LINQ sono disponibili in versione parallela e inoltre altri metodi implementano operatori aggiuntivi. Così, utilizzando la stessa sintassi vista per LINQ, è possibile sfruttare la potenza della programmazione in parallelo, adattando il livello di parallelismo alle capacità hardware della macchina.

Query parallele

Una query PLINQ assomiglia a una query LINQ to Objects non parallela. Le query PLINQ, analogamente alle query LINQ sequenziali, funzionano su qualsiasi origine dati in memoria IEnumerable o IEnumerable<T>. Vengono inoltre eseguite in modo differito, ovvero solo dopo l'enumerazione della query.

La differenza principale è che PLINQ tenta di sfruttare al massimo tutti i processori del sistema. Come già visto parlando dei metodi della classe Parallel, viene eseguito il partizionamento dell'origine dati in segmenti e quindi la query può essere eseguita su ogni segmento in thread distinti, e in parallelo su più processori. Il risultato è che generalmente le query vengono eseguite in tempi significativamente minori.

NOTA	In molti casi, anche parallelizzando una query, le prestazioni ottenute (quindi il tempo di esecuzione) sono peggiori che nella versione non parallela. Ciò è dovuto al fatto che le risorse necessarie per parallelizzare tali query introducono un overhead superiore ai vantaggi ottenuti: in generale, tale eventualità si verifica se la query non esegue calcoli pesanti e se la sorgente dati su cui viene eseguita è di dimensione ridotta.

La classe ParallelEnumerable

Le funzionalità di PLINQ vengono fornite dai metodi di estensione della classe System.Linq.ParallelEnumerable che, oltre alle implementazioni parallele degli operatori standard di LINQ, espone dei metodi specifici che rendono possibile l'esecuzione parallela.

Il punto di ingresso della classe PLINQ può essere considerato il metodo AsParallel della classe ParallelEnumerable, il cui utilizzo è immediato.

A partire da una qualunque sorgente dati di tipo IEnumerable/IEnumerable<T>, utilizzabile quindi per costruirci una query LINQ (vedere il Capitolo 11), è semplice eseguire la stessa in parallelo. Basta invocare AsParallel sulla collezione originale per ottenerne una versione parallela della stessa sequenza di oggetti, rappresentata con il tipo ParallelQuery<T>.

Il seguente esempio restituisce il calcolo della radice quadrata dei primi 10.000 numeri interi, trasformando la sequenza di interi ottenuta con il metodo Range in una sequenza ParallelQuery<int>.

```
var query = from n in Enumerable.Range(1, 10000).AsParallel()
select Math.Sqrt(n);
```

A questo punto, per ogni operatore LINQ applicato all'oggetto ParallelQuery sarà utilizzato il corrispondente metodo definito da ParallelEnumerable. Quindi, se durante la query è necessario applicare altri operatori PLINQ, non è necessario invocare ancora AsParallel: ogni operatore restituirà in ogni caso sempre una ParallelQuery<T>. Per esempio, se si vuol filtrare la sequenza con l'operatore Where:

```
var query = (from n in Enumerable.Range(1, 10000).AsParallel().Where(n=>n%2==0)
select Math.Sqrt(n);
```

L'operatore Where restituisce ancora un oggetto ParallelQuery<int>. Se si vuole convertire nuovamente la sequenza parallela ParallelQuery<T> in una IEnumerable<T>, per forzare la valutazione sequenziale della query si può utilizzare il metodo AsSequential.

NOTA	L'utilizzo di PLINQ è limitato a collezioni di oggetti locali, quindi non è possibile, per esempio, parallelizzare query in LINQ to SQL.

Domande di riepilogo

1) Quale namespace è necessario per utilizzare la programmazione multithread?

a. System.MultiThreading
b. System.Threading
c. System.Task
d. System.Parallel

2) Cosa fa l'istruzione Thread th=new Thread(new ThreadStart(Metodo1));?

a. Crea un'istanza di Thread ma non esegue Metodo1
b. Esegue il Metodo1 in un nuovo thread
c. Non esiste un costruttore di Thread
d. Esegue Metodo1 che restituisce un nuovo thread

3) Per sincronizzare l'accesso a una sezione critica, su quale tipo di membro si deve utilizzare l'istruzione lock?

a. Sull'oggetto this
b. Su un numero intero casuale
c. Su un campo privato di tipo riferimento
d. Su una proprietà pubblica

4) Per consentire l'interruzione di un Task, questo deve essere creato con un parametro di tipo:

a. CancellationToken
b. CancellationTokenSource
c. bool Cancel
d. CancellableTask

5) Per compilare correttamente il metodo seguente, quale parola chiave manca al posto dei puntini?

```
public ... Task MetodoAsync()
{
await Task.Run(()=>Console.Write("Hello"));
}
```

a. void
b. virtual
c. volatile
d. async

6) In quale caso può essere utilizzato del codice asincrono, cioè un'istruzione await?

a. In un blocco lock
b. Nel metodo Main
c. In un blocco catch
d. In un blocco unsafe

7) Un metodo asincrono, indicato dal modificatore async, può anche essere invocato senza await. Vero o falso?

8) Come vengono eseguite le iterazioni di un metodo Parallel.For?

a. In ordine sequenziale
b. In ordine non predicibile a priori
c. In ordine dipendente dal numero di iterazioni
d. Tutte contemporaneamente in parallelo

XML in C#

.NET Framework contiene diverse classi per manipolare dati in formato XML e interrogarli con XPath, mentre con LINQ è stato creato un apposito provider, chiamato LINQ to XML, per rendere più semplice ed efficace lavorare con tale formato.

XML (*Extensible Markup Language*) è un metalinguaggio che fornisce un meccanismo basato su marcatori (ed è quindi anche detto un linguaggio di *markup*, come HTML) per la definizione della struttura di documenti e dei dati in esso contenuti.

Poiché si tratta di un linguaggio testuale, XML permette di creare, conservare e scambiare dati utilizzando una sintassi basata su puro testo; da ciò deriva la sua versatilità, che lo ha portato a essere supportato in maniera praticamente universale da ogni sistema, piattaforma e linguaggio di programmazione.

.NET Framework supporta da sempre la manipolazione di dati in formato XML, anzi lo utilizza anche per scopi interni, per esempio per definire dei file di configurazione delle applicazioni o per l'invio e la ricezione di dati tramite servizi web, e in una sorta di forma dialettale, nota come *XAML*, per la definizione di interfacce grafiche (soprattutto per gli ultimi sistemi operativi di casa Microsoft: Windows 8 e Windows Phone).

Nonostante la sua apparente semplicità di base, però, XML può anche diventare molto complicato. Quindi trattare dati in tale formato, qualunque linguaggio di programmazione si utilizzi, può essere un vero e proprio incubo.

Un primo insieme di tipi e API di .NET dedicati a XML si trova all'interno del namespace System.Xml e dei suoi sotto-namespace, e permette di interagire con questo formato direttamente a basso livello, trattandone i vari elementi che ne costituiscono un documento, come nodi, attributi e così via, sia in fase di creazione sia in fase di lettura.

Per interrogare e ricavare informazioni da un documento XML, sarà poi possibile sfruttare le implementazioni dello standard *XPath*, che è un linguaggio appositamente pensato per scrivere query da eseguire sui dati XML. Con l'apparizione di LINQ, e quindi a partire da .NET 3.5, l'interrogazione di dati XML è divenuta più semplice grazie al provider dedicato, chiamato *LINQ to XML*.

In questo capitolo vedremo come lavorare con XML utilizzando tutti gli approcci, ma prima introdurremo in maniera molto rapida tale metalinguaggio e il modo di rappresentarne i dati.

Chi ha già una certa dimestichezza con XML o perlomeno lo conosce come linguaggio, può anche saltare il prossimo paragrafo e andare direttamente al sodo, passando all'utilizzo delle classi e dei metodi forniti da .NET per lavorare con documenti e dati XML.

Documenti XML

Un *documento* XML è un insieme di dati che segue la struttura, il formato e la sintassi definiti dal linguaggio XML. Questi dati, e quindi i documenti XML, possono essere letti e scritti in un file di testo, oppure essere manipolati anche direttamente in memoria, per esempio conservandoli in una stringa.

Un documento XML è costituito da diversi *elementi*, che possono essere a loro volta di diverse tipologie e che possono contenere a loro volta diversi altri sottoelementi, dando quindi all'intero documento una struttura gerarchica o ad albero.

Ogni elemento XML consiste di un *tag* di apertura e uno di chiusura, che racchiudono i dati contenuti nell'elemento.

Il tag di apertura racchiude fra parentesi angolari il nome dell'elemento. Per esempio:

```
<veicolo>
```

Il tag di chiusura è uguale a quello di apertura, con l'aggiunta di un carattere slash (/) dopo la prima parentesi angolare:

```
</veicolo>
```

Quindi un elemento che rappresenti un veicolo, contenente come dati il modello di un'automobile, potrebbe essere il seguente:

```
<veicolo>Alfa Romeo GT</veicolo>
```

Si faccia attenzione alle minuscole e maiuscole: <VEICOLO> è diverso da <veicolo>, quindi, chiudendo con </VEICOLO> il tag <veicolo>, il documento XML non sarà considerato valido.

NOTA	Chi conosce HTML noterà certamente la somiglianza nella modalità di scrittura dei tag. La differenza principale è che un linguaggio come HTML, espressamente dedicato alla creazione di documenti ipertestuali, definisce i vari tag utilizzabili, in XML invece non esistono elementi predefiniti e ognuno sceglie i nomi da assegnare ai loro per definire una struttura dati.

Ogni elemento può contenere al suo interno altri elementi, in maniera gerarchica. In questo modo è possibile, per esempio, includere altre informazioni all'interno di un elemento che definisce un veicolo, creando così degli elementi che definiscono marca e modello:

```
<veicolo>
<marca>Alfa Romeo</marca>
<modello>GT</modello>
</veicolo>
```

Un elemento può anche essere vuoto, per esempio se non vi è alcuna informazione associata a esso:

```
<veicolo></veicolo>
```

In questo caso è possibile abbreviare la sua scrittura aggiungendo lo slash di chiusura direttamente alla fine del tag di apertura:

```
<veicolo />
```

Oltre a poter memorizzare dati all'interno degli elementi, fra un tag di apertura e uno di chiusura, è possibile utilizzare degli attributi all'interno del tag di apertura assegnando un nome e un valore nel seguente modo:

```
<veicolo targa="AB123CD">
...
</veicolo>
```

Gli attributi possono essere considerati delle proprietà dell'elemento. Il valore dell'attributo deve essere racchiuso fra apici o fra doppi apici.

Unendo diversi elementi e attributi, ecco che abbiamo un primo esempio di documento XML completo:

```
<veicoli>
<veicolo targa="AB123CD">
<marca>Alfa Romeo</marca>
<modello>GT</modello>
</veicolo>
<veicolo targa="EF456GH">
<marca>Ferrari</marca>
<modello>F40</modello>
</veicolo>
</veicoli>
```

Ogni documento XML è quindi composto da un insieme di nodi, ma il nodo principale, o nodo radice, deve essere unico.

Per esempio, un documento come il seguente non è un documento XML valido perché ha due elementi che costituiscono entrambi i nodi radice del documento:

```
<veicolo targa="AB123CD">
<marca>Alfa Romeo</marca>
<modello>GT</modello>
</veicolo>
<veicolo targa="EF456GH">
<marca>Ferrari</marca>
<modello>F40</modello>
</veicolo>
```

L'unico nodo che può essere allo stesso livello della radice è un nodo speciale che rappresenta la *dichiarazione XML*, contenente informazioni sulla versione dello standard

XML utilizzato (1.0 è la versione corrente) e sulla codifica utilizzata per i caratteri contenuti nel documento.

L'elemento contenente la dichiarazione XML ha il seguente formato, e può essere inserito nel documento solo come primo elemento:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Inoltre, è possibile creare una particolare tipologia dei nodi di un documento XML per l'inserimento di *commenti*. Basta inserire il testo fra <!-- e -->, per esempio:

```
<!-- contiene dati sui veicoli -->
```

Namespace XML

Uno dei concetti più strani per chi si avvicina la prima volta a XML è quello di *namespace*.

Come i namespace di un programma C#, anche in XML possono essere utilizzati degli spazi di nomi per risolvere ambiguità fra oggetti omonimi, fornendo nomi univoci. Per esempio, uno stesso elemento denominato <modello> potrebbe servire a indicare il modello di un veicolo, come negli esempi precedenti, ma anche il modello di un qualunque altro oggetto da rappresentare all'interno dello stesso documento XML, e ciò quindi porterebbe a conflitti di nomi.

I namespace di XML vengono in genere assegnati utilizzando un *URI* (*Uniform Resource Identifier*), che tipicamente è quello dell'autore o dell'azienda che definisce i dati XML, in maniera da evitare conflitti di nomi fra i namespace stessi. In particolare, è possibile utilizzare un url, magari quello del proprio sito web, seguito da una stringa legata al particolare ambito dei dati XML.

Per riferirsi a un namespace si utilizza l'attributo speciale xmlns, associandolo all'elemento radice. Per esempio:

```
<veicoli xmlns="www.mionamespace.it/xmlveicoli">
```

Si noti che non è necessario che l'url punti a una pagina reale. In questo modo, il namespace di default del documento sarà quello indicato mediante l'attributo xmlns.

Un'altra possibile dichiarazione permette di assegnare un prefisso, nel formato xmlns:prefisso, in maniera da potersi riferire al namespace in oggetto mediante il solo prefisso. Per esempio:

```
<veicoli xmlns:xv="www.mionamespace.it/xmlveicoli">
```

Ogni elemento e attributo il cui nome inizia per xv: sarà così considerato come facente parte del namespace assegnato a tale prefisso:

```
<xv:veicoli xmlns:xv="www.mionamespace.it/xmlveicoli">
<xv:veicolo>
```

```
<xv:modello>GT</xv:modello>
<xv:marca>Alfa Romeo</xv:marca>
</xv:modello>
<xv:veicolo>
```

Le classi .NET che permettono di manipolare documenti XML mettono a disposizione le funzionalità necessarie per trattare anche gli spazi dei nomi XML.

XML DOM

L'*XML DOM* (*Document Object Model*) permette di accedere, aggiungere, rimuovere e manipolare gli elementi di un documento XML, rappresentandolo mediante una struttura ad albero.

NOTA	Il DOM è uno standard definito dal consorzio *W3C* (*World Wide Web Consortium*) ed è un'interfaccia indipendente da piattaforma e linguaggio che consente a programmi e script di accedere e modificare il contenuto, la struttura e lo stile di un documento. Ogni linguaggio o piattaforma, come .NET, può quindi fornire una o più implementazioni proprie del DOM.

In accordo a tale modello, tutto in un documento XML è un *nodo*: l'intero documento è un nodo, ogni elemento è un nodo, il testo contenuto in un elemento e ogni suo attributo sono nodi.

.NET fornisce una propria implementazione dell'XML DOM, contenuto all'interno del namespace System.Xml. Ogni nodo di un documento XML è rappresentato da una figlia della classe astratta XmlNode, mentre un documento è rappresentato dalla classe XmlDocument (che deriva anch'essa da XmlNode, visto che, come detto, anche un documento è un nodo).

La classe XmlDocument

La classe XmlDocument espone i metodi per leggere un file in memoria o per salvare un documento XML su un file. Una volta caricato un file o letta una stringa in formato XML, tramite altri metodi della classe sarà possibile creare, eliminare, modificare nodi.

Per creare un documento XML a partire da un file esistente, basta utilizzare il metodo Load:

```
XmlDocument doc = new XmlDocument();
string filename= "myfile.xml";
doc.Load(filename);
```

Il parametro accettato dal metodo Load può essere una stringa rappresentante il percorso di un file, oppure un oggetto Stream, un TextReader, o un XmlReader.

Se si ha una stringa, per esempio una variabile, già in formato XML, per caricarla in un XmlDocument si deve utilizzare il metodo LoadXml:

```
string xml = @"<?xml version=""1.0"" encoding=""utf-8"" ?>
<!--contiene dati sui veicoli-->
<veicoli>
<veicolo targa=""AB123CD"">
<marca>Alfa Romeo</marca>
<modello>GT</modello>
</veicolo>
</veicoli>";
```

```
doc.LoadXml(xml);
```

Al contrario, creato un documento XML, oppure modificatone uno esistente caricato come visto in precedenza, per esempio con l'aggiunta di nodi o la modifica di quelli già esistenti, esso può essere salvato mediante il metodo Save; tale quale può essere utilizzato specificando il percorso del file da salvare, oppure un oggetto Stream, un TextWriter o un XmlWriter:

```
doc.Save(filename);
```

Per esplorare e leggere i vari nodi di un XmlDocument esistono diverse possibilità. Il seguente esempio mostra come attraversare e leggere i vari elementi del documento XML caricato in precedenza, a partire dall'XmlElement di primo livello, che si ottiene per mezzo della proprietà DocumentElement:

```
XmlElement root= doc.DocumentElement;
```

La proprietà FirstChild di ogni XmlElement ricava il primo nodo figlio:

```
XmlNode nodeVeicolo= root.FirstChild;
```

Se un nodo ha degli attributi, possiamo ricavarne la collezione mediante la proprietà Attributes, che a sua volta permette di accedere a ognuno di essi mediante il nome o l'indice. Il valore di ogni attributo è accessibile per mezzo della proprietà Value:

```
XmlAttribute attrTarga = nodeVeicolo.Attributes["targa"];
string targa=attrTarga.Value;
```

A questo punto, si può procedere per leggere i nodi interni del nodo veicolo, utilizzando ancora FirstChild:

```
XmlNode nodeMarca = nodeVeicolo.FirstChild;
```

Il valore di ogni elemento, cioè il testo contenuto fra il tag di apertura e quello di chiusura, è accessibile mediante la proprietà InnerText:

```
string marca = nodeMarca.InnerText;
```

La proprietà InnerXml permette invece di ricavare direttamente il markup XML di tutti gli elementi figli. Per esempio, InnerXml di modeMarca sarebbe coincidente con il valore ottenuto mediante InnerText, ma se leggessimo InnerXml di nodeVeicolo il risultato sarebbe il frammento XML seguente:

```
<marca>Alfa Romeo</marca><modello>GT</modello>
```

Se interessa ottenere l'XML comprensivo dell'elemento contenitore, basta invece utilizzare la proprietà OuterXml, che restituirebbe per esempio:

```
<veicolo targa=""AB123CD"">
<marca>Alfa Romeo</marca>
<modello>GT</modello>
</veicolo>
```

Per passare da un nodo a uno immediatamente successivo, di pari livello, per esempio da quello contenente la marca a quello con il modello, si utilizza la proprietà NextSibling:

```
XmlNode nodeModello = nodeMarca.NextSibling;
```

Ogni nodo, cioè ogni istanza della classe XmlNode e le sue derivate, possiede anche una proprietà NodeType, del tipo enumerazione XmlNodeType, che permette di conoscere il tipo di elemento in esame. Per esempio, un elemento come nodeMarca sarà di tipo Element, mentre un attributo sarà un nodo di tipo Attribute e così via.

Con le stesse proprietà è possibile modificare i valori di attributi e il testo contenuto in ogni elemento. Per esempio, volendo modificare il valore dell'elemento modello, potremmo scrivere:

```
nodeModello.InnerText="Spider";
```

La classe XmlDocument permette così di creare un documento XML *ex novo*, aggiungendo nodi e impostando i loro attributi ed elementi.

Il seguente esempio ricrea l'XML del precedente esempio:

```
XmlDocument doc2 = new XmlDocument();
XmlDeclaration decl = doc2.CreateXmlDeclaration("1.0", "UTF-8", null);
doc2.AppendChild(decl);

XmlElement nodCommento= doc2.CreateElement("<!-- contiene dati sui veicoli -->");

XmlElement nodVeicoli= doc2.CreateElement("veicoli");

XmlElement nodVeicolo = doc2.CreateElement("veicolo");
nodVeicolo.SetAttribute("targa", "AB123CD");
XmlElement nodMarca=doc2.CreateElement("marca");
nodMarca.InnerText="Alfa Romeo";
nodVeicolo.AppendChild(nodMarca);
XmlElement nodModello = doc2.CreateElement("modello");
nodModello.InnerText = "GT";
nodVeicolo.AppendChild(nodModello);

nodVeicoli.AppendChild(nodVeicolo);
doc2.AppendChild(nodVeicoli);
```

Come avrete notato, il documento viene creato dal basso verso l'alto: vengono creati i singoli nodi, aggiunti ai propri nodi genitori, fino ad arrivare al nodo radice.

Ogni elemento può essere creato mediante il generico metodo CreateElement, ma esistono altri metodi del tipo CreateXXX riservati a particolari tipi di nodi. Si veda, per esempio, il nodo di testo creato nell'esempio precedente:

```
XmlElement nodModello = doc2.CreateElement("modello");
nodModello.InnerText = "GT";
```

È un nodo di testo che può essere direttamente creato e aggiunto come figlio di nodModello con il metodo CreateTextNode:

```
XmlText textNode= doc2.CreateTextNode("GT");
nodModello.AppendChild(textNode);
```

Analogamente, un attributo può essere creato con il metodo CreateAttribute, un commento con CreateComment e così via. In tal maniera, non sarà necessario ricordarsi di utilizzare la sintassi per ogni particolare tipo di nodo. Per esempio, il nodo di commento che va inserito fra <!-- e -->, se si utilizza il generico CreateElement, può essere creato specificando solo la stringa di commento:

```
XmlComment commentNode=doc.CreateComment("contiene i dati sui veicoli");
doc.AppendChild(commentNode);
```

Nell'esempio, inoltre, è sempre usato AppendChild per aggiungere un nodo come figlio di un nodo padre. Peraltro, sono tuttavia utilizzabili anche i metodi PrependChild, InsertBefore e InsertAfter.

NOTA	I documenti XML prevedono una codifica particolare per alcuni caratteri. In particolare, quelli utilizzati per definire la struttura e i tag del documento, come < e >, non possono essere usati direttamente all'interno di un elemento. Le classi .NET, com'è naturale, permettono di effettuare automaticamente la codifica. Per esempio < e > vengono codificati in < e > rispettivamente. In caso contrario, l'XML non risulterebbe valido.

Leggere e scrivere XML

XmlReader e XmlWriter sono due classi astratte che forniscono modalità di accesso differenti a file e dati XML rispetto al modello basato su DOM della classe XmlDocument. Esse permettono di leggere e scrivere gli elementi di un documento XML in maniera molto veloce, forward-only e senza cache, quindi molto efficace dal punto di vista delle performance.

La classe XmlReader

XmlReader è una classe astratta che fornisce l'interfaccia per la lettura di dati XML. Da essa sono derivate le classi concrete XmlTextReader, XmlNodeReader e anche XmlValidatingReader.

Per creare un'istanza della classe astratta XmlReader, è necessario utilizzare il metodo Create oppure istanziare direttamente una delle classi da essa derivate:

```
XmlReader reader=XmlReader.Create(new StreamReader(filename));
```

I metodi di lettura di XmlReader permettono di scorrere nodo per nodo l'intero documento XML. Per esempio, il metodo Read in generale continuerà ad avanzare una sorta di cursore finché

esisteranno nodi da leggere, restituendo un bool, che indica se si è letto un nuovo nodo o se si è raggiunta la fine. Quindi, un ciclo while può essere utilizzato per leggere l'intero documento in questo modo:

```
while(reader.Read())
{
…
}
```

Quando il metodo Read restituisce false, si è raggiunta la fine del documento, quindi si può chiudere l'XmlReader. Per questo motivo è consigliabile utilizzare la classe con un'istruzione using:

```
using(XmlReader reader=XmlReader.Create(new StreamReader(filename)))
{
while(reader.Read())
{
}
}
//reader chiuso
```

Ogni nodo, come già visto, può essere di un determinato tipo, ricavabile dalla proprietà NodeType e che può assumere uno dei valori dell'enumerazione XmlNodeType.

A questo punto, per mezzo delle proprietà di XmlReader, per esempio Name e Value, di tipo string, è possibile leggere l'eventuale nome dell'elemento e l'eventuale valore in esso contenuto. Non tutti i nodi restituiscono una stringa non vuota per entrambe le proprietà, dipende appunto dal tipo di nodo sotto esame. Un nodo di testo, per esempio, ha un valore, ma non un nome, mentre un attributo possiede entrambi.

Il seguente esempio ricrea la struttura testuale ricavata da un file XML e utilizza la proprietà Depth, che restituisce la profondità del nodo corrente per indentare correttamente il risultato:

```
using(XmlTextReader txtReader = new XmlTextReader(filename))
{
while (txtReader.Read())
{
Console.Write(new string(' ', txtReader.Depth));
Console.WriteLine("{0}: {1} {2}", txtReader.NodeType, txtReader.Name, txtReader.Value);
}
}
```

Eseguendo il codice precedente con l'XML di esempio creato negli scorsi paragrafi, si otterrà il seguente risultato:

```
XmlDeclaration: xml version="1.0" encoding="utf-8"
Whitespace:

Comment: contiene dati sui veicoli

Whitespace:
```

```
Element: veicoli
Whitespace:

Element: veicolo
Whitespace:

Element: marca
Text: Alfa Romeo
EndElement: marca
Whitespace:

Element: modello
Text: GT
EndElement: modello
Whitespace:

EndElement: veicolo
Whitespace:

EndElement: veicoli
```

Si noti che anche gli spazi bianchi, le tabulazioni e le interruzioni di linea sono dei particolari nodi, di tipo Whitespace. Infatti, dopo ogni elemento di tale tipo, vi è una riga vuota (il valore del nodo). Con un oggetto XmlReaderSettings è possibile utilizzare particolari impostazioni di lettura, per esempio ignorare tali spazi bianchi o i commenti:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreComments = true;
settings.IgnoreWhitespace = true;
using (XmlReader reader = XmlReader.Create(new StreamReader(filename), settings))
{
...
}
```

Altri metodi permettono di navigare la struttura del documento XML, spostandosi da un nodo all'altro, oppure analizzando attributi e valori. Per esempio, una volta posizionati sul nodo di tipo XmlDeclaration, possiamo stamparne i valori di tutti gli attributi spostandoci da uno all'altro con il metodo MoveToNextAttribute come segue:

```
using (XmlReader reader = XmlReader.Create(new StreamReader(filename), settings))
{
while (reader.Read())
{
switch(reader.NodeType)
{
case XmlNodeType.XmlDeclaration:
while (reader.MoveToNextAttribute())
{
Console.Write("{0}={1}", reader.Name, reader.Value);
}
Console.WriteLine();
break;
…
```

Ciò presuppone che in generale bisogna conoscere la struttura dell'XML per poterlo navigare correttamente.

La classe XmlReader può essere utilizzata per leggere in sequenza anche da sorgenti potenzialmente lente, come file di grossa dimensione, magari ricavati direttamente da url remoti. Essa espone, quindi, i metodi appena visti anche in versione asincrona, in maniera da non bloccare, per esempio, l'interfaccia grafica durante la lettura o la scrittura di un documento XML.

Se si intende usufruire delle versioni asincrone dei metodi di XmlReader, è necessario impostare la proprietà Async a true nell'oggetto XmlReaderSettings:

```
XmlReaderSettings settings = new XmlReaderSettings()
{
Async = true
};
```

A questo punto, è possibile usare i metodi asincroni corrispondenti a quelli sincroni già visti, facendoli precedere dalla parola chiave await (vedere il Capitolo 12). Per esempio, il metodo asincrono corrispondente a Read è il metodo ReadAsync.

Il seguente esempio legge il feed RSS (per chi non lo sapesse, è un formato per la distribuzione di contenuti basato su XML) del mio sito e poi ne ricava titoli e data dei post, utilizzando un altro metodo asincrono, ReadElementContentAsStringAsync, per leggere il contenuto degli elementi title e pubDate:

```
StringBuilder sb = new StringBuilder();
using (XmlReader reader = XmlReader.Create("http://antoniopelleriti.it/syndication.axd", settings))
{
try
{
while (await reader.ReadAsync())
{
switch (reader.NodeType)
{
case XmlNodeType.Element:
{
if (reader.Name == "title" )
{
sb.AppendLine(await reader.ReadElementContentAsStringAsync());
}
else if (reader.Name == "pubDate")
{
sb.AppendLine(await reader.ReadElementContentAsStringAsync());
sb.AppendLine();
}
break;
}
}
```

```
}
}
catch (Exception ex)
{
MessageBox.Show(ex.Message);
}
}
```

Le stringhe lette vengono aggiunte a uno StringBuilder (vedere l'Appendice A per l'uso di StringBuilder) in attesa, per esempio, di essere visualizzate dove previsto da una applicazione. Tale metodo potrebbe essere usato come base per creare un lettore RSS.

La classe XmlWriter

La controparte della classe XmlReader per scrivere e generare flussi di dati in formato XML è la classe XmlWriter. Anche in questo caso, è possibile creare delle impostazioni di scrittura, per esempio per utilizzare l'indentazione, mediante la classe XmlWriterSettings:

```
XmlWriterSettings ws=new XmlWriterSettings(){ Indent=true, Encoding=UTF8Encoding.UTF8};
```

XmlWriter aggiunge in modo automatico la dichiarazione XML ai documenti, a meno che non si specifichi esplicitamente di ometterla mediante la proprietà OmitXmlDeclaration di XmlWriterSettings. Una volta creata un'istanza di XmlWriter, è possibile utilizzare una serie di metodi per generare le varie parti di un documento XML. Il file di esempio precedente può essere ricreato con un XmlWriter come di seguito:

```
using (XmlWriter writer = XmlWriter.Create("generated.xml", ws))
{
writer.WriteComment("questa è una prova");
writer.WriteStartElement("veicoli");
writer.WriteStartElement("veicolo");
writer.WriteAttributeString("targa", "AB123DC");
writer.WriteElementString("marca", "Alfa Romeo");
writer.WriteElementString("modello", "GT");
writer.WriteEndElement();
writer.WriteEndElement();
}
```

La classe XmlWriter, come già detto, è astratta. Pertanto, nel caso precedente, si è usato il metodo statico Create per ottenerne un'istanza. Un'altra possibilità è usare la classe concreta da essa derivata XmlTextWriter.

XPath

XPath è uno standard W3C per l'interrogazione di dati in formato XML. Esso definisce un linguaggio per la definizione di query, le quali consentono di navigare un documento XML e ricavarne nodi che rispettano determinati criteri.

Il nome è l'acronimo di *XML Path Language*, che deriva, come vedremo a breve, dalla modalità di scrittura delle espressioni che vengono basate sul percorso dei nodi in un documento XML.

NOTA	L'argomento "XPath" è abbastanza complesso e sicuramente meriterebbe molto più spazio, ma in un testo dedicato completamente a XML. In questo libro si è focalizzata maggiormente l'attenzione su LINQ to XML, che permette di eseguire interrogazioni su documenti XML in maniera altrettanto potente e soprattutto più semplice.

Per mostrare le capacità di XPath utilizzeremo il seguente file XML:

```
<?xml version="1.0" encoding="utf-8" ?>
<veicoli>
<veicolo targa="AB123CD">
<marca>Alfa Romeo</marca>
<modello>GT</modello>
<alimentazione tipo="Gasolio">
<consumo>14</consumo>
</alimentazione>
</veicolo>
<veicolo targa="EF456GH">
<marca>Alfa Romeo</marca>
<modello>Spider</modello>
</veicolo>
<veicolo targa="AB999ZZ">
<marca>FIAT</marca>
<modello>UNO</modello>
</veicolo>
</veicoli>
```

Per poter eseguire delle query XPath su un documento XML, è possibile seguire diverse strade. La prima è invocare i metodi SelectNodes e SelectSingleNode della classe XmlDocument, passando come parametro la stringa contenente l'espressione XPath e ottenendo come risultato, rispettivamente, una lista di nodi oppure il primo dei nodi che soddisfano la query.

Per esempio, a partire dall'oggetto XmlDocument:

```
XmlDocument doc=new XmlDocument();
doc.Load("myfile.xml");
```

possiamo ricavare l'elenco dei nodi con nome veicolo e figli di un nodo padre con nome veicoli:

```
XmlNodeList nodiVeicoli = doc.SelectNodes("veicoli/veicolo");
```

e quindi stamparne l'attributo targa ricavandolo da ogni XmlNode, come visto nei precedenti paragrafi:

```
foreach (XmlNode nv in nodiVeicoli)
{
Console.WriteLine(nv.Attributes["targa"].Value);
}
```

La stringa passata come parametro al metodo SelectNodes è un'espressione XPath. Nel caso specifico, essa indica il percorso da ricercare nel documento. Quindi, partendo dal nodo veicoli, si passa a tutti i nodi figli con nome veicolo, ricavando per esempio dal nostro file esattamente tre nodi, di ognuno dei quali viene poi letto l'attributo targa all'interno del ciclo foreach.

Nei prossimi paragrafi vedremo come poter scrivere espressioni XPath più complesse utilizzando una specifica sintassi e specifici operatori. Intanto vediamo dove è possibile utilizzare le query XPath, oltre ai suddetti metodi di XmlDocument.

Il namespace System.Xml.XPath

Il namespace che contiene le classi necessarie a lavorare con XPath è denominato System.Xml.XPath. La classe fondamentale in questo caso è XPathNavigator, che fornisce un modello a cursore per navigare e modificare i nodi di un documento XML. Un oggetto XPathNavigator è istanziabile mediante il metodo CreateNavigator di XmlDocument, se si necessita di funzionalità di modifica, oppure tramite l'omonimo metodo esposto da XPathDocument.

La suddetta classe XPathDocument è una classe il cui unico scopo è permettere la navigazione dei documenti XML in sola lettura; di conseguenza le sue prestazioni e quelle del navigatore ricavato dalle sue istanze saranno notevolmente migliori rispetto a XmlDocument. Il rovescio della medaglia è che un XPathDocument carica interamente un documento XML in memoria e lo analizza, a differenza per esempio di XmlReader che utilizza un modello di lettura sequenziale di ogni singolo nodo.

Per creare un XPathDocument è possibile utilizzare un file esistente, oppure partire da un oggetto XmlReader, o ancora passare al costruttore uno stream. Per esempio, se si ha a disposizione un file basta passarne il percorso al costruttore come segue:

```
XPathDocument xpDoc=new XPathDocument("file.xml");
```

Oppure, avendo a disposizione il contenuto XML sotto forma di stringa, basta utilizzare un oggetto StringReader:

```
string str="<?xml version="1.0" encoding="utf-8" ?> ... "; //stringa xml da completare
```

```
StringReader reader=new StringReader(strXml);
XPathDocument doc=new XPathDocument(reader);
```

La classe XPathNavigator espone tutti i metodi per muoversi fra i nodi del documento o per selezionarne di specifici mediante una query XPath.

I metodi di selezione sono quelli elencati nella Tabella 13.1.

Tabella 13.1 - **Metodi di selezione di XPathNavigator.**

Metodo	Descrizione
Select	seleziona una lista di nodi utilizzando una query XPath
SelectAncestors	seleziona i nodi antenati
SelectChildren	seleziona i figli di un nodo
SelectDescendants	seleziona i discendenti di un nodo
SelectSingleNode	seleziona un singolo nodo utilizzando una query XPath

I metodi di selezione che restituiscono insiemi di nodi hanno come tipo di ritorno un oggetto XPathNodeIterator, che rappresenta un iteratore su una lista di nodi. I metodi come SelectSingleNode, che restituiscono un singolo elemento, hanno invece come tipo di ritorno un altro XPathNavigator, posizionato sul nodo che rappresenta il risultato stesso. Altri metodi di XPathNavigator, con nomi del tipo MoveToXXX, permettono di spostare il navigatore su un determinato nodo. Per esempio, sul primo figlio, sul prossimo attributo, sul seguente nodo.

Come già detto, un XPathNavigator creato da un XmlDocument può essere utilizzato anche per modificare i nodi di un documento XML. Per verificare se un XPathNavigator è utilizzabile in modalità di modifica, basta utilizzare la proprietà CanEdit. Nel caso in cui essa sia true, si potranno utilizzare metodi come InsertBefore e InsertAfter, che inseriscono un nuovo nodo, oppure metodi per eliminare o rimpiazzare nodi esistenti.

Sintassi XPath

XPath permette di eseguire delle interrogazioni su un documento XML scrivendo delle query che utilizzano una particolare sintassi e particolari operatori.

Le espressioni XPath specificano un percorso, come indica il nome completo XML Path Language, per riferirsi a specifiche parti di un documento XML. Ogni espressione, una volta valutata, può restituire un insieme di nodi, un bool, un numero o una stringa. Per esempio, l'espressione veicolo/marca già utilizzata in precedenza restituirà l'insieme degli elementi <marca> contenuti in un elemento <veicolo>.

In più, è possibile creare espressioni più articolate utilizzando dei filtri o delle funzioni. Per esempio, il risultato ottenuto dalla precedente espressione può essere limitato in maniera da

restituire solo i veicoli con l'attributo targa uguale al valore "AB123CD", scrivendo veicolo[@targa="AB123CD"].

Contesto di esecuzione

La valutazione e, quindi, il risultato di un'espressione dipendono dal contesto in cui essa viene eseguita, determinato dalla posizione corrente all'interno del documento XML. Per esempio, un'espressione potrà restituire un dato insieme di nodi se il contesto attuale è costituito dalla radice del documento, mentre se si parte da un nodo più in profondità il risultato sarà probabilmente un sottoinsieme, visto che i nodi su cui la query agirà saranno a loro volta un insieme ristretto rispetto a quelli dell'intero documento. Per mezzo di alcuni operatori, comunque, è possibile specificare e modificare il contesto di esecuzione.

I seguenti esempi mostrano alcune espressioni di base, che utilizzano operatori di contesto. Il carattere punto (.) indica il contesto corrente, mentre il carattere slash (/) indica tutti i nodi figli a partire dal contesto corrente. Per esplicitare, quindi, che il contesto da utilizzare è quello corrente e si vuole agire sui nodi figli, un'espressione può essere preceduta da un punto e uno slash (./), come in questo esempio:

```
./veicolo
```

Tale espressione si riferisce a tutti gli elementi <veicolo> nel contesto corrente: essa è equivalente a scrivere solo veicolo, ma la presenza di ./ consente di esplicitare il contesto, soprattutto se usati all'interno di un'espressione più complessa.

Creiamo per esempio un XPathNavigator in sola lettura, supponendo che il documento XML sia salvato nel file myfile.xml:

```
XPathDocument xpdoc = new XPathDocument("myfile.xml");
XPathNavigator nav= xpdoc.CreateNavigator();
```

A questo punto, con il metodo MoveToFirstChild è possibile spostarsi sul primo nodo del documento, l'elemento <veicoli>:

```
nav.MoveToFirstChild(); //contesto passa a primo nodo <veicoli>
```

Invocando ora il metodo Select con l'espressione precedente come parametro, si otterranno tutti gli elementi veicolo figli dell'elemento corrente veicoli:

```
XPathNodeIterator result= nav.Select("./veicolo"); //seleziona tutti i figli <veicolo>
foreach (XPathNavigator node in result)
{
Console.WriteLine(node.OuterXml);
}
```

Il ciclo foreach stampa ognuno degli elementi ottenuti (con il file di esempio saranno tre).

Se poi nell'espressione ci si vuole riferire direttamente alla radice del documento e ottenere quindi gli elementi figli a partire da questa, basta far iniziare l'espressione con l'operatore / senza punto. Per esempio, per riferirsi all'elemento <veicoli> che si trova alla radice, si può utilizzare l'espressione:

```
/veicoli
```

Un operatore molto utile, formato da un doppio slash (//) consente invece di eseguire una ricerca ricorsiva all'interno dell'intero documento a partire dall'elemento corrente. Per esempio, per trovare tutti gli elementi <veicolo> a partire dalla radice del documento e a qualunque livello di profondità, si scriverà:

```
var result=nav.Select(".//veicolo");
```

Espressioni e operatori XPath

Le espressioni XPath sono costruite utilizzando degli operatori e dei caratteri speciali mostrati nella seguente tabella.

Tabella 13.2 - **Operatori principali XPath.**

Operatore	Descrizione
/	selezione dei figli
//	selezione ricorsiva dei figli
.	indica il nodo nel contesto corrente
..	indica il padre del nodo corrente
*	Wildcard: selezione di tutti gli elementi a prescindere dal nome
@	prefisso per indicare un attributo
@*	Wildcard per attributi: seleziona tutti gli attributi a prescindere dal nome
:	separatore di namespace
( )	raggruppamento delle operazioni per esplicitare la precedenza
[ ]	applica un filtro
[ ]	operatore per indicizzare i nodi di una collezione
|	restituisce l'unione di due insiemi

Nel seguito, qualche esempio di utilizzo degli operatori per la scrittura di espressioni XPath:

```
//tutti gli elementi <veicolo> con un figlio <alimentazione>
var results = nav.Select("//veicolo[alimentazione]");

//tutti gli elementi marca figli di un elemento <veicolo> con un figlio <alimentazione>
var results = nav.Select("//veicolo[alimentazione]/marca");

//tutti gli elementi <veicolo> con figli <marca> e <modello>
var results = nav.Select("//veicolo[marca][modello]");
```

```
//tutti gli elementi <veicolo> con attributo targa
var results = nav.Select("//veicolo[@targa]");
```

Se si vuol ottenere l'unione di due diversi insiemi di risultati, è possibile utilizzare l'operatore di concatenamento |, per esempio:

```
//seleziona tutti gli elementi <marca> e tutti gli elementi <modello> contenuti in un elemento <veicolo>
var results = nav.Select("//veicolo/marca | //veicolo/modello");
```

Filtri in XPath

L'operatore [] di filtro (differente da quello di indicizzazione) permette di specificare delle condizioni di ricerca da applicare a una selezione. Tali condizioni, a loro volta, possono essere scritte utilizzando gli operatori di confronto =, !=, <, > e quelli logici and, or, not(). Ecco qualche esempio:

```
//tutti gli elementi <veicolo> con attributo targa = "AB123CD"
var results = nav.Select("//veicolo[@targa=\"AB123CD\"]");

//tutti gli elementi <marca> figli di <veicolo> con attributo targa = "AB123CD"
var results = nav.Select("//veicolo[@targa=\"AB123CD\"]/marca");

//selezione gli elementi <veicolo> con sottoelementi marca e modello ma senza alimentazione
var results = nav.Select("//veicolo[marca and modello and not(alimentazione)]");
```

LINQ to XML

LINQ to XML è la nuova modalità per interagire con dati in formato XML senza dover passare dal DOM di XmlDocument, utilizzare metodi sequenziali come XmlReader e XmlWriter, scrivere query con XPath.

LINQ to XML fornisce un'interfaccia di programmazione che permette di formulare ed eseguire delle query LINQ (vedere il Capitolo 11) per interrogare sorgenti di dati XML, ma anche le classi per poter creare e modificare documenti XML.

NOTA	LINQ to XML non è un LINQ Provider in senso stretto. Esso fornisce un modello a oggetti per i dati in formato XML, che rende più semplice creare documenti, interagire con essi ed eseguire query seguendo la sintassi LINQ.

Creare XML

LINQ to XML fornisce un Document Object Model, chiamato *X-DOM,* e un insieme di operatori supplementari per definire i nodi di un documento XML in maniera molto più semplice e intuitiva di quanto non sia possibile fare con XmlDocument. Infatti, uno degli obiettivi che LINQ to XML si pone è quello di fornire un approccio più orientato agli oggetti rispetto a quello di XmlDocument, che invece si può considerare orientato a XML e ai suoi nodi.

LINQ to XML è quindi basato su un insieme di classi (i cui nomi iniziano tutti per X) che rappresentano ognuna i possibili tipi di nodi che costituiscono un documento XML.

Queste classi sono contenute nel namespace System.Xml.Linq e la loro gerarchia è mostrata in Figura 13.1.

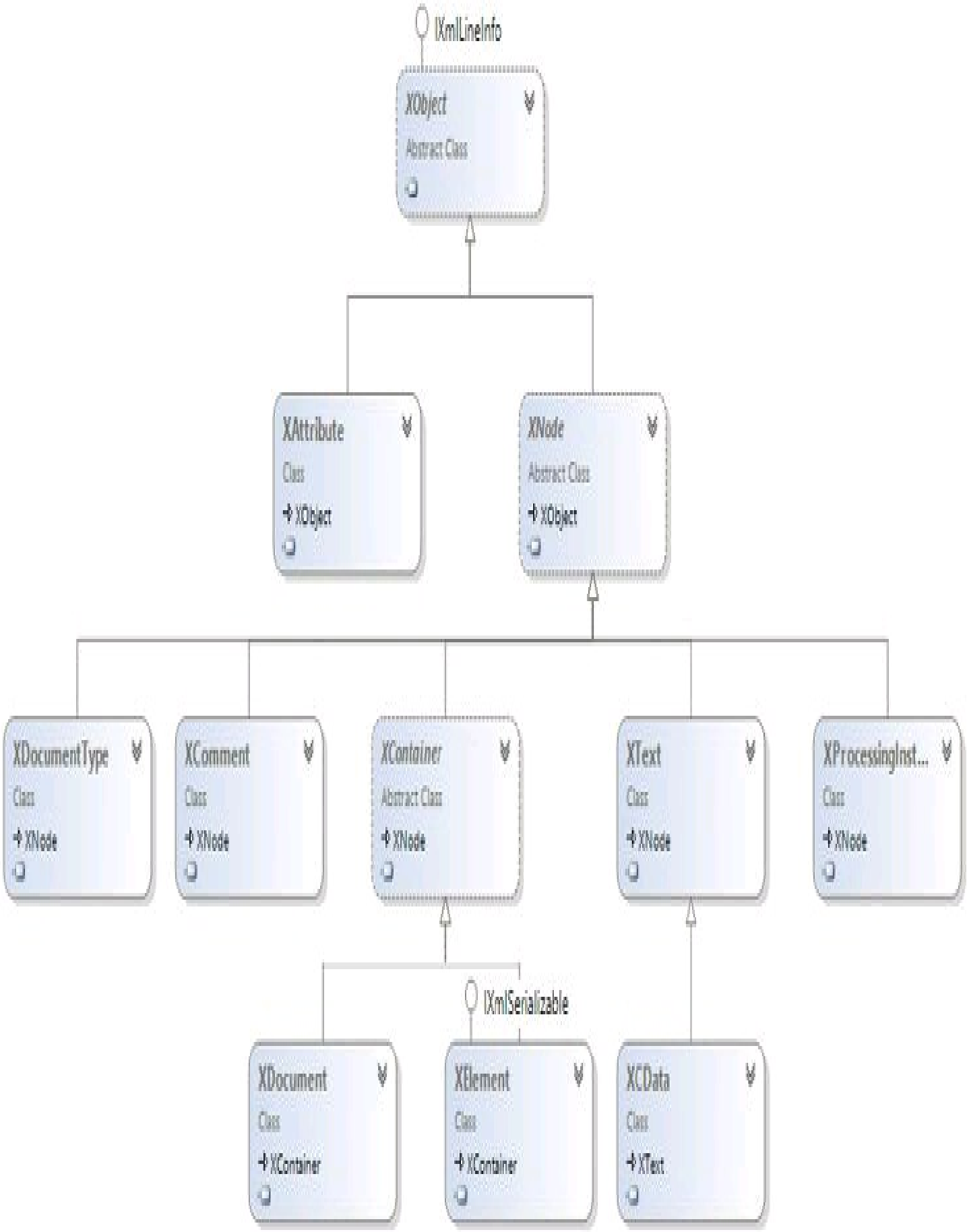

IXmlLineInfo
XObject
Abstract Class
XAttribute
Class
XObject
XNode
Abstract Class
XObject
XDocumentType
Class
XNode
XComment
Class
XNode
XContainer
Abstract Class
XNode
XText
Class
XNode
XProcessingInst...
Class
XNode
IXmlSerializable
XDocument
Class
XContainer
XElement
Class
XContainer
XCData
Class
XText

Figura 13.1 – Gerarchia delle classi di LINQ to XML.

Delle altre classi non presenti in figura, tra le più importanti citiamo XName e XNamespace, che rappresentano rispettivamente il nome di un elemento XML o di un attributo e un namespace XML. Utilizzando XDocument e le altre classi LINQ to XML, in particolare XElement, è semplice ricreare il documento XML di esempio usato finora:

```
XDocument xdoc = new XDocument(
new XDeclaration("1.0", "utf-8", null),
new XComment("contiene dati sui veicoli"),
new XElement("veicoli",
new XElement("veicolo", new XAttribute("targa", "AB123CD"),
new XElement("marca", "Alfa Romeo"),
new XElement("modello", "GT")
)
)
);

xdoc.Save("xfile.xml");
Console.WriteLine(xdoc); //il metodo ToString di XDocument stampa i dati in formato xml
```

Le classi LINQ to XML sono, come potete notare, totalmente object oriented e permettono in maniera molto immediata di creare un documento XML della struttura desiderata. A differenza del DOM di XmlDocument, non è necessario creare preventivamente i singoli nodi, aggiungendoli al proprio nodo padre, e quindi collegarli in maniera da ricreare la struttura.

Se date un'occhiata alla documentazione o all'intellisense di Visual Studio, noterete che i costruttori delle classi LINQ to XML utilizzati, per esempio quelli di XDocument e di XElement, non hanno fra i propri parametri degli oggetti di tipo string, ma nella maggioranza dei casi vorrebbero delle istanze di XName.

Questa classe non ha costrutti pubblici ma fornisce una conversione implicita da string che consente di creare le sue istanze. L'utilizzo tipico è quindi quello visto nel precedente esempio, che consiste nello specificare una stringa come parametro dove è previsto un XName.

La classe XDeclaration rappresenta la dichiarazione XML, che può essere aggiunta a un XDocument come nell'esempio sopra, oppure associata e letta mediante la proprietà Declaration della stessa classe.

La classe XComment consente di aggiungere rapidamente commenti a un documento XML. Un oggetto XComment può essere aggiunto come nodo figlio di un qualsiasi elemento oppure, come fatto nell'esempio sopra, direttamente come elemento di pari livello dell'elemento radice.

Consideriamo le istruzioni seguenti:

```
root = new XElement("root",
new XComment("commento di prova")
```

```
);
```

Esse producono questo XML:

```
<root>
<!--commento di prova-->
</root>
```

Per specificare un namespace si utilizza l'operatore di addizione, per il quale è definito un apposito overload che agisce su un XNamespace e una stringa per restituire un XName:

```
XNamespace ns = "http://www.mionamespace.it";
XElement root = new XElement(ns + "veicoli",
new XElement(ns + "veicolo")
);
```

Il quale produrrà un output come il seguente:

```
<veicoli xmlns="http://www.mionamespace.it">
<veicolo />
</veicoli>
```

Se si vogliono controllare i prefissi dei namespace, bisogna creare manualmente degli XAttribute con la dichiarazione di tali prefissi. Per farlo, è necessario assegnare all'attributo un nome composto dalla proprietà XNamespace.Xmlns e dal prefisso sotto forma di stringa, per esempio:

```
XNamespace ns = "http://www.mionamespace.it";
XElement root = new XElement(ns + "veicoli", new XAttribute(XNamespace.Xmlns+"nv", ns),
new XElement(ns + "veicolo")
);
```

In questo caso, l'XML prodotto sarebbe il seguente:

```
<nv:veicoli xmlns:nv="http://www.mionamespace.it">
<nv:veicolo />
</nv:veicoli>
```

LINQ to XML mette a disposizione un'altra possibilità: quella di creare direttamente un documento in formato XML a partire da una collezione di oggetti e utilizzare *LINQ to Objects* per ricavarne i dati.

Supponiamo per esempio di avere una classe Veicolo come la seguente:

```
public class Veicolo
{
public String Targa { get; set; }
public string Modello { get; set; }
public string Marca { get; set; }
public string Alimentazione { get; set; }
public double Consumo { get; set; }
}
```

E una lista di tali oggetti, per esempio creata come di seguito:

```
List<Veicolo> lista = new List<Veicolo>();
lista.Add(new Veicolo() { Marca = "Alfa Romeo", Modello = "GT", Targa = "AB123CD", Alimentazione = "Diesel", Consumo =
14.5 });
lista.Add(new Veicolo() { Marca = "FIAT", Modello = "Uno", Targa = "ME4563433", Alimentazione = "Benzina", Consumo=5
});
```

Volendo trasformare tale collezione in XML, basterà utilizzare LINQ per ricavare gli elementi dalla lista e costruire degli oggetti XElement, XAttribute e così via, come appena visto:

```
var xmlVeicoli = new XElement("veicoli", from v in veicoli
select new XElement("veicolo", new XAttribute("targa", v.Targa),
new XElement("marca", v.Marca),
new XElement("modello", v.Modello),
new XElement("alimentazione", new XAttribute("tipo",v.Alimentazione),
new XElement("consumo",v.Consumo)
)
)
);

Console.WriteLine(xmlVeicoli);
```

La variabile xmlVeicoli così creata, una volta convertita in string, conterrà l'XML corrispondente, con tanti elementi <veicolo> quanti sono gli oggetti presenti nella lista.

Le classi XElement e XDocument sono molto simili nel loro utilizzo, ma con sottili e importanti differenze: XDocument rappresenta un intero documento XML, normalmente composto da altri elementi; un XElement rappresenta invece un singolo elemento, che può essere dotato di attributi e figli, e che fa parte di un documento XML più ampio.

Leggere XML

LINQ to XML fornisce tutte le funzionalità necessarie per leggere e analizzare dati in formato XML.

Le classi XDocument e XElement espongono un metodo Load, il quale permette di caricare in memoria un file XML esistente indicandone il percorso, oppure di utilizzare uno stream, un TextReader o un XmlReader per leggerlo o per caricare direttamente una stringa contenente l'XML.

```
var element= XElement.Load(filename);
```

Un'altra possibilità per eseguire direttamente il *parsing*, cioè l'analisi, di una stringa che si ha già in formato XML è l'utilizzo del metodo Parse (anch'esso esposto da entrambe le classi suddette):

```
string xml=@"<veicoli>
<veicolo targa=""AB123CD""
<marca>Alfa Romeo</marca>
<modello>GT</modello>
<alimentazione tipo=""Gasolio""
<consumo>14</consumo>
</alimentazione>
</veicolo>
```

```
<veicolo targa=""EF456GH""">
<marca>Alfa Romeo</marca>
<modello>Spider</modello>
</veicolo>
<veicolo targa=""AB999ZZ""">
<marca>FIAT</marca>
<modello>UNO</modello>
</veicolo>
</veicoli>";

XElement element=XElement.Parse(xml);
```

Dopo aver creato oppure caricato in memoria un documento XML, è possibile eseguire delle query per ricavarne i vari nodi, come elementi e attributi, e i rispettivi valori. Per ricavare collezioni di nodi, si utilizzano dei metodi, detti *metodi axis*, di LINQ to XML. Alcuni di essi sono metodi di estensione della classe IEnumerable<T>, dove T è un tipo derivato da XNode o XContainer (in genere sarà XElement), contenuti nella classe Extensions, definita anch'essa all'interno dello stesso namespace System.Xml.Linq; altri sono metodi di XElement e XDocument, che restituiscono collezioni IEnumerable<T>.

Le query LINQ costruite mediante tali metodi costituiscono il modo più potente per esaminare ed estrarre dati da un documento XML. Come vedremo, esse restituiscono IEnumerable<T> di oggetti XElement oppure di XAttribute.

In aggiunta ai metodi axis suddetti, che restituiscono collezioni, altri due metodi comunemente utilizzati in LINQ to XML restituiscono invece oggetti singoli: il metodo Element restituisce un singolo XElement, mentre il metodo Attribute restituisce un singolo XAttribute.

La Tabella 13.3 riassume i metodi principali da utilizzare per l'esecuzione di query LINQ to XML.

Tabella 13.3 - **Metodi LINQ to XML.**

Metodo	Tipo ritorno	Descrizione
Ancestors	IEnumerable<XElement>	restituisce la collezione di elementi predecessori
Attributes	IEnumerable<XAttribute>	restituisce la collezione di attributi
DescendantsNodes	IEnumerable<XNode>	restituisce una collezione dei nodi discendenti
Descendants	IEnumerable<XElement>	restituisce gli elementi discendenti
Elements	IEnumerable<XElement>	restituisce la collezione di elementi figli
Nodes	IEnumerable<XNode>	restituisce la collezione di nodi figli
Parent	XElement	è una proprietà, restituisce l'elemento padre del nodo corrente
Element	XElement	restituisce il primo elemento figlio con il nome specificato
Attribute	XAttribute	restituisce l'attributo dell'elemento con il nome specificato

In generale i metodi permettono di filtrare la collezione utilizzando un XName come parametro (e quindi basta passare il nome sotto forma di string). Per esempio, in tal modo è possibile ottenere solo gli elementi o gli attributi con un dato nome e da essi poi ricavarne i rispettivi valori.

NOTA	*LINQPad*, già citato in precedenza come ottimo strumento per l'esecuzione di snippet di codice C#, è altrettanto utile, visto che nasce proprio per LINQ, per scrivere e testare le query LINQ to XML. Provate per esempio a caricare con esso dei documenti XML, usando il metodo Load, ed eseguire poi i metodi axis o le query LINQ.

Combinando i metodi e le proprietà appena viste con gli operatori o la sintassi per le query LINQ, è possibile ricavare da un documento XML qualsiasi informazione in maniera molto semplice ed efficace.

Il seguente esempio ricava tutti i valori dei modelli presenti nel file XML utilizzato finora:

```
XElement element = XElement.Parse(xml);
var queryModello = from veic in element.Descendants("veicolo")
where veic.Element("marca").Value == "Alfa Romeo"
select veic.Element("modello").Value;
```

Il metodo Descendants ricava tutti gli elementi <veicolo>, quindi mediante una query LINQ vengono selezionati i valori degli elementi <modello> che hanno come figlio l'elemento <marca> con valore "Alfa Romeo".

Naturalmente, è possibile anche ottenere direttamente gli XElement e poi eseguire un ciclo su di essi per stamparne nomi e valori:

```
var query2 = from veic in element.Descendants("veicolo")
select veic.Element("modello");

foreach (XElement el in query2)
{
Console.WriteLine("Name: {0}, Value: {1}", el.Name, el.Value);
}
```

Oppure costruire un tipo anonimo con i valori ricavati dall'XML:

```
var query3 = from veic in element.Descendants("veicolo")
select new {
Targa= veic.Attribute("targa"),
Modello = veic.Element("modello").Value,
Marca = veic.Element("marca").Value };

foreach (var obj in query3)
{
Console.WriteLine("Targa: {0}, Modello: {1}, Marca: {2}",obj.Targa, obj.Modello, obj.Marca);
}
```

L’esempio precedente utilizza anche il metodo Attribute per ricavare quello denominato "targa".

Una pratica molto comune e molto potente consiste nel concatenare più metodi di estensione fra di essi ed eventualmente anche con gli operatori standard di LINQ.

Il seguente esempio filtra gli elementi <veicolo> che hanno fra i figli degli elementi <alimentazione> con a loro volta un elemento figlio <consumo>:

```
var queryChain = from el in element.Descendants("veicolo")
let cons = el.Elements("alimentazione").Elements("consumo")
where cons.Any()
select el;
```

In particolare, viene utilizzato l’operatore Any come filtro per stabilire l’esistenza di tali tipi di elementi.

Domande di riepilogo

1) Quale istruzione permette di caricare in memoria un file XML?

a. new XmlDocument().Load(filename);
b. XmlDocument.Parse(filename);
c. new XmlDocument(filename);
d. XmlReader.Load(filename);

2) I metodi di XmlReader permettono di:

a. leggere e scrivere un documento XML
b. leggere l'intero documento XML in memoria
c. leggere nodo dopo nodo un documento XML
d. leggere solo gli attributi di ogni nodo di un documento XML

3) Lo standard XPath definisce:

a. un linguaggio per creare documenti XML
b. un linguaggio simile a XML ma più compatto
c. un linguaggio per ricavare il percorso di un nodo XML
d. un linguaggio per eseguire query su un documento XML

4) Le classi fornite da LINQ to XML per poter manipolare documenti XML sono contenute in:

a. System.Xml.Linq
b. System.Linq
c. System.Linq.Xml
d. System.Xml

5) Per effettuare il parsing di una stringa str contenente XML, si può usare:

a. XElement.Read(str)
b. XElement.Parse(str)
c. new XDocument(str)
d. new XElement(str)

Reflection, attributi e programmazione dinamica

C# possiede caratteristiche dinamiche come la reflection, con cui analizzare oggetti e generare codice a runtime, e gli attributi che consentono di aggiungere altre informazioni ai tipi. Il DLR fornisce poi l'infrastruttura necessaria al funzionamento del tipo dynamic.

La natura intrinseca degli assembly di .NET, che contengono oltre al codice eseguibile di programmi e librerie anche dei metadati descrittivi di classi, metodi, e ogni altro elemento, permette di ricavare e utilizzare tali informazioni direttamente all'interno di un'applicazione.

Il meccanismo che permette di accedere ai metadati e utilizzare i tipi e i loro membri a runtime è detto *reflection*.

Oltre a quanto detto, apposite classi consentiranno anche di creare metadati e codice IL direttamente a runtime, generando quindi dinamicamente nuovi tipi e relativo codice.

La reflection è utilizzata sia dal CLR sia da strumenti come Visual Studio per ricavare informazioni sui tipi e impostare i valori degli oggetti.

Il .NET Framework supporta inoltre i cosiddetti *attributi*, anch'essi memorizzati nei metadati, che sono in pratica delle annotazioni con cui è possibile decorare tipi, membri e altri elementi di codice, in maniera da aggiungere informazioni aggiuntive, utilizzabili per esempio dal compilatore per stabilire la modalità di generazione degli assembly, oppure per variarne il comportamento a tempo di esecuzione.

La reflection costituisce già un supporto notevole a una sorta di *programmazione dinamica*, soprattutto con la possibilità di invocare metodi scoperti a runtime, ma è divenuta realmente

integrata in C# a partire dalla versione 4.0, con l'introduzione del tipo *dynamic*. In tal modo, C# non si limita a essere un linguaggio fortemente tipizzato, ma acquisisce anche le caratteristiche di linguaggio dinamico, permettendo al compilatore di saltare la classica verifica sui tipi e delegando tale compito al CLR, che se ne occuperà a tempo di esecuzione.

Reflection

I tipi sono uno dei concetti fondamentali del .NET Framework e del funzionamento del CLR.

Ogni programma, compilato ed eseguito, è formato da oggetti di vario tipo che espongono vari membri e che il compilatore può esaminare per conoscerne le caratteristiche, stabilendo quindi se il codice stesso è scritto correttamente e può essere appunto compilato. Per esempio, supponiamo di voler compilare, all'interno di un programma C#, le due istruzioni seguenti:

```
Veicolo v=new Veicolo("abc");
v.Accelera();
```

Il compilatore, per creare un assembly a partire dal codice sorgente, deve essere a conoscenza del fatto che da qualche parte è definita una classe Veicolo, la quale è dotata di un costruttore con un parametro di tipo string e di un metodo Accelera.

L'analisi dei tipi e delle loro caratteristiche è possibile mediante un meccanismo noto come *reflection*, che fornisce l'accesso a ogni loro dettaglio.

La reflection è estremamente potente, perché permette di cercare e di analizzare tipi e membri a runtime, senza necessità di conoscerli a tempo di compilazione.

Vedremo come tramite apposite API sia possibile esaminare i tipi contenuti in un assembly e istanziarli.

NOTA	Esistono anche librerie di terze parti che permettono di analizzare il contenuto di un assembly o di generarne uno dinamicamente. Fra questi, la libreria open source Mono Cecil, che in molti casi specifici è addirittura da preferire alle classi standard: per esempio, non ha i classici problemi di utilizzo di memoria che incontrereste cercando di caricare un assembly e di scaricarlo dalla memoria alla fine dell'utilizzo.

A questo punto è possibile anche ricavare i membri esposti da un dato tipo, con ogni informazione sui loro parametri compresi i valori assunti da essi durante l'esecuzione del programma, e invocarli mediante appositi metodi. Al contrario, si possono anche generare tipi a tempo di esecuzione, emettendone il codice necessario. Naturalmente, tale potenzialità ha anche i suoi lati negativi, che sono essenzialmente due:

- il fatto che i tipi sono analizzabili e istanziabili a runtime, spesso mediante dei nomi specificati con una stringa, va contro la sicurezza dei tipi (non è detto, per esempio, che un nome di tipo specificato a tempo di esecuzione sia effettivamente esistente e istanziabile);

- il secondo svantaggio è che le performance di reflection sono peggiori rispetto a quelle ottenute con la tipizzazione forte, poiché, per quanto detto sopra, la ricerca di un tipo e la creazione di relative istanze si basano spesso sulla ricerca di nomi non conosciuti a tempo di compilazione, che avviene all'interno di uno o più assembly, il tutto eseguito a runtime.

Object Browser - Microsoft Visual Studio

FILE EDIT VIEW DEBUG TEAM TOOLS TEST ANALYZE DEVEXPRESS WINDOW HELP

Attach...

Toolbox

Object Browser | Start Page

Browse: .NET Framework 4.5.1

<Search>

- Single
- StackOverflowException
- STAThreadAttribute
- String
  - Base Types
    - ICloneable
    - IComparable
    - IComparable<string>
    - IConvertible
    - IEnumerable
    - IEnumerable<char>
    - IEquatable<string>
    - Object
  - Derived Types
- StringComparer
- StringComparison
- StringSplitOptions
- SystemException
- ThreadStaticAttribute
- TimeoutException
- TimeSpan
- TimeZone
- TimeZoneInfo
- TimeZoneInfo.AdjustmentRule
- TimeZoneInfo.TransitionTime
- TimeZoneNotFoundException

- LastIndexOfAny(char[ ])
- LastIndexOfAny(char[ ], int)
- LastIndexOfAny(char[ ], int, int)
- Normalize()
- Normalize(System.Text.NormalizationForm)
- PadLeft(int)
- PadLeft(int, char)
- PadRight(int)
- PadRight(int, char)
- Remove(int)
- Remove(int, int)
- Replace(char, char)
- Replace(string, string)
- Split(char[ ], int)
- Split(char[ ], int, System.StringSplitOptions)
- Split(char[ ], System.StringSplitOptions)
- Split(params char[ ])
- Split(string[ ], int, System.StringSplitOptions)
- Split(string[ ], System.StringSplitOptions)

public bool **StartsWith**(string *value*)

Member of System.String

Summary:

Determines whether the beginning of this string instance matches the specified string.

Figura 14.1 – L'object browser di Visual Studio esamina i tipi e relativi membri.

Nonostante ciò, le funzioni di reflection, se ci pensate bene, sono utili e adoperate in moltissimi ambiti.

Osservate Visual Studio durante il funzionamento del suo designer e, in particolare, la finestra delle proprietà che mostra i membri di un controllo selezionato: chiedetevi come fa l'IDE a conoscere il nome di ogni proprietà ed evento, e i rispettivi valori assunti. Se poi avete avuto modo di usarne anche l'*object browser* (vedi Figura 14.1), che consente di esaminare gli assembly referenziati all'interno della soluzione e di ricavarne tutti i tipi e i relativi membri, ponetevi la stessa domanda.

La risposta, a questo punto, è facile da dare. Basta "rifletterci" un po' sopra: reflection.

NOTA	L'object browser di Visual Studio è attivabile dal menu View, oppure mediante la scorciatoia da tastiera CTRL+W, J. Esso permette di sfogliare e analizzare il contenuto degli assembly, quindi tipi e relativi membri, sia facenti parte del .NET Framework sia quelli di terze parti o creati nell'ambiente stesso.

Metadati e tipi

Nel Capitolo 1, parlando degli assembly .NET, si è detto che essi sono le unità logiche contenenti il codice eseguibile dal CLR. Inoltre, abbiamo sottolineato come ogni assembly sia dotato dei cosiddetti metadati, che descrivono tutti i tipi contenuti a loro volta nello stesso assembly.

Il tipo fondamentale da cui partire è la classe che rappresenta tutti i tipi supportati da .NET, e cioè la già citata Type. Essa rappresenta qualunque tipo di .NET: classi, interfacce, struct, array, tipi valore e riferimento, enumerazioni, tipi generici, tipi generici costruiti aperti e chiusi. Per mezzo di un'istanza di Type è possibile ottenere praticamente ogni informazione sulla struttura e le caratteristiche del tipo: il nome, l'assembly in cui esso è definito, le classi eventuali da cui deriva, i modificatori di accesso, oltre a ogni suo membro e relative informazioni. Insomma, la classe Type è utilizzata talmente spesso ed è così fondamentale che non può che far parte del namespace System.

Altri tipi e metodi per leggere i metadati e per creare dinamicamente oggetti e invocarne i membri sono invece contenuti nel namespace System.Reflection.

La Tabella 14.1 elenca i tipi principali contenuti nel namespace suddetto, che verranno utilizzati più frequentemente.

Tabella 14.1 **- Tipi principali del namespace System.Reflection.**

Tipo	Descrizione
Assembly	rappresenta un assembly .NET e contiene metodi e proprietà per caricare e analizzare un eseguibile o una libreria di classi
AssemblyName	descrive l'identità di un assembly, quindi espone proprietà per ricavarne nomi e versione
ConstructorInfo	rappresenta il costruttore di un tipo e fornisce l'accesso alle sue informazioni
EventInfo	rappresenta un evento e fornisce l'accesso alle sue informazioni
FieldInfo	rappresenta un campo e fornisce l'accesso alle sue informazioni
MemberInfo	rappresenta un generico membro di un tipo e consente quindi di ottenere informazioni sulle caratteristiche comuni di EventInfo, FieldInfo, MethodInfo e PropertyInfo
MethodBody	rappresenta il corpo di un metodo e consente di ricavarne il codice intermedio e altri metadati
MethodInfo	rappresenta un dato metodo e fornisce l'accesso alle sue informazioni
Module	rappresenta un modulo contenuto in un assembly
ParameterInfo	rappresenta un parametro e fornisce l'accesso alle sue informazioni
PropertyInfo	rappresenta una proprietà e fornisce l'accesso alle sue informazioni
TypeInfo	rappresenta un qualunque tipo .NET e consente di ricavare informazioni dai suoi metadati

Con tali tipi è praticamente possibile ricavare ogni informazione dai metadati, fino a riuscire a implementare un decompilatore di codice .NET.

Analisi di un tipo

Per ottenere un'istanza della classe Type, vi sono diverse possibilità, il cui utilizzo dipende dalle circostanze in cui è necessario ricavarla.

La prima è l'uso dell'operatore typeof, che consente di ottenere un oggetto a tempo di compilazione a partire da un tipo esistente:

```
Type t=typeof(String);
```

Una seconda possibilità per ricavare un'istanza Type è quella di invocare il metodo GetType su un qualunque oggetto, dato che ogni tipo deriva dalla superclasse System.Object.

```
string str = "abc";
Type t2 = str.GetType();
```

Una volta ottenuto l'oggetto Type è possibile ottenere informazioni di vario genere utilizzandone le proprietà della classe.

Con .NET 4.5 sono stati apportati dei cambiamenti ai vari meccanismi di reflection e, quindi, alle relative API. In pratica, alcune funzioni della classe Type sono state spostate nella nuova classe TypeInfo, già presente nella Tabella 14.1, di cui è possibile ottenere un'istanza

mediante un apposito metodo di estensione della classe Type, denominato GetTypeInfo, anch'esso novità di .NET 4.5 e definito nel namespace System.Reflection.

L'oggetto TypeInfo, in particolare, rappresenta la definizione completa di un tipo, consentendo un'analisi più profonda e fornendo anche nuove proprietà e nuovi metodi. Inoltre, a differenza della classe Type e del metodo GetType, TypeInfo è utilizzabile anche in applicazioni Windows Store (cioè le applicazioni per Windows 8.x scaricabili dallo store di Microsoft), in cui invece è impedito l'uso della prima. In pratica, TypeInfo e i suoi metodi sono ora la modalità consigliata per l'utilizzo dei meccanismi di reflection, anche se in versioni precedenti alla 4.5 è possibile tranquillamente continuare a utilizzare la classe Type. Nel seguito, dove utilizzeremo la classe Type e i suoi membri, sarà sempre possibile sostituirla con i membri corrispondenti posseduti dalla classe TypeInfo, e viceversa.

Un tipo può essere ottenuto anche da una stringa che ne rappresenta il nome. In tal caso, è possibile utilizzare il metodo statico GetType, passando come parametro tale stringa, che deve essere in generale nel formato completamente qualificato, cioè comprensivo di nome dell'assembly che lo contiene (vedere il prossimo paragrafo). Se il tipo si trova però nell'assembly attualmente in esecuzione oppure nell'assembly standard Mscorlib.dll, è sufficiente indicare il nome del tipo completo di namespace. Per esempio:

```
Type ts=Type.GetType("System.String");
```

In questo caso, il tipo String è ottenuto semplicemente indicando il suo nome completo. Se invece si volesse fare lo stesso con un tipo contenuto in un assembly secondario, per esempio XmlDocument, sarebbe necessario indicare il nome in questo formato:

```
Type tx = Type.GetType("System.Xml.XmlDocument, System.Xml, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089");
```

Nel caso in cui il runtime non riesca a risolvere correttamente il nome del tipo, verrebbe restituito un riferimento null.

Il nome completo di un assembly può essere ricavato per mezzo della proprietà FullName. Per esempio, per ricavare quello dell'assembly corrente, si può scrivere:

```
string fullname = Assembly.GetExecutingAssembly().FullName;
```

Se invece si ha già a disposizione un oggetto Assembly, è possibile ricavarne un tipo in esso contenuto per mezzo dell'omonimo metodo GetType:

```
Assembly ass=Assembly.GetExecutingAssembly();
Type t=ass.GetType("MioNamespace.MiaClasse");
```

La classe Type espone infine dei metodi per ricavarne un'istanza di un tipo array o generico.

Per ottenere il tipo di un array bidimensionale di interi, basta invocare il metodo MakeArrayType su un'istanza del tipo System.Int32, con parametro pari a 2:

```
Type tabInteri= typeof(int).MakeArrayType(2); //restituisce un System.Int32[,]
```

Allo stesso modo, il metodo MakeGenericType permette di costruire un tipo generico, passando come argomento un array dei tipi da utilizzare come parametri.

Nel seguente caso, il tipo generico aperto List<T> prevede un unico parametro, che verrà sostituito dal tipo int:

```
Type listInteri = typeof(List<>).MakeGenericType(typeof(int));
```

Nomi dei tipi

Il seguente esempio mostra come ricavare e stampare informazioni sul nome di un tipo e del suo assembly:

```
Console.WriteLine(t.Name);
Console.WriteLine(t.Namespace);
Console.WriteLine(t.FullName);
Console.WriteLine(t.AssemblyQualifiedName);
```

Le proprietà Name, Namespace e FullName permettono di leggere nome, namespace e il nome completo (costituito dall'unione dei primi due) di ogni tipo.

La proprietà AssemblyQualifiedName restituisce invece il nome ancora più completo, in quanto include quello dell'assembly da cui il tipo è stato caricato.

Il codice precedente può essere modificato utilizzando la classe TypeInfo, per essere così eseguito su .NET 4.5 e quindi anche su applicazioni Windows 8.x/10:

```
TypeInfo ti= typeof(string).GetTypeInfo();
Console.WriteLine(ti.Name);
Console.WriteLine(ti.Namespace);
Console.WriteLine(ti.FullName);
Console.WriteLine(ti.AssemblyQualifiedName);
Console.WriteLine(ti.BaseType.FullName);
```

Come detto prima, è possibile ricavare informazioni su qualunque tipo supportato in .NET. Quindi, sia l'operatore typeof sia il metodo GetType possono ricavare un'istanza Type anche per tipi più complessi, come array o tipi generici:

```
Type listGeneric= typeof(List<>);
Type listGenericInt = typeof(List<int>);
Type arrayType = typeof(int[]);
```

Se provate a ricavare e stampare il nome di un tipo generico, noterete l'utilizzo di una particolare sintassi, che dipende dal numero di parametri di tipo utilizzati. Per esempio, la

proprietà Name del tipo List<int>, che utilizza un unico parametro di tipo, restituisce il nome della classe List e il numero 1, separati da un apice:

```
List'1
```

Lo stesso nome, però, sarebbe restituito anche dal tipo List<string> o dal tipo aperto List<>, dato che entrambi hanno un solo parametro di tipo. Se si utilizza invece la proprietà FullName, essa restituirà il nome completo del tipo generico seguito dall'AssemblyQualifiedName dei parametri di tipo. Per esempio:

```
Dictionary<int, string> dict = new Dictionary<int, string>();
TypeInfo tiDict = dict.GetType().GetTypeInfo();
Console.WriteLine(tiDict.FullName);
```

Il codice precedente stamperebbe la seguente stringa:

```
System.Collections.Generic.Dictionary'2[[System.Int32, mscorlib, Version=4.0.0.0
, Culture=neutral, PublicKeyToken=b77a5c561934e089],[System.String, mscorlib, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089]]
```

In essa il numero 2, che segue il nome completo della classe Dictionary, indica il numero di parametri di tipo. All'interno delle parantesi quadre, invece, vengono mostrati i nomi completi dei due parametri int e string, che fanno parte dell'assembly mscorlib, quello principale della libreria di classi di base di .NET.

Anche gli array necessitano di una sintassi particolare, che dipende appunto dalle dimensioni del vettore. In questi esempi vengono stampate le proprietà FullName di un array di interi e di uno bidimensionale di stringhe:

```
Type arrayType = typeof(int[]);
Console.WriteLine(arrayType.FullName); //System.Int32[]

string[,] matrice = new string[3, 3];
Console.WriteLine(matrice.GetType().FullName); //System.String[,]
```

Fra i casi particolari vi è quello dei tipi *innestati*. In tal caso, è possibile ricavarne un elenco sotto forma di array di oggetti Type utilizzando il metodo GetNestedTypes, oppure uno ben determinato, indicando il nome come parametro del metodo GetNestedType. Supponiamo, per esempio, di aver implementato la seguente classe Container con relativa classe innestata:

```
namespace Test
{
class Container
{
public class Nested
{
Nested() { }
}
}
}
```

Per ricavare tutti i tipi pubblici definiti all'interno di Container, basta scrivere:

```
Type[] nestedTypes= typeof(Container).GetNestedTypes();
foreach (Type nt in nestedTypes)
{
Console.WriteLine(nt.FullName);
}
```

Il nome del tipo in questo caso utilizzerà il carattere + come separatore fra il nome della classe esterna e il nome di quella interna, per esempio:

```
Test.Container+Nested
```

Caratteristiche e struttura dei tipi

La classe Type contiene numerosi membri utilizzabili per analizzare ogni caratteristica dei suoi metadati, la maggior parte dei quali restituisce oggetti elencati nella Tabella 14.1. Per esempio, oltre ai nomi visti nel paragrafo precedente, può essere di particolare interesse analizzare alcune caratteristiche per comprenderne la struttura e ottenere informazioni sulla particolare varietà di tipo. A tale scopo esistono delle proprietà booleane, il cui nome inizia per Is, il cui significato è facilmente intuibile:

- IsAbstract – è una classe astratta;
- IsArray – è un tipo array;
- IsClass – è una classe, cioè non è un'interfaccia o un tipo valore;
- IsEnum – è un'enumerazione;
- IsInterface – è un'interfaccia;
- IsNested – è un tipo definito all'interno di un altro;
- IsPointer – è un puntatore a un tipo;
- IsPrimitive – è un tipo primitivo;
- IsSealed – è una classe non derivabile;
- IsValueType – è un tipo valore.

Le precedenti proprietà trattano quindi della struttura di un tipo. Altre permettono inoltre di ricavare informazioni sulla loro visibilità; per esempio, le proprietà IsPublic e IsNotPublic indicano rispettivamente se il tipo in esame sia stato dichiarato come pubblico o meno.

Ecco cosa restituiscono le proprietà sopraelencate per il tipo Int32:

```
Type tipoInt=typeof(int);
Console.WriteLine(tipoInt.IsAbstract); //false
Console.WriteLine(tipoInt.IsArray); //false
Console.WriteLine(tipoInt.IsClass); //false
Console.WriteLine(tipoInt.IsEnum); //false
Console.WriteLine(tipoInt.IsInterface); //false
Console.WriteLine(tipoInt.IsNested); //false
Console.WriteLine(tipoInt.IsNotPublic); //false
```

```
Console.WriteLine(tipoInt.IsPointer); //false
Console.WriteLine(tipoInt.IsPrimitive); //true
Console.WriteLine(tipoInt.IsPublic); //true
Console.WriteLine(tipoInt.IsSealed); //true
Console.WriteLine(tipoInt.IsValueType); //true
```

Una classe può essere derivata da una classe madre, che se non altrimenti specificato sarà System.Object, e può implementare eventuali interfacce. Per ottenere la classe madre di un tipo si utilizza la proprietà BaseType, che naturalmente restituisce a sua volta un'istanza di Type:

```
Type exType=typeof(ArgumentException);
Type bt=exType.BaseType;
Console.WriteLine(bt.Name); // SystemException
```

Nel caso precedente la classe ArgumentException è figlia di SystemException.

NOTA	La classe System.Object, in quanto superclasse di ogni altro tipo, non ha classi da cui deriva, quindi nel suo caso la proprietà BaseType restituirà null.

Se si vuole verificare che una classe sia derivata da un'altra, è poi possibile utilizzare il metodo IsSubclassOf:

```
bool subclass=bt.IsSubclassOf(exType);
```

Inoltre, è possibile verificare che un tipo sia assegnabile in maniera sicura a un altro, cioè che esista una conversione. In questo caso, basta utilizzare il metodo IsAssignableFrom:

```
bool isAssignable=t1.IsAssignableFrom(t2); //il tipo t1 è assegnabile dal tipo t2
```

Per ricavare le interfacce implementate da una classe, si può utilizzare il metodo GetInterfaces.

```
Type[] interfaces= exType.GetInterfaces();
interfaces.ToList().ForEach(i => Console.WriteLine(i.Name));
```

Per analizzare i tipi generici sono disponibili delle proprietà dedicate. In particolare, la proprietà IsGenericType restituisce true per un tipo generico, sia esso la definizione di un tipo, un tipo generico costruito aperto o chiuso. Invece, IsGenericTypeDefinition è true solo se il tipo in esame rappresenta la definizione di un tipo generico, da cui un tipo può essere costruito. Un tipo generico, inoltre, può avere parametri di tipo non assegnati e in tal caso sarà true la proprietà ContainsGenericParameters. Ecco un paio di esempi che mostrano la differenza:

```
Type tgen = typeof(Dictionary<int, string>);
Console.WriteLine(tgen.IsGenericType); //true
Console.WriteLine(tgen.IsGenericTypeDefinition); //false, è un tipo costruito
Console.WriteLine(tgen.ContainsGenericParameters); //false, parametri assegnati

tgen = typeof(Dictionary<, >);
Console.WriteLine(tgen.IsGenericType); //true
Console.WriteLine(tgen.IsGenericTypeDefinition); //true
```

```
Console.WriteLine(tgen.ContainsGenericParameters); //true, parametri non assegnati
```

Membri di tipo

Di ogni oggetto Type è possibile ricavarne i membri per mezzo di metodi restituenti oggetti che ne rappresentano costruttori, metodi, eventi, campi e così via. Tali metodi hanno una struttura comune, e diversi overload, per poterne controllare e filtrare i risultati. Per esempio, per ottenere tutti i metodi pubblici della classe String, utilizziamo il metodo GetMethods e da ogni MethodInfo ottenuto possiamo leggere il tipo di ritorno e il nome:

```
Type ts=typeof(string);
MethodInfo[] methods = ts.GetMethods();
foreach (MethodInfo mi in methods)
{
Console.WriteLine("{0} {1}", mi.ReturnType.Name, mi.Name);
}
```

Com'è naturale, ogni metodo ha eventualmente dei parametri. Questi sono rappresentati dalla classe ParameterInfo e si possono ottenere tramite il metodo GetParameters di MethodInfo.

Riprendendo l'esempio precedente, ecco come ricavare e visualizzare anche i parametri, stampando l'intera firma di ogni metodo del tipo in esame:

```
List<string> pars=new List<string>();
foreach (MethodInfo mi in methods)
{
pars=new List<string>();
foreach (ParameterInfo par in mi.GetParameters())
{
pars.Add(String.Format("{0} {1},", par.ParameterType.Name, par.Name));
}
Console.WriteLine("{0} {1}({2})", mi.ReturnType.Name, mi.Name, string.Join(",", pars.ToArray()));
}
```

Il tipo di ogni parametro viene ricavato dalla proprietà ParameterType di Parameter-Info, mentre il nome dalla proprietà Name. In questo come in molti altri scenari, LINQ può tornare utile per eseguire query sui risultati ottenuti dai vari metodi di reflection. Per esempio, per ricavare i metodi di una classe che hanno almeno un parametro, possiamo scrivere la seguente espressione:

```
var query = from m in typeof(String).GetMethods()
where m.GetParameters().Any()
select String.Format("{0} {1}", m.ReturnType, m.Name);
```

Negli esempi precedenti abbiamo ricavato i metodi pubblici di un tipo, ma per ottenere anche quelli rimanenti è necessario servirsi di un apposito overload che utilizza un parametro di tipo BindingFlags, che è un'enumerazione decorata con l'attributo Flags, e quindi i suoi valori possono essere combinati bit a bit.

Per ottenere dei risultati, è innanzitutto necessario specificare almeno uno fra i due valori BindingFlags.Instance o BindingFlags.Static, che indicano rispettivamente membri di istanza e membri statici. Per esempio, per ottenere i metodi non pubblici della classe string, possiamo scrivere:

```
Console.WriteLine("non public methods");
var query = from m in typeof(string).GetMethods(BindingFlags.Instance | BindingFlags.NonPublic)
select String.Format("{0} {1}", m.ReturnType, m.Name);
```

Il metodo GetMembers restituisce tutti i membri di un tipo sotto forma di istanza di MemberInfo. Per esempio, per ottenere i membri pubblici e statici, possiamo scrivere:

```
MemberInfo[] members = typeof(int).GetMembers(BindingFlags.Static|BindingFlags.Public);
foreach (MemberInfo member in members)
{
Console.WriteLine("{0} {1}", member.MemberType, member.Name);
}
```

La proprietà MemberType di MemberInfo consente quindi di determinare il tipo esatto di membro, e a questo punto sarà possibile anche eseguire una conversione esplicita verso tale tipo.

La classe MemberInfo possiede due proprietà, DeclaringType e ReflectedType, che rispettivamente restituiscono il tipo in cui è dichiarato il membro e il tipo sul quale invece è stato chiamato il metodo GetMembers:

```
Type declaring=member.DeclaringType;
Type reflected= member.ReflectedType;
```

La differenza fra le due proprietà si nota nel caso di tipi derivati da altri e di cui ereditano quindi anche i membri. In questo caso, un membro può essere dichiarato nella classe madre, ottenibile per mezzo della proprietà DeclaringType, mentre il tipo della classe figlia è restituito da ReflectedType. In modo analogo, per mezzo dei metodi GetFields, GetEvents, GetProperties, magari filtrandoli con parametri BindingFlags, si possono ricavare direttamente le informazioni FieldInfo per i campi, EventInfo per gli eventi e PropertyInfo per le proprietà: tutte classi derivate da quella astratta comune MemberInfo.

Da quest'ultima deriva anche la classe astratta MethodBase, che a sua volta è classe madre di MethodInfo e ConstructorInfo.

La Figura 14.2 mostra la gerarchia della classi che rappresentano i vari membri di un tipo.

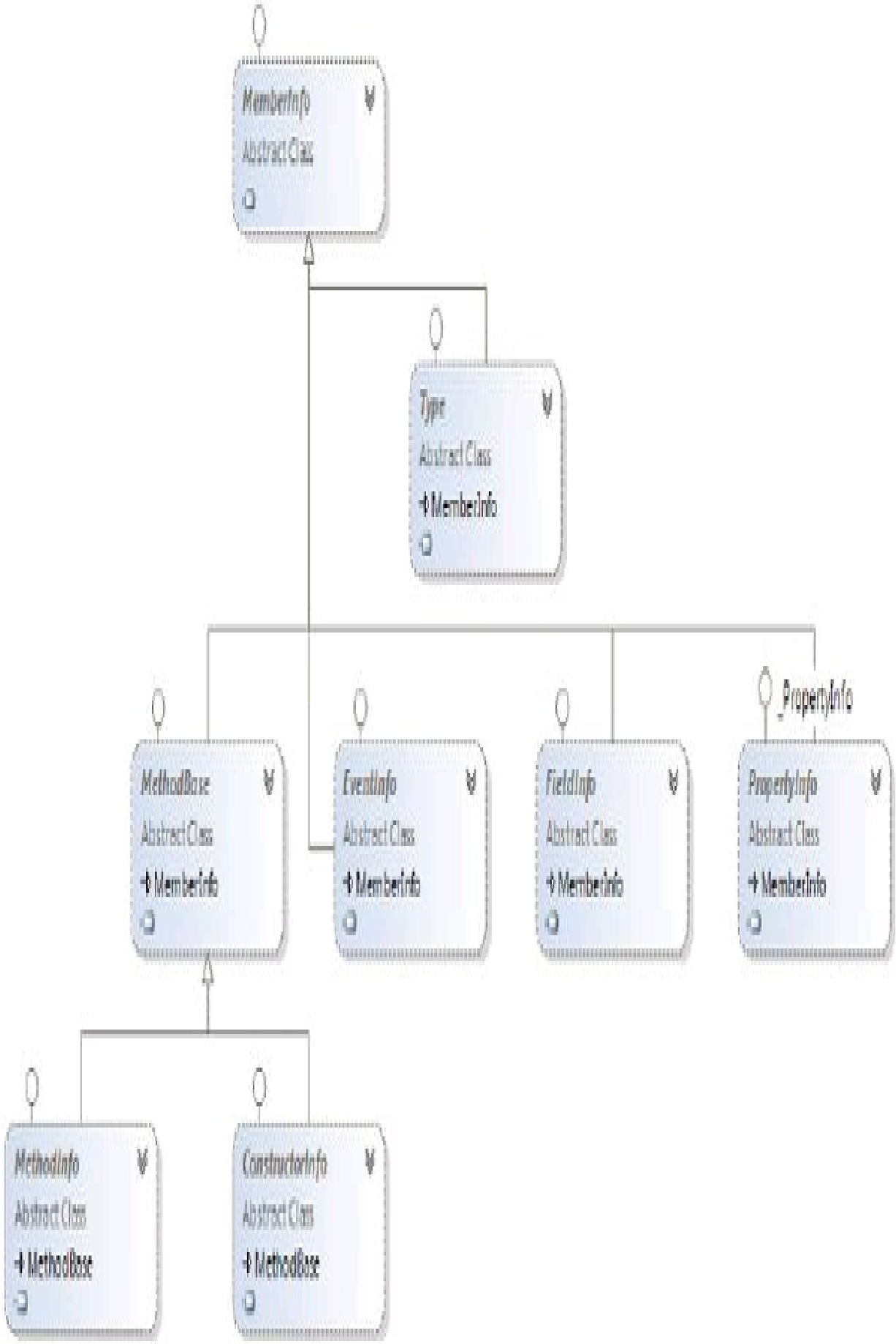

MemberInfo
Abstract Class
Type
Abstract Class
MemberInfo
PropertyInfo
MethodBase
Abstract Class
MemberInfo
EventInfo
Abstract Class
MemberInfo
FieldInfo
Abstract Class
MemberInfo
PropertyInfo
Abstract Class
MemberInfo
MethodInfo
Abstract Class
MethodBase
ConstructorInfo
Abstract Class
MethodBase

Figura 14.2 – Gerarchia delle classi che rappresentano i membri dei tipi.

Ognuna delle classi derivate da MemberInfo possiede a sua volta metodi e proprietà peculiari del tipo di membro. Come già abbiamo avuto modo di vedere e provare, MethodInfo possiede per esempio la proprietà ReturnType per ricavare il tipo restituito dal metodo.

Ognuno dei membri può essere ricavato anche singolarmente senza necessità di passare prima dall'intera collezione. Basta specificarne il nome ed eventuali altri parametri per indicarne i modificatori di accesso o altri aspetti peculiari del tipo di membro. Per esempio, per ricavare un metodo pubblico denominato MyMethod, esposto dalla classe Container, sarà sufficiente scrivere:

```
MethodInfo method=typeof(Container).GetMethod("MyMethod");
```

Se la classe Container possedesse più overload pubblici dello stesso MyMethod, si verificherebbe un'eccezione di tipo System.Reflection.AmbiguousMatchException. Infatti non è possibile, mediante il solo nome, determinare la versione corretta del metodo che si vuole ottenere: sarà necessario aggiungere ulteriori parametri al metodo GetMethod per restringere la ricerca e ottenere un singolo MethodInfo.

Se la classe Container possedesse un overload con parametro un valore di tipo int, si potrebbe ricavarlo nel seguente modo:

```
MethodInfo method = typeof(Container).GetMethod("MyMethod", new Type[]{typeof(int)});
```

La classe TypeInfo possiede delle proprietà aggiuntive che restituiscono delle collezioni di oggetti, esattamente degli IEnumerable<>, che rendono più semplice l'accesso ai membri dichiarati all'interno del tipo stesso. Per esempio, per ottenere i costruttori dichiarati all'interno di un tipo, si può utilizzare la proprietà DeclaredConstructors:

```
IEnumerable<ConstructorInfo> constructors= typeof(Container).GetTypeInfo().DeclaredConstructors;
constructors.ToList().ForEach(Console.WriteLine);
```

Allo stesso modo esistono i metodi DeclaredEvents, DeclaredFields, DeclaredMembers, DeclaredMethods, DeclaredNestedTypes e DeclaredProperties.

Invocazione dinamica

Una volta ottenuti i membri di un tipo, per esempio dei metodi o delle proprietà, li si può utilizzare dinamicamente, invocandoli a runtime. Questa possibilità è anche chiamata *late binding*, cioè *binding ritardato*, in quanto l'invocazione avverrà solo a tempo di esecuzione. La classe String, per esempio, possiede la proprietà Length, che rappresenta la lunghezza di una stringa; per ricavarne la corrispondente istanza di PropertyInfo, basta indicare il nome come parametro del metodo GetPropertyInfo:

```
Type ts= typeof(string);
PropertyInfo pi=ts.GetProperty("Length");
```

A questo punto, è possibile ottenerne il valore indicando l'istanza di cui si desidera leggere la proprietà:

```
int len=(int)pi.GetValue("abc");
```

Se la proprietà fosse anche accessibile in scrittura, sarebbe possibile invocarne il metodo SetValue per assegnarle un nuovo valore; lo stesso varrebbe per oggetti FieldInfo.

Un oggetto MethodInfo, invece, possiede il metodo Invoke, al quale passare eventuali parametri previsti dal metodo invocato dinamicamente:

```
Type[] parTypes=new Type[]{typeof(string)};
MethodInfo mIndexOf=ts.GetMethod("IndexOf", parTypes);
int index=(int)mIndexOf.Invoke("abc", new object[]{"b"});
```

Il metodo IndexOf ha diversi overload; in questo caso viene ricavato quello che accetta un singolo parametro di tipo string.

Invece, nel caso di metodi che accettano parametri di tipo ref o out, è necessario utilizzare il metodo MakeByRefType per creare un parametro correttamente. Per esempio, consideriamo il metodo TryParse di Int32, che ha la seguente firma:

```
public static bool TryParse(
string s,
out int result
)
```

Per essere eseguito dinamicamente, deve essere ricavato indicando i due parametri di tipo come segue:

```
Type[] intParseTypes = new Type[] { typeof(string), typeof(int).MakeByRefType() };
MethodInfo tryParse = typeof(int).GetMethod("TryParse", intParseTypes);
```

E a questo punto, per ottenere il risultato all'interno della variabile di uscita, si utilizza un array di object, in cui il primo elemento è la stringa da trasformare e il secondo, inizialmente null, contiene il valore finale:

```
object[] arguments={"123", null};
bool parseOk = (bool)tryParse.Invoke(null, arguments);
if (parseOk)
Console.WriteLine(arguments[1]);
```

Analisi di un assembly

La classe Assembly rappresenta un assembly .NET, cioè l'unità fondamentale di codice compilato, sotto forma di eseguibile o di libreria. Essa può essere utilizzata per caricare un assembly in memoria e analizzarne o utilizzarne i tipi in esso contenuti. Tramite i metodi e le

proprietà viste nei paragrafi precedenti, di essi potranno poi essere ricavate e analizzate a loro volta le varie caratteristiche.

Il primo passo è caricare un assembly in memoria o utilizzarne uno già disponibile. La classe Assembly possiede diversi metodi statici per farlo. Per esempio, per ottenere l'assembly attualmente in esecuzione, si utilizza il metodo GetExecutingAssembly:

```
Assembly assm=Assembly.GetExecutingAssembly();
```

Per ottenere invece l'assembly contenente un determinato tipo, è possibile utilizzare il metodo GetAssembly, come nel seguente caso:

```
Assembly assm=Assembly.GetAssembly(typeof(string));
```

Un'altra opzione è utilizzare direttamente la proprietà Assembly della classe Type o di TypeInfo.

Se si intende caricare un assembly da un file, è possibile invece utilizzare il metodo LoadFrom, indicando per esempio il percorso:

```
Assembly assm=Assembly.LoadFrom("nomeassembly.dll");
```

Il percorso del file può essere relativo alla directory di esecuzione attuale, oppure assoluto. Il caricamento dinamico da un file è particolarmente utile se si vuole implementare un sistema a plugin nelle nostre applicazioni, caricando assembly e poi ricavandone e istanziandone i tipi a runtime.

Si noti che, caricando l'assembly con il metodo LoadFrom e ricavandone i tipi, si possono avere dei possibili effetti collaterali, come l'esecuzione di costruttori statici nel momento in cui si utilizza un tipo che ne implementi qualcuno.

Per evitare questi problemi, si può utilizzare il metodo ReflectionOnlyLoadFrom, che indica esplicitamente di caricare solo i metadati dei tipi, senza istanziarli, al solo scopo di ottenerne informazioni sulla loro struttura:

```
Assembly assm=Assembly.ReflectionOnlyLoadFrom();
```

Una volta ottenuta un'istanza di Assembly, per ricavare i tipi definiti all'interno di un assembly esistono diverse possibilità. Quella più utilizzata prevede l'uso del metodo GetTypes, che restituisce tutti i tipi:

```
Type[] types=assm.GetTypes();
```

A partire da .NET 4.5 è disponibile anche la proprietà DefinedTypes, che invece restituisce una collezione dei TypeInfo corrispondenti ai tipi definiti nell'assembly:

```
var types=Assembly.GetExecutingAssembly().DefinedTypes;
```

Poiché un assembly può contenere anche tipi non pubblici, la proprietà Exported-Types consente invece di ricavare solo un'enumerazione di quelli visibili e utilizzabili all'esterno:

```
var exported=assm.ExportedTypes;
```

Anche in questi casi d'uso, è di notevole aiuto la sintassi delle query LINQ per ricavare solo particolari tipi, soprattutto per assembly con migliaia di classi.

Supponiamo di voler ricavare tutti i tipi enumerativi facenti parte dell'assembly standard di .NET. L'assembly mscorlib.dll è ottenibile facilmente, sapendo che esso è lo stesso assembly che contiene i tipi predefiniti di .NET, come int:

```
Assembly mscorlib = typeof(int).Assembly;

var enums = from type in mscorlib.DefinedTypes
where type.IsEnum
select type.FullName;

enums.ToList().ForEach(Console.WriteLine);
```

Nome di un assembly

Un *assembly* può essere identificato univocamente mediante l'insieme di quattro diverse componenti, tutte contenute fra i suoi metadati:

- il suo nome, che corrisponde a quello del file senza estensione (per esempio, il nome dell'assembly mscorlib.dll è semplicemente mscorlib);
- la versione dell'assembly, che è specificata nel formato major.minor.build.revision, dove ognuna di queste singole parti è un numero intero, e che viene assegnata mediante un attributo globale, come vedremo fra qualche paragrafo. Se non altrimenti indicato, essa assume il valore predefinito 0.0.0.0;
- le impostazioni di lingua;
- un token di chiave pubblica, che nel caso di assembly firmati è una piccola parte dell'intera chiave pubblica.

In tal modo, il nome completamente qualificato di un assembly, restituito per esempio dalla proprietà FullName di Assembly, è una stringa che contiene tutte e quattro queste parti secondo il seguente formato:

```
NomeSemplice, Version=versione, Culture=neutral, PublicKeyToken=chiave
```

Per esempio, nel caso dell'assembly principale del .NET Framework, si potrebbe avere il seguente nome completo:

```
mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
```

La classe AssemblyName consente di descrivere l'identità completa di un assembly ed è ricavabile dal metodo GetName della classe Assembly:

```
Assembly assm = Assembly.GetExecutingAssembly();
AssemblyName name = assm.GetName();
Console.WriteLine("name {0} version {1}",name.Name, name.Version);
```

Istanziare un tipo

Per creare un oggetto dinamicamente, a partire da un'istanza di Type, ci sono due modalità.

La prima, più istintiva, è quella di ricavare e invocare un oggetto ConstructorInfo, come già visto nei paragrafi precedenti, mediante il metodo GetConstructor.

L'altra, più semplice e immediata, è utilizzare la classe Activator. Essa possiede dei metodi CreateInstance ai quali è possibile passare i parametri opzionali da inviare al costruttore della classe da istanziare.

Supponiamo di avere la seguente classe:

```
class Test
{
public Test(string name)
{
Name = name;
}

public Test()
{
Name = "untitled";
}
}
```

Per creare un'istanza utilizzando il costruttore con il solo parametro string, si utilizzerà per esempio il metodo CreateInstance nel seguente modo:

```
Test t1=(Test)Activator.CreateInstance(typeof(Test), "nome"));
```

Un'altra versione generica, invece, permette di invocare automaticamente il costruttore predefinito senza parametri:

```
Test t2=(Activator.CreateInstance<Test>();
```

Generazione dinamica di codice

Il namespace System.Reflection.Emit contiene le classi e i metodi che permettono la generazione di codice IL e di metadati a runtime, creando degli assembly dinamici direttamente in memoria, che conterranno al loro interno dei tipi e i relativi membri.

Naturalmente, un assembly dinamico di tale tipo può, a questo punto, essere salvato anche su un normale file, divenendo un assembly classico, con niente di diverso da quelli che abbiamo visto e utilizzato finora.

La generazione dinamica di codice non è certamente una pratica comune e richiede la conoscenza abbastanza approfondita delle istruzioni del linguaggio intermedio IL, anche se le classi del namespace Emit nascondono buona parte della complessità.

Per tali motivi mostreremo solo un semplice esempio di generazione di un assembly, rinviando i lettori che volessero approfondire a testi più specifici o a pagine internet dedicate.

Tipi di System.Reflection.Emit

All'interno del namespace System.Reflection.Emit sono contenuti dei tipi che permettono di costruire un assembly e il suo contenuto, fino ad arrivare alla generazione del codice IL di ogni singolo metodo.

La Tabella 14.2 elenca i tipi principali che utilizzeremo nei prossimi esempi, o che possono tornare utili in generale in tali procedure.

Tabella 14.2 - **Tipi principali del namespace System.Reflection.Emit.**

Tipo	Descrizione
AssemblyBuilder	definisce e rappresenta un assembly dinamico, sia eseguibile sia sotto forma di libreria
ModuleBuilder	definisce e rappresenta un modulo in un assembly dinamico
EnumBuilder	descrive e rappresenta un'enumerazione
TypeBuilder	consente di descrivere e creare a runtime i vari tipi .NET
FieldBuilder	definisce e rappresenta un campo
LocalBuilder	rappresenta un campo e fornisce accesso alle sue informazioni
MethodBuilder	consente di creare un metodo di un tipo dinamico
CosntructorBuilder	consente di creare il costruttore di una classe dinamica
ParameterBuilder	crea i parametri e li associa ai metodi

PropertyBuilder	definisce le proprietà di un tipo dinamico
CustomAttributeBuilder	definisce gli attributi personalizzati
EventBuilder	definisce eventi per le classi
DynamicMethod	definisce e rappresenta un metodo dinamico che può essere compilato, eseguito e rimosso
ILGenerator	genera istruzione di codice intermedio IL
OpCodes	fornisce nei suoi campi le rappresentazioni delle varie istruzioni IL utilizzabili per l'emissione del rispettivo codice

La classe ILGenerator ricopre uno dei ruoli fondamentali nel namespace. Permette infatti di generare il codice intermedio che costituirà il corpo di un metodo o di un costruttore. Poiché essa non è istanziabile direttamente, non avendo un costruttore, una sua istanza viene creata utilizzando una delle altre classi Builder appena elencate. Una volta ottenuto un ILGenerator, potranno essere emesse le varie istruzioni di codice IL.

La classe DynamicMethod, invece, consente di generare un metodo dinamico, senza necessità di costruire un intero assembly e il tipo che lo contengano. Quindi è utilizzabile solo per compiti molto semplici, in generale per creare un metodo da eseguire dinamicamente e quindi da buttare via (facendolo eliminare dal Garbage Collector).

Generazione di metodi dinamici

Il seguente esempio mostra come utilizzare la classe DynamicMethod per creare e invocare un metodo che stampi la stringa "Hello World" a runtime.

L'obiettivo sarà cioè la generazione di un metodo come il seguente:

```
static void HelloMethod()
{
Console.WriteLine("Hello World");
}
```

Il primo passo è istanziare la classe, indicando il nome del metodo come primo parametro del costruttore. Il secondo e il terzo parametro rappresentano invece il tipo di ritorno e un array dei tipi di eventuali parametri. Nel nostro caso, il tipo di ritorno è void e non c'è alcun parametro:

```
var dynMethod = new DynamicMethod("HelloMethod", null, null);
```

A questo punto, possiamo ricavare un ILGenerator e usarlo per emettere le istruzioni di stampa e di ritorno del metodo:

```
ILGenerator gen = dynMethod.GetILGenerator();
gen.EmitWriteLine("Hello World");
gen.Emit(OpCodes.Ret);
```

Infine, il metodo può essere eseguito. Essendo statico non è necessario indicare alcun oggetto su cui invocarlo e non vi sono parametri da passare:

```
dynMethod.Invoke(null, null);
```

Il risultato sarà la stampa della stringa "Hello World".

Creazione di assembly dinamici

Nel precedente paragrafo si è visto come generare un metodo dinamico isolato, cioè non contenuto in alcun tipo e quindi in alcun assembly, facendo uso della classe DynamicMethod. Tale approccio è molto rapido e veloce, ma non consente di generare interi programmi o librerie dinamicamente.

Ora vedremo invece come utilizzare le altre classi del namespace System.Reflection.Emit per generare un intero assembly in memoria. In particolare, l'esempio genererà all'interno dell'assembly una singola classe dotata di un metodo e, infine, vedremo anche come salvare l'assembly in un file di libreria con estensione .dll.

Il primo passo è creare un AssemblyBuilder e, poiché un assembly vive all'interno di un application domain, utilizzeremo la classe AppDomain, esattamente il metodo DefineDynamicAssembly, per ricavarne un'istanza.

Il nome dell'assembly viene assegnato mediante un oggetto AssemblyName:

```
AssemblyName name = new AssemblyName("DynamicAssembly");
AssemblyBuilder assemblyBuilder= AppDomain.CurrentDomain.DefineDynamicAssembly(name,
AssemblyBuilderAccess.RunAndSave);
```

Il metodo DefineDynamicAssembly, oltre al parametro name, può utilizzare un secondo parametro di tipo AssemblyBuilderAccess, che indica la modalità di accesso all'assembly che si sta costruendo. In questo caso, il valore RunAndSave indica che esso potrà essere eseguito e salvato.

Con l'oggetto assemblyBuilder appena creato si ricava ora il modulo, all'interno del quale potrà trovare posto il seguente tipo:

```
ModuleBuilder moduleBuilder= assemblyBuilder.DefineDynamicModule("DynamicModule","DynamicAssembly.dll");
TypeBuilder tb= moduleBuilder.DefineType("HelloClass", TypeAttributes.Class | TypeAttributes.Public);
```

Si noti che è stato indicato anche il nome dell'assembly, che non è obbligatorio, ma ci servirà quando dovremo salvare l'assembly su file.

Il parametro TypeAttributes utilizzato all'interno del metodo DefineType permette di specificare, combinando i valori dell'enumerazione omonima, modificatori e attributi del tipo che si vuol creare. In questo caso, il tipo sarà una classe con accesso pubblico, denominata HelloClass.

Passiamo ora a generare il metodo PrintHello, che stamperà una stringa Hello combinandola con la stringa passata come argomento al metodo stesso:

```
MethodBuilder mb= tb.DefineMethod("PrintHello", MethodAttributes.Public, null, new Type[]{typeof(string)});
```

Il parametro null indica il tipo di ritorno, che quindi è void, mentre l'ultimo è l'array dei tipi passati come parametri, in questo caso un'unica stringa.

Infine, dal MethodBuilder si può ora ottenere l'oggetto ILGenerator con il quale creare il codice IL del corpo del metodo.

Il seguente blocco di istruzioni è commentato in maniera da mostrare i singoli passaggi:

```
ILGenerator myMethodIL = mb.GetILGenerator();
//definisce una stringa contenente il valore "Hello"
myMethodIL.Emit(OpCodes.Ldstr, "Hello");
//carica il valore del parametro passato al metodo
myMethodIL.Emit(OpCodes.Ldarg_1);
//concatena le due stringhe, per mezzo del metodo String.Concat
MethodInfo concatMethod = typeof(String).GetMethod("Concat", new Type[] { typeof(string), typeof(string) });
myMethodIL.Emit(OpCodes.Call, concatMethod);
//stampa la stringa per mezzo del metodo Console.Write
MethodInfo writeMethod = typeof(Console).GetMethod("Write", new Type[] { typeof(string) });
myMethodIL.Emit(OpCodes.Call, writeMethod);
//esce dal metodo
myMethodIL.Emit(OpCodes.Ret);
```

Ora che abbiamo creato il metodo, possiamo ricavare il tipo, istanziarlo e utilizzarlo, per mezzo dei metodi di reflection che abbiamo visto nei paragrafi precedenti:

```
Type helloType=tb.CreateType();
object helloObj= Activator.CreateInstance(helloType);
MethodInfo helloInstanceMethod= helloType.GetMethod("PrintHello", new Type[] { typeof(string) });
helloInstanceMethod.Invoke(helloObj, new object[]{ "Antonio" });
```

L'assembly dinamico così creato può essere salvato, visto che abbiamo usato la modalità RunAndSave.

L'altra operazione necessaria è l'assegnazione di un filename al modulo, che deve coincidere con quello dell'assembly se è l'unico modulo (e infatti per il ModuleBuilder abbiamo usato "DynamicAssembly.dll").

Non resta che invocare il metodo Save dell'oggetto assemblyBuilder:

```
assemblyBuilder.Save("DynamicAssembly.dll");
```

All'interno della directory di esecuzione dell'applicazione di base, quella che ha generato l'assembly dinamico, vi ritroverete la nuova DLL, utilizzabile come un qualsiasi assembly .NET, per esempio aggiungendo un riferimento a essa da un altro progetto in Visual Studio.

C:\Dev\esempi_csharp\Cap 14 Reflection\Code Generation\bin\Deb...

File View Help

- C:\Dev\esempi_csharp\Cap 14 Reflection\Code Generation\bin\Debug\DynamicAssembly.dll
  - MANIFEST
  - HelloClass
    - .class public auto ansi
    - .ctor : void()
    - PrintHello : void(string)

HelloClass::PrintHello : void(string)

Find Find Next

```
.method public instance void  PrintHello(string A_1) cil managed
{
  // Code size       17 (0x11)
  .maxstack  2
  IL_0000:  ldstr      "Hello "
  IL_0005:  ldarg.1
  IL_0006:  call       string [mscorlib]System.String::Concat(string,
                                                              string)
  IL_000b:  call       void [mscorlib]System.Console::Write(string)
  IL_0010:  ret
} // end of method HelloClass::PrintHello
```

```
.assembly DynamicAssembly
{
  .ver 0:0:0:0
```

Figura 14.3 – Codice IL di assembly dinamico decompilato da ILDasm.

Se si vuol modificare la directory di destinazione del salvataggio, bisogna specificarla come ulteriore parametro del metodo DefineDynamicAssembly.

La Figura 14.3 mostra l'assembly generato e salvato, all'interno di ILDasm, in cui il codice IL risultante dalla decompilazione corrisponde, com'è naturale che sia, a quello creato dinamicamente.

Attributi

Gli *attributi* sono un meccanismo di estensione degli elementi di un programma, che permette di aggiungere a tipi, membri, parametri, valori di ritorno, parametri di tipi generici e assembly, dei metadati personalizzati.

Ogni attributo è definito mediante una classe derivata dalla classe System.Attribute, quindi è anche possibile creare dei propri attributi personalizzati. Essi verranno poi compilati direttamente all'interno dell'assembly e potranno essere ricavati dinamicamente mediante reflection.

Per *decorare* un elemento con un attributo si utilizza un'apposita sintassi, facendo precedere l'elemento stesso dal nome dell'attributo racchiuso fra parentesi quadre:

```
[Obsolete]
public class MiaClasse
{
}
```

L'esempio precedente assegna alla classe MiaClasse un attributo chiamato Obsolete. Una volta compilata la classe, l'attributo verrà memorizzato fra i metadati dell'assembly. Il compilatore, o un altro strumento, può ricavare quindi gli attributi assegnati a un elemento e agire di conseguenza. In questo caso, per esempio, può avvisare lo sviluppatore con un warning, indicando che la classe non dovrebbe più essere utilizzata in quanto segnalata come obsoleta.

L'attributo Obsolete è definito nella libreria di classi di base per mezzo della classe ObsoleteAttribute. Esplorando la libreria .NET troverete molte classi di attributi che rispettano tale nomenclatura standard, cioè con il suffisso Attribute posto dopo il nome dell'attributo stesso. Essa, però, non è solo una convenzione: come nel caso dell'attributo Obsolete, permette di riferirsi all'attributo stesso e utilizzarlo per decorare elementi di codice omettendone il suffisso Attribute. Si noti che l'abbreviazione è consentita da C#, ma non tutti i linguaggi .NET la supportano. Ciò non toglie che anche in C# sia sempre possibile utilizzare il nome completo della classe:

```
[ObsoleteAttribute]
public class MiaClasse
{
}
```

Gli attributi possono essere applicati a più tipologie di elementi mediante delle regole stabilite da ogni classe che li implementa, come vedremo più avanti. Per esempio, l'attributo Obsolete può anche decorare un singolo metodo all'interno di una classe:

```
public class MiaClasse
{
[Obsolete]
public void OldMethod()
{
}
}
```

Ogni elemento inoltre supporta la presenza di molteplici attributi e in alcuni casi lo stesso attributo può essere applicato più volte allo stesso elemento. Per esempio, una classe può essere marcata come serializzabile e magari in un secondo tempo come obsoleta, semplicemente aggiungendo un attributo sotto l'altro:

```
[Serializable]
[Obsolete]
public class MiaClasse
{
}
```

Attributi multipli possono anche essere specificati con una sintassi differente, separandoli con una virgola, all'interno delle stesse parentesi quadre:

```
[Serializable, Obsolete]
public class MiaClasse
{
}
```

Un attributo, essendo alla fine una classe come un'altra, può anche accettare dei parametri per la sua costruzione. Per esempio, l'attributo Obsolete possiede, oltre al costruttore di default senza parametri, altri due costruttori. Il primo accetta un parametro string, utilizzabile per indicare un messaggio per cui l'elemento che decora non deve più essere utilizzato. Per passare tale parametro, si usa la classica sintassi mediante parentesi tonde, come se si stesse creando un'istanza dell'attributo mediante l'operatore new:

```
[Obsolete("Classe non più supportata")]
public class MiaClasse
{
}
```

Si noti che in questo caso non viene allocato in memoria alcun oggetto, il parametro string viene semplicemente memorizzato fra i metadati dell'assembly e può essere letto mediante reflection o da tool esterni.

Se si provasse ora a compilare la classe precedente e a utilizzarla in qualche parte del nostro codice, il compilatore e, quindi, Visual Studio avviserebbero dell'utilizzo di una classe obsoleta, fornendo un avviso e un tooltip con il messaggio impostato tramite l'attributo, come mostrato in Figura 14.4.

Cap 14 Reflection - Microsoft Visual Studio

FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ANALYZE WINDOW HELP DEVEXPRESS

Start Debug

Program.cs Object Browser ClassDiagram1.cd Program.cs App.config

CustomAttributes.Program Main(string[] args)

```
namespace CustomAttributes
{
    [Serializable]
    [Obsolete("Classe non più supportata")]
    class MiaClasse
    {
        [Obsolete]
        public void Method()
        { }

        public int MyProp { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            MiaClasse obj = new MiaClasse();
        }
    }
}
```

'CustomAttributes.MiaClasse' is obsolete: 'Classe non più supportata'

100 %

Error List

0 Errors | 2 Warnings | 0 Messages | Search Error List

	Description	File	Line	Colu...	Project
1	'CustomAttributes.MiaClasse' is obsolete: 'Classe non più supportata'	Program.cs	24	13	CustomAttributes
2	'CustomAttributes.MiaClasse' is obsolete: 'Classe non più supportata'	Program.cs	24	33	CustomAttributes

Figura 14.4 – L'attributo Obsolete provoca un warning del compilatore.

Un secondo costruttore di ObsoleteAttribute accetta anche un parametro booleano che, se viene impostato a true, permette di considerare l'utilizzo dell'elemento come un errore, e quindi ne impedisce la compilazione. Per esempio, se si volesse impedire completamente l'utilizzo del metodo MyMethod della classe, si potrebbe decorarlo come nel seguente esempio:

```
[Obsolete("Classe non più supportata")]
public class MiaClasse
{
[Obsolete("Il metodo provoca gravi errori", true)]
public void MyMethod()
{
...
}
}
```

Tentando di utilizzare il metodo, stavolta si avrebbe un errore di compilazione come mostrato in Figura 14.5.

```
namespace CustomAttributes
{
    [Serializable]
    [Obsolete("Classe non più supportata")]
    class MiaClasse
    {
        [Obsolete("Il metodo provoca errori", true)]
        public void Method()
        { }

        public int MyProp { get; set; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            MiaClasse obj = new MiaClasse();
            obj.Method();
        }
    }
}
```

[deprecated] void MiaClasse.Method()

Error:
'CustomAttributes.MiaClasse.Method()' is obsolete: 'Il metodo provoca errori'

100 %

Error List

1 Error | 2 Warnings | 0 Messages

	Description	File
1	'CustomAttributes.MiaClasse' is obsolete: 'Classe non più supportata'	Program.cs
2	'CustomAttributes.MiaClasse' is obsolete: 'Classe non più supportata'	Program.cs
3	'CustomAttributes.MiaClasse.Method()' is obsolete: 'Il metodo provoca errori'	Program.cs

Figura 14.5 – L'uso dell'attributo Obsolete per generare errori di compilazione.

Fra gli attributi di .NET più utili vi è anche ConditionalAttribute, che permette di decorare un elemento in maniera da renderlo dipendente da un simbolo di preprocessore. Anche qui Conditional è un alias di ConditionalAttribute e può essere applicato a un metodo o a una classe. Per esempio, se il metodo ShowMessage della classe seguente deve essere presente solo nel caso di una compilazione debug, si può marcarlo nel seguente modo:

```
class MiaClasse
{
[Conditional("DEBUG")]
public void ShowMessage(string message)
{
Console.WriteLine(message);
}
}

...
MiaClasse obj=new MiaClasse();
Obj.ShowMessage("messaggio stampato a debug");
```

Se utilizzate Visual Studio per compilare, nel caso di compilazione in configurazione Debug, la direttiva DEBUG è definita nelle proprietà di progetto e il metodo ShowMessage viene normalmente compilato e quindi eseguito. In modalità Release, invece, il metodo viene escluso dalla compilazione, così come ogni suo utilizzo.

Fra gli altri attributi e i loro utilizzi nel mondo reale, da notare quelli che permettono di assegnare una categoria alle proprietà. In questo modo, le proprietà di un controllo dell'interfaccia grafica selezionato nel designer appariranno raggruppate in diverse categorie all'interno della finestra delle proprietà di Visual Studio.

I nomi di queste categorie sono assegnati mediante l'attributo CategoryAttribute, applicato alle rispettive proprietà, e letti poi da Visual Studio o da altri IDE per creare il raggruppamento. Esistono diversi nomi standard, ottenibili da proprietà statiche della classe CategoryAttribute. che permettono così di rispettare delle regole e delle convezioni nella creazione di controlli personalizzati.

Attributi globali

Nel precedente paragrafo abbiamo visto cosa sono gli attributi e come sono utilizzabili per decorare degli elementi di un programma C#. Tuttavia, gli attributi possono anche essere usati a livello di assembly e moduli, e in questo caso sono detti *attributi globali*. Per applicare tali attributi, si utilizza il tag speciale [assembly:] facendolo precedere al nome dell'attributo stesso, oppure [module:] nel caso di attributi a livello di moduli.

Fra gli attributi globali, possiamo per esempio vedere come utilizzare l'attributo AssemblyTitle, che permette di assegnare un titolo all'assembly. Per applicare tale attributo all'assembly corrente, basta inserire in un file di codice sorgente l'istruzione seguente:

```
[assembly: AssemblyTitle("Titolo assembly")]
```

Dato che esso si riferisce all'intero assembly, non è necessario (né possibile, d'altronde) applicarlo a un particolare elemento di codice.

Gli attributi globali, per definizione, devono essere inseriti al di fuori di qualunque namespace, quindi all'inizio di un file di codice oppure in un apposito file. A tale scopo, Visual Studio utilizza un file di codice sorgente dedicato proprio a contenere questo e altri attributi dedicati all'assembly, chiamato AssemblyInfo.cs, che viene autogenerato ogni volta che si crea un progetto nell'ambiente di sviluppo.

Potete vederlo nel Solution Explorer espandendo il ramo Properties, come mostrato nella Figura 14.6.

Questi attributi permettono di specificare varie caratteristiche, come la descrizione, il titolo, l'azienda, informazioni di copyright e numero di versione.

Il seguente è il possibile contenuto standard di un file AssemblyInfo.cs:

Cap 14 Reflection - Microsoft Visual Studio

Quick Launch (Ctrl+Q)

FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ANALYZE DEVEXPRESS WINDOW HELP

Antonio Pellerti

Start · Debug

AssemblyInfo.cs* | AssemblyInfo.cs | Program.cs* | Object Browser

```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

// General Information about an assembly is controlled through the following
// set of attributes. Change these attribute values to modify the information
// associated with an assembly.
[assembly: AssemblyTitle("CustomAttributes")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("CustomAttributes")]
[assembly: AssemblyCopyright("Copyright © 2013")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

// Setting ComVisible to false makes the types in this assembly not visible
// to COM components.  If you need to access a type in this assembly from
// COM, set the ComVisible attribute to true on that type.
[assembly: ComVisible(false)]

// The following GUID is for the ID of the typelib if this project is exposed to COM
[assembly: Guid("c4bbe65e-6f8e-4890-90a9-7d9da5b4456a")]

// Version information for an assembly consists of the following four values:
//
//      Major Version
//      Minor Version
//      Build Number
//      Revision
//
// You can specify all the values or you can default the Build and Revision Numbers
// by using the '*' as shown below:
// [assembly: AssemblyVersion("1.0.*")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

100 %

Solution Explorer

Search Solution Explorer (Ctrl+;)

- CustomAttributes
  - Properties
    - AssemblyInfo.cs
  - References
  - App.config
  - Program.cs
  - VehicleTypeAttribute.cs

Solution Explorer | Team Explorer

Properties

AssemblyInfo.cs File Properties

Advanced

Build Action	Compile
Copy to Output Directory	Do not copy
Custom Tool	
Custom Tool Namespace	

Misc

File Name	AssemblyInfo.cs
Full Path	D:\dev\esempi_csharp\Cap 14 Re

Advanced

Web Publish Activity | Error List | Output | Find Symbol Results

Ready | Ln 1 | Col 1 | Ch 1 | INS

Figura 14.6 – Il file AssemblyInfo.cs in Visual Studio.

```
[assembly: AssemblyTitle("CustomAttributes")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("CustomAttributes")]
[assembly: AssemblyCopyright("Copyright © 2013")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
[assembly: ComVisible(false)]
[assembly: Guid("c4bbe65e-6f8e-4890-90a9-7d9da5b4456a")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

Visual Studio, inoltre, fornisce gli strumenti adatti per modificare tali valori senza costringere lo sviluppatore a mettere mano direttamente al codice. Infatti, aprendo le proprietà di un progetto, sarà possibile cliccare sul pulsante Assembly Information che a sua volta aprirà un'apposita finestra di inserimento dati (vedi Figura 14.7).

Fra gli attributi globali applicabili a un assembly, ve n'è uno che permette di rendere visibili tipi e membri con modificatore di accesso internal, quindi normalmente visibili solo all'interno del proprio assembly o anche all'interno di assembly "amici". Questo attributo è denominato InternalsVisibileTo; lo si utilizza specificando uno o più nomi di questi assembly amici nel seguente modo:

```
[assembly: InternalsVisibleTo("AssemblyFriend1")]
```

Così, i tipi e i membri internal dell'assembly corrente saranno visibili e utilizzabili all'interno dell'assembly AssemblyFriend1.


Assembly Information
Title:
CustomAttributes
Description:
Company:
Product:
CustomAttributes
Copyright:
Copyright © 2013
Trademark:
Assembly version:
1
0
0
0
File version:
1
0
0
0
GUID:
c4bbe63e-6f8e-4890-90a9-7d9da5b4456a
Neutral language:
(None)
Make assembly COM-Visible
OK
Cancel

Figura 14.7 – Modifica degli attributi di un assembly dalle proprietà di progetto in Visual Studio.

Leggere attributi

Nel precedente paragrafo abbiamo imparato cosa sono gli attributi e come sono utilizzati dal compilatore o da altri strumenti esterni.

Per mezzo delle classi, con relativi metodi e proprietà, viste parlando di reflection, possiamo leggere tali attributi e far anche eseguire al nostro codice delle azioni condizionate dalla loro presenza. Ciò risulterà particolarmente utile una volta visto come creare degli attributi custom.

Gli attributi presenti all'interno di un assembly possono essere elencati utilizzando il metodo GetCustomAttributes della classe Assembly:

```
var attributes = Assembly.GetExecutingAssembly().GetCustomAttributes();
foreach (Attribute attr in attributes)
{
Console.WriteLine(attr.GetType().Name);
}
```

Il metodo GetCustomAttributes restituisce in questo caso un'enumerazione di oggetti derivati dalla classe Attribute, di cui viene quindi ricavato il tipo e stampato il nome.

Noterete, eseguendo l'esempio, che per l'assembly in esecuzione, sebbene non sia stato definito alcun attributo personalizzato, sarà stampato un elenco abbastanza corposo di attributi. Sono in maggioranza quelli globali descritti nel precedente paragrafo e impostati nel file AssemblyInfo.cs.

Per leggere un particolare attributo, e le relative proprietà, si può utilizzare il metodo GetCustomAttribute di Assembly, per esempio nella versione generica, che permette di specificare il tipo dell'attributo stesso:

```
AssemblyTitleAttribute assemblyTitle = Assembly.GetExecutingAssembly().GetCustomAttribute<AssemblyTitleAttribute>();
Console.WriteLine("AssemblyTitle: {0}", assemblyTitle.Title);
```

Nel prossimo paragrafo, si vedranno più in dettaglio i vari attributi applicabili a livello di assembly e la relativa spiegazione.

Se si desidera invece leggere un attributo o l'elenco di attributi che decorano un elemento di codice, basta utilizzare il metodo GetCustomAttributes fornito dalla classe che rappresenta l'elemento stesso, ereditato da MemberInfo. Per esempio, data la classe MiaClasse vista in precedenza in questo capitolo, possiamo ricavarne gli attributi nel seguente modo:

```
var attributes= typeof(MiaClasse).GetCustomAttributes();
foreach (Attribute attr in attributes)
{
Console.WriteLine(attr.GetType().Name);
}
```

In questo caso, il metodo GetCustomAttributes restituisce gli attributi dell'istanza di Type, ricavata da MiaClasse. Se volessimo ricavare i membri della classe e gli eventuali attributi, potremmo utilizzare il metodo GetCustomAttributes su ogni elemento MemberInfo:

```
var members= typeof(MiaClasse).GetTypeInfo().DeclaredMembers;
foreach (MemberInfo mi in members)
{
attributes = mi.GetCustomAttributes();
if (attributes.Any())
{
Console.WriteLine("{0} {1} attributes", mi.MemberType, mi.Name);
foreach (Attribute attr in attributes)
{
Console.WriteLine(attr.GetType().Name);
}
}
}
```

Un'altra maniera decisamente più semplice permette di verificare se un attributo di un dato tipo è applicato a un determinato membro: essa prevede l'uso del metodo IsDefined;

```
Type t = typeof(MiaClasse);
bool defined=t.IsDefined(typeof(ObsoleteAttribute));
```

Nell'esempio precedente, il metodo IsDefined restituirà il valore true se la classe MiaClasse è decorata con l'attributo Obsolete.

Creare attributi personalizzati

Dato che ogni attributo è una classe derivata da System.Attribute, la creazione di un attributo personalizzato o custom è possibile mediante l'implementazione di una nuova classe derivata da essa. Rispettando le regole di nomenclatura, e riprendendo la nostra gerarchia di classi per veicoli più volte utilizzata nel corso del libro, possiamo per esempio creare un attributo per indicare la tipologia di veicolo:

```
class VehicleTypeAttribute : Attribute
{
public string TypeDescription { get; set; }

public VehicleTypeAttribute(string desc)
{
TypeDescription = desc;
}
}
```

In questo caso, la classe possiede una proprietà di tipo string destinata a memorizzare la descrizione del tipo di veicolo. Non cambia nulla, rispetto a quanto visto finora, sull'applicazione dell'attributo a una classe:

```
[VehicleType("Automobile")]
class Car
```

```
{
}
```

L'attributo prevede necessariamente l'uso di un parametro, altrimenti si verificherebbe un errore di compilazione a indicare che la classe VehicleTypeAttribute non possiede un costruttore predefinito senza parametri.

Un particolare attributo di .NET viene utilizzato proprio per decorare le classi di attributi personalizzati. Si tratta di AttributeUsage, che serve a istruire il compilatore su come l'attributo destinazione debba comportarsi.

La classe AttributeUsage ha tre proprietà. AllowMultiple, di tipo bool, indica se l'attributo può essere applicato più volte allo stesso elemento. Inherited, anch'essa di tipo bool, indica se l'attributo viene ereditato dalle classi derivate dall'attributo personalizzato, oppure, nel caso di metodi, se esso viene applicato anche ai loro override. Infine, la proprietà ValidOn restituisce l'insieme di elementi cui l'attributo può essere applicato; quest'ultima viene impostata per mezzo del costruttore, al quale può essere passata una qualunque combinazione di valori dell'enumerazione AttributeTargets, che sono i seguenti:

```
All
Assembly
Class
Constructor
Delegate
Enum
Event
Field
GenericParameter
Interface
Method
Module
Parameter
Property
ReturnValue
Struct
```

La classe ObsoleteAttribute, per esempio, ha questa definizione:

```
[AttributeUsage(AttributeTargets.Class|AttributeTargets.Constructor|AttributeTargets.Delegate|AttributeTargets.Enum|Attribute
Inherited = false)]
public sealed class ObsoleteAttribute : Attribute
{}
```

Quindi, nell'implementazione di un attributo personalizzato, un secondo passo è quello di applicare alla classe l'attributo AttributeUsage per indicare gli elementi sui quali si può applicare, ed eventualmente impostare anche le sue altre proprietà. Se, per esempio, il nostro attributo VehicleTypeAttribute deve poter essere utilizzato solo per classi e struct, si può specificare nel seguente modo:

```
[AttributeUsage(AttributeTargets.Class|AttributeTargets.Struct, Inherited = false, AllowMultiple = true)]
class VehicleTypeAttribute : Attribute
{
}
```

Per mezzo del costruttore è poi possibile indicare quali parametri saranno passati all'attributo stesso. Il costruttore di VehicleTypeAttribute prende come parametro una stringa, che rappresenta la descrizione del tipo di veicolo:

```
public VehicleTypeAttribute(string typeDescription)
{
this.typeDescription = typeDescription;
}
```

I parametri dell'attributo impostati per mezzo del costruttore sono anche detti *parametri posizionali*, in quanto l'assegnazione dei relativi argomenti dipende appunto dalla posizione. Per esempio, decorando la classe Car con l'attributo VehicleType, si può impostare il valore "Automobile" come descrizione:

```
[VehicleType("Automobile")]
class Car
{
}
```

A questo punto, se il compilatore trova un costruttore della classe VehicleTypeAttribute che prende una singola stringa come parametro, esso potrà generare i metadati appropriati e aggiungerli all'assembly.

Una seconda modalità di passaggio di parametri a un attributo è quella dei *parametri con nome*, che diverranno dei parametri opzionali. Per aggiungere un parametro opzionale, basta aggiungere alla classe dell'attributo personalizzato un campo o una proprietà pubblica. Per esempio, se la classe VehicleTypeAttribute avesse una proprietà pubblica, denominata Potenza, come nel seguente caso:

```
[AttributeUsage(AttributeTargets.Class|AttributeTargets.Struct, Inherited = false, AllowMultiple = true)]
class VehicleTypeAttribute : Attribute
{
public int Potenza {get;set;}
}
```

a essa potrebbe essere assegnato un valore in fase di impostazione dell'attributo, utilizzando una sintassi abbastanza naturale:

```
[VehicleType("Auto", Potenza=100)]
class Vehicle
{
…
}
```

In questo caso, il compilatore non cerca un costruttore con un secondo parametro, ma va ad assegnare alla proprietà Potenza il valore indicato.

Visual Studio consente di semplificare notevolmente l'implementazione di un attributo personalizzato mediante un apposito snippet di codice. Basta infatti digitare la parola Attribute, seguita da due pressioni del tasto TAB, ed esso genererà uno scheletro completo di classe di attributo, dotata di un costruttore con un parametro e di un'ulteriore proprietà, simile al seguente:

```
[AttributeUsage(AttributeTargets.All, Inherited = false, AllowMultiple = true)]
sealed class MyAttribute : Attribute
{
// See the attribute guidelines at
// http://go.microsoft.com/fwlink/?LinkId=85236
readonly string positionalString;

// This is a positional argument
public MyAttribute(string positionalString)
{
this.positionalString = positionalString;

// TODO: Implement code here
throw new NotImplementedException();
}
public string PositionalString
{
get { return positionalString; }
}

// This is a named argument
public int NamedInt { get; set; }
}
```

Ora è sufficiente apportare le proprie personalizzazioni per ottenere più rapidamente un nostro attributo custom.

Informazioni sul chiamante

Fra le novità di C# 5, vi è la funzionalità nota come *Caller Info Attributes*.

Avete presente il servizio di molti gestori telefonici che fornisce informazioni sul numero chiamante? Applicatelo a un programma C# e pensate al chiamante come al membro che invoca un altro metodo. Soprattutto per scopi di debug e diagnostica, spesso può essere utile conoscere il chiamante e ottenerne informazioni di vario genere. Prima di C# 5, era possibile ricavare buona parte di tali informazioni utilizzando la classe StackFrame e le funzionalità di reflection, oppure per mezzo di apposite librerie di logging, spesso open source, come Log4Net.

```
private static void Log(string message)
{
var sf = new StackFrame(1, true); // sale di 1 livello dal metodo corrente
var methodName = sf.GetMethod().Name;
var sourceFile = sf.GetFileName();
var lineNumber = sf.GetFileLineNumber();
Console.WriteLine("[{2} -- {0}] - {1} ({3}:{4})",
methodName, message, DateTime.Now, sourceFile, lineNumber);
}
```

In C# 5.0, gli attributi *Caller Info* possono essere applicati a dei parametri opzionali di un metodo e indicano al compilatore di assegnare a essi dei valori ottenuti dal metodo chiamante.

In particolare, esistono tre attributi:

- CallerMemberName è il nome del membro chiamante;
- CallerfilePath è il percorso del file contenente il sorgente del chiamante;
- CallerLineNumber è il numero di riga del chiamante all'interno del file sorgente.

Il seguente metodo mostra come usare i tre attributi, che verranno utilizzati dal compilatore per assegnare i valori ai corrispondenti parametri:

```
public static void LogInfo(
[CallerMemberName] string memberName = null,
[CallerFilePath] string filePath = null,
[CallerLineNumber] int lineNumber = 0)
{
Console.WriteLine(memberName);
Console.WriteLine(filePath);
Console.WriteLine(lineNumber);
}
```

Il metodo non fa altro che stampare i valori dei suoi argomenti. Normalmente, il compilatore assegnerebbe ai parametri opzionali i valori indicati nella firma del metodo, ma, con l'uso

degli attributi *Caller Info*, a essi sarà assegnato un valore ricavato direttamente dal contesto. Per esempio, utilizzando ora il metodo LogInfo, all'interno del metodo Main:

```
class Program
{
static void Main(string[] args)
{
LogInfo();
}
}
```

al parametro memberName sarà assegnato il nome del metodo chiamante, in questo caso Main; al secondo, il percorso completo del file in cui è definita la classe Program; al terzo, il numero di riga all'interno del file precedente al quale avviene l'invocazione del metodo LogInfo.

Se il metodo LogInfo venisse invocato all'interno di una proprietà, CallerMemberName restituirebbe il nome di quest'ultima. Tale evenienza è particolarmente utile nell'implementazione dell'interfaccia INotifyPropertyChanged, parecchio utilizzata in applicazioni con interfaccia grafica definita in XAML, in quanto richiede di ricavare e indicare il nome della proprietà, procedura che, senza tale attributo, andrebbe eseguita manualmente.

Programmazione dinamica

C# è soprattutto conosciuto come un linguaggio fortemente o staticamente tipizzato. Ciò significa che il tipo di ogni oggetto può essere determinato dal compilatore, prima dell'esecuzione stessa del programma.

La possibilità di assegnare e risolvere un tipo e i suoi membri a runtime, piuttosto che a tempo di compilazione, viene chiamata, al contrario, *binding dinamico*. Tale caratteristica è utile quando il compilatore non può ancora conoscere il tipo che, per esempio, una variabile assumerà a runtime, mentre lo sviluppatore è certo di esso e quindi sa già come poterlo utilizzare.

Alcuni esempi pratici di questi scenari possono essere quelli in cui degli oggetti vengono creati e restituiti da API COM, dal DOM di HTML (i cui oggetti possono essere creati con vari strumenti e linguaggi), da linguaggi prettamente dinamici come *IronPython* o *IronRuby*; oppure esempi con oggetti definiti direttamente in modo dinamico, come quelli *expando* o gli oggetti *dynamic* personalizzati che vedremo a breve.

In tali casi, è spesso possibile rivolgersi alle funzionalità di reflection in maniera da ricavare le caratteristiche dei tipi come visto in precedenza oppure, ove ciò non fosse possibile, sfruttare il supporto alla programmazione dinamica introdotto in C# 4.0, che quindi non può essere più considerato solo un linguaggio fortemente tipizzato.

Il concetto di linguaggio dinamico non è certamente nuovo o recente, ma negli ultimi anni, grazie a linguaggi come Python e Ruby, è tornato fortemente alla ribalta.

Nel caso di .NET, e quindi di C#, è in particolare il Dynamic Language Runtime (DLR) che fornisce delle nuove API e una keyword, `dynamic`, con cui rappresentare un nuovo tipo di dati.

DLR

Il *Dynamic Language Runtime* (DLR) è un ambiente di esecuzione che aggiunge al CLR un insieme di servizi dedicati a linguaggi che supportano la programmazione dinamica.

NOTA	Sebbene il nome possa fare intuire ciò, il DLR non è una versione dinamica del CLR. Esso è una libreria costituita da normali assembly, utilizzati dal CLR.

Il compito principale del DLR è unificare, mediante i suoi servizi, le procedure con cui i linguaggi possano utilizzare tipi e membri in maniera dinamica. In tal modo, un linguaggio statico come C# può utilizzare gli stessi servizi che userebbe un linguaggio dinamico come

IronPython. In generale, sarà anche più semplice sviluppare e aggiungere nuovi linguaggi dinamici al framework .NET.

I linguaggi dinamici possono identificare il tipo di un oggetto a runtime, mentre in genere un linguaggio fortemente tipizzato, come lo è C# nella stragrande maggioranza dei casi (e nella totalità di quelli visti finora), deve necessariamente specificare il tipo esatto di ogni oggetto, in maniera che il compilatore possa eseguire il binding statico di tipi e membri.

Con il termine *binding* si intende appunto il processo di verifica di tipi, membri e altre operazioni.

La Figura 14.8 mostra l'architettura generale del Dynamic Language Runtime.

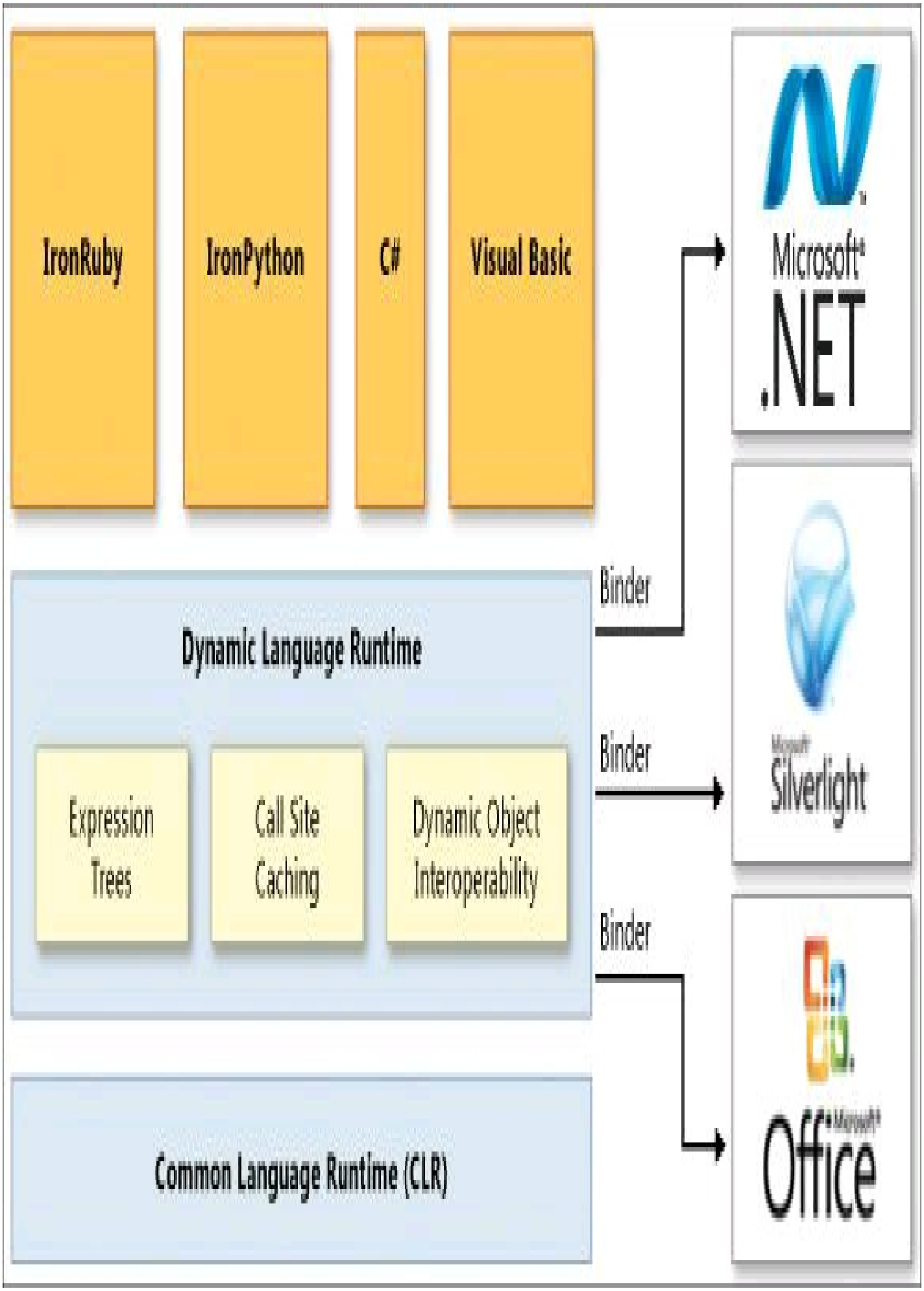

IronRuby
IronPython
C#
Visual Basic
Dynamic Language Runtime
Expression Trees
Call Site Caching
Dynamic Object Interoperability
Common Language Runtime (CLR)
Binder
Binder
Binder
Microsoft .NET
Microsoft Silverlight
Microsoft Office

Figura 14.8 – Architettura del DLR.

Ogni linguaggio con supporto dinamico ha il proprio insieme di *binder*, il cui compito è quello di identificare a runtime i target (per esempio tipi, metodi, proprietà e così via) in maniera da poterli poi utilizzare per eseguire le operazioni previste. Ricordiamoci, infatti, che in ogni caso il runtime ha bisogno di tipi precisi per poterli istanziare e invocarne i membri.

Nel caso di C#, questi tipi binder fanno parte del namespace Microsoft.CSharp.RuntimeBinder, contenuto nell'assembly Microsoft.CSharp.dll.

I servizi principali che il DLR aggiunge all'infrastruttura generale di .NET e del CLR sono i tre seguenti:

- *Expression tree* – il DLR usa gli alberi di espressioni per rappresentare la semantica di un linguaggio. Esso estende gli alberi di espressioni, già visti in LINQ, in maniera da includere controllo di flusso, assegnazione e altri meccanismi;
- *Call site caching* – per migliorare le prestazioni, il DLR mantiene una cache dei tipi utilizzati e informazioni sulle operazioni eseguite in precedenza, in maniera da poterle rieseguire più rapidamente e con maggiore efficacia nel caso in cui esse debbano essere eseguite nuovamente in seguito;
- *Dynamic object interoperability* – il DLR fornisce un insieme di classi e interfacce che rappresentano oggetti dinamici e le loro relative operazioni, e che possono essere utilizzati da linguaggi che implementano caratteristiche dinamiche. Questi tipi includono IDynamicMetaObjectProvider, DynamicMetaObject, DynamicObject e ExpandoObject.

Il tipo dynamic

L'unica aggiunta a livello di sintassi apportata in C# per il supporto alla programmazione dinamica è l'introduzione della parola chiave dynamic. A partire da C# 4.0, essa indica un nuovo tipo, chiamato appunto dynamic, che entra a far parte dei tipi predefiniti del linguaggio. Una variabile di tipo dynamic può mantenere il riferimento a un oggetto di qualunque tipo supportato da C#.

Ovunque sia possibile utilizzare uno dei tipi di C#, sarà possibile usare anche un tipo dynamic. Quindi, di tale tipo possono essere variabili, parametri, proprietà, tipi di ritorno, parametri di tipi generici e così via.

Internamente, il tipo dynamic può essere convertito in maniera implicita da e verso qualunque altro tipo .NET. Il tipo dynamic è quindi simile al tipo object: anche una variabile di tal tipo può essere utilizzata per assegnarle un qualunque oggetto.

Con dynamic, però, le cose vengono spinte oltre. Partiamo da questo esempio, che utilizza il tipo object, ma che, come intuirete facilmente, restituirebbe un errore se si tentasse di compilarlo:

```
object a=1;
object b=2;
object somma=a+b;
```

Nonostante le variabili a e b contengano due interi e siano memorizzate in due object (mediante boxing), l'istruzione di somma provocherebbe, così come è scritta, un errore di compilazione.

Il compilatore, infatti, non sa che i due oggetti a e b hanno a disposizione un operatore + di somma, perché esso li considera come object.

L'unica soluzione, nel caso specifico, sarebbe una conversione esplicita, cioè unboxing, dei numeri dai rispettivi oggetti, ma non è ciò che vogliamo.

A questo punto, proviamo a usare subito dynamic, sostituendo il tipo delle variabili:

```
dynamic a=1;
dynamic b=2;
dynamic somma=a+b;
```

E, quasi come per magia, il tutto funziona: il compilatore evita di effettuare il controllo sui tipi, che vengono ricavati e riconosciuti solo a tempo di esecuzione, ed è solo a questo punto quindi che il CLR si occupa di utilizzare l'operatore + disponibile per sommare due interi.

NOTA	Utilizzando Visual Studio per scrivere l'esempio precedente, noterete che l'IDE non offre e non può offrire intellisense per le variabili di tipo dynamic. Esso, infatti, per poter ricavare i membri supportati da un oggetto deve conoscerne il tipo, cosa che invece non ha a disposizione in questo scenario.

A molti potrebbe sembrare un passo indietro nella disciplina quasi ferrea a cui un linguaggio fortemente tipizzato ci ha abituati, portandoci dunque a scrivere programmi.

Cosa succederebbe se tentassimo allora di utilizzare un metodo non supportato dagli oggetti?

Per esempio:

```
dynamic a="a";
dynamic b="b";
dynamic quoziente= a/b;
```

Il tipo string non ha un operatore /. ma il compilatore evita di controllarlo, come da noi istruito mediante l'uso di dynamic, e a runtime si verificherebbe inevitabilmente un'eccezione non gestita, del tipo:

```
RuntimeBinderException: Operator '/' cannot be applied to operands of type 'string' and 'string'
```

Il tipo dynamic, quindi, indica al compilatore di differire il binding, cioè il momento in cui effettuare la verifica di tipi e membri, rinviandolo al momento dell'esecuzione.

NOTA	Le parole chiavi var e dynamic, sebbene si assomiglino agli occhi di un programmatore inesperto, hanno un significato totalmente differente. Anzi: opposto. Ricordate che var indica che una variabile assumerà implicitamente un tipo, che però il compilatore conosce perfettamente. Il tipo dynamic, invece, sarà determinato a tempo di esecuzione.

Il binding dinamico è utile quando lo sviluppatore del programma sa già che certi tipi o certi membri esistono, mentre il compilatore non lo sa. Come detto, ciò può capitare in scenari di interoperabilità con linguaggi dinamici come IronPython o IronRuby, oppure con COM.

Implementazione di oggetti dinamici

Il DLR fornisce l'implementazione per riconoscere e utilizzare i tipi .NET, i tipi COM, i tipi supportati da linguaggi come IronPython e IRonRuby e così via. Questi tipi sono tutti utilizzabili dal punto di vista del programmatore, referenziandoli mediante il tipo dynamic. Il DLR si occuperà poi di eseguire a runtime il binding agli oggetti reali, e quindi ai membri utilizzati.

È possibile però creare nuovi tipi di oggetti dinamici, implementando l'apposita interfaccia IDynamicMetaObjectProvider. Un oggetto che implementa tale interfaccia può partecipare alla risoluzione dinamica dei tipi, come descritto fino ad adesso.

Il .NET Framework fornisce già due classi che implementano IDynamicMetaObjectProvider, DynamicObject ed ExpandoObject, anch'esse facenti parte del namespace System.Dynamic.

Sebbene l'interfaccia definisca un solo metodo, GetMetaObject, che restituisce l'oggetto DynamicMetaObject che rappresenta l'oggetto dinamico, la sua implementazione è abbastanza complessa. Per tale motivo, l'approccio consigliato per creare nuove classi dinamiche, anziché implementare manualmente l'interfaccia, è quello di derivare nuove classi direttamente da DynamicObject.

La classe ExpandoObject

La classe ExpandoObject permette di creare oggetti dinamici, da utilizzare mediante la parola chiave dynamic, ai quali possono essere aggiunti ed eliminati membri direttamente a runtime. Possono inoltre essere impostati o letti i valori delle sue proprietà.

Tale classe ha un unico costruttore, senza parametri, quindi è possibile creare un oggetto ExpandoObject nel seguente modo:

```
dynamic exo = new ExpandoObject();
```

Sarà possibile aggiungervi proprietà e assegnarvi valori direttamente come segue:

```
exo.Name = "Antonio";
exo.Surname = "Pelleriti";
Console.WriteLine("{0} {1}", exo.Name, exo.Surname);
```

La classe ExpandoObject implementa, fra le altre, l'interfaccia IDictionaryObject<string, object>, in maniera che ogni proprietà dinamica aggiunta a una sua istanza sia utilizzabile come una chiave del dizionario, e il relativo valore come oggetto corrispondente alla chiave:

```
IDictionary<string, object> dict = exo;
foreach(var key in dict.Keys)
{
Console.WriteLine("{0}={1}", key, dict[key]);
}
```

In questo modo, il dizionario può anche essere utilizzato per rimuovere dinamicamente una proprietà aggiunta in precedenza all'oggetto ExpandoObject:

```
dict.Remove("Surname");
```

La classe DynamicObject

La classe DynamicObject permette di definire quali operazioni possano essere eseguite su un oggetto dinamico e come eseguirle. Per esempio, si può specificare cosa accade quando si legge o si assegna un valore a una proprietà dell'oggetto, oppure quando si invoca un suo metodo o operatore.

Per creare un oggetto dinamico, derivandolo da DynamicObject, si potranno implementare differenti metodi, a seconda del comportamento che si vuol ottenere dal nuovo oggetto. Oltre ai consueti metodi derivati da System.Object, si potrà fornire l'implementazione di una serie di metodi con nome del tipo TryXXX, che consentiranno di invocare dinamicamente dei membri dell'oggetto e di impostare delle proprietà e tutti i costrutti elencati nella Tabella 14.3.

Tabella 14.3 - **Metodi di DynamicObject.**

Metodo	Rappresenta implementazione di
TryBinaryOperation TryUnaryOperation	operatore binario e unario
TryConvert	conversione verso un altro tipo
TryCreateInstance	costruttore
TryDeleteIndex	eliminazione di oggetti tramite indice
TryDeleteMember	eliminazione di un membro dell'oggetto
TryGetIndex TrySetIndex	impostazione e lettura di indicizzatori
TryGetMember TrySetMember	impostazione e lettura di una proprietà

TryInvoke	invocazione di un'istanza stessa della classe
TryInvokeMember	invocazione di un metodo della classe

Tali metodi permettono di definire il comportamento dinamico dell'oggetto.

Il seguente è un semplice esempio di classe derivata da DynamicObject, che implementa solo l'override del metodo TryInvokeMember:

```
class MyDynamicObject: DynamicObject
{
public override bool TryInvokeMember(InvokeMemberBinder binder, object[] args, out object result)
{
Console.WriteLine("invoke member {0}",binder.Name);
result = null;
return true;
}
}
```

Grazie a esso, è possibile creare un'istanza di MyDynamicObject per mezzo della keyword dynamic e invocare un metodo qualunque, come nel seguente esempio:

```
dynamic dyn=new MyDynamicObject();
dyn.Metodo1();// stampa invoke member Metodo1
```

L'invocazione del Metodo1 causerà l'invocazione del metodo TryInvokeMember, con un argomento binder, avente proprietà Name uguale al nome del metodo stesso.

Una classe derivata da DynamicObject può anche definire dei metodi standard. Infatti, il sistema di binding del DLR tenta prima di trovare un membro standard della classe. Per esempio, invocando un metodo, prima verifica che esso esista come metodo reale, e solo in caso negativo, come nell'esempio precedente, invoca il metodo TryInvokeMember.

La seguente classe DynamicDictionary rappresenta l'implementazione di un dizionario dinamico, derivando la classe DynamicObject, che in pratica è una sorta di replica della classe ExpandoObject. Essa utilizza un campo Dictionary interno e implementa gli override di TryGetMember e TrySetMember per l'impostazione e la lettura di valori del dizionario stesso mediante proprietà aggiunte dinamicamente:

```
public class DynamicDictionary : DynamicObject
{
Dictionary<string, object> dictionary = new Dictionary<string, object>();

public override bool TryGetMember(GetMemberBinder binder, out object result)
{
return dictionary.TryGetValue(binder.Name, out result);
}

public override bool TrySetMember(SetMemberBinder binder, object value)
{
dictionary[binder.Name] = value;
return true;
```

```
}
public override bool TryInvokeMember(InvokeMemberBinder binder, object[] args, out object result)
{
Type dictType = typeof(Dictionary<string, object>);
try
{
result = dictType.InvokeMember(binder.Name, BindingFlags.InvokeMethod, null, dictionary, args);
return true;
}
catch
{
result = null;
return false;
}
public void Print()
{
foreach (var pair in dictionary)
Console.WriteLine(pair.Key + " " + pair.Value);
}
}
```

Possiamo ora istanziare un oggetto DynamicDictionary e impostarne delle proprietà dinamicamente, come nel seguente blocco di istruzioni:

```
dynamic veicolo = new DynamicDictionary();

//l'impostazione delle proprietà utilizza TrySetMember
veicolo.Marca = "Alfa Romeo";
veicolo.Modello = "GT";
veicolo.Targa = "AB123CD";

//la lettura delle proprietà utilizza TryGetMember
Console.WriteLine("{0}", veicolo.Marca);
```

Quando si imposta una proprietà, il runtime invoca il metodo TrySetMember, aggiungendo una nuova chiave al dizionario interno con nome uguale a quello della proprietà.

Al contrario, la lettura di una proprietà causa l'invocazione di TryGetMember, che restituisce il valore del dizionario corrispondente alla chiave con il nome della proprietà.

Ecco invece due esempi di utilizzo di metodi della classe:

```
//Print è un metodo Standard
veicolo.Print();
//invocazione di Clear utilizza tryInvoleMember
veicolo.Clear();
```

Si noti che l'invocazione di Print si traduce nell'utilizzo del metodo implementato in maniera standard all'interno della classe, mentre il metodo Clear provoca l'utilizzo del metodo TryInvokeMember, che cerca un metodo con il nome specificato fra quelli del tipo Dictionary.

Domande di riepilogo

1) La reflection è un meccanismo del .NET Framework per:

a. accedere ai metadati di un assembly e utilizzare i tipi e i loro membri a runtime
b. compilare un assembly a runtime a partire da codice IL
c. decompilare un assembly a runtime
d. convertire il codice IL di un assembly in codice eseguibile

2) Per ottenere il tipo di un oggetto x a runtime si utilizza:

a. Type type = typeof(x);
b. Type type = x.GetType();
c. Type type = new Type(x);
d. Type type = (Type)x;

3) Per ottenere i metodi privati definiti dal tipo String, si utilizza:

a. typeof(string).GetMethods();
b. typeof(string).GetMethods(BindingFlags.Private);
c. typeof(string).GetMethods(BindingFlags.NonPublic | BindingFlags.DeclaredOnly);
d. typeof(string).GetMethods(BindingFlags.Instance | BindingFlags.NonPublic | BindingFlags.DeclaredOnly);

4) Per ricavare l'assembly del programma in esecuzione, si utilizza:

a. Assembly.GetExecutingAssembly();
b. Assembly.GetCurrentAssembly();
c. Assembly.CurrentAssembly;
d. AppDomain.CurrentDomain.GetExecutingAssembly();

5) Per creare un'istanza t di un tipo T, con un costruttore predefinito senza parametri, si utilizza:

a. T t= T.CreateInstance(null);
b. T t= Type.GetType(typeof(T));
c. T t= (T)Activator.CreateInstance(typeof(T));
d. T t= Activator.CreateInstance<T>();

6) Per creare un attributo personalizzato, si deve derivare una classe dalla classe base:

a. System.CustomAttribute
b. System.Attribute

c. System.AttributeType

d. System.AttributeUsage

7) Invocando dal metodo Main il metodo Log seguente:

```
public static void Log([CallerMemberName] string memberName = null)
{ Console.Write(memberName);}
```

il valore stampato del parametro memberName sarà:

a. Log

b. memberName

c. Main

d. Main.Log

8) Qual è il valore di somma dopo l'esecuzione del seguente codice?

```
dynamic a = 1;
dynamic b = "b";
dynamic somma = a + b;
```

a. 1

b. "1b"

c. Eccezione a runtime

d. Errore di compilazione

Accesso ai dati

La gestione e l'accesso ai dati sono processi essenziali: questo capitolo mostra come eseguire input e output su file e directory, accedere a database relazionali usando ADO.NET o tramite strumenti ORM come LINQ to SQL e Entity Framework.

Variabili, campi di classe, collezioni, array: offrono tutti una locazione solo temporanea per la memorizzazione dei dati.

Un'applicazione reale, invece, avrà spesso a che fare con dati che dovranno essere conservati e d'altro canto letti per mezzo di diverse tipologie di sorgenti: file, database, servizi esposti da altre applicazioni tramite canali di vario genere, per esempio tramite Internet.

Tale possibilità è spesso necessaria: per esempio, l'applicazione dovrà poter memorizzare in maniera *persistente* i dati elaborati in una sessione di lavoro e, al successivo avvio, poter riprendere il lavoro dal punto in cui era stato lasciato. Altre casistiche potrebbero essere la necessità di dovere o potere inviare i dati elaborati a un'altra applicazione o a un altro utente, per esempio salvandoli su un supporto di memorizzazione di massa, come un CD o una Pen Drive USB, o direttamente mediante l'invio e il download attraverso Internet.

La scelta del tipo di memoria persistente dipende da vari fattori: dalla tipologia di dati, dalla loro quantità, dalla distribuzione che dovrà essere fatta di essi, da quanti e quali altri utenti o applicazioni dovranno poter accedere a essi.

Nel capitolo dedicato a XML abbiamo già avuto modo di eseguire operazioni per leggere e scrivere dati in questo formato, che di per sé costituisce un'ottima modalità di gestione dei dati. In questo capitolo mostreremo nel dettaglio quali sono le modalità e le principali classi

del .NET Framework che permettono di eseguire tutte quelle operazioni che possono essere raggruppate sotto l'unico termine di *I/O*, cioè *input/output*.

L'input/output consiste nel leggere e scrivere i vari tipi di dati (che possono essere per esempio dei classici file, che usano uno dei formati standard oppure un proprio formato, memorizzabili su un file system), oppure nell'accesso a dei veri database di differente tecnologia o architettura, o ancora nello sfruttare dei servizi remoti tramite una rete, per ricavare dati o per inviarli a dei destinatari.

Accedere al file system

Probabilmente, quando si parla di input e di output, il primo mezzo di memorizzazione dei dati che viene in mente è il file system, e quindi i file e le directory, che qualunque semplice utente di un computer conosce e sa utilizzare.

Il .NET Framework fornisce i tipi necessari e le API per interagire con il file system, e quindi per creare, leggere, scrivere, eliminare file e directory. La maggior parte di tali classi è contenuta nel namespace System.IO.

Informazioni sui Drive

Gli elementi fondamentali di un file system sono file e directory. Il namespace System.IO contiene le classi per eseguire operazioni su di essi e per ricavarne la struttura. Ogni file e directory è però di certo contenuto in un disco, che può essere un disco rigido, un supporto ottico, una Pen Drive e così via.

La classe DriveInfo consente di accedere alle informazioni su tutte le unità disco presenti su un computer. Il primo passo è elencarle per mezzo del metodo statico GetDrives, che restituisce un array di oggetti DriveInfo. La classe DriveInfo espone poi proprietà informative di vario genere, come il nome o l'etichetta dell'unità in esame, lo spazio totale o quello disponibile su di essa.

Il seguente esempio ottiene e stampa informazioni su ogni drive installato sul sistema che abbia la proprietà IsReady uguale a true. IsReady indica che un drive è pronto per la lettura. Per esempio, sarà false per un'unità CD-Rom che non contiene alcun disco.

```
DriveInfo[] drives = DriveInfo.GetDrives();
foreach (DriveInfo driveInfo in drives)
{
Console.WriteLine("Drive {0}", driveInfo.Name);
Console.WriteLine(" File type: {0}", driveInfo.DriveType);
if (driveInfo.IsReady == true)
{
Console.WriteLine("  Volume label: {0}", driveInfo.VolumeLabel);
Console.WriteLine("  File system: {0}", driveInfo.DriveFormat);
Console.WriteLine(" Spazio disponibile: {0,10} KBytes", (driveInfo.AvailableFreeSpace>>10));
Console.WriteLine("  Spazio totale:   {0,10} KBytes", driveInfo.TotalFreeSpace>>10);
Console.WriteLine("  Total size of drive: {0,10} KBytes", driveInfo.TotalSize>>10);
}
}
```

Nell'esempio vengono stampati il nome dell'unità, che è la lettera assegnata allo stesso, come C:\, D:\ e così via, e il tipo di unità, che è uno dei possibili membri dell'enumerazione DriveType.

Il tipo può essere quindi uno fra: CDRom, Fixed, Unknown, Network, NoRootDirectory, Ram, Removable.

Le proprietà sullo spazio totale e quello disponibile restituiscono un valore espresso in byte (che nell'esempio è convertito in kB, mediante un'operazione di shift a destra).

Lo spazio totale o dimensione del disco viene dato dalla proprietà TotalSize.

Si noti poi che vi sono due proprietà simili per lo spazio libero disponibile: TotalFreeSpace restituisce lo spazio libero in totale sull'unità, mentre AvailableFreeSpace, anche se spesso coincide con la precedente, tiene conto della possibile quota disco assegnata all'utente corrente.

Lavorare con i file e le cartelle

Per ottenere informazioni sulla struttura logica del file system, cioè su file e directory in esso contenuti, ed eseguire operazioni di vario genere su di essi, il namespace System.IO contiene due diverse coppie di classi da utilizzare a seconda dell'obiettivo perseguito.

Le prime due classi, File e Directory, sono statiche e forniscono quindi dei metodi anch'essi statici, senza dover istanziare alcun oggetto.

Le altre due invece si chiamano FileInfo e DirectoryInfo e devono essere istanziate indicando delle stringhe che rappresentino il percorso del file o della directory.

Con entrambe le coppie di classi possono essere ottenuti quasi gli stessi risultati, o eseguite le stesse operazioni. Quindi, in genere, si sceglie la versione statica, cioè le classi File e Directory, quando si devono eseguire operazioni singole o comunque ottenere valori più semplici, specificando per ogni chiamata il percorso di file e directory su cui agire. Mentre, se si devono o vogliono eseguire più operazioni su una stessa entità del file system, conviene istanziare un relativo oggetto FileInfo o DirectoryInfo, e quindi leggere le varie proprietà o invocare vari metodi sull'istanza ottenuta.

Manipolazione di nomi e percorsi

Prima di vedere più nel dettaglio le operazioni possibili su file e directory, introduciamo la classe Path.

La classe statica Path esegue operazioni su delle stringhe, che rappresentano percorsi relativi o assoluti di oggetti o nomi di file. Fra i metodi più usati e più utili vi è il metodo Combine, che consente di ottenere un percorso completo combinando, per esempio, un percorso che rappresenta una directory con un nome di file, oppure due diverse directory, senza preoccuparsi di inserire manualmente separatori o trattare con i caratteri di escape, e di usare

i metodi della classe String o i suoi operatori. Per esempio, supponiamo di avere le seguenti due stringhe:

```
string dirname=@"C:\temp";
string filename="file.txt";
```

Si noti che, nel primo caso, si è utilizzato il carattere *verbatim* @ per evitare di dover raddoppiare il backslash nel percorso della directory. In caso contrario, infatti, un singolo \ avrebbe agito insieme alla t seguente, formando una sequenza di escape \t, che indica un carattere di tabulazione.

Per ottenere il percorso completo del file, basterà ora utilizzare il metodo Combine e scrivere:

```
string fullpath= Path.Combine(dirname, filename); //C:\\temp\\file.txt"
```

La classe Path offre una serie di altri metodi di notevole utilità, che vi consiglio di provare ed esaminare, per evitare di scrivere funzioni già implementate.

Ecco una serie di esempi che lavorano sui precedenti nomi di file e directory e su un percorso completo; nei commenti, i risultati fanno capire lo scopo e il loro funzionamento:

```
string xml=Path.ChangeExtension(filename, "xml");// file.xml
string dir=Path.GetDirectoryName(fullpath); // c:\temp
string ext = Path.GetExtension(fullpath); // .txt
string file=Path.GetFileName(fullpath); // file.txt
string filewithoutext= Path.GetFileNameWithoutExtension(fullpath); // file

Directory.SetCurrentDirectory("c:\\windows"); // imposta la directory di lavoro corrente
string full=Path.GetFullPath(filename); // c:\windows\file.txt

string root=Path.GetPathRoot(fullpath); // C:
bool hasExt= Path.HasExtension("file.txt"); // true
bool pathRooted = Path.IsPathRooted("c:\\file.txt"); // true
```

Spesso capita di dover creare dei file temporanei, con un nome casuale, e di doverli memorizzare in una cartella anch'essa temporanea. La classe Path fornisce anche i metodi per ottenere tali risultati, basandoli sulle cartelle all'interno del profilo utente:

```
string randomFile = Path.GetRandomFileName(); // v3ybhjqf.0xd
string tempFile = Path.GetTempFileName(); // C:\Users\UserName\AppData\Local\Temp\tmp210E.tmp
string tempPath = Path.GetTempPath(); // C:\Users\UserName\AppData\Local\Temp
```

Oltre alla directory temporanea può anche capitare di dover lavorare sulle altre directory speciali del sistema operativo, per esempio quelle contenenti i documenti, il desktop e così via. In tal caso, un'utile compagna della classe Path è la classe Environment, che permette di ottenere un percorso di cartella speciale, specificandolo mediante uno dei valori dell'enumerazione SpecialFolder, da passare al metodo GetFolderPath:

```
string desktopPath=Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
```

NOTA	La classe Environment fornisce altre proprietà e metodi per ottenere informazioni e manipolare l'ambiente e la piattaforma corrente. Per esempio, è possibile ottenere il nome del computer, la versione del sistema operativo, il nome dell'utente loggato nel sistema e altre amenità di questo genere.

File *e* Directory

Torniamo ora alle classi per la manipolazione di file e directory, facendo una panoramica delle funzionalità che esse mettono a disposizione.

Come già detto, File e FileInfo, e poi Directory e DirectoryInfo, permettono di eseguire in maniera leggermente differente le stesse operazioni.

Il seguente esempio utilizza le classi Directory e DirectoryInfo per eliminare una cartella, specificandone il percorso completo, dopo averne verificato l'esistenza:

```
DirectoryInfo tempDir = new DirectoryInfo("c:\\temp\\tempdir");
if(tempDir.Exists)
tempDir.Delete(true);
```

```
if (Directory.Exists("c:\\temp\\tempdir"))
Directory.Delete("c:\\temp\\tempdir", true)
```

Il parametro bool del metodo Delete indica se eseguire l'eliminazione in maniera ricorsiva di eventuali file e sottocartelle in esse contenuti.

Le classi DirectoryInfo e FileInfo derivano da una classe astratta comune, FileSystem-Info, che contiene proprietà comuni per ricavare varie caratteristiche, come data e ora di creazione o dell'ultimo accesso, attributi di file o directory, metodi come l'appena utilizzato Delete per l'eliminazione di un elemento.

Intanto, come già visto sopra, è possibile verificare l'esistenza di un file o di una directory con Exists:

```
bool exists=File.Exists(fullpath);
FileInfo fi1=new FileInfo(fullpath);
exists=fi1.Exists;
```

Analogamente, per le classi Directory e DirectoryInfo possono essere usati rispettivamente il metodo e la proprietà omonimi.

In generale, le classi FileInfo e DirectoryInfo espongono le stesse funzionalità di File e Directory, con l'aggiunta di proprietà specifiche. Per esempio, una volta costruito un FileInfo è possibile ricavarne la DirectoryInfo che lo contiene:

```
FileInfo fi1 = new FileInfo(file1);
Console.WriteLine(fi1.Directory.FullName);
```

Se una directory non esiste, è possibile crearla per mezzo del metodo Create, presente sia in

Directory sia in DirectoryInfo:

```
DirectoryInfo dir=Directory.CreateDirectory(source);
```

I file e le directory sono dotati di attributi che permettono di specificare o stabilire delle particolari caratteristiche di tali elementi. Per esempio, un file può essere contrassegnato come di sola lettura, nascosto, di sistema e così via. L'enumerazione FileAttributes fornisce tali attributi ed, essendo essa di tipo Flags, i suoi membri possono essere combinati bit a bit per impostarne diversi.

Per leggere e scrivere gli attributi, è possibile utilizzare i metodi GetAttributes e Set-Attributes di File o la proprietà Attribute di FileInfo:

```
string filePath = @"c:\test.txt";
FileAttributes fileAttributes = File.GetAttributes(filePath);
```

Supponiamo di volere impostare l'attributo di sola lettura per un determinato file:

```
FileInfo fi=new FileInfo(path);
fi.Attributes = FileAttributes.Hidden | FileAttributes.ReadOnly;
```

Oppure, con il metodo statico di File, si può scrivere:

```
File.SetAttributes(file1, FileAttributes.Hidden | FileAttributes.ReadOnly);
```

Se si è interessati a verificare se un file o una directory hanno un preciso attributo, per esempio se esso è di sola lettura, bisogna utilizzare l'operatore & nel seguente modo:

```
bool isReadOnly = (fi1.Attributes & FileAttributes.ReadOnly) == FileAttributes.ReadOnly;
```

Se invece si vuole togliere un attributo, è necessario combinare gli attributi attuali (dopo averli ottenuti) mediante l'operatore & e utilizzare l'operatore ~ per ottenere il complemento dei bit specificati:

```
File.SetAttributes(filePath, File.GetAttributes(filePath) & ~FileAttributes.Hidden);
```

L'alternativa è impostare la proprietà Attributes, con l'operatore di OR esclusivo:

```
fi1.Attributes ^= FileAttributes.ReadOnly; //se l'attributo ReadOnly è presente, lo elimina
```

Passiamo adesso alle classiche operazioni di copia e spostamento di file e directory. I metodi sono abbastanza immediati da capire e utilizzare.

Per copiare un file è possibile utilizzare il metodo Copy di File o **CopyTo** di FileInfo:

```
File.Copy(@"c:\temp\file1.txt", @"C:\temp\file2.txt");

FileInfo fi = new FileInfo(@"c:\temp\file1.txt");
fi.CopyTo(@"C:\temp\file2.txt");
```

Se il file di destinazione già esiste, viene generata un'eccezione. In questo caso, è possibile passare ai metodi di copia un secondo parametro booleano, il quale indica se sovrascrivere il

file nel caso in cui sia esistente.

I metodi di copia mantengono naturalmente il file sorgente. Se invece si vuole spostare il file in un'altra cartella e quindi eliminarlo da quella di origine, basta utilizzare il metodo Move o MoveTo:

```
File.Move(@"c:\temp\file2.txt", @"C:\temp\file3.txt");

fi = new FileInfo(@"c:\temp\file3.txt");
fi.MoveTo(@"C:\temp\file4.txt");
```

Data una directory, fra le operazioni più interessanti e frequenti da eseguire vi sarà certamente l'enumerazione di file e sottocartelle presenti al suo interno. Tali funzionalità sono esposte da diversi metodi sia della classe Directory sia di DirectoryInfo.

I metodi Getfiles e GetDirectories di Directory ricavano un array di stringhe che rappresentano i nomi di file e sottocartelle presenti in un certo percorso:

```
string[] fileList= Directory.GetFiles("C:\\temp", "*.txt", SearchOption.AllDirectories);
string[] directoryList= Directory.GetDirectories("C:\\temp");
```

Si noti l'uso del parametro SearchOption che permette di stabilire se la ricerca debba essere effettuata in modo ricorsivo.

A partire da un oggetto DirectoryInfo, invece, si possono ricavare e filtrare sia file sia directory con i corrispondenti metodi di istanza. Alcuni metodi, presenti anche nelle classi statiche, che restituiscono un'enumerazione di oggetti, come EnumerateFileSystemInfos, EnumerateFiles, EnumerateDirectories, sono mostrati negli esempi seguenti:

```
DirectoryInfo diTemp = new DirectoryInfo("C:\\temp\\");
var entries= diTemp.EnumerateFileSystemInfos();
foreach (FileSystemInfo fsi in entries)
{
Console.WriteLine(fsi.FullName
}

var files = diTemp.EnumerateFiles();
foreach (FileInfo finfo in files)
{
Console.WriteLine("{0, -30} - {1}KB", finfo.Name, finfo.Length>>10);
}
```

Avendo a che fare in questo caso con array ed enumerazioni di file e directory, torna di grande utilità l'uso di query LINQ per filtrare tali collezioni in maniera molto più semplice e immediata.

Il seguente esempio mostra una query per stampare nome e dimensione dei file in C:\temp più grossi di 1 MB (dimensione in byte ricavata shiftando a sinistra di 20 posizioni il valore 1) e con estensione diversa da "txt":

```
var elenco= new DirectoryInfo("C:\\temp\\").EnumerateFiles("*", SearchOption.AllDirectories);
var query = from f in elenco
where f.Length > (1 << 20) && f.Extension != "txt"
select new { f.FullName, f.Length };

query.ToList().ForEach(Console.WriteLine);
```

Naturalmente, le cose più interessanti da fare con i file riguardano la lettura e la scrittura di dati. La classe File possiede dei metodi per impostare e leggere il contenuto di un file, ma nella maggior parte dei casi bisognerà avere a che fare con degli oggetti stream e con i relativi oggetti che permettono di trattarli in lettura e scrittura.

Tali argomenti saranno trattati nel prossimo paragrafo. Per il momento mostriamo qualcuno dei metodi rimanenti, che consentono un accesso al contenuto del file in maniera più immediata, ma anche limitata.

Per scrivere delle stringhe di testo sono disponibili i metodi WriteAllText e WriteAllLines:

```
string filepath = @"c:\temp\filetesto.txt";
StringBuilder sb=new StringBuilder();
sb.AppendLine("riga 1");
sb.AppendLine("riga 2");
sb.AppendLine("riga 3");
File.WriteAllText(filepath, sb.ToString());

string[] righe=new string[]{"riga 1", "riga 2", "riga 3"};
File.WriteAllLines(filepath, righe);
```

Entrambi i metodi azzerano il contenuto eventualmente esistente e lo impostano con i valori passati come parametri. Se si vuol aggiungere del testo, appendendolo a quello già contenuto, si può sfruttare il metodo AppendAllLines:

```
File.AppendAllLines(filepath, righe);
```

Il metodo WriteAllBytes, invece, permette di generare del contenuto binario a partire da un array di byte:

```
File.WriteAllBytes(filepath, Encoding.ASCII.GetBytes("contenuto"));
```

Al contrario, i metodi ReadLines, ReadAllLines, ReadAllText e ReadAllBytes permettono di leggere il contenuto di un file esistente:

```
var righeLette = File.ReadLines(filepath);
righeLette.ToList().ForEach(Console.WriteLine);
```

Monitoraggio del file system

La classe FileSystemWatcher permette di monitorare una cartella del file system e di essere avvisati di qualunque attività su di essa, per esempio la creazione, l'eliminazione o la modifica di un file al suo interno.

A cosa può servire? Pensate per esempio alla realizzazione di un'applicazione di backup automatico dei file: basta copiarli in una determinata cartella e, quando il FileSystemWatcher si accorge di una modifica, si può procedere alla copia del contenuto su un disco esterno, o magari inviarli su un computer remoto, come fanno tanti servizi tipo SkyDrive o Dropbox.

Il seguente codice mostra tutti i passi per creare un piccolo programma completo che esegua la copia dei file di una directory source verso una target al verificarsi di determinati eventi.

```
static string source = @"C:\temp\source";
static string target = @"C:\temp\target";

static void Main(string[] args
{
if(!Directory.Exists(source))
Directory.CreateDirectory(source);
if(!Directory.Exists(target))
Directory.CreateDirectory(target);
...
```

Per creare un FileSystemWatcher, basta indicare il percorso da monitorare ed eventualmente un filtro sui tipi di file che interessano.

La proprietà IncludeSubDirectories permette di estendere il monitoraggio alle sottocartelle:

```
FileSystemWatcher watcher = new FileSystemWatcher(source, "*.txt");
watcher.IncludeSubdirectories = true;
```

La proprietà NotifyFilter permette di specificare il tipo di eventi di cui l'oggetto watcher deve essere notificato. Come comportamento predefinito, essa monitora il cambiamento del nome di file e directory e l'istante di ultima scrittura. Nel seguente caso viene considerato anche il cambiamento della dimensione del file.

```
watcher.NotifyFilter = NotifyFilters.FileName | NotifyFilters.DirectoryName | NotifyFilters.LastWrite | NotifyFilters.Size;
```

Per gestire gli eventi, basta selezionare quelli cui si è interessati e scrivere il relativo gestore. Per esempio, per copiare un file quando esso viene creato, modificato o rinominato nella directory sorgente, scriveremo:

```
watcher.Created += watcher_Event;
watcher.Changed += watcher_Event;
watcher.Renamed += watcher_Event;
```

A questo punto, per iniziare il monitoraggio sarà sufficiente impostare la proprietà EnableRaisingEvents:

```
//inizia il monitoraggio
watcher.EnableRaisingEvents = true;
```

Nel gestore degli eventi impostati in precedenza, si ricava il file cambiato e lo si copia nella directory destinazione:

```
static void watcher_Event(object sender, FileSystemEventArgs e)
{
if (File.Exists(e.FullPath))
{
Console.WriteLine("copia di {0}", e.Name);
FileInfo fs = new FileInfo(e.FullPath);
fs.CopyTo(Path.Combine(target, Path.GetFileName(e.FullPath)), true);
}
}
```

La classe **Stream**

Ogni operazione di input e output in .NET coinvolge l'utilizzo dei cosiddetti *stream*. Uno stream può essere pensato come un'astrazione che rappresenta un flusso di dati, da leggere o da scrivere in maniera sequenziale. Il flusso potrà poi fisicamente corrispondere a un file, a una locazione di memoria, a una connessione di rete e così via.

La classe standard per rappresentare questo flusso di dati è la classe Stream, che permette di trattare genericamente una qualunque mole di dati in modo sequenziale, byte dopo byte, e quindi in maniera ottimizzata, perché non vi è la necessità di dover occupare una consistente quantità di memoria caricando, per esempio, un intero file che potrebbe anche essere di grosse dimensioni.

La classe Stream è astratta perché l'implementazione concreta, fornita da classi da essa derivate, dipende dall'entità reale cui essa è collegata: per esempio, FileStream rappresenta lo stream per un file su disco, MemoryStream un flusso mantenuto in memoria, o ancora NetworkStream consente di utilizzare un flusso di dati per l'accesso alla rete. In tal modo, lo sviluppatore non deve preoccuparsi degli specifici dettagli del sistema operativo o dei dispositivi fisici sottostanti.

La classe Stream espone i metodi e le proprietà per le tre operazioni fondamentali di lettura, scrittura e ricerca dei dati, oltre a quelli complementari per l'apertura e la chiusura del flusso.

La *lettura* da uno stream restituisce una sequenza di byte che dovranno poi essere trasformati in qualcosa di più intelligibile, per esempio stringhe di testo o oggetti.

La *scrittura* richiede il processo inverso, quindi la trasformazione di oggetto, testi, o altro in sequenze di byte, da inviare poi attraverso il flusso.

In alcuni casi, si può eseguire il posizionamento e la ricerca lungo uno stream per mezzo di un'operazione cosiddetta di *seeking*. Per esempio, tale possibilità è consentita per un file, nel quale ci si può posizionare al suo inizio, alla fine o in un punto qualsiasi. Mentre non ha senso per uno stream legato a una connessione di rete.

A seconda dell'archivio o sorgente di dati sottostante, uno stream potrà supportare solo alcune di queste operazioni. Le proprietà CanRead, CanWrite e CanSeek della classe Stream specificano quali fra esse sono consentite. Di conseguenza, la classe Stream espone anche i due metodi corrispondenti per eseguire le rispettive operazioni di lettura e scrittura, Read e Write, e una proprietà Position per ricavare e impostare la posizione della prossima lettura o scrittura di byte:

```
public abstract int Read(byte[] buffer, int offset, int count);
public abstract void Write(byte[] buffer, int offset, int count);
public abstract long Position { get; set; }
```

Da quanto detto finora, deriva che i dati possono essere trasferiti in maniera bidirezionale lungo uno stream.

È bene notare la terminologia usata. Si dice “leggere da uno stream” se l’operazione eseguita è quella di trasferire dati da una sorgente esterna verso il programma: per esempio, leggere un file, ricevere dati tramite una connessione di rete.

Al contrario, si dice “scrivere su uno stream”, e in tal caso il trasferimento avviene dal programma che usa il flusso verso l’esterno: per esempio, per scrivere su un file e inviare dati in rete.

Per poter creare uno stream, essendo la classe Stream astratta, necessitiamo di una classe come MemoryStream, su cui torneremo più avanti. Per il momento la utilizzeremo per ottenere un’istanza Stream su cui provare i metodi e le proprietà della classe madre:

```
Stream s = new FileStream("c:\\temp\\file.txt", FileMode.OpenOrCreate);
```

Intanto, c’è da notare che in generale uno stream è da chiudere al termine del suo utilizzo, per liberare così le risorse utilizzate, che possono essere l’handle di un file o di un socket di rete. Ciò è fattibile invocando esplicitamente il metodo Close:

```
s.Close();
```

Ma, poiché la classe Stream implementa l’interfaccia IDisposable, il metodo più indicato e corretto è utilizzare il pattern *dispose*, con l’uso della parola chiave using:

```
using(Stream s = new FileStream("c:\\temp\\file.txt", FileMode.OpenOrCreate))
{
//uso dello stream
}
```

Per scrivere dei byte su uno stream, si utilizza il metodo Write, passando come argomento un array di byte detto *buffer*, l’offset eventuale all’interno di questo buffer, e il numero di byte a partire dall’offset. Per esempio:

```
byte[] bytes=new byte[] {1,2,3,4};
using (Stream stream = new FileStream("c:\\temp\\file.txt", FileMode.OpenOrCreate))
{
stream.Write(bytes, 0, bytes.Length);
}
```

In questo caso, si è scritto sullo stream l’intero buffer composto da 4 byte. Il metodo WriteByte, invece, permette di scrivere un singolo byte:

```
s.WriteByte(123);
```

Analogamente, per leggere dati da uno stream è necessario indicare il buffer di destinazione, la posizione di partenza e il numero di byte da leggere. Nel seguente esempio, usiamo un MemoryStream riempito con 1000 byte casuali:

```
byte[] buffer = new byte[1000];
Random rnd=new Random();
rnd.NextBytes(buffer);
```

Il metodo di lettura Read restituisce il numero di byte letti. Infatti, anche se si specifica il numero di byte desiderati, si arriverà prima o poi alla fine del buffer originale e quindi, nel caso in cui i byte rimanenti siano in un numero minore di quello specificato, esso restituirà il numero 0:

```
byte[] dest=new byte[50000];
using(MemoryStream ms = new MemoryStream(buffer))
{
int read=0;
int total = 0;
while( (read = ms.Read(dest, 0, 100)) >0)
{
total += read;
}
Console.WriteLine(total);
}
```

Il metodo precedente continua a leggere dall'array buffer, 100 byte alla volta, finché il metodo Read restituisce un numero positivo.

Così come per la scrittura, è anche possibile cercare di leggere un byte alla volta:

```
int b = ms.ReadByte();
```

Se lo stream è giunto alla fine, il metodo ReadByte restituirà il valore -1.

Per i tipi di stream che consentono di impostare la posizione di lettura (e quindi la proprietà CanSeek restituisce true), possiamo utilizzare il metodo Seek per spostare la posizione e la proprietà Position per leggere quella attuale.

```
if (ms.CanSeek)
{
long pos = ms.Position;
ms.Seek(100, SeekOrigin.Begin); //posiziona a 100 dopo l'inizio
pos= ms.Position;
int b = ms.ReadByte();
}
```

La proprietà Position è anche utilizzabile per impostare direttamente la posizione.

Tipi di stream

Per lavorare con gli stream, il framework .NET mette a disposizione una serie di tipi che permettono di standardizzare le operazioni da eseguire su di essi.

Innanzitutto, esiste una prima categoria di tipi che, derivati dalla classe astratta Stream, sono delle classi che implementano specifiche operazioni di input/output e che rappresentano un particolare tipo di archivio.

Le principali e più comuni fra le classi derivate direttamente o indirettamente da Stream sono elencate e descritte di seguito:

- FileStream – per leggere e scrivere un file;
- IsolatedStorageStream – per leggere e scrivere un file all'interno dell'Isolated Storage (è derivata da FileStream);
- MemoryStream – per leggere e scrivere dati dalla memoria;
- NetworkStream – per leggere e scrivere dati attraverso connessioni di rete;
- PipeStream – per leggere e scrivere dati lungo un canale di comunicazione fra due processi.

Altre classi effettuano una sorta di trasformazione dei dati da utilizzare con i tipi precedenti, aggiungendo a essi una serie di diverse funzionalità:

- BufferedStream – ottimizza le operazioni di lettura e scrittura;
- CryptoStream – permette di trattare flussi dati crittografati;
- GZipStream – per compressione e decompressione dei flussi di dati;
- DeflateStream – per compressione e decompressione dei flussi mediante algoritmo Deflate.

Queste ultime sono eventualmente combinabili per fornire più funzionalità a uno stesso stream, in maniera da consentirgli di eseguire operazioni più complesse. Questo principio è chiamato *decorator pattern*. Per esempio, uno stream di dati potrebbe essere compresso usando un GZipStream, prima di essere crittografato mediante un CryptoStream, e poi spedito su una connessione di rete.

La classe FileStream

La classe FileStream restituisce un oggetto per la lettura e la scrittura di uno stream di dati associato a un file. L'abbiamo già usata per mostrare le funzioni di lettura e scrittura di un generico oggetto Stream, utilizzando il modo diretto di istanziarla, cioè passando un percorso di file al costruttore.

I diversi costruttori della classe consentono di specificare anche altri parametri, per indicare le modalità di apertura del file, i permessi di accesso e così via:

```
using (FileStream fs = new FileStream(path, FileMode.OpenOrCreate, FileAccess.ReadWrite))
{
fs.WriteByte(1);
}
```

L'enumerazione FileMode specifica come comportarsi in relazione al fatto che il percorso specificato potrebbe riferirsi a un file già esistente oppure a uno nuovo. Per esempio, se si utilizza il valore Open, verrà creato uno stream solo se il file esiste già, mentre verrebbe generata un'eccezione in caso contrario.

Utilizzando come nel caso precedente OpenOrCreate, qualora il file non esistesse, esso verrebbe creato.

L'enumerazione FileAccess, invece, permette di specificare cosa si intende fare con lo stream. Quindi si potrà creare uno stream di sola lettura, sola scrittura o in lettura e scrittura. Pertanto, se si aprisse uno stream specificando come FileAccess il valore Read, il seguente esempio genererebbe un'eccezione:

```
using (FileStream fsRead = new FileStream(path, FileMode.OpenOrCreate, FileAccess.Read))
{
try
{
fsRead.WriteByte(1);
}
catch(Exception ex)
{
Console.WriteLine(ex.Message);
}
}
```

Combinando i valori delle due enumerazioni si potrà creare e accedere in qualunque modalità a un file.

Quando si utilizzano le classi File e FileInfo viste nei paragrafi precedenti, si ha spesso a che fare con un oggetto FileStream. Infatti, alcuni dei metodi delle classi suddette permettono di ottenere un FileStream in maniera molto semplice ed immediata. Il metodo Open, in generale, restituisce un'istanza della classe, ma richiede di specificare come prima i parametri FileMode e FileAccess:

```
FileStream fs = File.Open(path, FileMode.OpenOrCreate, FileAccess.ReadWrite)
```

La classe File espone poi altri due metodi, OpenRead e OpenWrite, che evitano di impostare esplicitamente tali permessi. Il primo sarà utilizzato per aprire un FileStream in sola lettura, mentre il secondo ne permetterà anche la scrittura.

```
FileStream fs = File.OpenRead(path);
```

NOTA	FileStream non è disponibile per le applicazioni da eseguire su Windows RT/8.x: in questo ambiente è necessario utilizzare i tipi del namespace Windows.Storage.

La classe MemoryStream

La classe MemoryStream rappresenta uno stream che utilizza un array in memoria come archivio di memorizzazione dei dati. Essa è utile quando è necessario ricavare dei dati da manipolare in memoria, per esempio per leggere i byte di un'immagine da un file locale scaricato da Internet e applicare a essi dei filtri, oppure una compressione o un algoritmo di crittografia.

Avere i byte in memoria, anziché doverli leggere da uno stream potenzialmente più lento, migliora le perfomance; inoltre, essendo un tipo di stream che supporta la funzione di seek, è possibile spostarsi agevolmente fra i byte stessi.

Il seguente esempio apre un FileStream, legge i byte e li scrive su un MemoryStream.

```
using (MemoryStream ms = new MemoryStream())
{
using (FileStream file = new FileStream("c:\\temp\\test.jpg", FileMode.OpenOrCreate, FileAccess.ReadWrite))
{
byte[] filebuffer = new byte[file.Length];
file.Read(filebuffer, 0, (int)file.Length);
ms.Write(filebuffer, 0, (int)file.Length);
}
}
```

Operazioni asincrone

Le operazioni sugli stream, soprattutto su quelli potenzialmente lenti come i socket di rete, possono essere ottimizzate in maniera da rendere l'interfaccia grafica e le intere applicazioni più reattive e usabili.

A partire da .NET 4.5 sono presenti, infatti, anche le versioni asincrone dei metodi di lettura e scrittura di uno stream. Utilizzare questa modalità è solo questione di cambiare i metodi Read e Write con i corrispondenti asincroni ReadAsync e WriteAsync, naturalmente aggiungendo un await alla chiamata:

```
string path="c:\\temp\\asyncfile.txt";
using (FileStream fs = File.Open(path, FileMode.OpenOrCreate, FileAccess.ReadWrite))
{
await fs.WriteAsync(buffer, 0, 1000);

fs.Seek(0, SeekOrigin.Begin);
byte[] readbuffer = new byte[100];
int readBytes= await fs.ReadAsync(readbuffer, 0, 100);
}
```

Lettori e scrittori

Gli stream sono progettati per trattare direttamente con dati a livello di byte. Per consentire una più semplice manipolazione di tali dati, e per trattarli, per esempio, come stringhe, numeri

o, a livello ancora più alto, come elementi XML, il framework .NET fornisce delle classi che si occupano della loro conversione in un verso e nell'altro.

Queste classi sono implementate a coppie, in maniera da fornire per lo stesso meccanismo di conversione sia l'operazione di lettura sia quella di scrittura. Quindi utilizzano una nomenclatura che prevede, rispettivamente, il suffisso Reader e Writer.

Ecco un elenco delle principali fra queste classi:

- TextReader e TextWriter – sono classi astratte usate come base per altri tipi lettori e scrittori, e dedicate a dati di tipo testuale;
- StringReader e StringWriter – lettore e scrittore di caratteri da e verso oggetti di tipo string;
- StreamReader e StreamWriter – lettore e scrittore per convertire caratteri da e verso byte, utilizzando una data codifica;
- BinaryReader e BinaryWriter – utilizzate per leggere e scrivere dati primitivi in formato binario.

Nel capitolo dedicato a XML abbiamo anche visto e utilizzato le classi XmlReader e XmlWriter, che permettono l'accesso sequenziale agli elementi di dati in formato XML.

La classe StreamReader e StreamWriter

Teoricamente è possibile utilizzare FileStream per leggere e scrivere file o altri stream in formato testuale. Le classi StreamReader e StreamWriter, però, facilitano notevolmente il compito, lavorando a un livello leggermente più alto e focalizzando il loro compito sulla lettura e sulla scrittura di questa tipologia di stream.

Per esempio, uno StreamReader può leggere un file di testo in qualsiasi codifica, senza preoccuparsi di interpretare i byte in modo diverso a seconda del formato utilizzato dal file. Al contrario, StreamWriter consente di scrivere del testo su un file, utilizzando una qualsiasi codifica desiderata.

Per costruire un oggetto StreamReader, esiste una serie abbastanza vasta di possibilità; la più immediata è quella di utilizzare uno dei suoi costruttori. A essi è possibile passare come argomenti direttamente un percorso del file, oppure un oggetto FileStream costruito in precedenza, e una serie di altri parametri fra i quali il più utile, ove necessario, permette di specificare l'encoding da utilizzare.

Per esempio, per leggere un file di testo in codifica ASCII si può scrivere:

```
StreamReader reader = new StreamReader(path, Encoding.ASCII);
```

La classe astratta Encoding contiene una serie di proprietà che restituiscono un'istanza della codifica corrispondente; oltre a quella ASCII, per esempio, è possibile utilizzare Unicode,

UTF8, UTF7 e così via.

Come visto già per i FileStream, anche un oggetto StreamReader deve essere chiuso dopo il suo utilizzo, per evitare che il file rimanga bloccato:

```
reader.Close();
```

Quindi, anche in questo caso, è conveniente e consigliabile utilizzare un blocco using.

Altri modi di creare uno StreamReader sono collegati alle classi File e FileInfo che espongono entrambe il metodo OpenText:

```
using(StreamReader sr1= File.OpenText(path))
{}
```

Una volta ottenuto un oggetto StreamReader, leggere il contenuto di un file è semplicissimo. In particolare è a disposizione il metodo ReadLine, che tenta di leggere la linea di testo successiva o restituisce null quando si è raggiunta la fine del file.

Il seguente esempio usa un ciclo while per leggere tutte le righe e stamparle sulla console:

```
using(StreamReader sr1= File.OpenText(path))
{
string line;
while( (line=sr1.ReadLine())!=null)
{
Console.WriteLine(line);
}
}
```

Se invece si vuol ottenere in un solo colpo l'intero contenuto e memorizzarlo in una stringa, basta utilizzare il metodo ReadToEnd:

```
using(StreamReader sr= File.OpenText(path))
{
string str=sr.ReadToEnd();
}
```

Oppure, al contrario, se si vuol leggere un carattere alla volta, che verrà restituito sotto forma di int, eventualmente convertibile in char:

```
using (StreamReader sr2 = File.OpenText(path))
{
int ch;
while ((ch = sr2.Read()) != -1)
{
Console.WriteLine((char)ch);
}
}
```

ll valore -1 indicherà la fine del file.

Un altro overload di Read permette anche di leggere più caratteri in un array di char:

```
char[] array= new char[10];
sr.Read(array, 0 , 10);
```

La classe StreamWriter lavora in maniera analoga, permettendo di scrivere su un file di testo o su un altro stream qualsiasi tipo primitivo.

La costruzione di uno StreamWriter è analoga a quanto visto per StreamReader; si può utilizzare un costruttore, specificando eventualmente l'encoding che si vuole impiegare:

```
StreamWriter writer=new StreamWriter(path, true, Encoding.ASCII)
```

Se non si specifica nessun Encoding, verrà utilizzato come predefinito UTF8, che permette di gestire tutti i caratteri Unicode. Il parametro bool indica, in questo caso, che il file verrà aperto in modo da appendere al contenuto già esistente le nuove scritture.

Le altre possibilità permettono di passare un oggetto Stream al costruttore, oppure di partire anche qui da istanze delle classi File e FileInfo e usare i metodi CreateText o AppendText:

```
FileInfo fi = new FileInfo(path);
using(StreamWriter writer=fi.AppendText())
{}
```

Per scrivere su uno stream, possono essere utilizzati decine di overload dei metodi Write e WriteLine, che permettono di utilizzare parametri dei vari tipi primitivi:

```
using(StreamWriter writer=File.CreateText(path))
{
writer.WriteLine(1);
writer.WriteLine(true);
writer.WriteLine("hello");
}
```

Anche per le classi StreamReader e StreamWriter sono disponibili le versioni asincrone dei metodi di lettura e scrittura: ReadAsync, WriteLineAsync e WriteAsync.

La classe StringReader e StringWriter

Le classi StringReader e StringWriter sono utilizzabili quando, come sorgente dati sottostante, si ha già una stringa. Il loro scopo e la loro utilità possono sembrare leggermente oscure, dato che con una stringa si hanno già a disposizione una serie di metodi forniti dalla classe String.

In realtà, tali classi vengono usate quando si ha necessità di leggere parti di una stringa, per esempio riga per riga, oppure quando bisogna passare a qualche altro metodo un oggetto TextReader o TextWriter, che sono le classi madre.

Per esempio, abbiamo già utilizzato la classe XmlReader per leggere una stringa contenente dei dati in formato XML.

Per ottenere un XmlReader da una stringa è necessario utilizzare il metodo Create, ma fra i suoi overload non ve n'è uno che accetti direttamente il contenuto XML in formato string.

Però vi è quello che prevede un TextReader, quindi possiamo utilizzare uno StringReader al suo posto:

```
string xmlString="<node></node>";
var xmlReader= XmlReader.Create(new StringReader(xmlString));
```

Le classi BinaryReader e BinaryWriter

Le classi BinaryReader e BinaryWriter permettono di leggere e scrivere dati dei tipi primitivi in maniera più efficiente e compatta rispetto ai già visti StreamReader e StreamWriter, leggendoli e convertendoli direttamente da e verso byte. Queste classi sono dei wrapper di oggetti Stream, cioè degli oggetti che inglobano uno di questi, permettendo di leggere e scrivere tramite i propri metodi.

Per creare un oggetto BinaryWriter, quindi, è necessario avere a disposizione lo stream sul quale si desidera scrivere:

```
FileStream binFs=File.OpenWrite("c:\\temp\\file.dat");
BinaryWriter bw = new BinaryWriter(binFs);
```

I diversi overload del metodo Write permettono di scrivere dati di vari tipi:

```
bw.Write(true); //bool
bw.Write(123); //int
bw.Write("hello"); //string
```

Come sempre, anche in questo caso è necessario chiudere il writer o il reader, per evitare di bloccare uno stream. Basta anche qui invocare il metodo Close oppure usare il pattern dispose. Chiudendo un BinaryReader o BinaryWriter, si chiude automaticamente anche lo stream sottostante.

Al contrario, per leggere un file binario, si può utilizzare un BinaryReader, il che implica di dover conoscere il formato esatto del file, cioè l'ordine in cui i dati sono stati scritti.

Per esempio, a partire dal file precedentemente creato, possiamo ricavarne i dati leggendoli in variabili dei tipi corrispondenti:

```
using(BinaryReader br=new BinaryReader(File.OpenRead("c:\\temp\\file.dat")))
{
booleano = br.ReadBoolean();
intero = br.ReadInt32();
stringa = br.ReadString();
}
```

Fra i metodi di lettura, è molto comodo ReadBytes, che permette di leggere in un solo colpo tutti i byte di un flusso, di cui è possibile ricavare la lunghezza:

```
Stream streamb=File.OpenRead("c:\\temp\\file.dat");
```

```
using (BinaryReader br = new BinaryReader(streamb))
{
byte[] allBytes=br.ReadBytes((int)streamb.Length);
}
```

Si evita in tal modo di dover eseguire un ciclo e verificare di aver raggiunto la fine dello stream.

Isolated Storage

Ogni applicazione .NET può sfruttare una propria area riservata di memorizzazione dei dati, isolata dalle altre applicazioni e quindi più sicura perché queste ultime non avranno alcun accesso a essa.

L'*Isolated Storage*, inoltre, è in qualche caso anche l'unica soluzione utilizzabile. Per esempio, nelle applicazioni Silverlight che non hanno accesso al file system locale e quindi non possono leggere o creare elementi di file system neanche in directory speciali, come ApplicationData.

L'Isolated Storage è infatti una sorta di directory virtuale, gli utenti non hanno bisogno di conoscere la sua effettiva locazione sul disco, e fra l'altro essa varia per ogni sistema operativo.

Tale area è utile in diversi scenari e situazioni, per esempio per memorizzare impostazioni utente, impostazioni delle applicazioni, profili e così via.

Accesso all'Isolated Storage

Per accedere all'Isolated Storage è necessario utilizzare nuove e dedicate classi, che evitano quindi la necessità di sapere qual è il percorso fisico reale. Inoltre, è possibile avere un Isolated Storage separato e dedicato a ogni assembly o utente del sistema.

Le classi .NET per interagire con esso appartengono al namespace System.IO.IsolatedStorage.

La classe IsolatedStorageFile rappresenta l'area di Isolated Storage che può contenere file e directory.

Il primo passo è ottenerne una istanza, per mezzo del metodo statico GetStore al quale passare dei parametri che indicano il tipo o livello dell'area di memorizzazione. L'enumerazione IsolatedStorageScope contiene i membri che rappresentano tali livelli:

- Application;
- Assembly;
- Domain;
- Machine;
- None;
- Roaming;
- User.

Per esempio, per ottenere l'oggetto IsolatedStorageFile per l'utente e l'assembly corrente si può scrivere:

```
IsolatedStorageFile isoFile =
IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
IsolatedStorageScope.Assembly |
IsolatedStorageScope.User,
null,
null);
```

Come detto, la classe espone metodi che evitano di specificare a mano i valori dell'enumerazione. Per esempio, lo stesso risultato precedente potrà essere ottenuto semplicemente utilizzando il metodo GetUserStoreForAssembly.

A questo punto, l'oggetto IsolatedStorageFile può essere utilizzato per leggere e scrivere elementi, invocandone metodi e proprietà. Per esempio, se si vogliono ottenere i file presenti nell'area, oppure verificare la presenza di un preciso file, è possibile utilizzare il metodo GetFileNames:

```
string filename = "isofile.txt";
IsolatedStorageFile isoStore = IsolatedStorageFile.GetStore(IsolatedStorageScope.User | IsolatedStorageScope.Assembly, null,
null);

string[] fileNames = isoStore.GetFileNames(filename);
foreach (string file in fileNames)
{
if (file == filename)
{
Console.WriteLine("Il file esiste!");
}
}
```

La classe IsolatedStorageFileStream, che deriva da FileStream, fornisce l'accesso al contenuto di un file.

Per scrivere del testo sul file isofile.txt si può quindi utilizzare uno StreamWriter:

```
using (IsolatedStorageFileStream oStream = new IsolatedStorageFileStream(filename, FileMode.Create, isoStore))
{
StreamWriter writer = new StreamWriter(oStream);
writer.WriteLine("Prima linea nel file isolated storage.");
writer.WriteLine("seconda linea");
writer.Close();
}
```

Viceversa, per leggere il contenuto basterà utilizzare uno StreamReader:

```
using (IsolatedStorageFileStream iStream = new IsolatedStorageFileStream(filename, FileMode.Open, isoStore))
{
StreamReader reader = new StreamReader(iStream);
String line;
while ((line = reader.ReadLine()) != null)
```

```
{
Console.WriteLine(line);
}
}
```

Accesso ai database

L'accesso ai database è essenziale in un abbondante numero di applicazioni di qualunque genere, siano esse applicazioni eseguite su un sistema desktop, applicazioni web, app per smartphone.

Molto spesso, da sviluppatori, ci si ritroverà a dover eseguire una connessione a un database, ricavare o aggiornare i dati e visualizzarli all'interno dei nostri programmi.

La piattaforma .NET comprende un abbondante numero di classi, componenti e strumenti che consentono di accedere e interagire con diverse tipologie di database relazionali, per la creazione di applicazioni distribuite e orientate alla gestione e allo scambio di dati.

Questo insieme di API è chiamato nel complesso *ADO.NET*, ed è costituito da un'infrastruttura contenente le funzionalità comuni, che si specializza poi in differenti sezioni, ognuna dedicata a un diverso tipo di connessione o a una particolare categoria di *DBMS* (*Database Management System*), come SQL Server, Oracle, MySQL e così via.

ADO.NET

ADO.NET si è evoluto notevolmente negli anni e oggi può essere visto come una tecnologia che permette tre differenti approcci per consumare dati conservati su un database relazionale: in maniera connessa, in maniera disconnessa, e tramite un approccio orientato agli oggetti, quest'ultimo utilizzato per esempio da *LINQ to SQL* e da *Entity Framework* (oltre che da altri sistemi di terze parti).

Quando si usa la parte *connessa* di ADO.NET, ci si connetterà esplicitamente a un database per eseguire direttamente dei comandi o query in linguaggio *SQL* (*Structured Query Language*), per realizzare operazioni di creazione, lettura, scrittura ed eliminazione di dati. L'infrastruttura ADO.NET si occuperà di colloquiare con il DBMS scelto, inviare le query e ricevere i risultati.

La parte *disconnessa* di ADO.NET prevede invece l'utilizzo di oggetti che replicano in memoria la struttura fisica del database relazionale, come i *DataSet*, che contengono a loro volta delle *DataTable*. In tal modo, le applicazioni possono utilizzare questi oggetti senza essere connessi costantemente al DBMS, eseguire delle operazioni su di essi e poi sincronizzarsi con il database una volta stabilita la connessione vera e propria, il tutto in maniera trasparente per lo sviluppatore e quindi per l'applicazione.

Infine, un differente approccio di accesso ai database è costituito da una tecnica che in generale è nota con il nome di *Object Relational Mapping* (*ORM*) e che permette di utilizzare un approccio completamente orientato agli oggetti, trattando cioè classi e oggetti per lavorare su database relazionali senza necessità di scrivere codice per la connessione, esecuzione di query, disconnessione e altri aspetti esplicitamente legati a un DBMS. Ogni entità del database fisico (una tabella, per esempio) corrisponderà a una classe, i cui metodi permetteranno di leggere e scrivere i dati, rimandando all'ORM il compito di tradurre poi queste operazioni in comandi specifici del database.

L'ORM che fa parte di ADO.NET è noto con il nome di *Entity Framework*.

NOTA	Si tenga presente che esistono diversi altri sistemi (gratuiti, open source, commerciali) di tipo ORM. Molti di essi offrono API, strumenti di design, utility per sviluppare applicazioni di accesso ai database, nella maniera più rapida ed efficace possibile. La scelta di uno di essi è spesso una questione di gusto personale, legata al tipo di applicazione che si intende sviluppare, di prestazioni e, perché no, di costi di licenza.

Una considerazione a parte merita *LINQ to SQL*, che è l'implementazione di LINQ (cioè il LINQ Provider) dedicata solo ed esclusivamente a SQL Server.

Esso permette quindi di eseguire query SQL e inviarle al database utilizzando però la sintassi e gli operatori di LINQ. Quando il database restituisce i risultati, LINQ to SQL si occupa di convertirli di nuovo in oggetti dei tipi .NET, che possono essere utilizzati direttamente in C#.

Una trattazione completa degli argomenti introdotti fin qui richiederebbe naturalmente ben più di un testo a essi dedicato. Tuttavia, essendo questo un libro di programmazione in linguaggio C#, si cercherà di fornire una buona introduzione all'argomento, rimandando il lettore ad approfondire per conto proprio le parti che lo interessano maggiormente.

Introduzione ai database relazionali

Prima di metterli in pratica scrivendo del codice, riassumiamo brevemente in questo paragrafo un po' dei concetti che stanno dietro all'argomento "database", in particolare quelli *relazionali*, senza pretesa di essere esaustivi, ma con il solo scopo di permettere, anche a chi non ha mai sentito parlare di database, di iniziare a sviluppare applicazioni che li utilizzano.

Se avete già un minimo di esperienza in merito, o perlomeno sapete già cosa sono tabelle, campi e concetti del genere, potete tranquillamente saltare il paragrafo, o leggerlo abbastanza rapidamente.

Un *database* è un insieme strutturato e organizzato di dati. Esistono diversi modi e strategie per organizzare questi dati, permettere l'accesso a essi e la loro manipolazione.

Un *Database Management System*, o *DBMS*, è invece un software che si occupa di gestire uno o più database, consentendone l'amministrazione, la definizione, l'interrogazione, la modifica e l'aggiornamento.

Un DBMS rappresenta l'interfaccia fra i database, che sono i contenitori veri e propri dei dati, le applicazioni e quindi indirettamente gli utenti, che possono così accedere e utilizzare i dati stessi.

Esistono diverse categorie di database, suddivise a seconda della modalità di strutturazione e gestione dei dati. Fra i più popolari al giorno d'oggi, sebbene non sia un concetto nuovo (proviene anzi dagli anni Settanta), vi è il modello relazionale.

Un database relazionale organizza i dati in *tabelle*, che possono avere delle relazioni fra di loro. Per interrogare o manipolare un database relazionale si utilizza un linguaggio standard chiamato *SQL*, acronimo di *Structured Query Language*, e che spesso è pronunciato *siquel*.

Un'istruzione SQL, chiamata *query*, la si può inviare direttamente al DBMS, che la eseguirà per ottenerne dati o per manipolare quelli presenti nel database.

Nel primo caso, lo stesso DBMS si occuperà di restituire i risultati, per esempio i dati che rispettano determinati criteri di ricerca; nel secondo, penserà ad aggiornare i dati presenti nelle tabelle del database o a eliminarli.

Altre tecniche, come LINQ, possono agire a un livello più alto, evitando allo sviluppatore di dover scrivere e quindi imparare un linguaggio come SQL.

Tabelle e colonne

Come detto, un database relazionale contiene i dati organizzati in tabelle.

Le *tabelle* sono a loro volta composte da *righe* (chiamate anche *record*) e *colonne* (chiamate anche *campi*) che contengono i dati.

Supponiamo, per esempio, di dover memorizzare i dati di un veicolo in un database relazione. Possiamo progettare una tabella Veicolo strutturata come nella Tabella 15.1 (una tabella come questa raffigurata nel libro fornisce una rappresentazione praticamente fedele del concetto di tabella di database).

Tabella 15.1 **- La tabella Veicolo di un database.**

ID	Targa	Marca	Modello
1	AB123CD	Fiat	Uno
2	ME923984	Alfa Romeo	GT
3	JB007	Aston Martin	DB5

Questa tabella consiste di tre righe e quattro colonne. Ogni riga rappresenta un diverso veicolo, mentre ogni colonna rappresenta un attributo di ogni veicolo, o campo. Quindi, un veicolo è dotato in questo caso degli attributi ID, Targa, Marca e Modello.

L'ID rappresenta un campo unico, che non può essere duplicato, ed è quindi chiamato *chiave primaria* o *primary key* della tabella. In tal modo, ogni ID rappresenta in maniera univoca un veicolo.

Non è obbligatorio, naturalmente, che la chiave primaria sia un valore numerico. Per esempio, nel caso della tabella Veicolo, la chiave primaria potrebbe anche essere rappresentata dal campo Targa, che è univoco perché non possono essere presenti veicoli con uno stesso numero di targa. Inoltre, una chiave primaria potrebbe essere anche composta da più campi, l'importante è che l'insieme di tali campi sia una combinazione univoca.

La presenza di una chiave serve per permettere al DBMS, e quindi alle applicazioni, di identificare ogni record: se si vuol eliminare un determinato record, oppure aggiornarlo, si potrà così accedere alla tabella e al campo identificato dalla sua chiave. Una chiave primaria serve poi a collegare una tabella con altre contenenti dati in qualche modo connessi alla prima; ci torneremo a breve.

Gli altri campi, invece, possono anche contenere dati duplicati: per esempio, possono esistere più veicoli di una certa marca o di un determinato modello.

Ogni campo o attributo di una tabella è destinato a contenere dati di tipo differente. Per esempio, il campo ID è di tipo numerico, mentre il campo Targa, così come i campi Marca e Modello, sono di tipo testuale.

Sebbene lo standard SQL definisca dei tipi standard utilizzabili per i campi di una tabella, ogni DBMS può definire i propri tipi e chiamarli in maniera differente, così come fanno linguaggi di programmazione differente.

In generale, comunque, si avranno a disposizione tipi numerici, tipi per le stringhe o singoli caratteri, tipi per data e orario, per contenere oggetti sotto forma di byte.

A puro titolo di esempio, e visto che lo utilizzeremo a breve, fra i tipi utilizzabili in un database SQL Server vi sono int, float, datetime, char, varchar, text, binary (l'elenco non è completo).

Riprendendo la tabella Veicolo, essa potrebbe utilizzare i seguenti tipi di dati per i suoi campi:

- ID: int;
- Targa: varchar(10);

- Marca: varchar(30);
- Modello: varchar(30).

Il tipo varchar è una stringa di lunghezza variabile, ma limitata al numero indicato fra parentesi.

Relazioni

Il sistema di database relazionale deriva il suo nome dal concetto di *relazioni* che si possono stabilire fra differenti tabelle. Per esempio, un veicolo può appartenere a un proprietario, quindi definiamo una tabella Persona, come mostrato nella Tabella 15.2.

Tabella 15.2 **- La tabella Persona.**

ID	Nome	Cognome
1	Antonio	Pelleriti
2	Caterina	Marguccio
3	Felicia	Scaffidi

Anche una Persona ha un identificatore univoco, che è la chiave prima della tabella, e due campi testuali per il nome e il cognome.

Per collegare una Persona al proprio Veicolo bisogna modificare quest'ultima tabella, aggiungendo un campo che permetta di referenziare il proprietario. Per riferirsi a un determinato record si utilizza quindi logicamente la chiave primaria, in questo caso il campo ID di Persona. Quindi, aggiungendo per esempio un campo numerico, IDProprietario, saremo in grado di assegnare e ricavare il record Persona collegato a ogni record Veicolo.

Tabella 15.3 **- La tabella Veicolo modificata.**

ID	Targa	Marca	Modello	IDProprietario
1	AB123CD	Fiat	Uno	2
2	ME923984	Alfa Romeo	GT	1
3	JB007	Aston Martin	DB5	3

Il campo IDProprietario, poiché si riferisce a una chiave primaria di un'altra tabella, viene detto *chiave esterna* o *foreign key* della tabella.

Si noti anche che all'interno della tabella Veicolo potrebbero esserci più record con lo stesso valore di IDProprietario, cioè una stessa Persona può essere collegata a più entità Veicolo. In tal modo, si è creata una relazione *uno a molti* fra le tabelle, che permette quindi di collegare un unico record di una tabella con molti record di un'altra.

È molto comodo utilizzare dei diagrammi di rappresentazione delle tabelle e delle relazioni di un database, chiamati *Entity-Relationship Diagram*, sia in fase di analisi sia in fase di

progettazione del database.

La Figura 15.1 mostra per esempio le due tabelle Persona e Veicolo collegate dalla relazione uno a molti.

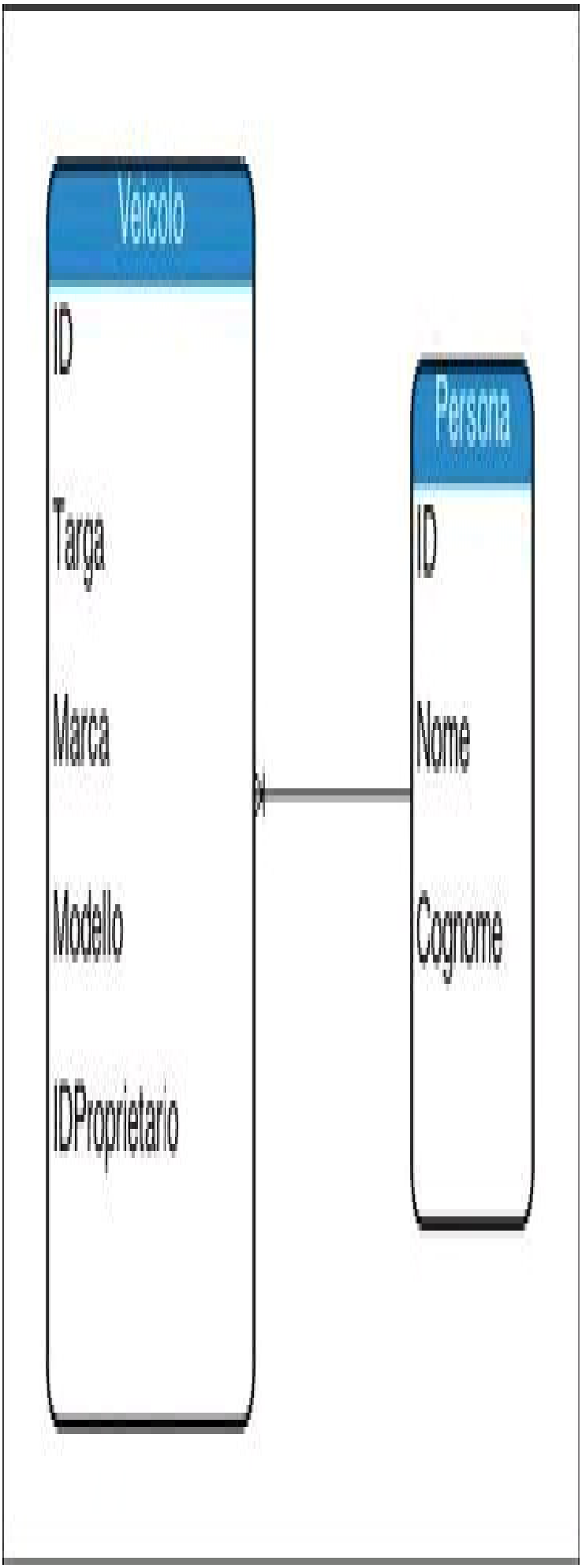

Veicolo
ID
Targa
Marca
Modello
IDProprietario
Persona
ID
Nome
Cognome

Figura 15.1 – Diagramma Entity-Relation per le tabelle Persona e Veicolo.

Per realizzare tali diagrammi esistono anche strumenti appositi, che ricavano le entità e le relazioni dal database vero e proprio o, viceversa, eseguono la creazione delle entità a partire dal diagramma.

In un database relazionale le tabelle sono collegabili utilizzando diversi tipi di relazione: uno a uno, uno a molti, molti a molti.

La relazione molti a molti è definibile solo utilizzando una terza tabella, chiamata *tabella di cross* o *di congiunzione*, che dispone di due chiavi esterne: una che si riferisce alla chiave primaria della prima tabella e una alla chiave primaria della seconda tabella.

Un tipico esempio di relazione molti a molti è quella presente fra una tabella Prodotto e una tabella Ordine, in quanto molti prodotti possono anche essere presenti in molti ordini.

Primi passi con SQL Server

Per iniziare a utilizzare un database, e passare poi allo sviluppo di applicazioni in C# che sfruttino le classi dei provider ADO.NET, LINQ to SQL, Entity Framework, o le altre tecniche che magari ognuno vorrà usare, abbiamo bisogno di un DBMS a nostra disposizione e qualche database di prova. Se non avete niente del genere, sfrutteremo una delle versioni gratuite e liberamente utilizzabili di *Microsoft SQL Server*. Infatti, così come Visual Studio, anche SQL Server è disponibile in più versioni express (esistono, come è lecito aspettarsi, versioni commerciali il cui costo può arrivare a migliaia di euro, come la Standard, la Enterprise, la Data Center e così via, ma credo che per il momento possiate farne a meno).

Il primo passo è quindi scegliere la versione che si desidera installare sul proprio sistema (che probabilmente non sarà un server vero e proprio, ma la stessa macchina che si usa per sviluppare), scaricarla e procedere alla sua installazione.

Versioni e installazione di SQL Server

Al momento della stesura di questo libro, l'ultima versione di SQL Server disponibile è la *SQL Server 2014*. Naturalmente, se disponete di una versione precedente già installata, 2012 o anche 2008, non siete obbligati ad aggiornare nulla: per lo scopo di questo libro è più che sufficiente, e per non far torto a nessuno, se siete anche in grado di amministrare e gestire un database di diverso tipo, per esempio *Microsoft Access*, *MySQL* o *Oracle*, siete liberi di continuare a usarlo, magari andando a installare solo i data provider necessari (per Access è sufficiente utilizzare il provider OleDb predefinito).

NOTA Anche Visual Studio comprende una versione ridotta di SQL Server denominata *LocalDB*, che viene installata automaticamente. Essa può in ogni caso essere installata a parte, come vedremo nel seguito, se l'avete rimossa durante l'installazione dell'IDE.

Le *Express Edition* hanno delle limitazioni che possiamo tranquillamente accettare, come quella nella dimensione di un singolo database, che è di 10 GB, oppure il fatto che non possano utilizzare processori multipli.

Avrete notato che ho usato un paio di volte il plurale nel parlare di versioni gratuite o di Express Edition. Infatti, sulla pagina di download (che potete raggiungere all'url https://www.microsoft.com/en-us/download/details.aspx?id=42299, oppure utilizzando un motore di ricerca qualsiasi con le keyword "SQL Server 2014 Express download"), una volta selezionata la lingua di installazione (vi consiglio di utilizzare sempre quella inglese), vi verrà proposto un elenco dei vari pacchetti disponibili (vedi Figura 15.2), che si differenziano sia per architettura, 32 o 64 bit, sia, come noterete facilmente, per dimensione del download.

Ogni pacchetto, infatti, contiene diversi eventuali componenti aggiuntivi, oltre al DBMS vero e proprio.

Per esempio, la versione più completa è quella da circa 1.1 GB che è chiamata *Express with Advanced Services*: essa comprende tutti i componenti, come lo strumento SQL Server Management Studio (SSMS), per la creazione e l'amministrazione dei database, e dei servizi avanzati di reporting e di ricerca Full Text.

Un buon compromesso fra dimensione e strumenti forniti è la versione *Express with Tools*, che fornisce il motore di database e il già citato SQL Server Management Studio. È spesso la scelta consigliata per un'installazione da utilizzare su una postazione di sviluppo.

Se invece non avete necessità di installare tutti i componenti forniti, oppure non volete dedicare tutto lo spazio richiesto, potete comunque scegliere una versione che è stata introdotta a partire da SQL Server 2012, denominata *LocalDB*, appositamente pensata per gli sviluppatori che vogliono in ogni caso avere a disposizione tutte le funzionalità di gestione e programmazione, compresa la possibilità di utilizzare SSMS per connettersi al DBMS e gestire i propri database, creando un piccolo ambiente di lavoro SQL Server.

Microsoft® SQL Server® 2014 Express

Select Language: English

Download

Microsoft SQL Server 2014 Express is a powerful and reliable free data management system that delivers a rich and reliable data store for lightweight Web Sites and desktop applications.

Details

Note: There are multiple files available for this download.Once you click on the "Download" button, you will be prompted to select the files you need.

Version:	Date Published:
12.0.2000.8	6/25/2014

File Name:	File Size:
Express 32BIT WoW64\SQLEXPR32_x86_ENU.exe	149.9 MB
Express 32BIT\SQLEXPR_x86_ENU.exe	168.4 MB
Express 64BIT\SQLEXPR_x64_ENU.exe	196.7 MB
ExpressAdv 32BIT\SQLEXPRADV_x86_ENU.exe	1.1 GB
ExpressAdv 64BIT\SQLEXPRADV_x64_ENU.exe	1.1 GB
ExpressAndTools 32BIT\SQLEXPRWT_x86_ENU.exe	840.8 MB
ExpressAndTools 64BIT\SQLEXPRWT_x64_ENU.exe	833.2 MB
LocalDB 32BIT\SqlLocalDB.msi	36.6 MB
LocalDB 64BIT\SqlLocalDB.msi	43.1 MB
MgmtStudio 32BIT\SQLManagementStudio_x86_ENU.exe	673.0 MB
MgmtStudio 64BIT\SQLManagementStudio_x64_ENU.exe	683.9 MB

Figura 15.2 – Pagina di download di SQL Server 2014 Express Edition.

La pagina di download delle versioni di SQL Server comprende anche un pacchetto dedicato a LocalDB (che infatti è quello con dimensione minore, circa 43 MB).

L'installazione della versione LocalDB è di per sé immediata e molto rapida, visto che non necessita di particolari configurazioni e non installa nessun servizio (viene eseguito come un normale processo, a differenza delle versioni più avanzate).

Per gestire e amministrare i database, è però molto utile avere a disposizione gli strumenti di Management Studio. Il pacchetto di installazione di quest'ultimo, a sua volta, permette di installare anche LocalDB, quindi, per iniziare a configurare la vostra postazione di sviluppo, scaricate il file denominato SQLManagementStudio_***.exe (gli asterischi dipendono dalla lingua e dall'architettura che andrete a scegliere: x86 oppure x64).

Database di esempio

Una volta installato il DBMS SQL Server e lo strumento di gestione Management Studio, siete già pronti a creare il vostro primo database, ma, per eseguire degli esempi su una sorgente dati abbastanza complessa e già riempita, potete scaricare e aggiungere alla vostra installazione del DBMS il database di esempio fornito da Microsoft, denominato *AdventureWorks*.

Andate alla pagina http://msftdbprodsamples.codeplex.com/, selezionate la sezione Downloads, e scaricate il file più recente compatibile con la versione 2014 di SQL Server. Sono disponibili sia i file con gli script da eseguire sul server per ricreare il database, sia i backup completi da ripristinare direttamente. Se avete difficoltà o non avete mai affrontato la procedura di creazione o ripristino di backup, sono disponibili anche dei documenti con le istruzioni, ma vedremo comunque a breve come eseguire tale ripristino.

Avviato SQL Server Management Studio, basta connettersi all'istanza di SQL Server su cui si desidera lavorare, che avete installato e configurato in precedenza.

Per effettuare la connessione a un qualunque server di database, è necessario conoscere, oltre all'indirizzo del server, il nome dell'istanza al quale connettersi e i relativi dati di accesso.


Connect to Server
Microsoft SQL Server 2014
Server type:
Database Engine
Server name:
(localdb)\MSSQLLocalDB
Authentication:
Windows Authentication
User name:
YODA\Antonio
Password:
Remember password
Connect
Cancel
Help
Options >>

Figura 15.3 – Finestra di connessione a SQL Server 2014.

L'installazione di SQL Server (anche nella versione LocalDB suddetta) eseguita con le opzioni predefinite vi permetterà di accedere usando la *Windows Authentication*, cioè la protezione integrata di Windows, che utilizza l'utente stesso del sistema operativo e non richiede quindi l'inserimento di nessun username e nessuna password. Basta pertanto indicare il nome dell'istanza di SQL Server alla quale connettersi (probabilmente troverete il campo Server Name già compilato con il nome di quella appena installata) e cliccare sul pulsante Connect.

Nel caso della versione LocalDB, il nome dell'istanza predefinita sarà in genere MSSQLLocalDB, mentre con SQL Server Express sarà SQLExpress. Nel campo Server Name va inserito il percorso completo del nome o dell'indirizzo del server, che se è in esecuzione sulla stessa macchina locale può essere sostituito dal punto, seguito poi dal carattere \ e dal nome dell'istanza. Per esempio, per connettersi all'istanza SQLExpress locale, basta inserire .\SQLEXPRESS. Nella versione LocalDB, invece, la stringa da utilizzare è (localdb)\MSSQLLocalDB.

Il nome dell'istanza sarà importante anche per effettuare la connessione da codice C# mediante un apposito metodo, ma ci torneremo a breve.

Restore Database - AdventureWorks2014

Ready

Select a page

- General
- Files
- Options

Script ▾ Help

Source

Database:

Device: D:\SqlServer Data\MSSQL10_50.SQLEXPRESS\MSSQL\Backup\Adventur ...

Database: AdventureWorks2014

Destination

Database: AdventureWorks2014

Restore to: The last backup taken (giovedì 17 luglio 2014 16:18:18) Timeline...

Restore plan

Backup sets to restore:

Restore	Name	Component	Type	Server	Datab
☑	AdventureWorks2014-Full Database Backup	Database	Full	VCG-SCULLEY\SQL2014MULTI	Adve

Verify Backup Media

Connection

.\SQLEXPRESS [sa]

View connection properties

Progress

Done

OK Cancel Help

Figura 15.4 – Ripristino di un backup di database SQL Server.

Una volta connessi, SSMS presenterà sulla sinistra una finestra Object Explorer, in cui è possibile esplorare i vari componenti e oggetti del server.

L'archivio di esempio scaricato in precedenza contiene il file di backup in formato BAK, che è possibile ripristinare nel proprio DBMS; intanto decomprimeteli nella sottocartella backup della vostra installazione di SQL Server, o comunque di quella del percorso che avete scelto per contenere appunto i database. La procedura seguente è generale e può essere utilizzata per ripristinare un qualunque backup di database.

Fate clic con il pulsante destro sul ramo Databases, nell'Object Explorer di SSMS, e cliccate poi sulla voce Restore Database. A questo punto si aprirà la finestra di selezione dei file da ripristinare, selezionate l'opzione Device, e con l'apposito pulsante Sfoglia, andate a selezionare il file contenente il backup. Infine fate clic sul pulsante Add.

Ora basta confermare con il pulsante OK e il database verrà riagganciato al vostro SQL Server.

Se invece volete creare un database vuoto cui aggiungere poi le vostre tabelle e i vostri dati, basta, dopo aver cliccato sul ramo Database, selezionare la voce New Database e quindi scegliere un nome da assegnargli.

Il ramo database dell'Object Explorer di SSMS conterrà tutti i database gestiti dal DBMS.

All'interno di ogni nodo di database sono contenuti i diversi oggetti che ne costituiscono la struttura, come tabelle e viste, oltre a quelli che permettono di gestire altri aspetti come utenti e ruoli di accesso.

Strumenti per database di Visual Studio

A partire da Visual Studio 2010, è possibile integrare tutti gli strumenti necessari a connettersi e lavorare con un database nell'IDE, quindi in molti casi potete anche fare a meno di strumenti esterni e lavorare esclusivamente all'interno di tale ambiente.

Dopo aver installato SQL Server, anche in versione LocalDB, o se avete già un'installazione di Visual Studio completa degli strumenti di database, potete creare una connessione al server e accedere a un database SQL Server, dalla finestra SQL Server Object Explorer. Se essa non è visualizzata, aprite il menu View e selezionate la voce omonima.

NOTA

Gli strumenti necessari a connettersi e lavorare con database SQL Server da Visual Studio sono denominati *SQL Server Data Tools* (o SSDT). Possono essere scaricati e installati

anche in maniera separata da Visual Studio. I file di installazione sono disponibili al seguente indirizzo: https://msdn.microsoft.com/en-us/library/mt204009.aspx.

Per iniziare a lavorare con un server SQL Server, è necessario aggiungerlo alla finestra. Basta fare clic con il pulsante destro sul nodo SQL Server e selezionare la voce Add SQL Server, oppure fare clic sul pulsante analogo nella barra in alto. A questo punto, si aprirà la finestra di inserimento dati di connessione simile a quella già vista in Figura 15.3. La procedura di connessione è analoga e, una volta connessi, apparirà il nodo con il nome del database e al suo interno gli oggetti che fanno parte di esso.

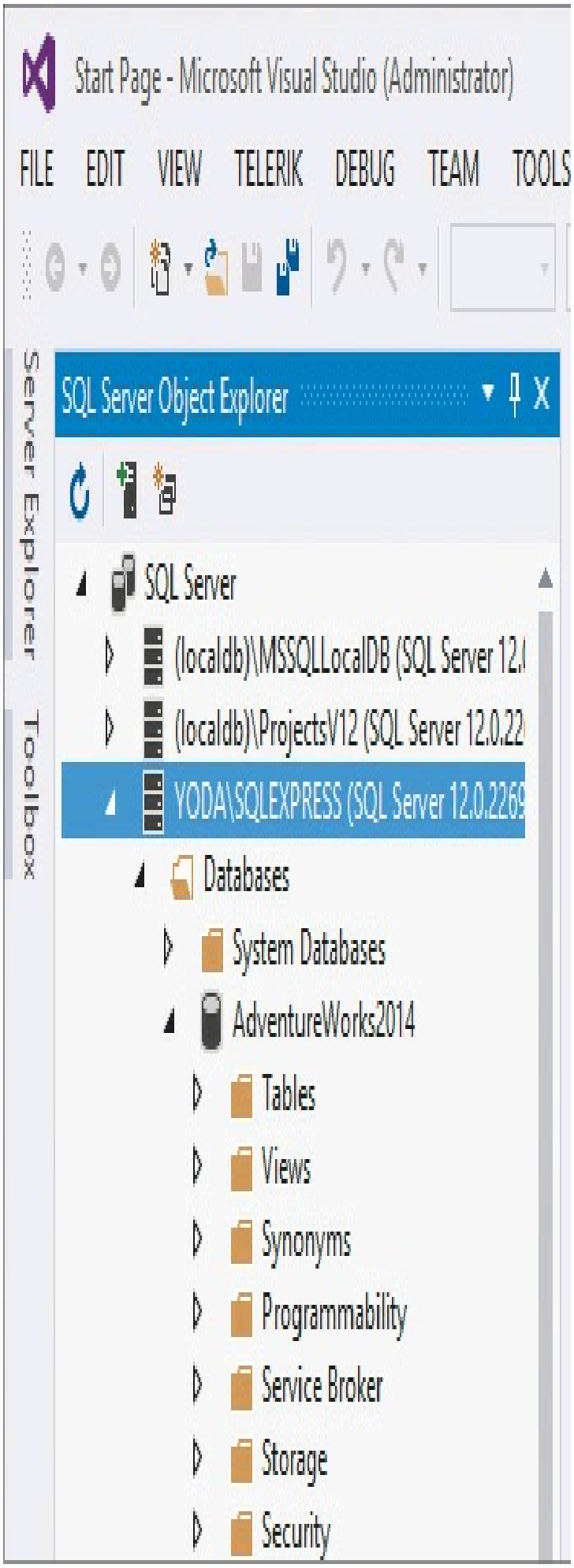

Start Page - Microsoft Visual Studio (Administrator)
FILE
EDIT
VIEW
TELERIK
DEBUG
TEAM
TOOLS
Server Explorer
Toolbox
SQL Server Object Explorer
SQL Server
(localdb)\MSSQLLocalDB (SQL Server 12.
(localdb)\ProjectsV12 (SQL Server 12.0.22
YODA\SQLEXPRESS (SQL Server 12.0.2269
Databases
System Databases
AdventureWorks2014
Tables
Views
Synonyms
Programmability
Service Broker
Storage
Security

Figura 15.5 – Connessione a SQL Server da Visual Studio 2015.

dbo.DimProduct [Design] - Microsoft Visual Studio (Administrator)

FILE EDIT VIEW TELERIK PROJECT BUILD DEBUG TEAM TOOLS ARCHITECTURE TEST DRIVER ANALYZE DEVEXPRESS WINDOW HELP

SQL Server Object Explorer

- SQL Server
  - (localdb)\MSSQLLocalDB (SQL Server 12.0
  - (localdb)\ProjectsV12 (SQL Server 12.0.22
  - YODA\SQLEXPRESS (SQL Server 12.0.2269
    - Databases
      - System Databases
      - AdventureWorks2014
      - AdventureWorksDW2014
        - Tables
          - System Tables
          - FileTables
          - dbo.AdventureWorksDWBu
          - dbo.DatabaseLog
          - dbo.DimAccount
          - dbo.DimCurrency
          - dbo.DimCustomer
          - dbo.DimDate
          - dbo.DimDepartmentGroup
          - dbo.DimEmployee
          - dbo.DimGeography
          - dbo.DimOrganization
          - dbo.DimProduct
          - dbo.DimProductCategory
          - dbo.DimProductSubcatego
          - dbo.DimPromotion
          - dbo.DimReseller
          - dbo.DimSalesReason
          - dbo.DimSalesTerritory
          - dbo.DimScenario
          - dbo.FactAdditionalInternati
          - dbo.FactCallCenter
          - dbo.FactCurrencyRate

dbo.DimProduct [Data] | dbo.DimProduct [Design]

Update Script File: dbo.DimProduct.sql

Name	Data Type	Allow Nulls	Default
ProductKey	int	☐	
ProductAlternateKey	nvarchar(25)	☑	
ProductSubcategoryKey	int	☑	
WeightUnitMeasureCode	nchar(3)	☑	
SizeUnitMeasureCode	nchar(3)	☑	
EnglishProductName	nvarchar(50)	☐	
SpanishProductName	nvarchar(50)	☐	
FrenchProductName	nvarchar(50)	☐	
StandardCost	money	☑	
FinishedGoodsFlag	bit	☐	
Color	nvarchar(15)	☐	
SafetyStockLevel	smallint	☑	
ReorderPoint	smallint	☑	

- **Keys** (2)
  - PK_DimProduct_ProductKey (Primary Key, Clustered: ProductKey)
  - AK_DimProduct_ProductAlternateKey_StartDate (ProductAlternate
- **Check Constraints** (0)
- **Indexes** (0)
- **Foreign Keys** (1)
  - FK_DimProduct_DimProductSubcategory (ProductSubcategoryKe
- **Triggers** (0)

Design | T-SQL

```
CREATE TABLE [dbo].[DimProduct] (
    [ProductKey]            INT            IDENTITY (1, 1) NOT NULL,
    [ProductAlternateKey]   NVARCHAR (25)  NULL,
    [ProductSubcategoryKey] INT            NULL,
    [WeightUnitMeasureCode] NCHAR (3)      NULL,
    [SizeUnitMeasureCode]   NCHAR (3)      NULL,
    [EnglishProductName]    NVARCHAR (50)  NOT NULL,
    [SpanishProductName]    NVARCHAR (50)  NOT NULL,
    [FrenchProductName]     NVARCHAR (50)  NOT NULL,
    [StandardCost]          MONEY          NULL,
```

100 %

Connection Ready | YODA\SQLEXPRESS | sa | AdventureWorksDW2014

Figura 15.6 – Visualizzazione della struttura di una tabella di SQL Server in Visual Studio 2015.

Espandendo il nodo del database AdventureWorks, appariranno le cartelle di oggetti del database, come tabelle, view e così via.

Facendo doppio clic su tali oggetti, appariranno all'interno della finestra di Visual Studio le relative finestre di gestione e modifica dell'oggetto selezionato, oppure sarà possibile visualizzare i dati di una tabella cliccando con il pulsante destro e poi scegliendo il comando View Data.

Il linguaggio SQL

Gli strumenti grafici come SSMS permettono di interagire facilmente con il DBMS e i suoi database, ma dal punto di vista dello sviluppatore è necessario conoscere anche la lingua parlata direttamente dal database, cioè il linguaggio SQL.

SQL (Structured Query Language) permette di eseguire interrogazioni su un database, sia per ricavarne i dati e la struttura, sia per modificarli ed eliminarli.

Per eseguire delle query SQL, anche solo per imparare i rudimenti, o quando è necessario eseguirle a mano in caso di necessità, è sempre possibile utilizzare il già visto SQL Server Management Studio. Basta fare clic sul pulsante New Query nella barra degli strumenti in alto e poi selezionare il database al quale si invieranno i comandi in SQL.

La stessa azione è possibile dalla finestra SQL Server Object Explorer di Visual Studio: basta cliccare con il pulsante destro sul database sul quale si vogliono eseguire le istruzioni SQL e selezionare anche qui New Query.

Utilizzando strumenti e tecniche, come LINQ to SQL, Entity Framework o ORM (Object Relational Mapping), di terze parti, sarà molto raro scrivere direttamente query in linguaggio SQL all'interno di un'applicazione, ma è molto importante conoscerne la sintassi e saperla utilizzare, anche per capire cosa accade dietro le quinte, soprattutto in caso di problemi. Se non avete grande dimestichezza con il linguaggio SQL, vi consiglio di prendere almeno confidenza e di approfondirlo: per motivi di spazio non è possibile farlo in questo libro, anche se utilizzeremo nei prossimi paragrafi qualche semplice esempio.

Data provider

I *data provider* .NET consentono di connettersi a un database relazionale, eseguire comandi e leggere i risultati.

Essi sono stati progettati e implementati con una struttura molto leggera, e in maniera da creare un livello intermedio il più leggero possibile tra il codice della propria applicazione e la

sorgente dati.

Ogni data provider è dedicato a uno specifico Database Management System, oppure a una particolare tecnologia utilizzata per l'accesso ai dati.

Il namespace principale di ADO.NET è System.Data, che contiene numerose classi e interfacce comuni, le quali vengono specializzate e implementate dai differenti data provider in namespace a essi dedicati.

Avendo una base comune, ogni provider sarà utilizzabile in una maniera e con una sintassi consistente, che permetteranno di utilizzare provider e quindi database differenti senza difficoltà.

Il .NET Framework include tre famiglie di provider: quello per l'accesso diretto a database Microsoft SQL Server, nel namespace System.Data.SqlClient, e due per l'accesso indiretto, cioè per mezzo di appositi driver: *ODBC* (*Open Database Connectivity*), che si trova nel namespace System.Data.Odbc, e *OleDb* (Object Linking and Embedding Database), che invece si trova in System.Data.OleDb.

In aggiunta a essi, se si vuole lavorare con un provider che fornisce accesso diretto a un database, sono disponibili numerosi provider di terze parti che permettono di lavorare con i diversi sistemi di database open source e commerciali disponibili.

In molti casi è il produttore stesso del database che fornisce i provider .NET adeguati, mentre in altri è possibile ottenerli da produttori specializzati. Fra tali provider, citiamo quelli per MySQL, Oracle, DB2, SQLite, PostgreSQL, per restare fra i più utilizzati. Quindi, usando C#, non avete praticamente alcun limite di scelta.

Nel seguito del capitolo, utilizzeremo per i nostri esempi un database SQL Server e quindi le classi che costituiscono il data provider a esso dedicato.

Oggetti e classi ADO.NET

La connessione alla sorgente dati e l'accesso ai dati veri e propri in essa contenuti sono delle operazioni fondamentali in ogni applicazione basata su database.

I provider di dati .NET si occupano di implementare tali operazioni, costituendo il livello intermedio fra l'applicazione e il database.

I data provider di ADO.NET sono basati su un modello comune, che utilizza quattro oggetti principali, elencati nella Tabella 15.4.

Tabella 15.4 - **Oggetti principali di un provider di dati .NET.**

Oggetto	Descrizione

Connection	stabilisce la connessione alla sorgente dati
Command	rappresenta un comando da inviare alla sorgente dati
DataReader	consente di leggere un flusso di dati ricavati da una sorgente dati in maniera sequenziale e in sola lettura
DataAdapter	consente di popolare un DataSet con i dati ricavati da una sorgente e, viceversa, di inviare dati aggiornati al database

Il framework .NET contiene un insieme di classi di base e di interfacce che implementano gli oggetti elencati, contenute nel namespace System.Data.Common, che poi verrà specializzato da ogni apposito provider dedicato a un particolare tipo di database. Per esempio, la classe base di un oggetto Connection è DbConnection, la classe che rappresenta un Command è DbCommand, la classe base di ogni oggetto DataReader è DbDataReader e la classe di base per tutti i DataAdapter è la classe DbDataAdapter.

Oltre alle classi appena elencate, ogni provider dati di .NET fornisce delle classi di contorno e di utilità, come DbTransaction, per raggruppare ed eseguire i comandi all'interno di un'unica transazione, DbCommandBuilder, che è una classe helper che serve a generare i comandi SQL e le relative proprietà, o ancora la classe DbParameter che definisce i parametri di input, output per comandi e transazioni.

Nel seguito del capitolo utilizzeremo le classi specifiche dedicate al database SQL Server, il cui nome è facilmente intuibile: basta sostituire nel nome delle classi suddette la parte Db con Sql. Per esempio, una connessione a un database sarà ottenuta mediante un oggetto SqlConnection.

Connessione a database

Il primo passo per connettersi a un database è fornire i parametri di connessione al database, per esempio il nome o l'indirizzo del server e le credenziali di accesso. Tali parametri vengono forniti sotto forma di stringa, chiamata quindi appropriatamente *stringa di connessione* o *connection string*, da passare come parametro

Ogni database utilizza uno o più formati per definire la stringa di connessione. Per esempio, per connettersi a un database SQL Server LocalDB, con autenticazione di Windows integrata, si dovrà utilizzare una stringa come la seguente:

```
string connString = "Server=(localdb)\\MSSQLLocalDb;Integrated Security=true;database=AdventureWorks2014";
```

Naturalmente, non è molto comodo e sicuro fissare e inserire le stringhe di connessione all'interno del codice sorgente. Pensate, per esempio, se ci fosse necessità di cambiare stringa per connettersi a un server differente e se le connessioni venissero create in diverse parti del programma. Meglio avere un file di configurazione in cui inserirle e ricavarle quando servono, con apposite classi.

Per esempio, sia in applicazioni Windows sia in applicazioni Web ASP.NET, l'approccio consigliato è di utilizzare dei file di configurazione XML (app.config nel primo caso e web.config nel secondo) in cui memorizzare stringhe di connessione e altri valori di configurazione, e utilizzare la classe ConfigurationManager del .NET Framework per leggerle.

Per un database SQL Server, con credenziali di accesso basate su username e password, in generale si utilizzerà il seguente formato:

```
Server=myServerAddress;Database=myDataBase;User Id=myUsername;Password=myPassword;
```

NOTA	Ogni provider di dati ha i propri parametri da utilizzare per formare una stringa di connessione al relativo database. Per evitare di doverli imparare a memoria, il sito www.connectionstrings.com costituisce un ottimo riferimento per ottenere le stringhe di connessione a qualunque sistema di database esistente.

Una volta impostata la stringa, è possibile creare una connessione, cioè una linea di comunicazione con il database.

La classe SqlConnection, derivata da DbConnection, rappresenta una connessione a un database SQL Server. Passando la stringa di connessione al costruttore della classe o impostandone la relativa proprietà ConnectionString, si crea e configura un'istanza della connessione.

Dopodiché, si procede ad aprire la connessione stessa mediante il metodo Open, a utilizzarla per creare comandi, e poi chiuderla con il metodo Close:

```
SqlConnection connection=new SqlConnection(connString);
connection.Open();
//utilizza la connessione
connection.Close();
```

Ogni connessione va chiusa al termine del suo utilizzo per non consumare inutilmente risorse e, in questo caso, per evitare che un database o i suoi oggetti rimangano bloccati e impediscano ad altri utenti o applicazioni di accedervi.

Il pattern *dispose* è quello più indicato e utilizzato anche per questo scenario. Con l'istruzione using, la connessione sarà automaticamente chiusa al termine del suo utilizzo:

```
using(SqlConnection connection = new SqlConnection(connString))
{
connection.Open();
Console.WriteLine(connection.ServerVersion);
}
```

Nell'esempio precedente viene stampata la versione del server ricavata a seguito dell'avvenuta connessione allo stesso.

Fra le proprietà di DbConnection, e quindi delle varie classi da essa derivate come SqlConnection, nella Tabella 15.5 sono riportate le più comuni.

Tabella 15.5 - **Oggetti principali di un provider di dati .NET.**

Oggetto	Descrizione
ConnectionString	imposta o ricava la stringa di connessione utilizzata per accedere al database
ConnectionTimeout	imposta il tempo in secondi trascorso il quale il tentativo di connessione fallisce e viene restituito un errore
Database	restituisce il nome del database a cui si è connessi o quello indicato nella stringa di connessione se la connessione non è stata aperta.
DataSource	restituisce il nome del server di database
ServerVersion	restituisce la versione del server
State	restituisce lo stato della connessione, per esempio Close o Opened

Esecuzione di comandi

Una volta stabilita una connessione a un database, è possibile creare dei comandi da inviare a esso. Tali comandi sono in genere delle stringhe contenenti a loro volta dei comandi SQL che, a seconda dei casi, possono anche restituire dei risultati di vario genere. Nel caso di un database SQL Server, è possibile per esempio costruire un'istruzione SELECT, la quale restituisce tutti o parte dei campi che costituiscono i record di una tabella. La classe DbCommand rappresenta la classe base per i comandi implementati da ogni provider. Un comando ha bisogno di una connessione, da passare al costruttore, oppure da impostare come proprietà. Per esempio, ecco come costruire un SqlCommand, che è l'implementazione di DbCommand per il database SQL Server:

```
using(SqlConnection connection = new SqlConnection(connString))
{
connection.Open();
string sql = "SELECT * FROM DimProduct";
SqlCommand cmd = new SqlCommand(sql, connection);
}
```

Nel caso precedente, si è costruito il comando passando direttamente al costruttore l'istruzione in linguaggio SQL.

La classe DbCommand ha una proprietà CommandType che definisce come interpretare la stringa utilizzata come comando, impostabile a sua volta anche tramite la proprietà CommandText. Se il valore di CommandType è Text, che è quello predefinito, il testo viene interpretato come istruzione SQL. Se si usa il valore TableDirect, il CommandText deve essere impostato con il nome di una tabella del database esistente (supportato solo nel provider OleDb).

Infine, nel caso di StoredProcedure, il comando eseguirà una procedura memorizzata all'interno del database stesso.

Una volta definito il comando è necessario eseguirlo. Ogni classe Command possiede diversi metodi di esecuzione, da scegliere a seconda della tipologia di comando. Se il comando non restituisce alcun valore, si utilizzerà il metodo ExecuteNonQuery. Se il risultato è un unico valore scalare, il metodo da utilizzare sarà ExecuteScalar. Invece, se ci si aspetta un insieme di risultati da poter leggere sequenzialmente con un DataReader (in caso di SQL Server sarà un SqlDataReader), verrà utilizzato il metodo ExecuteReader.

A partire da .NET 4.5, inoltre, sono presenti anche le versioni asincrone dei vari metodi di esecuzione, per esempio ExecuteReaderAsync, ExecuteScalarAsync e ExecuteNonQueryAsync, che permettono di non bloccare l'interfaccia di un'applicazione mentre si attende la conclusione di un comando.

Il metodo ExecuteNonQuery è per esempio utilizzato per eseguire istruzioni SQL di inserimento (INSERT), modifica (UPDATE) ed eliminazione (DELETE). Quindi, esso esegue l'istruzione impostata su una singola tabella e restituisce il numero di record coinvolti nel comando stesso, per esempio il numero di record inseriti, modificati o eliminati:

```
string sqlUpdate = " UPDATE DimProduct SET Color = 'Rosso', ListPrice = 100 WHERE ProductKey=12345 ";
SqlCommand cmdUpdate = new SqlCommand(sqlUpdate, connection);
int affected=cmdUpdate.ExecuteNonQuery();
Console.WriteLine("{0} record aggiornati", affected);
```

La classe SqlCommand ha diverse proprietà, che permettono di impostare l'istruzione SQL, la connessione e altri valori necessari al di fuori del costruttore. Inoltre, è molto utile poter passare dei parametri alle istruzioni SQL, anziché costruirle direttamente sotto forma di stringa. Per esempio, supponiamo di voler eseguire un comando che aggiorna il campo CompanyName con un valore arbitrario e per un record identificato da un valore di CustomerId anch'esso passabile come parametro.

In tal modo, la stringa SQL potrà essere riutilizzata più volte, con parametri diversi, all'interno del nostro programma:

```
cmdUpdate = new SqlCommand();
cmdUpdate.Connection = connection;
cmdUpdate.CommandType = System.Data.CommandType.Text;
cmdUpdate.CommandText = "UPDATE DimCustomer SET FirstName = @NAME WHERE CustomerKey=@CustomerKey";
cmdUpdate.Parameters.AddWithValue("@CustomerKey", 2);
cmdUpdate.Parameters.AddWithValue("@NAME", "L'ape maia");
affected= cmdUpdate.ExecuteNonQuery();
```

Si noti come i parametri di un SqlCommand siano identificati dal carattere @ prima del loro nome. Con altri provider, la modalità di identificazione dei parametri potrebbe essere

differente. Per esempio, un OleDbCommand non supporta i parametri denominati; in tal caso viene utilizzato il carattere ? come segnaposto di ogni parametro e i valori dei parametri corrispondenti vanno aggiunti nell'ordine corretto.

Il metodo ExecuteScalar viene utilizzato quando si vuol ottenere un singolo valore, per esempio per ottenere il numero di record restituiti da una query SQL, oppure il risultato di una funzione di aggregazione come SUM.

Il seguente esempio restituisce il numero di record della tabella Product che hanno 'Black' come valore del campo Color:

```
SqlCommand cmdCount = new SqlCommand("SELECT COUNT(*) FROM SalesLT.Product WHERE Color='Black'",
connection);
int numero = (int)cmdCount.ExecuteScalar();
```

Il metodo restituisce un valore di tipo object, quindi è necessario convertirlo esplicitamente nel tipo desiderato.

Quando il comando da eseguire prevede la restituzione di un insieme di risultati, per esempio di più record di una tabella, il modo più semplice è quello di ottenere un DbDataReader per mezzo del metodo ExecuteReader. Un DbDataReader, come un SqlDataReader nel caso di SQL Server, permette di iterare i risultati senza necessità di mantenere in memoria un intero elenco, potenzialmente molto lungo. Tale oggetto rimarrà connesso al database fino a quando non verrà chiuso.

L'utilizzo ideale del metodo è quindi quello per l'esecuzione di istruzioni SELECT.

Il seguente esempio ottiene un SqlDataReader che permette di leggere tutti i record della tabella Customer:

```
string sql = "SELECT * FROM SalesLT.Customer";
SqlCommand cmd = new SqlCommand(sql, connection);
SqlDataReader reader= cmd.ExecuteReader();
if (reader.HasRows)
{
while (reader.Read())
{
int id = reader.GetInt32(0);
string name = reader["FirstName"] as string;
string lastname = reader["LastName"] as string;
Console.WriteLine("{0} {1} {2}", id, name, lastname);
}
}
reader.Close();
```

Il metodo Read restituisce il valore true fino a quando ci sono altri record da leggere.

All'interno del ciclo while, a titolo di esempio, vengono ricavati i campi id, FirstName e LastName.

Si noti come sia possibile utilizzare un metodo basato sul tipo e la posizione del campo. È il caso di GetInt32(0), che ricava un numero intero presente nel record in esame all'indice 0, oppure un accesso tramite indicizzatore, indicando il nome del campo che si vuol ottenere. Nel secondo caso, è necessario convertire il valore ottenuto nel tipo desiderato e corretto. Ciò implica che bisogna conoscere la struttura del database.

NOTA	Se un campo del database contiene un valore NULL, in C# verrà restituito un oggetto DBNull.Value, che è diverso dal famigerato null che indica un riferimento nullo. Quindi si faccia attenzione a questi casi, che potrebbero comportare bug difficili da scovare.

Per mezzo di un DbDataReader, ed eventualmente anche di tutte le altre classi messe a disposizione da un provider .NET, è possibile comunque esaminare e ricavare l'intera struttura di un database.

Il seguente esempio ricava nome e tipo di tutte le colonne della tabella Product, per mezzo del metodo GetSchemaTable di DbDataReader:

```
SqlCommand cmdSchema = new SqlCommand("SELECT * FROM SalesLT.Product", connection);
using (var readerSchema = cmdSchema.ExecuteReader(CommandBehavior.SchemaOnly))
{
var tableSchema = readerSchema.GetSchemaTable();
foreach (DataRow row in tableSchema.Rows)
{
Console.WriteLine(row["ColumnName"] + " - " + row["DataTypeName"]);
}
}
```

In questo caso, visto che ci interessa solo ricavare lo schema della tabella, al metodo ExecuteReader è stato passato il parametro CommandBehavior.SchemaOnly, in maniera da evitare di leggere anche i dati.

Ci si ricordi di chiudere sempre l'oggetto reader, per evitare di mantenere attiva la connessione e impedire di utilizzarla per altri comandi.

Un altro medoto peculiare del provider SQL Server permette di ricavare dati direttamente in formato XML da un database. La classe SqlCommand, infatti, espone anche il metodo ExecuteXmlReader. Il comando deve utilizzare in questo caso la clausola speciale FOR XML:

```
SqlCommand cmdXml = new SqlCommand("SELECT TOP 5 ProductID, Name, Color FROM SalesLT.Product FOR XML
AUTO, XMLDATA", connection);
XmlReader xmlReader = cmdXml.ExecuteXmlReader();
while(xmlReader.Read())
Console.WriteLine("{0}", xmlReader.ReadOuterXml());
```

```
xmlReader.Close();
```

Anche qui, ricordatevi di chiudere sempre l'oggetto XmlReader.

Modalità disconnessa di ADO.NET

Nei paragrafi precedenti abbiamo visto come utilizzare un provider dati .NET per ricavare informazioni, manipolarle, e interagire con un database in una modalità detta quindi *connessa*.

In questo e nei seguenti paragrafi, si esporrà invece una seconda possibilità per ricavare risultati e aggiornare i dati: una modalità detta *disconnessa*.

Questa modalità sfrutta in particolare altre classi del namespace System.Data, che costituiscono un modello a oggetti del database e che permettono di rappresentare e mantenere in memoria sia la struttura sia i dati, dando l'illusione di essere connessi al database reale.

NOTA	La modalità disconnessa di ADO.NET è stata praticamente soppiantata dalle funzionalità offerte da Entity Framework, che verrà discusso più avanti, per cui daremo qui una rapida panoramica utile a completare l'argomento ADO.NET.

Le classi DataSet e DataTable

La classe DataSet rappresenta un contenitore di dati, che non ha alcuna nozione o conoscenza di database. Infatti, è possibile riempire un DataSet con dati non necessariamente provenienti da un database, ma per esempio leggendoli da file di vario genere, come XML, o manualmente via codice.

Ogni DataSet consiste di una o più DataTable, che rappresentano oggetti simili alle tabelle di un database, con colonne e righe, rappresentate rispettivamente da oggetti di tipo DataColumn e DataRow. Vi è inoltre la possibilità di definire una chiave primaria e relazioni fra gli oggetti DataTable di un DataSet.

La proprietà Tables di DataSet consente l'accesso alla collezione di DataTable in esso contenute.

La classe DataTable a sua volta possiede una proprietà Columns contenente gli oggetti DataColumn che rappresentano le sue colonne. Ogni DataColumn definisce la struttura comprendente il nome del campo, il tipo di dati, i valori di default, la nullabilità e così via.

Quando una DataTable viene riempita con i dati ricavati da un database, da un file o altro, le righe della tabella saranno ricavabili dalla proprietà Rows. Ogni record o riga di dati è rappresentato da un oggetto DataRow.

Data Adapter

L'anello di congiunzione fra le classi DataSet e DataTable e un database è costituito da un oggetto DataAdapter. Per mezzo di questo sarà possibile ricavare i dati da un database, riempire degli oggetti DataTable e inserirli in un DataSet. Al contrario, una volta modificati i dati, il Data Adapter si occuperà di inviare i dati aggiornati al database.

Il framework .NET anche in questo caso contiene una classe astratta DbDataAdapter, che sarà implementata concretamente dalle classi adapter dedicate ai vari tipi di database, per esempio SqlDataAdapter, OdbcDataAdapter e OleDbDataAdapter.

Un oggetto DbDataAdapter definisce in particolare quattro proprietà, SelectCommand, InsertCommand, UpdateCommand, DeleteCommand, le quali, come i rispettivi nomi lasciano facilmente intuire, rappresentano i DbCommand utilizzati per eseguire le rispettive istruzioni SQL.

Una volta definiti questi comandi, sarà possibile utilizzare i due metodi fondamentali della classe DbDataAdapter: Fill e Update.

Il metodo Fill esegue il comando SelectCommand per ricavare i dati e riempire DataSet e DataTable.

Il metodo Update, invece, permette di inviare le modifiche al database, eseguendo il comando o i comandi necessari, a seconda della manipolazione subita dai dati in memoria.

Nel seguente esempio, il DataSet viene riempito con un oggetto DataTable contenente i record della tabella Product del database. In questo caso, il SelectCommand viene impostato direttamente per mezzo del costruttore di SqlDataAdapter:

```
string connString = "Server=(LocalDb)\\v11.0;Integrated Security=true;database=AdventureWorksLT2012";

DataSet ds = new DataSet();
using (SqlConnection conn = new SqlConnection(connString))
{
conn.Open();
SqlDataAdapter adapter = new SqlDataAdapter("SELECT * FROM SalesLT.Product", conn);
adapter.Fill(ds, "Products");
}
```

A questo punto la connessione è stata chiusa all'uscita del blocco using, e il DataSet è ancora utilizzabile perché contiene tutti i dati in memoria:

```
DataTable dt=ds.Tables[0];
Console.WriteLine("{0} record", dt.Rows.Count);
foreach (DataRow row in dt.Rows)
{
Console.WriteLine(string.Format("Name: {0} , ProductNumber: {1}",
row["Name"], row["ProductNumber"]));
}
```

La classe DataSet espone la proprietà Tables per ricavare tutti i DataTable, in questo caso uno, contenenti oggetti DataRow restituiti dalla proprietà Rows.

Per accedere ai singoli campi, si può utilizzare l'indicizzatore della classe DataRow, per mezzo del nome della colonna o dell'indice numerico. Inoltre, la proprietà Count della collezione Rows permette di conoscere il numero di risultati, cioè di righe della tabella, restituiti dall'istruzione SELECT.

L'oggetto DataTable può essere manipolato, inserendo, modificando, eliminando oggetti DataRow al suo interno. Per replicare tali modifiche sul database, è necessario impostare i comandi corrispondenti all'interno del DataAdapter.

Nel seguente esempio vediamo come gestire l'eliminazione di un record, aggiungendo un comando DELETE al DataAdapter:

```
SqlConnection cn = new SqlConnection(connString);
cn.Open();
SqlDataAdapter da = new SqlDataAdapter("SELECT * FROM DimProduct", cn);
da.Fill(ds);

SqlCommand cmdDelete = new SqlCommand();
cmdDelete.Connection = cn;
cmdDelete.CommandType = CommandType.Text;
cmdDelete.CommandText = "DELETE FROM DimProduct WHERE ProductKey=@ID";
SqlParameter paramId=new SqlParameter("@ID", SqlDbType.Int, 0, "ProductID");
paramId.SourceVersion = DataRowVersion.Original;
cmdDelete.Parameters.Add(paramId);
da.DeleteCommand = cmdDelete;
```

Per eliminare una precisa riga è necessario identificarla in maniera esatta. In questo caso viene usata un'istruzione DELETE, che a sua volta utilizza un parametro basato sulla chiave primaria della tabella, cioè ProductKey.

L'eliminazione di uno o più oggetti DataRow dalla tabella si effettua con il metodo Delete di DataRow:

```
DataTable table = ds.Tables[0];
DataRow dr = table.Rows[count - 1]; //ultimi record
dr.Delete();
```

Il metodo Update è quello che si occupa di inviare le modifiche al database, in questo caso eliminerà quindi il record corrispondente all'ultimo DataRow della tabella, costruendo l'istruzione con il relativo valore di ProductID:

```
da.Update(table);
```

In maniera analoga è possibile aggiungere al DataAdapter i comandi per le istruzioni INSERT e UPDATE.

L'oggetto CommandBuilder

Il compito di configurazione di un DataAdapter può essere abbastanza lungo e noioso, soprattutto perché bisogna anche creare e configurare i parametri adeguati per i comandi. In ogni provider dati di .NET è stata inclusa una classe che permette di velocizzare e semplificare queste attività, generando automaticamente i comandi per inserimento, modifica ed eliminazione. Anche in questo caso esiste una classe madre astratta, denominata DbCommandBuilder.

Per SQL Server possiamo quindi utilizzare una classe SqlCommandBuilder, che viene istanziata passando al costruttore un oggetto SqlDataAdapter, già istanziato con un comando SELECT:

```
SqlDataAdapter da = new SqlDataAdapter("SELECT * FROM DimSales ", cn2);
SqlCommandBuilder builder = new SqlCommandBuilder(da);
```

In tal modo, grazie ai metadati ricavati dal database, il provider è in grado di creare le istruzioni SQL necessarie e impostare le proprietà UpdateCommand, InsertCommand e DeleteCommand dell'adapter. Per curiosità potete provare a stampare i comandi generati:

```
da.UpdateCommand = builder.GetUpdateCommand();
da.InsertCommand = builder.GetInsertCommand();
Console.WriteLine("UpdateCommand: {0}",da2.UpdateCommand.CommandText);
Console.WriteLine("InsertCommand: {0}", da2.InsertCommand.CommandText);
```

Ma allora perché non utilizzare sempre un oggetto CommandBuilder? La risposta è che ci sono delle limitazioni nell'utilizzo e inoltre esso influisce sulle performance, a causa della modalità utilizzata per ricavare le caratteristiche di una tabella.

Le limitazioni principali sono invece: che esso può essere utilizzato solo a partire da SELECT che agiscono su una unica tabella, che tale tabella deve possedere una chiave primaria, e che quest'ultima deve essere inclusa nell'istruzione stessa. Quindi, per applicazioni reali, dove i comandi SQL sono in genere molto più articolati, un CommandBuilder non è di grande utilità.

Data binding e interfacce grafiche

Utilizzando Visual Studio per creare un progetto dotato di interfaccia grafica, per esempio Windows Forms, ASP.NET e così via, è possibile costruire DataSet, DataAdapter e relativi comandi in maniera del tutto visuale, senza scrivere manualmente nessuna riga di codice, ma con la possibilità di personalizzare il tutto.

Il *data binding* è il processo di connessione di una sorgente di dati a un elemento di interfaccia utente. In tal modo è possibile separare completamente l'interfaccia dai dati. Per esempio, con il data binding si può visualizzare un dato proveniente da una tabella di database in una casella di testo, e poi modificarlo nella casella stessa per ottenere l'aggiornamento della fonte originaria, il tutto in maniera trasparente.

Sviluppando applicazioni basate su sorgenti dati, vi accorgerete che .NET Framework fornisce decine di classi e API per implementare il data binding e che non potrete praticamente farne a meno per incrementare la vostra produttività.

Un esempio molto veloce prevede l'uso di un controllo DataGridView, che è una classica griglia, per visualizzare i record di una tabella.

Anche se non avete mai provato a sviluppare applicazioni di tipo Windows Forms, la procedura è abbastanza semplice e servirà a mostrarvi, appunto, come uno strumento come Visual Studio permetta di creare applicazioni pratiche con il minor sforzo possibile.

Iniziate con il creare in Visual Studio un nuovo progetto Windows Forms, chiamandolo per esempio *ProductsViewer*.

Una volta visualizzata la finestra Form1 autogenerata, trascinate su di essa un controllo DataGridView dalla toolbox e ridimensionatelo a piacimento, per esempio come mostrato in Figura 15.7.

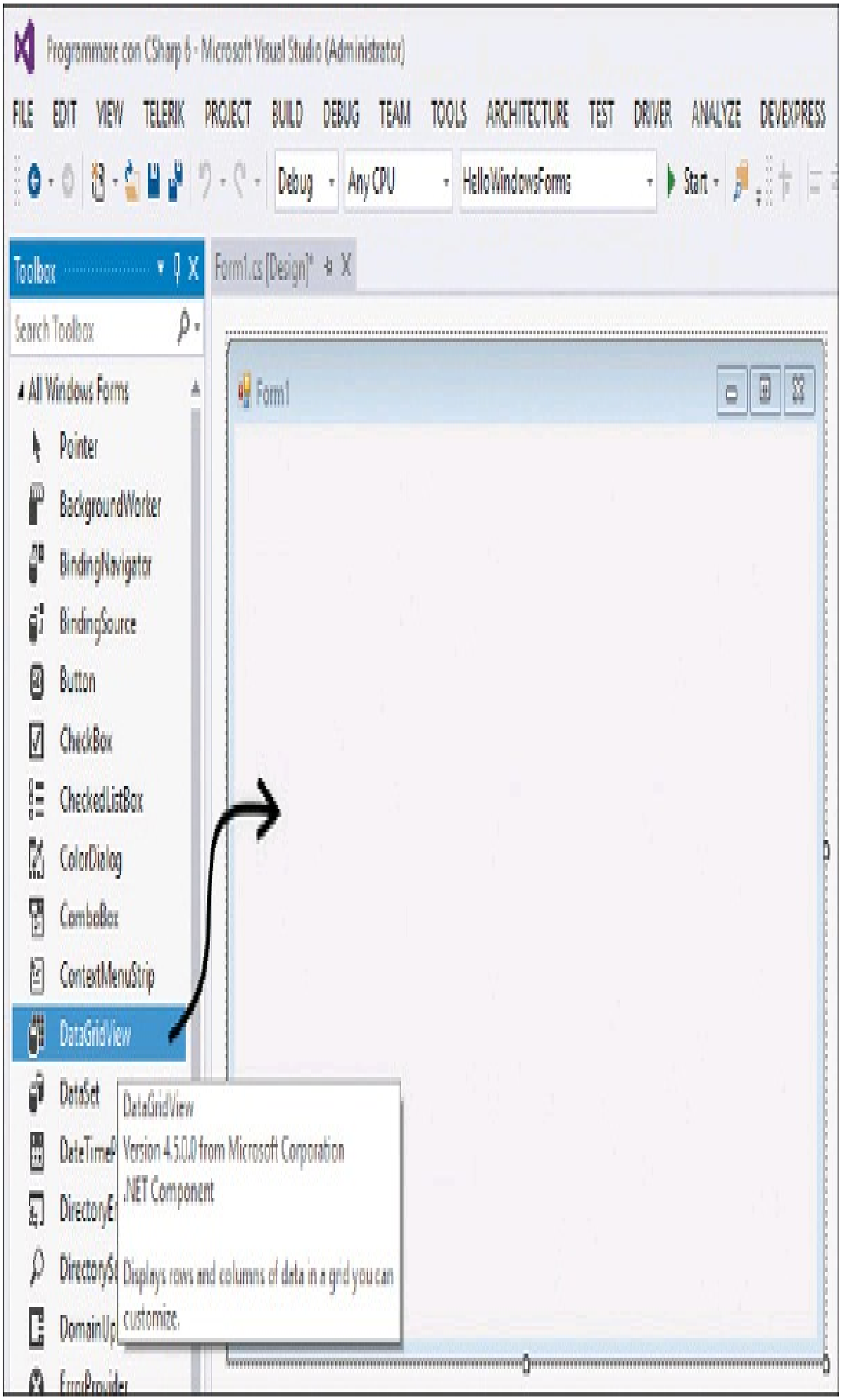

Programmare con CSharp 6 - Microsoft Visual Studio (Administrator)
FILE EDIT VIEW TELERIK PROJECT BUILD DEBUG TEAM TOOLS ARCHITECTURE TEST DRIVER ANALYZE DEVEXPRESS
Debug
Any CPU
HelloWindowsForms
Start
Toolbox
Form1.cs [Design]*
Search Toolbox
All Windows Forms
Pointer
BackgroundWorker
BindingNavigator
BindingSource
Button
CheckBox
CheckedListBox
ColorDialog
ComboBox
ContextMenuStrip
DataGridView
DataSet
DataGridView
Version 4.5.0.0 from Microsoft Corporation
.NET Component
Displays rows and columns of data in a grid you can customize.
Form1

Figura 15.7 – Creazione di un controllo DataGridView in Visual Studio.

A questo punto, fate clic sulla piccola freccia visualizzata in alto a destra sul controllo DataGridView (vedi Figura 15.8), che è detto *Smart tag*, e che sarà un compagno costante nelle vostre sessioni di sviluppo all'interno di Visual Studio.


GridViewTasks
Choose Data Source:
(none)
Edit Columns...
Add Columns...
Enable Adding
Enable Editing
Enable Deleting
Enable Column Reordering
Dock in Parent Container

Figura 15.8 – Smart tag per la configurazione di un DataGridView.

Così facendo apparirà una finestra in cui sarà possibile avviare vari comandi relativi in questo caso al controllo DataGridView, fra cui scegliere la sorgente dati o crearne una nuova.

Fate clic quindi sulla casella Choose Data Source e poi sul link Add Project Data Source. In questo modo, apparirà la finestra iniziale della procedura di configurazione della sorgente dati, che può essere un database, una servizio remoto, una normale collezione di oggetti o altro dipendente dalla nostra installazione (vedi Figura 15.9).

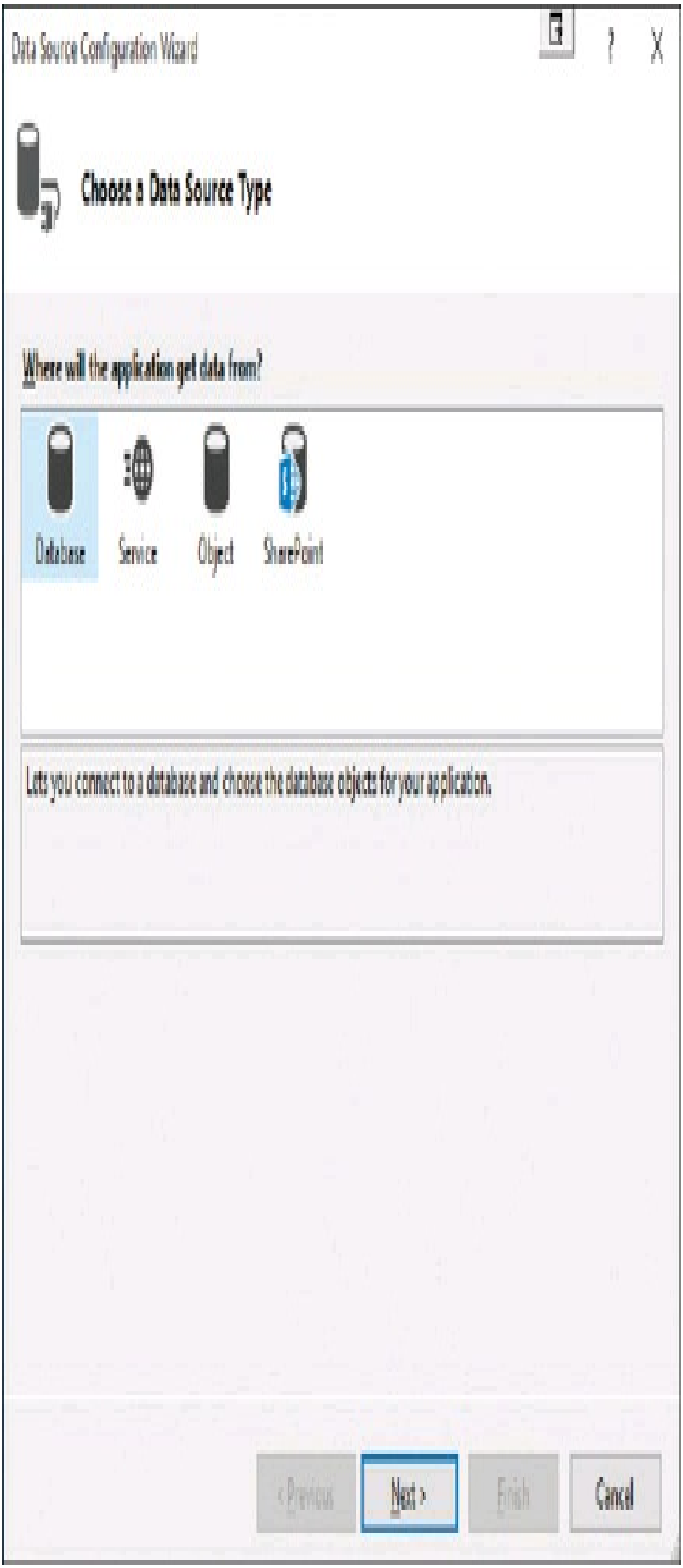

Data Source Configuration Wizard
Choose a Data Source Type
Where will the application get data from?
Database
Service
Object
SharePoint
Lets you connect to a database and choose the database objects for your application.
< Previous
Next >
Finish
Cancel

Figura 15.9 – Wizard di configurazione di una sorgente dati.

Per il nostro esempio selezionate Database e andate avanti. Nel passo successivo confermate di voler utilizzare un *DataSet* come modello di dati e andate ancora avanti.

La schermata successiva permette di configurare la connessione, inserendo i dati di accesso al vostro database SQL Server e mostrandovi automaticamente la stringa di connessione che verrà utilizzata. Se già in passato avete configurato una connessione, essa verrà mostrata nella casella di selezione, altrimenti il pulsante New Connection vi consentirà di crearne una *ex novo*.

Cliccate su Next, nel passo seguente lasciate tutto com'è, per aggiungere la stringa di connessione al file app.config di configurazione dell'applicazione, e andate avanti.

Finalmente si arriva al passo che permette di scegliere da quale entità del database ricavare i dati da visualizzare nel DataGridView. Selezionate il ramo Tables e poi, per esempio, la tabella DimProduct del database AdventureWorksDW2014 (vedi Figura 15.10).

Si noti come sia possibile anche selezionare solo i campi che interessano (nel nostro caso abbiamo selezionato l'intera tabella) e impostare un nome personalizzato per il DataSet che verrà creato.

Data Source Configuration Wizard

Choose Your Database Objects

Which database objects do you want in your dataset?

- Tables
  - AdventureWorksDWBuildVersion
  - DatabaseLog
  - DimAccount
  - DimCurrency
  - DimCustomer
  - DimDate
  - DimDepartmentGroup
  - DimEmployee
  - DimGeography
  - DimOrganization
  - DimProduct
  - DimProductCategory
  - DimProductSubcategory
  - DimPromotion
  - DimReseller

DataSet name:

AdventureWorksDW2014DataSet

< Previous | Next > | Finish | Cancel

Figura 15.10 – Selezione di una tabella come sorgente dati.

Dopo aver cliccato su Finish, Visual Studio creerà automaticamente un DataSet, il relativo DataAdapter e un oggetto di tipo BindingSource, che verrà utilizzato come sorgente dati vera e propria del DataGridView. Inoltre, notate come a questo punto il controllo griglia sia stato automaticamente configurato con le colonne corrispondenti a quelle della tabella scelta.

Avviate il progetto e vedrete la vostra griglia riempita con i dati ricavati dal database, il tutto senza aver scritto letteralmente nessuna istruzione di codice C#.

Per capire come funziona il tutto, potete anche provare a dare un'occhiata al codice che Visual Studio ha generato per noi. In parole povere, è stato creato un DataSet tipizzato, cioè una classe derivata da DataSet, contenente a sua volta un DataTable anch'esso tipizzato, con le colonne corrispondenti a quelle della tabella, e tutti i metodi e comandi necessari.

LINQ to SQL

LINQ to SQL è un provider LINQ dedicato all'accesso a database SQL Server, che permette quindi, usando la sintassi e gli operatori di LINQ, di eseguire interrogazioni e ricavare dati senza la necessità di imparare e impiegare un linguaggio dedicato come SQL.

Le funzionalità di LINQ to SQL sono realizzate creando una mappatura fra il modello del database e un modello a oggetti espresso in un linguaggio come C#.

Eseguendo l'applicazione, l'infrastruttura che sta dietro a questa tecnologia si occuperà di convertire le query scritte utilizzando la sintassi LINQ e i suoi operatori in istruzioni in linguaggio SQL, inviandole direttamente al database per l'esecuzione.

La mappatura suddetta viene creata scrivendo delle normali classi C#, che utilizzano particolari attributi, ognuna della quali rappresenta una tabella del database. Le colonne di tali tabelle a loro volta sono mappate mediante proprietà pubbliche, anch'esse marcate da attributi.

Come vedremo a breve, tale processo di scrittura delle classi può essere semplificato in maniera notevole, se non totalmente automatizzato, utilizzando gli appositi strumenti messi a disposizione da Visual Studio.

Modello a oggetti

Il primo passo nell'uso di LINQ to SQL per l'accesso a un database SQL Server consiste nel definire le classi che rappresentano le corrispondenti tabelle e relative colonne. Tale processo può avvenire manualmente o in maniera automatizzata in Visual Studio.

Riprendendo ancora il database AdventureWorks sin qui utilizzato, supponiamo di volere accedere ai dati contenuti nella tabella DimProduct.

NOTA	Per utilizzare LINQ to SQL è necessario aggiungere al proprio progetto un riferimento all'assembly System.Data.Linq.dll.

La classe seguente può rappresentare la tabella DimProduct e alcune delle sue colonne:

```
[Table(Name="DimProduct")]
class Product
{
[Column(IsPrimaryKey=true)]
public int ProductKey { get; set; }
[Column]
public string EnglishProductName { get; set; }
[Column]
public string Color { get; set; }
}
```

La classe è decorata mediante l'attributo Table, a indicare che essa rappresenta una tabella del database. Come comportamento predefinito, il nome della classe corrisponderà al nome della tabella, ma è possibile, utilizzando il parametro Name, indicare esplicitamente tale nome, come in questo caso in cui la classe Product rappresenta la tabella DimProduct.

Le proprietà pubbliche a loro volta sono decorate dall'attributo Column. Anche qui è possibile indicare il nome del campo reale mediante il parametro Name oppure, come nel caso della colonna ProductID, che si tratta della chiave primaria della tabella.

Altri parametri permettono di personalizzare e controllare più nel dettaglio il processo di mapping delle colonne.

Contesto dei dati

Una volta eseguito il mapping delle tabelle con le relative classi, è possibile stabilire la connessione fra il modello a oggetti, scritto nel mondo di C#, e il database vero e proprio.

Tale possibilità è resa possibile dalla classe DataContext, che si occuperà della comunicazione vera e propria, permettendo così di ricavare i dati costruendo gli oggetti, e di inviare al database quelli modificati trasformandoli in istruzioni SQL.

Il primo passo è quello di istanziare un oggetto DataContext, utilizzando per esempio una stringa di connessione come quella vista nei paragrafi precedenti.

```
string connString = "Server=(LocalDb)\\MSSQLLocalDB;Integrated Security=true;database=AdventureWorksDW2014";
DataContext context = new DataContext(connString);
```

A questo punto leggere un insieme di record dalla tabella Product è immediato. Innanzitutto si ottiene un oggetto Queryable invocando il metodo generico GetTable, per esempio:

```
var tableProd = context.GetTable<Product>();
var products=from prod in tableProd
where prod.Color=="Black"
select prod;
foreach(Product p in products)
{
Console.WriteLine(p.Name);
}
```

Tipicamente, come vedremo fra qualche paragrafo, si utilizzano degli oggetti DataContext tipizzati, derivati dalla classe DataContext vista sopra, e che Visual Studio si occupa di generare per noi. Essi evitano, per esempio, di utilizzare il metodo generico GetTable e implementano delle proprietà e dei metodi appositi che restituiscono le varie tipologie di entità.

La classe DataContext si occupa anche del processo inverso, cioè inviare al database le modifiche eseguite sulle entità ricavate in precedenza, invocando il metodo SubmitChanges.

Per esempio, ecco come modificare e salvare il colore del primo oggetto Product ottenuto dalla query precedente:

```
Product product = products.First();
product.Color = "Red";
context.SubmitChanges();
```

Generazione automatica del modello

In Visual Studio è disponibile un apposito strumento di generazione del modello a oggetti di un database, da utilizzare con LINQ to SQL. Tale procedura è certamente da preferire a quella vista nei precedenti paragrafi, in quanto permette di risparmiare tempo ed evitare possibili errori di scrittura manuale o dimenticanze.

Per generare il modello di un database, aggiungete un nuovo elemento a un progetto esistente e selezionate il tipo LINQ to SQL Classes, all'interno della categoria Data (vedi Figura 15.11).

Add New Item - LinqToSql

Installed

Sort by: Default

Search Installed Templates (Ctrl+E)

- Visual C# Items
  - Code
  - Data
  - General
  - Web
  - Windows Forms
  - WPF
  - Extensibility
  - SQL Server
  - Workflow
- Graphics
- Online

ADO.NET Entity Data Model	Visual C# Items
DataSet	Visual C# Items
EF 5.x DbContext Generator	Visual C# Items
EF 6.x DbContext Generator	Visual C# Items
LINQ to SQL Classes	Visual C# Items
Service-based Database	Visual C# Items
XML File	Visual C# Items
XML Schema	Visual C# Items
XSLT File	Visual C# Items

Type: Visual C# Items

LINQ to SQL classes mapped to relational objects.

Click here to go online and find templates.

Name: AdventureWorks.dbml

Add Cancel

Figura 15.11 – Selezione di una tabella come sorgente dati.

Date un nome al modello, che verrà salvato in un file con estensione .dbml, e cliccate sul pulsante Add. A questo punto potrete aggiungere al modello le tabelle del database cui siamo interessati, trascinandole dal Server Explorer o dall'SQL Server Object Explorer direttamente all'interno della superficie del designer. Il diagramma così generato mostra anche le relazioni fra le tabelle e le varie proprietà delle colonne ricavate.

Se non avete ancora eseguito la connessione al database, potete aggiungerne una nuova come visto in precedenza.

Una volta scelte e trascinate le tabelle sul designer, Visual Studio si occuperà di generare per noi tutto il codice necessario: le classi di mapping, comprensive di proprietà corrispondenti alle colonne e di proprietà quelle corrispondenti a eventuali relazioni, e la classe DataContext tipizzata, che assumerà un nome corrispondente a quello scelto per il modello, in questo caso AdventureDataContext.

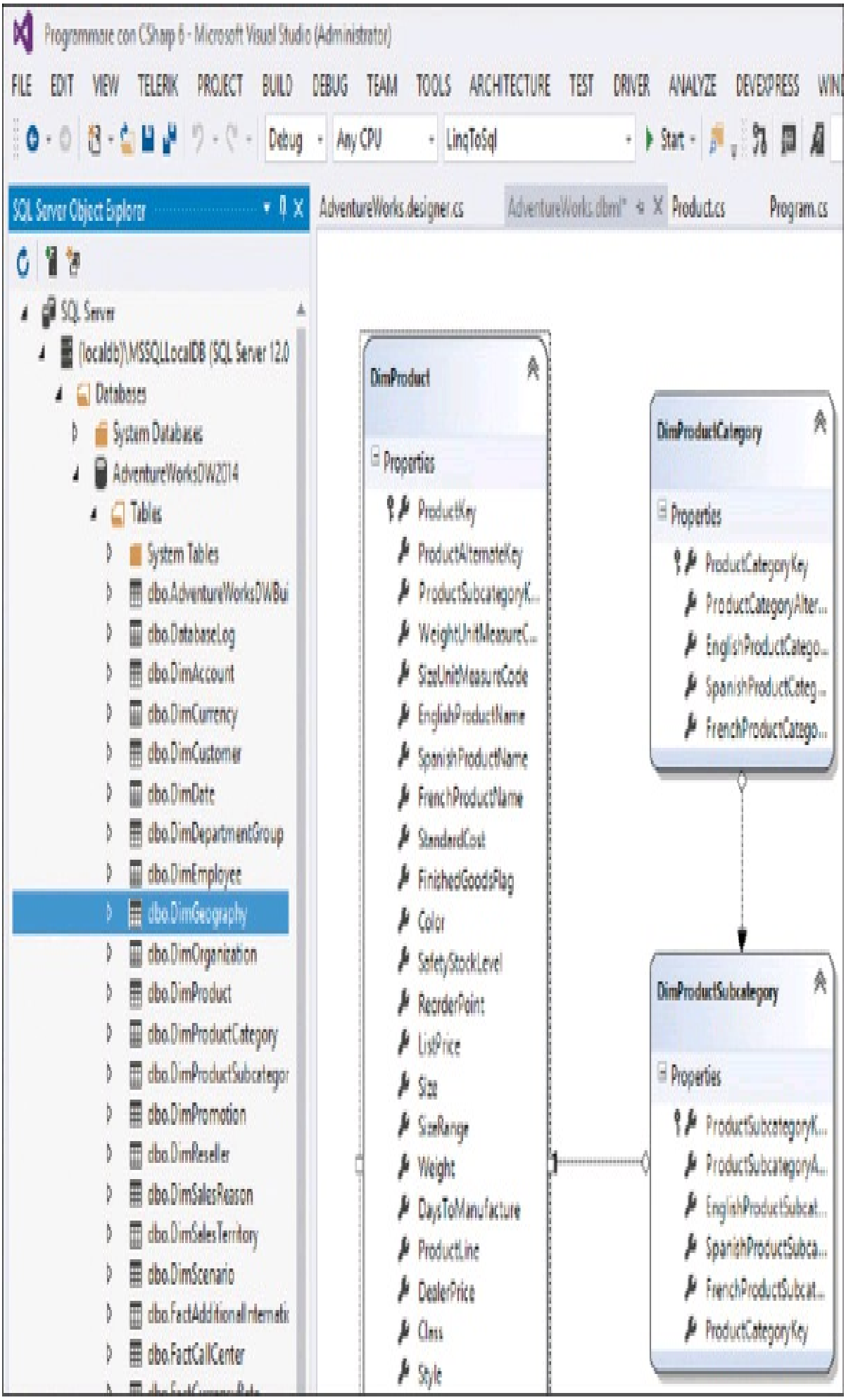

Programmare con CSharp 6 - Microsoft Visual Studio (Administrator)
FILE EDIT VIEW TELERIK PROJECT BUILD DEBUG TEAM TOOLS ARCHITECTURE TEST DRIVER ANALYZE DEVEXPRESS
Debug
Any CPU
LinqToSql
Start
SQL Server Object Explorer
AdventureWorks.designer.cs
AdventureWorks.dbml*
Product.cs
Program.cs
SQL Server
(localdb)\MSSQLLocalDB (SQL Server 12.0
Databases
System Databases
AdventureWorksDW2014
Tables
System Tables
dbo.AdventureWorksDWBui
dbo.DatabaseLog
dbo.DimAccount
dbo.DimCurrency
dbo.DimCustomer
dbo.DimDate
dbo.DimDepartmentGroup
dbo.DimEmployee
dbo.DimGeography
dbo.DimOrganization
dbo.DimProduct
dbo.DimProductCategory
dbo.DimProductSubcategor
dbo.DimPromotion
dbo.DimReseller
dbo.DimSalesReason
dbo.DimSalesTerritory
dbo.DimScenario
dbo.FactAdditionalInternatio
dbo.FactCallCenter
DimProduct
Properties
ProductKey
ProductAlternateKey
ProductSubcategoryK...
WeightUnitMeasureC...
SizeUnitMeasureCode
EnglishProductName
SpanishProductName
FrenchProductName
StandardCost
FinishedGoodsFlag
Color
SafetyStockLevel
ReorderPoint
ListPrice
Size
SizeRange
Weight
DaysToManufacture
ProductLine
DealerPrice
Class
Style
DimProductCategory
Properties
ProductCategoryKey
ProductCategoryAlter...
EnglishProductCatego...
SpanishProductCateg...
FrenchProductCatego...
DimProductSubcategory
Properties
ProductSubcategoryK...
ProductSubcategoryA...
EnglishProductSubcat...
SpanishProductSubca...
FrenchProductSubcat...
ProductCategoryKey

Figura 15.12 – Modello dbml generato da Visual Studio.

Ora è possibile usare il codice autogenerato in maniera analoga a quanto visto con il modello a oggetti che abbiamo creato manualmente. Per esempio, per ottenere tutte le categorie di prodotti contenuti nel database all'interno della tabella ProductCategory, basterà leggere la proprietà ProductCategories della classe AdventureDataContext:

```
AdventureWorksDataContext awdc = new AdventureWorksDataContext(connString);
var categories= awdc.ProductCategories;
```

Il modello generato tiene in considerazione, come detto, anche le relazioni fra le tabelle. Per esempio, ogni entità DimProductSubcategory possiede una proprietà DimProducts, marcata con l'attributo AssociationAttribute, che automaticamente restituisce la collezione di prodotti di una determinata categoria.

L'esempio seguente stampa tutte le categorie, sottocategorie e i relativi prodotti in esse contenuti:

```
foreach (DimProductCategory pc in categories)
{
Console.WriteLine(pc.EnglishProductCategoryName);
foreach (var sub in pc.DimProductSubcategories)
{
Console.WriteLine(sub.EnglishProductSubcategoryName);
foreach (var pd in sub.DimProducts)
Console.WriteLine(" {0}", pd.EnglishProductName);
}
}
```

Naturalmente, è possibile utilizzare la sintassi e gli operatori LINQ per eseguire query più o meno articolate. Di seguito vengono stampati tutti i prodotti della categoria "Bikes":

```
var query = from cat in categories
from sub in cat.DimProductSubcategories
from prod in sub.DimProducts
where cat.EnglishProductCategoryName.Contains("Bikes")
select prod;
foreach(var p in query)
{
Console.WriteLine("{0}", p.Name);
}
```

LINQ to SQL tiene traccia anche delle modifiche subite dalle entità e consente di inviarle al database di origine invocando il metodo SubmitChanges del DataContext:

```
awdc.DimProducts.First().Color = "Dark Red";
awdc.SubmitChanges();
```

Inoltre, è possibile inserire nuovi record o eliminarli, utilizzando i metodi InsertOnSubmit e DeleteOnSubmit della classe Table<T>:

```
DimProductCategory newCat = new DimProductCategory();
newCat.EnglishProductCategoryName = "New Category 1";
newCat.SpanishProductCategoryName = "Nueva Categoria 1";
newCat.FrenchProductCategoryName = "Nouvelle Categorie 1";
awdc.DimProductCategories.InsertOnSubmit(newCat);
awdc.SubmitChanges();
```

Entity Framework

ADO.NET Entity Framework, che quasi sempre è abbreviato in *Entity Framework*, è essenzialmente un Object Relational Mapper (ORM), cioè uno strumento che consente di creare un modello a oggetti che rappresenta un database. Tale modello si occupa di tradurre delle operazioni eseguite nel mondo degli oggetti (cioè utilizzando proprietà, metodi e spesso LINQ) in query eseguite sul database, quindi in linguaggio SQL, ottenendo dei risultati che a loro volta verranno restituiti nuovamente come oggetti.

In parole povere, Entity Framework (o un ORM in generale) permette agli sviluppatori di lavorare con dati nel contesto del dominio applicativo, per esempio Clienti, Ordini, Prodotti, senza dovere possedere alcuna nozione del database sottostante composto da tabelle e colonne.

NOTA	Esistono diversi ORM per .NET Framework, sia gratuiti sia commerciali, ognuno con proprie peculiarità. LINQ to SQL ed Entity Framework fanno parte dell'approccio scelto da Microsoft, ma potreste trovare più produttivo qualche altro tool, o avere necessità di utilizzare anche strumenti differenti per i vostri progetti. Nhibernate, DataObjects.Net, Open Access, XPO, LLBLGen Pro: sono solo alcuni dei nomi che incontrereste facendo una ricerca in tal senso.

Al primo approccio con Entity Framework, non si può far a meno di notare delle somiglianze o sovrapposizioni con la tecnica utilizzata dal provider LINQ to SQL che abbiamo già visto in precedenza, anche perché spesso verranno utilizzati la sintassi e gli operatori di LINQ per lavorare sugli oggetti corrispondenti alle entità del database.

In realtà, LINQ to SQL e Entity Framework sono due cose distinte e separate; probabilmente ciascuno presenta vantaggi rispetto all'altro, la cui analisi e conoscenza ci permetterà di scegliere il primo o il secondo in base alle necessità della propria applicazione.

LINQ to SQL, introdotto con .NET 3.5, è un provider LINQ utilizzabile esclusivamente con database Microsoft SQL Server, che permette di iniziare a lavorare in maniera molto immediata con un database, mappando in modo praticamente automatico ogni entità del database su una classe .NET.

Entity Framework, dal canto suo, è un motore esterno a LINQ, che permette un approccio molto più elaborato, per esempio consentendo di mappare su una singola classe due tabelle del database o viceversa; inoltre EF non è limitato a SQL Server ma è genericamente utilizzabile con database di differenti tipologie.

Per sfruttare LINQ al di sopra di Entity Framework ci si servirà di un LINQ Provider apposito, chiamato *LINQ to Entities*.

La prima versione di Entity Framework, detta anche EF 3.5, fu rilasciata all'interno di .NET 3.5 SP1; forniva un supporto ORM basilare usando un approccio di tipo Database First, cioè a partire da un database esistente era possibile generarne il relativo modello a oggetti.

Visual Studio 2015 include la versione 6.1.3, mentre dalla versione 7, come parte dello sforzo di Microsoft di abbracciare il mondo open source, è disponibile il codice sorgente anche su GitHub, all'indirizzo https://github.com/aspnet/EntityFramework.

Architettura di Entity Framework

L'architettura di Entity Framework è molto articolata ed è composta da diversi componenti. La Figura 15.13 mostra una visione di alto livello di tale struttura, che si andrà ora a spiegare nel dettaglio.


Entity Data Model
Conceptual Model
MAPPING
Storage Model
LINQ To Entities
Entity SQL
IEnumerable <T>
Object Services
Entity Client Data Provider
ADO.NET Data Provider
Database

Figura 15.13 – Architettura di Entity Framework.

L'*Entity Data Model* consiste di tre parti principali:

- il *conceptual model*, o *modello concettuale*, è il modello delle classi di dominio e delle relative relazioni fra di esse, indipendente dalle tabelle del database sottostante;
- lo *storage model*, o *modello di archiviazione* (chiamato anche *modello logico*), è il modello del database che include invece tabelle, campi, viste, stored procedure, relazioni, chiavi e tutto ciò che compone il database;
- il *mapping* consiste delle informazioni che permettono la *mappatura* fra le classi e il database.

Lo storage model e il mapping possono essere modificati in base alle esigenze applicative, senza che sia necessario modificare il modello concettuale, le classi di dati o il codice dell'applicazione. I tre componenti sono memorizzati e gestiti da Entity Framework mediante l'uso di tre file in formato XML, che Visual Studio permette di manipolare in buona parte mediante un wizard e un designer visuale, che vedremo più avanti. Per il momento vi basti sapere che esso genera un unico file con estensione .edmx, dal quale a runtime vengono ricavati i tre file suddetti.

Questi file del modello e di mapping vengono quindi utilizzati da Entity Framework per trasformare operazioni di creazione, lettura, aggiornamento ed eliminazione di entità e relazioni del modello concettuale, in operazioni equivalenti nell'origine dati, cioè direttamente sul database.

LINQ To Entities è il provider che fornisce il supporto per eseguire query LINQ sulle entità definite nel modello concettuale.

Entity SQL invece è un sotto-linguaggio SQL indipendente dal database fisico, che ha le stesse potenzialità e funzioni di LINQ to Entities, ma permette di scrivere le query direttamente in formato testuale. Quindi, con il vantaggio di poter creare dinamicamente le query, ma con lo svantaggio di dover imparare un nuovo linguaggio.

NOTA	L'esistenza di Entity SQL deriva dal fatto che, quando si è iniziato a sviluppare Entity Framework, non esisteva ancora LINQ e quindi si necessitava di un linguaggio apposito. Senza dubbio, LINQ to Entities è più semplice da utilizzare ed è da preferire a Entity SQL.

Il componente *Object Service* rappresenta il punto di ingresso principale per l'accesso ai dati, in quanto si occupa di inviare e convertire in classi dati dal formato restituito dallo strato

sottostante. Queste funzionalità sono implementate dalla classe ObjectQuery<T>, che si occupa di effettuare qualsiasi query.

Inoltre, l'Object Service tiene traccia dello stato degli oggetti, in maniera da poter capire in ogni momento quali di essi sono stati creati, modificati, o eliminati, e di conseguenza quali sono le operazioni da eseguire sul database.

Lo strato *Entity Client Data Provider* si occupa di convertire le query LINQ to Entities o Entity SQL nel formato SQL specifico del database, utilizzando il data provider ADO.NET sottostante che, come ormai ben sappiamo, è lo strato di comunicazione con il database.

Nei prossimi paragrafi inizieremo con il creare un Entity Data Model a partire da un database SQL Server esistente, per esempio il già utilizzato Adventure Works.

Workflow di sviluppo

Entity Framework, nella versione attuale, consente di utilizzare tre diversi approcci di modellazione e sviluppo, che lo sviluppatore può scegliere in base allo stato del database o del progetto.

Il modello *Code First* evita di utilizzare il designer visuale EDMX e consiste nello scrivere manualmente le proprie classi che rappresentano le entità, dette anche classi *POCO* (*Plain Old CLR Object*), e quindi creare il database a partire da queste classi, o mapparle su un database esistente.

Tale approccio, introdotto con EF 4.1, è comunemente utilizzato quando si ha un'applicazione esistente che è modellata con una struttura simile a un database, oppure quando si utilizza un processo di sviluppo Domain Driven. In questo caso, quindi, le classi create in C# potrebbero anche essere utilizzate senza necessariamente dover ricorrere a un database, ma mantenendo eventualmente i dati in memoria o, se proprio si necessità di renderli persistenti, salvandoli su un file (per esempio, in formato XML).

Il workflow *Model First*, introdotto in EF 4, permette di creare applicazioni in cui è richiesto il supporto di un database, senza necessità di dover utilizzare degli strumenti per la sua progettazione, come SQL Management Studio. Il modello viene creato all'interno di Visual Studio, in maniera visuale, e questo verrà poi utilizzato per generare il database finale.

Per concludere, l'approccio denominato *Database First* è quello per cui originariamente era necessario, o comunque utile, usare Entity Framework: a partire da un database esistente, o progettato in anticipo, si procede allo sviluppo di un'applicazione che utilizzi i dati in esso memorizzati. Grazie al modello e alle classi generate da Entity Framework, si nasconde la

complessità intrinseca del database e lo sviluppatore non deve necessariamente conoscere nei dettagli la sua struttura.

Creazione di un Entity Data Model (Database First)

Se si desidera sviluppare un'applicazione che sfrutti un database esistente e i relativi dati in esso contenuti, il primo passo consiste nel creare il modello di tale database e generare le classi da utilizzare nel nostro codice. Si tratta quindi dell'approccio che abbiamo chiamato *Database First*.

Creiamo allora un nuovo progetto Visual Studio, anche di tipo Console Application, e procediamo a creare l'Entity Data Model.

Facendo clic con il pulsante destro sul progetto nel Solution Explorer, selezionate al solito la voce Add e quindi New Item.

Nella finestra di selezione del template, all'interno della categoria Data, scegliete il tipo ADO.NET Entity Data Model, assegnate un nome, per esempio AdventureWorksModel.edmx, quindi fate clic sul pulsante Add. A questo punto verrà avviato il wizard di creazione del modello.

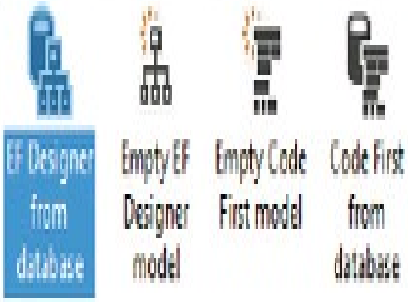
Nel primo passo della procedura (vedi Figura 15.14) saranno presenti quattro opzioni: *EF Designer from database* per usare l'approccio Database First, *Empty EF Designer model* per il Model First, e infine *Empty Code First model* e *Code First from database* per l'approccio Code First.

Procediamo con la prima scelta, visto che utilizzeremo il database Adventure Works creato in precedenza, e clicchiamo su Next. Nel passo successivo verrà richiesta la creazione o la selezione di una connessione al database; fate clic sul pulsante New Connection e inserite i parametri di connessione al database SQL Server. Tali parametri verranno conservati nel file di configurazione dell'applicazione, insieme al provider dati ADO.NET da utilizzare, che in questo caso è il provider SQLClient già usato in precedenza.

Entity Data Model Wizard

Choose Model Contents

What should the model contain?



Creates a model in the EF Designer based on an existing database. You can choose the database connection, settings for the model, and database objects to include in the model. The classes your application will interact with are generated from the model.

< Previous | Next > | Finish | Cancel

Figura 15.14 – Selezione della modalità di creazione dell'Entity Data Model.

Nel passo seguente è necessario scegliere la versione di Entity Framework da utilizzare: selezionate la 6.x. Si noti che questo step potrebbe anche non apparire se Entity Framework è stato installato nel progetto tramite pacchetto NuGet.

Entity Data Model Wizard

Choose Your Database Objects and Settings

Which database objects do you want to include in your model?

- Tables
- Views
- Stored Procedures and Functions

Pluralize or singularize generated object names

Include foreign key columns in the model

Import selected stored procedures and functions into the entity model

Model Namespace:

AdventureWorksDW2014Model

< Previous | Next > | Finish | Cancel

Figura 15.15 – Selezione della modalità di creazione dell'Entity Data Model.

Infine, al successivo passo, avviene la selezione degli oggetti del database da aggiungere al modello e che quindi si vogliono mappare sulle classi che verranno generate.

Selezionate tutte le caselle Tables, Views e Stored Procedures, in maniera da creare un modello completo dell'intero contenuto del database, e personalizzate, se desiderate, il namespace che conterrà le classi. Le altre impostazioni possono essere mantenute come proposto. Fate poi clic sul pulsante Finish.

Il wizard a questo punto verrà completato e in pochi secondi creerà il modello, le classi e i vari file di mapping, inglobandoli all'interno di un file EDMX. Dopo il completamento, si aprirà il designer visuale contenente il diagramma delle entità generate a partire dal database.

Programmare con CSharp 6 - Microsoft Visual Studio (Administrator)

FILE EDIT VIEW TEAM PROJECT BUILD DEBUG TEAM TOOLS ARCHITECTURE TEST DRIVER ANALYZE DEVEXPRESS WINDOW HELP

Server Explorer

Azure

Data Connections

Servers

SharePoint Connections

AdventureWorksMo...Edmx [Diagram]* | Program.cs | AdventureWorks.dbml | AdventureWorks.designer.cs | Product.cs | Program.cs

Solution Explorer

Mapping Details - DimProduct

Column	Operator	Value / Property
Tables		
Maps to DimProduct		
<Add a Condition>		
Column Mappings		
ProductKey : int	=	ProductKey : Int32
ProductAlternateKey : nvarchar	=	ProductAlternateKey : String
ProductSubcategoryKey : int	=	ProductSubcategoryKey : Int32
WeightUnitMeasureCode : nchar	=	WeightUnitMeasureCode : String
SizeUnitMeasureCode : nchar	=	SizeUnitMeasureCode : String
EnglishProductName : nvarchar	=	EnglishProductName : String
SpanishProductName : nvarchar	=	SpanishProductName : String

Properties

AdventureWorksDW2014Model.DimProduct Er

Abstract	False
Access	Public
Base Type	(None)
Documentation	
Entity Set Name	DimProduct
Fill Color	R: 122; 204
Name	DimProduct

Ready

Figura 15.16 – Il modello nell'Entity Framework Designer di Visual Studio.

All'interno del Solution Explorer potete ora espandere il file AdventureWorksModel.edmx, per analizzare tutti i file creati dalla procedura precedente.

Si noti fra gli altri la presenza dei vari file corrispondenti alle tabelle del database: DimProduct.cs, DimCustomer.cs e così via.

Utilizzo delle entità

Ora che abbiamo un modello, è possibile usarne le classi per accedere ai dati del database senza la necessità di conoscere nulla di SQL o della struttura del database stesso.

Lavorando con Entity Framework, la maggior parte delle classi necessarie risiedono all'interno del namespace System.Data.Entity.

La classe AdventureWorksDW2014Entities è stata generata derivandola da DbContext e rappresenta il punto di ingresso alle varie parti del database. La classe DbContext è un wrapper della classe ObjectContext ed espone le funzionalità più comuni di quest'ultima usando un'interfaccia più semplice e intuitiva; in definitiva, è quello che permette allo sviluppatore di interagire con lo strato Object Services.

In genere non bisogna nemmeno preoccuparsi di impostare manualmente una stringa di connessione, in quanto essa viene ricavata direttamente nel costruttore dal file app.config dell'applicazione, utilizzando il nome assegnato alla stringa stessa.

È sempre possibile variarla direttamente nel file di configurazione, oppure utilizzare un costruttore diverso di DbContext cui passare la stringa di connessione personalizzata.

Possiamo quindi creare e utilizzare un DbContext, istanziando direttamente la classe derivata:

```
using(AdventureWorksDW2014Entities db=new AdventureWorksDW2014Entities ())
{
...
}
```

Per selezionare i prodotti, cioè i record contenuti nella tabella DimProduct, basta leggere la proprietà omonima, scrivendo nel metodo Main dell'applicazione console le seguenti istruzioni:

```
using(AdventureWorksDW2014Entities db=new AdventureWorksDW2014Entities ())
{
var query = from prod in db.DimProduct
select prod;

foreach (DimProduct p in query)
{
Console.WriteLine($"{p.ProductKey}, {p.ModelName}");
}
}
```

In questo caso abbiamo usato la sintassi LINQ per scrivere la query che accede alla proprietà DimProduct. Tale proprietà restituisce una collezione DbSet<DimProduct> contenente tutti i record della tabella, senza necessità di scrivere direttamente una query SQL. Entity Framework si occupa di tutto dietro le quinte.

Una qualsiasi entità può anche essere ricavata specificando direttamente la sua chiave primaria, utilizzando il metodo Find della classe DbSet. L'entità cercata potrebbe già essere stata caricata nel contesto attuale; in tal caso verrebbe restituita direttamente da questo. Altrimenti, una query verrà eseguita sul database.

Ecco un esempio di utilizzo del metodo Find:

```
using(AdventureWorksDW2014Entities db = new AdventureWorksDW2014Entities ())
{
var product1 = db.DimProduct.Find(1); //ricerca il Product con chiave uguale a 1
}
```

Il framework si occupa di creare le proprietà necessarie a rappresentare le relazioni fra le tabelle. Facendo un esempio, la tabella DimProduct è correlata alla tabella DimProductSubcategory e, nel mondo di C#, la classe DimProduct espone una proprietà DimProductSubcategory che rappresenta questa relazione. Quindi, è possibile leggere tale proprietà e, a sua volta, la proprietà che restituisce il nome della categoria del prodotto:

```
string categoryName=p.DimProductSubcategory. EnglishProductSubcategoryName;
```

La seguente query esegue una join fra le due tabelle dei prodotti e delle categorie a cui questi appartengono, filtrando le categorie che contengono il termine "bike" nel nome, e stampa poi il nome di categoria e relativo prodotto:

```
var query2 = from cat in db.DimProductCategory
join prod in db.DimProduct on cat.ProductCategoryKey equals prod.ProductSubcategoryKey
where cat.EnglishProductCategoryName.ToLower().Contains("bike")
select prod;
foreach (var product in query2)
{
Console.WriteLine($"{product.DimProductSubcategory.EnglishProductSubcategoryName}, {product.ModelName}");
}
```

La precedente query LINQ, utilizzando le proprietà delle entità coinvolte, realizza in maniera molto semplice quella che in SQL avrebbe richiesto una join fra le tabelle.

L'inserimento di nuovi record è altrettanto semplice e immediato, basta usare il metodo Add:

```
DimProductCategory cat1 = new DimProductCategory();
cat1.EnglishProductCategoryName = "new category";
cat1.FrenchProductCategoryName = "nouvelle catégorie";
cat1.SpanishProductCategoryName = "nueva categoría";
using (AdventureWorksDW2014Entities dba = new AdventureWorksDW2014Entities())
{
```

```
dba.DimProductCategory.Add(cat1);
dba.SaveChanges();
}
```

Lo stesso vale per le modifiche e le eliminazioni: basta agire sugli oggetti esistenti, o rimuoverli dalle collezioni esistenti, e poi invocare ancora il metodo SaveChanges.

L'infrastruttura di Entity Framework si occupa di tenere traccia di ogni modifica e poi inviare al database le istruzioni necessarie.

Eseguire query SQL

La classe DbContext permette, qualora fosse necessario, di eseguire anche delle query in puro linguaggio SQL per ricavare risultati dal database, per eliminare e modificare dati.

Per esempio, per ricavare delle entità si può utilizzare il metodo SqlQuery dell'oggetto DbSet<T> restituito dalle proprietà corrispondenti a ogni tipologia di tabella:

```
using(AdventureWorksLT2012Entities dbquery = new AdventureWorksLT2012Entities())
{
var products = dbquery.Product.SqlQuery("select * from SalesLT.Product").ToList();
}
```

In questo caso, il metodo SqlQuery restituisce un oggetto DbSqlQuery<Product>, ma la query non è stata ancora eseguita. Per ottenere i risultati, come già abbiamo visto in LINQ, è necessario enumerarla, come fatto in questo esempio con la chiamata al metodo ToList.

Una query SQL è altresì eseguibile per ottenere istanze di altri tipi, inclusi quelli primitivi, utilizzando stavolta il metodo SqlQuery sulla classe Database, la cui istanza è ottenibile tramite la proprietà omonima del DbContext:

```
using (AdventureWorksDW2014Entities dbquery = new AdventureWorksDW2014Entities())
{
var productColorsQuery = dbquery.Database.SqlQuery<string>("select distinct Color from DimProduct");
List<string> names = productColorsQuery.ToList();
}
```

In questo caso, la query SQL ricava i nomi dei colori dei prodotti senza duplicati grazie alla clausola DISTINCT. Quindi, il risultato sarà ottenuto sotto forma di una lista di string.

I comandi non query, per esempio le istruzioni di UPDATE o DELETE, possono essere inviate al database per mezzo del metodo ExecuteSqlCommand della stessa classe Database:

```
int affected = dbquery.Database.ExecuteSqlCommand("UPDATE DimCustomer SET Title='Dr' WHERE
CustomerKey=1234");
```

Il valore di ritorno rappresenta il numero di record coinvolti, in questo caso quelli aggiornati.

Model First

Nell'approccio *Model First*, si inizia con il creare le entità e le relazioni fra di esse sfruttando il model designer EDMX di Visual Studio. Successivamente, sarà possibile generare sia lo schema del database a partire dal modello, sia le classi che verranno utilizzate nell'applicazione.

Quello che bisogna fare è innanzitutto avviare il wizard di creazione dell'Entity Data Model, già visto quando abbiamo creato il modello a partire da un database esistente, ma selezionando stavolta Empty EF Designer Model nel primo passo del wizard (vedi Figura 15.14), nel quale si potrà dare un nome al modello stesso.

Supponiamo di voler realizzare un'applicazione di gestione dei veicoli e dei rispettivi proprietari, chiamiamo quindi il modello CarModel e confermiamo la creazione del modello vuoto. A questo punto, si aprirà la finestra dell'Entity Framework Designer, nella quale sarà possibile aggiungere le entità.

La finestra delle proprietà mostrerà i valori di default delle varie proprietà del modello. Per esempio, Entity Container Name contiene il nome della classe di contesto, derivata da DbContext, che sarà generata una volta completato il modello. Il valore di default dovrebbe essere CarModelContainer.

Per aggiungere le entità, fate clic con il pulsante destro sulla finestra del designer e selezionate Add New -> Entity. Così facendo si aprirà una finestra di dialogo in cui inserire il nome dell'entità e della rispettiva chiave primaria.

Per esempio, impostiamo rispettivamente Car e IdCar, come mostrato in Figura 15.17, e clicchiamo su OK per fare in modo che l'entità venga creata e visualizzata nel designer.

Add Entity

Properties

Entity name:

Car

Base type:

(None)

Entity Set:

CarSet

Key Property

☑ Create key property

Property name:

IdCar

Property type:

Int32

OK Cancel

Figura 15.17 – Creazione di una nuova entità.

Ora bisogna aggiungere le proprietà dell'entità Car, come Targa e Modello, entrambe di tipo string. Per farlo basta cliccare con il pulsante destro sull'entità appena creata e selezionare Add New -> Scalar Property.

La finestra delle proprietà permette di personalizzare le varie caratteristiche, per esempio la possibilità di assegnare valori null, il valore di default, la visibilità dei membri, la lunghezza massima di una stringa e così via. Impostiamo a titolo di prova il valore true per la proprietà Nullable di Targa, in maniera da poter salvare nel database anche veicoli senza una targa impostata.

Dopo aver completato l'entità Car, bisogna aggiungere un'entità Person, che conterrà i proprietari delle auto. A essa aggiungiamo una chiave IdPerson e due proprietà, First-Name e LastName. A questo punto, abbiamo una coppia di entità che bisogna correlare con l'aggiunta di un'associazione. Quindi bisogna fare clic con il pulsante destro sull'area di progettazione e selezionare Add New -> Association. Si aprirà la finestra di configurazione dell'associazione, nella quale è possibile impostare che a ogni entità Person possono essere associate più entità Car.

Ci si assicuri quindi che l'estremità Person abbia molteplicità pari a 1, mentre l'estremità Car molteplicità Many.

Infine, la casella Add foreign key properties to the 'Car' Entity deve essere selezionata in maniera da far creare automaticamente una proprietà all'interno di Car con la chiave esterna necessaria.

Add Association

Association Name:

PersonCar

End

Entity:

Person

Multiplicity:

1 (One)

☑ Navigation Property:

Car

End

Entity:

Car

Multiplicity:

* (Many)

☑ Navigation Property:

Person

☑ Add foreign key properties to the 'Car' Entity

Person can have * (Many) instances of Car. Use Person.Car to access the Car instances.

Car can have 1 (One) instance of Person. Use Car.Person to access the Person instance.

OK

Cancel

Figura 15.18 – Configurazione dell'associazione fra due entità.

Confermando la creazione dell'associazione, nel designer apparirà il modello completo di relazione fra le entità, il quale può essere usato per leggere e scrivere dati, una volta generato il codice e il database.

Per procedere alla generazione del codice, fate clic con il destro sulla superficie vuota del designer e selezionate la voce Add Code Generation Item. Nella finestra di dialogo di scelta del template, potete scegliere la versione di EF per la quale generare DbContext e relative altre classi. Selezionate per esempio EF 6.x DbContext Generator, immettete il nome CarModel e fate clic su Add per confermare.

Nel Solution Explorer, all'interno del file CarModel.tt, potete navigare le classi e il relativo codice autogenerato.

La classe dell'entità Car sarà simile alla seguente:

```
public partial class Car
{
public int IdCar { get; set; }
public string Targa { get; set; }
public string Modello { get; set; }
public int PersonIdPerson { get; set; }
public virtual Person Person { get; set; }
}
```

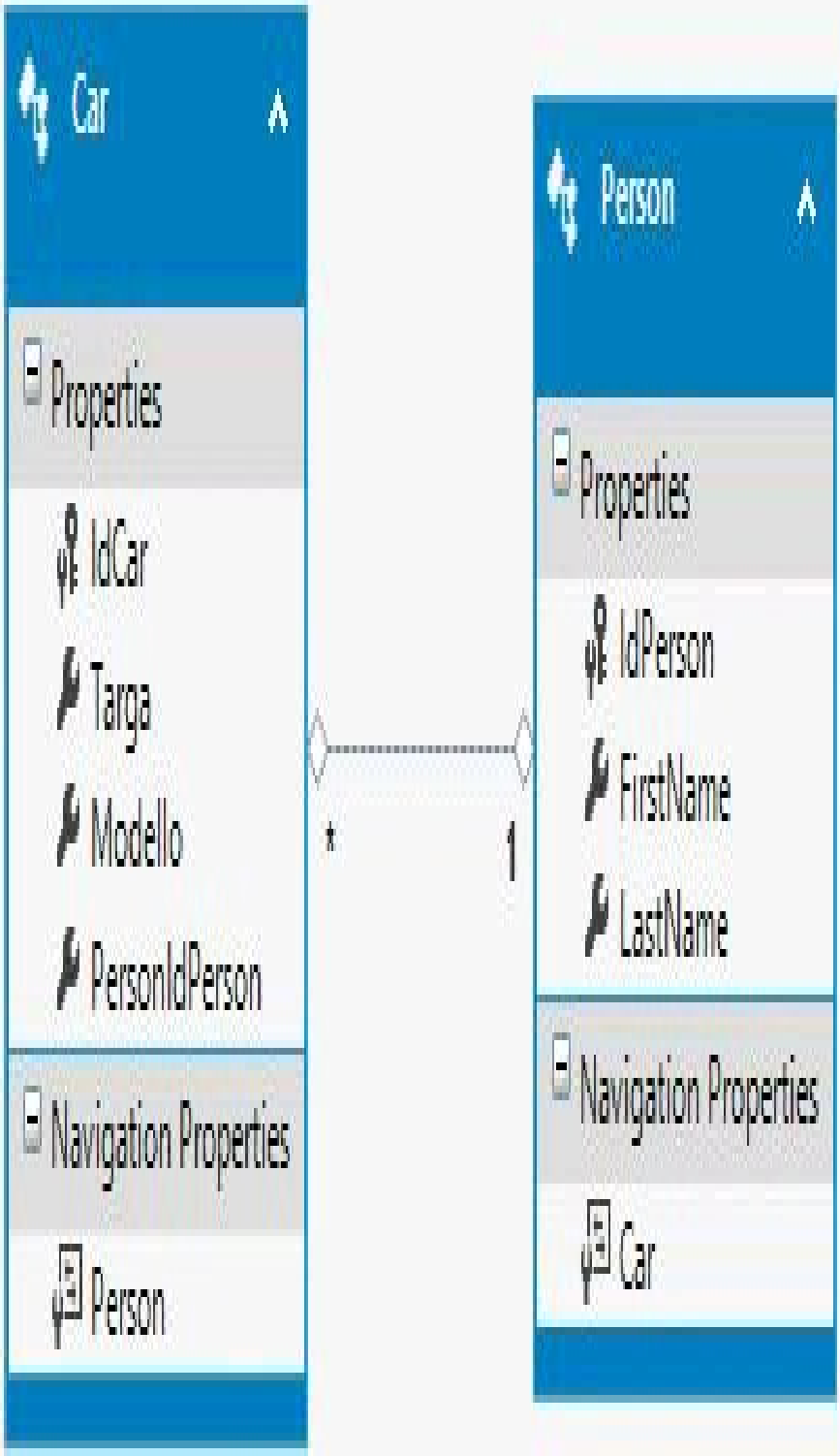

Car
Properties
IdCar
Targa
Modello
PersonIdPerson
Navigation Properties
Person
*
1
Person
Properties
IdPerson
FirstName
LastName
Navigation Properties
Car

Figura 15.19 – Il modello CarModel con le due entità.

Ora si può passare alla generazione del database, per esempio di tipo SQL Server LocalDb.

Facendo clic ancora una volta sul designer, selezionate stavolta il comando Generate Database from Model, che aprirà la finestra di creazione di una connessione al database, che abbiamo già utilizzato con il classico ADO.NET.

Aggiungete una nuova connessione, facendo clic sul pulsante apposito e inserendo il nome dell'istanza server da usare, per esempio (localdb)MSSQLLocalDb. Nel campo Database Name, per generare un nuovo database, inserite un nome non esistente, per esempio CarDb.

Selezionando OK verrà chiesta la conferma di creazione del nuovo database e si tornerà al wizard di generazione del database, che nell'ultimo passo mostrerà lo script in linguaggio SQL per la creazione del database.

Dopo aver cliccato su Finish, lo stesso script verrà aggiunto al progetto attuale di Visual Studio e si potrà procedere alla sua esecuzione facendo clic sull'apposito pulsante Execute, nella toolbar, oppure sull'omonima voce del menu contestuale che apparirà cliccando con il destro sullo stesso script.

Ora che si possiede sia il database sia il codice necessario per accedere a esso, possiamo provare a scrivere qualche esempio nel metodo Main, così da aggiungere qualche record al database:

Generate Database Wizard

Summary and Settings

Save DDL As: CarModel.edmx.sql

DDL

```
-- ----------------------------------------------------------
-- Entity Designer DDL Script for SQL Server 2005, 2008, 2012 and Azure
-- ----------------------------------------------------------
-- Date Created: 11/14/2015 00:26:31
-- Generated from EDMX file: C:\Users\Antonio\OneDrive\Scrittura libri
\CSharp6\programmare_con_csharp6\Programmare con CSharp 6\Capitolo 15\Cap15 Accesso Dati
\EFModelFirst\CarModel.edmx
-- ----------------------------------------------------------

SET QUOTED_IDENTIFIER OFF;
GO
USE [CarDb];
GO
IF SCHEMA_ID(N'dbo') IS NULL EXECUTE(N'CREATE SCHEMA [dbo]');
GO

-- ----------------------------------------------------------
-- Dropping existing FOREIGN KEY constraints
-- ----------------------------------------------------------
```

< Previous | Next > | Finish | Cancel

Figura 15.20 – Script di creazione del nuovo database.

```
static void Main(string[] args)
{
using(var db = new CarModelContainer())
{
Person persona = new Person();
persona.FirstName = "Antonio";
persona.LastName = "Pelleriti";
db.PersonSet.Add(persona);
db.SaveChanges();

Car car1 = new Car();
car1.Targa = "AB123CD";
car1.Modello = "Alfa Romeo GT";
car1.Person = persona;
db.CarSet.Add(car1);
db.SaveChanges();
}
}
```

Senza aver praticamente scritto una riga di linguaggio SQL e impostando semplici proprietà, la nostra applicazione creerà nel database un record nella tabella Person e uno a essa correlato nella tabella Car.

Code First

L'approccio di tipo *Code First*, introdotto con Entity Framework 4.1, prevede di iniziare con lo sviluppo via codice delle classi che rappresentano le entità, mentre in seguito sarà l'infrastruttura di EF a occuparsi della generazione dell'eventuale database.

Per mostrarne il funzionamento, iniziamo creando una nuova soluzione, con un progetto di tipo Class Library, da poter referenziare poi in altri progetti. Poiché esso utilizzerà le classi di Entity Framework, è necessario aggiungere allo stesso le librerie necessarie; il modo più semplice è usare il relativo pacchetto NuGet. Fate clic con il destro sul progetto Class Library e selezionate la voce Manage Nuget Packages. Nella finestra di gestione dei pacchetti, selezionate a sinistra il ramo Online, quindi nella casella di ricerca inserite il nome del pacchetto Entity Framework.

NuGet Package Manager: EFCodeFirst

Package source: nuget.org Filter: All Include prerelease entity framework X

EntityFramework
Entity Framework is Microsoft's recommended data access technology for new applications.

Microsoft.AspNet.Identity.EntityFramework
ASP.NET Identity providers that use Entity Framework.

EntityFramework.SqlServerCompact
Allows SQL Server Compact 4.0 to be used with Entity Framework.

Oracle.ManagedDataAccess.EntityFramework
The ODP.NET, Managed Driver Entity Framework package for EF 6 applications.

Nd.Framework.Repositories.EntityFramework
基于EntityFramework实现仓储 - 基础开发服务框架

Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

Do not show this again

EntityFramework

Action: Install **Version:** Latest stable 6.1.3

Install

Options

Show preview window

Dependency behavior: Lowest

File conflict action: Prompt

Learn about Options

Description

Entity Framework is Microsoft's recommended data access technology for new applications.

Author(s): Microsoft

License: http://go.microsoft.com/fwlink/?LinkID=320539

Downloads: 14,011,578

Project URL: http://go.microsoft.com/fwlink/?LinkID=320540

Report Abuse:

Figura 15.21 – Gestione dei pacchetti NuGet.

Una volta trovatolo, fate clic sul pulsante Install: verranno scaricati tutti gli assembly necessari e aggiunti ai riferimenti del progetto.

Un package NuGet può anche essere aggiunto al progetto mediante l'apposita console, azionabile dal menu Tools -> Library Package Manager di Visual Studio, eseguendo in questo caso il comando Install-Package EntityFramework. Lo stesso package deve essere aggiunto anche al progetto che utilizzerà la Class Library delle entità.

Al progetto aggiungiamo due classi Car e Person, contenenti il seguente codice:

```
public class Person
{
public int PersonId { get; set; }
public string FirstName { get; set; }
public string LastName { get; set; }
}

public class Car
{
public int CarId { get; set; }
public string Targa { get; set; }
public string Modello { get; set; }
public int PersonId { get; set; }

public Person Person { get; set; }
}
```

Le classi sono molto semplici e contengono il codice strettamente necessario a mantenere lo stato delle entità e a definire le associazioni fra di esse. Gli oggetti istanze di tali classi personalizzate sono spesso chiamati *POCO* (*Plain Old CLR Object*), data la loro struttura molto semplice, senza attributi particolari e altre responsabilità.

Nel nostro esempio, la classe Car ha due proprietà string che definiscono targa e modello di un'automobile; può avere un solo proprietario di tipo Person, identificato dalla proprietà PersonId. Inoltre essa possiede anche una proprietà di tipo Person, per contenere direttamente l'istanza Person che rappresenta il proprietario. Al contrario, ogni Person può avere associate molte entità Car ed è identificata da una chiave intera PersonId e da due stringhe per nome e cognome.

Le due proprietà CarId e PersonId rappresentano le chiavi primarie delle tabelle che verranno create all'interno del database. In questo caso, non è necessario nessun attributo, in quanto per convenzione Entity Framework utilizzerà come chiave primaria delle tabelle le proprietà a cui è stato dato nome ID (non case sensitive, quindi va bene anche qualsiasi variante in maiuscolo e minuscolo), oppure, come in questo caso, le proprietà denominate con NomeClasseID (o una

variante). Inoltre, se tali proprietà sono di tipo int o Guid, esse saranno anche create come colonne di tipo identity, cioè autogenerate o autoincrementali.

Se invece si vuol esplicitamente indicare quale proprietà dovrà fungere da chiave primaria, anche se non rispetta le convezioni suddette, si dovrà utilizzare l'attributo Key:

```
[Key]
public int Chiave {get;set;}
```

Tale attributo, insieme ad altri che permettono di aggiungere differenti annotazioni a proprietà e classi, è definito nel namespace System.ComponentModel.DataAnnotations.

In modo analogo, la convenzione prevista per le chiavi esterne consente di correlare automaticamente le entità. Nel nostro esempio, la proprietà PersonId di Car viene collegata alla chiave primaria PersonId di Person, in quanto esse sono dello stesso tipo int e perché esiste anche una proprietà di navigazione cui è stato assegnato lo stesso nome della classe dell'entità collegata:

```
public class Car
{
//Foreign Key
public int PersonId { get; set; }
//Navigation Property
public Person Person { get; set; }
}
```

Una volta create le classi delle entità, si necessita di un contesto dati per utilizzarle e fare in modo che Entity Framework possa generare un database.

Basta quindi creare una nuova classe, che chiameremo CarContext, derivandola dalla classe standard DbContext contenuta nel namespace System.Data.Entity. Al suo interno, si definiscono delle proprietà che rappresentano le tabelle contenenti le entità create in precedenza, di tipo DbSet<T>:

```
using System;
using System.Data.Entity;

namespace EFCodeFirstLibrary
{
public class CarContext:DbContext
{
public DbSet<Car> Cars {get;set;}
public DbSet<Person> People { get; set; }
}
}
```

Utilizzando il CarContext così definito, verrà creato un database a cui sarà assegnato un nome ricavato dal nome completo della classe, in questo caso EFModelFirst.Library.CarContext, con due tabelle Cars e People, che conterranno i dati delle entità Car e Person.

Se si vuol specificare un nome per il database, bisogna aggiungere un costruttore personalizzato cui passare sotto forma di string il parametro con il nome desiderato.

Altri costruttori possono anche permettere di personalizzare la stringa di connessione, e quindi il server e il database da utilizzare.

Vediamo ora come utilizzare la classe CarContext:

```
using (CarContext db = new CarContext())
{
Car c = new Car();
c.Targa = "abc";
c.Modello = "Alfa Romeo GT";

Person p = new Person();
p.FirstName = "Antonio";
p.LastName = "Pelleriti";
c.Person = p;

db.Cars.Add(c);
db.People.Add(p);
db.SaveChanges();
}
```

Il codice precedente crea il database, le due tabelle e quindi aggiunge un record in ognuna di esse.

Per la generazione del database, dato che non abbiamo specificato alcuna stringa di connessione, Entity Framework prima verifica se esiste un'installazione locale di SQL Server Express, altrimenti va alla ricerca di un'istanza di SQLServer LocalDb.

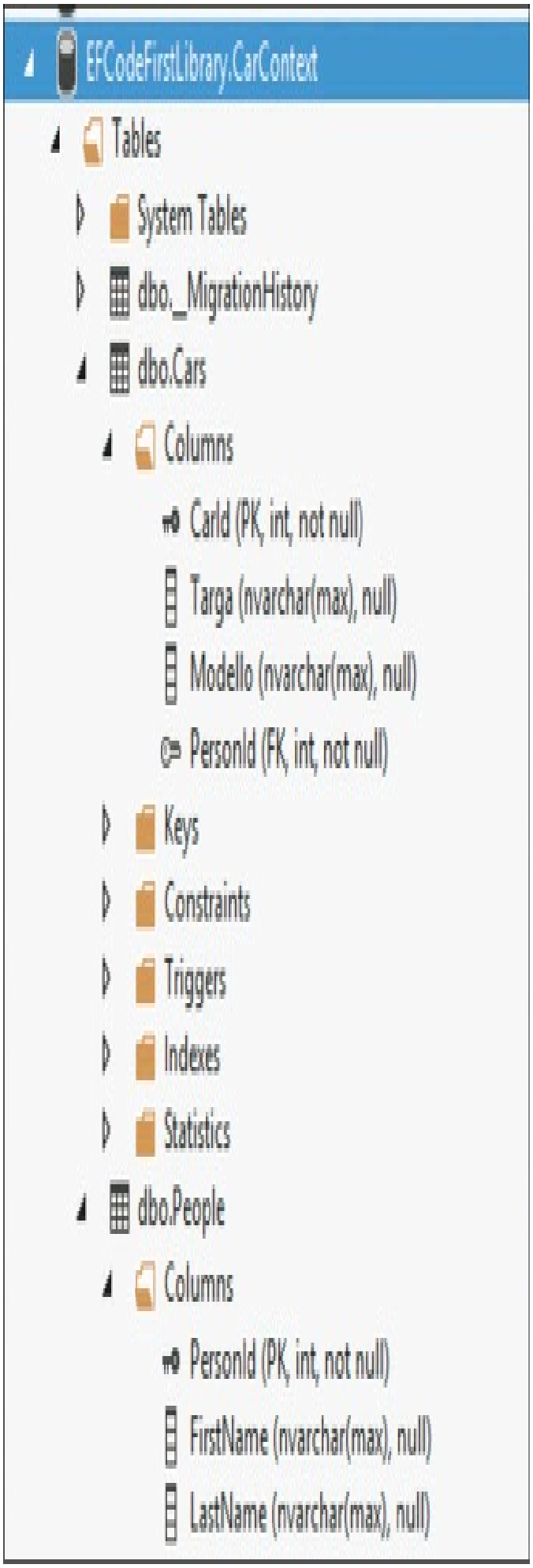

EFCodeFirstLibrary.CarContext
Tables
System Tables
dbo._MigrationHistory
dbo.Cars
Columns
CarId (PK, int, not null)
Targa (nvarchar(max), null)
Modello (nvarchar(max), null)
PersonId (FK, int, not null)
Keys
Constraints
Triggers
Indexes
Statistics
dbo.People
Columns
PersonId (PK, int, not null)
FirstName (nvarchar(max), null)
LastName (nvarchar(max), null)

Figura 15.22 – Il database generato da Entity Framework con l'approccio Code First.

Il nome del database da creare è assegnato automaticamente a partire dal nome completo della classe derivata da DbContext; può essere personalizzato passandolo al costruttore della classe base:

```
public CarContext() : base("CarDb") {}
```

La stringa di connessione può essere specificata invece inserendola nell'app.config:

```
<configuration>
<connectionStrings>
<add name="CarDbConnectionString"
connectionString="Data Source=.\SQLEXPRESS;Initial Catalog=CarDb;Integrated Security=true"
providerName="System.Data.SqlClient"/>
</connectionStrings>
</configuration>
```

e poi passando il nome della stringa al costruttore base, con il prefisso name=:

```
public CarContext() : base("name=CarDbConnectionString") {}
```

L'altra possibilità è di specificarla interamente impostando la proprietà dell'oggetto CarContext nel modo seguente:

```
db.Database.Connection.ConnectionString =
@"Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=CarDb;Integrated Security=True;";
```

La Figura 15.22 mostra il database generato all'interno della finestra SQL Server Object Explorer di Visual Studio.

NOTA	Code First, come anche gli altri workflow possibili con Entity Framework, non si può naturalmente sviscerare in un solo libro, figuriamoci in pochi paragrafi. Quindi, per i dettagli vi rinviamo alla documentazione ufficiale di MSDN, oppure a qualche testo dedicato all'argomento.

Domande di riepilogo

1) Qual è il metodo corretto per leggere il contenuto di un file di testo?

a. String txt=File.ReadAllText(path);
b. String txt=File.ReadAllLines(path);
c. String txt=File.ReadAllBytes(path) as string;
d. String txt=File.OpenText(path);

2) La classe FileSystemWatcher permette di:

a. esaminare il contenuto di una directory
b. monitorare i cambiamenti di una directory o dei file di una directory
c. esaminare il contenuto di un disco rigido
d. monitorare i cambiamenti di un disco rigido

3) Quale classe ADO.NET si utilizza per connettersi a un database?

a. Database
b. Command
c. Connection
d. DataAdapter

4) Quale metodo di un DbCommand si utilizza per eseguire una query di UPDATE?

a. ExecuteNonQuery
b. ExecuteQuery
c. ExecuteScalar
d. Execute

5) Quale metodo di un DbDataAdapter si utilizza per riempire un DataSet?

a. ExecuteQuery
b. Il costruttore
c. Load
d. Fill

6) La classe che rappresenta il punto di ingresso principale per il framework LINQ to SQL è:

a. DataContext
b. Database
c. DataSet

d. Table

7) Entity Framework può essere utilizzato solo per database SQL Server. Vero o falso?

8) Quale metodo del DbContext si usa in Entity Framework per inviare le modifiche al database?

a. Update
b. SaveChanges
c. SubmitChanges
d. Write

9) Quale non è uno dei possibili workflow utilizzabili in Entity Framework?

a. Code First
b. Model First
c. Database First
d. Framework First

10) Per ricavare l'entità dalla tabella Product con chiave primaria uguale a 1 si utilizza:

a. dbContext.Product.Load(1);
b. dbContext.LoadProduct(1);
c. dbContext.Product.Find(1);
d. dbContext.Product[1];

.NET Compiler Platform

La .NET Compiler Platform, conosciuta anche con il code name Roslyn, è utilizzabile dal proprio codice C# come un servizio o una interfaccia di programmazione per implementare strumenti o estensioni di creazione dinamica, analisi e miglioramento del codice.

Un compilatore di un linguaggio è in genere una scatola nera in cui entra del codice sorgente e che produce dei file eseguibili o librerie. Il funzionamento interno non è noto, o comunque la maggior parte degli sviluppatori può anche non avere interesse nei confronti di tali meccanismi.

D'altra parte, gli strumenti odierni di supporto allo sviluppo software, come Visual Studio o le sue estensioni, forniscono funzionalità di analisi del codice, di intellisense, di refactoring, e in generale di miglioramento della produttività, che naturalmente devono potere andare a fondo nel codice, quasi al livello di un compilatore. Per tali motivi, Microsoft ha deciso di "aprire" al mondo esterno il compilatore e anzi di esporne i servizi sotto forma di una piattaforma di compilazione, la .NET Compiler Platform (conosciuta anche con il nome in codice *Roslyn*), contenente servizi e API utilizzabili da ogni sviluppatore per creare i propri strumenti o per integrare tali funzionalità direttamente nelle proprie applicazioni.

.NET Compiler Platform

La .NET Compiler Platform espone le funzionalità del compilatore C# (oltre a quello VB) in maniera che gli sviluppatori possano utilizzarle come un qualsiasi servizio o interfaccia di programmazione.

NOTA	I sorgenti del progetto Roslyn, con relativi esempi e documentazione, sono disponibili su GitHub, all'indirizzo https://github.com/dotnet/roslyn.

Le API esposte replicano tutte le fasi di funzionamento di un compilatore, la cui sequenza è rappresentata in Figura 16.1.

Computer
Pipeline
Power
Network
Memory
Input
Output
Display
Keyboard
Mouse
Speaker

Figura 16.1 – Fasi di compilazione.

Nella prima fase, il parser analizza il codice sorgente e produce un albero sintattico basato sulla grammatica del linguaggio. A questo punto, nella seconda fase, le dichiarazioni e i metadati vengono importati e trasformati in simboli. Nella fase di binding, gli identificatori trovati nel programma sono collegati ai rispettivi simboli. Infine, tutte le informazioni raccolte dal compilatore sono utilizzate per emettere il codice IL dell'assembly finale.

Ogni compilatore combina tutte le fasi in una scatola nera, mentre per ognuna di esse la .NET Compiler Platform fornisce un modello a oggetti e delle API che permettono di accedere alle rispettive informazioni (vedi Figura 16.2).

Compiler API
Syntax tree API
Symbol API
Control flow graph API
Emit API

Figura 16.2 – API del compilatore esposte dalla .NET Compiler Platform.

Lo stesso Visual Studio utilizza questi servizi per esporre tutte le sue enormi potenzialità: per esempio, la formattazione automatica utilizza le Syntax API, le funzioni dell'object browser sono permesse dalla tabella dei simboli, il refactoring utilizza le API di semantica, la generazione automatica di snippet di codice usa le API di Emit.

NOTA	La .NET Compiler Platform permette di gestire anche un altro livello, quello dei workspace. Esso aggiunge il supporto alle soluzioni e ai progetti di Visual Studio.

Installazione di .NET Compiler Platform SDK

Per preparare la propria macchina di sviluppo e poter lavorare con Roslyn, è necessario innanzitutto avere a disposizione Visual Studio 2015 e avere installato anche i *Visual Studio Extensibility Tools* (eventualmente si possono aggiungere all'installazione anche in un secondo momento).

Per installare il .NET Compiler Platform SDK, potete scaricare il relativo pacchetto VSIX ricercandolo sul sito Visual Studio Gallery o dal menu Tools -> Extensions and Updates.

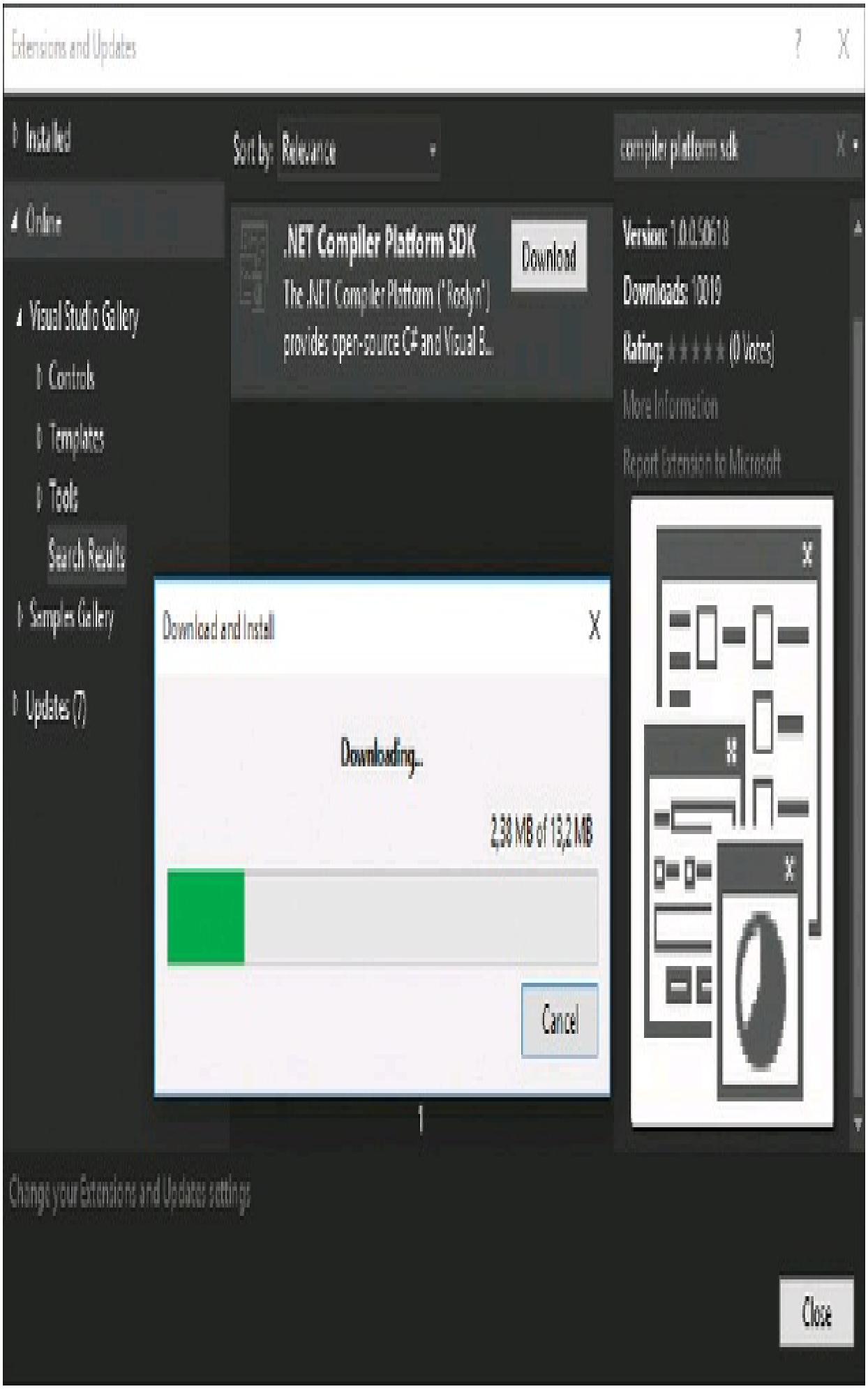

Extensions and Updates
Installed
Sort by: Relevance
compiler platform sdk
Online
.NET Compiler Platform SDK
Download
The .NET Compiler Platform ("Roslyn") provides open-source C# and Visual B...
Version: 1.0.0.50618
Downloads: 10019
Rating: (0 Votes)
More Information
Report Extension to Microsoft
Visual Studio Gallery
Controls
Templates
Tools
Search Results
Samples Gallery
Updates (7)
Download and Install
Downloading...
2,38 MB of 13,2 MB
Cancel
Change your Extensions and Updates settings
Close

Figura 16.3 – Download del .NET Compiler Platform SDK.

Un'altra possibilità è quella di iniziare la creazione di un nuovo progetto, selezionare la sezione Extensibility, che mostrerà fra i template la possibilità di scaricare direttamente il .NET Compiler Platform SDK.

Confermando la creazione del progetto si aprirà subito una pagina HTML con un pulsante dal quale scaricare il pacchetto Roslyn_SDK.vsix. Il .NET Compiler Platform SDK installerà tre diversi template, sia per C# sia per Visual Basic, per la creazione di:

- Analyzer with Code Fix – estensioni in formato NuGet o VSIX di Visual Studio per effettuare analisi e diagnostica, e applicare eventuali correzioni al codice;
- Code Refactoring – estensioni VSIX di Visual Studio per il refactoring del codice;
- Stand Alone Code Analysis Tool – applicazioni a riga di comando per generare, analizzare o trasformare codice C#.

Per sfruttare le funzionalità della .NET Compiler Platform all'interno delle applicazioni è possibile scaricare e installare il pacchetto NuGet Microsoft.CodeAnalysis. Questo è un pacchetto All-in-One contenente un insieme di tutti gli assembly. In alternativa, essi sono disponibili anche in singoli pacchetti suddivisi per linguaggio C#/VB o per funzionalità.

In qualche caso, i pacchetti sono disponibili (al momento in cui il libro sta andando in stampa) solo in versione pre-release. Quindi, quando andrete a installare i pacchetti dal NuGet Manager in Visual Studio, assicuratevi di abilitare l'apposita casella.

Syntax Visualizer

Installando il .NET Compiler Platform SDK, in Visual Studio 2015 sarà disponibile una nuova finestra denominata Syntax Visualizer. Essa permette di esplorare e capire meglio come Roslyn veda il codice sorgente.

Dal menu View -> Other Windows potete visualizzarla, se non fosse visibile, e posizionarla dove più vi piace. Ora aprite o create un progetto di Visual Studio e aggiungetegli una classe.

La finestra Syntax Visualizer mostrerà un albero i cui nodi rappresentano i vari elementi della sintassi. Provate, per esempio, a selezionare nell'editor di codice la dichiarazione di un namespace, o di una classe: automaticamente verrà evidenziato il nodo corrispondente. Viceversa, selezionando un nodo nell'albero del Syntax Visualizer, si evidenzierà la corrispondente porzione di codice sorgente.

Class1.cs

CSharpCompilerPlatfo | Download_the.NET_Cr | Main()

```
using System;

namespace Download_the.NET_Compiler_Platform_SDK
{
    0 referencias
    class Class1
    {
        0 references
        static void Main()
        {
        }
    }
}
```

Syntax Visualizer

Syntax Tree Legend

- CompilationUnit [0..155)
  - UsingDirective [0..13)
  - NamespaceDeclaration [17..153)
    - NamespaceKeyword [17..26)
    - QualifiedName [27..65)
      - IdentifierName [27..39)
      - DotToken [39..40)
      - IdentifierName [40..65)
    - OpenBraceToken [67..68)
    - ClassDeclaration [74..150)
      - ClassKeyword [74..79)
      - IdentifierToken [80..86)
      - OpenBraceToken [92..93)
      - MethodDeclaration [103..143)
      - CloseBraceToken [149..150)
    - CloseBraceToken [152..153)
      - Trail: EndOfLineTrivia [153..155)

Properties

Type ClassDeclarationSyntax

Kind ClassDeclaration

ContainsDiagnostic: False

ContainsDirectives False

ContainsSkippedTex False

Figura 16.4 – La finestra Syntax Visualizer.

Sintassi

Le *Syntax API* permettono di analizzare la struttura di un programma, utilizzando in particolare gli alberi sintattici, o Syntax Tree, che il compilatore utilizza per capire un programma C#. Queste strutture sono fondamentali per la compilazione, l'analisi del codice, il refactoring, la generazione di codice, funzioni di alto livello di un IDE e così via.

Un SyntaxTree è una struttura ad albero composta da nodi, token e trivia, che nella finestra Syntax Visualizer sono rispettivamente quelli rappresentati in blu, verde e rosso. La classe SyntaxTree è astratta: per l'analisi della sintassi in C# si utilizzano i metodi della classe CSharpSyntaxTree.

Un nodo rappresenta tutti i costrutti sintattici come istruzioni, dichiarazioni, espressioni. Ogni categoria di nodo è rappresentata da una classe derivata da SyntaxNode.

I nodi sono elementi, non foglie dell'albero: hanno sempre altri nodi o token come figli, che sono accessibili tramite la proprietà ChildNodes. Il nodo padre può essere invece ricavato dalla proprietà Parent. Ogni nodo ha poi una collezione di metodi per accedere agli elementi al di sotto di esso, come DescendantNodes, DescendantTokens, e DescendantTrivia. Ogni sottoclasse di SyntaxNode ha poi altri membri specifici di ogni categoria.

I token sono gli elementi foglia, cioè i terminali, e rappresentano i frammenti sintattici più piccoli: parole chiave, identificatori, literal e punteggiatura. Essi sono rappresentati dalla struttura SyntaxToken.

Infine, i nodi del tipo struct SyntaxTrivia rappresentano elementi come gli spazi e i commenti che possono apparire ovunque nel codice.

Parsing

Per ottenere un SyntaxTree e mostrare un esempio di utilizzo delle API, il modo più semplice è partire da una stringa contenente del codice sorgente e usare il metodo ParseText.

Innanzitutto, create un nuovo progetto di tipo Stand Alone Code Analysys Tool, presente nel gruppo Extensibility, che permette a sua volta di creare un'applicazione console per l'analisi del codice.

Inseriamo una costante per mantenere un programma di esempio da analizzare:

```
private const string code = @"using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace HelloWorld
```

```
{
class Program
{
static void Main(string[] args)
{
Console.WriteLine(""Hello, World!"");
}
}
}";
```

Nel metodo Main, inserite il seguente codice:

```
static void Main(string[] args)
{
SyntaxTree tree = CSharpSyntaxTree.ParseText(code);
var root = (CompilationUnitSyntax)tree.GetRoot();
}
```

Il metodo ParseText effettua l'analisi del codice restituendo un SyntaxTree, a partire da questo oggetto è possibile ricavare ogni tipo di informazione sul codice sorgente. Quest'ultimo utilizza per esempio delle istruzioni using, che possono essere lette dalla proprietà Usings:

```
foreach (var us in root.Usings)
{
Console.WriteLine(us.ToString());
}
```

A partire dalla radice root dell'albero, è possibile ora ricavare tutti i figli, per mezzo della proprietà Members. Il primo è la dichiarazione del namespace, che quindi, fortemente tipizzato, è un elemento NamespaceDeclarationSyntax:

```
var firstMember = root.Members[0];
var helloWorldDeclaration = (NamespaceDeclarationSyntax)firstMember;
```

La tipologia di membro può essere ottenuta dal metodo Kind, che restituisce un valore dell'enumerazione SyntaxKind, la quale contiene tutte le categorie di elementi che costituiscono la sintassi del linguaggio.

```
SyntaxKind kind=firstMember.Kind();
```

Continuando così, si possono ottenere la dichiarazione della classe, del metodo Main, dei suoi argomenti:

```
var programDeclaration = (ClassDeclarationSyntax)helloWorldDeclaration.Members[0];
var mainDeclaration = (MethodDeclarationSyntax)programDeclaration.Members[0];
var argsParameter = mainDeclaration.ParameterList.Parameters[0];
```

E stampare un riepilogo:

```
Console.WriteLine($@"Il metodo {mainDeclaration.Identifier} ha il parametro {argsParameter.Identifier} di tipo {argsParameter.Type} e restituisce
{mainDeclaration.ReturnType}");
```

Il risultato sarà una stringa come la seguente:

Il metodo Main ha il parametro args di tipo string[] e restituisce void

Il codice sorgente sottoposto al parsing può contenere naturalmente degli errori. Per verificare la loro presenza ed averne dettagli si può utilizzare il metodo GetDiagnostics di SyntaxTree:

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(errorCode);
var diagnostics=tree.GetDiagnostics();
```

Per attraversare e analizzare la struttura dell'albero, ogni classe derivata da SyntaxNode espone dei metodi che troverete comodi se avete già familiarità con LINQ. Per esempio cercare tutti i metodi che iniziano con la stringa Main è immediato con il metodo DescendantNodes:

```
var methods=root.DescendantNodes().OfType<MethodDeclarationSyntax>().Where(m =>
m.Identifier.Text.StartsWith("Method"));
```

L'approccio visto fino a qui funziona meglio quando si conosce già cosa si sta cercando, e si è interessati a determinate parti dell'albero. In alcuni casi, invece, può essere utile esplorare tutta la struttura, passando in rassegna ogni nodo e analizzandolo nell'ordine corretto.

Si può derivare una classe da quella astratta CSharpSyntaxWalker e scrivere gli override di uno o più dei suoi metodi Visit.

Il metodo Visit passa in rassegna tutti gli elementi. La seguente classe, invece, effettua l'override del metodo VisitMethodDeclaration, che verrà eseguito per ogni metodo trovato nel codice, e aggiunge i metodi con modificatore Public alla lista Methods:

```
class MethodCollector: CSharpSyntaxWalker
{
public readonly List<MethodDeclarationSyntax> Methods = new List<MethodDeclarationSyntax>();
public override void VisitMethodDeclaration(MethodDeclarationSyntax node)
{
if (node.Modifiers.Any(m => m.Kind() == SyntaxKind.PublicKeyword))
methods.Add(node);
}
}
```

Per utilizzare la classe basta creare una sua istanza e invocare il metodo Visit passando come parametro il SyntaxTree:

```
MethodCollector collector = new MethodCollector();

collector.Visit(root);
```

La proprietà Methods conterrà a questo punto tutti i nodi corrispondenti ai metodi pubblici trovati.

Costruzione

Per costruire un albero SyntaxTree, e quindi creare o modificare un programma servendosi delle Syntax API, è possibile utilizzare i metodi della classe SyntaxFactory.

Se si parte da un codice sorgente esistente, il metodo ParseText è ancora l'opzione più rapida ed efficiente per ricavare il corrispondente SyntaxTree.

I metodi della classe SyntaxFactory permettono di trattare ogni tipologia di nodo che costituisce un albero, così facendo il codice necessario diventa particolarmente lungo e complesso. Infatti, è necessario ricostruire tutto quello che in pratica viene visualizzato dalla finestra Syntax Visualizer. Per esempio, considerando la seguente semplice istruzione di invocazione di un metodo:

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

La finestra Syntax Visualizer permette di ricavarne il relativo albero, mostrato in Figura 16.5.

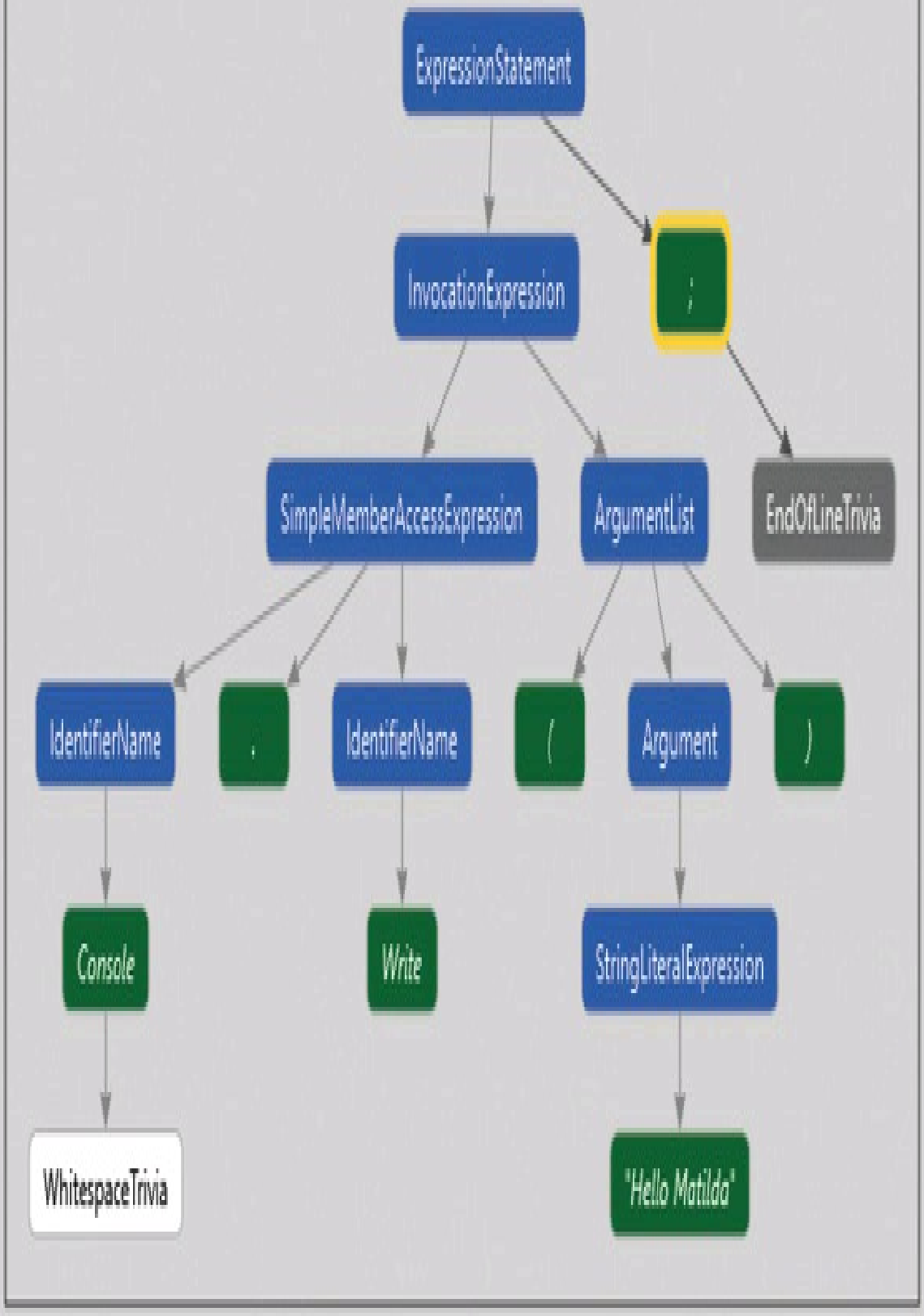

ExpressionStatement
InvocationExpression
;
SimpleMemberAccessExpression
ArgumentList
EndOfLineTrivia
IdentifierName
.
IdentifierName
(
Argument
)
Console
Write
StringLiteralExpression
WhitespaceTrivia
"Hello Matilda"

Figura 16.5 – L'albero sintattico per un'istruzione Console.Write.

Tutti i nodi possono essere ricreati per mezzo della classe SyntaxFactory. In pratica, in questo caso l'obiettivo è costruire una espressione rappresentata da ExpressionStatement, che conterrà una invocazione di un metodo, cioè InvocationExpression, con un argomento.

Il codice necessario, costruendo i vari pezzetti e poi rimettendoli insieme in una ExpressionStatement, è per esempio:

```
var console = SyntaxFactory.IdentifierName("Console");
var writeline = SyntaxFactory.IdentifierName("WriteLine");
var memberaccess = SyntaxFactory.MemberAccessExpression(SyntaxKind.SimpleMemberAccessExpression, console,
writeline);

var argument = SyntaxFactory.Argument(SyntaxFactory.LiteralExpression(SyntaxKind.StringLiteralExpression,
SyntaxFactory.Literal("Hello Matilda")));
var argumentList = SyntaxFactory.SeparatedList(new[] { argument });

var writeLineCall = SyntaxFactory.ExpressionStatement(
SyntaxFactory.InvocationExpression(memberaccess,
SyntaxFactory.ArgumentList(argumentList))
);
```

NOTA	Un buon punto di partenza per comprendere il funzionamento della classe SyntaxFactory è il progetto *Roslyn Quoter*, che mostra appunto il codice necessario per generare un dato sorgente. RoslynQuoter è open source e potete provarlo all'url http://roslynquoter.azurewebsites.net.

Trasformazione

Un concetto fondamentale da comprendere e tenere a mente è che un Syntax Tree è immutabile. Quindi, apportando delle modifiche, non si agisce su quello originale, ma si otterrà un albero aggiornato. Per tale motivo, le Syntax API non forniscono un meccanismo di modifica diretto, ma dei metodi che generano un nuovo albero contenente le modifiche apportate.

Ogni classe concreta derivata da SyntaxNode offre dei metodi il cui nome inizia con "With", utilizzabili appunto per modificare i nodi dell'albero.

Un altro metodo, ReplaceNode, consente invece di sostituire un nodo con uno nuovo.

Riprendendo il Syntax Tree costruito con il parsing del codice di esempio precedente, ecco come apportare la modifica del nome della classe Program:

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(code);
CompilationUnitSyntax root = tree.GetRoot() as CompilationUnitSyntax;
```

Viene ottenuta la dichiarazione di classe attuale e poi creata una nuova a partire da essa con un nuovo nome:

```
var classDeclaration = root.DescendantNodes().OfType<ClassDeclarationSyntax>().Single();
var identifier = SyntaxFactory.Identifier("ProgramNew");
var newClassDeclaration = classDeclaration.WithIdentifier(identifier);
```

Il nuovo nodo viene sostituito al precedente. Il metodo ReplaceNode restituisce la nuova radice dell'albero che può ora essere stampata:

```
root = root.ReplaceNode(classDeclaration, newClassDeclaration);
Console.WriteLine(root.GetText());
```

Analogamente, è possibile rimuovere e inserire nodi con il metodo RemoveNode o con quelli del tipo InsertNodeAfter e InsertNodeBefore. L'esempio che segue recupera la dichiarazione della classe, poi l'ultimo nodo figlio (che è il metodo Main, per esempio) e lo rimuove restituendo un nuovo nodo aggiornato:

```
var classDeclaration = root.DescendantNodes().OfType<ClassDeclarationSyntax>().Single();
var last = classDeclaration.ChildNodes().Last();
newClassDeclaration = classDeclaration.RemoveNode(last, SyntaxRemoveOptions.KeepEndOfLine);
root = root.ReplaceNode(classDeclaration, newClassDeclaration);
```

Per inserire un nodo dopo un altro già esistente, prima recuperiamo quest'ultimo. Riprendendo dal paragrafo precedente il nodo che rappresenta un'istruzione Console.WriteLine, memorizzato nella variabile writeLineCall, si può quindi scrivere:

```
var lastStatement= root.DescendantNodes().OfType<StatementSyntax>().Last();
root = root.InsertNodesAfter(lastStatement, new SyntaxNode[] { writeLineCall });
```

La variabile lastStatement contiene l'ultima istruzione dell'albero, e con il metodo InsertNodeAfter indichiamo che il nuovo nodo da creare dopo lastStatement è il nodo writeLineCall.

Come già visto con la classe CSharpSyntaxWalker, anche per modificare un albero sintattico esistente è possibile navigare i nodi, apportare i cambiamenti e ottenere un nuovo albero, utilizzando la classe CSharpSyntaxRewriter.

Derivando un'altra classe da essa, si possono visitare e selezionare i nodi di interesse, implementando l'override dei vari metodi e apportando, per esempio, al loro interno le modifiche necessarie.

La seguente classe ricerca nel codice in esame dei blocchi catch vuoti e li sostituisce con blocchi contenenti un'istruzione throw:

```
class CatchSyntaxRewriter: CSharpSyntaxRewriter
{

public override SyntaxNode VisitCatchClause(CatchClauseSyntax node)
{
if(!node.DescendantNodes().OfType<ThrowStatementSyntax>().Any())
{
BlockSyntax block = node.DescendantNodes().OfType<BlockSyntax>().Single();
if (!block.Statements.Any())
```

```
{
block = SyntaxFactory.Block(new StatementSyntax[] { SyntaxFactory.ThrowStatement() });
node = node.WithBlock(block);
return node;
}
}
return base.VisitCatchClause(node);
}
}
```

Il metodo VisitCatchClause sarà invocato per ogni istruzione catch trovata: se il blocco non contiene istruzioni throw, ne viene creata una. In grassetto è evidenziata la parte di creazione del nuovo blocco.

Per utilizzare la classe, basta creare una sua istanza e poi invocare il metodo Visit:

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(codeTryCatch);
CatchSyntaxRewriter rewriter = new CatchSyntaxRewriter();
var newRoot=rewriter.Visit(tree.GetRoot());
```

Compilazione

Nei paragrafi precedenti abbiamo utilizzato le Syntax API per analizzare o costruire un programma da zero. L'obiettivo finale e fondamentale che il codice sorgente deve attraversare per diventare un eseguibile, o una libreria utilizzabile direttamente o da altre applicazioni, è quello che prevede la compilazione vera e propria, cioè l'emissione di un assembly .NET.

Il primo passo da compiere è la creazione di un oggetto Compilation, una classe astratta che è la rappresentazione immutabile di una singola invocazione del compilatore, e che contiene tutte le informazioni necessarie: a partire dal Syntax Tree da compilare, ma anche altre necessarie, come i riferimenti ad altri assembly e le altre opzioni del compilatore.

Partendo dal codice sorgente di un programma, ricaviamo i SyntaxTree, e costruiamo un array di questi oggetti da dare in pasto al compilatore (infatti la compilazione avviene, in genere, su diversi sorgenti in ingresso):

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(code);
SyntaxTree[] sourceTrees = { tree };
```

L'indicazione dei riferimenti ad assembly differenti avviene poi creando un array di MetadataReference, per esempio:

```
MetadataReference mscorlib =
MetadataReference.CreateFromFile(typeof(object).Assembly.Location);
MetadataReference systemLinq =
MetadataReference.CreateFromFile(typeof(System.Linq.Enumerable).Assembly.Location);

MetadataReference[] references = { mscorlib, systemLinq };
```

Naturalmente, gli assembly possono anche essere ottenuti in altri modi, per esempio caricandoli da un file.

Nel caso C#, l'oggetto Compilation è a questo punto ricavabile con il metodo statico Create della classe concreta CSharpCompilation, passando come parametri, oltre ai SyntaxTree e ai MetadataReference, le opzioni di compilazione rappresentate da un oggetto CSharpCompilationOptions:

```
CSharpCompilation compilation = CSharpCompilation.Create("RoslynCS",
sourceTrees,
references,
new CSharpCompilationOptions(OutputKind.ConsoleApplication));
```

L'oggetto Compilation ha ora tutto quello che serve per procedere alla compilazione, che avverrà in memoria oppure direttamente su file, utilizzando il metodo Emit. Quest'ultimo restituisce un risultato, dal quale è possibile leggere se la compilazione è andata a buon fine, oppure ricavare gli errori riscontrati.

Nel seguente esempio, la compilazione avviene in memoria, usando un MemoryStream:

```
using (var ms = new MemoryStream())
{
EmitResult result = compilation.Emit(ms);

if (!result.Success)
{
IEnumerable<Diagnostic> failures = result.Diagnostics.Where(diagnostic =>
diagnostic.IsWarningAsError ||
diagnostic.Severity == DiagnosticSeverity.Error);

foreach (Diagnostic diagnostic in failures)
{
Console.Error.WriteLine("{0}: {1}", diagnostic.Id, diagnostic.GetMessage());
}
}
else
{
using (var file = File.Create( $"{compilation.AssemblyName}.exe"))
ms.WriteTo(file);
}
}
```

La proprietà Success di EmitResult indica l'eventuale riuscita della compilazione. Se essa è pari a false, gli errori vengono letti dalla proprietà Diagnostics e stampati.

Nell'esempio, in caso di compilazione corretta, viene creato un file eseguibile.

Analisi semantica

Finora abbiamo utilizzato la .NET Compiler Platform per analizzare, modificare o costruire la struttura sintattica di un programma. Il passo seguente è quello di ottenere informazioni sulla sua semantica o significato. Per esempio, oltre a poter ricavare la dichiarazione di una classe e dei suoi membri, sarà possibile comprendere quali e quanti riferimenti alla classe esistono, oppure dove un metodo viene invocato.

Ricavare informazioni sul modello semantico di un programma spesso richiede anche una compilazione.

In parole povere, una volta creato un oggetto Compilation (vedere il Paragrafo "Compilazione"), possiamo ottenere da esso un'istanza di SemanticModel:

```
var model = compilation.GetSemanticModel(tree);
```

Un programma C# è fatto da un insieme di elementi come namespace, tipi, membri e così via. Il compilatore si riferisce a essi mediante simboli che contengono informazioni sulla locazione in cui sono stati dichiarati (per esempio, nel sorgente oppure nei metadati di un assembly esterno), sul namespace o sul tipo in cui il simbolo è stato dichiarato, sui modificatori di accesso e così via.

Il modello semantico è quindi l'oggetto che consente di ottenere queste informazioni a partire dal SyntaxTree, cioè effettua il binding fra sintassi e simboli:

```
var methodMainSyntax =
tree.GetRoot().DescendantNodes().OfType<MethodDeclarationSyntax>()
.First(m => m.Identifier.Text == "Main");

var methodMainSymbol = model.GetDeclaredSymbol(methodMainSyntax);
```

Il metodo GetDeclaredSymbol ottiene informazioni sul simbolo corrispondente all'elemento sintattico passato come argomento. In questo caso, si è ricavato il simbolo relativo al metodo Main, dal quale possiamo poi ottenere svariate informazioni:

```
Console.WriteLine($"Informazioni semantiche su {methodMainSymbol.Name}");
Console.WriteLine($"tipo contenitore: {methodMainSymbol.ContainingType}");
Console.WriteLine($"tipo di ritorno: {methodMainSymbol.ReturnType}");
Console.WriteLine($"static: {methodMainSymbol.IsStatic}");
Console.WriteLine("parametri:");
foreach(var par in methodMainSymbol.Parameters)
Console.WriteLine($"{par.Type} {par.Name}");
```

Se l'esempio precedente mostra come effettuare il binding fra un nome, quello del metodo Main, e il rispettivo simbolo, in altri casi l'elemento contenuto nel programma C# di cui vogliamo ricavare il relativo simbolo potrebbe anche non avere un nome.

Riprendendo il codice sorgente usato fino ad ora, si pensi per esempio all'argomento stringa passato al metodo Console.WriteLine. Per ottenerlo, possiamo andare alla ricerca di un LiteralExpressionSyntax:

```
var helloMatildaString = tree.GetRoot().DescendantNodes()
.OfType<LiteralExpressionSyntax>()
.First();
```

Ottenere informazioni sul tipo del valore letterale (che sappiamo essere string) è possibile tramite il metodo GetTypeInfo:

```
var typeInfo = model.GetTypeInfo(helloMatildaString);
```

Il tipo dell'espressione ricavato è un simbolo che è possibile ora analizzare, per esempio ottenendone il nome e i membri:

```
ITypeSymbol typeSymbol = typeInfo.Type;
Console.WriteLine($"il tipo di {helloMatildaString} è {typeSymbol.Name}");

var staticMethods=typeSymbol.GetMembers()
.Where(m => m.Kind == SymbolKind.Method && m.IsStatic)
.Select(m=>m.Name).Distinct();
staticMethods.ToList().ForEach(name => Console.WriteLine(name));
```

In questo caso, vengono filtrati i metodi statici del tipo string.

NOTA	Le API relative al modello semantico sono certamente fra le più complesse e permettono di analizzare aspetti ancora più complicati, come il flusso dei dati e quello del controllo di un programma. Anche in questo caso, purtroppo, lo spazio è tiranno e sarebbe necessario ben più di un capitolo.

Scripting API

Le API di scripting (*scripting API*) permettono di istanziare il motore di compilazione di C# ed eseguire snippet di codice all'interno di un'applicazione .NET.

NOTA	Per utilizzare le API di scripting, è necessario installare il pacchetto NuGet Microsoft.CodeAnalysis.CSharp.Scripting e le relative dipendenze.

Esistono diverse modalità per sfruttare le API di scripting, molte delle quali mostrate negli esempi seguenti. In generale, è necessario utilizzare i due namespace seguenti:

```
using Microsoft.CodeAnalysis.CSharp;
using Microsoft.CodeAnalysis.CSharp.Scripting;
```

La classe fondamentale è CSharpScript. Il modo più semplice per utilizzare le funzionalità di scripting è quello di valutare un'espressione che restituisce un singolo valore. Il metodo EvaluateAsync accetta una stringa contenente l'espressione da valutare e restituisce il risultato:

```
var result = await CSharpScript.EvaluateAsync("1 + 2");
Console.WriteLine(result); // 3
```

Un secondo overload del metodo permette di indicare il tipo di ritorno, così da tipizzare fortemente le espressioni:

```
double d = await CSharpScript.EvaluateAsync<double>("5/(3-1)*3.5");
Console.WriteLine(d); // 7
```

L'espressione può contenere più istruzioni C# separate da punto e virgola:

```
var sum = await CSharpScript.EvaluateAsync("int x=1; int y=2; int z=x+y; z");
Console.WriteLine(sum); // 3
```

L'ultima non è una vera e propria istruzione, ma serve a ricavare il valore finale dell'espressione. Per gestire eventuali errori presenti in quest'ultima, è possibile gestire l'eccezione CompilationErrorException:

```
try
{
await CSharpScript.EvaluateAsync("1+a");
}
catch (CompilationErrorException e)
{
Console.WriteLine(string.Join(Environment.NewLine, e.Diagnostics)); //stampa gli errori di compilazione
}
```

In questo caso, l'errore nell'espressione è l'utilizzo di una variabile a mai dichiarata.

Per creare ed eseguire script più complessi, è possibile utilizzare il metodo RunAsync, che permette di esaminare lo stato durante le varie fasi di esecuzione. Per esempio, per analizzare il valore di una variabile e del valore finale restituito dallo script:

```
var state = await CSharpScript.RunAsync("int x=0;int y=x+1; y");
ScriptVariable y = state.Variables.First(sv => sv.Name == "y");
var returnVal = state.ReturnValue;
```

Uno script può anche essere creato ma non eseguito, utilizzando il metodo Create:

```
var script= CSharpScript.Create("int x=2;int y=x*x; y");
```

Così lo script può essere esaminato, per esempio per ottenerne il codice:

```
Console.WriteLine(script.Code);
```

E quindi compilato, leggendone gli eventuali errori:

```
var errors = script.Build();
```

Oppure eseguito come visto in precedenza:

```
var result = await script.RunAsync();
```

Per analizzare invece la semantica, usando l'intero insieme delle API della .NET Compiler Platform, è possibile ottenere un oggetto Compilation da uno script:

```
Compilation compilation = script.GetCompilation();
string lang=compilation.Language;
```

Naturalmente, uno script può anche essere molto complesso e formato da più righe di istruzioni. In questo caso è comodo costruirlo su più righe:

```
var script = CSharpScript.Create("int x=1;")
.ContinueWith("int y=2;")
.ContinueWith("int z=x+y;")
.ContinueWith("z");

var result=await script.RunAsync();
Console.WriteLine(result.ReturnValue);
```

Oppure è possibile eseguirlo passo dopo passo ed esaminare mano a mano il suo stato:

```
var state = await CSharpScript.RunAsync("int x=1;");
Console.WriteLine(state.Variables[0]);

state = await state.ContinueWithAsync("int y=2;");
state = await state.ContinueWithAsync("int z=x+y;");
state = await state.ContinueWithAsync("z");
var z = state.ReturnValue;
Console.WriteLine(z);
```

Inoltre, lo script può avere bisogno di accedere a classi e metodi di un assembly differente, quindi è possibile aggiungere un riferimento e le necessarie istruzioni using tramite un oggetto ScriptOptions:

```
ScriptOptions scriptOptions = ScriptOptions.Default;
```

```
var systemCore = typeof(System.Linq.Enumerable).Assembly;
//Aggiunge riferimento
scriptOptions = scriptOptions.AddReferences(systemCore);
//Aggiunge namespaces
scriptOptions = scriptOptions.AddImports("System");
scriptOptions = scriptOptions.AddImports("System.Linq");
scriptOptions = scriptOptions.AddImports("System.Collections.Generic");

var state = await CSharpScript.RunAsync(@"var list = new List<int>(){1,2,3,4,5};", scriptOptions);
state = await state.ContinueWithAsync("var sum = list.Sum();");
var sum=state.Variables.FirstOrDefault(v => v.Name == "sum");
Console.WriteLine(sum.Value);
```

L'esempio esegue uno script che crea una lista di interi e somma gli elementi usando il metodo `Sum` di LINQ.

REPL (Read Eval Print Loop)

Sfruttando le Scripting e le Syntax API analizzate nei paragrafi precedenti, è possibile scrivere una semplice applicazione REPL che replica la finestra C# Interactive, interpretando ed eseguendo le istruzioni C# inserite da riga di comando.

A ogni pressione del tasto Invio, si verifica se la riga inserita costituisce un'istruzione completa: in caso affermativo esegue lo script con `RunAsync`. Dopo l'esecuzione, se lo script ha generato un valore di ritorno, esso viene stampato.

Innanzitutto, impostiamo in una variabile `options` le opzioni da utilizzare per il parsing del codice sorgente, in particolare per indicare che esso è di tipo `Script`, con cui è possibile gestire istruzioni o espressioni isolate, inserite una dopo l'altra, come appunto avverrà in maniera interattiva:

```
options = new CSharpParseOptions(LanguageVersion.CSharp6,
DocumentationMode.Parse, SourceCodeKind.Script, null);
```

Viene poi creato uno script vuoto ed avviato un ciclo infinito, dal quale si uscirà solo digitando il comando speciale `:q`:

```
Script<object> script = CSharpScript.Create(string.Empty);
Console.Write("C# 6 REPL (inserisci :q per uscire)");
while (true)
{
Console.Write("> ");
var code = Console.ReadLine();
var newScript = script.ContinueWith(code);

ScriptState<object> result = null;
try
{
bool isComplete = IsCompleteSubmission(code);
if (code == ":q")
break;
else if (!isComplete)
```

```
Console.WriteLine("Istruzione incompleta");
else
{
result = await newScript.RunAsync();
script = newScript;
}
}
catch (Exception ex)
{
var error = ex.ToString();
WriteError(error);
}

if (result != null && result.ReturnValue != null)
Console.WriteLine(result.ReturnValue);
}
```

All'interno del ciclo, il metodo Console.ReadLine legge la prossima istruzione e la aggiunge allo script corrente. Solo quando il codice inserito rappresenta un'istruzione completa, esso viene eseguito e ne viene eventualmente stampato il risultato.

I due seguenti metodi sono usati nel codice precedente. Il primo stampa eventuali messaggi di errore, il secondo verifica che il codice inserito rappresenti un'istruzione completa, usando il metodo ParseSyntaxTree:

```
private static void WriteError(string error)
{
var color = Console.ForegroundColor;
Console.ForegroundColor = ConsoleColor.Red;
Console.WriteLine(error);
Console.ForegroundColor = color;
}

private static bool IsCompleteSubmission(string code)
{
var syntaxTree = SyntaxFactory.ParseSyntaxTree(code, options: options);
return SyntaxFactory.IsCompleteSubmission(syntaxTree);
}
```

Code Fix e Analyzer in Visual Studio

Nelle precedenti versioni di Visual Studio, scrivere delle estensioni di analisi e di correzione del codice non era un compito molto semplice. Con le *Diagnostics API* e quelle esposte in questo capitolo, facenti parte della .NET Compiler Platform, basta scrivere un po' di codice per identificare, per esempio, un particolare tipo di problema e apportare la modifica usando i metodi di trasformazione del codice. Il tutto integrandolo nell'ambiente di Visual Studio e, quindi, sfruttando il suo editor e le funzionalità di suggerimenti (le cosiddette *Light Bulb*) e le quick action.

Si supponga di voler creare uno strumento che identifichi il nome delle classi di un progetto, verificando che essi rispettino la convenzione che tale nome inizi con una lettera maiuscola e che, magari, non sia composto solo ed esclusivamente di lettere maiuscole: MiaClasse va bene, ma non sarà valido il nome MIACLASSE. Nel caso in cui si trovino nomi non validi, un apposito Code Fix permetterà di correggerli automaticamente con un clic.

Nei prossimi paragrafi verranno esposti i passi necessari per creare tale strumento.

Scrivere un analizzatore

Visual Studio, una volta installati gli strumenti di .NET Compiler Platform, metterà a disposizione anche il template Analyzer with Code Fix. Creato e dato un nome al nuovo progetto, per esempio ClassNameAnalyzer, e confermato con OK, verrà creata una soluzione composta da tre progetti.

- ClassNameAnalyzer: è il progetto principale che creerà l'assembly portable contententе l'analizzatore di codice.
- ClassNameAnalyzer.Vsix: è il progetto che permetterà di inglobare l'assembly precedente in un'estensione di Visual Studio. Questo progetto inoltre ci consentirà di eseguire il debug in una seconda istanza di Visual Studio.
- ClassNameAnalyzer.Test: è un progetto di test unitario, che permetterà di scrivere dei test per verificare il corretto funzionamento dell'analizzatore, senza necessità di avviare una seconda istanza dell'ambiente.

Compilando la soluzione si otterrà un assembly DLL, e due differenti pacchetti di distribuzione dell'analizzatore: un pacchetto NuGet package (.nupkg) che potrà essere installato in un differente progetto, e una estensione VSIX che eseguirà l'analizzatore nell'IDE e quindi in tutti i progetti aperti al suo interno.

Prima di procedere, assicuratevi che il progetto di avvio sia quello VSIX, in maniera che, quando si eseguirà il debug della soluzione, l'analizzatore verrà caricato ed eseguito in un'altra istanza di Visual Studio con l'estensione installata.

All'interno del file DiagnosticAnalyzer.cs, nel progetto principale, è presente la classe che rappresenta l'analizzatore, derivata da DiagnosticAnalyzer.

Il metodo Initialize permette di indicare su quali elementi agirà l'analizzatore, in questo caso il simbolo è un tipo denominato. Per esempio, il nome di una classe:

```
public override void Initialize(AnalysisContext context)
{
context.RegisterSymbolAction(AnalyzeSymbol, SymbolKind.NamedType);
}
```

La classe contiene poi il metodo AnalyzeSymbol, all'interno del quale viene eseguita l'analisi vera e propria di un simbolo. In questo metodo si può semplicemente modificare l'istruzione if, in maniera da identificare che il nome del tipo che si sta analizzando inizi per lettera minuscola, oppure che non contenga lettere minuscole, cioè che sia tutto in maiuscolo. In questo caso viene creato e visualizzato il relativo avviso:

```
private static void AnalyzeSymbol(SymbolAnalysisContext context)
{
var namedTypeSymbol = (INamedTypeSymbol)context.Symbol;
if (char.IsLower(namedTypeSymbol.Name.ToCharArray().First()) ||
(namedTypeSymbol.Name.Length > 1 &&
!namedTypeSymbol.Name.ToCharArray().Any(char.IsLower)))
{
var diagnostic = Diagnostic.Create(Rule, namedTypeSymbol.Locations[0], namedTypeSymbol.Name);

context.ReportDiagnostic(diagnostic);
}
}
```

Il progetto contiene poi un file di risorse con le stringhe da utilizzare per i messaggi visualizzati in Visual Studio: qui è necessario inserire le stringhe che si desidera visualizzare nei suggerimenti.

Eseguendo ora il debug, si avvierà come detto una seconda istanza dell'IDE. All'interno di questa, create un nuovo progetto, per esempio una applicazione Console.

Provate a modificare il nome della classe Program in program oppure in PROGRAM e vedrete apparire, in corrispondenza del nome della classe, il messaggio di avviso.

Il codice dell'Analyzer può essere normalmente debuggato, per esempio inserendo un breakpoint nel metodo Initialize e poi modificando il nome di una classe all'interno della seconda istanza di Visual Studio.

Correggere gli errori

Dopo aver identificato le possibili situazioni da correggere, il secondo compito è svolto da un oggetto CodeFixProvider che apporterà le modifiche al codice sorgente.

Il file CodeFixProvider.cs contiene la classe ClassNameAnalyzerCodeFixProvider, il cui metodo RegisterCodeFixesAsync si occupa di registrare le possibili azioni di correzione per ogni fix suggerito.

Ecco un metodo per correggere il nome di un tipo, trasformandone la prima lettera in maiuscolo e il resto in minuscolo:

```
private async Task<Solution> SetPascalNameAsync(Document document, TypeDeclarationSyntax typeDecl,
CancellationToken cancellationToken)
{
var identifierToken = typeDecl.Identifier;
var newName = Char.ToUpper(identifierToken.Text[0]).ToString();
if (identifierToken.Text.Length > 1)
newName += identifierToken.Text.Substring(1).ToLowerInvariant();

// ottiene il simbolo che rappresenta il tipo da rinominare.
var semanticModel = await document.GetSemanticModelAsync(cancellationToken);
var typeSymbol = semanticModel.GetDeclaredSymbol(typeDecl, cancellationToken);

// Produce una nuova soluzione con il tipo rinominato.
var originalSolution = document.Project.Solution;
var optionSet = originalSolution.Workspace.Options;
var newSolution = await Renamer.RenameSymbolAsync(document.Project.Solution, typeSymbol, newName, optionSet,
cancellationToken).ConfigureAwait(false);
return newSolution;
}
```

Il metodo prima ricava il simbolo corrispondente al tipo da rinominare, tramite il semantic model del documento corrente, e poi utilizza anche la classe Renamer, che in pratica si occupa di correggere tutti i riferimenti al tipo e crea una nuova soluzione aggiornata.

Per utilizzare il metodo, basta registrarlo all'interno di RegisterCodeFixAsync:

```
context.RegisterCodeFix(CodeAction.Create(
title: title,
createChangedSolution: c => SetPascalNameAsync(context.Document, declaration, c),
equivalenceKey: title),
diagnostic);
```

Ora, avviando nuovamente la soluzione e provando a dare un nome minuscolo a una classe, apparirà il suggerimento di correzione e sarà possibile applicare con un clic la correzione.

Domande di riepilogo

1) Qual è il pacchetto NuGet principale per utilizzare i servizi della .NET Compiler Platform?

a. Microsoft.Roslyn
b. Microsoft.CodeAnalysis
c. Microsoft.Compiler
d. Microsoft.Net.Platform;

2) Quale istruzione permette di ottenere l'albero sintattico a partire da una stringa code contenente del codice sorgente C#?

a. SyntaxTree tree = new SyntaxTree(code);
b. SyntaxTree tree = SyntaxTree.Analyze(code);
c. SyntaxTree tree = new CSharpSyntaxTree(code);
d. SyntaxTree tree = CSharpSyntaxTree.ParseText(code);

3) Qual è la classe che permette di compilare il codice C# e ottenere un assembly?

a. CSharpCompilation
b. Csc
c. Compiler
d. CSharpCompiler

4) Il modello semantico di un programma è ottenibile da un oggetto Compilation. Vero o falso?

5) Quale coppia di classi base è necessaria per implementare un Analyzer with Code Fix per Visual Studio?

a. DiagnosticAnalyzer, DiagnosticFix
b. DiagnosticAnalyzer, Compilation
c. CodeAnalyzer, CodeFixProvider
d. DiagnosticAnalyzer, CodeFixProvider

6) Il debug di un Analyzer with Code Fix può essere eseguito avviando una seconda istanza di Visual Studio. Vero o falso?

Applicazioni pratiche di C#

I campi di applicazione pratica di C# includono lo sviluppo desktop, con le librerie Windows Forms e WPF, oppure Universal Windows App su dispositivi Windows 10, passando al Web con applicazioni ASP.NET e MVC.

Nei capitoli precedenti del libro si è visto il linguaggio C# in tutte le sue sfaccettature: dalla sua sintassi e dalle sue parole chiave ai costrutti per scrivere e controllare il flusso di esecuzione di un programma, passando per la programmazione a oggetti e relativi principi, arrivando anche ad aspetti un po' più avanzati come reflection, gestione delle eccezioni, delegate ed eventi, multithreading e parallelismo, programmazione asincrona, coprendo così tutte le sfaccettature del linguaggio fino alla versione 6.

Fra i vari argomenti, alcuni di essi hanno già dimostrato la loro utilità pratica. Per esempio, LINQ, le classi ADO.NET, le collezioni e i tipi generici permettono quotidianamente agli sviluppatori di svolgere il proprio lavoro nella maniera più efficace e rapida possibile.

Quasi sempre, però, ogni esempio scritto ed eseguito è stato basato su una applicazione di tipo console, cioè eseguibile direttamente dal prompt dei comandi, e che riceve input e stampa l'output sempre al suo interno in forma testuale.

Al giorno d'oggi, la stragrande maggioranza delle applicazioni in commercio e di quelle richieste da un qualsiasi possibile cliente impiega un'interfaccia grafica senza dubbio più elaborata e accattivante, non solo per questioni di bellezza o per stupire l'utilizzatore, ma per essere il più user friendly possibile, in maniera da aumentare la produttività dell'utente ed essere integrata nel sistema su cui viene eseguita.

Quindi, nel quotidiano lavoro, ogni sviluppatore potrà e dovrà cimentarsi nella progettazione e nell'implementazione di interfacce grafiche per sistemi desktop, web, mobile e chi più ne ha più ne metta.

Per tale motivo, in questo ultimo capitolo si proporrà una panoramica dall'aspetto molto pratico, che mostri alcune fra le varie possibilità offerte da C# e .NET per la costruzione dei diversi tipi di applicazioni, senza la pretesa di essere esaustiva (ogni tipologia richiederebbe almeno un testo a essa dedicato).

C# è un linguaggio ormai universale, tanto che, oltre a quelli mostrati in questo capitolo, vi sono altri sistemi, ambiti e piattaforme in cui esso è utilizzato al giorno d'oggi. Fra quelli più interessanti, e non approfonditi di seguito, la possibilità di sviluppare applicazioni per il Cloud, per esempio sulla piattaforma *Azure* di Microsoft, per gli smartphone e i tablet *iOS* e *Android* grazie alla piattaforma *Xamarin*. E, ricordando l'apertura all'open source di Microsoft e l'ormai maturo progetto *Mono*, il supporto anche a sistemi operativi diversi da Windows, come *Linux* e *Mac OS*.

Infine, non possiamo non citare, dato il crescente interesse per la cosiddetta *Internet Of Things*, la possibilità di sviluppare soluzioni per dispositivi con Windows 10 IoT Core: fra gli altri, *Raspberry Pi* e *Arduino*.

Windows Forms

Windows Forms è il nome dell'API per la programmazione di applicazioni con interfaccia grafica per Windows incluse nel framework .NET fin dalla prima versione. Con Windows Forms è possibile creare applicazioni *smart client*, che posseggono cioè una *GUI* (*Graphical User Interface*) e che vengono eseguite in genere all'interno di sistemi operativi della famiglia Windows, quindi come applicazioni a finestre all'interno del desktop.

NOTA	Le librerie Windows Forms non sono un'implementazione esclusiva di Microsoft Windows: ne esiste infatti una versione alternativa creata all'interno del progetto open source Mono, utilizzabile per applicazioni che girano su tale piattaforma e quindi in ogni sistema operativo che la supporta, inclusi Linux e Mac OS.

Infatti, il nome di Windows Forms deriva dal termine *Form*, il quale indica la classica finestra di un'applicazione sulla cui superficie possono essere ospitati altri elementi grafici, che comporranno l'interfaccia utente.

Questi elementi sono anche detti *controlli* e il loro compito è di visualizzare dati di output, o di accettare dati in input, interagendo con l'utente e rispondendo alle sue azioni, per mezzo di eventi.

Le librerie Windows Forms contengono decine di controlli e componenti che possono essere utilizzati per realizzare interfacce grafiche per applicazioni desktop.

Sono poi utilizzabili anche librerie di terze parti, fornite da produttori specializzati nello sviluppo di suite di controllo commerciali, oppure disponibili gratuitamente, permettendo di integrare nuovi controlli nell'interfaccia: per realizzare compiti specifici in particolari ambiti, per esempio.

Infine, qualora non fosse disponibile un controllo che implementi un compito specifico, oppure se si volesse personalizzare l'aspetto e il comportamento in base alle proprie esigenze o a quelle dei propri clienti, sarebbe comunque possibile implementare dei controlli personalizzati.

Creazione di applicazioni Windows Forms

Visual Studio è uno strumento fondamentale per costruire applicazioni basate su Windows Forms, anche se nulla vieta di creare tutto il codice necessario manualmente con un editor di testo, compilandolo eventualmente con il compilatore csc.

Dal punto di vista del compilatore, infatti, la differenza fra un'applicazione console e una Windows è questione di uno switch, quello chiamato target o abbreviato in t:

```
csc program.cs /target:exe
```

```
csc program.cs /target:winexe
```

L'unica differenza fra le due modalità è che con la prima, appena lanciato l'eseguibile, viene creata una finestra di console, mentre il secondo modo di compilazione evita di crearla in quanto si aspetta che l'applicazione mostri come punto di ingresso una finestra.

NOTA	In Visual Studio, il tipo di output o target può essere modificato all'interno della finestra delle proprietà del progetto, più esattamente nella scheda Application di quest'ultima.

Per creare invece un'applicazione per Windows basata sulla libreria Windows Forms utilizzando Visual Studio, basta selezionare il template Windows Forms Application all'interno della finestra di creazione di un nuovo progetto, esattamente nella categoria Visual C# -> Windows (vedere la Figura 17.1).

La finestra precedente permette di selezionare la versione del .NET Framework da utilizzare per la compilazione: potrete notare che un progetto Windows Forms è realizzabile anche con .NET 2.0 (in realtà sarebbe sufficiente anche la 1.0, ma in Visual Studio 2015 non è più disponibile).

New Project

Recent

Installed

Templates

Visual C#

Windows

Universal

Windows 8

Classic Desktop

Web

Office/SharePoint

Android

Cloud

Documentation

Extensibility

iOS

LightSwitch

Mobile Apps

Silverlight

Test

WCF

Windows Driver

Online

.NET Framework 4.5 | Sort by: Default | Search Installed Templates (Ctrl+E)

Blank App (Universal Windows)	Visual C#
Windows Forms Application	Visual C#
WPF Application	Visual C#
Console Application	Visual C#
Shared Project	Visual C#
Class Library	Visual C#
Class Library (Portable)	Visual C#
Class Library (Universal Windows)	Visual C#
Windows Runtime Component (Universal Windows)	Visual C#
Telerik WPF Application	Visual C#

Type: Visual C#

A project for creating an application with a Windows Forms user interface

Click here to go online and find templates.

Name: WindowsFormsApplication1

Location: C:\Users\Antonio\OneDrive\Scrittura libri\CSharp6\programmare_con_csharp6\Programmare con

Browse...

Solution: Create new solution

Solution name: WindowsFormsApplication1

☑ Create directory for solution

☐ Add to source control

OK | Cancel

Figura 17.1 – Creazione di un progetto Windows Forms in Visual Studio.

Selezionando una versione successiva del framework noterete altre tipologie di progetti nella categoria Windows: *WPF Application*, per esempio, di cui parleremo più avanti.

Assegnato il nome all'applicazione e premuto il tasto OK, verrà creato lo scheletro di un'applicazione Windows.

Le finestre

L'applicazione creata da Visual Studio con il template Windows Forms Application contiene una singola finestra, istanza della classe Form, pronta a ospitare gli altri controlli che costituiranno l'interfaccia.

Ogni applicazione Windows Forms ha almeno una finestra principale, detta anche *Main Form*: essa costituisce la *madre*, o *proprietaria*, di tutte le altre finestre che verranno eventualmente create durante il ciclo di vita dell'applicazione; inoltre, quando essa verrà chiusa, anche l'intera applicazione terminerà la sua esecuzione.

Ciò si evince anche dalle istruzioni nel metodo Main che permettono l'esecuzione dell'applicazione. Le potete osservare aprendo il file *Program.cs*:

```
static void Main()
{
Application.EnableVisualStyles();
Application.SetCompatibleTextRenderingDefault(false);
Application.Run(new Form1());
}
```

Tralasciando le prime due righe, si noti come l'applicazione venga lanciata eseguendo il metodo statico Run della classe Application, a cui viene passato come argomento una nuova istanza della form principale dell'applicazione, denominata *Form1*, che verrà visualizzata all'interno del desktop di Windows.

La classe Application, contenuta nel namespace System.Windows.Forms, espone proprietà e metodi statici per la gestione di un'applicazione.

Per provare il funzionamento dell'applicazione basta avviare il debug da Visual Studio: dopo pochi secondi apparirà una finestra vuota o con i controlli eventualmente inseriti da voi sulla superficie della finestra nel designer.

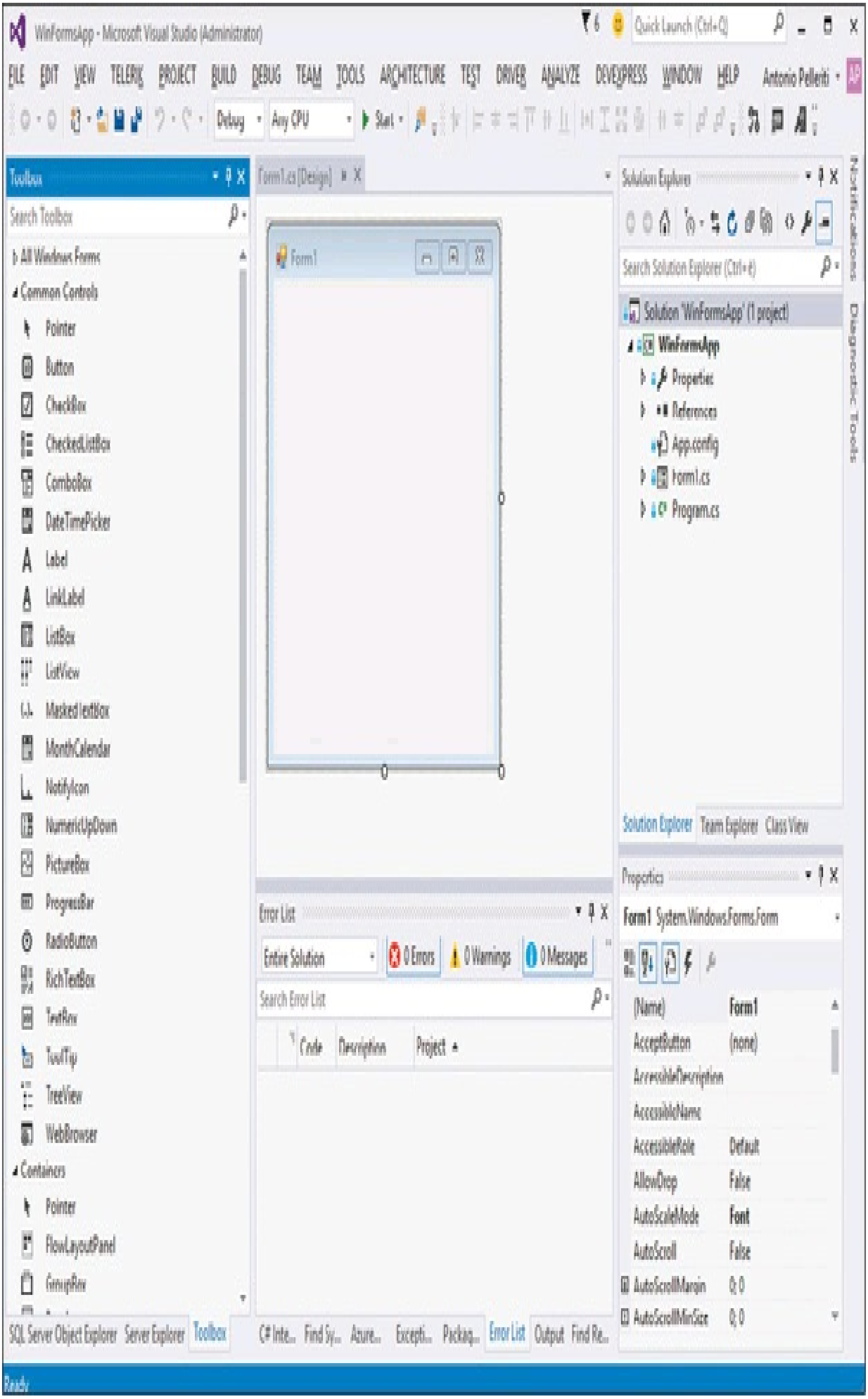

WinFormsApp - Microsoft Visual Studio (Administrator)
Quick Launch (Ctrl+Q)
FILE EDIT VIEW TELERIK PROJECT BUILD DEBUG TEAM TOOLS ARCHITECTURE TEST DRIVER ANALYZE DEVEXPRESS WINDOW HELP
Antonio Pellerti
Debug
Any CPU
Start
Toolbox
Search Toolbox
All Windows Forms
Common Controls
Pointer
Button
CheckBox
CheckedListBox
ComboBox
DateTimePicker
Label
LinkLabel
ListBox
ListView
MaskedTextBox
MonthCalendar
NotifyIcon
NumericUpDown
PictureBox
ProgressBar
RadioButton
RichTextBox
TextBox
ToolTip
TreeView
WebBrowser
Containers
Pointer
FlowLayoutPanel
GroupBox
Form1.cs [Design]
Form1
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'WinFormsApp' (1 project)
WinFormsApp
Properties
References
App.config
Form1.cs
Program.cs
Solution Explorer
Team Explorer
Class View
Error List
Entire Solution
0 Errors
0 Warnings
0 Messages
Search Error List
Code
Description
Project
Properties
Form1 System.Windows.Forms.Form
(Name) Form1
AcceptButton (none)
AccessibleDescription
AccessibleName
AccessibleRole Default
AllowDrop False
AutoScaleMode Font
AutoScroll False
AutoScrollMargin 0; 0
AutoScrollMinSize 0; 0
SQL Server Object Explorer
Server Explorer
Toolbox
Error List
Output
Ready

Figura 17.2 – Una Windows Form all'interno di Visual Studio.

Una volta chiusa la finestra, l'applicazione verrà, come detto, terminata e Visual Studio sarà a sua volta scollegato dal debug della stessa.

Inoltre, Visual Studio permette di passare dal designer visuale al codice in ogni momento. Per esempio, basta fare clic con il pulsante destro sulla finestra in progettazione e selezionare dal menu contestuale la voce View Code oppure premere F7; al contrario, dalla finestra di codice di una form o di un controllo, è possibile passare al designer con il comando View Designer o con SHIFT+F7.

Per personalizzare l'interfaccia dell'applicazione, aggiungendo dei controlli, basta trascinarli dalla toolbox dell'IDE sulla superficie della finestra in costruzione, ridimensionarli o spostarli a proprio piacimento, impostarne le varie proprietà ed eventualmente gestirne gli eventi, che dipendono naturalmente dalla tipologia di controllo.

L'impostazione delle proprietà di una finestra o del controllo selezionato nell'area di design avviene per mezzo di un'apposita finestra, detta appunto finestra delle proprietà o Property Window. Qualora non fosse visibile, può essere attivata dal menu View -> Properties Window. Tramite tale finestra è possibile personalizzare l'aspetto di ogni controllo o componente di interfaccia grafica dell'applicazione, che è un processo molto importante di ogni tipologia di applicazione desktop.

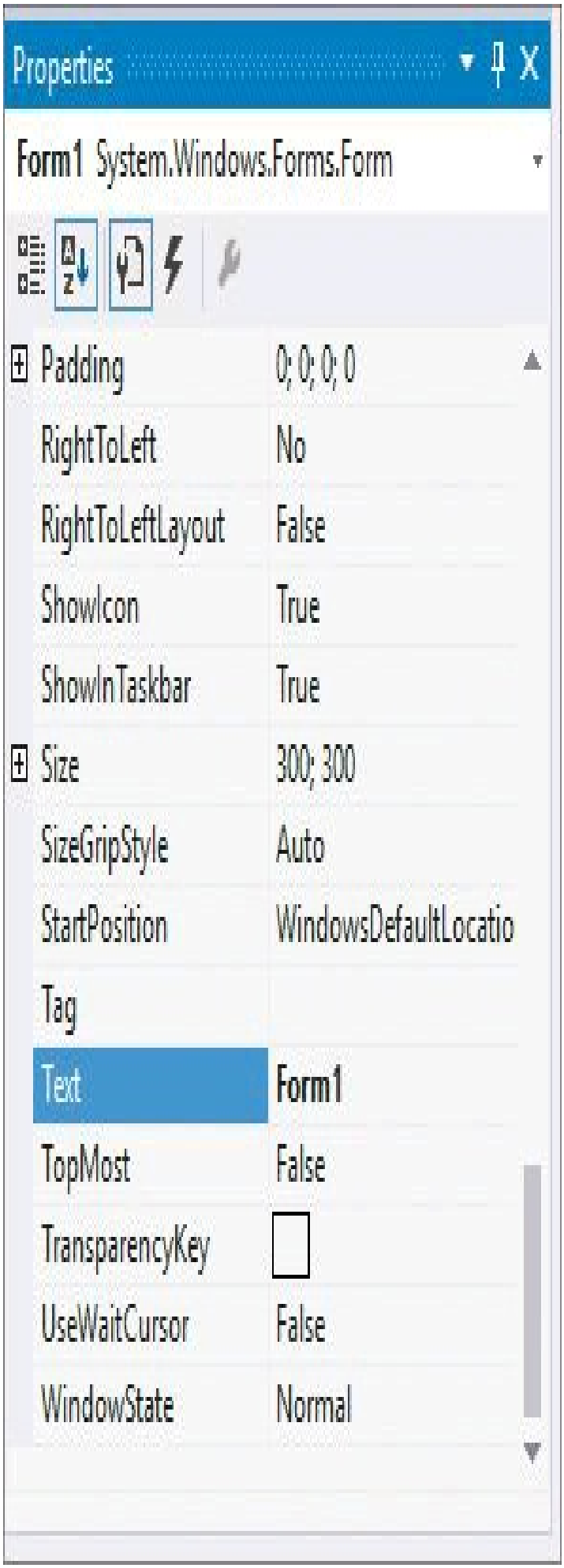

Properties
Form1 System.Windows.Forms.Form
Padding 0; 0; 0; 0
RightToLeft No
RightToLeftLayout False
ShowIcon True
ShowInTaskbar True
Size 300; 300
SizeGripStyle Auto
StartPosition WindowsDefaultLocatio
Tag
Text Form1
TopMost False
TransparencyKey
UseWaitCursor False
WindowState Normal

Figura 17.3 – La finestra delle proprietà in Visual Studio.

Le proprietà sono visualizzabili in due modalità: raggruppate per categoria o in ordine alfabetico, attivabili per mezzo delle icone in alto a sinistra, nella finestra delle proprietà

Un'applicazione in genere può visualizzare diverse form, oltre a quella principale di avvio. Quest'ultima può fungere da contenitore delle finestre figlie, quando l'applicazione utilizza la modalità cosiddetta *MDI* (*Multiple Document Interface*), come fa Microsoft Word, che permette di aprire più documenti contemporaneamente.

L'altra modalità è invece quella in cui si apriranno altre finestre, separate e al di fuori di quella principale.

Nel primo caso, la finestra che funge da contenitore deve possedere la proprietà IsMdiContainer impostata al valore true, cosa che è fattibile mediante la finestra delle proprietà o sempre via codice:

```
this.IsMdiContainer=true;
```

Ogni finestra figlia, aperta all'interno dell'applicazione per mezzo del metodo Show, avrà la proprietà MdiParent impostata con il riferimento alla finestra madre:

```
Form figlia=new Form2();
figlia.MdiParent=this;
figlia.Show();
```

Per mostrare invece una finestra in maniera modale, cioè che richiede all'utente la sua chiusura prima di interagire con le altre finestre, bisogna utilizzare il metodo ShowDialog:

```
figlia.ShowDialog();
```

In molti casi è necessario passare parametri o oggetti da una finestra all'altra. Esistono diversi modi per farlo. Per esempio, alla finestra che deve accettare tali parametri è possibile aggiungere dei costruttori con degli argomenti, quindi istanziarle passando i parametri stessi:

```
private void btShowChild_Click(object sender, EventArgs e)
{
Form form2=new Form2("parametro");
form2.Show();
}
```

Un'altra possibilità è implementare proprietà o metodi, da impostare o invocare prima del metodo Show o ShowDialog, e quindi all'interno di questi impostare eventualmente dei campi di classe che costituiscono lo stato della finestra.

Finestre di dialogo predefinite

Le classi Windows Forms contengono diverse finestre di dialogo predefinite, utilizzabili per gestire aspetti comuni come la selezione di un colore, l'apertura o il salvataggio di un file e così via.

In generale, una finestra di dialogo, visualizzata per mezzo del metodo ShowDialog, restituisce un valore di tipo DialogResult, indicante la scelta eseguita dall'utente che ha portato alla chiusura della finestra stessa. Per esempio, una MessageBox può visualizzare dei pulsanti e restituire un valore indicante quale di essi è stato cliccato per chiuderla. Il valore viene restituito dal metodo Show:

```
private void button2_Click(object sender, EventArgs e)
{
DialogResult result = MessageBox.Show("continuare?", "scegli", MessageBoxButtons.OKCancel, MessageBoxIcon.Question);
if(result== System.Windows.Forms.DialogResult.OK)
{
label1.Text = "Hai cliccato OK";
}
else label1.Text = "Hai cliccato annulla";
}
```

Le finestre di dialogo predefinite sono dei componenti selezionabili dalla toolbox, ma che non appariranno sulla superficie di design della finestra. Il componente comparirà invece sulla barra posta subito al di sotto del designer, dove sarà selezionabile come un qualsiasi altro controllo, in maniera da impostarne le proprietà e gestire gli eventi. Per esempio, se si necessita di aggiungere all'applicazione la funzione di scelta e apertura di un file dal disco, è possibile utilizzare il componente OpenFileDialog. Tali finestre di dialogo sono comunque istanziabili e utilizzabili anche via codice, istanziando le rispettive classi:

```
OpenFileDialog ofd = new OpenFileDialog();
ofd.Filter = "File di test (*.txt)|*.txt|Tutti i file (*.*)|*.*";
ofd.InitialDirectory = Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
if(ofd.ShowDialog()== System.Windows.Forms.DialogResult.OK)
{
string filename = ofd.FileName;
MessageBox.Show("Hai selezionato il file "+filename);
}
```

Le finestre predefinite contenute nelle classi Windows Forms sono le seguenti:

- ColorDialog – selezione di un colore;
- OpenFileDialog – apertura di un file;
- SaveFileDialog – salvataggio di un file;
- FolderBrowserDialog – selezione di una cartella;
- FontDialog – selezione di uno dei font installati nel sistema;
- PageSetupDialog – finestra di impostazione delle proprietà di stampa;
- PrintDialog – finestra di selezione della stampante;

- PrintPreviewDialog – finestra di anteprima di stampa.

Ognuna di esse ha delle proprietà specifiche per ottenere gli eventuali dati selezionati o impostati a seguito del loro utilizzo. Per esempio, nel precedente spezzone di codice viene ricavato il nome del file selezionato.

I controlli

È possibile aggiungere a una Windows Form due tipi di controlli: quelli grafici visualizzati direttamente sulla superficie della finestra, come pulsanti ed etichette, e i componenti che invece non hanno una interfaccia visibile all'utente.

Il modo più semplice per aggiungere dei controlli a una finestra, in modalità visuale, è utilizzare la casella degli strumenti, o toolbox. Provate, per esempio, a trascinare sulla Form1 un pulsante e a cambiare il testo dello stesso, modificando la proprietà Text, per esempio in "Click Me".

Spesso può essere necessario modificare le proprietà di un controllo direttamente da codice, e poiché ogni controllo posizionato sulla finestra, oltre alla finestra stessa, viene istanziato all'interno del codice di una classe parziale, in un file denominato *Form1.designer.cs*, ogni istanza è anche utilizzabile direttamente da codice C#. Per esempio, se si vuole modificare il titolo della finestra da codice C#, si potrebbe scrivere all'interno del costruttore il seguente codice:

```
public Form1()
{
InitializeComponent();
this.Text = "Hello Windows";
}
```

Il metodo InitializeComponent viene implementato automaticamente da Visual Studio, all'interno del file *Form1.designer.cs*. Esso contiene tutto il codice necessario a realizzare l'interfaccia grafica corrispondente a quella creata nel designer visuale. Per esempio, il metodo subito dopo la creazione del progetto contiene il codice necessario alla configurazione delle proprietà di Form1:

```
private void InitializeComponent()
{
this.components = new System.ComponentModel.Container();
this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
this.Text = "Form1";
}
```

Una volta aggiunti controlli sulla superficie della finestra, potete provare a riesaminare il file *Form1.designer.cs* per capire come le modifiche si riflettono su di esso e comprendere il funzionamento del tutto. Fra l'altro, ciò dimostra come un'applicazione con interfaccia grafica

Windows Forms possa essere realizzata totalmente con la scrittura di codice manuale (anche se, per fortuna, c'è Visual Studio!).

Windows Forms contiene decine di classi che definiscono vari controlli utilizzabili nella costruzione di un'interfaccia grafica. All'interno della toolbox di Visual Studio, potete vedere i controlli più utilizzati suddivisi in categorie:

- *Common Controls* – controlli comuni come campi di testo, pulsanti, caselle di selezione ecc.;
- *Containers* – contenitori che permettono di gestire e raggruppare altri controlli;
- *Menus & Toolbars* – controlli che permettono di aggiungere barre di menu, di strumenti, di stato e così via;
- *Data* – controlli per lavorare e visualizzare sorgenti di dati, per esempio griglie e grafici;
- *Components* – componenti che non appaiono sulla superficie della finestra, ma permettono di interagire con il sistema operativo o di eseguire funzioni particolari, come timer, utilizzo di porte seriali ecc.;
- *Printing* – oggetti per aggiungere funzioni di stampa alle applicazioni;
- *Dialogs* – finestre di dialogo predefinite del sistema operativo;
- *Reporting* – visualizzazione di report di stampa.

La toolbox può essere estesa con controlli di terze parti, gratuiti o acquistabili, forniti all'interno di suite specializzate, che implementano funzionalità particolari o che migliorano quelle standard fornite con .NET.

Eventi

Per ogni controllo aggiunto alla finestra, o per la finestra stessa, è possibile gestire i vari eventi definiti nella rispettiva classe.

Per esempio, un pulsante prevede, fra gli altri eventi, la gestione di quelli di interazione con il mouse, come il clic su di esso. Quindi, all'interno della finestra delle proprietà è possibile aprire la sezione degli eventi cliccando sull'icona con l'immagine di un fulmine e scegliere l'evento da gestire, per esempio il suddetto Click. A questo punto, basta fare doppio clic sulla casella vuota a destra del nome dell'evento, oppure scrivere direttamente il nome del metodo di gestione e premere Invio. Visual Studio aprirà il codice della form nell'editor di testo, creando lo scheletro dell'event handler relativo all'evento in oggetto. Nel file *Form1.designer.cs* verrà creata l'istruzione che imposta e aggiunge il gestore dell'evento, nel seguente modo:

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

Lo stesso codice potrebbe essere aggiunto manualmente, per esempio all'interno del costruttore della classe, facendolo così eseguire a runtime nel momento in cui lo stesso costruttore viene invocato.

Nel file *Form1.cs* verrà invece implementato il corpo vuoto del metodo:

```
private void button1_Click(object sender, EventArgs e)
{
}
```

All'interno del metodo, naturalmente, andrà inserito il codice da eseguire al clic sul pulsante.

Ipotizziamo di voler visualizzare una classica finestra di messaggio utilizzando la classe MessageBox, utile in molti casi nella programmazione Windows Forms anche a scopi di debug:

```
MessageBox.Show("Hai cliccato il pulsante");
```

Gli argomenti dei metodi di gestione in genere sono un generico oggetto sender e uno contenente eventuali parametri aggiuntivi.

Il primo rappresenta l'oggetto che ha generato l'evento, utile se si utilizza lo stesso gestore per più controlli e si vuole individuare quello che ha scatenato l'evento.

Il secondo argomento deriva, in genere, dalla classe EventArgs e contiene eventuali dati associati all'evento. Nel caso di un evento associato al mouse, un oggetto MouseEventArgs contiene dati sulla posizione o sul tasto del mouse cliccato.

WPF

WPF (*Windows Presentation Foundation*) è un framework di presentazione per la creazione di GUI, visivamente ricche e accattivanti, per applicazioni Windows oppure da ospitare all'interno di un browser web.

La prima versione del nuovo sottosistema grafico ha visto la luce con l'introduzione di .NET 3.0.

Esso può essere considerato uno dei principali cambiamenti di tecnologia per lo sviluppo di interfacce grafiche degli ultimi anni, in quanto utilizza DirectX per il rendering di contenuto vettoriale, consentendo lo sfruttamento dell'accelerazione hardware quando possibile, permettendo così la realizzazione di interfacce veloci, scalabili e indipendenti dalla risoluzione dello schermo, con animazioni realizzabili in maniera molto immediata.

Quindi, a differenza di Windows Forms, le classi WPF non sono dei semplici wrapper .NET delle API native di Windows e non si basano sul precedente sistema *GDI* (*Graphics Device Interface*).

A partire da Windows Vista, le librerie runtime di WPF sono incluse nel sistema operativo, quindi danno certamente il meglio di sé nelle versioni più nuove di Windows stesso.

Anche per WPF vengono periodicamente rilasciati aggiornamenti delle librerie e nuovi controlli, in genere in concomitanza con le nuove versioni del .NET Framework.

Finora sono state rilasciate cinque versioni principali di WPF: WPF 3.0 (nel novembre 2006), WPF 3.5 (nel 2007), WPF 3.5 SP1 (nel 2008), WPF 4 (nel 2010) e WPF 4.5 (nell'agosto 2012).

Creazione di un'applicazione WPF per Windows

Per creare un'applicazione con interfaccia grafica WPF, basta creare un nuovo progetto in Visual Studio 2015 scegliendo il template WPF Application.

A differenza di Windows Forms, WPF è disponibile solo a partire da .NET 3.0, quindi selezionando un framework precedente il template non sarà visualizzato.

Una volta confermata la creazione, verrà generata l'applicazione con un'unica finestra, che sarà una istanza della classe Window.

A differenza di Windows Forms, il designer di Visual Studio permetterà la gestione della parte visuale vera e propria e in parallelo del codice XAML (vedremo nel prossimo paragrafo di

cosa si tratta), come mostrato in Figura 17.4.

WpfApplication1 - Microsoft Visual Studio (Administrator)

FILE EDIT VIEW TELERIK PROJECT BUILD DEBUG TEAM DESIGN FORMAT TOOLS ARCHITECTURE TEST DRIVER ANALYZE DEVEXPRESS WINDOW HELP

Antonio Pelleriti

Debug · Any CPU · Start

Toolbox

Search Toolbox

Common WPF Controls

- Pointer
- Border
- Button
- CheckBox
- ComboBox
- DataGrid
- Grid
- Image
- Label
- ListBox
- RadioButton
- Rectangle
- StackPanel
- TabControl
- TextBlock
- TextBox

All WPF Controls

- Pointer
- Border
- Button
- Calendar
- Canvas
- CheckBox
- ComboBox
- ContentControl
- DataGrid
- DatePicker
- DockPanel
- DocumentViewer
- Ellipse
- Expander

MainWindow.xaml | MainWindow.xaml.cs

57.71% · Design · XAML

Window · Window

```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xa
        xmlns:d="http://schemas.microsoft.com/expression/bl
        xmlns:mc="http://schemas.openxmlformats.org/markup-
        xmlns:local="clr-namespace:WpfApplication1"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

Solution Explorer

Search Solution Explorer (Ctrl+é)

Solution 'WpfApplication1' (1 project)

- WpfApplication1
  - Properties
  - References
  - App.config
  - App.xaml
  - MainWindow.xaml

Solution Explorer | Team Explorer | Class View

Properties

Notifications | Diagnostic Tools

SQL Server O... | Server Explorer | Toolbox | Document O...

C# Interactive | Find Symbol Results | Azure App Service Activity | Exception Settings | Package Manager Console | Error List | Output | Find Results 1

Ready

Figura 17.4 – Designer di un'applicazione WPF in Visual Studio.

Visual Studio crea una soluzione con due file .xaml, App.xaml e MainWindow.xaml, e i rispettivi file di code behind App.xaml.cs e MainWindow.xaml.cs.

App.xaml rappresenta il punto d'ingresso dell'applicazione e contiene le risorse utilizzabili globalmente da quest'ultima e la definizione della finestra principale:

```
<Application x:Class="WpfApplication1.App"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:local="clr-namespace:WpfApplication1"
StartupUri="MainWindow.xaml">
<Application.Resources>
</Application.Resources>
</Application>
```

L'attributo StartupUri contiene il nome della finestra principale che verrà visualizzata all'avvio dell'applicazione.

NOTA	Dove si trova il metodo Main dell'applicazione WPF? Abbiamo imparato che ogni programma C# deve avere un punto di ingresso, definito metodo Main, da cui inizia la sua esecuzione. Nella soluzione dell'applicazione WPF generata dal template di Visual Studio non vi è traccia di questo metodo. In realtà, esso è presente ed è generato in modo automatico all'interno di un file *App.g.cs*, che potete trovare nella cartella obj/[Debug/Release] una volta compilata l'applicazione.

L'attributo x:Class, invece, indica il nome della classe C# che contiene l'implementazione dell'applicazione stessa. Aprendo il file App.xaml.cs, troverete la classe App, derivata dalla classe Application.

La classe System.Windows.Application rappresenta una istanza globale di un'applicazione WPF; essa espone un metodo Run per il suo avvio e una serie di eventi e proprietà.

Gli altri attributi, del tipo xmlns, definiscono invece i namespace XML all'interno di un file XAML, e hanno lo stesso scopo e significato dei namespace visti in C#, cioè sono equivalenti all'istruzione using, ma in questo caso il nome del namespace è indicato in formato URI. Per esempio, la prima dichiarazione xmlns:

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

indica il namespace generale del framework XAML di WPF. In tal modo, tutti i controlli in esso contenuti saranno istanziabili tramite il relativo elemento senza nessun prefisso.

Analogamente, gli elementi contenuti nel namespace indicato con il prefisso xmlns:x saranno istanziabili utilizzando appunto il prefisso x. Per esempio, l'attributo x:Class visto sopra è definito in tale namespace.

Il secondo file, *MainWindow.xaml*, contiene la definizione della finestra principale:

```
<Window x:Class="WpfApplication1.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:local="clr-namespace:WpfApplication1"
mc:Ignorable="d"
Title="MainWindow" Height="350" Width="525">
<Grid>
</Grid>
</Window>
```

All'interno dell'elemento Window, che definisce la finestra vera e propria, vi è un elemento vuoto Grid, che rappresenta il contenitore principale dei controlli che verranno eventualmente aggiunti.

Ecco invece il contenuto del file *MainWindow.xaml.cs*, con la definizione della classe MainWindow:

```
namespace WpfApplication1
{
/// <summary>
/// Interaction logic for MainWindow.xaml
/// </summary>
public partial class MainWindow : Window
{
public MainWindow()
{
InitializeComponent();
}
}
}
```

Avviando l'applicazione, apparirà la finestra principale vuota, visto che non abbiamo ancora aggiunto nulla al codice XAML.

XAML

La rivoluzione principale apportata da WPF è l'utilizzo di un linguaggio dichiarativo chiamato XAML (si pronuncia *zam'l*), acronimo di *eXtensible Application Markup Language*, che è una sorta di dialetto di XML con cui è possibile definire e sviluppare i vari elementi che costituiscono l'interfaccia grafica di un'applicazione, in maniera totalmente indipendente dalla logica che vi sta dietro.

Allo stesso tempo, per questa sua natura, è possibile lavorare all'interfaccia in strumenti appositi, come *Microsoft Expression Blend*, mentre uno sviluppatore C# lavora alla parte di logica vera e propria.

Per molti versi, quindi, il suo ruolo è analogo all'HTML utilizzato per lo sviluppo della parte grafica delle web form di ASP.NET: anche qui, come vedremo più avanti, il code behind, indipendente dall'interfaccia, può essere scritto in un linguaggio come C#.

Ogni elemento dichiarato in XAML, per mezzo di codice di markup, tag, attributi e così via, viene istanziato come un normale oggetto .NET con cui è quindi possibile interagire da normale codice scritto in C#. Viceversa, è possibile creare delle classi in C# e poi dichiararle e accedere alle loro istanze via XAML.

Mentre in Windows Forms i controlli e i componenti aggiunti alla GUI dal designer di Visual Studio corrispondono a normale codice C#, generato automaticamente dall'IDE in una classe parziale, il markup di XAML viene semplicemente serializzato e compilato in un assembly managed, e quindi caricato quando è necessario generare l'interfaccia grafica dell'applicazione stessa. Naturalmente, anche l'editor visuale di Visual Studio è stato aggiornato per permettere il disegno di interfacce XAML; allo stesso modo, come accennato prima, è possibile progettarle mediante l'editor di Microsoft Expression Blend, specializzato in tale ambito.

Fra le altre caratteristiche, grazie ancora alla separazione fra aspetto e logica, è possibile cambiare e modificare agevolmente il look di un qualsiasi controllo. In maniera analoga agli stili CSS e agli Skin utilizzati per definire dei temi per il Web, in XAML è possibile creare dei template da applicare ai controlli delle proprie applicazioni.

XAML è un protagonista primario dello sviluppo .NET anche al di fuori da WPF. Infatti, esso viene utilizzato fra gli altri ambiti anche nello sviluppo di applicazioni per lo store Windows 8.x (le cosiddette *Windows Store App*), di app per il sistema operativo mobile Windows Phone e per applicazioni Silverlight eseguibili per mezzo di un plugin all'interno di un browser Internet (sebbene quest'ultima tecnologia sia stata quasi abbandonata, con grande rammarico del sottoscritto).

Sintassi ed elementi XAML

XAML consente di istanziare oggetti .NET mediante la sintassi dichiarativa del suo codice di markup. In questo modo, gli elementi XML aggiunti al codice XAML rappresenteranno degli oggetti, mentre i relativi attributi rappresentano e permettono di impostare le proprietà.

Il seguente frammento di XAML mostra un esempio di codice per la dichiarazione di un normale pulsante all'interno della finestra principale dell'applicazione:

```
<Window x:Class="WpfApplication1.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="MainWindow" Height="350" Width="525">
<Grid>
<Button Name="bt" Width="100" Height="30">
Click me!
</Button>
</Grid>
</Window>
```

L'elemento Button istanzierà un oggetto di classe System.Windows.Controls.Button con nome bt e con larghezza e altezza impostate mediante le relative proprietà Width e Height.

Il pulsante è contenuto all'interno dell'elemento Grid, in quanto in WPF è sempre necessario avere un contenitore all'interno del quale definire il layout. La classe Grid permette di creare un layout a griglia, mediante righe e colonne. In questo caso, in maniera sottintesa, la griglia ha un'unica riga e un'unica colonna.

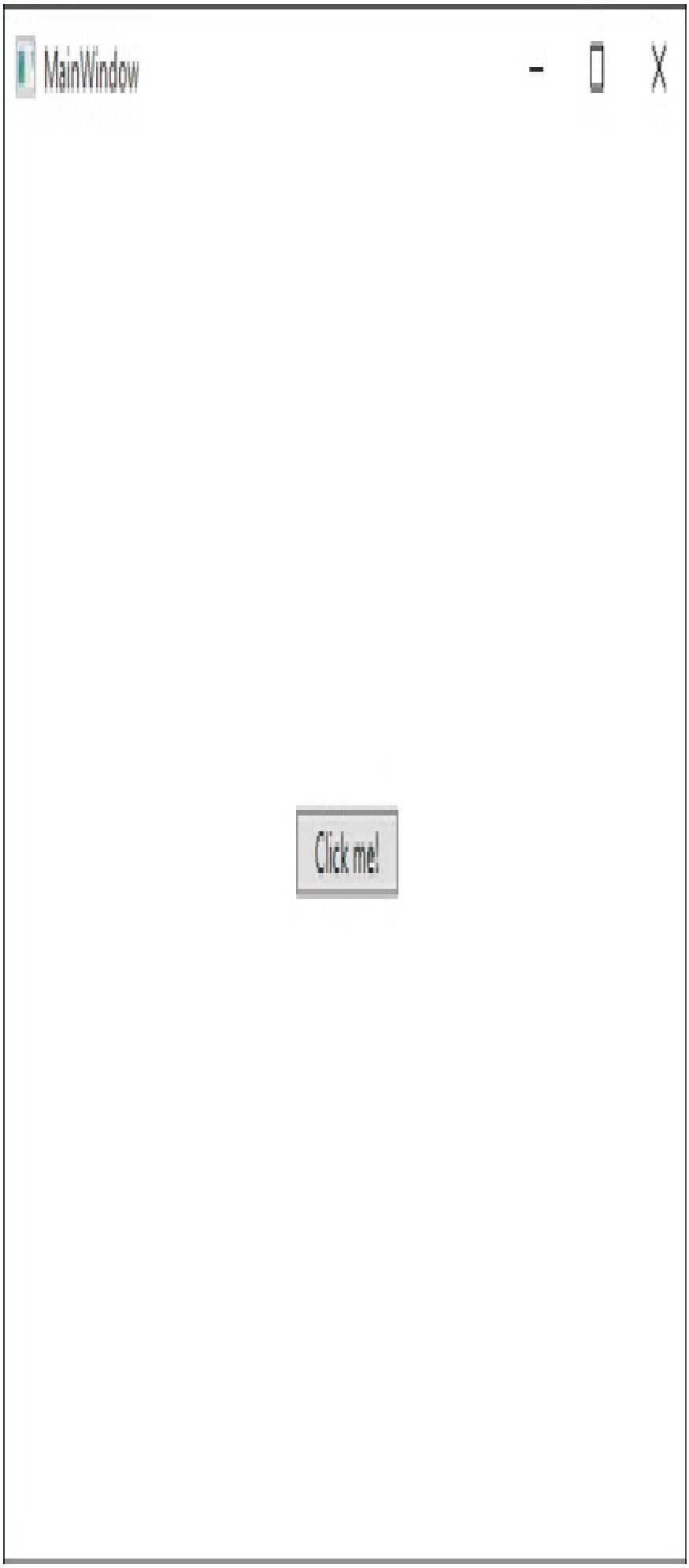

MainWindow
Click me!

Figura 17.5 – Applicazione WPF in esecuzione.

Si noti poi come il contenuto del pulsante, in questo caso una semplice stringa, sia impostato direttamente fra gli elementi di apertura e chiusura Button. In tal modo, WPF permette di creare pulsanti o altri oggetti inserendo sulla sua superficie altri elementi a loro volta definiti tramite XAML.

Per esempio, il seguente Button contiene al suo interno un'immagine e del testo:

```
<Button>
<StackPanel Orientation="Horizontal">
<Image Source="nomefile.png" Stretch="Uniform"/>
<TextBlock Text="Click me" />
</StackPanel>
</Button>
```

Grazie a un altro oggetto contenitore, in questo caso uno StackPanel, è semplice disporre il contenuto, qui affiancato orizzontalmente.

Per gestire gli eventi, per esempio il clic sul pulsante, basta aggiungere un attributo con il nome dell'evento stesso e avente come valore il nome del metodo gestore.

In Visual Studio è possibile, come già visto per Windows Forms, utilizzare la finestra delle proprietà o fare doppio clic sul pulsante stesso.

```
<Button Name="bt" Width="100" Height="30" Click="bt_Click">
Click me!
</Button>
```

All'interno del file di code behind dovrà essere inserito il metodo bt_Click che rispetti naturalmente la firma prevista dal delegate indicato dall'evento:

```
private void bt_Click(object sender, RoutedEventArgs e)
{
MessageBox.Show("Hello WPF");
}
```

Come per Windows Forms, tutto quello che è realizzabile tramite XAML in WPF corrisponde a codice C#, quindi al contrario è possibile realizzare un'interfaccia grafica anche esclusivamente via codice, creando istanze di oggetti e aggiungendole al layout di una finestra.

Per esempio, il precedente pulsante può essere ricreato e configurato così come segue:

```
Button bt = new Button();
bt.Name = "bt";
bt.Height = 30;
bt.Width = 100;
bt.Content = "Click me!";
bt.Click+=bt_Click;
```

Poi, per visualizzare un controllo, bisogna aggiungerlo a uno di layout, come l'elemento Grid presente nella finestra precedente. Assegnando per esempio a questa Grid il nome "grid", possiamo quindi inserire il pulsante aggiungendolo alla collezione di elementi figli restituita dalla proprietà Children:

```
grid.Children.Add(bt);
```

Nel caso di una Grid a più righe o a più colonne, per indicare la cella in cui inserire un elemento bisogna impostare delle corrispondenti proprietà, con una sintassi particolare, come la seguente:

```
<Button Name="bt" Grid.Column="1" Grid.Row="2" Grid.ColumnSpan="2"/>
```

In questo caso, l'oggetto Button può impostare dei valori di proprietà ereditate da un altro oggetto padre, che nell'esempio è Grid. Queste particolari proprietà sono dette attached property ed è una pratica che sarà utilizzata molto spesso in XAML.

Naturalmente, le attached property possono essere utilizzate anche in codice C#. Il precedente Button creato in XAML può essere creato e configurato come segue:

```
Button bt = new Button();
Grid.SetRow(bt, 2);
Grid.SetColumn(bt, 1);
Grid.SetColumnSpan(bt, 2);
```

Un'altra caratteristica molto importante e potente di XAML è quella delle conversioni implicite di tipo. Ogni attributo di un elemento XAML viene infatti impostato tramite una stringa. D'altronde non può essere fatto altrimenti, trattandosi di testo in formato XML.

In maniera implicita, il framework converte il valore testuale nel preciso valore desiderato dalla proprietà.

Riprendiamo e modifichiamo l'esempio precedente, in cui si è definito il pulsante Button:

```
<Button Name="bt" Width="100" Height="30" Background="Red" Click="bt_Click">
Click me!
</Button>
```

In questo caso, l'elemento Button ha due elementi di tipo double, Width e Height, a cui sono assegnati i valori testuali "100" e "30", che sono convertiti implicitamente in valori double. La proprietà Background, invece, è di tipo Brush e il valore "Red" viene implicitamente convertito in un'istanza di tale tipo.

Layout

Le applicazioni Windows Forms dispongono i controlli utilizzando un posizionamento assoluto e una gestione delle dimensioni legata alla risoluzione dello schermo. WPF, invece, misura i controlli in maniera indipendente da questa, utilizzando delle unità *device*

independent e in maniera relativa; l'aspetto di una stessa applicazione viene così ottimizzato per la configurazione hardware dell'utente.

Come visto nel precedente paragrafo, i controlli dell'interfaccia grafica da visualizzare su una finestra devono essere aggiunti necessariamente a un controllo di layout: anzi, è da questo che si inizia la realizzazione di una GUI.

I controlli di layout o pannelli, a differenza dei controlli di contenuto (come, per esempio, un pulsante), permettono di mantenere al loro interno più elementi gestendone il posizionamento.

WPF fornisce dei controlli predefiniti per definire il layout dell'interfaccia. Di questi i più utilizzati sono i seguenti:

- Grid – posizionamento con righe e colonne;
- UniformGrid – come Grid, ma con righe e colonne dimensionate in maniera uniforme;
- Canvas – posizionamento con coordinate assolute;
- DockPanel – i controlli vengono allineati lungo i lati dello schermo;
- StackPanel – i controlli vengono impilati uno dopo l'altro, verticalmente oppure orizzontalmente;
- WrapPanel – i controlli sono posizionati uno dopo l'altro da sinistra a destra andando automaticamente a capo.

Il controllo Grid è quello più flessibile in quanto permette di configurare il numero di righe e colonne e di posizionare così gli elementi figli impostando la cella, o le celle, di destinazione.

Il controllo Canvas invece è quello che più assomiglia al posizionamento di Windows Forms, in quanto utilizza delle coordinate assolute.

Gli altri controlli di layout, infine, utilizzano degli algoritmi di posizionamento più o meno automatici, che permettono di gestire rapidamente il layout su ogni dimensione di schermo dell'utente. Per esempio, lo StackPanel posiziona i controlli uno dopo l'altro, orizzontalmente o verticalmente, a seconda della proprietà Orientation.

In generale, la realizzazione di una interfaccia grafica per una applicazione abbastanza complessa prevede l'utilizzo di differenti pannelli, innestandoli uno dentro l'altro.

Data binding

Il *data binding* consente di visualizzare dati ricavati da una qualunque sorgente all'interno degli elementi dell'interfaccia grafica. In tal modo, per esempio, il testo di una casella di testo TextBox può essere collegato alla proprietà Name di una classe Customer, impostando direttamente in XAML tale corrispondenza, senza necessità di codice aggiuntivo.

Ogni volta che i dati nel modello sottostante cambiano, automaticamente essi si rifletteranno sull'interfaccia utente.

Supponiamo di avere una classe rappresentante un cliente, con una unica proprietà Name:

```
public class Customer
{
public string Name { get; set; }
}
```

Il seguente metodo genera un'istanza di Customer e la imposta come origine dati della finestra attuale:

```
public void GenerateData()
{
Customer c=new Customer() {Name="Antonio"};
this.DataContext=c;
}
```

Ogni controllo WPF derivato da FrameworkElement (che è una delle classi base di WPF che definisce le funzionalità base per tutti gli elementi di interfaccia grafica) ha una proprietà DataContext.

La proprietà DataContext definisce l'oggetto radice da cui tutti gli elementi figli della finestra otterranno i dati con cui impostare il proprio valore mediante binding, a meno di non impostarlo esplicitamente mediante altre modalità.

Il seguente controllo TextBlock, che è un'etichetta di testo, definito in XAML ha l'attributo Text collegato alla proprietà Name ricavata dal DataContext che, essendo un oggetto Customer, corrisponde alla proprietà Name di quest'ultima classe:

```
<TextBlock Text="{Binding Path=Name}"></TextBlock>
```

La sintassi usata per definire il data binding usa una cosiddetta *estensione di markup*, la quale prevede l'utilizzo delle parentesi graffe, altrimenti il testo corrisponderebbe direttamente al valore inserito fra doppi apici.

Allo stesso modo, possono essere creati dei controlli che visualizzano collezioni di dati, oppure dei moduli di inserimento e modifica dei dati che visualizzano diverse proprietà di uno stesso oggetto. Per esempio, si può creare un modulo per la modifica dei campi di un oggetto Customer, inserendo diverse TextBox in uno stesso StackPanel, la cui proprietà DataContext verrà ereditata dai controlli figli:

```
<StackPanel DataContext="{StaticResource myCustomer}">
<TextBox Text="{Binding FirstName}"/>
<TextBox Text="{Binding LastName}"/>
<TextBox Text="{Binding Street}"/>
<TextBox Text="{Binding City}"/>
</StackPanel>
```

XAML permette di creare anche dei template di visualizzazione; nel seguente frammento viene creata una ListBox che visualizzerà i nomi di diversi clienti, utilizzando sempre una TextBlock:

```
<ListBox Name="lista" ItemsSource="{Binding}" Grid.Row="1">
<ListBox.ItemTemplate>
<DataTemplate>
<TextBlock Text="{Binding Path=Name}" Foreground="Black"></TextBlock>
</DataTemplate>
</ListBox.ItemTemplate>
</ListBox>
```

La sorgente dati è impostata mediante la proprietà ItemsSource che a sua volta, in quanto impostata con il valore {Binding}, utilizzerà il DataContext della finestra che la contiene. Il DataContext in questo caso sarà impostato con una List<Customer>:

```
public MainWindow()
{
InitializeComponent();
this.DataContext = GenerateData();
}

private List<Customer> GenerateData()
{
List<Customer> clienti = new List<Customer>();
clienti.Add(new Customer() { Name = "Antonio" });
clienti.Add(new Customer() { Name = "John" });
clienti.Add(new Customer() { Name = "James" });
clienti.Add(new Customer() { Name = "Bill" });
clienti.Add(new Customer() { Name = "Matthew" });

return clienti;
}
```

Value Converter

All'interno delle applicazioni che utilizzano XAML per la definizione dell'interfaccia grafica, una delle pratiche più comuni è il binding di proprietà di un tipo differente da quello previsto dalla proprietà del controllo.

Per rendere meglio l'idea, la visibilità di un controllo è determinata dai membri dell'enumerazione Visibility e quindi può assumere i valori Hidden, Visible e Collapsed.

Supponiamo di dover configurare il data binding che determina la visibilità di un controllo in base al valore booleano di una proprietà Visible. In questo caso, sarà necessario associare il valore true al valore Visible e il valore false a Collapsed. La soluzione, in questa come in evenienze simili, è l'utilizzo di un Value Converter.

WPF contiene già diverse classi di tale genere, ma in caso di necessità sarà possibile creare la propria classe di conversione implementando l'interfaccia IValueConverter.

La seguente classe BooleanToVisibilityConverter esegue la conversione dal tipo bool al tipo Visibility e viceversa:

```
public class BooleanToVisibilityConverter : IValueConverter
{
public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
{
if (value is Boolean)
{
return ((bool)value) ? Visibility.Visible : Visibility.Collapsed;
}

return value;
}

public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)
{
throw new NotImplementedException();
}
}
```

Si noti che essa fa già parte, come altre, del .NET Framework e in particolare della libreria di classi WPF nel namespace System.Windows.Controls. Per utilizzarla all'interno di XAML, basta creare una risorsa che indichi il convertitore e impostare la proprietà Converter dell'estensione Binding:

```
<Grid>
< Grid.Resources>
<BooleanToVisibilityConverter x:Key="boolToVis" />
</ Grid.Resources>

<CheckBox x:Name="chkShowDetails" Content="Show Details" />
<StackPanel x:Name="panel" Visibility="{Binding IsChecked, ElementName=chkShowDetails,
Converter={StaticResource boolToVis}}">
</StackPanel>
</ Grid >
```

In questo caso, la visibilità dello StackPanel è collegata direttamente al valore della proprietà IsChecked di un controllo CheckBox.

Applicazioni WPF nel browser

WPF può essere utilizzato anche per la creazione di applicazioni eseguibili all'interno di un browser web, attualmente solo Internet Explorer o Firefox, dette anche applicazioni XBAP (*XAML Browser Applications*).

Com'è naturale, ci sono delle restrizioni che vengono applicate alle applicazioni eseguite in questa modalità, in quanto l'ambiente di esecuzione è una sorta di sandbox in cui per sicurezza non è possibile accedere completamente alle risorse del computer locale: non si possono aprire connessioni di rete, salvare file su disco in locazioni arbitrarie, e in genere non sono

permesse operazioni che potrebbero consentire ad applicazioni malevole di compromettere il PC dell'utente.

Per creare un'applicazione XBAP basta utilizzare l'apposito template di Visual Studio, quindi procedere alla sua implementazione, come visto già con WPF per le applicazioni Windows. La differenza principale è che, anziché avere a che fare con finestre, istanze della classe Window, l'applicazione sarà costituita da diverse pagine, istanze della classe Page, fra le quali l'utente potrà navigare.

Ecco un esempio di pagina in un'applicazione XBAP che contiene un pulsante al cui clic si vuole navigare alla pagina *Page2.xaml*:

```
<Page x:Class="WpfBrowserApplication1.Page1"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
mc:Ignorable="d"
d:DesignHeight="300" d:DesignWidth="300"
Title="Page1">
<Grid>
<Grid.RowDefinitions>
<RowDefinition Height="1*"/>
<RowDefinition Height="10*"/>
</Grid.RowDefinitions>
<Label Content="XBAP Application" HorizontalAlignment="Left" Margin="0,0,0,0" VerticalAlignment="Top"/>
<Button Content="click me" HorizontalAlignment="Left" Margin="109,104,0,0" Grid.Row="1" VerticalAlignment="Top"
Width="75" Click="Button_Click"/>

</Grid>
</Page>
```

Il codice C# per eseguire la navigazione sfrutta la classe NavigationService:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
this.NavigationService.Navigate(new Uri("/Page2.xaml", UriKind.RelativeOrAbsolute));
}
```

Distribuzione di applicazioni XBAP

Quando si compila un'applicazione XBAP, oltre all'eseguibile .exe, nell'output sono inclusi due file di manifesto: il primo, con estensione .manifest, che contiene i metadati associati all'applicazione, e un secondo, con estensione .xbap, che contiene le informazioni per installarla e distribuirla.

Le applicazioni XBAP, essendo eseguite all'interno di un browser web, possono essere ospitate in una pagina HTML inserendo un apposito link ipertestuale al file di manifesto .xbap dell'applicazione stessa:

```
<html>
<head></head>
<body>
<a href="application1.xbap">Clicca qui per lanciare l'applicazione</a>
</body>
</html>
```

All’interno del server web andranno copiati tutti e tre i file prodotti dalla compilazione.

Quando l’utente farà clic sul collegamento o punterà il browser direttamente al percorso del file .xbap, verrà effettuato il download dell’applicazione e subito dopo essa sarà avviata.

Universal Windows App

Una applicazione Windows universale (UWA) è un'app destinata a girare su un sistema operativo Windows 10, basata sulla piattaforma UWP (Universal Windows Platform), introdotta per la prima volta come Windows Runtime in Windows 8. Il termine "universale" sottintende il fatto che le applicazioni sfrutteranno un'unica interfaccia di programmazione, o API, e saranno così eseguibili su un qualunque sistema di Windows mediante un'esperienza utente unificata e portatile tra diverse tipologie di dispositivi, per esempio desktop, tablet, smartphone, Xbox.

Ciò significa, fra l'altro, che un unico pacchetto potrà essere installato su questa ampia varietà di dispositivi, e che lo stesso singolo pacchetto sarà distribuibile sul Windows Store per raggiungere tutti i tipi di device sui quali è eseguibile.

Strumenti di sviluppo

Per sviluppare delle Universal Windows Apps con Visual Studio 2015, anche in versione Community, è necessario utilizzare Windows 10 come sistema operativo e aver selezionato durante il setup dell'IDE anche le relative funzionalità di sviluppo (vedere la Figura 17.6).

In caso contrario è sempre possibile aggiungerle in un secondo momento, aggiornando l'installazione di Visual Studio, oppure aggiungendo gli strumenti necessari in fase di creazione di un nuovo progetto.

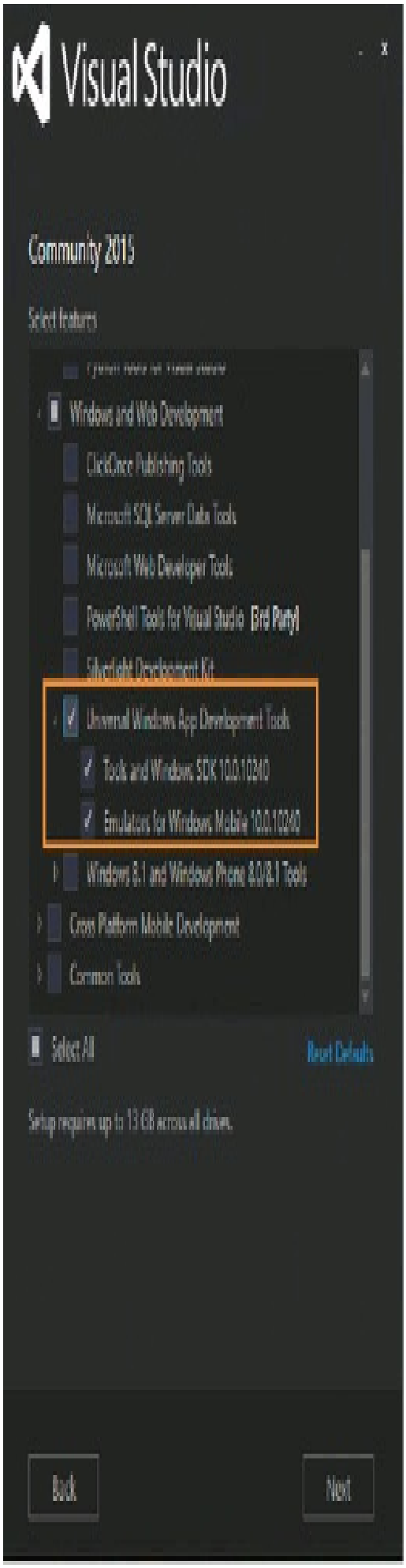

Visual Studio
Community 2015
Select features
Windows and Web Development
ClickOnce Publishing Tools
Microsoft SQL Server Data Tools
Microsoft Web Developer Tools
PowerShell Tools for Visual Studio (3rd Party)
Silverlight Development Kit
Universal Windows App Development Tools
Tools and Windows SDK 10.0.10240
Emulators for Windows Mobile 10.0.10240
Windows 8.1 and Windows Phone 8.0/8.1 Tools
Cross Platform Mobile Development
Common Tools
Select All
Reset Defaults
Setup requires up to 13 GB across all drives.
Back
Next

Figura 17.6 – Installazione degli strumenti di sviluppo per le Universal Windows Apps.

Con un unico progetto di Visual Studio, è possibile sviluppare un'app eseguibile su qualsiasi dispositivo su cui gira Windows 10. Gli strumenti di sviluppo includono inoltre gli emulatori per effettuare il test e il debug delle applicazioni anche senza possedere un dispositivo reale.

Creare una Universal Windows App

Utilizzando C# come linguaggio di sviluppo di una Universal Windows App, è possibile definire le interfacce grafiche con il già citato XAML. Per creare una nuova app, dal menu File di Visual Studio scegliete la voce File -> New Project. Nella finestra di creazione del nuovo progetto, selezionate nel riquadro a sinistra il gruppo Visual C# -> Windows -> Universal: così facendo, appariranno nell'elenco al centro i template di progetto per le UWA.

Scegliete il modello Blank App per creare una nuova applicazione vuota da cui iniziare lo sviluppo, date un nome, per esempio HelloWorld, e fate clic su OK per creare il progetto.

NOTA	La prima volta che si crea un progetto vi apparirà un messaggio di avviso che invita ad abilitare la modalità sviluppatore Windows 10 sul sistema in uso. Cliccate sul link presente nella stessa finestra per aprire le impostazioni Sviluppatori e abilitate la modalità suddetta selezionando l'apposita opzione.

Per provare che tutto funzioni per il verso giusto, provate a compilare la soluzione e quindi ad avviarla in debug.

Dalla casella combinata vicino al pulsante di avvio del debug, è possibile selezionare uno fra diversi possibili target, per esempio Local Machine per eseguire l'app direttamente sulla macchina attuale, oppure un simulatore, un emulatore di Windows 10 mobile, o anche un dispositivo fisico collegato con cavo USB.

Selezionate la voce Local Machine e avviate il debug: entro pochi secondi l'app verrà eseguita mostrando una finestra bianca, con il nome nella barra del titolo e un Frame Counter al suo interno.

Fermate il debug, o chiudete la finestra, e analizziamo i file creati dal template.

Il file App.xaml.cs è il punto di ingresso dell'app e contiene i metodi che ne gestiscono l'attivazione e la sospensione.

Il metodo OnLaunched, per esempio, crea e imposta il Frame che agirà da contesto principale di navigazione e, se si tratta del primo lancio, mostra la vista iniziale navigando verso la pagina MainPage:

```
if (rootFrame.Content == null)
{
rootFrame.Navigate(typeof(MainPage), e.Arguments);
}
```

Viene infine attivata la finestra corrente con il metodo seguente:

```
Window.Current.Activate();
```

Si noti che è possibile disattivare il Frame Counter visto all'interno della finestra impostando a false il valore della proprietà EnableFrameRateCounter:

```
#if DEBUG
if (System.Diagnostics.Debugger.IsAttached)
{
this.DebugSettings.EnableFrameRateCounter = false;
}
#endif
```

In ogni caso, il blocco di codice è eseguito solo in modalità debug.

La principale e unica vista dell'applicazione è al momento definita all'interno dei file MainPage.xaml e del suo file di code behind MainPage.xaml.cs. Essi definiscono la classe MainPage derivata da Page.

Nel file .xaml sarà possibile disegnare l'interfaccia grafica, mentre il codice gestore della logica e degli eventi sarà implementato nel file .cs.

NOTA	Un'introduzione a XAML è stata già fatta nel paragrafo relativo a WPF, all'inizio di questo capitolo. Naturalmente, ogni piattaforma ha i propri controlli e oggetti utilizzabili in XAML, ma i concetti e la sintassi sono identici.

Il contenuto generato dal modello Blank App è minimo, con un controllo Grid vuoto.

All'interno della cartella Assets sono contenute delle risorse grafiche personalizzabili, come i loghi in vari formati, e l'immagine per la schermata di avvio, o *splash screen*.

Facendo doppio clic sul file Package.appxmanifest, è possibile visualizzare e modificare varie impostazioni dell'applicazione, come i loghi da utilizzare nei vari formati, oltre a impostare le caratteristiche hardware e concedere i diritti di utilizzo. Per esempio, se l'app deve accedere alla fotocamera o al microfono di sistema, è possibile configurarli dalla sezione Capabilities.

Per visualizzare del contenuto all'interno dell'unica pagina attuale dell'app, aprite il file MainPage.xaml e modificate il contenuto della Grid, inserendo per esempio uno StackPanel con due controlli TextBlock e Image:

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
```

```
<StackPanel HorizontalAlignment="Center">
<TextBlock FontSize="24">Hello World</TextBlock>
<Image Source="Assets/Square150x150Logo.png" Width="160" Height="160"></Image>
</StackPanel>
</Grid>
```

Eseguendo nuovamente l'app in debug, sarà visualizzato un testo "Hello World" e in basso l'immagine con il logo di default.

Rilevamento del dispositivo

Una UWA è eseguibile su ogni dispositivo che supporti Windows 10, quindi da codice è spesso utile rilevarne la tipologia e le caratteristiche per poter eventualmente personalizzare le funzionalità e l'aspetto dell'app.

Potrebbe essere utile attivare o disattivare alcune funzioni a seconda che si stia eseguendo l'app su uno smartphone o, viceversa, su un normale desktop di PC.

Aggiungiamo una proprietà alla classe MainPage, di tipo string, chiamandola DeviceFamily:

```
public string DeviceFamily { get; } = AnalyticsInfo.VersionInfo.DeviceFamily;
```

Essa viene inizializzata con il valore della proprietà AnalyticsInfo.VersionInfo.DeviceFamily. La classe AnalyticsInfo è contenuta nel namespace Windows.System.Profile per cui è necessario il relativo using.

Per visualizzare il valore di DeviceFamily sulla MainPage, aggiungiamo un nuovo controllo TextBlock, e usiamo il binding per collegarlo alla proprietà. Con la Universal Windows Platform di Windows 10 è stata introdotta una nuova estensione di markup utilizzabile in XAML, x:Bind, per cui oltre al binding classico, come visto in WPF, è possibile ottenere lo stesso risultato con la seguente sintassi

```
<TextBlock Text="{x:Bind DeviceFamily}"/>
```

La differenza è che il binding di x:Bind è creato a tempo di compilazione, quindi offre migliori prestazioni a runtime.

Eseguendo l'app, verrà visualizzato anche il nome della famiglia di dispositivo corrente. Per esempio, la Figura 17.7 mostra l'app su un desktop di PC.

HelloWorld
Hello World
WindowsDesktop

Figura 17.7 – Rilevamento della famiglia di dispositivo in una Universal Windows App.

Navigazione su altre pagine

Molto spesso vi capiterà di dover sviluppare una applicazione costituita da più schermate o viste. Per vedere come funziona la navigazione da una schermata all'altra, aggiungiamo una nuova pagina alla nostra app di esempio, facendo clic con il tasto destro sul progetto e poi selezionando la voce Add -> New Item. Selezionate il template Blank Page per creare una pagina vuota e assegnate il nome Page2.xaml. A questo punto, aggiungete del contenuto alla pagina, per esempio un TextBlock come il seguente:

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
<TextBlock>Questa è la pagina 2</TextBlock>
</Grid>
```

Tornando alla pagina principale MainPage.xaml, aggiungiamo un controllo Button che al clic avvierà la navigazione verso la pagina Page2.

Per creare, per esempio, il pulsante con il testo Vai a pagina 2 posizionato al centro, scrivete il seguente codice XAML:

```
<Button HorizontalAlignment="Center" Click="Button_Click">Vai a pagina 2</Button>
```

Ora bisogna gestire l'evento Click, il cui metodo gestore sarà definito nel code behind MainPage.xaml.cs:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
Frame.Navigate(typeof(Page2));
}
```

Il metodo Frame.Navigate fa in modo che venga caricato il contenuto della pagina specificata. Se occorre passare dei parametri, è possibile farlo usando un secondo parametro:

```
Frame.Navigate(typeof(Page2), "Hello from MainPage");
```

Per gestire il parametro in arrivo, che non necessariamente deve essere una stringa, ma può essere un qualsiasi object, nella pagina di destinazione è possibile implementare l'override del metodo OnNavigatedTo e visualizzare una finestra di dialogo con i dati ricevuti:

```
protected async override void OnNavigatedTo(NavigationEventArgs e)
{
await new MessageDialog(e.Parameter.ToString()).ShowAsync();
base.OnNavigatedTo(e);
}
```

Non rimane che gestire il tasto Back per tornare alla pagina precedente. Infatti, se eseguite l'app nell'emulatore mobile e passate alla pagina Page2, anche premendo il tasto indietro non

si tornerà alla MainPage, ma si chiuderà l'app tornando al sistema operativo o all'app precedentemente aperta.

La gestione della navigazione all'indietro rappresenta un tipico caso in cui la gestione è differente fra i diversi sistemi. Uno smartphone Windows è dotato di un apposito pulsante hardware, mentre l'app eseguita in una finestra in un desktop di PC dovrà prevedere un'apposita funzione.

Windows 10 fornisce a tal proposito un pulsante Back quando un'app è eseguita in modalità Tablet, ma non quando è in normale modalità Desktop. Per abilitare questo pulsante, è possibile sfruttare la classe SystemNavigationManager. Alla fine del metodo OnLaunched del file App.xaml.cs, gestite l'evento Navigated del frame principale nel seguente modo:

```
Window.Current.Activate();
rootFrame.Navigated += RootFrame_Navigated;
}

private void RootFrame_Navigated(object sender, NavigationEventArgs e)
{
Frame rootFrame = Window.Current.Content as Frame;
SystemNavigationManager.GetForCurrentView().AppViewBackButtonVisibility = rootFrame.CanGoBack ?
AppViewBackButtonVisibility.Visible :
AppViewBackButtonVisibility.Collapsed;
}
```

Il codice fa in modo da mostrare il pulsante Back sulla barra del titolo della finestra, o nella task bar se in modalità Tablet, solo quando effettivamente c'è una view verso cui navigare all'indietro. Tuttavia, il pulsante è ancora soltanto visualizzato e un clic su di esso non ha alcun effetto.

Sempre nel metodo OnLaunched bisogna gestire l'evento BackRequested:

```
SystemNavigationManager.GetForCurrentView().BackRequested += App_BackRequested;

private void App_BackRequested(object sender, BackRequestedEventArgs e)
{
if (!e.Handled)
{
Frame frame = Window.Current.Content as Frame;
if(frame.CanGoBack)
{
frame.GoBack();
e.Handled = true;
}
}
}
```

In tal modo, sarà gestito sia il pulsante hardware su un Windows 10 mobile, sia il pulsante software visualizzato in modalità Desktop o Tablet, senza necessità di dover scrivere codice differente o dedicato.

Mobile Emulator

L'applicazione creata è già pronta a essere eseguita su un dispositivo mobile. Potete verificarlo selezionando come target del debug uno degli emulatori mobile installati, che in genere hanno un nome del tipo *Mobile Emulator 10.0*, seguito dalla versione e da altre caratteristiche, come la quantità di RAM e la risoluzione.

La Figura 17.8 mostra la stessa applicazione sviluppata finora, senza alcun cambiamento, che eseguita sull'emulatore mostrerà come DeviceFamily la stringa Windows.Mobile.


4:57
Hello World
Windows.Mobile

Figura 17.8 – Esecuzione di una Universal Windows App sull'emulatore mobile di Windows 10.

Pubblicazione sullo store

Per distribuire ad altri utenti e magari vendere la tua Universal Windows App, è necessario creare un pacchetto dell'app. Per Windows 10, ne viene creato uno solo che può essere installato in qualsiasi dispositivo supporti il sistema operativo.

Il primo passo per la creazione del pacchetto è la configurazione per mezzo del file Package.appxmanifest. Basta fare doppio clic per aprire l'editor grafico. A questo punto, è possibile configurare le varie informazioni all'interno della scheda Application, come nome e descrizione, e le immagini necessarie all'interno della scheda Visual Assets. Nella scheda Packaging, invece, si possono immettere i dati di pubblicazione. Innanzitutto, tutte le Universal Windows Apps devono essere firmate con un certificato che può essere selezionato tramite l'apposito pulsante.

Dopo aver apportato tutte le modifiche necessarie, basta salvare il file. Prima di procedere alla creazione dei pacchetti, bisogna accedere al proprio account sul Windows Dev Center (e quindi, se non lo avete già fatto, dovete registrarvi).

L'indirizzo per accedere alla pagina di pubblicazione dell'applicazione è https://dev.windows.com/it-it/dashboard/Application/New. Qui dovete scegliere e riservare il nome della vostra app e magari immettere già altre informazioni, come l'eventuale prezzo di vendita, immagini promozionali, una descrizione e così via.

NOTA	Il nome dell'applicazione deve essere univoco e non utilizzato da altri. Potete riservare il nome dell'app direttamente da Visual Studio, eseguendo l'accesso al vostro account sviluppatori.

A questo punto, da Visual Studio andate al menu Project -> Store e selezionate la voce Create App Packages. Verrà avviata la procedura di creazione dei pacchetti; al primo passo potrete scegliere se crearli per inviarli allo store, o se trasferirli e installarli direttamente su altri dispositivi (tale procedura è anche detta *sideload*). In questo secondo caso, non sarà richiesto l'accesso all'account sviluppatore.

Fate clic su Avanti, eseguite eventualmente l'accesso, e a questo punto potete configurare i dettagli per la creazione del pacchetto.

Create App Packages

Select and Configure Packages

Output location:

C:\Users\Antonio\OneDrive\Scrittura libri\CSharp6\programmare_con_csharp6\Programmare con CSharp 6\Capitolo 16\HelloWorld\Ap ...

Version:

1 . 1 . 0 . 0

☑ Automatically increment

Generate app bundle:

Always

What does an app bundle mean?

Select the packages to create and the solution configuration mappings:

	Architecture	Solution Configuration
☐	Neutral	None
☑	x86	Release (x86)
☑	x64	Release (x64)
☑	ARM	Release (ARM)

The Windows Store will only accept the generated .appxupload package. Any other .appx packages are created for testing purposes only.

☑ Include full PDB symbol files, if any, to enable crash analytics for the app. Learn More

Previous | Create | Cancel

Figura 17.9 – Configurazione per la creazione dei pacchetti di un'app universale

Per convalidare l'app e poterla pubblicare, ricordate che è necessario usare la configurazione di release. Infine, cliccate sul pulsante Create e, al termine della generazione, verrà visualizzata una finestra di conferma in cui sarà possibile vedere il percorso di creazione del pacchetto, in maniera da recuperarlo per distribuirlo. È anche possibile avviare una procedura di convalida dello stesso prima di inviarlo allo store.

I pacchetti andranno poi inviati tramite la dashboard del Windows Dev Center, nella sezione Invio -> Pacchetti, in cui è possibile scegliere di caricare file in uno dei formati supportati. Per le Universal App di Windows 10, i file sono in formato appxbundle o appxupload.

Per la distribuzione diretta, detta anche *sideload*, basta copiare la cartella contenente i pacchetti generati, all'interno della quale è presente anche un file con uno script di installazione PowerShell, in genere denominato Add-AppDevPackage.ps1. Basta fare clic su di esso con il tasto destro e scegliere il comando Run with PowerShell.

Applicazioni web con ASP.NET

Fin dalla versione 1.0 del .NET Framework e di C#, Microsoft ha puntato molto sullo sviluppo web, con l'introduzione della piattaforma ASP.NET.

La prima release ha costituito un cambiamento netto rispetto alla precedente tecnologia *ASP* (*Active Server Pages*), ora conosciuta anche come *Classic ASP*.

Il termine ASP.NET copre un insieme piuttosto ampio di tecnologie web, che vanno dalle applicazioni web ospitate all'interno di un web server come *IIS* e fruite direttamente in un normale browser web, fino alle varie tipologie di web service che invece sono invocabili da parte di applicazioni client mediante richieste HTTP.

Per quanto riguarda la parte web direttamente visibile all'utente, il framework ASP.NET include una serie di funzionalità che permettono di eseguire all'interno del web server tutto il codice necessario a compilare, e quindi produrre, il codice HTML che verrà renderizzato dai browser.

Nei seguenti paragrafi si farà una breve introduzione delle possibilità offerte da ASP.NET per la costruzione di interfacce grafiche web-based, mentre si consiglia di approfondire anche aspetti architetturali della piattaforma, se si intende utilizzarla per lo sviluppo professionale di applicazioni web e di servizi.

Versioni di ASP.NET

ASP.NET 1.0 è stata la prima versione della nuova tecnologia web di Microsoft basata su .NET, ma è a partire dalla versione 2.0 che le cose hanno cominciato a farsi seriamente divertenti con una serie di importanti novità, come il rilascio dell'*ASP.NET Development Web Server* per evitare l'uso di IIS nei computer di sviluppo, l'introduzione di molti nuovi web control, delle master page e dei temi.

Le master page, in particolare, consentono la gestione del layout delle diverse pagine di un sito, permettendo di definire delle zone comuni, come header e footer, e delle parti invece variabili, in cui inserire il contenuto personalizzato per ogni determinata pagina.

Con ASP.NET 3.5 viene aggiunta anche nel mondo web la possibilità di sfruttare le potenzialità di LINQ e il data binding tra controlli ed entità ADO.NET, oltre al supporto integrato per lo sviluppo basato su AJAX.

ASP.NET 4.0 e 4.5 aggiungono ancora altre funzionalità, fra cui la possibilità di ottimizzare il layout delle pagine web in maniera da essere eseguite su diversi dispositivi mobili e su nuovi

browser, miglior supporto di CSS e di HTML 5, supporto del pattern di sviluppo MVC (*Model View Controller*) e, naturalmente, integrazione della programmazione asincrona.

La versione ASP.NET 5 costituisce un'importante novità, introducendo concetti e cambiamenti architetturali con cui è possibile sviluppare moderne web app in maniera più rapida ed efficiente. In particolare, ASP.NET 5 segue quelle che sono le novità del .NET Framework, che si presenta in due edizioni, .NET Core 5 e .NET Framework 4.6, e sulle quali ASP.NET 5 può funzionare. Quindi, essendo il primo focalizzato per lo sviluppo cross-platform, anche ASP.NET 5 è supportato su Linux e Mac OS, oltre che su Windows.

Web form

L'innovazione principale apportata da ASP.NET, con le sue web form, è la possibilità di separazione fra codice e presentazione, in maniera simile a quanto fatto dall'introduzione di XAML per le applicazioni desktop in WPF.

Infatti, mentre le pagine ASP classiche all'interno dello stesso file mischiano parti di HTML e parti di logica, rendendo difficile la manutenzione e la modifica, con ASP.NET ogni pagina, che viene qui chiamata *web form*, viene suddivisa in due parti.

La parte di layout o presentazione, basata sempre su HTML, viene gestita e salvata in un file con estensione .aspx. La parte complementare di logica, è scritta in un linguaggio .NET, per esempio C#, e salvata in un file detto di c*ode behind*, con estensione .aspx.cs.

In tal modo, anche per lo sviluppo web è possibile sfruttare le caratteristiche Object Oriented del linguaggio C#, in congiunzione alle librerie di classi fornite da .NET Framework (in particolare i web control) o da terze parti e alle funzionalità visuali di design messe a disposizione ancora una volta da Visual Studio.

Web control

L'architettura delle web form di ASP.NET introduce nel mondo web un modello simile a quello già utilizzato in ambito desktop, con la possibilità di utilizzare una serie di controlli e componenti server-side, detti web control, che rappresentano oggetti dell'interfaccia utente da rappresentare all'interno di una pagina web. ASP.NET converte tali web control direttamente in HTML, in quanto ogni browser web non può far altro che renderizzare tale linguaggio di markup.

Grazie ai web control, tuttavia, si può continuare a mantenere uno sviluppo Object Oriented, basato su classi e sulla programmazione a eventi. Per esempio, per visualizzare un pulsante all'interno di una pagina web e gestirne il clic, basta utilizzare il relativo web control, da inserire per mezzo di un tag HTML <asp:Button> (o, come vedremo, trascinando il relativo

oggetto dalla toolbox) e quindi implementare all'interno del file di code behind, in C#, il codice da eseguire.

Il seguente codice potrebbe essere la parte HTML contenuta nel file *Default.aspx*:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeBehind="Default.aspx.cs" Inherits="WebApplication1.Default" %>

<!DOCTYPE html>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
<title></title>
</head>
<body>
<form id="form1" runat="server">
<asp:Button ID="Button1" runat="server" OnClick="Button1_Click" Text="Button" />
<br />
<asp:Label ID="Label1" runat="server" Text="Label"></asp:Label>

</form>
</body>
</html>
```

Chi già conosce un minimo di HTML, noterà che i tag con prefisso asp, come Button e Label, non sono degli elementi HTML standard. Sono invece dei web control standard forniti da ASP.NET.

L'attributo runat="server" indica che gli elementi corrispondenti saranno disponibili anche lato server come oggetti .NET e quindi potranno essere utilizzato mediante il nome indicato dal relativo attributo ID.

Il concetto è simile a quanto già visto con XAML, anche se in realtà sono state le web form di ASP.NET a vedere per prime la luce.

Il file di code behind, con estensione .aspx.cs, contiene la definizione di una classe, il cui nome è quello indicato nell'attributo Inherits della direttiva Page del file ASPX precedente:

```
public partial class Default : System.Web.UI.Page
{
protected void Page_Load(object sender, EventArgs e)
{
}
}
```

Anche ogni classe che definisce una pagina web ha il proprio ciclo di vita e i propri eventi. Essa deriva infatti dalla classe Page nel namespace System.Web.UI. Per esempio, il metodo Page_Load è quello eseguito all'apertura della pagina stessa, e quindi può contenere del codice da eseguire prima di ogni altro metodo.

La classe Page espone diverse proprietà, fra cui le più importanti e utilizzate sono Request, Response e Session.

La proprietà Request, di tipo HttpRequest, rappresenta e fornisce informazioni sulla richiesta HTTP. Per esempio, contiene i valori passati come variabili della form, oppure quelli passati come parametri nell'url, la cosiddetta QueryString.

Se la pagina *default.aspx* è stata aperta con un url simile a *default.aspx?id=123*, si può ricavare il valore di id esaminando la richiesta come segue:

```
protected void Page_Load(object sender, EventArgs e)
{
string id=Request.QueryString["id"];
}
```

La proprietà Response, della classe HttpResponse, consente invece di inviare informazioni al client all'interno della risposta, per esempio per scrivere dell'HTML sulla pagina o per reindirizzare l'utente su una pagina differente.

La proprietà Session contiene informazioni sulla sessione corrente, salvando al suo interno le variabili dette appunto di sessione, in quanto specifiche dell'utente attuale.

Tali proprietà si utilizzano estensivamente nella programmazione ASP.NET e dimostrano come con un approccio totalmente Object Oriented sia possibile gestire aspetti tipici del protocollo HTTP senza preoccuparsi di quello che accade a basso livello.

L'attributo OnClick del controllo Button, indica il nome del gestore dell'evento, implementato nel file di code behind *default.aspx.cs*:

```
protected void Button1_Click(object sender, EventArgs e)
{
Label1.Text = "Hello ASP.NET";
}
```

In questo caso, al clic del pulsante viene semplicemente assegnato un valore stringa "Hello ASP.NET" alla proprietà Text del controllo identificato dall'ID Label1.

Dal punto di vista del browser, questo si tradurrà per esempio in codice HTML, esattamente in qualcosa del tipo:

```
<span id="Label1">Hello ASP.NET</span>
```

Anche in questo caso, senza necessità di conoscere e dover gestire manualmente HTML e i suoi tag, ma soltanto con codice C#, analogamente a quanto si scriverebbe in normali applicazioni Windows.

Dietro le quinte, com'è naturale, il funzionamento di ASP.NET è differente rispetto a quello desktop, in quanto il clic vero e proprio avviene nel browser dell'utente, mentre il gestore dell'evento viene eseguito sul web server.

Ciò che accade al clic è un cosiddetto Postback, cioè un invio di informazioni verso il server, che le riceve e genera la nuova pagina da ricaricare nel browser.

ASP.NET contiene diverse decine di web control di svariate categorie che possono essere usati nelle applicazioni web; ognuno di essi possiede delle proprietà comuni e altre specifiche, utilizzabili sia da codice C# sia dalla finestra delle proprietà di Visual Studio.

Inoltre, è possibile gestire i vari eventi di ogni controllo in maniera totalmente analoga a quanto fatto con i controlli Windows Forms e WPF.

Altri produttori di terze parti forniscono ancora diverse librerie e controlli specializzati, utilizzabili anch'essi come i controlli server standard.

Blocchi di codice inline

ASP.NET consente anche di inserire dei blocchi di codice direttamente nell'HTML (in maniera simile a quanto era necessario fare in Classic ASP). In questo caso, il codice C# può essere utilizzato per scrivere delle espressioni il cui risultato verrà valutato lato server ma visualizzato all'interno dell'HTML.

Per inviare del testo sulla pagina web, generato in maniera dinamica a partire da istruzioni C#, l'espressione deve essere scritta fra <%= e %>, per esempio:

```
<p>
Data e ora: <%= DateTime.Now.ToString() %>
</p>
```

Un'altra possibilità, invece, è di eseguire blocchi interi di istruzioni C#, utilizzando <% e %> per racchiudere i blocchi contenenti istruzioni:

```
<p>
<%
for (int i = 0; i < 10; i++)
{
Response.Write(i);
%>
<br />
<%
}
%>
</p>
```

Si noti come il testo al di fuori dei blocchi di codice, per esempio il tag <br/> nel ciclo for, venga inviato alla pagina web direttamente in formato HTML.

Controlli personalizzati

ASP.NET ha introdotto anche la possibilità di creare dei controlli utente personalizzati, da riutilizzare all'interno delle diverse web form delle proprie applicazioni.

Un controllo utente, o *web user control*, ha estensione .ascx ed è realizzabile come una normale web form, quindi includendo sia una parte di presentazione sia una di code behind in C#.

Lo scopo principale di un web user control è quello di raggruppare diversi controlli standard, in maniera da poter riutilizzare il risultato in più web form.

A differenza di una web form, in cui si utilizza `Page`, un controllo utente inizia con la direttiva Control:

```
<%@ Control Language="C#" AutoEventWireup="true" CodeBehind="WebUserControl1.ascx.cs"
Inherits="WebApplication1.WebUserControl1" %>
<table style="width:100%;">
<tr>
<td>
<asp:Label ID="Label1" runat="server" Text="Nome"></asp:Label>
</td>
<td>
<asp:Label ID="Label2" runat="server" Text="Cognome"></asp:Label>
</td>
</tr>
<tr>
<td>
<asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
</td>
<td>
<asp:TextBox ID="TextBox2" runat="server"></asp:TextBox>
</td>
</tr>
</table>
```

La parte di code behind conterrà l'implementazione di una classe C# che deriva da UserControl:

```
public partial class WebUserControl1 : System.Web.UI.UserControl
{
protected void Page_Load(object sender, EventArgs e)
{
}
}
```

Per utilizzare un controllo all'interno di una pagina .aspx, è necessario aggiungere un riferimento al controllo per mezzo della direttiva Register:

```
<%@ Register src="WebUserControl1.ascx" tagname="WebUserControl1" tagprefix="uc1" %>
```

Tale direttiva indica il prefisso da utilizzare per creare l'istanza del controllo sulla pagina. In questo caso avverrà nel seguente modo:

```
<uc1:WebUserControl1 ID="WebUserControl1" runat="server" />
```

Una seconda possibilità di creare controlli riutilizzabili e distribuibili è quella dei cosiddetti *custom control*: è una della attività più complesse da eseguire in ASP.NET, ma consente di ottenere e sfruttare il massimo delle risorse messe a disposizione dello sviluppatore. Un custom control viene infatti realizzato implementando una classe interamente in C#, derivata da Control o da una classe più specializzata, e inserendo al suo interno tutto il codice necessario per renderizzarlo su una pagina web:

```
public class MyControl: Control
{
protected override void Render(HtmlTextWriter writer)
{
writer.Write(DateTime.Now.ToShortTimeString());
}
}
```

Anche in questo caso, per il suo utilizzo è necessario registrare il controllo con una direttiva Register, che però indica il namespace contenente la classe e il tag prefix:

```
<%@ Register assembly="WebApplication1" namespace="WebApplication1" tagprefix="cc1" %>
```

Così, utilizzando cc1 come prefisso, possiamo istanziare un controllo indicando il nome della sua classe:

```
<cc1:MyControl ID="MyControl1" runat="server">
</cc1:MyControl>
```

Web Page e Razor

ASP.NET fornisce una seconda possibile sintassi per integrare all'interno del markup HTML del codice C#, in maniera da creare pagine web dal contenuto dinamico. Tale possibilità è fornita da un view engine chiamato Razor, introdotto originariamente come estensione di Visual Studio 2010, e poi integrato direttamente nel framework ASP.NET MVC 3.

La sintassi di Razor può tuttavia essere utilizzata anche per lo sviluppo di applicazioni web abbastanza semplici, con poche pagine, che possono essere considerate un'altra variante di ASP.NET, che per contrapposizione a web form, viene detta *Web Page*.

Le ASP.NET Web Page utilizzano singoli file, contenenti del codice HTML inframezzato da blocchi di codice da eseguire lato server. L'estensione del singolo file è, nel caso di codice C#, .cshtml, e ogni pagina è un'istanza della classe WebPage (differente dalla classe Page delle web form). In tal modo, il codice HTML, che da solo produrrebbe pagine statiche, viene potenziato dalla possibilità di creare contenuto dinamico.

Il design della pagina può essere creato solo scrivendo interamente il codice HTML necessario: non abbiamo controlli lato server da inserire nella pagina, come visto nelle web form.

NOTA	Microsoft offre gratuitamente anche un altro strumento, chiamato WebMatrix, che si sposa particolarmente bene con lo sviluppo di applicazioni basate su Razor. WebMatrix integra un editor di pagine web, un web server per il test delle applicazioni web, e degli strumenti per database e per il deploy su Internet delle proprie applicazioni. WebMatrix può essere ottenuto all'url http://www.asp.net/web-pages.

In Razor, i blocchi di codice da interpretare come C# sono identificati dal carattere @:

```
<html>
<head>
<title></title>
</head>
<body>
Data e ora: @DateTime.Now
</body>
</html>
```

In tal modo, è possibile generare codice al volo in maniera molto più semplice rispetto alla sintassi di scripting vista con le web form, basata su <% e %>, e creare pagine dinamiche di cui si può controllare il layout più facilmente.

Inoltre, con il carattere @ possiamo identificare un blocco di più istruzioni da racchiudere poi fra parantesi graffe. In quest'ultimo, magari, si può eseguire l'assegnazione di una variabile, da utilizzare nella stessa pagina più avanti, per esempio per popolare il contenuto di un tag h1:

```
<div>definizione una variabile...
@{
var dt= DateTime.Now;
string str = dt.ToShortTimeString();
}
che utilizzo più avanti stampandola: <h1>@str</h1>
</div>
```

All'interno di blocchi di codice, ogni istruzione deve essere terminata dal punto e virgola. Come nel caso precedente, è possibile creare variabili, anche di tipo implicito, e poi utilizzarle all'interno della stessa pagina referenziandole con @.

Naturalmente, anche all'interno delle Web Page possono essere utilizzate le classi fornite dal .NET Framework e le relative proprietà.

Il seguente esempio crea una tabella HTML con alcuni valori ricavati dalle proprietà dell'oggetto Request:

```
<table border="1">
<tr>
<td>Requested URL</td>
<td>Relative Path</td>
<td>Full Path</td>
<td>HTTP Request Type</td>
</tr>
<tr>
<td>@Request.Url</td>
<td>@Request.FilePath</td>
<td>@Request.MapPath(Request.FilePath)</td>
<td>@Request.RequestType</td>
</tr>
</table>
```

È ovviamente possibile creare e utilizzare codice più strutturato, servendosi di istruzioni per il controllo di flusso, in maniera da rendere la creazione della pagina dipendente da particolari condizioni.

Fra le possibili variabili da controllare vi è la proprietà IsPost, che indica se la richiesta HTTP è una POST o, viceversa, una richiesta GET.

Il seguente esempio mostra come effettuare la somma di due numeri, inseriti in due campi di testo, al clic su un pulsante Add, che appunto invierà i valori da sommare al server con una richiesta POST:

```
@{
var total = 0;
var totalMessage = "";
if (IsPost)
{
var num1 = Request["text1"];
var num2 = Request["text2"];
total = num1.AsInt() + num2.AsInt();
totalMessage = "Totale = " + total;
}
}

<!DOCTYPE html>
<html lang="en">
<head>
<title>Somma</title>
<meta charset="utf-8" />
<style type="text/css">
body {
font-family: Verdana, Arial;
margin: 50px;
}

form {
padding: 10px;
border-style: solid;
width: 250px;
```

```
}
</style>
</head>
<body>
<p>inserisci i numeri da sommare a clicca <strong>Somma</strong>.</p>
<form action="" method="post">
<p>
<label for="text1">Numero 1:</label>
<input type="text" name="text1" />
</p>
<p>
<label for="text2">Numero 2:</label>
<input type="text" name="text2" />
</p>
<p><input type="submit" value="Somma" /></p>
</form>

<p>@totalMessage</p>

</body>
</html>
```

La pagina trasforma il testo inserito nelle caselle in interi, mediante il metodo di estensione della classe String, AsInt (ne esistono anche per convertire in altri tipi), anche se è perfettamente lecito e possibile utilizzare altri metodi, come la classe Convert.

I numeri ricavati vengono sommati e quindi il risultato mostrato, stampandolo direttamente sulla pagina per mezzo dell'istruzione @totalMessage.

Le Web Page permettono anche di validare l'input, per esempio per evitare di provare a sommare due valori che non rappresentano dei numeri. La classe WebPage espone una proprietà Validation con cui indicare i campi da validare. Per esempio, all'inizio della pagina si può scrivere:

```
@{
Validation.RequireField("text1", "Inserisci il primo numero.");
Validation.RequireField("text2", "Inserisci il secondo numero.");
Validation.Add("text1", Validator.Integer());
Validation.Add("text2", Validator.Integer());
```

Quindi, per ognuno dei due campi destinati a contenere i numeri, si invocherà il metodo ValidationMessage dell'oggetto HtmlHelper restituito dalla proprietà HTML della pagina, per verificare così la presenza di eventuali errori. Per ripristinare il valore inserito prima del POST, invece, si imposta il value usando il valore contenuto nella form:

```
<label for="text1">Numero 1:</label>
<input type="text" name="text1" value="@Request.Form["text1"]" />
@Html.ValidationMessage("text1")
```

Provando a lanciare la pagina in un browser, si può vedere il funzionamento di quanto scritto in C# e Razor.

MVC

ASP.NET MVC è la tecnologia per creare applicazioni web in ASP.NET in cui, al contrario delle web form dove a farla da padrone è il codice server side, HTML e JavaScript sono i linguaggi utilizzati per la generazione del layout delle pagine.

MVC implementa il pattern Model View Controller, che, seguendo un modello nato negli anni Settanta ma sempre in voga anche ai nostri giorni, permette di organizzare il codice in tre parti distinte:

- Model – contiene i metodi di accesso ai dati;
- View – visualizza i dati nell'interfaccia e gestisce l'interazione con l'utente;
- Controller – riceve i comandi dell'utente attraverso una View, eseguendo operazioni che coinvolgono il Model, e che in genere portano a un aggiornamento della View.

MVC permette di separare i ruoli in maniera da disaccoppiare il modello e la logica di controllo dall'interfaccia grafica. In tal modo, il codice HTML sarà gestibile in maniera separata dal resto dell'applicazione, rendendo manutenzione, modifica e test più efficaci e semplici.

Il codice che manipola i dati sarà contenuto solo nel Model, quello che crea l'interfaccia grafica soltanto nelle View e, infine, quello che si occupa di gestire le richieste dell'utente solo nei controller.

Diciamo subito che MVC non è un sostituto delle ASP.NET Web Form. Ognuno dei due approcci ha i propri vantaggi e ognuno è il più adatto in contesti differenti.

Le web form sono state create con l'obiettivo di aggiungere uno strato di astrazione utile alla gestione dello stato al di sopra di un protocollo come HTTP, che invece è fondamentalmente da considerarsi *stateless*, utilizzando i concetti di PostBack e ViewState. Ciò rende possibile lo sviluppo seguendo una modalità molto simile a quella Desktop con Windows Forms, con la possibilità di creare un'interfaccia grafica molto articolata grazie, per esempio, all'uso di controlli server.

MVC riabbraccia la vera natura di HTTP e, fornendo un'infrastruttura senza dubbio più leggera e semplice, è certamente al passo con le attuali tendenze seguite nel mondo dello sviluppo web.

Con l'attuale versione di ASP.NET è altresì possibile mischiare gli approcci di sviluppo, per esempio inserendo delle web form, all'interno di un'applicazione MVC.

ASP.NET MVC

Nell'interpretazione MVC di ASP.NET, i controller sono delle normali classi C#, in genere derivate dalla classe Controller presente nel namespace System.Web.Mvc. Ogni metodo pubblico in una classe Controller è chiamato *action method* e viene associato a un url configurabile per mezzo dei meccanismi di routing di ASP.NET.

Quando viene fatta una richiesta a un indirizzo associato a un action method, vengono eseguite le istruzioni di quest'ultimo che permetteranno di eseguire delle operazioni su un modello sottostante e quindi di selezionare una view da visualizzare nel browser.

Per la creazione della view, nelle prime versioni di ASP.NET MVC erano utilizzate della classiche web form, mentre a partire dalla versione MVC 3 viene usato il view engine di Razor.

Visual Studio fornisce il supporto e l'intellisense per entrambi i tipi di view engine.

NOTA	ASP.NET MVC è open source: il codice sorgente è stato pubblicato da Microsoft nell'aprile 2009, sotto licenza Microsoft Public License (MS-PL).

Esempio di applicazione ASP.NET MVC

Per creare un'applicazione web basata su ASP.NET MVC, si può utilizzare come sempre uno dei template di Visual Studio. In particolare, avviata la procedura di creazione di un nuovo progetto e scelta la voce ASP.NET Web Application, si aprirà una seconda finestra in cui sarà possibile selezionare il template denominato MVC. Verrà così creata una applicazione ASP.NET MVC completa e funzionante, senza dover fare nulla, una sorta di Hello World, che costituisce un buon punto di partenza sia per costruire la propria applicazione, sia per esaminare la struttura di un progetto MVC.

New ASP.NET Project - WebApplication1

Select a template:

ASP.NET 4.6 Templates

Empty | Web Forms | MVC | Web API | Single Page Application

Azure API App (Preview) | Azure Mobile Service

ASP.NET 5 Preview Templates

Empty | Web API | Web Application

A project template for creating ASP.NET MVC applications. ASP.NET MVC allows you to build applications using the Model-View-Controller architecture. ASP.NET MVC includes many features that enable fast, test-driven development for creating applications that use the latest standards.

Learn more

Change Authentication

Authentication: Individual User Accounts

Add folders and core references for:

☐ Web Forms ☑ MVC ☐ Web API

Microsoft Azure

☐ Host in the cloud

Web App

☐ Add unit tests

Test project name: WebApplication1.Tests

OK | Cancel

Figura 17.10 – Scelta del template ASP.NET MVC per una applicazione web in Visual Studio.

Infatti, dando un'occhiata al Solution Explorer noterete che l'IDE ha creato per noi diversi file, suddividendoli in una serie di cartelle che seguono una struttura standard e adatta al pattern MVC.

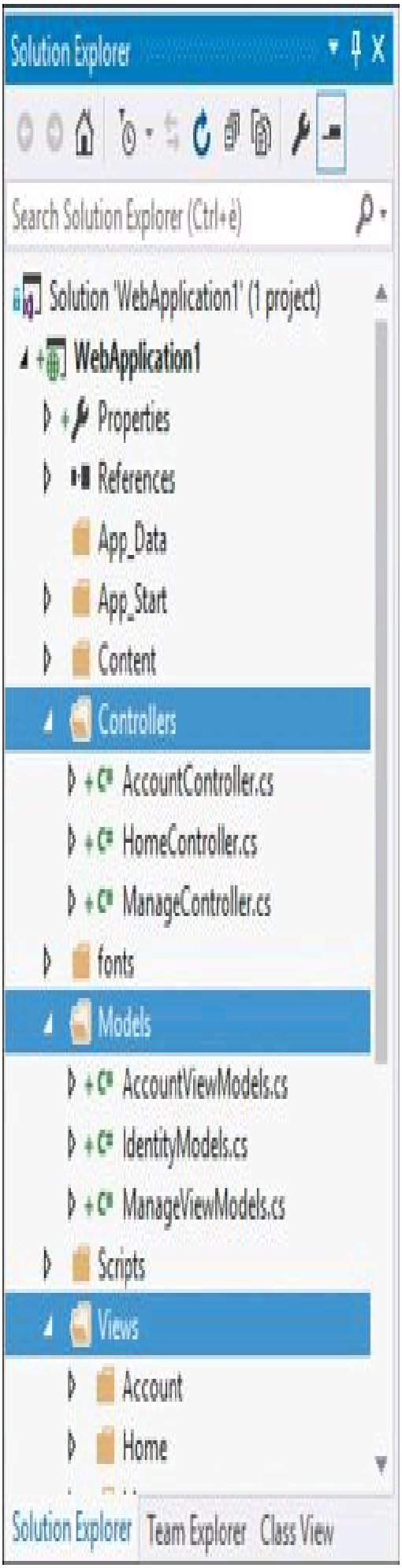

Solution Explorer
Search Solution Explorer (Ctrl+è)
Solution 'WebApplication1' (1 project)
WebApplication1
Properties
References
App_Data
App_Start
Content
Controllers
AccountController.cs
HomeController.cs
ManageController.cs
fonts
Models
AccountViewModels.cs
IdentityModels.cs
ManageViewModels.cs
Scripts
Views
Account
Home
Solution Explorer
Team Explorer
Class View

Figura 17.11 – Struttura di un'applicazione ASP.NET MVC in Visual Studio.

In particolare, la cartella Controllers è quella che contiene le classi Controller, che come vedrete seguono la convenzione che prevede di assegnarvi dei nomi che terminano appunto con la parola "Controller": potete per esempio notare le due classi generate da Visual Studio e denominate HomeController e AccountController.

Di seguito, l'implementazione C# della classe HomeController, che gestisce gli url Index, About e Contact:

```
public class HomeController : Controller
{
public ActionResult Index()
{
return View();
}

public ActionResult About()
{
ViewBag.Message = "Your application description page.";
return View();
}

public ActionResult Contact()
{
ViewBag.Message = "Your contact page.";

return View();
}
}
```

La cartella Model contiene le classi che rappresentano oggetti e dati usati nell'applicazione e che definiscono la logica per l'interazione con un archivio dati sottostante.

Nel progetto creato potete notare, per esempio, la classe LoginViewModel che contiene le proprietà necessarie a inviare i dati per effettuare il login di un utente e che verrà utilizzata dall'AccountController:

```
public class LoginViewModel
{
[Required]
[Display(Name = "User name")]
public string UserName { get; set; }

[Required]
[DataType(DataType.Password)]
[Display(Name = "Password")]
public string Password { get; set; }

[Display(Name = "Remember me?")]
public bool RememberMe { get; set; }
}
```

I file all'interno della cartella Views rappresentano naturalmente le pagine dell'applicazione web, cioè l'interfaccia grafica.

Come noterete già dall'estensione .cshml, essi utilizzano il view engine di Razor e, quindi, la sintassi inline già vista nei paragrafi precedenti. Inoltre, grazie ad apposite istruzioni è possibile associare ogni view a un model. Per esempio, la prima riga del file *login.cshtml* contiene l'istruzione seguente:

```
@model MVCApplication.Models.LoginViewModel
```

La direttiva `@model` associa il model `LoginViewModel` alla vista attuale.

Eseguendo l'applicazione creata da Visual Studio, senza nessuna modifica, verrà avviato il browser che mostrerà la web app servita da IIS Express.

ASP.NET MVC da zero

Per comprendere meglio i meccanismi, proviamo ora a creare un nuovo progetto ASP.NET MVC partendo da zero. Quindi, utilizzando ancora Visual Studio, scegliamo il template Empty aggiungendo il supporto MVC per fare in modo che l'IDE crei la struttura di cartelle vista prima e aggiunga i riferimenti alle librerie necessarie.

Una volta assegnato il nome, per esempio `HelloMVC`, e creato il progetto, si noti come all'interno del Solution Explorer siano state create anche in questo caso le cartelle Controllers, Models e Views, vuote perché abbiamo scelto il template di base.

Lanciando l'applicazione in debug, nel browser verrà restituito un errore 404 di risorsa non trovata, e in effetti non abbiamo ancora creato nessuna pagina.

Come detto prima, le richieste vengono gestite in ASP.NET MVC dai controller, quindi il primo passo è la creazione di una nuova classe derivata da `Controller`. In Visual Studio è possibile farsi aiutare facendo clic con il pulsante destro sulla cartella Controllers nel Solution Explorer, quindi Add e infine Controller.

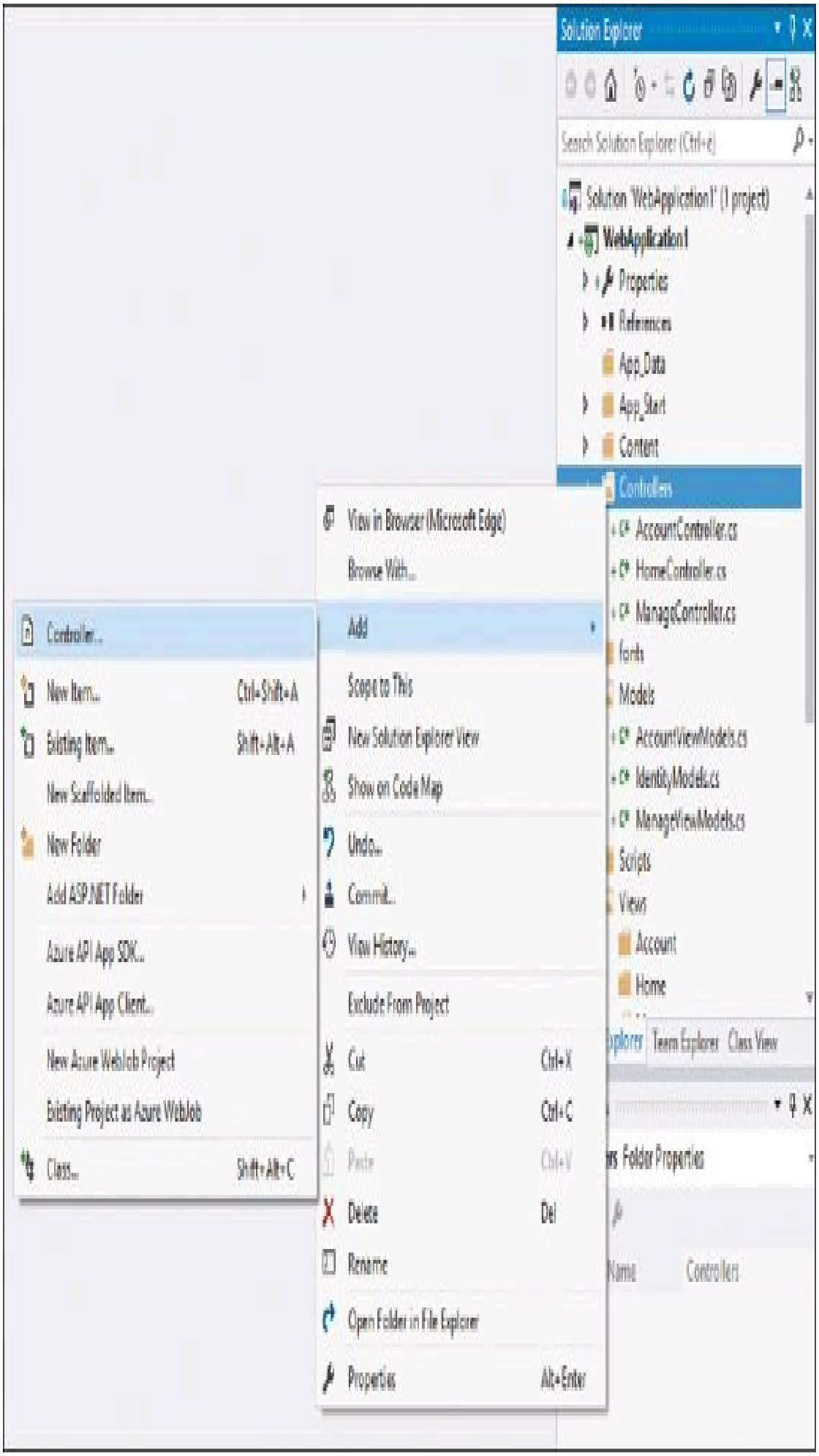

Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'WebApplication1' (1 project)
WebApplication1
Properties
References
App_Data
App_Start
Content
Controllers
AccountController.cs
HomeController.cs
ManageController.cs
fonts
Models
AccountViewModels.cs
IdentityModels.cs
ManageViewModels.cs
Scripts
Views
Account
Home
View in Browser (Microsoft Edge)
Browse With...
Add
Scope to This
New Solution Explorer View
Show on Code Map
Undo...
Commit...
View History...
Exclude From Project
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Delete Del
Rename
Open Folder in File Explorer
Properties Alt+Enter
Controller...
New Item... Ctrl+Shift+A
Existing Item... Shift+Alt+A
New Scaffolded Item...
New Folder
Add ASP.NET Folder
Azure API App SDK...
Azure API App Client...
New Azure Weblob Project
Existing Project as Azure Weblob
Class... Shift+Alt+C
Team Explorer
Class View
Folder Properties
Name
Controllers

Figura 17.12 – Creazione di un controller per l'applicazione ASP.NET MVC.

In Visual Studio, si aprirà a questo punto una finestra per la scelta della tipologia di controller da creare; selezionate MVC 5 Controller Empty (o una voce simile, a seconda della versione di MVC installata) e quindi assegnate un nome. Per convenzione, ogni controller termina con la parola "Controller" e deve naturalmente avere un nome descrittivo: qui assegnate il nome HomeController e confermate la vostra scelta. Verrà creata una classe simile alla seguente nella cartella Controllers:

```
public class HomeController : Controller
{
//
// GET: /Home/
public ActionResult Index()
{
return View();
}
}
```

Mentre all'interno della cartella Views è stata creata la sottocartella Home, destinata a contenere il file con la vista relativa a Home.

Il metodo View invocato in Index cercherà quindi di creare un oggetto ViewResult, che renderizzerà la vista inviandola come risposta alla richiesta ricevuta. Se proverete a eseguire l'applicazione, vi verrà restituito un errore differente, in quanto il view engine non riuscirà a trovare alcuna vista all'interno della cartella Home.

Per il momento, quindi, modificate il metodo Index nel seguente modo:

```
public string Index()
{
return "Hello MVC";
}
```

Così facendo, l'applicazione in esecuzione mostrerà sulla pagina iniziale una semplice stringa di testo.

Ma come ha fatto l'applicazione a collegare la pagina iniziale al controller HomeController?

Per capirne il funzionamento, bisogna intanto sapere che MVC utilizza il sistema di *routing* di ASP.NET, che si occupa di collegare gli indirizzi inseriti nel browser ai controller e alle relative azioni.

Il progetto creato con il template di Visual Studio contiene una cartella App_Start, all'interno della quale vi è un file *RouteConfig.cs* contenente a sua volta la configurazione di un percorso predefinito:

```
public class RouteConfig
```

```
{
public static void RegisterRoutes(RouteCollection routes)
{
routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

routes.MapRoute(
name: "Default",
url: "{controller}/{action}/{id}",
defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
);
}
}
```

In tal modo la richiesta alla radice del sito / verrà reindirizzata al controller Home. Non solo, come potete notare dal formato del parametro url, anche le richieste nel formato /Home e /Home/Index porteranno all'esecuzione della stessa azione. Quindi, il corretto funzionamento dell'applicazione deriva dal fatto che abbiamo seguito la convenzione corretta di chiamare il primo controller con il nome HomeController.

Il passo successivo è quello di produrre una pagina HTML un po' più complessa di quella precedente, mediante la realizzazione di una vera e propria view.

Intanto è necessario riportare il metodo Index, modificato in precedenza, al suo contenuto iniziale:

```
public ActionResult Index()
{
return View();
}
```

In questo modo, l'oggetto ViewResult restituito dal metodo indica al view engine di creare e restituire al browser una view, in questo caso invocando il metodo View. MVC cercherà di creare la vista predefinita, che non esiste e che restituirà nel browser l'errore evidenziato in precedenza.

Lo stesso errore mostra i percorsi all'interno dei quali MVC ha tentato di trovare la view: le cartelle Home oppure Shared, con file del tipo *Index.**, con estensioni del tipo .aspx, .ascx, .cshtml e .vbhtml.

Passiamo quindi alla creazione della view associata al controller Home. In Visual Studio basta fare clic con il pulsante destro sul nome o sul contenuto del metodo Index, contenuto nel file *HomeController.cs*, e quindi selezionare dal menu la voce Add View. Nella finestra di configurazione della view, deselezionate la casella Use a layout page e cliccate su Add. Visual Studio creerà il file *Index.cshtml*, cioè un file che utilizza l'engine Razor, all'interno della cartella Views/Home, con il seguente contenuto standard:

```
@{
```

```
Layout = null;
}
```

```
<!DOCTYPE html>
<html>
<head>
<meta name="viewport" content="width=device-width" />
<title>Index</title>
</head>
<body>
<div>
</div>
</body>
</html>
```

Come noterete, il contenuto è quasi tutto composto da normale HTML, a eccezione del blocco iniziale, che sarà interpretato dall'engine Razor e che indica semplicemente che la pagina non utilizza, come indicato da noi, nessuna pagina di layout.

Potete modificare l'HTML a vostro piacimento, per esempio inserendo il testo Hello MVC all'interno del div principale, ed eseguire nuovamente l'applicazione, che stavolta aprirà la pagina *index.cshtml* appena creata. Per rendere dinamico tale contenuto, possiamo utilizzare l'oggetto dynamic ViewBag, all'interno del metodo Index, impostando per esempio una proprietà Hello con una stringa:

```
public ViewResult Index()
{
ViewBag.Hello = "Hello MVC from Controller";
return View();
}
```

La proprietà ViewBag può essere quindi utilizzata all'interno della view con la sintassi Razor:

```
<div>
<h1>@ViewBag.Hello</h1>
</div>
```

In questo modo, quando MVC invocherà il metodo View all'interno del metodo Index del controller, verrà caricato il file *index.cshtml* e Razor effettuerà il parsing, processando l'espressione appena inserita.

Visual Studio e ASP.NET

Sebbene per sviluppare applicazioni web con ASP.NET sia sufficiente un editor di testo come Notepad, anche in ambito web abbiamo visto che Visual Studio fornisce strumenti e funzioni con le quali è possibile velocizzare sia la parte di design del layout, sia quella di programmazione della logica nei file di code behind.

Riepilogando, per creare un'applicazione web basata su ASP.NET e con l'ausilio di Visual Studio, si inizia come sempre dal menu File -> New Project, selezionando poi la categoria

Web e quindi la tipologia ASP.NET Web Application.

NOTA	A partire dalla versione 2013 di Visual Studio, sono state apportate diverse novità nello sviluppo ASP.NET e quindi anche nei template di progetto. In particolare, è ora possibile mischiare i diverse possibili approcci architetturali di sviluppo, per esempio iniziando un progetto basato sul pattern MVC e poi aggiungendo delle web form o ancora il supporto di Web API al progetto iniziale. Se utilizzate una versione precedente di Visual Studio, potreste quindi notare qualche differenza rispetto a quanto descritto in questo paragrafo.

Una volta confermata la creazione del progetto web, con l'impostazione del percorso e del nome dell'applicazione, verrà aperta una nuova finestra di selezione del template (vedere la Figura 17.10), nella quale selezionare la tecnologia ASP.NET di base per l'applicazione web, con la possibilità di aggiungere sin da ora i riferimenti necessari per l'utilizzo delle altre tecniche possibili.

Oltre a quelli visti nei paragrafi precedenti, come web form e MVC, Visual Studio fornisce anche i template necessari a creare applicazioni basate sulla modalità Single Page Application o applicazioni da ospitare all'interno di Facebook, oltre al template per lo sviluppo di servizi REST basati sulle Web API ASP.NET.

Riepilogo

Questo capitolo ha concluso l'avventura nel mondo dello sviluppo in C#, mostrando le opportunità pratiche che si presenteranno una volta che sarete diventati degli sviluppatori più o meno esperti. Probabilmente vi dedicherete a degli ambiti particolari, a seconda del core business della vostra azienda, oppure dei vostri interessi.

Quello che ho cercato di farvi capire è che .NET, e quindi C#, costituiscono una potenzialità molto elevata per poter entrare in diversi mondi legati allo sviluppo software.

Potete dedicarvi allo sviluppo di applicazioni desktop per Windows con WPF o Windows Forms, di applicazioni universali per Windows 10 da pubblicare sullo store ed eseguire anche su smartphone, di applicazioni web in ASP.NET, e non sono certamente le uniche possibilità.

Diversi progetti, piattaforme, librerie, permettono di utilizzare C# su piattaforme un tempo lontanissime da quella Microsoft, per esempio iOS e Android, oppure Linux e Mac OS X.

Una cosa è certa: con C# non starete a guardare i programmatori che hanno scelto altre piattaforme e linguaggi.

Stringhe ed espressioni regolari

Le classi String e StringBuilder del .NET Framework forniscono le proprietà e i metodi necessari a trattare le stringhe di testo. Per utilizzare invece le espressioni regolari, C# fornisce la classe Regex e i suoi membri.

Ogni applicazione, non solo C#, necessita di manipolare e trattare stringhe di testo, quindi è fondamentale poter fare affidamento su classi che permettano di utilizzare metodi e proprietà su tale tipologia di dati.

Questa appendice è dedicata a un rapido riepilogo delle funzionalità offerte dalle classi String e StringBuilder; se le funzionalità di ricerca offerte da String non fossero sufficienti, si potrà fare affidamento sulle espressioni regolari, utilizzabili per mezzo della classe Regex.

La classe **String**

Tutte le stringhe, cioè le sequenze di caratteri che rappresentano dei testi, sono in .NET istanze della classe String, che fa parte del namespace System, in quanto tipo fondamentale del framework. Inoltre, come ben sappiamo, essendo un tipo primitivo C# permette di utilizzare un apposito alias per il tipo System.String, denominato string.

Una stringa rappresenta un insieme immutabile e ordinato di caratteri, e in particolare in .NET le stringhe sono sequenze di caratteri *Unicode*, a loro volta istanze del tipo System.Char.

La classe String fornisce metodi e proprietà per eseguire molte operazioni fondamentali sulle stringhe che riepilogheremo nei seguenti paragrafi.

Costruzione di stringhe

String è un tipo riferimento, ereditato direttamente da System.Object e non ulteriormente derivabile, in quanto classe sealed.

Per costruire una stringa a partire dalla sequenza di caratteri che essa rappresenta, si utilizza l'assegnazione diretta del suo valore letterale fra doppi apici:

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

Non è possibile utilizzare in tal caso l'operatore new passando la stringa come argomento:

```
string str=new string("hello world"); //ERRORE
```

Altri costruttori permettono però di passare come argomento l'array di char e costruire una stringa a partire da esso o da una sua porzione:

```
str = new string('a', 10); // "aaaaaaaaaa"

char[] arr={'a','b','c','d','e'};
str = new string(arr, 1, 3); //"bcd"
```

Le stringhe possono anche contenere *sequenze di escape* formate con il carattere backslash (\), sia per inserire caratteri particolari sia per rappresentare caratteri Unicode di cui si conosce il codice, facendo precedere il relativo valore dalla sequenza \u:

```
str = "\\\u0045"; // \\ rappresenta il carattere \, \u0045 rappresenta il carattere E
```

Le stringhe possono anche essere scritte con valori letterali preceduti dal carattere *verbatim* @.

Il vantaggio di usare il carattere @ è che le sequenze di escape all'interno della stringa non verranno processate, ma interpretate letteralmente:

```
string str=@"c:\temp\file\a.txt"; //non è necessario raddoppiare i \
```

Per includere i doppi apici all'interno di una stringa costruita con @, è necessario raddoppiare il carattere ":

```
string str=@""""Questa stringa è fra doppi apici" // "Questa stringa è fra doppi apici"
```

Usando l'operatore + è possibile concatenare due stringhe:

```
str="hello";
str=str+"world"; //helloworld
```

Essendo il tipo string immutabile, in realtà viene costruita una nuova istanza come risultato dell'operazione di concatenamento fra str e "world".

Fra i metodi per ottenere una stringa, non dimenticate il metodo ToString della classe Object, che è ereditato da qualsiasi oggetto: ogni classe può implementarne un override per produrre una rappresentazione testuale delle proprie istanze.

Stringhe nulle o vuote

Una stringa vuota è un'istanza della classe String che non contiene caratteri e quindi è usata per rappresentare un campo di testo vuoto.

Una stringa vuota è un oggetto perfettamente valido e può essere creata così:

```
string empty="";
int len= empty.Length; //0
```

oppure utilizzando la proprietà Empty:

```
empty=String.Empty;
```

Una stringa null invece, come qualunque altro oggetto null, non è un riferimento valido. Quindi, se si tentasse di invocarne metodi e proprietà, lancerebbe una NullReferenceException:

```
str=null;
int len=str.Length; //eccezione
```

È invece possibile utilizzare una stringa nulla nelle concatenazioni e in tal caso verrà considerata come una stringa vuota:

```
String sn=null;
str="abc"+sn; //abc
```

La classe String fornisce anche il metodo statico IsNullOrEmpty per verificare se una stringa è nulla oppure vuota:

```
bool isNullOrEmpty = string.IsNullOrEmpty(str);
```

Il metodo IsNullOrWhiteSpace, invece, restituisce true anche se la stringa è formata solo da spazi bianchi:

```
string space = " ";
bool isNullOrEmptyOrSpace = string.IsNullOrWhiteSpace(space);
```

Esaminare le stringhe

Per ottenere la lunghezza di una stringa, si può utilizzare la proprietà Length:

```
string str="hello";
int len=str.Length; //5
```

Per ricavare un carattere in una data posizione, è possibile utilizzare l'operatore [] con l'indice del carattare da ottenere:

```
char ch=str[0]; //h
```

In tal modo, è possibile scrivere un ciclo for o foreach per iterare lungo tutti i caratteri:

```
for(int i=0;i<str.Length;i++)
{
Console.WriteLine(str[i]);
}

foreach (char ch in str)
{
Console.WriteLine(ch);
}
```

Se interessa ottenere l'array di caratteri che compongono la stringa, è possibile utilizzare il metodo ToCharArray:

```
char[] chars=str.ToCharArray(); //h,e,l,l,o
chars=str.ToCharArray(0,2); // h,e
```

Per verificare se una stringa contiene un determinato carattere, è possibile usare il metodo IndexOf, che restituisce l'indice del primo carattere trovato oppure -1:

```
str = "hello world";
int index = str.IndexOf('h'); //0
```

IndexOf può anche accettare altri parametri per limitare la ricerca a partire da un certo indice di partenza e per una data lunghezza.

Analogamente, il metodo LastIndexOf e i suoi overload ricercano l'ultima occorrenza di un carattere:

```
int index = str.LastIndexOf('l'); //3
```

Il metodo IndexOfAny funziona come IndexOf, ma può esaminare l'occorrenza di uno qualunque dei caratteri contenuti nell'array passato come parametro:

```
index= str.IndexOfAny( new char[]{'l', 'w'}); //2
```

LastIndexOfAny, invece, esegue la ricerca per l'ultimo carattere.

Volendo verificare se una stringa inizia o finisce con una particolare sequenza di caratteri, sono disponibili i metodi StartsWith ed EndsWith:

```
str = "hello world";
bool sb = str.StartsWith("hel"); //true
bool eb = str.EndsWith("abc"); //false
```

Il metodo Contains consente di verificare se una data stringa contiene un'altra stringa al suo interno:

```
str = "hello world";
bool cb=str.Contains("wo");//true
bool cb=str.Contains("abc");//false
```

Confronto di stringhe

Sebbene String sia un tipo riferimento, gli operatori == e != non agiscono sui riferimenti, ma direttamente sui valori letterali.

In questo modo è possibile confrontare due stringhe per l'uguaglianza usando tali operatori:

```
string str1 = "hello";
bool eq = str1 == "hello"; //true
```

Gli operatori == e != utilizzano internamente il metodo Equals, che verifica se le due stringhe contengono gli stessi caratteri. Esso ha diversi overload ed è disponibile anche come metodo statico di String:

```
str1="hello";
str2="HELLO";
eq = str1.Equals(str2); //false
eq = str1.Equals(str2, StringComparison.OrdinalIgnoreCase); //true
```

Nel secondo caso, il precedente esempio esegue il confronto ignorando maiuscole e minuscole. In realtà, c'è da notare che il .NET Framework utilizza anche dei meccanismi particolari per evitare di mantenere in memoria più istanze della stessa sequenza di caratteri:

```
string str1,str2;
str1="hello";
str2="hello";
```

Per verificare se due riferimenti puntano allo stesso oggetto, è possibile utilizzare il metodo statico ReferenceEquals di Object. In questo caso, anche se le due variabili sono diverse, puntano a una stessa stringa in memoria:

```
bool b= Object.ReferenceEquals(str1,str2); //true
```

Tale meccanismo è detto *interning*. Quando viene creata la seconda stringa, il CLR verifica se ne esiste già una uguale e, in caso affermativo, le assegna un riferimento.

La classe String fornisce altri metodi per eseguire operazioni di confronto.

Se, oltre a determinare l'uguaglianza o meno di due stringhe, si vuole anche ottenere un ordinamento alfabetico, è possibile utilizzare i metodi Compare, CompareOrdinal e

CompareTo.

Il primo è un metodo statico di String che restituisce un valore intero il quale, a seconda del valore di due stringhe A e B, sarà:

- = 0, se A è uguale a B;
- < 0, se A è minore di B;
- > 0, se A è maggiore di B.

Ecco un esempio:

```
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 accetta anche altri parametri per perfezionare la modalità di confronto: per esempio, per eseguirlo tenendo conto di maiuscole e minuscole o meno.

Il metodo CompareOrdinal esegue un confronto carattere per carattere; verifica in tal modo anche l'uguaglianza logica di eventuali caratteri particolari, per esempio contenuti in stringhe scritte in lingue straniere:

```
bool b=String.CompareOrdinal("Strass", "Straß");//restituisce false
b=String.Compare("Strass", "Straß");//restituisce true
```

In questo caso, in lingua tedesca il carattere ß è considerato equivalente a ss.

Il metodo CompareTo è invece un metodo di istanza e quindi viene invocato su una stringa, passando come parametro la seconda stringa con cui effettuare il confronto. I valori restituiti sono gli stessi del metodo statico Compare.

Manipolazione di stringhe

Svariati metodi consentono di manipolare e modificare stringhe, e di ottenere da esse caratteri o sottostringhe. Ognuno di questi metodi ha più overload per agire su diversi numeri o tipi di parametri.

Il metodo Concat concatena fino a quattro stringhe o oggetti; nel caso di oggetti, verrà invocato il metodo ToString per ottenerne la rappresentazione testuale:

```
string str1="hello";
string str2="world";
string str3=String.Concat(str1,str2); //helloworld
```

Il metodo CopyTo copia un determinato numero di caratteri da una certa posizione in una stringa e li inserisce in un array:

```
chars=new char[3];
```

```
str3.CopyTo(0, chars, 0, chars.Length); //H,e,l
```

Il metodo Insert restituisce una nuova stringa con una sottostringa inserita in una certa posizione:

```
str3 = str1.Insert(0, "World");//WorldHello
```

Il metodo Join unisce un array di stringhe o oggetti, utilizzando un eventuale stringa di separazione:

```
str3 = String.Join("-", str1, str2); //Hello-World
```

I metodi PadLeft e PadRight consentono di allineare la stringa all'interno di un campo di una certa lunghezza, aggiungendo dei caratteri rispettivamente a sinistra o a destra, fino a raggiungere la specificata larghezza totale. Inoltre, può essere specificato un carattere di riempimento che, come valore predefinito, sarà lo spazio:

```
string str1="hello";
str3 = str1.PadLeft(10); //" hello"
str3 = str1.PadRight(10, '.'); //"hello....."
```

Il metodo Remove restituisce una stringa ottenuta eliminando un determinato numero di caratteri da una stringa originale, oppure tutti i caratteri a partire da un certo indice:

```
str3 = str1.Remove(2); //He
str3 = str1.Remove(0, 2); //llo
```

Il metodo Replace crea una nuova stringa sostituendo tutte le occorrenze di un carattere con un altro carattere, o le occorrenze di una sottostringa con un'altra stringa:

```
str3 = str1.Replace('e', '3'); //h3llo
str3 = str1.Replace("ll", String.Empty); //heo
```

Il metodo Split crea un array di stringhe suddividendo una stringa originale. È possibile indicare uno o più caratteri o stringhe da utilizzare come separatori:

```
string longString="Ecco una stringa lunga";
string[] splitted = longString.Split(' '); //crea un array cone le 4 parole separate
```

Per estrarre una sottostringa da una originale, da un determinato indice fino al termine o per una certa lunghezza, si può usare il metodo Substring:

```
str3 = "helloworld".Substring(3, 4);
```

I metodi ToLower o ToUpper restituiscono le versioni convertite in minuscolo o in maiuscolo di una determinata stringa di partenza.

Il metodo Trim rimuove tutti gli spazi da una stringa, mentre TrimEnd e TrimStart rimuovono gli spazi rispettivamente alla fine e all'inizio della stringa originale.

NOTA

Per esaminare e manipolare stringhe e collezioni di stringhe, tornano notevolmente utili gli operatori e i metodi di estensione forniti da LINQ. Essi possono essere combinati con i metodi della classe String e con le espressioni regolari per ottenere funzionalità molto efficaci nel trattamento di testi.

Formattazione di stringhe

Il metodo statico Format fornisce una funzionalità molto utile per la costruzione di stringhe in maniera parametrica e dinamica. In particolare, è in questo modo possibile ricavare delle stringhe a partire da variabili o altri oggetti, senza dover ricorrere alla concatenazione e all'utilizzo dell'operatore +.

Fra l'altro, concatenare più stringhe potrebbe provocare dei decadimenti di prestazioni, visto che, poiché le stringhe in .NET sono immutabili, ogni operazione di tale tipo si traduce nella creazione di nuove istanze di stringhe in memoria.

Per utilizzare il metodo Format, è necessario indicare una *stringa di formato* seguita dal preciso numero di parametri che essa si attende. Per esempio:

```
string titolo="IT";
string autore="Stephen King";
int pagine=1317;
string formatted = String.Format("Titolo del libro {0}, Autore {1}, pagine {2}", titolo, autore, pagine);
// Titolo del libro IT, Autore Stephen King, pagine 1317"
```

In questo caso, la stringa di formato contiene tre parametri, indicati mediante un indice fra parentesi graffe, a partire da 0, che verrà sostituito dal corrispondente oggetto passato come parametro del metodo.

Oltre all'indice, ogni parametro di formato può anche essere seguito da una virgola accompagnata dalla minima larghezza che deve occupare nella stringa finale e dai due punti seguiti a loro volta da una stringa di formato del parametro. La larghezza minima è utile per allineare il testo; il valore può essere negativo per allineare a sinistra o positivo a destra:

```
formatted = String.Format("Titolo: {0, -20} prezzo: {1,10:c}", titolo, 10);
Console.WriteLine(formatted);
//Titolo: IT prezzo: € 10,00
```

In questo esempio, il parametro intero 10 viene formattato con la stringa di formato c, che sta per "currency", cioè "valuta".

Nel Capitolo 2, abbiamo già elencato e mostrato il funzionamento delle stringhe di formato, usandole con i metodi di scrittura della classe Console. Potete far riferimento a tale capitolo per i valori utilizzabili con il metodo String.Format.

Interfaccia IFormatProvider

Diversi override del metodo String.Format accettano come parametro anche un oggetto che implementa l'interfaccia IFormatProvider, che permette di fornire una formattazione o effettuare l'analisi delle stringhe in maniera personalizzata. In particolare, esistono tre implementazioni standard dell'interfaccia.

La classe NumberFormatInfo fornisce informazioni usate nella formattazione dei numeri, come simboli di valuta e separatori dei decimali e delle migliaia personalizzati per determinate impostazioni di cultura.

La classe DateTimeFormatInfo fornisce informazioni impiegate nella formattazione di data e ora, per esempio ordine e formato dei componenti di anno, mese, giorno, dei separatori, e così via, tutti ancora una volta personalizzati per impostazioni precise di cultura.

La classe CultureInfo rappresenta le impostazioni di una specifica cultura. Un oggetto CultureInfo può essere costruito in diversi modi, per esempio con il costruttore della classe o con il metodo statico CreateSpecificCulture, passando una stringa che rappresenta il nome di una cultura. Ecco il caso dell'italiano:

```
CultureInfo ci=new CultureInfo("it-IT");
```

Utilizzando l'oggetto con il metodo StringFormat per stampare una data e un numero, la stringa risulterà formattata secondo la cultura italiana, per esempio / per separare i componenti della data, il simbolo € per la valuta, la virgola per separare i decimali e il punto per il separatore delle migliaia:

```
string str=String.Format(ci, "prezzo del {0}: {1:c}", DateTime.Now, 1234567.89);
// prezzo del 25/11/2015 16:25:05: € 1.234.567,89
```

Le proprietà NumberFormat e DateTimeFormat permettono di ottenere e modificare le impostazioni per i numeri e le date. Per esempio:

```
ci.NumberFormat.CurrencyDecimalDigits=3;
ci.DateTimeFormat.DateSeparator="-";
```

In tal modo, i numeri saranno formattati con tre cifre decimali e le date avranno il trattino al posto della barra.

Interpolazione di stringhe

C# 6 ha introdotto una nuova sintassi per la formattazione del testo, chiamata *interpolazione di stringhe*, che permette di definire una sorta di template contenente delle espressioni da sostituire a runtime con i valori attuali.

La struttura generale di una stringa interpolata è la seguente:

```
$" <testo> { <espressione interpolata> <,larghezza campo opzionale> <:formato campo> } <testo> ... } "
```

Quindi, basta anteporre il carattere $ e poi all'interno della stringa inserire normalmente il testo, mentre le espressioni vanno racchiuse fra parentesi graffe. Come nel caso della formattazione (vedere il paragrafo precedente), è possibile specificare larghezza, allineamento e formato dell'espressione calcolata. Per esempio:

```
string s = $"{person.Name,-20}: altezza {person.Altezza,10:n2}";
```

Il valore della proprietà Name viene formattato in un campo largo 20 caratteri allineato a sinistra, mentre quello della proprietà Altezza in un campo largo 10, allineato a destra e con due cifre decimali.

Per inserire delle parantesi graffe all'interno della stringa risultato, esse vanno raddoppiate.

Le stringhe interpolate possono essere implicitamente convertite in IFormattable e FormattableString (quest'ultima disponibile in .NET 4.6):

```
decimal p=1.99m;
System.IFormattable ifs = $"{p:c}";
System.FormattableString fss= $"{p:c}";
```

Si ottengono così degli oggetti con cui è possibile esaminare i risultati dell'interpolazione e utilizzarli per formattare le stringhe in culture specifiche (vedere il paragrafo precedente):

```
Console.WriteLine(ifs.ToString(null, new CultureInfo("it-IT"))); //formato italiano, € 1,99
Console.WriteLine(fss.ToString(new CultureInfo("en-US"))); //formato inglese, $ 1.99
```

La classe **StringBuilder**

Il tipo String è immutabile, quindi la creazione, la modifica e il rilascio continui di istanze si tradurranno in un possibile spreco di memoria e nel decadimento delle prestazioni. La classe StringBuilder, contenuta nel namespace System.Text, è stata introdotta per aggirare tali problematiche.

Essa, anziché basarsi sul tipo string, mantiene al suo interno un array di caratteri modificabile con i metodi che la classe stessa fornisce.

Il costruttore di StringBuilder può opzionalmente accettare una stringa iniziale, a partire dalla quale si potranno poi apportare modifiche, aggiunte, rimozioni e sostituzioni:

```
StringBuilder builder = new StringBuilder("Hello");
```

Diverse proprietà permettono di esaminare la lunghezza attuale e la capacità attuale e massima dell'oggetto:

```
Console.WriteLine(builder.Length);
Console.WriteLine(builder.Capacity);
Console.WriteLine(builder.MaxCapacity);
```

Un'istanza di StringBuilder permette di accedere direttamente all'array dei caratteri che sta gestendo per mezzo dell'operatore []:

```
Console.WriteLine("primo carattere: {0}",builder[0]);
```

L'uso principale della classe StringBuilder avviene per mezzo dei metodi Append e AppendLine, che permettono di aggiungere all'istanza attuale un nuovo oggetto, sia di tipo string sia di un altro qualunque tipo predefinito:

```
builder.Append("world");
```

Il metodo AppendLine, dopo aver aggiunto l'eventuale stringa opzionale, va a capo.

Un altro metodo è AppendFormat, il quale aggiunge un nuovo valore all'istanza utilizzando una stringa di formato:

```
builder.AppendFormat("{0}: {1}", "prova", 123);
```

Per ottenere l'attuale stringa memorizzata nello StringBuilder, basta invocare il metodo ToString:

```
string str=builder.ToString();
```

Gli altri metodi della classe sono Insert, Remove e Replace, che permettono di inserire, rimuovere o effettuare sostituzioni al suo interno.

Le espressioni regolari

Se le funzionalità di ricerca e sostituzione fornite dalla classe String non fossero sufficienti per le proprie esigenze, soprattutto per eseguire ricerche di particolari pattern all'interno di testi, sarà necessario, o comunque estremamente comodo, ricorrere all'utilizzo delle espressioni regolari.

Il concetto di *espressione regolare* non è esclusivo di .NET e C# (esistono libri interi su tale argomento), in quanto esso costituisce un vero e proprio linguaggio per l'identificazione di sequenza di caratteri all'interno di altre sequenze.

Un'espressione regolare è un pattern che un motore di tali espressioni utilizza per trovare delle corrispondenze all'interno di un testo di input. Un pattern è formato da uno o più caratteri e comprende eventuali operatori e particolari costrutti.

Le espressioni regolari trovano parecchia utilità in ambiti applicativi, come la validazione di valori da inserire in moduli (per esempio, per controllare la validità di un indirizzo email o di un codice fiscale), per ricavare particolari stringhe scritte in un dato formato (per esempio, con un'espressione regolare si possono trovare tutti gli url presenti in una pagina HTML, o tutti i tag HTML e così via), o per elaborare un testo effettuando delle sostituzioni parametriche.

La classe Regex

Un'espressione regolare è rappresentata in .NET dalla classe Regex, contenuta nel namespace System.Text.RegularExpressions. In particolare, è il metodo Match che si occupa di effettuare la verifica e la ricerca di un'espressione regolare all'interno di una stringa di input.

Proviamo intanto a vedere un paio di esempi, prima di mostrare il particolare linguaggio utilizzato per la costruzione di espressioni:

```
string pattern=@"\d{2}";
string input="oggi ci sono 25 gradi";
Regex rex=new Regex(pattern);
Match match=rex.Match(input);
if(match.Success)
{
Console.WriteLine(match.Value);
Console.WriteLine(match.Index);
Console.WriteLine(match.Length);
}
```

La variabile pattern contiene un'espressione regolare che indica, in questo caso, una sequenza di due numeri interi. Più precisamente, la sequenza \d indica una cifra, mentre {2} è un

quantificatore che indica la presenza di due occorrenze della sequenza precedente.

Nel caso in cui non siano specificati caratteri speciali, come quelli usati nell'esempio precedente o quelli che vedremo nel seguito, verrà ricercata una corrispondenza del testo esatto utilizzato come pattern. Per esempio, se come pattern utilizziamo la stringa "i", con lo stesso input precedente, otterremo tre corrispondenze; verrà infatti ricercata la stringa esatta, che nella frase appare in tre posizioni:

```
MatchCollection matches = Regex.Matches(input, "i");
Console.WriteLine("trovate {0} corrispondenze di 'i' in {1}", matches.Count, input);
foreach(Match m in matches)
{
Console.WriteLine("indice {0}", m.Index);
}
```

Poiché il carattere backslash ha un significato speciale e lo si userà in maniera abbastanza comune nella definizione di espressioni regolari, spesso i pattern utilizzano il carattere *verbatim* @ come prefisso della stringa.

Per mezzo del metodo Match, si verifica se la stringa di input contiene una tale sequenza e, in caso affermativo determinato dal valore della proprietà Success, è possibile ricavarne il valore trovato per mezzo della proprietà Value.

La proprietà Index restituisce invece l'indice all'interno della stringa originale con il quale si è trovata la corrispondenza, mentre la proprietà Length rappresenta la lunghezza del risultato in esame. La stessa espressione può essere utilizzata per mezzo della versione statica del metodo Match:

```
Match match= Regex.Match(input, pattern);
```

Una stringa può anche contenere più risultati. In tal caso è possibile utilizzare la proprietà Matches per ottenere una loro enumerazione, esattamente di tipo MatchCollection.

Nel seguente esempio, vengono ricercate tutte le parole di una frase che hanno una lunghezza di due caratteri. Le parole sono identificate come stringhe che all'inizio o alla fine hanno uno spazio bianco, indicato dalla sequenza \b; la sequenza \w identifica invece un carattere alfanumerico e con il quantificatore {2} si intende una sequenza di due di questi caratteri:

```
input = "La stringa attuale contiene più parole di lunghezza due caratteri";
pattern= @"\b\w{2}\b";
var risultati= Regex.Matches(input, pattern, RegexOptions.IgnoreCase);
foreach(Match m in risultati)
{
Console.WriteLine(m.Value);
}
```

Con la stringa di input usata nell'esempio si otterranno due diversi risultati, la parola "La" e la parola "di".

Opzioni delle espressioni

Il costruttore di Regex, o altri metodi della classe, tra cui Match e Replace, permettono di specificare delle particolari opzioni con cui eseguire la ricerca dell'espressione regolare.

Tali opzioni sono quelle costituite dai membri dell'enumerazione RegexOptions:

- Compiled – specifica che l'espressione regolare è compilata in un assembly per ottimizzare la velocità di esecuzione;
- CultureInvariant – ignora le differenze culturali della lingua;
- ECMAScript – consente un comportamento conforme a ECMAScript (consentito solo con IgnoreCase, Multiline e Compiled);
- ExplicitCapture – consente solo la cattura di gruppi esplicitamente denominati o numerati (per la cattura di gruppi, vedere più avanti);
- IgnoreCase – ignora maiuscole e minuscole;
- IgnorePatternWhitespace – ignora gli spazi nell'espressione regolare;
- Multiline – modalità multiriga che modifica il significato dei simboli ^ e $ in modo che corrispondano all'inizio e alla fine di ogni riga, e non solo dell'intera stringa di input;
- None – indica che non è stata impostata nessuna opzione;
- RightToLeft – la ricerca avviene da destra a sinistra;
- Singleline – modalità a riga singola.

Per esempio, per invocare il metodo Match su una stringa costituita da più linee, magari ricavata da un file di testo e in maniera case insensitive, si può scrivere:

```
Match match= Regex.Match(input, pattern, RegexOptions.Multiline| RegexOptions.IgnoreCase);
```

Caratteri di escape

Nel linguaggio delle espressioni regolari, alcuni caratteri speciali sono utilizzati per la definizione di particolari tipi di caratteri, sequenze, gruppi e così via. Tali caratteri sono detti anche *metacaratteri*.

Senza di essi, le espressioni regolari non sarebbero così utili e basterebbero i metodi della classe String per ricercare corrispondenze all'interno di un testo.

I metacaratteri usati nelle espressioni regolari sono quelli riportati qui di seguito:

```
\ * + ? | { [ () ^ $ . #
```

Tra le espressioni di qualche paragrafo fa, per esempio, ne abbiamo scritta una regolare del tipo \d{2}, contenente il carattere \ e le parentesi graffe. Nel caso in cui si voglia effettuare la

ricerca in senso letterale di tali caratteri all'interno di una stringa di input, è necessario far precedere al carattere stesso un ulteriore \.

Il seguente esempio cerca di trovare all'interno della stringa di input una corrispondenza della stringa "chiami?", compreso il punto interrogativo, quindi è necessario inserire lo stesso all'interno del pattern, con il prefisso \:

```
input = "Come ti chiami?";
pattern = @"chiami\?";
match = Regex.Match(input, pattern);
if(match.Success)
{
Console.WriteLine(match.Value);
}
```

La tabella seguente elenca i caratteri di escape supportati dalle espressioni regolari in .NET.

Tabella A.1 **- Caratteri di escape.**

Carattere di escape	Descrizione
Tutti i caratteri diversi da . $ ^ { [(|) * + ? \	non hanno alcun significato speciale e quindi, quando preceduti da \, rappresentano se stessi
\a	carattere di allarme (*Bell*)
\b	in un gruppo di caratteri [ ] (vedere il paragrafo seguente), rappresenta il backspace, altrimenti rappresenta un confine di parola
\t	tabulazione
\r	ritorno a capo
\v	tabulazione verticale
\n	nuova linea
\e	carattere di escape
\f	carattere di avanzamento carta
\nnn	carattere ASCII corrispondente al codice ottale indicato con il valore numerico nnn
\xnn	carattere ASCII corrispondente al codice esadecimale di due cifre nn
\cX	carattere di controllo ASCII; per esempio, \cC corrisponde a CTRL+C
\unnnn	carattere Unicode corrispondente al codice di quattro cifre nnnn

Molto utile la sequenza \b per identificare le parole di un testo, in quanto tale carattere di escape indica i confini di una parola. Per trovare le parole che iniziano per a, possiamo utilizzare l'espressione:

```
string pattern = @"\ba";
```

Al contrario, le parole che finiscono per z possono essere identificate con la seguente:

```
string pattern = @"z\b";
```

La classe Regex fornisce anche il metodo Escape per evitare di inserire a mano il carattere di escape:

```
pattern = Regex.Escape("?"); \\restituisce "\?"
```

Il metodo Unescape esegue la procedura inversa.

Come visto in precedenza, il carattere @ consente di interpretare le stringhe in maniera letterale. Quindi, nel caso in cui esse contengano il carattere \, lo stesso backslash sarà considerato come carattere di escape da utilizzare in un'espressione regolare.

Per ogni carattere diverso dai metacaratteri prima elencati, verrà ricercata una corrispondenza esatta del carattere stesso o della stringa formata da tali caratteri. In pratica, in tal modo, l'uso dell'espressione regolare è probabilmente inutile ed è sostituibile semplicemente con l'utilizzo dei metodi IndexOf o Contains della classe String.

In altri casi, il backslash serve a indicare particolari caratteri (vedere la Tabella A.1): per esempio, \t indica un carattere di tabulazione, \n indica un carattere di newline, \uxxxx indica il carattere Unicode corrispondente al codice numerico indicato con xxxx e così via.

Classi di caratteri

Molto spesso è utile indicare particolari classi o tipologie di caratteri anziché specifici caratteri o stringhe. Gli insiemi o *classi di caratteri* permettono di ricercare corrispondenze di un carattere appartenente a un dato insieme, all'interno di una stringa.

La Tabella A.2 mostra le diverse modalità per indicare insiemi di caratteri.

Tabella A.2 **- Classi di caratteri.**

Espressione	Descrizione
[abc]	ricerca uno dei caratteri contenuti nell'insieme fra parentesi
[^abc]	negazione dell'insieme, ricerca un carattere qualsiasi al di fuori dell'insieme indicato fra parentesi
[a-f]	ricerca uno dei caratteri nell'intervallo indicato
.	indica un carattere qualunque, escluso \n; se si vuol ricercare il punto, bisogna utilizzare la sequenza \.
\p{ categoria }	ricerca uno dei caratteri nella categoria specificata
\P{ categoria }	ricerca un carattere al di fuori della categoria specificata

\w	ricerca un carattere alfanumerico, dipende dalla lingua, in generale coincide con l'insieme [a-zA-Z_0-9]
\W	ricerca un carattere non alfanumerico, quindi è la negazione del precedente
\s	ricerca un qualunque spazio
\S	ricerca un qualunque carattere diverso da uno spazio
\d	ricerca una qualunque cifra decimale
\D	ricerca un carattere che non sia una cifra decimale

Per esempio, per trovare occorrenze di uno dei caratteri abcde si può scrivere:

```
input = "hello antonio";
pattern = "[abcde]";
var matches = Regex.Matches(input, pattern);
foreach(Match m in matches)
Console.WriteLine("at {0}: {1}",m.Index, m.Value); //trova e e a
```

Gli insiemi possono essere combinati per creare espressioni più complesse.

Il seguente ricerca una lettera dalla a alla f, seguita da un numero:

```
input = "a1 h1 b2 h2";
pattern = @"[a-f]\d";
matches = Regex.Matches(input, pattern);
foreach (Match m in matches)
Console.WriteLine("at {0}: {1}", m.Index, m.Value);//a1 b2
```

La sequenza \p, oppure \P, seguita da un nome di categoria di caratteri racchiusa fra parentesi graffe accetta uno dei nomi indicati nella Tabella A.3.

Tabella A.3 - **Categorie di caratteri.**

Categoria	Descrizione
\p{L}	lettere
\p{Lu}	lettere maiuscole
\p{Ll}	lettere minuscole
\p{N}	numeri
\p{P}	punteggiatura
\p{M}	caratteri con segni diacritici e accenti
\p{S}	simboli
\p{Z}	separatori
\p{C}	caratteri di controllo

Ancoraggi

I *caratteri di ancoraggio* specificano la posizione all'interno del testo di input dove verificare la presenza di una corrispondenza.

La Tabella A.3 mostra i caratteri di ancoraggio utilizzabili nelle espressioni regolari.

Tabella A.3 **- Caratteri di ancoraggio.**

Punto di ancoraggio	Descrizione
^	inizio della stringa di input o della riga
$	fine della stringa di input o della riga
\A	solo inizio della stringa di input
\Z	solo fine della stringa, oppure prima di \n alla fine della stringa
\z	la corrispondenza deve verificarsi solo alla fine della stringa
\G	la corrispondenza deve verificarsi solo nel punto in cui è terminata la corrispondenza precedente
\b	confine di parola
\B	la corrispondenza non deve verificarsi su un confine di parola

Per esempio, un'espressione che ricerca l'occorrenza di un numero di una o più cifre all'inizio di una stringa potrebbe essere la seguente, in quanto ^ indica l'inizio della stringa:

```
^[0-9]+
```

Senza il simbolo ^, il pattern troverebbe una corrispondenza per ogni numero all'interno della stringa di input.

Se si utilizza ^ con l'opzione RegexOptions.Multiline (vedere il Paragrafo "Opzioni delle espressioni"), verrà verificata la corrispondenza all'inizio di ogni riga.

Il seguente esempio mostra invece come utilizzare la sequenza \b per verificare le corrispondenze su un confine di parola:

```
string input = "area bare arena mare";
string pattern = @"\bare\w*\b";
Console.WriteLine("Parola che iniziano per 'are':");
foreach (Match match in Regex.Matches(input, pattern))
Console.WriteLine("'{0}' found at position {1}",match.Value, match.Index);
```

Nell'espressione memorizzata nella variabile pattern, il primo \b indica l'inizio di una parola, quindi si ricerca la stringa "are", seguita da zero o più caratteri alfanumerici specificati con \w*, e quindi termina con un confine di parola \b.

Quantificatori

I *quantificatori* consentono di specificare il numero di occorrenze dei caratteri o dei gruppi da ricercare in una stringa.

Tabella A.4 - **Quantificatori.**

Quantificatore	Descrizione
*	zero o più occorrenze
+	almeno una occorrenza
?	zero o una occorrenza
{n}	esattamente n occorrenza
{n,}	almeno n occorrenze
{n,m}	da n a m occorrenze

Tali quantificatori seguono il carattere o il gruppo di caratteri che si vuol ricercare.

Il seguente esempio cerca un numero di cinque cifre all'interno di una stringa:

```
input = "98065 123 abc 9210";
pattern = @"\d{5}";
matches = Regex.Matches(input, pattern);
foreach (Match m in matches)
Console.WriteLine("at {0}: {1}", m.Index, m.Value);//98065
```

Il quantificatore \d{5} permette di scrivere in maniera più concisa l'espressione \d\d\d\d\d.

Nel precedente paragrafo abbiamo già visto un esempio con un quantificatore *:

```
string pattern = @"\bare\w*\b";
```

La parte \w* indica la corrispondenza di una sequenza di caratteri alfanumerici che seguono la stringa "are".

Costrutti di raggruppamento

I *costrutti di raggruppamento* permettono di delineare sotto-espressioni all'interno di un'espressione regolare e catturare le relative corrispondenze in una stringa di input.

Questi costrutti sono utili per diversi scopi, fra cui:

- ricercare le corrispondenze ripetute;
- applicare un quantificatore a una sotto-espressione;
- sostituire le corrispondenze trovate con altre espressioni.

Il modo più semplice per indicare una sotto-espressione è racchiuderla fra parentesi tonde.

```
(sottoespressione)
```

Ogni sotto-espressione così definita e catturata da una stringa di input costituisce un *gruppo*. I gruppi possono essere così ottenuti mediante la proprietà Groups della classe Match.

Il seguente esempio ricava i numeri con prefisso internazionale (cioè specificati nel formato +XX) presenti in una stringa:

```
pattern = @"(\+\d{2})-(\d{3}-\d{4})";
input = "In questa stringa ci sono dei numeri 212-555-6666 +39-932-1111 +415-222-3333 +01-888-9999 alcuni dei quali sono
telefoni con prefisso";
matches = Regex.Matches(input, pattern);

foreach (Match mn in matches)
{
Console.WriteLine("Prefisso nazione: {0}", mn.Groups[1].Value);
Console.WriteLine("numero tel.: {0}", mn.Groups[2].Value);
Console.WriteLine();
}
```

Il pattern `\+\d{2}` indica che il primo gruppo deve iniziare con il carattere `+` seguito da due numeri. Il primo gruppo è poi seguito da un carattere `-` e quindi da un secondo gruppo, costituito da due numeri di tre e quattro cifre, separati a loro volta da un altro trattino.

Con la stringa di input usata sopra si otterrà il risultato seguente:

```
Prefisso nazione: +39
numero tel.: 932-1111

Prefisso nazione: +01
numero tel.: 888-9999
```

Ci si può riferire a un gruppo anche all'interno di una stessa espressione, in maniera da riutilizzare la corrispondenza catturata. Per far ciò, si utilizza il meccanismo del *backreference*, che può usare il numero del gruppo.

Il seguente esempio mostra come trovare parole duplicate all'interno di un testo:

```
pattern = @"(\w+)\s(\1)";
input = "In questa frase frase ci sono sono delle parole ripetute";
foreach (Match mr in Regex.Matches(input, pattern, RegexOptions.IgnoreCase))
Console.WriteLine("Duplicato '{0}'", mr.Groups[1].Value);
```

Il pattern identifica una sequenza di caratteri seguiti da uno spazio e quindi la stessa sequenza a cui ci si riferisce con `\1`.

A ogni sotto-espressione può essere anche assegnato un nome, in maniera da potersi riferire a essa tramite quest'ultimo, e non soltanto tramite il numero come nel caso precedente.

La sintassi da utilizzare nell'espressione per assegnare il nome è la seguente:

```
(?<nome>sotto-espressione)
```

Per riferirsi alla sotto-espressione con un dato nome, invece, si usa la sintassi `\k<nome>`.

Ecco il precedente esempio che trova parole duplicate assegnando il nome dup alla sotto-espressione:

```
pattern = @"(?<dup>\w+)\s(\k<dup>)";
foreach (Match mr in Regex.Matches(input, pattern, RegexOptions.IgnoreCase))
Console.WriteLine("Duplicato '{0}'", mr.Groups["dup"].Value);
```

Anche per accedere alla proprietà Groups, anziché l'indice, viene usato direttamente il nome del gruppo.

Espressioni alternative

Per creare delle espressioni regolari con possibili alternative, cioè inserire una sorta di condizione all'interno, esistono particolari costrutti. Per esempio, per indicare che si vuole trovare una corrispondenza di una fra una serie di alternative, si utilizza il carattere |.

Il seguente pattern identifica la stringa ab oppure cd all'interno di un testo:

```
(ab|cd)
```

Naturalmente, le alternative possono anche essere sotto-espressioni più complesse.

```
string pattern = @"\b(\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b";
```

Il pattern precedente identifica parole intere costituite da numeri nel formato due cifre seguiti da sette cifre, oppure da tre cifre, poi due cifre e infine quattro cifre.

Un'altra possibilità è un costrutto che assomiglia al costrutto ternario di C#:

```
(?( sotto-espressione ) yes | no )
```

Essa permette di indicare una prima sotto-espressione di cui ricercare l'occorrenza, seguita da due sotto-espressioni alternative. Se è stata trovata la corrispondenza relativa alla prima, allora verrà ricercato il pattern yes, altrimenti quello alternativo no.

Sostituzioni

Un altro utilizzo delle espressioni regolari è quello che permette di sostituire le corrispondenze trovate all'interno di un testo con delle stringhe o con altre espressioni.

La classe Regex espone il metodo Replace, con diversi overload per effettuare le sostituzioni. La versione più semplice è quella con due parametri, la stringa di input e l'espressione di sostituzione.

Il seguente esempio crea una istanza di Regex con un pattern per identificare una sequenza di uno o più spazi bianchi e quindi sostituirlo con un solo spazio:

```
input = "Questa stringa ha troppi spazi bianchi ";
pattern = "\\s+";
Regex rgx = new Regex(pattern);
```

```
string result = rgx.Replace(input, replacement);
```

Lo stesso metodo può essere eseguito nella sua versione statica:

```
result=Regex.Replace(input, pattern, replacement);
```

Per effettuare sostituzioni più complesse, in base alle corrispondenze trovate è possibile seguire diverse strategie.

La prima è quella di utilizzare il metodo `Replace` con un secondo parametro di tipo MatchEvaluator, che è un delegate che definisce un metodo personalizzato di sostituzione.

Il seguente esempio ricava tutte le parole presenti in una stringa e le sostituisce con le stesse parole, convertendone però l'iniziale in maiuscolo:

```
input = "una frase tutta in minuscolo";
result= Regex.Replace(input, @"\b[a-z]\w+", delegate(Match matchToEval)
{
string v = matchToEval.ToString();
return char.ToUpper(v[0]) + v.Substring(1);
});
Console.WriteLine("Originale: {0}", input);
Console.WriteLine("Dopo sostituzione: {0}", result); //Una Frase Tutta In Minuscolo
```

I pattern di sostituzione possono contenere anche dei caratteri speciali per definire su quali parti e con che modalità eseguire la sostituzione stessa.

In particolare, nella stringa di sostituzione viene spesso utilizzato il carattere `$` seguito da un numero che indica quello del gruppo catturato dall'espressione regolare, oppure dal nome del gruppo fra parentesi graffe. Per esempio, il criterio `$1` indica che la sottostringa corrispondente deve essere sostituita dal primo gruppo catturato.

Il seguente esempio converte le date presenti nella stringa originale e scritte come mese/giorno/anno, nel formato giorno-mese-anno:

```
input = "12/31/2014";
pattern = @"\b(\d{1,2})\/(\d{1,2})\/(\d{2,4})";
replacement="$2-$1-$3";
result = Regex.Replace(input, pattern, replacement);
Console.WriteLine("{0} convertita in {1}", input, result);
```

L'espressione regolare definisce quindi tre gruppi per catturare il mese, il giorno e l'anno. A questi gruppi si riferisce la stringa di sostituzione per creare il formato `$2-$1-$3`, in quanto il gruppo 2 contiene il mese catturato, il gruppo 1 è il giorno e il 3 l'anno.

Il risultato sarà quindi:

```
12/31/2014 convertita in 31-12-2014
```

Allo stesso modo si possono assegnare dei nomi ai gruppi e utilizzarli nella stringa di sostituzione:

```
pattern = @"\b(?<mese>\d{1,2})\/(?<giorno>\d{1,2})\/(?<anno>\d{2,4})";
replacement = "${giorno}-${mese}-${anno}";
result = Regex.Replace(input, pattern, replacement);
Console.WriteLine("{0} convertita in {1}", input, result);
```

Librerie di espressioni regolari

Spesso capita di dover creare ed elaborare espressioni che probabilmente qualcun altro ha già avuto modo di scrivere e testare. Infatti, cercando su Internet, non sarà difficile trovare librerie di espressioni regolari di vario genere e diversi ambiti applicativi.

Per esempio, fra le espressioni più ricorrenti vi capiterà di dover scrivere quelle per cercare corrispondenze di indirizzi email, url, indirizzi IP, percorsi su file system, codice fiscale, partita IVA, CAP, date e orari in vario formato, numeri di carte di credito e molto altro ancora.

Anziché riportare nel testo questi esempi di espressioni regolari, che sarebbero anche difficili da ricopiare, vi segnalo fra le varie risorse disponibili online i siti regexlib.com e regexhero.net che, oltre a permettere di scrivere e testare le proprie espressioni direttamente online, contengono anche una sezione con decine di esempi pronti all'uso.

Riepilogo

Questa appendice è stata dedicata a classi e metodi utili nel trattamento di testi. La classe String, in particolare, è uno dei tipi primitivi più utilizzati e contiene quindi diverse proprietà e metodi per lavorare con sequenze di caratteri.

Essendo le istanze del tipo string immutabili nell'implementazione di .NET, Microsoft ha introdotto anche la classe StringBuilder, che permette di lavorare con le stringhe in maniera da non sprecare memoria e ottimizzare quindi le prestazioni.

Infine, abbiamo visto cosa sono e come utilizzare le espressioni regolari in C#, con le quali è possibile eseguire operazioni più complesse sulle stringhe, laddove il solo uso del tipo String implicherebbe la scrittura di parecchio codice e senz'altro molte difficoltà in più.

Interoperabilità

C# consente l'utilizzo dei puntatori all'interno di contesti di codice unsafe. Per mezzo dei servizi P/Invoke è inoltre possibile invocare funzioni native di codice non gestito.

In questa appendice verranno introdotti degli argomenti da utilizzare specificamente per l'interoperabilità con codice nativo e puntatori.

Si noti che l'utilizzo di puntatori è abbastanza raro, ma, per completezza, mostreremo qui quali sono le possibilità offerte in tale ambito da C#, principalmente mostrandone le istruzioni disponibili, la modalità e la sintassi da utilizzare.

Contesto unsafe

Per garantire la sicurezza, la gestione automatica della memoria e l'ormai nota sicurezza rispetto ai tipi, C# non supporta come comportamento predefinito l'utilizzo dei puntatori.

È possibile però definire dei metodi o blocchi di codice, identificati dalla parola chiave unsafe, all'interno dei quali l'uso dei puntatori è consentito e quindi programmare con essi in perfetto stile C/C++.

Oltre a ciò è necessario indicare al compilatore csc la presenza di questi blocchi all'interno del codice, quindi bisognerà utilizzare l'opzione di compilazione /unsafe.

Visual Studio invece permette di impostare tale modalità di compilazione direttamente all'interno delle proprietà del progetto, nella sezione Build, selezionando la casella Allow unsafe code.

AppendixB-UnsafeCode - Microsoft Visual Studio

FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ANALYZE DEVEXPRESS WINDOW HELP

Start Debug

Server Explorer Toolbox

AppendixB-UnsafeCode* Program.cs

Application

Build*

Build Events

Debug

Resources

Services

Settings

Reference Paths

Signing

Security

Publish

Configuration: Active (Debug) Platform: Active (Any CPU)

General

Conditional compilation symbols:

Define DEBUG constant

Define TRACE constant

Platform target: Any CPU

Prefer 32-bit

Allow unsafe code

Optimize code

Figura B.1 – Impostazione per compilazione di codice unsafe in Visual Studio.

L'utilizzo di codice unsafe e quindi dei puntatori è richiesto in casi particolari e principalmente per l'interoperabilità con le API Win32 di Windows (in tal caso infatti è spesso necessario interagire con puntatori a strutture) oppure per la scrittura di codice critico dal punto di vista delle performance.

I contesti unsafe definibili in C# sono diversi. Il primo caso è costituito dalle dichiarazioni di tipi come classi, struct, interface e delegate, in cui l'intero corpo del tipo è considerato essere un contesto unsafe:

```
public unsafe struct Node
{
public int Value;
public Node* Left;
public Node* Right;
}
```

La dichiarazione della struct precedente contiene come membri della stessa struttura due campi pubblici Left e Right, definiti come puntatori alla struttura stessa.

Nel secondo caso una dichiarazione di un membro, per esempio un campo, metodo, proprietà o evento di una classe, può includere il modificatore unsafe e, in tal modo, l'intero membro e relativo contenuto (nel caso di proprietà e metodi) costituisce un contesto unsafe.

Nel seguente esempio i due campi Left e Right sono dichiarati come unsafe e quindi possono essere dei puntatori:

```
public struct Node
{
public int Value;
public unsafe Node* Left;
public unsafe Node* Right;
}
```

Infine un blocco di codice può utilizzare dei puntatori dichiarandoli come unsafe:

```
unsafe
{
// unsafe context: qui si possono usare i puntatori
}
```

Per esempio:

```
unsafe
{
int i = 123;

int* pInt1=&i;
int* pInt2;
pInt2 = pInt1;

Console.Write("{0} {1}", *pInt1, *pInt2);
```

}

Tipi puntatore

In contesto *unsafe* è possibile dichiarare un tipo puntatore semplicemente facendo seguire al nome del tipo stesso l'operatore *. Per esempio:

```
int* pInt;
```

definisce un tipo puntatore a int, cioè la variabile pInt punta a una locazione di memoria nella quale è contenuto un valore int.

A differenza dei tipi riferimento, i puntatori non sono gestiti dal Garbage Collector, il quale non ha alcuna conoscenza dei dati ai quali punta il puntatore in oggetto. Per questo motivo non è possibile definire un puntatore a un tipo riferimento o a una struct che contiene riferimenti.

I tipi con i quali è possibile utilizzare i puntatori sono anche detti tipi *unmanaged* e, riassumendo, sono i seguenti:

- sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool;
- tipi enum;
- struct personalizzate senza campi di tipo riferimento;
- un altro puntatore.

Il valore di un puntatore di tipo T* rappresenta l'indirizzo della variabile, cioè della locazione di memoria contenente il valore di tipo T, per esempio il precedente pInt contiene l'indirizzo della variabile, ma non il suo valore.

Gli operatori utilizzati con i puntatori sono quelli elencati nella Tabella B.1.

Tabella B.1 **- Operatori per i puntatori.**

Operatore	Descrizione
*	Operatore dereference: restituisce la variabile contenuta nella locazione puntata.
&	Operatore address-of: restituisce un puntatore all'indirizzo della variabile.
->	L'operatore puntatore a membro permette di accedere al membro di una puntatore a struct.

Ecco un esempio che utilizza tutti e tre gli operatori, con un puntatore alla struct Node vista in precedenza:

```
Node node=new Node();
Node* pn=&node;
```

Il puntatore pn viene inizializzato con l'indirizzo della variabile node:

```
pn->Value = 1; //accede al membro Value
Console.WriteLine(node.Value);
```

Lo stesso risultato si può ottenere accedendo prima alla variabile tramite l'operatore * e quindi impostando il campo Value per mezzo del normale operatore .:

```
(*pn).Value = 2; //
Console.WriteLine(node.Value);
```

Altri operatori permettono di eseguire altre operazioni classiche sui puntatori. Chi avrà la necessità di farlo conoscerà sicuramente la sintassi standard e quindi saprà come usare operatori di incremento, decremento, confronto e così via.

Istruzione fixed

L'istruzione fixed impedisce che il Garbage Collector esegua la rilocazione di una variabile gestita durante il suo funzionamento, bloccandola in una locazione di memoria per un determinato periodo di tempo. Questa tecnica è detta anche *pinning* dichiarativo.

Nell'esempio seguente la variabile pt è una variabile gestita normalmente e quindi soggetta alla procedura di Garbage Collection:

```
unsafe static void TestMethod()
{
Point pt = new Point();

fixed (int* p = &pt.x)
{
*p = 1;
}
}
```

Utilizzando un'istruzione fixed, viene ricavato in questo caso il valore del membro x, assegnandolo al puntatore p. In tal modo all'interno del blocco fixed, la locazione della variabile pt non potrà essere modificata.

L'istruzione fixed può essere utilizzata per creare un buffer con un array di dimensioni fisse, all'interno di una struttura:

```
unsafe struct UnsafeStruct
{

public fixed byte Buffer[30];

}
```

In questo caso l'istruzione fixed ha il significato di bloccare la dimensione dell'array, esattamente a 30 byte.

In generale un array a dimensione fissa viene poi usato con un'altra istruzione fixed, come visto in precedenza, per bloccarne la locazione in memoria:

```
UnsafeStruct us;
fixed (byte* p = us.Buffer)
```

```
{
//utilizzo p
}
```

Istruzione stackalloc

La parola chiave stackalloc consente di allocare un blocco di memoria esplicitamente nello stack, all'interno di un blocco di codice unsafe:

```
int* block=stackalloc int[100];
```

Il seguente esempio mostra come calcolare la sequenza di Fibonacci, passando il numero di risultati al quale fermare la sequenza come argomento del metodo:

```
public unsafe static void Fibonacci(int arraySize)
{
int* fib = stackalloc int[arraySize];
int* p = fib;
//La sequenza inizia con 1, 1.
*p++ = *p++ = 1;
for (int i = 2; i < arraySize; ++i, ++p)
{
// Somma i due precedenti
*p = p[-1] + p[-2];
}
for (int i = 0; i < arraySize; ++i)
{
Console.WriteLine(fib[i]);
}
}
```

L'array contenente i risultati viene puntato dal puntatore fib. L'array è allocato nello stack con l'istruzione stackalloc. La memoria dell'array non è quindi sottoposta alla Garbage Collection e non è necessario utilizzare l'istruzione fixed.

Platform Invoke

Il *Platform Invocation Service*, spesso abbreviato in *P/Invoke*, consente al codice managed di invocare funzioni unmanaged implementate in una DLL nativa.

Non tutte le funzionalità delle API di Windows sono messe a disposizione all'interno delle classi del .NET Framework, in questo caso è necessario ricorrere a P/Invoke.

Allo stesso modo, potreste aver implementato le vostre funzioni in maniera nativa, qualche anno fa magari e quindi, per non reinventare la ruota, potete tranquillamente riutilizzarla per mezzo di P/Invoke.

Il CLR, per mezzo di tali servizi, può caricare in memoria le DLL indicate per mezzo di particolari attributi e si occupa inoltre di tradurre eventuali parametri e valori di ritorno.

Il primo passo per utilizzare una funzione contenuta in una DLL nativa è quello di identificare la sua firma e dichiararla all'interno del codice C#, scrivendo un metodo dotato dei modificatori static e extern. Inoltre è necessario utilizzare l'attributo DllImport per indicare quale DLL contiene la funzione nativa.

NOTA	Un ottimo riferimento per ricavare le firme delle funzioni contenute nelle DLL Win32 è il sito pinvoke.net. Fra l'altro per ogni funzione è anche riportato l'eventuale metodo managed corrispondente.

Per esempio, la visualizzazione di una classica finestra di messaggio è possibile per mezzo della funzione MessageBox contenuta in user32.dll la cui firma C++ è la seguente:

```
int WINAPI MessageBox(
_In_opt HWND hWnd,
_In_opt_ LPCTSTR lpText,
_In_opt_ LPCTSTR lpCaption,
_In_ UINT uType
);
```

Creiamo una classe NativeMethods (potete scegliere il vostro nome a piacimento), destinata a contenere i metodi nativi importati da una DLL e inseriamo il metodo MessageBox con i parametri corrispondenti a quelli della funzione:

```
class NativeMethods
{
[DllImport("user32.dll")]
public static extern int MessageBox(IntPtr hWnd, String text, String caption, int options);
}
```

I metodi extern devono essere sempre static e non includono alcuna implementazione: extern indica appunto che l'implementazione si trova in una libreria esterna. È infatti l'attributo DllImport che indica in quale DLL è contenuta l'implementazione.

Il CLR tenterà a questo punto di trovare all'interno della user32.dll la funzione con la stessa firma.

Per utilizzare il metodo, e indirettamente la funzione MessageBox nativa, basta invocarlo come un normale metodo:

```
static void Main(string[] args)
{
NativeMethods.MessageBox(IntPtr.Zero, "World", "Hello", 0);
}
```

Si noti come il primo argomento, che prevede un handle di finestra, può essere omesso, utilizzando il valore IntPtr.Zero, che rappresenta un puntatore o handle inizializzato a zero.

Non è obbligatorio che il metodo abbia lo stesso nome della funzione. Basta utilizzare il parametro EntryPoint dell'attributo DllImport, che deve essere inizializzato al nome originale della funzione:

```
[DllImport("user32.dll", EntryPoint="MessageBox")]
public static extern int MsgBox(IntPtr hWnd, String text, String caption, int options);
```

In questo caso il metodo corrispondente alla funzione MessageBox è stato chiamato MsgBox.

Spesso le funzioni native utilizzano dei valori di ritorno numerici, che servono a indicare i vari risultati possibili ottenibili dalla funziona stessa.

La stessa funzione MessageBox restituisce un valore intero alla sua chiusura, valore di ritorno che dipende dal pulsante cliccato per chiuderla: per esempio il valore 1 indicherà il clic sul pulsante OK, mentre 2 indica il pulsante Cancel e così via.

In questi casi può quindi essere utile definire le enumerazioni dei possibili valori e utilizzare il tipo enum come tipo di ritorno della funzione:

```
public enum MessageBoxResult : uint
{
Ok = 1,
Cancel,
Abort,
Retry,
Ignore,
Yes,
No,
Close,
Help,
TryAgain,
```

```
Continue,
Timeout = 32000
}
```

La funzione importata da user32.dll potrà quindi essere riscritta come segue:

```
[DllImport("user32.dll")]
public static extern MessageBoxResult MessageBox(IntPtr hWnd, String text, String caption, int options);
```

Set di caratteri

Non tutte le versioni di Windows supportano lo stesso set di caratteri e all'interno delle API sono presenti funzioni che eseguono lo stesso compito, con formati di testo differenti.

L'attributo DllImport permette di specificare il set di caratteri da utilizzare, nel caso di funzioni che trattano con stringhe di testo, per mezzo della proprietà opzionale CharSet.

Nella maggior parte dei casi è possibile utilizzare il valore Auto, ma se si vuol esplicitamente indicare quale versione della funzione utilizzare possiamo usare un valore Ansi o Unicode:

```
[DllImport("user32.dll", EntryPoint="MessageBox", CharSet=CharSet.Ansi)]
public static extern int MsgBox(IntPtr hWnd, String text, String caption, int options);
```

Il valore predefinito, se non indicato altrimenti, è Ansi.

Gestione degli errori

La gestione degli errori è un aspetto fondamentale dello sviluppo e nel caso della programmazione tramite API native di Windows, è spesso fondamentale ricavare il valore dell'ultimo errore verificatosi, per mezzo della funzione GetLastError.

Pertanto, in P/Invoke, sarà quindi necessario impostare a true il valore della proprietà SetLastError dell'attributo DllImport:

```
[DllImport("user32.dll", EntryPoint="MessageBox", CharSet=CharSet.Ansi, SetLastError=true)]
public static extern int MsgBox(IntPtr hWnd, String text, String caption, int options);
```

In tal modo, il CLR memorizzerà l'eventuale valore di errore restituito dalla chiamata alla funzione, e sarà quindi possibile ottenere lo stesso con una chiamata al metodo GetLastWin32Error:

```
NativeMethods.MsgBox(new IntPtr(123132), "Press OK...", "Press OK Dialog", 0);
int error = Marshal.GetLastWin32Error();
```

Nell'esempio precedente viene invocato il metodo MsgBox con un valore non valido per l'handle della finestra. Quindi la finestra stessa non verrà visualizzata e il metodo GetLastWin32Error restituirà un valore non nullo.

Parametri puntatori

Alcune funzioni native possono utilizzare i puntatori, visti nel paragrafo precedente, come parametri.

Tali parametri potranno essere eventualmente inizializzati all'interno della funzione stessa e quindi usati come parametri di ritorno.

In tali casi P/Invoke non richiede che venga scritto del codice unsafe per usare i puntatori anche in C#. Il modo più semplice di convertire le firme di tali funzioni è quello di utilizzare il modificatore ref oppure out.

Per esempio la funzione FileEncryptionStatus della DLL Advapi32.dll serve a verificare se un file è criptato. La firma C++ è la seguente:

```
BOOL WINAPI FileEncryptionStatus(
_In_ LPCTSTR lpFileName,
_Out_ LPDWORD lpStatus
);
```

Se la funzione viene eseguita correttamente, il valore di ritorno sarà un numero diverso da zero, ma il valore dello stato sarà restituito all'interno del secondo parametro, che è un puntatore a un intero a 32bit.

Quindi, per importare la funzione in C#, possiamo utilizzare il modificatore out come segue:

```
[DllImport("Advapi32.dll", CharSet=CharSet.Auto)]
static extern Boolean FileEncryptionStatus(String filename, out UInt32 status);
```

Riepilogo

Questa appendice è stata dedicata a fornire dei cenni sull'interoperabilità possibile fra codice gestito e codice nativo. Il codice unsafe consente di utilizzare i puntatori all'interno di codice C#, mentre P/Invoke è l'insieme di servizi che permettono di invocare funzioni presenti in librerie di codice nativo.

Risposte alle domande

Le risposte alle domande poste alla fine di ogni capitolo sono riportate in questa appendice.

Ognuna è indicata nel formato Domanda=Risposta, per esempio 1=A, indica che la risposta corretta alla domanda 1 è la A.

Capitolo 2

1=C, 2=B, 3=C, 4=D, 5=A

Capitolo 3

1=Falso, 2=D, 3=C, 4=A, 5=A, 6=B, 7=C, 8=A

Capitolo 4

1=A, 2=A, 3=C, 4=B, 5=C, 6=D, 7=C, 8=A

Capitolo 5

1=B, 2=C, 3=D, 4=Vero; 5=A, 6=Vero, 7=B

Capitolo 6

1=C, 2=Falso, 3=D, 4=A, 5=C, 6=D, 7=A, 8=B, 9=Falso, 10=A

Capitolo 7

1=C, 2=B e C, 3=C, 4=D, 5=A, 6=D, 7=C, 8=Vero

Capitolo 8

1=A, 2=B, 3=A, 4=D, 5=Vero, 6=B, 7=A-B-C, 8 = Vero

Capitolo 9

1=B, 2=C, 3=B, 4=D, 5=A, 6=A, 7=B, 8=Vero, 9=B

Capitolo 10

1=A, 2=B, 3=C, 4=D, 5=B, 6=Vero, 7=B

Capitolo 11

1=C, 2=D, 3=Falso, 4=B, 5=D, 6=D, 7=A, 8=B

Capitolo 12

1=B, 2=A, 3=C, 4=A, 5=D, 6=C, 7=Vero, 8=B

Capitolo 13

1=A, 2=C, 3=D, 4=A, 5=B

Capitolo 14

1=A, 2=B, 3=D, 4=A, 5=C e D, 6=B, 7=C, 8=B

Capitolo 15

1=A, 2=B, 3=C, 4=A, 5=D, 6=A, 7= Falso, 8=B, 9=D, 10=C

Capitolo 16

1=B, 2=D, 3=A, 4=Vero, 5=D, 6=Vero