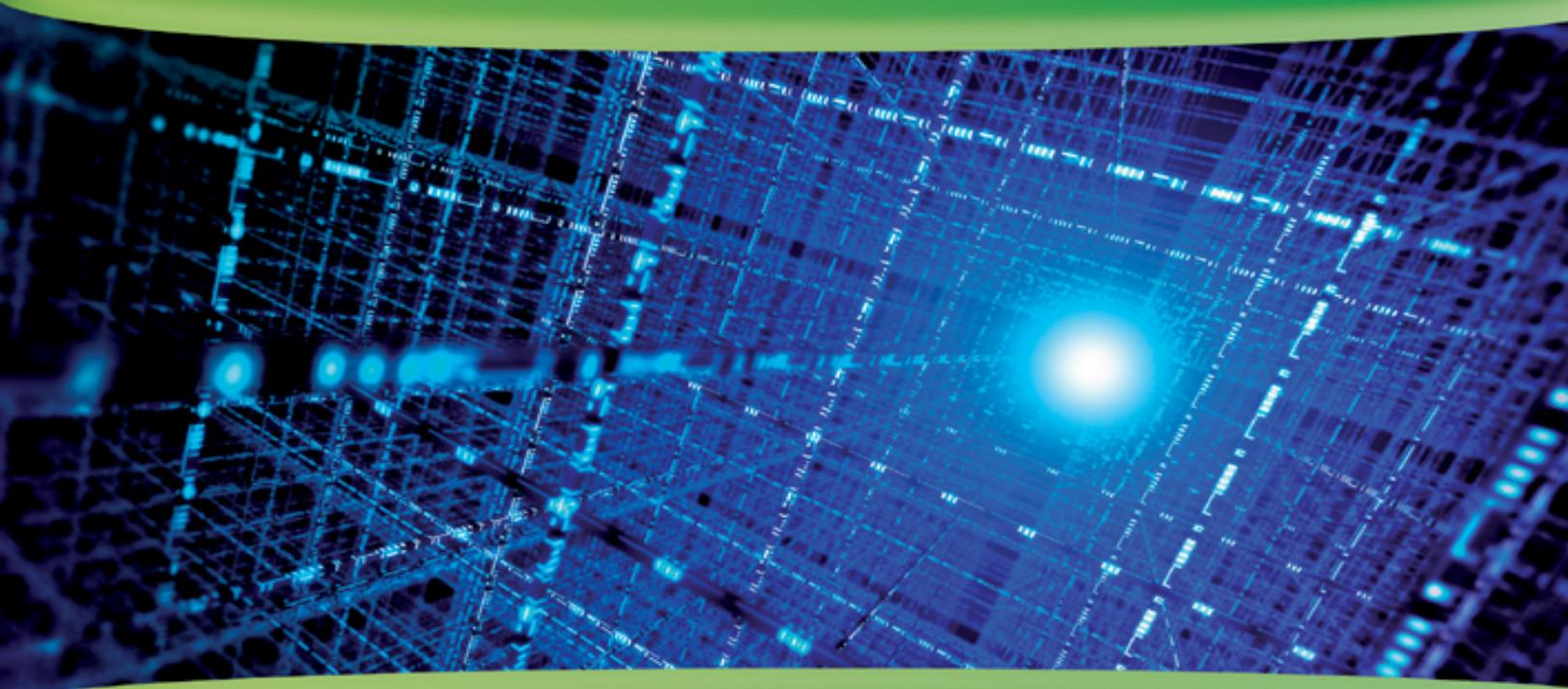


C# 8 E VISUAL STUDIO 2019

**Guida completa
per lo sviluppatore**



**Daniele Bochicchio, Cristian Civera, Marco Leoncini,
Stefano Mostarda, Matteo Tumiatì**

 **aspitalia.com**
The ASP.NET Community

HOEPLI
INFORMATICA

C# 8
e Visual Studio 2019

**Daniele Bochicchio,
Cristian Civera, Marco Leoncini,
Stefano Mostarda, Matteo Tumiati**

**C# 8
e Visual Studio 2019**

**Guida completa
per lo sviluppatore**



EDITORE ULRICO HOEPLI MILANO

Copyright © Ulrico Hoepli Editore S.p.A. 2019

via Hoepli 5, 20121 Milano (Italy)

tel. +39 02 864871 – fax +39 02 8052886

e-mail hoepli@hoepli.it

www.hoepli.it

Seguici su Twitter: @Hoepli_1870

Tutti i diritti sono riservati a norma di legge
e a norma delle convenzioni internazionali

ISBN EBOOK 978-88-203-8968-0

Progetto editoriale e realizzazione: Maurizio Vedovati - Servizi editoriali (info@iltrio.it)

Copertina: Sara Taglialegne

Realizzazione digitale: Promedia, Torino

Nessun albero è stato abbattuto per la realizzazione di questo eBook.

Contenuti del libro

Questa guida completa a C# 8 è l'espressione corale di un gruppo di sviluppatori che utilizza questo linguaggio sin dalla prima versione per costruire applicazioni di ogni tipo, da quelle web fino a complessi sistemi enterprise.

Il libro include le ultime novità introdotte dal linguaggio e dal framework: vengono trattate le basi del linguaggio e mostrati concetti più avanzati, viene spiegato l'uso dell'OOP in C#, per poi passare alle tecnologie: LINQ, Entity Framework, WPF, ASP.NET, XAML, Universal Windows Platform.

C# 8 Guida completa per lo sviluppatore è l'ideale sia per il novizio (che abbia una buona esperienza anche con altri linguaggi di programmazione) sia per chi ha la necessità di apprendere tutte le novità di C# 8, trattando tutto quanto ruota intorno a .NET, da .NET Framework a .NET Core.

Gli autori fanno parte dello staff di ASPIItalia.com, storica community italiana che dal 1998 si occupa di sviluppo su piattaforme Microsoft.

Il libro è suddiviso in cinque parti, ciascuna delle quali risponde a un insieme di esigenze di sviluppo specifiche.

Le informazioni di base, riguardanti .NET, Visual Studio e una prima introduzione al linguaggio compongono la **prima parte**.

Nella **seconda parte** viene approfondito il linguaggio, con tre capitoli che entrano maggiormente nel dettaglio, dopo aver introdotto i principi della programmazione orientata agli oggetti. Chiude questa parte la trattazione di LINQ.

L'accesso e la persistenza dei dati sono invece l'argomento trattato nella **terza parte** del libro, che include ADO.NET, Entity Framework e LINQ to XML.

La **quarta parte** comprende una serie di capitoli completamente dedicata alle tecnologie che fanno uso del linguaggio: ci sono due capitoli dedicati a XAML, che poi viene approfondito per Windows 10. Ci sono anche un capitolo dedicato ad ASP.NET (WebForms e MVC), per creare applicazioni web, e uno dedicato alla creazione di applicazione distribuite.

La **quinta parte** è dedicata agli argomenti più avanzati, in molti casi fondamentali per lo sviluppo di un'applicazione: parliamo di sicurezza e accesso al file system, alla rete e a risorse esterne, oltre che della distribuzione delle applicazioni.

A chi si rivolge questo libro

L'idea che sta alla base di questo libro è di fornire un rapido accesso alle informazioni principali che caratterizzano la versione 8 di C#. Quando sono presenti, le novità rispetto alle versioni precedenti sono messe in risalto, ma questo libro è indicato anche per chi è a digiuno di C# e desidera imparare l'uso di questa tecnologia partendo da zero. Sono presentate alcune nozioni relative alla programmazione object oriented, ma è consigliabile avere già nozioni base di programmazione anche con altri linguaggi.

Inoltre, all'interno del libro la trattazione degli argomenti è svolta in maniera classica, non utilizzando un approccio per esempi (che sono comunque disponibili a supporto degli argomenti teorici).

Il libro è stato organizzato in modo da avere nella prima parte un'introduzione a .NET, allo scopo di fornire al lettore un'infarinatura generale dell'infrastruttura e della tecnologia su cui andare a costruire le applicazioni. Contiene quindi una trattazione completa del linguaggio ma non approfondisce in maniera specifica tutte le tecnologie incluse in .NET Framework e in .NET Core, soffermandosi solo su quelle principali.

Per comprendere appieno molti degli esempi e degli ambiti che incontreremo nel corso del libro, potrebbe essere necessario per il lettore approfondire maggiormente alcuni aspetti, che di volta in volta sono comunque evidenziati e per i quali forniamo opportuni link a risorse online.

Convenzioni

All'interno di questo volume abbiamo utilizzato stili differenti secondo il significato del testo, così da rendere più netta la distinzione tra tipologie di contenuti differenti.

I termini importanti sono spesso indicati in **grassetto**, così da essere più facilmente riconoscibili.

Il testo contenuto nelle note è scritto in questo formato. Le note contengono informazioni aggiuntive relativamente a un argomento o ad aspetti particolari ai quali vogliamo dare una certa rilevanza.

Gli esempi contenenti codice o markup sono rappresentati secondo lo schema riportato di seguito. Ciascun esempio è numerato in modo da poter essere referenziato più facilmente nel testo e recuperato dagli esempi a corredo.

Esempio 1.1 - Linguaggio

```
Codice
Codice importante, su cui si vuole porre l'accento
Altro codice
```

Per namespace, classi, proprietà, metodi ed eventi è utilizzato questo stile. Per attirare la vostra attenzione su uno di questi elementi, per esempio perché è la prima volta che viene menzionato, lo stile che troverete sarà **questo**.

Materiale di supporto ed esempi

Allegata a questo libro è presente una nutrita quantità di esempi, che riprendono sia gli argomenti trattati sia quelli non approfonditi. Il codice può essere scaricato agli indirizzi www.hoeplieditore.it/8465-4 e <http://books.asptalia.com/CSharp-8/>, dove saranno anche disponibili gli aggiornamenti e tutto il materiale collegato al libro.

Requisiti software per gli esempi

Questo è un libro dedicato a C#, con un particolare riferimento al linguaggio in quanto tale. Pertanto non è necessario nient'altro che il .NET Core SDK o il .NET Framework SDK, a seconda del tipo di applicazione che si vuole sviluppare. Il libro è stato scritto con la Preview 6 di .NET Core 3, ma tutti gli esempi e le tematiche trattate dovrebbero essere valide anche dopo il rilascio della versione finale. Rimandiamo ai siti prima menzionati per eventuali aggiornamenti.

Ove si eccettuino pochi casi particolari, comunque evidenziati, per visionare e testare gli esempi è possibile utilizzare una qualsiasi versione di Visual Studio 2019 (o successivo) (per Windows o macOS), oppure Visual Studio Code (su Windows, Linux e macOS). Visual Studio Code è scaricabile gratuitamente senza limitazioni particolari e utilizzabile liberamente, anche per sviluppare applicazioni a fini commerciali, all'indirizzo <http://code.visualstudio.com/>. Visual Studio Code e .NET Core sono realmente cross-platform e la maggior parte dei contenuti di questo libro (con l'eccezione di quelli pensati per lo sviluppo di interfacce native per Windows, basate su XAML) consente di creare facilmente applicazioni cross-platform basate su C#.

Per quanto concerne l'accesso ai dati, nel libro facciamo riferimento principalmente a SQL Server. Vi raccomandiamo di utilizzare la versione Express di SQL Server, liberamente scaricabile all'indirizzo <http://www.microsoft.com/express/sql/>. Il tool per gestire questa versione si chiama SQL Server Management Tool Express, ed è disponibile allo stesso indirizzo.

Contatti con l'editore

Per qualsiasi necessità, potete contattare direttamente l'editore attraverso il sito

<http://www.hoepli.it/>

Contatti, domande agli autori

Per rendere più agevole il contatto con gli autori, abbiamo predisposto un forum specifico, raggiungibile all'indirizzo <http://forum.aspitalia.com/>, in cui saremo a vostra disposizione per chiarimenti, approfondimenti e domande legate al libro.

Potete partecipare, previa registrazione gratuita, alla community di ASPItalia.com Network.

Vi aspettiamo!

Gli autori



Daniele Bochicchio

E-mail: daniele@aspitalia.com

Twitter: <http://twitter.com/dbochicchio>

Daniele Bochicchio è Chief Digital Officer in iCubed, uno dei più importanti Microsoft Gold Partner, con sede a Milano (<http://www.icubed.it>). Aiuta i clienti a costruire applicazioni innovative

basate sulle tecnologie web, mobile e cloud ed è responsabile della divisione training, grazie alla sua esperienza di oltre vent'anni nel settore.

È Microsoft Regional Director, un ruolo di advisor indipendente riservato a un centinaio di persone nel mondo, che aiutano Microsoft a percepire meglio il mercato. È inoltre Microsoft MVP dal 2002, a riconoscimento del suo impegno nelle community.

Nel 1998 ha ideato e sviluppato ASPIItalia.com, una delle storiche community italiane di sviluppatori, di cui coordina ancora le attività e che conta 70 mila utenti. Pubblica codice con licenza Open Source dal 1998, aiutando gli sviluppatori a far girare più facilmente le conoscenze.

Ha all'attivo più di 30 libri su tematiche tecniche e potete incontrarlo abitualmente ai più importanti eventi e conferenze tecniche italiane e internazionali.



Cristian Civera

E-mail: cristian@aspitalia.com

Twitter: <http://twitter.com/CristianCivera>

Cristian è Senior Software Architect e opera nello sviluppo di applicazioni web. Le sue competenze si basano sull'intero .NET, di cui si è sempre interessato fin dalla prima versione, sulla piattaforma cloud

Microsoft Azure e sui linguaggi e framework client, come Angular e TypeScript. Numerose le sue esperienze lavorative che spaziano dalla domotica, passando dai gestionali fino ad arrivare all'automazione industriale. Contribuisce alla community di ASPIItalia.com ed è Microsoft Azure MVP dal 2004. Partecipa regolarmente a diversi eventi, anche per Microsoft Italia, in qualità di speaker.



Marco Leoncini

E-mail: nostromo@aspitalia.com

Marco Leoncini si occupa di User eXperience e sviluppo di applicazioni web, mobile e desktop, in ambienti ASP.NET e WPF,.

È autore di libri, speaker, content manager di WinRTItalia.com.



Stefano Mostarda

E-mail: sm@aspitalia.com

Twitter: <http://twitter.com/sm15455>

Stefano è Software Architect e si occupa di progettazione e sviluppo di applicazioni principalmente Web. Sin dagli inizi della sua carriera ha continuato a seguire le innovazioni tecnologiche aiutando le aziende, in diversi settori chiave come domotica, media e gaming, nel loro processo di automazione.

Dal 2004 è membro dello staff del network ASPIItalia.com e Content Manager dei siti LINQItalia.com e HTML5Italia.com. È MVP nella categoria Visual Studio and Development Technologies ed è autore di diversi libri di questa collana e di libri in lingua inglese, sempre dedicati allo sviluppo .NET, oltre che speaker nelle maggiori conferenze italiane.



Matteo Tumiatì

E-mail: matteot@aspitalia.com

Twitter: <http://twitter.com/xtumiox>

Matteo è Senior Consultant in iCubed nella divisione AppDev e nel suo lavoro di consulenza si occupa di training e sviluppo software, con un focus principale su tecnologie Microsoft come .NET. Aiuta i suoi clienti nella migrazione e nella costruzione di piattaforme cloud-ready con Microsoft Azure e Kubernetes e sfruttando metodologie alla base di DevOps. Dal 2016 è membro dello staff di ASPIItalia.com e Content Manager del sito WinRTItalia.com.

È inoltre Microsoft MVP nella categoria Windows Development dal 2015 grazie ai numerosi contributi per la community e partecipazioni come speaker presso le principali conferenze nazionali.

ASPItalia.com

ASPItalia.com Network, nata dalla passione dello staff per la tecnologia, è supportata da quasi 20 anni di esperienza con ASPItalia.com per garantirvi lo stesso livello di approfondimento, aggiornamento e qualità dei contenuti su tutte le tecnologie di sviluppo del mondo Microsoft. Con 70.000 iscritti alla community, i forum rappresentano il miglior luogo in cui porre le vostre domande riguardanti tutti gli argomenti trattati!

ASPItalia.com si occupa principalmente di tecnologie dedicate al Web, da ASP.NET a IIS, con un'aggiornata e nutrita serie di contenuti pubblicati negli oltre vent'anni di attività che spaziano da ASP a Windows Server, passando per security e XML.

Il network comprende:

- ❑ HTML5Italia.com con HTML5, CSS3, ECMAScript 5 e tutto quello che ruota intorno agli standard web per costruire applicazioni che sfruttino al massimo il client e le specifiche web.
- ❑ LINQItalia.com, con le sue pubblicazioni, approfondisce tutti gli aspetti di LINQ, passando per i vari flavour LINQ to SQL, LINQ to Objects, LINQ to XML oltre a Entity Framework.
- ❑ WindowsAzureItalia.com, che tratta di cloud computing e di Microsoft Azure, in particolare dal punto di vista dello sviluppo.

- ❑ WinFXItalia.com, in cui sono presenti contenuti su Windows Presentation Foundation, Windows Communication Foundation, Windows Workflow Foundation e, più in generale, su tutte le tecnologie legate allo sviluppo per .NET.
- ❑ WinRTItalia.com copre gli aspetti legati alla creazione di applicazioni per Windows 10, in tutte le sue declinazioni, dall'UX fino allo sviluppo.

Ringraziamenti

Daniele ringrazia Noemi, Alessio e Matteo e tutta la sua famiglia, il team editoriale Hoepli e i coautori, con i quali la collaborazione è sempre più ricca di soddisfazioni. Un ringraziamento va anche ai suoi soci e colleghi in iCubed, che gli lasciano il tempo di dedicarsi a queste attività.

Cristian ringrazia Chiara, Sonia e Arianna. La passione per la tecnologia e la famiglia richiedono tempo e fortunatamente riesce a conciliare entrambe le cose.

Marco ringrazia il piccolo Stefano e Manuela, i genitori Mario e Fosca, le sorelle Monica e Sandra e gli amici e autori di questo libro.

Stefano ringrazia tutta la sua famiglia, i suoi amici. Grazie in maniera speciale agli altri autori per aver preso parte a quest'avventura. Aho!

Matteo desidera ringraziare la sua famiglia e i suoi amici, per il supporto e la pazienza, gli altri autori, lo staff di ASPIItalia.com e in particolare Daniele per la stima e le numerose opportunità.

Il team tiene particolarmente a ringraziare la community di ASPIItalia.com Network, a cui anche quest'ultimo libro è, come sempre, idealmente dedicato!

Introduzione a .NET

Dalla prima versione del .NET Framework, rilasciata nel 2002, le tecnologie hanno subito un'evoluzione che le ha portate a includere funzionalità sempre più innovative e a introdurre costanti miglioramenti a quanto già presente fin dall'inizio.

C#, come linguaggio, nasce insieme al .NET Framework, con l'obiettivo ambizioso di dare la possibilità a una nuova generazione di sviluppatori di sfruttare un linguaggio nuovo, pensato per le applicazioni moderne e che non avesse vincoli con il passato.

Questa evoluzione, nel corso degli anni, è stata caratterizzata dal rilascio di diverse versioni del .NET Framework, a cui si sono aggiunti miglioramenti anche dal punto di vista del linguaggio. La release 2.0, rilasciata nel novembre del 2005 insieme a Visual Studio 2005, ha senz'altro rappresentato un evento importante e un vero punto di svolta, introducendo novità significative e numerosi cambiamenti alle funzionalità già esistenti nelle versioni 1.0 e 1.1. Successivamente, la release 3.0, uscita tra il 2006 e il 2007 insieme a Windows Vista, e la versione 3.5, rilasciata ufficialmente nel corso del 2008 con Visual Studio 2008, hanno rappresentato un ulteriore passo in avanti, introducendo a loro volta nuove tecnologie finalizzate a migliorare la produttività degli sviluppatori nella realizzazione di applicazioni basate su servizi e ad alto impatto estetico. L'obiettivo primario della versione 4.5 del .NET Framework e del relativo Visual Studio 2012 è stato quello di continuare la tradizione di .NET Framework 4 e Visual Studio 2010, ovvero quella di semplificare e potenziare, allo stesso tempo, lo sviluppo di applicazioni di tutti i tipi. Visual Studio 2013 ha continuato a migliorare le funzionalità offerte dalla versione 2012, senza l'introduzione di una versione major del framework.

In questi anni il linguaggio ha subito numerosi cambiamenti, per lo più legati alle novità introdotte nella sintassi, per supportare al meglio la comparsa nelle versioni più recenti di nuove tecnologie come LINQ, AJAX, WCF o il cloud computing. Visual Studio 2015 ha continuato questa tradizione, con l'introduzione del supporto a nuovi linguaggi, nuovi paradigmi e un nuovo approccio alla distribuzione del .NET Framework.

Visual Studio 2017, invece, ha segnato un nuovo approccio al rilascio di Visual Studio, con update più frequenti e la comparsa di .NET Core, che ha rivoluzionato (e continuerà a farlo) il modo di ragionare intorno a .NET.

Oggi siamo arrivati a Visual Studio 2019, che continua sulle innovazioni introdotte da Visual Studio 2017.

Questo libro è focalizzato su tutto quanto C#, come linguaggio, consente di fare sfruttando .NET. Nel corso dei prossimi capitoli impareremo la sintassi di base e costruiremo applicazioni di tutti i tipi. Prima di partire, però, è necessario analizzare quello che .NET è in grado di offrirci.

Cos'è .NET

In principio C# è stato sempre legato alla piattaforma .NET Framework: è sempre stato importante conoscerne i dettagli di funzionamento, per poter sfruttare il linguaggio.

Da qualche anno a questa parte, però, non si parla più di .NET Framework, ma genericamente di .NET e il motivo è che, a fianco a .NET Framework, è stato introdotto un nuovo framework, chiamato .NET Core. Si tratta di una piattaforma **open source**, di un reboot totale della piattaforma di sviluppo, pensata per il futuro. .NET Framework risale ai primi anni 2000 e il mercato dello sviluppo era decisamente differente all'epoca: la maggioranza degli sviluppatori, infatti, sviluppava applicazioni native per Windows su Windows. Oggi la bilancia è sicuramente spostata da un'altra parte: Linux è un sistema operativo sempre più apprezzato in ambito server e il cloud (da non confondersi con il cloud pubblico di Amazon AWS o Microsoft Azure) è uno dei fattori principali di sviluppo.

La release più recente del .NET Framework, a cui questo libro fa riferimento, è la versione 4.8, che include **C# 8**, mentre la release più recente di .NET Core è la 3.

Essendo il linguaggio attraverso il quale sviluppare applicazioni, C# consente di fare tutto quello che le due versioni di .NET mettono a

disposizione.

Grazie al fatto che le specifiche di .NET sono **standard ISO**, si possono creare alternative al .NET Framework (come è successo con .NET Core o Xamarin), per cui oggi imparare a sfruttarne i vantaggi rappresenta un ottimo modo di investire il proprio tempo, anche in proiezione futura. Questo concetto è certamente valido anche per C#, le cui specifiche sono disponibili per implementazioni esterne alla piattaforma Windows. Per consentire a implementazioni differenti di .NET di referenziare le stesse librerie, Microsoft ha introdotto .NET Standard, su cui torneremo a breve in questo stesso capitolo.

.NET = .NET Framework + .NET Core

.NET Core e .NET Framework possiedono molti aspetti in comune: sono l'uno l'evoluzione in chiave moderna dell'altro. Il .NET Framework è confinato solo a Windows e contiene tutto ciò a cui siamo abituati tradizionalmente. .NET Core, invece, è una nuova versione, pensata soprattutto per il cloud e in grado di funzionare, oltre che su Windows, anche su Mac OSX e Linux. Rappresenta un radicale cambio di strategia da parte di Microsoft, che per la prima volta supporta nativamente il proprio Runtime su piattaforme differenti rispetto al classico Windows.

Microsoft era di fronte a una scelta: provare a rendere .NET Framework cross-platform, con il rischio di rompere la compatibilità, oppure provare a evolvere i concetti alla base di .NET in una maniera nuova. Alla fine, ha prevalso questo orientamento: così è nato .NET Core.

Cos'è .NET Core? È una nuova implementazione dei concetti alla base di .NET, così come lo è .NET Framework. Se avete sviluppato con Silverlight, Xamarin o Universal Windows Platform, sapete che questi toolkit sono implementazioni di .NET, e sono creati per risolvere problemi specifici, in ambiti in cui il .NET Framework sarebbe stato eccessivo. Sono, insomma, delle versioni di .NET pensate per risolvere problematiche specifiche, mentre .NET Framework è un framework più completo, frutto dell'insieme di alcuni toolkit (come ASP.NET, WPF, WinForms).

Volendo tirare le somme, **.NET Core è solo l'ultima delle varianti di .NET**, creata per consentire di sviluppare applicazioni cross-platform per il web, basandosi sui concetti tipici di .NET Framework e di ASP.NET, nella sua variante chiamata ASP.NET Core.

.NET Core è un reboot di .NET. .NET Framework, infatti, non consente di re-ingegnerizzare tutti i pezzi di cui è composto per garantirne una portabilità cross-platform. Piuttosto che creare versioni di .NET Framework incompatibili tra loro, Microsoft ha preferito differenziarle totalmente, a partire dal nome stesso.

.NET Core, prima della versione 3, è stato limitato al solo supporto di ASP.NET Core, con la possibilità di creare un numero decisamente inferiore di tipologie di applicazioni rispetto a .NET Framework.

Questa limitazione è stata voluta: .NET Core è pensato per le performance e per essere **cross platform**, in un'ottica cloud based, quindi principalmente web. Questo, però, a partire da .NET Core 3 è cambiato con l'aggiunta di una serie di estensioni (valide solo per Windows). Queste estensioni, chiamate **Desktop Pack**, aggiungono il supporto a WinForms, WPF e WinUI, che sono diventate open source. L'obiettivo di Microsoft è quello di consentire l'utilizzo di .NET Core a una platea sempre più ampia, senza rinunciare alle innovazioni, che vengono rilasciate all'interno di questa piattaforma. Poiché .NET Framework è installato su milioni di server e client, Microsoft non ha in programma di creare sconvolgimenti particolari nei prossimi anni. .NET Framework è una tecnologia stabile e matura e se il vostro interesse è avere applicazioni che non funzionano senza intervenire sulle stesse, allora siete già nella situazione giusta. Tuttavia, se preferite essere più moderni e utilizzare un framework che è rilasciato più velocemente, allora la risposta giusta è .NET Core. Nella creazione di nuove applicazioni, infatti, non c'è dubbio: .NET Core è sempre da preferire.

Se siete sviluppatori .NET di lungo corso, non dovete preoccuparvi: .NET Core e .NET Framework hanno moltissimo in comune. Tutto quello che presenteremo in questo libro, infatti, è pensato per entrambi: linguaggi, compilatori e una serie di funzionalità comuni. Grazie a questo approccio, diventa possibile sfruttare .NET Core conoscendo .NET Framework (e viceversa). Per questo motivo, nel corso del libro generalmente ci riferiremo a .NET e, eventualmente, sottolineeremo eventuali specificità di .NET Core o .NET Framework, anche in previsione di .NET 5, di cui parleremo tra un attimo in questo capitolo.

.NET Core 3 non è ancora stato rilasciato in versione definitiva nel momento in cui questo libro viene pubblicato ed è disponibile come preview. Il rilascio definitivo avverrà a settembre 2019. La maggior parte dei contenuti di questo libro sono comunque da considerarsi definitivi.

In realtà, .NET Core e .NET Framework consentono di sfruttare le stesse librerie, a patto che queste siano state create come **librerie .NET Standard**. Questo tipo di librerie è stato appositamente pensato per rendere compatibili tra loro librerie create per differenti runtime. ASP.NET Core, quindi, è di fatto implementato come una serie di librerie .NET Standard.

A meno che non dobbiate per forza utilizzare una libreria non compatibile, consigliamo di utilizzare .NET Framework come runtime e di sfruttare .NET Core, così da poter beneficiare di tutte le funzionalità introdotte da quest'ultimo, come il supporto cross-plataform e un runtime più snello e perciò più performante.

Diamo un'occhiata a .NET Standard per cercare di capire meglio come funziona.

.NET Standard

.NET Standard è una specifica che definisce formalmente come le API devono essere implementate all'interno di tutti i runtime .NET ed è stato creato per consentire a differenti runtime di lavorare all'interno di un solo ecosistema.

.NET è basato sullo **standard ECMA 335**, che però non prevede come debbano comportarsi internamente le librerie della BCL (Base Class Library, l'insieme di librerie di base di .NET).

Per questo motivo, .NET Standard cerca di porre una serie di regole che consentano, a prescindere dall'implementazione del runtime, di avere accesso a un set di API comune. Questo consente agli sviluppatori di creare librerie che siano più facilmente portabili su runtime differenti, senza dover ricompilare codice o creare differenti versioni compilate di uno stesso output. Le librerie .NET Standard, insomma, sono compatibili in maniera binaria tra tutte le implementazioni che supportano lo standard.

Le diverse versioni di .NET implementano una versione differente di .NET Standard. Ogni versione di .NET Standard porta con sé una serie di API comuni, che se un determinato runtime supporta, ci garantisce di avere codice che funziona in maniera simile.

I runtime che supportano direttamente .NET Standard sono differenti e spaziano da .NET Framework e .NET Core, per arrivare a Mono (una implementazione Open Source di .NET, alla cui base si trovano alcuni prodotti legati a Xamarin), ma anche Xamarin e Universal Windows Platform (UWP).

Al momento di scrivere questo libro, la versione corrente di .NET Standard è la 2.1, che è supportata da .NET Core 3, ma non da .NET Framework 4.8, che è rimasto alla versione 2. Questa scelta è stata fatta perché le innovazioni introdotte da .NET Standard 2.1 richiedono cambiamenti al runtime, che si è deciso di non portare su .NET Framework 4.8, per non sacrificarne la stabilità. .NET Standard 2.1 sarà anche supportato da Xamarin, Mono e Unity.

Creando una libreria basata su .NET Standard 2.0, saremo in grado di farla funzionare su .NET Core 2 o successivo, .NET Framework 4.6.1 o successivo e UWP 10.0.16299 o successivo. Viceversa, scegliendo .NET Standard 2.1, non saremo in grado di referenziare questa libreria anche da .NET Framework 4.8 o precedenti.

Le versioni di .NET Standard differiscono dalle altre solo per le aggiunte: una volta che una API è confluita in .NET Standard, infatti, questa non verrà mai più rimossa. Inoltre, le API non vengono modificate.

La gestione di .NET Standard è demandata a un comitato, che decide cosa debba finire all'interno dello standard e cosa no, garantendo una governance dell'intero ecosistema .NET.

Rispetto alle **Portable Class Library**, che .NET Standard sostituisce, i vantaggi sono tanti. Entrambi consentono di definire un set di API portabili tra runtime differenti, ma .NET Standard ha il vantaggio di non richiedere direttive di compilazione e di essere, come il nome suggerisce, uno standard curato. Come gran parte di ciò che ricade intorno a .NET Core, le specifiche sono open source e disponibili su <https://aspit.co/bo5>.

Tecnicamente, una libreria .NET Standard referencia un metapackage chiamato `NETStandard.Library`, che è in realtà composto da una serie di package NuGet. Al loro interno, questi pacchetti hanno come target la specifica versione del framework a cui fanno riferimento.

I metapackage sono una convenzione di NuGet per raggruppare logicamente una serie di package. Come vedremo, anche ASP.NET Core è distribuito come metapackage, ma resta possibile referenziare direttamente, se necessario, anche i singoli package.

Creando una libreria di questo tipo, faremo riferimento a una piattaforma *ad hoc*, chiamata `netstandard`. Ognuna delle versioni è caratterizzata da un nome che ne ricorda la versione. Un esempio di creazione di una libreria di questo tipo è contenuto nel prossimo capitolo. Quando creiamo una libreria per .NET Core, possiamo decidere di crearla sia con .NET Standard, per compatibilità

con .NET Framework, sia direttamente per .NET Core, indicando quello che viene chiamato *target framework*.

I componenti di .NET

.NET Framework e .NET Core sono composti da una serie di funzionalità che si incastrano in maniera perfetta all'interno di un immaginario puzzle, per rendere lo sviluppo più semplice, a prescindere dalla tipologia di applicazione sviluppata.

La release 4.8 di .NET Framework è evolutiva. Si basa, infatti, sulla versione 4.5, di cui è un aggiornamento. Installando la versione 4.8 possiamo godere di tutti i benefici introdotti dalla versione 4.5, potendo contare anche sulle novità che andremo a presentare nel corso di questa guida. Di fatto, installando il .NET Framework 4.8 viene installato anche il Service Pack del .NET Framework 4.5 (e delle versioni tra queste contenute), che resta la base su cui si poggiano le funzionalità.

Uno degli obiettivi di .NET è infatti quello di unificare le metodologie di progettazione e sviluppo nell'ambito delle piattaforme Microsoft, offrendo uno strato comune sul quale gli sviluppatori possano basare le proprie applicazioni, indipendentemente dalla loro tipologia.

I concetti esposti in questo capitolo (come in quelli immediatamente successivi) rimangono del tutto validi in presenza di applicazioni di tipo differente. Questo aspetto rappresenta un vero vantaggio anche in termini produttivi per lo sviluppatore, dal momento che può imparare a sviluppare applicazioni .NET in modo più semplice e immediato, utilizzando lo stesso linguaggio, la stessa tecnologia e il medesimo approccio per scopi differenti.

Indipendentemente dalla versione, .NET presenta una serie di caratteristiche e componenti che, nel tempo, sono rimasti sostanzialmente invariati o quasi, ossia:

- ❑ **Common Language Runtime (CLR):** è il cuore della tecnologia, la parte responsabile di gestire l'esecuzione delle applicazioni;
- ❑ **Common Language Specification (CLS):** è un insieme di specifiche che rendono possibile l'interoperabilità tra linguaggi differenti;
- ❑ **Common Type System (CTS):** rappresenta delle specifiche per un insieme comune di tipi scritti in linguaggi differenti, che rende possibile

lo scambio d'informazioni tra le applicazioni .NET, fornendo un meccanismo di rappresentazione dei dati condiviso dal runtime;

- ❑ **linguaggi:** è ciò che consente di sviluppare le applicazioni, cioè principalmente **C# 8** e **Visual Basic 16**;
- ❑ **un insieme di tecnologie diverse:** come ASP.NET, WinForms, Windows Presentation Foundation, Windows Runtime e ADO.NET, in generale contenuti all'interno di componenti (concetto su cui avremo modo di tornare a breve).

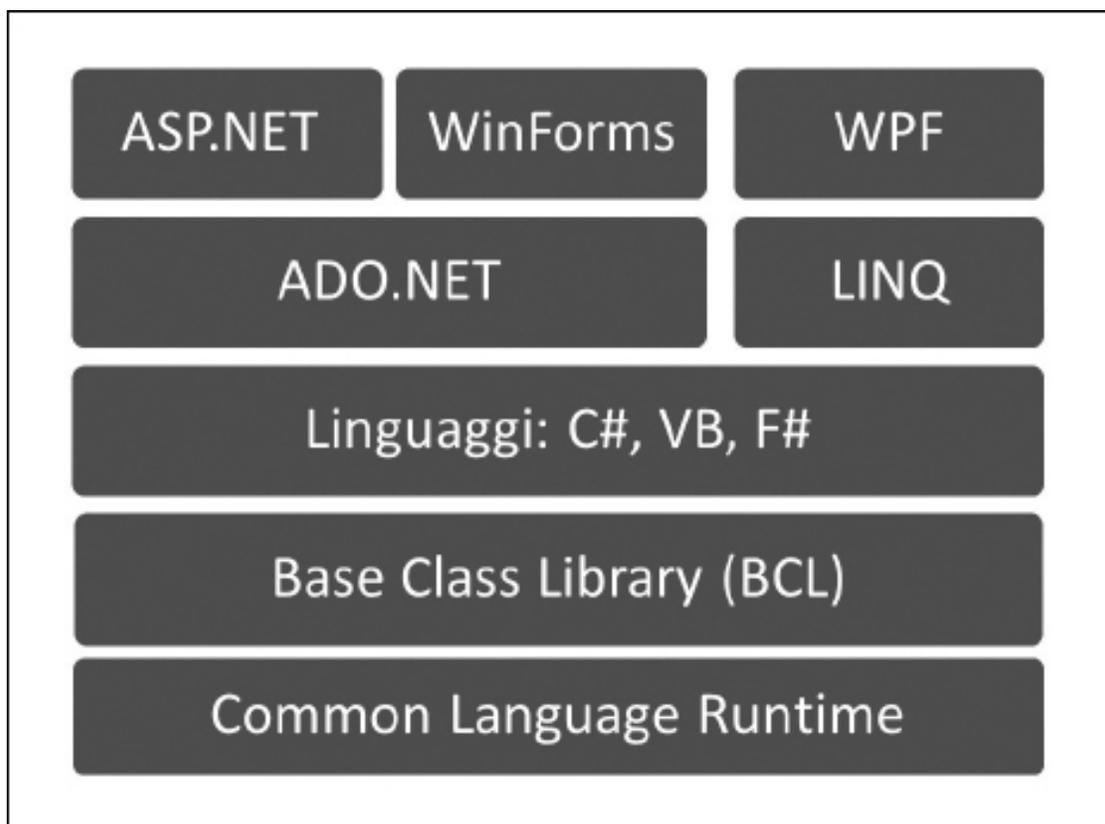


Figura 1.1 – Le tecnologie alla base di .NET.

Questa caratteristica comporta indubbiamente vantaggi nella creazione del software: in particolare, diventa molto più semplice passare dallo sviluppo web a quello per Windows (o viceversa), visto che .NET offre le medesime funzionalità a prescindere “dall’interfaccia grafica” che si utilizza, oppure riutilizzare le medesime componenti per tipologie eterogenee di applicazioni.

I vantaggi non finiscono per farsi sentire solo nella fase di sviluppo, ma sono importanti anche in quella di progettazione. Se quest'aspetto è spesso sottovalutato, esso riveste invece un'importanza straordinaria quando abbiamo a che fare con progetti complessi.

Il percorso di apprendimento iniziale, per tutti questi motivi, è praticamente lo stesso, a prescindere dalla tipologia di applicazione che andremo a creare, per poi differenziarsi per quanto riguarda le problematiche specifiche di ogni ambiente, dato che un'applicazione web ha peculiarità differenti da quelle di un'applicazione Windows.

All'interno di un ambiente del genere, anche il linguaggio rappresenta (quasi) un dettaglio. Di fatto, ciascun linguaggio supportato da .NET (nel nostro caso C#, ma il concetto è identico anche per Visual Basic) accede alle stesse funzionalità offerte dalla piattaforma applicativa, utilizzando semplicemente notazioni sintattiche diverse, ognuna dotata di proprie motivazioni storiche e tecniche.

Una classe non è nient'altro che un insieme di funzionalità offerte allo sviluppatore, tali da non dover essere implementate ogni volta da zero, ma disponibili per essere riutilizzate in fase di sviluppo laddove risulti necessario.

Nel corso della guida, impareremo a utilizzare le classi in maniera del tutto naturale, tanto che questi termini, alla fine, sembreranno scontati.

I due componenti utilizzati maggiormente sono il Common Language Runtime (CLR), che sarà approfondito nella prossima sezione di questo capitolo, e la Base Class Library (BCL) che, come il nome stesso suggerisce, contiene alcuni tipi di base comuni a tutto l'ambiente.

Common Language Runtime (CLR)

Ogni linguaggio di programmazione si basa su un componente denominato "runtime" che, a seconda delle diverse situazioni, può essere più semplice, come nel caso di C++, dove la sua funzione è quella di fornire un'infrastruttura comune all'esecuzione del codice macchina, o più complesso, come nel caso di VB6, dove il runtime esegue il codice compilato in p-code, o di Java, in cui la virtual machine si occupa di servizi analoghi a quelli offerti da .NET. Il Common Language Runtime (CLR) rappresenta il runtime per .NET.

Il ruolo del Common Language Runtime è semplicemente quello di eseguire tutte le applicazioni .NET scritte in uno dei linguaggi supportati dalla piattaforma (si parla in tal caso di **linguaggi managed** o gestiti) e di fornire un insieme di servizi affinché le applicazioni possano sfruttare e condividere una serie di funzionalità tali da rendere l'ambiente di esecuzione più stabile ed efficiente.

Nel caso specifico, durante la fase di esecuzione, il CLR provvede a caricare ed eseguire il codice, offrendo caratteristiche quali la gestione della memoria, l'allocazione di thread e, più in generale, politiche di accesso alle risorse del computer.

Per poter eseguire un pezzo di codice scritto in C# o in uno qualsiasi dei linguaggi supportati dal CLR, occorre utilizzare un compilatore specifico che non produce il classico codice macchina ma una forma ibrida e parziale, denominata MSIL (Microsoft Intermediate Language) o CIL (Common Intermediate Language), più comunemente detta per semplicità IL (Intermediate Language). Si tratta di un insieme di istruzioni indipendenti dal linguaggio e dall'architettura dell'hardware su cui esso verrà poi eseguito. All'atto dell'esecuzione, queste istruzioni sono gestite da un JIT-ter, cioè da un compilatore Just-In-Time che ha il compito di trasformarle finalmente in codice macchina, ottimizzandole per il tipo di hardware in uso e fornendo al tempo stesso funzionalità di gestione dell'accesso alle risorse di sistema.

Il meccanismo descritto è possibile perché il codice, per essere eseguito, deve essere convertito in qualcosa che l'architettura per la quale viene compilato è in grado di comprendere, cioè codice macchina. L'IL per questo motivo è solo un linguaggio intermedio, che necessita di una conversione per essere eseguito.

Esistono JIT-ter specifici per ognuna delle piattaforme hardware supportate. Al momento, Microsoft fornisce implementazioni del .NET per ambienti a **32 bit** (x86) e a **64 bit** (x64), con il JIT compiler introdotto da alcuni anni, chiamato RyuJIT. Come abbiamo detto, .NET Framework è disponibile solo per Windows, mentre .NET Core è disponibile anche per macOS e Linux, e nelle varianti per processori ARM32 e ARM64.

Questo nuovo compilatore nasce all'evolversi della tecnologia cui abbiamo assistito negli ultimi anni: in un mondo in cui si va sempre di più verso i 64 bit, RyuJIT va a sopperire a una mancanza storica del .NET Framework, in cui il compilatore era più veloce per 32 bit che per 64 bit, garantendo, a parità di codice, performance superiori anche di 20 volte e un tempo di startup più rapido delle applicazioni. Possiamo scaricare gratuitamente l'ultima versione

del .NET Framework e di .NET Core per la nostra piattaforma da questo URL: <http://www.dot.net/>.

Sia .NET Framework sia .NET Core vengono distribuiti in due versioni, una chiamata **Redistributable**, perché contiene solo il necessario per far funzionare le applicazioni (si installa tipicamente sui server di testing e produzione) e una che, invece, contiene anche il **Software Development Kit (SDK)**, composto da un insieme di strumenti utili per lo sviluppo e dalla documentazione di supporto. Quest'ultima è quella di cui facciamo uso in fase di sviluppo.

Quando il compilatore produce IL, aggiunge una serie di metadati, che servono per descrivere il contenuto dell'entità risultante dalla compilazione. I metadati rappresentano informazioni aggiuntive che sono allegate al codice IL per descriverne meglio il contenuto.

La presenza di queste informazioni aggiuntive consente al codice di auto-descriversi, cioè di essere in grado di funzionare senza librerie di tipi o linguaggi come IDL (Interface Definition Language).

I metadati del CLR sono contenuti all'interno del file che viene generato in seguito alla compilazione del sorgente, scritto in C#. Il contenuto del file, che è chiamato assembly, è strettamente legato all'IL prodotto e consente al CLR di caricare ed estrarre i metadati in fase di esecuzione, quando sono necessari. In questo modo possono essere ricavati: la definizione di ogni tipo contenuto, la firma di ogni membro, eventuali riferimenti a librerie esterne e ad altre informazioni, necessarie a runtime affinché il codice venga eseguito al meglio.

Internamente, il Common Language Runtime è composto di due elementi principali:

- ❑ Un runtime: **MSCOREE (per .NET Framework) o CoreCLR (per .NET Core)**;
- ❑ Un insieme di librerie di base: **MSCORLIB**.



Figura 1.2 – Il Common Language Runtime (CLR) in dettaglio.

MSCOREE è il componente all'interno del CLR (su .NET Framework) responsabile di compilare il codice IL nel codice macchina, secondo il meccanismo che abbiamo analizzato finora, ovvero tramite un JIT-ter specifico per ogni architettura hardware. È anche responsabile dell'allocazione e della deallocazione degli oggetti in memoria, oltre che della gestione della sicurezza e delle politiche di accesso alle risorse da parte del codice. In .NET Core questa funzionalità è demandata a un componente chiamato **CoreCLR**, che tecnicamente è implementato in `coreclr.dll` (in Windows), in `libcoreclr.so` (in Linux) o in `libcoreclr.dylib` (in macOS), ma i concetti qui esposti sono gli stessi.

MSCORLIB contiene le librerie di sistema che compongono il cosiddetto "core", il cuore di .NET, su cui si basa ogni applicazione gestita dal Common Language Runtime. È stato rinominato `System.Private.CoreLib` in .NET Core, anche se sia nel codice sia nella documentazione si continua a far genericamente riferimento a **MSCORLIB**.

Queste librerie diventano di facile utilizzo dopo un po' di pratica, dato che consentono di effettuare le operazioni più comuni. Avremo modo di usare la Base Class Library (BCL), di cui anche **MSCORLIB** fa parte, nel corso dei prossimi capitoli.

Il concetto di codice managed

Non di rado capiterà di imbatterci nel concetto di codice **managed** (spesso anche detto codice gestito). Con questo termine si indicano tutte quelle applicazioni eseguite tramite il CLR. Il codice managed beneficia di funzionalità come l'integrazione dei linguaggi, con la relativa gestione delle eccezioni, la sicurezza, il versioning e il deployment, favorendo l'interazione tra componenti scritti in linguaggi differenti, dal momento che, all'atto della compilazione, tutto il codice è comunque trasformato in IL. Il termine "managed" non è stato scelto a caso, dato che il Common Language Runtime prevede un meccanismo di sandboxing, cioè di inscatolamento, che fa sì che il codice sia eseguito isolato da altri contesti e con la possibilità per il CLR, attraverso MSCOREE/CoreCLR, di gestire anche le politiche di accesso alle risorse, la gestione della memoria e gli aspetti di sicurezza.

Viceversa, il codice che non viene eseguito dal Common Language Runtime viene comunemente chiamato **unmanaged** (non gestito). Un programma scritto in VB6 o C++ è un'applicazione unmanaged, per capirci meglio.

Quando scriviamo un'applicazione in uno dei linguaggi managed, come C#, in realtà, il processo di esecuzione che sta dietro è molto più complesso di quello che possiamo immaginare, benché venga tutto consumato nell'arco di poche frazioni di millesimi di secondo. Nella [Figura 1.3](#) viene schematizzato il funzionamento di questo processo.

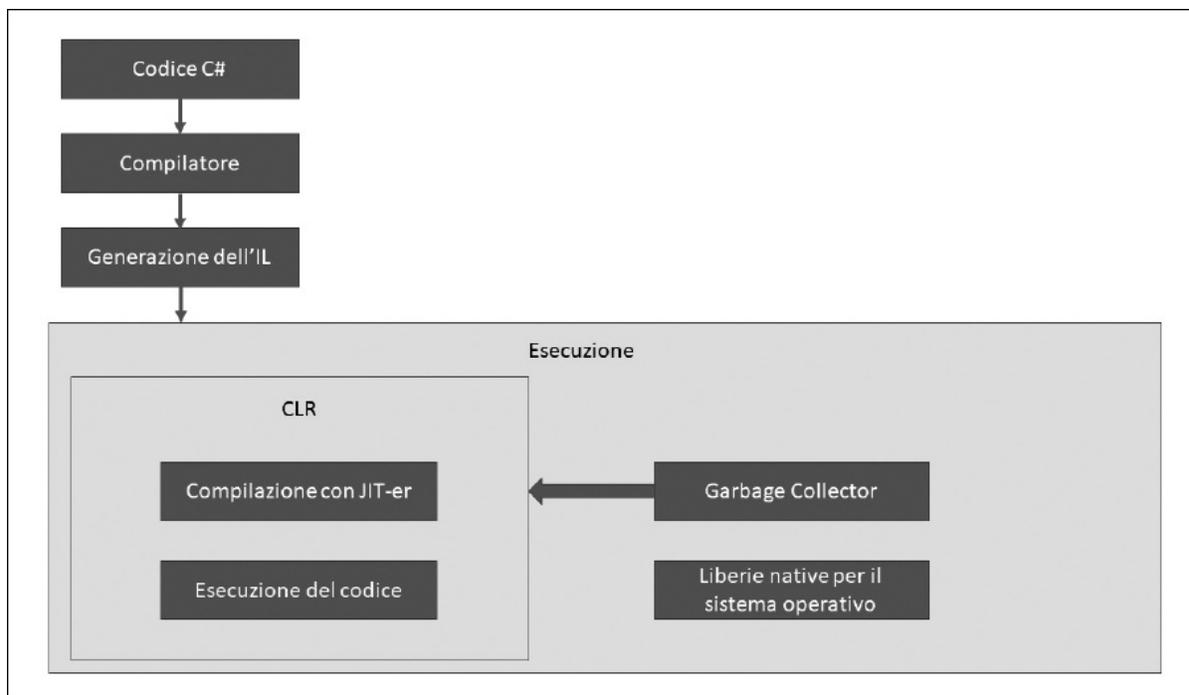


Figura 1.3 – Il processo di esecuzione di codice da parte del CLR.

Come abbiamo già detto, durante la prima fase di compilazione viene prodotto codice IL. All'atto dell'esecuzione il passo successivo da parte del CLR consiste dapprima nella compilazione del codice da MSIL in linguaggio macchina (operazione svolta dal JIT-ter) e, successivamente, nel controllo della congruenza dei tipi e nell'applicazione delle politiche di sicurezza. Infine, il CLR crea il contesto di esecuzione vero e proprio.

Il percorso compiuto dal Common Language Runtime, quando esegue codice managed, può essere schematizzato come segue:

- cerca i metadati associati al membro richiesto;
- scorre lo stack di esecuzione;
- gestisce le eventuali eccezioni sollevate;
- gestisce le informazioni relative alla sicurezza.

La serie di operazioni mostrate in [Figura 1.3](#) avviene solo alla prima richiesta mentre, per le volte successive, esiste un meccanismo che fa sì che questa fase venga saltata, andando a recuperare direttamente il risultato della

compilazione, in modo da evitare sprechi di risorse e ottenere performance di tutto rispetto. Questo vuol dire che una chiamata a un certo metodo di una classe fa sì che il JITter compili l'IL soltanto la prima volta: dalle successive richieste, verrà utilizzata la versione già compilata. .NET Core 3, in tal senso, aggiunge una nuova funzionalità, attiva di default da questa versione, chiamata **Tiered Compilation**: si tratta di un meccanismo che, attraverso un'ottimizzazione che si ripete nel tempo, continua a effettuare passaggi di codice, al fine di migliorarne le performance.

Per avere accesso ai servizi offerti dal CLR, i compilatori devono fare affidamento su una serie di regole che rendano possibile l'interazione tra componenti. Questa parte dell'architettura di .NET prende il nome di Common Type System.

Common Type System

Il Common Type System (CTS) rappresenta un'altra parte importante dell'architettura di .NET, dato che stabilisce come i tipi debbano essere dichiarati, utilizzati e gestiti dal Common Language Runtime; si tratta di un elemento di importanza fondamentale per garantire il supporto e l'integrazione multilinguaggio, poiché ogni linguaggio ha la propria struttura e le proprie regole, che si basano su convenzioni che spesso stanno agli antipodi le une rispetto alle altre.

Il Common Type System ha il compito di rendere possibile che un componente scritto in Visual Basic (che utilizza il tipo `Integer`) e uno in C# (che fa uso del tipo `int`) possano scambiarsi informazioni e ragionare sullo stesso identico concetto, che è appunto quello di numero intero. Questo è possibile poiché entrambi i tipi, `int` e `Integer`, sono convertiti in fase di compilazione nel tipo `System.Int32`, che è la rappresentazione del numero intero all'interno del .NET Framework.

Il Common Type System consente di scrivere codice senza dover far riferimento per forza al tipo del .NET Framework. Per denotare i tipi, lo sviluppatore può continuare a utilizzare le parole chiave specifiche del linguaggio in uso (`int` al posto di `System.Int32` in C#; `Integer` al posto di `System.Int32` in Visual Basic), dato che queste non sono altro che alias del tipo effettivo.

Il Common Type System garantisce inoltre che, in fase di compilazione, la tipologia iniziale venga preservata. Tutto questo è possibile grazie al fatto che

il compilatore C# è **CLS-compliant**, il che significa che genera codice compatibile con le specifiche CLS, che verranno trattate nella prossima sezione.

Common Language Specification

La Common Language Specification (CLS) rappresenta una serie di specifiche che il compilatore e il relativo linguaggio devono rispettare per fare in modo che un componente sia in grado di integrarsi con componenti scritti in linguaggi diversi. In pratica, stiamo parlando del sistema attraverso il quale il compilatore espone i tipi affinché il codice risultante possa essere eseguito dal Common Language Runtime.

Perché tutto ciò avvenga, i tipi e i metodi pubblici devono essere CLS-compliant, cioè compatibili con tali specifiche. Quelli privati, che non vengono esposti all'esterno, possono anche non esserlo, dal momento che solo i tipi e i metodi pubblici sono esposti direttamente al CLR.

Dato che il Common Language Runtime è in grado solo di eseguire IL, i compilatori devono essere in grado di produrre codice IL corretto, tale da permettere al CLR di far parlare componenti differenti, grazie all'uniformità garantita dalla CLS. È qui che entra in gioco il Common Type System, il quale consente la portabilità dei tipi, in un contesto dove il codice IL è generato a partire da altri linguaggi. L'esempio più semplice di quanto la CLS sia importante è rappresentato dal fatto che C#, come altri linguaggi, è “**case sensitive**”, cioè fa differenza tra le lettere maiuscole e minuscole nella nomenclatura e nelle parole chiave, laddove invece, in Visual Basic, questo non rappresenta una differenza. In un contesto del genere, la definizione in C# di una variabile di nome MiaVar ha un significato differente rispetto a Miavar, mentre in Visual Basic i due nomi identificano lo stesso oggetto. È proprio in casi come questo che le specifiche della CLS entrano in gioco poiché, proprio per questa potenziale ambiguità, esse vietano di esporre membri pubblici con lo stesso nome e “case” differente.

Se il codice è CLS-compliant, cioè rispetta tutte le regole previste dalla CLS, avremo la certezza che l'accesso ai membri pubblici possa essere fatto da un componente managed, anche se scritto in un linguaggio differente. D'altra parte, la BCL, che contiene tutte le classi di base del .NET Framework, è interamente scritta in C#, ma può essere tranquillamente utilizzata da applicazioni scritte in Visual Basic.

La CLS regola pertanto le modalità in cui i tipi devono essere esposti, strutturati ed organizzati. In prima approssimazione, possiamo identificare due gruppi principali di tipi: i cosiddetti **tipi primitivi** elencati nella [Tabella 1.1](#) (tra cui il tipo `System.Object`) e i **tipi derivati**, che ereditano dal tipo base `System.Object`. Tutti gli elementi definiti tramite un linguaggio managed sono, infatti, oggetti. Il Common Language Runtime *capisce* solo oggetti e, pertanto, .NET favorisce l'utilizzo di tecniche e linguaggi di programmazione chiamati **object oriented**, cioè orientati agli oggetti. Il concetto di ereditarietà e la programmazione orientata agli oggetti verranno trattati nel corso dei prossimi capitoli.

Tabella 1.1 – I tipi della CLS

Tipo	Descrizione
<code>System.Boolean</code>	Rappresenta un valore booleano (<code>True</code> o <code>False</code>). La CLS non prevede conversioni implicite da altri tipi primitivi, cioè da <code>boolean</code> a stringa o viceversa, senza un'operazione esplicita di conversione.
<code>System.Byte</code>	Rappresenta un tipo byte senza segno, cioè interi compresi tra 0 e 255.
<code>System.Char</code>	Rappresenta un carattere UNICODE.
<code>System.DateTime</code>	Rappresenta un tipo data e ora, in un intervallo compreso tra 01/01/01 e 31/12/9999 e 0:00:00 e 23:59:59.
<code>System.Decimal</code>	Rappresenta un tipo <code>Decimal</code> , con un numero massimo di ventotto cifre.
<code>System.Double</code>	Rappresenta un tipo numerico a 64 bit, a doppia precisione e in virgola mobile.
<code>System.Int16</code>	Rappresenta un intero a 16 bit con segno.
<code>System.Int32</code>	Rappresenta un intero a 32 bit con segno.
<code>System.Int64</code>	Rappresenta un intero a 64 bit con segno.
<code>System.Single</code>	Rappresenta un tipo numerico a 4 byte, a doppia precisione e in virgola mobile.
<code>System.TimeSpan</code>	Rappresenta un intervallo di tempo, anche negativo.
<code>System.String</code>	Rappresenta un insieme, anche vuoto, di caratteri UNICODE.
<code>System.Array</code>	Rappresenta un array (vettore) di oggetti monodimensionale.
<code>System.Object</code>	Il tipo base da cui tutti gli altri derivano.

[La Cross-Language Interoperability](#)

La Cross-Language Interoperability (CLI) è la possibilità del codice di interagire con altro codice scritto in un linguaggio differente. Questa caratteristica consente di scrivere meno codice, riutilizzando quello già presente e favorendo uno sviluppo non ripetitivo.

Il codice managed beneficia della CLI perché i debugger e i vari strumenti di sviluppo devono essere in grado di *capire* solamente l'IL e i suoi metadati, piuttosto che ognuno dei diversi linguaggi con le sue peculiarità. Inoltre, la gestione delle eccezioni viene trattata nello stesso modo per tutti i linguaggi, in modo che un errore possa essere intercettato da un componente scritto in un linguaggio diverso da quello che l'ha sollevato. Infine, poiché ogni linguaggio poggia su un modello di dati comune, diventa possibile, senza particolari vincoli, scambiare oggetti tra componenti scritti in linguaggi differenti.

L'integrazione tra i linguaggi è dovuta primariamente alla Common Language Specification, in base alla quale i compilatori generano il codice. La CLS garantisce, infatti, una base di regole condivise tali da permettere l'interoperabilità richiesta.

Tipi di valore e tipi di riferimento

A questo punto è ormai chiaro che i tipi sono alla base del CLR, poiché sono il meccanismo attraverso il quale lo sviluppatore rappresenta nel codice le funzionalità derivanti dall'ambito logico e pratico dell'applicazione. Un tipo serve per descrivere il ruolo di una variabile nell'ambito del codice e ne caratterizza le funzionalità.

All'interno del Common Language Runtime troviamo il supporto per due categorie fondamentali di tipi:

- ❑ **tipi di valore:** sono rappresentati dalla maggior parte dei tipi primitivi (come `System.Int32`, `System.Boolean`, `System.Char`, `System.DateTime`, ecc.), dalle enumerazioni o da tipi definiti dallo sviluppatore come **strutture**;
- ❑ **tipi di riferimento:** sono rappresentati dalle **classi**. Il loro scopo è di fornire un meccanismo di strutturazione del codice e memorizzazione dei dati in un'ottica object oriented.

La differenza tra tipi di valore e tipi di riferimento è che i primi contengono direttamente il valore dei dati, mentre i secondi contengono solo un

riferimento a una locazione in memoria (sono, in pratica, un puntatore a una regione di memoria). I tipi di valore non devono essere istanziati esplicitamente tramite un'azione di creazione e non possono contenere un valore nullo. Per i tipi di riferimento vale, in entrambi i casi, l'esatto contrario. I tipi di valore derivano dalla classe `System.ValueType` (o da `System.Enum` nel caso delle enumerazioni) e, tra questi, si possono contemplare gran parte dei tipi primitivi, come `System.Int32` o `System.Boolean`. `System.Object` e, in generale, i suoi derivati, come pure `System.String`, sono tipi di riferimento. Analizzeremo entrambi questi concetti nel quarto capitolo.

Tecnicamente parlando, la differenza principale tra le due tipologie è rappresentata dal fatto che i tipi di valore sono allocati direttamente nello stack, mentre i tipi di riferimento sono gestiti nel managed heap del Common Language Runtime.

Come abbiamo detto, il managed heap è utilizzato per allocare tipi di riferimento, mentre lo stack è usato solo per i tipi di valore. L'accesso allo stack è più veloce ma gli oggetti al suo interno vengono sempre copiati, quindi non sarebbe adatto nei casi in cui gli oggetti siano complessi e la relativa copia risulti dispendiosa. Il managed heap, d'altro canto, è gestito dal Garbage Collector, un componente in grado di deallocare in automatico la memoria, per cui i vantaggi, in caso di oggetti complessi la cui copia sia dispendiosa, si fanno sentire maggiormente.

Il [Capitolo 4](#) contiene una trattazione più esaustiva circa l'uso dei tipi di riferimento e affronta l'argomento da un punto di vista meno teorico e più orientato all'utilizzo di questi concetti all'interno di .NET.

[Conversioni tra tipi, boxing e unboxing](#)

Il fatto che il Common Language Runtime supporti un generico tipo base `System.Object` rappresenta, dal punto di vista dello sviluppo, un vantaggio di non poco conto. Dato che tutti gli oggetti derivano da `System.Object`, possiamo scrivere codice che utilizzi un generico `Object` e associare a quest'ultimo un'istanza qualsiasi di una classe o di un tipo di valore. In effetti, nel .NET Framework esistono moltissime funzionalità che si basano proprio su questa caratteristica dato che, dal punto di vista della programmazione a oggetti, è perfettamente lecito passare un oggetto derivato da `Object` laddove

un membro si aspetti quest'ultimo. Questa caratteristica è particolarmente importante perché consente di scrivere codice che è in grado di gestire diverse tipologie di tipi con le stesse istruzioni. Dato che tutti gli oggetti derivano da `object`, è assolutamente lecito assegnare un intero a una variabile di tipo `object`. Dato che `Integer` è un tipo valore (e quindi sta nello stack), in corrispondenza di una tale operazione il CLR è costretto a copiare il contenuto nel managed heap, dove risiedono i tipi riferimento come `object`.

Quest'operazione e la sua contraria sono chiamate rispettivamente **boxing** e **unboxing** e hanno un costo non trascurabile in termini di performance, per cui bisogna fare attenzione a non abusarne.

In conclusione, anche se la tentazione di scrivere codice utilizzando un generico `object` potrebbe essere forte, l'utilizzo diretto del tipo specifico, laddove possibile, consente di evitare queste due operazioni e dunque equivale a garantire maggiori performance all'applicazione oltre a comportare il beneficio implicito di scrivere codice `type safe`, cioè che usa direttamente il tipo più corretto, evitando problemi derivanti da codice non `strongly typed`.

La gestione della memoria: il Garbage Collector

Una delle tante funzionalità interessanti offerte dal CLR è la gestione automatica della memoria, che consente allo sviluppatore di evitare i cosiddetti **memory leak** (ossia quei casi in cui dimentichiamo di compiere le operazioni legate alla deallocazione dell'oggetto dalla memoria). Nelle applicazioni del mondo COM (Visual Basic 6 o ASP) queste problematiche sono più che diffuse e hanno come effetto quello di peggiorare le performance, se non di bloccare l'intera applicazione.

Per ovviare a questi problemi, tutti gli oggetti allocati nel managed heap vengono gestiti da un componente particolare, il cui ruolo è quello di liberare la memoria dagli oggetti non più utilizzati secondo un algoritmo non deterministico, perché, in generale, non possiamo controllare il momento in cui quest'ultimo eseguirà le sue funzionalità.

Il **Garbage Collector** (GC) entra in azione ogni qual volta vi sia la necessità di avere maggiori risorse a disposizione. Peraltro, un oggetto non viene necessariamente rimosso nel momento in cui non è più utilizzato dall'applicazione, ma può essere eliminato in una fase successiva.

Schematizzando, il Garbage Collector agisce seguendo questi passaggi:

- ❑ segna tutta la memoria managed come “garbage”, cioè spazzatura;
- ❑ cerca i blocchi di memoria ancora in uso e li marca come validi;
- ❑ scarica i blocchi non utilizzati;
- ❑ infine, compatta il managed heap.

Il Garbage Collector, per ottimizzare le sue prestazioni, utilizza un algoritmo di tipo generazionale.

Nella cosiddetta generation zero, che è la prima di cui viene fatto il collecting (il processo di raccolta spiegato sopra), vengono inseriti gli oggetti appena allocati. Quest’area è certamente il posto dove incontriamo la più alta probabilità di trovare un blocco di memoria non utilizzato e, al tempo stesso, è anche quella con una dimensione minore, quindi più rapida da analizzare. È talmente piccola che è l’unica delle tre generation a stare dentro la cache L2 della CPU. Quando un oggetto sopravvive a un collecting, perché raggiungibile, viene promosso alla generation successiva. Inoltre, il Garbage Collector lavora sulla generation 0, ma se la memoria dovesse non essere sufficiente, passa alla generation 1, e così via.

Il Garbage Collector alloca la memoria in modo tale che il managed heap rimanga il meno possibile frammentato. Questo rappresenta una grossa differenza rispetto ai classici unmanaged heap, dove la dimensione e la frammentazione possono crescere molto rapidamente.

Gli oggetti di grandi dimensioni meritano peraltro un discorso a parte. Questi oggetti, una volta allocati, rimangono di solito in memoria per lunghi periodi di tempo, per cui vengono mantenuti in un’area speciale del managed heap, che non viene mai compattata. Tenerli separati dal resto offre maggiori benefici rispetto a quanto si possa immaginare, perché il costo di spostare un oggetto di grandi dimensioni è ripagato dal fatto che il managed heap non viene più frammentato.

I concetti appena esposti valgono per quegli oggetti che fanno uso esclusivamente di risorse managed. Spesso invece, come nel caso di utilizzo di connessioni a database o handle di Windows (accesso a file su disco o ad altre risorse di sistema), le risorse sfruttate internamente sono di tipo unmanaged.

*Nel caso di utilizzo di risorse unmanaged, aspettare che il Garbage Collector faccia il proprio lavoro comporta un serio degrado della performance. A tale scopo è stato introdotto il cosiddetto **Pattern***

Dispose, cioè un approccio unificato al problema del rilascio delle risorse unmanaged. Attraverso l'implementazione di un'apposita interfaccia (il concetto di interfaccia è spiegato nel [Capitolo 4](#)), chiamata `IDisposable`, si fa in modo che tutti gli oggetti che la implementano abbiano un metodo `Dispose`, che deve essere invocato alla fine dell'utilizzo dell'oggetto stesso. Il metodo ha la responsabilità di chiudere e deallocare le risorse unmanaged in uso.

Il ruolo del Garbage Collector è molto importante, poiché consente allo sviluppatore di non curarsi dell'allocazione della memoria, che viene così gestita in automatico, in base alle reali necessità di risorse. Dilungarsi troppo nella sua analisi in questa fase costringerebbe a fare uno sforzo inutile, dato che dovrebbero essere dati per scontati molti concetti che, invece, saranno oggetto dei prossimi capitoli.

Componenti che fanno uso di risorse unmanaged devono anche avere un Finalizer per non incorrere in fenomeni di memory leak che, lo ripetiamo, al di fuori del codice gestito sono comunque probabili. In C# questo è possibile, utilizzando un override del metodo `Finalize` della classe `System.Object` oppure creando un distruttore della classe. È peraltro utile sottolineare che non è possibile utilizzare contemporaneamente sia `Finalize` sia `Dispose`. Il ruolo del `Finalize`, rispetto a quello del `Dispose`, è di liberare le risorse quando l'oggetto viene rimosso dal managed heap. Per questo motivo non va mai implementato quando non ce n'è davvero bisogno, dato che questa operazione consuma risorse. Inoltre, `Finalize` e `Dispose` vengono invocati in momenti differenti. Tuttavia, è utile sottolineare come, a differenza che nel mondo COM, un'applicazione managed non abbia bisogno di avere un'apposita sezione, generalmente collocata alla fine di un blocco di codice in cui vengono deallocati e chiusi gli oggetti gestiti, poiché questo servizio è offerto dal Runtime attraverso il CLR e, in particolare, tramite il Garbage Collector. Tutto questo offre il duplice vantaggio di rendere più semplice lo sviluppo e di migliorare, come abbiamo già detto, le performance e la stabilità delle applicazioni.

[Il concetto di Assembly](#)

Uno dei problemi più grandi del mondo COM è, senza dubbio, la forte presenza del concetto di dipendenza e di versione di un componente. Spesso,

l'installazione di un'applicazione che aggiorna la versione di un certo componente può avere effetti devastanti sulle altre, portando anche a un blocco totale.

Queste situazioni accadono perché, nel mondo COM, può esistere una e una sola versione di uno stesso componente caratterizzato da un identificativo univoco, detto **ProgID**. La sovrascrittura di un componente COM ha effetto globale, ossia riguarda tutte le applicazioni che puntano al suo identificativo. Per fare un esempio, una tipica situazione di malfunzionamento può riguardare ADO, l'insieme di librerie che nel mondo COM (e quindi con Visual Basic 6 o ASP) viene utilizzato per l'accesso ai database. In questi casi, il problema deriva spesso dal fatto di aver sovrascritto fisicamente un file, chiamato **Dynamic Link Library (DLL)**. Sostituendo una DLL, che attraverso il registro di Windows è associata al suo ProgID, le applicazioni smettono di utilizzare la precedente versione e cominciano a far riferimento a quella nuova. Se l'aggiornamento con una nuova versione, in genere, non causa gravi problemi, molto spesso la sovrascrittura con una versione precedente ne può portare in quantità.

Purtroppo, questo limite di COM è tale e non può essere superato, se non utilizzando alcuni trucchi, che però, di fatto, fanno sì che il componente sia, in ogni caso, diverso da quello originale.

L'altro grande problema di COM è il suo modello di deployment, cioè il sistema attraverso il quale, una volta creati, i componenti vengono registrati e distribuiti nell'ambiente di produzione. La registrazione di un oggetto COM richiede un accesso interattivo al computer, per poter lanciare da riga di comando `regsvr32.exe`. Questo strumento si occupa di aggiungere nel registro di Windows i riferimenti di cui si è appena detto, in modo che le applicazioni possano sapere dove trovare il file fisico che implementa le funzionalità necessarie.

Infine, COM comporta problemi anche durante la fase di sviluppo, dato che una DLL è fisicamente bloccata finché viene utilizzata e, in tali casi, per farne l'aggiornamento, è necessario fermare il servizio che la sta utilizzando.

.NET differisce molto dal mondo unmanaged, perché il corrispondente della DLL, cioè l'**assembly**, è un file che al proprio interno contiene il risultato della compilazione espresso in **MSIL**, i **metadati**, il cosiddetto **manifest**, che contiene informazioni sull'assembly, e le **risorse**, ossia elementi di varia natura (come immagini o altro) incluse direttamente dentro lo stesso file fisico per rendere minime le dipendenze esterne, come mostrato nella [Figura 1.4](#).

Il vantaggio di avere i metadati che descrivono i tipi è rappresentato dal fatto che, in collaborazione con le informazioni contenute nel manifest, l'assembly è in grado di comunicare all'esterno tutte le informazioni necessarie affinché possa essere utilizzato al meglio, rendendo quindi superflua la registrazione nel registro di Windows.

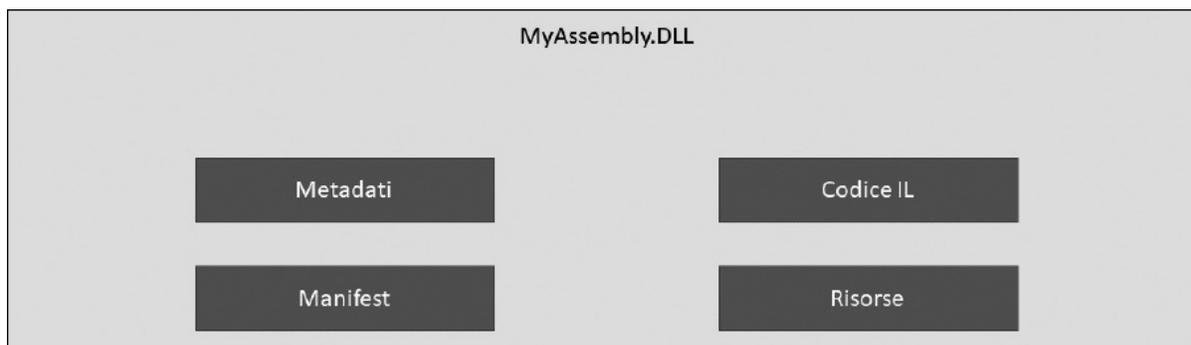


Figura 1.4 – Lo schema semplificato di un assembly.

*IL non è un linguaggio macchina e, così com'è, disponibile nel suo formato sorgente, in chiaro all'interno dell'assembly, non è propriamente immediato da capire. Ciò comporta che possiamo leggerlo direttamente (tramite un tool chiamato ILDASM, incluso in Visual Studio) o, addirittura, disassemblare il codice e ottenere il sorgente C# corrispondente. Ci sono tool come **.NET Reflector** che, sfruttando una caratteristica di .NET chiamata **Reflection**, consentono di navigare all'interno di un assembly per mostrarne tutti i tipi e i relativi elementi costitutivi, con la possibilità di disassemblarne il contenuto. Questi strumenti possono essere utilizzati per capire come funzionano internamente alcune caratteristiche delle classi che stiamo utilizzando, per leggere il codice in formato IL o in un linguaggio diverso da quello d'origine. Dal momento che, durante la fase di compilazione, tutte le variabili interne ai membri sono rinominate, vengono rimossi tutti i commenti e vengono effettuate alcune ottimizzazioni nel codice, con .NET Reflector non possiamo visualizzare il codice in un formato esattamente identico al sorgente originale. Peraltro, il risultato del disassemblaggio è tale da essere molto simile al codice di partenza. Esistono dei tool, chiamati obfuscator, che rendono il codice più*

difficile da decifrare. Si tratta comunque di sistemi che non garantiscono però una sicurezza al 100%.

Gli assembly, in genere, sono spesso utilizzati in forma privata, cioè senza essere registrati nel sistema. Secondo questo modello, per utilizzare un componente esterno è sufficiente copiare il file in un determinato percorso, insieme ai file dell'applicazione, senza la necessità di effettuarne la registrazione nel sistema. Questo vuol dire che in un computer possono coesistere più copie dello stesso assembly, anche in versioni diverse, senza che per questo motivo si verifichi alcuna interferenza tra loro. Tuttavia, in alcuni contesti, può essere necessario avere una sola copia centralizzata dello stesso componente, registrata a livello di sistema, evitando così di replicare le varie copie private per il file system.

Per soddisfare questa necessità, .NET sfrutta un sistema chiamato Global Assembly Cache (GAC), ovvero un repository centralizzato dove vengono inseriti gli assembly visibili a livello di sistema. Gli assembly della Base Class Library di .NET, su cui si basano tutte le applicazioni .NET, rappresentano l'esempio più evidente e scontato di assembly che risiedono in GAC.

Nella GAC può essere presente una sola copia per ciascuna versione di un assembly, dove per determinare univocamente lo stesso si usa una combinazione di nome, versione, culture e public key. Questa limitazione non rappresenta, di fatto, un problema, perché .NET implementa il concetto di versioning e non quello di compatibilità binaria, propria del mondo COM. Il Common Language Runtime è, infatti, in grado di caricare più versioni dello stesso assembly nello stesso istante, eliminando in un colpo solo i due problemi maggiori di COM.

L'esecuzione side-by-side di più versioni è supportata anche dall'intero .NET Framework, nel senso che sullo stesso sistema è possibile avere versioni diverse del .NET Framework installate contemporaneamente. Questa caratteristica contempla anche la possibilità di avere tutte le versioni del .NET Framework dalla 1.0 alla 4.8 installate sullo stesso computer, senza che esse interferiscano minimamente fra loro. In realtà, questo meccanismo negli anni si è un po' andato a perdere, poiché le versioni minor (per esempio la 4.8) in realtà modificano il comportamento delle precedenti (per esempio, la 4.5 o la 4.6). Questa caratteristica, invece, è presente in .NET Core, in cui ogni versione è isolata completamente dalle altre, può essere installata a livello globale (di intero server o client), oppure distribuita insieme all'applicazione. In questo modo, dunque, .NET Core consente di tenere legata a una specifica versione (anche una preview) una singola applicazione e di tenere più versioni

di .NET Core installate globalmente, senza che si disturbino tra di loro. GAC è una caratteristica di .NET Framework che non è stata portata in .NET Core, dove solo il runtime può installare librerie globali (mentre la GAC consente in .NET Framework di farlo a chiunque).

Per le applicazioni moderne l'uso della GAC è in genere limitato a casi particolari, vale a dire in quelle situazioni in cui un gruppo di applicazioni abbia bisogno di condividere uno o più assembly. In tutti gli altri casi possiamo limitarci a usare gli assembly in forma privata. Comunque, questa pratica si sta diffondendo sempre più, anche per lo sviluppo di applicazioni client, a tal punto che perfino il .NET Core viene distribuito insieme all'applicativo stesso, invece che richiederne l'installazione sulla macchina.

All'interno del manifest di un assembly, la versione ha il formato Major.Minor.Build.Revision. Un assembly viene considerato differente rispetto a un altro quando a variare tra i due sono i valori di Major o Minor.

Coloro che provengono dal mondo COM, in particolare gli sviluppatori ASP o VB6, hanno da sempre utilizzato il **late-binding** nel loro codice. Nel caso del motore di scripting di ASP, questo approccio consente di demandare alla fase di esecuzione la verifica dell'esistenza di un membro all'interno di un componente, con vantaggi dal punto di vista della scrittura del codice e altrettanti svantaggi dal punto di vista delle performance, dato che questo controllo deve essere eseguito ogni volta a Runtime. Per capire il significato di quanto detto, basti pensare di nuovo al caso di ADO citato in precedenza: aggiornando MDAC dalla versione 2.0 a una differente, il codice di una pagina ASP rimane comunque invariato, proprio perché gli oggetti COM sono referenziati nel codice tramite l'utilizzo del late-binding.

Il Common Language Runtime offre un servizio simile a quello descritto, con la differenza peraltro che l'associazione viene fatta sin dalla fase di caricamento dell'assembly (e non al momento dell'esecuzione), secondo un meccanismo noto come **early-binding**. Sfruttando i metadati, infatti, il CLR è in grado di conoscere a priori quali sono gli elementi esposti da ogni tipo presente in un assembly.

Esiste un meccanismo mediante il quale un assembly può essere firmato attraverso una chiave, chiamata Public Key Token, riportata all'interno del manifest insieme all'autore o al numero di versione. In questo modo, un assembly può essere identificato

tramite uno strong name, cioè un riferimento univoco dato dal nome dell'assembly, la sua versione, un hash e questa chiave pubblica. Lo strong name serve a .NET quando vogliamo utilizzare un tipo referenziato da una specifica versione di un assembly. Gli assembly possono anche essere firmati attraverso un certificato digitale, per identificare chi ha creato l'assembly, un po' come già avviene per gli ActiveX. Gli assembly registrati nella GAC devono essere provvisti di strong name.

Se nel mondo COM spesso viene utilizzato il trucco di sfruttare le interfacce per mantenere la compatibilità tra le diverse versioni, con .NET tutto questo non risulta necessario, dal momento che il Common Language Runtime è in grado di indirizzare la chiamata all'assembly più idoneo fino a quando un certo membro mantiene la stessa firma (ovvero non cambia nome né tipo, sia di ritorno sia dei parametri utilizzati).

Questa caratteristica fa sì che un'applicazione compilata, per esempio, per il .NET Framework 4.0, possa girare perfettamente con la versione 4.8 senza che debba essere necessario ricompilarla. Analogamente, un'applicazione per .NET Core 2 può girare con il runtime in versione 2.2.

Interoperabilità tra .NET e COM

Sbarazzarsi di tutta l'esperienza e il codice già scritto non è mai una pratica consigliabile. Per fortuna COM può essere sfruttato nel CLR attraverso un meccanismo che prende il nome di **interop** (interoperabilità). Affinché gli oggetti COM siano visibili al CLR, occorre che vengano creati dei tipi, chiamati proxy, che fungono da tramite tra le chiamate unmanaged e quelle managed, permettendo il passaggio di dati secondo le diverse regole di rappresentazione proprie dei due contesti di esecuzione. D'altra parte, Windows stesso è in pratica tutto unmanaged, per cui tale meccanismo consente, ad esempio, di invocare funzionalità native implementate proprio come oggetti COM.

L'interoperabilità offre comunque il vantaggio di consentire una meno brusca e traumatica migrazione del codice unmanaged, grazie al fatto che esso potrà essere ancora utilizzato all'interno delle applicazioni managed.

Questo concetto, su .NET Framework e Windows, resta ancora valido. .NET Core, invece, usa un approccio basato su file di risorse e referenze a

librerie, che gli consente di offrire implementazioni basate su librerie cross platform in maniera più semplice.

Nel caso fosse necessario sfruttare l'interop all'interno delle proprie applicazioni, esistono strumenti, documenti e whitepaper che facilitano il lavoro. A questo proposito possiamo trovare maggiori informazioni (in inglese) su MSDN, all'indirizzo: <https://aspit.co/a6r>.

Analisi di .NET Core

Attualmente .NET Core, a differenza di .NET Framework, che è limitato solo a Windows, supporta direttamente questi sistemi operativi:

- Windows Client: 7, 8.1, 10 (dalla build 1607);
- Windows Server: 2008 R2 SP1 o successivo;
- macOS: 10.12 o successivo;
- RHEL: 7 o successivo;
- Fedora: 26 o successivo;
- openSUSE: 42.3 o successivo;
- Debian: 8 o successivo;
- Ubuntu: 14.04 o successivo;
- Alpine Linux: 3.6 o successivo;
- SLES: 12 o successivo.

Il supporto è disponibile per architetture Windows x86 e x64, macOS x64, Linux x64 e Linux ARM32 (il processore utilizzato da diversi dispositivi, tra cui Raspberry PI). Alcune piattaforme, pur non essendo supportate direttamente, lo sono grazie al lavoro fatto dalla community. Le piattaforme e i sistemi operativi direttamente supportati, però, godono del normale **ciclo di supporto di Microsoft**, con la possibilità, per le aziende, di acquistare anche i pacchetti di supporto *ad hoc*, che garantiscono aggiornamenti e supporto

specifici. Pur essendo un prodotto open source, insomma, .NET Core gode dello stesso supporto di tutti gli altri prodotti Microsoft.

Tornando all'analisi delle piattaforme supportate, dobbiamo notare, in particolare, il supporto a **ARM32**, che apre la porta a tutta una serie di dispositivi (come le TV) che, a una prima analisi, mai penseremmo che possano beneficiare di un runtime per creare applicazioni web.

In realtà, ormai il web è oltre una serie di semplici pagine HTML ed è sempre più sfruttato per creare **endpoint raggiungibili via HTTP**, per offrire, per esempio, risultati in formato JSON.

La [Figura 1.5](#) mostra alcuni dei componenti chiave di .NET Core e li raffronta con i corrispondenti in .NET Framework e Xamarin, piattaforma con cui condividono alcune funzionalità.

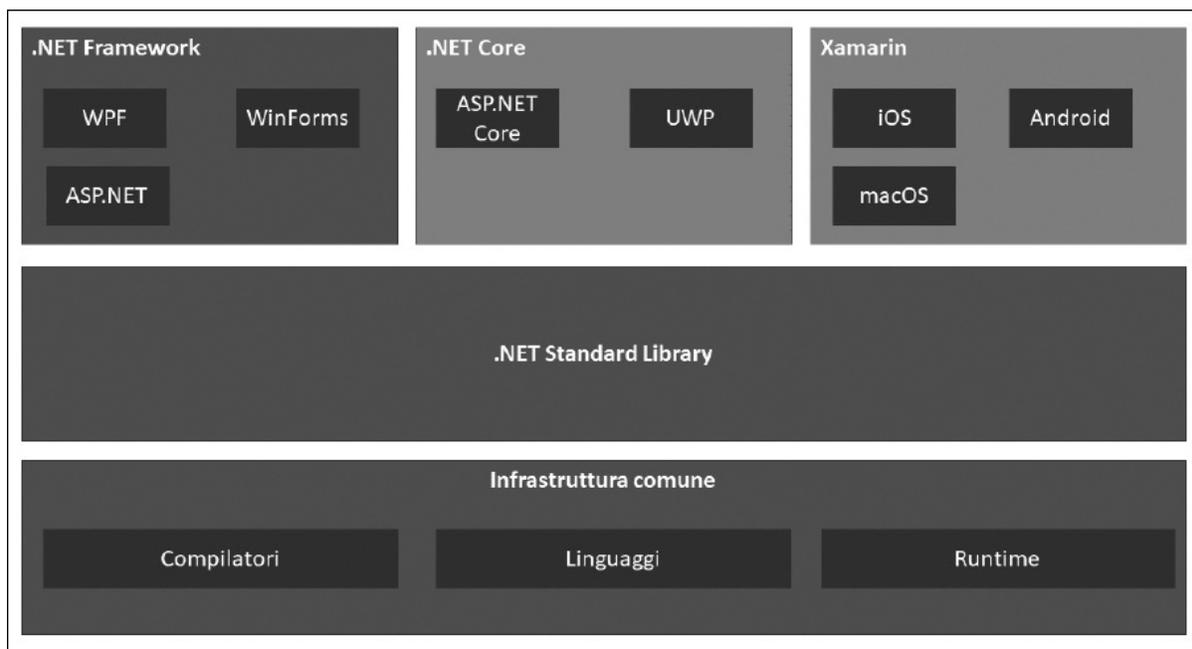


Figura 1.5 - I vari componenti di .NET Framework, .NET Core e Xamarin.

Un'altra particolarità di .NET Core, rispetto a .NET Framework, è che tutte le librerie sono distribuite sotto forma di **pacchetti NuGet**. Questo consente di ottimizzare l'applicazione e di includere le dipendenze necessarie, aggiornandole singolarmente e senza la necessità di aggiornare o distribuire l'intero framework.

NuGet è il package manager di riferimento per il mondo .NET. Consente di referenziare facilmente dipendenze attraverso il suo repository centralizzato. Maggiori informazioni sono disponibili su <http://www.nuget.org/>.

Un vantaggio significativo è che differenti versioni di .NET Core possono essere presenti nello stesso server (o, in generale, nella stessa macchina host), sia installate direttamente sia distribuite insieme all'applicazione stessa. In questo caso, si dice che .NET Core supporta il concetto di **installazione side-by-side** (fianco a fianco).

Scegliere .NET Core o .NET Framework

A questo punto, una domanda potrebbe sorgere spontanea: quando è meglio scegliere .NET Core e quando .NET Framework? La risposta dipende molto dal tipo di applicazione che andremo a creare. Volendo schematizzare, possiamo dire che .NET Core è ideale quando:

- Vogliamo il supporto cross-platform.
- Vogliamo sfruttare un toolkit più moderno e in continuo aggiornamento.
- Dobbiamo sviluppare un'applicazione con un'architettura a microservice o cloud.
- Vogliamo sfruttare i container basati su Docker.
- Abbiamo bisogno di performance e scalabilità.
- Vogliamo distribuire il runtime insieme all'applicazione stessa.

D'altro canto, .NET Framework è indicato se:

- Abbiamo già fatto investimenti su .NET Framework: in questo caso è più indicato non migrare l'applicazione, a meno che uno dei punti precedenti non dovesse risultare determinante.

- ❑ Ci affidiamo a librerie che non supportano .NET Core.
- ❑ Abbiamo bisogno di sfruttare componenti nativi di Windows.

In generale, la migrazione di un'applicazione esistente non è sempre semplice, poiché non esistono percorsi (o tool) preferenziali. Benché gran parte delle API siano compatibili, di fatto, la migrazione vuol dire creare una nuova applicazione da zero e procedere con l'incorporare le funzionalità esistenti, piuttosto che in una migrazione vera e propria, avvicinandosi maggiormente a una riscrittura dell'applicazione stessa.

Pur essendo molto simili tra loro, .NET Core non ha alcuni componenti che in .NET Framework hanno ricoperto un ruolo fondamentale, come gli AppDomain e la Code Access Security, perché fondamentalmente il runtime è differente e le stesse funzionalità di isolamento si possono ottenere isolando i processi o lavorando con i container.

Tra i fattori che possono bloccare il passaggio a .NET Core, vale la pena segnalare la mancanza di alcune funzionalità, che restano appannaggio esclusivo di .NET Framework. Tra queste è d'obbligo segnalare ASP.NET Web Forms, WCF (Windows Presentation Foundation) (in realtà, è presente solo una libreria per creare client, restando impossibile creare la parte server) e WF (Windows Workflow Foundation).

All'atto pratico, in un'applicazione moderna, tutte queste mancanze non si rivelano tali ma è comunque doveroso menzionarle.

Infine, una menzione particolare sul linguaggio: non tutti i tipi di progetto sono disponibili per tutti i linguaggi. .NET Core supporta direttamente **C#**, **VB** e **F#**, ma questi ultimi sono relegati ad alcuni tipi di applicazioni. Nel caso di VB è solo possibile creare applicazioni console e librerie, mentre F# consente di creare anche applicazioni basate su ASP.NET Core. C# resta il linguaggio con il supporto totale e, anche per questo, quello più diffuso nell'ambito .NET.

.NET 5 e il futuro di .NET

Il team di sviluppo di .NET ha annunciato nel maggio del 2018 che .NET Core, Xamarin e Mono confluiranno in un solo ecosistema, chiamato per semplicità .NET, che includerà tutte le funzionalità di queste piattaforme all'interno di una sola, per semplificare la scelta degli sviluppatori. In tal senso, ci sarà una major release di .NET ogni anno, per cui è già stato annunciato questi ciclo di rilasci:

- ❑ .NET 5: novembre 2020
- ❑ .NET 6: novembre 2021
- ❑ .NET 7: novembre 2022
- ❑ .NET 8: novembre 2023

Questo approccio consentirà a sviluppatori e aziende di pianificare con certezza le proprie applicazioni. Eventuali versioni minori potranno essere rilasciate secondo necessità. Poiché in .NET Core esiste il concetto di Long Term Support (LTS) e Current Release, tipico dei prodotti Open Source, è stata decisa anche la policy di supporto, che prevede che le versioni pari siano soggette a LTS: questa policy offre il supporto (vale l'opzione più lunga) per 3 anni dalla release, oppure un anno dalla release LTS successiva. Maggiori informazioni su LTS sono disponibili su <https://aspit.co/bvp>.

Conclusioni

.NET rappresenta la base su cui C# poggia tutta la propria infrastruttura, pertanto la conoscenza degli argomenti affrontati in questo capitolo si rivelerà preziosa nel corso del libro per comprendere i concetti che tratteremo in seguito.

Il Common Language Runtime, insieme al Common Type System, alla Common Language Specification, alla Cross-Library Interoperability e al modello di sicurezza, consente di sfruttare un insieme di funzionalità decisamente comode da utilizzare, inserite in una piattaforma applicativa completa, matura e pensata tanto per le applicazioni più semplici quanto per quelle più complesse.

In più, la garanzia di poter scrivere componenti in linguaggi differenti e fare in modo che possano comunque scambiarsi informazioni è essenziale, poiché consente di aggiungere alle proprie applicazioni anche componenti di terze parti che, una volta compilate, diventano IL e sono dunque convertite in un linguaggio comune.

Seppure molto vasta, la Base Class Library non copre tutte le necessità possibili, dunque è tutt'altro che raro che sia necessario ricorrere a funzionalità aggiuntive.

È su queste solide fondamenta che C# consente di creare le applicazioni di ogni tipo. La conoscenza delle basi, così come della sintassi del linguaggio, rappresenta un prerequisito essenziale, da cui non si può prescindere. Per questo motivo, nel prossimo capitolo ci soffermeremo su quelle che sono le peculiarità di base del linguaggio stesso.

Visual Studio

Per sviluppare applicazioni basate su .NET, tendenzialmente si utilizza Visual Studio, giunto ormai alla versione 2019.

Visual Studio è un'IDE (Integrated Development Editor), cioè un ambiente all'interno del quale è possibile gestire l'intero ciclo di sviluppo di un'applicazione. È disponibile in diverse versioni, che racchiudono funzionalità specifiche per ambiti ben definiti. La versione Community, per esempio, è indicata qualora volessimo iniziare a sperimentare le caratteristiche di C#, senza necessità di acquistare una versione commerciale: è gratuita per progetti open source, ed entro certi limiti anche per i prodotti commerciali. Maggiori informazioni sulle limitazioni di questa versione e su tutti i pacchetti di Visual Studio sono disponibili all'indirizzo <http://www.visualstudio.com/>.

All'interno di questo capitolo vedremo come l'IDE e le sue funzionalità possono essere utilizzate per creare applicazioni in C# di qualsiasi tipo. Vedremo poi come sfruttare le funzionalità dell'Intellisense, come funzionano le solution, i progetti, la compilazione e il debugger.

L'IDE di Visual Studio

Visual Studio è un ambiente di sviluppo che consente di gestire lo sviluppo di applicazioni, di qualsiasi tipo esse siano, grazie a un meccanismo di estendibilità. Possiamo creare applicazioni basate su ASP.NET, su Universal Windows Platform (per Windows 10), WPF, WCF, Entity Framework, LINQ, oppure console, mantenendo lo stesso ambiente e

sfruttando lo stesso linguaggio. Visual Studio è un'applicazione MDI (Multiple Document Interface), in grado di consentire l'apertura contemporanea di più documenti, all'interno di tab.

Questo capitolo, così come l'intero libro, è basato sulla versione in inglese di Visual Studio. Questo fatto non rappresenta un vero problema, dato che tra le due versioni ciò che cambia è solo il nome delle voci nei menu, non la posizione degli stessi, né le relative funzionalità.

L'IDE di Visual Studio si presenta con una pagina iniziale come quella illustrata nella [Figura 2.1](#).

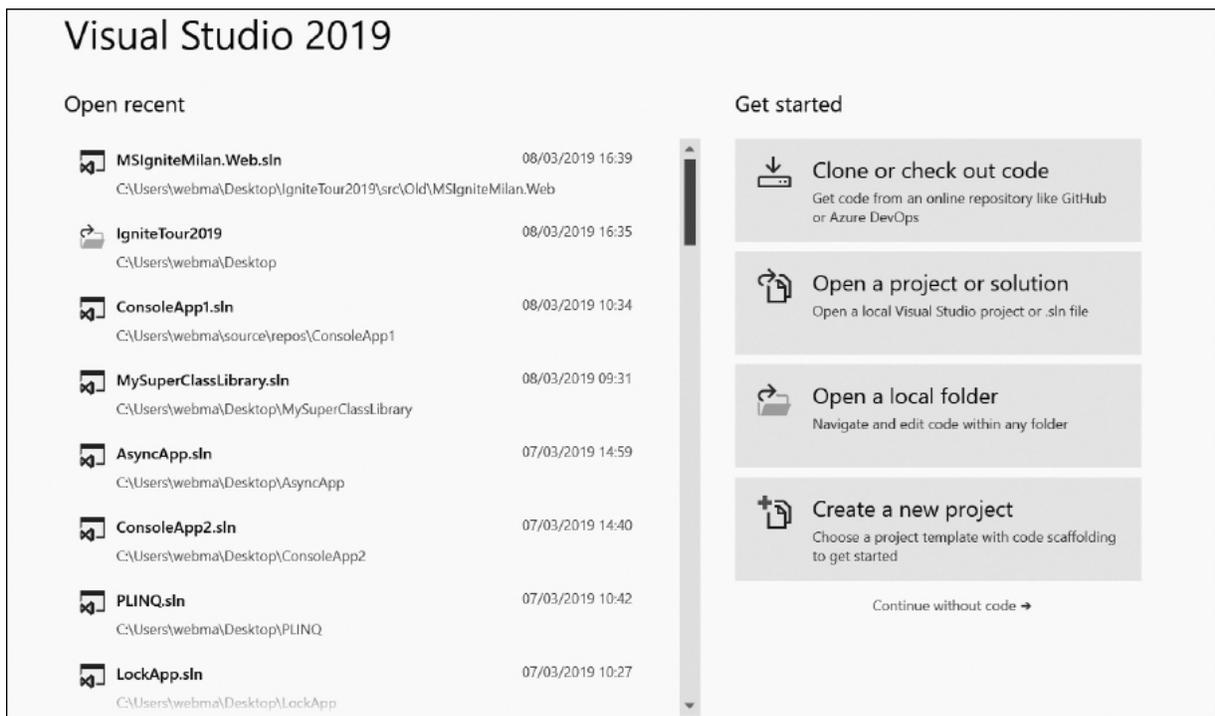


Figura 2.1 – La pagina iniziale di Visual Studio.

Come si può notare nella figura precedente, l'area di lavoro è suddivisa in zone, dove nella parte centrale spicca quella denominata “Text Editor”. Esiste tutta una serie di zone che è opportuno approfondire, per comprenderne al meglio l'utilizzo all'interno di Visual Studio.

Nella versione 2019, Visual Studio continua ad avere quel particolare design, mutuato dal Microsoft Design Language (lo stesso che ha portato alla UI delle ultime versioni di Windows), che privilegia la semplicità (nei colori e nello stile). Quest'ultimo aggiornamento si differenzia dagli altri perché introduce una nuova start screen, visibile nella [Figura 2.1](#), e semplifica ulteriormente il design, aumentando l'area di visualizzazione del codice.

Text Editor, designer e Intellisense

A partire da Visual Studio 2010, il **Text Editor** è liberamente posizionabile, anche in un eventuale secondo monitor. Non è più necessario che sia ancorato alla schermata principale: in questa modalità, chiamata docked, la finestra segue il normale comportamento di un ambiente MDI. All'interno di questo editor, come il nome suggerisce, viene scritto il codice, sia esso C#, markup HTML, codice XAML (per WPF) o altro codice. Questa funzionalità è stata ulteriormente rifinita e migliorata, così da offrire un supporto ancora migliore in scenari multi-monitor e, novità di Visual Studio 2019, anche per monitor con DPI differenti (per esempio quello principale di un portatile e un monitor secondario esterno).

All'interno del text editor è fornita una funzionalità di completamento del codice, che aiuta nello scrivere le diverse varianti possibili: si tratta dell'**Intellisense**.

In Visual Studio 2019, come da tradizione, l'Intellisense è stato ulteriormente potenziato rispetto alle versioni precedenti: oggi è disponibile una nuova funzionalità, chiamata IntelliCode, che, grazie a un set di funzionalità basate su Intelligenza Artificiale (AI), è in grado di analizzare il nostro codice e proporre, in base all'uso che ne facciamo, codice che rispecchia il nostro modo di scrivere.

Si può sempre riportare in primo piano l'Intellisense, semplicemente premendo la sequenza `Ctrl+Spazio`, mentre ci si trova nel Text Editor. La [Figura 2.2](#) mostra l'Intellisense all'opera.

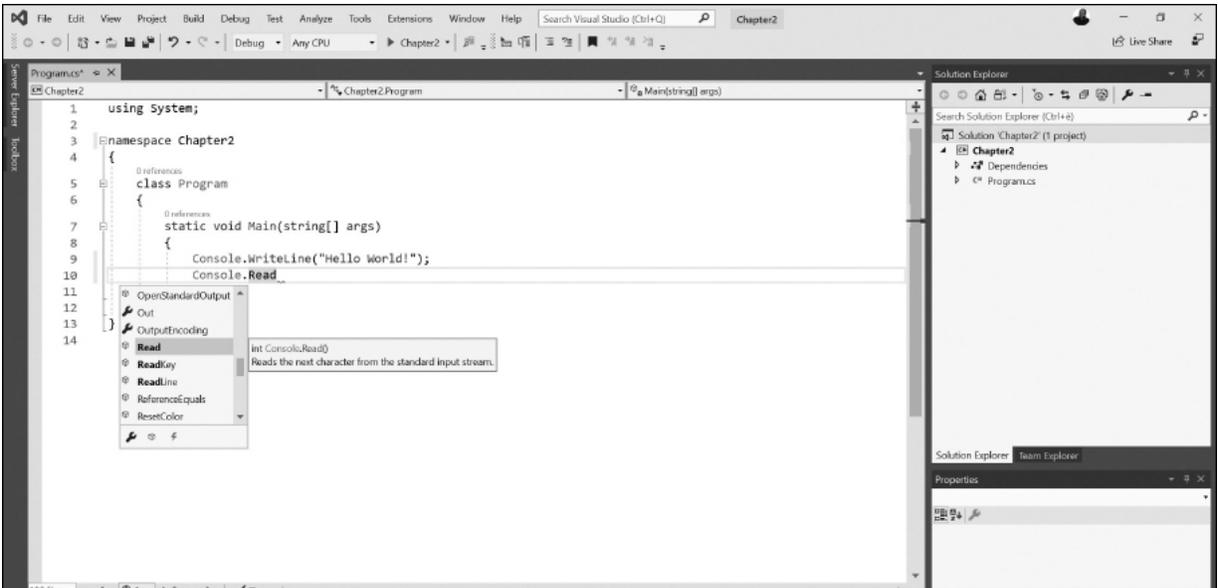


Figura 2.2 – L’Intellisense all’opera.

In alcuni scenari, come per le web form di ASP.NET o per le applicazioni WPF, il Text Editor può occupare metà dello spazio in altezza, utilizzando una modalità chiamata “**Split View**”. In questo caso, l’altra metà dello spazio è occupata dal **designer**, che è il sistema attraverso il quale è possibile avere una rappresentazione visuale dell’interfaccia su cui si sta lavorando. La “Split View” è visibile nella [Figura 2.3](#).

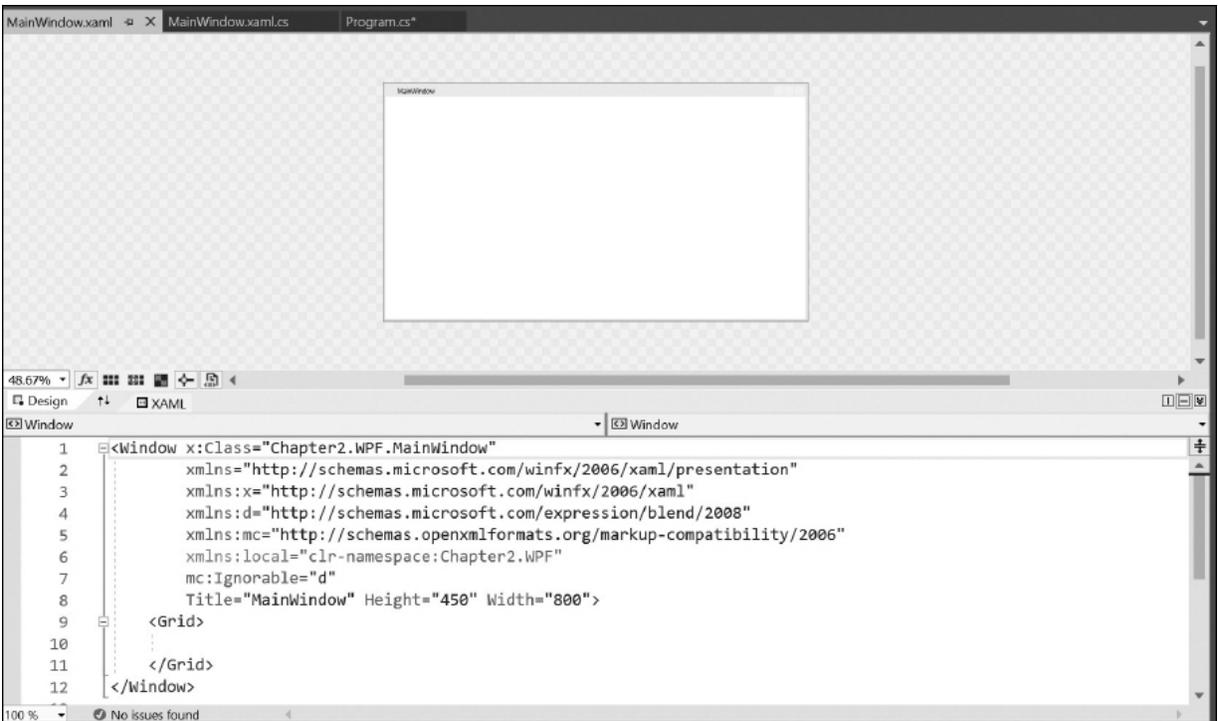


Figura 2.3 – La Split View di Visual Studio.

Il **designer** è una modalità già nota se in passato abbiamo utilizzato un ambiente RAD, come Visual Basic 6, e consente di sviluppare trascinando gli oggetti all'interno dell'area, posizionandoli e gestendone le proprietà. In questo caso, sono disponibili due nuovi tipi di toolbar, che prendono il nome di toolbox e property editor.

[Toolbox](#)

La toolbox è la zona all'interno della quale sono contenuti i controlli, da poter trascinare nel designer. Generalmente è possibile compiere la stessa operazione anche verso il Text Editor, con l'effetto che in questo caso, quando possibile, viene generato il corrispondente codice (o markup). La toolbox è visibile in dettaglio nella [Figura 2.4](#).

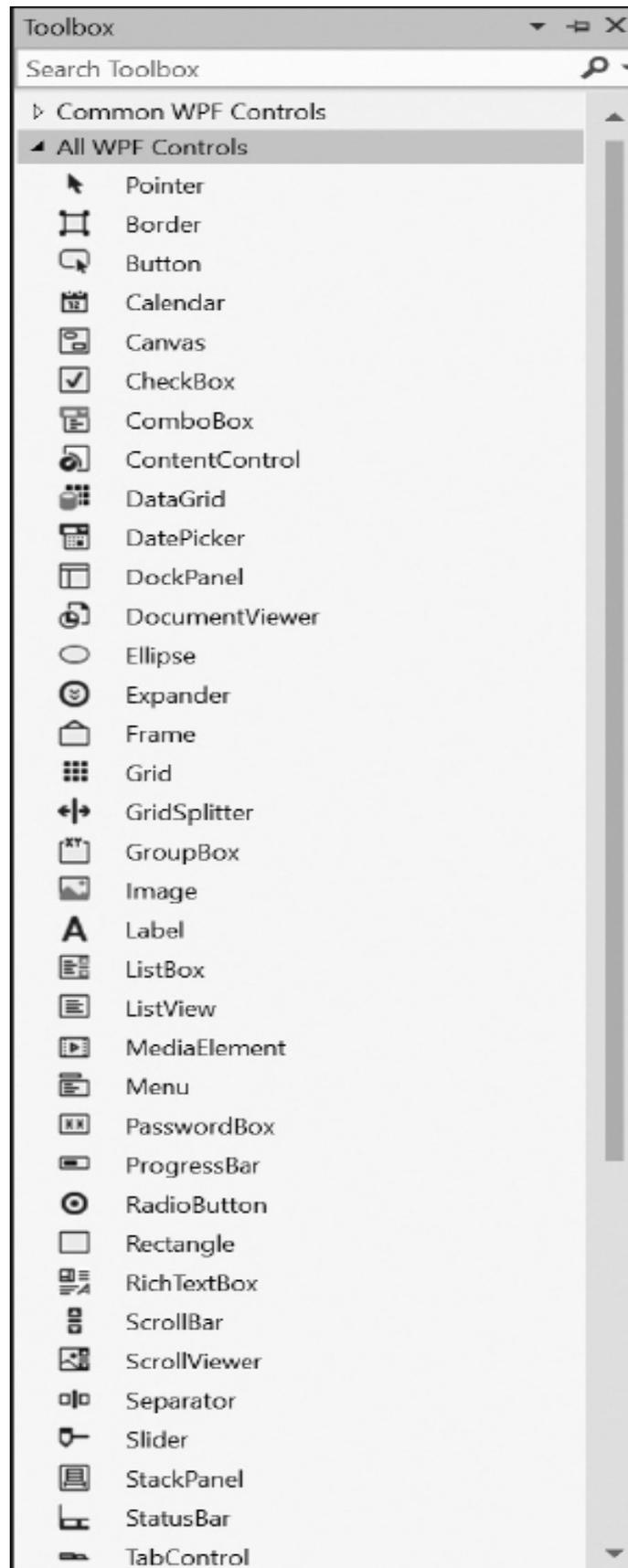


Figura 2.4 – La toolbox di Visual Studio.

Eventuali oggetti di terze parti vengono registrati in automatico e raggruppati in maniera opportuna, cosa che, tra l'altro, avviene comunque anche per gli oggetti già inclusi.

Property Editor

Il Property Editor è l'area all'interno della quale possono essere definite le proprietà. Generalmente viene utilizzato in combinazione con il designer, ma è possibile sfruttarlo anche quando si utilizza il Text Editor. È visibile in dettaglio nella [Figura 2.5](#).

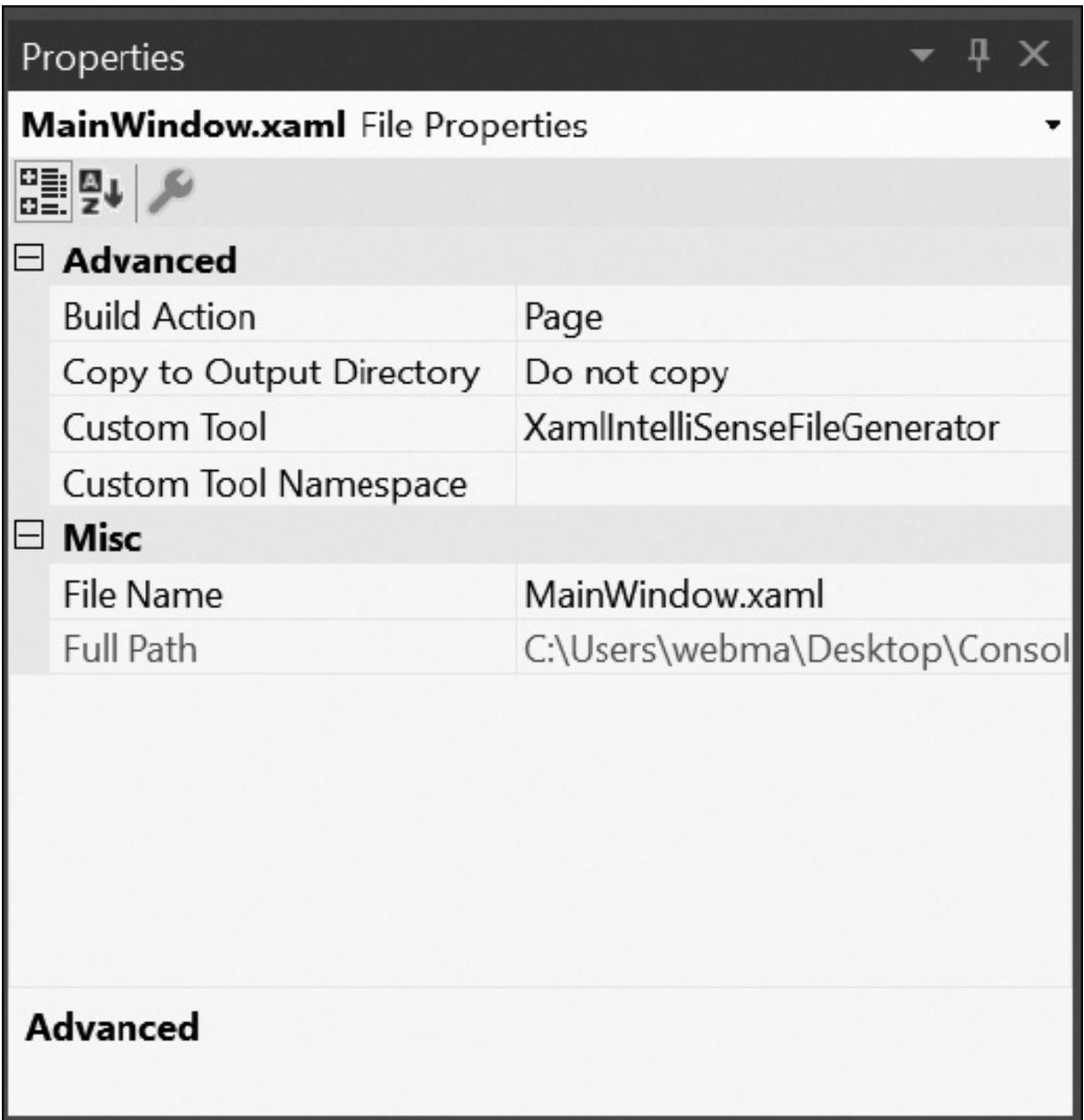


Figura 2.5 – Il Property Editor di Visual Studio.

Si tratta di un editor vero e proprio, che di default ordina le proprietà degli oggetti raggruppandoli per tipologia, ma che può mostrare anche il classico ordinamento alfabetico. Le proprietà di tipo complesso possono, a loro volta, aprire menu o finestre aggiuntive, piuttosto che offrire comportamenti che vanno oltre la semplice immissione di testo. Questi comportamenti sono definiti in fase di creazione dei controlli stessi e sfruttano l'estendibilità di Visual Studio per fornirci un ambiente più semplice da utilizzare.

Altre aree dell'IDE

Chiudiamo questa rapida carrellata sulle aree di Visual Studio parlando della **toolbar**. In quest'ultima vengono aggiunte una serie di barre che semplificano l'utilizzo dei task più comuni. Se facciamo uso di altri linguaggi durante il nostro sviluppo, oltre a C#, è consigliabile, quando Visual Studio parte per la prima volta, scegliere come profilo "General development", che aggiunge in tal senso le funzionalità più diffuse.

La nostra prima analisi dell'IDE finisce qui: non ci dilungheremo molto riguardo le singole voci dei menu, non essendo questo un libro su Visual Studio, il cui uso comunque è intuitivo e impareremo a padroneggiare nel corso dei prossimi capitoli.

Continuiamo a dare un'occhiata all'ambiente, passando al primo passo da compiere per la nostra nuova applicazione: creare il relativo progetto.

Creare un progetto

Il progetto rappresenta il punto da cui iniziare per creare un'applicazione. Possiamo creare un nuovo progetto dall'apposita voce, sotto il menu File. Visual Studio contiene un numero molto elevato di template, alcuni dei quali sono visibili nella [Figura 2.6](#).

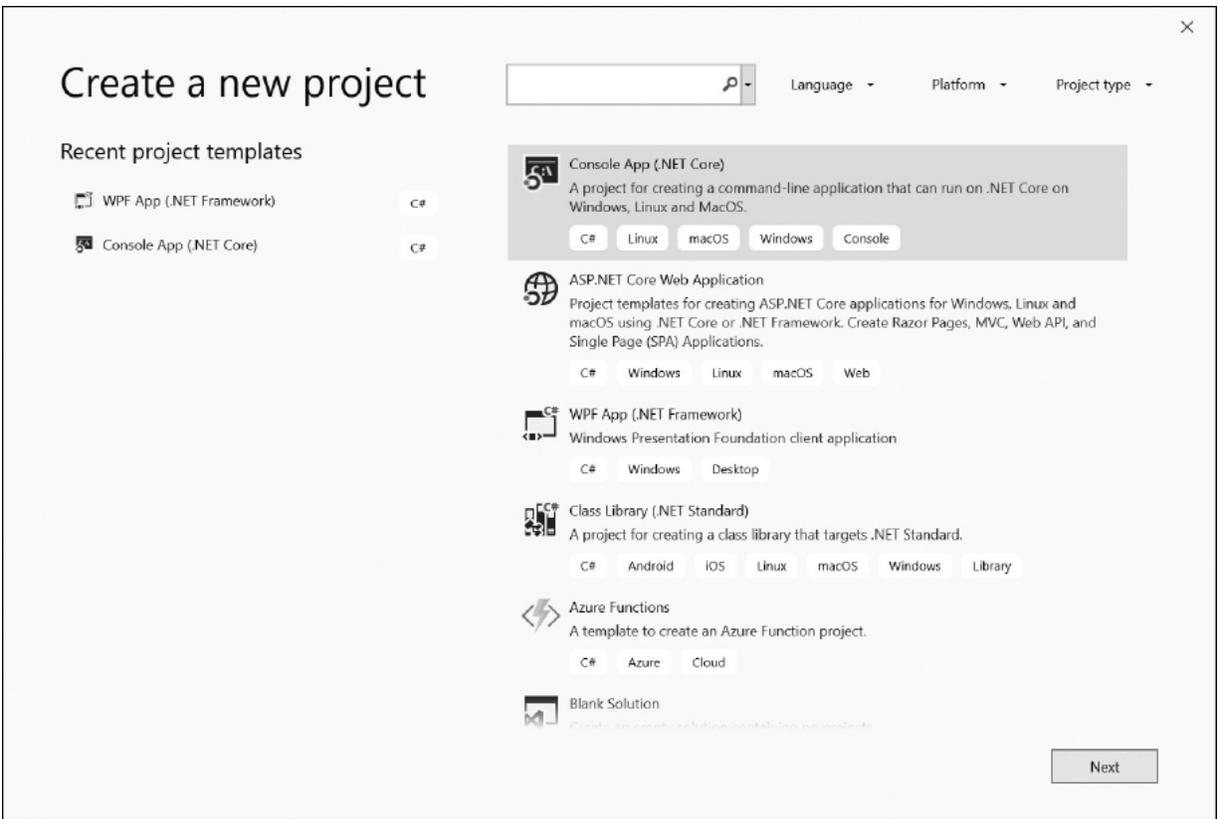


Figura 2.6 – Il wizard per creare un nuovo progetto.

A seconda della versione, questi template di progetto potrebbero essere in numero differente. La versione Express, per esempio, ne contiene un numero limitato.

In linea di massima i template di progetti non sono altro che un insieme già pronto di file configurati per sviluppare uno specifico tipo di progetto, per cui per implementare una data funzionalità è necessaria meno fatica. Visual Studio raggruppa questi template all'interno di aree, così che sia più facile individuare quelli di nostro interesse. Per esempio, nella sezione “web” sono presenti un insieme di template specifici per ASP.NET, mentre quella denominata “Windows” racchiude una serie di template specifici per questo tipo di applicazione.

Il multi-targeting di .NET in Visual Studio

Dando un'occhiata più da vicino alla [Figura 2.6](#), non può sfuggirci la possibilità di selezionare da un menu apposito quale versione di .NET

vogliamo utilizzare per il nostro progetto. Visual Studio 2019 offre – così come accade ormai dalla versione 2010 – il supporto a più versioni di .NET Framework e .NET Core, onorando per ciascuna le relative caratteristiche. Questo vuol dire che scegliendo una versione, Visual Studio adatterà l'IDE, i compilatori, l'Intellisense e, in certi casi, anche le opzioni disponibili in alcuni menu. Con l'attuale versione possiamo creare applicazioni per .NET Framework 2.0 o superiori e .NET Core (tutte le versioni). Da notare che il supporto per .NET Core 3 è disponibile solo a partire da Visual Studio 2019: utilizzando Visual Studio 2017, dunque, è possibile sviluppare con versioni precedenti a .NET Core 3.

Questa capacità di ospitare qualsiasi versione di .NET, permette di avere una sola versione di Visual Studio installata per lavorare con applicazioni basate su versioni precedenti.

Visual Studio è in grado di gestire i progetti pensati per le versioni precedenti: in altre parole, anziché migrare il file di progetto, come avveniva in passato, lo stesso viene lasciato com'è, garantendo la possibilità di essere sfruttato in team nei quali gli sviluppatori fanno uso di versioni diverse di Visual Studio.

Il concetto di progetto e soluzione

A questo punto è necessario fare chiarezza sui concetti di progetto e soluzione, menzionati già diverse volte.

Il progetto è l'insieme dei file (sorgenti o risorse), che generalmente concorrono alla creazione dell'applicazione, mentre la soluzione è l'insieme dei progetti, cioè l'applicazione stessa. Molto spesso, specie in applicazioni semplici, la soluzione contiene un solo progetto. Viceversa, in scenari più complessi, è molto diffusa la presenza di più progetti all'interno della soluzione.

I progetti saranno compilati e creeranno il risultato della compilazione che, lo ricordiamo, nel gergo di .NET si chiama assembly.

Data la natura di Visual Studio, all'interno di una soluzione possono convivere tranquillamente progetti scritti in linguaggi diversi. Possiamo quindi avere, a fianco di un progetto in C#, uno in Visual Basic o C++ managed.

Facendo un paragone con Visual Basic 6, la soluzione corrisponde ai vecchi file .vb6, mentre il progetto al file .vbproj. Una volta creato il progetto, il passo successivo consiste nel gestire lo stesso.

È possibile cambiare in qualsiasi momento il nome della soluzione e dei progetti. Nel caso di questi ultimi, occorre prestare attenzione al fatto che per cambiare il nome del namespace utilizzato dalle classi è necessario accedere alle proprietà del progetto ed agire sulla schermata che apparirà. Cambiare il nome degli elementi all'interno del solution explorer rappresenta la scelta migliore, perché ha effetto sul nome fisico del file e, comunque, consente di mantenere intatta e valida la struttura della soluzione.

Gestire soluzione e progetto

La parte di creazione del progetto è quella tutto sommato più semplice. Durante lo sviluppo ci capiterà, invece, di dover gestire la soluzione e i progetti che contiene.

Da un punto di vista pratico, è importante notare come, nel caso di soluzioni con più progetti, si possa specificare quello predefinito (che viene lanciato in fase di debug) attraverso la voce "Set as start-up project" che viene visualizzata quando si accede, con il tasto destro del mouse, al menu contestuale del progetto stesso. Non c'è limite alla tipologia di progetti che possono essere contenuti, anzi in applicazioni complesse è spesso possibile trovare, nella stessa soluzione, tipologie di progetti diversi: applicazioni console, WPF, ASP.NET e class library. Queste ultime sono molto comode, in quanto consentono di raggruppare all'interno di un solo assembly una serie di funzionalità che possono essere condivise tra più progetti all'interno della stessa soluzione.

Aggiungere un progetto alla soluzione

Abbiamo visto che Visual Studio consente di avere più progetti all'interno della stessa soluzione. Per dimostrarlo, procediamo creando una applicazione di tipo Console. Successivamente, per fare in modo che alcune funzionalità possano essere condivise tra più progetti, proviamo ad aggiungere un altro, questa volta di tipo class library. Per compiere questa

operazione è sufficiente premere con il tasto destro del mouse sulla soluzione e selezionare la voce “Add” e quindi “New Project”, come si vede nella [Figura 2.7](#).

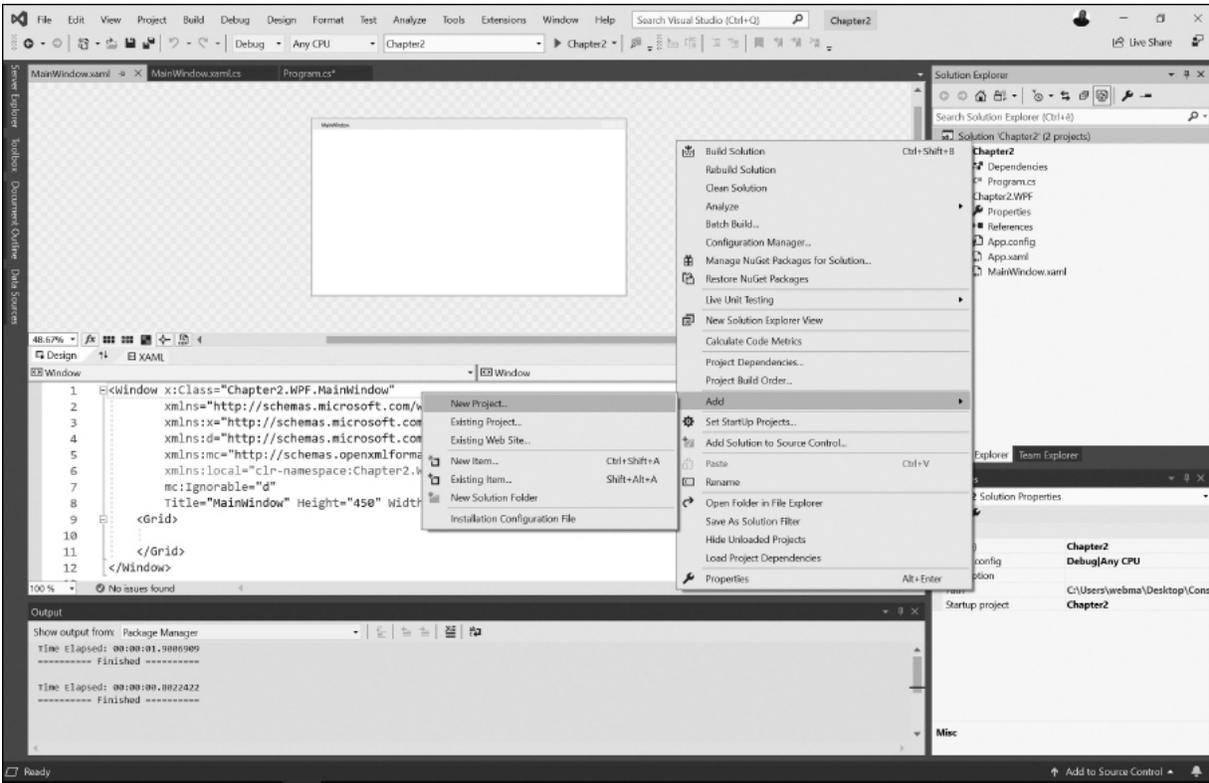


Figura 2.7 – L’aggiunta di un nuovo progetto alla soluzione.

È possibile aggiungere anche un progetto esistente, che può ritornare comodo quando abbiamo un progetto esistente e vogliamo gestirlo da una sola soluzione. In questo caso, la voce da selezionare è “Existing Project”, utilizzando sempre la stessa identica sequenza di passaggi appena vista.

Perché il progetto con l’applicazione console veda quello con la class library, dobbiamo imparare a gestire le referenze.

Gestione delle referenze

Le referenze servono al compilatore (e quindi anche all’ambiente) per capire dove si trovano le risorse esterne. All’interno del solution explorer sono visibili sotto il ramo References (se .NET Framework) o

Dependencies (se .NET Core). Questa voce, secondo il tipo di progetto, potrebbe non essere sempre visibile. Ad ogni modo, per poter aggiungere una reference, Visual Studio offre un apposito menu, raggiungibile premendo il tasto destro sul progetto stesso all'interno del Solution Explorer, oppure dall'apposito menu.

In realtà, ormai, la maggior parte delle dipendenze arriva da NuGet, che è il package manager di .NET. Un package manager consente di avere una sorgente centralizzata in cui sono pubblicate le librerie, che quando vengono aggiornate possono poi essere prese direttamente dalle applicazioni nell'ultima versione disponibile, semplificando la vita allo sviluppatore: .NET Core, in effetti, utilizza questo sistema per tutte le librerie, anche quelle considerate di sistema, che possono quindi essere aggiornate più facilmente. Agendo su questa voce, si apre una schermata come quella visibile nella [Figura 2.8](#).

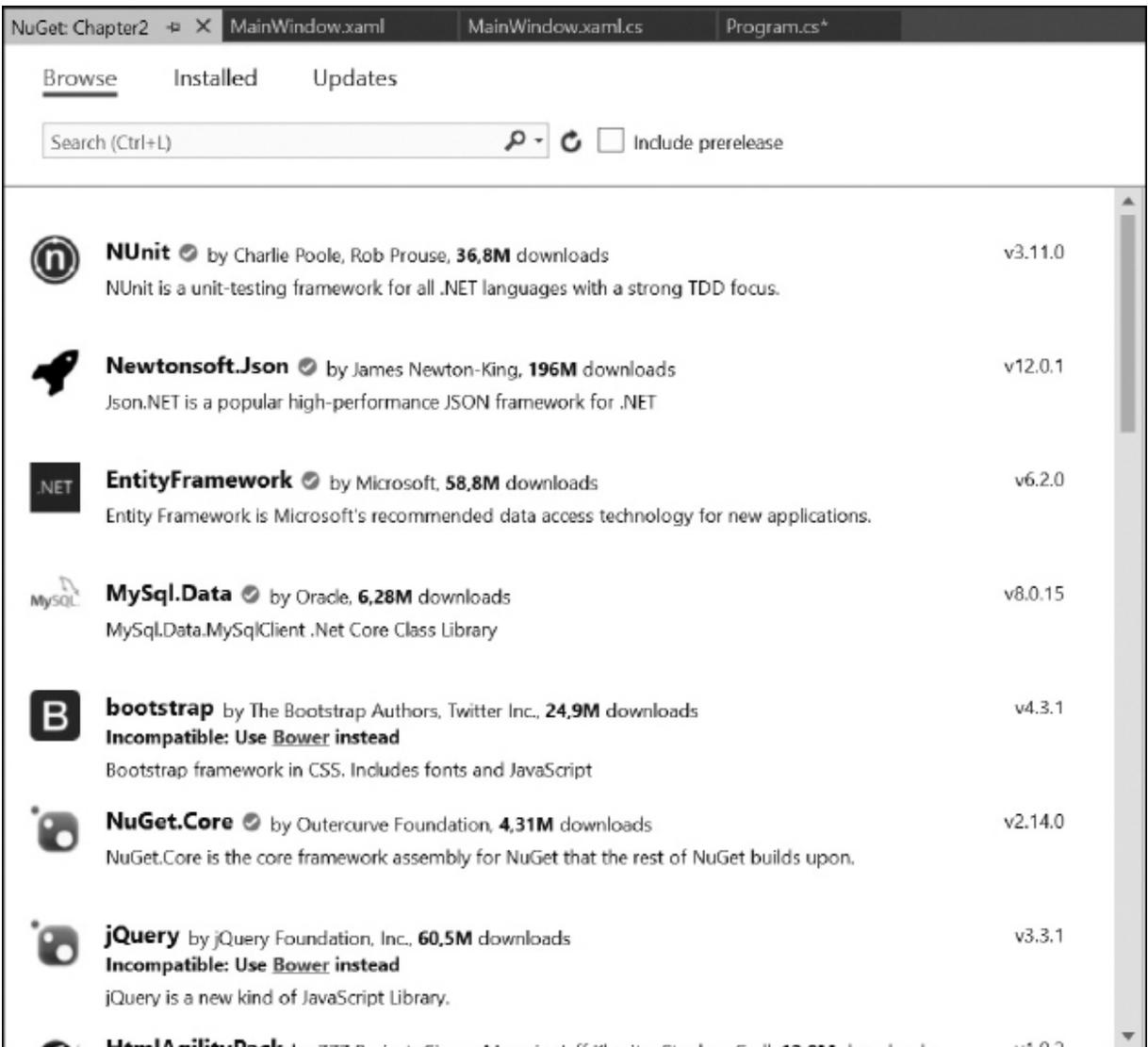


Figura 2.8 – Il menu per l’aggiunta delle dipendenze da NuGet al progetto corrente.

Nel caso di .NET Framework, dove molte dipendenze sono ancora locali, dovremo utilizzare la voce “Add Reference...” e poi selezionare il tab “Projects”, da cui poi indicheremo l’altro progetto. È anche possibile aggiungere referenze ad assembly presenti in GAC (come quelli della BCL), a oggetti COM, o ad assembly compilati, seguendo lo stesso approccio.

Completata questa operazione, siamo in grado, in un colpo solo, di compilare tutta la soluzione.

Gestione di directory nella solution

È possibile tenere nella soluzione delle directory che non siano effettivamente progetti, ma possano servire per raggruppare logicamente gli stessi, oppure per includere altre risorse, come file di testo, o documentazione. Questo è possibile dalla voce “Add new solution folder”, raggiungibile dal menu contestuale sulla solution, come è visibile nella [Figura 2.9](#).

I file contenuti in queste directory, se non sono all'interno di un progetto, non saranno compilate insieme alla soluzione.

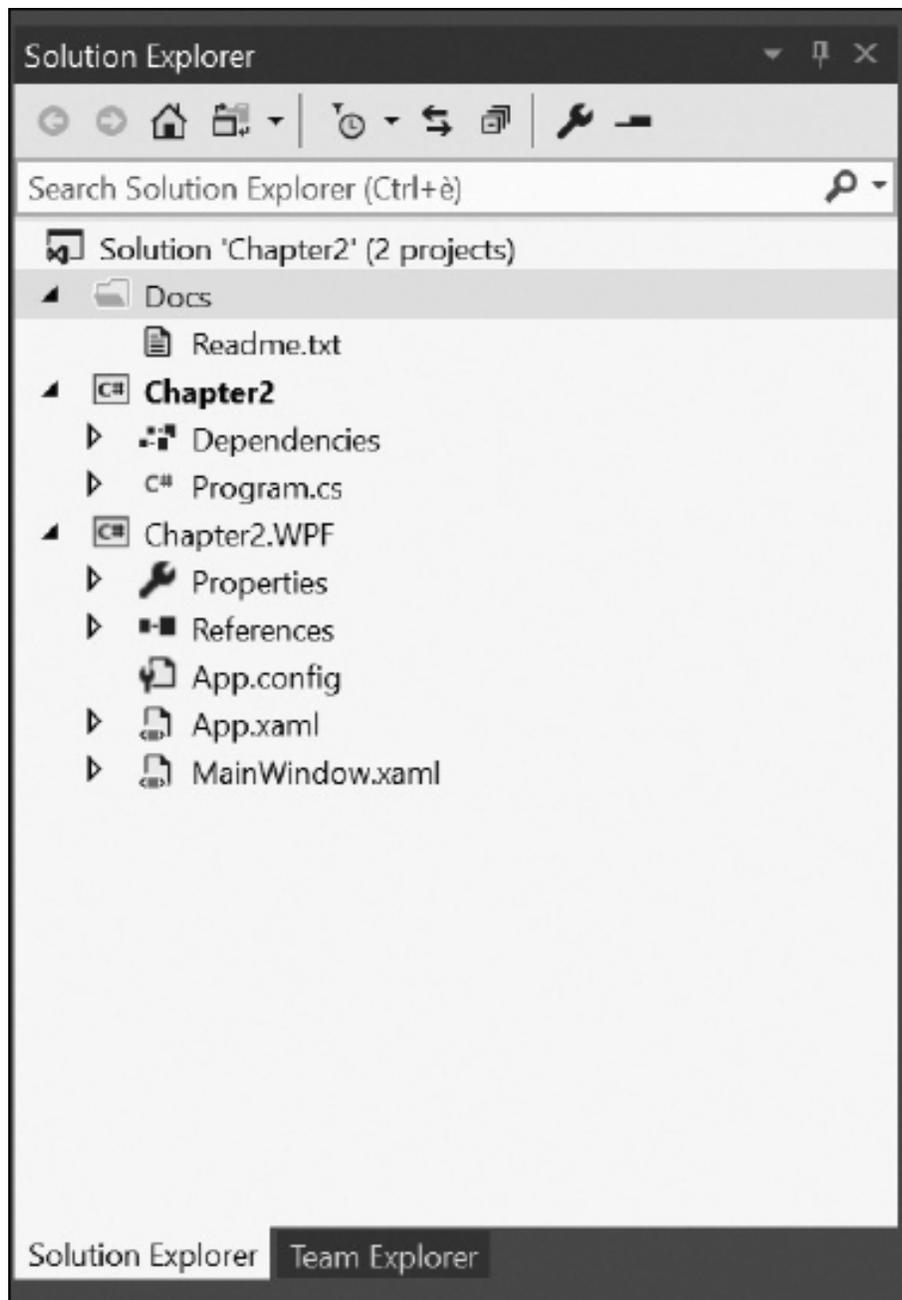


Figura 2.9 – Aggiunta di una directory alla soluzione.

Gestione del codice sorgente

È buona norma gestire le modifiche al codice sorgente appoggiandosi a un tool apposito, che ne gestisca le varie versioni. In tal senso, si parla di source control repository, ovvero di un sistema in grado di garantire che noi

possiamo effettuare il versioning del codice sorgente, potendo tornare indietro nel caso una data modifica non fosse qualitativamente adeguata. Questo approccio dovrebbe essere perseguito tanto da chi lavora in team, dove sarebbe impensabile gestire le modifiche scambiandosi file, tanto dal singolo sviluppatore, che così può avere una copia di backup del proprio sorgente.

Storicamente, gli sviluppatori Windows sono stati abituati a utilizzare Team Foundation Server (TFS), ma quest'ultimo è stato ormai del tutto soppiantato da **Azure DevOps** e **Azure DevOps Server**, che, come TFS, non sono limitati al versioning del codice sorgente ma sono in grado di gestire l'intero ciclo di vita di un'applicazione, godendo anche, tra le altre cose, di funzionalità di team management e di gestione dei task e delle build.

In Azure DevOps (la versione cloud) e Azure DevOps Server (la versione installabile su un proprio server) è ormai suggerito di utilizzare il protocollo Git per il source control. Pertanto, all'interno della [Figura 2.10](#), è visibile la schermata che consente di gestire il codice sorgente, raggiungibile dalla voce "Team Explorer", un tab che in genere è posizionato di default di fianco al tab "Solution Explorer".

Visual Studio poi supporta in maniera completa qualsiasi repository Git (non per forza basato su Azure DevOps), consentendo di accedere a repository basati su questo source control server di tipo distribuito, che va molto in voga in questo periodo e spesso è utilizzato come preferito dai progetti open source, anche per via del suo supporto cross-platform. In Visual Studio, dunque, è perfettamente integrato anche il supporto ai repository di GitHub.

Consigliamo di utilizzare un repository anche per progetti a cui si lavora da soli e non in team, poiché aggiunge uno storico alle modifiche del codice e consente di avere sempre i backup dello stesso. Tra l'altro, ci sono delle versioni di Azure DevOps che hanno delle licenze gratuite da poter utilizzare con repository privati.

Maggiori informazioni su Visual Studio e Azure DevOps sono disponibili su <http://www.visualstudio.com>

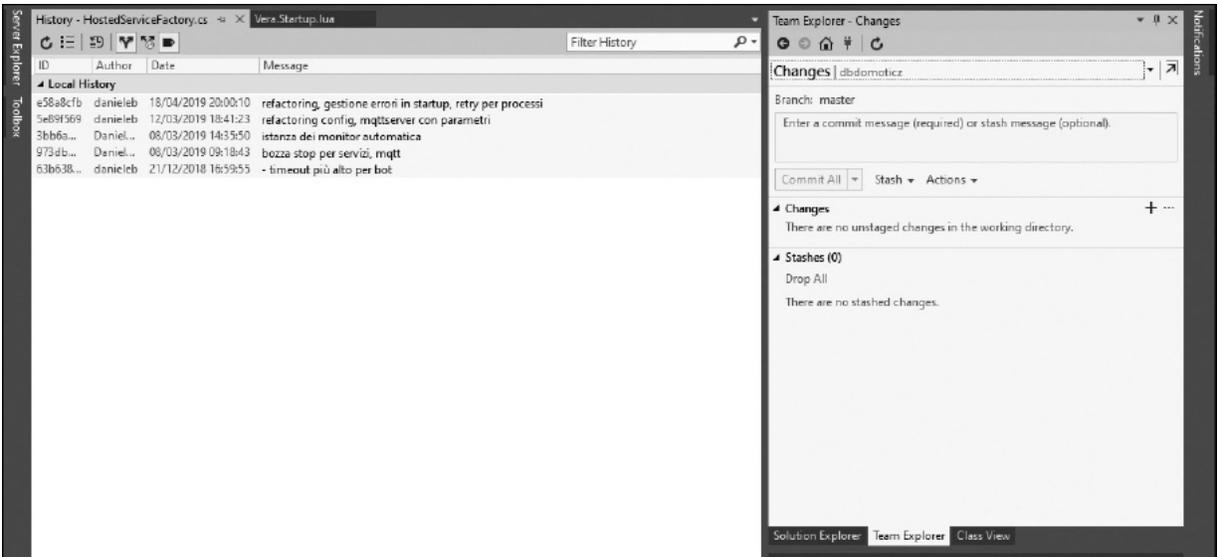


Figura 2.10 – Gestione del codice sorgente con TFS.

Compilare un progetto

La compilazione può essere effettuata dal menu “Build”, oppure eseguendo l’applicazione, anche con il debugger. Se preferiamo utilizzare una sequenza di tasti, la compilazione si può scatenare premendo `Ctrl+Shift+B`, mentre l’applicazione si può avviare premendo `Ctrl+F5`, o semplicemente `F5` per mandarla in debug.

In questa fase vengono tenute in conto le referenze tra i progetti, quindi Visual Studio provvede a compilare i progetti nell’ordine necessario, affinché quelli che hanno referenze trovino l’assembly già pronto.

Possiamo referenziare in un progetto A un progetto B, ma non è possibile fare anche il contrario. .NET non supporta le referenze circolari.

Come default, il risultato della compilazione di un’applicazione .NET Framework è contenuto nella directory “bin\”, posta sotto il progetto indicato come principale. All’interno di questo ci sono due directory, che si riferiscono a due profili che sono disponibili di default e su cui torneremo nella prossima sezione. Con .NET Core, invece, bisogna procedere a un deployment, cioè a un rilascio: in questo caso esiste un menu posto all’interno del progetto e chiamato “Publish”, che avvia un wizard. Poiché

la compilazione è cross platform, in questo caso gli artefatti generati sono di tipo differente. Il risultato sarà in una directory “Publish\” messa all’interno di quella “bin\”, contenuto in altre directory in base alla piattaforma di compilazione.

Gestire le configurazioni

Visual Studio supporta il concetto di configurazione: all’atto pratico, quando creeremo una nuova soluzione, troveremo già pronte due configurazioni, denominate “**Release**” e “**Debug**”. La prima è indicata per compilare (e gestire) la soluzione per la distribuzione, mentre la seconda è specifica per il debug, come il nome stesso suggerisce.

La modalità di debug andrebbe usata solo per lo sviluppo, perché non è ottimizzata per la produzione. Resta possibile utilizzare i simboli (che danno maggiori informazioni in caso di eccezioni) anche in release. In questo caso, dobbiamo generare gli stessi in fase di compilazione, agendo sulle proprietà di compilazione del progetto.

Queste modalità, in genere, sono attivabili da un apposito menu a tendina, disponibile sulla toolbar di Visual Studio e visibile nella [Figura 2.11](#).

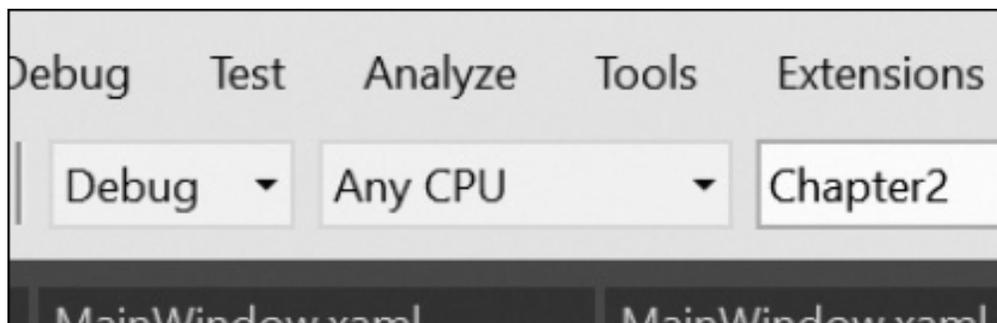


Figura 2.11 – La configurazione può essere scelta direttamente dalla toolbar.

Agendo sul valore di questo elenco, cambia il modo in cui viene compilata la soluzione. Per esempio, la modalità Debug non ha nessuna

ottimizzazione, mentre quella Release evita che vengano generate istruzioni utili al debug e che tutto sia ottimizzato per l'esecuzione.

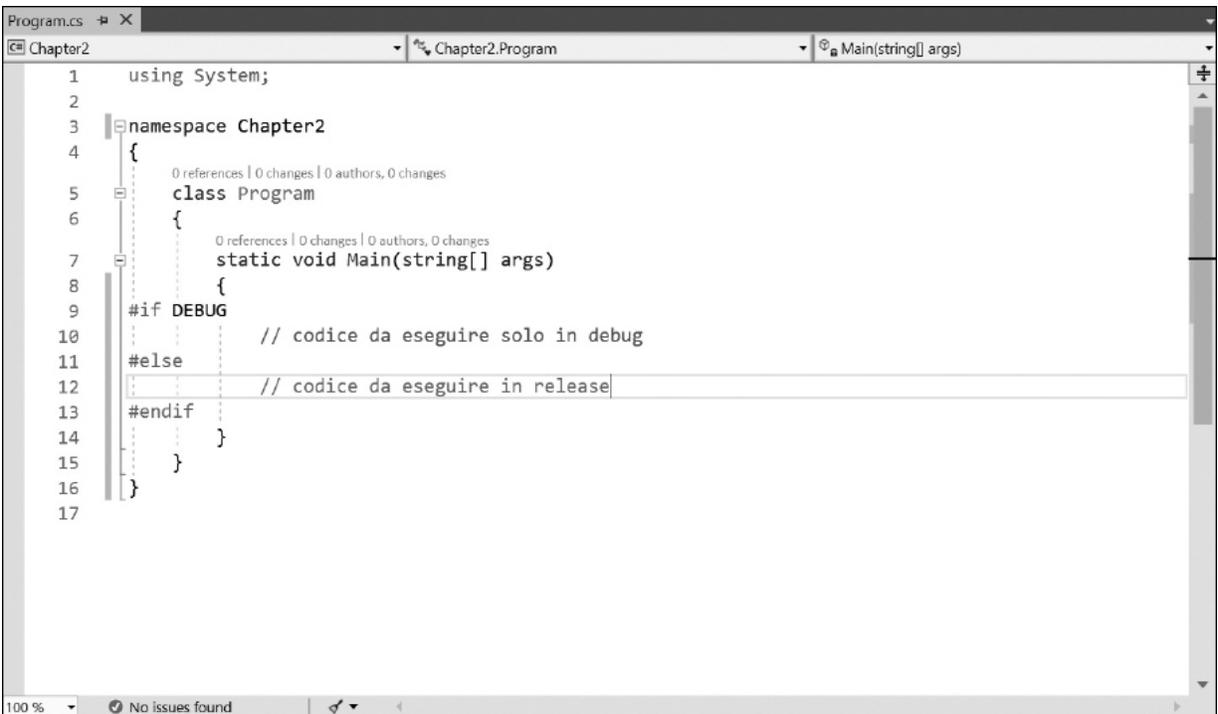
Possiamo anche scrivere codice che il compilatore, in base alla configurazione scelta, interpreta (e compila) solo se necessario. Questo ci consente di poter gestire il caricamento di informazioni particolari solo in fase di debug, piuttosto che differenziare la verbosità di una funzione di logging. Una dimostrazione in tal senso è visibile nell'[esempio 2.1](#).

Esempio 2.1

```
#if DEBUG // l'alternativa è RELEASE
    // codice da eseguire solo in debug
#else
    // altro codice
#endif
```

Rispetto a una soluzione di tipo differente, questa lavora a stretto contatto con il compilatore, quindi garantisce la certezza che, se la configurazione selezionata è differente, il codice non sarà nemmeno processato, come si può intuire guardando la colorazione che Visual Studio dà allo stesso, visibile nella [Figura 2.12](#).

Possiamo anche creare nostre impostazioni custom, per esempio per supportare un ambiente di staging, dove le applicazioni saranno testate prima della messa in produzione.

The image shows a screenshot of the Visual Studio code editor. The window title is 'Program.cs'. The file explorer on the left shows a project structure with 'Chapter2' containing a 'Program' class. The code in the editor is as follows:

```
1 using System;
2
3 namespace Chapter2
4 {
5     class Program
6     {
7         static void Main(string[] args)
8         {
9             #if DEBUG
10                // codice da eseguire solo in debug
11            #else
12                // codice da eseguire in release
13            #endif
14        }
15    }
16 }
17
```

The status bar at the bottom indicates '100%' zoom and 'No issues found'.

Figura 2.12 – Le direttive di compilazione consentono di generare il codice in base alla configurazione scelta.

Debug di un progetto

Quando l'applicazione diventa mediamente complessa, il debug è in grado di garantirci la possibilità di intervenire in maniera più proficua nell'analizzare e correggere eventuali problematiche. Se abbiamo già utilizzato C#, il concetto è già noto: si tratta di una particolare tecnica che consente di entrare nell'esecuzione del codice, consentendoci di analizzare lo stato della variabili e di intervenire in maniera proficua.

Usare il debugger

Visual Studio include un debugger: si tratta di un componente che consente di agganciarsi al processo che esegue il codice e, grazie alla presenza dei simboli, di mostrare il flusso all'interno di Visual Studio. Questo ci consente di poter intervenire durante l'esecuzione dell'applicazione, per esempio per individuare le cause che portano a un'anomalia della stessa.

A prescindere dalla configurazione corrente, è sempre possibile mandare un'applicazione in debug premendo il tasto F5.

Anche se è molto potente, usare il debugger non sostituisce tecniche di analisi del codice o di test dello stesso. Si tratta di un'arma in più, quando abbiamo necessità di poter verificare il flusso di un particolare blocco di codice, così da vedere come lo stesso viene eseguito. Nella [Figura 2.13](#) è visibile la schermata di selezione del processo, raggiungibile attraverso il menu “Debug”, alla voce “Attach to process”.

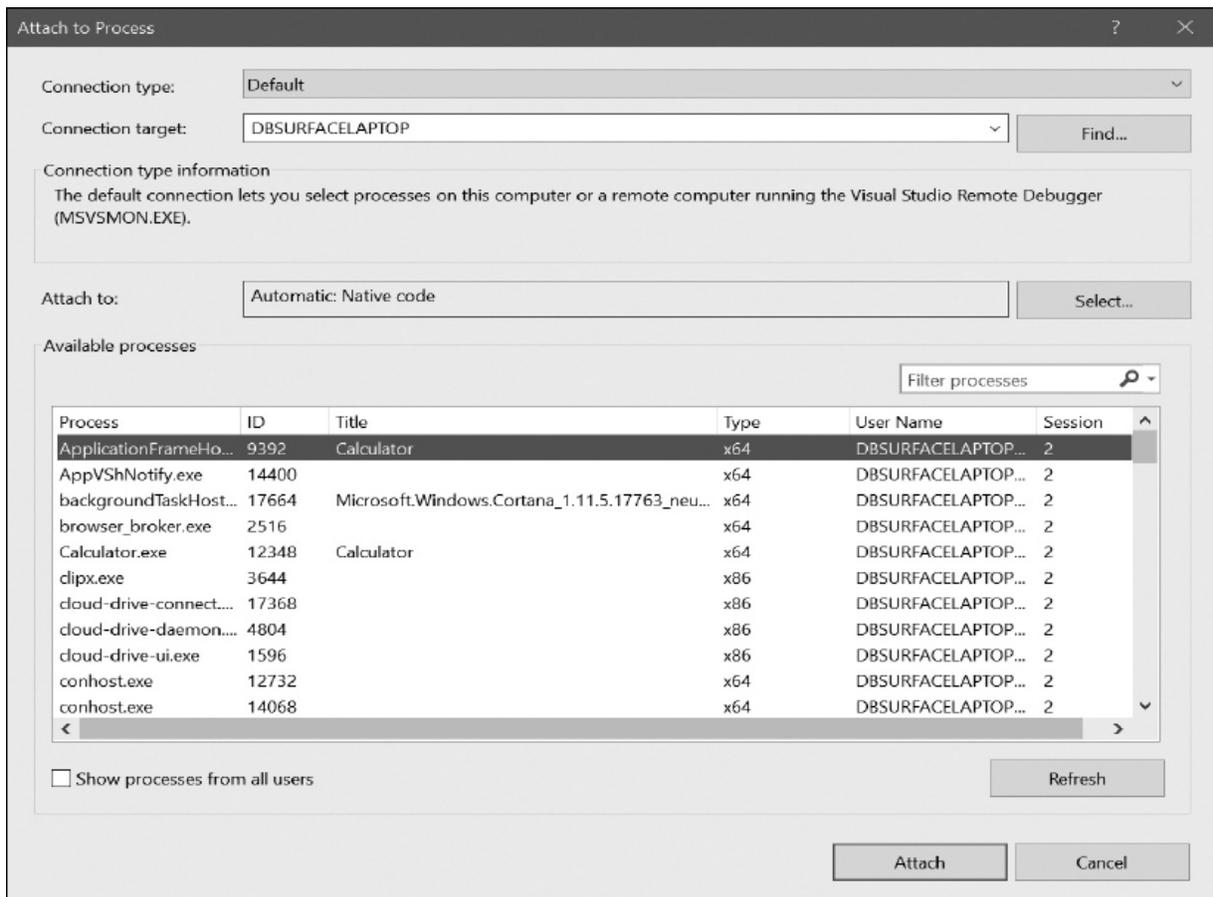


Figura 2.13 – Selezione del processo su cui effettuare il debug.

È possibile far partire l'applicazione in debug, oppure agganciarsi a un'applicazione già in esecuzione. Si può controllare al meglio il flusso sfruttando due caratteristiche, note con il nome di breakpoint e watch. Inoltre, le chiamate che hanno portato al codice corrente sono visibili in

un'apposita finestra, di norma ancora in basso, che prende il nome di **Call Stack Window**. All'interno di quest'ultima troveremo le chiamate che hanno originato il flusso attuale: questa è una funzionalità molto comoda quando un dato metodo è richiamato da più punti dell'applicazione e vogliamo capire come siamo arrivati a un determinato punto della stessa.

Breakpoint e watch

Come comportamento predefinito il debugger non si blocca, se non in caso di eccezioni. I breakpoint consentono di fissare un punto specifico, al cui raggiungimento il debugger si ferma, in attesa che l'utente possa verificare lo stato di una particolare variabile all'interno del codice. Per impostare un breakpoint è sufficiente cliccare con il mouse a fianco della riga, fino a che un pallino di colore rosso non compare, come è possibile notare nella [Figura 2.14](#). Si può anche impostare un breakpoint premendo sul tasto F9, dopo che ci siamo posizionati sulla riga interessata.

Possiamo anche impostare un breakpoint selettivo, per cui il debugger si fermerà solo al verificarsi di una determinata situazione. Possiamo visualizzare in ogni momento i breakpoint attivi, agendo attraverso il menu "Debug".

Se passiamo sopra a una variabile, viene visualizzato in automatico il relativo valore, attraverso quello che viene chiamato **quick watch**. A partire da questa versione di Visual Studio, è anche possibile tenere in primo piano il watch, utilizzando l'apposita funzione di pin, come possiamo notare nella [Figura 2.15](#).

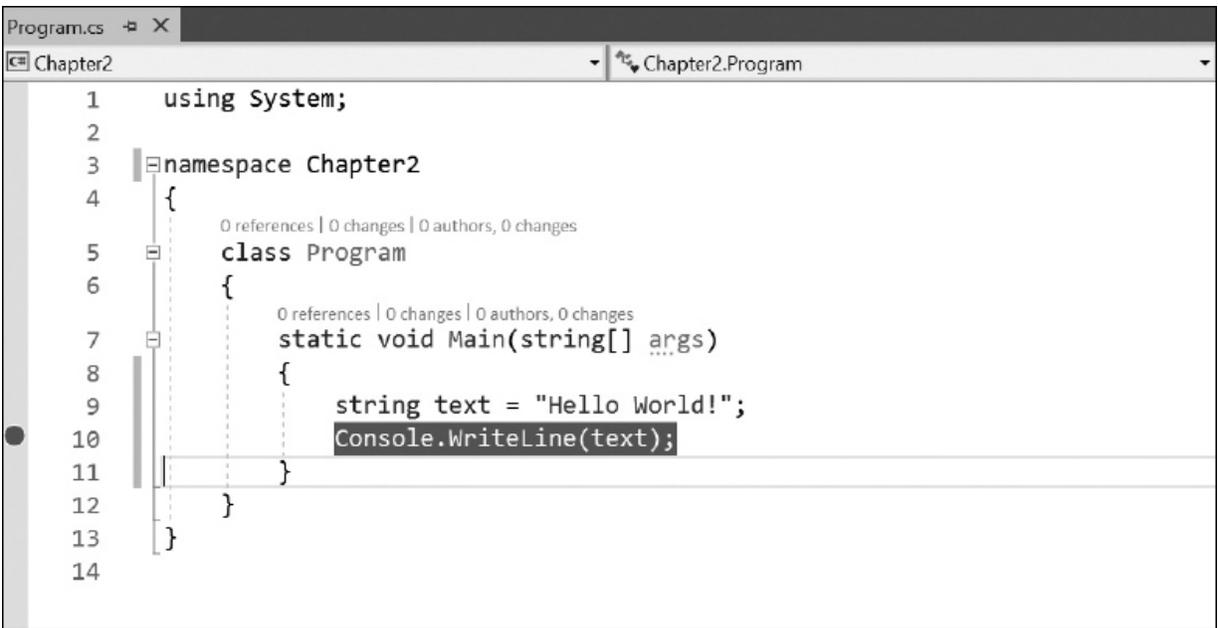


Figura 2.14 – Aggiungere un breakpoint.

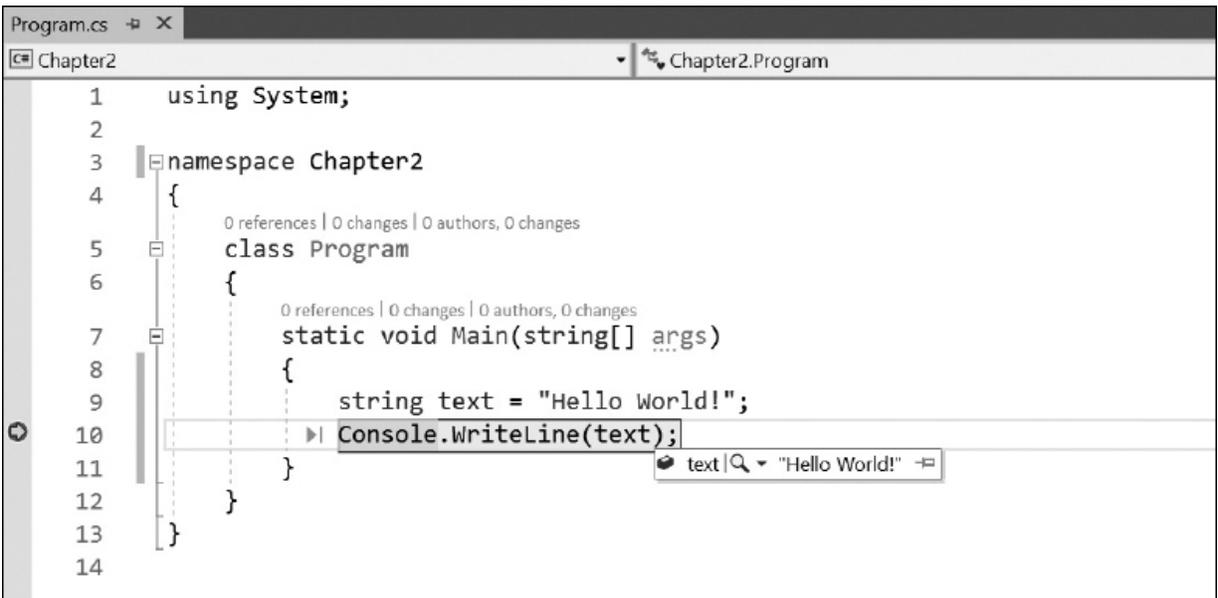


Figura 2.15 – Un quick watch con pin.

Nel caso in cui decidessimo, dall'apposito menu che compare facendo click con il tasto destro sulla variabile, di scegliere la voce "Add Watch", il valore dell'oggetto verrà mostrato, per tutto il suo ciclo di vita, dentro un'apposita finestra, denominata **Watch Window**. Questo è utile quando è

necessario tenere sott'occhio il valore di una data variabile durante tutto il ciclo di vita dell'applicazione o invocare delle funzioni, comprese le interrogazioni con LINQ. Nella maggior parte dei casi, comunque, finiremo per utilizzare molto di più la funzione di quick watch, che consente di interrogare i membri di un oggetto e controllare le relative proprietà in maniera molto semplice. In Visual Studio sono anche inclusi dei viewer specializzati, per visualizzare grandi quantità di testo, file in formato XML o JSON.

Alcuni oggetti sono mostrati in automatico, attraverso l'**Autos Window**. Si tratta di una finestra che appare in automatico in debug e mostra le ultime variabili utilizzate durante l'esecuzione. Ne esiste anche un'altra, denominata **Local Window**, che invece mostra le variabili utilizzate nel contesto corrente (per esempio, all'interno di una routine).

Intellitrace e historical debug

Visual Studio 2010 ha introdotto l'Intellitrace, che, come avviene ormai da diverse versioni anche in Visual Studio 2019, è stato ulteriormente migliorato. Si tratta di una tecnologia che consente di creare una funzione del tutto identica a quella di una scatola nera, generalmente posta all'interno di un aereo: consente di registrare tutto quello che è accaduto, rendendo possibile l'**historical debug**, cioè il debug a posteriori.

Attraverso la registrazione di informazioni importanti, durante la normale esecuzione di un'applicazione diventa possibile mandare in esecuzione il codice e capire cosa è successo in quel particolare momento.

Questo consente, per esempio, di ricevere un report da un tester e riprodurre in locale il bug, piuttosto che comprendere meglio le cause di un problema, che a volte analizzando il call stack o il log di un bug non possono essere immediatamente riprodotte.

Questa funzionalità è specifica di Visual Studio Ultimate, la versione di punta.

Refactoring

Può capitare che, durante la scrittura di codice, si presti poca attenzione ad alcuni dettagli e si decida di effettuare un controllo successivo per

intervenire sugli stessi, per esempio, per modificare il nome di una classe o di un membro.

In questi casi si effettua quello che viene chiamato **refactoring**, che consiste nel sistemare il codice a posteriori. Con il refactoring cambiamo il funzionamento interno del nostro codice, migliorandolo, senza che lo stesso subisca un cambiamento esterno, che gli impedisca di continuare a funzionare come in precedenza.

In C# c'è un ampio supporto a queste funzionalità, che si attivano in base al testo selezionato nell'editor, attraverso il pulsante destro, alla voce "Quick Actions":

- ❑ **Rename:** rinomina un membro o una variabile, cercando i riferimenti all'interno del codice, per assicurare che lo stesso non smetta di funzionare;
- ❑ **Extract Method:** estrae un metodo, dato un codice selezionato. È utile quando è necessario estrarre una parte di codice, che prima invece era contenuto esclusivamente all'interno di un metodo, per creare un metodo a sé stante, che possa essere riutilizzato in altri contesti;
- ❑ **Extract Interface:** consente di creare un'interfaccia a partire dai membri pubblici di una classe;
- ❑ **Encapsulate Field:** trasforma un campo privato in una proprietà pubblica, all'interno della quale viene incapsulato il campo originale;
- ❑ **Remove Parameters:** rimuove i parametri di una funzione, alterandone la firma;
- ❑ **ReorderParameters:** riordina i parametri di una funzione, modificandone la firma.

Tutte queste azioni non comportano soltanto la modifica del codice, ma anche un adeguamento di quelle parti che fanno riferimento a ciò che viene modificato dalla nostra azione, assicurando che il nostro codice non smetta di funzionare.

Molto interessante (e raggiungibile nello stesso modo) è anche il menu “**Organize Using**”, che consente di organizzare al meglio i blocchi using:

- ❑ **Remove Unnecessary Usings**: rimuove i riferimenti ai namespace che non sono sfruttati nella classe;
- ❑ **Sort Usings**: ordina i namespace referenziati nella classe in ordine alfabetico;
- ❑ **Remove And Sort Usings**: applica entrambe le azioni riportate.

Altre funzionalità di refactoring, come “**Generate Method Stub**”, sono invece direttamente accessibili in contesti particolari: questo, per esempio, è un aiuto che ci viene offerto quando aggiungiamo un riferimento a un tipo (o a un membro) che non viene riconosciuto, consentendoci di crearne una implementazione vuota, che consenta al codice di compilare correttamente e a noi di continuare a completarlo. Basta posizionarsi sul codice per avere un pulsante, alla cui pressione appare un menu come quello mostrato nella [Figura 2.16](#).

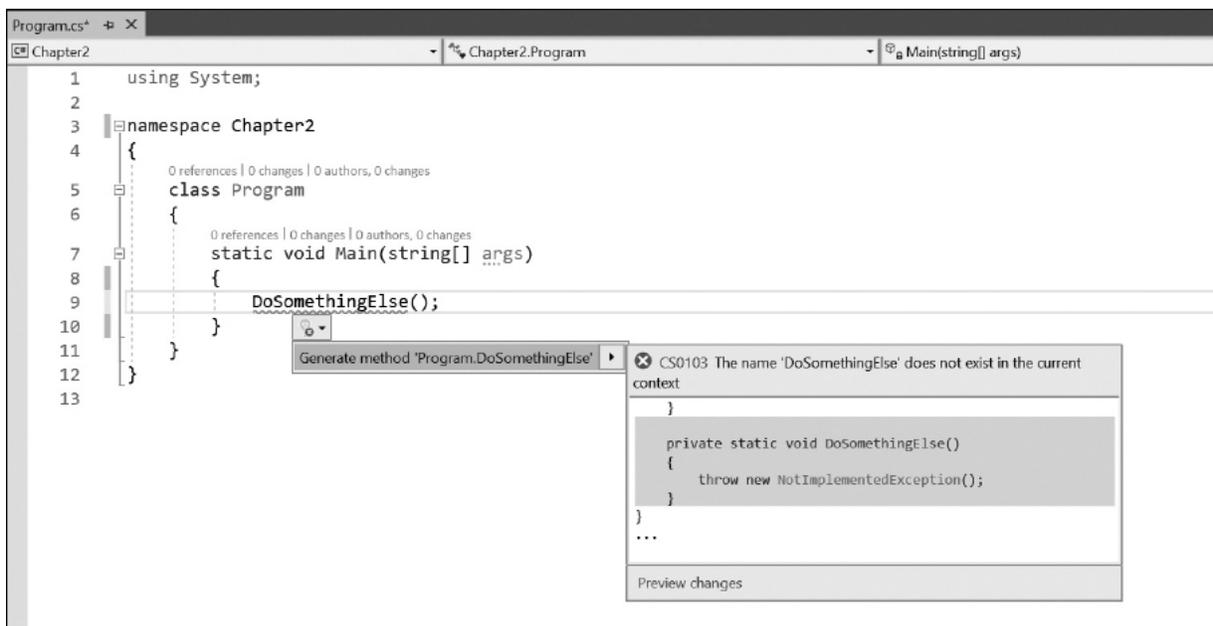


Figura 2.16 – La funzione di generazione automatica di un metodo che non esiste.

Seppure non paragonabile a quello di add-in commerciali nati appositamente per questo scopo (come ReSharper di JetBrains), il supporto al refactoring incluso in Visual Studio 2015, quando si utilizza C# come linguaggio, può aiutare a svolgere meglio le normali operazioni quotidiane di rifinitura del codice.

Grazie all'introduzione di Roslyn, il nuovo compilatore offerto come servizio, le funzionalità di refactoring sono state ulteriormente migliorate, in quanto Visual Studio può ora compilare più facilmente il codice mentre scriviamo, garantendoci, tra l'altro, anche una precisione migliore dell'Intellisense in fase di autocompletion.

In tal senso, una nuova funzionalità è quella offerta dagli analyzer: si tratta della possibilità di aggiungere estensioni a Visual Studio, sotto forma di codice, che vanno a controllare il codice (mentre lo stiamo scrivendo), di aggiungere funzionalità che consentano di forzare stili di scrittura del codice (per esempio il case) e fornire anche suggerimenti in fase di scrittura del codice stesso, come avviene per Visual Studio stesso, tramite le funzionalità che suggeriscono di aggiungere, per esempio, una direttiva di import.

Tutte queste feature, comunque, sono più interessanti per gli sviluppatori di plug-in, ma conoscerne l'esistenza può sicuramente tornare utile a tutti. Maggiori informazioni sono disponibili nella documentazione ufficiale su <http://aspit.co/a6t>.

Novità di Visual Studio 2019

Visual Studio 2019 introduce alcune migliorie in generale, con diverse novità nell'ambito della produttività e del lavoro in team. In particolare, oltre alla già citata nuova *Start Page*, le novità principali sono le seguenti.

- ❑ **Ricerca migliorata:** una nuova funzionalità che va a sostituire il *Quick Launch* e ora include anche un elenco delle shortcut associate a una certa funzionalità.

- ❑ **One-click code cleanup:** grazie a un nuovo menu, diventa possibile ripulire il codice in un colpo solo, anche basandosi sulle impostazioni contenute nei file in formato *editorConfig*, di cui parleremo a breve.

- ❑ **Migliorie al debugger:** aggiunta la possibilità, tra le altre cose, di cercare all'interno delle *Watch Window*.
- ❑ **Visual Studio Live Share:** si tratta di un servizio che consente di lavorare in condivisione, in tempo reale sul codice. Disponibile come add-on per le versioni precedenti, il servizio è ora integrato direttamente in Visual Studio.
- ❑ **Supporto a .NET Core 3:** la nuova release di .NET Core necessita dell'ultima versione di Visual Studio.
- ❑ **Visual Studio IntelliCode:** come abbiamo già accennato, è una nuova funzionalità che va a completare l'Intellisense che conosciamo. Visual Studio 2019 la include in automatico, mentre in precedenza si trattava di un add-on da installare.

Supporto a editorConfig

In particolare, il supporto a editorConfig, introdotto da Visual Studio 2017, è ora stato integrato all'interno di una funzionalità nota come One-click code cleanup, che in realtà prende le proprie impostazioni da diverse sorgenti, come per esempio le impostazioni personali di Visual Studio. A cosa serve questa funzionalità? A formattare con spazi o tab le indentazioni, piuttosto che a forzare un certo stile di scrittura, forzando le parentesi graffe a capo e così via. editorConfig è un formato aperto di file documentato a questo indirizzo: <https://editorconfig.org/>. In Visual Studio serve a fare in modo che un certo progetto (e quindi un team) possa condividere le stesse regole di formattazione del codice.

Perché funzioni, è necessario creare nella root della solution un nuovo file di nome `.editorconfig` (sì, senza nome e solo con l'estensione). Una serie di regole sono contenute nell'esempio che segue.

Esempio 2.1

```
root = true
[*.{cs,vb}]
indent_style = tab
```

```
indent_size = 4
trim_trailing_whitespace = true

[*.*cs]
csharp_new_line_before_open_brace = methods
```

Nell'esempio precedente abbiamo forzato lo stile delle indentazioni impostando i tab. Questo vuol dire che, a prescindere dalle impostazioni locali di Visual Studio, all'interno dei file verranno sempre utilizzati i tab per indentare, e non gli spazi. Questo comportamento verrà applicato in fase di formattazione o durante l'uso del *One-click code cleanup*, che si attiva con il tasto all'interno della finestra con il codice. Poiché le opzioni configurabili all'interno di questo file sono tantissime e variano in base al linguaggio (si possono applicare a file HTML, JavaScript, tanto quanto a file C# o VB), rimandiamo alla documentazione ufficiale per approfondimenti.

Conclusioni

Visual Studio è un ambiente vero e proprio, all'interno del quale, come sviluppatori di C#, passeremo gran parte del nostro tempo. Questa nuova versione si differenzia dalle altre grazie alle migliorie apportate un po' ovunque, dal Text Editor fino all'Intellisense, passando per l'introduzione di nuove funzionalità, come l'Intellitrace.

Questo capitolo non vuole essere una guida esaustiva a Visual Studio, che è molto complesso e tutt'altro che semplice da spiegare in poche pagine, quanto una prima introduzione, che mostri come poter sfruttare al meglio l'ambiente per iniziare a lavorare.

Prima di addentrarci maggiormente nello sviluppo di applicazioni basate su C#, è necessario che esaminiamo brevemente le caratteristiche del linguaggio. Nel prossimo capitolo inizieremo a dedicare uno sguardo più approfondito al linguaggio, cominciando dalla sintassi di base.

Introduzione al linguaggio e sintassi di base

Basato su uno standard ECMA, il linguaggio C# è stato introdotto nel 2002 per proporre un linguaggio nato con il .NET Framework e in grado di offrire l'evoluzione in chiave managed di linguaggi già esistenti come C e C++. Infatti, C# utilizza numerose caratteristiche e notazioni sintattiche proprie di C e C++ relativamente alle istruzioni, alle espressioni e agli operatori ma, rispetto ai suoi predecessori, include notevoli innovazioni e facilitazioni che riguardano principalmente la gestione della memoria e dei tipi.

Nonostante il numero ridotto di parole chiave, la sintassi di C# è altamente espressiva e, di conseguenza, abbastanza semplice da imparare. Inoltre, essendo basata sull'uso delle parentesi graffe, risulta immediatamente riconoscibile per chiunque abbia familiarità con i cosiddetti linguaggi "C-like" come, per esempio, Java, JavaScript e lo stesso C++. In genere, i programmatori che hanno dimestichezza con uno di questi linguaggi sono in grado di iniziare a sviluppare utilizzando C# senza grandi sforzi, dopo un breve periodo di apprendimento.

In questo capitolo cominciamo a vedere il linguaggio più da vicino, introducendo le regole sintattiche di base.

Introduzione al linguaggio

Per iniziare a parlare di C# e della sua sintassi, facciamo riferimento a un semplice esempio, allo scopo di fissare i concetti di base. L'esempio classico che in genere viene usato per presentare un linguaggio e al quale, anche in

questo caso, facciamo ricorso per non tradire la consuetudine, è il cosiddetto “Hello World”.

Esempio 3.1

```
using System;

namespace ASPItalia.Books.Chapter3
{
    class HelloWorld
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World");
            Console.ReadLine();
        }
    }
}
```

L'[esempio 3.1](#) riporta la porzione di codice relativa a una semplice applicazione che si limita a scrivere a video la stringa “Hello World”. Il codice proposto si compone di una serie di **dichiarazioni** e **istruzioni** (dette anche **statement**), ciascuna delle quali include uno o più vocaboli speciali come `class`, `static`, `void` detti **keyword** o anche **parole chiave**.

Alcuni concetti anticipati nell'[esempio 3.1](#) vengono affrontati in modo specifico nei prossimi capitoli del libro. In particolare, il concetto di classe verrà spiegato nel corso del prossimo capitolo.

C# contiene un certo numero di keyword, riportate nella [Tabella 3.1](#), che sono termini riservati e predefiniti che hanno un significato specifico per il compilatore e non possono essere utilizzate come identificatori per definire variabili, a meno che non abbiano il prefisso `@`. Per esempio, `@if` è un identificatore valido, mentre `if` non lo è, in quanto è una parola chiave.

Tabella 3.1 – Parole chiave riservate del linguaggio

<code>abstract</code>	<code>as</code>	<code>base</code>	<code>bool</code>
<code>break</code>	<code>byte</code>	<code>case</code>	<code>catch</code>
<code>char</code>	<code>checked</code>	<code>class</code>	<code>const</code>
<code>continue</code>	<code>decimal</code>	<code>default</code>	<code>delegate</code>
<code>do</code>	<code>double</code>	<code>else</code>	<code>enum</code>

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	safe	sbyte	sealed
short	sizeof	stackalloc	static
string	struct	switch	this
throw	true	try	typeof
uint	ulong	unchecked	unsafe
ushort	using	virtual	void
volatile	while		

Oltre alle keyword riservate, il linguaggio include una serie di parole chiave contestuali che assumono uno speciale significato solo in determinati contesti d'utilizzo e che riportiamo nella [Tabella 3.2](#). Dato che le parole chiave contestuali non sono riservate, possono essere utilizzate per i nomi degli identificatori. Consigliamo tuttavia di evitare di usare le keyword contestuali per definire variabili e quant'altro, poiché il codice potrebbe risultare di difficile lettura e si potrebbero verificare errori gravi non facilmente rilevabili.

Tabella 3.2 – Parole chiave contestuali

add	alias	ascending	async
await	descending	dynamic	from
get	global	group	into
join	let	orderby	partial
remove	select	set	value
var	where	yield	

Alcune delle parole chiave contestuali riportate nella [Tabella 3.2](#) sono state introdotte specificatamente per supportare alcune delle caratteristiche del linguaggio, in particolare la type inference (`var`), i tipi dinamici (`dynamic`) e le query expression di LINQ (per esempio, `from`, `group`, `join`, `orderby`, `select`,

where e altre). Avremo modo di scoprire queste caratteristiche del linguaggio nei capitoli a seguire.

Prendendo in considerazione tutte le parole chiave elencate nelle [Tabelle 3.1](#) e [3.2](#), è comunque interessante sottolineare come le keyword introdotte con C# 5 siano solo due, ovvero `async` e `await`, che trovano impiego negli scenari di esecuzione asincrona del codice, introdotti con il .NET Framework 4.5. Il loro scopo e utilizzo sarà spiegato nello specifico nel corso del [Capitolo 9](#). Né C# 6, né C# 7, né C# 8 hanno introdotto nuove keyword, segno di una maturità ormai raggiunta dal linguaggio.

Dall'[esempio 3.1](#) emerge in modo evidente uno degli aspetti sintattici che caratterizzano maggiormente il linguaggio C#, cioè l'utilizzo delle parentesi graffe. Qualsiasi gruppo di istruzioni (detto anche **blocco**), il corpo delle routine e degli oggetti e, in generale, il contenuto delle varie sezioni di codice è sempre racchiuso all'interno di una coppia di parentesi graffe. Ad una parentesi aperta deve sempre corrispondere una parentesi chiusa, anche in caso di annidamento; diversamente, otteniamo un errore in fase di compilazione.

Come un blocco di istruzioni è sempre racchiuso in una coppia di parentesi graffe, così ogni singola istruzione del codice è conclusa da un ";" (punto-e-virgola), che funge da carattere separatore. La presenza del *punto-e-virgola* permette tra l'altro di scrivere più istruzioni consecutivamente sulla stessa riga, anche se questo può comportare una perdita di leggibilità del codice, oppure dover suddividere una stessa istruzione su più righe. L'omissione del punto-e-virgola al termine di un'istruzione produce sempre un errore in fase di compilazione.

Un altro aspetto importante da sottolineare è il fatto che C# è un linguaggio **case sensitive**. Un diverso utilizzo delle lettere maiuscole e minuscole per keyword, nomi di variabili, di routine e di oggetti, comporta inevitabilmente un errore in fase di compilazione. Nomi uguali con "case" diversi corrispondono sempre a variabili e oggetti differenti (`myVar` è diverso da `myvar`). In relazione all'[esempio 3.1](#), inserendo al posto di `Console.WriteLine` l'istruzione `console.writeline`, otteniamo un errore in fase di compilazione, in quanto non viene rispettata la formattazione corretta per il nome della routine.

[Uso dei commenti](#)

Una pratica che ogni programmatore dovrebbe attuare durante le attività di sviluppo riguarda l'inserimento di commenti a corredo del codice per migliorare la comprensione e la leggibilità.

In C# esistono due modalità per inserire commenti all'interno del codice:

- ❑ **commento su singola riga:** il testo di commento viene posto dopo una coppia di slash "//" ([esempio 3.2](#)). Tutto il testo a seguire nell'ambito della riga viene considerato un commento e ignorato dal compilatore;
- ❑ **commento ripartito su più righe:** il testo di commento è inserito tra la sequenza "/*" (slashasterisco) e "*/" (asterisco-slash) e può essere suddiviso su più righe ([esempio 3.2](#)).

Esempio 3.2

```
// Testo di commento su singola riga  
Console.WriteLine("Hello, World!"); // Commento dopo il codice  
/* Testo di commento  
   ripartito su più righe */
```

La prima modalità è indicata nelle situazioni in cui vogliamo aggiungere un commento dopo un'istruzione di codice oppure introdurre un testo esplicativo su una singola riga. Invece, la seconda modalità si rivela molto comoda per commentare intere sezioni di codice da mascherare al compilatore oppure per introdurre un testo esplicativo ripartito su più righe.

Tipi di base

A ciascuna variabile in C# corrisponde un tipo di dato che, come abbiamo visto nel primo capitolo, può essere di valore o di riferimento. Oltre ai tipi dichiarati dallo sviluppatore (e, in particolare, le classi che vedremo in dettaglio nel prossimo capitolo) e ai tipi forniti nella Base Class Library (anche detta BCL), il linguaggio è dotato di un insieme di tipi di dato intrinseci, comunemente chiamati **primitive**, a cui fanno riferimento molte delle parole chiave elencate nella [Tabella 3.1](#).

Le primitive sono tutti tipi di valore (per esempio, `int` oppure `bool`), fatta eccezione per `object` e `string` che sono tipi di riferimento e che quindi non

contengono direttamente un valore effettivo, ma puntano a un'area di memoria gestita nell'ambito del managed heap. Come possiamo notare nell'[esempio 3.3](#), per azzerare il riferimento di una stringa o di un oggetto in genere, in C# dobbiamo usare la parola chiave `null`.

Esempio 3.3

```
int x = 2;      // Intero che contiene 2
bool y = true; // Valore booleano che vale true
string s1 = "Hello"; // Punta ad un'area di memoria
string s2 = null; // Non punta a nulla
object obj1 = new object(); // Crea un nuovo oggetto
object obj2 = null; // Non punta a nulla
```

Le primitive di C# sono elencate nella [Tabella 3.3](#). A ciascuna di esse corrisponde un tipo contenuto in .NET, del quale la parola chiave rappresenta semplicemente un alias.

Tabella 3.3 – Primitive di C#

Parola chiave	Tipo di dato	Valore / Riferimento
<code>bool</code>	<code>System.Boolean</code>	Tipo di valore
<code>byte</code>	<code>System.Byte</code>	Tipo di valore
<code>char</code>	<code>System.Char</code>	Tipo di valore
<code>decimal</code>	<code>System.Decimal</code>	Tipo di valore
<code>double</code>	<code>System.Double</code>	Tipo di valore
<code>float</code>	<code>System.Single</code>	Tipo di valore
<code>int</code>	<code>System.Int32</code>	Tipo di valore
<code>long</code>	<code>System.Int64</code>	Tipo di valore
<code>object</code>	<code>System.Object</code>	Tipo di riferimento
<code>sbyte</code>	<code>System.SByte</code>	Tipo di valore
<code>short</code>	<code>System.Int16</code>	Tipo di valore
<code>string</code>	<code>System.String</code>	Tipo di riferimento
<code>uint</code>	<code>System.UInt32</code>	Tipo di valore
<code>ulong</code>	<code>System.UInt64</code>	Tipo di valore

ushort	System.UInt16	Tipo di valore
--------	---------------	----------------

Non tutti i tipi indicati nella [Tabella 3.3](#) sono CLS-compliant. In particolare, `sbyte`, `ushort`, `ulong` e `uint`, che corrispondono rispettivamente ai tipi della BCL `System.SByte`, `System.UInt16`, `System.UInt32` e `System.UInt64`, non rispettano la Common Language Specification e ad essi, in genere, dovrebbero essere preferiti i tipi corrispondenti compatibili con la CLS (`byte`, `short`, `int` e `long`).

Namespace

Quando uno sviluppatore definisce un tipo, si trova nella necessità di assegnargli un nome che sia non solo significativo ma anche univoco. Questo vincolo è necessario per permettere al compilatore di riferirsi al tipo senza ambiguità. D'altra parte, assegnare nomi lunghi e complessi tali da garantire l'univocità richiesta può rappresentare un problema e minare non poco la leggibilità del codice.

Per questo motivo, in C# è possibile definire i **namespace** (detti anche **spazi dei nomi**), ovvero contenitori logici che hanno il duplice scopo di raggruppare e organizzare i tipi dichiarati dallo sviluppatore e di fornire un meccanismo per semplificare la nomenclatura.

I namespace rappresentano un meccanismo di organizzazione gerarchica del codice, in quanto possono includere, a loro volta, altri namespace. I nomi dei namespace annidati sono separati tra loro da un punto. Possiamo dichiarare in file e assembly diversi tipi appartenenti allo stesso namespace.

Per assegnare un tipo a un namespace, dobbiamo inserire la sua dichiarazione nell'ambito di un blocco di codice, contrassegnato con la parola chiave `namespace` seguita da un nome identificativo ([esempio 3.4](#)).

Esempio 3.4

```
namespace ASPItalia.Books.Chapter3
{
    // Dichiarazione di un tipo appartenente al
    // namespace ASPItalia.Books.Chapter3
    class HelloWorld
    {
        // ...
    }
}
```

Il nome completo del tipo dichiarato nell'[esempio 3.4](#) è: `ASPItalia.Books.Chapter3>HelloWorld`. Oltre a poter dichiarare altri tipi nello stesso namespace, come per esempio: `ASPItalia.Books.Chapter3.AnotherType`, possiamo definire anche tipi omonimi in altri namespace, come `System>HelloWorld`.

Il nome di un tipo viene connotato dal nome del namespace al quale appartiene; di conseguenza, per riferirsi ad un tipo, dobbiamo sempre indicare l'intera gerarchia dei namespace. Peraltro, se un namespace è usato frequentemente nel codice, può risultare assai scomodo ripetere ogni volta il nome del namespace prima del nome dei tipi. Utilizzando la parola chiave `using`, in `C#` è possibile inserire un riferimento a un namespace in modo tale da poter omettere l'indicazione del suo nome davanti al nome dei tipi, come mostrato nell'[esempio 3.5](#).

Esempio 3.5

```
using ASPItalia.Books.Chapter3;

// Il nome del namespace può essere omissso
HelloWorld hello1;

// In alternativa possiamo indicare il nome completo
// comprensivo del namespace
ASPItalia.Books.Chapter3>HelloWorld hello2;
```

Quando un tipo viene dichiarato senza indicare alcun namespace, viene aggiunto al cosiddetto namespace globale.

`C# 6` introduce una novità, cioè quella dell'uso dello `static using`. Questa caratteristica consente di evitare l'uso di un riferimento esplicito a un tipo durante l'invocazione di un metodo statico. Approfondiremo questo aspetto nel prossimo capitolo, quando parleremo di `Extension Method`.

Dichiarazione di variabili

`C#` è un linguaggio per il quale la dichiarazione delle variabili e la definizione dei tipi rappresentano, senza dubbio, due aspetti fondamentali e non prescindibili nell'ambito della programmazione.

Le **variabili** sono entità che contengono o puntano ai dati utilizzati nell'ambito di un blocco di codice. La **dichiarazione** è il tipo d'istruzione che permette di creare una nuova variabile. Questa può essere inserita praticamente ovunque, ma la validità della variabile creata, in genere, si esaurisce al termine del blocco in cui essa viene definita e usata (comunemente si dice che la variabile esce dal suo **scope**, ovvero dal suo ambito di validità).

Le variabili relative ai tipi di valore rilasciano le informazioni contenute in modo deterministico al termine della sezione di codice di appartenenza. Per le variabili relative ai tipi di riferimento, vale un discorso diverso: il rilascio non è deterministico, in quanto, in questo caso, le informazioni vengono gestite nel managed heap dal Garbage Collector, che, come abbiamo visto nel primo capitolo, decide in piena autonomia quando liberare la memoria non più in uso.

Già nell'[esempio 3.3](#) sono riportati alcuni casi di dichiarazione di variabili e oggetti. In generale, al nome del tipo segue il nome dell'identificatore, eventualmente seguito a sua volta da un'assegnazione a un valore costante o a un'altra variabile.

Possiamo dichiarare più variabili consecutivamente nell'ambito della stessa istruzione, indicando solamente una volta il tipo e separando i vari identificatori con una virgola, come mostrato nell'[esempio 3.6](#).

Esempio 3.6

```
int i = 1, j = 2, k = 3;  
string x, y, z;
```

Una variabile che non può essere modificata in fase di esecuzione si dice **costante** e viene contrassegnata con la parola chiave `const`. Una costante deve essere obbligatoriamente inizializzata a “compile-time” e non può essere modificata, in seguito, tramite un'operazione di assegnazione ([esempio 3.7](#)).

Esempio 3.7

```
const double PI = 3.1416;  
const string HELLO = "Hello World";
```

Nel caso di variabili relative a tipi di riferimento, l'assegnazione associata alla dichiarazione può riguardare un'istanza preesistente oppure una nuova istanza, generata mediante un'operazione di costruzione specificando la parola chiave `new` ([esempio 3.8](#)). Una variabile relativa a un tipo di riferimento a cui non venga assegnata alcuna istanza, viene inizializzata implicitamente con `null` (ovvero non punta ad alcun oggetto).

Esempio 3.8

```
object x = new object();    // Punta ad una nuova istanza
object y = null;           // Punta a null
object z;                  // Punta a null
object obj = x;            // obj si riferisce alla stessa istanza di x
```

C# supporta la **type inference**, ossia la capacità di ricavare il tipo di una variabile dalla sua espressione di inizializzazione. La type inference può essere applicata unicamente nella dichiarazione di variabili locali interne a una routine. In tali casi, il tipo della variabile viene ricavato dall'espressione usata per l'inizializzazione; se l'espressione viene omessa, otteniamo un errore in fase di compilazione. Questo tipo di dichiarazione richiede l'uso della parola chiave `var` da inserire al posto del tipo della variabile da inizializzare.

La type inference può essere applicata sia nel caso di variabili il cui tipo è di valore (in particolare i tipi primitivi), sia nel caso di variabili il cui tipo è di riferimento (classi). Nel caso dei tipi di riferimento non è ammessa l'inizializzazione a `null`, ma solo all'istanza di una classe.

Esempio 3.9

```
// Dichiarazione di una variabile intera senza e con type inference
int i = 1;    // i è di tipo intero
var j = 1;    // j è di tipo intero (type inference)

// Dichiarazione di una stringa senza e con type inference
string x = "Hello World";    // x è di tipo string
var y = "Hello World";    // y è di tipo string (type inference)
```

L'[esempio 3.9](#) mostra alcuni semplici casi d'utilizzo. In questo esempio le variabili `j` e `y` sono legate alla loro dichiarazione originale; non possono cioè

variare di tipo. Il codice MSIL, creato in fase di compilazione, è il medesimo che otteniamo normalmente quando indichiamo per esteso il tipo della variabile da dichiarare (come nel caso di `i` e `x`).

Espressioni e operatori

Come tutti i linguaggi, anche C# permette di costruire espressioni che utilizzano uno o più operatori, a seconda dei casi. Esistono varie tipologie di operatori, in funzione del numero e del tipo di operandi. Un **operatore** è un elemento di codice che esegue un'operazione su uno o più **operand**i, ovvero elementi che contengono un valore di qualche tipo (variabili, costanti, oggetti ed espressioni).

Un'**espressione** è una sequenza di operandi, combinati con operatori allo scopo di ritornare un valore finale. Gli operatori agiscono sugli operandi eseguendo calcoli, confronti o altre operazioni di vario genere.

Sebbene siano presenti operatori unari e anche un operatore ternario, la maggior parte degli operatori accettano due operandi. I principali operatori riguardano le operazioni aritmetiche, le operazioni di confronto, le espressioni booleane e le assegnazioni di variabili ([esempio 3.10](#)).

Esempio 3.10

```
bool x, y;           // Variabili booleane (default = false)
x = true;           // Assegnazione del valore true
x = !y;             // Negazione booleana (x vale true)
x = (18 > 8);        // A x viene assegnato il valore booleano
                    // risultante dalla valutazione
                    // dell'espressione di confronto (true)

int i, j, k;         // Variabili intere
i = 2;              // Assegnazione
j = i + 1;          // Somma e assegnazione
k = i * 2;          // Prodotto e assegnazione

int num;            // Variabile intera
num = i * j + k;    // Prodotto, somma e assegnazione

bool isOdd;         // Variabile booleana
isOdd = (num % 2) != 0; // Espressione che valuta se num è dispari

string hello;       // Variabile di tipo string
hello = "Hello " + "World"; // Concatenazione di stringhe
```

La [Tabella 3.4](#) elenca gli operatori predefiniti in C# in funzione della loro categoria di appartenenza.

Tabella 3.4 – Categorie degli operatori in C#

Categoria	Operatori
Aritmetici	+ - * / %
Logici (booleani e bit per bit)	& ^ ! ~ && true false
Concatenamento di stringhe	+
Incremento e decremento	++ --
Spostamento	<< >>
Relazionali	== != < > <= >=
Assegnazione	= += -= *= /= %= &= = ^= <<= >>= ??
Accesso a membri	.
Indicizzazione (array)	[]
Slice e span (array)	.. ^
Casting	()
Condizionale	?:
Concatenazione e rimozione di delegati	+ -
Creazione di oggetti (classi)	new
Informazioni sui tipi	as is sizeof typeof
Controllo delle eccezioni di overflow	checked unchecked
Riferimento indiretto e indirizzo	* -> [] &
Lambda	=>

Con l'eccezione dell'operatore di assegnazione, gli operandi di un'espressione vengono sempre valutati da sinistra verso destra. Dato che un'espressione può contenere più operatori, essi vengono valutati secondo un determinato ordine di precedenza. Per esempio, l'operatore di moltiplicazione o divisione aritmetica viene sempre valutato prima di quello relativo alla somma; l'operatore di negazione logica viene valutato prima dell'AND o dell'OR logico. Quest'ordine può essere variato impiegando le parentesi tonde per aggregare le diverse parti di un'espressione in modo personalizzato.

Nell'[esempio 3.11](#), l'uso delle parentesi tonde fa variare in modo significativo il risultato dell'operazione aritmetica tramite cui viene assegnata la variabile `num`, dal momento che la somma viene eseguita prima della moltiplicazione.

Esempio 3.11

```
int num = 0;      // Variabile intera
num = 2 * 3 + 4 * 5;    // num vale 26
num = 2 * (3 + 4) * 5;  // num vale 70
```

La [Tabella 3.5](#) elenca gli operatori predefiniti in C# in funzione del loro ordine di precedenza.

Tabella 3.5 – Priorità degli operatori in C#

Tipologia	Operatori
Primari	<code>x.y f(x) a[x] x++ x-- new typeof (un)checked nameof</code>
Unari	<code>+ - ! ~ ++x --x (T)x</code>
Moltiplicativi	<code>* / %</code>
Additivi	<code>+ -</code>
Shift	<code><< >></code>
Relazionali	<code>< > <= >= is as</code>
Uguaglianza	<code>== !=</code>
AND su bit	<code>&</code>
XOR su bit	<code>^</code>
OR su bit	<code> </code>
AND logico	<code>&&</code>
OR logico	<code> </code>
Condizionale	<code>x ? y : z</code>
Assegnazione	<code>= *= /= %= += -= <<= >>= &= ^= = ??</code>
Lambda	<code>=></code>

Come possiamo notare, insieme all'operatore `=>` usato nelle lambda expression, l'assegnazione è, in generale, l'operatore a più bassa priorità, preceduto, nell'ordine, dagli operatori logici, da quelli di confronto e da quelli aritmetici.

Conversione tra tipi

Tra gli operatori elencati nella [Tabella 3.3](#), quello di casting merita un approfondimento. Per quanto abbiamo visto finora, una volta che una variabile viene dichiarata, ad essa viene associato un tipo che può essere di valore o di riferimento. Peraltro, in C# esiste la possibilità di convertire il tipo di una variabile in un altro tipo affine (operazione anche nota col nome di **casting**): per esempio, un `int` in un `long`.

Quando si parla di casting, dobbiamo fare una distinzione tra le conversioni fra tipi di valore, in particolare i tipi numerici, e le conversioni fra tipi di riferimento. Nel primo caso, le modalità di casting si suddividono in conversioni implicite e conversioni esplicite. Le **conversioni implicite** sono quelle che avvengono senza il rischio di perdita d'informazioni, sono del tutto trasparenti e lo sviluppatore non si deve preoccupare di nulla. Per esempio, una variabile di tipo `byte` può essere convertita implicitamente in un intero perché il range dei valori validi per il tipo `byte` è incluso nel range dei valori di `int`. Le **conversioni esplicite** presuppongono, piuttosto, una perdita d'informazioni, in quanto il tipo di destinazione non è in grado di rappresentare l'intero dominio dei valori del tipo origine. In questo caso, lo sviluppatore è tenuto a specificare tramite l'operatore di casting l'intenzione di voler effettuare una conversione di tipo.

Per effettuare una conversione esplicita, dobbiamo anteporre alla variabile da convertire il tipo di destinazione racchiuso in una coppia di parentesi tonde. Omettendo l'operatore di casting, il compilatore segnalerà un errore. Per esempio, per convertire una variabile di tipo intero in una di tipo `byte`, siamo tenuti a specificare l'operatore di conversione (`byte`). In ogni caso, dobbiamo tenere in considerazione l'eventualità di poter generare un'eccezione di overflow a runtime qualora il valore convertito non appartenga all'intervallo di valori validi per il tipo di destinazione. Nell'[esempio 3.12](#) sono riportate alcune casistiche significative di conversione implicita ed esplicita dei tipi numerici.

Esempio 3.12

```
int x = 0;      // Variabile intera (4 byte)
byte y = 100;   // Variabile di tipo byte (1 byte)
x = y;         // Conversione implicita
y = (byte) x;   // Conversione esplicita
y = x;         // Errore di compilazione
x = 1000;      // Valore non incluso nel range del tipo byte
y = (byte) x;   // Errore di overflow a runtime
```

Per quanto riguarda le conversioni tra tipi di riferimento, dobbiamo innanzitutto precisare che in questi casi non è l'oggetto a essere convertito, ma solo il suo riferimento. Anche per i tipi di riferimento possiamo fare la distinzione tra conversioni implicite ed esplicite (l'operatore è lo stesso), ma in questo caso la compatibilità tra i tipi è garantita dai legami di ereditarietà o implementazione che caratterizzano le classi e le interfacce. Questi argomenti verranno affrontati nel corso del prossimo capitolo.

Array

Un **array** (detto anche **array monodimensionale** o **vettore**) è una variabile composta da un gruppo di oggetti dello stesso tipo. Gli elementi che fanno parte di un array sono contrassegnati con un indice, tramite il quale è possibile accedere a ciascuno di essi. In C# un array è un'istanza del tipo `System.Array` contenuto in .NET (si tratta di una classe .NET Standard) ed è pertanto un tipo di riferimento a tutti gli effetti.

Per richiamare un elemento di un array, dobbiamo utilizzare l'operatore d'indicizzazione riportato nella [Tabella 3.3](#). In C# quest'operatore è rappresentato da una coppia di parentesi quadre contenenti l'indice dell'elemento. L'indice vale zero per il primo elemento dell'array e viene incrementato di un'unità per ogni elemento successivo. L'elemento nella posizione *i*-esima è sempre contrassegnato con l'indice *i*-1 ([esempio 3.13](#)).

Esistono principalmente due modi per dichiarare un array, entrambi mostrati nell'[esempio 3.13](#). Nel primo caso, viene creato un array composto da tre elementi di tipo `string` e la dimensione viene specificata in modo esplicito. Nel secondo caso, gli elementi vengono inizializzati con altrettanti numeri interi e la dimensione dell'array risultante è pari al numero dei valori racchiusi

tra le parentesi graffe. A seconda di come viene dichiarato, un array può contenere sia tipi di valore, sia tipi di riferimento.

Esempio 3.13

```
string[] x = new string[3];    // Vettore composto da tre stringhe
x[0] = "Hello ";             // Contiene "Hello "
x[1] = "World";              // Contiene "World"
x[2] = x[0] + x[1];           // Contiene "Hello World"

int[] y = { 1, 2, 3 };        // Vettore composto da tre interi
int z = y[1];                 // La variabile z vale 2
```

Come avviene per le variabili semplici, in C# possiamo dichiarare un array sfruttando la type inference. In questa modalità di dichiarazione, il tipo di tutti gli elementi dell'array deve essere sempre lo stesso (o comunque implicitamente convertibile). L'utilizzo di tipi eterogenei o per i quali non sia possibile eseguire una conversione implicita, produce un errore in fase di compilazione. L'[esempio 3.14](#) mostra alcuni casi significativi di utilizzo di questa forma sintattica.

Esempio 3.14

```
var x1 = new[] { 0, 1, 2, 3, 4, 5 };    // int[]
var x2 = new[] { 0, 0.5, 1, 1.5, 2 };    // double[]
var x3 = new[] { "Riccardo", "Golia" };  // string[]
var x4 = new[] { 1, "x", 2, "y", 3, "z" }; // Errore
```

Gli array possono essere multidimensionali. La loro dichiarazione è analoga a quanto visto per gli array monodimensionali ([esempio 3.15](#)).

Esempio 3.15

```
int[, ,] coords = new int[3, 3, 3];    // Coordinate spaziali (dim. 3)
coords[0, 0, 0] = 1;                    // Origine (x = 0, y = 0, z = 0)
coords[1, 1, 1] = 2;                    // x = 1, y = 1, z = 1
coords[2, 2, 2] = 3;                    // x = 2, y = 2, z = 2
```

La notazione per separare tra loro le dimensioni è rappresentata da una virgola e il numero di virgole stabilisce l'ordine di grandezza dell'array. Per esempio, due virgole indicano che l'array ha tre dimensioni.

In C# sono contemplati anche gli array di array (**array jagged**), ma questo tipo di struttura dati non rientra nella Common Language Specification, ovvero l'insieme di regole che permettono di definire oggetti compatibili tra i diversi linguaggi e le diverse implementazioni di .NET (vedi [Capitolo 1](#)). Ove possibile, sconsigliamo di usare gli array jagged e suggeriamo di preferire soluzioni alternative.

Enumerazioni

In C# esiste il tipo **enum** (enumerazione) che permette di definire un insieme chiuso di valori. Questo tipo di dato torna comodo per obbligare una variabile a contenere solo una serie finita di valori preindividuati (per esempio, i mesi dell'anno, i giorni della settimana ecc.).

La parola chiave **enum** permette di definire l'elenco dei valori ammissibili per un particolare tipo di valore che, per default, è **int** a partire da zero ([esempio 3.16](#)). Dal momento che possiamo considerare le enumerazioni come sottoinsiemi di un particolare tipo numerico, esse sono, a tutti gli effetti, tipi di valore.

Esempio 3.16

```
enum Gender      // Enumerazione di tipo int (default) - Sesso
{
    Male,        // Vale 0 (int) - Maschio
    Female       // Vale 1 (int) - Femmina
}
```

Per le enumerazioni possiamo specificare un tipo numerico diverso da **int** e forzare il valore per ogni elemento dell'insieme tramite un'espressione valida per il tipo in questione ([esempio 3.17](#)). Oltre a **int**, i tipi di dati utilizzabili con le enumerazioni possono essere **byte**, **long**, **sbyte**, **short**, **uint**, **ulong** e **ushort**.

Esempio 3.17

```
enum DayOfWeek : byte    // Giorno della settimana
{
    Monday = 1,          // Vale 1 (byte) - Lunedì
    Tuesday = 2,         // Vale 2 (byte) - Martedì
    Wednesday = 3,       // Vale 3 (byte) - Mercoledì
    Thursday = 4,        // Vale 4 (byte) - Giovedì
    Friday = 5,          // Vale 5 (byte) - Venerdì
    Saturday = 6,        // Vale 6 (byte) - Sabato
    Sunday = 7           // Vale 7 (byte) - Domenica
}
```

Dato che, implicitamente, gli elementi di un'enumerazione contengono valori di tipo numerico, possiamo effettuare nei due sensi una conversione esplicita.

Esempio 3.18

```
Gender x = Gender.Male;    // Variabile di tipo Gender
int y = 0, z = 2;         // Variabili intere
y = (int) Gender.Female;  // y vale 1
y = (int) x;              // y vale 0
x = (Gender) y;           // Valore ammissibile
x = (Gender) z;           // Errore - Valore 2 non ammissibile
```

L'[esempio 3.18](#) prende in considerazione l'enumerazione `Gender` definita nell'[esempio 3.16](#). Come mostrato, possiamo fare il casting di un elemento di `Gender` al tipo intero e, viceversa, convertire una variabile intera al tipo `Gender` se il valore contenuto nella variabile è ammissibile per l'enumerazione.

Funzioni e procedure

Una **routine** rappresenta un insieme d'istruzioni racchiuse in un unico blocco a formare un'entità richiamabile più volte nell'ambito del codice. Scopo delle routine è quello di eseguire un qualche tipo di elaborazione, sfruttando, eventualmente, un insieme di parametri di input, e fornire, qualora previsto, un risultato sotto forma di un valore di ritorno.

Come detto, una routine può essere richiamata in più punti del codice. L'invocazione di una routine interrompe l'esecuzione del codice chiamante e

inserisce un'entry in cima allo stack delle chiamate. Al termine della sua esecuzione, la chiamata viene rimossa dallo stack e l'elaborazione del codice riprende dall'istruzione successiva a quella d'invocazione. Una routine può, a sua volta, richiamare altre routine: in tal caso, l'elaborazione della routine chiamante viene interrotta fino a che non termina l'elaborazione della routine chiamata.

In C# si distinguono due tipologie di routine: le **funzioni**, che restituiscono un valore, e le **procedure**, che non generano risultati, limitandosi a eseguire una serie di istruzioni ([esempio 3.19](#)).

Esempio 3.19

```
// Funzione che calcola la somma di due interi int Sum(int x, int y)
{
    return x + y;
}

// Procedura che scrive un messaggio a video
void Write(string text)
{
    Console.Write(text);
}
```

Come possiamo notare, la dichiarazione di una routine è, in generale, composta da una serie di elementi:

- ❑ un tipo di ritorno per le funzioni oppure la parola chiave `void` qualora non venga restituito alcun valore;
- ❑ un nome identificativo;
- ❑ l'elenco dei parametri compresi in una coppia di parentesi tonde.

Una routine è pienamente identificata dalla sua **firma**. Detta anche **signature**, la firma di una funzione o di una procedura è rappresentata dal suo nome identificativo, dal numero dei parametri e dal loro tipo; il tipo dell'eventuale valore di ritorno non fa parte della firma. Il concetto di firma è importante in quanto, nell'ambito del loro contesto di validità, tutte le routine devono avere una signature univoca. Questo significa che, in determinate situazioni, come nel caso della definizione dei tipi di riferimento, più funzioni o procedure possono avere lo stesso nome identificativo, ma il numero dei parametri e/o il

loro tipo deve essere necessariamente differente (**overloading**). Riprenderemo quest'argomento nel corso del prossimo capitolo.

Un altro aspetto importante che riguarda le routine è il **passaggio dei parametri**. In C# i parametri possono essere passati:

- ❑ per valore (**by value**): si tratta della modalità di default, per la quale non viene specificata alcuna parola chiave;
- ❑ per riferimento (**by reference**): viene utilizzata la keyword `ref` e la modalità si applica solamente ai parametri inizializzati in ingresso;
- ❑ per riferimento in uscita (**by reference**): viene usata la keyword `out` e la modalità si applica ai parametri che vengono valorizzati nell'ambito della routine e che sono usati come risultato in uscita, indipendentemente dal loro valore in ingresso;
- ❑ per riferimento in ingresso (**by reference**): viene usata la keyword `in` e la modalità si applica ai parametri che non possono essere modificati all'interno della funzione chiamante. Questa è una novità introdotta da C# 7.2.

L'[esempio 3.20](#) mostra due delle tre casistiche di passaggio dei parametri per una routine che somma due numeri.

Esempio 3.20

```
// Passaggio per valore
int Sum(int x, int y)
{
    return x + y;
}

// Passaggio per valore e per riferimento
void Sum(int x, int y, out int result)
{
    result = x + y;
}

// Passaggio per riferimento in ingresso
int Test(in int x)
{
    x = 11;    // questa istruzione produce un errore di compilazione
}

int num = Sum(11, 22);    // num vale 33
Sum(111, 222, out num);  // num vale 333
```

```
Sum(111, 222, out _);    // con _ ignoriamo il valore di ritorno  
  
Sum(111, 222, out var y);    // il valore in uscita sarà nella variabile y
```

Quando il parametro è un tipo di valore, nel caso di passaggio by value, la variabile viene copiata all'interno della routine e un'eventuale riassegnazione non influenza il chiamante. Se il passaggio è di tipo by reference, ogni variazione al parametro viene propagata anche al chiamante.

Quando il parametro è un tipo di riferimento, nel caso di passaggio by value, la variabile non viene copiata, ma viene semplicemente passato il suo riferimento che è un puntatore a un'area di memoria. In questo caso, la variabile non può essere assegnata nuovamente. Se il passaggio è di tipo by reference, ancora una volta viene passato semplicemente un riferimento, ma, diversamente dal caso precedente, la variabile può essere riassegnata liberamente.

Le ultime due versioni sono una novità di C# 7.x e 8, rispettivamente, e consentono di ignorare il valore di ritorno (con il carattere `_`) attraverso una funzionalità chiamata *discard*, oppure di creare in un colpo solo una variabile in uscita, senza dichiararla in precedenza (con `out var`) e potendola utilizzare normalmente nel flusso di codice successivo.

A partire dalla versione 4, per funzioni e procedure, C# permette di usare sia **parametri opzionali**, sia **parametri nominali**. I parametri opzionali ci consentono di poter omettere il loro valore nell'invocazione della routine, dal momento che per essi viene indicato un valore di default nella dichiarazione. I parametri nominali ci permettono invece di specificare gli argomenti in funzione del loro nome, indipendentemente dalla loro posizione all'interno della firma. Per spiegare la sintassi da usare nei due casi, facciamo riferimento all'[esempio 3.21](#).

Esempio 3.21

```
// Funzione che somma tre numeri con due parametri opzionali  
int Sum(int x, int y = 0, int z = 0)  
{  
    return x + y + z;  
}  
  
// Numeri interi  
int i = 18, j = 8, k = 5;  
  
// Invocazione classica: il risultato è 31  
Sum(i, j, k);
```

```

// Uso dei parametri opzionali
Sum(i, j);      // Equivale a: Sum(i, j, 0) e il risultato è 26
Sum(i);        // Equivale a: Sum(i, 0, 0) e il risultato è 18

// Uso dei parametri nominali
Sum(z: i, y: j, x: k);    // Equivale a: Sum(k, j, i) e il risultato è 31
Sum(z: k, i);            // Errore di compilazione

// Uso di un parametro nominale per assegnare un valore diverso da
// quello di default all'ultimo parametro opzionale
Sum(i, z: k);           // Equivale a: Sum(i, 0, k) e il risultato è 23
Sum(i, ,k);            // Errore di compilazione

```

Come possiamo notare, i parametri opzionali vengono definiti tramite un'espressione di assegnazione all'interno della dichiarazione della routine. Il valore assegnato dev'essere obbligatoriamente una costante.

All'interno della firma, dobbiamo specificare i parametri opzionali in fondo alla lista degli argomenti. In caso contrario, otteniamo un errore di compilazione. Inoltre, nell'invocazione di una funzione o di una procedura, non possiamo omettere un parametro opzionale se non è l'ultimo di quelli specificati. Anche in questo caso, ci troviamo a ottenere un errore di compilazione, sempre che non usiamo un parametro nominale.

Nella chiamata di una routine, i parametri nominali devono essere riportati sempre dopo gli argomenti posizionali, qualora siano presenti. Essi vengono specificati facendo precedere al valore parametrico il nome dell'argomento seguito dal carattere ":" (due-punti). Il loro utilizzo ci consente di poter variare l'ordine di apparizione degli argomenti nella chiamata della routine e di poter associare a un parametro opzionale un valore diverso da quello di default, indipendentemente dal numero di argomenti specificati e dalla posizione del parametro opzionale nella firma.

C# 8 introduce il concetto di **Nullable reference type**: si tratta della possibilità di non consentire di impostare a `null` un reference type, evitando problemi legati a riferimento che puntano a variabili non istanziate. L'ambito di applicazione è spiegato nell'[esempio 3.22](#).

Esempio 3.22

```

string name = null;      // warning
string? name = null;    // OK

void PrintName(string? s)
{
    Console.WriteLine(s.Length);    // warning
}

```

```
if (s != null)
    Console.WriteLine(s.Length);    // OK
}
```

Grazie al fatto che abbiamo specificato di accettare un valore `null`, il compilatore farà un'analisi del nostro codice e ci avviserà, con un warning, che stiamo cercando di accedere ad una proprietà di una variabile che molto probabilmente conterrà valori `null`, generazione un errore. L'obiettivo di questa nuova funzionalità è di evitare questo genere di situazioni, che in genere sono identificabili solo a runtime, quando è troppo tardi.

Istruzioni condizionali

Le istruzioni condizionali servono per determinare l'esecuzione di un blocco di istruzioni in funzione di un'espressione booleana. In C# le istruzioni di selezione sono due, `if` e `switch`, con l'aggiunta dell'operatore ternario condizionale.

Istruzione if

L'istruzione `if` valuta un'espressione booleana e, in funzione del suo valore (`true` oppure `false`), esegue il blocco di codice ad essa associato. Se l'espressione vale `true`, il blocco viene eseguito. In caso contrario, l'esecuzione riprende dalla prima istruzione successiva. Facoltativamente, può essere aggiunto un blocco d'istruzioni preceduto dalla parola chiave `else` oppure dalla combinazione `else if` che viene eseguito in alternativa al blocco principale. Se un blocco è composto da una sola istruzione, le parentesi graffe possono essere omesse ([esempio 3.23](#)).

Esempio 3.23

```
if (name == "C#")
{
    // Blocco principale (obbligatorio)
}
else
{
    // Blocco alternativo (facoltativo)
}
```

Un blocco dell'istruzione `if...else` può contenere, a sua volta, altre istruzioni `if...else`. In genere, questo produce una certa confusione nel codice. In taluni casi il ricorso all'istruzione `switch` può produrre un miglioramento della leggibilità.

Istruzione switch

L'istruzione `switch` permette di stimare il valore di una variabile (che può essere anche una stringa) e, in funzione di questo valore, eseguire un particolare blocco di codice. Il blocco `switch` contiene a sua volta una serie di blocchi contrassegnati con la parola chiave `case`, seguita da un valore costante dello stesso tipo della variabile in esame. Un blocco viene eseguito in modo esclusivo rispetto agli altri fino alla prima istruzione di salto `break` (oppure `return`) se il valore della variabile combacia con quello della costante. La parola chiave `default` identifica il blocco finale che viene eseguito se nessun valore costante corrisponde al valore della variabile. In passato questo blocco andava posto sempre per ultimo, mentre nelle ultime versioni di C# può essere messo in una qualsiasi posizione. Fin dalla versione 1, non è obbligatorio specificarlo.

Esempio 3.24

```
switch (name)
{
    case "Matteo":      // Valore singolo
        // ...
        break;
    case "Cristian":   // Viene eseguito lo stesso blocco di
    case "Daniele":    // codice nel caso in cui il valore di
    case "Marco":      // name sia uno dei tre indicati
        // ...
        break;
    default:           // Blocco di default opzionale
        // ...
        break;
}
```

Nel caso in cui il blocco di codice da eseguire corrisponda a più valori, possiamo raggruppare le istruzioni `case` seguite dalle diverse costanti, come mostrato nell'[esempio 3.24](#). In questo modo, il blocco di codice viene eseguito

qualora la variabile in esame assuma almeno uno dei valori indicati consecutivamente.

C# 7 ha introdotto una funzionalità, nota come **pattern matching**, che poi C# 8 ha esteso, con la possibilità di utilizzare istruzioni di questo tipo in contesti maggiori. Un esempio in tal senso è contenuto qui di seguito.

Esempio 3.25

```
switch (person)
{
    case Customer a when (a.Active):    // se è un Customer attivo
        // ...
        break;
    case Supplier b when (b.IsValid):    // se è un Supplier con validità in corso
        // ...
        break;
    default:                            // blocco di default opzionale
        // ...
        break;
    case null:                          // caso speciale per gestire i null
        // ...
        break;
}
```

In questo momento non abbiamo ancora illustrato una serie di concetti legati all'OOP (saranno introdotti nel prossimo capitolo), ma possiamo notare alcuni punti all'interno del codice:

- ❑ qualora la variabile dovesse essere `null`, anche se l'istruzione è l'ultima della serie, si arriverebbe direttamente a quest'ultima;
- ❑ la sequenza è importante, poiché, come in un blocco `if`, la prima istruzione vera farebbe saltare la valutazione delle altre.

Nell'intento di rendere sempre più compatto il codice, C# 8 introduce una variante molto interessante di questa istruzione, contenuta nell'[esempio 3.26](#).

Esempio 3.26

```
public string GetDescription(Person person)
{
    return person switch
    {
```

```
Customer c => $"Cliente {p.CustomerName}",
Supplier s => $"Fornitore {p.SupplierName}",
_ => "{person.FirstName} {person.LastName}" // default
};
}
```

La particolarità di questo codice è di adattarsi più facilmente a LINQ e alle sue espressioni, come vedremo nei prossimi capitoli. Il carattere “_” (underscore) è un carattere speciale, che si può utilizzare tutte le volte che il valore non è necessario nel resto del codice.

Operatore binario

L’operatore condizionale permette di definire un’espressione composta da tre operandi: un’espressione booleana, che funge da condizione di controllo, e due espressioni separate dal carattere “:” (due-punti), che forniscono i valori nel caso in cui la condizione sia, rispettivamente, vera oppure falsa. Una variante dell’operatore condizionale è l’operatore binario ?? (doppio punto-di-domanda), che consente di ritornare un valore alternativo qualora una determinata variabile relativa a un tipo di riferimento sia pari a null ([esempio 3.27](#)).

Esempio 3.27

```
// Se i è dispari, isOdd vale true
bool isOdd = (i % 2 == 1) ? true : false;

// Se x è null, ritorna y, altrimenti ritorna x
string x = null;
string y = "Hello World";
string z = x ?? y; // z vale "Hello World"
```

Data la forma estremamente compatta dell’espressione basata sull’operatore condizionale, è consigliabile non abusarne per non compromettere la leggibilità del codice e preferire l’uso dell’istruzione `if` laddove questo possa comportare una maggiore chiarezza.

Operatore null-conditional

Introdotta da C# 6, l'operatore null-conditional ?. (punto-di-domanda, punto) è molto potente, poiché consente di valutare l'espressione alla sua destra solo se quella alla sinistra è diversa da `null`.

Esempio 3.28

```
// codice precedente
string name = "";
if (customer != null)
    name = customer.Name;

// nuova sintassi
string name = customer?.Name;    // null se customer è null
```

Come per gli altri operatori condizionali, anche in questo caso vale la regola del corto circuito: se uno dei valori specificati dovesse essere `null`, non vengono valutate le istruzioni alla destra, evitando problemi derivanti da un accesso a una proprietà di una classe non istanziata (che genererebbe un errore). Come si può apprezzare nell'[esempio 3.28](#), questa nuova sintassi semplifica decisamente il codice e lo rende più elegante e semplice da comprendere. Anche in questo caso, ne apprezzeremo l'eleganza quando avremo a che fare con LINQ.

Istruzioni iterative

Le istruzioni d'iterazione servono per eseguire un blocco di istruzioni ripetutamente, in modo ciclico. In C# esistono diverse istruzioni per eseguire iterazioni nel codice, ciascuna caratterizzata da una diversa modalità d'uscita dal ciclo.

Istruzione while

L'istruzione `while` permette di definire un ciclo dove la condizione booleana di uscita viene sempre valutata prima dell'iterazione. Il ciclo termina nel momento in cui la condizione di controllo risulti essere falsa. Nel caso in cui il ciclo sia composto da una sola istruzione, le parentesi graffe che delimitano il blocco delle istruzioni possono essere omesse.

Esempio 3.29

```
int[] x = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int y = 0;      // Somma dei numeri contenuti nel vettore
int i = 0;      // Indice del vettore

// Somma i dieci numeri contenuti nel vettore
// Alla fine del ciclo y vale 55 e i vale 10
while (i < 10)
{
    y += x[i];
    i++;      // Incremento di 1
}
```

L'[esempio 3.29](#) mostra come eseguire un'iterazione per sommare dieci numeri contenuti in un vettore di interi. In tal caso, a ogni ciclo, l'indice dell'array deve essere esplicitamente incrementato di 1, affinché la condizione d'uscita (indice minore di 10) risulti falsa una volta letti tutti i valori.

Istruzione do...while

L'istruzione `do...while` permette di definire un ciclo in cui la condizione booleana di uscita viene sempre valutata dopo l'iterazione. Il ciclo termina nel momento in cui la condizione di controllo risulti essere falsa.

Esempio 3.30

```
int[] x = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int y = 0;      // Somma dei numeri contenuti nel vettore
int i = 0;      // Indice del vettore

// Somma i dieci numeri contenuti nel vettore
// Alla fine del ciclo y vale 55 e i vale 10
do
{
    y += x[i];
    i++;      // Incremento di 1
}
while (i < 10);    // Condizione finale
```

L'[esempio 3.30](#) mostra l'uso dell'iterazione basata sullo statement `do...while` per eseguire la somma dei numeri contenuti in un vettore. Anche in questo caso, come in precedenza, affinché la condizione di uscita sia vera, l'indice dell'array deve essere esplicitamente incrementato di 1.

Istruzione for

L'istruzione `for` permette di definire un ciclo per un intervallo di indici numerici. Essa si compone di tre parti racchiuse in una coppia di parentesi tonde: l'inizializzazione dell'indice con l'eventuale dichiarazione, la condizione booleana di uscita basata sul controllo del valore dell'indice rispetto al suo valore massimo o minimo e l'istruzione di incremento o decremento dell'indice. Nonostante le tre parti dell'istruzione `for` non siano obbligatorie, è vivamente consigliato mantenerle sempre. Anche in questo caso, qualora il ciclo sia composto da una sola istruzione, possiamo omettere le parentesi graffe che delimitano il blocco delle istruzioni.

Esempio 3.31

```
int[] x = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int y = 0;      // Somma dei numeri contenuti nel vettore

// Somma i dieci numeri contenuti nel vettore      // Alla fine del ciclo y vale 55
for (int i = 0; i < 10; i++)
{
    y += x[i];
}
```

Come possiamo notare nell'[esempio 3.31](#), diversamente dai due casi precedenti, non dobbiamo specificare l'istruzione d'incremento dell'indice, ma solamente definire il suo range di validità, ovvero il valore di partenza e quello di arrivo.

Istruzione foreach

L'istruzione `foreach` permette di compiere iterazioni sui tipi enumerabili, come possono essere gli array, e di scorrere gli elementi senza la necessità di definire e utilizzare un indice, come invece avviene per l'istruzione `for` ([esempio 3.32](#)). Come nei casi precedenti, qualora il ciclo sia composto da una sola istruzione, possiamo ancora una volta omettere le parentesi graffe che delimitano il blocco delle istruzioni.

Esempio 3.32

```
int[] x = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int y = 0;    // Somma dei numeri contenuti nel vettore

// Somma i dieci numeri contenuti nel vettore
// Alla fine del ciclo y vale 55
foreach (int item in x)
{
    y += item;
}
```

È importante sottolineare il fatto che gli elementi che vengono iterati non possono essere modificati all'interno del ciclo, pena il sollevamento di un'eccezione a runtime.

Istruzioni di salto

Le istruzioni di salto permettono di modificare il normale flusso di esecuzione sequenziale del codice, introducendo spostamenti tra istruzioni non contigue.

Istruzione break

Posta all'interno di un blocco `switch` oppure in un ciclo, l'istruzione `break` ne termina l'esecuzione, trasferendo il controllo all'istruzione immediatamente successiva.

Esempio 3.33

```
int[] x = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int y = 0;    // Somma dei numeri contenuti nel vettore

// Somma i primi quattro numeri diversi da 5 contenuti nel vettore
// Alla fine del ciclo y vale 10
foreach (int item in x)
{
    if (item == 5)
        break;
    y += item;
}
```

Nell'[esempio 3.33](#), gli elementi dell'array superiori al 4 non vengono considerati nel computo della somma, dal momento che il ciclo s'interrompe definitivamente quando il valore diventa pari a 5.

Istruzione continue

Posta all'interno di un ciclo, l'istruzione `continue` termina l'iterazione corrente e trasferisce il controllo all'iterazione successiva.

Esempio 3.34

```
int[] x = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int y = 0;      // Somma dei numeri contenuti nel vettore

// Somma i numeri diversi da 5 contenuti nel vettore
// Alla fine del ciclo y vale 50
foreach (int item in x)
{
    if (item == 5)
        continue;
    y += item;
}
```

Nell'[esempio 3.34](#) l'elemento dell'array con valore 5 non viene considerato nel computo della somma. Infatti, il ciclo s'interrompe momentaneamente, per riprendere dall'elemento successivo con una conseguente variazione del risultato finale.

Istruzione return

L'istruzione `return` termina l'esecuzione di una routine, eventualmente restituendo un valore di ritorno, qualora previsto. Nelle procedure l'istruzione va usata senza specificare il valore di ritorno.

Esempio 3.35

```
// Somma i numeri contenuti nel vettore passato come parametro
// Se il vettore è nullo ritorna zero, altrimenti esegue la somma
int Sum(int[] x)
{
    if (x == null)
    {
        return 0;
    }

    int y = 0;
    foreach (int item in x)
    {
        y += item;
    }
}
```

```
}  
    return y;  
}
```

Le funzioni, in quanto routine che ritornano necessariamente un valore, devono contenere sempre almeno un'istruzione `return`. In realtà, il numero d'istruzioni di ritorno dentro a una funzione è determinato dal numero di flussi d'esecuzione presenti in quest'ultima. L'omissione di un'istruzione di ritorno per un flusso in una funzione produce inevitabilmente un errore in fase di compilazione.

Istruzione goto

L'istruzione `goto` consente di trasferire il controllo in corrispondenza di un punto del codice contrassegnato da un'etichetta, ovvero un nome seguito dal carattere ":" (due-punti) che precede, a sua volta, una serie di istruzioni.

Esempio 3.36

```
int[] x = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
int y = 0;    // Somma dei numeri contenuti nel vettore  
  
// Somma i primi quattro numeri diversi da 5 contenuti nel vettore  
// Alla fine del ciclo y vale 10 e la somma viene visualizzata a video  
foreach (int item in x)  
{  
    if (item == 5)  
        goto finish;  
    y += item;  
}  
  
finish:    // Etichetta  
    Console.WriteLine("Somma: " + y);
```

Sebbene C# includa questa istruzione, ne sconsigliamo decisamente l'utilizzo. In genere, una buona strutturazione del codice permette di evitarne l'uso, con il grande vantaggio di accrescere notevolmente la leggibilità del codice.

Formattazione di stringhe

Le stringhe sono tra i tipi più utilizzati all'interno del codice perché ci consentono di produrre un output visualizzabile a video. C# ha storicamente utilizzato il metodo `Format` della classe `string`, per consentire di formattare le stringhe utilizzando un posizionamento basato su segnaposti. Nell'[esempio 3.37](#) vediamo uno spezzone di codice che mostra questa tecnica.

Esempio 3.37

```
string name = "Daniele";
string city = "Rionero in Vulture";

string fullName = string.Format("Ti chiami {0} e vieni da {1}",
                                name, city);
```

Il risultato di questa istruzione sarà che al posto di `{0}` e `{1}` verranno inseriti i valori prelevati dalla variabile specifica successivamente. Non c'è limite alle variabili specificabili, e il loro identificativo sarà determinato in base alla posizione di dichiarazione.

Questo approccio va bene quando il numero di variabili è minimo, altrimenti diventa complesso da gestire. Per questo motivo, C# 6 ha introdotto una nuova funzionalità, chiamata **string interpolation** (interpolazione di stringhe), che consente di avere una sintassi più semplice. Il codice dell'esempio precedente si trasforma in quello contenuto nell'[esempio 3.38](#).

Esempio 3.38

```
string name = "Daniele";
string city = "Milano";

string fullName = $"Ti chiami {name} e vivi a {city}";
```

Il carattere `$` (dollaro) è utilizzato per far comprendere al compilatore che siamo di fronte a un'interpolazione di stringhe. I segnaposti interni, anziché utilizzare il posizionamento basato su indice, saranno equivalenti al nome della variabile, che sarà poi inserito al posto giusto. Questo approccio rende molto più semplice variare il formato del testo per adattarlo alle nostre necessità.

Conclusioni

C# è un linguaggio managed, nato direttamente con il .NET Framework e poi evolutosi per supportare tutte le forme di .NET, come .NET Core, Xamarin o Unity, derivato da linguaggi preesistenti come il C e il C++. Per questo motivo, C# presenta non poche analogie con la sintassi del C e del C++, per quanto riguarda istruzioni, espressioni e operatori.

In questo capitolo abbiamo affrontato solamente gli aspetti base della sintassi di C#. Nel prossimo capitolo avremo modo di applicare le nozioni che abbiamo appreso nella definizione delle classi, che rappresentano gli elementi fondamentali per l'organizzazione e la strutturazione del codice nella programmazione orientata agli oggetti.

Object Oriented Programming

La programmazione procedurale, propria di linguaggi ormai datati come il Pascal o il Linguaggio C, si basa principalmente sull'organizzazione e suddivisione del codice in funzioni e procedure. A ciascuna operazione corrisponde una routine, che accetta un insieme di parametri e, in funzione di questi, produce eventualmente un risultato. Con un simile approccio, la separazione tra i dati e la logica applicativa risulta essere netta: una procedura riceve le informazioni utili all'esecuzione dell'operazione richiesta solamente sotto forma di parametri e il suo stato interno, in genere, viene del tutto perduto una volta terminata l'elaborazione. In molti casi, questa netta separazione tra dati e codice rappresenta un vincolo importante, tale da rendere questo paradigma di programmazione poco efficace.

La **programmazione orientata agli oggetti** (in modo abbreviato OOP, che in inglese corrisponde a Object Oriented Programming) introduce un modo diverso o, se vogliamo, più efficiente per strutturare il codice e la logica applicativa. OOP è, infatti, un paradigma di programmazione che si basa sulla definizione e sull'utilizzo di una serie di entità tra loro collegate e interagenti, ciascuna delle quali è caratterizzata da un insieme di informazioni di stato e di comportamenti specifici. Queste entità sono denominate **oggetti** e ciascuna di esse può contenere contemporaneamente dati, funzioni e procedure. In questo modo, una routine associata a un oggetto può sfruttare lo stato interno dell'oggetto di appartenenza per ricavare le informazioni utili all'esecuzione dell'elaborazione prevista.

Nel corso di questo capitolo vedremo come l'OOP si applichi al linguaggio C# e quali vantaggi comporti.

Vantaggi dell'Object Oriented Programming

Il grande vantaggio dell'approccio object-oriented rispetto agli altri paradigmi di programmazione consiste nel fatto che, per strutturare le applicazioni, lo sviluppatore si trova a utilizzare una logica che è molto vicina a quella che porta alla percezione comune del mondo reale. Pensare a oggetti significa, infatti, saper riconoscere gli aspetti che caratterizzano una particolare realtà e saper fornire, di conseguenza, una rappresentazione astratta in un'ottica OOP.

Per fare un esempio, consideriamo la contabilità di un'azienda. In quanto le fatture o le bolle sono sicuramente elementi del mondo reale che riguardano l'amministrazione aziendale, un sistema gestionale realizzato secondo l'approccio OOP deve necessariamente includere una rappresentazione di queste entità sotto forma di oggetti. L'idea essenziale di OOP consiste, infatti, nell'individuare l'insieme degli oggetti che costituiscono e caratterizzano la realtà in esame e, al tempo stesso, nel definire il modo con cui essi interagiscono tra loro ed evolvono nel tempo. Ciascun oggetto individuato non deve necessariamente corrispondere a un elemento del mondo reale; in taluni casi esso può consistere in una pura invenzione introdotta dallo sviluppatore per uno specifico scopo.

C# è un linguaggio orientato agli oggetti che permette di realizzare applicazioni che sfruttano .NET come ambiente di esecuzione. Esso sfrutta i principi base della programmazione orientata agli oggetti e utilizza, come elementi fondamentali per l'organizzazione e la strutturazione del codice, le **classi**, di cui gli oggetti sono istanze particolari.

Principi fondamentali di OOP

Come abbiamo avuto modo di dire nel corso del primo capitolo, i tipi (sia quelli di valore sia quelli di riferimento) sono gli elementi portanti di ogni applicazione che gira nell'ambito del Common Language Runtime. Oltre ai tipi built-in, inclusi nella Base Class Library di .NET, possiamo peraltro definire i nostri tipi personalizzati.

Le **classi** rappresentano i tipi di riferimento definiti dallo sviluppatore, mentre le **strutture** sono i tipi di valore. I loro elementi costitutivi (detti anche **membri**) possono essere sia dati, sia funzioni e procedure. I principi

fondamentali su cui si fonda la programmazione orientata agli oggetti e che riguardano le classi (e in parte anche le strutture) sono principalmente tre: **ereditarietà**, **polimorfismo** e **incapsulamento**. Avere dimestichezza con questi concetti è di fondamentale importanza per capire e, al tempo stesso, sfruttare al meglio i meccanismi che regolano la programmazione orientata agli oggetti.

Ereditarietà

Il principio di ereditarietà si basa sul fatto di poter definire un legame di dipendenza di tipo gerarchico tra classi diverse. Una classe deriva da un'altra se da essa ne eredita il comportamento e le caratteristiche. La classe figlia si dice **classe derivata**, mentre la classe padre prende il nome di **classe base** (oppure **superclasse**). La classe base contiene il codice comune a tutte le classi da essa derivate. Ogni classe derivata rappresenta invece una specializzazione della superclasse, ma eredita in ogni caso dalla classe base i comportamenti in essa definiti.

*Tutte le classi contenute in .NET (comprese quelle create ex-novo dallo sviluppatore) derivano implicitamente dal tipo System.Object. Questa caratteristica permette di poter utilizzare variabili di tipo System.Object per referenziare le istanze di una qualsiasi classe, tramite un'operazione nota come **up-casting** (casting da una classe derivata verso una superclasse). L'operazione inversa prende il nome di **down-casting**.*

Per fare un esempio chiarificatore, supponiamo di voler rappresentare il genere animale definendo una struttura di classi equivalente (vedi [Figura 4.1](#)). Possiamo individuare una prima classe base, Animale, da cui tutte le altre devono derivare. A questo punto le classi derivate possono essere: Mammifero, Rettile, Insetto, Anfibio, Pesce, Uccello, ecc. La classe Uomo deriva direttamente da Mammifero e indirettamente da Animale. La classe Rana deriva da Anfibio, mentre la classe Rondine deriva da Uccello; entrambe discendono indirettamente da Animale.

Possiamo notare che in tutti i casi vale la relazione “è un tipo di” (l'uomo è un tipo di mammifero, la rana è un tipo di anfibio, la rondine è un tipo di uccello; l'uomo, la rana e la rondine sono un tipo di animale).

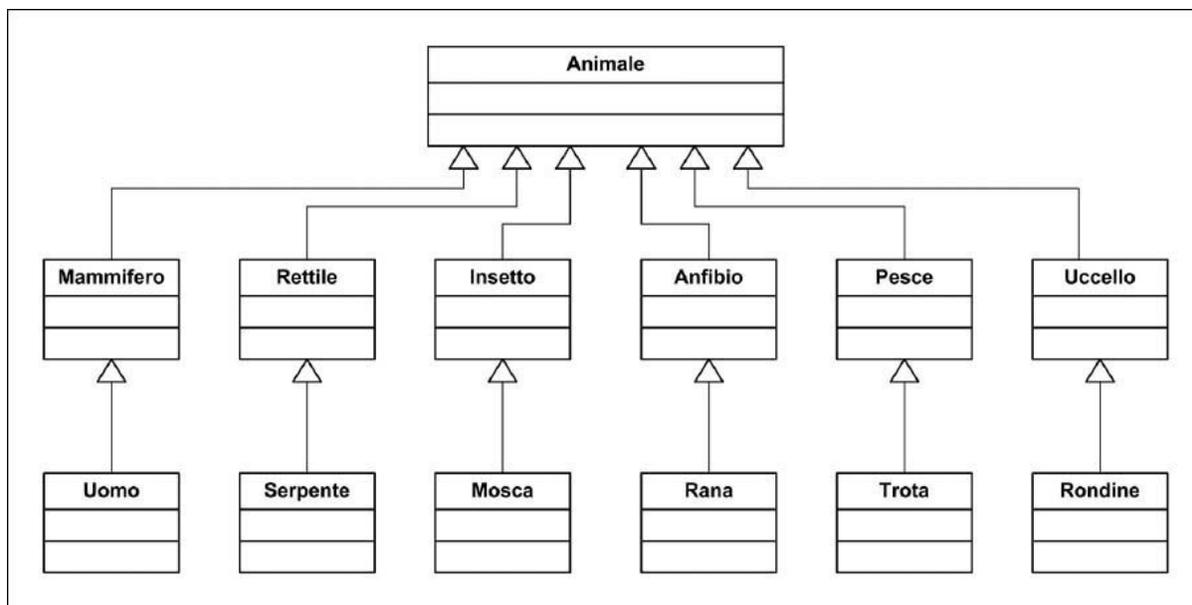


Figura 4.1 - La classe base `Animale` e le sue classi derivate.

Polimorfismo

Il polimorfismo rappresenta il principio in funzione del quale diverse classi derivate possono implementare uno stesso comportamento definito nella classe base in modo differente.

Dato che in .NET ogni classe deriva direttamente o indirettamente da `System.Object`, ogni oggetto presenta un certo numero di comportamenti comuni, alcuni dei quali possono essere specializzati sfruttando il polimorfismo. Tra questi comportamenti figurano le funzioni `ToString`, `GetHashCode` e `Equals`, la cui implementazione predefinita è inclusa nella classe base `System.Object`. Avremo modo di riprendere il discorso sul polimorfismo nel corso del capitolo.

Riprendendo l'esempio proposto nel caso dell'ereditarietà, consideriamo due comportamenti comuni a tutti gli animali: `Respira` e `Mangia`. Nel mondo animale questi comportamenti vengono messi in atto secondo modalità peculiari a seconda delle specie: un carnivoro mangia in modo differente rispetto a un erbivoro, un pesce respira in modo diverso rispetto a un uccello. Sulla base di queste considerazioni, i comportamenti `Mangia` e `Respira`, definiti nelle diverse classi derivate da `Animale`, possono essere implementati

in modo specifico e del tutto indipendente dalle altre implementazioni (e, in particolar modo, da quello della classe base `Animale`).

Incapsulamento

L'incapsulamento rappresenta il principio in base al quale una classe può mascherare la sua struttura interna e proibire ad altri oggetti di accedere ai suoi dati o di richiamare le sue funzioni che non siano direttamente accessibili dall'esterno.

Nei linguaggi di .NET l'incapsulamento è ottenuto grazie all'uso dei modificatori di accessibilità, di cui avremo modo di parlare nel corso del prossimo paragrafo.

Lo scopo principale dell'incapsulamento è di dare accesso allo stato e ai comportamenti di un oggetto solo attraverso un sottoinsieme di elementi pubblici. L'incapsulamento permette quindi di considerare una classe come una sorta di black-box, in altre parole una scatola nera che permette di mostrare solo ciò che è necessario, mascherando ciò che non deve trasparire verso l'esterno.

Classi

Una volta introdotti i principi della programmazione orientata agli oggetti, siamo pronti per vedere nel dettaglio come il linguaggio permetta di sfruttare il paradigma a oggetti tramite la definizione e l'uso delle classi.

Come abbiamo già detto nell'introduzione del capitolo, le classi sono i tipi di riferimento definiti dallo sviluppatore. In C# esse vengono dichiarate tramite la parola chiave `class`, seguita dal nome identificativo. L'[esempio 4.1](#) mostra la sintassi valida per la dichiarazione di una classe che rappresenta una persona.

Esempio 4.1

```
class Person
{
    // ...
}
```

Una classe è un contenitore che può includere una serie di membri sotto forma sia di dati, sia di comportamenti. Tra i membri di una classe possiamo distinguere i campi, i metodi, gli eventi e le proprietà.

Membri di una classe

I **campi** rappresentano le variabili interne alla classe (i dati), mentre i **metodi** sono le funzioni e le procedure (i comportamenti). Per i campi e i metodi valgono le stesse notazioni sintattiche viste nel corso del capitolo precedente per la dichiarazione delle variabili locali e delle routine ([esempio 4.2](#)).

Esempio 4.2

```
class Person
{
    string _fullName = "Sconosciuto";
    int _age = 18;

    string GetFirstName()
    {
        // ...
    }
}
```

Gli **eventi** sono elementi che permettono di inviare notifiche verso l'esterno in corrispondenza di certi avvenimenti che tendono a variare lo stato interno di una classe. A ciascun evento possono essere associate una o più azioni (dette **handler**), che vengono eseguite in corrispondenza della notifica dell'evento stesso. In .NET gli eventi sfruttano un tipo di dato particolare per agganciare i rispettivi handler. Si tratta dei **delegate**, che non sono altro che l'equivalente .NET dei puntatori a funzione presenti nei linguaggi come il C++. Agli eventi e ai delegate è dedicato l'intero [Capitolo 6](#).

Le **proprietà** sono membri che permettono di accedere in lettura e scrittura ai campi di una classe. In genere, una proprietà definisce una coppia di metodi (uno per la lettura di un campo e uno per la sua scrittura) e a questi associa un nome identificativo. La dichiarazione di una proprietà viene effettuata specificando due blocchi di codice contrassegnati dalle parole chiave `get` e `set`, corrispondenti rispettivamente alle operazioni di lettura e di scrittura ([esempio 4.3](#)). Questi due blocchi sono detti anche **accessor** (o funzioni di

accesso). Per una proprietà non è obbligatorio definire sempre entrambi gli accessor.

Esempio 4.3

```
public class Person
{
    // Campo privato che contiene il nome completo (nome e cognome)
    // Il campo è inizializzato con il valore "Sconosciuto"
    private string _fullName = "Sconosciuto";

    // Campo privato che contiene l'età
    // Il campo è implicitamente inizializzato con il valore 0
    private int _age;

    // Proprietà pubblica che espone il campo contenente il nome
    public string FullName
    {
        get { return __fullName; }
        set { _fullName = value; }
    }

    // Proprietà pubblica che espone il campo contenente l'età
    public string Age
    {
        get { return _age; }
        set { _age = value; }
    }
}
```

Generalmente le proprietà vengono associate a un campo privato, che è detto anche backing field e che contiene effettivamente il valore. In questo modo diventa possibile applicare una logica ai due metodi legati al caricamento e salvataggio dei dati, chiamati anche getter e setter.

In C# possiamo definire le proprietà di una classe in modo alternativo rispetto a quanto mostrato nell'[esempio 4.3](#), attraverso le **proprietà automatiche (automatic properties)**. Queste ultime rappresentano una modalità di dichiarazione molto compatta che ci evita di dover dichiarare il campo privato, reso accessibile tramite la proprietà, e di dover implementare gli accessor per la scrittura e la lettura in modo esplicito. In questo tipo di dichiarazione è il compilatore che crea in modo totalmente trasparente per lo sviluppatore sia il campo privato, sia il contenuto degli accessor della proprietà.

L'[esempio 4.4](#) mostra un semplice caso di utilizzo della sintassi per le proprietà automatiche, nella definizione della classe `Person` vista in precedenza. Fatte salve le inizializzazioni, il codice è in tutto e per tutto equivalente a quello riportato nell'[esempio 4.3](#). Possiamo peraltro notare

come, in questo secondo caso, la compattezza e l'essenzialità del codice siano molto maggiori, con un conseguente miglioramento della leggibilità.

Esempio 4.4

```
public class Person
{
    public string FullName { get; set; }
    public int Age { get; set; }
}
```

C# 6 ha introdotto, a fianco delle proprietà automatiche, una funzionalità nota come **auto-property initializer** (inizializzatori per proprietà automatiche), che consente di scrivere un codice ancora più compatto e di inizializzare, in un colpo solo, le proprietà, come possiamo vedere nell'[esempio 4.5](#).

Esempio 4.5

```
public class Person
{
    public string FirstName { get; set; } = "Daniele";
    public string LastName { get; set; } = "Bohicchio";
    public int Age { get; set; } = 40;
    public bool IsActive { get; } = true;
    public string LastName => $"{FirstName} {LastName}";
}
```

Contestualmente a questa sintassi, ne è stata aggiunta una nuova che è valida solo per le proprietà dotate di getter. Nel nostro [esempio 4.5](#), la quarta proprietà ha solo il getter: in questo caso, viene creato comunque un backing field, che è implicitamente dichiarato come `readonly` (caratteristica che lo rende inizializzabile nel costruttore).

Particolarmente interessante è la sintassi dell'ultima proprietà, che è un'altra novità di C# 6. Si tratta di una nuova sintassi, che fa uso delle lambda (che sono funzioni espresse in linea e che vengono approfondite in maniera specifica nel [Capitolo 6](#)). Consente di definire subito una funzione associata al getter, evitando di specificare tutto il codice che sarebbe necessario in questi scenari e risultando molto più compatta e immediata da comprendere (nell'esempio, poi, abbiamo fatto uso anche dell'interpolazione di stringhe, presentata nel capitolo precedente). Questa sintassi, peraltro, può essere

applicata anche ai metodi, qualora il risultato sia riconducibile a un'espressione esprimibile come lambda.

Livelli di accessibilità

Negli esempi precedenti, le parole chiave come `private` e `public` rappresentano i cosiddetti **access modifier** o **modificatori di accessibilità**, che hanno lo scopo di definire il grado di visibilità di un determinato elemento.

In base a quanto detto relativamente all'incapsulamento all'inizio del capitolo, le classi e i loro membri (campi, proprietà e metodi) possono avere un diverso grado di accessibilità. Questo significa che i membri di una classe possono essere mascherati e resi invisibili agli altri oggetti. L'insieme degli elementi accessibili è detto **interfaccia della classe**. Esistono diversi livelli di accessibilità, a ciascuno dei quali corrisponde una parola chiave specifica, come mostrato nella [Tabella 4.1](#).

Tabella 4.1 – Modificatori di accessibilità in C#

Parola chiave	Visibilità
<code>public</code>	Da tutte le classi
<code>protected</code>	Solo dalle classi derivate
<code>private</code>	Non accessibile
<code>internal</code>	All'interno dell'assembly
<code>protected internal</code>	Sia da una sottoclasse sia da classi derivate
<code>private protected</code>	Solo da una sottoclasse contenuta nello stesso assembly di origine.

Il modificatore di accessibilità per una classe si riferisce alla sua visibilità in relazione all'assembly di appartenenza. Una classe pubblica può essere richiamata anche dagli oggetti presenti in altri assembly; una classe definita come `internal` è accessibile unicamente all'interno dell'assembly di appartenenza. Gli access modifier sono facoltativi. In caso di omissione, i membri di una classe vengono considerati di tipo privato, mentre le classi diventano visibili unicamente all'interno dell'assembly di appartenenza.

È buona norma definire sempre i campi di una classe come privati e renderli, eventualmente, accessibili dall'esterno tramite una proprietà corrispondente. Del resto, la presenza all'interno del linguaggio delle proprietà automatiche facilita questo approccio.

Come comportamento predefinito, gli accessor di una proprietà hanno lo stesso livello di accessibilità indicato per il membro di classe cui fanno riferimento. Possiamo peraltro differenziare i livelli di accessibilità dei due blocchi, in modo tale che risultino diversi. Nell'[esempio 4.6](#), l'operazione di lettura è pubblica, mentre l'operazione di scrittura ha un livello di accesso limitato.

Esempio 4.6

```
public string FullName
{
    get;
    private set;
}
```

I modifier possono essere combinati, fintanto che non vanno in conflitto. A partire da C# 7.2, è possibile utilizzare `private protected`: in questo caso, la classe (o un membro di una classe) è accessibile solo da una sottoclasse contenuta nello stesso assembly di origine.

[Creazione delle istanze di classe](#)

Le istanze delle classi sono gli oggetti definiti nell'introduzione del capitolo. Dal momento che, per ogni oggetto, il Common Language Runtime alloca un'area di memoria distinta nel managed heap, ciascuno di essi è caratterizzato da un'identità univoca, da uno stato particolare (memorizzato nei campi) e da comportamenti specifici (metodi). Questo significa che oggetti diversi di una stessa classe mantengono in memoria copie differenti dei dati (quindi presentano uno stato distinto da tutti gli altri oggetti simili), anche se, in generale, condividono gli stessi comportamenti.

Un oggetto viene istanziato invocando un metodo particolare, che prende il nome di **costruttore**. Il costruttore di default non presenta parametri, ma per una classe è possibile definire più costruttori, ciascuno caratterizzato da una firma differente. Questo è possibile in quanto, all'interno delle classi, i metodi

possono essere soggetti a **overloading**, ovvero possono esistere più metodi con lo stesso nome, ma con parametri diversi. Come già anticipato nel corso del [Capitolo 3](#), la condizione di overloading è applicabile a tutti i metodi di una classe in base alla firma. L'[esempio 4.7](#) mostra un caso di overloading del costruttore.

Il costruttore è un tipo particolare di procedura (non presenta un valore di ritorno) ed è contrassegnato dallo stesso nome della classe cui si riferisce (non occorre utilizzare la keyword `void`). Come abbiamo visto nel capitolo precedente, parlando della dichiarazione di variabili e oggetti, esso viene richiamato durante la creazione di un'istanza di classe, utilizzando la parola chiave `new` ([esempio 4.7](#)).

Esempio 4.7

```
public class Person
{
    private string _fullName;
    private readonly int _age;

    // Costruttore di default
    public Person()
    {
        _fullName = "Sconosciuto";
        _age = 18;
    }

    // Costruttore con parametri
    public Person(string name, int age)
    {
        _fullName = name;
        _age = age;
    }
}

Person x = new Person();
Person y = new Person("Daniele Bochicchio", 40);
```

In una classe possiamo utilizzare una o più costanti, secondo le regole sintattiche già esposte nel [Capitolo 3](#). In aggiunta possiamo definire un campo come **readonly** (la parola chiave è `readonly`), che può essere inizializzato con un valore, non necessariamente costante, unicamente all'interno di un costruttore ([esempio 4.7](#)). Una volta che una classe viene istanziata, i suoi campi `readonly` non sono più modificabili.

*Un altro tipo particolare di metodo è il **distruttore**. Questo metodo viene richiamato ogni qualvolta un oggetto deve essere finalizzato.*

Peraltro, essendo la distruzione di un'istanza non deterministica, in quanto gli oggetti obsoleti contenuti nel managed heap vengono rimossi in modo autonomo e non prevedibile dal Garbage Collector, il distruttore è un metodo che solitamente non è necessario definire. D'altra parte, questa regola viene meno nel momento in cui una classe utilizza risorse non gestite come, per esempio, handle Win32. In questi casi, le risorse unmanaged devono essere finalizzate in modo esplicito, dal momento che il loro rilascio non può essere effettuato dal Garbage Collector in modo automatico.

Internamente ai metodi e alle proprietà di una classe, possiamo utilizzare la keyword `this` come riferimento all'istanza corrente creata tramite l'invocazione di un costruttore. L'uso di questa parola chiave permette, in molti casi, di migliorare la leggibilità del codice e, al tempo stesso, di eliminare qualsiasi possibile ambiguità, in caso di omonimia degli identificatori. Nell'[esempio 4.8](#) la keyword `this` viene utilizzata sia nel costruttore, sia nel metodo d'istanza per richiamare la proprietà `FullName`.

Esempio 4.8

```
public class Person
{
    public string FullName { get; set; }

    public Person(string name)
    {
        this.FullName = name;
    }

    public string GetFirstName()
    {
        return this.FullName.Split(' ')[0];
    }
}

Person x = new Person("Daniele Bochicchio"); // Creazione dell'istanza
string y = x.GetFirstName(); // y vale "Daniele"
```

In un'ottica di riduzione della verbosità del codice, C# prevede una sintassi molto compatta per l'inizializzazione di oggetti, array e collezioni (queste ultime verranno trattate in modo specifico nel prossimo capitolo). Per l'inizializzazione degli oggetti dobbiamo utilizzare la parola chiave `new`,

seguita dal nome del tipo e da un elenco di coppie proprietà/valore, separate da una virgola e racchiuse in una coppia di parentesi graffe.

L'[esempio 4.9](#) riporta la creazione di un'istanza della classe `Person` con inizializzazione delle proprietà. Questa forma sintattica compatta è del tutto equivalente all'invocazione del costruttore della classe `Person`, seguita dall'inizializzazione di ciascuna proprietà con il valore corrispondente.

Esempio 4.9

```
// Sintassi di creazione con inizializzazione delle proprietà
Person p = new Person { FullName = "Daniele Bochicchio", Age = 40 };

// Sintassi equivalente: costruttore di default ed inizializzazione
Person y = new Person();
y.FullName = "Daniele Bochicchio";
y.Age = 40;
```

Anche per gli oggetti enumerabili, come gli array e le collezioni, possiamo utilizzare una sintassi d'inizializzazione compatta. In questo caso essa si riferisce alla definizione degli elementi costituenti.

In C# l'inizializzazione di un oggetto enumerabile è composta dall'insieme degli inizializzatori di ciascun elemento, separati da una virgola. La sequenza degli elementi è racchiusa in una coppia di parentesi graffe ed è preceduta dalla dichiarazione del tipo enumerabile.

L'[esempio 4.10](#) mostra la sintassi per inizializzare un array di elementi di tipo `Person`.

Esempio 4.10

```
var persons = new Person[]
{
    new Person { FullName = "Cristian Civera", Age = 38 },
    new Person { FullName = "Daniele Bochicchio", Age = 40 }
};
```

C# permette di usare la keyword `new` anche nella definizione di **tipi anonimi** (detti pure **anonymous type**), caratterizzati dal fatto di non avere un nome identificativo dichiarato esplicitamente (il nome viene comunque generato automaticamente, in fase di compilazione). Un anonymous type deriva

direttamente dalla classe `System.Object` ed è composto semplicemente da un insieme di proprietà.

Per definire un anonymous type, dobbiamo utilizzare la stessa sintassi già vista poco sopra nel caso degli inizializzatori di oggetti ([esempio 4.11](#)). La sintassi di creazione è, infatti, caratterizzata dalla dichiarazione di una serie di proprietà, il cui tipo viene ricavato tramite type inference in base all'espressione di inizializzazione.

Esempio 4.11

```
// definizione di una coordinata spaziale
var point = new { X = 1, Y = 2, Z = 3 };

// definizione implicita da contesto
Points[] ps = { new (1, 2, 3), new {3, -2, 0}, new {1, 2, 3} }; // tutti Point
```

La seconda variante, che è una novità di C# 8, consente di scrivere codice più compatto quando il compilatore riesce a individuare il tipo dal contesto (in questo caso, dalla dichiarazione che lo precede).

Possono esistere più variabili assegnate a un anonymous type e corrispondenti allo stesso tipo ([esempio 4.12](#)). Infatti, due dichiarazioni di anonymous type producono lo stesso risultato se nelle due definizioni il nome e il tipo di ciascuna proprietà corrispondono rispettando l'ordine di apparizione. In tal caso è possibile eseguire l'assegnazione tra le due variabili dichiarate come anonime.

Esempio 4.12

```
var point1 = new { X = 21, Y = 10, Z = 71 };
var point2 = new { X = 18, Y = 8, Z = 5 };

// La seconda proprietà di point3 è di tipo string
var point3 = new { A = 20, B = "1", C = 67 };

point1 = point2; // Assegnazione valida (stesso tipo anonimo)
point1 = point3; // Errore (type mismatch)
```

Nell'[esempio 4.12](#), i due tipi anonimi relativi alle variabili `point1` e `point3` differiscono tra loro in quanto non esiste un'esatta corrispondenza tra i tipi

degli elementi costituenti. Un tentativo di assegnazione tra due istanze relative a tipi anonimi diversi produce un errore in fase di compilazione.

Classi statiche e parziali

Finora abbiamo parlato solamente degli elementi associati a una particolare istanza di una classe (detti **membri d'istanza**). Oltre a questi ultimi, in una classe, possiamo definire anche un insieme di elementi associati direttamente al tipo e condivisi da tutte le sue istanze (detti **membri statici**).

Classi statiche

I metodi, i campi e le proprietà statiche possono essere utilizzati senza dover invocare il costruttore per il tipo cui si riferiscono. Essi sono contrassegnati con la keyword `static`, che li identifica come statici e li differenzia dai membri d'istanza ([esempio 4.13](#)).

Esempio 4.13

```
public class Person
{
    public string FullName { get; set; }

    public Person(string name)
    {
        this.FullName = name;
    }

    // Metodo d'istanza
    public string GetFirstName()
    {
        return this.FullName.Split(' ')[0];
    }

    // Metodo statico
    public static string GetFirstName(string name)
    {
        return name.Split(' ')[0];
    }
}

string x = Person.GetFirstName("Crisian Civera");
```

I membri statici possono essere contenuti in una classe indipendentemente dal fatto che, in essa, siano presenti anche membri d'istanza. Una classe che

contiene unicamente membri statici, viene anch'essa indicata come **statica**.

Per definire una classe statica, è sufficiente contrassegnare la classe con la keyword `static` e dichiarare tutti i suoi membri come statici ([esempio 4.14](#)).

Esempio 4.14

```
public static class Calculator
{
    public static int Sum(int x, int y)
    {
        // Ritorna la somma dei due numeri
        return x + y;
    }

    public static int Subtract(int x, int y)
    {
        // Ritorna la differenza tra i due numeri
        return x - y;
    }
}

int i = Calculator.Sum(18, 8);
int j = Calculator.Subtract(18, 8);
```

Come possiamo notare negli esempi precedenti, per accedere ai membri statici di un tipo o agli elementi di un'istanza, dobbiamo utilizzare il carattere “.” (punto), seguito dal nome identificativo del membro e, nel caso dei metodi, dall'elenco dei parametri inclusi in una coppia di parentesi tonde.

Un tipo particolare di metodo statico è il cosiddetto **extension method**. Gli extension method possiedono la particolarità di poter essere invocati per un particolare tipo, come se fossero metodi di istanza, e permettono di estendere classi preesistenti con metodi aggiuntivi, sia che i tipi a cui essi si riferiscono siano direttamente inclusi nella Base Class Library di .NET sia che siano tipi personalizzati creati dallo sviluppatore.

Per dichiarare un extension method, in C# dobbiamo modificare la firma di un metodo statico, aggiungendo prima del primo parametro la parola chiave `this`. Il tipo del primo parametro rappresenta la classe che viene estesa; il metodo statico diventa parte integrante dell'interfaccia e può essere invocato come un normale metodo di istanza.

L'[esempio 4.15](#) mostra l'estensione del tipo `System.String` con l'aggiunta del metodo `IsNull`. L'istruzione `x.IsNull()` equivale alla chiamata del metodo statico `Extensions.IsNull(x)`.

Esempio 4.15

```
public static class Extensions
{
    public static bool IsNull(this string value)
    {
        return value == null;
    }
}

string x = null;    // Stringa nulla
bool y = x.IsNull();    // y vale true
```

C# 6 ha introdotto un nuovo costrutto, chiamato `using static`, che consente di importare, così come avviene con i namespace (lo abbiamo illustrato nel capitolo precedente) tutti i metodi di una classe statica. Nell'[esempio 4.16](#) viene mostrato come fare.

Esempio 4.16

```
using System;    // namespace
using static System.Console;    // classe statica

class program
{
    static void Main()
    {
        WriteLine("Hello world");    // equivale a Console.WriteLine
    }
}
```

Questa sintassi è particolarmente utile quando abbinata agli `extension method`, perché consente di importare solo quelli che si vogliono specificamente utilizzare, anziché tutti quelli contenuti all'interno di un namespace.

Partial class

Un aspetto interessante delle classi è rappresentato dal fatto che esse possono essere definite come **parziali**. La dichiarazione di una classe può essere, infatti, distribuita in più file separati, ciascuno dei quali può includere un sottoinsieme dei membri. La classe deve essere contrassegnata con la parola chiave `partial` per indicare che si tratta di una dichiarazione parziale e non completa ([esempio 4.17](#)). L'omissione della parola chiave comporta un errore

in fase di compilazione, in quanto, in questo caso, al compilatore risultano essere presenti due tipi aventi lo stesso nome identificativo. Durante la fase di compilazione, infatti, le varie dichiarazioni parziali vengono unite a formare un'unica entità finale.

Esempio 4.17

```
public partial class Person
{
    public void Write()
    {
        // ...
        OnWrite();
        // ...
    }

    // Dichiarazione del metodo parziale privato
    partial void OnWrite();
}

public partial class Person
{
    partial void OnWrite()
    {
        // Implementazione
    }
}
```

Anche un metodo privato di una classe parziale può essere dichiarato come parziale, ma solo se è una procedura che non ritorna un valore. In questo caso, una parte della classe contiene solamente la definizione, mentre l'altra può contenere, facoltativamente, l'implementazione ([esempio 4.17](#)). Nel caso in cui un metodo parziale non venga implementato, il metodo stesso e tutte le sue chiamate vengono rimosse in fase di compilazione.

Ereditarietà e polimorfismo

In C# l'ereditarietà multipla non è contemplata. Per questo motivo una classe può derivare unicamente da una sola classe base. Questo aspetto non rappresenta tuttavia un vincolo di particolare rilievo: l'**ereditarietà singola** risulta essere decisamente sufficiente nella maggior parte dei casi.

Il legame di ereditarietà va indicato con il carattere ":" (due-punti), posto subito dopo la dichiarazione della classe e seguito dal nome del tipo base.

L'[esempio 4.18](#) mostra la dichiarazione della classe `Employee`, che eredita dalla classe base `Person`.

Esempio 4.18

```
// Employee (impiegato) deriva da Person (persona)
public class Employee : Person
{
    // ...
}
```

I membri pubblici e protetti del tipo base sono accessibili anche nelle classi derivate. Inoltre, tutte le forme di visibilità, ovvero gli access modifier di ogni membro del tipo base, vengono ereditate, a meno che esse non siano reimpostate diversamente nella classe derivata.

I membri statici, in quanto associati al tipo e non alle istanze, non vengono derivati, indipendentemente dal loro livello di accessibilità. Una classe statica non può essere derivata.

Oltre alla keyword `this`, che identifica l'istanza corrente, in C# possiamo usare la parola chiave `base` come riferimento al tipo base nelle classi derivate. Questa parola chiave torna molto utile per richiamare in modo esplicito i metodi e, in generale, i membri pubblici o protetti del tipo base nei metodi e negli accessor delle proprietà di una classe derivata ([esempio 4.19](#)).

Esempio 4.19

```
public class Employee : Person
{
    private string _firstName;

    public Employee(string name, int age)
    {
        base.FullName = name;
        base.Age = age;
        _firstName = base.GetFirstName();
    }
}
```

In base a come viene dichiarata, una classe può non essere derivabile oppure può non essere istanziabile direttamente. Una classe che non può essere

derivata deve essere contrassegnata con la keyword `sealed` ([esempio 4.20](#)).

Esempio 4.20

```
// Customer (cliente) deriva da Person e non può avere classi derivate public
sealed class Customer : Person
{
    // ...
}
```

Una classe che non può essere direttamente istanziata e che, quindi, deve essere obbligatoriamente derivata, si dice **astratta**. Per dichiarare una classe astratta, dobbiamo specificare la parola chiave `abstract`. Le classi astratte possono avere opzionalmente tutti i metodi e le proprietà o un loro sottoinsieme definiti anch'essi come astratti.

Un **membro astratto** è un elemento della classe per il quale viene semplicemente riportata la dichiarazione insieme alla parola chiave `abstract` ([esempio 4.21](#)). Per esso, infatti, non viene specificata l'implementazione, che deve essere obbligatoriamente fornita nelle classi derivate.

Esempio 4.21

```
// Person è una classe astratta e deve essere derivata
public abstract class Person
{
    // Metodo astratto
    public abstract GetFirstName();

    // Proprietà astratta
    public abstract string FullName { get; set; }
}
```

Come abbiamo avuto modo di dire nella prima parte del capitolo, il polimorfismo rappresenta la possibilità di ridefinire le proprietà e i metodi nelle classi derivate rispetto a quelli dichiarati nel tipo base (**overriding**). Affinché possa essere polimorfico, un membro deve essere marcato come **astratto** oppure come **virtuale**.

Dato che nel tipo base un membro astratto riporta unicamente la sua dichiarazione, è chiaro che esso debba essere implementato ogni volta in ciascuna delle classi derivate. Questo implica che ogni classe derivata è

caratterizzata da comportamenti simili, che devono peraltro adottare necessariamente diverse strategie implementative.

A differenza di quanto succede per i membri astratti, nel caso dei membri virtuali, esiste sempre nella superclasse un'implementazione di base. Peraltro, un membro virtuale può essere ridefinito in modo personalizzato nelle classi derivate. Questo significa che, oltre all'implementazione di base, possono esistere implementazioni diverse dello stesso membro, ciascuna delle quali afferente a una diversa classe derivata.

Se per proprietà e metodi astratti è sempre obbligatorio definire un'implementazione nella classe derivata, questo non è vero per i membri virtuali: nel caso in cui un membro virtuale non venga ridefinito nel tipo derivato, rimane valida la versione presente nella superclasse.

I campi di una classe non possono essere soggetti a overriding né definiti come astratti (del resto contengono solamente dati, non definiscono comportamenti). Gli unici membri che possono essere polimorfici sono i metodi e le proprietà.

La keyword che permette di contrassegnare un membro come virtuale è `virtual` ([esempio 4.22](#)). Per eseguire l'overriding di un membro in una classe derivata, dobbiamo invece specificare la parola chiave `override` ([esempio 4.22](#)). Per indicare che un membro non può essere ulteriormente soggetto a overriding nelle classi derivate, dobbiamo infine specificare la keyword `sealed`.

*La parola chiave `new` permette di nascondere in modo esplicito un membro ereditato da una classe base. Questo significa che la versione derivata del membro sostituisce la versione della classe base. Lo scopo principale dello **shadowing** è, infatti, quello di proteggere la definizione dei membri di una classe. Se nel tipo base viene aggiunto un nuovo elemento con il nome uguale a quello del membro già definito nella classe derivata, la keyword `new` impone che i riferimenti attraverso la classe vengano comunque risolti nel membro della classe derivata anziché nel nuovo elemento della superclasse.*

È sempre bene ricordare che un membro virtuale viene risolto in base al tipo dell'istanza su cui viene richiamato, non in funzione del tipo del riferimento.

L'[esempio 4.22](#) illustra questo importante concetto, illustrando una casistica significativa di invocazione di un metodo virtuale.

Esempio 4.22

```
public class Person
{
    // Metodo virtuale
    public virtual string GetFirstName()
    {
        // Implementazione di base del metodo
    }
}

public class Employee : Person
{
    public Employee(string name, int age)
    {
        // ...
    }

    public override string GetFirstName()
    {
        // Nuova implementazione del metodo
    }
}

Employee x = new Employee("Matteo Tumiatì", 28);
Person y = x;

// Viene comunque eseguito il metodo di Employee
// anche se il riferimento è di tipo Person
string z = y.GetFirstName();
```

Nell'[esempio 4.22](#) la variabile `y` di tipo `Person` punta a un'istanza della classe derivata `Employee`. L'invocazione del metodo virtuale `GetFirstName` produce l'esecuzione dell'implementazione presente in `Employee` indipendentemente dal tipo del riferimento (ovvero la variabile `y`).

Interfacce

Un'**interfaccia** è un tipo simile a una classe astratta pura, ossia composta solamente da metodi e da proprietà astratte. Essa è, infatti, priva di qualsiasi implementazione, dato che il suo scopo è semplicemente quello di definire un contratto valido per le classi che la vanno a implementare.

Il grosso vantaggio nell'utilizzo delle interfacce è rappresentato dal fatto che una classe può implementare più di un'interfaccia contemporaneamente.

Questo aspetto va a compensare, almeno parzialmente, la mancanza dell'ereditarietà multipla per le classi.

Per definire un'interfaccia, dobbiamo specificare la parola chiave `interface` seguita dal nome identificativo, evitando di riportare gli access modifier per i suoi elementi. Per ognuno di essi è, infatti, sufficiente inserire unicamente la dichiarazione, omettendo qualsiasi forma d'implementazione ([esempio 4.23](#)). In un'interfaccia possono essere incluse dichiarazioni di metodi, proprietà ed eventi.

Esempio 4.23

```
// Interfaccia che definisce un metodo per la scrittura
public interface IWritable
{
    void Write();
}
```

L'implementazione di un'interfaccia in C# segue le stesse regole sintattiche utilizzate nel caso dell'ereditarietà: il tipo base e le interfacce implementate vengono indicate nello stesso elenco. Nel caso di implementazione multipla, dobbiamo usare il carattere “,” (virgola) per separare tra loro le diverse interfacce che sono associate alla classe.

Oltre all'implementazione, l'[esempio 4.24](#) riporta anche una semplice casistica di utilizzo. Come possiamo notare, ogni istanza relativa a una classe che implementa un'interfaccia può essere assegnata a una variabile dell'interfaccia stessa. In questo caso, i membri che possono essere richiamati sono solamente quelli associati all'interfaccia e non quelli esposti dalla classe che implementa l'interfaccia stessa. Per poter invocare i membri della classe aggiuntivi rispetto a quelli dell'interfaccia, dobbiamo necessariamente effettuare un'operazione di casting al tipo della classe concreta.

Esempio 4.24

```
// Deriva dalla classe base Person
// Implementa l'interfaccia IWritable
public class Employee : Person, IWritable
{
    public void Write()
    {
        Console.Write(base.FullName);
    }
}
```

```

}

// Il riferimento è di tipo IWritable, ma l'istanza è di tipo Employee
IWritable x = new Employee("Stefano Mostarda", 40);

// Scrive il nome completo
x.Write();

// È necessario effettuare il casting per invocare il metodo
string y = ((Employee)x).GetFirstName();

```

A questo punto può sorgere una domanda: quando usare le interfacce e quando, invece, utilizzare le classi astratte pure o parzialmente implementate?

Abbiamo detto che entrambe definiscono un contratto per le classi alle quali sono associate, dato che, internamente, non contengono implementazioni, ma solo dichiarazioni. Peraltro, le classi astratte pure forniscono un tipo di contratto “più forte”, in quanto, oltre a definire i comportamenti per le classi associate, ne rappresentano anche il tipo base.

Diversamente le interfacce permettono una maggiore flessibilità, in quanto sono tipi indipendenti e trasversali rispetto alla gerarchia delle classi definita tramite i legami di ereditarietà. L’uso delle interfacce è quindi da preferire nel caso in cui vogliamo definire contratti di natura generale, che possano essere usati indipendentemente dai legami di ereditarietà e che non impongano comportamenti specifici ed esclusivi.

C# 8 introduce il supporto ai **default interface methods**, cioè ai metodi di default nelle interfacce.

Riprendendo gli esempi 4.23 e 3.24, possiamo scrivere una nuova interfaccia da utilizzare come base per implementare un sistema di logging, come nell’esempio che segue.

Esempio 4.25

```

// Interfaccia che definisce un metodo per la scrittura
public interface ILogger
{
    void Log(string what, string category);
    void Log(string what) => Log(what, "Default");
}

```

Nell’[esempio 4.25](#) abbiamo aggiunto un metodo che accetta due parametri, senza implementazione e, contestualmente, abbiamo aggiunto una variante che richiama questo metodo, passando un valore dei default. Benché sia possibile aggiungere anche implementazioni complete, che fanno riferimento a una

qualsiasi classe, questo è considerato scorretto dal punto di vista dell'OOP e andrebbe evitato.

Questa funzionalità è disponibile solo con .NET Core 3 e non con .NET Framework 4.8. I tipi come le interfacce e le classi astratte, che hanno una valenza soprattutto dichiarativa, vengono genericamente identificati come **tipi astratti**, in contrapposizione alle classi che contengono le implementazioni vere e proprie, che vengono denotate come **tipi concreti**.

Strutture

Oltre alle enumerazioni e alle classi, lo sviluppatore può definire in modo personalizzato anche i tipi di valore, attraverso le **strutture** o **struct**.

Una struttura è un aggregato di dati simile a una classe, dal momento che può contenere campi, metodi, proprietà ed eventi. La differenza rispetto a una classe risiede principalmente nel modo in cui una struttura viene mantenuta in memoria: essendo un tipo di valore, essa non viene inserita nel managed heap, come avviene per le classi, ma viene allocata direttamente sullo stack.

Nel Common Language Runtime (CLR), il Garbage Collector ha lo scopo di gestire automaticamente la memoria. Il managed heap è un'area gestita dal Garbage Collector, in cui quest'ultimo alloca e gestisce gli oggetti. Quest'area è chiamata managed heap (poiché utilizzata da codice managed, quello gestito dal CLR), in contrapposizione al native heap, che è l'area gestita dal sistema operativo.

Quanto abbiamo detto finora per le classi rimane generalmente valido anche per la definizione di strutture. Peraltro, le struct presentano alcune limitazioni rispetto alle classi. La [Tabella 4.2](#) riassume le principali differenze esistenti tra classi e strutture.

Tutti i tipi primitivi di valore che sono stati trattati sono strutture. Per esempio, System.Int32, System.DateTime, System.Boolean oppure System.Decimal sono struct.

Tabella 4.2 – Differenze tra classi e strutture

Classi	Strutture
--------	-----------

Possono definire campi, proprietà, metodi ed eventi.	Possono definire campi, proprietà, metodi ed eventi.
Supportano tutti i tipi di costruttori e l'inizializzazione dei membri.	Supportano tutti i tipi di costruttori (novità di C# 6) e l'inizializzazione dei membri.
Supportano il metodo <code>Finalize</code> , ovvero il distruttore.	Non supportano il metodo <code>Finalize</code> .
Supportano l'ereditarietà.	Non supportano l'ereditarietà.
Sono tipi di riferimento.	Sono tipi di valore.
Vengono allocati nel managed heap.	Vengono allocati sullo stack.

Come possiamo notare nella tabella riepilogativa, le strutture non possono derivare da altre strutture né essere un tipo base per altri tipi. Da C# 6 le struct supportano anche il costruttore di default. Come abbiamo fatto in precedenza, possiamo dichiarare, in modo personalizzato, costruttori con parametri in overload a quello di default.

Per definire una struttura, dobbiamo specificare la parola chiave `struct`. La dichiarazione segue le stesse regole sintattiche viste per le classi nel corso del capitolo ([esempio 4.26](#)).

Esempio 4.26

```
// Definizione di una struttura per i numeri complessi
public struct ComplexNumber
{
    public float Real { get; set;}    // Parte reale
    public float Imaginary { get; set;} // Parte immaginaria

    // Costruttore con parametri
    // Deve essere richiamato esplicitamente il costruttore di
    // default per poter utilizzare le proprietà automatiche
    public Complex(float r, float i) : this()
    {
        this.Real = r;
        this.Imaginary = i;
    }

    // Metodo statico per la somma di due numeri complessi
    public static ComplexNumber Sum(Complex x, Complex y)
    {
        return new ComplexNumber(x.Real + y.Real, x.Imaginary + y.Imaginary);
    }
}

Complex u = new ComplexNumber(); // u vale 0.0 + 0.0i
Complex v = new ComplexNumber(1.0F, 2.0F); // v vale 1.0 + 2.0i
```

L'[esempio 4.26](#) si riferisce alla dichiarazione di una struttura per i numeri complessi. In essa è presente un costruttore con due parametri per la parte reale e per la parte immaginaria del numero. Questo costruttore può essere invocato per inizializzare in modo mirato le due proprietà della struct. In alternativa è comunque possibile invocare il costruttore di default; in questo caso i valori assegnati alla parte reale e alla parte immaginaria del numero complesso sono pari a zero (ovvero il valore di default per il tipo `float`).

Al di là delle differenze riportate nella [Tabella 4.2](#), quando è opportuno utilizzare una struttura al posto di una classe? Le strutture nascono come tipi orientati a gestire poche informazioni di breve durata, per lo più contenute, a loro volta, in altri oggetti. Esse devono rappresentare principalmente valori singoli o tipi primitivi (per esempio, valori numerici più o meno strutturati, coordinate spaziali, ecc.), caratterizzati da una dimensione ridotta (in genere 16 byte) e immutabili. In tutti gli altri casi vanno usate le classi.

Overload degli operatori

L'[esempio 4.26](#) presenta in realtà un problema: per sommare due istanze della classe che rappresenta i numeri reali, è necessario creare un'istanza che sommi separatamente la parte reale e quella complessa, creando una nuova istanza che contenga i nuovi valori.

C# ci consente di definire un comportamento speciale per gli operatori, così che sia possibile, per esempio, sommare direttamente due istanze della nostra classe che rappresenta i numeri complessi. La caratteristica di definire un comportamento personalizzato per gli operatori è nota come `operator overloading` o anche come `overload degli operatori`.

Perché sia possibile utilizzare la keyword `overload`, che rappresenta la dichiarazione di un operatore, bisogna che lo stesso sia marcato sia come `public` sia come `static`. Gli operatori unari accettano un solo parametro, quelli binari due e almeno uno deve essere di tipo `T` o `T?`.

Possiamo riscrivere quindi l'[esempio 4.26](#), aggiungendo il comportamento mancante ([esempio 4.27](#)).

Esempio 4.27

```
// Definizione di una struttura per i numeri complessi
public struct ComplexNumber
{
```

```

// ... aggiungere il codice preso dall'esempio precedente
public static Complex operator +(Complex x, Complex y)
{
    return new Complex(x.Real + y.Real, x.Imaginary + y.Imaginary);
}

public static Complex operator -(Complex x, Complex y)
{
    return new Complex(x.Real - y.Real, x.Imaginary - y.Imaginary);
}
}

Complex a = new Complex();    // vale 0.0 + 0.0i
Complex b = new Complex(1.0F, 2.0F);    // vale 1.0 + 2.0i
Complex c = a + b;    // vale 1.0 + 2.i

```

Questo tipo di approccio è utilizzabile anche per gestire conversioni tra tipi: sebbene in questo scenario non avrebbe molto senso, è il sistema attraverso il quale è possibile sommare un `int` e un `float` e ottenere comunque un `float` come tipo risultante.

Questa funzionalità si può applicare a tutti gli operatori che abbiamo introdotto in precedenza. Possiamo definire gli operatori che desideriamo, ma alcuni (`==` e `!=`, `<` e `>`, `<=` e `=>`) vanno sempre definiti in coppia.

Tuple

Le tuple in C# rappresentano una sintassi semplificata, più leggera, che offre vantaggi come regole per la conversione basate sulla cardinalità e sui tipi degli elementi, nonché regole consistenti per copia, uguaglianza e assegnazioni. Non supportano alcune delle funzionalità OOP, tipicamente associate all'ereditarietà, e sono state notevolmente potenziate a partire da C# 7.

Benché il .NET Framework includesse già un tipo `Tuple` (il cui limite principale è quello di avere delle proprietà di nome fisso, oltre al fatto di essere una classe, quindi un `reference type`), è necessario assicurarsi di avere una reference al package `System.ValueTuple`, che è disponibile anche per tutte le incarnazioni di .NET. In questo assembly si trovano una serie di `struct` (quindi, dei `value type`), tra cui anche `ValueTuple`, che quindi rappresentano un vantaggio in termini di performance, rispetto all'uso di un `reference type`.

A cosa servono esattamente le tuple? Il caso d'uso migliore è quello di un metodo che restituisce un valore di ritorno, che in realtà all'interno incapsula diversi valori. Questo ci consente di restituire facilmente un risultato senza dover costruire una classe o una `struct`, quindi senza la necessità di aggiungere

comportamenti e dati in un contesto in cui non ne trarremmo giovamento. L'[esempio 4.28](#) include alcune dichiarazioni possibili.

Esempio 4.28

```
var unnamed = ("a", "b");    // valori in .Item1, .Item2
var named = (first: "a", second: "b");    // valori in .first, .second

var a = "a";
var b = "b";
var c = (a, b);    // valori in .a, .b
var d = (e: a, f: b);    // valori in .e, .f
```

In particolare, negli ultimi esempi possiamo notare una funzionalità, nota come tuple projection initializers, che consente di definire dei nomi personalizzati per i field che sono creati in automatico, secondo vari meccanismi. Questa funzionalità automatica non funziona in caso di duplicazione di nomi, oppure in caso di nomi riservati (come per esempio `Item1` o `ToString`). In tutti i casi, è possibile accedere ai valori anche posizionalmente, con la sintassi `Itemx`, dove `x` può essere un valore da 1 in poi.

A partire da C# 7.3, le tuple supportano gli operatori di eguaglianza `==` e `!=`.

Questi operatori comparano tutti i membri della parte sinistra con quelli della parte destra, nell'ordine di definizione.

Per rendere lo sviluppo più semplice, le tuple usano un meccanismo di lifted conversion se uno dei due valori fosse nullo. L'[esempio 4.29](#) presenta una dimostrazione in tal senso.

Esempio 4.29

```
var first = (a: "a", b: "b");
var second = (a: "a", b: "b");
bool result = first == second;    // true

// lifted conversion
(string a, string b)? nullableTuple = second;
bool result2 = first == nullableTuple;    // true

// altre conversioni implicite
var a = (first: 5, second: 4);
(long a, long b) b = (first: 5, second: 4);
bool result3 = a == b;    // true
```

C# consente l'assegnazione di tuple che hanno lo stesso numero di elementi e il cui tipo può essere convertito.

Lo scenario in cui sono più utili, come abbiamo già sottolineato, è come valori di ritorno di un metodo, quando c'è bisogno di più valori da restituire: una funzione, infatti, può comunque avere un solo valore di ritorno e una tupla consente facilmente di onorare questa necessità, aggiungendo in più la possibilità di passare all'interno di una sola struttura tutti i valori di ritorno dovessero essere necessari. L'[esempio 4.30](#) introduce meglio la questione.

Esempio 4.30

```
public static (string[] Results, int Count) GetPagedData(int pageIndex)
{
    var data = new[] { "a", "b", "c", "d", "e", "f", "g" };
    var results = new[] { data[0], data[1], data[2] }; // finta paginazione
    return (results, data.Length);
}

// esempio di utilizzo "classico"
var data = GetPagedData(0);
var count = items.Count;
var results = items.Results;

// esempio di utilizzo con deconstruction
var (results, count) = GetPagedData(0);
```

Di fatto, diventa possibile molto facilmente restituire struct personalizzate, con una sintassi molto semplice. Il nome nella dichiarazione del tipo di ritorno non è obbligatorio.

Infine, è possibile effettuare la deconstruction delle tuple, cioè trasformare il tipo di ritorno all'interno di variabili locali. Questa funzionalità si può applicare anche tra classi differenti e consente di trasformare tuple in tipi e viceversa, definendo un metodo di nome `Deconstruct`, che trovate approfondito su <https://aspit.co/bwt>.

In generale, le tuple sono molto comode per semplificare il codice e, in unione con il pattern matching, rappresentano la parte più funzionale di un linguaggio Object Oriented come C#.

[Regole di nomenclatura](#)

Un aspetto importante, che non dobbiamo sottovalutare quando creiamo e definiamo i nostri tipi personalizzati, è rappresentato dalle scelte di nomenclatura che riguardano in particolare la selezione degli identificatori per le classi, le strutture, i namespace, le variabili locali e i parametri dei metodi.

I nomi assegnati agli identificatori nel codice devono iniziare necessariamente con un carattere alfabetico o il carattere di sottolineatura (underscore) e non possono contenere caratteri speciali come gli elementi di punteggiatura o altro.

Possiamo identificare tre notazioni fondamentali per il naming delle variabili:

- ❑ **notazione ungherese:** al nome dell'identificatore viene aggiunto un prefisso che ne indica il tipo (esempio: `intNumber` identifica una variabile intera);
- ❑ **notazione Pascal:** l'iniziale di ogni parola che compone il nome dell'identificatore è maiuscola, mentre tutte le altre lettere sono minuscole (esempio: `FullName`);
- ❑ **notazione Camel:** come la notazione Pascal, a differenza del fatto che la prima iniziale deve essere minuscola (esempio: `fullName`).

Sebbene sia una pratica ancora diffusa tra gli sviluppatori, **la notazione ungherese è considerata superata**, in quanto si addice poco alle logiche e alle tecniche di progettazione e sviluppo object oriented. Per questo motivo, ne sconsigliamo vivamente l'utilizzo. Del resto, il nome di tutti i tipi in .NET e dei loro membri pubblici seguono la notazione Pascal, mentre, in generale, la notazione Camel viene usata per variabili locali, campi privati e parametri. La [Tabella 4.3](#) riassume le notazioni da applicare per l'assegnazione dei nomi agli identificatori.

Tabella 4.3 – Notazioni da applicare per il naming delle variabili e dei tipi

Elemento/i	Notazione
Namespace	Notazione Pascal
Classi	Notazione Pascal
Interfacce	Notazione Pascal

Strutture	Notazione Pascal
Enumerazioni	Notazione Pascal
Campi privati	Notazione Camel, eventualmente preceduta dal carattere di sottolineatura (esempio: <code>_fullName</code>)
Proprietà, metodi ed eventi	Notazione Pascal
Parametri dei metodi e delle funzioni in generale	Notazione Camel
Variabili locali	Notazione Camel

La convenzione prevede di utilizzare il prefisso “I” (i maiuscola) per il nome identificativo delle interfacce, in modo tale da poterle sempre distinguere senza ambiguità dagli altri tipi e, in particolare, dalle classi.

Conclusioni

La programmazione orientata agli oggetti è un paradigma di programmazione che si basa sulla definizione e sull'utilizzo di una serie di entità collegate tra loro, ciascuna delle quali è caratterizzata da un insieme di informazioni di stato e di comportamenti specifici. Queste entità sono denominate oggetti e ciascuna di esse può contenere contemporaneamente dati, funzioni e procedure.

Le classi, di cui gli oggetti sono istanze particolari, rappresentano gli elementi fondamentali per l'organizzazione e la strutturazione del codice in C#. Le classi sono i tipi di riferimento definiti dallo sviluppatore, contenenti campi, proprietà, metodi ed eventi.

Per le classi valgono i principi fondamentali della programmazione orientata agli oggetti, cioè ereditarietà, polimorfismo e incapsulamento. In C# una classe può derivare solamente da un'altra classe (ereditarietà singola), può ridefinire in modo personalizzato le implementazioni dei suoi membri, anche se ereditati dal tipo base, e può mascherare la sua struttura interna, utilizzando in modo appropriato gli access modifier.

Le interfacce sono tipi astratti privi di qualsiasi implementazione, dato che il loro scopo è semplicemente quello di definire un contratto valido per le classi che le implementano. Il grosso vantaggio nell'utilizzo delle interfacce è rappresentato dal fatto che ciascuna classe può implementare più interfacce contemporaneamente. Questo aspetto compensa la mancanza dell'ereditarietà multipla.

Oltre alle classi e alle interfacce, C# include anche le strutture. Queste ultime rappresentano i tipi di valore definiti dallo sviluppatore e servono per definire tipi primitivi personalizzati. Anche se non supportano l'ereditarietà, le struct sono elementi che, da un punto di vista sintattico, presentano molte similitudini con le classi, dal momento che possono contenere campi, metodi, proprietà ed eventi.

Un tipo particolare di classe sono i tipi enumerabili, come gli array e le collezioni. Già nel corso di questo capitolo e di quello precedente, abbiamo avuto occasione di menzionare questa tipologia di oggetti. Ora è arrivato il momento di affrontare l'argomento nello specifico, all'interno del prossimo capitolo.

Collection e generics

Nei capitoli precedenti abbiamo introdotto i concetti fondamentali del linguaggio C# e abbiamo visto come sia possibile utilizzarlo per rappresentare classi o strutture. Una delle necessità più comuni nello sviluppo di applicazioni è quella di raggruppare oggetti omogenei e, fin dai tempi dei linguaggi procedurali, la soluzione a questo problema è stato il ricorso agli array. In questo capitolo ci spingeremo un passo più avanti, mostrando come il paradigma a oggetti ha permesso di sviluppare contenitori di alto livello, chiamati collection, e come sia possibile sfruttarne la versatilità all'interno delle nostre applicazioni.

A partire dal .NET Framework 2.0, .NET si è arricchito grazie al concetto dei generics che, sebbene molto utilizzati proprio nell'ambito della realizzazione di collezioni di oggetti, offrono in realtà tutta una serie di vantaggi e di prerogative grazie alla semplicità con cui è possibile scrivere codice fortemente tipizzato. Abbiamo dedicato a loro la seconda parte del capitolo.

Introduzione alle collection

Raggruppare informazioni all'interno di contenitori, siano esse numeri, caratteri o oggetti complessi, è da sempre una delle necessità più comuni nell'ambito dello sviluppo di applicazioni. Storicamente presenti fin dai tempi dei linguaggi procedurali, gli array sono sopravvissuti fino ai giorni odierni, tanto che .NET prevede addirittura una classe ad-hoc, chiamata `System.Array`, da cui ereditano tutti gli oggetti (anche un array è, a tutti gli effetti, un oggetto) di questo tipo.

In C# un array può essere definito (e successivamente utilizzato) con i costrutti mostrati nell'[esempio 5.1](#). `System.Array` contiene una serie di metodi e proprietà, sia di tipo sia d'istanza, che possono essere utilizzati per interagirvi.

Esempio 5.1

```
int[] myIntArray = { 1, 5, 2, 3, 6 };

int[] anotherArray = new int[5];
anotherArray[3] = 2;

var length = myIntArray.Length;    // Recupera il numero di elementi
var index =
    Array.IndexOf(myIntArray, 2);    // Indice di un elemento

var item = myIntArray[3];           // Ritorna il quarto elemento

Array.Sort(myIntArray);             // Ordina gli elementi

Array.Resize(ref myIntArray, 7);    // Modifica il numero di elementi
myIntArray[6] = 11;
```

Il numero di elementi che lo compongono è statico e deciso all'atto della sua inizializzazione, utilizzando la notazione esplicita o quella implicita mostrate nell'esempio precedente; nel caso questo debba essere modificato in un momento successivo, è necessario invocare il metodo `Resize`, passando per riferimento la variabile contenente l'array e la nuova dimensione. Proprio questa caratteristica rende una tale struttura particolarmente scomoda da utilizzare quando il numero di elementi non è noto a priori, oppure varia con una certa frequenza. In tutti i casi in cui si abbia la necessità di un contenitore più "snello", cui sia possibile aggiungere o rimuovere elementi con facilità, è molto più comodo utilizzare una **collection**.

.NET le supporta fin dalla sua primissima versione, grazie a una serie di classi e interfacce presenti all'interno del namespace `System.Collections`. Esse rappresentano tutte insiemi del tipo più generico possibile, ossia `System.Object`, e pertanto possono essere utilizzate con qualsiasi oggetto. In realtà, dall'avvento dei **generics**, avvenuto con la versione 2.0 del .NET Framework, esse sono cadute un po' in disuso, ma vale comunque la pena studiarne il funzionamento perché comunque la logica che lo regola è condivisa con le loro controparti generiche.

La classe ArrayList

Tra le varie tipologie di collection disponibili, quella che più si avvicina al concetto di array è la classe `ArrayList`. Essa, infatti, rappresenta una collezione di elementi a cui possiamo accedere tramite un indice numerico, ma il cui numero può variare dinamicamente, come mostrato nell'[esempio 5.2](#).

Esempio 5.2

```
var sample = new ArrayList();    // Creazione di un arraylist
sample.Add(2);                  // Aggiunta di elementi

object value = sample[1];       // Ritorna il secondo elemento

int count = sample.Count;       // Recupera il numero di elementi

sample.Remove(2);               // Rimuove l'intero 2
sample.Clear();                 // Rimuove tutti gli elementi
```

Il primo passo per poter utilizzare un `ArrayList` è quello di crearne un'istanza tramite il suo costruttore. A questo punto, ciò che abbiamo a disposizione è un contenitore vuoto, a cui possono essere aggiunti oggetti e che può essere popolato mediante i metodi `Add` o `AddRange`, che accettano rispettivamente un singolo oggetto o un'altra collezione (per esempio un array, o un altro `ArrayList`) di oggetti. Un elemento di un `ArrayList` può essere recuperato tramite il suo indice oppure rimosso tramite il metodo `Remove`, mentre il contenuto di tutta la lista può essere scorso utilizzando i costrutti `for` o `foreach`, come mostrato dall'[esempio 5.3](#).

Esempio 5.3

```
for (int index = 0; index < sampleList.Count - 1; index++)
{
    Console.WriteLine(sampleList[index]);
}

foreach (object element in sampleList)
{
    Console.WriteLine(element);
}
```

Oltre a quelli indicati, l'oggetto `ArrayList` contiene diversi metodi e proprietà che possono essere utilizzati per manipolarne il contenuto. La [Tabella 5.1](#) contiene un elenco dei più utilizzati.

Tabella 5.1 – Membri della classe `ArrayList`

Metodo/Proprietà	Descrizione	Esempio
<code>Add(item)</code>	Aggiunge un elemento alla lista	<code>myList.Add("A string");</code>
<code>AddRange(items)</code>	Aggiunge un insieme di elementi alla lista	<code>myList.AddRange(new[] {1, 5, 9});</code>
<code>Clear()</code>	Svuota la lista	<code>myList.Clear();</code>
<code>Contains(item)</code>	Verifica se un elemento appartiene alla lista	<code>if (myList.Contains("item"))...</code>
<code>Count</code>	Ritorna il numero di elementi	<code>int items = myList.Count;</code>
<code>IndexOf(item)</code>	Ritorna l'indice della prima occorrenza di un elemento, o -1 nel caso in cui questo non sia presente	<code>int index = myList.IndexOf("item");</code>
<code>Item(index)</code>	Ritorna l'elemento corrispondente all'indice fornito. La proprietà <code>Item</code> può essere eventualmente omessa.	<code>object item = myList.Item[2];</code> oppure <code>object item = myList[2];</code>
<code>LastIndexOf(item)</code>	Ritorna l'indice dell'ultima occorrenza di un elemento, o -1 nel caso in cui questo non sia presente	<code>int index = myList.LastIndexOf(5);</code>
<code>Insert(index, item)</code>	Inserisce un nuovo elemento alla posizione specificata da <code>Index</code>	<code>myList.Insert(0, "First Element");</code>
<code>InsertRange(index,</code>		<code>myList.InsertRange(0, new[] {1,2,3});</code>

items)	Inserisce un insieme di elementi a partire dalla posizione specificata da Index	
Remove(item)	Rimuove la prima occorrenza di un determinato elemento	myList.Remove("A string");
RemoveAt(index)	Rimuove l'elemento il cui indice è pari a quello fornito	myList.RemoveAt(0);
ToArray()	Genera un array a partire dal contenuto dell'ArrayList	object[] myArray = myList.ToArray();

Non sempre, però, accedere a un particolare oggetto tramite il suo indice è sufficiente; spesso, infatti, abbiamo la necessità di gestire delle collezioni di coppie chiave-valore, che in letteratura vengono comunemente chiamate **dizionari** e saranno l'oggetto del prossimo paragrafo.

Dizionari in .NET tramite la classe Hashtable

Esistono molteplici casistiche in cui è utile gestire collezioni di elementi la cui discriminante, invece che un indice numerico, è rappresentata da una qualsiasi altra tipologia di oggetto: pensiamo, per esempio, a una serie di impostazioni di configurazione dell'applicazione, in cui vogliamo associare a una determinata **chiave**, per esempio "UserFontSize", il corrispettivo **valore** "15px". È ovviamente possibile creare una classe con due proprietà, *Key* e *Value*, per poi raggrupparne le istanze in un ArrayList, ma recuperare il valore associato a una data chiave vorrebbe dire scorrere ogni volta tutta la lista fino a identificare l'elemento desiderato.

All'interno del framework e, in particolare, sempre nel namespace System.Collections, trova posto la classe Hashtable che, invece, rappresenta la soluzione ottimale per questo tipo di problematiche, com'è possibile notare nell'[esempio 5.4](#).

Esempio 5.4

```
var myDictionary = new Hashtable();           // Creazione dell'HashTable

myDictionary.Add("someIntValue", 5);         // Aggiunta di elementi
myDictionary.Add("someClass", new StringWriter());
myDictionary.Add(DateTime.Today, "Today's string");

object value =
    myDictionary["someClass"];               // Recupera elemento dalla chiave

myDictionary.Remove("someClass");           // Rimuove un elemento

int count = myDictionary.Count;             // Recupera il numero di elementi
myDictionary.Clear();                       // Rimuove tutti gli elementi
```

A un'istanza di `Hashtable` possono essere aggiunte coppie *chiave-valore* tramite il metodo `Add`. Come possiamo notare, non è definita una particolare tipizzazione e, pertanto, esse possono essere composte da una qualsiasi tipologia di oggetto; attenzione però al fatto che, mentre nessun vincolo è posto sui valori, è invece richiesto che le chiavi siano univoche, pena il sollevamento di un errore a runtime. Una volta inserito un valore all'interno di una `Hashtable`, esso può essere direttamente recuperato, utilizzando la relativa chiave. L'istruzione:

```
object value = myDictionary[key];
```

ritornerà `null` nel caso in cui la chiave specificata non sia presente all'interno del dizionario e, pertanto, è necessario gestire nel codice questa eventualità.

La classe `Hashtable` è così chiamata perché tutte le chiavi sono memorizzate internamente in una struttura basata su hash numerici; ogni oggetto .NET è infatti in grado di generare un proprio hash tramite il metodo `GetHashCode`. Tutto ciò rende l'operazione di ricerca di un elemento estremamente veloce ma, allo stesso tempo, richiede un certo overhead per la generazione dell'hash e per la manutenzione della tabella interna.

Per queste ragioni, nel caso di dizionari con un numero limitato di elementi (inferiore a dieci) è preferibile utilizzare la classe `ListDictionary`, che implementa le medesime funzionalità, ma con una strategia differente. Una terza classe, chiamata

HybridDictionary, ha la capacità di configurarsi internamente prima come un ListDictionary, per poi convertirsi a Hashtable nel momento in cui questa soglia viene superata. Entrambe queste classi appartengono al namespace System.Collection.Specialized.

Invece, quando abbiamo la necessità di scorrere il contenuto di un dizionario nella sua totalità, possiamo utilizzare le tre modalità differenti, mostrate nell'[esempio 5.5](#).

Esempio 5.5

```
// Enumerazione di tutte le coppie chiave-valore
foreach (DictionaryEntry item in myDictionary)
{
    Console.WriteLine(item.Key + " " + item.Value);
}

// Enumerazione di tutte le chiavi
foreach (object key in myDictionary.Keys)
{
    Console.WriteLine(key);
}

// Enumerazione di tutti i valori
foreach (object value in myDictionary.Values)
{
    Console.WriteLine(value);
}
```

In particolare, se applichiamo l'istruzione `foreach` a `Hashtable`, l'oggetto restituito a ogni iterazione è un `DictionaryEntry` che consente di accedere sia alla chiave sia al valore vero e proprio. In altre occasioni, invece, può essere necessario scorrere tutte le chiavi o tutti i valori separatamente e, in questi casi, possiamo utilizzare le proprietà `keys` e `values`.

Queste ultime, insieme con altri importanti membri della classe `Hashtable`, sono sintetizzate nella [Tabella 5.2](#).

Tabella 5.2 – Membri della classe Hashtable

Metodo/Proprietà	Descrizione	Esempio
<code>Add(key, value)</code>	Aggiunge un elemento alla Hashtable	<code>myDict.Add("someKey", 5);</code>

<code>Clear()</code>	Svuota il contenuto della Hashtable	<code>myDict.Clear();</code>
<code>Contains(key)</code>	Ritorna <code>true</code> nel caso in cui la chiave specificata sia presente nel dizionario	<code>if (myDict.Contains("someKey")) ...</code>
<code>ContainsValue(value)</code>	Ricerca un determinato valore nel dizionario e restituisce <code>true</code> nel caso in cui questo sia presente	<code>if (myDict.ContainsValue(5))...</code>
<code>Item(key)</code>	Recupera un valore tramite la corrispondente chiave. La proprietà <code>Item</code> può eventualmente essere omessa	<code>object value = myDict.Item["somekey"];</code> oppure <code>object value = myDict["someKey"];</code>
<code>Keys</code>	Restituisce una collection di tutte le chiavi presenti	<code>foreach (var key in myDict.Keys)...</code>
<code>Remove(key)</code>	Elimina un elemento, data la corrispondente chiave	<code>myDict.Remove("SomeKey");</code>
<code>Values</code>	Restituisce una collection di tutti i valori presenti	<code>foreach (var val in myDict.Values)...</code>

Nonostante la loro versatilità, utilizzando le classi del framework è piuttosto raro imbattersi in metodi che accettino o restituiscano `Hashtable` o `ArrayList`. Molto più comune, invece, è trovare riferimenti ad alcune interfacce, come `IEnumerator`, `ICollection` o `IDictionary`, e ciò dipende dalla particolare architettura che è stata data all'infrastruttura delle collection, che sarà oggetto della prossima sezione.

[Le interfacce in System.Collections](#)

Come abbiamo avuto modo di apprendere dalla lettura del capitolo precedente, nel paradigma della programmazione a oggetti si usa rappresentare con

un'interfaccia la capacità, da parte di un determinato oggetto, di svolgere un particolare compito. In questo modo, è possibile scrivere codice senza legarsi a una particolare implementazione concreta, basandosi esclusivamente sui requisiti che l'oggetto deve avere affinché siamo in grado di utilizzarlo. Si tratta di concetti avanzati, che risulteranno più chiari proseguendo nella lettura del libro, ma che trovano una netta applicazione proprio all'interno del namespace `system.collections`. Esso, infatti, oltre a contenere le definizioni di diverse tipologie di collection, dichiara una serie di interfacce, ognuna delle quali è rappresentativa di un certo insieme di funzionalità di una collezione; interfacce e classi concrete costituiscono la gerarchia di tipi mostrata nella [Figura 5.1](#).

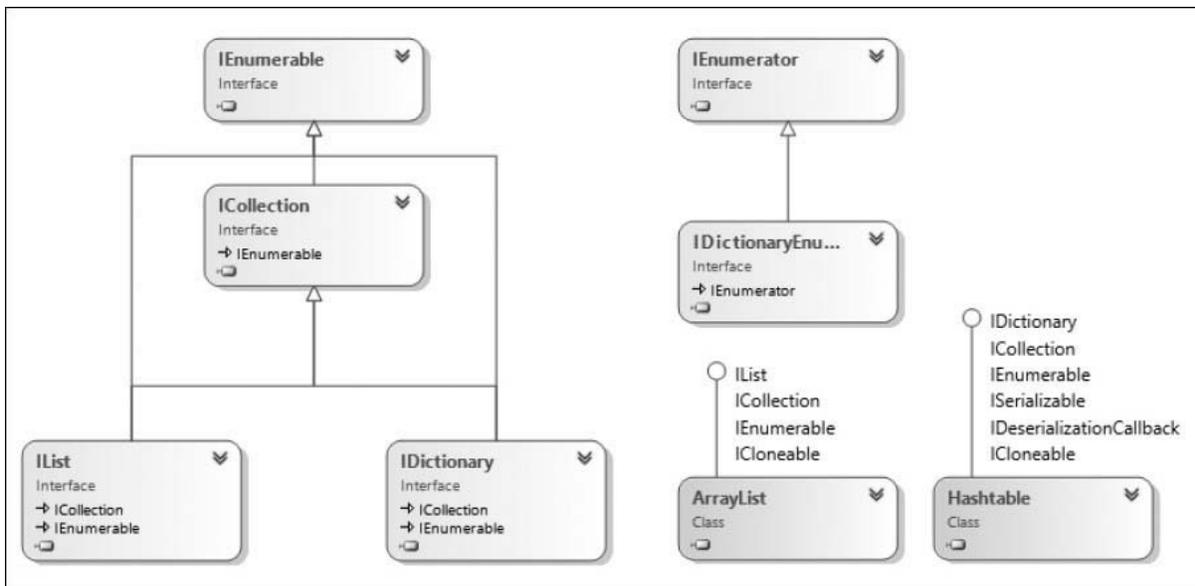


Figura 5.1 – Interfacce e classi in System.Collections.

Tra queste, quella più utilizzata è sicuramente `IEnumerable`, che rappresenta la possibilità di scorrere un oggetto tramite un enumeratore (che a sua volta implementa l'interfaccia `IEnumerator`). Questo, in pratica, si traduce nella possibilità di utilizzare il costrutto `foreach` per esplorare il contenuto della collezione. Pertanto, questa interfaccia è implementata un po' da tutte le tipologie di collection che abbiamo avuto modo di incontrare finora, dai semplici array fino ad arrivare all'`Hashtable`. A queste si affiancano `ICollection`, `IList` e `IDictionary`, che via via aggiungono funzionalità più avanzate; nella [Tabella 5.3](#) possiamo trovare un piccolo riepilogo di ognuna di

esse, in ordine di complessità, unitamente alle caratteristiche che le contraddistinguono.

Tabella 5.3 – Interfacce in System.Collections

Interfaccia	Descrizione
IEnumerable	Esponde il metodo GetEnumerator che ritorna un oggetto di tipo IEnumerator tramite il quale è possibile scorrere l'intera collection. Si traduce nella possibilità di navigarne il contenuto tramite l'istruzione foreach.
IEnumerator	Esponde la proprietà Current e i metodi Reset e MoveNext. Un IEnumerator è automaticamente creato e utilizzato dal costrutto foreach per esplorare il contenuto di una collection.
IDictionaryEnumerator	Eredita da IEnumerator, si tratta di un enumeratore specifico per i dizionari.
ICollection	Eredita da IEnumerable e aggiunge, tra le altre, la proprietà Count per conoscere la numerosità della collezione.
ICollection	Eredita da ICollection e contiene le funzionalità di aggiunta e rimozione dinamica di elementi, esponendo metodi come Add, Clear, Contains, Insert o Remove, oltre alla proprietà Item tramite la quale recuperare un elemento in base al suo indice.
IDictionary	Eredita da ICollection ed espone metodi e proprietà simili a quelli di ICollection, ma relativi ai dizionari e quindi basati sulla chiave piuttosto che sull'indice.

Ulteriori tipologie di collection

Il supporto da parte di .NET alla rappresentazione di collezioni non si ferma di certo alla presenza di ArrayList e Hashtable. Oltre a liste e dizionari, all'interno di System.Collections trovano posto ulteriori classi, pensate per soddisfare esigenze specifiche, che vale comunque la pena citare. Concetti come **pile** o **code** sono, per esempio, implementati rispettivamente dalle classi Stack e Queue. Si tratta di collezioni che non implementano l'interfaccia ICollection e che, quindi, non possono essere popolate con i metodi standard Add o Remove.

In particolare, Stack rappresenta un contenitore di oggetti che può essere riempito e svuotato secondo una logica **LIFO** (Last In First Out): l'aggiunta di elementi avviene tramite il metodo Push, mentre per la rimozione bisogna

utilizzare `Pop`, che ritorna l'ultimo elemento inserito in ordine di tempo, rimuovendolo al contempo dalla collezione.

`Queue`, invece, opera in modo differente, implementando una logica di tipo **FIFO** (First In First Out): la collezione viene riempita tramite il metodo `Enqueue`, mentre con `Dequeue` è possibile svuotarla, recuperando di volta in volta l'elemento più "anziano" al suo interno. Per chiarire meglio i concetti sin qui espressi, consideriamo il codice dell'[esempio 5.6](#).

Esempio 5.6

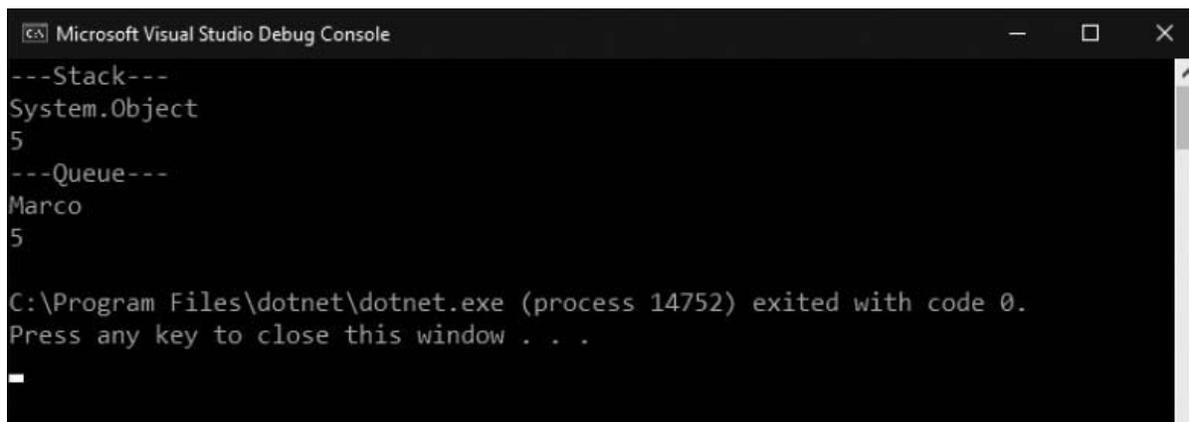
```
Console.WriteLine("---Stack---");
var myStack = new Stack();
myStack.Push("Marco");
myStack.Push(5);
myStack.Push(new object());

Console.WriteLine(myStack.Pop());
Console.WriteLine(myStack.Pop());

Console.WriteLine("---Queue---");
var myQueue = new Queue();
myQueue.Enqueue("Marco");
myQueue.Enqueue(5);
myQueue.Enqueue(new object());

Console.WriteLine(myQueue.Dequeue());
Console.WriteLine(myQueue.Dequeue());
```

In esso, una `Queue` e uno `Stack` vengono riempiti con i medesimi valori e successivamente svuotati, mostrando gli elementi recuperati sulla console. Eseguendo il codice, otteniamo l'output mostrato nella [Figura 5.2](#), in cui è possibile apprezzare il diverso ordine secondo cui viene restituito il contenuto.



```
Microsoft Visual Studio Debug Console
---Stack---
System.Object
5
---Queue---
Marco
5

C:\Program Files\dotnet\dotnet.exe (process 14752) exited with code 0.
Press any key to close this window . . .
```

Figura 5.2 – Esempio di utilizzo di Stack e Queue.

Ulteriori tipologie di collezioni, quali, per esempio, `StringCollection` o `StringDictionary`, presenti in `System.Collections.Specialized`, sono attualmente da considerarsi superate e non più utilizzate, grazie all'avvento dei **generics**, che saranno l'argomento della restante parte del capitolo.

I Generics e la tipizzazione forte

Le varie tipologie di collezioni presentate hanno ovviamente il requisito di poter essere utilizzabili con qualsiasi tipo di oggetto, tant'è che il metodo `Add` o la proprietà `Item` di `ArrayList`, per esempio, lavorano con il tipo `object`; questa versatilità, però, ha un costo. Immaginiamo di avere, per esempio, una lista di oggetti `DateTime` e di dover poi recuperare un elemento. Il codice da scrivere è simile a quello dell'[esempio 5.7](#).

Esempio 5.7

```
var dateList = new ArrayList();
dateList.Add(DateTime.Now);
dateList.Add(new DateTime(2000, 1, 10));
dateList.Add(DateTime.Today.AddYears(-1));

DateTime firstItem = (DateTime) dateList[0];
```

Come possiamo notare, avere a che fare con una lista di `object` implica la necessità di dover effettuare ogni volta il casting al tipo desiderato (`DateTime` in questo caso), visto che il dato non è direttamente disponibile come tale. Ciò, oltre che scomodo e ripetitivo, è anche potenzialmente molto pericoloso, perché, di fatto, non è possibile attuare nessun tipo di controllo sull'effettivo contenuto di una lista. Se guardiamo l'[esempio 5.8](#), infatti, notiamo come il suo contenuto possa essere quanto più eterogeneo.

Esempio 5.8

```
var list = new ArrayList();

list.Add(10);
list.Add("Some string");
list.Add(DateTime.Now);
list.Add(new ArrayList());
list.Add(new System.Globalization.CultureInfo("en-US"));
```

```
DateTime secondItem = (DateTime) list[0];
```

Il codice precedente viene ritenuto come perfettamente valido dal compilatore, ma se eseguito genera un errore (viene sollevata una `InvalidCastException`) in quanto nell'ultimo statement si sta provando a convertire un dato di tipo `int` in un `DateTime`. Il grosso problema è che purtroppo, finché il tutto non è in esecuzione, non abbiamo nessuno strumento per accorgercene; in gergo, si dice che una classe come `ArrayList` (e, in generale, tutte le `collection` e le interfacce di `System.Collections`) supporta solo la **tipizzazione debole**.

.NET utilizza un sistema di notifica degli errori a runtime tramite l'utilizzo di eccezioni: quando l'esecuzione di un metodo non termina correttamente, il chiamante riceve una segnalazione nella forma di un oggetto che deriva dalla classe `System.Exception` e il cui tipo identifica la particolare tipologia di errore che si è verificata. Affronteremo questo argomento maggiormente in dettaglio nel corso del [Capitolo 7](#).

Noi vogliamo invece codice con rigidi vincoli di tipo, o meglio **fortemente tipizzati**, così che, in una lista di `DateTime`, sia effettivamente consentito aggiungere solo dati omogenei, pena errori già in fase di compilazione, quindi facilmente identificabili e tracciabili. Proprio questo è il vantaggio introdotto dai generics.

[Le collezioni generiche](#)

Dalla versione 2.0 del .NET Framework, è stato introdotto il nuovo namespace `System.Collections.Generic` che contiene una serie di collezioni che, a differenza delle precedenti, non fanno più riferimento a `object`, ma consentono di volta in volta, quando utilizzate, di specificare con esattezza il tipo che dovranno contenere. Il modo migliore per rendere più chiaro questo concetto è sicuramente quello di esplorare nel dettaglio il funzionamento di quella che probabilmente è la `collection` generica attualmente più utilizzata: `List<T>`.

[La lista nel mondo dei generics: List<T>](#)

`List<T>` rappresenta l'alter-ego generico di `ArrayList` ed è pertanto adatta a creare delle strutture simili ad array, ma in grado di supportare l'aggiunta e la rimozione di elementi, adattando dinamicamente la loro dimensione. L'[esempio 5.9](#) mostra come debba essere utilizzata per realizzare una lista di stringhe.

Esempio 5.9

```
var strings = new List<string>(); // Inizializzazione di List<T>
strings.Add("Matteo Tumiati"); // Possiamo aggiungere solo string
strings.Add("C# 8");
strings.Insert(0, "Primo elemento");

var index = strings.IndexOf(
    "Matteo Tumiati"); // Ritorna 1

var mySubstring =
    strings[0].Substring(5); // Già string: cast non necessario
```

Il parametro generico `T` presente nel nome di questa collection, sta a indicare il tipo degli elementi contenuti al suo interno. Può essere pensato come una sorta di segnaposto che deve poi essere obbligatoriamente specificato all'atto della dichiarazione della variabile. Nel codice in alto, per esempio, dovendo realizzare una lista di stringhe, è stata dichiarata e istanziata una variabile di tipo `List<string>`.

Il primo vantaggio consiste nel fatto che, recuperando un elemento dalla collection, mentre `ArrayList` restituiva un `object` e quindi era necessario scrivere

```
string myString = (string) arrayList[0];
```

nel caso di `List<T>` ciò non è più necessario, in quanto l'oggetto restituito è già di tipo `string`, con il conseguente vantaggio di scrivere meno codice e di evitare un'istruzione come quella di cast esplicito, che, comunque, potrebbe fallire e generare un errore. Ciò è possibile perché l'infrastruttura dei generics assicura che il contenuto della lista sia composto esclusivamente da oggetti di tipo `string` e che una riga di codice simile a:

```
strings.Add(DateTime.Now);
```

provochi un errore **già in fase di compilazione**. Questa è in assoluto la peculiarità più importante dei tipi generici, grazie alla quale siamo in grado di scrivere codice **fortemente tipizzato**, in cui sia il compilatore, ma anche l'IntelliSense di Visual Studio, come mostra la [Figura 5.3](#), sono in grado di intercettare eventuali errori già nella fase di scrittura dell'applicazione.

```
var strings = new List<string>(); // Inizializzazione di List<T>
strings.Add()~
```

```
void List<string>.Add(string item)
Adds an object to the end of the List<T>.
item: The object to be added to the end of the List<T>. The value can be null for reference types.
```

Figura 5.3 – Supporto dell'IntelliSense ai generics.

*Solitamente il primo dubbio che viene in mente una volta posti davanti ai vantaggi dei tipi generici, riguarda eventuali penalità, a livello di prestazioni, che essi possono avere rispetto ai corrispettivi non generici. Nella realtà dei fatti emerge invece esattamente il contrario: il parametro generico, infatti, funge esclusivamente da segnaposto e viene rimpiazzato con il tipo reale **in fase di compilazione**, senza pertanto comportare alcuna perdita nelle performance. Quando poi il parametro generico viene valorizzato con un value type, si ha addirittura un notevole effetto benefico nelle prestazioni: senza spingersi troppo nei dettagli, basti sapere che List<int>, per esempio, è estremamente più veloce di ArrayList per gestire liste di interi, in quanto consente di evitare il ricorso a operazioni dispendiose, come **boxing** e **unboxing**, per trasformare un tipo valore in un tipo riferimento e viceversa.*

L'infrastruttura delle collection generiche è basata su un insieme di interfacce che rappresentano le funzionalità elementari che ogni singola collezione implementa. Esse saranno l'argomento del prossimo paragrafo.

[Le interfacce nelle collezioni generiche](#)

Da un punto di vista architetturale, la struttura delle collection generiche non si discosta più di tanto da quanto già visto in precedenza e consta innanzitutto di un insieme di interfacce, per lo più omonime delle controparti non generiche.

Il diagramma presente nella [Figura 5.4](#) mostra i vincoli di ereditarietà che sussistono tra loro:

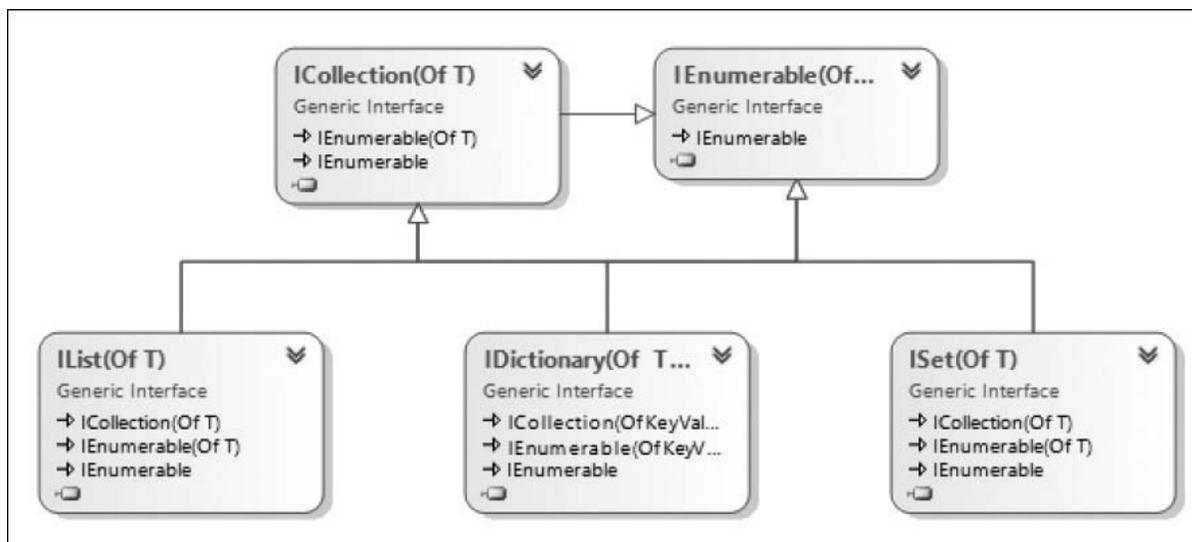


Figura 5.4 – Interfacce delle collection generiche.

Come si può notare, i capisaldi sono costituiti da `IEnumerable<T>` e `ICollection<T>`, che espongono alcune funzionalità di base, e in particolare:

- ❑ `IEnumerable<T>` - e `IEnumerator<T>`- consentono di scorrere il contenuto di una collection e abilitano il supporto all'istruzione `foreach`; possiamo notare che, al fine di mantenere la compatibilità anche con i tipi non generici, `IEnumerable<T>` eredita da `IEnumerable`;
- ❑ `ICollection<T>` espone le funzionalità di base di una collezione e, a differenza del caso non generico, oltre a `Count` annovera anche metodi quali `Add`, `Remove` o `Contains`.

Queste due interfacce vengono utilizzate come base di partenza per le funzionalità relative alle tre tipologie principali di collection previste da .NET:

- ❑ `IList<T>` rappresenta le logiche tipiche delle liste, introducendo il concetto di indice in una collezione: fanno la loro comparsa pertanto i membri basati sul concetto di posizione all'interno della lista, quali la proprietà `Item` e metodi `IndexOf`, `Insert` e `RemoveAt`;

- ❑ `IDictionary<TKey, TValue>` espone le logiche tipiche dei dizionari, similmente a `IDictionary`, consentendo però di specificare i tipi da utilizzare per la chiave e per i valori;
- ❑ `ISet<T>` è utilizzata per rappresentare insiemi di oggetti univoci.

Inizializzazione di una collezione

Finora abbiamo sempre costruito istanze vuote di `List<T>`, per poi aggiungere gli elementi nelle istruzioni successive. In realtà esistono delle modalità più concise per far creare e popolare le nostre collezioni. La più semplice è quella dell'[esempio 5.10](#), e si basa su un particolare overload del costruttore, grazie al quale possiamo inizializzare la collection a partire da un oggetto qualsiasi che implementi `IEnumerable<T>`.

Esempio 5.10

```
var source = new string[] { "String1", "String2", "String3" };  
  
// qui abbiamo usato un array, ma un qualsiasi  
// IEnumerable<string> va bene  
var strings = new List<string>(source);
```

Nel codice precedente abbiamo creato un array di stringhe, da usare come parametro per il costruttore del nostro oggetto `strings`. Una sintassi ancora più concisa è quella dell'[esempio 5.11](#):

Esempio 5.11

```
// Inizializzazione inline  
var strings = new List<string>  
{  
    "String1",  
    "String2",  
    "String3"  
};  
  
// Inizializzazione inline con indici espliciti  
var strings1 = new List<string>  
{  
    [0] = "String1",  
    [4] = "String5",  
    [2] = "String3"  
};
```

In pratica, possiamo specificare gli elementi in modalità “in linea”, ossia subito dopo aver invocato il costruttore. Come possiamo notare, da C# 6 c’è anche la possibilità di specificare esplicitamente gli indici a cui inserire i vari elementi.

Finora ci siamo occupati esclusivamente di `List<T>`. Il framework, tuttavia, contiene diverse altre tipologie di collezioni, che vale la pena conoscere perché possono essere utilizzate per scopi differenti: ci stiamo riferendo in particolare alle classi `Dictionary<TKey, TValue>` e `HashSet<T>`.

[Un dizionario fortemente tipizzato: Dictionary<TKey, TValue>](#)

Le funzionalità tipiche di un dizionario sono implementate, nelle collection generiche, dalla classe `Dictionary<TKey, TValue>`. Per molti versi ha un comportamento simile a quello visto in precedenza per `Hashtable`, ma espone due parametri generici per specificare i tipi ammissibili per chiavi e valori, come mostrato nell’[esempio 5.12](#).

Esempio 5.12

```
var holidays = new Dictionary<string, DateTime>();
holidays.Add("Natale", new DateTime(2015, 12, 25));
holidays.Add("Capodanno", new DateTime(2015, 1, 1));
holidays.Add("Compleanno", new DateTime(2015, 7, 10));

DateTime vigiliaCompleanno = holidays["Compleanno"].AddDays(-1);
```

Anche in questo caso, com’è lecito attendersi, il codice è fortemente tipizzato e un tentativo di usare un tipo diverso da `string` per la chiave o `DateTime` per il valore causa un errore in compilazione; ovviamente lo stesso vale anche in fase di recupero del dato, come mostra in maniera eloquente la [Figura 5.5](#):

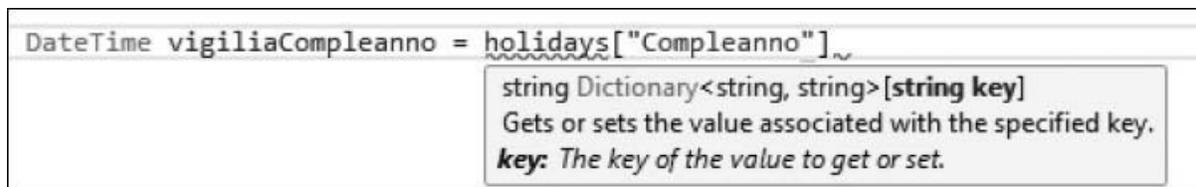


Figura 5.5 – Type safety del dictionary.

`Dictionary<TKey, TValue>` utilizza internamente il medesimo sistema di memorizzazione basato su un hash utilizzato da `Hashtable`, con il risultato di rendere estremamente veloci le ricerche basate sulla chiave. A differenza di quest'ultima, però, il tentativo di recupero di un elemento tramite una chiave inesistente, non restituisce `null`, ma risulta in un errore a runtime (viene sollevata un'eccezione di tipo `KeyNotFoundException`). Pertanto, prima di utilizzare la proprietà `Item` è consigliabile invocare il metodo `ContainsKey` per verificarne preventivamente la presenza:

```
DateTime date;
if (holidays.ContainsKey("chiave"))
    date = holidays["chiave"];
else
    Console.WriteLine("La data cercata non esiste");
```

In alternativa, tramite `TryGetValue` è possibile scrivere del codice funzionalmente identico in maniera un po' più concisa:

```
DateTime date;
if (!holidays.TryGetValue("chiave", out date))
    Console.WriteLine("La data cercata non esiste");
```

Questo metodo, infatti, oltre al valore della chiave, accetta un secondo argomento passato per riferimento e restituisce un valore `bool` che indica il successo dell'operazione.

Anche `Dictionary<TKey, TValue>` ha una modalità di inizializzazione in linea, analoga a quella che abbiamo già visto in precedenza per `List<T>`, come possiamo vedere nell'esempio seguente:

Esempio 5.13

```
var holidays = new Dictionary<string, DateTime>()
{
    ["Natale"] = new DateTime(2015, 12, 25),
    ["Capodanno"] = new DateTime(2016, 1, 1),
    ["Compleanno"] = new DateTime(2015, 7, 10)
};
```

[Una collection con elementi univoci: HashSet<T>](#)

Un tipo di collezione che non ha un corrispondente non generico è l'HashSet. Si tratta di una collezione simile alla lista, poiché consente l'inserimento di valori, ma con uno scopo completamente diverso: rappresentare un insieme non ordinato di elementi univoci.

Consideriamo l'[esempio 5.14](#):

Esempio 5.14

```
var holidays = new HashSet<DateTime>();  
  
holidays.Add(new DateTime(2015, 12, 25));  
holidays.Add(new DateTime(2015, 1, 1));  
holidays.Add(new DateTime(2015, 5, 1));  
holidays.UnionWith(ferieEstive);  
  
var someDate = new DateTime(2015, 11, 1);  
  
bool willWork = holidays.Contains(someDate);
```

Il codice in alto mostra quello che è il tipico utilizzo di un HashSet, ossia non quello di fungere da contenitore di elementi con lo scopo di recuperarli, bensì di utilizzarlo per verificare l'appartenenza di un dato all'insieme. Questa classe, infatti, utilizza un sistema basato sui codici hash per rendere operazioni di questo tipo (come l'esecuzione del metodo Contains) estremamente veloci ed efficienti. Un esempio tipico può essere quello mostrato in precedenza, in cui tramite un HashSet di festività, si determina se una certa data è lavorativa o meno.

Proprio per questo scopo, HashSet non espone membri tramite i quali recuperare il contenuto (se si esclude il fatto che comunque è ammesso l'uso del costrutto foreach per scorrerlo), bensì presenta una serie di metodi tipici della teoria degli insiemi, sintetizzati nella [Tabella 5.4](#).

Tabella 5.4 – Metodi di HashSet<T>

Metodo/Proprietà	Descrizione
Add(value)	Aggiunge un elemento all'HashSet se non è già presente e ritorna un boolean per indicare se l'aggiunta è stata effettuata o meno
Clear()	Cancella tutti gli elementi
Contains(value)	Effettua una ricerca tramite codice hash e ritorna true se l'elemento appartiene alla collezione

<code>ExceptWith(collection)</code>	Elimina dal set tutti gli elementi presenti nella collezione in input, realizzando un'operazione di differenza
<code>IntersectWith(collection)</code>	Elimina dal set tutti gli elementi non presenti nella collezione in input, realizzando un'operazione di intersezione
<code>IsSubsetOf(collection)</code> <code>IsProperSubsetOf(collection)</code>	Verifica se l'HashSet è un sottoinsieme (eventualmente in senso stretto) della collezione in input
<code>IsSupersetOf(collection)</code> <code>IsProperSupersetOf(collection)</code>	Verifica se la collezione in input è un sottoinsieme (eventualmente in senso stretto) dell'HashSet
<code>Overlaps(collection)</code>	Ritorna true se il set e la collection in input hanno almeno un elemento in comune
<code>Remove(value)</code>	Elimina, se presente, un elemento dal set
<code>UnionWith(collection)</code>	Realizza un'operazione di unione con la collection in input

Altre tipologie di collezioni generiche

Oltre a quelle presentate fino a questo punto, .NET e in particolare il namespace `System.Collections.Generic`, contengono una serie di ulteriori tipologie di collection, di uso più specifico, che vale però comunque la pena citare:

- ❑ `Stack<T>` e `Queue<T>` sono le versioni generiche delle analoghe collection già viste in precedenza in questo capitolo;
- ❑ `SortedList<TKey, TValue>` e `SortedDictionary<TKey, TValue>` sono due differenti implementazioni di un dizionario i cui elementi sono ordinati in base al valore della chiave; differiscono nella loro implementazione interna: la prima è in generale meno esosa di memoria, mentre la seconda è più veloce nelle operazioni di inserimento e rimozione del dato;
- ❑ `LinkedList<T>` è una lista di elementi ognuno dei quali ha un puntatore all'elemento successivo e uno al precedente. L'uso di questo tipo di collezione è estremamente vantaggioso quando si presenta spesso la necessità di inserire e rimuovere elementi, in quanto queste operazioni risultano estremamente veloci;
- ❑ `BindingList<T>` e `ObservableCollection<T>` non appartengono al namespace `System.Collections.Generic`, ma rispettivamente a `System.ComponentModel` e `System.Collections.ObjectModel`.

Funzionalmente si comportano come `List<T>`, e implementano alcune interfacce aggiuntive indispensabili per l'utilizzo come sorgente dati in un'applicazione Windows. Si tratta di concetti che esulano dallo scopo di questo capitolo e che verranno trattati nel prosieguo del libro.

Le potenzialità dei tipi generici non si esauriscono con le collection, e sono molteplici gli esempi all'interno di .NET di tipi che ne fanno uso. Ovviamente esse possono essere sfruttate anche per la creazione di classi personalizzate. Nelle prossime pagine vedremo come.

[Creazione di tipi generici](#)

Costruire una classe che sfrutti i generics è estremamente semplice e, in pratica, richiede esclusivamente di specificare, all'atto della dichiarazione, il numero e il nome dei tipi generici da utilizzare. Supponiamo, per esempio, di avere la necessità di realizzare una sequenza di elementi, in cui ogni membro sia in grado di memorizzare un valore e un riferimento all'elemento successivo, come mostrato nella [Figura 5.6](#).

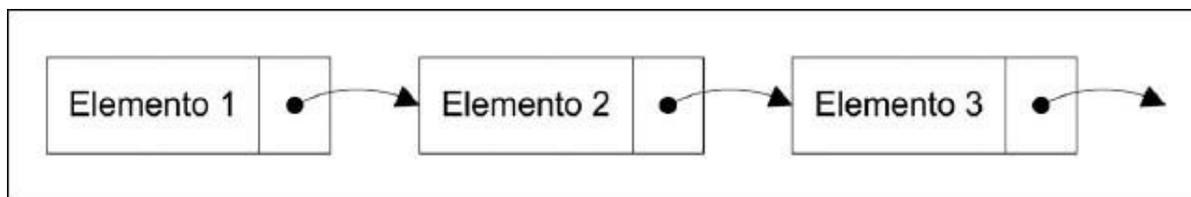


Figura 5.6 – Sequenza di elementi.

*Si tratta di quella che, in letteratura informatica, è comunemente chiamata “**lista**”; questo termine ha però un’accezione differente in .NET e pertanto la indicheremo genericamente come “**sequenza**”. Anche se abbiamo già introdotto in precedenza una struttura dati simile a questa, ossia `LinkedList<T>`, da un punto vista didattico ha comunque senso provare a costruire un esempio personalizzato, per comprendere come sfruttare i generics anche in questo caso.*

Ogni elemento della sequenza può essere implementato tramite una classe e deve esporre una proprietà `value` contenente il valore memorizzato. Senza i generics, saremmo costretti a realizzare un’implementazione specifica per ogni

tipo da supportare, o a utilizzare una proprietà di tipo `object`, perdendo così i vantaggi della tipizzazione forte. Grazie ai generics, invece, il codice da scrivere risulta estremamente semplice e comprensibile, come possiamo notare nell'[esempio 5.15](#).

Esempio 5.15

```
public class ListNode<T>
{
    public T Value { get; set; }
    public ListNode<T> NextNode { get; set; }
}
```

La dichiarazione della classe presenta la stessa sintassi vista ormai diverse volte durante questo capitolo; una volta indicato, tramite il segnaposto `T`, il tipo indeterminato, esso diviene a tutti gli effetti utilizzabile nel codice come ogni altro, come possiamo notare dalla dichiarazione delle proprietà `Value` e `NextNode`.

I parametri generici possono essere dichiarati e utilizzati anche su un singolo metodo, senza dover necessariamente costruire una classe generica che lo contenga, come mostrato nel codice seguente:

Esempio 5.16

```
public class Utils
{
    public static void Swap<T>(ref T first, ref T second)
    {
        T temp = first;
        first = second;
        second = temp;
    }
}
```

L'[esempio 5.16](#) mostra un metodo in grado di scambiare due oggetti passati come argomenti. Come possiamo notare, il metodo è contenuto in una classe `Utils` che non è dichiarata come generica, mentre il parametro generico è definito direttamente sul metodo `Swap`. Un aspetto interessante è che il compilatore, in alcune situazioni, è in grado di determinare autonomamente il valore da assegnare al tipo `T` senza che siamo costretti a specificarlo esplicitamente:

```
int a = 5;
int b = 8;

Utils.Swap(ref a, ref b);
```

Questa caratteristica è denominata **type inference** e consente di scrivere codice in maniera estremamente concisa e leggibile.

[Impostare dei vincoli sul tipo generico](#)

Spesso definire un parametro generico `T` non è sufficiente, ma è necessario specificarne anche alcuni requisiti, come il fatto che sia una classe piuttosto che una struttura, o che implementi una determinata interfaccia. Supponiamo, per esempio, di voler realizzare un metodo per calcolare il massimo valore all'interno di una lista, simile a:

```
public static T Max<T>(IEnumerable<T> list)
{
    ...
}
```

In questo caso non è possibile accettare un qualsiasi tipo per il parametro `T`: nel codice del metodo `Max`, infatti, saremo sicuramente costretti a effettuare operazioni di confronto, che non è detto siano applicabili a un tipo qualsiasi, come `System.IO.File` o `System.Exception`. Pertanto, è necessario specificare come vincolo che `T` implementi l'interfaccia `IComparable`, in modo che siamo in grado di confrontare due valori della collection in input e determinare qual è il maggiore. La sintassi è quella mostrata, unitamente all'implementazione, nell'[esempio 5.17](#).

Esempio 5.17

```
public static T Max<T>(IEnumerable<T> list) where T : IComparable
{
    bool isFirst = true;

    T res = default(T);
    foreach (T item in list)
    {
        if (isFirst)
        {
            res = item;
            isFirst = false;
        }
    }
}
```

```

    if (res.CompareTo(item) < 0)
        res = item;
    }
    return res;
}

```

Il risultato è che sarà possibile utilizzare `Max` con una lista di interi o con array di stringhe, perché entrambi implementano `IComparable`, ma solleverà un errore in compilazione se invocato con una lista di `object`.

La [Tabella 5.5](#) mostra le possibili tipologie di vincoli che possono essere imposti sui parametri di un tipo generico.

Tabella 5.5 – Vincoli sui tipi generici

<code>where T : class</code>	Il tipo T deve essere una classe (tipo riferimento)
<code>where T : struct</code>	Il tipo T deve essere una struct (tipo valore)
<code>where T : new()</code>	Il tipo T deve avere un costruttore senza parametri
<code>where T : Interface</code>	Il tipo T deve implementare l'interfaccia specificata
<code>where T : BaseClass</code>	Il tipo T deve ereditare dalla classe base specificata

Un altro aspetto cui dobbiamo prestare attenzione è il modo in cui abbiamo inizializzato la variabile `res`. A priori, infatti, non ci è dato sapere se `T` sia un tipo di riferimento o un tipo di valore e non vogliamo porre vincoli in questo senso: fintanto che questo tipo implementa `IComparable` il nostro algoritmo è perfettamente funzionante!

In casi simili, se assegnassimo `null` alla variabile `res`, otterremmo un errore in compilazione, in quanto non si tratta di un'operazione lecita per qualsiasi tipo, ma solo per le classi:

```

T res = null; // genera un errore in fase di compilazione!

```

Ecco la ragione per cui, nell'[esempio 5.14](#), ci siamo invece avvalsi dell'istruzione `default(T)`, che restituisce `null` nel caso di un tipo di riferimento e il valore di default per un tipo di valore (per esempio `0`, nel caso di un `int`). A partire da C# 7.1 è stata introdotta la **default literal expression**, pertanto è possibile semplificare l'istruzione `default(T)` in `default`, poiché il tipo verrà recuperato automaticamente dal compilatore per type inference.

Un particolare tipo generico: Nullable<T>

Come abbiamo appena avuto modo di evidenziare, i tipi di valore come `int`, `DateTime` o `bool`, tanto per citarne alcuni, presentano tutti la caratteristica di non poter assumere un valore `null`. Spesso, invece, soprattutto nell'ambito di applicazioni che si interfacciano con basi di dati, abbiamo la necessità di rappresentare, anche con dei tipi valore, il dato "indeterminato".

A questo scopo, .NET mette a disposizione degli oggetti chiamati **Nullable Value Types**, realizzati tramite il tipo generico `Nullable<T>`, che consentono di aggiungere anche tale stato a un tipo di valore. Un aspetto interessante è che essi ridefiniscono internamente gli operatori di uguaglianza e casting, in modo che possano essere utilizzati al posto delle controparti non nullabili in maniera quasi del tutto trasparente. L'[esempio 5.18](#) mostra alcune tipiche modalità di utilizzo e contribuisce a chiarire questo concetto.

Esempio 5.18

```
int myInt = default(int);
Nullable<int> nullableInt = default(Nullable<int>);

Console.WriteLine(myInt);      // Stampa 0
Console.WriteLine(nullableInt); // Stampa una riga vuota

myInt = 5;
nullableInt = 7;      // Assegnazione di un valore intero

var res = myInt + nullableInt;      // res è di tipo Nullable<int>

if (nullableInt.HasValue)          // Verifica presenza di un valore
{
    var value = nullableInt.Value;  // Contiene il valore int
}
```

In particolare, tramite le proprietà `HasValue` e `Value` è possibile, rispettivamente, verificare la presenza di un valore all'interno del tipo nullable ed eventualmente recuperarlo. Per dichiarare un `Nullable Value Type` possiamo utilizzare anche la notazione contratta mostrata in basso.

```
int? nullableInt;
```

Assegnazioni tra tipi generici: covarianza e controvarianza

Un aspetto che sulle prime lascia piuttosto perplessi utilizzando i generics è la complessità nel gestire le assegnazioni tra tipi diversi quando tra i relativi parametri sussistono comunque relazioni di ereditarietà. Per esempio, non è lecito (e provoca un errore di compilazione) il codice dell'[esempio 5.19](#).

Esempio 5.19

```
public static void SomeMethod(List<object> list)
{
    foreach (var item in list)
    {
        Console.WriteLine(item);
    }
}

static void Main(string[] args)
{
    var strings = new List<string>();
    SomeMethod(strings);
}
```

In esso, infatti, si prova a invocare un metodo che accetta una `List<object>`, passando una `List<string>`, al fine di stamparne il contenuto sulla console. In realtà, sebbene sulle prime possa sembrare strano, il compilatore C# ha tutte le ragioni per rifiutare questo codice: a priori, infatti, non c'è alcuna garanzia che `SomeMethod` non provi a modificare il contenuto della lista, aggiungendo magari elementi leciti per una `List<object>` ma non per una `List<string>`.

Questo limite può essere superato se si riscrive `SomeMethod` utilizzando, in luogo di `List<object>`, l'interfaccia `IEnumerable<object>`: quest'ultima, infatti, presenta la caratteristica di esporre il tipo `T` **solo in uscita**, come risultato cioè di funzioni o proprietà in sola lettura, e **mai in ingresso**, superando di fatto il problema che affliggeva invece il codice precedente:

```
public static void SomeMethod(IEnumerable<object> list)
{
    foreach (var item in list)
    {
        Console.WriteLine(item);
    }
}
```

Questa versione di `SomeMethod` consente al codice dell'[esempio 5.19](#) di compilare; si dice che, in `IEnumerable<T>`, il `T` è **covariante**, ossia che può essergli assegnato un `IEnumerable<TDerivato>` con un tipo generico più specifico di quello originale.

D'altro canto, esistono casi in cui il parametro generico `T` viene utilizzato all'interno di un'interfaccia come un argomento per i metodi da essa esposti. Una `List<T>`, per esempio, può essere ordinata tramite il metodo `Sort` e un opportuno oggetto `IComparer<T>`.

L'interfaccia `IComparer<T>` serve a rappresentare una particolare strategia per confrontare due istanze del tipo `T`; essa espone il metodo `Compare` che accetta le due istanze su cui effettuare il confronto e restituisce un intero, negativo nel caso in cui la prima sia minore della seconda, positivo in caso contrario. Utilizzare un `IComparer<T>` nel metodo `Sort` di una lista, consente di cambiare criteri di ordinamento in maniera estremamente versatile, fornendo come argomento, di volta in volta, dei `comparer` differenti, secondo la necessità.

Supponiamo, allora, di avere una `List<Car>`, dove `Car` eredita dalla classe base `Vehicle`, e di volerla ordinare in base alla velocità massima tramite uno `SpeedComparer`; i tre oggetti sono mostrati nell'[esempio 5.20](#).

Esempio 5.20

```
public class Vehicle
{
    public int Speed { get; set; }
}

public class Car : Vehicle
{
}

public class SpeedComparer : IComparer<Vehicle>
{
    public int Compare(Vehicle x, Vehicle y)
    {
        return x.Speed - y.Speed;
    }
}
```

Come possiamo notare, essendo la proprietà `Speed` definita in `Vehicle`, `SpeedComparer`, implementa l'interfaccia `IComparer<Vehicle>`; questo può

essere comunque utilizzato per ordinare una `List<Car>`, nonostante il metodo `Sort` di quest'ultima accetti un oggetto di tipo `IComparer<Car>`.

```
var cars = new List<Car>();  
// ... si popola la lista  
  
cars.Sort(new SpeedComparer());
```

Ciò è possibile perché `IComparer<T>` utilizza il tipo `T` esclusivamente **in ingresso**, ossia come argomento dei suoi metodi, ma mai come risultato di una funzione. Pertanto, si dice che in `IComparer<T>`, `T` è **controvariante**, cioè che ad esso può essere assegnato un oggetto che implementi `IComparer<TBase>`, con `T` che eredita da `TBase`.

Basandosi su quanto detto in questo paragrafo, si può facilmente dedurre che interfacce come `IList<T>` o `ICollection<T>` non possono essere né covarianti, né controvarianti. La prima, infatti, espone metodi che coinvolgono `T` sia come argomento sia come risultato, mentre la seconda ha metodi che accettano `T` solo in ingresso, ma non può essere controvariante in quanto eredita da `IEnumerable<T>`, che è a sua volta covariante, dato che espone il tipo `T` in uscita.

Creazione di interfacce covarianti e controvarianti

Ovviamente covarianza e controvarianza non sono concetti che rimangono circoscritti alle sole interfacce del framework, ed è possibile realizzarne di nuove con supporto a queste particolari modalità di assegnazione. Ciò è possibile associando le parole chiave `in` e `out` alla definizione dell'interfaccia generica, come mostrato dall'[esempio 5.21](#).

Esempio 5.21

```
public interface ICovariant<out T>  
{  
    T SomeCovariantMethod();  
}  
  
public interface IContravariant<in T>  
{  
    void SomeContravariantMethod(T arg);  
}
```

Anteponendo la parola chiave `out` al tipo generico `T`, ci impegniamo a definire esclusivamente metodi che utilizzino il tipo `T` come risultato e mai come argomento. Viceversa, dichiarando l'interfaccia come `<in T>`, siamo obbligati a usare `T` sempre e solo come argomento, pena un errore in compilazione.

Index e Range

Nonostante non abbiamo ancora avuto modo di vedere come si possono filtrare i dati aggiunti in array e liste per recuperare dei valori ben precisi, poiché sarà oggetto dei capitoli successivi quando introdurremo LINQ, è bene sapere che C# 8 (e in particolare .NET Standard 2.1) introduce il concetto dello slicing. Questa tecnica permette di creare una sorta di “finestra” al di sopra dell'array che si vuole filtrare a livello di memoria, così che i dati non abbiano bisogno di essere allocati nuovamente per essere letti (ad esempio creando un array di appoggio), ma possono essere recuperati direttamente dalla “finestra” stessa.

Prendiamo in esempio la seguente collezione di oggetti:

```
var peopleArray = new string[]
{
    "Daniele Bochicchio",
    "Stefano Mostarda",
    "Matteo Tumiatì",
    "Cristian Civera",
    "Marco Leoncini"
};
```

Per recuperare il primo e l'ultimo valore, abbiamo già visto che è sufficiente identificare l'array secondo l'indice degli elementi interessati:

```
var first = peopleArray[0]; // Daniele Bochicchio
var last = peopleArray[peopleArray.Length - 1]; // Marco Leoncini
```

Qualora volessimo recuperare un singolo elemento situato in una posizione specifica rispetto all'inizio, oppure alla fine, immaginiamo il terzultimo, le cose iniziano a farsi complicate con il metodo che abbiamo appena visto e, pertanto, conviene fare uso dell'operatore `hat (^)`.

Esempio 5.22

```
var last = peopleArray[^1];    // ultimo
var twoFromLast = peopleArray[^3];    //terzultimo
```

Nel caso mostrato dall'[esempio 5.22](#) la finestra creata ha dimensione uno poiché l'oggetto recuperato tramite l'operatore `hat` è unico. Nulla viene modificato per quanto riguarda l'inizio dell'array, sempre identificato a partire dall'indice zero. Questa rappresentazione, in particolare, è la versione abbreviata di quanto viene offerto dalla struttura `Index`, che permette la riscrittura come segue:

```
Index lastIndex = ^1;
Index twoFromLastIndex = ^3;
var last = peopleArray[lastIndex];
var twoFromLast = peopleArray[twoFromLastIndex];
```

Qualora dovessimo invece non solo recuperare un elemento in una posizione specifica, ma un range di elementi contenuti tra due posizioni note, dobbiamo fare uso della struttura `Range` (e l'operatore `..`) che identifica una finestra più grande per scorrere l'array:

Esempio 5.23

```
var peopleFiltered = peopleArray[1..3];

foreach (var person in peopleFiltered)
    Console.WriteLine(person);
```

Come visto per gli indici, anche i `Range` sono delle strutture che possono essere referenziate all'interno del codice e pertanto è possibile riscrivere l'[esempio 5.23](#) come segue:

```
Range range = 1..3;
var peopleFiltered = peopleArray[range];

foreach (var person in peopleFiltered)
    Console.WriteLine(person);
```

È bene sottolineare che i nomi delle persone stampati dal ciclo dell'[esempio 5.23](#) sono “*Stefano Mostarda*” e “*Matteo Tumiati*”: non c’è un errore e non sono stati persi elementi, ma semplicemente l’indice di partenza, definito prima dell’operatore `..`, è inclusivo, mentre l’operatore di fine, definito dopo l’operatore `..`, è esclusivo e pertanto estromette il valore “*Cristian Civera*” dall’output. Questo è sempre vero, a patto che venga specificato l’indice finale: esiste infatti una versione “abbreviata”, che permette di stampare tutti gli elementi a partire da un certo indice fino alla fine dell’array.

Esempio 5.24

```
// stamperà tutti (compreso l'ultimo) ad esclusione di Daniele Bochicchio
// poiché è il primo e l'indice è inclusivo
var peopleFiltered = peopleArray[1..];

// stamperà tutti perché entrambi gli indici sono inclusivi
var peopleFiltered = peopleArray[..];
```

Se invece si vuole fare in modo (ed è fattibile per via del fatto che si conosce l’indice) di includere anche l’elemento finale, è possibile combinare i `Range` con gli `Index`:

Esempio 5.25

```
foreach (var person in peopleArray[1..^1])
    Console.WriteLine(person);
```

Come si può notare dall'[esempio 5.25](#), infatti, verranno stampate tutte le persone che nell’array si trovano a partire dalla seconda posizione (indice uno dell’array) fino al penultimo, rendendolo, di fatto, inclusivo.

Qualora si volesse realizzare una sorta di finestra mobile, per leggere, per esempio, degli elementi a coppie all’interno dell’array, si può fare riferimento, a partire da [C# 7.2](#), alla struttura `Span<T>` che permette nuovamente, di creare una finestra su un array di memoria contigua allocata nello stack, ma di lavorare anche con memoria unmanaged. L’oggetto `Span<T>` è in realtà un wrapper perché contiene al suo interno un riferimento alla prima porzione di memoria da leggere e uno `slice` che identifica la grandezza della porzione da

leggere (una lunghezza di due nel caso in cui gli elementi vengano letti a coppie).

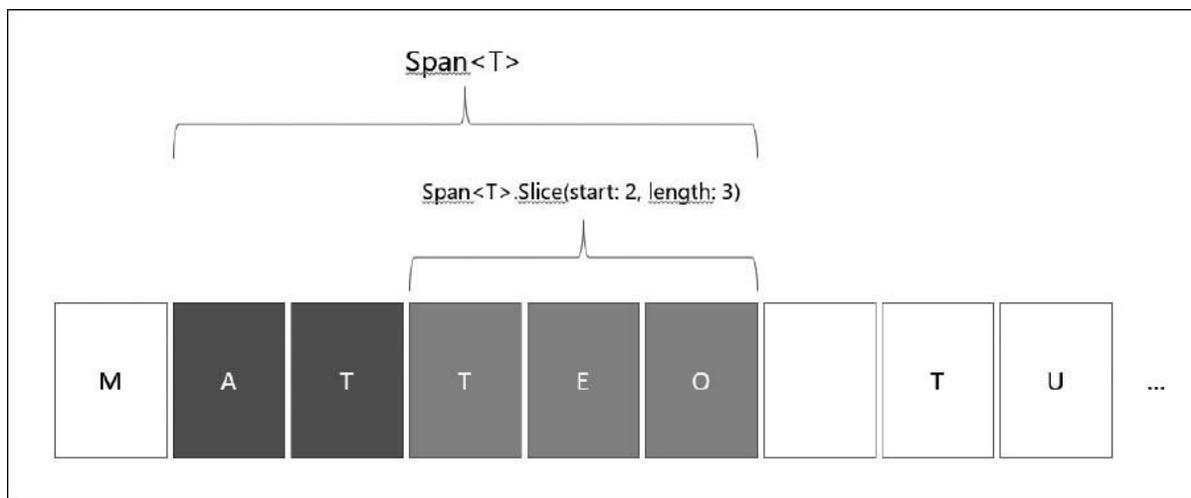


Figura 5.7 – Struttura di uno Span<T> in memoria.

Come si può notare nell'[esempio 5.26](#), le coppie vengono stampate in sequenza a due a due, fino al termine dell'array. Lo slice viene quindi ricalcolato a partire dallo span, recuperato dall'array contenente le persone, mentre la finestra rimane con una dimensione invariata di due elementi (l'indice finale è esclusivo) ma potrebbe anch'essa essere variata secondo le logiche di business richieste.

Esempio 5.26

```
Span<string> span = new Span<string>(peopleArray);  
for (var i = 0; i < peopleArray.Length; i = i + 2)  
{  
    Span<string> slice = span.Slice(start: i, length: 2);  
    Console.WriteLine($"Coppia: {slice[0]}, {slice[1]}");  
}
```

Le stringhe, invece, rappresentano un tipo particolare in quanto sono per loro natura immutabili all'interno del framework: il tipo `ReadOnlySpan<T>` viene introdotto proprio per ovviare a questa tipologia di oggetti e propone un accesso in sola lettura tramite la finestra. Supponendo di avere un metodo che

va a recuperare il cognome, dato il nome completo di una persona, saremo abituati a scrivere qualcosa di simile all'esempio che segue:

Esempio 5.27

```
public string GetLastName(string fullName)
{
    var lastSpaceIndex = fullName.LastIndexOf(" ");

    return lastSpaceIndex == -1
        ? string.Empty
        : fullName.Substring(lastSpaceIndex + 1);
}
```

Il problema, come abbiamo già evidenziato, è che sebbene questo metodo funzioni egregiamente, va ad allocare continuamente memoria per effettuare i calcoli e restituire il nome come una nuova stringa, porzione del nome completo separata da uno spazio. Lo stesso esempio può essere riscritto facendo un uso migliore della memoria tramite `Span<T>` e `ReadOnlySpan<T>`:

Esempio 5.28

```
public ReadOnlySpan<char> GetLastName(ReadOnlySpan<char> fullName)
{
    var lastSpaceIndex = fullName.LastIndexOf(' ');

    return lastSpaceIndex == -1
        ? ReadOnlySpan<char>.Empty
        : fullName.Slice(lastSpaceIndex + 1);
}
```

Sebbene lo slicing sia un'ottima funzionalità per applicare una prima tipologia di filtro sugli array, è bene tenere presente che la stessa tecnica non è applicabile ad oggetti di tipo `List<T>` poiché questo oggetto non è di tipo `LinkedList<T>` e dunque non permette, almeno in modo diretto e senza perdita di performance, di lavorare con `Index`, `Range` e `Span<T>`. I tipi `Memory<T>` (e `ReadOnlyMemory<T>`) rappresentano la parte complementare di `Span<T>`, poiché non hanno le stesse limitazioni (in particolare non sono una `ref struct`) e pertanto possono lavorare anche su memoria heap, con collezioni, con `Stream` e con il pattern `async/await` (che vedremo nei capitoli successivi quando parleremo di programmazione asincrona). Essendo queste due strutture di tipo general purpose, sono meno efficienti di `Span<T>` ma permettono, nel

caso, di salvare le informazioni in un tipo, piuttosto che effettuare il boxing del tipo stesso.

Esempio 5.29

```
private void GetNames()
{
    // creazione di un buffer di lettura
    byte[] buffer = ArrayPool<byte>.Shared.Rent(16000 * 8);
    int bytesRead;

    // apertura del file
    var stream = new FileStream(@"C:\people.txt", FileMode.Open, FileAccess.Read);

    // lettura all'interno del buffer
    while ((bytesRead = stream.Read(buffer, 0, buffer.Length)) > 0)
    {
        // creazione dell'oggetto Memory<T>
        var memory = new ReadOnlyMemory<byte>(buffer, 0, length: bytesRead);

        // lettura come Span<T>
        ReadOnlySpan<byte> slice = memory.Span;

        // conversione e stampa su console...
    }
}
```

Come abbiamo dimostrato all'interno dell'[esempio 5.29](#), in questo caso `Span<T>` e `Memory<T>` lavorano insieme per prelevare i dati da uno stream, ovvero il file aperto in lettura e, in seconda istanza, per leggere dal buffer di volta in volta i nomi delle persone che sono presenti all'interno.

Conclusioni

In questo capitolo abbiamo parlato di alcuni concetti chiave per la realizzazione di applicazioni object oriented tramite C#.

Innanzitutto, abbiamo visto come l'infrastruttura delle collection e il gran numero di varianti presenti in .NET consentano di rappresentare diverse tipologie di strutture dati: `ArrayList` e `Hashtable` garantiscono una flessibilità estremamente più elevata rispetto ai semplici array, mentre `Stack` e `Queue` possono essere utilizzate per gestire problematiche più specifiche.

La seconda parte del capitolo, invece, è stata completamente dedicata ai tipi generici, mettendo in luce gli enormi vantaggi che la scrittura di codice fortemente tipizzato offre dal punto di vista dell'affidabilità del codice. Le collezioni generiche, infatti, permettono di evitare il ricorso alle conversioni di

tipo e di intercettare eventuali errori già in fase di compilazione. Abbiamo comunque mostrato come le potenzialità dei generics possano essere sfruttate anche nella stesura di codice personalizzato. Nella parte finale, invece, abbiamo introdotto il concetto dello slicing, ovvero della possibilità di filtrare oggetti conoscendone l'indice, tramite l'uso di `Index` e `Range`, piuttosto che creando delle finestre mobili utilizzando `Span<T>` che non creano ulteriori allocazioni in memoria, rendendo di fatto le applicazioni più performanti. Infine abbiamo visto le limitazioni di tutti questi nuovi tipi, che sono rappresentate maggiormente dalla loro struttura in memoria e quindi abbiamo visto come `Memory<T>` riesca a lavorare con molti più tipi – per via del fatto che la sua allocazione è a livello di heap – e a interfacciarsi comunque con `Span<T>`.

Nel prossimo capitolo riutilizzeremo questi concetti, esplorando un altro importante argomento nella programmazione orientata agli oggetti: i delegate e lo sviluppo di codice basato su eventi.

Delegate ed eventi

Finora abbiamo imparato a utilizzare C# per costruire oggetti, ossia entità autonome in grado di mantenere uno stato interno tramite i campi e di esporre proprietà e metodi in modo che essi possano interagire con altri oggetti, effettuando elaborazioni e scambiando dati. Spesso siamo abituati a pensare a questi ultimi sempre come stringhe, numeri, al limite anche ulteriori oggetti, ma tutti i linguaggi evoluti, in realtà, consentono di considerare come “dato” anche una funzione o una procedura. C# (o più precisamente .NET) non fa eccezione e, grazie all’infrastruttura dei delegate, permettono di memorizzare dei riferimenti a funzioni e trattarle come se fossero comuni variabili, che possono quindi essere assegnate o passate a metodi sotto forma di argomenti, oltre che, ovviamente, eseguite.

Proprio i delegate costituiscono l’argomento principale di questo capitolo: nella sua prima parte impareremo a conoscerli e a sfruttarne le potenzialità, mostrando come spesso rappresentino una soluzione elegante ed efficace a problematiche ricorrenti. Introdurremo anche una sintassi particolare per definire i delegate in maniera estremamente concisa ed elegante: le lambda expression. Queste ultime rappresentano un concetto di estrema importanza che risulterà indispensabile nei capitoli successivi, visto che è alla base della logica funzionale di LINQ.

Nella seconda parte del capitolo, invece, riusciremo finalmente a colmare un limite proprio degli oggetti che siamo stati in grado di costruire finora: basandoci sulle conoscenze dei delegate fin qui apprese, infatti, introdurremo il concetto di evento, tramite cui potremo fare in modo che i nostri oggetti abbiano la capacità di inviare notifiche ai loro utilizzatori, instaurando quindi una vera e propria comunicazione bidirezionale.

I Delegate in .NET

Quando, nel capitolo precedente, abbiamo parlato delle collection, non abbiamo citato alcun metodo per eseguire delle ricerche all'interno delle stesse. Sicuramente abbiamo la possibilità di scorrerle tramite il costrutto `foreach` e recuperare gli elementi desiderati, ma in realtà sono disponibili sistemi più avanzati ed eleganti: una `List<string>` per esempio, contiene un metodo `Find` che accetta come unico argomento un oggetto di tipo `Predicate<string>`, come mostra la [Figura 6.1](#):

```
// utilizzo del predicate come argomento di Find
var foundString = strings.Find(

```

```
string List<string>.Find(Predicate<string> match)
Searches for an element that matches the conditions defined by the specified predicate, and returns the first occurrence within the entire List<T>.
match: The Predicate<in T> delegate that defines the conditions of the element to search for.
```

Figura 6.1 – Intellisense del metodo Find.

Per il momento non addentriamoci nei dettagli su come utilizzare questo metodo, sarà tutto più chiaro tra poco, ma cerchiamo di capire meglio il significato di questa definizione e, in particolare, il ruolo di `Predicate<T>`. Quest'oggetto, contrariamente a tutti quelli che abbiamo incontrato finora, non è un dato inteso come un numero o una stringa, ma rappresenta, invece, una funzione che possiamo gestire come se fosse una normale variabile, tant'è che siamo in grado di passarla a un metodo come argomento. Per esempio, possiamo ricercare la prima stringa che inizia per la lettera "a", scrivendo un metodo `FindStringsWithA` e utilizzandolo come argomento per `Find` come nell'[esempio 6.1](#).

Esempio 6.1

```
private static bool FindStringsWithA(string item)
{
    return item.StartsWith("a");
}

public static void Main()
{
    var strings = new List<string>();

    strings.Add("This is a test");
    strings.Add("abcdefgh");
}
```

```
strings.Add("a1234567");

// Utilizzo del predicate come argomento di Find
var foundString = strings.Find(FindStringsWithA);

// Stampa abcdefgh sulla console
Console.WriteLine(foundString);
}
```

Il fatto che `FindStringsWithA` sia utilizzabile come criterio di ricerca dipende unicamente dalla sua firma: in generale infatti, un `Predicate<T>` è un metodo che accetta in ingresso un oggetto di tipo `T` e restituisce un `bool`; nel caso questo contratto non venga rispettato, il risultato che si ottiene è un errore in fase di compilazione.

Al di là del risultato finale del codice mostrato, che comunque può rivelarsi utile in molteplici occasioni, ciò che è importante comprendere è che in `C#` siamo effettivamente in grado di definire e istanziare puntatori a funzioni fortemente tipizzati, che all'interno di `.NET` sono chiamati **delegate**.

Definizione e utilizzo di un delegate

Un delegate è univocamente identificato da un nome e reca con sé le informazioni relative ai requisiti di forma, in termini di signature e tipo del risultato, che una funzione deve rispettare affinché possa essergli assegnata. `Predicate<T>`, che abbiamo visto nel paragrafo precedente, non è altro che uno delle centinaia di tipi di delegate all'interno del framework. Ovviamente in `C#` c'è la possibilità di definirne di personalizzati tramite l'utilizzo della parola chiave `delegate`.

Cerchiamo di capire, allora, come realizzarli e sfruttarli nel nostro codice per implementare un logger, una classe cioè in grado di memorizzare delle informazioni cronologiche scrivendole su diversi supporti, quali file, e-mail o la stessa console dell'applicazione. Invece di essere costretti a realizzare diverse versioni della classe `Logger` per ognuno di essi, o magari di utilizzare molteplici blocchi `if` al suo interno, possiamo pensare di definire un delegate apposito, dichiarandolo come se si trattasse di una qualsiasi classe:

```
public delegate void StringLogWriter(
    DateTime timestamp, string message);
```

Si tratta di una soluzione estremamente versatile, perché in questo modo può essere l'utente stesso a decidere su quale supporto scrivere i messaggi di log. Il codice completo della classe `Logger` è indicato nell'[esempio 6.2](#).

Esempio 6.2

```
public delegate void StringLogWriter(
    DateTime timestamp, string message);

public class Logger
{
    private StringLogWriter writer;

    public Logger(StringLogWriter writer)
    {
        this.writer = writer;
    }

    public void Log(string message)
    {
        if (this.writer != null)
            this.writer(DateTime.Now, message);
    }
}
```

Come possiamo notare, essa accetta uno `StringLogWriter` come argomento del costruttore, per poi utilizzarlo all'interno del metodo `Log` allo stesso modo di come faremmo con una normale procedura. In realtà, dietro le quinte lo statement:

```
this.writer(DateTime.Now, message);
```

viene sostituito con la chiamata al metodo `Invoke` del delegate che abbiamo definito:

```
this.writer.Invoke(DateTime.Now, message);
```

Per utilizzare la classe `Logger`, a questo punto, è sufficiente creare una procedura che rispetti la firma stabilita da `StringLogWriter` e utilizzarla per

crearne un'istanza, referenziandola nel costruttore in modo simile all'[esempio 6.3](#).

Esempio 6.3

```
private static void ConsoleWriter(DateTime timestamp, string message)
{
    Console.WriteLine("{0} - {1}", timestamp, message);
}

public static void Main()
{
    var myLogger = new Logger(ConsoleWriter);

    // Stampa sulla console il messaggio in basso
    myLogger.Log("Messaggio di esempio");
}
```

Fino a questo punto abbiamo appreso come definire, istanziare e utilizzare un delegate, e abbiamo anche accennato al fatto che esso, come ogni altra entità in .NET, è rappresentato da un vero e proprio modello a oggetti: proprio quest'ultimo sarà argomento del prossimo paragrafo.

Modello a oggetti dei delegate

Quando dichiariamo un delegate in C#, ciò che accade dietro le quinte è che il compilatore genera per noi un particolare tipo di classe, come possiamo facilmente verificare con un qualsiasi strumento in grado di esplorare il contenuto di un assembly, come ILDASM o Reflector. Per esempio, il progetto del paragrafo precedente contiene la definizione di un oggetto di tipo `StringLogWriter` simile a quello della [Figura 6.2](#).

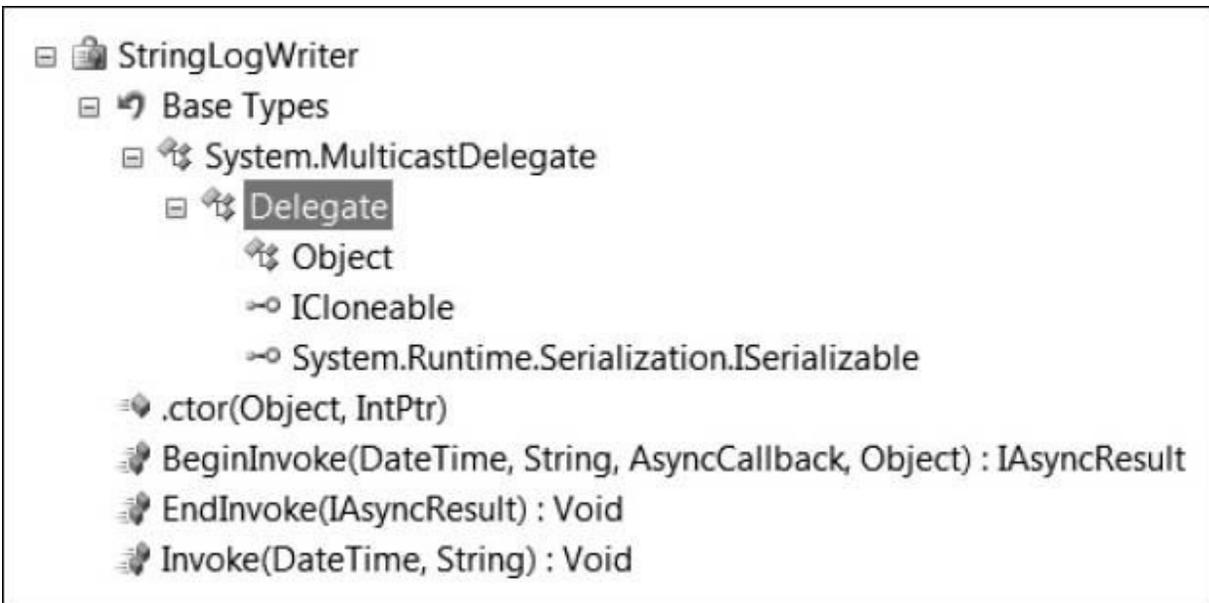


Figura 6.2 – Gerarchia di StringLogWriter.

Al suo interno troviamo il metodo `Invoke`, già citato in precedenza, che corrisponde esattamente alla firma del delegate stesso e che consente l'esecuzione della procedura assegnata. Accanto a quest'ultimo, `BeginInvoke` e `EndInvoke` realizzano il supporto a quello che si chiama **Asynchronous Programming Model (APM)**, ossia la capacità da parte di un delegate di essere eseguito in un thread separato rispetto al chiamante; si tratta di un concetto avanzato, che verrà comunque trattato in un prossimo paragrafo e successivamente esteso in un capitolo interamente dedicato alle tecniche di programmazione multithreading.

Scorrendo la gerarchia di classi, vediamo che, in ultima analisi, `StringLogWriter` eredita dalla classe `System.Delegate`. Quest'ultima, oltre a contenere la logica implementativa necessaria al funzionamento dei delegate, espone una coppia di proprietà che consentono di recuperare informazioni relative al metodo a cui il delegate punta; in particolare:

- ❑ `Method` contiene informazioni relative al metodo e alla sua firma, come il tipo che lo definisce, i tipi degli argomenti in ingresso e quello dell'eventuale risultato;
- ❑ `Target` contiene un riferimento alla specifica istanza a cui il metodo assegnato al delegate appartiene.

Per capire meglio questo concetto, consideriamo il codice dell'[esempio 6.4](#).

Esempio 6.4

```
public class FileLogWriter
{
    public string FileName { get; set; }

    public FileLogWriter(string filename)
    {
        this.FileName = filename;
    }

    public void Write(DateTime timestamp, string message)
    {
        // Scrittura messaggio su file...
    }
}

public static void Main()
{
    var writer = new FileLogWriter(@"c:\somefile.txt");
    var writerDelegate = new StringLogWriter(writer.Write);

    Console.WriteLine(writerDelegate.Method);

    Console.WriteLine(
        ((FileLogWriter) writerDelegate.Target).FileName);
}
```

Esso contiene la definizione di una classe in grado di scrivere i messaggi di log su file, la cui procedura `write` viene poi utilizzata per creare un'istanza di `StringLogWriter`. Si tratta di una novità, rispetto a quanto abbiamo visto nel paragrafo precedente, in cui invece abbiamo costruito delegate associati a procedure statiche. In questo caso, il delegate punta a un metodo di istanza e pertanto la proprietà `Target` contiene il riferimento all'istanza di `FileLogWriter` a cui esso appartiene. Il risultato di tale codice è quello mostrato nella [Figura 6.3](#).

```
CA\WINDOWS\system32\cmd.exe
Void Write(System.DateTime, System.String)
c:\somefile.txt
Press any key to continue . . .
```

Figura 6.3 – Output dell’[esempio 6.4](#).

Vale la pena rimarcare ancora una volta che, quando si costruisce un delegate tramite un metodo di istanza, come nel caso dell’esempio precedente, esso mantiene un riferimento all’oggetto a cui tale metodo appartiene. Ciò vuol dire che tale oggetto non potrà essere distrutto dal Garbage Collector fintanto che non verrà rimosso dal delegate stesso. Questo concetto avrà un’implicazione importante nella seconda parte del capitolo, quando parleremo degli eventi.

Negli esempi che abbiamo visto finora, abbiamo sempre costruito delegate che puntassero a una singola procedura. Nel prossimo paragrafo cercheremo di capire quali sono gli strumenti che ci consentono di superare questo limite.

[Combinazione di delegate: la classe MulticastDelegate](#)

Una caratteristica che rende i delegate estremamente potenti e versatili è costituita dalla capacità di mantenere al loro interno una **invocation list** e quindi, di fatto, puntare contemporaneamente più metodi da eseguire poi in sequenza. Questa componibilità è internamente implementata dalla classe MulticastDelegate, anch’essa, come System.Delegate, classe base per tutti i delegate, e può essere sfruttata utilizzando il metodo statico Combine, come mostrato nell’[esempio 6.5](#).

Esempio 6.5

```

// Definizione del primo delegate
var writer = new FileLogWriter(@"c:\somefile.txt");
var fileDelegate = new StringLogWriter(writer.Write);

// Definizione del secondo delegate
var consoleDelegate = new StringLogWriter(ConsoleWriter);

// Combinazione dei delegate
StringLogWriter combinedDelegate =
    (StringLogWriter)Delegate.Combine(consoleDelegate, fileDelegate);

var myLogger = new Logger(combinedDelegate);

// Scrive sulla console e su file il messaggio in basso
myLogger.Log("Messaggio di esempio");

```

In questo codice, due differenti istanze di `StringLogWriter` vengono utilizzate per crearne una terza, `combinedDelegate`, con cui poi effettivamente inizializzare la classe `Logger`. Il risultato che otteniamo è che ogni chiamata al metodo `Log` provoca l'esecuzione di entrambi i delegate, e il tutto avviene in maniera trasparente per la nostra classe `Logger`, tant'è che non siamo stati costretti ad apportare alcuna modifica al codice scritto nei paragrafi precedenti.

Quando si utilizza il metodo `Delegate.Combine` bisogna comunque fare attenzione ad alcuni aspetti: innanzi tutto, i delegate da combinare devono essere tutti dello stesso tipo, altrimenti viene sollevato un errore a runtime; in secondo luogo, come si nota dal codice precedente, questo metodo restituisce un oggetto di tipo `Delegate` e pertanto, prima di poterlo effettivamente utilizzare, è necessario effettuare un'operazione di casting.

Come è facile comprendere, i multicast delegate risultano estremamente utili quando si vogliono concatenare più invocazioni a procedure in maniera trasparente per l'utilizzatore. Quando sono però applicati a funzioni, è importante avere cognizione del fatto che il valore di ritorno restituito dalla chiamata è sempre pari a quello dell'ultima funzione invocata.

Altri due metodi della classe `Delegate` che risultano utili nella gestione dei multicast delegate sono `GetInvocationList`, tramite cui è possibile recuperare l'elenco dei delegate che compongono la sua lista, e `Remove`, che consente di creare un multicast delegate a partire da un delegate esistente, rimuovendo un membro dalla sua invocation list. Il loro utilizzo è mostrato nell'[esempio 6.6](#).

Esempio 6.6

```
StringLogWriter combinedDelegate =
    (StringLogWriter)Delegate.Combine(consoleDelegate, fileDelegate);

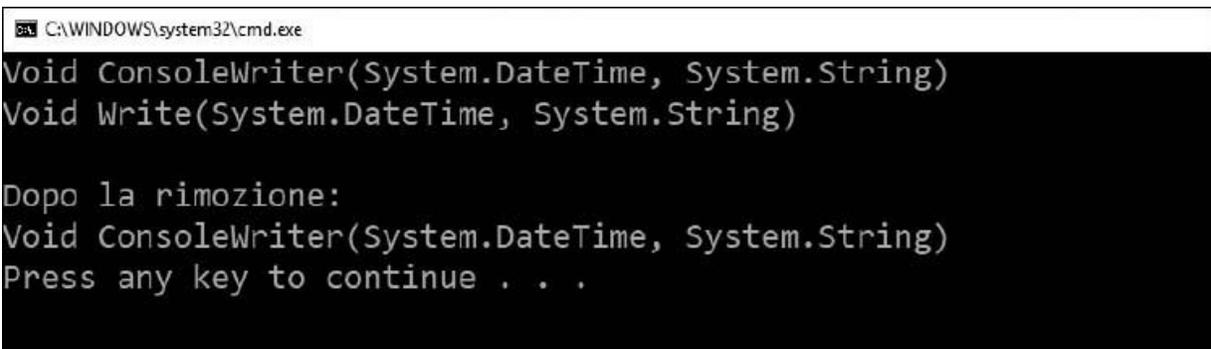
foreach (Delegate item in combinedDelegate.GetInvocationList())
{
    Console.WriteLine(item.Method);
}

combinedDelegate = (StringLogWriter)
    Delegate.Remove(combinedDelegate, fileDelegate);

Console.WriteLine("\r\nDopo la rimozione:");

foreach (Delegate item in combinedDelegate.GetInvocationList())
{
    Console.WriteLine(item.Method);
}
```

Il codice precedente utilizza questi due metodi per visualizzare e manipolare lo stesso `combinedDelegate` che abbiamo introdotto all'inizio di questo paragrafo, in particolare creando un nuovo delegate dopo aver rimosso uno dei suoi membri dalla invocation list. L'output di questo snippet di codice è quello mostrato nella [Figura 6.4](#).



```
C:\WINDOWS\system32\cmd.exe
Void ConsoleWriter(System.DateTime, System.String)
Void Write(System.DateTime, System.String)

Dopo la rimozione:
Void ConsoleWriter(System.DateTime, System.String)
Press any key to continue . . .
```

Figura 6.4 – Output dell'[esempio 6.6](#).

[Cenni sull'esecuzione asincrona di un delegate](#)

Le applicazioni che abbiamo utilizzato negli esempi visti finora presentano tutte la caratteristica di essere eseguite in un singolo thread: si tratta di un concetto avanzato, che sarà spiegato in maniera estesa nel corso del libro, ma che sostanzialmente può tradursi nel fatto che i vari statement sono

processati in sequenza, facendo sì che al termine di ognuno si avvii l'esecuzione del successivo.

Quando un metodo coinvolge risorse tipicamente lente, quali la rete o dispositivi di input/output, il flusso dell'applicazione resta bloccato fintanto che questa operazione non viene completata. In situazioni simili, il modo migliore per ottimizzare le prestazioni dell'applicazione è quello di sfruttare le funzionalità di esecuzione asincrona dei delegate tramite la coppia di metodi `BeginInvoke` ed `EndInvoke`. Per capire meglio quali sono i vantaggi, guardiamo il codice dell'[esempio 6.7](#).

Esempio 6.7

```
delegate string SampleDelegate(string input);

public static string VeryLongEchoFunction(string input)
{
    Thread.Sleep(3000);
    return "Hello " + input;
}

public static void Main()
{
    var myEcho = new SampleDelegate(VeryLongEchoFunction);

    Console.WriteLine(myEcho("Marco"));
    Console.WriteLine(myEcho("Daniele"));
    Console.WriteLine(myEcho("Matteo"));
}
```

In esso abbiamo reso la funzione `VeryLongEchoFunction` artificialmente lenta, utilizzando l'istruzione `Thread.Sleep` che ne blocca l'esecuzione per tre secondi, in modo da simulare la latenza tipica, per esempio, dell'interazione con un dispositivo di rete. Il risultato è che l'applicazione impiega circa nove secondi per essere completata, intervallo pari alla somma del tempo necessario per completare le tre invocazioni al delegate.

In realtà, durante tutto questo tempo, l'applicazione resta quasi sempre in attesa, senza far nulla, e pertanto si tratta di uno dei tipici casi in cui l'esecuzione asincrona può fare effettivamente la differenza. Proviamo allora a riscrivere il metodo `Main` per sfruttare questa funzionalità dei delegate.

Esempio 6.8

```
public static void Main()
{
    var myEcho = new SampleDelegate(VeryLongEchoFunction);

    var results = new List<IAsyncResult>();

    results.Add(myEcho.BeginInvoke("Marco", null, null));
    results.Add(myEcho.BeginInvoke("Daniele", null, null));
    results.Add(myEcho.BeginInvoke("Matteo", null, null));

    // Recupero dei risultati e stampa sulla console.
    foreach (IAsyncResult result in results)
    {
        Console.WriteLine(myEcho.EndInvoke(result));
    }
}
```

In questa versione, invece di utilizzare `Invoke` come in precedenza, l'esecuzione del delegate viene scatenata tramite il metodo `BeginInvoke`, anch'esso generato automaticamente dal compilatore in base alla dichiarazione di `SampleDelegate` e della sua signature. In questa fase non ci interessa entrare nel dettaglio degli argomenti addizionali che esso prevede, bensì è importante capire la sostanziale differenza rispetto all'esempio precedente: ognuno dei metodi `BeginInvoke`, infatti, ha la caratteristica di avviare l'esecuzione di `VeryLongEchoFunction` in un thread parallelo e ritornare immediatamente il controllo al thread principale, che, quindi, può passare subito a processare la chiamata successiva. Il risultato consiste nel fatto che le tre invocazioni vengono scatenate parallelamente, come schematizzato nella [Figura 6.5](#).

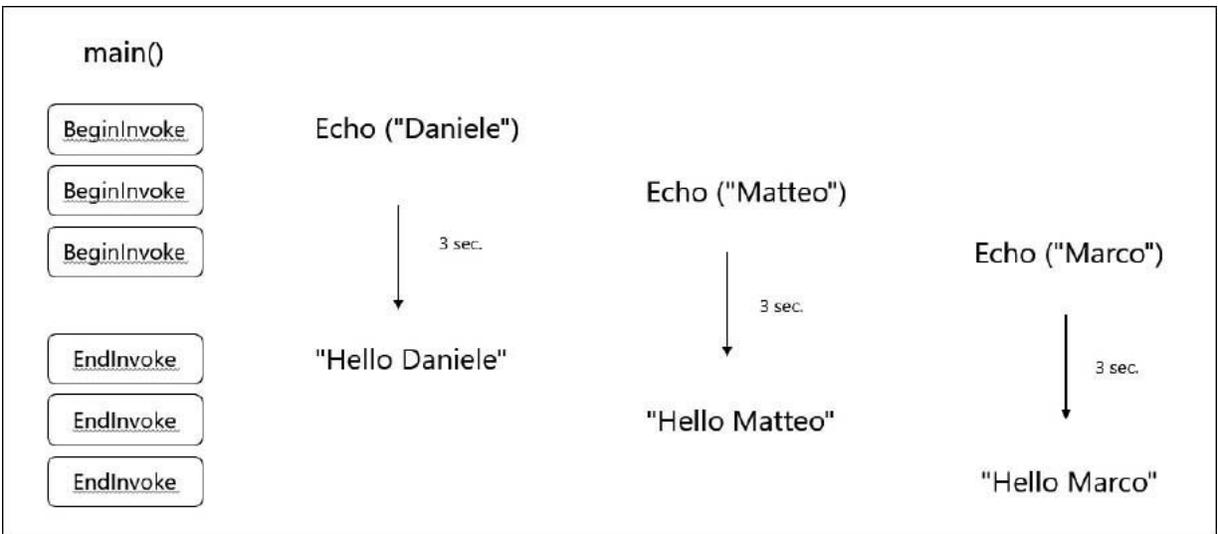


Figura 6.5 – Schema di esecuzione multithreaded.

Per recuperare il valore di ritorno di `VeryLongEchoFunction` e stamparlo finalmente sulla console bisogna, invece, utilizzare il metodo `EndInvoke` che accetta come argomento l'oggetto di tipo `IAsyncResult` restituito da ognuna delle invocazioni precedenti. `EndInvoke` ovviamente blocca l'esecuzione del thread principale fino al termine di `VeryLongEchoFunction`, ma, il fatto che le tre elaborazioni vengano eseguite quasi contemporaneamente, consente comunque all'applicazione di essere decisamente più veloce rispetto alla versione dell'[esempio 6.7](#).

I delegate e i generics

Essendo i delegate delle classi, è possibile sfruttare i generics per creare delle strutture facilmente riutilizzabili e versatili e pertanto sono supportate tutte le funzionalità che abbiamo imparato a conoscere nel [Capitolo 5](#).

Esempio 6.9

```
public delegate string MyDelegate<T>(T input) where T : struct;
private static string DateTimeWriter(DateTime input)
{
    return input.ToShortDateString();
}
public static void Main()
```

```
{
    var sample = new MyDelegate<DateTime>(DateTimeWriter);
}
```

Nell'[esempio 6.9](#), abbiamo definito un delegate impostando un vincolo sul tipo generico, che deve essere un tipo di valore. In .NET sono già disponibili un gran numero di delegate generici utilizzabili, e in particolare:

- ❑ `Action<T1, ..., T8>`: si tratta di una serie di delegate che rappresentano una procedura che accetta fino a otto differenti argomenti in ingresso;
- ❑ `Func<T1, ..., TResult>`: consiste in otto differenti delegate per rappresentare funzioni che accettano fino a otto differenti argomenti per restituire un tipo `TResult`.

Usando queste ultime, per esempio, possiamo riscrivere il codice precedente come nell'[esempio 6.10](#).

Esempio 6.10

```
private static string DateTimeWriter(DateTime input)
{
    return input.ToShortDateString();
}

public static void Main()
{
    var sample = new Func<DateTime, string>(DateTimeWriter);
}
```

Come vedremo più avanti nel libro, `Action` e `Func` rivestono una fondamentale importanza all'interno della tecnologia LINQ, dato che gran parte dei metodi che ne costituiscono l'infrastruttura utilizzano (seppur indirettamente) proprio queste tipologie di delegate per costruire espressioni complesse. In generale, però, non è comodo utilizzare in maniera intensiva i delegate se, per ognuno di essi, siamo costretti a realizzare una funzione o una procedura in qualche altra parte del nostro codice. Per ovviare a questo inconveniente, C# supporta due differenti sintassi, entrambe più compatte, per rappresentarli; esse saranno argomento del prossimo paragrafo.

Anonymous method e lambda expression

Quando all'inizio del capitolo abbiamo mostrato come, tramite il metodo `Find` e l'utilizzo di un `Predicate`, sia possibile ricercare elementi all'interno di una lista di oggetti, siamo stati costretti a scrivere parecchio codice per dichiarare la funzione di ricerca e utilizzarla per inizializzare il `delegate`.

```
private static bool FindStringsWithA(string item)
{
    return item.StartsWith("a");
}

public static void Main()
{
    var strings = new List<string>();
    // ...
    var foundString = strings.Find(FindStringsWithA);
}
```

La medesima funzionalità può essere ottenuta tramite una sintassi più breve, che ci consente di definire la funzione in-line, ossia direttamente nel momento in cui invociamo il metodo `Find`.

Esempio 6.11

```
public static void Main()
{
    var strings = new List<string>();
    // ...
    var foundString = strings.Find(
        delegate(string item)
        {
            return item.StartsWith("a");
        });
}
```

La forma sintattica utilizzata nell'[esempio 6.11](#) prende il nome di **anonymous method**, grazie al quale possiamo specificare un `delegate` senza per forza dover definire esplicitamente un metodo; questo compito è, infatti, assolto dal compilatore, che si occupa di generarlo in maniera automatica. Quando, come nell'[esempio 6.11](#), il `delegate` è in realtà

composto da una sola istruzione, è possibile esprimerlo in una forma ancora più contratta:

Esempio 6.12

```
public static void Main()
{
    var strings = new List<string>();
    // ...
    var foundString = strings.Find(i => i.StartsWith("a"));
}
```

La sintassi utilizzata nell'[esempio 6.12](#) prende il nome di **lambda expression** e risulta di gran lunga la modalità più utilizzata e semplice per istanziare delegate in C#. Cerchiamo di comprenderne l'anatomia.

La prima parte è costituita dalla dichiarazione degli argomenti dell'espressione; solitamente vanno specificati utilizzando le parentesi tonde, che possono essere omesse nel caso in cui il numero dei parametri sia uguale a uno:

```
(a, b) => a + b;    // Lambda expression con due parametri

() => someMethod(); // Lambda expression priva di parametri

(a) => someMethod(a); // Parentesi sono opzionali
a => someMethod(a);
```

Come possiamo notare, non è necessario specificarne il tipo, visto che il compilatore C# è in grado di determinarlo in base al contesto in cui l'espressione è utilizzata.

La seconda metà della lambda expression è il vero e proprio codice, in cui risiede la logica dell'espressione stessa e, nella forma utilizzata finora, ammette un unico statement; pertanto, per esempio, invece del costrutto `if..else` siamo costretti a utilizzare l'operatore condizionale `?`. Un'ultima osservazione, infine, riguarda il fatto che il risultato dello statement è automaticamente utilizzato come valore di ritorno dell'espressione e quindi la parola chiave `return` è implicita e non dev'essere indicata.

Nel caso in cui si voglia utilizzare questa notazione per esprimere logiche più complesse, è possibile scrivere metodi multi-riga utilizzando la

seguinte sintassi:

```
var foundString = strings.Find(  
    i =>  
    {  
        statement1();  
        statement2();  
        return i.StartsWith("a");  
    }  
);
```

Attenzione che, in questo caso, l'uso della parola chiave `return` torna a essere necessario.

Una funzionalità estremamente comoda e interessante delle lambda expression è costituita dalla cosiddetta **closure**, ossia dalla possibilità di accedere, dal corpo dell'espressione, a eventuali altre variabili definite nel metodo in cui la lambda è dichiarata; l'[esempio 6.13](#) mostra come possiamo sfruttare tale caratteristica per ricercare stringhe che inizino per un qualsiasi carattere.

Esempio 6.13

```
var strings = new List<string>();  
// ...  
var searchString = "B";  
var foundString = strings.Find(i => i.StartsWith(searchString));
```

Implementare una tale logica tramite i soli delegate, avrebbe richiesto la costruzione di una classe ausiliaria, detta per l'appunto closure, che in questo caso è invece automaticamente generata dal compilatore.

A partire da C# 6, le lambda expression possono anche essere utilizzate per definire metodi, al posto dei consueti blocchi di codice:

```
public static string SayHello(string name) => "Hello " + name;
```

I delegate in .NET, insomma, non sono solo semplici puntatori a funzione, ma veri e propri oggetti dotati di funzionalità avanzate e di un ottimo supporto da parte del compilatore. Essi sono alla base dell'infrastruttura degli eventi, che esamineremo nei successivi paragrafi di questo capitolo.

I delegate come strumento di notifica: gli eventi

Oltre a quello visto nelle pagine precedenti, l'ulteriore tipico campo di utilizzo dei delegate è in tutte quelle occasioni in cui si voglia fare in modo che una classe sia in grado di fornire delle notifiche ai suoi utilizzatori. Supponiamo, per esempio, di dover leggere un dato da una porta del PC e di realizzare quindi una classe che si occupi di incapsulare tutta la logica di comunicazione, come quella dell'[esempio 6.14](#).

Esempio 6.14

```
public class PortReceiver
{
    public delegate void DataReceivedCallback();
    private DataReceivedCallback callback;

    public void Subscribe(DataReceivedCallback callback)
    {
        this.callback = callback;
    }

    public string Data { get; set; }

    public void ReceiveData()
    {
        this.Data = "Data received";
        if (this.callback != null)
            this.callback();
    }
}
```

La classe `PortReceiver` definisce al suo interno un particolare tipo di delegate, chiamato `DataReceivedCallback`, che viene invocato ogni volta che vengono ricevuti dei dati. Un oggetto che voglia ricevere una notifica dell'avvenuta ricezione del dato, deve utilizzare il metodo `subscribe` per fornire a `PortReceiver` una propria procedura da eseguire al verificarsi di questo evento, come avviene nell'[esempio 6.15](#).

Esempio 6.15

```
public class SomeClass
```

```

{
    private PortReceiver receiver;

    public SomeClass(PortReceiver receiver)
    {
        this.receiver = receiver;
        this.receiver.Subscribe(this.DataReceived);
    }

    private void DataReceived()
    {
        Console.WriteLine(this.receiver.Data);
    }
}

```

In questo modo, siamo riusciti effettivamente a instaurare una comunicazione bidirezionale tra i due oggetti, senza tra l'altro introdurre dipendenze all'interno di `PortReceiver`, che, effettivamente, non ha cognizione alcuna di chi sia il proprio sottoscrittore, a parte il fatto che contiene una procedura con una signature ben definita. Siamo stati però costretti a scrivere una discreta quantità di codice che andrebbe replicato per ogni nuova tipologia di notifica, introducendo, di volta in volta, un'ulteriore versione del metodo `Subscribe`.

Fortunatamente tutto questo sforzo non è necessario, perché .NET contiene già gli strumenti adatti a sopperire a questo tipo di necessità: gli **eventi**.

Definizione e uso di un evento in un oggetto

Gli eventi rappresentano in C# il modo più efficace e semplice per rendere possibile l'invio di notifiche da parte di un oggetto a un numero arbitrario di sottoscrittori. L'[esempio 6.16](#) mostra come possiamo riscrivere `PortReceiver` per dotarlo dell'evento `PortDataReceived`, utilizzando la parola chiave `event`.

Esempio 6.16

```

public class PortReceiver
{
    public delegate void PortDataReceivedEventHandler(
        PortReceiver receiver);
    public event PortDataReceivedEventHandler PortDataReceived;

    public string Data { get; set; }
}

```

```

public void ReceiveData()
{
    this.Data = "Data received";
    if (this.PortDataReceived != null)
        this.PortDataReceived(this);
}

// .. codice di interfacciamento con la porta ..
}

```

La signature del nuovo evento è univocamente identificata dal delegate a questo associato, nel nostro caso `PortDataReceivedEventHandler`. Non è più necessario esporre un metodo `subscribe` e, per inviare una notifica ai sottoscrittori, possiamo utilizzare la sintassi mostrata nel metodo `ReceiveData`, mediante la quale verificare la presenza di almeno un gestore ed eventualmente eseguire il delegate, valorizzandone i parametri in modo opportuno.

Nell'accezione propria di .NET, il metodo che viene invocato in corrispondenza del sollevamento di un evento è chiamato **handler** e, di riflesso, per convenzione si usa denominare il delegate utilizzato nella dichiarazione dell'evento usando il suffisso "EventHandler". Per agganciare un handler possiamo utilizzare la sintassi dell'[esempio 6.17](#), e in particolare l'operatore `+=`, specificando un metodo la cui firma corrisponda esattamente a quella dell'evento stesso, pena un errore in compilazione.

Esempio 6.17

```

public class SomeClass
{
    public SomeClass(PortReceiver receiver)
    {
        receiver.PortDataReceived += DataReceived;
    }

    private void DataReceived(PortReceiver receiver)
    {
        Console.WriteLine(receiver.Data);
    }
}

```

Dato che, come abbiamo già accennato, gli eventi sono gestiti nel framework tramite l'utilizzo dei delegate, è ovviamente ancora valida la peculiarità dei multicast delegate, ovvero la possibilità di concatenare più

chiamate: grazie a quest'ultima, a ogni evento possono essere associati più gestori e il tutto avviene in maniera trasparente, senza essere costretti ad adottare alcun accorgimento nella scrittura delle definizioni degli stessi.

Come abbiamo visto in precedenza, delegate legati a metodi di istanza mantengono un riferimento a quest'ultima, impedendo quindi che il Garbage Collector possa rimuoverla dalla memoria. Ciò è ovviamente valido anche nel caso degli handler a eventi; è possibile cancellare la sottoscrizione grazie all'operatore -=.

Ora che conosciamo le nozioni di base della gestione degli eventi in C#, è tempo di capire quali sono le possibilità di personalizzazione di questi strumenti e le best practice consigliate.

[Creare eventi personalizzati](#)

Nel paragrafo precedente abbiamo accennato al fatto che gli eventi, in .NET, poggiano le loro fondamenta sull'infrastruttura dei delegate e abbiamo visto come possiamo utilizzarli per specificare la firma che un metodo deve avere per poter essere ritenuto un gestore valido. Un aspetto migliorabile del codice scritto nel paragrafo precedente, però, riguarda il fatto che solleviamo l'evento direttamente all'interno del metodo `ReceiveData`.

Infatti, sebbene tutto funzioni correttamente, la nostra attuale implementazione non consente a un'eventuale classe derivata, di modificare le logiche che determinano se sollevare o meno l'evento stesso. Per questa ragione, una *best practice* è quella di utilizzare allo scopo un metodo definito come `protected virtual`, il cui nome convenzionalmente è "On", seguito dal nome dell'evento, e di invocare l'eventuale gestore solo mediante quest'ultimo. A valle di queste considerazioni, allora, l'implementazione di `PortReceiver` diviene quella dell'[esempio 6.18](#).

Esempio 6.18

```
public class PortReceiver
{
```

```

public delegate void PortDataReceivedEventHandler(
    PortReceiver receiver);
public event PortDataReceivedEventHandler PortDataReceived;

protected virtual void OnPortDataReceived()
{
    if (this.PortDataReceived != null)
        this.PortDataReceived(this);
}

public string Data { get; set; }

public void ReceiveData()
{
    this.Data = "Data received";
    this.OnPortDataReceived();
}

// .. codice di interfacciamento con la porta ..
}

```

Ora che abbiamo capito quali sono i modi migliori per strutturare un oggetto che sia in grado di sollevare eventi, ci resta ancora da capire come utilizzarli per scambiare informazioni con i sottoscrittori e questo sarà argomento del prossimo paragrafo.

[Scambiare dati tramite eventi: la classe EventArgs e le sue derivate](#)

Quando abbiamo realizzato `PortDataReceived` non abbiamo posto più di tanto l'enfasi sulla particolare signature che esso deve avere; nel corso degli esempi che abbiamo mostrato siamo arrivati a un'implementazione in cui, quasi per caso, abbiamo deciso di includere l'istanza di `PortReceiver` responsabile di aver sollevato l'evento. Si tratta invece di un dato che dovrebbe essere sempre obbligatoriamente inviato: una best practice che Microsoft consiglia di adottare, per gli eventi più semplici, la signature:

```

delegate void MyEventHandler(object sender, EventArgs e);

```

tanto che .NET contiene un delegate apposito, chiamato `EventHandler`.

Il secondo argomento della signature è un'istanza della classe `EventArgs`; si tratta di un tipo che non contiene alcuna informazione, ma la cui adozione è comunque consigliata per un requisito di forma che sarà più chiaro fra poco. Il modo più corretto e semplice per valorizzarlo è tramite il suo campo statico `Empty`:

```
if (this.PortDataReceived != null)
    PortDataReceived(this, EventArgs.Empty);
```

Nel caso in cui sia invece necessario inviare informazioni di stato, il consiglio è quello di realizzare una classe personalizzata che derivi da `EventArgs` e il cui nome convenzionalmente termini con tale suffisso. Pertanto, se, per esempio, volessimo passare tramite `PortDataReceived` anche il dato appena ricevuto, dovremmo definire un nuovo oggetto `PortDataReceivedEventArgs` e modificare il relativo delegate. L'[esempio 6.19](#) mostra l'implementazione di `PortReceiver` comprensiva di queste ulteriori migliorie.

Esempio 6.19

```
public class PortDataReceivedEventArgs : EventArgs
{
    public string Data { get; set; }
}

public class PortReceiver
{
    public delegate void PortDataReceivedEventHandler(
        object sender, PortDataReceivedEventArgs e);
    public event PortDataReceivedEventHandler PortDataReceived;

    protected virtual void OnPortDataReceived(string data)
    {
        if (this.PortDataReceived != null)
        {
            var e = new PortDataReceivedEventArgs();
            e.Data = data;
            this.PortDataReceived(this, e);
        }
    }

    public string Data { get; set; }

    public void ReceiveData()
    {
        this.Data = "Data received";
        this.OnPortDataReceived(Data);
    }

    // .. codice di interfacciamento con la porta ..
}
```

Come possiamo notare, l'introduzione di `PortDataReceivedEventArgs` comporta anche la modifica di `OnPortDataReceived`, che possiamo però

sfruttare per incapsulare tutta la logica di creazione degli argomenti dell'evento.

Anche i tipi in .NET seguono le regole che abbiamo enunciato per dotare gli eventi di informazioni di stato e, pertanto, esiste un gran numero di classi che derivano da EventArgs (insieme, ovviamente, ai relativi delegate) che possiamo eventualmente sfruttare nel nostro codice. CancelEventArgs, per esempio, contiene una proprietà booleana chiamata Cancel ed è tipicamente utilizzata come argomento dei cosiddetti eventi di preview, cioè eventi sollevati prima di una certa operazione, per dare la possibilità all'utilizzatore di cancellarne l'esecuzione. L'[esempio 6.20](#) mostra come sfruttare questa funzionalità all'interno di PortReceiver.

Esempio 6.20

```
public class PortReceiver
{
    // .. altro codice qui ..

    public event CancelEventHandler PortDataReceiving;

    protected virtual void OnPortDataReceiving(CancelEventArgs e)
    {
        if (this.PortDataReceiving != null)
            this.PortDataReceiving(this, e);
    }

    public void ReceiveData()
    {
        var e = new CancelEventArgs(false);
        this.OnPortDataReceiving(e);

        // qui verificiamo se un sottoscrittore
        // ha cancellato la ricezione dati
        if (!e.Cancel)
        {
            this.Data = "Data received";
            this.OnPortDataReceived(Data);
        }
    }
}
```

In generale, per ogni tipologia di classe derivante da EventArgs, è disponibile il delegate generico EventHandler<T> che possiamo utilizzare se non riteniamo necessario definirne uno personalizzato.

In quest'ultimo esempio è lampante che l'utilizzo dei delegate come base per l'infrastruttura degli eventi in .NET comporti un

differenti modo di ragionare rispetto al passato; dato che un evento può avere sottoscrittori multipli, il modo migliore per far sì che anche questi ultimi possano condividere informazioni di stato è proprio quello di utilizzare una classe EventArgs, ossia un tipo di riferimento, e far sì che la medesima istanza venga inviata a ognuno di essi. Una funzione con un risultato di tipo bool, invece, avrebbe dato solo all'ultimo metodo della invocation list il privilegio di impostare o meno la cancellazione dell'evento.

Un ulteriore aspetto riguarda la ragione per cui è consigliato l'uso della signature standard, composta dai due argomenti di tipo object e EventArgs (o una sua classe derivata), per definire un evento anche nel caso in cui esso non debba tramandare informazioni di stato. Il vantaggio risiede nel fatto che, grazie alla controvarianza dei delegate, possiamo pensare di creare dei gestori universali, in grado cioè di essere assegnati a qualsiasi evento. Ciò non sarebbe ovviamente possibile se ognuno esponesse un numero diverso di argomenti o se questi non appartenessero alla medesima gerarchia.

Definizione esplicita di eventi

In alcuni casi può essere necessario eseguire del codice personalizzato in corrispondenza dell'aggiunta o rimozione di un handler a un evento, o in corrispondenza della sua esecuzione. Quelle che abbiamo utilizzato finora per definire PortDataReceiving e PortDataReceived sono le forme contratte (ma allo stesso tempo più utilizzate) per la definizione di eventi. Alla stessa stregua di ciò che accade per le proprietà, infatti, possiamo utilizzare la forma estesa dell'[esempio 6.21](#).

Esempio 6.21

```
private EventHandler customEvent;
public event EventHandler CustomEvent
{
    add
    {
        this.customEvent += value;
    }
    remove
}
```

```
{
    this.customEvent -= value;
}
}
```

In questo modo si possono ottenere effetti particolarmente interessanti, come far sì che un evento sia sempre gestito al massimo da un solo handler, in modo simile all'[esempio 6.22](#).

Esempio 6.22

```
private EventHandler customEvent;
public event EventHandler CustomEvent
{
    add
    {
        this.customEvent = value;
    }
    remove
    {
        this.customEvent = null;
    }
}
```

Conclusioni

In questo capitolo abbiamo introdotto una serie di concetti di fondamentale importanza per costruire applicazioni complesse in C#. I delegate rappresentano in .NET quello che i puntatori a funzione sono in linguaggi meno evoluti e, oltre alle funzionalità base, incapsulano logiche complesse quali multicasting o esecuzione asincrona. Una certa complessità per quanto riguarda il codice necessario a utilizzarli è compensata dall'introduzione delle lambda expression, che consentono di scrivere delegate semplici in una sola riga di codice.

Nella seconda parte del capitolo abbiamo visto come possiamo dotare i nostri tipi della capacità di inviare notifiche all'esterno: ne abbiamo mostrato le caratteristiche principali, indicando le best practice da adottare in fase di definizione e come sia anche possibile personalizzarli a ogni livello, addirittura intervenendo nelle fasi in cui un evento viene sottoscritto o sollevato.

Nel prossimo capitolo continueremo a esplorare le peculiarità di C# e di .NET, occupandoci di un aspetto cruciale nello sviluppo di un'applicazione come la gestione degli errori, per poi arrivare a trattare una tecnologia, chiamata Reflection, con cui è possibile ispezionare a runtime i membri degli oggetti e invocarne dinamicamente l'esecuzione.

Approfondimenti al linguaggio

Grazie a quanto abbiamo illustrato negli ultimi capitoli, la nostra visione del linguaggio C# inizia ormai a diventare abbastanza completa, tanto da consentirci di scrivere le prime applicazioni. Prima di questo passo, però, c'è un ultimo aspetto infrastrutturale del framework che è indispensabile conoscere e dal quale qualsiasi programmatore non può prescindere: gli errori a runtime, che sono l'argomento al quale è dedicata la prima parte di questo capitolo. In questa parte della guida impareremo quali sono le azioni che possiamo intraprendere nel momento in cui abbiamo la necessità di gestire delle situazioni non previste e anche le modalità secondo le quali possiamo usare le eccezioni per segnalare situazioni anomale a chi ha invocato un nostro metodo.

La seconda parte del capitolo sarà invece dedicata a concetti avanzati, in particolare a reflection, ossia un set di classi di cui possiamo avvalerci sia per esplorare i metadati contenuti negli assembly sia per istanziare e interagire dinamicamente con i membri in essi definiti. Vedremo inoltre quali sono gli strumenti forniti da .NET tramite i quali realizzare oggetti in grado di mutare la loro struttura in maniera dinamica e come invece sfruttare i custom attribute per scrivere codice dichiarativo.

Infine, daremo un brevissimo cenno su Roslyn, la libreria di Microsoft che permette di accedere alle funzionalità del compilatore direttamente da .NET e vedremo alcune novità dell'ultima versione del linguaggio.

Gestione delle eccezioni

Il fatto che l'esecuzione di un determinato programma possa generare un errore è un'eventualità che va presa in considerazione nel momento in cui si decide di sviluppare un'applicazione; il codice, potenzialmente, fallisce a prescindere dalla bravura di chi l'ha scritto, semplicemente perché le ragioni per cui ciò avviene non sono sempre prevedibili o gestibili: un metodo può sollevare un errore a causa di una svista dello sviluppatore, ma anche in risposta a un problema hardware o al tentativo di accedere a una risorsa non disponibile.

Ecco perché qualsiasi linguaggio e tecnologia di sviluppo sono dotati di sistemi per gestire gli errori. .NET, in particolare, chiama questi eventi imprevisti con il nome di **eccezioni**, ed essendo completamente orientato agli oggetti, sfrutta proprio il paradigma della programmazione a oggetti per realizzare un'infrastruttura di gestione affidabile e facilmente espandibile. Per apprezzarne appieno le caratteristiche, però, è necessario comprendere cosa accade in COM e Win32.

Gli errori prima del .NET Framework

Quando un metodo genera un errore, la prima azione che tipicamente è necessario intraprendere è quella di interrompere l'esecuzione del codice, cercare di identificarne la natura ed eventualmente mettere in atto delle contromisure, per evitare che l'applicazione si comporti in maniera imprevista; nel caso in cui ciò non sia possibile, l'errore deve essere notificato al chiamante del metodo che, a sua volta, deve provare a ripercorrere i medesimi passi.

In ambito COM e Win32, in assenza di un'infrastruttura dedicata alla gestione di questo tipo di situazioni, solitamente ci si limita a esporre metodi che restituiscano un valore intero mediante il quale indicare se l'esecuzione sia andata a buon fine o meno. Le API di Windows, per citare un caso, non appartengono alla categoria del codice managed e continuano, pertanto, a seguire questo sistema, come possiamo notare, per esempio, dalla signature funzione `GetDateFormat`:

```
int GetDateFormat(  
    __in    LCID Locale,  
    __in    DWORD dwFlags,  
    __in    const SYSTEMTIME *lpDate,  
    __in    LPCTSTR lpFormat,  
    __out   LPTSTR lpDateStr,
```

```
__in    int cchDate  
);
```

Quest'ultima restituisce un valore intero che, nel caso sia pari a 0, indica che si è verificato un errore durante la sua esecuzione. In generale, poi, una volta appurato che l'esecuzione non si è conclusa con successo, è necessario invocare ulteriori funzioni di sistema per recuperare informazioni circa la natura dell'errore stesso.

Si tratta, insomma, di una gestione non proprio elementare e assolutamente non standardizzata, che tra l'altro ha il grave difetto di demandare totalmente all'utilizzatore il compito di determinare il successo o meno dell'invocazione. Quest'ultimo aspetto è particolarmente pericoloso, in quanto dimenticarsi di verificare il valore di ritorno o controllarlo in maniera non corretta equivale in genere a perdere la notifica di un eventuale errore, generando quindi pericolosi bug di sicurezza e minando, in generale, la stabilità e l'affidabilità dell'applicazione stessa.

D'altro canto, una tecnologia di sviluppo come Visual Basic 6.0, per fare un altro esempio, ha utilizzato un approccio differente, che si basava sull'utilizzo di una clausola `On Error`, tramite la quale, in caso di errore, era possibile rimandare a una porzione di codice. Le informazioni sull'errore erano disponibili all'interno di un particolare oggetto, chiamato `Err`, che veniva automaticamente valorizzato dal runtime quando necessario e comunque codificate numericamente.

Si tratta, in buona sostanza, di due modalità decisamente differenti, ognuna con le proprie peculiarità e i propri difetti, ma comunque non standardizzate. In .NET, invece, la gestione delle eccezioni diviene infrastrutturale e comune a tutti i linguaggi, consentendoci di liberarci dal concetto di "errore identificato da un codice numerico" e risolvendo praticamente tutte le problematiche evidenziate in Win32. Cerchiamo di capire come.

[Gestione strutturata degli errori tramite le exception](#)

Per iniziare a comprendere quali siano i vantaggi di una gestione strutturata degli errori, come quella che possiede .NET, vediamo cosa accade nel codice dell'[esempio 7.1](#).

Esempio 7.1

```
public static void Main()
{
    var result = Division(5, 0);

    // Questo codice non viene mai eseguito
    Console.WriteLine("Il risultato è {0}", result);
}

public static int Division(int a, int b)
{
    int result = a / b;

    // in caso di errore questo codice non viene mai eseguito
    Console.WriteLine("Risultato calcolato con successo");

    return result;
}
```

Quando viene eseguito il calcolo all'interno della funzione `Division` utilizzando il valore 0 come quoziente, la sua esecuzione viene immediatamente interrotta e, al contesto di runtime, viene associato un particolare oggetto di tipo `DivideByZeroException`, rappresentativo della tipologia di errore che si è verificata. Successivamente, lo stesso comportamento si ripete all'interno del metodo `Main`, che viene anch'esso terminato, provocando quindi la chiusura dell'applicazione stessa, come mostrato nella [Figura 7.1](#).

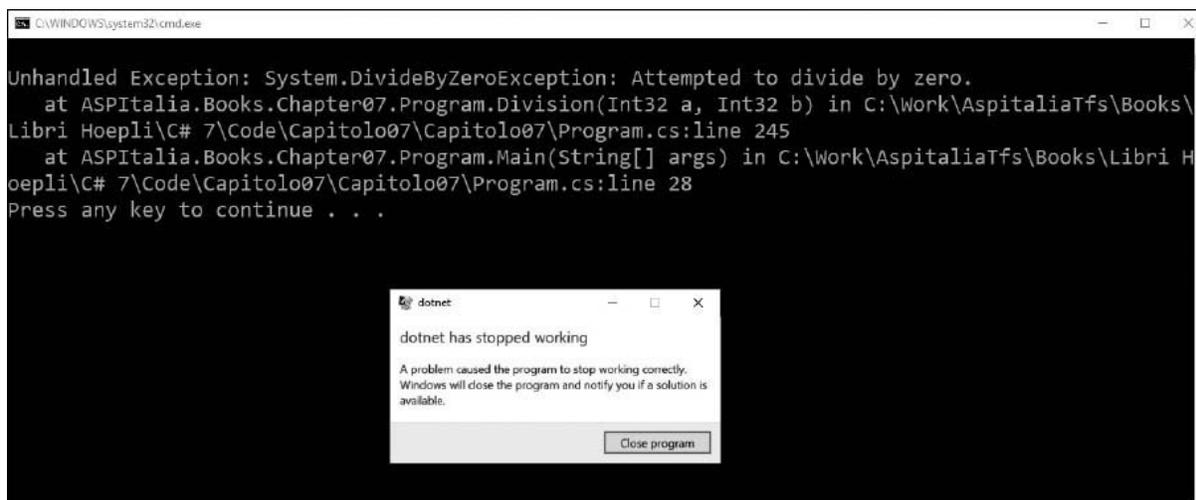


Figura 7.1 – Applicazione terminata per eccezione non gestita.

Il comportamento generale della gestione eccezioni in .NET è schematizzato nella [Figura 7.2](#).

In generale, ciò che accade quando un metodo solleva un'eccezione è che il runtime interrompe il flusso logico in esecuzione, per passare alla ricerca di un eventuale **gestore**, ossia di un blocco di codice opportunamente marcato come in grado di prenderla in carico. Nel caso in cui questo non venga trovato, l'eccezione viene propagata al metodo chiamante dove si ripete il medesimo pattern, via via risalendo tutto lo stack. Se tale ricerca si conclude con successo, il gestore viene eseguito e questo flusso di notifica viene interrotto, a meno che esso non decida di risollevarla a sua volta.

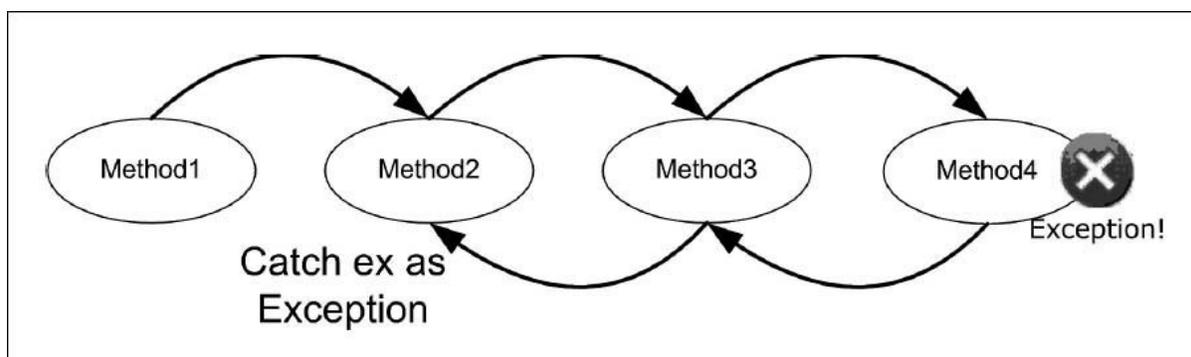


Figura 7.2 – Flusso di una exception.

In caso contrario, invece, una volta giunti all'origine dello stack, l'eccezione viene marcata come **non gestita** e provoca la chiusura dell'applicazione, come nel caso dell'[esempio 7.1](#).

Come possiamo notare, si tratta di un approccio strutturato molto differente se paragonato alla tecnologia Win32 che abbiamo illustrato nel paragrafo precedente. In .NET i valori di ritorno delle funzioni adempiono al ruolo per cui sono pensati, ossia quello di rappresentare il risultato logico dell'operazione, mentre la gestione di eventuali errori è completamente a carico del runtime che provvede da solo a interrompere l'esecuzione del codice e a farsi carico di propagarli lungo lo stack.

Abbiamo più volte accennato al fatto che le eccezioni in .NET sono rappresentate come oggetti. Nel prossimo paragrafo cercheremo di capire meglio le caratteristiche di questo modello.

[La classe System.Exception](#)

.NET è una tecnologia completamente orientata verso la programmazione a oggetti e sfrutta questo paradigma anche per rappresentare gli errori che vengono sollevati a runtime. Abbiamo accennato al fatto che, al verificarsi di un'eccezione, il CLR associa al contesto di esecuzione un'istanza di un particolare oggetto il cui tipo è (o deriva da) `Exception`.

Proprio il tipo utilizzato è la prima discriminante che possiamo usare per determinare la natura dell'eccezione che si è verificata, come abbiamo potuto verificare nell'[esempio 7.1](#): una divisione per zero solleva infatti una `DivisionByZeroException`, mentre invece, accedere in lettura a un file inesistente, genera un errore rappresentato dal tipo `FileNotFoundException`. La classe `Exception`, in particolare, è capostipite di una numerosa gerarchia di tipi, ognuno dei quali è associato a una particolare casistica di errore. Essa espone alcuni membri che, pertanto, sono comuni a tutte le eccezioni definite nel framework, e sono sintetizzati nella [Tabella 7.1](#)

Tabella 7.1 – Membri della classe `System.Exception`

Nome	Significato
<code>Message</code>	Contiene un messaggio di errore descrittivo dell'eccezione.
<code>Source</code>	Se non impostato diversamente, contiene il nome dell'assembly che ha sollevato l'eccezione.
<code>StackTrace</code>	Lo stack di chiamate al momento in cui si verifica l'eccezione.
<code>Data</code>	Una dictionary che consente di specificare ulteriori informazioni sull'errore verificatosi.
<code>HelpLink</code>	È una proprietà in cui è possibile memorizzare un URL per rimandare a una pagina di help.
<code>InnerException</code>	Un oggetto <code>Exception</code> può fungere da contenitore per un'ulteriore eccezione più specifica, che viene esposta tramite questa proprietà. Questo concetto sarà spiegato meglio nel corso del capitolo.
<code>TargetSite</code>	La definizione del metodo che ha sollevato l'eccezione.
<code>ToString()</code>	Questo metodo è ridefinito nella classe <code>System.Exception</code> e produce una stringa contenente la natura dell'eccezione, il messaggio e lo stack trace.

A questi membri se ne possono aggiungere altri, propri della particolare tipologia d'eccezione che è stata sollevata e dell'errore che essa deve rappresentare: la classe `SQLException`, per esempio, viene sollevata in risposta a un errore segnalato da SQL Server e contiene, fra le altre, le proprietà

LineNumber o Procedure, che indicano rispettivamente la riga di T-SQL e la stored procedure che hanno causato il problema.

Realizzare eccezioni personalizzate

Essendo le eccezioni nient'altro che oggetti, nulla vieta di costruire delle proprie eccezioni personalizzate per modellare errori applicativi particolari, che non trovano riscontro tra quelli già previsti nel framework. La gerarchia di ereditarietà che ha origine dalla classe `System.Exception` trova, al secondo livello, due classi che, almeno nelle intenzioni, hanno lo scopo di suddividere le eccezioni in altrettante categorie:

- ❑ `SystemException` rappresenta la classe base per tutte le eccezioni sollevate dal runtime.
- ❑ `ApplicationException` è invece da intendersi come la classe base per tutte le eccezioni applicative e, pertanto, è consigliabile utilizzarla come tale per tutte le eccezioni personalizzate che abbiamo bisogno di realizzare.

Per realizzare una exception personalizzata, quindi, non dobbiamo far altro che ereditare da quest'ultima classe base per creare una classe il cui nome, per convenzione, dovrebbe terminare con il suffisso `-Exception`.

Lo scopo di una tale suddivisione è quello di dare la possibilità allo sviluppatore di distinguere facilmente la natura di un'eccezione, in modo che, per esempio, si possano gestire in maniera centralizzata tutte quelle generate dal runtime e, in maniera puntuale, quelle applicative. In realtà, nello sviluppo .NET questa regola non è stata seguita fino in fondo e oggi esistono eccezioni di runtime che derivano direttamente da `Exception` o addirittura da `ApplicationException`. Ciò nonostante, è comunque consigliabile seguire la linea guida generale ed ereditare quindi da quest'ultima.

L'[esempio 7.2](#) mostra il codice necessario a definire un'eccezione personalizzata per rappresentare un errore di mancata validazione di un oggetto `Customer`.

Esempio 7.2

```
[Serializable]
public class InvalidCustomerException : ApplicationException
{
    public Customer Customer { get; set; }

    public InvalidCustomerException(Customer customer)
        : this(customer, "Customer is invalid")
    { }

    public InvalidCustomerException(Customer customer, string message)
        : base(message)
    {
        this.Customer = customer;
    }

    public InvalidCustomerException(Customer customer, string message, Exception
innerException)
        : base(message, innerException)
    {
        this.Customer = customer;
    }

    public InvalidCustomerException(SerializationInfo info, StreamingContext
context)
        : base(info, context)
    { }
}
```

Sebbene realizzare una classe che derivi da `Exception` equivalga a costruire un'eccezione personalizzata perfettamente valida, il codice mostrato si spinge un po' oltre per adottare alcune best practice consigliate nella realizzazione di queste particolari tipologie di oggetti. In particolare:

- ❑ L'eccezione deve essere marcata con l'attributo `Serializable`. Il concetto di attributo sarà presentato più avanti in questo capitolo; per il momento ci basti sapere che quella particolare notazione consente a .NET di rappresentare l'eccezione con uno stream binario, in modo che possa essere eventualmente inviata attraverso la rete o persistita su una memoria durevole.
- ❑ La custom exception può, a tutti gli effetti, contenere delle proprietà aggiuntive, nel nostro caso la particolare istanza di `Customer` che non è stata ritenuta valida.
- ❑ Devono essere presenti due costruttori che consentano di specificare un messaggio personalizzato e una `InnerException`, ossia un'eccezione di cui si vuole tener traccia (solitamente perché è stata la causa prima

dell'errore) e che si vuole memorizzare all'interno della nostra classe personalizzata.

- ❑ Deve essere presente un costruttore che accetti i due tipi `SerializationInfo` e `StreamingContext`; si tratta di un costruttore “di servizio”, che viene utilizzato nel momento in cui si vuole ricostruire una eccezione da dati serializzati.

Ora che abbiamo appreso nel dettaglio il modello a oggetti che in .NET è utilizzato per rappresentare gli errori a runtime, è finalmente arrivato il momento di capire come, all'interno del codice, sia possibile interagire con essi, sia per quanto riguarda la gestione di eventuali eccezioni, sia nell'ottica di sollevarle per notificare altri metodi di eventuali situazioni anomale. Le prossime pagine saranno dedicate proprio a questo argomento.

Lavorare con le eccezioni nel codice

La capacità di intercettare le eccezioni all'interno del codice è interamente basata sul tipo di `Exception` che viene sollevata che, come abbiamo visto in precedenza, rappresenta la natura dell'errore che si è verificato.

Intercettare e gestire situazioni di errore significa scrivere particolari blocchi di codice che vengono eseguiti in queste situazioni specifiche. Il primo fra quelli che analizzeremo è il costrutto `try...catch`, in grado di attivarsi al verificarsi di un particolare tipo di eccezione.

Intercettare le eccezioni

Il codice visto nell'[esempio 7.1](#) sollevava un errore a causa della divisione per zero e, trattandosi di un'eccezione non gestita, portava al crash e alla chiusura dell'applicazione stessa. Per evitare queste conseguenze, lo possiamo modificare come mostrato nell'[esempio 7.3](#).

Esempio 7.3

```
public static void Main()
{
    try
    {
        var result = Division(5, 0);
    }
}
```

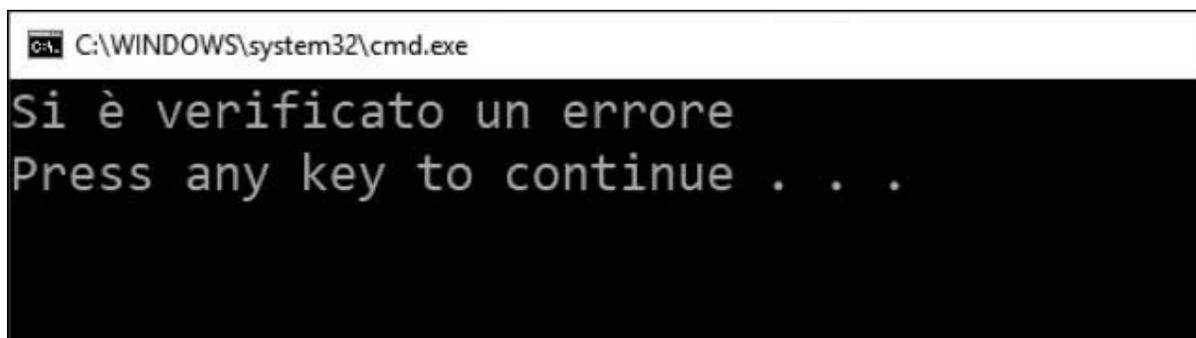
```
// Questo codice non viene mai eseguito
Console.WriteLine($"Il risultato è {result}");
}
catch (Exception ex)
{
    Console.WriteLine("Si è verificato un errore");
}
}
```

Tutte le volte in cui abbiamo a che fare con del codice che potenzialmente può sollevare un'eccezione e vogliamo essere in grado di gestire questa eventualità, possiamo avvalerci del costrutto `try...catch` appena mostrato, grazie al quale, quando il codice all'interno del blocco `try` genera un'eccezione, ne viene interrotta l'esecuzione e il controllo viene passato al blocco `catch` che corrisponde al tipo dell'eccezione sollevata.

Nel nostro esempio, in particolare, alla clausola `catch` è associato il tipo `Exception` ed essendo quest'ultimo il tipo base per qualsiasi eccezione del framework, è considerato un gestore valido per la `DivideByZeroException` sollevata dallo statement

```
var result = Division(5, 0);
```

Il risultato finale, quindi, è che l'eccezione risulta gestita e l'applicazione non termina più in maniera anomala come accadeva in precedenza, come si può notare dall'output mostrato dalla [Figura 7.3](#).



```
C:\WINDOWS\system32\cmd.exe
Si è verificato un errore
Press any key to continue . . .
```

Figura 7.3 – Eccezione gestita.

Il risultato sarebbe il medesimo se si indicasse come eccezione da gestire la stessa `DivideByZeroException` o `ArithmeticException`, da cui essa deriva, mentre l'applicazione tornerebbe a chiudersi con un errore se il gestore fosse relativo a un tipo differente come, per esempio, `OutOfMemoryException`.

In generale, un blocco `try...catch` può contenere molteplici clausole `catch`, in modo da prevedere diversi gestori per specifiche tipologie di eccezione, come mostra l'[esempio 7.4](#).

Esempio 7.4

```
try
{
    var result = Division(5, 0);

    // Questo codice non viene mai eseguito
    Console.WriteLine("Il risultato è {0}", result);
}
catch (DivideByZeroException ex)
{
    Console.WriteLine("Errore: non si può dividere per zero");
}
catch (OutOfMemoryException ex)
{
    Console.WriteLine("Memoria terminata");
}
catch (Exception ex)
{
    Console.WriteLine("Si è verificato un errore generico");
}
```

Quando si utilizza questo particolare approccio, bisogna prestare attenzione all'ordine secondo cui vengono disposti i blocchi `catch`, dato che vengono valutati in sequenza dal runtime finché non ne viene trovato uno adatto; pertanto, se nel codice dell'[esempio 7.4](#) il blocco

```
catch (Exception ex)
```

si trovasse in prima posizione, essendo in grado di gestire qualsiasi tipo di eccezione, impedirebbe di fatto l'esecuzione di quelli più specifici.

Un altro concetto piuttosto importante nella gestione delle eccezioni, è quello degli `exception filter`, ossia la possibilità di attivare un determinato blocco `catch` in base a una condizione specifica.

Esempio 7.5

```
try
{
    ExecuteSqlQuery();
}
```

```
catch (SQLException e) when (e.Class > 19)
{
    Console.WriteLine("Errore fatale");
}
catch (SQLException e)
{
    Console.WriteLine("Errore durante la query");
}
```

Il codice dell'[esempio 7.5](#) intercetta un'eccezione di tipo `SQLException` in due differenti blocchi `catch`. Il primo, in particolare, grazie alla keyword `when`, viene attivato solo quando il livello di severity dell'errore è superiore a 19, ossia in presenza di un errore fatale, mentre tutti gli altri casi saranno gestiti dal secondo blocco `catch`. Anche in questo caso è importante l'ordine dei blocchi `catch`, poiché la validazione degli statement `where` verrà eseguita in sequenza ed è quindi bene tenere, nuovamente, i blocchi più generici in fondo.

[Il blocco finally](#)

Alle volte la necessità che abbiamo non è quella di intercettare una particolare tipologia d'eccezione, ma di assicurarci che una determinata porzione di codice venga eseguita comunque, sia in presenza che in assenza di errore. Tipicamente si tratta di codice detto **di cleanup**, ossia utilizzato per liberare risorse. Consideriamo il caso dell'[esempio 7.6](#), che utilizza un oggetto di tipo `StreamReader` per leggere il contenuto di un file.

Esempio 7.6

```
StreamReader sr = null;
try
{
    sr = new StreamReader(@"c:\test.txt");
    string content = sr.ReadToEnd();
}
finally
{
    if (sr != null)
        sr.Close();
}
```

In casi simili abbiamo bisogno di essere certi che il metodo `close` venga comunque invocato, in quanto necessario per liberare l'handle di Windows associato al file a cui si sta accedendo; il blocco `finally` garantisce l'esecuzione del codice in esso contenuto, a prescindere dal fatto che le

istruzioni all'interno del corrispondente blocco `try` sollevino eccezioni o meno.

Le parole chiave `try`, `catch` e `finally` possono essere utilizzate anche contemporaneamente, come mostra l'[esempio 7.7](#).

Esempio 7.7

```
StreamReader sr = null;
try
{
    sr = new StreamReader(@"c:\test.txt");
    string content = sr.ReadToEnd();
}
catch (FileNotFoundException ex)
{
    Console.WriteLine("Errore di I/O:\r\n{0}", ex);
}
finally
{
    if (sr != null)
        sr.Close();
}
```

Solitamente gli oggetti che, come `StreamReader`, allocano risorse di sistema, implementano anche una particolare interfaccia per consentirne il rilascio in maniera deterministica, per esempio, all'interno di un blocco `finally`. Si tratta di un argomento di estrema importanza, dato che una gestione errata delle risorse spesso si accompagna a leak, blocchi dell'applicazione e a sovraccarichi del sistema, ai quali è interamente dedicato il paragrafo successivo.

[L'interfaccia `IDisposable` e il blocco `using`](#)

Quando, nel corso del primo capitolo, abbiamo parlato della gestione della memoria da parte del CLR, abbiamo visto come il processo di deallocare oggetti non più utilizzati avvenga in maniera del tutto trasparente allo sviluppatore, grazie a un gestore chiamato **Garbage Collector** e, in particolare, abbiamo visto come questo operi in maniera non deterministica, in base alle richieste di risorse da parte dell'applicazione e alle condizioni del sistema.

Esistono però casi in cui un comportamento del genere non è accettabile: si pensi al precedente [esempio 7.6](#), nel quale sfruttiamo uno `StreamReader` per accedere a un file, impegnando quindi una risorsa non gestita quale un handle

di Windows. Quando questo oggetto viene coinvolto in una garbage collection, ne viene invocato il distruttore, in modo che tali risorse vengano effettivamente liberate.

Si tratta però di una sorta di “ultima spiaggia”, in quanto non è assolutamente accettabile che lasciamo il file impegnato dall’applicazione, anche quando abbiamo effettivamente terminato di utilizzarlo, attendendo l’intervento del Garbage Collector, sulla cui cadenza temporale non abbiamo alcun controllo. In tutti i casi in cui vogliamo invece liberare una risorsa in maniera deterministica, possiamo utilizzare il **pattern Disposable**, che consiste nell’implementare l’interfaccia `IDisposable` come evidenziato nell’[esempio 7.8](#).

Esempio 7.8

```
public class DisposableObject : IDisposable
{
    protected virtual void Dispose(bool disposing)
    {
        if (disposing)
        {
            // qui si liberano le risorse gestite
        }

        // qui si liberano le risorse non gestite
    }

    public void Dispose()
    {
        this.Dispose(true);
        GC.SuppressFinalize(this);
    }

    ~DisposableObject()
    {
        this.Dispose(false);
    }
}
```

Cerchiamo di capire in dettaglio il significato e il funzionamento di questi tre metodi:

- ❑ Il primo metodo `Dispose` accetta un parametro in ingresso che indica se è stato invocato deterministicamente dall’utente (valore `true`) o dal finalizzatore (valore `false`) a causa di una garbage collection. Nel primo caso, devono essere liberate tutte le risorse, sia gestite che non, nel secondo caso solo queste ultime, in quanto il garbage collector potrebbe aver già distrutto le prime;

- ❑ Il secondo overload di `Dispose`, senza parametri, è quello previsto dall'interfaccia `IDisposable` e, dopo aver opportunamente invocato il metodo descritto in precedenza, comunica al Garbage Collector che non è più necessario eseguire il finalizzatore per quella particolare istanza, visto che le risorse sono state già liberate.
- ❑ Il terzo metodo è il distruttore dell'oggetto e si occupa di invocare il metodo `Dispose` forzando la liberazione delle sole risorse non gestite.

L'implementazione dell'interfaccia `IDisposable`, come si può capire da quanto abbiamo detto finora, consente di liberare in maniera deterministica, invocandone il metodo `Dispose`, eventuali risorse non gestite, utilizzate direttamente o indirettamente. Essa va pertanto implementata, assieme al finalizzatore, quando utilizziamo direttamente risorse non gestite (per esempio un `handle` o un oggetto `COM`) oppure nei casi in cui il nostro oggetto mantenga riferimenti ad altri oggetti che implementano a loro volta `IDisposable`. In quest'ultimo caso, però, non è necessario dotare la nostra classe di un distruttore. In ogni modo, è importante sottolineare che `IDisposable` non ha nulla a che vedere con il concetto di liberare memoria, che resta invece totalmente una prerogativa del Garbage Collector.

Quando istanziamo un oggetto `IDisposable` nel codice di un metodo, dovremmo sempre utilizzarlo all'interno di un blocco `try...finally` per essere sicuri di invocare correttamente il metodo `Dispose` anche in eventuali casi di errore, come mostrato dall'[esempio 7.9](#).

Esempio 7.9

```
public static void Main()
{
    var myObject = new DisposableObject();

    try
    {
        myObject.SomeMethod();
    }
    finally
    {
        myObject.Dispose();
    }
}
```

C# mette a disposizione una sintassi particolare, di fatto equivalente a quella precedente, utilizzando il blocco `using` come nell'[esempio 7.10](#).

Esempio 7.10

```
using (var myObject = new DisposableObject())
{
    myObject.SomeMethod();
}
```

Esso è riconosciuto come valido dal compilatore solo nel caso in cui l'oggetto istanziato nel primo statement implementi l'interfaccia `IDisposable` e abbia la caratteristica di invocare automaticamente il metodo `Dispose` al termine del blocco, sia in presenza, sia in assenza, di eventuali eccezioni a runtime. Novità introdotta da C# 8 è la **using declaration**, ovvero la possibilità di fare assegnazioni in linea con la keyword `using`:

Esempio 7.11

```
private void MyMethod()
{
    using var myObject = new DisposableObject();
    myObject.SomeMethod();
} // Il metodo Dispose viene chiamato qui
```

In questo caso non è necessario creare un blocco `using`, poiché il metodo `Dispose` verrà chiamato automaticamente alla chiusura dello scope in cui è contenuto: come mostrato dall'[esempio 7.11](#) questo avviene al completamento del metodo `MyMethod`.

Dopo questo breve excursus su come gestire oggetti che allocano risorse in .NET anche in caso di eccezioni, dal prossimo paragrafo torneremo all'argomento principale della prima parte di questo capitolo, ossia la gestione degli errori a runtime, con particolare attenzione alle best practice consigliate per l'utilizzo di questo strumento all'interno del nostro codice.

[Sollevare eccezioni nel codice e best practice](#)

Ovviamente, durante l'esecuzione di un'applicazione, non è detto che ci troviamo sempre dalla parte di coloro che "subiscono" il verificarsi di un errore, bensì può capitare che sia necessario e doveroso sollevare un'eccezione per rispondere a una situazione anomala. A questo scopo possiamo utilizzare la parola chiave `throw`, che accetta come argomento un'istanza di un oggetto che derivi dalla classe `System.Exception`. L'[esempio 7.12](#) mostra un tipico utilizzo di questa parola chiave per rispondere a una delle linee guida fornite da Microsoft in merito al design delle nostre classi, ossia il verificare preventivamente, all'interno di ogni metodo, che i parametri obbligatori siano correttamente valorizzati, sollevando una `ArgumentNullException` in caso contrario.

Esempio 7.12

```
public static void SomeMethod(List<int> items)
{
    if (items == null)
    {
        // Si solleva l'eccezione indicando il nome dell'argomento non
        // correttamente valorizzato
        throw new ArgumentNullException("items");
    }

    // Codice del metodo qui
}
```

Un possibile problema dell'esempio precedente è costituito dall'utilizzo di una stringa per indicare il nome del parametro errato `items`. Il codice funziona perfettamente, ma non supporta le funzionalità di refactoring esposte da Visual Studio. Per esempio, se rinominiamo il parametro, l'ambiente di sviluppo non è in grado di individuare questa stringa e aggiornarla.

L'alternativa più corretta è quella di utilizzare la parola chiave `nameof`, che accetta un qualsiasi membro e restituisce una stringa con il suo nome. L'[esempio 7.13](#) mostra la differenza:

Esempio 7.13

```
public static void SomeMethod(List<int> items)
{
    if (items == null)
    {
        // Si solleva l'eccezione indicando il nome dell'argomento non
        // correttamente valorizzato
    }
}
```

```
        throw new ArgumentNullException(nameof(items));
    }

    // Codice del metodo qui
}
```

L'istruzione `throw` non va utilizzata solo nel caso in cui abbiamo la necessità di generare una nuova eccezione ma anche quando, dopo averne intercettata una tramite un blocco `catch`, vogliamo **rilanciarla** per notificare comunque il chiamante.

Gestione e rilancio delle eccezioni

Quando un'eccezione viene sollevata e intercettata all'interno di un blocco `catch`, a meno che non sia possibile porvi rimedio e aggirare il problema, solitamente le due azioni più opportune da intraprendere sono:

- ❑ eseguire del codice per **gestire almeno parzialmente l'errore**, per esempio, limitando i danni che una stored procedure possa aver provocato eseguendo un rollback di una transazione, oppure registrando l'eccezione ottenuta in un file di log, così che possa essere investigata in futuro;
- ❑ **rilanciare l'eccezione al chiamante**, per notificarlo dell'anomalia, e così via fino a raggiungere la root dell'applicazione, dove un gestore centralizzato possa valutare la gravità della situazione, mostrare un messaggio d'errore o, eventualmente, terminare l'esecuzione.

Il concetto di rilanciare un'eccezione non completamente gestita, quindi, è di fondamentale importanza all'interno del blocco `catch`, per far sì che gli altri chiamanti nello stack siano anch'essi notificati, e si esplicita utilizzando la parola chiave `throw`, come nell'[esempio 7.14](#).

Esempio 7.14

```
try
{
    var myObject = new SomeClass();
    myObject.SomeProblematicMethod();
}
catch (Exception ex)
{
    logger.Log(ex.ToString);
}
```

```
    throw;
}
```

Quando l'istruzione `throw` viene utilizzata senza specificare alcun argomento, l'eccezione sollevata è quella dell'attuale contesto di runtime; quello dell'[esempio 7.14](#) è sempre il modo più corretto per rilanciare un'eccezione, in quanto consente di preservarne lo stack trace originale, grazie al quale è possibile avere evidenza del metodo che, in ultima analisi, ha generato il problema. Questa informazione verrebbe invece persa utilizzando la sintassi

```
catch (Exception ex)
{
    // ...
    throw ex
}
```

Utilizzo delle InnerException

In alcune situazioni, invece, può essere necessario inglobare l'eccezione ottenuta in una di più alto livello. Immaginiamo, per esempio, che l'esecuzione di una query SQL per validare le credenziali dell'utente fallisca, generando una `SQLException`. Visto che, in ultima analisi, ciò comporta anche la mancata validazione delle credenziali utente, rilanciare un'eccezione di tipo `SecurityException` può rappresentare forse la decisione più corretta, facendo però in modo che quest'ultima contenga anche le informazioni relative alla vera natura dell'errore che si è verificato.

Esempio 7.15

```
public bool ValidateCredentials(string username, string password)
{
    try
    {
        // codice per validare username e password su database
    }
    catch (SQLException ex)
    {
        throw new SecurityException(
            "Non è stato possibile validare le credenziali", ex);
    }
}
```

Il codice dell'[esempio 7.15](#), come possiamo notare, rilancia una `SecurityException` fornendo come argomento del costruttore l'istanza dell'eccezione originale, in modo che sia accessibile tramite la proprietà `InnerException`.

Considerazioni a livello di prestazioni sull'uso delle Exception

Un'ultima riflessione deve essere invece fatta dal punto di vista delle performance. La gestione delle eccezioni è, infatti, **molto onerosa** per il runtime e sovraccarica molto la CPU, rendendo estremamente lenta la nostra applicazione. Per questa ragione, esse vanno utilizzate per gestire eventi, per l'appunto, eccezionali. Pertanto non è corretto, per esempio, segnalare con una `SecurityException` la mancata validazione delle credenziali utente quando username e password semplicemente non corrispondono: si tratta di un caso prevedibile e, per gestirlo, è molto più opportuno utilizzare una funzione che restituisca un valore `bool`, come quella dell'esempio precedente, mentre l'eccezione deve essere utilizzata solo per un evento non prevedibile a priori, come un errore di connessione al database o di esecuzione della query.

Con queste considerazioni si conclude la prima parte del capitolo, dedicata alla gestione delle eccezioni a runtime nelle applicazioni basate su .NET. Nelle prossime pagine introdurremo invece un'altra peculiarità di questa tecnologia, grazie alla quale è possibile ispezionare e invocare dinamicamente i membri dei nostri oggetti: reflection.

Esplorare i tipi a runtime con Reflection

Nel corso del primo capitolo, abbiamo visto come il risultato della compilazione di codice sorgente nel framework sia un **assembly**, ossia un contenitore di alto livello, in grado di immagazzinare non solo codice MSIL ma anche informazioni aggiuntive relativamente al suo contenuto, catalogate tramite i metadati; il meccanismo grazie al quale è possibile interrogare questi metadati ed eventualmente utilizzare nel codice le informazioni in essi contenute, è denominato **Reflection** ed è implementato mediante una serie di classi, appartenenti al namespace `System.Reflection`. Grazie a queste ultime, per esempio, possiamo esplorare a runtime il contenuto di un assembly e recuperare l'elenco dei tipi in esso definiti.

Esempio 7.16

```
// Recupera una reference all'assembly corrente
Assembly currentAssembly = Assembly.GetExecutingAssembly();

// Ricerca dei tipi definiti nell'assembly in esecuzione
foreach (Type type in currentAssembly.GetTypes())
{
    // Stampa il nome del tipo su console
    Console.WriteLine(type.Name);
}
```

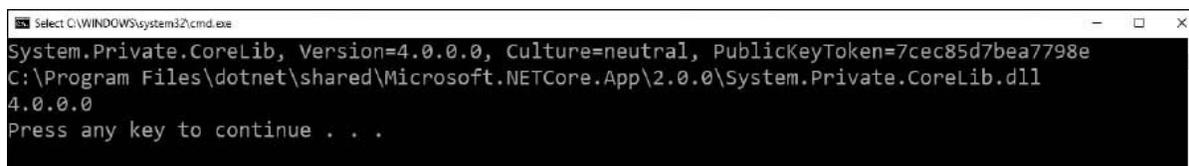
La classe `Assembly`, utilizzata nell'[esempio 7.16](#), espone una serie di metodi statici per ottenere a runtime l'accesso a un particolare assembly. Nel nostro caso abbiamo utilizzato il metodo `GetExecutingAssembly`, che ritorna un riferimento a quello correntemente in esecuzione, ma è possibile caricare una libreria in memoria a partire da un tipo in essa definito, dal nome logico, dal nome del file che la contiene, o addirittura da uno stream di byte che ne rappresenta il contenuto. Per esempio, tramite il tipo `int`, possiamo recuperare un riferimento a `mscorlib`, cui esso appartiene.

Esempio 7.17

```
Assembly MsCoreLib = Assembly.GetAssembly(typeof(int));

Console.WriteLine(MsCoreLib.FullName);
Console.WriteLine(MsCoreLib.Location);
Console.WriteLine(MsCoreLib.GetName().Version);
```

Come possiamo notare nell'[esempio 7.17](#), un'istanza di `Assembly` può essere utilizzata per molteplici scopi, per esempio, per reperire informazioni quali il fully qualified name, il path fisico o il numero di versione. La [Figura 7.4](#) mostra il risultato dell'esecuzione del codice precedente.



```
Select C:\WINDOWS\system32\cmd.exe
System.Private.CoreLib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=7cec85d7bea7798e
C:\Program Files\dotnet\shared\Microsoft.NETCore.App\2.0.0\System.Private.CoreLib.dll
4.0.0.0
Press any key to continue . . .
```

Figura 7.4 – Recupero informazioni tramite la classe `Assembly`.

Torniamo ora, per un momento, all'[esempio 7.16](#), in cui siamo stati in grado di recuperare un elenco di tipi definiti in un assembly, tramite il metodo `GetTypes`.

```
foreach (Type type in currentAssembly.GetTypes())
{
    ..
}
```

Il risultato di questo metodo è un array di oggetti di tipo `System.Type`, una classe che, nonostante non appartenga al namespace `System.Reflection`, costituisce il cardine di questo set di API, come avremo modo di vedere nel prossimo paragrafo.

[La classe System.Type](#)

La classe `Type` rappresenta il modello a oggetti di un qualsiasi tipo del runtime e svolge una duplice funzione:

- ❑ espone contenuto informativo sulla natura del tipo stesso, permettendoci, per esempio, di capire se si tratta di un tipo di riferimento o di valore, se è astratto, se è un'interfaccia, ecc.;
- ❑ consente l'accesso a tutti i membri in esso definiti, come campi, proprietà, metodi ed eventi affinché, data una particolare istanza di un'oggetto, siamo poi in grado di invocarli dinamicamente.

La [Tabella 7.2](#) mostra i membri di questa classe più comunemente utilizzati.

Tabella 7.2 – Membri della classe System.Type

Nome	Significato
<code>Assembly</code>	Restituisce un riferimento all'assembly in cui il tipo è definito.
<code>Name</code> <code>FullName</code> <code>Namespace</code> <code>AssemblyQualifiedName</code>	Restituiscono rispettivamente il nome del tipo, il nome completo di namespace, il solo namespace e il nome completo di namespace e assembly di definizione.
<code>IsAbstract</code> <code>IsClass</code> <code>IsValueType</code>	Si tratta di una serie di proprietà tramite le quali possiamo identificare la natura del tipo in questione e il livello di

IsEnum IsInterface IsArray IsGenericType IsPublic IsSealed	accessibilità.
GetProperty() GetProperties() GetField() GetFields() GetMethod() GetMethods() GetConstructor() GetConstructors() GetEvent() GetEvents()	Restituiscono oggetti che derivano dalla classe MemberInfo, rappresentativi rispettivamente di proprietà, campi, metodi, costruttori ed eventi. È possibile recuperare tutti questi elementi o ricavarne uno in particolare in base al nome e, in seguito, utilizzare l'oggetto ottenuto per manipolare o creare un'istanza del tipo stesso.
FindInterfaces()	Restituisce un elenco delle interfacce implementate dal tipo.
IsInstanceOfType(o as Object)	Restituisce true se l'argomento è un'istanza valida per il tipo.
IsAssignableFrom(t as Type)	Restituisce true nel caso in cui un'istanza del tipo fornito come argomento sia assegnabile al tipo.

`System.Type` è una classe astratta, quindi non è possibile istanziarla direttamente; l'[esempio 7.18](#), tuttavia, mostra le diverse modalità grazie alle quali è possibile recuperarne un'istanza valida.

Esempio 7.18

```
Type integerType;

// Utilizzo dell'istruzione typeof
integerType = typeof(int);

// Utilizzo del metodo object.GetType() a partire
// da un'istanza dell'oggetto
int value = 5;
integerType = value.GetType();

// Utilizzo del metodo Type.GetType() a partire
// dal nome in formato stringa del tipo
integerType = Type.GetType("System.Int32");
```

L'ultima riga di codice, in particolare, è un piccolo esempio di come sia possibile sfruttare reflection per dotare le nostre applicazioni di codice dinamico: come vedremo nelle prossime pagine, a partire dal semplice nome di un tipo, sia esso inserito dall'utente in una finestra di dialogo o letto da un file di configurazione, siamo infatti in grado di recuperare la relativa definizione e, successivamente, interagire con esso.

Scrittura di codice dinamico

Quando si parla di codice dinamico, generalmente s'intende definire la possibilità di eseguire istruzioni che in fase di compilazione non siano ancora note: si tratta di una caratteristica che non appartiene a nessuno degli esempi che abbiamo avuto modo di analizzare fino a questo punto del libro, tutti caratterizzati dal fatto che il codice scritto in C# veniva compilato in MSIL e memorizzato all'interno di un assembly in forma statica. Nei prossimi paragrafi vedremo le differenti modalità messe a disposizione da C# per scrivere codice dinamico.

Realizzazione di codice dinamico con reflection

Abbiamo già visto, nel precedente [esempio 7.18](#), come possiamo utilizzare il metodo `Type.GetType()` per ottenere il riferimento a un particolare tipo, dato il suo nome in formato stringa. Supponiamo allora di aver realizzato una libreria di classi chiamata `ClassLibrary`, all'interno della quale sia presente l'oggetto `Person` dell'[esempio 7.19](#).

Esempio 7.19

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public override string ToString() =>
        $"{Name} ha {Age} anni";
}
```

Per utilizzare quest'oggetto da un'applicazione console, una volta arrivati a questo punto, è sufficiente usare i vari metodi presentati nella [Tabella 7.2](#) per costruirne, per esempio, un'istanza e interagire con i suoi membri. Il tutto funziona anche se la nostra applicazione non possiede una reference verso l'assembly in cui `Person` è definito, a patto che quest'ultimo sia comunque presente nella stessa cartella dell'applicazione principale o sia installato in GAC.

Esempio 7.20

```

// Recuperiamo un riferimento al tipo
Type personType = Type.GetType("ClassLibrary.Person, ClassLibrary");

// Tramite il costruttore di default (senza parametri)
// ne costruiamo un'istanza
ConstructorInfo constructor =
    personType.GetConstructor(Type.EmptyTypes);
object person = constructor.Invoke(null);

// Tramite PropertyInfo valorizziamo Name e Age
PropertyInfo nameProperty =
    personType.GetProperty("Name");
nameProperty.SetValue(person, "Matteo Tumiati", null);

PropertyInfo ageProperty =
    personType.GetProperty("Age");
ageProperty.SetValue(person, 28, null);

// Visualizziamo su Console il risultato di Person.ToString()
Console.WriteLine(person.ToString());

```

Come possiamo notare nel [esempio 7.20](#), a partire da un `Type` siamo in grado di recuperare oggetti quali `PropertyInfo`, `ConstructorInfo` (e, oltre a questi, anche i corrispettivi di metodi, campi ed eventi) semplicemente in base al loro nome e, di fatto, utilizzarli per costruire un'istanza di `Person`, valorizzarla e visualizzare l'output del metodo `ToString` sulla console, realizzando quindi algoritmi che non sarebbero possibili con codice statico.

In alternativa all'utilizzo esplicito di `ConstructorInfo`, nel caso di costruttori senza parametri può essere utilizzata la classe `Activator`, per ottenere un'istanza del tipo desiderato:

```

// Creazione di un'istanza di Person
Type personType =
    Type.GetType("ClassLibrary.Person, ClassLibrary");
object person = Activator.CreateInstance(personType);

```

Una tale versatilità si paga tuttavia in termini di performance, essendo l'utilizzo di reflection piuttosto oneroso e soprattutto perché il codice scritto in questo modo non è più fortemente tipizzato: assegnare un valore stringa alla proprietà `Age`, per esempio, risulta perfettamente lecito in fase di compilazione, dato che il metodo `PropertyInfo.SetValue` accetta un generico tipo `object`, ma provocherà un errore a runtime.

[Codice dinamico tramite la parola chiave `dynamic`](#)

Una delle caratteristiche in assoluto più apprezzate di `C#` è il fatto di essere un linguaggio fortemente tipizzato, in cui tutte le assegnazioni e le invocazioni

sono ritenute valide, facendo sì che assegnazioni e invocazioni di metodi possano essere risolte direttamente in fase di compilazione. Questo tipo di approccio è chiamato **early binding**.

In alcuni (rari) casi, soprattutto quando si fa uso di codice dinamico simile a quello visto in precedenza, può risultare comodo disattivare questa opzione, in modo che il compilatore dilazioni fino al runtime la risoluzione dei metodi e dei campi utilizzati, usando la tecnica chiamata **late binding**. In questo modo, quindi, il codice dell'esempio precedente può essere riscritto in una forma notevolmente più concisa.

Esempio 7.21

```
// Creazione di un'istanza di Person
Type personType =
    Type.GetType("ClassLibrary.Person, ClassLibrary");
dynamic person = Activator.CreateInstance(personType);

person.Name = "Matteo Tumiati";
person.Age = 28;

// Visualizziamo su Console il risultato di Person.ToString()
Console.WriteLine(person.ToString());
```

Nell'[esempio 7.21](#), come avveniva in precedenza, abbiamo creato un'istanza dell'oggetto `Person` tramite reflection, ma invece di dichiararla come `object`, abbiamo utilizzato la parola chiave `dynamic`, introdotta a partire da C# 4, che ci consente di attivare le funzionalità di late binding e, in ultima analisi, di utilizzare direttamente le proprietà `Name` e `Age`, come faremmo se potessimo istanziare esplicitamente un oggetto di questo tipo.

Pertanto, anche se a prima vista può non sembrare così, anche quello dell'[esempio 7.21](#) è, a tutti gli effetti, da considerarsi codice dinamico, visto che fa uso di proprietà la cui effettiva esistenza dipende dal particolare risultato del metodo `CreateInstance` e, pertanto, sarà nota solo a runtime.

[Le classi `DynamicObject` e `ExpandoObject`](#)

Come abbiamo visto, la parola chiave `dynamic` indica al compilatore C# di utilizzare dei particolari oggetti, chiamati **binder**, per associare a runtime il nome di un membro alla relativa definizione all'interno della classe; il risultato, tornando al precedente [esempio 7.19](#), è che `Name` e `Age` devono essere

effettivamente proprietà definite all'interno del tipo `Person`, altrimenti viene sollevato un errore in fase di esecuzione.

In alcuni casi, invece, può essere comodo voler modificare questo comportamento di default, intervenendo sulle logiche secondo le quali il meccanismo di late binding di C# funziona per creare oggetti dinamici, tramite cui possiamo avere il totale controllo sulle modalità di risoluzione di questi riferimenti. Un risultato di questo tipo si può ottenere creando una classe che erediti dal tipo `DynamicObject`, come accade per la classe nell'[esempio 7.22](#).

Esempio 7.22

```
public class MyDynamicObject : DynamicObject
{
    public override bool TryGetMember(
        GetMemberBinder binder, out object result)
    {
        result = "Property " + binder.Name;
        return true;
    }

    public override bool TryInvokeMember(
        InvokeMemberBinder binder, object[] args, out object result)
    {
        result = "Method " + binder.Name;
        return true;
    }
}

public class Program
{
    public static void Main()
    {
        dynamic myObject = new MyDynamicObject();
        Console.WriteLine(myObject.Property1);
        Console.WriteLine(myObject.SomeMethod("Test"));
    }
}
```

`DynamicObject` è stato introdotto con il .NET Framework 4.0 ed espone alcuni metodi che vengono eseguiti in conseguenza di un tentativo di late binding, che possiamo ridefinire per plasmarne il funzionamento secondo le nostre necessità. `MyDynamicObject`, per esempio, effettua l'override di `TryGetMember` e `TryInvokeMember`, utilizzati rispettivamente in fase di lettura di una proprietà ed esecuzione di un metodo, ritornando, come risultato, una stringa contenente il nome del membro stesso. Il risultato dell'esecuzione dell'[esempio 7.22](#) è mostrato nella [Figura 7.5](#).

La classe `DynamicObject` può, in fin dei conti, essere utilizzata per realizzare classi dinamiche, prive cioè di una struttura fissa, ma di utilizzo generale e che possiamo quindi plasmare a runtime secondo le nostre necessità. Per contro, però, si tratta di una classe astratta e, come tale, è possibile sfruttarla solo come classe base di un nostro tipo personalizzato, costringendoci quindi, di fatto, a scrivere del codice.

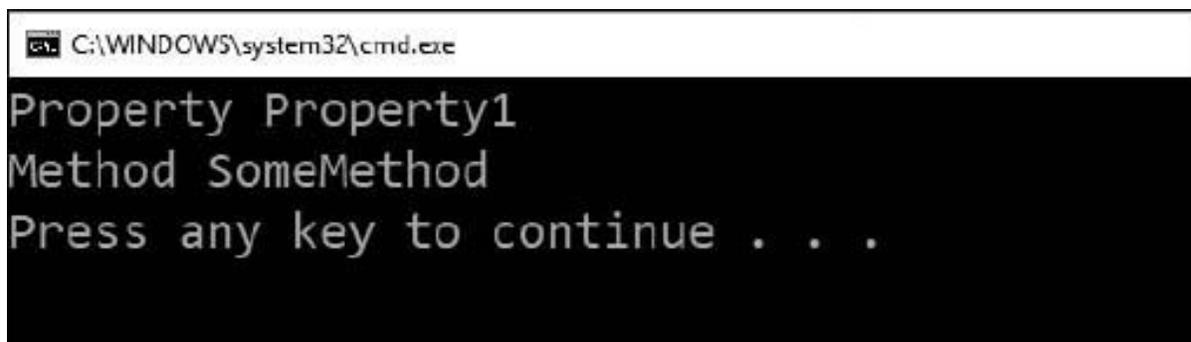


Figura 7.5 – Utilizzo di `DynamicObject`.

Un oggetto simile negli scopi a `DynamicObject`, ma per contro già dotato di una logica in grado di renderlo direttamente utilizzabile nel codice, è `ExpandoObject`, una classe i cui membri, siano essi metodi, proprietà o eventi, possono essere dinamicamente aggiunti e rimossi a runtime, come mostra l'[esempio 7.23](#).

Esempio 7.23

```
dynamic myObject = new ExpandoObject();

// Aggiunta di un metodo all'oggetto tramite lambda expression
// ToText restituisce una string e pertanto è di tipo Func<string>
myObject.ToText =
    (Func<string>)(
        () => string.Format("{0} ha {1} anni",
            myObject.Name, myObject.Age)
    );

// Aggiunta dinamica di due proprietà
myObject.Name = "Matteo Tumiati";
myObject.Age = 28;

// Invocazione del metodo definito in precedenza
Console.WriteLine(myObject.ToText());
```

Ovviamente sia `DynamicObject` sia `ExpandableObject` richiedono che sia attivo il late binding per funzionare correttamente e, pertanto, è necessario dichiarare le rispettive variabili come `dynamic`.

L'introduzione di `ExpandableObject` e `DynamicObject`, e soprattutto della parola chiave `dynamic` in C# hanno sollevato non pochi dubbi nella comunità di sviluppatori, perché consentono a tutti gli effetti di aggirare i vincoli della tipizzazione forte di C#. In realtà, il loro impiego è particolarmente comodo quando si utilizzano oggetti COM, le cui API spesso costringono lo sviluppatore a continui cast, rendendo molto onerosa la scrittura del codice e minandone la leggibilità.

Le funzionalità sin qui mostrate consentono di limitare al massimo le informazioni fornite al compilatore e di scrivere codice anche quando le definizioni degli oggetti utilizzati sono incomplete o, addirittura, assenti. Il prossimo paragrafo, invece, tratterà di un argomento che, in un certo senso, si muove in una direzione opposta, garantendoci la possibilità di integrare ulteriormente i metadati dei nostri tipi, grazie all'uso di speciali strumenti: gli attributi.

Codice dichiarativo tramite gli attributi

Nelle prime pagine di questo capitolo, quando abbiamo visto come costruire eccezioni personalizzate, abbiamo incontrato una particolare notazione tramite la quale indicare al runtime che la classe realizzata poteva essere serializzata:

```
[Serializable]
public class InvalidCustomerException : ApplicationException
{
    ...
}
```

Tale dichiarazione è realizzata con l'ausilio di un oggetto particolare, detto **attributo**, che viene utilizzato per integrare i metadati associati alla definizione di `InvalidCustomerException`; la classe `SerializableAttribute` non implementa alcun tipo di funzionalità particolare ed è totalmente estranea alla logica del tipo a cui è associata. Svolge, in pratica, semplicemente la funzione di marcatore, in modo che un oggetto interessato a determinare se

`InvalidCustomerException` sia serializzabile o no possa scoprirlo visionando gli attributi a esso associati.

I nomi delle classi che rappresentano attributi terminano convenzionalmente con il suffisso `Attribute`. Quando vengono utilizzate, questo suffisso può facoltativamente essere omissso e questa è la ragione per cui nel codice precedente si è potuto scrivere `Serializable` invece che `SerializableAttribute`. È indifferente utilizzare l'uno o l'altro identificativo.

Gli attributi possono essere associati a diversi elementi di codice: l'[esempio 7.24](#) mostra alcuni tipici casi di utilizzo.

Esempio 7.24

```
// A livello di assembly
[assembly: AssemblyVersion("1.0.0.0")]

// a livello di classe
[Serializable]
public class SomeClass
{
    // A livello di proprietà
    [XmlIgnore]
    public string MyProperty { get; set; }

    // A livello di field
    [NonSerialized]
    private string myField;

    // A livello di argomento di un metodo
    public void SomeMethod(
        [MarshalAs(UnmanagedType.LPWSTR)] string value)
    {
        // ...
    }
}
```

Gli argomenti che possiamo specificare sono quelli corrispondenti ai diversi overload del costruttore; non è necessario però che ne esista uno per ogni possibile combinazione delle proprietà da valorizzare, dato che è eventualmente possibile utilizzare la sintassi dell'[esempio 7.25](#) per specificare quelle non esplicitamente previste.

Esempio 7.25

```
[WebMethod(BufferResponse=true)]
public void SomeMethod()
{
    // ....
}
```

Quando utilizziamo gli attributi, comunque, non dobbiamo mai dimenticare la loro reale natura, ossia quella di metadati aggiuntivi a quelli già previsti dal CLR. Come tali, non vengono valutati a runtime: essendo strutture statiche, quindi, è possibile valorizzarle solo con dati costanti o con dei tipi. È quindi lecito scrivere qualcosa come

```
[MyAttribute("Test", SomeType = typeof(int))]
```

mentre invece non è possibile scrivere espressioni che necessitino una valutazione a runtime:

```
[MyAttribute("Test", new DateTime(2000, 1, 1))] // NON compila
```

L'uso degli attributi può essere un'ottima scelta in tutte le casistiche in cui stiamo realizzando delle API e vogliamo fornire all'utilizzatore la possibilità di utilizzare dei marcatori all'interno del codice, in modo che possiamo poi valutarli a runtime, dando quindi l'illusione di scrivere codice dichiarativo. Un esempio pratico di questa tecnica ha come prerequisito l'argomento del prossimo paragrafo, ossia la creazione di un attributo personalizzato.

[Costruire e utilizzare attributi custom: la classe System.Attribute](#)

Immaginiamo, per esempio, di voler realizzare un sistema in grado di produrre una tabella da una lista di oggetti e che, per ogni proprietà, vogliamo essere in grado di specificare l'intestazione della relativa colonna. L'idea è di realizzare un attributo personalizzato in grado di memorizzare queste informazioni, così che l'utilizzatore che voglia far uso della nostra funzionalità si limiti a indicarle sulle proprietà dell'oggetto da rappresentare. Per costruire un oggetto di questo tipo, non dobbiamo far altro che realizzare una classe che erediti da `System.Attribute`, come quella dell'[esempio 7.26](#).

Esempio 7.26

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple = false)] public class
ReportPropertyAttribute : Attribute
{
    public string Header { get; set; }

    public ReportPropertyAttribute(string header)
    {
        this.Header = header;
    }
}
```

Oltre a rispettare la convenzione sul naming del tipo, che richiede l'uso del suffisso `Attribute`, abbiamo specificato tramite l'attributo `AttributeUsage` i membri per i quali vogliamo dare la possibilità di utilizzare il nostro `ReportPropertyAttribute`, pena un errore in fase di compilazione: nel nostro caso sarà possibile utilizzarlo per decorare proprietà, indicandolo, al più, una sola volta. L'[esempio 7.27](#) ne mostra un tipico utilizzo.

Esempio 7.27

```
public class Person
{
    [ReportProperty("Nome")]
    public string FirstName { get; set; }

    [ReportProperty("Cognome")]
    public string LastName { get; set; }

    [ReportProperty("Età")]
    public int Age { get; set; }
}
```

A questo punto, affinché la nostra libreria di reporting sia effettivamente realizzabile, resta solo da capire come leggere questi metadati a runtime, ossia utilizzando reflection.

Esempio 7.28

```
public string BuildReport<T>(IEnumerable<T> items)
{
    var headers = new Dictionary<PropertyInfo, string>();

    // Lettura dei metadati
    foreach (PropertyInfo prop in typeof(T).GetProperties())
    {
```

```

object[] attributes =
    prop.GetCustomAttributes(typeof(ReportPropertyAttribute), true);

if (attributes.Length > 0)
{
    ReportPropertyAttribute reportAttribute =
        (ReportPropertyAttribute)attributes[0];

    headers.Add(prop, reportAttribute.Header);
}
}

// ... Qui logica per produrre il report in base
// al contenuto del dictionary headers ...
}

```

L'[esempio 7.28](#) mostra la definizione di un metodo generico `BuildReport`, che riceve in ingresso una lista di oggetti per produrre un elenco in formato `string`. Tramite reflection, esso scorre tutte le proprietà del tipo in ingresso e, per ognuna di esse, usa il metodo `GetCustomAttributes` per ricercare attributi di tipo `ReportPropertyAttribute` in uno qualsiasi degli oggetti della gerarchia di `T`. In caso affermativo, ne utilizza la proprietà `Header` per popolare un dictionary, che potremo poi utilizzare per la vera e propria logica di generazione del report, qui omessa per semplicità.

Si tratta di un semplice esempio, che tuttavia mette in luce le potenzialità dei custom attribute: possiamo immaginarli come una sorta di tool, da mettere a disposizione dei membri del nostro team, per indicare, in maniera dichiarativa, le proprietà da includere nel report e la relativa intestazione.

Grazie alle potenzialità di Visual Studio, in realtà, possiamo spingerci ancora oltre, arrivando a controllare persino il processo di interpretazione e compilazione del codice: accenneremo a queste funzioni avanzate nella prossima sezione, dedicata a Roslyn.

[Il compilatore-as-a-service: Roslyn](#)

L'introduzione di Roslyn in .NET rappresenta una vera e propria rivoluzione copernicana nell'ambito dello sviluppo software. Per la prima volta, infatti, siamo in grado di interfacciarci e sfruttare, all'interno del nostro codice, un componente dell'ambiente di sviluppo che abbiamo utilizzato migliaia di volte come se fosse una scatola nera: il compilatore.

Roslyn è una serie di librerie, rigorosamente open source e scaricabili da GitHub, che mappano le varie fasi del processo di compilazione del codice, a partire dal parsing del testo fino alla generazione degli assembly. Le possibilità

Il Syntax Visualizer è interattivo: man mano che ci spostiamo nel codice, infatti, esso si aggiorna per visualizzare il nodo corrispondente alla posizione del cursore. In maniera del tutto analoga, possiamo anche cliccare su un nodo per vedere il testo relativo evidenziato nell'editor di codice.

Se ora proviamo a modificare il testo, introducendo un errore, il visualizzatore si aggiornerà di conseguenza, mostrando in maniera evidenziata alcune delle righe che sono state impattate da esso. Per esempio, se eliminiamo il punto e virgola dopo la `get` della proprietà `LastName`, il risultato sarà quello che possiamo vedere nella [Figura 7.7](#).

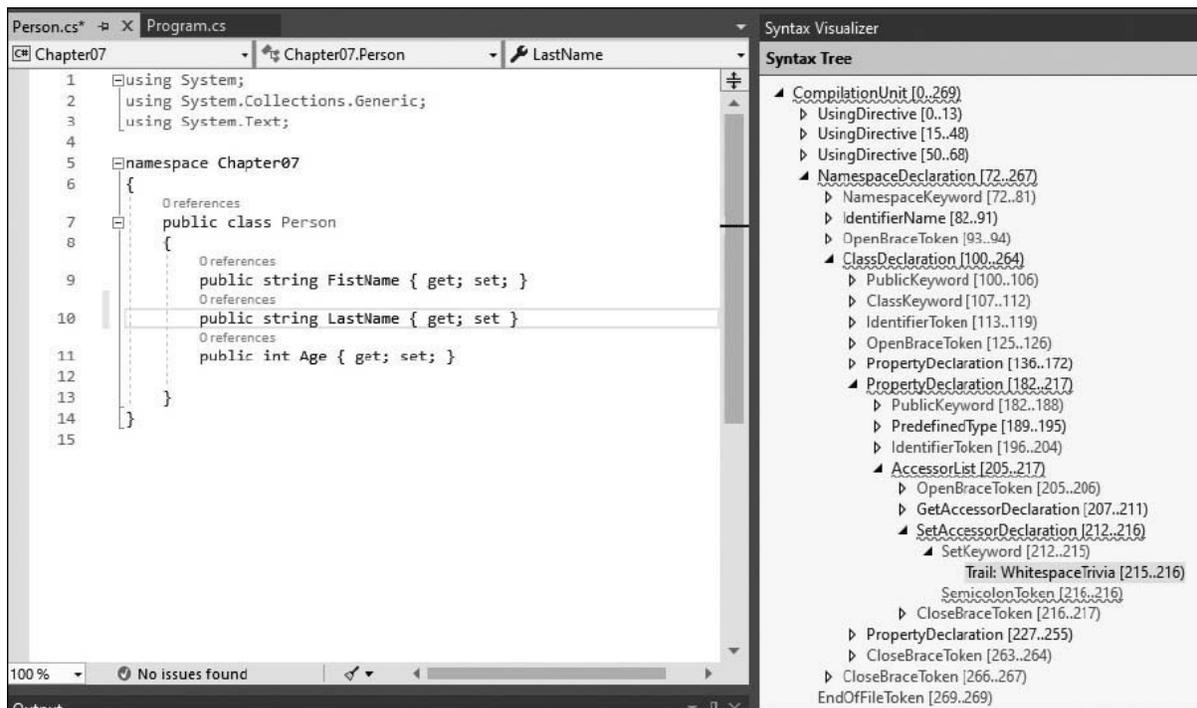


Figura 7.7 – Syntax Visualizer in Visual Studio.

Come possiamo notare, Roslyn è ancora in grado di interpretare la totalità del nostro codice, e di generare il Syntax Tree. Alcuni elementi, tuttavia, sono segnalati come errati, a causa dell'errore di sintassi che abbiamo introdotto. In molti casi, il compilatore è anche in grado di proporre una modifica per risolvere il problema. Infatti, all'interno del nodo `SetAccessorDeclaration`, è comunque presente un nodo di tipo `SemicolonToken`, che però è segnalato come "missing", cioè mancante.

Queste informazioni possono essere molto utili se vogliamo, per esempio, realizzare un add-on per Visual Studio che, sfruttando Roslyn, fornisca

funzionalità di generazione automatica del codice o di refactoring.

Analisi della sintassi

Ovviamente possiamo invocare queste API anche da codice C#, per effettuare analisi sintattica, semantica e anche riscrittura e modifica del codice. Questi argomenti sono parecchio avanzati e fuori dagli obiettivi di questo libro, tuttavia possiamo capire le potenzialità di Roslyn anche con un semplice esempio.

Creiamo quindi un nuovo progetto di tipo Code Analysis Tool (all'interno del gruppo Extensibility) e proviamo a scrivere il codice dell'[esempio 7.29](#):

Esempio 7.29

```
using Microsoft.CodeAnalysis.CSharp;
namespace ConsoleApplication2
{
    class Program
    {
        static void Main(string[] args)
        {
            var syntaxTree = CSharpSyntaxTree.ParseText(@"
public class Test
{
    public string MyProperty { get; set; }
}");
            var root = syntaxTree.GetRoot();
            var nodes = root.DescendantNodes();
        }
    }
}
```

Questo esempio costruisce un syntax tree a partire da un vero e proprio blocco di testo: grazie al metodo `ParseText` di `CSharpSyntaxTree`, infatti, possiamo chiedere al compilatore di interpretare il contenuto di una stringa e analizzarlo, generando un albero del tutto analogo a quello che abbiamo visto nella sezione precedente.

Possiamo verificarlo semplicemente eseguendo l'esempio e mettendo un breakpoint nell'ultima riga.

Conclusioni

Nella prima parte del capitolo abbiamo illustrato come le applicazioni basate sul .NET Framework possano avvalersi di un sistema strutturato per la gestione delle eccezioni a runtime: non siamo più costretti a controllare manualmente il corretto esito dell'esecuzione di un metodo, ma è il runtime stesso a notificarci, tramite un sistema interamente basato sulla modellazione a oggetti del concetto di errore, eventuali anomalie che si sono verificate. Il normale flusso logico dell'applicazione viene interrotto e, tramite opportuni blocchi di codice, possiamo intercettare particolari eccezioni per porre in atto opportune contromisure, oppure possiamo decidere semplicemente di limitarci a liberare risorse per poi inoltrare nuovamente l'errore lungo tutto lo stack di chiamate.

La seconda parte del capitolo è stata invece incentrata su concetti avanzati dello sviluppo di applicazioni tramite C# e .NET in generale. Abbiamo mostrato come, tramite l'uso di reflection, possiamo recuperare a runtime informazioni sui tipi definiti all'interno di un assembly o come sfruttare il medesimo set di classi per interagire dinamicamente con oggetti. È stato questo il primo esempio di programmazione dinamica, una tecnica avanzata di sviluppo che è implementabile anche tramite la keyword `dynamic`, che abilita funzionalità di late binding in C#. Infine, abbiamo introdotto due classi,

`DynamicObject` ed `ExpandObject`, che possiamo sfruttare per realizzare tipi in grado di modificare la propria struttura a runtime.

Successivamente abbiamo trattato gli attributi, grazie ai quali ci è possibile integrare i metadati degli oggetti con delle ulteriori informazioni, che possono poi essere recuperate tramite reflection. Questa infrastruttura consente di realizzare codice dichiarativo con grande facilità, come abbiamo avuto modo di apprezzare realizzando un metodo in grado di produrre report generici basandosi sull'uso di attributi personalizzati.

Nell'ultima parte, abbiamo accennato a Roslyn, una libreria che può essere descritta con la frase "Compiler-as-a-service": si tratta infatti di una serie di API, tramite cui possiamo accedere alle funzionalità del compilatore, come parser e tokenizer. Si tratta di un progetto open source molto ben documentato, mediante il quale possiamo realizzare add-on per Visual Studio, come nuovi refactoring o generatori di codice. Abbiamo visto come, una volta installato l'SDK, abbiamo a disposizione un Syntax Visualizer che ci aiuta nella comprensione dell'object model e del funzionamento del parser.

Il prossimo capitolo servirà invece a presentare una delle peculiarità in assoluto più interessanti e utilizzate del mondo .NET, che trova applicazione in un ampio spettro di situazioni, ovvero l'esecuzione di codice asincrono.

Async, multithreading e codice parallelo

Al giorno d'oggi praticamente tutti i personal computer sono dotati di processori multi-core. Realizzare applicazioni in grado di gestire simili architetture rappresenta una vera e propria sfida, dato che è necessario ripensare e ristrutturare gli algoritmi finora utilizzati affinché possano essere eseguiti in parallelo.

Il **multithreading** rappresenta la prima tecnica che, come sviluppatori, abbiamo a disposizione per migliorare le prestazioni della nostra applicazione. Proprio a questa tecnica è dedicata la prima metà di questo capitolo, in cui vedremo le differenti possibilità che il framework ci fornisce per eseguire diverse porzioni di codice nello stesso momento.

Ma avviare più thread contemporaneamente non basta a garantire che il carico di lavoro sia equamente distribuito tra le diverse CPU a disposizione. Nella seconda parte del capitolo, introdurremo una serie di classi, presenti a partire dal .NET Framework 4.0 e denominate **Parallel Extensions**, tramite le quali possiamo scrivere codice parallelo, tralasciando i dettagli relativi allo scenario hardware in cui esso viene eseguito. Successivamente vedremo come l'utilizzo delle keyword `async` e `await`, presenti già a partire da C# 5, semplifichino notevolmente la scrittura di codice asincrono.

Infine, avremo modo di vedere come la vera e propria sfida in questo ambito di sviluppo risieda nella facilità con cui l'accesso concorrente alle risorse, se mal gestito, può comportare bug a runtime difficilmente riproducibili e individuabili. Analizzeremo anche in questo caso gli strumenti messi a disposizione dal sistema operativo e da .NET per evitare questo tipo di problemi.

Processi e thread

Quando richiediamo l'esecuzione di un programma, il sistema operativo crea un'istanza di un particolare oggetto del kernel, chiamato **process** (processo), a cui assegna un ben definito (e isolato) spazio di indirizzamento in memoria. Un processo, di per sé, non è in grado di eseguire alcun codice e svolge il compito di puro e semplice contenitore di quelle che potremmo definire come le entità funzionali elementari del sistema operativo: i thread. Ogni processo dispone di almeno un thread, chiamato **primary thread**, al termine del quale esso viene distrutto, liberando lo spazio di memoria e le risorse ad esso assegnate.

Il normale ciclo di vita di un'applicazione consiste nell'esecuzione sequenziale di numerosi blocchi di codice richiamati dal thread principale, che a loro volta possono richiamare ulteriori blocchi di codice. Quando questo avviene, il metodo chiamante risulta ovviamente bloccato fino a che la routine invocata non giunge al termine. Si tratta di un approccio con diversi limiti e, per forza di cose, non sempre percorribile: se i nostri processi fossero in grado di eseguire un solo thread, il controllo ortografico di Microsoft Word causerebbe continue interruzioni nella digitazione del testo oppure ogni richiesta pervenuta a un'applicazione ASP.NET impegnerebbe il server per tutta la durata di elaborazione, fino alla generazione della risposta. Fortunatamente ogni thread ha la possibilità di assegnare a un thread secondario l'esecuzione di un metodo. In questo caso, la chiamata a quest'ultimo ritorna immediatamente il controllo al thread chiamante e i due blocchi di codice possono effettivamente essere eseguiti in parallelo.

.NET fornisce un gran numero di classi per gestire questo tipo di situazioni all'interno dell'applicazione e rendere possibile l'esecuzione contemporanea di diversi metodi. Esse sono, per lo più, contenute all'interno del namespace `System.Threading`, al quale la classe `Thread` appartiene.

La classe `System.Threading.Thread`

Il codice di un'applicazione .NET viene eseguito da un **managed thread**, ossia da un thread gestito dal Common Language Runtime. Si tratta di un'astrazione di più alto livello rispetto al thread del sistema operativo; più thread gestiti, infatti, possono essere in concreto implementati utilizzando il medesimo thread di sistema, al fine di risparmiare risorse, oppure un managed

thread può essere eseguito parzialmente tramite un windows thread per poi essere spostato su un altro. Si tratta di ottimizzazioni che, per noi utilizzatori, risultano assolutamente trasparenti, dato che è il runtime preoccuparsi di gestire tutte queste tipologie di operazioni, in assoluta autonomia, a seconda del particolare scenario di esecuzione.

All'interno del framework, un thread è rappresentato dall'omonima classe Thread. Il suo costruttore accetta un delegate di tipo ThreadStart (o ParametrizedThreadStart, nel caso dobbiamo passare anche dei parametri), tramite il quale possiamo fornire al thread il codice che esso dovrà eseguire, come mostrato nell'[esempio 8.1](#).

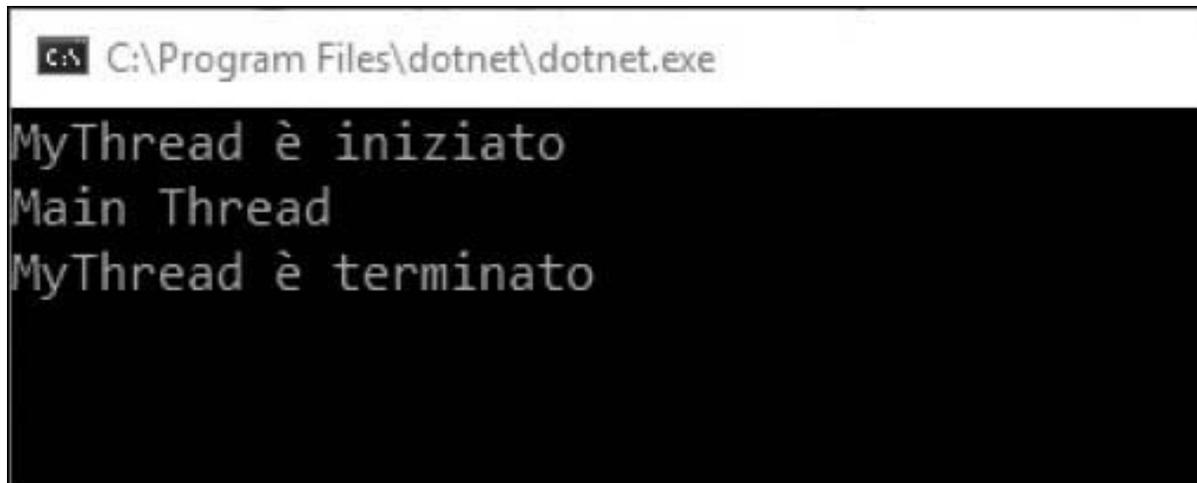
Esempio 8.1

```
public static void Main()
{
    Thread myThread = new Thread(() =>
    {
        Console.WriteLine("MyThread è iniziato");
        Thread.Sleep(1000);
        Console.WriteLine("MyThread è terminato");
    });

    // Esecuzione di myThread
    myThread.Start();

    Thread.Sleep(500);
    Console.WriteLine("Main Thread");
}
```

Nel codice abbiamo creato una nuova istanza della classe Thread, a cui abbiamo assegnato un metodo, definito tramite la notazione delle lambda expression, che visualizza un messaggio in console dopo un secondo di attesa. Questo delegate viene preso in consegna da myThread ed eseguito in un thread separato, chiamato **worker thread**, all'invocazione del metodo start. Il risultato dell'esecuzione è quello mostrato nella [Figura 8.1](#).



```
C:\Program Files\dotnet\dotnet.exe
MyThread è iniziato
Main Thread
MyThread è terminato
```

Figura 8.1 – Esecuzione dell'[esempio 8.1](#).

Come possiamo notare, ciò che accade è che effettivamente i due blocchi di codice, appartenenti al thread principale e a `myThread`, sono eseguiti in parallelo, tant'è che l'output su console prodotto dal primo appare nel mezzo dei due messaggi stampati dal secondo, secondo lo schema mostrato nella [Figura 8.2](#).

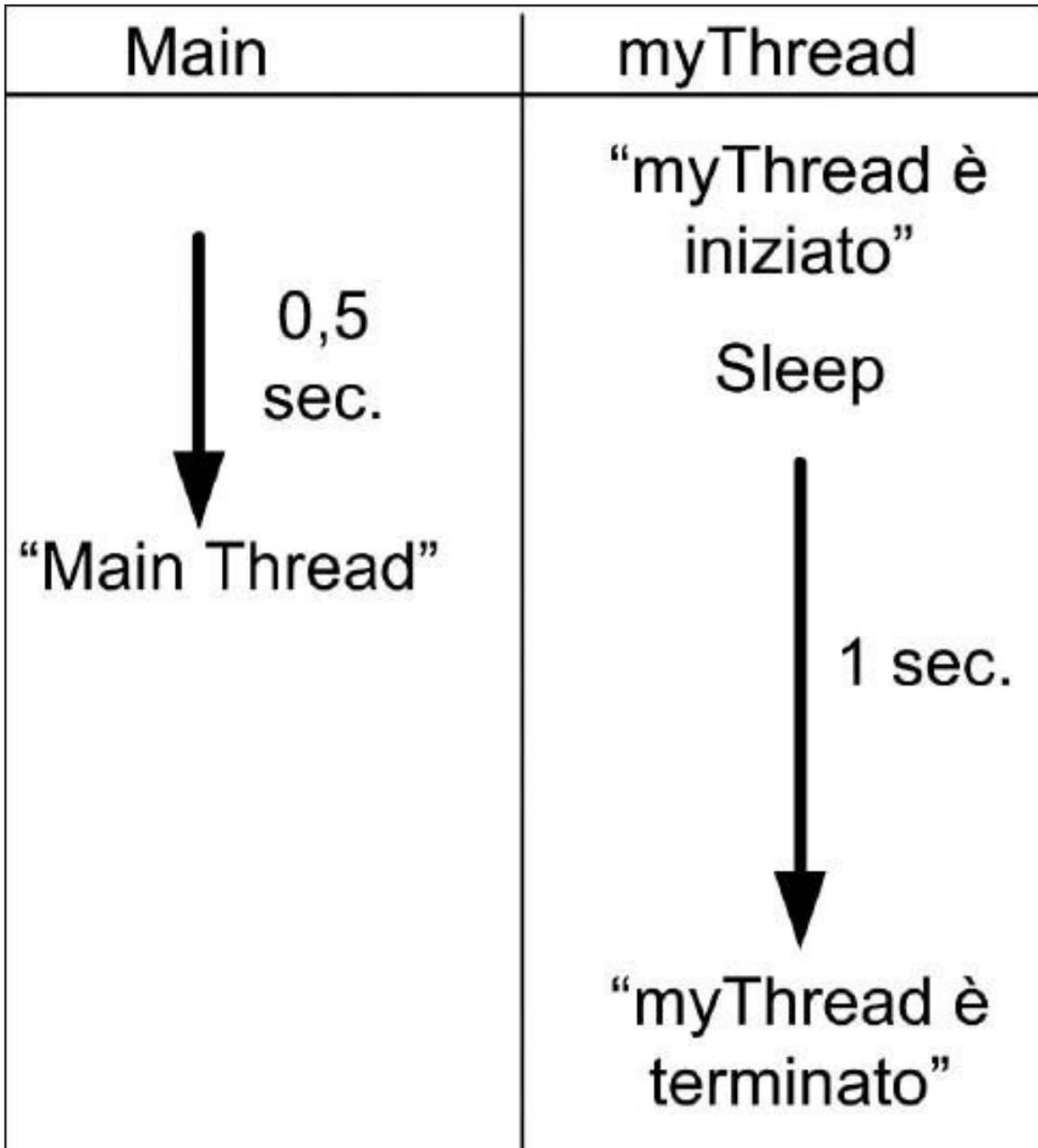


Figura 8.2 – Parallelismo dei thread dell’[esempio 8.1](#).

*Quando viene eseguita l’istruzione `Console.WriteLine` del thread principale, quest’ultimo termina la sua esecuzione; in condizioni normali, ciò comporterebbe la chiusura dell’applicazione senza visualizzare il messaggio prodotto dal worker thread “MyThread è terminato”. In realtà, nell’[esempio 8.1](#) ciò non accade perché quest’ultimo è un **thread di foreground** e, come tale, è in grado di mantenere in vita l’applicazione finché non è concluso. Tramite la*

*proprietà `IsBackground`, si può configurare `myThread` come **thread di background**; in questo caso l'applicazione non deve attenderne il completamento per poter terminare.*

L'esempio che abbiamo mostrato esegue codice isolato, che non accetta parametri dall'esterno. Si tratta di una limitazione che elimineremo nel prossimo paragrafo.

Passare parametri a un worker thread

Nel caso in cui dobbiamo passare un parametro al codice eseguito nel worker thread, possiamo avvalerci di un delegate di tipo `ParametrizedThreadStart`, che accetta un argomento di tipo `object`, come nell'[esempio 8.2](#).

Esempio 8.2

```
string someVariable = "Matteo Tumiati";
var workerThread = new Thread(o =>
{
    Console.WriteLine("Saluti da: {0}", o);
});
workerThread.Start(someVariable);
```

Il parametro può poi essere valorizzato tramite l'overload del metodo `start`, che abbiamo utilizzato, nel nostro caso, per passare la variabile `someVariable` al thread secondario. Sebbene, tramite l'utilizzo delle lambda expression, possiamo referenziare direttamente questa variabile dal delegate, evitando quindi l'utilizzo di un `ParametrizedThreadStart`, quello mostrato è l'unico modo corretto per passare parametri a un worker thread, evitando **race condition**, ossia comportamenti anomali dell'applicazione che dipendono dalle tempistiche di esecuzione dei vari thread. Per comprendere meglio il problema, consideriamo l'[esempio 8.3](#).

Esempio 8.3

```
string someVariable = "Matteo Tumiati";
var workerThread = new Thread(() =>
{
    Thread.Sleep(500);
});
```

```
        Console.WriteLine("Saluti da: {0}", someVariable);
    });

workerThread.Start();
someVariable = "Daniele Bochicchio";
```

Nel codice precedente, abbiamo referenziato `someVariable` direttamente in fase di costruzione del thread, supponendo pertanto di visualizzare il nome “*Matteo Tumiati*”. Questa variabile, però, viene in realtà valutata solo al momento dell’esecuzione di `Console.WriteLine`, non quando viene istanziato `workerThread`, né quando ne viene invocato il metodo `start`, stampando a console “*Daniele Bochicchio*”. In sostanza, il risultato dell’[esempio 8.3](#) dipende da chi, fra thread principale e worker thread, accede per primo a `someVariable` (da qui il termine *race condition*). Si tratta di un problema che, sovente, è fonte di bug difficilmente individuabili e riproducibili (come, in generale, tutti quelli causati da un utilizzo improprio del multithreading) e pertanto è necessario porre estrema attenzione a questi aspetti, passando i parametri in maniera corretta, come abbiamo visto nell’[esempio 8.2](#).

Finora ci siamo già scontrati con le prime problematiche di sincronizzazione e abbiamo sfruttato il metodo `sleep`, che blocca un thread per un tempo prestabilito, per regolare l’esecuzione delle nostre applicazioni di esempio. Un simile approccio non è ovviamente possibile in un contesto reale; per queste necessità esiste uno strumento più adeguato, di cui ci occuperemo nel prossimo paragrafo.

[Controllare il flusso di esecuzione di un thread](#)

Il vantaggio di utilizzare un oggetto di tipo `Thread` nella nostra applicazione è quello di poterne controllare in maniera esplicita il flusso di esecuzione. Una problematica estremamente comune da dover gestire, per esempio, è quella di ricevere una notifica quando un worker thread ha terminato il proprio lavoro. Questo risultato può essere raggiunto tramite l’uso del metodo `Join`.

Esempio 8.4

```
var list = new List<Thread>();

// Qui creiamo ed eseguiamo cinque worker thread for (int index = 0; index < 5; index++)
{
    var myThread = new Thread((currentIndex) =>
    {
        Console.WriteLine("Thread {0} è iniziato", currentIndex);
    });
}
```

```

        Thread.Sleep(500);
        Console.WriteLine("Thread {0} è terminato", currentIndex);
    });
    myThread.Start(index);
    list.Add(myThread);
}

// Attesa del completamento di ognuno dei worker thread
foreach (Thread thread in list)
{
    thread.Join();
}

Console.WriteLine("Esecuzione di tutti i thread terminata");

```

L'obiettivo del codice dell'[esempio 8.4](#) è quello di visualizzare un opportuno messaggio su console solo nel momento in cui tutti i worker thread costruiti all'interno del ciclo `for` abbiano completato la loro esecuzione. Invocando il metodo `Join` dal thread principale, se ne blocca l'esecuzione finché l'istanza del worker thread non abbia terminato di eseguire il relativo codice. Eventualmente, `Join` accetta come argomento anche un timeout, che rappresenta il tempo massimo di attesa al termine del quale restituire il controllo.

```

// Attesa di massimo un secondo thread.Join(1000);

```

Un'altra necessità piuttosto comune è quella di interrompere del tutto un worker thread: si pensi, per esempio, al caso in cui si voglia dare la possibilità all'utente di annullare l'esecuzione di una lunga operazione. In questi casi, il metodo da utilizzare è `Abort`, come mostrato nell'[esempio 8.5](#).

Esempio 8.5

```

var workerThread = new Thread(() =>
{
    Console.WriteLine("Inizio di un thread molto lungo");
    Thread.Sleep(5000);
    Console.WriteLine("Termine worker thread");
});

workerThread.Start();
workerThread.Join(500);

// Se il worker thread è ancora in esecuzione lo si cancella
if (workerThread.ThreadState != ThreadState.Stopped)
{
    workerThread.Abort();
}

Console.WriteLine("Termine applicazione");

```

Il codice precedente, dopo aver invocato il metodo `Join` per attendere il completamento del worker thread, verifica tramite la proprietà `ThreadState` se quest'ultimo sia ancora in esecuzione e, in caso affermativo, lo cancella, invocando il metodo `Abort`. Quando l'esecuzione di un thread viene cancellata, il relativo codice viene interrotto da una `ThreadAbortException`, che possiamo eventualmente gestire per introdurre del codice di cleanup.

```
var workerThread = new Thread(() =>
{
    try
    {
        Console.WriteLine("Inizio di un thread molto lungo");
        Thread.Sleep(5000);
        Console.WriteLine("Termine worker thread");
    }
    catch (ThreadAbortException ex)
    {
        // qui codice per gestire l'eccezione
    }
});
```

Finora abbiamo implementato funzionalità multithreading nelle nostre applicazioni, creando in maniera esplicita istanze della classe `Thread`. Una tale pratica è consigliata solo nei casi in cui vogliamo controllare in maniera diretta il ciclo di vita e le caratteristiche del thread secondario, per esempio, rendendolo o no di background tramite la proprietà `IsBackground` oppure impostandone la priorità:

```
workerThread.IsBackground = false; workerThread.Priority = ThreadPriority.Lowest;
```

Per tutte le altre casistiche, invece, questo non è un approccio consigliabile, dato che il Common Language Runtime mantiene già, per ogni applicazione, un certo numero di thread in esecuzione, in modo da ottimizzare l'utilizzo delle risorse di sistema in relazione al particolare ambiente di esecuzione: il `ThreadPool`.

[*Il ThreadPool per applicazioni multithreading*](#)

Il numero ottimale di thread per eseguire codice parallelo varia secondo il particolare scenario hardware e software, in cui l'applicazione viene eseguita. In generale, però, quando realizziamo applicazioni multithreading, dobbiamo tenere presente che:

- ❑ ogni thread consuma risorse di sistema e impegna una certa quantità di memoria;
- ❑ la creazione e la distruzione di un thread sono, in generale, operazioni costose in termini prestazionali.

Per limitare questi inconvenienti e consentire allo stesso tempo l'esecuzione parallela di codice, l'idea è quella di non distruggere un thread una volta che questi abbia terminato l'esecuzione ma di mantenerlo attivo per un certo periodo di tempo in stato **Idle**, in modo che possa essere riutilizzato in seguito. Il CLR mette pertanto a disposizione un contenitore, chiamato **Thread Pool**, all'interno del quale mantiene una lista di thread attivi e tramite il quale permette di gestire le code dei task che vengono a esso via via assegnati. La classe statica `ThreadPool` espone il metodo `QueueUserWorkItem`, che accetta un delegate di tipo `WaitCallback`, tramite il quale possiamo accodare un nuovo task da gestire in parallelo.

Esempio 8.6

```
ThreadPool.QueueUserWorkItem((o) =>
{
    Console.WriteLine("Inizio worker thread");
    Thread.Sleep(1000);
    Console.WriteLine("Termine worker thread");
});

Thread.Sleep(500);
Console.WriteLine("Metodo main");
Thread.Sleep(2000);
```

L'[esempio 8.6](#) è molto simile concettualmente al precedente 8.1, ma non istanzia direttamente un oggetto di tipo `Thread` e si avvale del thread pool del CLR per eseguire un task in parallelo.

Tutti i worker thread appartenenti al pool sono thread di background e quindi l'applicazione non deve attenderne la

conclusione per terminare. Pertanto, nell'[esempio 8.6](#) siamo stati costretti a usare il metodo `Thread.Sleep` per dare tempo al pool di completare il task assegnato.

Quando un task richiede un parametro in ingresso, possiamo specificarlo tra gli argomenti di `QueueUserWorkItem`, come mostrato nell'[esempio 8.7](#).

Esempio 8.7

```
string someVariable = "Matteo Tumiatì";

ThreadPool.QueueUserWorkItem((argument) =>
{
    Thread.Sleep(500);
    Console.WriteLine("Saluti da: {0}", argument);
}, someVariable);

someVariable = "Daniele Bochicchio";
Thread.Sleep(2000);
```

Per questa particolare necessità, infatti, valgono tutte le considerazioni che abbiamo fatto nelle pagine precedenti a proposito delle **race condition**. Pertanto, sebbene sia possibile utilizzare variabili esterne al codice accodato tramite le lambda expression, è comunque assolutamente sconsigliato procedere in questo senso. Specificando invece il valore del parametro in fase di accodamento del task, come avviene nel codice dell'[esempio 8.7](#), siamo sicuri che quest'ultimo venga eseguito con il valore atteso, anche nel caso in cui la variabile di partenza dovesse cambiare per opera di un altro thread.

La sincronizzazione dei task eseguiti in questo modo richiede alcuni accorgimenti nel codice, dato che, non essendo disponibile un'istanza di `Thread`, non è possibile utilizzarne il metodo `Join` per poterne attendere la conclusione. Allo scopo, possiamo sfruttare la classe `ManualResetEvent`, come nell'[esempio 8.8](#).

Esempio 8.8

```
var list = new List<ManualResetEvent>();
try
{
    for (int index = 0; index < 5; index++)
    {
        // Creazione del waitHandle per il task corrente
        var waitHandle = new ManualResetEvent(false);
        list.Add(waitHandle);
    }
}
```

```

// Oggetto da passare al task accodato
var state = new Tuple<int, ManualResetEvent>(
    index, waitHandle);

ThreadPool.QueueUserWorkItem((untypedState) =>
{
    // WaitCallback accetta un object, pertanto
    // è necessario un cast
    var taskState = (Tuple<int, ManualResetEvent>)untypedState;

    // Visualizzazione dei messaggi su console
    Console.WriteLine("Thread {0} è iniziato", taskState.Item1);
    Thread.Sleep(500);
    Console.WriteLine("Thread {0} è terminato", taskState.Item1);

    // Segnalazione del termine dell'esecuzione del task
    // utilizzando il Set del ManualResetEvent
    taskState.Item2.Set();
}, state);
}

// Attesa che tutti i ManualResetEvent siano in stato Set
foreach (ManualResetEvent handle in list)
{
    handle.WaitOne();
}
}
finally
{
    foreach (ManualResetEvent handle in list)
    {
        handle.Dispose();
    }
}

Console.WriteLine("Esecuzione terminata");

```

In questo caso il codice è divenuto leggermente più complesso rispetto a quello che istanziava i thread in maniera esplicita. Ad ogni iterazione del ciclo `for`, viene costruita una nuova istanza di un oggetto di tipo `Tuple`, che contiene le due informazioni che vogliamo passare al task, ossia:

- ❑ l'indice dell'iterazione, che sarà poi stampato a video;
- ❑ l'istanza di `ManualResetEvent`, che il task può poi utilizzare per segnalare il termine della sua esecuzione, invocandone il metodo `set`.

Quest'ultimo è anche memorizzato all'interno di una lista, così che, in seguito, possiamo invocare per ognuno dei `ManualResetEvent` istanziati il metodo `waitone`. Esso è molto simile a `Thread.Join` e agisce bloccando l'esecuzione del thread corrente finché non sia impostato in stato **Set**, permettendoci quindi di attendere, di fatto, la conclusione di tutti i task accodati.

Oltre a `ManualResetEvent`, .NET mette a disposizione anche una versione meno onerosa per il sistema, chiamata `ManualResetEventSlim`. Essa possiede la limitazione di non poter essere condivisa tra più processi, ma assicura migliori prestazioni qualora i tempi d'attesa siano brevi.

La classe `ManualResetEvent` implementa l'interfaccia `IDisposable` e, pertanto, è buona norma invocarne il metodo `Dispose` all'interno di un blocco `try...finally`.

[Asynchronous programming model](#)

Nel corso del [Capitolo 6](#) abbiamo accennato al fatto che i delegate in .NET possono essere utilizzati per eseguire codice in modalità asincrona, tramite il metodo `BeginInvoke`. Come abbiamo visto, quest'ultimo presenta la caratteristica di ritornare immediatamente il controllo al thread chiamante e di iniziare l'esecuzione del delegate, utilizzando internamente il thread pool del CLR, come mostrato dal codice dell'[esempio 8.9](#).

Esempio 8.9

```
public delegate int SomeDelegate(string parameter);

public static void Main()
{
    var method = new SomeDelegate((parameter) =>
    {
        Thread.Sleep(1000);
        Console.WriteLine("Ciao da {0}", parameter);
        return parameter.Length;
    });

    method.BeginInvoke("Matteo Tumiati", null, null);
    Console.WriteLine("Esecuzione avviata");
    Thread.Sleep(2000);
}
```

Da un punto di vista funzionale, questo tipo di implementazione non si discosta molto da quelle viste in precedenza, salvo il fatto che, questa volta, il passaggio di parametri al metodo asincrono avviene in maniera tipizzata, mentre sia nel caso dell'utilizzo diretto della classe `Thread` sia avvalendosi del thread pool, l'eventuale parametro è sempre di tipo `object`. Oltre che per questo importante vantaggio, l'utilizzo dei delegate per eseguire operazioni

asincrona risulta estremamente comodo per la sua versatilità nell'intercettare il termine dell'elaborazione parallela e, in particolare, per recuperarne il risultato. Esistono tre diverse modalità per raggiungere questo scopo, e saranno l'argomento dei prossimi paragrafi.

Utilizzo del metodo EndInvoke

Accanto al metodo `BeginInvoke` che abbiamo visto in precedenza, ogni delegate espone anche un metodo `EndInvoke` che può essere utilizzato per attendere la conclusione dell'operazione asincrona e recuperarne il risultato. L'[esempio 8.9](#), nel quale abbiamo ancora una volta impropriamente utilizzato per questo scopo il metodo `Thread.Sleep`, può essere riscritto come mostrato qui nell'[esempio 8.10](#).

Esempio 8.10

```
var method = new SomeDelegate((parameter) =>
{
    Thread.Sleep(1000);
    Console.WriteLine("Ciao da {0}", parameter);
    return parameter.Length;
});

IAsyncResult asyncResult =
    method.BeginInvoke("Matteo Tumiati", null, null);

Console.WriteLine("Esecuzione avviata");

// Attesa del termine dell'operazione e recupero risultato
int result = method.EndInvoke(asyncResult);

Console.WriteLine("Il risultato è {0}", result);
```

Il metodo `BeginInvoke` avvia l'esecuzione in parallelo del codice del delegate e ritorna immediatamente il controllo al chiamante, dandoci quindi la possibilità di eseguire altre istruzioni nel thread principale. Esso restituisce un oggetto di tipo `IAsyncResult`, che dobbiamo utilizzare come argomento nella successiva chiamata a `EndInvoke`. Quest'ultimo ha la caratteristica di bloccare il thread in esecuzione fino al termine dell'operazione asincrona, consentendoci quindi, a tutti gli effetti, di sincronizzare i due thread; nel caso in cui il delegate preveda un valore di ritorno, `EndInvoke` lo restituisce come risultato, come possiamo notare nell'esempio precedente, in cui l'abbiamo visualizzato su console.

Sincronizzazione tramite `IAsyncResult` e polling

Una modalità alternativa per attendere il termine dell'elaborazione in parallelo è quella di utilizzare direttamente l'oggetto restituito dal metodo `BeginInvoke`. Esso implementa l'interfaccia `IAsyncResult`, che espone i membri elencati nella [Tabella 8.1](#).

Tabella 8.1 – Membri dell'interfaccia `IAsyncResult`

Nome	Descrizione
<code>AsyncState</code>	Proprietà di tipo <code>object</code> che può essere utilizzata per passare un oggetto arbitrario tra le varie fasi dell'invocazione asincrona.
<code>AsyncWaitHandle</code>	Contiene un riferimento a un <code>waitHandle</code> che può essere utilizzato per sincronizzare l'esecuzione del delegate asincrono.
<code>CompletedSynchronously</code>	Proprietà di tipo <code>bool</code> che indica se il task è stato completato in maniera sincrona.
<code>IsCompleted</code>	Proprietà di tipo <code>bool</code> che indica se l'esecuzione del task è stata completata.

In particolare, la proprietà `AsyncWaitHandle` restituisce un oggetto di tipo `waitHandle`, del tutto analogo al `ManualResetEvent`, che abbiamo utilizzato nei paragrafi precedenti e che possiamo sfruttare anche nel caso dei delegate, come nell'[esempio 8.11](#).

Esempio 8.11

```
var method = new SomeDelegate((parameter) =>
{
    Thread.Sleep(1000);
    Console.WriteLine("Ciao da {0}", parameter);
    return parameter.Length;
});

IAsyncResult asyncResult =
    method.BeginInvoke("Matteo Tumiatì", null, null);

Console.WriteLine("Esecuzione avviata");

// Polling sul WaitHandle
while (!asyncResult.IsCompleted)
{
    Console.WriteLine("Esecuzione in corso...");
    asyncResult.AsyncWaitHandle.WaitOne(200);
}
```

```
// Attesa del termine dell'operazione e recupero risultato
int result = method.EndInvoke(asyncResult);

Console.WriteLine("Il risultato è {0}", result);
```

Come possiamo notare, l'utilizzo diretto di `IAsyncResult` offre il vantaggio di consentirci una maggiore flessibilità nel gestire l'attesa per la conclusione del task. Nel codice precedente abbiamo implementato un **algoritmo di polling** che effettua periodicamente questa verifica, inviando, nel frattempo, delle notifiche all'utente. Al termine dell'attesa, possiamo comunque utilizzare il metodo `EndInvoke` (che, a questo punto, ritorna immediatamente il controllo al thread chiamante, visto che il delegate ha terminato il suo lavoro), per recuperare il risultato dell'invocazione.

Utilizzo di un metodo di callback

In alcune occasioni può capitare che, una volta iniziata l'esecuzione asincrona di un delegate, vogliamo evitare di gestirne il termine dal metodo chiamante, perché quest'ultimo dovrà successivamente occuparsi di altre funzionalità della nostra applicazione. In casi simili, in luogo delle due tecniche viste in precedenza, possiamo intercettare la conclusione dell'invocazione, configurando il delegate in modo che esegua un **metodo di callback**, come avviene nel codice dell'[esempio 8.12](#).

Esempio 8.12

```
public static void Main()
{
    var method = new SomeDelegate((parameter) =>
    {
        Thread.Sleep(1000);
        Console.WriteLine("Ciao da {0}", parameter);
        return parameter.Length;
    });

    method.BeginInvoke("Matteo Tumiati", MyCallback, null);

    Console.WriteLine("Esecuzione avviata");
    Console.ReadLine();
}

public static void MyCallback(IAsyncResult ar)
{
    Console.WriteLine("Esecuzione terminata");
}
```

La signature di `MyCallback` deve rispettare quella del delegate `AsyncCallback`, che accetta in ingresso l'istanza di `IAAsyncResult` associata all'esecuzione del delegate asincrono.

Nel caso esso produca un risultato e vogliamo recuperarlo, possiamo avvalerci della proprietà `AsyncState` di `IAAsyncResult`, che abbiamo citato nel paragrafo precedente. Si tratta di un generico contenitore a cui, in corrispondenza del metodo `BeginInvoke`, possiamo associare qualsiasi tipo di oggetto, e quindi anche l'istanza del delegate stesso, in modo che possiamo poi utilizzarlo secondo i nostri scopi, come nell'[esempio 8.13](#).

Esempio 8.13

```
public static void Main()
{
    var method = new SomeDelegate((parameter) =>
    {
        Thread.Sleep(1000);
        Console.WriteLine("Ciao da {0}", parameter);
        return parameter.Length;
    });

    method.BeginInvoke("Matteo Tumiati", MyCallback, method);

    Console.WriteLine("Esecuzione avviata");
    Console.ReadLine();
}

public static void MyCallback(IAAsyncResult ar)
{
    Console.WriteLine("Esecuzione terminata");

    var method = (SomeDelegate)ar.AsyncState;
    int result = method.EndInvoke(ar);

    Console.WriteLine("Il risultato è {0}", result);
}
```

Quando ci troviamo all'interno del metodo di callback, l'esecuzione del delegate asincrono è terminata e pertanto possiamo invocare il metodo `EndInvoke`, certi che esso completi immediatamente l'invocazione, ritornando il risultato desiderato.

Questa tecnica risulta particolarmente utile nell'ambito di applicazioni desktop o RIA, come quelle realizzate con WPF o Silverlight: in questi scenari, infatti, usare il thread principale per intercettare il termine di un'esecuzione asincrona equivale a bloccare l'interfaccia utente, peggiorando quindi la user experience. Possiamo pensare allora di avviare il task tramite `BeginInvoke`, di ritornare immediatamente il controllo all'utente e di gestirne

successivamente la chiusura all'invocazione del metodo di callback. Tuttavia, dobbiamo prestare estrema attenzione al fatto che il metodo di callback viene eseguito all'interno del worker thread associato al delegate asincrono. Pertanto, esso non può essere utilizzato direttamente per aggiornare gli elementi dell'interfaccia utente.

Le funzionalità viste finora fanno parte di .NET sin dalle primissime versioni, quando avere più di una CPU a disposizione era una peculiarità solo dei migliori server. Anche in simili ambienti, però, il sistema operativo (e in particolar modo lo scheduler), rende l'illusione di riuscire a processare più istruzioni contemporaneamente, assegnando ogni flusso di codice alla CPU per un certo periodo di tempo. Ma, come è facilmente immaginabile, possiamo parlare di vero e proprio parallelismo solo quando due o più thread siano assegnati ad altrettanti distinti core messi a disposizione dall'hardware. Il framework dispone di una libreria specifica per l'esecuzione parallela di codice, che sarà argomento delle prossime pagine.

Esecuzione parallela con Parallel Extensions

Con l'avvento e il diffondersi di macchine multi-core, per riuscire a sfruttare al massimo le potenzialità dell'hardware è necessario strutturare le applicazioni secondo architetture innovative volte, in particolare, a favorire l'uso del parallelismo. Non si tratta di un'operazione semplice ma avere a disposizione strumenti evoluti rappresenta sicuramente un enorme vantaggio, soprattutto quando forniscono a noi sviluppatori la possibilità di concentrarci solo sugli aspetti logici del requisito da soddisfare, come se creassimo semplice codice multithread, senza doverci preoccupare dei dettagli tecnici con cui il parallelismo viene implementato.

All'interno del .NET Framework 4.0 trova spazio una libreria, chiamata **Parallel Extensions**, che invece è in grado di gestire thread multipli in maniera ottimizzata in base al numero di CPU, così che anche le nostre applicazioni possano effettivamente eseguire codice in parallelo. Essa si compone di due elementi fondamentali:

- ❑ **Task Parallel Library (TPL)**, cioè un insieme di classi, tramite le quali siamo in grado di eseguire in parallelo porzioni di codice personalizzate;

- ❑ **Parallel LINQ (PLINQ)**, ossia una serie di extension method che ci consentono di sfruttare il parallelismo per calcolare il risultato di query di LINQ to Objects.

Lo scopo del prossimo paragrafo è proprio quello di imparare a conoscere la Task Parallel Library per capire nel dettaglio come sfruttarla al meglio. Parleremo di Parallel LINQ nel prossimo capitolo.

La Task Parallel Library

La classe `Task` appartiene al namespace `System.Threading.Tasks` e fornisce un'interfaccia evoluta per la creazione e il controllo sull'esecuzione di codice parallelo. Essa basa la sua interfaccia sull'utilizzo dei delegate `Action` e `Func`, a seconda del fatto che eseguiamo una procedura o una funzione. L'[esempio 8.14](#) mostra alcuni casi tipici di utilizzo.

Esempio 8.14

```
// Costruzione di un semplice task
var simpleTask = Task.Factory.StartNew(() =>
{
    Thread.Sleep(1000);
    Console.WriteLine("Ciao da simpleTask");
});

// Costruzione di un task con parametro in input
var parameterTask = Task.Factory.StartNew((name) =>
{
    Thread.Sleep(1000);
    Console.WriteLine("Ciao da parameterTask, {0}", name);
}, "Matteo Tumiatì");

// Costruzione di un task che ritorna un risultato
var resultTask = Task.Factory.StartNew((inputValue) =>
    PerformSomeLongCalculation(inputValue), 5000D);
```

La creazione di un task può essere realizzata tramite l'oggetto `TaskFactory`, a cui possiamo accedere tramite la proprietà statica `Task.Factory` e, in particolare, sfruttando il metodo `StartNew`. Come possiamo notare nell'[esempio 8.14](#), si tratta di un metodo estremamente versatile, grazie al quale possiamo creare istanze di `Task` che eseguono procedure o funzioni, con o senza parametri in ingresso.

I task creati in questo modo vengono automaticamente schedulati per l'esecuzione e quindi avviati non appena possibile. In alternativa, possiamo avvalerci del costruttore per generare un'istanza della classe `Task` e, successivamente, chiamare il metodo `Start`.

Esempio 8.15

```
// Creazione esplicita di un Task
var resultTask = new Task((inputValue) =>
    PerformSomeLongCalulation(inputValue), 5000D);

// Esecuzione
resultTask.Start();
```

Il codice dell'[esempio 8.15](#) risulta utile quando vogliamo limitarci a istanziare un oggetto `Task` senza procedere immediatamente all'esecuzione, magari perché la nostra intenzione è quella di passarlo come parametro a un metodo. Negli altri casi, l'utilizzo di `TaskFactory` comporta le prestazioni migliori.

Quando creiamo un task passando un argomento di tipo `Func`, ossia una funzione, l'oggetto che in realtà viene costruito è di tipo `Task<Result>`, una classe che deriva da `Task` e che espone la proprietà `Result`, tramite la quale possiamo recuperare il risultato dell'invocazione, come nell'[esempio 8.16](#).

Esempio 8.16

```
var resultTask = Task.Factory.StartNew((inputValue) =>
    PerformSomeLongCalulation(inputValue), 5000D);

// .. altro codice qui ..

// Determinazione del risultato
Console.WriteLine("Il risultato è: {0}", resultTask.Result);
```

Utilizzando questa proprietà, attiviamo automaticamente la sincronizzazione del task con il thread chiamante, che rimane bloccato finché il risultato non viene reso disponibile, ossia fino al termine dell'elaborazione asincrona.

La proprietà `Result` è molto comoda per recuperare il risultato dell'esecuzione di un task perché ne consente anche implicitamente la sincronizzazione. Tuttavia, bisogna prestare grande attenzione al suo utilizzo perché non ci viene permesso di specificare un timeout.

Pertanto, accedere in lettura a un task bloccato o non ancora avviato, può comportare il blocco del thread chiamante.

In generale, quando abbiamo la necessità di attendere che l'esecuzione di uno o più task sia conclusa, possiamo utilizzare uno dei tre metodi riassunti nella [Tabella 8.2](#).

Tabella 8.2 – Metodi di attesa della classe Task

Nome	Descrizione
Wait	Metodo di istanza tramite il quale possiamo attendere la conclusione del task corrente.
WaitAll	Metodo statico esposto dalla classe Task, che accetta in ingresso un array di task da sincronizzare e ritorna il controllo al thread chiamante quando tutti hanno terminato l'esecuzione.
WaitAny	Metodo statico esposto dalla classe Task, che accetta in ingresso un array di task da sincronizzare e ritorna il controllo al thread chiamante quando almeno uno ha terminato l'esecuzione.

Essi consentono anche di specificare un eventuale timeout, oltre il quale il controllo viene comunque restituito al thread chiamante. L'[esempio 8.17](#) mostra diversi utilizzi di questi metodi.

Esempio 8.17

```
// Attesa senza timeout
var myTask = Task.Factory.StartNew(SomeMethod); myTask.Wait();

// Attesa con timeout di un secondo
myTask = Task.Factory.StartNew(SomeMethod); myTask.Wait(1000);

// Attesa conclusione di uno tra myTask e anotherTask
myTask = Task.Factory.StartNew(SomeMethod);
var anotherTask = Task.Factory.StartNew(SomeMethod);
Task.WaitAny(myTask, anotherTask);

// Attesa conclusione di una lista di Task, con timeout di 2 secondi
List<Task> taskList = GetTaskList();
Task.WaitAll(taskList.ToArray(), 2000);
```

Essi, in generale, non devono essere utilizzati solo nei casi in cui abbiamo la necessità di attendere la conclusione di un task, ma anche quando vogliamo essere sicuri che l'esecuzione sia andata a buon fine. Quando un task solleva

internamente un'eccezione, infatti, quest'ultima viene notificata al chiamante solo in caso di invocazione dei metodi `Wait`, `WaitAll` o `WaitMany`, o di accesso alla proprietà `Result`.

Esempio 8.18

```
var problematicTask = Task.Factory.StartNew(() =>
{
    throw new ApplicationException("Errore!");
});

try
{
    problematicTask.Wait();
}
catch (AggregateException ex)
{
    Console.WriteLine("Il task ha sollevato la seguente eccezione:");
    Console.WriteLine(ex.InnerException);
}
```

Nel caso dell'[esempio 8.18](#), l'eccezione sollevata da `problematicTask` viene rilevata solo nel momento in cui richiamiamo il metodo `wait` ed è incapsulata come `inner exception` all'interno di una `AggregateException`.

Composizione di task

Una caratteristica della classe `Task`, che spesso risulta estremamente comoda, è la possibilità di combinare l'esecuzione di più delegate, utilizzando il metodo `ContinueWith`.

Esempio 8.19

```
var compositeTask = Task.Factory.StartNew(() =>
{
    Thread.Sleep(1000);
    Console.WriteLine("Primo task");
}).ContinueWith((task) =>
{
    Thread.Sleep(1000);
    Console.WriteLine("Secondo task");
});

// Accodamento di una funzione
var resultTask = compositeTask.ContinueWith((task) =>
{
    string result = "Funzione del terzo task";
    Console.WriteLine(result);
});
```

```
        return result;
    });

    Console.WriteLine("Il risultato è: {0}", resultTask.Result);
```

Il codice dell'[esempio 8.19](#) utilizza questa funzionalità per concatenare tre differenti task, i primi due contenenti procedure e il terzo, invece, relativo a una funzione, recuperando al termine il risultato di quest'ultima. Nel caso vogliamo attendere la conclusione di molteplici elaborazioni prima di procedere all'esecuzione di una nuova istanza di Task, possiamo utilizzare i metodi `ContinueWhenAll` o `ContinueWhenAny` dell'oggetto `TaskFactory`, come nell'[esempio 8.20](#). Essi accettano in ingresso un array di istanze di Task da monitorare e il delegate da avviare quando, rispettivamente, tutte o almeno una di queste termina con successo.

Esempio 8.20

```
List<Task> taskList = GetTaskList();

// Task da eseguire al termine di tutti quelli contenuti in tasklist
var finalTask = Task.Factory.ContinueWhenAll(
    taskList.ToArray(),
    (tl) => Console.WriteLine("Tutti i task completati con successo"));
```

Nested task e child task

Il delegate fornito come parametro al metodo `TaskFactory.StartTask` o al costruttore di `Task` può, ovviamente, essere arbitrariamente complesso e magari istanziare e avviare al suo interno ulteriori task, come nell'[esempio 8.21](#).

Esempio 8.21

```
var outerTask = Task.Factory.StartNew(() =>
{
    var innerTask = Task.Factory.StartNew(() =>
    {
        Thread.Sleep(2000);
        Console.WriteLine("Nested task");
    });
    Console.WriteLine("Outer Task");
});

outerTask.Wait();
Console.WriteLine("outerTask terminato");
```

Un'istanza, come `innerTask` nell'[esempio 8.21](#), è chiamata **nested task** (task annidato) e ha la caratteristica di possedere un ciclo di vita proprio, del tutto indipendente da quello del task da cui è stata generata. Pertanto, l'invocazione al metodo `outerTask.Wait` attende il termine del solo `outerTask`. In fase di costruzione però, possiamo specificare la volontà di sincronizzare `innerTask` e `outerTask`, usando il parametro opzionale `TaskCreationOptions.AttachedToParent`.

Esempio 8.22

```
var outerTask = Task.Factory.StartNew(() =>
{
    var innerTask = Task.Factory.StartNew(() =>
    {
        Thread.Sleep(2000);
        Console.WriteLine("Nested task");
    }, TaskCreationOptions.AttachedToParent);

    Console.WriteLine("Outer Task");
});

outerTask.Wait();
Console.WriteLine("outerTask terminato");
```

Nel caso dell'[esempio 8.22](#), `innerTask` è denominato **child task** (task figlio) e presenta un comportamento sensibilmente diverso rispetto al precedente, dato che i due task non sono più indipendenti: `outerTask`, infatti, riceve le eventuali notifiche delle eccezioni da `innerTask` e, comunque, ne attende il completamento prima di chiudersi, producendo su console il risultato mostrato nella [Figura 8.3](#).

```
C:\Program Files\dotnet\dotnet.exe
Outer Task
Nested task
outerTask terminato
```

Figura 8.3 – Risultato dell’utilizzo dei child task.

Tramite la classe `Task`, in conclusione, siamo in grado di eseguire in parallelo procedure e funzioni, sfruttando al massimo i core a disposizione, ma senza doverci preoccupare dei dettagli hardware. La Task Parallel Library e, come vedremo nel prossimo capitolo, PLINQ, sicuramente costituiscono un validissimo aiuto nella realizzazione di codice parallelo e asincrono.

A partire da C# 6, però, abbiamo a disposizione un nuovo costrutto che semplifica ulteriormente il nostro lavoro. Sarà l’argomento delle prossime pagine.

[Programmazione asincrona con `async` e `await`](#)

Finora abbiamo visto diversi sistemi per realizzare codice multithread, in grado cioè di essere eseguito su più thread contemporaneamente. Un particolare effetto dello sviluppo multithread è costituito dall’esecuzione asincrona di un metodo, cioè in cui il thread che effettua la chiamata non resta bloccato in attesa del completamento del metodo invocato.

I vantaggi della programmazione asincrona si notano soprattutto nell’ambito di applicazioni UWP o WPF, dato che in questi casi il thread chiamante è quasi sempre quello dell’interfaccia utente: sono metodi che durano a lungo, se non sono eseguiti in maniera asincrona, bloccano l’interfaccia e quindi l’applicazione non è più

responsiva, mentre ciò non accade nel caso di esecuzione asincrona. Tuttavia, anche in altri ambiti, quali, per esempio, quello delle applicazioni web, lo sviluppo asincrono presenta immensi vantaggi, perché consente in ogni caso di liberare un thread che può essere sfruttato dal server per servire un'altra richiesta.

Nonostante il modello dei `Task` visto finora sia sicuramente uno dei sistemi più avanzati e intuitivi per scrivere codice asincrono, è innegabile che il codice risultante sia sicuramente più complesso e meno leggibile. Ciò può rappresentare un problema quando la logica da implementare è complicata e, per certi versi, ha sempre rappresentato una delle difficoltà principali all'adozione della programmazione asincrona. C# 6 semplifica di molto questo scenario grazie all'introduzione di due nuove parole chiave, `async` e `await`. Cerchiamo di chiarirne il funzionamento con un esempio.

Esempio 8.23

```
public static void Download()
{
    Console.WriteLine("---- Download ----");

    WebRequest client = HttpWebRequest.Create("http://www.google.com");
    var response = client.GetResponse();

    using (var reader = new StreamReader(response.GetResponseStream()))
    {
        // esecuzione sincrona della richiesta
        var result = reader.ReadToEnd();

        Console.WriteLine(result);
    }
}
```

Il codice dell'[esempio 8.23](#) sfrutta la classe `WebRequest` e, successivamente, `StreamReader`, per recuperare il contenuto dell'URL specificato e, fintanto che il risultato non viene recuperato, il thread resta bloccato e in attesa. Sfruttando `async` e `await` possiamo realizzarne la versione sincrona in maniera estremamente semplice.

Esempio 8.24

```
public static async Task DownloadAsync()
```

```

{
    Console.WriteLine("---- DownloadAsync ----");

    WebRequest client = HttpWebRequest.Create("http://www.google.com");
    var response = client.GetResponse();

    using (var reader = new StreamReader(response.GetResponseStream()))
    {
        // esecuzione asincrona della richiesta
        var result = await reader.ReadToEndAsync();

        Console.WriteLine(result);
    }
}

```

Questa nuova versione del metodo ha richiesto intanto di specificare la keyword `async` nella sua dichiarazione, mentre il tipo restituito è variato da `void` a `Task`. Si tratta di una direttiva che indica al compilatore la nostra intenzione di gestire all'interno di questo metodo delle chiamate asincrone. Il passo successivo è stato quello di utilizzare il metodo `ReadToEndAsync`, che rappresenta la variante asincrona di `ReadToEnd`: esso, infatti, restituisce un oggetto di tipo `Task<string>` ed esegue l'operazione di download in un altro thread.

A partire dal .NET Framework 4.5, la maggior parte dei metodi la cui durata può essere potenzialmente “lunga” (oltre i 50ms) possiede una variante asincrona. Anche l'aggiunta di riferimenti a web service esterni consente di generare metodi asincroni. Se, per un particolare caso, il metodo asincrono non risultasse disponibile, sarebbe sufficiente eseguirlo all'interno di un `Task` per poter sfruttare comunque la tecnica che abbiamo mostrato. In .NET Standard, pertanto, spesso e volentieri esistono solo le rispettive varianti asincrone.

In condizioni normali, dovremmo in qualche modo gestire il callback su questo oggetto per recuperare l'informazione prelevata dalla rete; grazie alla parola chiave `await`, invece, è il compilatore stesso a preoccuparsi di iniettare tutto il codice necessario, e noi possiamo semplicemente limitarci a prelevare il risultato dell'esecuzione asincrona e assegnarlo alla variabile `result` di tipo `string`, senza curarci del fatto che, in realtà, il metodo invocato restituisca un `Task`.

Il vantaggio, ovviamente, è che pur sfruttando le potenzialità e le caratteristiche della programmazione asincrona, il codice che siamo chiamati a scrivere è assolutamente analogo alla versione sincrona dell'[esempio 8.23](#).

Eseguire operazioni in parallelo con async e await

L'estrema comodità di `async` e `await` si nota, in modo particolare, nel momento in cui dobbiamo eseguire operazioni asincrone in cascata, come nel caso dell'[esempio 8.25](#).

Esempio 8.25

```
public static async Task SequentialOperations()
{
    Console.WriteLine("---- SequentialOperations ----");

    HttpClient client = new HttpClient();

    var firstResult = await client.GetStringAsync("http://www.google.com");
    Console.WriteLine(firstResult);

    var secondResult = await client.GetStringAsync("http://www.yahoo.com");
    Console.WriteLine(secondResult);

    var thirdResult = await client.GetStringAsync("http://www.bing.com");
    Console.WriteLine(thirdResult);
}
```

In questo caso, per esempio, abbiamo concatenato tre invocazioni, ognuna delle quali necessita del risultato della precedente per essere eseguita. Scrivere manualmente il codice sincrono analogo porterebbe sicuramente a un risultato molto meno leggibile e suscettibile di errori. Ogni volta che utilizziamo `await` in un'operazione asincrona, il flusso del metodo chiamante viene interrotto per attendere la disponibilità del risultato. Ciò impone che le chiamate asincrone siano eseguite tutte in sequenza. Quando queste chiamate sono logicamente indipendenti le une dalle altre, invece, può aver senso ristrutturare il codice come nell'[esempio 8.26](#), in modo che vengano avviate in parallelo.

Esempio 8.26

```
public static async Task ParallelOperations()
{
    Console.WriteLine("---- ParallelOperations ----");

    HttpClient client = new HttpClient();

    var tasks = new List<Task<string>>()
    {
        client.GetStringAsync("http://www.google.com"),
    }
```

```
client.GetStringAsync("http://www.yahoo.com"),
client.GetStringAsync("http://www.bing.com")
};

await Task.WhenAll(tasks);

foreach (var task in tasks)
{
    Console.WriteLine(task.Result);
}
}
```

In questo caso, abbiamo evitato di utilizzare `await` in corrispondenza di ogni singola chiamata, memorizzando le istanze dei `Task<string>` restituite all'interno della lista `tasks`. Il risultato è che il flusso del codice non viene più interrotto per attendere i risultati e le operazioni vengono effettivamente avviate in parallelo. Per recuperare, poi, i risultati, dobbiamo attendere il completamento di tutte le chiamate asincrone, effettuando un `await` sul metodo `whenAll` della classe, come abbiamo mostrato nel codice.

Realizzare metodi asincroni

Negli esempi precedenti abbiamo mostrato come trasformare un metodo sincrono in uno asincrono: questa operazione ha comportato, tra le varie modifiche, la trasformazione del tipo restituito dal metodo `DownloadAsync` da `void` a `Task`. In generale, non si tratta di un'operazione necessaria, visto che possiamo applicare `async` anche a un metodo che ritorna un `void`, ma lo è nel momento in cui vogliamo consentire a chi utilizzerà `DownloadAsync` di effettuare, a sua volta, un `await`. Cercando di riassumere le modalità con cui dichiarare un metodo asincrono, possiamo fare riferimento ai seguenti punti:

- ❑ Se dichiariamo una funzione `async void`, essa sarà in grado di eseguire metodi asincroni al suo interno, ma il suo chiamante non sarà in grado di effettuarne l'`await`. Si dice, allora, che la sua esecuzione sarà di tipo “fire and forget”;
- ❑ Un metodo di tipo `async Task` è un metodo in grado di invocare operazioni asincrone; la differenza rispetto al caso precedente è che il chiamante può effettuare l'`await` e attenderne il completamento. Nonostante questo metodo restituisca un'istanza di `Task`, a conti fatti non c'è nessun dato da ritornare: il `Task`, infatti, è utilizzato internamente solo per la sincronizzazione;

- ❑ Un metodo di tipo `async Task<T>` è una funzione che possiede tutte le caratteristiche del punto precedente, ma che è in grado di restituire un oggetto di tipo `T`.

Un'operazione piuttosto comune da compiere è quella di realizzare una versione asincrona di un metodo. Abbiamo visto, all'inizio di questa sezione, che se gli oggetti che utilizziamo espongono a loro volta delle versioni asincrone, la trasformazione è piuttosto semplice. Quando invece questi metodi non esistono, possiamo sfruttare un `Task` come nell'[esempio 8.27](#).

Esempio 8.27

```
public static Task<string> SomeMethodAsync()
{
    return Task.Run(() => SomeMethod());
}

public static string SomeMethod()
{
    // codice qui...
    Thread.Sleep(3000);

    return "test";
}
```

L'[esempio 8.27](#) contiene un metodo `SomeMethod` che restituisce una `string` e non è pensato per l'utilizzo asincrono. Il successivo `SomeMethodAsync` ne consente l'esecuzione asincrona invocandolo all'interno di un `Task`. Come possiamo notare, non è necessario che questo metodo sia marcato come `async`, dato che internamente non ha la necessità di effettuare l'`await` di alcuna chiamata. Nonostante questo, visto che comunque restituisce un `Task`, può essere invocato in modalità asincrona all'interno di un metodo chiamante (`Execute`, nel nostro caso).

Finora, parlando di multithreading e parallelismo, abbiamo trascurato un paio di aspetti di estrema importanza: l'accesso alle risorse condivise e la gestione della concorrenza, che rappresentano fattori chiave in applicazioni di questo tipo. Questi due aspetti saranno argomento del prossimo paragrafo.

Task per esecuzioni sincrone

I `Task` rappresentano una promessa rispetto al fatto che una determinata operazione verrà eseguita in modo asincrono e che l'operazione riuscirà a ritornare al chiamante, se gestita in modo corretto come mostrato negli esempi precedenti. Ci sono diverse casistiche in cui, però, non sempre i metodi marcati con la parola chiave `async` hanno necessità di essere eseguiti in modalità asincrona.

Esempio 8.28

```
public async Task WriteAsync(byte value)
{
    if (_bufferedCount == _buffer.Length)
    {
        await FlushAsync();
    }

    _buffer[_bufferedCount++] = value;
}
```

Come si può notare dall'[esempio 8.28](#), infatti, il metodo `WriteAsync` è marcato come asincrono perché una chiamata al suo interno è asincrona: questa chiamata, però, viene eseguita pochissime volte, solamente nel caso in cui il buffer ha raggiunto la lunghezza massima consentita per la scrittura. Questo rappresenta perciò uno spreco: l'oggetto `Task` che viene usato per cattura il contesto asincrono e ritornare il risultato dell'operazione al chiamante, alloca in realtà una state machine piuttosto complessa per tenere traccia dello stato dell'esecuzione per cui, in scenari in cui l'accesso a `WriteAsync` è frequente, si possono verificare troppe allocazioni e, potenzialmente, `OutOfMemoryException` che potrebbero causare l'arresto della nostra applicazione.

Per risolvere questo problema, a partire da .NET Core 2 è stato introdotto l'oggetto `ValueTask`, una struttura che è in grado di fare da contenitore per due oggetti: il primo, rappresentato dallo stato dell'operazione sincrona – che verrà eseguita la maggior parte delle volte – e, il secondo, rappresentato dallo stato dell'operazione asincrona, ovvero l'oggetto `Task<T>` classico. L'[esempio 8.28](#) può essere riscritto come mostrato nell'[esempio 8.29](#) qui di seguito.

Esempio 8.29

```
public async ValueTask WriteAsync(byte value)
```

```
{
    if (_bufferedCount == _buffer.Length)
    {
        await FlushAsync();
    }
    _buffer[_bufferedCount++] = value;
}
```

È di fondamentale importanza però non abusare dei `valueTask`: nel caso in cui le possibilità di esecuzioni asincrone aumentino proporzionalmente con le invocazioni del metodo stesso, diventa sconveniente perché andrebbe ad allocare in memoria molti più oggetti di `Task<T>` stesso che il Garbage Collector, in scenari di forte I/O, non sarebbe in grado di gestire.

Entrypoint asincroni

Per quanto riguarda le applicazioni console e anche per altre determinate tipologie di applicazioni (inclusi mondi non .NET), l'entrypoint applicativo risiede nel metodo `Main` della classe `Program`. Questo metodo viene invocato automaticamente all'avvio dell'applicazione per eseguire il codice che è stato scritto ed è definito da una serie di overload ben definiti (con o senza tipo di ritorno e parametri) e non modificabili, pertanto, qualora dovessimo effettuare ed aspettare chiamate a metodi asincroni, come abbiamo visto nei paragrafi precedenti, dovremo andare di workaround:

Esempio 8.30

```
static int Main()
{
    return MyAsyncMethod().GetAwaiter().GetResult();
}
```

Come si può notare, il workaround mostrato nell'[esempio 8.30](#) non è molto leggibile e, di fatto, non sta davvero eseguendo il metodo in modalità asincrona, ma, anzi, sta aspettandone il completamento in modalità sincrona. A partire da C# 7.1 sono state introdotte nuove firme per il metodo `Main` che permettono di lavorare con i costrutti `async/await` che abbiamo avuto modo di vedere precedentemente:

Esempio 8.31

```
static async Task<int> Main()
{
    return await MyAsyncMethod();
}
```

Nell'esempio si nota, infatti, come tutta la sintassi sia più leggibile, con codice che viene eseguito in modalità asincrona end-to-end, dall'entrypoint fino al termine dell'esecuzione dell'applicazione stessa, permettendo inoltre l'uscita con un codice di errore wrappato all'interno del `Task`.

Concorrenza e thread safety

Nelle pagine precedenti, e precisamente nell'ambito dell'[esempio 8.3](#), ci siamo imbattuti in una problematica tipica delle applicazioni multithread: l'accesso non regolamentato da parte di thread concorrenti a risorse condivise, può dar luogo a comportamenti anomali denominati **race condition**, in cui l'effettivo risultato dell'algoritmo dipende dalla tempistica con cui i thread evolvono.

In generale, ogni volta che invochiamo metodi e proprietà di un oggetto in uno scenario parallelo, dobbiamo interrogarci sulla sua capacità di gestire accessi contemporanei da parte di più thread o, per meglio dire, sulla sua **thread safety**. .NET garantisce, per ognuno degli oggetti definiti all'interno della Base Class Library, la thread safety di tutti i membri statici. I membri di istanza, invece, non sono in generale thread safe, salvo diverse specifiche nella documentazione.

Sincronizzare l'accesso alle risorse

Cerchiamo ora di capire in che modo le race condition possono minare la stabilità del nostro codice e, per farlo, consideriamo l'[esempio 8.32](#), in cui due task accedono alla medesima collezione, uno leggendo un elemento, l'altro effettuandone la rimozione.

Esempio 8.32

```
var myList = new List<string>();
```

```

myList.Add("Elemento di test");

var firstTask = Task.Factory.StartNew(() =>
{
    if (myList.Count > 0)
    {
        // race condition!
        Console.WriteLine(myList[0]);
    }
});

var secondTask = Task.Factory.StartNew(() =>
{
    if (myList.Count > 0)
        myList.RemoveAt(0);
});

Task.WaitAll(firstTask, secondTask);

```

Il codice riportato nell'[esempio 8.32](#) sembra formalmente corretto, dato che entrambi i task verificano che sia presente almeno un elemento all'interno della lista prima di procedere con il proprio compito. Tuttavia esiste una possibilità per cui, dopo che `firstTask` ha verificato la presenza di almeno un elemento della lista, `secondTask` riesca a rimuoverlo prima che questo sia effettivamente visualizzato sulla console, generando quindi un'eccezione a runtime; inoltre, non essendo il tipo `List<T>` thread safe, l'accesso concorrente di due thread ai suoi membri potrebbe generare errori nella valutazione di `myList.Count` o di `myList[0]`. Cosa ancora peggiore, questi malfunzionamenti non sono deterministici, dato che dipendono dalle tempistiche di esecuzione dei due task che, come abbiamo avuto modo di apprendere nel corso del capitolo, sono soggette a un elevato numero di variabili, quali il numero di CPU presenti o le condizioni di carico della macchina.

Pertanto, l'unico modo per consentire un corretto funzionamento del codice visto in precedenza, è quello di regolare l'accesso alla risorsa condivisa `myList` da parte dei due thread, come nell'[esempio 8.33](#).

Esempio 8.33

```

var syncObject = new object();

var myList = new List<string>();
myList.Add("Elemento di test");

var firstTask = Task.Factory.StartNew(() =>
{
    bool lockTaken = false;
    Monitor.Enter(syncObject, ref lockTaken);
    try

```

```

    {
        if (myList.Count > 0)
        {
            Console.WriteLine(myList[0]);
        }
    }
    finally
    {
        if (lockTaken)
            Monitor.Exit(syncObject);
    }
});
var secondTask = Task.Factory.StartNew(() =>
{
    bool lockTaken = false;
    Monitor.Enter(syncObject, ref lockTaken);
    try
    {
        if (myList.Count > 0)
            myList.RemoveAt(0);
    }
    finally
    {
        if (lockTaken)
            Monitor.Exit(syncObject);
    }
});

Task.WaitAll(firstTask, secondTask);

```

Nel codice precedente abbiamo utilizzato la classe `Monitor` per far sì che ogni task acquisisca l'accesso esclusivo a una risorsa, nel nostro caso l'oggetto `syncObject`. Questa pratica garantisce l'assenza del malfunzionamento visto in precedenza perché, in presenza di un lock esclusivo da parte di un thread, qualsiasi altra invocazione al metodo `Enter` da parte di altri thread risulta bloccata fino al rilascio del lock stesso tramite il metodo `Exit`.

Per evitare che una risorsa resti comunque bloccata in caso di eccezione, è sempre necessario invocare il metodo `Exit` all'interno di un blocco `finally`. Il flag `lockTaken` viene passato per riferimento al metodo `Enter` e, se valorizzato a `true`, indica che il lock è stato acquisito e, pertanto, è necessario procedere al suo rilascio.

Invece di usare direttamente la classe `Monitor`, è possibile sfruttare la parola chiave `lock` di C#, che produce risultati del tutto equivalenti, consentendoci però di scrivere una minore quantità di codice.

```

var firstTask = Task.Factory.StartNew(() =>
{
    lock (syncObject)

```

```
{
    if (myList.Count > 0)
    {
        Console.WriteLine(myList[0]);
    }
}
});
```

Vale la pena di notare che, ovviamente, l'uso di lock esclusivi pone un limite al parallelismo e, pertanto, risulta piuttosto penalizzante dal punto di vista delle prestazioni complessive della nostra applicazione.

Anche per questa ragione è determinante, per esempio, la presenza nel framework di collezioni che garantiscano la thread safety limitando al massimo l'utilizzo di lock, come vedremo nelle prossime pagine.

Collezioni con supporto alla concorrenza

Utilizzare le collezioni in uno scenario parallelo non è banale e anche le operazioni più semplici rischiano di creare non pochi problemi: immaginiamo di dover scorrere una `List<string>` mentre un altro thread tenta di modificarne il contenuto.

Esempio 8.34

```
List<string> myList = GetList();

// Creazione del task che scorre la collection
var firstTask = Task.Factory.StartNew(() =>
{
    foreach (string item in myList)
    {
        DoSomething(item);
    }
});

// Creazione del task che modifica la collection
var secondTask = Task.Factory.StartNew(() =>
{
    myList.Add("Task element");
});

Task.WaitAll(firstTask, secondTask);
```

Se proviamo a eseguire il codice dell'[esempio 8.34](#), noteremo che alcune volte l'applicazione si conclude regolarmente mentre in altri casi viene sollevata una

`InvalidOperationException` a causa del fatto che la collezione è stata modificata quando l'enumerazione era ancora in corso.

Riuscire a gestire questo tipo di situazioni implica, per esempio, di dover acquisire un lock esclusivo sulla collezione e di mantenerlo fino al termine del blocco `foreach`, ma si tratta di una soluzione che, come abbiamo avuto modo di vedere in precedenza, mina profondamente i vantaggi prestazionali derivanti dal parallelismo.

.NET, e in particolare le `Parallel Extensions`, contengono le definizioni di una serie di collezioni `thread safe` sia in lettura sia in scrittura, profondamente riviste nella loro struttura interna, in modo da minimizzare l'uso del lock e garantire, in questo modo, prestazioni elevate. Queste ultime fanno parte del namespace `System.Collections.Concurrent` e sono elencate nella [Tabella 8.3](#).

Tabella 8.3 – Collezioni contenute in `System.Collections.Concurrent`

Nome	Descrizione
<code>ConcurrentBag</code>	Si tratta di un contenitore generico di oggetti, in cui l'ordine non è importante e che ammette la presenza di duplicati.
<code>ConcurrentStack</code>	Implementazione <code>thread safe</code> di una collezione con accesso basato su logica LIFO (Last In First Out).
<code>ConcurrentQueue</code>	Implementazione <code>thread safe</code> di una collezione con accesso basato su logica FIFO (First In First Out).
<code>ConcurrentDictionary</code>	Rappresenta un dizionario, ossia un insieme non ordinato di coppie chiave-valore.

Esse non corrispondono esattamente alle loro controparti contenute nel namespace `System.Collections.Generic` ed espongono un set limitato di metodi e proprietà: lo scopo, infatti, non è quello di replicare tutte le funzionalità tipiche, per esempio, della classe `List<T>`, ma quello di fornire un insieme di strumenti basilari che supportino scenari in cui agiscono, in scrittura e lettura, diversi thread contemporaneamente.

Semplicemente riscrivendo il codice mostrato in precedenza, utilizzando un'istanza di `ConcurrentBag` in luogo della lista di tipo `List<string>`, l'applicazione si conclude sempre correttamente senza sollevare alcun errore ([esempio 8.35](#)).

Esempio 8.35

```
var myBag = new ConcurrentBag<string>(GetList());
// Creazione del task che scorre la collection
var firstTask = Task.Factory.StartNew(() =>
{
    foreach (string item in myBag)
    {
        DoSomething(item);
    }
});
// Creazione del task che modifica la collection
var secondTask = Task.Factory.StartNew(() =>
{
    myBag.Add("Task element");
});
Task.WaitAll(firstTask, secondTask);
```

Quando un thread utilizza il blocco `foreach` per enumerare la collezione, viene infatti generato internamente uno snapshot rappresentativo del suo contenuto nel momento in cui l'enumeratore viene istanziato, in modo che eventuali successivi inserimenti non creino problemi, pur in assenza di lock esclusivi. `ConcurrentBag` è anche in grado di gestire internamente code di elementi, differenziate a seconda del thread che li inserisce, in modo che, fintanto che un thread accede in lettura ai medesimi elementi che ha scritto, non sia necessario alcun meccanismo di sincronizzazione.

[Stream asincroni con IAsyncEnumerable](#)

Supponiamo ora di avere un metodo che aggiunge gli elementi ad una lista secondo dati che arrivano da un servizio web, come mostrato nell'esempio seguente.

Esempio 8.36

```
public static async Task<IEnumerable<int>> AddNumbersAsync()
{
    var collection = new List<int>();
    for (int i = 0; i < 20; i++)
    {
```

```
        collection.Add(await GetDataAsync(i));
    }

    return collection;
}
```

In questo caso, la lista deve essere creata in modo specifico prima dell'esecuzione del ciclo e, sebbene il codice possa essere ottimizzato tramite `Task.WhenAll` per eseguire le operazioni in parallelo, come abbiamo visto in precedenza, purtroppo non possiamo fare uso di `yield` per tenere i dati e restituirli in modo diretto, perché questo operatore non supporta le operazioni asincrone. Questo significa che non solo avremo più allocazioni del necessario, ma che anche la funzione che invocherà la `AddNumbersAsync` dovrà avere codice specifico per leggere i dati in modo asincrono.

A partire da C# 8 è stato perciò introdotto il concetto di stream asincroni: la nuova collezione di tipo `IAsyncEnumerable<T>` (contenuta nel namespace `System.Collections.Generic`) consente proprio la lettura dei dati in modalità asincrona e pertanto ci permette di riscrivere l'[esempio 8.36](#) più semplicemente sfruttando l'operatore `yield`.

Esempio 8.37

```
public static async IEnumerable<int> AddNumbersAsync()
{
    for (int i = 0; i < 20; i++)
    {
        yield return await GetDataAsync(i);
    }
}
```

Il codice risulta molto più leggibile, ma introduce due breaking change: la prima, più evidente, è che non è più richiesto l'uso di `Task` come tipo di ritorno dalla funzione perché sarà `IAsyncEnumerable` a fare da wrapper per `Task<T>` e a gestire l'esecuzione asincrona, mentre la seconda, meno evidente, è sul chiamante:

Esempio 8.38

```
await foreach (var number in AddNumbersAsync())
{
    Console.WriteLine(number);
}
```

Come si può notare dall'[esempio 8.38](#), infatti, anziché fare l'`await` di tutta la funzione `AddNumbersAsync` e quindi eseguire un controllo su tutti i suoi risultati, creando una nuova allocazione della lista di ritorno, è sufficiente fare l'`await` di tutto il blocco `foreach` che eseguirà la valutazione di volta in volta dello stream, continuando a lavorarlo in modalità asincrona, se necessario.

Conclusioni

In questo capitolo abbiamo trattato argomenti che possono essere ritenuti avanzati ma che, negli anni a venire, rappresenteranno un bagaglio indispensabile per ogni sviluppatore. Scrivere codice multithread, in grado cioè di eseguire contemporaneamente diverse operazioni sfruttando l'infrastruttura di threading del sistema operativo, consente di incrementare notevolmente le prestazioni delle nostre applicazioni. Nella prima parte del capitolo abbiamo visto quali sono gli oggetti messi a disposizione dal framework per la gestione dei thread, che possono essere istanziati esplicitamente o prelevati da un pool gestito dal CLR.

Successivamente abbiamo introdotto il concetto di parallelismo, ossia la capacità di un'applicazione multithread di assegnare i task creati dallo sviluppatore alle unità di calcolo (core e/o CPU) messe a disposizione dall'hardware. Le Parallel Extensions sono una porzione di librerie, introdotte dal .NET Framework 4.0 e disponibili anche in .NET Core, il cui compito è proprio quello di fornire strumenti evoluti per realizzare codice in grado di essere eseguito in parallelo. Tramite la classe `Task` possiamo facilmente eseguire delegate in maniera asincrona, ottimizzandone la schedulazione in base al numero di unità di calcolo presenti nel sistema. Scrivere applicazioni di questo tipo non è però semplice, dato che codice non thread safe e race condition possono minarne la stabilità.

Nell'ultima parte del capitolo ci siamo occupati proprio di queste problematiche, mostrando quando è necessario utilizzare sistemi di sincronizzazione per evitare bug legati alla concorrenza, e come le collezioni concorrenti possano garantire, allo stesso tempo, thread safety ed elevate prestazioni, grazie al limitato utilizzo dei lock esclusivi.

A partire dal prossimo capitolo cambieremo radicalmente argomento, passando a un tema assolutamente centrale nell'ambito dello sviluppo di applicazioni reali: l'accesso ai dati.

Introduzione a LINQ

Fino a questo punto abbiamo analizzato la struttura e le potenzialità del linguaggio C#. Da questo capitolo cominciamo a mettere in pratica quanto abbiamo imparato, sfruttando uno degli autentici gioielli di .NET: **LINQ**.

Una delle più grandi feature di .NET è senza dubbio LINQ (Language INtegrated Query), ovvero una tecnologia realizzata per permettere la ricerca all'interno di qualunque struttura dati, sfruttando direttamente il linguaggio. Grazie a questa tecnologia, possiamo scrivere query tipizzate per ricercare dati in una lista di oggetti, nodi in una struttura XML o tabelle in un database SQL.

Inoltre, grazie alla tipizzazione, Visual Studio può fornire l'intellisense e il compilatore può eseguire il controllo immediatamente, individuando gli errori senza la necessità di eseguire l'applicazione. Queste caratteristiche incrementano notevolmente la nostra produttività e ci permettono di scrivere codice vicino allo stile funzionale.

Nel corso del capitolo vedremo come LINQ permetta di risparmiare una enorme quantità di codice, rendendolo anche più leggibile (qualora non ne abusiamo). Grazie ai vari metodi messi a disposizione, possiamo svolgere in maniera molto più semplice ed elegante non solo le ricerche ma anche molte altre operazioni.

LINQ si basa su caratteristiche del .NET quali lambda expression, extension method, anonymous type e object, collection initializer e array initializer, tutte funzionalità che sono state ampiamente trattate nei precedenti capitoli. È importante dire che molte di queste funzionalità sono state create proprio per rendere il più semplice possibile l'utilizzo di LINQ.

I perché di LINQ

Qualunque applicazione deve manipolare dati. Per loro natura, questi dati possono essere memorizzati in diversi formati e in diversi tipi di storage. Molte applicazioni salvano i dati su un database, altre utilizzano file XML, mentre altre ancora sono astratte dalla persistenza dei dati tramite web service. In più, tutte queste tipologie di applicazione spesso trattano anche i dati in memoria e hanno bisogno di compiere operazioni su questi ultimi.

Il vero problema, in queste applicazioni, è che dobbiamo usare API differenti per ogni tipo di dato da trattare. Alla fine però quello che dobbiamo eseguire è semplicemente un accesso ai dati. Partendo da questo presupposto, Microsoft ha lavorato a una piattaforma che permettesse di effettuare query allo stesso modo, qualunque fosse il tipo di storage sottostante: XML, database, web service, oggetti in memoria. Questa piattaforma è LINQ.

Un'altra idea che ha guidato Microsoft nella creazione di LINQ è l'integrazione nei linguaggi di programmazione. In questo modo il compilatore è in grado di controllare le query scritte e di restituire eventuali errori già in fase di compilazione.

Il risultato è che il codice di interrogazione diventa **univoco** e ci permette di ignorare le diversità dello storage dei dati. La [Figura 9.1](#) mostra come LINQ ottenga questo risultato.

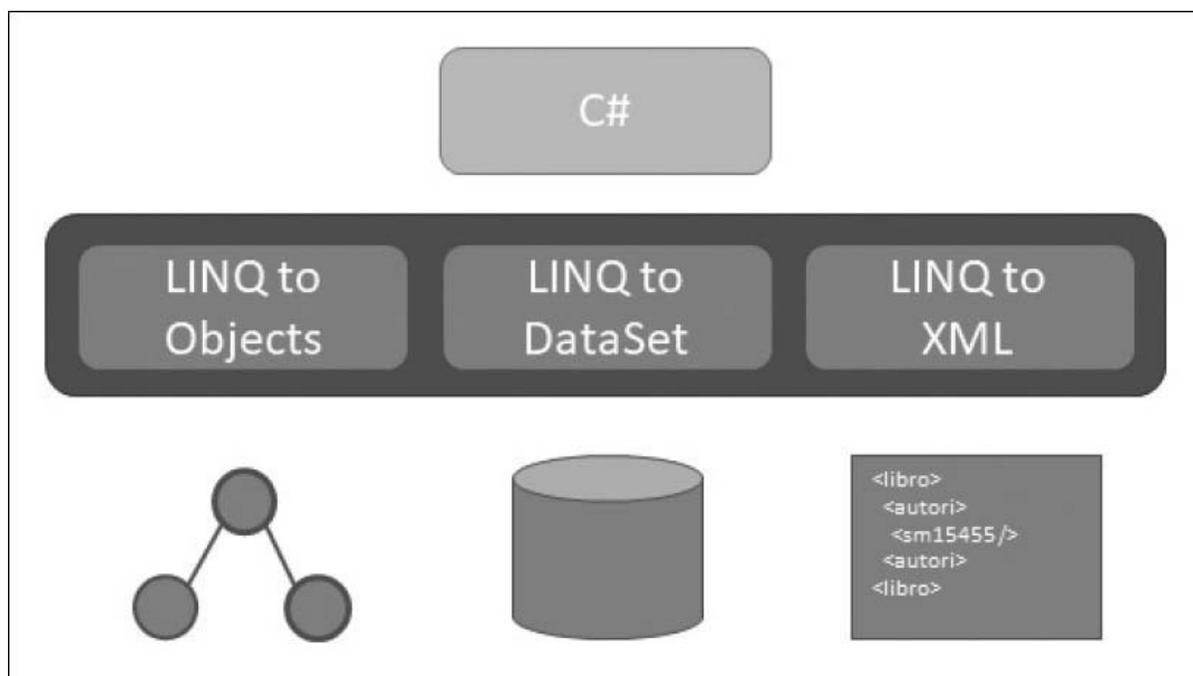


Figura 9.1 – La struttura di LINQ.

LINQ ha diversi *flavor*. LINQ to Objects permette di eseguire query su oggetti in memoria, LINQ to Dataset abilita query su oggetti DataSet e, infine, LINQ to XML permette di eseguire query verso strutture XML.

I flavour di LINQ sono trasparenti per il programmatore. Anche se alcuni aggiungono ulteriori funzionalità, la base è comune a tutti. Per esempio nel [Capitolo 11](#), in cui parleremo di Entity Framework, vedremo come questo prodotto aggiunge metodi asincroni a quelli sincroni esistenti. Nel corso di questo capitolo analizzeremo solo LINQ to Objects.

Prima di vedere LINQ in azione, diamo uno sguardo all'interno di questa tecnologia, per capire come funziona.

Come funziona LINQ

Alla base di LINQ vi è la classe `Enumerable`, situata nel namespace `System.Linq`. Questa classe contiene una serie di extension method che estendono l'interfaccia `IEnumerable<T>`, aggiungendo i metodi che vengono utilizzati nelle query. Poiché nel .NET ogni lista generica implementa, direttamente o indirettamente, l'interfaccia `IEnumerable<T>`, automaticamente ogni lista generica è interrogabile via LINQ. Inoltre, la classe `Enumerable` contiene anche i metodi che estendono l'interfaccia `IEnumerable` non generica, consentendo di ottenere da quest'ultima una versione generica pronta per essere interrogata. Grazie a questo design delle classi e senza alcuno sforzo, abbiamo a disposizione il motore di interrogazione, integrato direttamente nel linguaggio.

Inoltre, molti dei metodi presenti nella classe `Enumerable` sono funzioni che, a loro volta, restituiscono un oggetto `IEnumerable<T>` e quindi rendono possibile la concatenazione di più metodi nella stessa query. Il funzionamento base di LINQ sta tutto qui; non ci sono altre cose da capire.

Se poi vogliamo entrare nel merito di alcune specializzazioni di LINQ, come LINQ to SQL e LINQ to Entities, cominciamo ad avere a che fare con Expression Tree e LINQ Provider. In poche parole, prima la query viene convertita in un albero di funzioni e poi il LINQ Provider si preoccupa di trasformare questa struttura ad albero in un'entità che possa interrogare la sorgente dati per cui il provider è realizzato. Nel caso di LINQ to SQL l'albero viene trasformato in un comando SQL mentre, nel caso di LINQ to Entities, l'albero viene trasformato in un linguaggio di interrogazione dello schema logico (vedi [Capitolo 11](#)).

Potenzialmente, grazie a questa struttura, possiamo interrogare sorgenti dati di ogni tipo come web service, motori di ricerca, Active Directory, NHibernate, JSON, Excel e molto altro ancora. Molti di questi provider sono stati già realizzati da terze parti e sono facilmente rintracciabili sul web.

Sul sito di LINQItalia.com è disponibile un ottimo articolo che spiega come creare un LINQ provider personalizzato per effettuare query verso il motore di ricerca Live Search. Lo trovate all'URL: <http://aspit.co/aei>.

Ora che abbiamo compreso il funzionamento interno di LINQ, possiamo cominciare a scrivere query. Prima però, diamo un veloce sguardo alla struttura degli oggetti su cui andremo a eseguire le interrogazioni.

[Introduzione all'esempio del capitolo](#)

Nel corso del capitolo, il modello a oggetti su cui eseguiamo le query prevede una classe Ordine, che contiene una proprietà Dettagli con la lista dei dettagli dell'ordine. La lista degli ordini è valorizzata come nell'[esempio 9.1](#).

Esempio 9.1

```
List<Ordine> ordini = new List<Ordine>()
{
    new Ordine { Id = 1, Data = DateTime.Now,
        Dettagli = new List<DettaglioOrdine>()
        {
            new DettaglioOrdine { Id = 1, Prezzo = 50, Quantita = 10 },
            new DettaglioOrdine { Id = 2, Prezzo = 30, Quantita = 1 },
            new DettaglioOrdine { Id = 3, Prezzo = 12, Quantita = 13 }
        }.ToArray()
    },
    new Ordine { Id = 2, Data = DateTime.Now.AddDays(-10),
        Dettagli = new List<DettaglioOrdine>()
        {
            new DettaglioOrdine { Id = 4, Prezzo = 32, Quantita = 18 },
            new DettaglioOrdine { Id = 5, Prezzo = 15, Quantita = 4 }
        }.ToArray()
    },
    new Ordine { Id = 3, Data = DateTime.Now.AddDays(-5),
        Dettagli = new List<DettaglioOrdine>()
        {
            new DettaglioOrdine { Id = 6, Prezzo = 2, Quantita = 9 },
            new DettaglioOrdine { Id = 7, Prezzo = 15, Quantita = 4 },
            new DettaglioOrdine { Id = 8, Prezzo = 21, Quantita = 150 },
            new DettaglioOrdine { Id = 9, Prezzo = 1, Quantita = 400 },
            new DettaglioOrdine { Id = 10, Prezzo = 3, Quantita = 84 }
        }
    }
}
```

```

        }.ToArray()
    },
    new OrdineEx { Id = 4, IsInternational = true,
        Data = DateTime.Now.AddDays(-5),
        Dettagli = new List<DettaglioOrdine>()
        {
            new DettaglioOrdine { Id = 11, Prezzo = 2, Quantita = 30 },
            new DettaglioOrdine { Id = 12, Prezzo = 2, Quantita = 2 }
        }.ToArray()
    },
};

```

Oltre alla lista degli oggetti da interrogare, prima di eseguire una query dobbiamo anche sapere quali sono i metodi di LINQ. Nella prossima sezione ci occuperemo di questo problema.

Gli extension method di LINQ

Gli extension method messi a disposizione da LINQ sono tantissimi e sono suddivisi in categorie a seconda del loro scopo. Questi metodi sono riportati nella [Tabella 9.1](#).

Tabella 9.1 – Gli extension method di LINQ (in ordine di categoria)

Tipologia	Operatori
Aggregazione	Aggregate, Average, Count, LongCount, Max, Min, Sum
Concatenamento	Concat, Zip
Conversione	Cast, ToArray, ToDictionary, ToList, ToLookup
Elemento	DefaultIfEmpty, ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault
Generazione	Empty, Range, Repeat,
Intersezione	GroupJoin, Join
Ordinamento	OrderBy, ThenBy, OrderByDescending, ThenByDescending, Reverse
Partizionamento	Skip, SkipWhile, Take, TakeWhile
Proiezione	Select, SelectMany
Quantificazione	All, Any, Contains
Uguaglianza	SequenceEqual
Raggruppamento	GroupBy

Restrizione	OfType, Where
Insieme	Distinct, Except, Union, Intersect

Dalla tabella possiamo desumere tutte le informazioni per eseguire query nel codice: quindi, dalla prossima sezione, potremo finalmente iniziare a utilizzare LINQ.

La filosofia alla base LINQ

Per capire la filosofia che sta alla base di LINQ, è meglio partire direttamente da un esempio, in modo da rendere il tutto più semplice da comprendere. Prendiamo la seguente query.

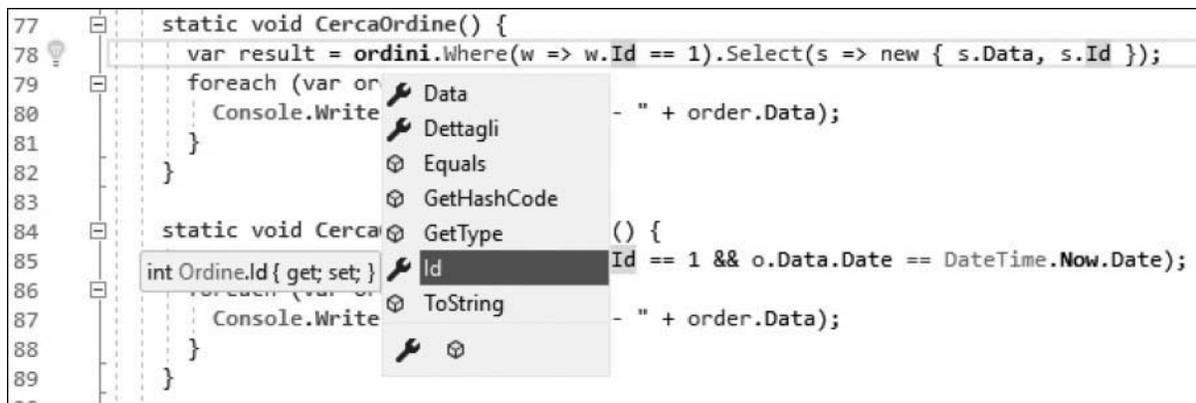
Esempio 9.2

```
var o = Ordini
    .Where(w => w.Id == 1)
    .Select(s => new { s.Data, s.Id });
```

Questa query LINQ prende la lista degli ordini, ricerca l'ordine con la proprietà `Id` uguale a 1 e ritorna un anonymous type con le sole proprietà `Data` e `Id`. Senza entrare troppo in dettaglio, vi sono alcune cose che dobbiamo approfondire.

Innanzitutto l'intellisense. Come detto nell'introduzione del capitolo, Visual Studio offre il pieno supporto a LINQ. Questo vale sia per gli extension method sia per le proprietà delle classi interrogate.

Per abilitare gli extension method nel codice, bisogna importare il namespace `System.Linq` tramite `using`.



```
77 static void CercaOrdine() {  
78     var result = ordini.Where(w => w.Id == 1).Select(s => new { s.Data, s.Id });  
79     foreach (var or  
80         Console.Write - " + order.Data);  
81     }  
82 }  
83  
84 static void Cerca  
85     int Ordine.Id { get; set; }  
86     foreach (var or  
87         Console.Write - " + order.Data);  
88 }  
89 }
```

IntelliSense popup menu items:
Data
Dettagli
Equals
GetHashCode
GetType
Id
ToString

Figura 9.2 – Intellisense in Visual Studio con C#.

Il secondo concetto sul quale dobbiamo soffermarci è la completa innovazione di LINQ in termini di modalità di ricerca. Nella query dell'[esempio 9.2](#), non abbiamo scritto **come** il codice debba effettuare la ricerca, bensì abbiamo scritto **cosa** dev'essere cercato; il vero lavoro di ricerca viene fatto dal codice degli extension method. Il fatto di scrivere codice funzionale e non imperativo rappresenta una novità per il mondo .NET al momento dell'uscita di LINQ e rimane tutt'ora una grande caratteristica.

La concatenazione di più operatori e la possibilità di crearne di personalizzati permette di ottenere, in una sola query un risultato che altrimenti richiederebbe di scrivere molte righe di codice.

Ora che abbiamo capito come scrivere una semplice query con LINQ, possiamo passare alla fase successiva e approfondire il funzionamento degli extension method.

[Anatomia di una query.](#)

Nell'[esempio 9.2](#) è stato introdotto un primo esempio di ricerca dei dati. La ricerca è stata effettuata tramite l'operatore `where`, che permette di filtrare i dati. La firma di questo metodo è, in effetti, molto più verbosa di quanto possa sembrare dal codice, come possiamo vedere nell'[esempio 9.3](#).

Esempio 9.3

```
IEnumerable<TSource> Where<TSource>(  
    this IEnumerable<TSource> source, Func<TSource, bool> predicate)
```

Innanzitutto, `where` è un metodo generico ma nell'[esempio 9.2](#) non vi è traccia del parametro. Questo è possibile grazie alla `type inference` in quanto, essendo la lista interrogata generica, il compilatore utilizza automaticamente il tipo della lista, evitandoci di doverlo scrivere esplicitamente.

Il metodo `where` accetta un solo parametro in input. Questo parametro è un predicato che rappresenta la condizione in base alla quale eseguire il filtro ed è formulato tramite una `lambda expression`. Il predicato che passiamo in input al metodo `where` possiede una firma in cui il parametro in input è l'oggetto corrente della lista e il risultato in output è un `Boolean`, che specifica se la condizione di ricerca è soddisfatta o no.

La classe `Func` rappresenta un delegato che accetta da uno a diciassette parametri tipizzati tramite `generics`. L'ultimo parametro generico rappresenta il tipo di ritorno del delegato.

Ovviamente, i concetti appena visti sono applicabili non solo al metodo `where` ma, in generale, a tutti gli `extension method`.

Finalmente abbiamo acquisito tutte le conoscenze di base indispensabili per scrivere qualunque tipo di query e quindi, da qui in avanti, esamineremo in dettaglio tutti i metodi di LINQ.

[Gli operatori di restrizione](#)

Nel paragrafo precedente abbiamo già illustrato, in maniera abbastanza dettagliata, l'operatore `where`, che è il più importante fra quelli che fanno parte della categoria in esame. Nel prossimo paragrafo utilizzeremo questo metodo con un taglio più pratico.

La `lambda expression` con cui esprimiamo il filtro è scritta in codice `managed` e, pertanto, possiamo inserire più condizioni di ricerca, come nel seguente esempio.

Esempio 9.4

```
var result = ordini
    .Where(o => o.Id == 1 && o.Data.Date == DateTime.Now.Date);
```

Molto spesso le query di ricerca sono dinamiche, nel senso che, in base a determinate condizioni, dobbiamo applicare alcuni filtri e non altri. Poiché il metodo `Where` ritorna un `IEnumerable<T>`, il risultato della chiamata è nuovamente interrogabile con un'altra chiamata a un secondo metodo `Where`, come mostra l'[esempio 9.5](#).

Esempio 9.5

```
var result = ordini.Where(o => o.Id == 1);  
result = result.Where(o => o.Data.Date == DateTime.Now.Date);
```

Il metodo `where` ha un secondo overload che si differenzia dal primo in quanto cambia la firma del predicato. Infatti, il delegato, oltre che accettare in input l'oggetto della lista, accetta anche il suo indice. Questo può tornare pratico in scenari dove, per esempio, vogliamo prendere solo gli oggetti con indice pari o dispari o, comunque, dove l'indice può essere una discriminante. Nella definizione della lambda, il secondo parametro viene separato dal primo tramite un carattere “,” (virgola) e non necessita della definizione del tipo, così come avviene già per il primo parametro, grazie sempre alla type inference. L'[esempio 9.6](#) mostra il codice che sfrutta questo overload.

Esempio 9.6

```
var result = ordini.Where((o, i) => i % 2 == 0);
```

L'altro operatore che fa parte di quelli di restrizione è `OfType`, il quale filtra gli oggetti in base ad altri criteri.

OfType

Il secondo metodo della categoria degli operatori di restrizione è `OfType`. Questo metodo serve per filtrare gli oggetti in una lista in base al loro tipo. Il modo in cui questo metodo compara il tipo degli oggetti è identico all'operatore `typeof..Is`. Questo significa che, se cerchiamo il tipo `ordine`, ci vengono restituiti tutti gli oggetti `ordine` e quelli che, direttamente o indirettamente, ereditano da questa classe. Se invece cerchiamo gli oggetti di

tipo `OrdineEx` allora viene restituito solo l'ordine con `Id` pari a 4. Se volessimo recuperare le istanze dei soli oggetti di tipo `Ordine`, dovremmo usare il metodo `where` ed effettuare manualmente il controllo sul tipo. L'[esempio 9.7](#) mostra l'utilizzo di questo metodo.

Esempio 9.7

```
var res1 = ordini.OfType<Ordine>()      // Torna tutti gli oggetti
var res2 = ordini.OfType<OrdineEx>()    // Torna solo l'ordine 4
```

La particolarità di questo metodo è che il tipo di oggetto che stiamo cercando non è specificato tramite un parametro, bensì tramite generics.

Ora che sappiamo come filtrare, passiamo a vedere come possiamo formattare l'output delle nostre query.

Gli operatori di proiezione

Gli operatori di proiezione permettono di modificare il risultato di output di una query. Quando eseguiamo una query su una lista generica, il risultato è una nuova lista generica nella quale il tipo è lo stesso della lista di input. A volte, dobbiamo modificare questo comportamento al fine di avere, in output, un oggetto più snello perché, per esempio, lo dobbiamo collegare a una griglia oppure perché dobbiamo restituire l'oggetto all'esterno e non vogliamo esporre tutte le proprietà.

Select

Il primo operatore di proiezione è `select`. Questo metodo permette di effettuare diverse trasformazioni del risultato di una query.

Supponiamo di avere un metodo che deve restituire gli ID degli ordini. Tornare l'intero oggetto non ha senso, né in termini logici né in termini prestazionali. La via migliore è restituire una lista con gli ID, come si può vedere nell'[esempio 9.8](#).

Esempio 9.8

```
var result = ordini.Select(o => o.Id);
```

In altri casi, un metodo potrebbe dover tornare degli oggetti diversi da quelli interrogati, per scopi di ottimizzazione oppure perché alcuni dati sono sensibili e non devono essere esposti all'esterno. In questi scenari possiamo riversare il contenuto della classe di input in un'altra, che è quella che viene poi restituita al chiamante. Grazie agli Object Initializers, questo può essere fatto in una sola riga di codice, come possiamo notare nell'[esempio 9.9](#).

Esempio 9.9

```
var result = ordini.Select(o =>  
    new OrdineDTO { Id = o.Id, Data = o.Data });
```

Un altro scenario che può trarre beneficio dalla modifica del risultato è la visualizzazione dei dati. Se abbiamo una griglia dove vogliamo presentare una serie di fatture senza mostrarne i dettagli, ci basta creare un oggetto in output che contenga tutte le proprietà tranne `Dettagli`. Il caso è identico a quello che abbiamo appena visto ma con la differenza che `OrdineDTO` è una classe che deve essere creata e mantenuta mentre, in questo caso, grazie agli anonymous type, possiamo creare la classe al volo e collegarla subito alla griglia. L'[esempio 9.10](#) dimostra questa tecnica in pratica.

Esempio 9.10

```
var result = ordini.Select(o => new { o.Id, o.Data });
```

Il nome delle proprietà dell'oggetto anonimo viene automaticamente ricavato dalle proprietà dell'oggetto originale. Se dobbiamo cambiare il nome delle proprietà, dobbiamo anteporre il nuovo nome seguito dal carattere `=`. Nell'[esempio 9.11](#) mostriamo come ottenere questo risultato.

Esempio 9.11

```
var result = ordini.Select(o => new { IdOrdine = o.Id,
```

```
DataOrdine = o.Data});
```

Il metodo `select` è tutto qui e pensiamo non necessiti di ulteriori spiegazioni, quindi possiamo passare al prossimo operatore di proiezione.

SelectMany

L'operatore `selectMany` serve per appiattare quegli oggetti che, altrimenti, avrebbero una struttura ad albero. Sebbene la definizione possa sembrare complicata, in realtà `selectMany` svolge una funzione molto semplice e facile da capire, una volta vista in azione. Prendiamo la query dell'[esempio 9.12](#), che torna i dettagli di un ordine.

Esempio 9.12

```
var ordine = Ordini
    .Where(o => o.Id == 1)
    .Select(o => o.Dettagli);
```

Ciò che otteniamo da questa query è una lista con un solo elemento, contenente un elenco con i dettagli dell'ordine; in pratica la query ritorna una lista di liste. Sebbene scorrere questi dati sia semplice, avere a disposizione direttamente la lista è molto più comodo. Questo è ancora più vero quando vogliamo ritornare i dettagli di tutti gli ordini.

A questo punto entra in campo il metodo `selectMany` che appiattisce tutti i dettagli in una lista unica, più semplice da scorrere. Il suo utilizzo è mostrato nell'[esempio 9.13](#).

Esempio 9.13

```
var ordine = Ordini.SelectMany(o => o.Dettagli);
```

Gli operatori di selezione sono molto importanti e il loro utilizzo è frequente durante la scrittura delle query. Fortunatamente la loro estrema semplicità ci permette di comprenderne con facilità il funzionamento. Ora passiamo agli operatori che permettono di ordinare le liste.

Gli operatori di ordinamento

Gli operatori di ordinamento hanno il semplice scopo di ordinare una lista in base a una o più proprietà. Grazie a questi metodi, non dobbiamo più creare una classe ad hoc, che implementi `IComparable` per ordinare le liste, e quindi otteniamo un notevole risparmio di codice. Vediamo ora i primi metodi di questa categoria.

OrderBy, OrderByDescending, ThenBy e ThenByDescending

I metodi `OrderBy` e `OrderByDescending` permettono di ordinare la lista, rispettivamente in maniera ascendente e discendente, in base a una proprietà degli oggetti contenuti. L'[esempio 9.14](#) mostra un primo utilizzo di questi metodi.

Esempio 9.14

```
var result = ordini.OrderBy(o => o.Data);
```

La caratteristica peculiare di questi due metodi è che il risultato non è un oggetto di tipo `IEnumerable<T>`, bensì un oggetto `IOrderedEnumerable<T>` (che eredita da `IEnumerable<T>`). Il motivo dell'esistenza di questa interfaccia è dovuto al fatto che la possibilità di ordinare per più campi non è rappresentabile in una lambda expression. Infatti, quando usiamo il metodo `where`, è facile unire due condizioni, in quanto il linguaggio già lo permette, ma nel caso dell'ordinamento non esiste nulla che possa esprimere due o più proprietà in base a cui ordinare.

Per questo motivo, sono stati introdotti due ulteriori operatori tramite l'interfaccia `IOrderedEnumerable<T>`: `ThenBy` e `ThenByDescending`. Questi metodi permettono di specificare ulteriori campi per effettuare l'ordinamento dopo una prima chiamata a `OrderBy` o `OrderByDescending`. In questo modo siamo sicuri di aver già fatto un ordinamento prima di invocare questi metodi. Il loro utilizzo è mostrato nell'[esempio 9.15](#).

Esempio 9.15

```
var result = ordini
    .OrderBy(o => o.Data)
    .ThenBy(o => o.Id);
```

In questo esempio viene effettuato l'ordinamento degli ordini prima in base alla data e all'ID.

In una query può esserci un solo metodo `OrderBy` o `OrderByDescending`. La presenza di due o più chiamate a questi metodi comporta che l'ultima chiamata è quella che vince. Questo discorso non si applica, invece, per gli altri operatori `ThenBy` o `ThenByDescending`.

Oltre agli operatori che ordinano in base a uno o più campi, abbiamo a disposizione `Reverse`, un operatore che effettua un altro tipo di ordinamento.

Reverse

L'ultimo operatore della categoria dell'ordinamento è `Reverse`. Già il nome esprime in modo chiaro lo scopo di questo metodo. Infatti, chiamando il metodo `Reverse`, otteniamo una lista, con gli oggetti in ordine completamente invertito.

Vi sono un paio di osservazioni da fare su questo metodo. Innanzitutto, se ci sono oggetti di tipo differente nella lista di partenza, il metodo va in errore. Inoltre, poiché esiste già un metodo `Reverse` per alcuni oggetti che gestiscono liste (vedi `List<T>`), in tali casi dobbiamo specificare, via `generic`, il tipo dell'oggetto della lista, altrimenti il compilatore cerca di utilizzare il metodo non generico e scatena un errore. Se invece stiamo facendo la query direttamente su un oggetto `IEnumerable<T>`, non c'è bisogno di specificare il tipo. L'[esempio 9.16](#) mostra l'utilizzo di questo metodo.

Esempio 9.16

```
var result = ordini
    .Where(o => typeof(Ordine) == o.GetType())
    .Reverse<Ordine>(); // Filtra e inverte
```

Oltre che permettere di filtrare, tornare dati in formati diversi e ordinare, LINQ permette anche di effettuare raggruppamenti. Nella prossima sezione analizzeremo questa capacità.

Gli operatori di raggruppamento

Nella categoria oggetto di questo paragrafo, esiste un solo operatore: `GroupBy`. Questo metodo permette di raggruppare i dati in base a una chiave che può essere una proprietà o l'unione di più proprietà in una sola. Per esempio, se vogliamo raggruppare le fatture per giorno, il codice da utilizzare è quello mostrato nell'[esempio 9.17](#).

Esempio 9.17

```
var result = ordini.GroupBy(o => o.Data.Date);
```

Quello che otteniamo da questa query è un oggetto di tipo `IEnumerable<IGrouping<T, K>>`, dove `T` è di tipo `DateTime` e `K` di tipo `IEnumerable<J>` con `J` di tipo `ordine`. A prima vista, questa struttura può sembrare molto complessa ma, in realtà, è più semplice di quel che possiamo immaginare. Basta pensare a una struttura ad albero, contenente gli ordini classificati in base alla loro data di emissione. A ogni data corrisponde un nodo dell'albero e, per ognuno di questi nodi, esistono tanti figli quanti sono gli ordini per quella data. Detto questo, è chiaro che il modo in cui scorriamo la lista non è lo stesso che abbiamo usato in precedenza, ma cambia in virtù del fatto che la lista è a due livelli. L'[esempio 9.18](#) mostra come cambia il modo di ciclare tra gli elementi del risultato.

Esempio 9.18

```
StringBuilder sb = new StringBuilder(); foreach (var item in result)
{
    sb.AppendLine(item.Key);
    foreach (var obj in item)
    {
        sb.AppendLine(" ordine numero " + obj.Id);
    }
}
```

Il primo ciclo scorre tutti i nodi di primo livello della struttura (le date) e per ognuno di essi va a esaminare i figli (gli ordini). Il valore della chiave di raggruppamento è contenuto nella proprietà `Key` del tipo `IGrouping<T, K>` (l'oggetto di primo livello).

`GroupBy` ha diversi overload, fra i quali uno molto importante è quello che permette di scegliere quale oggetto inserire nella lista legata alla chiave. Di default, quando passiamo al metodo `GroupBy` solo la chiave di raggruppamento, l'oggetto della lista di input viene inserito in quella di output. Tuttavia, se vogliamo modificare questo comportamento, possiamo passare un secondo parametro che specifichi cosa inserire in output. Così come per il metodo `select`, possiamo scegliere di inserire un oggetto personalizzato, come `orderedTo` oppure un anonymous type (soluzione mostrata nell'[esempio 9.19](#)).

Esempio 9.19

```
var result = ordini
    .GroupBy(o => o.Data.Date,
            o => new { o.Id, Dettagli = o.Dettagli.Count() });
StringBuilder sb = new StringBuilder();
foreach (var item in result)
{
    sb.AppendLine(item.Key);
    foreach (var obj in item)
    {
        sb.AppendLine(" ordine numero " + obj.Id + " Dettagli " +
            obj.Dettagli);
    }
}
```

In questo caso otteniamo un anonymous type con l'ID dell'ordine e il numero di dettagli per ognuno. È importante sottolineare che, pur utilizzando gli anonymous type, Visual Studio rimane in grado di offrire l'intellisense in qualunque momento.

Un'altra caratteristica che LINQ mette a nostra disposizione, è la possibilità di eseguire calcoli su una lista di valori tramite gli operatori di aggregazione.

[Gli operatori di aggregazione](#)

Gli operatori di aggregazione hanno lo scopo di calcolare un singolo valore da un insieme di dati. Questi metodi tornano utili quando vogliamo calcolare il valore massimo di un numero, contare gli elementi in una lista o altro ancora. Cominciamo analizzando gli operatori che eseguono calcoli sui dati.

[Average, Min, Max, Sum](#)

I metodi `Average`, `Min`, `Max` e `Sum` calcolano rispettivamente la media, il valore minimo, quello massimo e la somma di un particolare insieme di dati. Il modo per specificare su quale dato si debba eseguire il calcolo è sempre tramite predicato e lambda expression, come possiamo vedere nell'[esempio 9.20](#).

Esempio 9.20

```
var maxdata = ordini.Max(o => o.Data);
var mindata = ordini.Min(o => o.Data);
var avgvendite = ordini.Average(o => o.Dettagli.Count());
var fatturato = ordini.Sum(o => o.Dettagli.Sum(
    d => d.Prezzo * d.Quantita));
```

La variabile `maxdata` contiene la data dell'ultimo ordine piazzato, mentre `mindata` contiene il primo; `avgvendite` indica la media dei dettagli per ordine, mentre `fatturato` contiene il totale di tutti gli ordini.

Il modo di calcolare il fatturato è una delle funzioni più potenti di LINQ. Infatti, non c'è una sola query, ma ve ne sono due innestate. Per calcolare il risultato, LINQ scorre la lista degli ordini e per ogni ordine scorre i dettagli, sommando il prezzo moltiplicato per la quantità tramite `sum`. Infine, questi valori vengono sommati nuovamente tramite `sum`, arrivando così al risultato finale.

Ora che abbiamo visto come utilizzare questo tipo di operatori, possiamo passare a quelli che contano gli elementi.

[Count, LongCount](#)

Come è facile desumere dai loro nomi, `Count` e `LongCount` ritornano il numero di elementi che sono presenti in una lista. La differenza tra i due risiede nel fatto che `Count` restituisce un `Int32`, mentre `LongCount` un `Int64`. Il loro utilizzo è mostrato nell'[esempio 9.21](#).

Esempio 9.21

```
var count = ordini.Count();
var longcount = ordini.LongCount();
```

Ora che abbiamo trattato anche questa categoria, passiamo ad analizzare la prossima, relativa agli operatori che filtrano i dati in base alla loro posizione.

Gli operatori di elemento

Gli operatori che fanno parte di questa categoria hanno lo scopo di far restituire alla query un solo elemento e non una lista. Infatti, quando eseguiamo una query LINQ, il risultato è sempre un `IEnumerable<T>` (oppure un'interfaccia derivata). Se sappiamo già che stiamo cercando un solo elemento, possiamo utilizzare questi operatori per impostare direttamente l'oggetto come tipo di ritorno.

Il primo metodo è `First` e serve per restituire il primo elemento della lista, come mostrato nell'[esempio 9.22](#).

Esempio 9.22

```
var first = ordini.First();
```

Se la lista non contiene alcun elemento, questo metodo solleva un'eccezione. Per prevenire questo comportamento, esiste il metodo `FirstOrDefault` che, nel caso in cui la lista sia vuota, torna il valore di default del tipo ("0" per interi, "false" per booleani, "null" per i tipi per riferimento, come le stringhe ecc.).

Entrambi i metodi appena menzionati hanno anche un overload che permette di specificare una condizione di ricerca. In questo caso, viene restituito il primo oggetto che corrisponde ai criteri di ricerca. L'impiego di questo overload permette di risparmiare codice, in quanto evita l'utilizzo del metodo `where`, come possiamo notare nell'[esempio 9.23](#).

Esempio 9.23

```
Ordine first = ordini.First(o => o.Id == 1);
```

Mentre `First` e `FirstOrDefault` tornano il primo elemento, `Last` e `LastOrDefault` svolgono il compito esattamente opposto, restituendo l'ultimo elemento della lista. Anche per `Last`, vale la regola delle eccezioni vista per `First`. Se non vogliamo che sia sollevata un'eccezione, dobbiamo usare `LastOrDefault`.

Un altro metodo importante è `single`. Questo operatore ritorna sempre il primo membro di una lista, ma presenta la caratteristica di sollevare un'eccezione non solo quando non ci sono elementi o se il filtro di ricerca non ha prodotto risultati, ma anche quando la lista ha più di un elemento. Questo metodo è molto comodo perché, alla presenza di situazioni anomale con dati duplicati, questi ultimi vengono subito individuati.

L'utilizzo di `singleOrDefault` tiene al riparo da eccezioni di dati non presenti ma, in caso di più oggetti nella lista, l'eccezione viene comunque sollevata.

Gli ultimi due metodi di questa tipologia sono `ElementAt` e `ElementAtOrDefault`. Questi metodi restituiscono l'elemento che si trova a uno specifico indice nella lista. Se utilizziamo `ElementAt` e l'elemento non esiste, viene sollevata un'eccezione. Se, invece, utilizziamo `ElementAtOrDefault`, l'eccezione non viene sollevata e otteniamo il valore di default del tipo cercato.

Sin qui questi due metodi si comportano in maniera del tutto analoga agli altri della stessa famiglia. Tuttavia c'è una piccola diversità dovuta al fatto che questi metodi non hanno un overload per effettuare una ricerca ma una sola firma, che accetta in input un `Int32` che contiene l'indice.

`DefaultIfEmpty` è un metodo completamente diverso rispetto agli altri. Il suo scopo non è quello di restituire un solo elemento, ma di aggiungere un elemento vuoto a una lista, nel caso in cui questa sia vuota. Nel caso di query che fanno uso di `outer join`, possiamo utilizzare questo metodo per associare un elemento vuoto laddove non ci siano corrispondenze.

Ora che sappiamo come estrarre un oggetto in base alla sua posizione, vediamo come estrarre gruppi di oggetti, utilizzando lo stesso criterio.

[Gli operatori di partizionamento](#)

Gli operatori di partizionamento hanno lo scopo di estrarre oggetti da una lista in base alla loro posizione. Un esempio classico dell'utilizzo di questi metodi

si verifica quando dobbiamo fare una paginazione. I metodi principali di questa categoria sono `Take` e `Skip`.

Take e Skip

Il metodo `Take` serve per restituire le prime n righe di una lista. Questo metodo accetta in input un solo parametro, di tipo `Int32`, che specifica il numero di elementi da estrarre. Il suo utilizzo è mostrato nell'[esempio 9.24](#).

Esempio 9.24

```
var result = ordini.Take(2);
```

Di per sè il metodo `Take` non è di grande utilità, in quanto può estrarre solo i primi record. Tuttavia, la vera potenza di questo metodo si svela quando ne facciamo un utilizzo combinato con `skip`. L'operatore `skip` permette di saltare l'estrazione delle prime n righe di una lista. A questo punto, possiamo facilmente creare query in grado di estrarre pagine di dati con una sola istruzione, utilizzando a dovere il metodo `skip` per saltare i primi n elementi e `Take` per prendere i primi n elementi successivi. L'[esempio 9.25](#) dimostra come effettuare una paginazione.

Esempio 9.25

```
var result = ordini.Skip(2).Take(2);
```

In questa query recuperiamo un'ipotetica seconda pagina composta di due elementi poiché i primi due oggetti vengono saltati.

TakeWhile e SkipWhile

Il metodo `TakeWhile` serve per restituire i primi elementi di una lista, che soddisfano una determinata condizione tramite una lambda expression. La caratteristica principale di questo metodo è che, appena incontra un oggetto

che non soddisfa la condizione, l'iterazione si arresta e tutti i successivi record vengono ignorati.

Il metodo `skipwhile` è, in tutto e per tutto, l'opposto di `takewhile`. Questo metodo salta tutti i primi record che soddisfano la condizione espressa tramite la lambda expression `e`, appena trova un record non corrispondente, smette di effettuare il controllo e ritorna indiscriminatamente tutti gli altri oggetti della lista.

Ora che anche gli operatori di partizionamento sono stati trattati, possiamo passare all'ultima categoria, che si occupa di compiere operazioni d'insiemistica sulle liste.

Operatori di insieme

Gli operatori di insieme hanno il compito di confrontare le liste e individuare gli oggetti uguali, quelli differenti e quelli doppi. Oltre a questo, tramite l'operatore di unione, possiamo fondere insieme due o più liste.

Per fare le comparazioni negli esempi proposti di seguito, dobbiamo introdurre una seconda lista, che possiamo vedere nell'[esempio 9.26](#).

Esempio 9.26

```
ordiniSet = new List<Ordine>() {  
    new Ordine { Id = 1, Data = DateTime.Now },  
    new Ordine { Id = 2, Data = DateTime.Now },  
    new Ordine { Id = 5, Data = DateTime.Now },  
    new Ordine { Id = 6, Data = DateTime.Now },  
    new Ordine { Id = 6, Data = DateTime.Now }  
};
```

Il primo operatore che andiamo ad analizzare è `Except`.

Except

Il metodo `Except` permette di estrarre da una lista gli oggetti che non hanno una corrispondenza con quelli presenti in una seconda lista. Questo metodo ha due overload: un primo, dove passiamo solo la lista da confrontare e un secondo, dove possiamo passare un `IEqualityComparer` per decidere quando due oggetti sono uguali o non lo sono. Se non vogliamo creare una classe che

implementa `IEqualityComparer`, possiamo ricorrere alla modifica della classe `Ordine` per gestire le uguaglianze. Per fare questo, basta dare l'override dei metodi `GetHashCode` e `Equals`, così come possiamo notare nell'[esempio 9.27](#).

Esempio 9.27

```
public override bool Equals(object obj)
{
    return ((Ordine)obj).Id == Id;
}

public override int GetHashCode()
{
    return Id;
}
```

Facendo l'override di questi metodi, LINQ è in grado di capire da solo quando due oggetti sono uguali. In questo caso, due ordini presenti in entrambe le liste sono uguali quando hanno lo stesso ID.

A questo punto, possiamo utilizzare il metodo `Except` senza difficoltà, come mostrato nell'[esempio 9.28](#).

Esempio 9.28

```
var result = ordini.Except(ordiniSet);
```

Questa query produce come risultato la restituzione degli ordini 3 e 4, in quanto gli ordini 1 e 2 sono presenti anche nella seconda lista.

[Intersect](#)

Questo metodo ha lo scopo di creare una nuova lista dove siano presenti gli elementi che sono in entrambe le liste comparate. Anche questo metodo ha due overload: il primo che accetta la lista da comparare e il secondo che accetta un `IEqualityComparer`, per specificare a LINQ come identificare i doppioni (inutile in questo caso, dato che la classe `Ordine` implementa i metodi `Equals` e `GetHashCode`). L'[esempio 9.29](#) ne mostra l'utilizzo.

Esempio 9.29

```
var result = ordini.Intersect(ordiniSet);
```

Il risultato di questa query è una lista con gli ordini 1 e 2, che sono quelli presenti in entrambe le liste ispezionate.

Distinct

L'operatore `Distinct` permette di ottenere una lista in cui gli eventuali doppioli vengono eliminati. Poiché i doppioli vengono cercati nella lista stessa, questo metodo non accetta parametri, a meno che non usiamo l'overload che accetta un `IEqualityComparer`. L'utilizzo viene mostrato nell'[esempio 9.30](#).

Esempio 9.30

```
Dim result = ordiniSet.Distinct();
```

Il risultato di questa query è una lista di quattro elementi, poiché l'ordine con ID 6 è incluso due volte nella lista.

Union

`Union` è l'ultimo metodo della categoria in esame e permette di fondere insieme due liste. Nel caso in cui LINQ sia in grado di determinare quando due oggetti sono uguali, i doppioli vengono eliminati tramite `IEqualityComparer` o override dei metodi della classe. Nell'[esempio 9.31](#), diamo una dimostrazione di utilizzo di questo metodo.

Esempio 9.31

```
var result = ordiniSet.Union(ordiniSet);
```

Il risultato di questa query è una lista con gli ordini 1, 2, 3, 4, 5 e 6 senza il doppione con ID pari a 6. A questo punto abbiamo illustrato i principali metodi che permettono di eseguire query con LINQ. Ovviamente non abbiamo coperto il 100% delle possibilità ma abbiamo analizzato solo quelli che vengono utilizzati nella maggior parte dei casi. Il passo successivo sarà quello di scoprire un modo diverso di scrivere le query.

La Query Syntax

L'utilizzo degli extension method in combinazione con le lambda expression è molto comodo. Tuttavia, molti sviluppatori non si trovano perfettamente a loro agio usando questa sintassi; soprattutto quelli che possiedono un background orientato al linguaggio SQL.

Per questi sviluppatori è stata introdotta la cosiddetta **Query Syntax**. Questa funzionalità permette di scrivere query LINQ con una sintassi molto simile al linguaggio SQL. In questo modo possiamo approcciare con una maggior familiarità la scrittura delle query LINQ, come mostrato nell'[esempio 9.32](#).

Esempio 9.32

```
var result = from o in ordini
              where o.Id == 1
              select o;
```

Da questo esempio possiamo vedere come la sintassi sia molto simile a quella SQL, a eccezione della prima istruzione, che è una `From` e non una `select`. Il motivo di questa scelta risiede nel fatto che, per permettere a Visual Studio di offrire l'intellisense, dobbiamo specificare da subito quali oggetti devono essere interrogati.

Bisogna tenere sempre presente una cosa: quando il compilatore incontra questa query, la converte in chiamate agli extension method. Questo significa che, dal punto di vista delle performance, non vi è alcun vantaggio nell'usare l'una o l'altra tecnica. La scelta è dettata esclusivamente dai nostri gusti personali.

Sebbene sia molto comoda, la Query Syntax non ha le stesse potenzialità offerte dagli extension method associati alle lambda expression. Pertanto, è

utilizzabile solo in alcuni scenari non troppo complessi. Di fatto, sono molto pochi gli extension method che hanno un mapping con le clausole della Query Syntax.

Tabella 9.2 – Parole chiave con la Query Syntax

Query Syntax Operator	Extension Method
From	
group by	Groupby
join	Join
let	
order by	Orderby
select	Select
where	Where

Questa tabella mostra che se da un lato abbiamo a disposizione molti metodi, dall'altro questi non sono sufficienti a coprire tutte le esigenze. In base alla nostra esperienza, possiamo affermare che, nella maggior parte dei casi, la sintassi con gli extension method è l'unica strada percorribile.

Nel precedente capitolo abbiamo accennato come Parallel LINQ sia in grado di migliorare sensibilmente le prestazioni delle nostre query LINQ to Objects. Nella prossima sezione affronteremo questo argomento.

Parallel LINQ

All'interno delle Parallel Extension, trova posto un'implementazione di LINQ grazie alla quale possiamo facilmente fare in modo che le nostre query sfruttino al massimo l'architettura multicore e multiprocessore messa a disposizione dall'hardware di oggi. Ciò è possibile grazie alla presenza dell'extension method `AsParallel`, tramite il quale possiamo istruire il runtime per far sì che la query venga eseguita in parallelo. Ovviamente, affinché ne possiamo apprezzare i vantaggi, i test devono essere eseguiti su una grande quantità di dati, come nel caso dell'[esempio 9.33](#).

Esempio 9.33

```

var myList = new List<int>();
var rnd = new Random();

for (int index = 1; index <= 10000000; index++)
{
    myList.Add(rnd.Next(0, 100000));
}

// Query per recuperare i numeri primi
var primes = from n in myList.AsParallel()
             where IsPrime(n)
             select n;

var sw = Stopwatch.StartNew();
primes.Count();
sw.Stop();

```

Semplicemente aggiungendo `AsParallel()` e quindi attivando l'engine di PLINQ, una query come quella dell'esempio precedente riceve tangibili miglioramenti prestazionali; la lista in ingresso viene infatti partizionata in diversi sottogruppi di dati, in modo che possa essere processata in parallelo dalle diverse unità di calcolo del sistema (core e/o CPU). Ciò si evince anche dando un'occhiata al task manager di Windows, monitorando l'utilizzo dei processori durante l'esecuzione dell'applicazione, come mostrato nella [Figura 9.3](#).

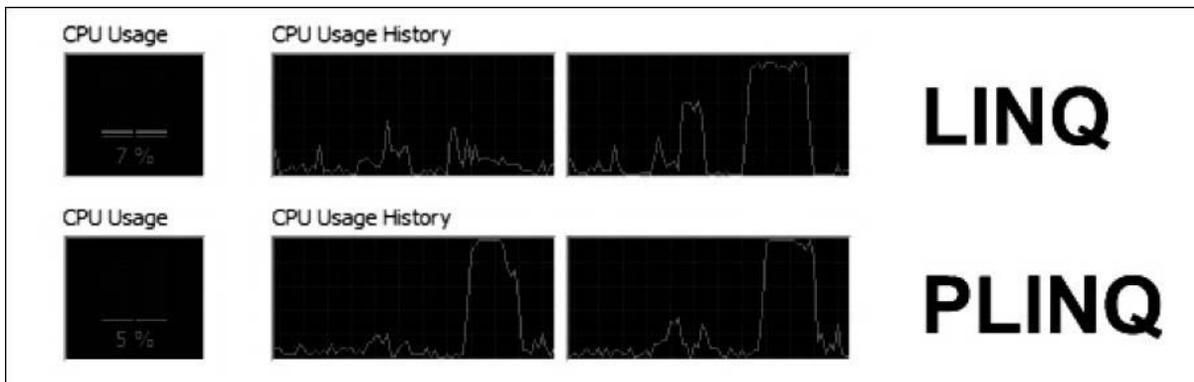


Figura 9.3 – Utilizzo delle CPU senza e con PLINQ.

I vantaggi prestazionali derivanti dall'uso di PLINQ non sono i medesimi per tutte le query e dipendono pesantemente dalla logica di ognuna di esse. In generale, i migliori guadagni si avvertono quando esse coinvolgono algoritmi di calcolo potenzialmente lunghi in corrispondenza delle clausole **where** o **select** e per i quali l'ordine degli elementi non rappresenti un fattore.

Esempio 9.34

```
var query = from item in list.AsParallel()  
            where VeryLongAlgorithm(item)  
            select item;
```

In generale, l'esecuzione di una query su diverse CPU non garantisce che l'ordine degli elementi in ingresso venga preservato; nel caso in cui questo sia necessario, possiamo indicarlo tramite il metodo `AsOrdered`.

Esempio 9.35

```
var query = from item in list.AsParallel().AsOrdered()  
            where VeryLongAlgorithm(item)  
            select item
```

È comunque l'engine stesso che deve valutare, caso per caso, se il parallelismo possa rappresentare un vantaggio, decidendo eventualmente di eseguire la query in modo sequenziale, anche in presenza della clausola `AsParallel`.

La deferred execution comporta che le query non vengano eseguite finché non accediamo al loro contenuto. Se volessimo iterare il risultato di una query con un `foreach`, sebbene PLINQ riesca a sfruttare il parallelismo per calcolare il risultato dell'espressione, dovremmo comunque popolare una collezione temporanea con i dati estratti, in modo che l'iteratore possa funzionare. Si tratta, ovviamente, di una condizione penalizzante che, qualora l'ordine degli elementi non fosse importante, potrebbe essere evitata utilizzando l'extension method `ForEachAll`, come nell'[esempio 9.36](#).

Esempio 9.36

```
var primes = from n in myList.AsParallel().AsOrdered()  
            where IsPrime(n)  
            select n;  
  
primes.ForEachAll((n) => DoSomething(n));
```

In questo caso, la `Action` che abbiamo specificato come argomento di `ForEachAll` verrà eseguita sui singoli partizionamenti in cui PLINQ ha suddiviso la lista

iniziale, godendo quindi dei vantaggi del parallelismo e senza l'overhead di ricostruire una collezione sul thread chiamante.

Esistono molti casi in cui, in generale, vogliamo realizzare cicli che potenzialmente possano essere eseguiti in parallelo, senza che questi coinvolgano necessariamente query PLINQ. Per gestire situazioni simili, .NET dispone della classe `Parallel`, che espone due metodi statici, `For` e `ForEach`. Questi ultimi sono rappresentativi degli omonimi costruiti di C# ma sfruttano internamente l'infrastruttura dei task per ripartire l'onere computazionale tra i core disponibili.

Conclusioni

LINQ introduce una serie di nuovi strumenti per lo sviluppatore, ampliandone notevolmente le potenzialità. Grazie a questa tecnologia, si realizza in maniera concreta la possibilità di scrivere meno codice e di renderlo, allo stesso tempo, molto più leggibile che in passato.

Inoltre, LINQ ha un'espressività intrinseca che avvicina il linguaggio C# alla programmazione funzionale; tutto questo, sommato all'architettura a provider, pone LINQ al centro delle strategie di accesso verso le più svariate sorgenti dati.

Infine, grazie a Parallel LINQ, abbiamo la capacità di migliorare sensibilmente le prestazioni delle nostre query LINQ to Objects, per via di un engine in grado di valutare se queste possono trarre benefici dall'esecuzione parallela e, eventualmente, utilizzare l'infrastruttura dei task per implementarla. Ora che abbiamo spiegato come scrivere query all'interno del codice, nel prossimo capitolo analizzeremo come eseguire query verso il database utilizzando le librerie native di .NET.

Accedere ai dati con ADO.NET

La tecnologia che .NET mette a disposizione per accedere ai dati è ADO.NET. ADO.NET è composto da classi che lavorano a basso livello con il database astraendone le complessità e permettendo a noi di scrivere solamente il codice di business rilevante per la nostra applicazione. Oltre a queste classi, ADO.NET ne mette a disposizione anche altre per manipolare in locale i dati letti dal database per poi aggiornarli sullo stesso database in un momento successivo.

L'utilizzo di ADO.NET assicura l'uniformità della tecnologia per l'accesso ai dati: un sistema di tipi comune, i modelli di progettazione e le convenzioni di denominazione vengono infatti condivisi da tutti i componenti. Inoltre, ADO.NET fornisce un'architettura estendibile che offre una base comune sulla quale creare provider per ogni tipologia di database (SqlServer, Oracle, MySql e così via).

Sulla base di ADO.NET, Microsoft ha costruito un O/RM denominato Entity Framework. Questo semplifica notevolmente lo sviluppo del codice di accesso ai dati offrendo funzionalità come la trasformazione dei dati dal database in oggetti, il tracking delle modifiche fatte agli oggetti e la loro persistenza, il supporto a LINQ e molto altro ancora.

Architettura di ADO.NET

Come abbiamo detto, ADO.NET fornisce un sistema di tipi comune e un'architettura estendibile che permette di creare provider (data provider d'ora in poi) per connettersi a qualunque tipo di database (SqlServer, SQLite, Oracle, MySql e così via). Per raggiungere questo scopo, ADO.NET è

composto da una serie di namespace che raccolgono le diverse classi per l'accesso ai dati in funzione del loro scopo e della loro implementazione:

- ❑ il namespace `System.Data` include le classi base che ogni data provider deve implementare per dialogare con lo specifico database;
- ❑ i **data provider** contengono classi che ereditano da quelle del namespace `System.Data.Common` e che implementano il codice necessario per connettersi a uno specifico database. Generalmente, ogni data provider ha un proprio namespace. Un tipico esempio è il data provider per `SqlServer`, già presente nel .NET, che si trova nel namespace **`System.Data.SqlClient`**. Oltre a questo provider, ne esiste un altro per `SqlServer` denominato **`Microsoft.Data.SqlClient`** di cui parleremo più avanti;
- ❑ il namespace `System.Data.SqlTypes` contiene le classi che rappresentano i tipi di dati utilizzati in ambito SQL.
- ❑ il namespace `System.Data` racchiude le classi di uso generale, indipendenti dalla tipologia di sorgente dati (database, file XML, file JSON e così via), che permettono di lavorare con i dati in locale.

Con quest'architettura ogni gruppo di classi ha un suo specifico ruolo. Vediamo nel dettaglio cosa contengono i namespace menzionati partendo dal primo.

[Il namespace `System.Data.Common`](#)

In `System.Data.Common` si gettano le basi per un modello di programmazione comune, per quanto riguarda l'interfacciamento con il database, in quanto tutti i data provider devono ereditare da queste classi. Le classi più importanti di questo namespace sono:

- ❑ **`DbConnection`**: permette di stabilire una connessione con il database;
- ❑ **`DbCommand`**: permette di eseguire comandi sul database sia per leggere che per scrivere dati;

- ❑ **DbDataReader**: permette di leggere i dati recuperati tramite una query eseguita con il `DbCommand`;
- ❑ **DbTransaction**: permette di aprire una transazione con il database garantendo che le modifiche ai dati siano effettive solamente se ogni comando va a buon fine. In caso di errori le modifiche vengono annullate;
- ❑ **DbParameter**: rappresenta un parametro di input e/o output per una stored procedure o un comando testuale;
- ❑ **DbConnectionStringBuilder**: è l'oggetto builder per la costruzione della stringa di connessione;
- ❑ **DbDataAdapter**: incapsula la logica di connessione, transazione e comandi permettendo di eseguire query di lettura e scrittura in maniera molto semplice integrandosi anche con le classi in `System.Data`. Con i moderni paradigmi di programmazione a oggetti, questa classe è divenuta obsoleta ed è presente in .NET Standard esclusivamente per retrocompatibilità con lo scopo di favorire il porting di applicazioni legacy da .NET Framework a .NET Core. Per questo motivo non approfondiremo l'utilizzo di questa classe e delle sue derivate nei data provider ma ne parleremo solo superficialmente.

Queste classi sono astratte e rappresentano la base per il colloquio con il database. Questa lista non è completa in quanto il numero completo di classi presenti nel namespace è 33. Tuttavia elencarle tutte sarebbe inutile in quanto questa lista comprende le classi che vengono usate nel 99% dei casi quindi è più che sufficiente per capire come è fatta la base di un data provider. Vediamo ora come vengono ereditate e implementate queste classi dai data provider.

Data provider

Le classi dei data provider sono quelle che fisicamente usiamo per interagire con il database: connetterci, eseguire query, scrivere dati e così via. Queste classi non sono altro che un'estensione di quelle viste nel precedente paragrafo. .NET contiene già un data provider per `SqlServer`, contenuto nel namespace `System.Data.SqlClient`. Questo provider è incluso in .NET per

retrocompatibilità con il codice esistente e, a parte aggiornamenti di bug e di sicurezza, non verrà ulteriormente evoluto.

In sostituzione, Microsoft ha sviluppato un nuovo provider, scaricabile da NuGet, `Microsoft.Data.SqlClient`. Questo nuovo provider sarà lo standard per l'accesso a `SqlServer`, sia su `.NET Framework` sia su `.NET Core`, e tutte le nuove funzionalità verranno incluse solo in questo pacchetto. Per minimizzare le modifiche necessarie a migrare dal provider legacy al nuovo, le classi hanno mantenuto lo stesso nome, quindi, al netto del cambio di namespace da `System.Data.SqlClient` a `Microsoft.Data.SqlClient`, non ci sono modifiche da fare. Prendiamo questo provider come esempio per mostrare come un provider implementi le classi base esposte da `System.Data.Common`:

- ❑ **SqlConnection:** eredita da `DbConnection` e permette di stabilire una connessione a `Sql-Server`;
- ❑ **SqlCommand:** eredita da `DbCommand` e permette di eseguire comandi su `SQL Server`;
- ❑ **SqlDataReader:** eredita da `DbDataReader` e permette di leggere i dati recuperati tramite una query eseguita con il `SqlCommand`;
- ❑ **SqlTransaction:** eredita da `DbTransaction` e permette di aprire una transazione su `SqlServer`;
- ❑ **SqlParameter:** eredita da `DbParameter` e permette di specificare un parametro per il comando `SQL` lanciato su `SqlServer`;
- ❑ **ConnectionStringBuilder:** eredita da `ConnectionStringBuilder` e permette di costruire programmaticamente una stringa di connessione al database `SqlServer`;
- ❑ **SqlDataAdapter:** eredita da `DbDataAdapter` e ne incapsula le logiche specializzandole per `SqlServer`;

Quello che appare evidente da questa lista, è che per creare un data provider tutto quello che bisogna conoscere sono le classi base da cui ereditare e le competenze tecnologiche del database verso il quale ci si vuole interfacciare. Non è un caso che i vari vendor di database abbiano già creato e distribuito,

tramite pacchetto NuGet, il loro data provider sia per .NET Core che per .NET Framework.

Oltre al provider legacy e al nuovo provider per SqlServer disponibili sia per .NET Core sia per .NET Framework, Microsoft fornisce altri provider come quello per SQLite (solo per .NET Core) e quelli per Odbc e OleDb. Provider per altri tipi di database come Oracle, MySql, PostgreSQL e Fire-bird, solo per nominare quelli più comuni, sono forniti da terze parti.

Una volta capito come lavorare fisicamente con il database è importante capire come mappare i dati verso i tipi analoghi .NET.

[Il namespace System.Data.SqlTypes](#)

Una stringa in .NET è differente da una stringa nel database in quanto la loro rappresentazione interna cambia. Tuttavia, in .NET è molto più comodo usare la rappresentazione nativa delle stringhe piuttosto che quella del database. Per questo motivo, sono state create nel namespace `System.Data.SqlTypes` classi che agiscono da mapper tra i tipi .NET e i tipi del database. In questo modo noi possiamo lavorare usando i tipi nativi .NET mentre quando i data provider interagiscono con il database usano queste classi. All'interno di `System.Data.SqlTypes` abbiamo classi come `SqlString`, `SqlInt32`, `SqlByte`, `SqlBinary`, `SqlBoolean`, `SqlDateTime` e altre ancora. Il nome delle classi è abbastanza esplicativo quindi non occorre spiegare quali tipi queste mappino.

[Il namespace System.Data](#)

Le classi in `System.Data` permettono di accedere ai dati recuperati dal database, di manipolarli ed eventualmente persisterli sul database. Le classi principali di questo namespace sono quattro:

- ❑ **DataTable**: classe che contiene la lista di record recuperati da una query sul database;
- ❑ **DataRow**: classe che contiene un singolo record letto dal database. Un oggetto `DataTable` contiene una lista di `DataRow`;

- ❑ **DataColumn**: classe che rappresenta una colonna all'interno di una riga. Un oggetto DataRow contiene una lista di DataColumn;
- ❑ **DataSet**: classe che rappresenta un contenitore di DataTable.

Queste classi ricalcano la struttura di un database. Un oggetto DataSet è come un database, un oggetto DataTable è paragonabile a una tabella con tante righe (oggetti DataRow) ognuna delle quali composte da tante colonne (oggetti DataColumn). Grazie a questa struttura, queste classi rendono il modello di programmazione molto familiare con il database.

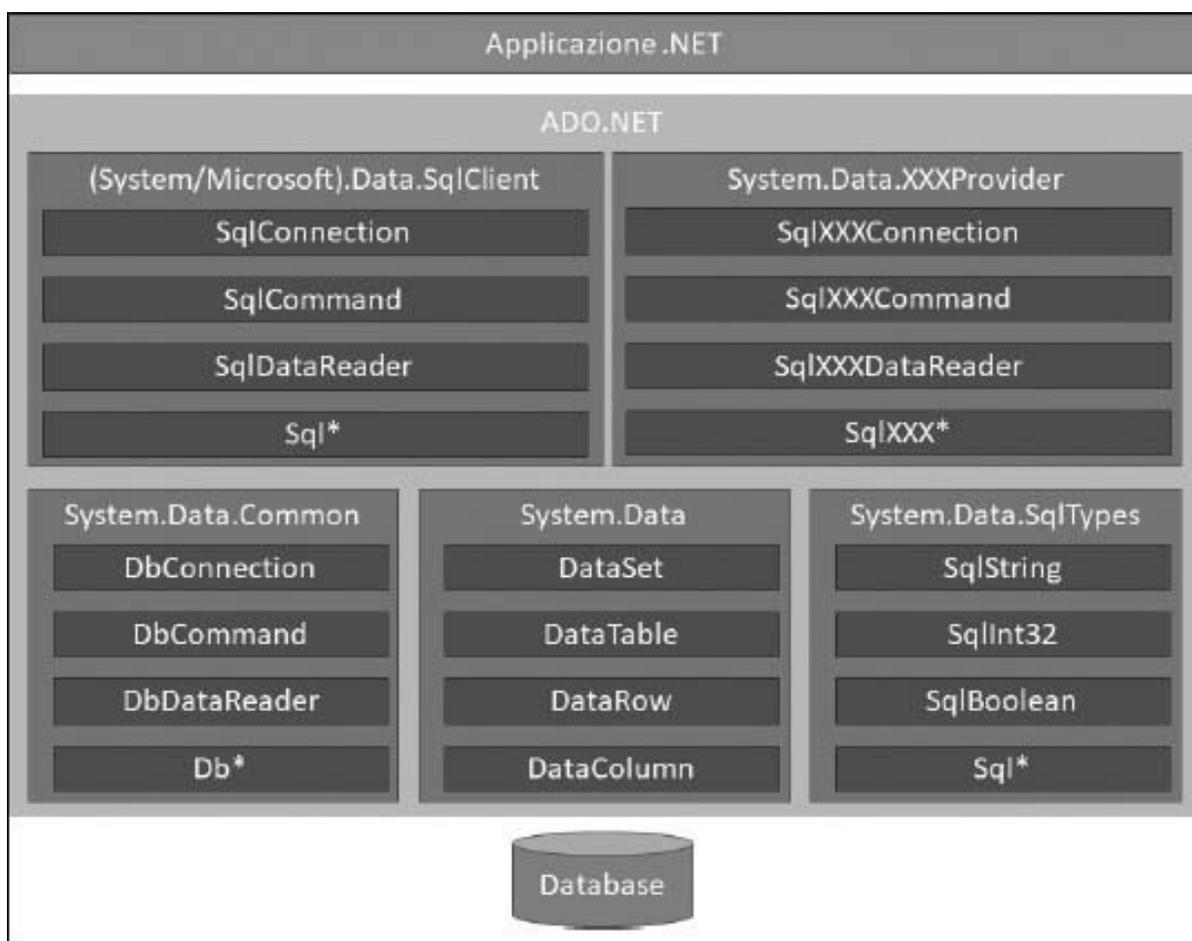


Figura 10.1 – Architettura di ADO.NET.

Conoscere l'architettura di ADO.NET è importante, perché ci fornisce le basi per scrivere codice che accede ai dati. Nella prossima sezione metteremo a frutto queste conoscenze scrivendo questo codice.

Lavorare con ADO.NET

Il primo passo per lavorare con qualunque database è stabilire una connessione. Infatti, per comunicare con un database al fine di eseguire comandi di lettura e di aggiornamento, è sempre necessario instaurare un certo tipo di collegamento, che dipende inevitabilmente dal tipo di database. Nel caso dei database relazionali, l'attivazione di una connessione fisica al server è propedeutica all'inoltro di qualsiasi comando SQL per la lettura o la modifica dei dati contenuti nelle tabelle. Vediamo come stabilire una connessione con il database utilizzando SqlServer.

Stabilire una connessione

Come detto in precedenza, per stabilire una connessione con il database bisogna usare la classe del data provider che eredita da `DbConnection`. Questa classe espone la proprietà `ConnectionString` che rappresenta una stringa tramite la quale specifichiamo i parametri di connessione al database. La stringa di connessione per una particolare istanza può essere specificata sia tramite la proprietà appena menzionata, sia in fase di creazione tramite un costruttore che accetta la stringa di connessione come parametro.

Una volta impostata la stringa di connessione, apriamo e chiudiamo la connessione utilizzando rispettivamente i metodi `Open` (o la sua controparte asincrona `OpenAsync`) e `Close`. L'[esempio 10.1](#) mostra come aprire e chiudere una connessione verso un database SQL Server, definendo la stringa di connessione tramite il costruttore parametrico della classe `SqlConnection`. L'utilizzo del blocco di gestione delle eccezioni permette di intercettare in modo appropriato gli eventuali errori, derivanti, per esempio, da un'errata configurazione delle credenziali dell'utente oppure da un problema di connessione al server. Una volta aperta, la connessione a una sorgente dati va sempre chiusa in modo esplicito, per evitare sprechi di risorse.

Esempio 10.1

```
// Stringa di connessione
var connectionString = "Server=localhost;Database=Northwind;
    User ID=appUser; Password=p@$$w0rd";

// Creazione dell'istanza di SqlConnection
var conn = new SqlConnection(connectionString);
```

```

try
{
    // Apertura della connessione
    await conn.OpenAsync();
    // ...
}
catch(SQLException ex)
{
    // Gestione dell'eccezione
}
finally
{
    // Chiusura della connessione
    if (conn.State == ConnectionState.Open)
        conn.Close();
}

```

Dal momento che la classe `DbConnection` implementa l'interfaccia `IDisposable`, il codice precedente può essere scritto in modo più compatto, sfruttando il costrutto `using`.

Esempio 10.2

```

var connectionString = "...";

using(var connection = new SqlConnection(connectionString))
{
    try
    {
        await connection.OpenAsync();
        // ...
    }
    catch(SQLException ex)
    {
        // Gestione dell'eccezione
    }
}

```

Il codice riportato nell'[esempio 10.2](#) è del tutto equivalente a quello mostrato nell'[esempio 10.1](#). La connessione viene chiusa in modo trasparente al termine del blocco `using`, mediante una chiamata implicita del metodo `Dispose` sia in caso di successo sia in caso di errore.

La stringa di connessione è composta da una serie di coppie nome/valore separate dal carattere “;” (punto-e-virgola), dove il nome corrisponde a una parola chiave. Le principali parole chiave sono elencate qui di seguito:

- ❑ `Data Source`: (equivalente a `server`) specifica il percorso dove risiede la sorgente dati;

- ❑ `Database` (equivalente a `Initial Catalog`) identifica il database predefinito;
- ❑ `User ID` (equivalente a `uid`) identifica il nome dell'utente nel caso in cui sia necessario specificare le credenziali di accesso (per esempio, autenticazione `SqlServer`);
- ❑ `Password` (equivalente a `Pwd`) identifica la password dell'utente nel caso in cui sia necessario specificare le credenziali di accesso;
- ❑ `Integrated Security` (equivalente a `Trusted_Connection`) permette di abilitare l'autenticazione Windows (autenticazione integrata). In questo caso, le credenziali dell'utente possono essere omesse.

Oltre a queste parole chiave, ne esistono altre di cui alcune valide per ogni database e altre specifiche per tipologia di database. Elencare tutte le parole chiave è un'operazione che esula dagli scopi di questo testo in quanto richiederebbero molto spazio e la conoscenza di ogni parola chiave specifica per database.

Negli esempi analizzati in precedenza, abbiamo specificato la stringa di connessione direttamente nel codice, ma questa pratica non è utilizzabile in un contesto reale dove la stringa di connessione deve provenire dalla configurazione. Se usiamo .NET Framework, possiamo inserire la stringa di connessione nella sezione `connectionStrings` del file `app.config` o `web.config`, mentre se usiamo .NET Core, possiamo inserire la stringa di connessione nella sezione `ConnectionStrings` nella root. Ogni stringa di connessione è specificata da una coppia chiave/valore dove la chiave è un identificativo della stringa e il valore è la stringa di connessione ([esempio 10.3](#)).

Esempio 10.3

```
.NET Framework
<configuration>
  <connectionStrings>
    <add name="SQL"
        connectionString="
            Server=dbserver;
            Database=northwind;Integrated security=true" />
  </connectionStrings>
</configuration>

.NET Core
```

```
{
  "ConnectionStrings": {
    "SQL": "Server=dbserver;Database=northwind;Integrated security=true"
  }
}
```

Se usiamo .NET Framework possiamo accedere alle stringhe di connessione tramite la proprietà statica `ConnectionStrings` della classe `ConfigurationManager`. Questa proprietà accetta come indexer il nome della chiave della stringa di connessione e restituisce l'elemento di configurazione che contiene la proprietà `ConnectionString` che espone la stringa di connessione.

Se usiamo .NET Core le stringhe di connessione sono accessibili tramite il metodo `GetConnectionString` dell'interfaccia `IConfiguration`. Questo metodo accetta in input il nome della chiave relativa alla stringa di connessione e restituisce la stringa. Nell'[esempio 10.4](#) vediamo come utilizzare queste classi nel codice.

Esempio 10.4

```
.NET Framework
public class CustomerService
{
  private readonly string connectionString;

  public CustomerService()
  {
    connectionString =
      ConfigurationManager.ConnectionStrings["SQL"].ConnectionString;
  }
}

.NET Core
public class CustomerService
{
  private readonly string connectionString;

  public CustomerService(IConfiguration configuration)
  {
    connectionString = configuration.GetConnectionString("SQL");
  }
}
```

Oltre a essere recuperata dal file di configurazione, una stringa di connessione può essere costruita in modo programmatico sfruttando l'implementazione della classe `DbConnectionStringBuilder` che ogni data provider fornisce.

La classe `DbConnectionStringBuilder` permette di assemblare la stringa di connessione, fornendo un controllo intrinseco sul formato e sulla validità

dei vari parametri. A ciascun parametro della stringa di connessione corrisponde una proprietà dell'oggetto builder, che deve essere impostata in modo opportuno. Una volta valorizzati i parametri necessari, la stringa finale è accessibile tramite la proprietà `ConnectionString`. L'[esempio 10.5](#) mostra la costruzione di una stringa di connessione per accedere a un database SQL Server tramite `SqlConnectionStringBuilder`.

Esempio 10.5

```
var builder = new SqlConnectionStringBuilder();
builder.DataSource = serverName;
builder.InitialCatalog = database;
builder.IntegratedSecurity = true;
var connectionString = builder.ConnectionString;
```

Nel caso in cui le parti della stringa di connessione derivino da un input da parte dell'utente, l'utilizzo dell'oggetto builder migliora decisamente la sicurezza, riducendo in modo significativo il rischio di attacchi basati su input malevoli.

Quando si lavora con SQL Azure possiamo sfruttare la managed service identity al posto della username e la password come meccanismo di autenticazione del client. In questo modo, la stringa di connessione non deve contenere dati sensibili. Per sfruttare questa funzionalità, dobbiamo impostare la proprietà `AccessToken` con l'access token ottenuto dal servizio di Azure. Maggiori informazioni su questa tecnica sono disponibili all'indirizzo <https://aspit.co/bvn>.

Stabilire la connessione con il database è il primo passo per dialogare, il secondo consiste nell'esecuzione di comandi sul database a cui si è connessi.

[Esecuzione di un comando](#)

Una volta che la connessione è stata attivata, possiamo inviare comandi alla sorgente dati per la lettura e per la scrittura. A tale scopo, il modello a oggetti di ADO.NET fornisce il tipo base astratto `DbCommand`, che include una serie di

membri comuni a tutte le implementazioni, presenti nei vari data provider. La [Tabella 10.1](#) riporta le proprietà e i metodi di uso più frequente.

Tabella 10.1 - Membri principali della classe DbCommand

Proprietà o metodo	Descrizione
CommandText	Proprietà che imposta lo statement SQL o il nome della stored procedure da eseguire.
CommandType	Proprietà di tipo CommandType (enumerazione) che definisce la tipologia del comando. I valori possibili sono: Text (default), StoredProcedure, TableDirect.
Connection	Proprietà che associa una connessione al comando.
Parameters	Proprietà che rappresenta la collezione dei parametri di input e output utilizzati dal comando.
Transaction	Proprietà che permette di definire a quale transazione associata alla connessione il comando appartiene.
ExecuteNonQuery ExecuteNonQueryAsync	Metodi per l'esecuzione di un comando diverso da una query. Il primo viene eseguito in modalità sincrona e ritorna il numero di righe interessate. Il secondo viene eseguito in modalità asincrona e ritorna un oggetto Task<int> che espone il numero di righe interessate.
ExecuteReader ExecuteReaderAsync	Metodi per l'esecuzione di una query. Il primo viene eseguito in modalità sincrona e ritorna un cursore di tipo forward-only e read-only, contenente il risultato. Il secondo viene eseguito in modalità asincrona e ritorna un oggetto Task<DbDataReader> che espone il cursore.
ExecuteScalar ExecuteScalarAsync	Metodi per l'esecuzione di una query di cui viene preso il primo valore della prima colonna della prima riga ignorando altri dati. Il primo viene eseguito in modalità sincrona e ritorna il risultato (di tipo object). Il secondo viene eseguito in modalità asincrona e ritorna un oggetto Task<object> che espone il risultato.

Come possiamo vedere nella [Tabella 10.1](#), i metodi per l'esecuzione di un comando sono tre (in realtà sei se consideriamo che ci sono sia le versioni sincrone che quelle asincrone). Ciascuna funzione restituisce un valore di ritorno differente, a testimonianza del fatto che i metodi sono stati concepiti per un utilizzo specifico. ExecuteNonQuery ed ExecuteNonQueryAsync permettono di invocare un comando di inserimento (INSERT), aggiornamento (UPDATE) o cancellazione (DELETE) e ritornano il numero di righe

interessate. `ExecuteReader` ed `ExecuteReaderAsync` consentono di eseguire interrogazioni sui dati (`SELECT`), restituendo uno o più resultset a seconda del numero di query che inviamo. `ExecuteScalar` ed `ExecuteScalarAsync` permettono infine di recuperare un valore singolo da una query e si rivelano particolarmente efficaci nel caso di comandi che recuperano valori aggregati come, per esempio, `SELECT COUNT`, `SELECT MAX`, `SELECT MIN` e così via.

L'[esempio 10.6](#) riporta le tre casistiche d'esecuzione di un comando sincrono e asincrono verso un database `SqlServer`.

Esempio 10.6

```
//Metodo sincrono
private void Method(SqlConnection connection)
{
    // Aggiornamento
    var cmdUpdate = new SqlCommand("UPDATE Products SET ...", connection);
    int affectedRows = cmdUpdate.ExecuteNonQuery();

    // Query
    var cmdQuery = new SqlCommand("SELECT * FROM Products", connection);
    SqlDataReader reader = cmdQuery.ExecuteReader();

    // Conteggio
    var cmdCount = new SqlCommand("SELECT COUNT(*) FROM Products", connection);
    var count = (int)cmdCount.ExecuteScalar();
}

//Metodo asincrono
private async Task MethodAsync(SqlConnection connection)
{
    // Aggiornamento
    var cmdUpdate = new SqlCommand("UPDATE Products SET ...", connection);
    int affectedRows = await cmdUpdate.ExecuteNonQueryAsync();

    // Query
    var cmdQuery = new SqlCommand("SELECT * FROM Products, connection ");
    SqlDataReader reader = await cmdQuery.ExecuteReaderAsync();

    // Conteggio
    var cmdCount = new SqlCommand("SELECT COUNT(*) FROM Products", connection);
    var count = (int)(await cmdCount.ExecuteScalarAsync());
}
```

Il testo del comando non deve essere necessariamente espresso per esteso. Infatti, `DbCommand` permette di eseguire anche stored procedure, specificando la tipologia del comando mediante la proprietà `CommandType`. Tra le opzioni possibili, contenute nell'enumerazione `System.Data.CommandType`, il valore `StoredProcedure` consente di specificare l'intenzione di invocare una stored procedure sul database. In tal caso, la proprietà `CommandText` del comando

deve contenere il nome della stored procedure invece del testo SQL formattato esplicitamente.

Per eseguire comandi SQL o stored procedure dotate di valori in ingresso, possiamo usare i parametri. Essi sono istanze delle classi che derivano dal tipo base `DbParameter` e sono caratterizzati da un nome identificativo, un valore, un tipo, una dimensione e una direzione (input/output). Ciascun parametro può essere associato a un comando tramite il metodo `Add` della proprietà `Parameters` ([esempio 10.7](#)).

Esempio 10.7

```
var query = new SqlCommand(
    "SELECT * FROM Orders " +
    "WHERE EmployeeID = @EmployeeID " +
    "AND OrderDate = @OrderDate " +
    "AND ShipCountry = @ShipCountry" +
    "ORDER BY OrderDate DESC", connection);

var p1 = new SqlParameter
{
    ParameterName = "@EmployeeID";
    DbType = DbType.Int32;
    Direction = ParameterDirection.Input;
    Value = 1;
}
var p2 = new SqlParameter
{
    ParameterName = "@OrderDate";
    DbType = DbType.DateTime;
    Direction = ParameterDirection.Input;
    Value = new DateTime(1996, 8, 7);
}
var p3 = new SqlParameter
{
    ParameterName = "@ShipCountry";
    DbType = DbType.String;
    Direction = ParameterDirection.Input;
    Value = "Italy";
}
query.Parameters.Add(p1);
query.Parameters.Add(p2);
query.Parameters.Add(p3);
```

Un approccio alternativo all'uso dei parametri (purtroppo ancora diffuso fra gli sviluppatori) consiste nell'utilizzare la concatenazione di stringhe, allo scopo di comporre il testo del comando SQL includendo i valori in ingresso.

Anche se, come soluzione, può sembrare equivalente a quella basata su parametri e anche più veloce da implementare, **l'uso della concatenazione rappresenta un approccio sbagliato** e, quindi, assolutamente da evitare per

motivi di sicurezza applicativa. Infatti, la semplice concatenazione di stringhe non permette di controllare se i valori in ingresso sono formattati correttamente. Pertanto, la concatenazione consente l'iniezione di codice maligno all'interno del testo del comando SQL, con conseguenze che, nella maggior parte dei casi, si possono rivelare disastrose. Molti attacchi alle applicazioni sfruttano proprio l'uso della concatenazione di stringhe nella formattazione del codice SQL per poter eseguire comandi non previsti dallo sviluppatore e per modificare i dati contenuti nelle tabelle del database. L'approccio basato su parametri garantisce il giusto livello di sicurezza, dal momento che non consente in alcun modo l'iniezione di codice SQL. Per questo motivo, questa soluzione è **sempre** da preferire. Essa permette di proteggersi dagli attacchi di tipo SQL-Injection (iniezione di codice SQL maligno) e garantisce il controllo semantico dei dati in ingresso. Infatti, grazie all'uso dei parametri, la formattazione di date e numeri oppure la codifica dei caratteri speciali, come l'apice singolo, vengono eseguite in modo trasparente, indipendentemente dalle impostazioni internazionali di sistema e dai settaggi del database (come nel caso del secondo parametro dell'[esempio 10.7](#)).

Possiamo usare i parametri con tutti i data data provider. La regola generale prevede di utilizzare il nome del parametro preceduto dal carattere "@" come marcatore all'interno del testo del comando ([esempio 10.7](#)).

Quando inviamo più comandi di scrittura, possiamo inglobarli in una transazione per garantire che tutti siano eseguiti o annullati senza lasciare dati inconsistenti sul database. Vediamo ora come gestire le transazioni.

Scrivere dati in transazione

Quando inviamo un comando di scrittura, il database esegue l'aggiornamento dei dati e ritorna il numero di righe modificate. Tuttavia, spesso capita di dover eseguire più comandi e che questi comandi debbano essere eseguiti in transazione, per garantire che vengano tutti confermati o annullati.

I data provider supportano le transazioni attraverso una classe che eredita da `DbTransaction`. Questa classe viene istanziata sfruttando il metodo `BeginTransaction` della classe che eredita da `DbConnection`. `BeginTransaction` ritorna un oggetto `DbTransaction` sul quale poi possiamo invocare i metodi `Commit` e `Rollback` che, rispettivamente, confermano e annullano i comandi inviati nel contesto della transazione. Nell'[esempio 10.8](#)

vediamo come creare una transazione, inviare comandi e, infine, confermare o annullare la transazione.

Esempio 10.8

```
using (var connection = new SqlConnection(connectionString))
{
    connection.Open();
    using (SqlTransaction transaction = connection.BeginTransaction())
    {
        try
        {
            // inserisce un ordine
            // inserisce i dettagli
            // aggiorna il magazzino
            transaction.Commit();
        }
        catch (SqlException ex)
        {
            transaction.Rollback();
            // Gestione dell'eccezione
        }
    }
}
```

In questo esempio, prima apriamo la connessione e poi iniziamo la transazione; successivamente inviamo i comandi al database e, se non ci sono errori, eseguiamo il commit della transazione. Se invece ci sono errori, il controllo del codice passa per il blocco `catch`, all'interno del quale facciamo il rollback della transazione che annulla tutti i comandi inviati nel contesto della transazione stessa. Infine, alla chiusura dei blocchi `using`, viene fatto il dispose della transazione e della connessione.

Questa modalità di gestione delle transazioni è sicuramente semplice quando abbiamo un metodo che manipola i dati ma, quando abbiamo più metodi che devono scrivere in transazione, dobbiamo fare in modo che questa sia accessibile, o passandola a ogni metodo come parametro o rendendola disponibile in una classe di contesto accessibile dai metodi. Il codice diventa sicuramente più complicato da scrivere in virtù anche del fatto che la transazione potrebbe non essere sempre attiva, a seconda della nostra logica di business.

Per semplificare gli scenari transazionali, ADO.NET mette a disposizione le classi del namespace `System.Transaction` la cui classe principale è `TransactionScope`. Grazie a questa classe possiamo creare una transazione esplicita che viene memorizzata nelle informazioni del thread (e che

sopravvive anche ai cambi di thread qualora usassimo `async/await`) e che viene automaticamente utilizzata dagli oggetti di ADO.NET. In questo modo, nel nostro codice tutto quello che dobbiamo fare è iniziare la transazione e invocarne il `commit` quando il nostro codice viene eseguito con successo.

Esempio 10.9

```
public void CreateOrder(Order order)
{
    using (var scope = new TransactionScope())
    {
        SaveOrder(order);
        UpdateStock(order);
        scope.Complete();
    }
}

private void SaveOrder(Order order)
{
    using (var connection = new SqlConnection(connectionString))
    {
        connection.Open();
        //comandi di salvataggio dell'ordine
    }
}

private void UpdateStock(Order order)
{
    using (var connection = new SqlConnection(connectionString))
    {
        connection.Open();
        //comandi di aggiornamento del magazzino prodotti
    }
}
```

Nel metodo principale di salvataggio, prima creiamo un oggetto `TransactionScope`, così da creare un contesto transazionale. Successivamente chiamiamo i metodi che compiono fisicamente le operazioni sul database e infine invociamo il metodo `Complete` di `TransactionScope`, che indica che, quando lo scope viene chiuso (in questo caso quando si arriva alla fine del blocco `using`) deve essere eseguito il `commit` della transazione. Se non invociamo il metodo `Complete`, al momento della chiusura la transazione viene annullata, lanciando in automatico un `rollback`.

I metodi che scrivono sul database sono completamente agnostici rispetto alla transazione. Gli oggetti di ADO.NET sono in grado di capire che si trovano in un contesto transazionale e utilizzano la transazione in automatico. Questo, per la leggibilità del codice, è un grosso passo in avanti rispetto all'utilizzo della classe `DbTransaction` e derivate.

Ora che siamo in grado di scrivere dati sul database, cambiamo argomento e vediamo come leggerli.

Letture del risultato di una query

Come detto, l'esecuzione dei metodi `ExecuteReader` e `ExecuteReaderAsync` comporta la restituzione di un oggetto contenente i risultati dell'interrogazione. Questo oggetto, detto genericamente **data reader**, è un'istanza di una delle classi che derivano dal tipo astratto `DbDataReader`, contenuto nel namespace `System.Data.Common`. Un data reader rappresenta un cursore client-side di tipo forward-only e read-only, che consente di scorrere e leggere uno o più resultset, generati da un comando associato a una connessione.

Grazie al data reader, possiamo accedere ai dati un record alla volta, utilizzando il metodo `Read` o il suo corrispondente asincrono `ReadAsync`. Dal momento che la funzione ritorna il valore `true` finché tutti i dati non sono stati consumati, può essere usata come condizione d'uscita in un ciclo iterativo di lettura. Chiaramente, il blocco associato al ciclo deve contenere il codice per trattare i dati relativi al record corrente ([esempio 10.10](#)).

La lettura dei campi può essere eseguita in due modi:

- ❑ mediante la proprietà `indexer`, che permette di recuperare il valore di una colonna nel formato nativo in base all'indice o al nome del campo; in questo caso dobbiamo eseguire un'operazione di casting, in funzione del tipo di destinazione;
- ❑ tramite i metodi `Getxxx`, che permettono di leggere i campi in funzione della loro posizione all'interno del record, ritornando direttamente uno specifico tipo di dato (per esempio, `GetInt32` ritorna un intero, `GetDateTime` ritorna una data, `GetString` ritorna una stringa, ecc.).

Esempio 10.10

```
using (var cmd = new SqlCommand("SELECT * FROM Products"), connection)
{
    using (SqlDataReader reader = await cmd.ExecuteReaderAsync())
    {
        while(reader.Read())
        {
            // Viene utilizzata la proprietà indexer
        }
    }
}
```

```

var productID = (int)reader["ProductID"];

// ProductName è il secondo campo del record
var productName = reader.GetString(1);

// ...
}
}
}

```

Il data reader alloca risorse che devono poi essere eliminate nel momento in cui finiamo di leggere i dati. Il modo più semplice per deallocare le risorse è invocando il metodo `close`. Dal momento che la classe `DbDataReader` implementa l'interfaccia `IDisposable`, possiamo usare la sintassi che fa uso del blocco `using` così da invocare automaticamente il metodo `Dispose` che, internamente, chiama il metodo `close`.

Un data reader può contenere più di un resultset. Questo avviene quando al comando che ha generato il data reader sono associate più query. In questo caso, il data reader, una volta creato, viene sempre posizionato sul primo resultset. Per spostarsi da un resultset a quello successivo è necessario utilizzare il metodo `NextResult` o la sua controparte asincrona `NextResultAsync`. Il metodo restituisce il valore *false* se non esistono altri resultset da leggere all'interno del data reader corrente.

Esempio 10.11

```

var cmd = new SqlCommand("SELECT * FROM Products; SELECT * FROM Customers",
    connection);
using (SqlDataReader reader = await cmd.ExecuteReaderAsync())
{
    IterateOverProducts(reader);
    await reader.NextResultAsync();
    IterateOverCustomers(reader);
}

```

L'uso più comune di un data reader è quello di leggere i dati per poi riversarli in uno o più oggetti. L'iterazione degli elementi di un data reader richiede, però, una connessione aperta. Vediamo ora come leggere i dati senza mantenere una connessione aperta con il database.

[Modalità disconnessa in ADO.NET](#)

Oltre alla modalità connessa che, come abbiamo visto, contempla l'utilizzo dei data reader, ADO.NET permette di lavorare sui dati anche in modalità disconnessa, ovvero senza che sia attiva una connessione verso la sorgente dati. Tuttavia, è utile sottolineare che la modalità disconnessa, sebbene sia stata ampiamente adottata dagli sviluppatori nelle primissime versioni del .NET Framework, oggi rappresenta un metodo assolutamente superato e, di conseguenza, sconsigliato per accedere ai dati. L'avvento di tecnologie alternative e più evolute, come Entity Framework, oggetto del prossimo capitolo, ne hanno decretato inevitabilmente l'obsolescenza. Per completezza, in questo contesto ci limitiamo a riportare un veloce richiamo dei concetti principali, senza entrare nei dettagli.

Per poter lavorare sui dati in modalità disconnessa, ADO.NET include un oggetto particolare, simile a un comando, detto genericamente **data adapter**, tramite il quale possiamo popolare un oggetto container con le informazioni recuperate dalla sorgente dati. Un **container di dati** è un oggetto finalizzato a raccogliere, in modo strutturato e ordinato, le informazioni che in esso vengono inserite, fornendo al tempo stesso una serie di funzionalità per il trattamento e la lettura del suo contenuto.

Il data adapter si comporta come tramite, in entrambi i sensi, tra la sorgente dati e il container. Infatti, a differenza del comando dove il resultset viene ritornato sotto forma di cursore read-only, il data adapter sfrutta l'oggetto container come raccogliitore delle informazioni recuperate dalla sorgente dati. In ADO.NET esistono due tipi di container di dati: il DataSet e il DataTable.

Le classi container di ADO.NET non sono elementi specifici di un particolare data provider. Sono altresì dei semplici contenitori, che presentano una serie di funzionalità simili a quelle offerte da un database classico, come l'organizzazione dei dati in tabelle, le relazioni, l'integrità referenziale, i vincoli, l'indicizzazione e così via. Lo scopo dei container è quello di ospitare insiemi di dati, strutturati secondo uno schema specifico, mantenendoli attivi in memoria affinché possano essere letti e modificati in modo semplice e immediato. In particolare, il DataSet è un oggetto composto da un insieme di tabelle, rappresentate da altrettante istanze della classe DataTable e da relazioni di tipo DataRelation. Ciascuna tabella, a sua volta, è composta da righe (classe DataRow) e colonne (classe DataColumn) e può includere vincoli di integrità referenziale e di univocità dei dati.

Il data adapter è una classe che deriva dal tipo base `DbDataAdapter`. Ogni data provider presenta una sua implementazione specifica, ma tutte le specializzazioni includono i membri utili al popolamento e all'aggiornamento di un particolare container di dati. La [Tabella 10.2](#) riporta le proprietà e i metodi principali.

Tabella 10.2 - Membri principali della classe `DbDataAdapter`

Proprietà o metodo	Descrizione
<code>SelectCommand</code>	Proprietà che permette di impostare il comando di selezione delle informazioni provenienti dalla sorgente dati. Questo comando viene utilizzato dal data adapter durante l'operazione di popolamento del container di dati di destinazione (<code>DataSet</code> o <code>DataTable</code>).
<code>InsertCommand</code>	Proprietà che permette di impostare il comando di inserimento di nuovi record nell'ambito della sorgente dati, utilizzato durante l'operazione di salvataggio (batch update).
<code>UpdateCommand</code>	Proprietà che permette di impostare il comando di aggiornamento dei record nell'ambito della sorgente dati, utilizzato durante l'operazione di salvataggio (batch update).
<code>DeleteCommand</code>	Proprietà che permette di impostare il comando di cancellazione dei record nell'ambito della sorgente dati, utilizzato durante l'operazione di salvataggio (batch update).
<code>Fill(DataSet)</code>	Metodo per il popolamento di un <code>DataSet</code> . Il metodo è soggetto a overloading.
<code>Fill(DataTable)</code>	Metodo per il popolamento di una <code>DataTable</code> . Il metodo è soggetto a overloading.
<code>Update(DataSet)</code>	Metodo per il salvataggio (batch update) del contenuto di un <code>DataSet</code> verso la sorgente dati. Il metodo è soggetto a overloading.
<code>Update(DataTable)</code>	Metodo per il salvataggio (batch update) del contenuto di una <code>DataTable</code> verso la sorgente dati. Il metodo è soggetto a overloading.

Dal momento che funge da tramite “da e verso” la sorgente dati, il data adapter include internamente quattro comandi (a cui corrispondono le quattro proprietà in tabella) che vengono invocati per le operazioni di lettura e di salvataggio dei dati ([esempio 10.12](#)). Sia durante l'operazione di popolamento sia durante quella di aggiornamento (detta anche **batch update**), il data adapter attiva, in

modo trasparente, una connessione verso la sorgente dati e invoca i comandi in relazione all'operazione che sta compiendo.

Esempio 10.12

```
var conn = new SqlConnection("...");

// In fase di creazione occorre specificare la connessione
var adapter = new SqlDataAdapter("SELECT * FROM Products", conn);

// Creazione della DataTable
var dt = new DataTable();

// Popolamento della DataTable
adapter.Fill(dt);

// Modifica dei dati contenuti nella DataTable...

// Batch update
adapter.Update(dt);
```

La connessione, associata al comando di selezione all'atto della chiamata del metodo di popolamento `Fill`, deve essere valida, ma non necessariamente aperta. Se la connessione risulta essere chiusa prima della chiamata della funzione di popolamento, essa viene automaticamente aperta e, successivamente, chiusa dal data adapter in modo del tutto trasparente. Se, invece, la connessione risulta essere aperta prima della chiamata del metodo `Fill`, essa viene lasciata aperta dal data adapter e deve essere chiusa in modo esplicito.

[Ricerca dati in un DataSet](#)

Una volta popolato, possiamo effettuare ricerche all'interno di un dataset attraverso **LINQ to DataSet**. Questi sono extension method aggiunti alla classe `DataTable` che ne trasformano le righe in una collection tipizzata e, successivamente, ne permettono la ricerca nello stile LINQ.

Il metodo che trasforma le righe in una collection ricercabile è `AsEnumerable`. Una volta ottenuta la collection, possiamo usare il metodo `Where` per filtrare i dati. Questo metodo accetta in input una funzione che prende la riga come parametro. Per recuperare i valori dei campi all'interno della riga, possiamo usare l'extension method `Field`, che accetta il tipo del campo, come parametro generico, e il nome. L'[esempio 10.13](#) mostra come filtrare i clienti del database.

Esempio 10.13

```
var ds =          // popola dataset
EnumerableRowCollection<DataRow> enumDt = c.Tables[0].AsEnumerable();
var custs = enumDt.Where(t => t.Field<string>("CustomerID").StartsWith("A");
```

Oltre al metodo `where`, ci sono anche i metodi `Select` e `OrderBy` che svolgono le medesime funzioni svolte dagli omonimi metodi LINQ di base. Il loro funzionamento è lo stesso visto per il metodo `where`.

Conclusioni

L'accesso ai dati è probabilmente la parte più importante di ogni applicazione. ADO.NET rappresenta il sottosistema di accesso ai dati all'interno di .NET. ADO.NET fornisce agli sviluppatori tutti gli strumenti necessari per accedere ai dati, per leggerli e per modificarli anche in un contesto transazionale. Grazie alla sua struttura a provider, ADO.NET permette di scrivere questo codice in maniera quasi agnostica rispetto al tipo di database e permette di creare facilmente specializzazioni di ADO.NET per ogni tipo di database (SQL Server, Oracle, MySql e altro ancora).

Sulla base di ADO.NET, Microsoft ha costruito un O/RM chiamato Entity Framework. L'utilizzo di un O/RM rende il nostro codice di accesso ai dati estremamente più pulito e robusto semplificandone lo sviluppo. Nel prossimo capitolo ci occuperemo di questo argomento

Accedere ai dati con Entity Framework Core

Nel capitolo precedente abbiamo illustrato come ADO.NET fornisca un'ottima base per accedere ai dati. Oggetti come `DbConnection`, `DbCommand`, `DbDataReader` e `DataSet` offrono tutto ciò di cui abbiamo bisogno per interagire con un database.

Tuttavia, lavorare con questi oggetti in maniera diretta nel nostro codice significa legarlo al database e alla sua struttura. Per questo motivo, le applicazioni moderne sfruttano una tecnica più elaborata per manipolare i dati. I dati vengono recuperati dal database utilizzando le classi di ADO.NET (in uno strato dedicato all'accesso ai dati) e riversati in un insieme di classi (l'insieme è noto come `object model`, mentre le classi sono note come `entity`) che vengono restituite all'applicazione. L'applicazione manipola i dati contenuti nelle classi dell'`object model` e demanda allo strato di accesso ai dati la loro persistenza sul database. In questo modo, la nostra applicazione lavora principalmente con l'`object model` senza preoccuparsi della struttura del database se non in uno strato dedicato. Grazie all'astrazione creata dall'`object model` e dallo strato di accesso ai dati, la nostra applicazione è agnostica rispetto al database e, quindi, più semplice sia da sviluppare che da mantenere.

Quando sviluppiamo applicazioni seguendo questo pattern, possiamo trovare un valido aiuto nei cosiddetti **Object/Relational Mapper** o **O/RM** (O/RM d'ora in poi). Un O/RM è un framework che permette di dialogare con il database astruendo le classi di ADO.NET e restituendo direttamente istanze di oggetti dell'`object model`. Permette anche di tracciare le modifiche fatte agli oggetti e di persisterle sul database.

Microsoft offre da anni un suo O/RM chiamato Entity Framework, di cui sono disponibili due versioni: Entity Framework 6 ed Entity Framework Core. La prima versione è quella pensata per girare su .NET Framework mentre la seconda su .NET Core. Tuttavia, insieme a .NET Core 3.0, Microsoft ha rilasciato una nuova versione di Entity Framework 6 (la 6.3) che gira anche su .NET Core. Questa nuova versione permette alle applicazioni che usano Entity Framework 6 su .NET Framework di essere più facilmente portate a .NET Core.

Essendo stato inizialmente sviluppato oltre dieci anni fa, Entity Framework 6 è molto più maturo di Entity Framework Core, ma è stato dichiarato completo e non verrà ulteriormente migliorato, mentre Entity Framework Core continuerà a essere attivamente sviluppato. Ci si può quindi ragionevolmente aspettare che in futuro Entity Framework Core sarà superiore a Entity Framework 6. I due framework sono molto simili nell'impostazione, ma profondamente diversi tra loro a livello di runtime e LINQ provider. In questo capitolo parleremo di entrambi i framework, evidenziandone le differenze.

Nel corso del capitolo, quando parleremo di Entity Framework intenderemo entrambi i framework. Quando parleremo di un framework specifico, ci riferiremo a questo come Entity Framework 6 (aggiornato alla versione 6.3 se non diversamente specificato) o Entity Framework Core. Inoltre, ricorrerà spesso la notazione parolachiave/parolachiave. Il primo valore indica il nome della classe, proprietà o metodo usato in Entity Framework 6, mentre il secondo valore indica quello usato in Entity Framework Core.

Partiamo tuttavia dallo spiegare cosa sia un O/RM e quale ne sia l'idea di base.

Cosa è un O/RM

Prima di iniziare a utilizzare uno dei due framework è bene accennare cosa sia un O/RM e quale idea risieda alla base di questo strumento di sviluppo. Come detto poco sopra, mascherare la struttura e l'interazione con il database dietro alle classi, permette di costruire applicazioni fortemente disaccoppiate dal database; questa è un'ottima cosa a livello di semplicità di sviluppo e manutenzione.

Attraverso l'uso di un O/RM possiamo creare delle classi che rappresentino il dominio della nostra applicazione, indipendentemente da come i dati sono

strutturati nel database. Sarà poi compito di uno specifico strato dell'applicazione tradurre i risultati delle query in oggetti e, viceversa, tradurre gli oggetti in comandi per aggiornare il database.

Prendendo come esempio il database Northwind, possiamo creare le classi Order, OrderDetail e Customer. In questo caso, le classi hanno una struttura speculare con le tabelle del database, ma non sempre è così. La teoria che sta dietro agli oggetti è completamente diversa dalla teoria che è alla base dei dati relazionali e questa diversità porta spesso (ma non sempre) ad avere classi diverse dalle tabelle. Il primo esempio di questa diversità risiede nella diversa **granularità**. Un cliente, in genere, ha un indirizzo di fatturazione e uno di spedizione (che possono coincidere o meno). Per rappresentare questi dati nel database, creiamo una tabella customers con i campi indirizzo, C.a.p., città e nazione ripetuti per entrambe le tipologie di indirizzo. Quando invece creiamo le classi, la cosa migliore è crearne una (AddressInfo) con le proprietà di un indirizzo e poi nella classe Customer aggiungere due proprietà (BillingAddress e ShippingAddress) di tipo AddressInfo. Questo significa avere una tabella lato database e due classi lato Object Model, come è mostrato nella [Figura 11.1](#).

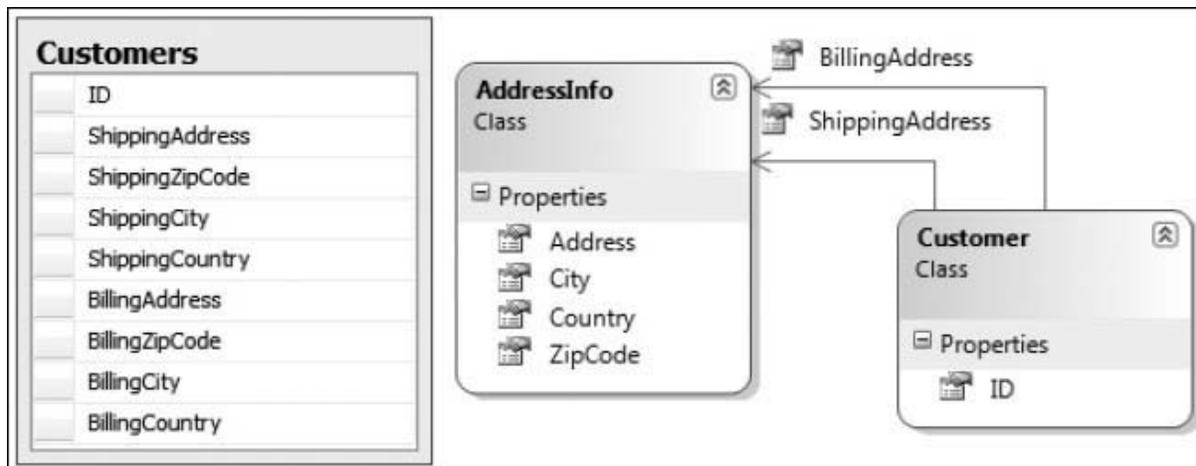


Figura 11.1 – La tabella clienti è descritta in due classi.

Un altro esempio è fornito dalla diversa modalità di relazione tra i dati. In un database, le relazioni tra i record sono mantenute tramite colonne con un vincolo di foreign key. Per esempio, per associare l'ordine a un cliente, mettiamo nella tabella degli ordini una colonna che contenga l'id del cliente. Nell'object model le relazioni si esprimono usando direttamente gli oggetti.

Quindi, per mantenere l'associazione tra l'ordine e il cliente, aggiungiamo la proprietà `customer` alla classe `order`, come è mostrato nella [Figura 11.2](#).

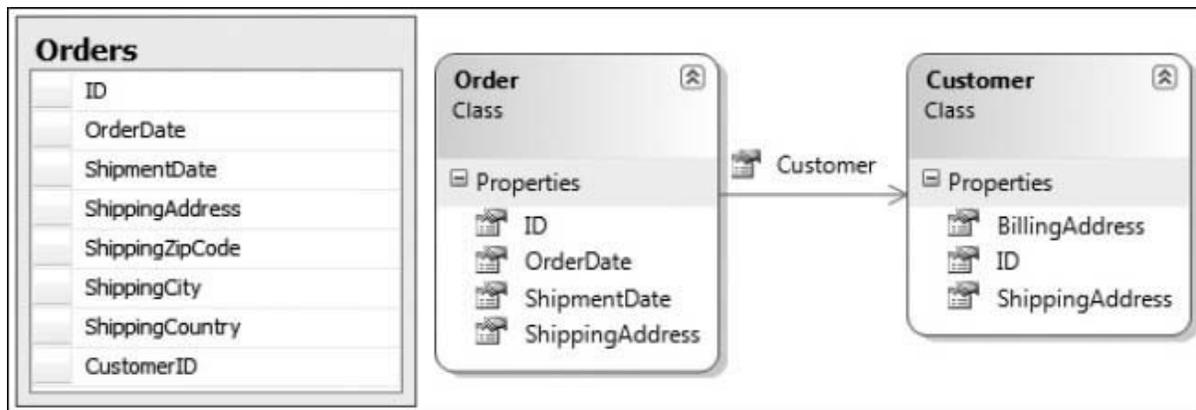


Figura 11.2 – La relazione tra ordini e clienti è mantenuta con una foreign key sul database e con una proprietà sul modello.

Le differenze si amplificano ulteriormente quando usiamo l'ereditarietà. Utilizzare questa tecnica nel mondo a oggetti è una cosa normalissima. Tuttavia, nel mondo relazionale non esiste il concetto di ereditarietà. Supponendo di avere un modello con le classi `customer` e `supplier` che ereditano dalla classe `company`, come possiamo avere una simile rappresentazione nel database? Possiamo sicuramente creare degli artifici che ci permettano poi di ricostruire le classi, ma si tratta comunque di accorgimenti volti a coprire una diversità di fondo tra il mondo relazionale e quello a oggetti.

Risolvere manualmente tutte queste complessità (e anche altre) non è affatto banale ed è per questo motivo che, da tempo, esistono dei framework che coprono queste e altre necessità. Questi framework prendono il nome di O/RM, in quanto lavorano come (M)apper tra (O)ggetti e dati (R)elazionali. In questo modo le diversità tra i due mondi sono gestite dall'O/RM, lasciandoci liberi di preoccuparci del solo codice di business.

Gli O/RM agiscono come mapper, cioè mappano le classi e le relative proprietà con le tabelle e le colonne nel database. Il vantaggio che ne deriva è nel fatto che possiamo evitare di scrivere query verso il database, ma possiamo scriverle verso l'Object Model in un linguaggio specifico dell'O/RM, che poi si occuperà di creare il codice SQL necessario. In termini di logica di business questo rappresenta un enorme vantaggio, in quanto gli oggetti rappresentano la logica in maniera molto più semplice delle tabelle. Lo stesso

ragionamento vale per gli aggiornamenti sul database. Noi ci preoccupiamo solo di modificare gli oggetti e poi demandiamo all'O/RM la persistenza di questi sul database. Ora dovrebbe essere più chiaro cosa significhi avere un'applicazione disaccoppiata dal database.

In conclusione, un O/RM è una parte di software molto potente ma, allo stesso tempo, molto complessa e pericolosa, poiché il livello di astrazione che introduce rischia di farci dimenticare che c'è un database, e questo è negativo. Dobbiamo sempre controllare le query generate dall'O/RM e verificare le istruzioni di manipolazione dati, per essere sicuri che le performance corrispondano ai requisiti.

Ora che abbiamo capito quali compiti svolge un O/RM possiamo vedere come questi compiti siano svolti da Entity Framework e come possiamo usare questo strumento per semplificare lo sviluppo del codice di accesso ai dati. Il primo passo consiste nel creare le classi del modello a oggetti per poi mapparle sul database.

Mappare il modello a oggetti sul database

Visto che la M di O/RM sta per Mapper, è facile immaginare che la fase di mapping tra il modello a oggetti e il database sia molto importante. Per mappare le classi verso il database, dobbiamo svolgere tre attività: prima scriviamo le classi, poi creiamo la classe di contesto (semplicemente contesto d'ora in poi) e quindi tramite quest'ultima mappiamo le classi.

L'operazione di mapping viene effettuata tramite codice utilizzando specifiche API oppure decorando con attributi le classi e le proprietà dell'object model. Inoltre, se scriviamo i nomi delle classi, delle loro proprietà e delle proprietà del contesto sfruttando determinate convenzioni, non abbiamo nemmeno la necessità di scrivere il codice di mapping, in quanto Entity Framework è già in grado di dedurre automaticamente dai nomi come il modello debba essere mappato. Le API permettono di effettuare qualunque tipologia di mapping Entity Framework metta a disposizione, mentre gli attributi e le convenzioni permettono di mappare solo un sottoinsieme delle possibilità di Entity Framework. Per questo motivo parleremo in maniera approfondita solo del mapping tramite API. Vediamo ora come creare e mappare le classi introdotte nelle precedenti sezioni.

Disegnare le classi

Scrivere una classe dell'object model è estremamente semplice in quanto consiste nel creare una classe con delle proprietà, esattamente come faremmo per qualunque altra classe. Non è richiesta alcuna integrazione con Entity Framework, come possiamo notare nel codice dell'[esempio 11.1](#).

Esempio 11.1

```
public class AddressInfo
{
    public string Address { get; set; }
    public string City { get; set; }
    public string Region { get; set; }
    public string PostalCode { get; set; }
    public string Country { get; set; }
}

public partial class Customer
{
    public string CustomerId { get; set; }
    public string CompanyName { get; set; }
    public AddressInfo Address { get; set; }

    public ICollection<Order> Orders { get; set; } = new HashSet<Order>();
}

public class Order
{
    public int OrderId { get; set; }
    public string CustomerId { get; set; }
    public AddressInfo ShipAddress { get; set; }

    public Customer Customer { get; set; }
    public ICollection<OrderDetail> OrderDetails { get; set; } =
        new HashSet<OrderDetail>();
}

public partial class OrderDetail
{
    public int OrderId { get; set; }
    public int ProductId { get; set; }
    public decimal UnitPrice { get; set; }
    public short Quantity { get; set; }
    public float Discount { get; set; }

    public Order Order { get; set; }
}
```

Il codice mostra chiaramente alcune caratteristiche del nostro modello:

- ❑ le classi sono semplici classi POCO (Plain Old CLR Object), con proprietà che rappresentano i dati e nessuna relazione con l'O/RM.

Questo modello potrebbe essere usato da Entity Framework così come da altri O/RM senza bisogno di alcuna modifica;

- ❑ le relazioni sono espresse tramite proprietà (dette **Navigation Property**) che si riferiscono direttamente a un oggetto in caso di relazioni uno-a-uno (come da dettaglio a ordine), o una lista di oggetti in casi di relazioni uno-a-molti (come da ordine a dettagli).
- ❑ il tipo `AddressInfo` è un tipo senza una chiave, referenziato da altri oggetti. Questo tipo di classe, denominata **complex type** in Entity Framework 6 e **owned type** in Entity Framework Core, agisce come semplice contenitore di proprietà e non come tipo da mappare verso una tabella;
- ❑ anche se non presenti nell'esempio, gli `enum` sono supportati come qualunque altro tipo nativo.

Ora che abbiamo visto il codice dell'object model andiamo a vedere il codice del contesto.

Creare il contesto

Il contesto è la classe che agisce da ponte tra il mondo a oggetti dell'object model e il mondo relazionale del database. Infatti, è attraverso questa classe che possiamo mappare l'object model verso il database ed effettuare tutte le operazioni, siano esse query o modifiche di dati negli oggetti.

Il contesto è una classe che eredita da `DbContext` e che definisce una proprietà di tipo `DbSet<T>`, chiamata **entityset**, per ogni classe dell'object model mappata verso una tabella del database (il tipo `T` corrisponde al tipo della classe). Nel nostro caso, il contesto conterrà tre proprietà: una per la classe `Customer`, una per la classe `Order` e una per la classe `Order_Detail`. Queste proprietà rappresentano il punto di entrata per recuperare e modificare oggetti nel database.

Nel caso in cui usiamo l'ereditarietà nel nostro object model (ad esempio una classe `Company` da cui derivano `Customer` e `Supplier`), abbiamo un solo entityset per tutta la gerarchia. Il tipo dell'entityset è quello della classe base.

L'[esempio 11.2](#) mostra il codice del contesto.

Esempio 11.2

```
public class NorthwindContext : DbContext
{
    public DbSet<Customer> Customers { get; set; }
    public DbSet<OrderDetail> OrderDetails { get; set; }
    public DbSet<Order> Orders { get; set; }
}
```

Se usiamo Entity Framework 6 su .NET Core, è importante creare un costruttore che accetti la stringa di connessione come parametro e che richiami il costruttore base con la stessa firma. Questo passo, mostrato nell'[esempio 11.3](#), è necessario in quanto al momento non c'è altro modo di passare la stringa di connessione al DbContext.

Esempio 11.3

```
public class NorthwindContext : DbContext
{
    public MyContext(string connectionString) : base(connectionString) { }
    ...
}
```

Sebbene il team non lo abbia esplicitamente specificato, è probabile che nella versione finale di Entity Framework 6 per .NET Core verranno aggiunti altri modi per configurare la stringa di connessione.

Ora che abbiamo visto come creare la classe di contesto, vediamo come eseguire il mapping delle classi verso il database attraverso questa classe.

[Mapping tramite convenzioni](#)

Quando un contesto viene istanziato la prima volta e viene eseguita la prima operazione, Entity Framework esegue il codice di mapping. Per prima cosa vengono analizzate le proprietà di tipo `DbSet<T>` del contesto per recuperarne le relative classi e verificare se queste rispettino determinate convenzioni che

ne permettono il mapping senza necessità di scrivere codice. Per chiarire meglio questo concetto, analizziamo le convenzioni applicandole alla classe `Order`:

- ❑ La classe viene mappata verso una tabella che ha il nome dell'entityset. Nel nostro caso, la classe viene mappata verso la tabella `Orders`;
- ❑ Le colonne della tabella hanno lo stesso nome delle proprietà della classe. Nel caso di proprietà all'interno di owned/complex type, il nome del campo sulla tabella viene calcolato unendo il nome della proprietà di tipo owned type con quello della proprietà al suo interno e separandoli con il carattere “_”. Nel nostro caso, la proprietà `city` all'interno della proprietà `ShipAddress` viene mappata sulla colonna `ShipAddress_City`. Se un owned/complex type contiene a sua volta un altro type, i nomi vengono concatenati ricorsivamente;
- ❑ Se una proprietà si chiama, indipendentemente dal case, `ID`, `Key`, `{NomeClasse}ID` o `{NomeClasse}Key` (dove il segnaposto `NomeClasse` viene rimpiazzato dal nome della classe che contiene la proprietà) questa viene automaticamente eletta a chiave primaria della classe. Se la proprietà è di tipo intero, questa è trattata come `Identity`. Nel nostro caso, la proprietà `orderId` è automaticamente identificata come chiave primaria;
- ❑ Il tipo del campo su cui la proprietà è mappata è analogo al tipo della proprietà (`int` per i tipi `Int32`, `bit` per i tipi `Boolean`, e così via). Le proprietà di tipo `Nullable<T>` e le proprietà di tipo `string` sono considerate `null` sul database, le altre proprietà sono considerate `not null`. Infine, le proprietà di tipo `string` sono considerate `Unicode` a lunghezza massima. Nel nostro caso, la proprietà `customerId` è mappata su una colonna di tipo `int`, mentre la proprietà `shipName` è mappata su una colonna di tipo `nvarchar`;
- ❑ Le navigation property con una reference uno-a-uno vengono automaticamente mappate utilizzando come nome della colonna il nome della proprietà chiave della classe a cui la proprietà si riferisce. Nel nostro caso, la navigation property `customer` punta a un oggetto di tipo `customer` la cui proprietà chiave è `customerId`. Questo significa che

Entity Framework sfrutta la colonna `CustomerId` nella tabella `Orders` per mappare la relazione tra le due entità.

Alla luce di queste convenzioni appare chiaro che buona parte del codice di mapping può essere gestito automaticamente dalle convenzioni. Tuttavia, c'è una parte di mapping che va gestita a mano come la lunghezza massima delle stringhe, la configurazione dei nomi dei campi quando sono diversi dalle proprietà, la configurazione di chiavi primarie composte da più proprietà, gli indici, le foreign key e altro ancora.

Mapping tramite API

Il mapping tramite API è molto simile tra i due framework, ma ci sono comunque delle differenze: in questa sezione vedremo come eseguire il mapping evidenziando queste differenze. Per mappare una classe verso il database usando le API di Entity Framework, dobbiamo eseguire l'override del metodo `OnModelCreating` nella classe di contesto. Questo metodo accetta in input un oggetto di tipo `DbModelBuilder/ModelBuilder`. Il metodo principale di questa classe è `Entity`; in Entity Framework questo metodo accetta il tipo della classe da mappare come parametro generico e ritorna un oggetto di tipo `EntityTypeConfiguration<T>` che espone i metodi per mappare l'entity. In Entity Framework Core, il metodo `Entity` accetta il tipo della classe da mappare come parametro generico e un metodo che specifica il mapping della classe. Questo metodo prende in input un oggetto di tipo `EntityTypeBuilder<T>`, dove `T` rappresenta la classe, tramite il quale possiamo eseguire tutte le operazioni di mapping della classe. I metodi principali della classe `EntityTypeConfiguration<T>/EntityTypeBuilder<T>` sono esposti nella [Tabella 11.1](#).

Tabella 11.1 – Metodi principali di mapping di `EntityTypeConfiguration<T>/EntityTypeBuilder<T>`

Metodo	Scopo
<code>Property</code>	Accetta una lambda che rappresenta una proprietà della classe e restituisce un oggetto che la rappresenta e sul quale possiamo eseguire metodi per mapparne le caratteristiche (nome della colonna sulla tabella, dimensioni, precisione, nullabilità e così via).
<code>HasIndex</code>	Accetta una lambda che rappresenta una o più proprietà che formano un

	indice sulla tabella e restituisce un oggetto su cui applicare metodi di mapping per specificare alcune caratteristiche dell'indice (nome, univocità). Valido solo per Entity Framework Core.
<code>ToTable</code>	Specifica il nome della tabella su cui la classe viene mappata nel database
<code>OwnsOne</code>	Accetta una lambda che rappresenta una proprietà della classe che si riferisce a un owned type e restituisce un oggetto che la rappresenta. Su questo oggetto possiamo usare metodi per mappare le proprietà dell'owned type. Questi metodi sono quasi gli stessi visti in questa tabella. Valido solo per Entity Framework Core.
<code>HasOne - HasMany</code>	Accettano una lambda che rappresenta una navigation property della classe e restituisce un oggetto che permette di specificarne le caratteristiche di mapping (nome della colonna sul database, relazione con la classe corrispondente, nome della foreign key). Valido solo per Entity Framework Core.
<code>HasRequired -HasOptional -HasMany</code>	Come sopra ma validi per Entity Framework 6.
<code>HasFilter</code>	Accetta una lambda che specifica un filtro che verrà applicato ad ogni query che riguarda la tabella senza bisogno di specificarlo in ogni query. Valido Solo per Entity Framework Core.
<code>HasKey</code>	Accetta una lambda che rappresenta una o più proprietà che formano la chiave primaria della tabella e restituisce un oggetto su cui applicare metodi di mapping per specificare alcune caratteristiche della chiave (nome, clustered se il database è SqlServer)

I metodi di `EntityTypeConfiguration<T>/EntityTypeBuilder<T>` si dividono in due categorie: quelli che mappano informazioni tra la classe e la tabella (`HasKey`, `ToTable`, `HasFilter`) e quelle che tornano le proprietà per poi specificarne il mapping (`Property`, `OwnsOne`, `HasOne`, `HasMany` e così via). Vediamo nella [Tabella 11.2](#) quali sono i metodi di mapping relativi alle proprietà.

Tabella 11.2 – Metodi di mapping

Metodo	Applicabile su	Scopo
<code>HasMaxLength</code>	<code>String</code>	Specifica la lunghezza massima del campo
<code>IsUnicode</code>	<code>String</code>	Specifica se il campo supporta caratteri Unicode
<code>HasColumnName</code>	Tutti i tipi	Specifica il nome del campo mappato
<code>HasColumnType</code>	Tutti i tipi	Specifica il tipo del campo mappato

IsRequired	Tutti i tipi	Specifica se il campo è null o meno (not null per default)
IsConcurrencyToken	Tutti i tipi	Specifica che la proprietà fa parte del token per gestire la concorrenza ottimistica negli aggiornamenti
IsRowVersion	Tutti i tipi	Specifica che la proprietà contiene la versione della riga (ogni database può interpretare il campo mappato in modo diverso). Valido solo per Entity Framework Core.
ValueGeneratedNever	Tutti i tipi	Specifica che il valore della proprietà viene generato dal client (la nostra applicazione). Valido solo per Entity Framework Core.
ValueGeneratedOnAdd	Tutti i tipi	Specifica che il valore della proprietà può essere generato dal client in fase di inserimento, ma spetta al database decidere se usare quello del client o generarne un altro. Per fare un esempio, se usiamo un'identity con SqlServer, un eventuale valore fornito dal client viene scartato e sostituito con quello generato dal server. Valido solo per Entity Framework Core.
ValueGeneratedOnAddOrUpdate	Tutti i tipi	Specifica che il valore della proprietà può essere generato dal client sia in fase di inserimento che di aggiornamento. Così come per ValueGeneratedOnAdd, spetta al database decidere se usare il valore inviato dal del client o generarne un altro. Valido solo per Entity Framework Core.
HasDefaultValueSql	Tutti i tipi	Specifica che il valore di default della proprietà sul database. Valido solo per Entity Framework Core.

I metodi di mapping appena elencati sono agnostici rispetto al tipo di database. I provider per uno specifico tipo di database possono aggiungere ulteriori metodi specifici (identity, sequence e così via).

Ora che abbiamo illustrato i metodi di mapping, vediamo come applicarli per mappare la classe `AddressInfo` all'interno del metodo `OnModelCreating`.

Esempio 11.4

```
//Entity Framework 6
modelBuilder.ComplexType<AddressInfo>().Property(e =>
    e.City).HasMaxLength(15);
modelBuilder.ComplexType<AddressInfo>().Property(e =>
    e.Country).HasMaxLength(15);

//Entity Framework Core
modelBuilder.Entity<AddressInfo>(entity =>
{
    entity.Property(e => e.City).HasMaxLength(15);
    entity.Property(e => e.Country).HasMaxLength(15);
});
```

In questo esempio sfruttiamo il metodo `Property` per recuperare il riferimento a una proprietà del complex/owned type, e specifichiamo la massima lunghezza sfruttando il metodo `HasMaxLength`. Il tipo `AddressInfo` è piuttosto semplice da mappare quindi possiamo passare al prossimo: `Customer`.

Esempio 11.5

```
//Entity Framework 6
modelBuilder.Entity<Customer>().HasKey(e => e.CustomerId);
modelBuilder.Entity<Customer>().Property(e => e.CustomerId)
    .HasColumnType("nchar(5)")
    .ValueGeneratedNever();

modelBuilder.Entity<Customer>().Property(e => e.Address.Address)
    .HasColumnName("Address");
modelBuilder.Entity<Customer>().Property(e => e.Address.City)
    .HasColumnName("City");

modelBuilder.Entity<Customer>().Property(e => e.CompanyName)
    .IsRequired()
    .HasMaxLength(40);
});

//Entity Framework Core
modelBuilder.Entity<Customer>(entity =>
{
    entity.HasKey(e => e.CustomerId);

    entity.Property(e => e.CustomerId)
        .HasColumnType("nchar(5)")
        .ValueGeneratedNever();

    entity.OwnsOne(e => e.Address)
        .Property(e => e.Address)
        .HasColumnName("Address");

    entity.OwnsOne(e => e.Address)
        .Property(e => e.City)
        .HasColumnName("City");

    entity.Property(e => e.CompanyName)
        .IsRequired();
});
```

```
.HasMaxLength(40); });
```

La classe `Customer` è decisamente più complessa rispetto a `AddressInfo`. Innanzitutto, specifichiamo che la proprietà che mappa sulla chiave primaria è `CustomerId`. Questo mapping non sarebbe necessario in quanto il framework è in grado di capirlo dalle convenzioni, ma è comunque riportato per dare un esempio del suo utilizzo.

Le istruzioni successive mappano le proprietà del complex/owned type `AddressInfo` verso la tabella `Customers`. Qui tra i due framework cambia il modo in cui recuperiamo le proprietà. Se usiamo Entity Framework 6, recuperiamo la proprietà tramite una lambda mentre, se usiamo Entity Framework Core, recuperiamo prima la proprietà `Address` all'interno di `Customer` e poi le sue proprietà interne. A prescindere dal framework, una volta recuperata la proprietà da mappare, usiamo il metodo `HasColumnName` per mapparla verso la relativa colonna sul database. Quest'operazione è necessaria in quanto in sua assenza entrambi i framework utilizzerebbero la convenzione predefinita e userebbero i nomi `Address_City`, `Address_Address`, e così via, come nomi delle colonne sul database, generando quindi un'eccezione a runtime, in quanto queste colonne non esistono.

Oltre a impostare i nomi delle colonne, usiamo i metodi `IsRequired` e, di nuovo, `HasMaxLength` per specificare ulteriori caratteristiche delle proprietà.

Ora cambiamo classe e analizziamo il mapping di `Order` di cui mostriamo un estratto nell'[esempio 11.6](#).

Esempio 11.6

```
//Entity Framework 6
modelBuilder.Entity<Order>().HasKey(e => e.OrderId);
modelBuilder.Entity<Order>().HasIndex(e => e.CustomerId)
    .HasName("CustomersOrders");

modelBuilder.Entity<Order>().Property(e => e.OrderID)
    .HasColumnName("OrderID");

modelBuilder.Entity<Order>().HasRequired(d => d.Customer)
    .WithMany(p => p.Orders);
});

//Entity Framework Core
modelBuilder.Entity<Order>(entity =>
{
    entity.HasKey(e => e.OrderId);
    entity.HasIndex(e => e.CustomerId).HasName("CustomersOrders");
    entity.Property(e => e.OrderId).HasColumnName("OrderID");
});
```

```
entity.HasOne(d => d.Customer).WithMany(p => p.Orders);
});
```

Il mapping della classe `Order` utilizza metodi già visti nei precedenti esempi. Infatti vengono prima specificati gli indici e poi mappate le proprietà.

Il metodo più interessante è `HasRequired/HasOne`. Questo viene utilizzato per mappare la navigation property `Customer` verso l'omonima classe. Per convenzione, Entity framework usa `CustomerId` come nome della colonna che agisce da foreign key verso la tabella `Customers`. Questo perché la colonna che rappresenta la chiave primaria sulla tabella `Customers` si chiama `CustomerId`. Possiamo modificare questo comportamento usando il metodo `HasForeignKey` e passando in input il nome della colonna sulla tabella `Orders`. Il successivo metodo `WithMany` specifica che la navigation property `Orders` di `Customer` è mappata verso `Order` sfruttando la stessa foreign key. Ora non rimane che mappare la classe `OrderDetail`.

Esempio 11.7

```
//Entity Framework 6
modelBuilder.Entity<OrderDetail>().ToTable("Order Details");
modelBuilder.Entity<OrderDetail>().HasKey(e => new {e.OrderId,e.ProductId });
modelBuilder.Entity<OrderDetail>().Property(e => e.OrderId)
    .HasColumnName("OrderID");
modelBuilder.Entity<OrderDetail>().Property(e => e.ProductId)
    .HasColumnName("ProductID");
modelBuilder.Entity<OrderDetail>().Property(e => e.Discount);
modelBuilder.Entity<OrderDetail>().Property(e => e.Quantity);
modelBuilder.Entity<OrderDetail>().Property(e => e.UnitPrice)
    .HasColumnType("money");

modelBuilder.Entity<OrderDetail>().HasRequired(d => d.Order)
    .WithMany(p => p.OrderDetails);

//Entity Framework Core
modelBuilder.Entity<OrderDetail>(entity =>
{
    entity.ToTable("Order Details");
    entity.HasKey(e => new { e.OrderId, e.ProductId });
    entity.Property(e => e.OrderId).HasColumnName("OrderID");
    entity.Property(e => e.ProductId).HasColumnName("ProductID");
    entity.Property(e => e.Discount).HasDefaultValueSql("((0))");
    entity.Property(e => e.Quantity).HasDefaultValueSql("((1))");
    entity.Property(e => e.UnitPrice)
        .HasColumnType("money")
        .HasDefaultValueSql("((0))");

    entity.HasOne(d => d.Order)
        .WithMany(p => p.OrderDetails)
        .onDelete(DeleteBehavior.ClientSetNull)
});
```

Il mapping di `OrderDetail` utilizza altri metodi molto importanti. Il primo è `toTable` che specifica il nome della tabella verso cui la classe mappa. In questo caso, siamo obbligati a usare questo metodo in quanto il nome della tabella dedotto dalle convenzioni, `OrderDetails`, è errato.

Il secondo è `HasKey`. Sebbene abbiamo già visto questo metodo prima, in questo caso il suo utilizzo mostra come specificare una chiave composta da più proprietà. Infatti, il metodo non ritorna una sola proprietà, bensì un `anonymous type` con le proprietà che fanno parte della chiave primaria. Questa stessa tecnica è utilizzabile anche nel metodo `HasIndex` per specificare indici composti.

Infine, solo per Entity Framework Core, il mapping delle relazioni mostra anche come specificare il cascading attraverso il metodo `onDelete`.

Come detto in precedenza, oltre che tramite convenzioni e API, esiste un terzo metodo di mapping che utilizza le data annotation e di cui ci occuperemo nella prossima sezione.

[Mapping tramite data annotation](#)

Il mapping tramite le data annotation prevede che in testa alla classe e alle proprietà siano presenti alcuni attributi che vengono interpretati dal motore di Entity Framework. Questi attributi permettono di specificare il nome della tabella su cui una classe mappa, il nome della colonna su cui una proprietà mappa, il tipo specifico della colonna, se è una `primary key`, se è obbligatoria, la sua lunghezza massima e altro ancora. Sebbene sia comodo, il mapping tramite data annotation non copre tutte le esigenze come invece fa il mapping tramite API. La [Tabella 11.3](#) mostra le principali data annotation.

Tabella 11.3 – Data annotation per il mapping

Attributo	Applicabile su	Scopo
<code>Index</code>	Classe	Specifica che la proprietà fa parte di un indice
<code>Table</code>	Classe	Specifica la tabella su cui la classe mappa
<code>Key</code>	Proprietà	Specifica che la proprietà fa parte della chiave primaria
<code>Column</code>	Proprietà	Specifica il nome e il tipo della colonna su cui la proprietà mappa
<code>DatabaseGenerated</code>	Proprietà	Specifica se la colonna è su cui la proprietà mappa è un'identity, calcolata o normale

Required	Proprietà	Specifica che la proprietà è obbligatoria
StringLength	Proprietà	Specifica la lunghezza massima della proprietà

A prescindere dalla tecnica di mapping utilizzata, abbiamo scritto il codice necessario per cominciare a utilizzare Entity Framework. Tuttavia, finora siamo ricorsi alla scrittura manuale del codice, ma Entity Framework permette di generare il codice partendo dalle tabelle del database.

Generare automaticamente le classi dal database

Quando abbiamo già il database a disposizione, Entity Framework mette a disposizione degli strumenti per generare l'object model e il suo mapping partendo dal database. Questi strumenti variano a seconda del framework che usiamo.

Se usiamo Entity Framework 6 su .NET Framework, possiamo utilizzare il wizard di Visual Studio. Questo wizard permette di selezionare prima il database a cui connettersi e poi quali siano le tabelle, viste, funzioni e stored procedure per cui generare l'object model. Una volta completato il wizard, Visual Studio mette a disposizione anche un designer che rappresenta le classi dell'object model e le relazioni tra queste, offrendo anche la possibilità di effettuare modifiche e di aggiornarne le proprietà qualora il database venga modificato. Sia il wizard sia il designer si basano su un file XML chiamato EDMX che contiene la descrizione delle classi, del database e il relativo mapping. In fase di compilazione questo file viene suddiviso in tre file separati che vengono embeddati nell'assembly e letti a runtime dal `DbContext`. La conseguenza dell'uso del wizard e del designer è che il mapping da codice non è necessario in quanto specificato nel file EDMX.

A seconda delle nostre scelte, possiamo istruire il wizard per generare non il file EDMX, bensì direttamente le classi dell'object model e il relativo codice di mapping. La conseguenza di questa scelta è l'assenza del designer e l'utilizzo del solo codice.

Al momento della stesura del libro, se usiamo Entity Framework 6 su .NET Core, il wizard e il designer non sono disponibili e quindi non possiamo usarli per la generazione del codice. Inoltre, stando alle metriche di utilizzo, il wizard

e il designer sono poco utilizzati e quindi, per Entity Framework Core, il team ha deciso di non renderli disponibili.

Per sopperire alla mancanza di questi strumenti, con Entity Framework Core abbiamo a disposizione il comando `scaffold-dbcontext`. Questo comando prende in input la stringa di connessione al database, il provider di Entity Framework da usare e genera una classe per ogni tabella mappando automaticamente le proprietà, le navigation property e generando anche la classe di contesto con tutti gli entityset.

Oltre ai parametri base, possiamo specificare anche di quali tabelle vogliamo eseguire il mapping, in quale cartella mettere i file generati, il nome del contesto, se creare un log e altro ancora. Per conoscere tutti i parametri rimandiamo alla pagina della documentazione disponibile su <http://aspit.co/bk0>.

Nell'[esempio 11.8](#) possiamo vedere il comando da utilizzare per generare le classi, il contesto e il mapping dell'object model che abbiamo analizzato.

Esempio 11.8

```
scaffold-dbcontext
-connection "connection string"
-provider "Microsoft.EntityFrameworkCore.SqlServer"
-tables "Customers", "Order Details", "Orders"
-output "data"
-context "NorthwindContext"
-verbose
```

Questo comando va lanciato dalla finestra Package Manager Console di Visual Studio all'interno della quale va prima selezionato su quale progetto creare i file e poi lanciato il comando.

Se non usiamo Visual Studio, possiamo usare le estensioni di Entity Framework per la CLI di .NET: `dotnet ef`. Queste estensioni sono installabili da NuGet e vengono installate globalmente; quindi, oltre a specificare i parametri relativi a stringhe di connessione, provider, tabelle e altro ancora, dobbiamo anche posizionarci con il prompt nella directory dove risiede il progetto in cui vogliamo creare i file.

Esempio 11.9

```
dotnet ef dbcontext scaffold
"connection string"
Microsoft.EntityFrameworkCore.SqlServer
-t "Customers", "Order Details", "Orders"
-o "data"
-c "NorthwindContext"
-v
```

Il generatore creerà classi e proprietà mappandole una ad una con le tabelle e le colonne del database. Questo significa che non prende in considerazione la creazione di owned type come `AddressInfo`.

Oltre al comando per generare le classi e il contesto partendo dal database, ne esistono molti altri. Esistono comandi per manipolare e applicare le migrazioni, per eliminare il database e per ottenere informazioni sui contesti presenti nel progetto.

Avere a disposizione il codice di mapping (sia esso scritto a mano o generato da Entity Framework) è la prima parte di ciò che serve per lavorare con Entity Framework mentre la seconda è il recupero e la configurazione del contesto.

Istanziare il contesto

Il modo di configurare e istanziare il contesto cambia di molto in base al framework e all'ambiente. La prima cosa da configurare è la stringa di connessione e questo possiamo farlo attraverso il file di configurazione.

Se usiamo Entity Framework 6 su .NET Framework, aggiungiamo la stringa di connessione nella sezione `connectionStrings` del file `web.config` o `app.config`, a seconda che stiamo creando un'applicazione web o no. Per minimizzare il codice, il nome della chiave della stringa di connessione deve corrispondere al nome della classe del contesto. Questo fa sì che quando istanziamo il contesto, questo sia automaticamente in grado di ritrovare autonomamente la stringa di connessione.

Se invece usiamo Entity Framework Core, mettiamo la stringa di connessione nella sezione `ConnectionStrings` del file `appsettings.json`. Nel prossimo esempio mostriamo le stringhe di connessione nei diversi file.

Esempio 11.10

```

--app.config o web.config
<configuration>
  <connectionStrings>
    <add name="NorthwindContext"
          connectionString="
            Server=dbserver;
            Database=northwind;Integrated security=true" />
  </connectionStrings>
</configuration>

--appsettings.json
{
  "ConnectionStrings": {
    "NorthwindContext":
      "Server=dbserver;Database=northwind;Integrated
        security=true"
  }
}

```

Con Entity Framework Core, se ci troviamo in un'applicazione Windows dobbiamo eseguire l'override del metodo `onConfiguring` e al suo interno invocare il metodo `useSqlServer` (o equivalente per altri database) passando in input la stringa di connessione presa dalla configurazione. Se invece ci troviamo in un'applicazione web .NET Core, nel metodo `ConfigureServices` della classe `Startup` dobbiamo invocare il metodo `AddDbContext` che aggiunge al motore di dependency injection il contesto. Questo metodo accetta in input una funzione all'interno della quale usiamo il metodo `useSqlServer` (o equivalente per altri database) per impostare la stringa di connessione.

Esempio 11.11

```

--Entity Framework Core su Windows
public class NorthwindContext : DbContext
{
  protected override void OnConfiguring(DbContextOptionsBuilder o)
  {
    var connectionString = "stringa di connessione da configurazione";
    o.UseSqlServer(connectionString);
  }
});

--Entity Framework Core su ASP.NET Core
public class Startup
{
  public void ConfigureServices(IServiceCollection services)
  {
    services.AddDbContext<NorthwindContext>(o =>
    {
      var connectionString = "stringa di connessione da configurazione";
      o.UseSqlServer(connectionString);
    });
  }
}

```

A questo punto, siamo pronti per creare un'istanza del contesto. Con Entity Framework 6 su .NET Framework, possiamo ottenere un'istanza del contesto utilizzando il suo costruttore senza parametri, a patto che la chiave della stringa di connessione abbia lo stesso nome del contesto, indipendentemente dal fatto che ci troviamo in un'applicazione web o Windows.

Con Entity Framework 6 su .NET Core, possiamo istanziare il contesto tramite il suo costruttore, passando in input la stringa di connessione che possiamo ottenere dalla configurazione o in altro modo. La cosa più importante da ricordare è che va registrata la provider factory a mano nello startup dell'applicazione, che essa sia web o Windows.

Con Entity Framework Core, se siamo in un'applicazione Windows ci basta istanziare il costruttore usando `new`, mentre se siamo in un'applicazione ASP.NET Core, possiamo accettare il contesto come parametro nel costruttore delle nostre classi. Se siamo in un controller, possiamo anche passare il contesto come parametro di una action, decorando il parametro con l'attributo `FromServices`. In entrambi i casi sarà compito del motore di dependency injection di ASP.NET Core creare l'istanza e fornirla al nostro costruttore. Tutti i metodi di creazione del contesto sono mostrati nel prossimo esempio.

Esempio 11.12

```
--Entity Framework 6 su .NET Framework
class Program
{
    static void Main(string[] args)
    {
        using (var ctx = new NorthwindContext()) { ... }
    }
}

--Entity Framework 6 su app Windows basata .NET Core
class Program
{
    static void Main(string[] args)
    {
        DbProviderFactories.RegisterFactory("System.Data.SqlClient",
            SqlConnectionFactory.Instance);
        using (var ctx = new NorthwindContext()) { ... }
    }
}

--Entity Framework 6 su ASP.NET Core
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        DbProviderFactories.RegisterFactory("System.Data.SqlClient",
```

```

        SqlClientFactory.Instance);
    }
}

public class CustomersController() {
    public IActionResult MyAction()
    {
        using (var ctx = new NorthwindContext()) { ... }
    }
}

--Entity Framework Core su Windows
class Program
{
    static void Main(string[] args)
    {
        using (var ctx = new NorthwindContext()) { ... }
    }
}

--Entity Framework Core su ASP.NET Core
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddDbContext<NorthwindContext>(o =>
        {
            var connectionString = "stringa di connessione da configurazione";
            o.UseSqlServer(connectionString);
        });
    }
}

public class CustomersController() {
    private readonly NorthwindContext _ctx;

    public CustomersController(NorthwindContext ctx)
    {
        _ctx = ctx;
    }
}

```

La classe `DbContext` implementa l'interfaccia `IDisposable`. Se l'istanziamento avviene tramite il motore di dependency injection, questo si occupa anche di chiamare il metodo `Dispose`. Se siamo noi a istanziare il contesto tramite costruttore, allora dobbiamo anche preoccuparci di chiamare il metodo `Dispose` e lo facciamo implicitamente, usando il blocco `using` come abbiamo visto nell'esempio.

L'istanziamento del contesto è il primo passo verso la manipolazione dei dati sul database. Il prossimo passo è leggere i dati presenti sul database.

[Recuperare i dati dal database](#)

Ora che abbiamo il contesto, dobbiamo sfruttare i suoi entityset. Poiché il tipo `DbSet<T>` implementa indirettamente l'interfaccia `IQueryable<T>`, questo espone tutti gli extension method di LINQ. Questo non significa assolutamente che i dati siano in memoria. Il provider LINQ di Entity Framework intercetta l'esecuzione della query verso l'entityset e la trasforma in un *expression tree*, che viene poi convertito in SQL grazie alle informazioni di mapping. Il risultato è che noi scriviamo query LINQ e tutto il lavoro di conversione è affidato a Entity Framework. L'[esempio 11.13](#) mostra come sia semplice recuperare la lista dei clienti italiani.

Esempio 11.13

```
var customers1 = from c in ctx.Customers
                 where c.Address.Country == "Italy"
                 select c;

var customers2 = ctx.Customers.Where(c => c.Address.Country == "Italy");
```

L'esempio porta ad alcune importanti considerazioni. La prima è che non dobbiamo gestire né connessioni né transazioni né altri oggetti legati all'interazione col database. Entity Framework astrae tutto per noi, restituendoci direttamente oggetti.

La seconda è che il codice appena visto non esegue alcuna query. La *deferred execution* è perfettamente supportata dal provider LINQ di Entity Framework, quindi finché non enumeriamo fisicamente i risultati, la query non viene effettuata sul database. È importante sottolineare che a causa di questo comportamento, se eseguiamo la query quando il contesto è stato eliminato otteniamo un'eccezione a runtime. Il caso più comune in cui ci troviamo in questa situazione è quando assegniamo la variabile `customers1` a una proprietà del modello e poi nella view enumeriamo la proprietà. La query viene scatenata quando enumeriamo la proprietà, ma quando siamo nel contesto di esecuzione della view il nostro contesto è già stato eliminato in quanto siamo usciti dal suo scope.

La terza cosa da notare è che nelle query possiamo utilizzare sia la query syntax sia la normale sintassi basata sugli extension method, ma nel secondo caso dobbiamo prestare attenzione a quali extension method utilizzare. Il provider di Entity Framework non supporta tutti i metodi LINQ e i relativi overload poiché alcuni non trovano una controparte sui database, mentre altri non hanno la possibilità di essere tradotti in SQL. Gli extension method

disponibili sono quelli di aggregazione, di intersezione, di ordinamento, di partizionamento, di proiezione e di raggruppamento oltre a quelli di insieme.

Per esempio, quando vogliamo recuperare un solo oggetto, i metodi `First` e `Single` sono validi. La differenza risiede nel fatto che mentre `First` viene tradotto in una `TOP 1`, `Single` viene tradotto in una `TOP 2`, per verificare che non sia presente più di un elemento. Inoltre, se la ricerca dell'oggetto avviene per chiave primaria, possiamo anche utilizzare il metodo `Find` (o la sua controparte asincrona `FindAsync`) della classe `DbSet<T>` come mostrato nel prossimo esempio.

Esempio 11.14

```
var c1 = ctx.Customers.First(c => c.CustomerID == "ALFKI");
var c2 = ctx.Customers.Find("ALFKI");
var c3 = await ctx.Customers.FindAsync("ALFKI");
```

Una cosa che torna utilissima in LINQ sono le proiezioni. Molto spesso non abbiamo bisogno di tutta la classe, ma solo di alcuni suoi dati. Specificando in una proiezione quali proprietà dobbiamo estrarre (come nell'[esempio 11.15](#)), faremo in modo che il SQL generato estragga solo quelle, ottenendo così un'ottimizzazione delle performance.

Esempio 11.15

```
ctx.Customers.Select(c => new { c.CustomerID, c.CompanyName});
```

Esempio 11.15 – SQL

```
SELECT
    c.CustomerID,
    c.CompanyName
FROM [dbo].[Customers] AS c
```

Il provider LINQ di Entity Framework 6 è molto potente e riesce a convertire in modo corretto, anche se non sempre efficiente, anche query molto complesse. Il provider LINQ di Entity Framework Core è stato riscritto da zero ed è stato pensato per supportare molte più funzionalità rispetto a quelle

offerte dal provider di Entity Framework 6. Come conseguenza, le potenzialità del provider sono enormi in quanto traduce molti metodi LINQ in SQL, permette di mischiare parzialmente codice SQL e codice LINQ per generare una unica query SQL sul server, e altro ancora. Il provider ha sofferto di alcuni problemi di gioventù, ma con Entity Framework Core 3 la situazione è molto migliorata, in quanto la pipeline di generazione del codice SQL a partire da LINQ è stata rivista pesantemente e alcuni errori di impostazione sono stati risolti.

Tuttavia, a prescindere da quale framework si usi, è sempre bene controllare il codice SQL generato (utilizzando il profiler del database o il logging di Entity Framework), per essere sicuri che corrisponda ai propri criteri di qualità.

Per fare un esempio di quando sia necessario controllare il codice SQL, quando usiamo Entity Framework Core alcune query possono soffrire del problema 1+n rallentando vistosamente le performance dell'applicazione. Un caso in cui questo accade è quando usiamo una projection che include campi di una navigation property che punta a una lista di oggetti. Prendiamo in esame il seguente codice.

Esempio 11.16

```
var orders = ctx.Orders
    .Where(o => o.Customer.CustomerId == "ALFKI")
    .Select(c => new {
        CustomerId = c.Customer.CustomerId,
        Products = c.OrderDetails.Select(d => d.ProductId)
    })
    .ToList();    // query che estrae dati ordine
foreach (var order in orders)
{
    foreach (var p in order.Products)    // query che estrae prodotti per ordine
    {
    }
}
```

La query filtra gli ordini e per ogni ordine estrae l'id del cliente e gli id dei prodotti coinvolti nell'ordine. In questi casi ci si aspetta che venga eseguita una sola query che estrae i dati utilizzando una `join` SQL. In realtà, il motore esegue prima una query che estrae solo i dati degli ordini (`CustomerId` in questo caso), e poi, quando accediamo in ciclo alla lista dei prodotti per l'ordine corrente esegue una query per estrarre i prodotti. Questo significa che se la query torna 10 ordini, il codice esegue 1 query per estrarre i dati

dell'ordine, e poi una query per ogni ordine per estrarre i prodotti per un totale di 11 query. Fortunatamente possiamo aggirare il problema usando il metodo `ToList` sulla subquery dei prodotti. Questo forza il caricamento dei prodotti in una query sola, eliminando il problema.

Finora abbiamo parlato dei metodi sincroni `First`, `Single`, `ToList` e così via per eseguire le query. Entity Framework offre la possibilità di eseguire query asincrone tramite i metodi `ToListAsync`, `ToArrayAsync`, `FirstAsync`, solo per nominarne alcuni, su cui deve essere fatta l'`await` per aspettarne la fine dell'esecuzione. L'utilizzo di questi metodi è consigliato per ottimizzare l'utilizzo dei thread dell'applicazione, ma va specificato che la classe di contesto **non è thread safe**, quindi l'asincronia non può essere sfruttata per eseguire più query in parallelo. Se si lanciano query in parallelo, il funzionamento non è garantito e si possono ottenere eccezioni o addirittura dati non congruenti.

Ottimizzare il fetching

Quando recuperiamo i dati dal database, spesso abbiamo bisogno anche di altri dati collegati. Per fare un esempio, quando recuperiamo gli ordini, capita spesso di dover recuperare anche il cliente associato e i dettagli degli ordini. Nella maggioranza dei casi, la soluzione ideale è quella di recuperare tutti gli oggetti in una singola query. Il problema risiede nel fatto che Entity Framework ritorna solo gli oggetti specificati dall'`entityset`. Questo significa che quando eseguiamo una query sugli ordini, i dettagli vengono ignorati. Fortunatamente possiamo modificare questo comportamento e dire quali dati collegati vogliamo caricare insieme all'entità principale.

Per fare questo dobbiamo utilizzare il metodo `Include` della classe `DbSet`. Questo metodo, in input, accetta una lambda che specifica la navigazione property da caricare, così come è visibile nell'[esempio 11.17](#).

Esempio 11.17

```
ctx.Orders.Include(o => o.OrderDetails);  
ctx.OrderDetails.Include(o => o.Order)
```

Quando dobbiamo caricare navigation property che fanno parte dell'oggetto caricato tramite `Include`, (per esempio se vogliamo eseguire una query sugli

ordini e caricare contemporaneamente i dettagli e i prodotti legati a essi) la soluzione cambia a seconda del framework.

Con Entity Framework 6, se la proprietà caricata tramite la prima chiamata a `Include` punta a un oggetto singolo, per esempio se dal dettaglio carichiamo l'ordine, allora possiamo accedere alla proprietà di secondo livello semplicemente accedendovi come proprietà. Se invece la proprietà a cui accediamo con la prima `Include` è una lista, per esempio se dall'ordine carichiamo i dettagli, dobbiamo usare il metodo `Select`.

Con Entity Framework Core possiamo concatenare il metodo `ThenInclude` al metodo `Include`, specificando la lambda che accede alla seconda proprietà da caricare.

Esempio 11.18

```
--Entity Framework
ctx.OrderDetails.Include(o => o.Order.Customer);
ctx.Orders.Include(o => o.OrderDetails.Select(d => d.Product));

--Entity Framework Core
ctx.OrderDetails.Include(o => o.Order).ThenInclude(d => d.Customer);
ctx.Order.Include(o => o.OrderDetails).ThenInclude(c => c.Product);
```

Entity Framework 6 traduce le `Include` in una `join SQL`. Questa tecnica ha il vantaggio di estrarre tutti i dati in una sola query, ma ha anche lo svantaggio di recuperare una enorme quantità di dati duplicati. Entity Framework Core, al contrario, produce più query che da un lato causano una maggior lentezza nel recupero dei dati, ma ottimizza la quantità di dati tornati dal database.

Sebbene recuperare immediatamente le navigation property sia ottimale nella maggior parte dei casi, vi sono situazioni in cui è meglio recuperare solo l'entity principale e caricare le navigation property sfruttando il **lazy loading**.

Lazy loading

Tramite questa tecnica effettuiamo una query per recuperare una o più entity principali e ne carichiamo le navigation property solo se fisicamente vi accediamo. Per fare un esempio, potremmo dover recuperare tutti gli ordini di un giorno, ma recuperare i dettagli solo di quelli spediti in certi stati. In questo caso recuperiamo gli ordini in una query unica e solo quando accediamo ai dettagli viene effettuata una query da Entity Framework (per noi trasparente in

quanto accediamo semplicemente alla proprietà `OrderDetails`). Come affrontato nella precedente sezione, questo significa introdurre il problema del 1+n poiché si esegue una query per ogni ordine di cui recuperiamo i dettagli quindi è bene valutare l'uso del lazy loading.

Oltre a questa problematica, per supportare il lazy loading dobbiamo anche apportare modifiche al codice. Se usiamo Entity Framework 6, il lazy loading è abilitato di default. Se usiamo Entity Framework Core, dobbiamo aggiungere il package `Microsoft.EntityFrameworkCore.Proxies` e, nella configurazione del contesto, invocare il metodo `UseLazyLoadingProxies`.

Esempio 11.19

```
services.AddDbContext<NorthwindContext>(o =>
{
    o.UseLazyLoadingProxies();
});
```

Infine, per entrambi i framework, dobbiamo modificare le classi dell'object model rendendole tutte non sealed, e impostando come virtual tutte le navigation property. Questa operazione è necessaria in quanto a runtime Entity Framework crea un nuovo tipo (proxy) che eredita dalla nostra classe dell'object model e che sovrascrive getter e setter delle navigation property al fine di iniettare il codice necessario a effettuare il lazy loading (cioè andare sul database e recuperare i dati). L'[esempio 11.20](#) mostra come il lazy loading sia trasparente per il nostro codice.

Esempio 11.20

```
var orders = _ctx.Orders.ToList ();
foreach (var order in orders)
{
    if (order.ShipAddress.Country == "Italy")
    {
        foreach (var detail in order.OrderDetails)
        {
            }
        }
    }
}
```

Il risultato di questo codice è che per ogni ordine spedito in Italia, noi accediamo ai dettagli ed Entity Framework scatena una query in maniera trasparente per noi.

Lavorare con i proxy è l'unica possibilità per ottenere il lazy loading con Entity Framework 6. Con Entity Framework Core, invece, ci sono anche altre possibilità, come iniettare in un costruttore privato della nostra classe l'interfaccia `ILazyLoader` oppure un delegato di tipo `Action<object, string>` il cui nome deve essere `lazyLoader`. Grazie all'iniezione di questi oggetti, possiamo invocare il caricamento dei dati scrivendo noi il codice senza dover ricorrere ai proxy e a tutto quello che il loro utilizzo comporta. Per maggiori informazioni su queste tecniche rimandiamo alla documentazione disponibile all'indirizzo <http://aspit.co/bnu>.

A seconda della versione, Entity Framework offre altre funzionalità avanzate per il recupero dei dati, come la possibilità di mischiare codice LINQ e codice SQL (Entity Framework Core), l'invocazione di stored procedure e table-valued function direttamente dal contesto (Entity Framework 6) o tramite SQL (Entity Framework Core), il mapping di viste (entrambi i framework), il supporto agli Enum e molto altro ancora. Parlare di tutte le funzionalità relative alle query richiederebbe un intero libro, quindi rimandiamo alla documentazione ufficiale di Entity Framework all'indirizzo <https://aspit.co/buj>.

Salvare i dati sul database

Quando un oggetto viene recuperato dal database, viene anche memorizzato all'interno del contesto, in un componente chiamato **Change Tracker**.

Sono due i motivi per cui questo comportamento è necessario. Innanzitutto, ogni volta che eseguiamo una query, il contesto (che è responsabile della creazione degli oggetti) verifica che non esista già un oggetto corrispondente (ovvero con la stessa chiave primaria) nel change tracker. In caso affermativo, il record proveniente dal database viene scartato e il relativo oggetto nel change tracker viene restituito. In caso negativo, viene creato il nuovo oggetto, messo nel change tracker e, infine, restituito all'applicazione. Questo garantisce che vi sia un solo oggetto a rappresentare lo stesso dato sul database.

Inoltre, il contesto deve mantenere traccia di tutte le modifiche fatte agli oggetti, così che poi queste possano essere riportate sul database. Il fatto di

avere gli oggetti in memoria semplifica questo compito. Salvare i dati sul database significa persistere la cancellazione, l'inserimento e le modifiche delle entità. Questo processo viene scatenato attraverso la chiamata al metodo `saveChanges` (o la sua controparte asincrona `saveChangesAsync`) del contesto. Questo metodo non fa altro che scorrere gli oggetti modificati memorizzati nel contesto, iniziare una transazione, costruire ed eseguire il codice SQL per la persistenza e infine eseguire il commit o il rollback a seconda che vada tutto bene.

Se abbiamo più chiamate al metodo `SaveChanges` e vogliamo che queste siano eseguite in un'unica transazione, dobbiamo usare la classe `TransactionScope`.

Per poter generare il codice SQL corretto, il contesto deve conoscere lo stato di ogni singolo oggetto quando effettua il salvataggio. Un oggetto può essere in quattro stati:

- ❑ `Unchanged`: l'oggetto non è stato modificato. Il processo di persistenza non scatena alcun comando per gli oggetti in questo stato;
- ❑ `Modified`: qualche proprietà semplice dell'oggetto è stata modificata. Il processo di persistenza genera un comando SQL di `UPDATE` per ogni oggetto in questo stato includendo solo le colonne modificate;
- ❑ `Added`: l'oggetto è stato marcato per l'aggiunta sul database. Il processo di persistenza genera un comando SQL di `INSERT` per ogni oggetto in questo stato;
- ❑ `Deleted`: l'oggetto è stato marcato per la cancellazione dal database. Il processo di persistenza genera un comando SQL di `DELETE` per ogni oggetto in questo stato.

Quando un oggetto viene creato da una query, il suo stato è `Unchanged`. Nel momento in cui andiamo a modificare una proprietà semplice o una complessa, lo stato passa automaticamente a `Modified`. `Added` e `Deleted` si differenziano dagli stati precedenti, in quanto devono essere impostati manualmente. Per settare lo stato di un oggetto su `Added`, dobbiamo utilizzare il metodo `Add` della classe `DbSet<T>`, mentre per impostare un oggetto su `Deleted`, dobbiamo invocare `Remove`. Analizziamo ora queste casistiche in dettaglio.

Persistere un nuovo oggetto

In ogni applicazione che gestisce il ciclo completo di un'entità, il primo passo è l'inserimento. Come detto poco sopra, per persistere un nuovo oggetto basta utilizzare il metodo `Add` della classe `DbSet<T>`, passando in input l'oggetto da persistere. L'effetto è quello di aggiungere l'oggetto al contesto nello stato di `Added`.

Esempio 11.21

```
using (var ctx = new NorthwindEntities())
{
    var c = new Customer { /*Imposta proprietà customer*/ };
    ctx.Customers.Add(c);
    ctx.SaveChanges();
}
```

Come possiamo notare, inserire un nuovo cliente è estremamente semplice. Quando si deve inserire un ordine, invece, la situazione cambia leggermente, poiché un ordine contiene riferimenti ai dettagli e al cliente che l'ha emesso.

Infatti, il metodo `Add` aggiunge l'oggetto passato in input al contesto ma, oltre a fare questo, scorre tutte le navigation property dell'oggetto aggiunto e ne aggiunge gli oggetti al change tracker in stato di `Added`. In questo modo, possiamo richiamare una volta soltanto il metodo `Add`, passando l'ordine, con il vantaggio che tanto i dettagli quanto il cliente verranno aggiunti al contesto in fase di inserimento.

Tuttavia, se per i dettagli questo rappresenta l'approccio corretto, per il cliente non possiamo dire altrettanto, visto che, in realtà, non deve essere inserito nuovamente ma solo referenziato, in quanto già esistente nel database. L'approccio più semplice consiste nel non valorizzare la proprietà `Customer` dell'ordine, bensì `CustomerId`. In Entity Framework 6, l'utilizzo della foreign key è praticamente obbligatorio per mantenere le relazioni ma con Entity Framework Core non è così, e potremmo non avere la proprietà `CustomerId` nell'ordine; in questo caso possiamo ricorrere ai metodi `Attach` e `Update`. Entrambi i metodi, come `Add`, prendono un oggetto in input e ne scorrono tutte le navigation property per attaccarne gli oggetti al contesto, ma cambiano il modo di impostare lo stato. Per ogni oggetto, se il valore della chiave primaria è uguale al valore di default del suo tipo (0 per interi, `null` per stringhe e così via) allora il relativo stato viene impostato su `Added`, altrimenti viene

impostato su Unchanged o Modified a seconda del metodo usato. Entrambe le tecniche sono mostrate nel prossimo esempio.

Esempio 11.22

```
//Inserimento con foreign key
using (var ctx = new NorthwindEntities())
{
    var o = new Order {
        CustomerId = "ALFKI",
        //Imposta altre proprietà ordine ma non la chiave perché ordine nuovo
    };
    o.OrderDetails.Add(new OrderDetail {
        //Imposta proprietà dettaglio ma non la chiave perché ordine nuovo
    });
    o.OrderDetails.Add(new OrderDetail {
        //Imposta proprietà dettaglio ma non la chiave perché ordine nuovo
    });
    ctx.Orders.Add(o);
    ctx.SaveChanges();
}

//Inserimento con navigation property (Solo Entity Framework Core)
using (var ctx = new NorthwindEntities())
{
    var o = new Order {
        Customer = new Customer { CustomerId = "ALFKI" },
        //Imposta altre proprietà ordine ma non la chiave perché ordine nuovo
    };
    o.OrderDetails.Add(new OrderDetail {
        //Imposta proprietà dettaglio ma non la chiave perché ordine nuovo
    });
    o.OrderDetails.Add(new OrderDetail {
        //Imposta proprietà dettaglio ma non la chiave perché ordine nuovo
    });
    ctx.Orders.Attach(o);
    ctx.SaveChanges();
}
```

Ora che abbiamo imparato a inserire nuovi oggetti, passiamo ad analizzarne la rispettiva modifica.

Persistere le modifiche a un oggetto

Per modificare un cliente, basta leggerlo dal database, modificarne le proprietà e invocare il metodo SaveChanges/SaveChangesAsync come mostrato nel prossimo esempio.

Esempio 11.23

```
using (var ctx = new NorthwindEntities())
{
    var cust = ctx.Customers.Find("ALFKI");
    cust.Address.Address = "Piazza del popolo 1";
    ctx.SaveChanges();
}
```

Questa modalità di aggiornamento viene definita come *connessa*, poiché l'oggetto è modificato mentre il contesto che lo ha istanziato è ancora in vita.

Tuttavia, non è sempre così. Prendiamo, per esempio, un Web Service che metta a disposizione un metodo che torna i dati di un cliente e un altro che accetti un cliente e lo salvi sul database. La cosa corretta, in questi casi, è creare un contesto diverso in ogni metodo. Questo significa che non c'è nessun tracciamento delle modifiche fatte dal client e quindi il secondo contesto non può sapere cosa è cambiato. Quando incontriamo questo tipo di problema, si parla di modalità *disconnessa*, in quanto le modifiche all'oggetto sono fatte fuori dal contesto che lo ha creato.

In questi casi abbiamo a disposizione due modalità per risolvere il problema. Con la prima effettuiamo nuovamente la query per recuperare il cliente e impostiamo le proprietà con i dati che vengono passati in input al metodo. In questo caso il contesto riesce a tracciare le modifiche quindi il codice è identico a quello visto nell'[esempio 11.23](#).

La seconda consiste nell'aggiungere l'oggetto modificato al contesto, marcandolo come `Modified`. Se usiamo Entity Framework 6, dobbiamo attaccare l'oggetto al contesto con il metodo `Attach`, recuperare la relativa entry dal change tracker con il metodo `Entry` e modificare lo stato in `Modified`. Se usiamo Entity Framework Core, possiamo agire sullo stato come con Entity Framework 6, ma possiamo più semplicemente invocare il metodo `Update` già analizzato in precedenza. L'[esempio 11.24](#) mostra entrambe le possibilità.

Esempio 11.24

```
//Metodo del servizio che torna il cliente
public Customer GetCustomer()
{
    using (var ctx = new NorthwindEntities())
    {
        return ctx.Customers.Find("STEMO");
    }
}
```

```

}

//Il client aggiorna il cliente e chiama il servizio di aggiornamento
Cust.Address.Address = "Piazza Venezia 10";
UpdateCustomer(Cust);

//Il servizio aggiorna il cliente modificando lo stato dell'entity
public void SaveCustomer(Customer modifiedCustomer)
{
    using (var ctx = new NorthwindEntities())
    {
        ctx.Customers.Attach(modifiedCustomer);
        ctx.Entry(modifiedCustomer).State = EntityState.Modified;
        ctx.SaveChanges();
    }
}

//Il servizio aggiorna il cliente tramite il metodo Update
public void SaveCustomer(Customer modifiedCustomer)
{
    using (var ctx = new NorthwindEntities())
    {
        ctx.Customers.Update(modifiedCustomer);
        ctx.SaveChanges();
    }
}

```

Quando si parla di ordini e dettagli, o più in generale di classi che hanno navigation property, è importante sottolineare una cosa: se aggiungiamo, modifichiamo o cancelliamo un dettaglio in modalità disconnessa, anche impostando l'ordine come modificato non otteniamo l'aggiornamento dei dettagli. Per eseguire un corretto aggiornamento, dobbiamo fare una comparazione manuale tra gli oggetti presenti sul database e quelli presenti nell'oggetto modificato e cambiare manualmente lo stato di ogni dettaglio. Dopo la modifica, è arrivato il momento di passare alla cancellazione dei dati.

Cancellare un oggetto dal database

L'ultima operazione di aggiornamento sul database è la cancellazione. Cancellare un oggetto è estremamente semplice, in quanto basta invocare il metodo `Remove` della classe `DbSet<T>`, passando in input l'oggetto. C'è comunque una particolarità molto importante da tenere presente. L'oggetto da cancellare **deve** essere attaccato al contesto. Questo significa che, anche in caso di cancellazione, abbiamo una modalità connessa e una disconnessa.

Nella modalità connessa possiamo semplicemente eseguire una query per recuperare l'oggetto e invocare poi `Remove` come nel prossimo esempio.

Esempio 11.25

```
using (var ctx = new NorthwindEntities())
{
    var cust = ctx.Customers.Find("STEMO");
    ctx.Customers.Remove(cust);
    ctx.SaveChanges();
}
```

Anche nella modalità disconnessa abbiamo due tipi di scelta: recuperare l'oggetto dal database e invocare `Remove` (stesso codice dell'[esempio 11.25](#)) oppure attaccare l'oggetto al contesto e poi chiamare il metodo `Remove` come nel prossimo esempio.

Esempio 11.26

```
using (var ctx = new NorthwindEntities())
{
    ctx.Customers.Attach(custToDelete);
    ctx.Customers.Remove(custToDelete);
    ctx.SaveChanges();
}
```

Come detto, gli oggetti attaccati al contesto vengono salvati nel change tracker. Questo componente espone alcuni metodi che possono essere utili per manipolare gli oggetti al suo interno.

[Manipolare gli oggetti nel change tracker](#)

Il contesto espone il change tracker tramite la proprietà `ChangeTracker` di tipo `DbChangeTracker/ChangeTracker`. Il metodo più importante di questa classe è `Entries` che ritorna una lista di oggetti di tipo `DbEntityEntry/EntityEntry`. Ogni istanza (detta entry) contiene i dati di una entity attaccata al contesto, li espone attraverso le sue proprietà e permette di manipolarli attraverso i suoi metodi. Quelli principali sono esposti nella [Tabella 11.4](#).

Tabella 11.4 – Membri di `DbEntityEntry/EntityEntry`

Membro	Scopo
<code>Entity</code>	Proprietà che restituisce l'entity associata all'entry

State	Proprietà che restituisce e imposta lo stato dell'entity
IsKeySet	Specifica se la chiave dell'entity è impostata (solo Entity Framework Core)
Metadata	Restituisce i dati di mapping dell'entity (Solo Entity Framework Core)
OriginalValues	Restituisce i valori dell'entity com'era quando è stata attaccata al contesto (recuperandola dal database o usando uno dei metodi per attaccarla al contesto)
CurrentValues	Restituisce i valori attuali dell'entity
Reload/ReloadAsync	Ricarica l'entity prendendo i valori dal database

Come detto, `Entries` restituisce una lista di entry. Questo significa che possiamo filtrare le entry per recuperarne una specifica e modificarne lo stato, oppure tutte quelle in uno stato di `Added` per loggare le operazioni di inserimento o altro ancora.

Quando abbiamo un riferimento a una entity e vogliamo recuperare l'entry associata, possiamo usare il metodo `Entry` del contesto che accetta in input la entity e restituisce l'entry.

Aggiornare i dati con Entity Framework è tutt'altro che complesso, quindi non necessita di ulteriori approfondimenti. Nella prossima sezione invece vedremo brevemente altre caratteristiche che possono tornare utili durante lo sviluppo.

Funzionalità aggiuntive di Entity Framework

Ovviamente, in questo capitolo abbiamo trattato solo gli argomenti principali che ci permettono di sviluppare con Entity Framework. Infatti, esistono molte altre funzionalità che entrambi i framework ci mettono a disposizione. In questa sezione elenchiamo quelle più comuni:

- ❑ **Concorrenza:** Entity Framework gestisce la concorrenza ottimistica sull'aggiornamento dei dati, per evitare che un utente possa aggiornare dati obsoleti. L'unica cosa che dobbiamo fare per abilitare la concorrenza è indicare nel mapping quali campi devono essere controllati.

- ❑ Code-First migration: permette di creare il database a design-time, partendo dalle classi del modello a oggetti. Inoltre, permette di modificare il database al cambio del codice delle classi mappate. Questi comandi non sono disponibili per Entity Framework 6 su .NET Core.
- ❑ Validazione: poiché nel mapping specifichiamo molte informazioni, come per esempio la lunghezza di una proprietà di tipo `string`, il contesto prima di persistere le notifiche verifica che le proprietà siano conformi al loro mapping, evitando così di arrivare al database con dati non validi.
- ❑ Logging: Entity Framework ha un motore di logging integrato, dove possiamo intercettare l'esecuzione dei comandi e generare un log. Va detto che il codice di logging è profondamente differente tra i due framework.
- ❑ Mapping avanzato: Entity Framework Core permette di mappare i campi di una tabella su più classi, di mappare l'ereditarietà secondo il modello TPH (Entity Framework 6 supporta anche il modello TPT), di mappare campi della tabella verso membri privati di una classe (Solo Entity Framework Core) e di specificare chiavi univoche alternative.

Limitazioni di Entity Framework 6.3 su .NET Core

Come abbiamo detto in questo capitolo, Entity Framework 6.3 soffre di alcune limitazioni quando viene eseguito su .NET Core. Oltre al dover registrare le provider factory a mano e il dover passare obbligatoriamente la stringa di connessione nel costruttore, ce ne sono altre. Per esempio, non è possibile compilare applicazioni che usano Entity Framework 6 su .NET Core e che usano il file EDMX. Questo perché il task di compilazione che scompatta il file nei tre che vengono letti a runtime dal `DbContext` non è disponibile su .NET Core. Per chiudere, è necessaria una modifica ai provider di Entity Framework 6 per fare in modo che questi funzionino sia su .NET Core sia su .NET Framework. Attualmente, solo il provider per `SqlServer` è stato modificato per girare su entrambi i framework. Il team ha già fatto sapere che queste limitazioni sono temporanee e saranno superate nel momento in cui

Entity Framework 6.3 sarà definitivamente rilasciato, ma ce ne sono altre che invece sono permanenti e non verranno rimosse.

La prima limitazione consiste nell'assenza del provider per Sql Server Compat, dato che non esiste un provider per questo database su .NET Core. La seconda limitazione consiste nel fatto che i tipi spaziali e il tipo HierarchyId non saranno supportati. Applicazioni esistenti che si basano su questi tipi non possono essere migrate senza una rivisitazione del codice che faccia a meno di queste funzionalità.

Conclusioni

In questo capitolo abbiamo parlato delle caratteristiche principali di Entity Framework, così da poter cominciare a utilizzare questo O/RM. Ora siamo in grado di costruire e modificare un modello sia scrivendo a mano l'object model, il contesto e il mapping, sia sfruttando i tool che Entity Framework ci mette a disposizione per generare le stesse cose partendo dal database. Abbiamo visto come recuperare i dati dal database sfruttando il provider LINQ e come persistere sia oggetti semplici, sia grafi complessi con poche righe di codice.

Tuttavia, c'è molto di più: come abbiamo visto nella penultima sezione del capitolo, Entity Framework offre una serie di funzionalità che lo rendono uno strumento valido per l'accesso ai dati. Questo, unito alla capacità di Entity Framework Core di girare anche su piattaforme Linux e MacOS, fa capire come Microsoft stia investendo molto su questa tecnologia, che raffigura il presente e il futuro dell'accesso ai dati.

Ora che abbiamo visto come accedere ai dati residenti su un database, nel prossimo capitolo esamineremo un altro modo di persistere i dati: XML.

XML e LINQ to XML

L'**eXtensible Markup Language** (XML) è un metalinguaggio standard creato dal World Wide Web Consortium (W3C) presente sul mercato ormai da parecchi anni ed è diventato uno strumento d'uso comune nelle applicazioni, insieme a JSON. Lo utilizziamo quotidianamente per le configurazioni, per lo scambio di dati tra diverse piattaforme oppure, grazie alla sua semplicità di utilizzo con strumenti di disegno e alla sua immediata leggibilità come linguaggio di markup e di layout, tanto nell'ambito web quanto in quello desktop: ne sono un esempio SOAP (per la creazione di servizi) o XAML (linguaggio per Windows Presentation Foundation e Universal Windows Platform app).

Il supporto a XML con .NET

Fin dalla prima versione di .NET sono presenti classi per la lettura e la manipolazione dell'XML di invenzione da parte di Microsoft, ognuna delle quali aderisce a specifici standard ed è più o meno immediata e comoda da utilizzare a seconda della situazione che dobbiamo fronteggiare. Sono contenute inoltre classi, racchiuse sotto il nome di **LINQ to XML**, che consentono di sfruttare LINQ e la query syntax come metodologia alternativa di gestione dell'XML. Questi strumenti sono ormai consolidati e considerati primari tanto da essere disponibili all'interno di .NET Standard, in modo completo nella versione 2.0, e quindi utilizzabili su tutti i runtime .NET.

L'obiettivo di questo capitolo è illustrare il migliore utilizzo di tutte queste classi, al fine di capire qual è lo strumento più adatto per la soluzione di un determinato problema, calcolandone oneri e benefici. Il documento mostrato nell'[esempio 12.1](#) è utilizzato all'interno dell'intero capitolo come sorgente degli esempi d'applicazione delle varie tecnologie di manipolazione dei documenti XML.

Esempio 12.1

```
<?xml version="1.0" encoding="utf-8" ?>
<products
  xmlns="http://schemas.aspitalia.com/book40/products">
  <!-- Lista prodotti -->
  <product idProduct="1"
    idCategory="1">
    <description>Prodotto 1</description>
    <details
      xmlns="http://schemas.aspitalia.com/book40/details">
      <detail name="Size"
        value="10x20" />
      <detail name="Weight"
        value="2" />
    </details>
  </product>
  <product idProduct="2"
    idCategory="3">
    <description>Prodotto 2</description>
    <details
      xmlns="http://schemas.aspitalia.com/book40/details">
      <detail name="Size"
        value="40x35" />
      <detail name="Weight"
        value="8" />
    </details>
  </product>
  <product idProduct="3"
    idCategory="1">
    <description>Prodotto 31</description>
    <details
      xmlns="http://schemas.aspitalia.com/book40/details">
      <detail name="Size"
        value="10x25" />
      <detail name="Weight"
        value="2" />
    </details>
  </product>
</products>
```

L'infoset d'esempio racchiude, sotto il nodo principale `products`, una serie di prodotti con relative caratteristiche e dettagli, quali la dimensione e il

peso.

Gestire l'XML con la classe XmlDocument

Lo strumento più semplice presente in .NET per la gestione di documenti XML è rappresentato dalla classe `XmlDocument` che descrive, secondo le specifiche DOM livello 2 (uno standard W3C), un documento XML in base a una struttura ad albero, quindi con padri e figli, e ne permette l'interrogazione e la modifica.

Il fatto che questa classe aderisca alle specifiche DOM, significa che se proveniamo da altri linguaggi o tecnologie, come JavaScript, PHP, Python o Java, avremo già una certa familiarità con i metodi e le proprietà messe a disposizione. La classe `XmlDocument` rappresenta dunque il documento e ne permette il caricamento mediante i metodi `Load` o `LoadXml`: il primo accetta, con vari overload, un URI, uno `Stream`, un `TextReader` o un `XmlReader` (affrontato in seguito), mentre il secondo accetta direttamente l'XML come stringa nella sua forma testuale.

Esempio 12.2

```
var doc = new XmlDocument();

// Uri: percorso relativo al file
doc.Load("test.xml");

// Apertura di uno stream su file
using (Stream stream = File.OpenRead("test.xml"))
{
    // Oppure caricamento da uno Stream
    doc.Load(stream);
}

// Oppure caricamento XML diretto
doc.LoadXml("<test />");
```

Quando carichiamo un documento XML attraverso questa classe, questo viene rappresentato in memoria con istanze di oggetti specifici per ogni elemento, attributo o, in generale, per qualsiasi tipologia di nodo dal quale è

costituito. Questa distinzione è implementata nelle classi offerte all'interno di .NET e possiamo capirne la logica analizzando la [Figura 12.1](#).

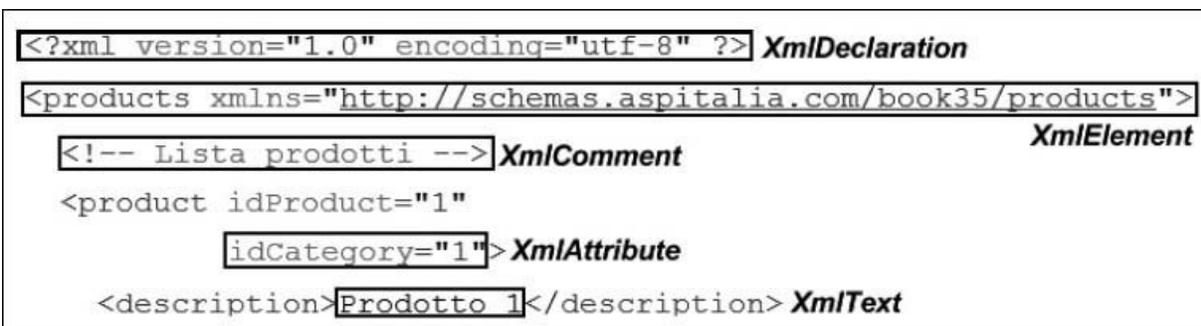


Figura 12.1 – Frazionamento di un documento XML secondo DOM.

Nella [Figura 12.2](#) è invece visibile l'intero modello a oggetti per la rappresentazione di un documento XML.

La struttura dei nodi, che deriva dal caricamento del documento, prende origine dalla proprietà `DocumentElement` della classe `XmlDocument`, di tipo `XmlElement`, e rappresenta il nodo radice obbligatorio. Da esso possiamo accedere agli altri elementi attraverso una serie di proprietà base, definite a livello della classe `XmlNode`, come `ChildNodes`, per ottenere una collezione di nodi figli, `FirstChild` e `LastChild`, per ottenere il primo e l'ultimo figlio, `PreviousSibling` e `NextSibling`, e i nodi fratelli precedenti o successivi, presenti sullo stesso livello.

Possiamo poi ricavare informazioni generiche sul nodo, sfruttando la proprietà `LocalName`, per conoscere il nome del tag o dell'attributo, la proprietà `NamespaceURI`, per conoscere il namespace del nodo, oppure `OuterXml` e `InnerXml`, per ottenere il nodo corrente come stringa XML, comprensivo dei figli o dei soli nodi che contiene.

Sempre sulla classe `XmlNode` è disponibile la proprietà `Attributes` che restituisce una collezione di attributi (solo se si tratta di un elemento) mentre, con la proprietà predefinita, possiamo ottenere gli elementi figli con un certo nome. Inoltre, a seconda della tipologia di nodi, abbiamo a disposizione proprietà e metodi specifici. Per esempio, la classe `XmlElement` ha i metodi `HasAttribute`, `GetAttribute` e `GetElementsByTagName` per capire, rispettivamente, se un elemento ha un attributo, per ottenerne il valore, oppure per recuperare la lista degli elementi figli in base al nome. Certamente questo comporta l'introduzione di approcci diversi per ottenere

informazioni di tipo simile, tuttavia questi metodi risultano più comodi da utilizzare, essendo tipizzati per gli elementi specifici.

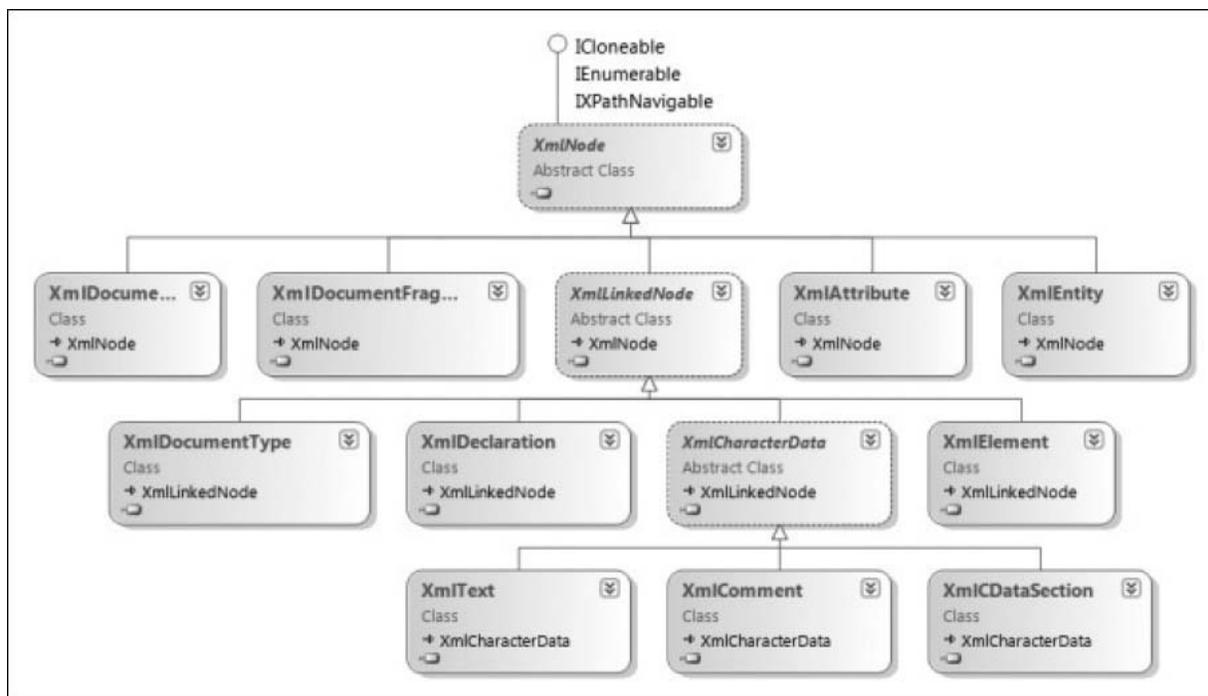


Figura 12.2 – Struttura gerarchica delle classi System.Xml.

Esempio 12.3

```

// Nome del tag e namespace dell'ultimo figlio
Console.WriteLine($"LocalName: {doc.DocumentElement.LastChild.LocalName} -
Namespace: {doc.DocumentElement.LastChild.NamespaceURI}");

// Attributo idProduct sull'ultimo figlio (tag product)
var product = (XmlElement)doc.DocumentElement.LastChild;
Console.WriteLine($"idProduct: {product.Attributes("idProduct").Value}");
Console.WriteLine($"idProduct: {product.GetAttribute("idProduct")}");

// XML del primo figlio di products (il commento)
Console.WriteLine($"Comment: {doc.DocumentElement.FirstChild.OuterXml}");

// Tag description del secondo figlio (tag product)
XmlElement description = doc.DocumentElement.ChildNodes[1]
    ["description", "http://schemas.aspitalia.com/book40/products"];

// InnerText interroga direttamente il value dei figli
Console.WriteLine($"Description: {description.InnerText}");
// Entro direttamente nel figlio che è un XmlText
Console.WriteLine($"Description: {description.FirstChild.Value}");
    
```

Nel codice rappresentato nell'[esempio 12.3](#), possiamo vedere all'opera le varie proprietà e quali risultati si ottengono con l'XML di esempio illustrato a inizio capitolo. Notiamo inoltre l'indicazione del namespace del tag `description`, che eredita quello del tag contenitore `products`. Il risultato dell'output della Console è visibile nella [Figura 12.3](#).

A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window displays the following text:

```
LocalName: product - Namespace: http://schemas.aspitalia.com/book40/products
idProduct: 3
idProduct: 3
Comment: <!-- Lista prodotti -->
Description: Prodotto 1
Description: Prodotto 1
Press any key to continue . . .
```

Figura 12.3 – Output ottenuto mediante interrogazione con XmlDocument.

Otteniamo così il nome e il namespace del tag `product`, il valore dell'attributo `idProduct`, il valore di `OuterXml` per un commento e la lettura del tag `description`.

Qualificare i nodi con un namespace è importante per identificare in modo univoco una struttura XML in base a uno schema XSD. Di conseguenza, anche nella ricerca di un elemento, non dobbiamo limitarci al `LocalName`, ma diventa obbligatorio specificare anche il namespace.

Oltre a leggere un documento XML, possiamo modificarlo sempre sfruttando le stesse classi. Le proprietà `value`, `InnerText` e `InnerXml` non sono in sola lettura perciò, una volta che abbiamo ottenuto un nodo, lo possiamo modificare.

A questi si affiancano i metodi `AppendChild`, `InsertAfter`, `InsertBefore`, `RemoveAll`, `RemoveChild` e `ReplaceChild` per aggiungere un nodo in coda – prima o dopo un altro nodo di riferimento – rimuovere tutti i nodi o uno specifico oppure sostituirne uno con un altro.

Per nodo s'intende un oggetto che fa parte di un documento XML. Qualsiasi commento, attributo, elemento, testo o dichiarazione XML rientra in questa categoria, come viene dimostrato nella [Figura 12.2](#).

Dobbiamo rilevare che nuovi nodi non si possono creare direttamente tramite il costruttore delle classi, ma occorre invocare una serie di metodi `Create[tipo Nodo]` per creare una nuova istanza e poterla aggiungere a un nodo, sfruttando i metodi illustrati in precedenza.

Infine, una volta terminata l'elaborazione del documento, possiamo salvarlo con il metodo `save` della classe `XmlDocument`, come viene mostrato nell'[esempio 12.4](#).

Esempio 12.4

```
// Rimuovo il terzo prodotto
doc.DocumentElement.RemoveChild(doc.DocumentElement.LastChild);
// Tolgo tutti i figli del secondo prodotto
doc.DocumentElement.LastChild.RemoveAll();

// Creo l'elemento product
XmlElement product = doc.CreateElement("", "product",
    "http://schemas.aspitalia.com/book40/products");
// Imposto l'attributo idProduct
product.SetAttribute("idProduct", "4");
// Aggiungo l'elemento prima degli altri prodotti
doc.DocumentElement.InsertBefore(product,
    doc.DocumentElement.ChildNodes[1]);

// Commento all'interno del tag product
XmlComment comment = doc.CreateComment("Nuovo prodotto");
product.AppendChild(comment);

// Sezione CDATA
XmlCDataSection cdata = doc.CreateCDataSection("Testo libero con <complesso>");
product.AppendChild(cdata);

// Creo elemento description
XmlElement description = doc.CreateElement("", "description",
    "http://schemas.aspitalia.com/book40/products");
// Creo un nodo XmlText con la descrizione
description.InnerText = "Prodotto 4";
// Aggiungo l'elemento al tag product
product.AppendChild(description);

// Salvo il documento
var sw = new StringWriter();
doc.Save(sw);
```

L'XML che otteniamo in output è mostrato nell'[esempio 12.5](#): ha perso un prodotto, ne contiene un altro senza nodi figli, mentre ne è presente uno nuovo con commento e sezione CDATA.

Esempio 12.5

```
<?xml version="1.0" encoding="utf-16"?>
<products xmlns="http://schemas.aspitalia.com/book40/products">
  <!-- Lista prodotti -->
  <product idProduct="4">
    <!-- Nuovo prodotto -->
    <![CDATA[Testo libero con <complesso>]]>
    <description>Prodotto 4</description>
  </product>
  <product idProduct="1" idCategory="1">
    <description>Prodotto 1</description>
    <details xmlns="http://schemas.aspitalia.com/book40/details">
      <detail name="Size" value="10x20" />
      <detail name="Weight" value="2" />
    </details>
  </product>
  <product>
  </product>
</products>
```

La classe `XmlDocument`, quindi, è adatta per una manipolazione facile e molto intuitiva del documento XML, soprattutto se conosciamo già le specifiche DOM. Con questa tecnica l'intero documento viene analizzato e caricato in memoria per poter essere interrogato o manipolato fino a quando non lo salviamo con il metodo `save` verso uno `stream`, un file fisico o un `TextWriter`. Di fatto, è quindi adatto per documenti di piccola dimensione, nei quali vogliamo prediligere la facilità di gestione.

Letture e scrittura rapida e leggera

La classe `XmlDocument` appena analizzata è molto comoda e intuitiva da utilizzare, ma presenta il difetto di caricare l'intero documento in memoria, inclusi i metadati aggiuntivi, e quindi potrebbe non essere la soluzione migliore se intendiamo elaborare documenti XML di dimensioni considerevoli. Per ovviare a questo limite, in .NET sono presenti le classi astratte `XmlReader` e `XmlWriter`, che ci consentono di leggere e scrivere documenti XML sfruttando lo stesso tipo di accesso ai nodi, ma con un

approccio forward-only (senza possibilità di andare indietro), in sola lettura per `XmlReader` e in sola scrittura per `XmlWriter`. Le principali implementazioni sono `XmlTextReader` e `XmlTextWriter`, che sono in grado di leggere e scrivere documenti testuali.

È bene ricordare che le specifiche XML InfoSet non vincolano al solo formato testuale dato che quello che conta è la rappresentazione logica e dunque l'XML potrebbe benissimo basarsi su una codifica binaria.

[Leggere con XmlReader](#)

Per la lettura, il metodo statico `XmlReader.Create` dispone di molti overload per aprire `Stream`, `TextReader` o un URI e accetta eventualmente un parametro di tipo `XmlReaderSettings`, per impostare alcune opzioni nella lettura, come ignorare commenti o spazi bianchi oppure scegliere di effettuare validazioni mediante schema, piuttosto che chiudere lo `Stream` alla chiusura dell'`XmlReader`.

Esempio 12.6

```
// Impostazioni di caricamento del file
var settings = new XmlReaderSettings();
settings.IgnoreComments = true;
settings.IgnoreProcessingInstructions = true;
settings.IgnoreWhitespace = true;

// Using per chiudere il file e liberarlo
using (var reader = XmlReader.Create("Test.xml", settings))
{
    // Lettura reader
}
```

Come abbiamo già accennato in precedenza, la lettura avviene con un approccio forward-only, mediante un cursore che decodifica l'XML progressivamente alla lettura, passando da un nodo all'altro, considerando come tali anche gli spazi bianchi tra i tag, le istruzioni di processamento, i commenti, le sezioni CDATA e, più in generale, qualsiasi contenuto del documento. È per questo che, nell'[esempio 12.6](#), richiediamo esplicitamente di saltare alcuni nodi che non riteniamo necessari.

Per scorrere un documento possiamo utilizzare il metodo principale `Read`, che restituisce un boolean per indicare se il documento è giunto alla fine. Ogni sua invocazione determina il passaggio al nodo successivo e, tramite la classe `XmlReader`, possiamo ottenerne le relative informazioni. Le principali proprietà sono `LocalName`, `NamespaceURI`, `NodeType`, `Value`, `HasAttributes` e `HasValue`: non tutte sono però sempre disponibili, poiché variano nel significato a seconda del nodo in cui l'`XmlReader` è posizionato.

Possiamo sfruttare, in alternativa o come complemento al metodo `Read`, una serie di metodi `MoveTo[TipoElemento]` che indicano di far posizionare il cursore su un determinato tipo di nodo, restituendo `true` o `false` in base all'avvenuto successo dell'operazione. Per facilitare ancor di più lettura dei nodi, i metodi `ReadContentAs[tipo CLR]` e `ReadElementContentAs[tipo CLR]` permettono la lettura tipizzata, con conversione automatica del contenuto (a seconda del nodo) o del contenuto dell'elemento in cui si trova il cursore. L'[esempio 12.7](#) mostra come usare questi metodi, sfruttando sempre lo stesso documento XML presentato all'inizio del capitolo.

Esempio 12.7

```
// Leggo il prossimo nodo dallo stream
while (reader.Read())
{
    // Stampo nome del nodo (attributo o elemento) e tipo
    Console.WriteLine($"LocalName: {reader.LocalName} - NodeType:
        {reader.NodeType}");

    // Intercetto l'inizio di un nuovo elemento
    if (reader.NodeType == XmlNodeType.Element)
    {
        // Identifico in che elemento mi trovo
        switch (reader.LocalName)
        {
            case "description":
                // Controllo che l'elemento non sia vuoto
                if (!reader.IsEmptyElement)
                {
                    Console.WriteLine(
                        $"Description: {reader.ReadElementContentAsString()}");
                }
                break;

            case "product":
                // Mi sposto sull'attributo idProduct
                if (reader.MoveToAttribute("idProduct"))
                {
                    // Se è andato a buon fine, leggo il valore
                    Console.WriteLine($"idProduct: {reader.Value}");
                }
        }
    }
}
```

```
        break;
    }
}
```

Data la natura di `xmlReader`, non abbiamo la sicurezza della validità del documento quando lo si apre con `create`, cosa che, invece, può assicurare la classe `xmlDocument`. In questo caso, infatti, dobbiamo sfogliare l'intero documento con il metodo `read` così da procedere passo per passo nell'analisi dello stesso, fino ad arrivare alla fine.

Scrivere con XmlWriter

Per la scrittura di documenti XML, la classe `xmlWriter` si comporta come `xmlReader` e dispone anch'essa di un metodo statico `create` per scrivere XML con un approccio forward-only.

Il suo utilizzo è abbastanza semplice, poiché è sufficiente invocare i metodi `write[tipo nodo]` o `writeStart[tipo nodo]` e `writeEnd[tipo nodo]` per i nodi complessi, dove "tipo nodo" dipende dalla posizione del cursore. Infatti è a nostro carico conoscerla e invocare i metodi più appropriati, pena un'eccezione a runtime. Nel codice 12.8 è mostrato come possiamo creare un documento XML e come sia necessario prestare attenzione alla corretta apertura e chiusura dei nodi. In realtà, in certe circostanze, il motore è in grado di chiuderli in modo automatico ma è buona norma farlo in modo esplicito, così da ottenere una migliore visione di ciò che stiamo creando.

Esempio 12.8

```
// Impostazioni di scrittura
var settings = new XmlWriterSettings();
settings.Indent = true;
settings.NewLineOnAttributes = true;

string ns = "http://schemas.aspitalia.com/book40/products";

var stringWriter = new StringWriter();

using (XmlWriter writer = XmlWriter.Create(stringWriter, settings))
{
    writer.WriteStartDocument();

    // Nuovo tag products
```

```

writer.WriteStartElement("products", ns);
writer.WriteComment("Nuovo prodotto");

// Nuovo tag product con attributo
writer.WriteStartElement("product", ns);
writer.WriteAttributeString("idProduct", "4");

// Noto testuale
writer.WriteElementString("description", ns, "Prodotto 4");

// Chiudo i tag
writer.WriteEndElement();
writer.WriteEndElement();

writer.WriteEndDocument();
}

```

La classe `xmlWriter` consente di specificare alcune opzioni per la formattazione del documento che viene generato, tra cui l'indentazione (`Indent` e `NewLineOnAttributes`), le tipologie dei caratteri e l'encoding da utilizzare.

Nella fase di scrittura abbiamo una certa tranquillità nel produrre un documento ben formattato, dato che abbiamo un aiuto nella chiusura degli elementi, ma non abbiamo alcuna certezza che l'XML sia valido secondo uno schema prefissato.

Le classi `xmlReader` e `xmlWriter` possono quindi sembrare scomode perché sono verbose e richiedono di non sbagliare nella navigazione tra gli elementi ma sono, di fatto, il miglior modo per leggere e scrivere un documento se vogliamo impiegare il numero minore possibile di risorse e raggiungere il massimo delle prestazioni.

[LINQ to XML](#)

Gli strumenti visti finora sono i principali tra quelli disponibili con .NET; ve ne sono anche altri – come **XPath** affrontato successivamente e il cui scopo è più orientato a supportare le trasformazioni XSLT – i quali, purtroppo, richiedono la conoscenza di ulteriori linguaggi, che sono standard e utilizzabili anche con altri ambienti e linguaggi di programmazione, ma che richiedono tempo, ulteriore esperienza e, spesso, spingono il programmatore a limitarsi e a ripiegare sul semplice DOM.

L'intenzione di **LINQ** è quella di fornire un unico modo di interrogare i dati, in modo da permettere di muoversi agilmente con diverse sorgenti tra

cui anche documenti XML. A questo scopo disponiamo di classi ([Figura 12.4](#)), racchiuse sotto il nome di LINQ to XML, in grado di decodificare l'XML, permettendoci di interrogare con facilità un documento mediante la query syntax e le estensioni di LINQ to Object.

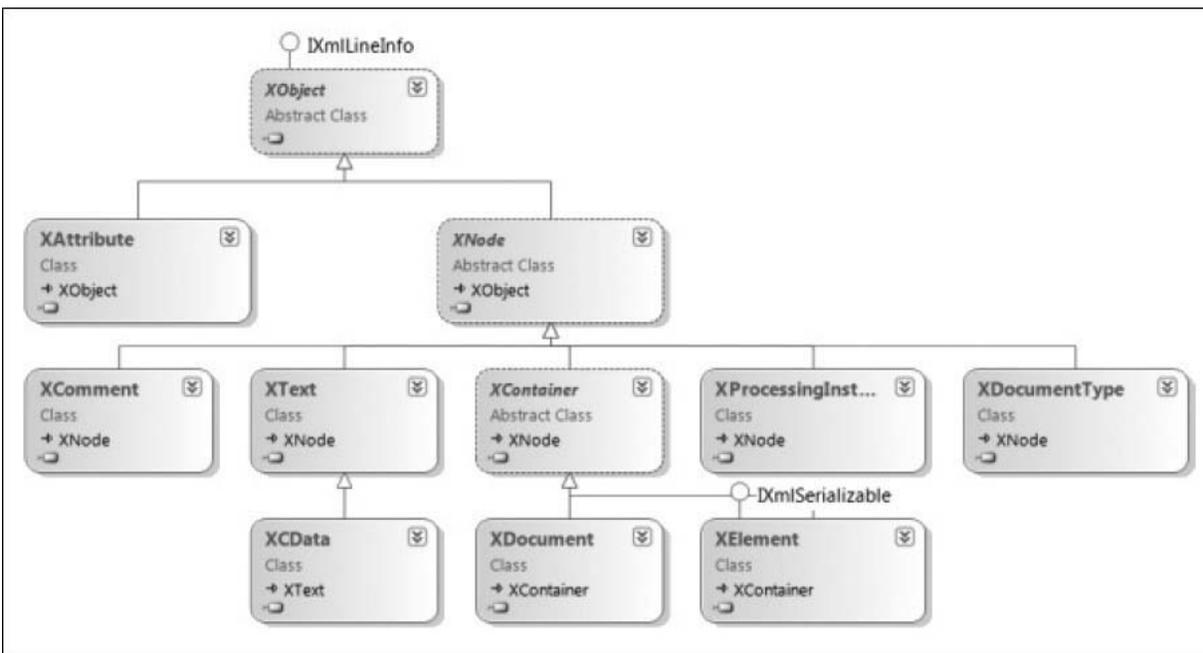


Figura 12.4 – Struttura gerarchica delle classi del namespace System.Xml.Linq.

La loro struttura è peraltro molto simile a quella vista nel caso di `XmlNode`, ma è maggiormente ottimizzata, poiché non sono presenti membri inutili per certe tipologie di nodi.

Interrogare i nodi con LINQ

La classe principale è rappresentata da `XDocument` che con i metodi `Load`, `Parse` e `Save`, permette di caricare, decodificare una stringa o salvare un documento XML. Da questa, attraverso la proprietà `Root`, abbiamo accesso all'`XElement` radice, sul quale possiamo invocare una serie di metodi per ottenere uno specifico elemento figlio o un attributo (classe `XAttribute`). L'[esempio 12.9](#) mostra come leggere l'attributo `idProduct` sul primo tag `product`. È da notare come gli oggetti `XElement` e `XAttribute` dispongano

di operatori di conversione espliciti, con i quali possiamo quindi effettuare il cast sul tipo desiderato; nel caso non sia possibile, otterremo un'eccezione.

Esempio 12.9

```
// Caricamento del documento
XDocument doc = XDocument.Load("test.xml");
// Recupera l'elemento product
XElement product = doc.Root.Element(
    "{http://schemas.aspitalia.com/book40/products}product");
// Lettura attributo idProduct
var idProduct = (int)product.Attribute("idProduct");
```

Nell'[esempio 12.9](#) sicuramente ci colpisce la particolare sintassi “{namespace}localName” utilizzata per qualificare correttamente l'elemento. In realtà, il metodo `Element`, che restituisce la lista degli elementi, accetta un tipo `XName` che unisce il namespace al localName.

Grazie a un operatore di conversione implicito, possiamo utilizzare direttamente una stringa con la sintassi speciale, vista nell'[esempio 12.9](#). In alternativa, possiamo usare il metodo statico `XName.Get` o la classe `XNamespace`, per riutilizzare il namespace concatenandolo con il localName una o più volte, come viene illustrato nell'[esempio 12.10](#).

Esempio 12.10

```
XNamespace productNs =
    "http://schemas.aspitalia.com/book40/products"
XElement product = doc.Root.Element(productNs + "product")
```

Quanto visto finora non basta comunque a giustificare la creazione di nuove API. Infatti, da esse traiamo il massimo con una serie di metodi come `Nodes`, `Elements`, `Attributes` o `Descendants`, i quali restituiscono un `IEnumerable<XNode>` o tipi derivati da quest'ultimo.

Questi enumerabili sono ulteriormente gestibili con LINQ to Object e con alcuni extension method della classe `System.Xml.Linq.Extensions`, che permettono, a loro volta, di ottenere nodi, attributi ed elementi discendenti oppure ascendenti.

L'[esempio 12.11](#) mostra all'opera alcuni di questi metodi combinabili con alcune estensioni della classe `Enumerable`, applicandoli al documento XML d'esempio per questo capitolo, per effettuare la somma oppure ottenere il primo nodo.

Esempio 12.11

```
// Tutti gli attributi idProduct degli elementi product
foreach (XmlAttribute attribute in
    doc.Root.Elements().Attributes("idProduct"))
{
    Console.WriteLine($"idProduct {(string)attribute}");
}

// Il primo nodo (commento)
Console.WriteLine($"Comment: {doc.Root.Nodes().First()}");

// Il primo elemento product
XElement product = doc.Root.Elements(
    productNs + "product").ElementAt(1);
Console.WriteLine("Description: {0}",
    (string)product.Element(productNs + "description"));

// Somma dell'attributo value per tutti
// i detail di tipo Weight
int sum = (from d in doc.Root.Descendants(dns + "detail")
    where d.Attribute("name").Value == "Weight"
    select (int)d.Attribute("value")).Sum();
Console.WriteLine($"Total weight: {sum}");
```

Otteniamo di conseguenza l'output visibile nella [Figura 12.5](#).



```
C:\WINDOWS\system32\cmd.exe
idProduct 1
idProduct 2
idProduct 3
Comment: <!-- Lista prodotti -->
Description: Prodotto 2
Total weight: 12
Press any key to continue . . . _
```

Figura 12.5 – Output ottenuto interrogando con XDocument.

È da notare come il `ToString` di un nodo (il commento, in questo caso) restituisca in forma testuale l'intero nodo XML.

Manipolazione dei nodi

Anche la modifica è resa semplice grazie a metodi ed extension method molto elastici, che consentono di eliminare liste di nodi, di elementi e di attributi oppure di creare facilmente nuove istanze degli stessi, usando l'apposito costruttore. Quest'ultima caratteristica, mostrata nell'[esempio 12.12](#), risulta infatti molto comoda e versatile, poiché durante la creazione di un `XElement` possiamo passare uno o più nodi figli (anche un `IEnumerable` o a sua volta uno o più `XElement`) che vengono gestiti automaticamente e, a seconda del tipo di nodo, emessi come XML nel corretto ordine. Nel caso di tipi primitivi, viene usato come nodo `XText`, da inserire nell'elemento. La compattezza e la comprensione del codice che scriviamo rende questa soluzione adatta anche in scenari misti, nei quali, con **LINQ to Entities**, interroghiamo database e produciamo come risultato un documento XML.

Esempio 12.12

```
// Rimuovo gli attributi del primo elemento
doc.Root.Elements().First().RemoveAttributes();
// Rimuovo il secondo e il terzo elemento product
doc.Root.Elements(productNs + "product").Skip(1).Take(2).Remove();
// Nuovo tag product dentro products
var product =
    new XElement(productNs + "product",
        new XComment("Nuovo prodotto"),
        new XAttribute("idProduct", 4),
        new XElement(productNs + "description", "Prodotto 4"));
doc.Root.Add(product);
```

Nell'[esempio 12.12](#) vengono messi in mostra i principali metodi che abbiamo a disposizione per la manipolazione dei nodi: `RemoveAttributes` rimuove tutti gli attributi appartenenti al nodo, `Remove` rimuove l'intero nodo, mentre il metodo `Add` permette di aggiungere qualsiasi tipologia di nodo. Nell'esempio lo invochiamo passando un `XElement` creato mediante il costruttore, che oltre al nome dell'elemento può ricevere la lista di nodi figli. Il documento XML creato è visibile nell'[esempio 12.13](#).

Esempio 12.13

```
<products
  xmlns="http://schemas.aspitalia.com/book40/products">
  <!-- Lista prodotti -->
  <product>
    <description>Prodotto 1</description>
    <details
      xmlns="http://schemas.aspitalia.com/book40/details">
      <detail name="Size" value="10x20" />
      <detail name="Weight" value="2" />
    </details>
  </product>
  <product idProduct="4">
    <!--Nuovo prodotto-->
    <description>Prodotto 4</description>
  </product>
</products>
```

Tra le varie altre funzionalità che offre la classe base `XObject`, da cui deriva `XNode`, i metodi `AddAnnotation`, `Annotation` e `RemoveAnnotations` possono essere sfruttati per aggiungere informazioni aggiuntive (`Object`) all'oggetto, che non vengono poi emesse nel documento XML ma che possono servirci per l'elaborazione del documento stesso; gli eventi `Changing` e `Changed` permettono invece di intercettare l'aggiunta, la rimozione, la rinomina o la valorizzazione.

Di particolare importanza è la classe `XStreamingElement`, che permette di generare un documento XML, consumandone i contenuti. Infatti, nonostante `XDocument` offra molti vantaggi, ha il difetto di mantenere in memoria l'intera struttura dati, comprese le eventuali liste di tipo `IEnumerable` passate. Queste ultime, in realtà, vengono subito processate, e modificarle in un secondo momento non produce alcun effetto. La classe `XStreamingElement`, invece, mantiene il contenuto così com'è e lo processa al momento dell'emissione del markup, che avviene invocando il metodo `Save`.

Se incontra una lista enumerabile, questa viene sfogliata sul momento, riducendo al minimo le risorse impiegate: ovviamente questo comportamento varia in base al tipo di lista, poiché una normale collezione dispone già dei dati in memoria (portando quindi pochi benefici), mentre un `DbSet<T>` di **Entity Framework** carica una singola riga per volta dal database, eseguendo la query solo all'effettiva consumazione della riga stessa, con un netto miglioramento delle risorse usate.

Infine, la particolarità del metodo `save` è data dal fatto che, comunque, utilizza al suo interno un `xmlwriter` per produrre il documento, rendendo di fatto LINQ to XML solo un wrapper sulle classi basilari di .NET.

Interrogare rapidamente con XPathDocument

Tra i numerosi strumenti messi a disposizione da .NET rientrano anche classi che permettono l'utilizzo di **XPath** per l'interrogazione di documenti XML. XPath è un linguaggio standard promosso dal W3C, che semplifica la ricerca dei dati mediante una sintassi rivolta alla navigazione tra i nodi e il filtro di questi ultimi. La creazione di questo nuovo linguaggio ha principalmente lo scopo di supportare le trasformazioni XSLT (che affronteremo nel prossimo paragrafo), ma possono essere sfruttate in alternativa a LINQ per l'accesso a XML, se già si conosce tale sintassi.

Non è fra gli obiettivi di questo libro spiegare questo linguaggio, ma possiamo vederne alcuni esempi di utilizzo, associati alle relative classi del .NET, che permettono poi di ottenerne i risultati ed eventualmente di manipolare l'XML.

Navigare tra i nodi

L'oggetto principale per l'accesso mediante XPath è rappresentato da `XPathDocument`. Lo troviamo nel namespace `System.Xml.XPath` e permette di caricare l'intero documento XML in memoria mediante `URI`, `Stream` o `xmlReader`, perciò dobbiamo fare attenzione alle dimensioni del file che intendiamo caricare perché questo strumento è sottoposto al medesimo limite che ha `xmlDocument`. Una volta creata l'istanza, l'unico metodo che abbiamo a disposizione è `CreateNavigator`, il quale crea un oggetto di tipo `XPathNavigator`, fulcro di tutta la navigazione tra i nodi. Lo stesso metodo è disponibile anche dalla classe `xmlDocument`, dato che l'approccio è simile, e ci permette di unire DOM alla facilità di ricerca di XPath.

Attraverso il navigatore possiamo appunto spostarci tra gli elementi con una serie di metodi di nome `MoveTo***` che sono in grado di navigare sul rispettivo attributo, elemento, figlio, id o namespace. L'[esempio 12.14](#)

mostra come caricare il documento preso in esame in questo capitolo e come spostarsi, con una serie di metodi, all'attributo `idCategory`.

Esempio 12.14

```
' Carico il documento
var doc = new XPathDocument("test.xml");
XPathNavigator navigator = doc.CreateNavigator();

// Navigo sulla root <products>
navigator.MoveToFirstChild();

// Navigo sul primo <product>
navigator.MoveToChild("product",
    "http://schemas.aspitalia.com/book40/products");

// Mi sposto sull'attributo della categoria
navigator.MoveToAttribute("idCategory", "");

// Leggo il valore
int idCategory = navigator.ValueAsInt;
```

Possiamo notare che diversi metodi, come `MoveToFirstChild`, `MoveToChild` e `MoveToAttribute` permettono di navigare sul nodo desiderato secondo certe logiche. Ve ne sono anche altri, come `MoveToParent` o `MoveToRoot`, i quali sono tutti auto esplicativi, come possiamo vedere nell'[esempio 12.14](#).

Qualunque strada utilizziamo per raggiungere il nodo che desideriamo, ciò che conta è che abbiamo alcune proprietà le quali, in modo simile all'`XmlReader`, permettono di conoscere il tipo di nodo (`NodeType`), il nome (`LocalName`) e il namespace (`NamespaceURI`). La proprietà più importante è sicuramente `Value`, che restituisce sotto forma di stringa il valore del nodo, se questo è disponibile. Abbiamo inoltre a disposizione proprietà di nome `ValueAsBoolean`, `ValueAsDateTime`, `ValueAsInt` e `ValueAsLong` per ottenere il valore già tipizzato, se la conversione va a buon fine. Tutto questo, comunque, non paga la creazione di un ulteriore strumento di interrogazione XML, perché il vero scopo è quello di supportare XPath. I metodi `Select` e `SelectSingleNode` servono proprio a questo, in quanto accettano una stringa XPath da utilizzare per la selezione. Il primo restituisce l'intero resultset di nodi che ha trovato mentre il secondo restituisce semplicemente il primo dei nodi trovati.

Quindi, per leggere la categoria del primo prodotto, possiamo sfruttare la sintassi di navigazione tra i nodi, come nell'[esempio 12.15](#).

Esempio 12.15

```
var doc = new XPathDocument("test.xml");
XPathNavigator navigator = doc.CreateNavigator();

// Creo il gestore dei namespace
var nsManager = new XmlNamespaceManager(navigator.NameTable);
// Associo il prefisso al namespace
nsManager.AddNamespace("p",
    "http://schemas.asptalia.com/book40/products");

// Navigo direttamente sull'attributo
navigator = navigator
    .SelectSingleNode("/p:products/p:product[1]/@idCategory", nsManager);

// Leggo il valore
int idCategory = navigator.ValueAsInt;
```

La sintassi `/p:products/p:product[1]/@idCategory` vuole indicare che, partendo dal nodo radice, il motore deve ricercare i tag `products` e, a sua volta, i tag `product`. Mediante le parentesi quadre, indichiamo un'espressione di filtro e, in questo caso, di prendere solo il primo elemento; da questo, poi, navighiamo nell'attributo `idCategory`. Otteniamo così, a nostra volta, un nuovo `XPathNavigator`, che ha un riferimento al medesimo `XPathDocument` ma si trova in un'altra posizione. Possiamo fare ulteriori spostamenti, ritornare a elementi padre oppure, come nell'[esempio 12.15](#), interrogare le proprietà di nostro interesse.

La classe `XmlNamespaceManager` permette di mappare una serie di prefissi con relativi namespace, per utilizzarli in un successivo momento nelle espressioni XPath. Sempre nell'[esempio 12.15](#), il prefisso `p` viene utilizzato per fare riferimento a un preciso namespace. Dobbiamo infatti considerare che XPath è molto potente, permette di filtrare, convertire numeri e stringhe, utilizzare funzioni, lavorare per posizione, combinare i namespace e usare speciali keyword (dette axis) per ricercare nodi discendenti, ascendenti, fratelli e padri.

Dato che l'espressione è una stringa che va interpretata dal motore di XPath per migliorare le prestazioni di esecuzione, possiamo anche compilare tale stringa con il metodo `Compile` e ottenere una `XPathExpression` che la rappresenta, pronta per essere eseguita una o più volte, come viene mostrato nell'[esempio 12.16](#).

Esempio 12.16

```
// Navigo direttamente sull'attributo
XPathExpression exp =
    navigator.Compile("number(/p:products/p:product[1]/@idCategory)");
// Imposto il resolver dei namespace
exp.SetContext(nsManager);

// Leggo il valore
int idCategory = Convert.ToInt32(navigator.Evaluate(exp));
```

Nell'esempio compiliamo l'espressione e ne teniamo il riferimento, dato che a ogni suo utilizzo dobbiamo impostare il contesto e specificare qual è il resolver dei namespace da utilizzare. L'espressione può essere passata ai metodi `Select` e `SelectSingleNode`, oppure, se restituisce direttamente un valore (in questo caso il numero convertito della categoria), può essere passata al metodo `Evaluate` per ottenere immediatamente il valore ricercato.

Modificare i nodi

Oltre a permettere la navigazione tra i nodi, l'`xPathNavigator` prevede la possibilità di modificarli in modo molto simile a DOM. Tuttavia, non tutti i navigatori supportano questa funzionalità perché ciò dipende dalla loro origine: l'`xPathDocument`, per esempio, permette solo la lettura, a differenza dell'`XmlDocument`, che la permette attraverso il proprio navigatore.

Nell'[esempio 12.17](#) sfruttiamo proprio questa classe per caricare il file XML; creiamo un navigatore e usiamo il metodo `setValue` per cambiarne il valore.

Esempio 12.17

```
// Carico il documento
var doc = new XmlDocument();
doc.Load("test.xml");

// Creo il gestore dei namespace
var nsManager = new XmlNamespaceManager(doc.NameTable);
// Associo il prefisso al namespace
nsManager.AddNamespace("p",
    "http://schemas.aspitalia.com/book40/products");

// Creo il navigatore sui XmlNode
```

```
XPathNavigator navigator = doc.CreateNavigator();  
  
// Ricerca l'attributo  
navigator = navigator  
    .SelectSingleNode("/p:products/p:product[1]/@idCategory", nsManager);  
navigator.SetValue("5");
```

Abbiamo inoltre a disposizione un insieme di metodi per creare elementi e attributi oppure per spostare o eliminare, il tutto sempre applicandoli al nodo corrente di navigazione.

Nell'[esempio 12.18](#), partiamo quindi da un `XmlDocument` nuovo e con il metodo `AppendChildElement` creiamo il tag radice specificando il prefisso, il nome e il namespace. Successivamente ci spostiamo sull'elemento appena creato e impostiamo un attributo, valorizzandolo.

Esempio 12.18

```
var doc = new XmlDocument();  
var navigator = doc.CreateNavigator();  
  
// Creo il tag <products>  
navigator.AppendChildElement("p", "products",  
    "http://schemas.aspitalia.com/book40/products", "");  
navigator.MoveToFirstChild();  
  
// Creo l'attributo count  
navigator.CreateAttribute("", "count", "", "5");
```

Il metodo `AppendChildElement` prevede eventualmente di impostare il valore testuale del nodo e anche `CreateAttribute` permette, come opzione, di specificare il prefisso e il namespace.

In ogni caso, una volta manipolato il documento con il navigatore, per proseguire le modifiche dobbiamo ricorrere ai metodi di salvataggio appartenenti al creatore dell'`XPathNavigator`. Nell'[esempio 12.18](#), dobbiamo quindi invocare il metodo `save` sull'oggetto `XmlDocument`.

Trasformare i documenti con XSLT

In unione a XPath vi è un altro strumento che può essere molto utile quando dobbiamo processare documenti XML e produrne un secondo: **XSLT**. Acronimo di eXtensible Stylesheet Language Transformations, è un

linguaggio standard che ha lo scopo di produrre documenti di testo, HTML o XML, data una sorgente XML. Su di essa possiamo ricercare nodi e leggere i valori, utilizzando tutta la sintassi XPath, eventualmente anche sfruttando funzioni esterne da noi sviluppate.

In .NET le trasformazioni XSLT sono supportate e si possono effettuare mediante la classe `xslCompiledTransform` del namespace `System.Xml.Xsl`. Usando questo oggetto, quello che più conta è conoscere il linguaggio XSLT, dato che ciò che gli serve è lo stylesheet e il documento XML da trasformare.

Lo stylesheet, di fatto, è anch'esso un XML che, attraverso alcuni tag del namespace <http://www.w3.org/1999/XSL/Transform>, permette di unire XML con markup HTML o normale testo. Anche se non è scopo di questo libro spiegare in modo approfondito questo linguaggio, possiamo vederne un campione nell'[esempio 12.19](#).

Esempio 12.19 - test.xslt

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:msxsl="urn:schemas-microsoft-com:xslt"
  exclude-result-prefixes="msxsl p"
  xmlns:p="http://schemas.aspitalia.com/book40/products">
  <xsl:output method="html"
    indent="yes"/>

  <xsl:template match="/">
    <html>
      <body>
        <table>
          <xsl:for-each select="/p:products/p:product">
            <tr>
              <td>
                <xsl:value-of select="p:description"/>
              </td>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>

  </xsl:template>
</xsl:stylesheet>
```

Partendo dall'elemento radice di nome `stylesheet`, identifichiamo la trasformazione da applicare. Con l'elemento `output` impostiamo alcune direttive, come l'indentazione e il tipo di output che vogliamo produrre:

HTML. Attraverso il tag `template`, invece, andiamo a indicare che, a fronte del match sull'elemento radice (lo slash /), il motore deve applicare il suo contenuto. Nell'[esempio 12.19](#), andiamo quindi a preparare le porzioni di HTML che sono sempre presenti, fino alla tabella. Con un `for-each` andiamo a ciclare il risultato di una `select` effettuata su un'espressione XPath, la stessa vista nel paragrafo precedente, che va a ciclare sui prodotti. Per ognuno di essi, creiamo una riga con relativa cella, da valorizzare con il contenuto della descrizione.

Lo scopo di questa trasformazione è quello di preparare una pagina HTML dato un XML aderente allo schema preso in esame nell'arco di tutto questo capitolo. Creiamo quindi un'istanza della classe `XslCompiledTransform` e carichiamo la trasformazione `test.xslt` dell'[esempio 12.19](#). Come il nome della classe suggerisce, una volta caricata la trasformazione, questa viene compilata e, mantenendone un riferimento, ci permette di usarla più volte per ottenere il massimo delle prestazioni.

Successivamente, con il metodo `Transform`, passiamo l'XML da trasformare e indichiamo dove caricare il risultato, come mostrato nell'[esempio 12.20](#).

Esempio 12.20

```
var xslt = new XslCompiledTransform();
// Carico la trasformazione XSLT
xslt.Load("test.xslt");

// Trasformo l'XML e lo mostro nella finestra di Output
xslt.Transform("test.xml", null, Console.Out);
```

Poche linee di codice bastano per utilizzare un linguaggio standard e per produrre dell'HTML, con moltissimi vantaggi nella visualizzazione di come risulterà l'output finale.

Possiamo infatti vedere, nell'[esempio 12.21](#), come l'output prodotto rispetti esattamente quello che era intuibile tramite la trasformazione dell'[esempio 12.19](#).

Esempio 12.21

```
<html>
  <body>
    <table>
      <tr>
        <td>Prodotto 1</td>
      </tr>
      <tr>
        <td>Prodotto 2</td>
      </tr>
      <tr>
        <td>Prodotto 3</td>
      </tr>
    </table>
  </body>
</html>
```

Il secondo parametro del metodo `Transform` accetta facoltativamente un `XsltArgumentList`, il quale permette di passare parametri utili alla trasformazione o `extension object`: oggetti che permettono di estendere la trasformazione con funzioni personalizzate.

Conclusioni

In questo capitolo abbiamo illustrato i principali strumenti per l'interrogazione, la creazione e la modifica di documenti XML presenti in .NET Standard.

Abbiamo visto le classi `XmlReader` e `XmlWriter`, le quali ci consentono di ottenere il massimo delle prestazioni con un certo onere di implementazione ma che, di fatto, sono alla base di tutti gli altri strumenti messi a disposizione da .NET.

Con `XmlDocument` ci viene invece offerto un semplice approccio DOM e perciò risulta utile se si ha esperienza con quest'ultimo mentre, sfruttando LINQ to XML, possiamo ottenere il massimo dei benefici derivanti dall'uso della query syntax e degli `extension method` offerti da LINQ. Nel corso del capitolo abbiamo osservato come sia migliore l'organizzazione delle classi e come coprano correttamente tutte le tipologie di nodi disponibili. La versatilità di `XElement` permette di creare facilmente elementi, attributi e commenti, e di manipolarli spostandoci facilmente tra i nodi con metodi dai nomi auto esplicativi.

Qualora invece volessimo usare strumenti standard – che dovremmo già conoscere – abbiamo a disposizione `XPath` per la ricerca dei nodi e `XSLT`

per effettuare trasformazioni sui documenti.

A questo punto, abbiamo offerto una buona illustrazione di tutti gli strumenti a disposizione, così da essere in grado di determinare quello più adatto alle varie circostanze che possiamo incontrare nel corso dello sviluppo delle nostre applicazioni.

Introduzione a XAML

XAML è un linguaggio dichiarativo di markup basato su XML, usato per inizializzare strutture e oggetti. XAML era l'acronimo di Extensible Avalon Markup Language, dove Avalon era il nome in codice della prima tecnologia che lo ha usato in modo estensivo: Windows Presentation Foundation (WPF). Decaduto il nome in codice, XAML è diventato ufficialmente l'acronimo di Extensible Application Markup Language.

Prima di XAML lo sviluppo delle applicazioni in ambiente Windows era interamente basato sulle GDI/GDI+, usate ampiamente in WinForms: una tecnologia rilasciata nel 2002 con la prima versione del .NET Framework.

Con XAML, per la prima volta in ambiente Windows viene introdotto un linguaggio dichiarativo, per certi versi simile al ben più famoso e diffuso HTML. Da XML e HTML, XAML prende la leggibilità tipica dei linguaggi dichiarativi. Il Markup è di semplice interpretazione e non richiede particolari conoscenze tecniche.

Questa semplicità ne ha portato l'adozione come linguaggio di markup anche su altre piattaforme di sviluppo come Xamarin, tecnologia con la quale possiamo realizzare app mobile per iOS e Android.. Con l'avvento di Windows 8/10, inoltre, è stato introdotto un nuovo runtime che permette la realizzazione di applicazioni su più piattaforme che prende il nome di Universal Windows Platform (UWP), il quale anch'esso fa uso di XAML per la definizione dell'interfaccia. Ai giorni nostri, a seguito dell'unificazione della piattaforma di sviluppo, lo XAML è ancora la scelta più produttiva per la definizione dell'interfaccia. Questa uniformità va ad appannaggio del miglioramento dei tempi di sviluppo e della semplicità di

manutenzione. In questo capitolo vogliamo analizzare XAML applicandolo alle tecnologie Windows attualmente adottabili: WPF o UWP.

L'ambiente di sviluppo

A prescindere dalla tipologia di applicazione che vogliamo realizzare, a fornire l'infrastruttura di base è la classe `Application`. Nonostante le ovvie differenze, questa classe accomuna tutte le piattaforme che utilizzano XAML come linguaggio dichiarativo per la realizzazione delle interfacce grafiche.

La classe `Application` ha lo scopo di fornire l'infrastruttura necessaria a gestire il ciclo di vita dell'applicazione, le risorse (vedremo in dettaglio di cosa si tratta più avanti nel libro), l'UI e il sistema di navigazione.

Ogni applicazione WPF, UWP è realizzata utilizzando Visual Studio, ed è composta da un insieme di asset che ne costituiscono il nucleo minimo di funzionamento. L'organizzazione dell'IDE di Visual Studio è visibile nella [Figura 13.1](#).

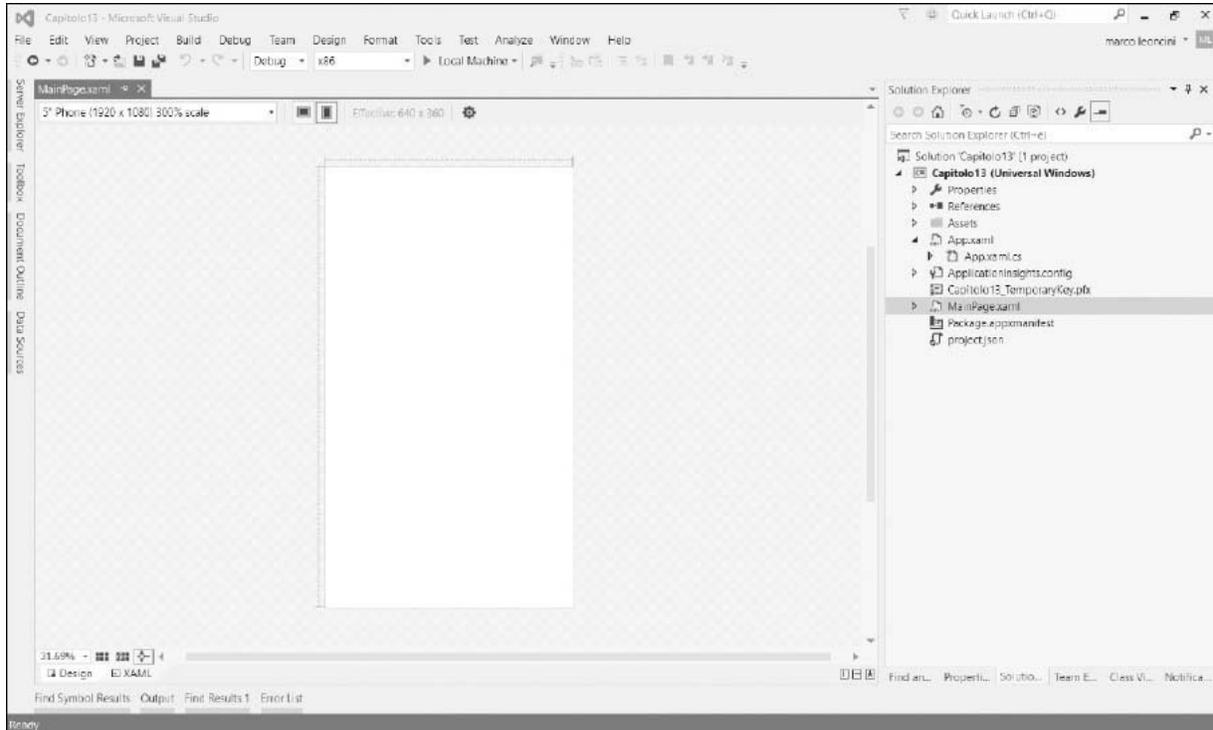


Figura 13.1 – L'IDE di Visual Studio.

La parte di definizione dell'interfaccia avviene mediante l'utilizzo del linguaggio XAML (si legge zamel) all'interno di file con l'omonima estensione `.xaml`. Per ognuno di questi file esiste il corrispettivo `.cs`, chiamato file di code behind. Generalmente, questo tipo di file è usato per la gestione degli eventi sollevati dall'interfaccia utente, come la pressione di un bottone o la gestione delle animazioni. Il file di code behind non è strettamente necessario, e se la nostra UI è estremamente semplice possiamo tranquillamente rimuoverlo. Visual Studio non permette la creazione di file xaml che ne siano privi. Quindi è necessario, una volta rimosso il file `.cs` inutilizzato, rimuovere il riferimento a quest'ultimo. Per farlo è sufficiente identificare nel file xaml l'attributo `x:Class`, che generalmente troviamo applicato sul nodo root, e rimuoverlo.

Fin dall'introduzione dello XAML come linguaggio di markup, Microsoft decise di affiancare a Visual Studio un tool pensato per sfruttarne le potenzialità multimediali: Blend. Con il tempo, molte delle funzionalità di Blend sono state incorporate in Visual Studio, con un'unica eccezione: la creazione delle animazioni. L'interfaccia utente di Blend è ormai uniformata a quella di Visual Studio, del quale ne adotta il medesimo layout.

La [Figura 13.2](#) mostra l'IDE di Blend for Visual Studio: quest'ultimo può essere utilizzato sia come programma stand alone sia, in alternativa, richiamato da Visual Studio semplicemente dal menu contestuale su qualsiasi file con estensione XAML.

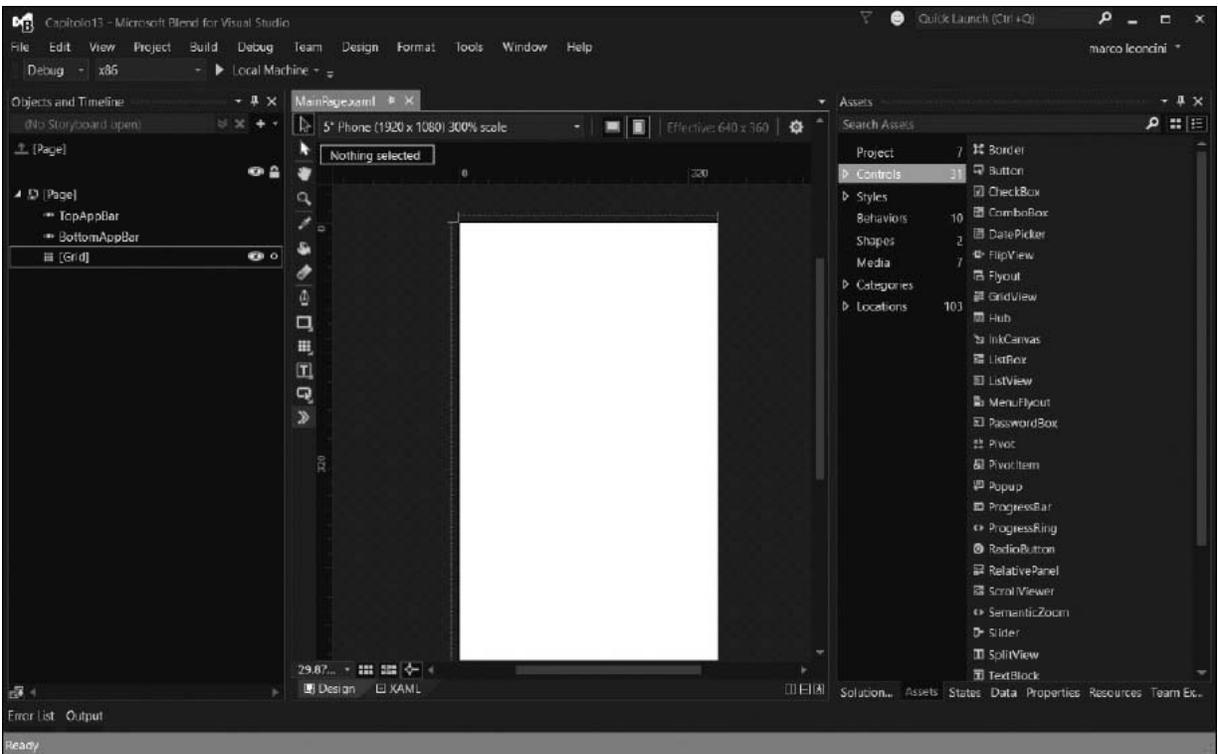


Figura 13.2 – L’IDE di Blend for Visual Studio.

Il markup XAML

XAML è un linguaggio dichiarativo, come abbiamo detto, simile all’HTML e all’XML. Dal primo prende la semplicità mentre dal secondo ne trae il rigore della sintassi e della correttezza del formato. Un file XAML dichiara al suo interno una serie di elementi, ognuno dei quali rappresenta un’istanza di un oggetto. Dichiarare l’elemento `<Grid>` è l’equivalente da codice della creazione di una nuova istanza del controllo `Grid`. Il compito di interpretare i tag presenti nello XAML e di istanziare gli oggetti relativi è a carico del parser XAML.

La sintassi

Allo scopo di poter gestire la complessità necessaria alla realizzazione d’interfacce utente complesse e articolate, composte da decine di oggetti,

XAML dispone di alcune comode regole di sintassi da usare in base al problema da risolvere.

La sintassi Object element

Questo tipo di sintassi rappresenta un'istanza di un oggetto; ogni elemento è formato da una parentesi angolare aperta "<" il nome della classe da istanziare, seguito da una slash e da una parentesi angolare chiusa ">". L'[esempio 13.1](#) mostra il markup necessario a creare un'istanza del tipo `Button`.

Esempio 13.1 – XAML

```
<Button>  
  Hello!  
</Button>
```

Nel frammento di codice precedente, l'inserimento della stringa di testo "hello" è equivalente a impostare la proprietà `Content` della classe `Button`.

La sintassi Property Attribute

Se la semplice istanza dell'oggetto non è sufficiente, ne possiamo impostare le proprietà utilizzando il nome della stessa come attributo di un elemento. Nell'[esempio 13.2](#) impostiamo il colore di sfondo di un bottone. Quindi il nome dell'attributo rappresenta il nome della proprietà e la stringa dopo il simbolo dell'uguale ne rappresenta il valore.

Esempio 13.2 – XAML

```
<Button Background="Blue" />
```

Ci sono proprietà che comunque possono difficilmente essere rappresentate da una semplice stringa. In questi casi possiamo ricorrere a una sintassi

alternativa, chiamata **Property Element**.

La sintassi Property Element

In questo tipo di sintassi, il valore della proprietà è espresso utilizzando l'Object Element, mentre la proprietà da impostare segue la sintassi `Type.PropertyName`, come possiamo vedere nell'[esempio 13.3](#), e prende forma come elemento interno all'oggetto che la espone.

Esempio 13.3 – XAML

```
<Button>
  <Button.Background>
    <SolidColorBrush Color="Red" />
  </Button.Background>
</Button>
```

Nel precedente esempio creiamo un'istanza di una classe `Button` e impostiamo la proprietà `Background`, utilizzando una nuova istanza di `SolidColorBrush`.

Affinché sia possibile creare un'istanza della classe dichiarata nel markup, è necessario che il tipo disponga di un costruttore pubblico privo di parametri.

I namespace

In un documento XAML, l'elemento root dichiara uno o più namespace comuni a ogni piattaforma:



`xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation`: con questa dichiarazione è mappata una grande quantità di namespace delle librerie; è il namespace di default e non necessita prefissi;

- ❑ `xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml`: è un namespace a supporto delle caratteristiche del linguaggio XAML; tipicamente ha il prefisso `x` e può servire, come abbiamo visto mediante l'attributo `x:class`, a legare il file `.xaml` al relativo file di codebehind.

I namespace, indicano al parser in quale assembly cercare i tipi da utilizzare per creare le istanze degli elementi definiti nel documento XAML. Poiché un namespace XAML può mappare più assembly e più namespace del CLR, è necessario che non vi sia sovrapposizione dei nomi tra le classi, altrimenti il parser non sarà in grado di distinguere quale tipo istanziare.

Oltre ai namespace predefiniti, possiamo definirne di nuovi per utilizzare dei tipi definiti da noi.

Esempio 13.4

```
namespace Chapter13
{
    public class TestClass
    {
        public TestClass(){}
        public override string ToString()
        {
            return "Questo è un tipo non definito nel CLR";
        }
    }
}
```

Per utilizzare la classe dell'[esempio 13.4](#) in un documento XAML, è sufficiente aggiungere in WPF all'elemento root il seguente namespace: `xmlns:local="clr-namespace:Chapter13"`, mentre nelle Windows Universal app: `xmlns:local="using:Capitolo13"`. Con questa è possibile associare il namespace del CLR al namespace XAML.

Esempio 13.5 – XAML

```
<UserControl x:class="Chapter13.UserControl1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    WindowTitle="Capitolo 12"
    WindowWidth="300"
```

```
WindowHeight="300"  
  xmlns:local="clr-namespace:Chapter13">  
  <local:MyClass></local:MyClass>  
</UserControl>
```

Quando il parser si trova a visualizzare un tipo definito dall'utente, nel caso in cui non si tratti di un elemento visuale o non sia specificato diversamente, viene richiamato il metodo `ToString`, che ogni oggetto del CLR espone, come è possibile vedere nella [Figura 13.3](#).



Questo è un tipo non definito nel CLR

Figura 13.3 – La renderizzazione del nostro tipo custom.

Come possiamo comprendere dall'[esempio 13.5](#) e dalla [Figura 13.3](#), il rischio di riempire l'elemento root con definizioni di namespace è davvero alto. Per evitarlo solo nelle applicazioni WPF, possiamo utilizzare l'attributo `xmlnsDefinitionAttribute` e associare più namespace del CLR a un solo namespace XML.

Il layout system

Ogni elemento dell'interfaccia occupa uno spazio che è chiamato bounding box, il quale viene definito dal layout system: è un processo ricorsivo che misura, arrangia, dispone e si occupa della renderizzazione. Questo processo è fondamentale per il funzionamento delle applicazioni realizzate utilizzando XAML, che si differenzia da molti altri framework per la distinzione tra gli elementi fisici e logici che ne compongono l'interfaccia.

È possibile recuperare tutte le informazioni sullo slot occupato da ogni elemento, utilizzando la classe `LayoutInformation`.

Elementi fisici e logici

Quando in un documento XAML, attraverso il markup, definiamo gli elementi dell'interfaccia, andiamo a delineare, a tutti gli effetti, una grafo che viene chiamato **Logical Tree**. Nell'[esempio 13.5](#), in pratica, abbiamo creato una gerarchia che ha come radice l'oggetto `UserControl` e un unico figlio costituito dall'oggetto `Button`.

Ogni elemento logico può essere formato da uno o più elementi grafici. Per esempio, il controllo `Button` è costituito da più elementi visuali, e questa gerarchia prende il nome di **Visual Tree**.

In genere, i controlli espongono proprietà che permettono di navigare e popolare il Logical Tree. Per esempio, possiamo accedere al contenuto degli oggetti che ereditano dalla classe `ContentControl`, attraverso la proprietà `Content`.

Diversamente, nessun controllo espone la propria gerarchia visuale, che rimane "occultata". Possiamo navigare attraverso il Visual Tree, utilizzando la classe `VisualTreeHelper` la quale espone una serie di metodi utili per recuperare ogni informazione sull'aspetto visuale di un elemento.

La disposizione degli elementi

La disposizione degli elementi di un'interfaccia ha sempre un'importanza fondamentale per l'usabilità di un'applicazione. Nell'ambito delle tecnologie Microsoft la disposizione degli elementi è affidata a un tipo particolare di oggetti. Queste classi che estendono il tipo `Panel` sono predisposte per posizionare gli elementi contenuti, in base ad alcune regole dipendenti dal tipo utilizzato.

I pannelli

La classe `Panel` è una classe astratta che fornisce l'infrastruttura per la realizzazione di classi specializzate per il posizionamento degli elementi dell'interfaccia utente. A questo scopo, esiste una serie di pannelli che risponde alle più semplici esigenze:

- ❑ `Canvas`: è il più semplice dei pannelli. Gli elementi figlio sono disposti ,mediante coordinate assolute, relativamente a una coppia di margini, attraverso le `AttachedProperty Left, Top, Right e Bottom`;
- ❑ `Grid`: gli elementi sono disposti in righe e colonne;
- ❑ `StackPanel`: gli elementi sono disposti l'uno di seguito all'altro, verticalmente oppure orizzontalmente, in base alle proprietà `Orientation`;
- ❑ `VirtualizingStackPanel`: la disposizione degli elementi è identica a quella eseguita dallo `StackPanel` ma è ottimizzata per un numero elevato di elementi, calcolando solamente quelli effettivamente visibili.

Esempio 13.6 – XAML

```
<UserControl 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"
  Width="300"
  Height="300">
```

```
<StackPanel>  
  <Button>1</Button>  
  <Button>2</Button>  
  <Button>4</Button>  
  <Button>5</Button>  
  <Button>6</Button>  
</StackPanel>  
</ UserControl >
```

Nella [Figura 13.4](#), nell'immagine viene visualizzato il risultato dell'[esempio 13.6](#).

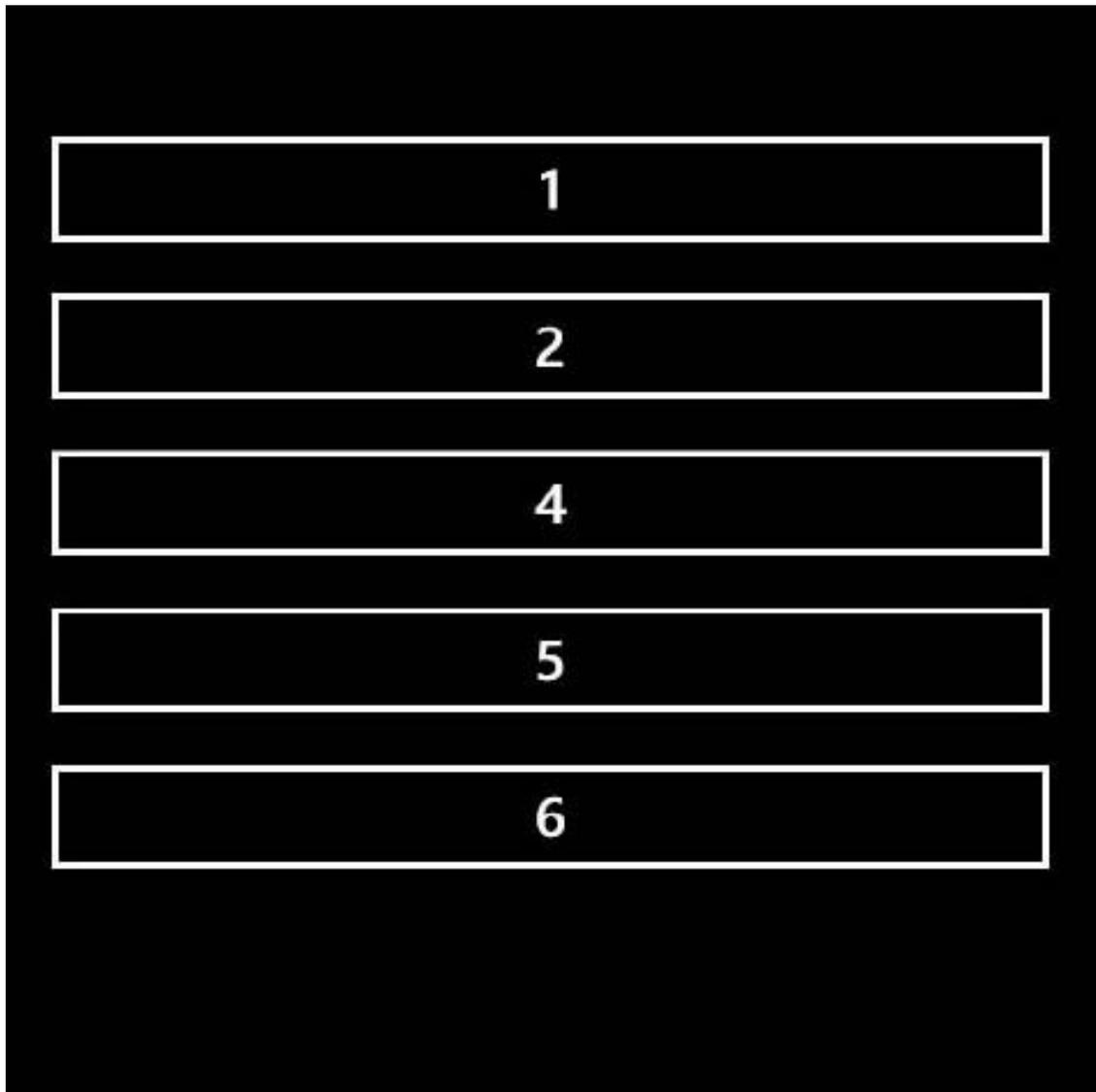


Figura 13.4 – I Button ordinati da uno StackPanel.

Nelle UWP sono disponibili pannelli specializzati per creare le tipiche interfacce delle applicazioni moderne. Come, per esempio, il `VariableSizedWrapGrid`, con il quale è possibile disporre gli elementi sotto forma di griglia e decidere lo spazio da allocare a ogni singolo elemento.

Nell'[esempio 13.7](#), invece, sostituiamo il pannello `StackPanel` con un `Canvas`. Oltre a cambiare il tipo di pannello, è necessario indicare come disporre i vari bottoni all'interno del pannello stesso: possiamo farlo usando le `AttachedProperty`.

Esempio 13.7 – XAML

```
<UserControl 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"
  Width="300"
  Height="300">
  <Canvas>
  <Button>1</Button>
  <Button Canvas.Top="90">2</Button>
  <Button Canvas.Top="190">4</Button>
  <Button Canvas.Top="300">5</Button>
  <Button Canvas.Top="390">6</Button>
  </Canvas>
</UserControl >
```

Nella [Figura 13.5](#) possiamo apprezzare, oltre alla differente posizione degli elementi, le diverse dimensioni. Questo perché il `Canvas`, in modo differente da quanto fa lo `StackPanel`, alloca agli elementi figli solo lo spazio di cui hanno effettivamente bisogno e non tutto lo spazio disponibile.

Anche solo grazie a questi due semplici esempi, è possibile intravedere la flessibilità dell'uso dei pannelli rispetto al vecchio e unico modello “cartesiano” tipico di Windows Form o di altri sistemi operativi.

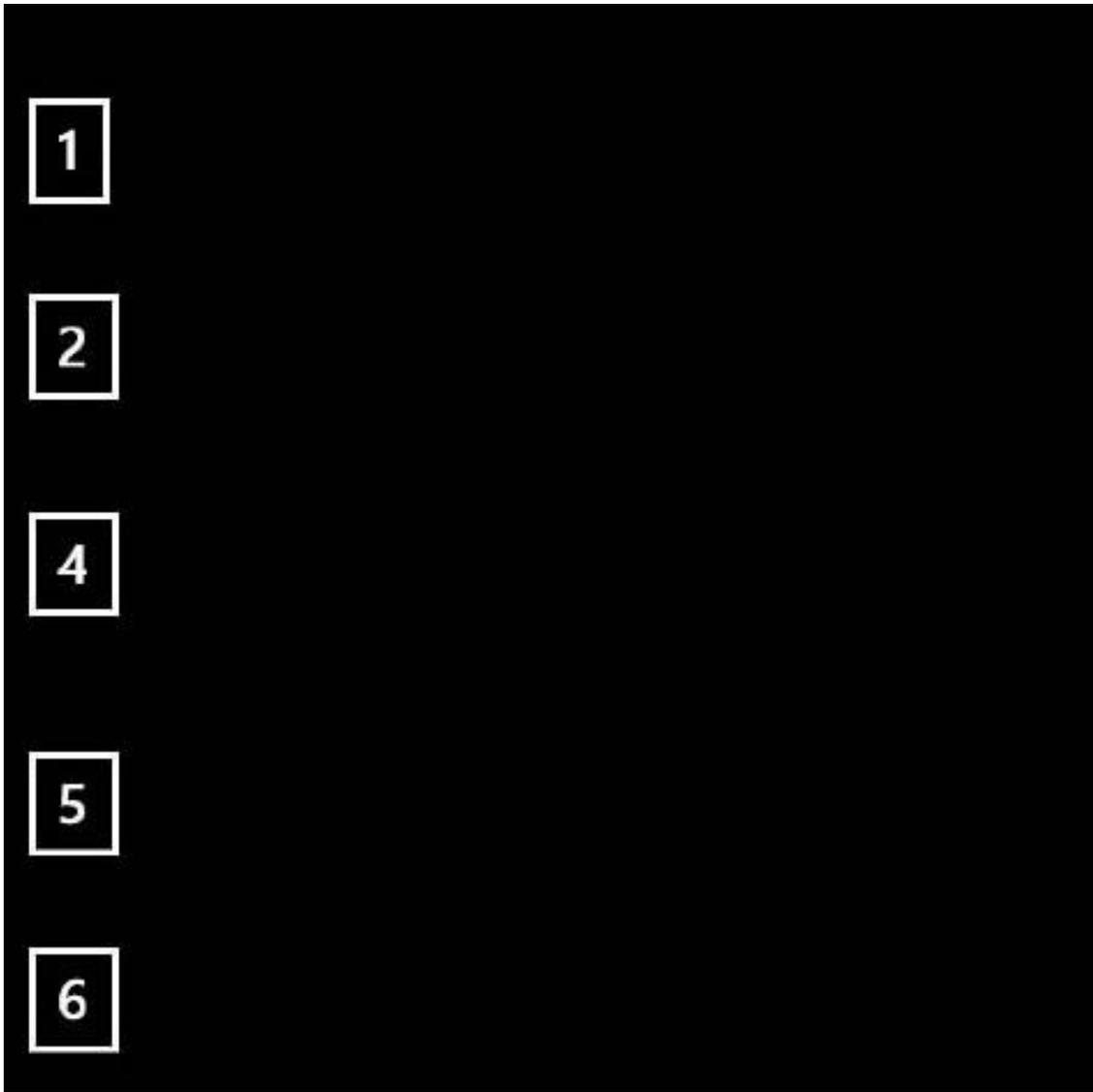


Figura 13.5 – I Button ordinati all'interno di un canvas.

I Pannelli, da soli, non possono essere usati per creare un'applicazione; serve qualcosa con cui l'utente possa interagire: i controlli.

I controlli

L'interfaccia di ogni moderna applicazione è formata da una serie di elementi che permette all'utente di interagire con i dati dell'applicazione stessa. Questi elementi prendono il nome di controlli. Le applicazioni Universal e WPF dispongono di un set completo e general purpose di

controlli. Questi controlli permettono la realizzazione anche di interfacce grafiche molto complesse.

I controlli sono definiti lookless, cioè privi di look ma, benché questo non sia letteralmente corretto, si meritano questo appellativo per la loro capacità di cambiare completamente aspetto, ridefinendo il proprio Visual Tree con l'impostazione della proprietà `Template`.

Le classi principali: UIElement e FrameworkElement

Ogni controllo eredita indirettamente dal tipo `FrameworkElement` che fornisce i servizi di Layout, il supporto al `DataBinding` agli `Style` e ai `Template` ed estende il supporto alle animazioni, già offerto dalla classe `UIElement`.

La classe `UIElement` fornisce un supporto base alla gestione dell'input dell'utente alle animazioni ed espone i metodi che le classi derivate possono sovrascrivere per partecipare al sistema di Layout.

I controlli

Volendo dividere i controlli, possiamo distinguerli in due grandi categorie: i controlli che derivano direttamente o indirettamente dalla classe `ContentControl` e quelli che derivano da `ItemsControl`. Un esempio di classe derivata dal tipo `ContentControl` è il controllo `Button`. Nella sua semplicità, introduce concetti che possiamo applicare anche a controlli nettamente più complessi.

Tutti i controlli derivati dalla classe `ContentControl` espongono la proprietà `Content` del tipo `Object`. Ciò permette, dal semplice `Button` allo `UserControl`, di ospitare qualsiasi tipo di contenuto, dalla semplice stringa di testo fino a un complesso `Logical Tree`.

Per esempio, per visualizzare un'immagine all'interno di un `Button`, non dobbiamo creare una classe personalizzata o un controllo diverso ma possiamo più semplicemente, come è possibile vedere nell'immagine 12.8, creare una nuova istanza del tipo `Image` e assegnarla alla proprietà `Content`.

Utilizzando XAML, la procedura non è diversa da quella di impostare una qualsiasi proprietà.

Esempio 13.8 – XAML

```
<Button>
  <Image Height="50"
    Source="Desert.jpg"
    Stretch="Fill" />
</Button>
```

Naturalmente il contenuto non è limitato a un solo elemento ma, utilizzando i `Panel`, possiamo arrangiare più elementi all'interno di un `Button`, esattamente come accade per uno `UserControl`. Anche se il `ContentControl` può contenere, non direttamente, più di un elemento, va considerato che ha come contenuto sempre e solo un figlio. Possiamo superare questo limite utilizzando la classe `ItemsControl`.

Il tipo `ItemsControl` espone il proprio contenuto attraverso la proprietà `Items` ed è la classe base per una serie di controlli, come la `ListBox`, un controllo utile per gestire la visualizzazione e la selezione di uno o più elementi. La capacità di selezionare non è fornita dalla classe `ItemsControl` ma da `selector`, uno dei suoi tipi derivati. Diversamente da quanto avviene in altre piattaforme di sviluppo, il contenuto della `ListBox` può essere qualsiasi tipo di oggetto, dalla semplice stringa, passando per le immagini, fino a poter utilizzare, per un singolo item, un grafo complesso di elementi.

Esempio 13.9 – XAML

```
<ListBox>
  <Image Height="80"
    Source="3.1.png"
    Stretch="None" />
  <Image Height="80"
    Source="xp.png"
    Stretch="none" />
  <Image Height="80"
    Source="vista.png"
    Stretch="none" />
  <Image Height="80"
    Source="8.png"
    Stretch="none" />
```

```
</ListBox>
```

Non esistono limiti alla composizione del layout: possiamo realizzare le più complesse interfacce utente, combinando tutti i controlli presenti nelle varie piattaforme. Nella [Figura 13.6](#) possiamo vedere il controllo `ListBox` creato nell'[esempio 13.9](#).

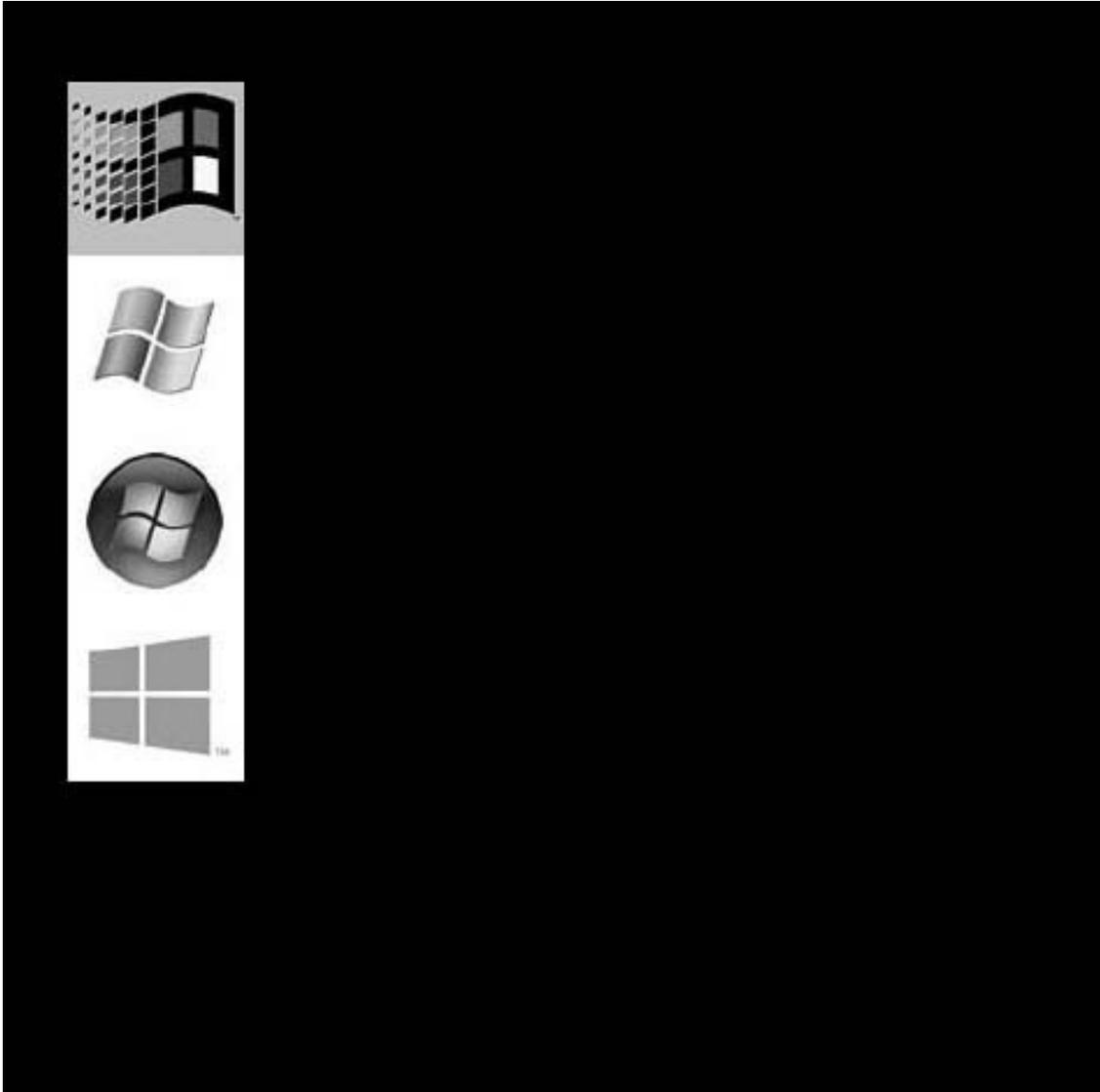


Figura 13.6 – Il controllo `ListBox`.

La possibilità di utilizzare per ogni singolo elemento della lista qualsiasi tipo di contenuto non pone limiti alla quantità o all'aspetto che ognuno di

essi può assumere. Nel nostro caso ogni elemento è rappresentato da un'immagine, ma possiamo adattare l'output, in caso di necessità, a visualizzazioni differenti.

La grafica

Con XAML è possibile soddisfare tanto le esigenze del grafico quanto quelle del programmatore. Con poche righe di markup è possibile creare una nutrita serie di oggetti, finalizzati a renderizzare forme sullo schermo. Poiché questi elementi ereditano da `FrameworkElement`, possono essere inseriti direttamente all'interno di `Panel` o in controlli come `Button` o `ListBox`. Tutti questi elementi, che ereditano dalla classe base `Shape`, espongono le seguenti proprietà:

- ❑ `Fill`: di tipo `Brush`, rappresenta il riempimento della forma;
- ❑ `Stroke`: di tipo `Brush`, rappresenta il riempimento del bordo della forma;
- ❑ `StrokeThickness`: di tipo `Thickness`, specifica lo spessore del bordo della forma.

Utilizzando i tipi derivati da `Shape`, possiamo disegnare: `Ellipse`, `Line`, `Path`, `Polygon`, `Polyline`, `Rectangle`, nell'ordine, per disegnare un'ellisse o un cerchio, una linea, una figura complessa, un poligono, una polilinea o un rettangolo.

Esempio 13.10 – XAML

```
<Button>
  <Grid>
    <Rectangle Fill="Red"
      Width="45.6"
      Height="48.8"
      HorizontalAlignment="Left"
      VerticalAlignment="Top" />
    <Ellipse Fill="#FF3694FF"
      Margin="157.4,0,45.6,0"
      Width="70.4" />
    <Path Fill="#FF7F7F7F"
```

```
Stretch="Fill"
HorizontalAlignment="Left"
Margin="72.3,6.5,0,3.7"
Width="58.9"
Data="M72.299999,30.2 L86.999997,6.5000006 130.2,32.100001
98.199997,44.100001 z" />
</Grid>
</Button>
```

Nell'[esempio 13.10](#) abbiamo inserito, all'interno di un `Button`, una serie di forme; per ognuna abbiamo specificato il colore di riempimento, impostando la proprietà `Fill` e alcune proprietà per aggiustarne la posizione.

[I pennelli: il Brush](#)

Disegnare una forma non avrebbe senso se, una volta lanciata l'applicazione, questa non fosse visibile. Dobbiamo quindi riuscire a visualizzare una forma che sia colorata.

Con XAML possiamo colorare gli oggetti, utilizzando le classi derivate dal tipo `Brush`, le quali forniscono un modo differente di applicare il colore a una forma:

- ❑ `SolidColorBrush`: è il riempimento più semplice, rappresentato da un riempimento pieno di un colore uniforme, che può essere impostato attraverso la proprietà `Color`;
- ❑ `LinearGradientBrush`: rappresenta una sfumatura che ha un'origine e una fine, che possiamo specificare impostando le proprietà `StartPoint` e `EndPoint`. Tra questi due punti, i colori che costituiscono la sfumatura possono essere illimitati e possono essere impostati mediante `GradientStops`;
- ❑ `RadialGradientBrush`: (solo WPF) è simile al `LinearGradientBrush` ma la sfumatura è radiale e possiamo impostarne l'aspetto mediante le proprietà `RadiusX` e `RadiusY`;
- ❑ `ImageBrush`: il riempimento non è più un colore ma un'immagine, per coprire completamente l'oggetto al quale è applicata.

Nell'[esempio 13.11](#) creiamo un cerchio e lo coloriamo mediante un gradiente composto da tre colori.

Esempio 13.11 – XAML

```
<Ellipse>
  <Ellipse.Fill>
    <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
      <GradientStop Color="#FFEA0F0F" Offset="0"/>
      <GradientStop Color="Black" Offset="1"/>
      <GradientStop Color="White" Offset="0.477"/>
    </LinearGradientBrush>
  </Ellipse.Fill>
</Ellipse>
```

Possiamo disegnare forme di qualsiasi genere sfruttando i `Path` ma, per effetti davvero sorprendenti, non possiamo non citare le trasformazioni.

Le trasformazioni sugli oggetti

Nessun sistema grafico potrebbe dirsi completo se non disponesse di un sistema per elaborare l'aspetto dei propri elementi. Con XAML possiamo applicare trasformazioni, come ad esempio, la scala non uniforme, semplicemente impostando la proprietà `RenderTransform`, esposta da qualsiasi tipo erediti direttamente o indirettamente dalla classe `UIElement`. Tutte le trasformazioni messe a disposizione ereditano dal tipo `Transform` e ognuna di esse conferisce un effetto differente all'elemento al quale è applicata:

- ❑ `TranslateTransform`: trasporta l'elemento al quale è applicata, specificando le coordinate `x` e `y` mediante le proprietà `x` e `y`;
- ❑ `SkewTransform`: applica una trasformazione di slittamento, impostando le proprietà `AngleX` e `AngleY`. Per esempio, un rettangolo cui viene applicata una trasformazione di questo tipo può diventare un trapezio;
- ❑ `ScaleTransform`: applica una trasformazione di scalatura, controllabile attraverso le proprietà `ScaleX` e `ScaleY`;

- ❑ `RotateTransform`: applica una trasformazione di rotazione, della quale possiamo controllare l'angolo mediante la proprietà `Angle`, e il centro mediante `CenterX` e `CenterY`.

Esempio 13.12 – XAML

```
<Rectangle Fill="#FFFE1F1F"
  Width="50"
  Height="50">
  <Rectangle.RenderTransform>
    <RotateTransform Angle="-42" />
  </Rectangle.RenderTransform>
</Rectangle>
<Rectangle Fill="#FFFE1F1F"
  Width="50"
  Height="50"
  Grid.Column="1">
  <Rectangle.RenderTransform>
    <ScaleTransform ScaleX="0.6" />
  </Rectangle.RenderTransform>
</Rectangle>
<Rectangle Fill="#FFFE1F1F"
  Width="50"
  Height="50"
  Grid.Row="1"
  Grid.Column="1">
  <Rectangle.RenderTransform>
    <TranslateTransform X="42"
      Y="35" />
  </Rectangle.RenderTransform>
</Rectangle>
<Rectangle Fill="#FFFE1F1F"
  Width="50"
  Height="50"
  Margin="46.4,41"
  Grid.Row="1">
  <Rectangle.RenderTransform>
    <SkewTransform AngleX="-31" />
  </Rectangle.RenderTransform>
</Rectangle>
```

Nell'[esempio 13.12](#), al medesimo rettangolo viene applicata ogni volta una trasformazione differente, producendo gli effetti che possiamo vedere nella [Figura 13.7](#).

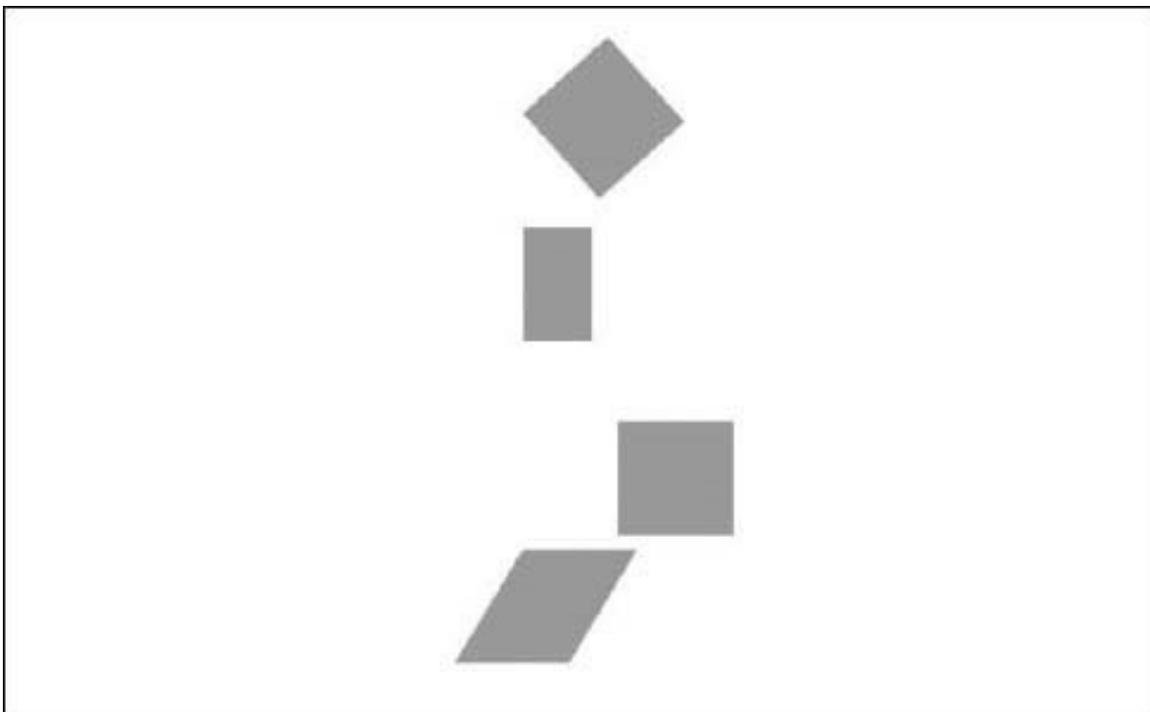


Figura 13.7 – Una serie di trasformazioni applicata ad alcuni elementi.

La proprietà `RenderTransform` accetta una sola istanza del tipo `Transform`, impedendo, di fatto, la possibilità di applicare una collezione di trasformazioni. Per ovviare a questo problema, possiamo utilizzare il tipo `TransformGroup`, il quale eredita dal tipo `Transform`, ma applicando più trasformazioni, aggiungendole alla collezione `children`. Dobbiamo prestare attenzione all'ordine nel quale sono dichiarate le trasformazioni all'interno della collezione, in quanto l'ordine può produrre effetti differenti se, per esempio, prima di ruotare un oggetto, lo abbiamo precedentemente spostato. Le trasformazioni applicate mediante `RenderTransform` non influenzano altri elementi se non quello che imposta la proprietà. È possibile invece, solo in WPF, mediante la proprietà `LayoutTransform`, modificare anche il layout degli elementi circostanti. Per esempio, se ingrandiamo un elemento, quelli nelle vicinanze si sposteranno di conseguenza.

[Le animazioni](#)

Quello che differenzia le nuove tecnologie da quelle esistenti su Windows prima dell'avvento di WPF è la totale integrazione del sistema di animazioni direttamente nei tipi che costituiscono le basi di .NET.

Con XAML possiamo animare gli elementi dell'interfaccia, alternandone le proprietà (per esempio, modificando la proprietà `opacity` per far apparire dal nulla un oggetto).

Affinché questo sia possibile, sono necessari alcuni requisiti: la proprietà deve essere una `DependencyProperty` mentre l'oggetto che la espone deve ereditare da `DependencyObject`.

Le animazioni sono create utilizzando la classe `Storyboard` e impostandone una o alcune. Difficilmente queste ultime sono create utilizzando l'ambiente RAD di Visual Studio o create a mano, ma è più realistico pensare che il tutto avvenga utilizzando Blend, il quale nasce non solo per facilitare il disegno delle interfacce basate su XAML ma, soprattutto, per creare accattivanti animazioni.

Con Blend, realizzare un'animazione è semplicissimo: è sufficiente creare un nuovo `Storyboard` dal pannello "Object and Timeline", posizionare la testina di riproduzione (la riga gialla sullo storyboard) e cambiare i valori delle proprietà che vogliamo animare. Per esempio, per animare la posizione di un elemento dell'interfaccia, sarà sufficiente trascinarlo nell'artboard: Blend for Visual Studio registrerà i nostri spostamenti e creerà l'animazione.

Le animazioni supportate possono essere lineari, ovvero con un andamento costante nel tempo, variando una proprietà solo da un valore all'altro. Ma possono anche utilizzare `keyframe`, ovvero fotogrammi chiave, nei quali una determinata proprietà deve raggiungere un determinato valore.

La [Figura 13.8](#) mostra l'interfaccia di Blend for Visual Studio durante la registrazione di uno `Storyboard`: a sinistra possiamo notare la timeline, lo strumento che ci consente di controllare il tempo di ogni singola animazione.

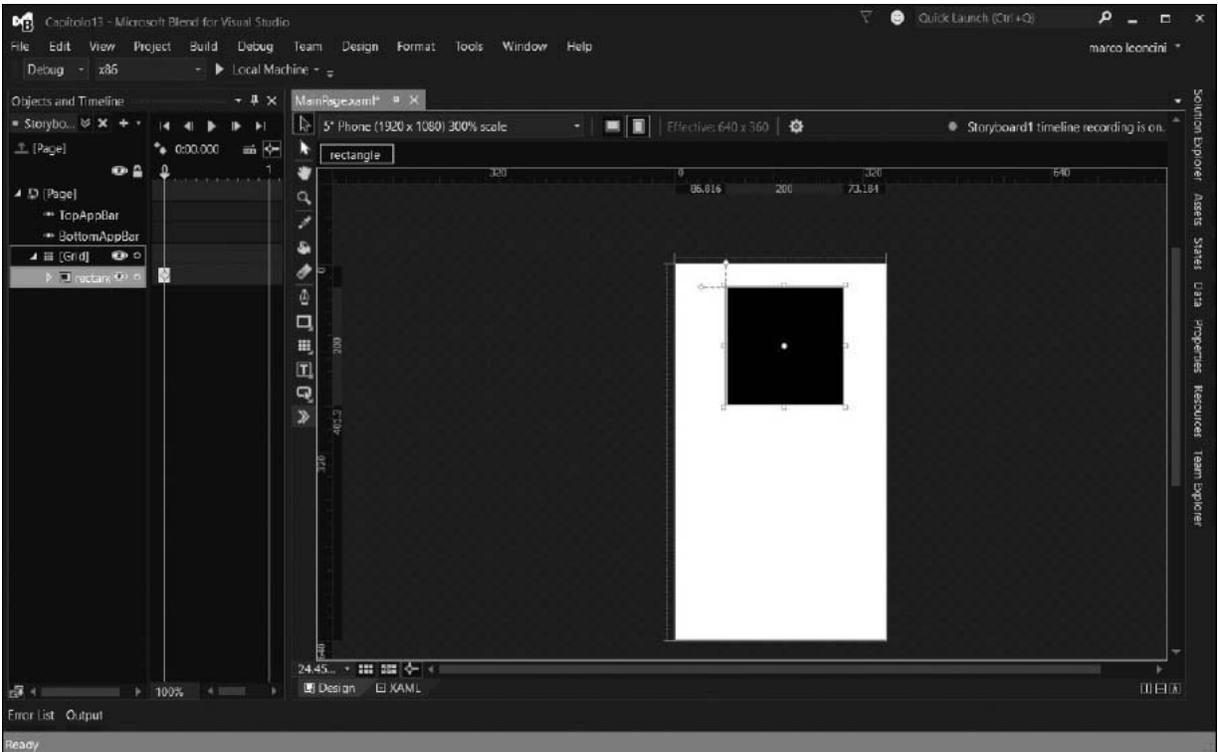


Figura 13.8 – Blend for Visual Studio durante la registrazione di un'animazione.

La registrazione dello storyboard nella [Figura 13.8](#), ovvero la registrazione di come le proprietà cambiano nel tempo, genera lo XAML che possiamo vedere nell'[esempio 13.13](#).

Esempio 13.13 – XAML

```
<Storyboard x:Name="Chapater13Storyboard">
  <DoubleAnimation Duration="0"
    To="6.816"

Storyboard.TargetProperty="(UIElement.RenderTransform).
(CompositeTransform.TranslateX)"
    Storyboard.TargetName="rectangle"
    d:IsOptimized="True" />
  <DoubleAnimation Duration="0"
    To="-181.3"
    Storyboard.TargetProperty="(UIElement.
RenderTransform).(CompositeTransform.TranslateY)"
    Storyboard.TargetName="rectangle"
    d:IsOptimized="True" />
</Storyboard>
```

Vista la complessità dello XAML necessario a creare anche una semplice animazione, Blend for Visual Studio è, senza dubbio, uno strumento indispensabile per la realizzazione di animazioni, permettendoci di creare e visualizzare in tempo reale l'animazione senza la necessità di eseguire l'applicazione. Spesso le animazioni sono eseguite in seguito a un determinato evento; per iniziare l'esecuzione di un'animazione, è sufficiente utilizzare nel code behind il metodo `Begin`: quindi, utilizzando il nome assegnato allo Storyboard, possiamo scrivere: `Chapater13Storyboard.Begin`.

Conclusioni

XAML rappresenta, senza dubbio, una rivoluzione per il layer di presentazione di Windows. L'integrazione con le DirectX e la possibilità di creare grafiche raffinate senza alterarne la logica di funzionamento, rende questo strumento indispensabile per creare rapidamente interfacce grafiche complesse e all'avanguardia.

La possibilità di creare grafi complessi di oggetti risponde alle esigenze di produrre interfacce ricche di dettagli e sempre più orientate a fornire all'utente un'esperienza appagante.

La possibilità di utilizzare riempimenti, come immagini, oggetti visuali, video e geometrie, consente di ottenere una gamma di effetti molto complessi.

La possibilità di animare ogni singola proprietà di ogni oggetto apre scenari che, in passato, non erano semplicemente immaginabili, lasciando uno spazio sempre più ampio alla creatività.

Nel prossimo capitolo proseguiremo il viaggio all'interno XAML, affrontando tematiche più avanzate quali il data binding, gli stili, i template, in modo da comprendere e approfondire le potenzialità che offre questa tecnologia e saper scegliere come e quando applicarla.

Sviluppare con XAML: concetti avanzati

Nel capitolo precedente abbiamo introdotto XAML, analizzando quali sono le sue principali caratteristiche ed evidenziando qual è il netto distacco rispetto alle vecchie applicazioni basate sul disegno con posizioni assolute, che contraddistingue Visual Basic (fino alla versione 6), WinForm, MFC e, in generale, Win32. Proseguiamo ora questo viaggio, affrontando le funzionalità più evolute di XAML, molte delle quali sono nuove per il mondo dello sviluppo di applicazioni.

XAML presenta un buon numero di differenze tra i vari framework, soprattutto in WPF: trattare ogni singola differenza richiederebbe la scrittura di un altro libro. Perciò, come abbiamo già fatto nel capitolo precedente, anche in questo cercheremo di illustrarne tutte le caratteristiche comuni tra i framework, abbracciate da XAML Standard, senza entrare troppo nel dettaglio, ma cercando di esporre quali sono le enormi potenzialità di questo linguaggio.

Come abbiamo già illustrato nel capitolo precedente, tramite XAML dichiariamo classi, impostiamo proprietà e definiamo layout complessi, il tutto sfruttando controlli che rappresentano funzionalità logiche. Rispetto ad altri framework però, XAML va oltre e ci permette di personalizzare tutti i controlli in ogni loro aspetto grafico, attraverso i **template**. L'insieme dei colori, dei pennelli, delle trasformazioni e delle animazioni possono quindi essere racchiusi sotto gli **style** ed essere poi applicati più volte su diversi oggetti. Il **data binding** è un altro incredibile strumento dalle enormi potenzialità, che permette di automatizzare il processo di visualizzazione e

modifica dei dati, permettendo anche scenari master/detail e il pieno supporto alla validazione.

In questo capitolo vedremo tutto questo, partendo da come possiamo definire oggetti e utilizzarli quando ne abbiamo la necessità. Iniziamo quindi dalle risorse.

Definire e riutilizzare le risorse

Uno dei comportamenti più naturali tenuti tanto dagli sviluppatori quanto dai designer, è quello di cercare di organizzare i contenuti di un'applicazione in modo da favorirne la manutenzione e il riutilizzo, dividendo i contenuti secondo le logiche più adeguate.

Nelle app basate su XAML, questa esigenza non viene a mancare, dato che è molto comune riutilizzare oggetti e valori, come colori, pennelli e immagini. Per soddisfare questa esigenza, ogni elemento visuale ha a disposizione una proprietà `Resources` che permette di mantenere un dizionario di chiavi con i loro relativi valori, per poi farne riferimento nel contesto in cui sono state definite.

Poniamo di voler definire un pennello, usato come sfondo in più parti dell'applicazione; per farlo, dobbiamo dichiarare il pennello allo stesso modo di come normalmente facciamo sull'elemento al quale vogliamo applicarlo. Tutto questo è illustrato nell'[esempio 14.1](#).

Esempio 14.1 - XAML

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Page.Resources>
    <SolidColorBrush x:Key="RedBrush">Red</SolidColorBrush>
  </Page.Resources>
</Page>
```

Nell'esempio precedente sfruttiamo la property syntax per popolare la proprietà `Resources`, in questo caso di una pagina, nella quale definire gli oggetti. L'unico requisito richiesto è identificare la risorsa con una chiave

(attraverso l'attributo `x:key`), che è univoca all'interno del contesto in cui è stata definita.

Per contesto intendiamo l'elemento all'interno del quale la risorsa è stata definita e tutto il ramo di figli che gli appartengono. Ciò significa che l'elemento stesso (e quelli discendenti) può utilizzarla, e per farlo dobbiamo sfruttare una delle numerose markup extension presenti in XAML, in modo specifico quella di nome `StaticResource`.

Le markup extension sono una particolare caratteristica del linguaggio XAML e si contraddistinguono dalle parentesi graffe nella valorizzazione degli attributi. Permettono di ampliare il processo di inizializzazione degli elementi attraverso processi personalizzati. Vengono sfruttate per fornire funzionalità quali le risorse, il data binding e l'assegnazione dei valori.

Poniamo quindi di avere un pulsante sul quale impostare la proprietà `Background` con la risorsa `RedBrush`. Utilizziamo la markup extension per farne riferimento in base alla chiave, come mostrato nell'[esempio 14.2](#).

Esempio 14.2 - XAML

```
<Page>
  <Page.Resources>
    <SolidColorBrush x:key="RedBrush">Red</SolidColorBrush>
  </Page.Resources>
  <Button Background="{StaticResource RedBrush}">StaticResource</Button>
</Page>
```

Il risultato dell'[esempio 14.2](#) è visibile nella [Figura 14.1](#), in una Universal Windows Platform App.



Figura 14.1 – Il pulsante con background impostato tramite risorsa.

Poiché la risorsa è definita a livello di finestra, il bottone, che è figlio, può accedere alla risorsa e valorizzare lo sfondo. Questa caratteristica, da un lato comporta una maggiore attenzione nel porre le risorse, ma permette anche scenari più complessi. Per esempio, possiamo definire delle risorse all'interno di un pannello, per il solo fine di renderle disponibili per i propri figli. In caso di omonimie, tra l'altro, viene utilizzata quella più "vicina" a chi ne fa richiesta, dato che la ricerca della risorsa avviene partendo dall'elemento e quindi salendo nell'albero degli elementi, fino ad arrivare all'applicazione. Questo comporta la possibilità di definire risorse a livello globale, attraverso il file `App.xaml` presente in tutte le tipologie di applicazione, agendo sulla proprietà `Resources`. In Expression Blend o in Visual Studio, infatti, per ogni risorsa che andiamo a creare, viene sempre chiesto se bisogna definirla in uno degli elementi padre rispetto a quello che ne ha bisogno, a livello di finestra, oppure a livello di applicazione.

Inoltre, nelle risorse possiamo inserire qualsiasi oggetto managed, dato che XAML non è un linguaggio strettamente legato alla definizione di layout. Possiamo dichiarare numeri che rappresentano dimensioni per certi aspetti dell'applicazione, ma anche liste, pennelli, colori, animazioni, trasformazioni e perfino elementi visuali.

Le risorse di XAML non vanno confuse con quelle degli assembly .NET con estensione .resx. Nel caso degli assembly, le risorse sono file binari inclusi nei file .dll e possono essere letti e processati a seconda della tipologia di file. Vengono sfruttate per includere il binario di immagini e font, mentre le risorse del markup XAML contengono le dichiarazioni degli oggetti da istanziare a runtime.

Bisogna evidenziare che la risoluzione delle risorse avviene in fase d'inizializzazione del markup e cioè, in pratica, nel costruttore della classe rappresentante l'applicazione, la pagina o la finestra, quindi non c'è un decadimento delle prestazioni e, di fatto, questo è equivalente ad aver dichiarato il valore di una proprietà nel classico modo.

Sebbene sia possibile cambiare le risorse di un oggetto da codice, a runtime, in virtù di quanto detto, tali modifiche non si rispecchieranno sugli

oggetti caricati ma solo su quelli che verranno istanziati successivamente.

In WPF e Xamarin Forms possiamo utilizzare la markup extension `DynamicResource` che risolve questo problema, monitorando le risorse e rispecchiando in autonomia i cambiamenti apportati.

Nel caso vogliamo permettere all'utente di selezionare un tema o uno skin da dare all'applicazione, sarà necessario riavviare l'applicazione e cambiare le risorse prima che esse vengano utilizzate, oppure, ricaricare la finestra o la pagina per un approccio meno invadente.

Creare e gestire gli Style

L'utilizzo delle risorse ci permette di centralizzare e organizzare meglio le informazioni, in particolar modo quelle inerenti il layout. Con gli `style` possiamo fare molto di più: racchiudere in un unico nome un insieme di proprietà e comportamenti, da associare poi a un elemento. Rispetto al mondo web, gli `style` sono paragonabili agli **stili CSS** e sono fondamentali per il rendering a video dei controlli. Questi ultimi, infatti, rappresentano funzionalità logiche, ma non definiscono niente nell'aspetto, poiché tutto è affidato allo stile predefinito.

Per capire quindi gli `style`, supponiamo di avere un pulsante al quale vogliamo dare lo sfondo rosso e impostare il colore del testo in bianco.

Invece di impostare tali proprietà sul controllo, definiamo uno stile tra le risorse di nome "RedButton", come mostrato nell'[esempio 14.3](#).

Esempio 14.3 - XAML

```
<Page.Resources>
  <Style x:Key="RedButton"
        TargetType="Button">
    <Setter Property="Background"
            Value="Red" />
    <Setter Property="Foreground"
            Value="White" />
  </Style>
</Page.Resources>
```

Nell'esempio possiamo notare come, prima di tutto, per ogni stile vada specificata la tipologia di elemento a cui è dedicato, in questo caso il `Button`. Seguono poi, mediante oggetti `setter`, le valorizzazioni delle proprietà appartenenti all'oggetto `Button`, mediante la coppia `Property` e `Value`.

Una volta che abbiamo definito lo stile, non ci resta che utilizzarlo come una risorsa, come mostrato nell'[esempio 14.4](#), impostandolo attraverso la proprietà `style`, disponibile per ogni elemento visuale.

Esempio 14.4 - XAML

```
<Button Style="{StaticResource RedButton}">First button</Button>  
<Button Style="{StaticResource RedButton}">Second button</Button>
```

Nella [Figura 14.2](#) possiamo vedere che lo stile viene applicato e il risultato è il medesimo che otterremmo impostando le proprietà sull'elemento stesso.



Figura 14.2 – Due bottoni con uno style applicato.

Sebbene dobbiamo specificare la destinazione di ogni `style` mediante la proprietà `TargetType`, gli stili non sono vincolati a uno specifico di

oggetto. Infatti, possiamo assegnarlo a una classe/elemento base, e ottenere così uno stile applicabile a più oggetti.

Nella dichiarazione dello `style` non è però obbligatorio indicare la chiave della risorsa. Se omessa, la risorsa assume il nome stesso della tipologia. Quello che otteniamo è uno stile implicito per tutti gli elementi di quel tipo. Questo significa che possiamo omettere la valorizzazione della proprietà `style`, vista nell'[esempio 14.4](#), e ottenerne il medesimo risultato.

Le possibilità però non finiscono qui. Gli stili possono ereditare da un altro stile, impostando la proprietà `BasedOn`. Questo ci permette di organizzare gli stili con impostazioni base che poi specializziamo man mano a seconda del controllo, come illustrato nell'[esempio 14.5](#).

Esempio 14.5 - XAML

```
<Style x:Key="BaseStyle" TargetType="Control">
  <Setter Property="Margin"
    Value="4" />
</Style>
<Style x:Key="RedButton"
  BasedOn="{StaticResource BaseStyle}"
  TargetType="Button">
  <Setter Property="Background"
    Value="Red" />
  <Setter Property="Foreground"
    Value="White" />
</Style>
```

Non è fondamentale che lo stile dal quale ereditiamo abbia il medesimo `TargetType`. Infatti, nell'[esempio 14.5](#) lo stile base è applicato a tutti i controlli (view in caso di Xamarin Forms), i quali dispongono di una proprietà `Margin`.

Occorre evidenziare che, anche nelle risorse e in particolare, negli stili, è pienamente utilizzabile tutta la sintassi XAML, quindi la proprietà `value` di un `setter` può contenere oggetti più complessi e, mediante la `property element syntax`, valorizzare trasformazioni, animazioni e quant'altro, così come utilizzare una risorsa che abbiamo definito in precedenza.

[Modellare il layout con i Template](#)

Abbiamo già accennato al fatto che i controlli delle applicazioni basate su XAML rappresentano una funzione logica. Il `Button` ha un evento `Click` ed esegue un comando, la `ListView` contiene una lista di elementi e restituisce l'elemento selezionato, la `CheckBox` e lo `Switch` hanno uno stato di selezionato o non selezionato. Questi sono solo alcuni esempi di controlli nella cui implementazione non c'è alcun riferimento a come devono apparire, né informazioni su come cambiare in funzione dello stato. Queste caratteristiche sono affidate agli stili e ai template, che determinano l'aspetto di ogni controllo e, allo stesso tempo, ci permettono di ridefinirli in base alle nostre necessità.

L'aspetto predefinito di ogni controllo è determinato dal tipo di applicazione (WPF, Universal Windows Platform o Xamarin Forms) e dal tema del sistema operativo (Windows, iOS, Android). In Windows Presentation Foundation, sono supportati gli stili di Windows Classic, Windows XP, Windows Vista e Windows 10, sfruttando le tabelle dei colori del tema corrente. Nelle Universal Windows Platform, invece, il tema è univoco e può essere chiaro o scuro, e possiamo scegliere se sfruttare gli accenti scelti dall'utente a livello di sistema operativo. Nelle applicazioni mobile, infine, i temi sono aderenti al sistema operativo del dispositivo.

I **template** rappresentano la definizione di come un controllo deve essere rappresentato in maniera visuale. Rappresentano un modello, basato a sua volta su elementi primitivi, come rettangoli, bordi e pannelli, e si appoggiano sugli stati per fornire l'interazione con l'utente dell'interfaccia.

Le tipologie di template sono molteplici, a seconda dello scopo che devono soddisfare:

- ❑ `ControlTemplate`: utilizzato per modellare i controlli e il loro aspetto;
- ❑ `DataTemplate`: utilizzato per mostrare i dati all'interno di liste o `ContentControl`;
- ❑ `ItemsPanelTemplate`: utilizzato per modellare il pannello contenitori di controlli che mostrano liste.

Partiamo dai `ControlTemplate`, che sono fondamentali per tutto il motore dell'interfaccia di un'applicazione.

Personalizzare un controllo con il ControlTemplate

Per capire meglio il concetto, proviamo a personalizzare una `CheckBox`, in cui il normale look&feel è quello previsto dal sistema. I template da applicare ai controlli sono di tipo `ControlTemplate` e, normalmente, vengono definiti nelle risorse. Al loro interno non dobbiamo fare altro che inserire gli elementi visuali, per dare alla nostra `CheckBox` l'aspetto che desideriamo.

Nell'[esempio 14.6](#), attraverso un `Grid`, posizioniamo un rettangolo a fianco del contenuto, in modo che questo riempi tutto lo spazio disponibile. Indichiamo inoltre il `TargetType` del template, in modo da specializzare il template e le proprietà a cui facciamo riferimento: la `CheckBox`.

Esempio 14.6 - XAML

```
<ControlTemplate x:Key="PointCheckBox"
                TargetType="CheckBox">
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto" />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>

    <Ellipse Stroke="DarkRed"
             x:Name="square"
             Width="14"
             Height="14" />
    <ContentPresenter Grid.Column="1" />

  </Grid>
</ControlTemplate>
```

Di fondamentale importanza è il `ContentPresenter`, che funge da segnaposto, a indicare che in quel punto va posto il contenuto della `CheckBox`. La proprietà `Content`, infatti, che il controllo espone, può contenere qualsiasi testo o elemento.

Nell'[esempio 14.7](#) non ci resta che definire le checkBox e utilizzare il template quando lo desideriamo, attraverso l'omonima proprietà `Template`.

Esempio 14.7 - XAML

```
<CheckBox IsChecked="True"
          Content="IsChecked a true" />
<CheckBox Template="{StaticResource PointCheckBox}"
          IsChecked="True"
          Content="IsChecked a true" />
<CheckBox Template="{StaticResource PointCheckBox}"
          Content="IsChecked a false" />
```

C'è ancora un'ultima cosa da fare: abbiamo detto, infatti, che la `CheckBox` implementa la logica, perciò espone una proprietà `IsChecked` che indica se l'utente ha premuto il pulsante sull'elemento. Tale proprietà determina anche lo stato visuale del controllo, e attraverso quest'ultimo possiamo personalizzare l'aspetto del controllo stesso a seconda delle situazioni previste. Nel caso della `CheckBox`, sono previsti due stati di nome `Checked` e `Unchecked`, che possiamo definire nel `Template` e utilizzare per animare gli elementi,

Per esempio, possiamo riempire di colore rosso il rettangolo che abbiamo definito nell'[esempio 14.6](#). Modifichiamo quindi il `ControlTemplate` aggiungendo un oggetto `VisualStateManager`, come nell'[esempio 14.8](#).

Esempio 14.8 - XAML

```
<ControlTemplate x:Key="PointCheckBox"
                 TargetType="CheckBox">
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto" />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Ellipse Stroke="DarkRed"
             x:Name="square"
             Width="14"
             Height="14"
             Margin="2"
             VerticalAlignment="Center" />
    <ContentPresenter VerticalAlignment="Center"
```

```

        Grid.Column="1" />
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup x:Name="CheckStates">
    <VisualState x:Name="Checked">
      <Storyboard>
        <ObjectAnimationUsingKeyFrames Storyboard.TargetName="square"
          Storyboard.TargetProperty="Fill">
          <DiscreteObjectKeyFrame KeyTime="0:0:0">
            <DiscreteObjectKeyFrame.Value>
              <SolidColorBrush Color="Red" />
            </DiscreteObjectKeyFrame.Value>
          </DiscreteObjectKeyFrame>
        </ObjectAnimationUsingKeyFrames>
      </Storyboard>
    </VisualState>
    <VisualState x:Name="Unchecked">
      </VisualState>
    </VisualStateGroup>
  </VisualStateManager.VisualStateGroups>
</Grid>
</ControlTemplate>

```

Come possiamo vedere nell'esempio, la collezione `visualStateGroups` ci permette di definire uno o più `visualStateGroup` il quale rappresenta un gruppo di stati visuali. Ogni stato, identificato dall'oggetto `visualState`, ci permette di definire una `storyboard`, quindi una serie di animazioni da eseguire quando lo stato visuale cambia. Quindi, grazie agli strumenti di animazione, nell'[esempio 14.8](#) cambiamo la proprietà `Fill` del rettangolo di nome `square`.

Nella [Figura 14.3](#) possiamo vedere il risultato a partire dai controlli definiti nell'[esempio 14.7](#): il primo con lo stile predefinito, il secondo e il terzo con quello personalizzato.

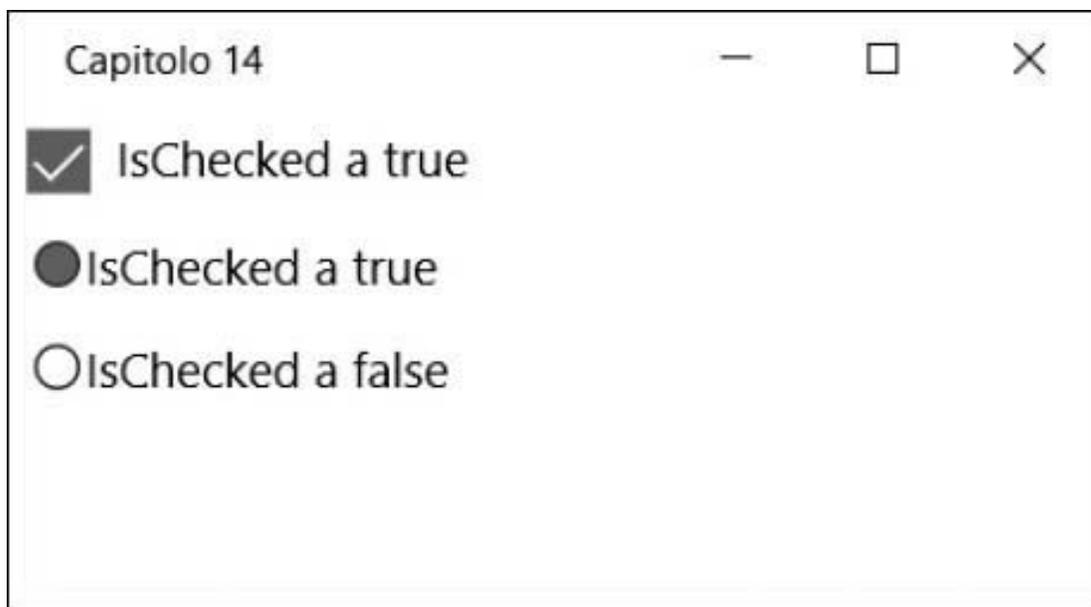


Figura 14.3 – Le CheckBox con il template personalizzato in un'applicazione WPF.

L'esempio della `checkbox` è semplificato, ma il meccanismo descritto sta alla base del funzionamento di tutti i controlli di ogni tipologia di applicazione.

Ogni `VisualStateGroup`, infatti, rappresenta un insieme logico di stati che un controllo può avere, e in genere disponiamo di più gruppi per poter impostare più animazioni contemporaneamente. La maggior parte dei controlli predispone i gruppi `CommonStates` e `FocusStates`, i quali possono avere rispettivamente gli stati `Normal`, `PointerOver` (o `MouseOver` se su WPF) e `Pressed` per il primo, mentre per il secondo `Focus` e `Unfocused`. Ogni gruppo può essere in uno di questi stati, perciò possiamo specificare contemporaneamente animazioni per il passaggio del mouse e animazioni per il fatto che il controllo abbia o no il focus da tastiera. Vi sono poi gli stati specifici di un controllo, che possiamo personalizzare consultando la documentazione ufficiale, oppure affidandoci a Visual Studio o a Expression Blend per ottenere un aiuto da parte del designer, che ci suggerisce gli stati che possiamo personalizzare, come viene mostrato nella [Figura 14.4](#).

Oltre ai `VisualState` definiti dai framework, possiamo anche sfruttare stati personalizzati definibili su qualsiasi elemento, anche al di fuori del template. Sono molto utili, per esempio, a livello di pagina, perché ci

permettono di variare elementi all'interno di essa, cambiando lo stato corrente. Nelle Universal Windows Platform app, inoltre, disponiamo di uno speciale trigger, di nome `AdaptiveTrigger`, che è in grado di innescare lo stato quando viene soddisfatta la condizione impostata in funzione della dimensione della pagina. Lo scopo è quello di poter adattare i contenuti all'interno della pagina a seconda che l'applicazione venga eseguita su mobile, tablet o desktop. Nell'[esempio 14.9](#) possiamo vedere all'opera questo speciale trigger. Con esso possiamo indicare la larghezza minima della pagina (`MinWindowWidth`) e l'altezza minima della pagina (`MinWindowHeight`) affinché lo stato venga cambiato.

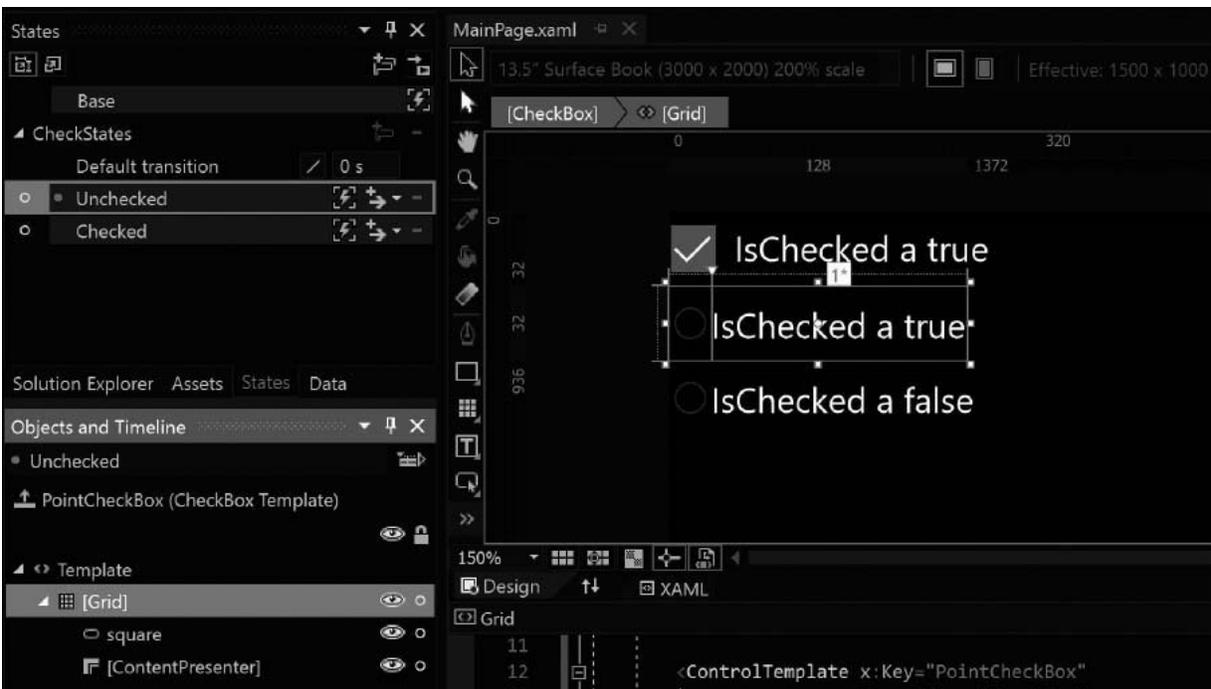


Figura 14.4 – Expression Blend e il supporto ai Visual State del designer.

Esempio 14.9 - XAML

```
<Grid>
  <VisualStateManager.VisualStateGroups>
    <VisualStateGroup>
      <VisualState>
        <VisualState.StateTriggers>
          <AdaptiveTrigger MinWindowWidth="700" />
        </VisualState.StateTriggers>
```

```

        <VisualState.Setters>
            <Setter Target="mainPanel.Orientation" Value="Horizontal" />
        </VisualState.Setters>
    </VisualState>
</VisualStateManager.VisualStateGroups>

<StackPanel x:Name="mainPanel" Orientation="Vertical">
    <TextBlock>Primo elemento</TextBlock>
    <TextBlock>Secondo elemento</TextBlock>
</StackPanel>
</Grid>

```

Oltre a specificare il trigger, lo stato ci consente di usare la storyboard, come già fatto nell'[esempio 14.8](#), ma anche uno o più Setter, oggetti che in modo più immediato imposta una proprietà di un elemento (`nomeElemento.Proprieta`), ma senza transizione. Nell'[esempio 14.9](#) andiamo a variare l'orientamento del pannello qualora la larghezza della pagina superi i 700 pixel effettivi (pixel dello schermo rapportati ai DPI).

Ritornando alla funzionalità che può sfruttare i VisualState, i template, dobbiamo aggiungere che sono importanti anche per il caricamento dei dati. Infatti, possiamo sfruttarli per preparare `DataTemplate`, modelli che hanno il compito di caricare il layout per uno specifico oggetto, presente, per esempio, all'interno di una lista.

[Il data binding](#)

Nella maggior parte delle applicazioni, lo scopo principale è mostrare dati provenienti da database, servizi, oppure dal web, per permetterne la consultazione, la modifica, la cancellazione o l'inserimento. Poiché questa esigenza è molto frequente e presenta spesso le medesime dinamiche, in XAML abbiamo un meccanismo che ci facilita queste operazioni: il data binding.

Con questo termine indichiamo il legame che creiamo tra controlli e sorgente dati, in modo da riflettere le modifiche apportate al controllo sulla sorgente, e viceversa. Questo meccanismo è complesso ma, allo stesso tempo risulta semplice e molto potente, specie se lo confrontiamo con meccanismi simili di altri framework. Iniziamo a esplorarlo.

Mostrare le informazioni con il data binding

Per comprendere le potenzialità del data binding, supponiamo di avere una struttura di informazioni fatta di categorie e relativi prodotti. Questi dati sono definiti manualmente, come nell'[esempio 14.10](#), con due categorie e relativi prodotti, ma possono provenire da qualsiasi altro tipo di fonte, come il web o un database.

Esempio 14.10

```
Category[] list = {
    new Category {
        Products = new Product[] {
            new Product {Description = "Cat 1 - Prodotto 1", Id = 1},
            new Product {Description = "Cat 2 - Prodotto 2", Id = 2}
        },
        Description = "Categoria 1",
        Id = 1
    },
    new Category {
        Products = new Product[] {
            new Product {Description = "Cat 2 - Prodotto 3", Id = 1},
            new Product {Description = "Cat 2 - Prodotto 4", Id = 2}
        },
        Description = "Categoria 2",
        Id = 2
    }
};
```

Possiamo decidere di caricare queste informazioni scegliendo, tra i controlli disponibili, quello che più rispecchia le funzionalità a noi necessarie poiché, indipendentemente da quello scelto, il modo di procedere è autonomo.

Esistono molti modi per caricare i dati, ma quello più usato cerca di mantenere il più possibile la separazione dell'interfaccia dal codice, evitando che quest'ultimo faccia riferimenti diretti a elementi. L'oggetto "list", definito nell'[esempio 14.10](#), infatti, può essere associato alla proprietà DataContext (BindingContext nel caso di Xamarin Forms) della finestra o della pagina in fase di caricamento iniziale, come mostrato nell'[esempio 14.11](#).

Esempio 14.11

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
    this.DataContext = list;
}
```

Con questa istruzione, impostiamo la lista di categorie come contesto dati dell'elemento radice. L'elemento stesso e quelli figli, da questo momento, possono accedere a tale oggetto e leggerne le proprietà. Controlli come `ListView`, `GridView`, per citarne alcuni, dispongono di una proprietà `ItemsSource`, che può essere valorizzata anche da codice oppure mediante il data binding. La markup extension `{Binding}` serve proprio a questo: legge dal contesto dati attuale. Nell'[esempio 14.12](#), carichiamo con tale sintassi la lista di categorie.

Esempio 14.12 - XAML

```
<ListView ItemsSource="{Binding}"
           x:Name="categories">
</ListView>
```

Nell'esempio precedente non viene passata una specifica informazione aggiuntiva, a indicare che il motore di data binding deve caricare l'oggetto stesso, cioè la lista di categorie. Eseguendo il codice, otteniamo la lista delle categorie, come è visibile nella [Figura 14.5](#). Da subito possiamo notare che non abbiamo dovuto preoccuparci dei tempi e della creazione degli elementi visuali figli, necessari a popolare la `ListView`.



Figura 14.5 – ListView con data binding applicato.

Possiamo anche notare che l'aspetto non è quello che ci attendiamo, dato che otteniamo il nome completo del tipo dell'oggetto `Category`. Per ottenere l'aspetto visivo desiderato possiamo sfruttare i `DataTemplate`.

Tramite questi ultimi, in modo simile al `ControlTemplate`, possiamo definire l'aspetto da dare a ogni elemento della lista e visualizzare le informazioni che vogliamo. Ampliamo quindi il markup dell'[esempio 14.12](#) e definiamo un `DataTemplate` tra le risorse. In esso possiamo mettere qualsiasi elemento che desideriamo, tra cui `shape`, immagini o, addirittura, video. Nell'[esempio 14.13](#) ci limitiamo a porre un rettangolo a fianco della descrizione della categoria.

Esempio 14.13 - XAML

```
<DataTemplate x:Key="CategoryTemplate">
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto" />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Rectangle Fill="Red"
      Width="10"
      Height="10" />
    <TextBlock Text="{Binding Path=Description}" Grid.Column="1" />
  </Grid>
</DataTemplate>
```

Il termine `DataTemplate` sta a indicare che il template ha lo scopo di servire la visualizzazione di dati. Sfruttando lo stesso meccanismo del `DataContext`, visto nell'[esempio 14.11](#), troviamo all'interno del template il contesto dati: questo è il singolo elemento della lista e, nel caso dell'[esempio 14.13](#), è l'oggetto `Category`. In questo modo, possiamo effettuare il data binding facendo riferimento alla proprietà `Description` attraverso l'attributo `Path` dell'espressione. Definito poi il template, non rimane altro che utilizzarlo, referenziandolo attraverso la proprietà `ItemTemplate` della `ListView`.

Esempio 14.14 - XAML

```
<ListView ItemsSource="{Binding Path=."
    x:Name="categories"
    ItemTemplate="{StaticResource CategoryTemplate}">

</ListView>
```

Rispetto ad altri approcci, quello offerto da XAML presenta innumerevoli vantaggi, dato che non ci sono limiti all'interfaccia che possiamo realizzare, rendendola dinamica, a seconda del contesto dei dati. Inoltre, il meccanismo di data binding è molto versatile e consente anche scenari più complessi, dove i dati e le liste sono relazionati tra loro.

Scenari master/detail con il data binding

Il meccanismo di data binding copre la maggior parte delle esigenze che possiamo incontrare. Tra queste la necessità di supportare scenari in cui la selezione su una prima lista, determina il caricamento di un dettaglio o di una seconda lista, a cascata.

Poniamo ora una seconda `ListView` che deve caricare i prodotti della categoria selezionata sulla prima `ListView`. La proprietà `Path` può contenere, separando dal punto, la navigazione sotto proprietà, perciò attraverso `SelectedItem` possiamo ottenere la `Category` selezionata, e al suo interno leggere le proprietà che vogliamo. Nel caso dell'[esempio 14.15](#), carichiamo la lista dei prodotti nella seconda `ListBox`.

Esempio 14.15 - XAML

```
<ListView ItemsSource="{Binding Path=SelectedItem.Products,  
    ElementName=categories}">  
</ListView>
```

Bastano poche righe di markup per ottenere due liste collegate tra loro: nel selezionare la categoria, la lista dei prodotti cambierà automaticamente. Nell'esempio possiamo notare che dobbiamo inoltre specificare su quale oggetto interrogare la proprietà `SelectedItem`: in questo caso la prima `ListView` di nome `categories` (vedi [esempio 14.12](#)). Nella [Figura 14.6](#) possiamo vedere sia il `DataTemplate` applicato alla prima lista, sia la lista dei prodotti funzionale alla categoria selezionata.

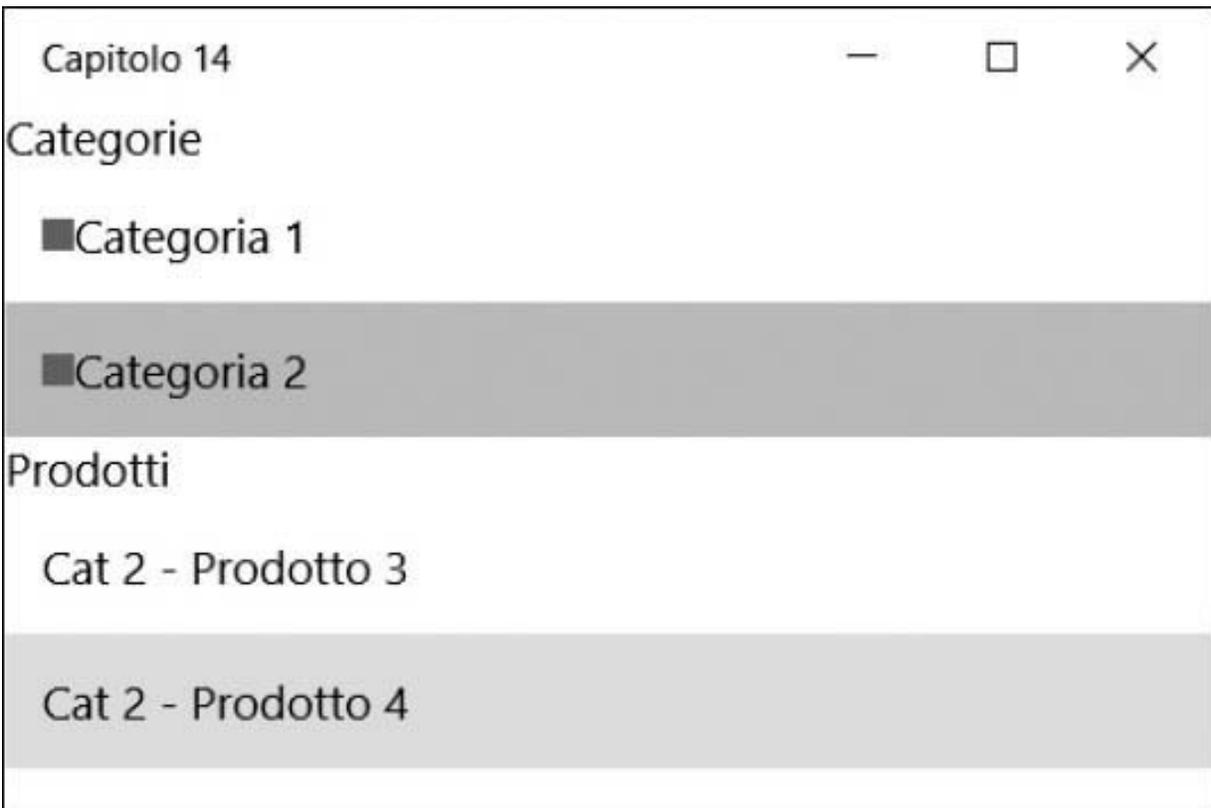


Figura 14.6 – Due `ListView` collegate mediante data binding.

Il meccanismo di data binding qui descritto non ha limiti per quanto riguarda il numero di liste annidate che possiamo caricare, e in questo modo

ci permette di soddisfare ogni nostra esigenza. Per esempio, nel `DataTemplate` delle categorie potremmo caricare direttamente la lista dei prodotti, mostrandoli come un unico elemento di selezione, insieme alla categoria. Le fonti dati, come abbiamo potuto vedere, non si limitano al `DataContext` impostato da codice ma possono coprire ogni esigenza. Nella prossima sezione daremo uno sguardo alle possibilità in tal senso.

[Le fonti dati per il data binding](#)

Le espressioni di data binding affrontate nei primi esempi fanno un uso implicito del contesto dati corrente. Nella finestra principale è il `DataContext` di uno degli elementi mentre, nel `DataTemplate` è il contesto specifico di ogni riga, come nel caso dell'[esempio 14.13](#).

In alternativa, però, possiamo sfruttare, come già anticipato, la proprietà `ElementName` dell'oggetto `Binding` per interrogare, sempre mediante la proprietà `Path`, le proprietà di un elemento.

Con Xamarin Forms possiamo ottenere lo stesso risultato con la markup extension `x:Reference`, che ci permette di ottenere il riferimento a un altro elemento dell'interfaccia e associarlo alla proprietà `Source` del `Binding`.

Se abbiamo una casella di testo e un'etichetta in cui riportare il testo immesso dall'utente, ci basta dare un nome al primo controllo e referenziarlo nell'espressione di data binding, come mostrato nell'[esempio 14.16](#).

Esempio 14.16 - XAML

```
<TextBox x:Name="myText" />
<TextBlock Text="{Binding Path=Text, ElementName=myText}" />
```

La markup extension `Binding`, così come tutte le altre di XAML, prevede di separare ogni proprietà con una virgola e di valorizzarla con un'assegnazione. Nell'esempio, indichiamo di caricare la proprietà `Text`

dell'oggetto `TextBox` di nome `myText`. Inoltre, dato che tale proprietà può variare (nello specifico è una `DependencyProperty`), il motore la aggancia e fa sì che ogni modifica che effettuiamo alla casella di testo, si rifletta automaticamente sull'etichetta sottostante, come visibile nella [Figura 14.7](#).

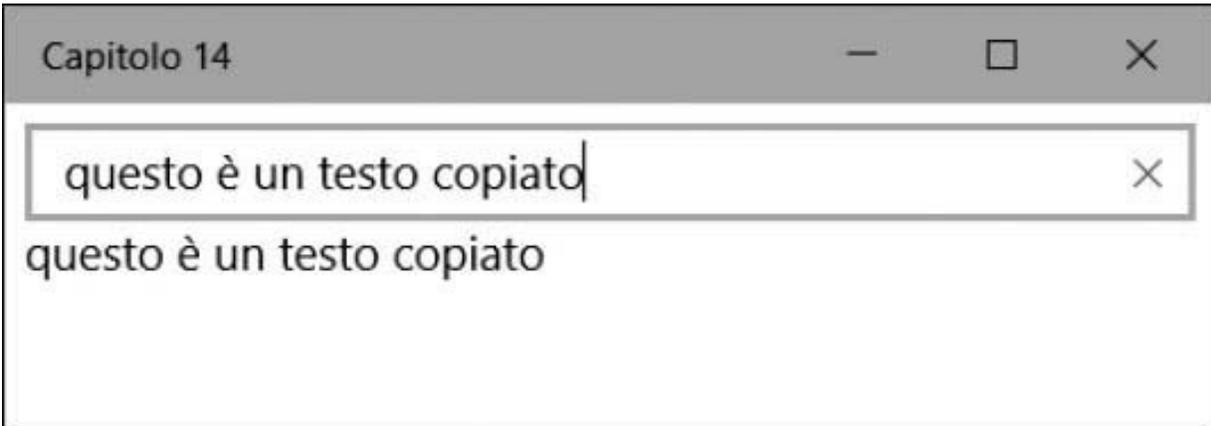


Figura 14.7 – Il data binding mediante `ElementName` in azione.

La proprietà `ElementName` è specifica per sorgenti di tipo elemento ma, come abbiamo visto nel paragrafo precedente, non vi sono limiti sul tipo di sorgente dati.

Abbiamo visto, nella sezione dedicata alle risorse, come in esse possiamo dichiarare qualsiasi oggetto; per questo motivo istanziamo la classe `Product` come abbiamo già fatto nell'[esempio 14.10](#). Questa volta però lo facciamo da markup, come mostrato nell'[esempio 14.17](#).

Esempio 14.17 - XAML

```
<Page xmlns:l="using:ASPIItalia.Books.Chapter14">
  <Page.Resources>
    <l:Product x:Key="myProduct"
              Id="1"
              Description="Prodotto 1" />
  </Page.Resources>
```

Non c'è alcuna differenza rispetto all'istanziare questa classe da codice, con l'eccezione del fatto che l'istanza viene mantenuta all'interno delle risorse.

Possiamo quindi caricare tale oggetto mediante la proprietà `source` e la markup extension `StaticResource`: l'[esempio 14.18](#) mostra come fare.

Esempio 14.18 - XAML

```
<TextBlock Text="{Binding Path=Description,  
                Source={StaticResource myProduct}}" />
```

Nell'esempio notiamo, prima di tutto, l'utilizzo di una markup extension all'interno di un'altra, che rende comunque leggibile tutta l'espressione. Dato che la sorgente dati è il prodotto, non ci resta che indicare di leggere la descrizione attraverso la proprietà `Path`.

La formattazione dei dati

Finora le espressioni di data binding che abbiamo analizzato si sono limitate a mostrare semplicemente il valore della proprietà di un oggetto. Sempre nell'ottica di limitare l'uso di codice per personalizzare l'interfaccia, il motore di data binding offre la possibilità di formattare i valori.

Il metodo, comune a tutti i framework basati su XAML, consiste nello sfruttare la proprietà di nome `converter`, che ci permette di specificare un convertitore di valori: una classe che implementa l'interfaccia `IValueConverter`.

Creare un converter è piuttosto semplice, dato che i membri richiesti sono due:

- ❑ `Convert`: richiamato quando il motore vuole prendere il valore originale e convertirlo nel valore finale;
- ❑ `ConvertBack`: richiamato quando il motore deve riversare i cambiamenti effettuati sul destinatario di un'espressione di binding sulla sorgente dati.

Supponiamo di voler cambiare il colore della data in funzione del fatto che il giorno attuale sia pari o dispari. Prima di tutto creiamo il nostro convertitore, come nell'[esempio 14.19](#).

Esempio 14.19

```
public class DateToColorConverter : IValueConverter
{
    public static readonly DateToColorConverter Instance =
        new DateToColorConverter();

    public object Convert(object value,
        Type targetType,
        object parameter,
        string language)
    {
        System.DateTime d = (System.DateTime)value;
        if (d.Day % 2 == 0)
            return new SolidColorBrush(Colors.Green);
        else
            return new SolidColorBrush(Colors.Red);
    }

    public object ConvertBack(object value,
        Type targetType,
        object parameter,
        string language)
    {
        throw new NotImplementedException();
    }
}
```

L'implementazione del convertitore non fa altro che leggere il parametro `value`, che diamo per scontato sia una data e, in base al giorno del mese, restituisce un pennello verde piuttosto che rosso.

Xamarin Form e WPF hanno la stessa interfaccia, prevedono la ricezione della lingua come tipo `CultureInfo` e differiscono solo in questo, mantenendo intatto il concetto.

Tornando al markup, non ci resta quindi che valorizzare la proprietà `Foreground` del testo, sempre con l'espressione `Binding`, indicando come `Path` la data attuale mentre, come convertitore, quello creato nell'[esempio 14.20](#).

Esempio 14.20 - XAML

```
<Page.Resources>
    <l:DateToColorConverter x:Key="converter" />
</Page.Resources>
```

```
<TextBlock Text="{Binding Path=Data}"
    Foreground="{Binding Path=Data, Converter={StaticResource
    converter}}" />
```

Il motore leggerà allora la data, la passerà al convertitore e quest'ultimo restituirà un Brush, che verrà poi assegnato alla proprietà Foreground del testo. Da notare come, anche per far riferimento al `DateToColorConverter`, dobbiamo prima di tutto istanziarlo nelle risorse.

Nella [Figura 14.8](#), poiché il 22 è un numero pari, l'etichetta sarà di colore verde.

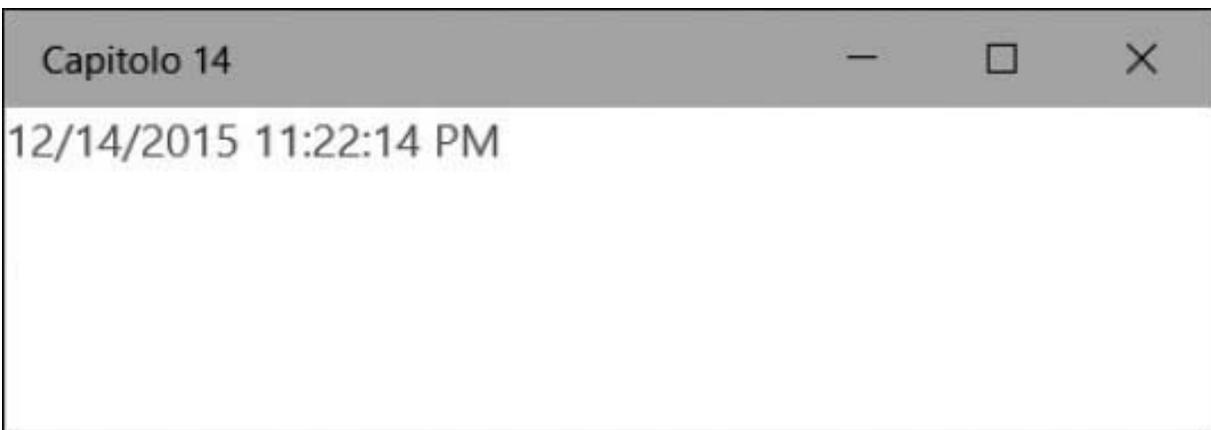


Figura 14.8 – Converter applicato al Foreground di un TextBlock.

Le possibilità offerte dal data binding non finiscono qui. Possiamo infatti gestire anche come il motore lega la sorgente al controllo che mostra le informazioni.

Le modalità di data binding

Finora abbiamo utilizzato il motore di data binding per leggere i valori e mostrarli a video. Abbiamo già visto, però, come nell'esempio 13.20 il motore sia in grado di ripetere l'operazione di caricamento dei dati ogni volta che la sorgente cambia. Questo è possibile grazie alla proprietà `Mode`, il cui valore è automatico e dipende dalla sorgente dati, ma può assumere i seguenti valori:

- ❑ **OneWay**: i valori della sorgente dati vengono riversati sul destinatario e questa operazione viene ripetuta ogni volta che la sorgente cambia;
- ❑ **OneTime**: i valori della sorgente vengono riversati una sola volta, ignorando poi i successivi cambiamenti;
- ❑ **TwoWay**: i valori della sorgente e del destinatario vengono sincronizzati, perciò ogni modifica apportata a una delle parti si riflette sull'altra.

Di particolare importanza è la modalità **TwoWay** che, di fatto, ci aiuta nel preparare maschere di modifica dei dati. Mediante controlli come **TextBox**, **CheckBox** o **Slider**, possiamo infatti permettere la modifica dei dati e, automaticamente, riversarli sulla sorgente dati, senza dover ricorrere all'uso di codice.

Nell'[esempio 14.21](#) utilizziamo tale proprietà per forzare il data binding a eseguire una volta sola l'operazione di caricamento.

Esempio 14.21 - XAML

```
<TextBlock Text="{Binding Path=Text,  
                Mode=OneTime  
                ElementName=myText}" />
```

Le potenzialità del data binding sono dunque innumerevoli. Purtroppo alcune sono specifiche di WPF o di Xamarin Forms e non trovano posto all'interno di questa guida. Tra queste rientra la possibilità di effettuare data binding multipli, di dare una priorità alle espressioni, di interagire con le interfacce di validazione presenti in .NET e fornire feedback visivi in caso di errore.

Comunque, all'interno del portale web della community WinFXItalia, troviamo articoli e script di maggiore approfondimento.

[Gestire gli eventi](#)

Gli eventi sono una caratteristica che permette di reagire, mediante codice, al verificarsi dei cambi di stato. Abbiamo già intuito che in XAML, diversamente che in altri ambiti, gli eventi spesso non sono necessari, perché le logiche dei controlli, il data binding e i visualState sono sufficienti a coprire la maggior parte delle esigenze.

In ogni caso, tutti gli elementi visuali hanno un set di eventi base per ogni aspetto legato all'input da parte dell'utente. Su ogni `UIElement` troviamo per questo eventi come `KeyDown` o `KeyUp`, per la gestione della tastiera, `Tapped` o `DoubleTapped` per intercettare il click con mouse o la pressione da touch. Questi sono solo alcuni esempi dato che, in realtà, gli eventi sono molteplici e in grado di farci intercettare ogni dinamica di input. A questi dobbiamo poi aggiungere quelli specifici dei controlli: per esempio, un pulsante ha l'evento `Click`, mentre una `ListView` ha l'evento `SelectionChanged` o `ItemTapped`., con Xamarin Forms.

Il modo più semplice per intercettare uno di questi eventi, è dichiarare l'evento come attributo sull'elemento e specificare il nome del metodo da chiamare. Nell'[esempio 14.22](#), possiamo vedere come intercettare l'evento `Loaded` sulla pagina.

Esempio 14.22 - XAML

```
<Page x:Class="Events"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      Loaded="Page_Loaded">
```

La funzione `Page_Loaded` è poi definita nel code-behind, per essere richiamata non appena la pagina ha completato l'operazione di caricamento.

Esempio 14.23

```
private void Page_Loaded(Object sender, RoutedEventArgs e)
{
}
```

Gli eventi non sono però tutti dello stesso tipo; quello `Loaded`, visto nell'[esempio 14.22](#), è di tipo diretto (perciò viene invocato solo

sull'elemento sui cui è stato scatenato) ma ne esistono anche di tipo bubble, cioè che si propagano dall'elemento che li ha generati e risalgono tutto l'albero degli elementi, fino ad arrivare a quello radice. Si prestano a soddisfare eventi come `Tapped`, per essere intercettati abbracciando più elementi contemporaneamente.

Nella gestione degli eventi, se WPF e UWP trovano molta uniformità, nel caso di Xamarin Forms ci sono molte differenze in ciò che possiamo intercettare e nei nomi, essendo più orientato al mondo mobile, e non è presente la propagazione degli eventi.

Con il markup dell'[esempio 14.24](#), dove abbiamo una `Grid` e un `Rectangle`, intercettiamo a livello di pagina e di rettangolo l'evento di pressione, di tipo bubble.

Esempio 14.24 - XAML

```
<Window x:Class="Events"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  MouseUp="Window_MouseUp">
  <Grid Background="LightGray">
    <Rectangle Width="100"
      Height="100"
      MouseUp="Rectangle_MouseUp"
      Fill="Red" />
  </Grid>
</Window>
```

Nel code-behind definiamo quindi i due eventi. Per provare le funzionalità di bubbling, inibiamo i click effettuati sul rettangolo e notificiamo con il nome dell'oggetto che ha generato l'evento.

Esempio 14.25

```
void Page_Tapped(object sender, TappedRoutedEventArgs e)
{
  var md = new MessageDialog("Pulsante premuto su " +
    e.OriginalSource.GetType().Name);
  md.ShowAsync();
}

void Rectangle_Tapped(object sender, TappedRoutedEventArgs e)
```

```
{
    e.Handled = true;
}
```

Eseguendo il codice, notiamo che l'evento `Tapped` della pagina viene sollevato a prescindere dall'area in cui clicchiamo, anche se non abbiamo intercettato direttamente l'evento sulla griglia o sul rettangolo. Grazie alla proprietà `Handled`, inoltre, quando cliccheremo sul rettangolo inibiremo la propagazione dell'evento su quello a livello di finestra. Come vediamo dalla [Figura 14.9](#), di fatto, solo cliccando sulla `Grid` otterremo il messaggio, mentre cliccando sul `Rectangle` non otterremo nulla.

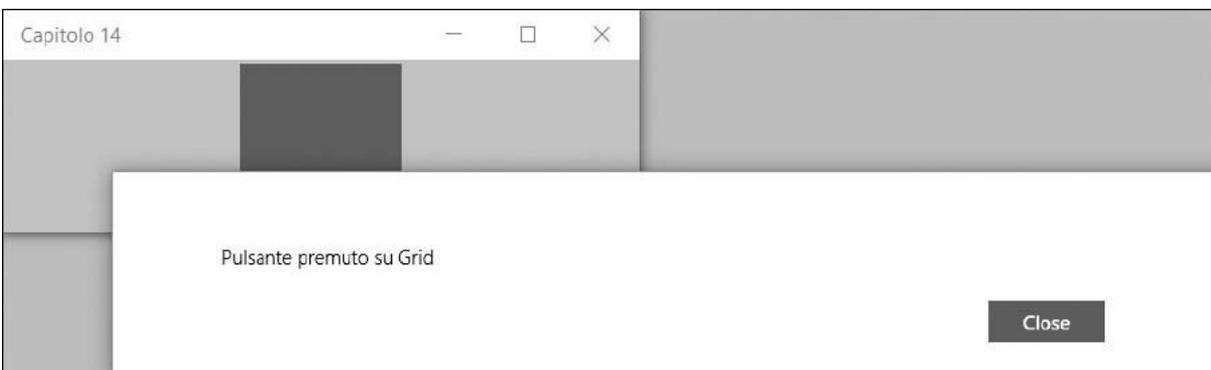


Figura 14.9 – La propagazione degli eventi intercettata e mostrata attraverso una `MessageBox` di UWP.

Il meccanismo descritto regola tutta la gestione degli input all'interno di un'applicazione, per cui la sua comprensione è molto importante. È grazie a questa caratteristica che controlli come `Button` possono esporre l'evento `Click`, indipendentemente dal loro template. In questo modo, tutti gli eventi `Tapped` generati dagli elementi che danno l'aspetto al pulsante, vengono intercettati in modo da fornire un unico evento che rappresenta la funzionalità logica del pulsante: il click.

Conclusioni

XAML è un linguaggio che rivoluziona il modo di creare applicazioni per desktop, tablet o mobile. Introduce molti concetti nuovi ed estende il lavoro

di realizzazione di un'applicazione, comprendendo anche una nuova figura professionale: quella del designer.

In questo capitolo abbiamo dimostrato come, attraverso stili e template, possiamo cambiare il modo di concepire i controlli e quindi l'interfaccia grafica. Possiamo modificare completamente il loro aspetto, condividere informazioni tra più elementi e, con i `visualState`, fornire un'esperienza più ricca, senza l'ausilio di codice.

Abbiamo quindi illustrato come il motore di data binding sia capace di soddisfare moltissime esigenze – gestisce la conversione dei dati, la formattazione, scenari master/details in modo completamente automatizzato – il tutto utilizzando un linguaggio dichiarativo.

Inoltre, abbiamo analizzato il sistema di eventi che sta alla base degli elementi e permette il funzionamento dei controlli.

Due capitoli non sono sufficienti per comprendere appieno tutto quello che sta dietro a XAML, ma possono bastare per aiutarvi a capirne le potenzialità. A questo punto non ci resta che applicare quanto appreso su questo linguaggio nel mondo delle Universal Windows Platform app.

Usare XAML

Nei due capitoli precedenti abbiamo visto come funziona XAML, che è alla base delle recenti tecnologie di sviluppo Microsoft per i sistemi operativi client. L'abbiamo utilizzato in diversi contesti, senza però entrare nel merito dei runtime e degli ambiti in cui lo possiamo utilizzare.

Con XAML, infatti, possiamo sviluppare applicazioni per il Windows Store tramite **Universal Windows Platform app** (UWP) e per **Windows Presentation Foundation** (WPF).

Ciascuna di queste implementazioni offre caratteristiche differenti, che rendono XAML in grado di sfruttare il massimo vantaggio e offrire il meglio allo sviluppatore.

Storicamente WPF è nato nel lontano 2006 e si contrappone alle WinForms: un wrapper .NET che utilizza Win32 del sistema operativo per realizzare interfacce sfruttando le sue primitive. Con WPF, invece, la finestra Win32 è un contenitore di un'interfaccia realizzata interamente con primitive DirectX e grazie a XAML ne permette la definizione di layout, controlli e animazioni. Questa tecnologia ha portato un importante cambiamento dal punto vista dell'impatto dell'interfaccia, tanto che con Windows 8, e poi con il perfezionamento in Windows 10, Microsoft ha deciso di adottare XAML per introdurre un nuovo modo di fare app che garantisca l'isolamento delle stesse e facilitasse la loro distribuzione tramite lo store. Quest'ultima tecnologia, inoltre, cerca di spostare e facilitare la creazione di app dal desktop verso i dispositivi mobile, grazie alla presenza di controlli pensati per questo scopo. Data la presenza massiva di applicazioni sviluppate in WPF e la scarsa adozione del nuovo paradigma, Microsoft ha deciso di permettere la realizzazione di app in WPF anche con

.NET Core 3.0, finora consentito solo tramite .NET Framework. Lo scopo di questa scelta è di permetterci di sfruttare le nuove caratteristiche del runtime e di C# anche per le app già in essere.

In questo capitolo vi presentiamo una rapida panoramica di queste implementazioni, sviluppando alcuni esempi per comprendere le differenze e capire quali si adattano ai nostri scopi.

Universal Windows Platform

Universal Windows Platform supporta piattaforme differenti dalle classiche Intel x86/x64, poiché gira anche su ARM (i cosiddetti SOC, System On a Chip), consentendo di creare applicazioni che possono girare sui tradizionali sistemi desktop, sui portatili, oltre che sui tablet, sia puri, sia ibridi. Per questo motivo, Microsoft ha introdotto un nuovo run time, denominato Windows Runtime o, più semplicemente, WinRT.

WinRT offre una serie di funzionalità comuni alle app, come i servizi di accesso alla UI, alle risorse hardware, allo storage o al networking, come tradizionalmente è sempre avvenuto in Windows. Lo fa mettendo a disposizione degli sviluppatori una serie di funzionalità direttamente sopra al Windows Core, cioè al cuore di Windows.

Inoltre, WinRT gestisce il sandboxing delle applicazioni, facendo in modo che ogni singola applicazione sia limitata ad accedere solo a un'area specifica di memoria e disco, migliorando la stabilità complessiva e la sicurezza del sistema, grazie all'uso di un run time broker che controlla, per esempio, che solo le API dichiarate dal creatore dell'app nel manifest (una sorta di carta d'identità che lo sviluppatore deve allegare a ogni app) vengano effettivamente invocate.

Per uno sviluppatore abituato a .NET, come vedremo, si tratta solo di utilizzare quello che già conosce, ma in una chiave differente, con l'aggiunta di alcune novità, come le XAML Island che consentono di portare la potenza del runtime addirittura su applicazioni Windows Forms.

Le applicazioni moderne sono utilizzabili sia con la tastiera sia con il mouse come da tradizione, ma anche attraverso il touch. In questo capitolo cercheremo di inquadrare meglio i tool e le metodologie necessarie a costruire applicazioni per Windows.

In realtà, WinRT è basato su un'anima COM, lo stesso COM che probabilmente abbiamo utilizzato in qualche applicazione in passato (e imparato ad amare e odiare): tuttavia, i problemi maggiori di COM, come la gestione delle reference e il versioning, sono assorbiti in toto da WinRT, con il risultato che non dovremo mai subirne le conseguenze. I componenti, per esempio, non sono più di sistema, ma locali a ogni applicazione. Questo è un concetto già noto a chi ha dimestichezza, per esempio, con il meccanismo di gestione delle reference locali, tipico di .NET.

In realtà, siamo di fronte a un'evoluzione di COM che prende molto dal .NET Framework, con un sistema di namespace, un modello a oggetti ben ingegnerizzato, un type system espandibile.

Grazie a questa impostazione, WinRT supporta essenzialmente tre modelli differenti di sviluppo:

- ❑ XAML, C# .NET;.
- ❑ HTML e JavaScript, sfruttando Chakra, l'engine di Internet Explorer 10.
- ❑ XAML e C++, per applicazioni native.

Ciascuna di queste opzioni è identica, in termini di funzionalità accessibili allo sviluppatore, a una qualsiasi delle altre: questo meccanismo risulta possibile perché le funzionalità implementate attraverso le API di WinRT sono rese accessibili agli sviluppatori attraverso un ponte, che le adatta in base alle differenti necessità dei differenti linguaggi: questi ponti prendono il nome di projection.

Le projection sono il sistema attraverso il quale uno dei linguaggi supportati proietta se stesso in WinRT. O, se vogliamo, il sistema attraverso il quale le API di WinRT sono proiettate per supportare il linguaggio/tecnologia nelle sue peculiarità.

Le proiezioni, poi, si occupano di adattare le varie tipologie di tipi al linguaggio. Questo fa sì una collection venga rappresentata da un oggetto WinRT di tipo `Ivector`, che in realtà ha un metodo `Add` in .NET, ma usa la funzione `push` se utilizzato da JavaScript, consentendo agli sviluppatori di adattarsi molto più facilmente al framework, senza dover rinunciare alle rispettive abitudini e convenzioni.

In base a queste considerazioni, la preferenza nell'utilizzare una strada piuttosto che un'altra è essenzialmente soggettiva: andremo a utilizzare la tecnologia con la quale già siamo in grado di produrre un risultato migliore, con la certezza che, tra l'altro, eventualmente è possibile mischiare le varie opzioni durante la creazione delle nostre applicazioni.

Chiariti questi aspetti, cosa ci serve per iniziare a sviluppare app per il Windows Store? Ci basta conoscere un po' di XAML, introdotto nei due capitoli precedenti, (ovviamente) C# e avere Visual Studio installato su Windows 10..

I tool per sviluppare

Per sviluppare applicazioni per WinRT è necessario, obbligatoriamente, dotarsi di un PC con installato Windows 10.

Attraverso il download dei tool, si ha accesso a una serie di funzionalità che di seguito saranno analizzate nel dettaglio. Tutti i tool offerti sono disponibili gratuitamente.

Troviamo un riepilogo dei download all'indirizzo: <http://www.winrtitalia.com/sviluppo/>.

I tool gratuiti, offerti all'interno di Visual Studio Community, presentano alcune limitazioni rispetto alle versioni a pagamento, ma consentono di partire con lo sviluppo di app in WinRT senza grosse limitazioni. Tutti i concetti e i suggerimenti contenuti in questo libro sono sfruttabili anche con Visual Studio Community.

Per quanto riguarda lo sviluppo di Universal Windows App, vengono aggiunti nuovi template di progetto, oltre a un nuovo designer. Torneremo fra poco su queste due tipologie di progetti, analizzandone le differenze.

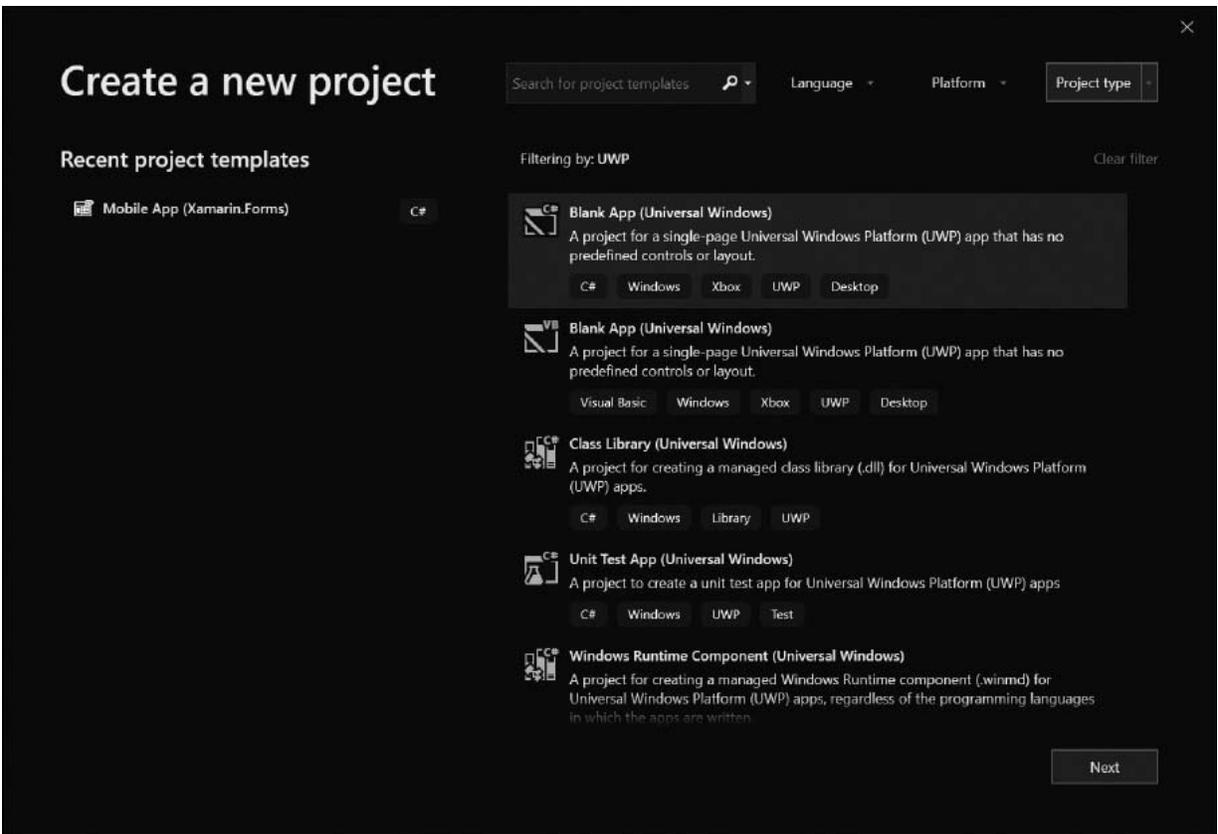


Figura 15.1 – I progetti aggiuntivi offerti in Visual Studio per creare applicazioni Universal Windows Platform.

Creando un nuovo progetto di tipo “Blank App”, come è visibile nella [Figura 15.1](#), possiamo notare come il designer di Visual Studio sia integrato con molti concetti mutuati da Blend e porti diverse novità per chi è abituato a sviluppare con WPF.

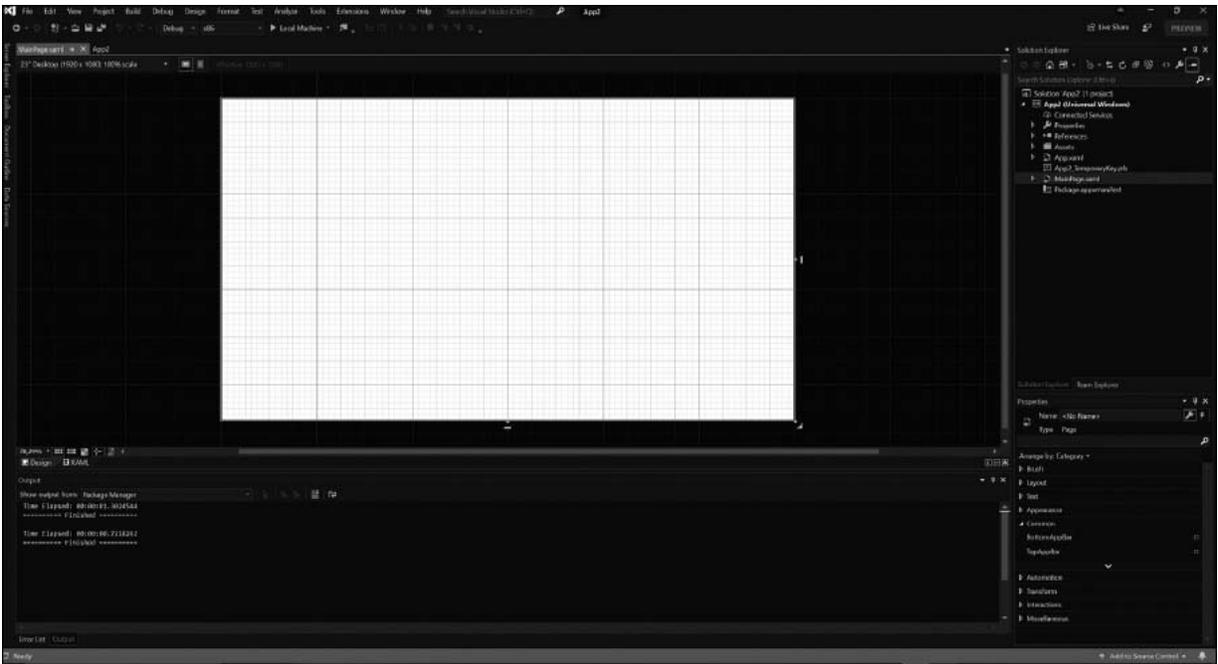


Figura 15.2 – L’ambiente specifico per lo sviluppo di UWP, all’interno di Visual Studio.

Come abbiamo accennato, il nuovo designer dello XAML è mutuato da quello di Blend, un tool che affianca Visual Studio per gestire design, e UX.

Visual Studio introduce tipi differenti di progetti, per le UWP i principali sono:

- ❑ **Blank App:** è un template minimo, all’interno del quale viene aggiunto solo quello che è strettamente necessario.
- ❑ **Class Library:** consente di creare un componente sotto forma di class library.
- ❑ **Windows Runtime Component:** per creare componenti che possano essere riutilizzati in WinRT, a prescindere dal linguaggio.
- ❑ **Unit Test Library:** consente di creare un progetto ad hoc per fare Unit Testing in applicazioni UWP.

L’uso di ciascuna di queste tipologie di template di progetto va a coprire esigenze particolari. Tuttavia, al contrario di quanto succedeva in

precedenza, non esistono template per la configurazione iniziale di un'applicazione..

Rispetto al passato, il numero di template si è ridotto, Per compensare questa perdita è stato creato un nuovo progetto su GitHub, denominato Template 10 (<https://github.com/Windows-XAML/Template10>). Questo progetto affronta le problematiche ricorrenti nello sviluppo non solo delle Universal Windows App, spaziando dal modello di layout fino ad affrontare tematiche come il MVVM. Queste problematiche sono affrontate in modo più aderente ai pattern che in passato; inoltre il progetto, essendo ospitato su GitHub, beneficia di uno sviluppo continuo e costante, che garantisce miglioramenti continui.

[La prima app per il Windows Store](#)

Aggiungiamo una nuova soluzione, in cui andremo a creare la nostra prima app. Dopo aver scelto il linguaggio (C#, nel nostro caso) vedremo apparire una serie di file che variano a seconda del tipo di template scelto.

Partiamo, per semplicità, con il template denominata “Blank App”. Per mandare in esecuzione l'applicazione, ci basterà premere F5, come siamo già abituati a fare. Quello che avviene dietro le quinte è che la nostra app viene compilata all'interno di un package, che prende l'estensione .appx.

Questo è un concetto già noto, se abbiamo dimestichezza con i precedenti modelli di sviluppo di applicazioni moderne: di fatto, questi package sono degli archivi compressi, con estensione custom, al cui interno, una volta scompattati, troveremo il contenuto vero e proprio. Nel caso di un .appx, oltre alla nostra app compilata, troveremo anche un insieme di informazioni, tra cui un manifest, all'interno del quale l'app dichiara le capabilities, cioè le caratteristiche che supporta.

La gestione delle capabilities è essenziale ed è contenuta in un file XML, chiamato `Package.appxmanifest`, che può essere anche aperto da Visual Studio con un editor visuale, che possiamo vedere nella [Figura 15.3](#).

È attraverso questo tool che possiamo specificare le informazioni relative al titolo dell'app, alle sue live tile (wide e normali) e, tra le altre cose, alle icone visualizzate con le notifiche push.

Dato che non tutti i computer di sviluppo saranno dotati di monitor ruotabili nell'immediato, Visual Studio integra un tool per testare questi

aspetti. Quest'ultimo, tra l'altro, è molto interessante a prescindere da questa funzionalità, perché consente di emulare anche risoluzioni differenti, simulare la posizione geografica, simulare il touch e catturare screenshot.

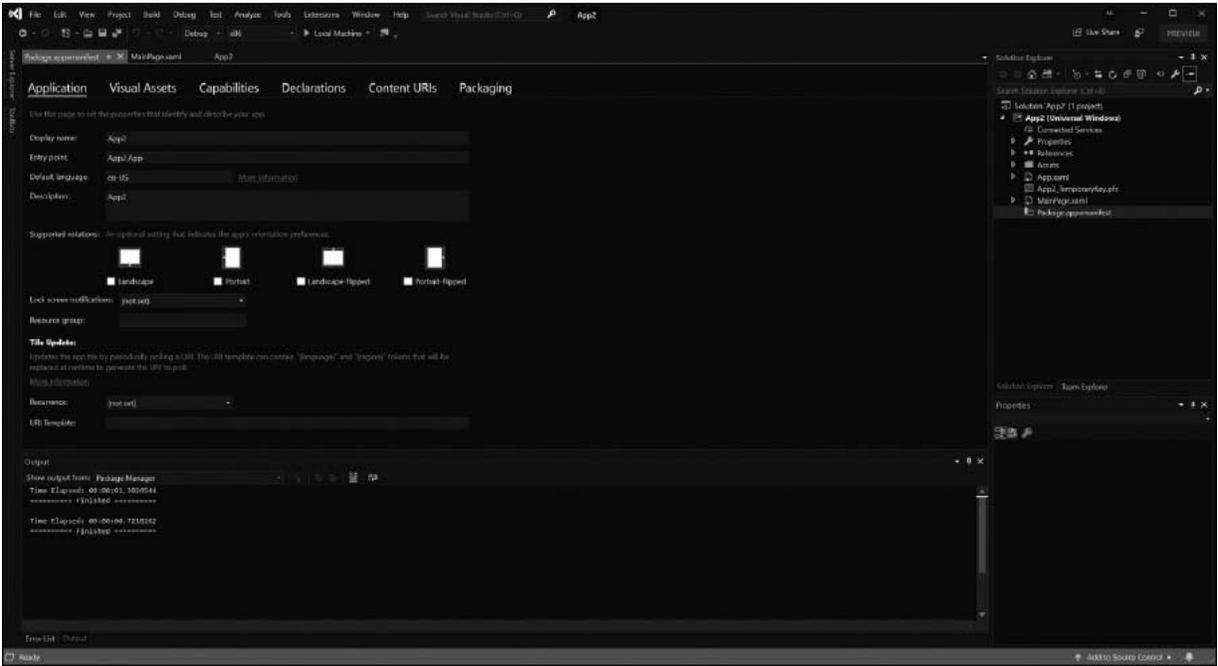


Figura 15.3 – Visual Studio ci consente di gestire le informazioni del manifest dell'applicazione in maniera visuale.

Nella [Figura 15.4](#) possiamo vedere come l'app possa essere testata a una risoluzione molto elevata (27" – 2560x1440) in portrait, senza per questo aver bisogno di un hardware che sia effettivamente dotato di queste caratteristiche.



Figura 15.4 – Il Simulator simula risoluzioni differenti, ruota il device, gestisce la geolocalizzazione e simula il touch.

Avviandoci per la prima volta alla scrittura di applicazioni UWP, potremmo rimanere spiazzati dalla sua caratteristica tipica di eseguire tutte le operazioni in maniera **asincrona**, come già anticipato nell'introduzione, all'inizio di questo capitolo. Nell'idea di Microsoft, tutte le applicazioni devono essere fast and fluid, cioè veloci e fluide: inevitabilmente, questo si traduce in una serie di best practice che lo sviluppatore deve seguire.

La scelta dell'asincrono è dettata da una ragione precisa: c'è un solo thread che ha una coda delle operazioni da compiere sulla UI, per cui, ogni volta che viene richiesta l'esecuzione di un'operazione, è necessario attendere che le altre siano state completate. In tutto questo, se le chiamate non fossero asincrone, **il thread principale resterebbe bloccato**. Per capirci meglio, in uno scenario del genere, anche il semplice clic su un bottone inibirebbe qualsiasi altra operazione come, per esempio, un'animazione, perché il flusso sarebbe inevitabilmente legato al completamento del codice nell'event handler.

Sfruttando il modello asincrono, invece, WinRT è in grado di non rallentare in alcun modo l'applicazione e, al tempo stesso, di **eseguire più operazioni in parallelo**. È possibile, infatti, avere più thread, che vengono utilizzati in maniera automatica per rendere le operazioni il più fluide possibile, benché quello legato al rendering dell'UI resti soltanto uno.

WinRT è **multi-threading**, con tutti i vantaggi che questo comporta in quegli scenari in cui è necessario eseguire più operazioni in contemporanea come, per esempio, nel caso di animazioni molto complesse. Il .NET dispone dei costrutti ad hoc, che si basano (ed estendono) sulla Task Parallel Library (TPL), già introdotta con il .NET Framework 4.0 rivista per essere integrata direttamente tanto in C# 2012 quanto anche in C# . Per una trattazione della TPL vi rimandiamo al [Capitolo 8](#).

Tutto questo si traduce, per noi sviluppatori, nel poter scrivere il codice dell'[esempio 15.1](#).

Esempio 15.1

```
private async void MyMethod()
{
    var results = await GetDataAsync();
    if (results.Count>0)
    {
        // altro codice...
    }
}
```

Come si può notare da questo esempio, grazie all'uso di `async/await`, il codice diventa semplice e leggibile, pur mantenendo la sua asincronicità: per noi sviluppatori è un doppio vantaggio, perché abbiamo codice semplificato e applicazioni che non bloccano l'UI.

Supponiamo di voler caricare delle informazioni da remoto, da un feed RSS. Ci basterà scrivere un codice come quello dell'[esempio 15.2](#), avendo cura di salvare le informazioni all'interno di una `ObservableCollection`.

Esempio 15.2

```
private async Task<ObservableCollection<SampleDataItem>> GetDataAsync()
{
    SyndicationClient client = new SyndicationClient();
```

```

Uri feedUri = new Uri("http://feed.aspitalia.com/feed.xml");

var feed = await client.RetrieveFeedAsync(feedUri);
var feedItems =
    feed.Items.OrderByDescending(x => x.PublishedDate).Take(10);

var group = new SampleDataGroup("ID", "ASPIItalia.com", "", "", "");

var items = new ObservableCollection<SampleDataItem>();
foreach (var item in feedItems)
    items.Add(new SampleDataItem(item.Id,
        System.NET.WebUtility.HtmlDecode(item.Title.Text),
        "", null, "", "", @group));
return items;
}

```

Come si può notare, utilizzando il tipo `Task<T>` e specializzandolo passando una collection, nel nostro caso una `ObservableCollection<T>`, è possibile rendere tutto il metodo asincrono. Dietro le quinte a sua volta recupera gli elementi del feed aggiungendoli mano a mano alla collezione di ritorno, il binding di XAML ci semplifica tutti questi aspetti, poiché mostrerà in automatico gli elementi, man mano che li andiamo a inserire. Per ottenere la UI rappresentata nella [Figura 15.5](#), abbiamo utilizzato un `ListView`, andando a mettere in binding gli oggetti recuperati dal feed. Il risultato è che il controllo `ListView` mostrerà i nostri dati con i template predefiniti, su cui possiamo apportare personalizzazioni.

Il risultato è visibile nella [Figura 15.5](#), nella quale abbiamo creato un semplice `DataTemplate` composto da un controllo `Image` e un template nel quale andiamo a mostrare direttamente i dati del feed.

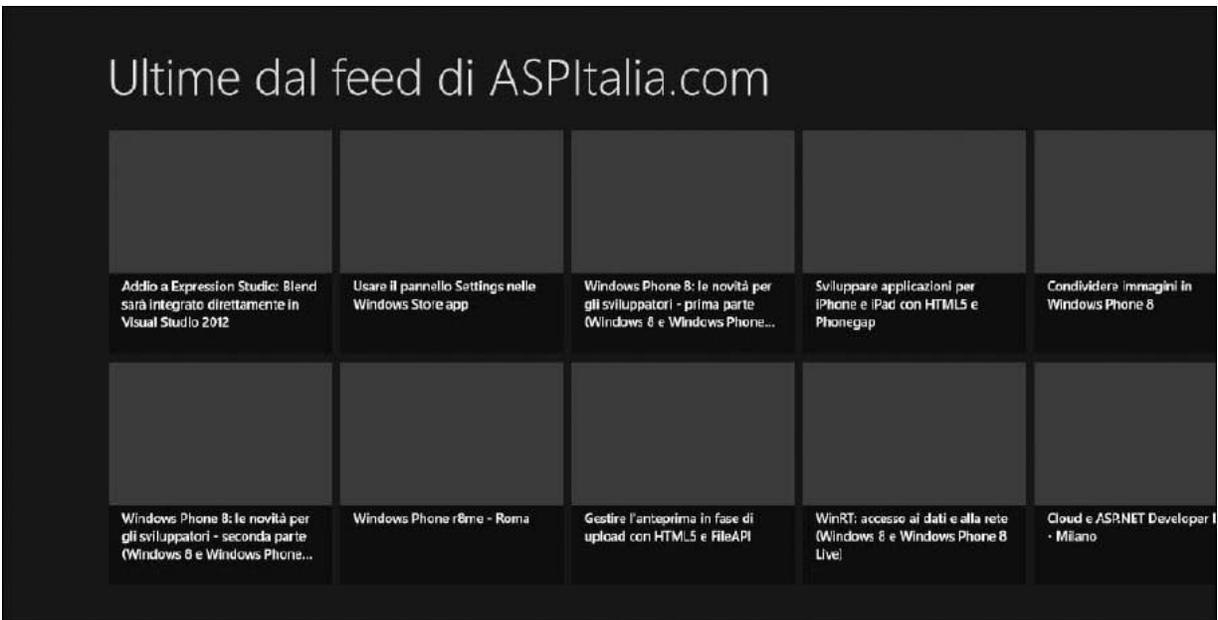


Figura 15.5 – Il risultato della nostra semplice app che mostra i feed.

A questo punto, dopo aver caricato il feed, dobbiamo gestire il click sul singolo elemento. L'[esempio 15.3](#) mostra come abbiamo fatto.

Esempio 15.3 - XAML

```
<ListView
  x:Name="itemListView"
  AutomationProperties.AutomationId="ItemsListView"
  AutomationProperties.Name="Items"
  TabIndex="1"
  Grid.Row="1"
  Visibility="Collapsed"
  Margin="0,-10,0,0"
  Padding="10,0,0,60"
  ItemsSource="{Binding Source={StaticResource itemsViewSource}}"
  ItemTemplate="{StaticResource Standard80ItemTemplate}"
  SelectionMode="None"
  IsSwipeEnabled="false"
  IsItemClickEnabled="True"
  ItemClick="ItemView_ItemClick"/>
```

Esempio 15.4 - C#

```
private void ItemView_ItemClick(object sender, ItemClickEventArgs e)
{
```

```
string url = ((MyData)e.ClickedItem).Id;
this.Frame.Navigate(typeof(DetailsPage), url);
}
```

La pagina su cui andremo a navigare mostrerà un controllo `webView`, che presenterà l'URL all'interno di un browser: il relativo codice è disponibile con gli esempi del libro.

Quanto abbiamo detto fino in questo momento vale sia che la nostra applicazione si trovi a essere eseguita sul Desktop sia su altri dispositivi, come Tablet e IoT.

In quest'ultimo caso, però, dobbiamo prestare attenzione ad alcuni dettagli, come, per esempio, la gestione del tasto back.

Windows 10 uniforma anche questo aspetto. Adesso le applicazioni eseguite in Tablet Mode offrono un tasto back di sistema.

La gestione è molto semplice e adotta il paradigma `GetForCurrentView`. Utilizzando quindi la classe statica `SystemNavigationManager`, possiamo richiamare il metodo `GetForCurrentView` e recuperare così l'oggetto `SystemNavigationManager` sul quale possiamo sottoscrivere l'evento `BackRequested`, come viene mostrato nell'[esempio 15.5](#).

Esempio 15.5

```
public MainPage()
{
    this.InitializeComponent();
    var navigationManager = Windows.UI.Core.SystemNavigationManager.
    GetForCurrentView();
    navigationManager.BackRequested += NavigationManager_BackRequested;
}
```

In questo modo è semplicissimo costruire un'applicazione che, grazie al layout fluido che le nuove Universal Windows App mettono a disposizione, si adatti alle varie risoluzioni disponibili nonché ai vari tipi di device.

Costruire applicazioni per WinRT richiede una conoscenza approfondita della piattaforma: in queste poche pagine abbiamo solo visto quanto, conoscendo XAML e C#, diventi facile e veloce costruire semplici applicazioni. Ma per muoversi in scenari più complessi è necessario approfondire in maniera specifica l'argomento.

Adesso che, con l'analisi delle caratteristiche dello XAML abbiamo introdotto le applicazioni per Universal, possiamo procedere rapidamente ad analizzare quanto sia possibile fare con XAML e WPF.

Applicazioni desktop con Windows Presentation Foundation

Dal momento che abbiamo analizzato le UWP, è giunta l'ora di affrontare un'altra tipologia di applicazioni che possiamo realizzare, sempre sfruttando XAML: le applicazioni desktop. Come abbiamo anticipato, questa tipologia è stata la prima ad adottare questo linguaggio per la definizione del layout, permettendo poi l'evolversi di tutte le altre piattaforme. Il framework di riferimento per le applicazioni desktop è Windows Presentation Foundation (WPF) ed è stato introdotto con il .NET Framework 3.0, già nel 2006. È un'alternativa alle Windows Forms e, in generale, a tutte quelle tecnologie che sfruttano Win32 (le API native di Windows) per lo sviluppo dell'interfaccia desktop.

Rappresenta quindi una svolta molto importante rispetto al passato, perché porta con sé una differenza fondamentale: in Win32 ogni framework che utilizziamo non è altro che un wrapper verso le API native, le quali disegnano l'interfaccia con i pregi e i limiti di API in C, che portano con sé più di dieci anni di evoluzione. Questo fa sì che, indipendentemente dal linguaggio utilizzato, le applicazioni abbiano tutte lo stesso stile grafico e aderiscano al tema del sistema. Con WPF, invece, Win32 è messo completamente da parte e sostituito con Direct3D come engine grafico. Questo permette di creare un nuovo modello di layout da sfruttare con XAML, per la generazione di un'interfaccia che viene renderizzata a video in modo accelerato, tramite GPU, consentendoci di utilizzare anche strumenti più avanzati, quali trasformazioni, effetti e 3D.

Creare un'applicazione WPF è piuttosto semplice, come del resto lo è per le altre tipologie, e non richiede nessun tool aggiuntivo, perché già supportato da Visual Studio. Nella creazione di un nuovo progetto, troviamo le relative voci con il prefisso `WPF` all'interno del menu `Windows`, come viene mostrato nella [Figura 15.6](#).

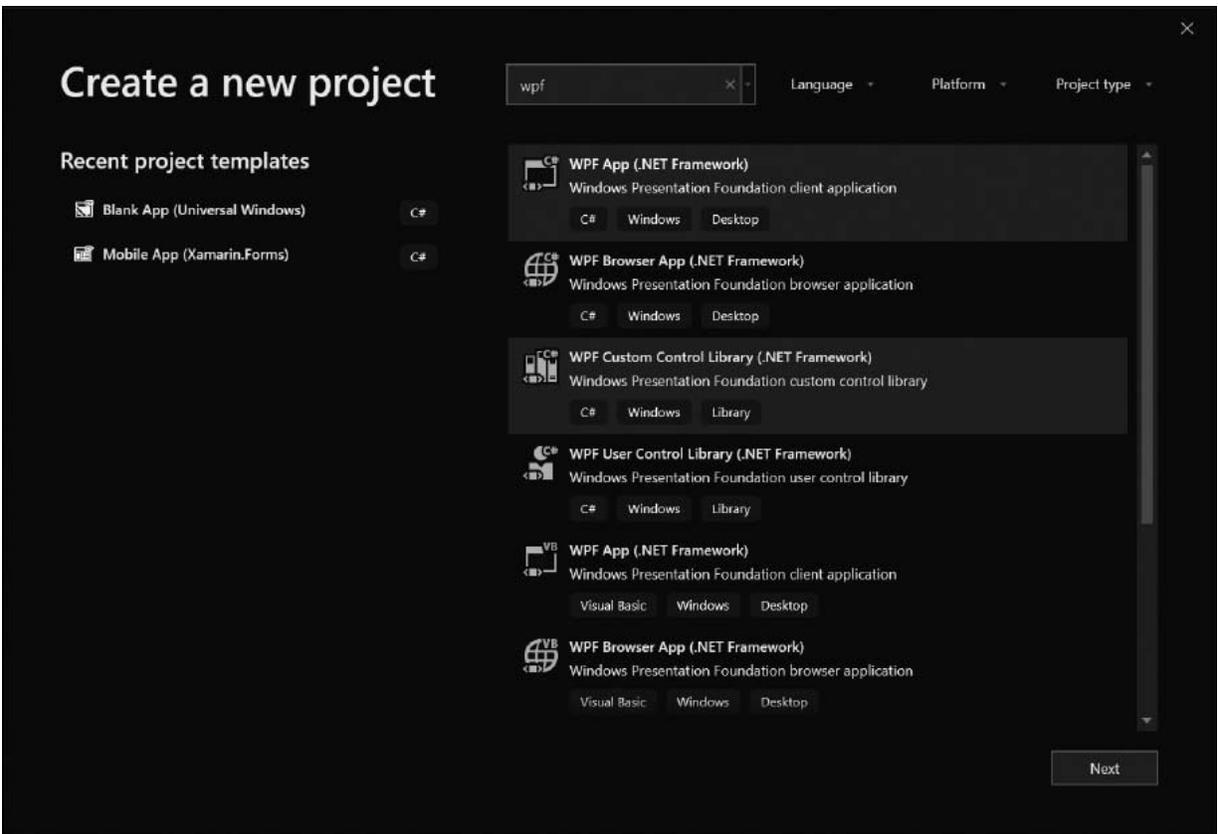


Figura 15.6 – Creazione di un nuovo progetto WPF con .NET Core e .NET Framework.

È importante non confondere tali voci con Windows Forms Application, poiché queste ultime rappresentano il vecchio modello di sviluppo, basato su Win32. In questa tipologia di progetto possiamo utilizzare i componenti tipici anche di ambienti server oppure ADO.NET, creare servizi distribuiti o sfruttare COM. Il mondo desktop, inoltre, non pone limiti: possiamo eseguire codice in background, accedere a qualsiasi file o cartella, interagire con tutto l'ecosistema Win32 e sfruttare al massimo tutte le caratteristiche della macchina. Per contro, questo mondo è maggiormente soggetto a problemi di sicurezza ed è disponibile solo su Windows, quindi su piattaforme x86 e x64.

Sempre dalla [Figura 15.6](#) notiamo la presenza della voce WPF App declinata nei due runtime disponibili: .NET Framework e .NET Core. Se stiamo creando un nuovo progetto, la scelta consigliata è quella di utilizzare .NET Core. In questo modo, la nostra app godrà di più longevità (il .NET

Framework 4.8 è l'ultima versione rilasciata) e potrà sfruttare tutte le caratteristiche di C# 8.

Creazione di un progetto

Partendo dalla [Figura 15.6](#), nella quale selezioniamo la voce WPF Application, proseguiamo creando il progetto. Se abbiamo scelto il .NET Framework, troveremo un progetto di vecchia tipologia, che referencia PresentationCore, PresentationFramework e WindowsBase, le DLL di WPF, visibili nella [Figura 15.7](#), già presenti sulla nostra macchina perché installate tramite il .NET Framework.

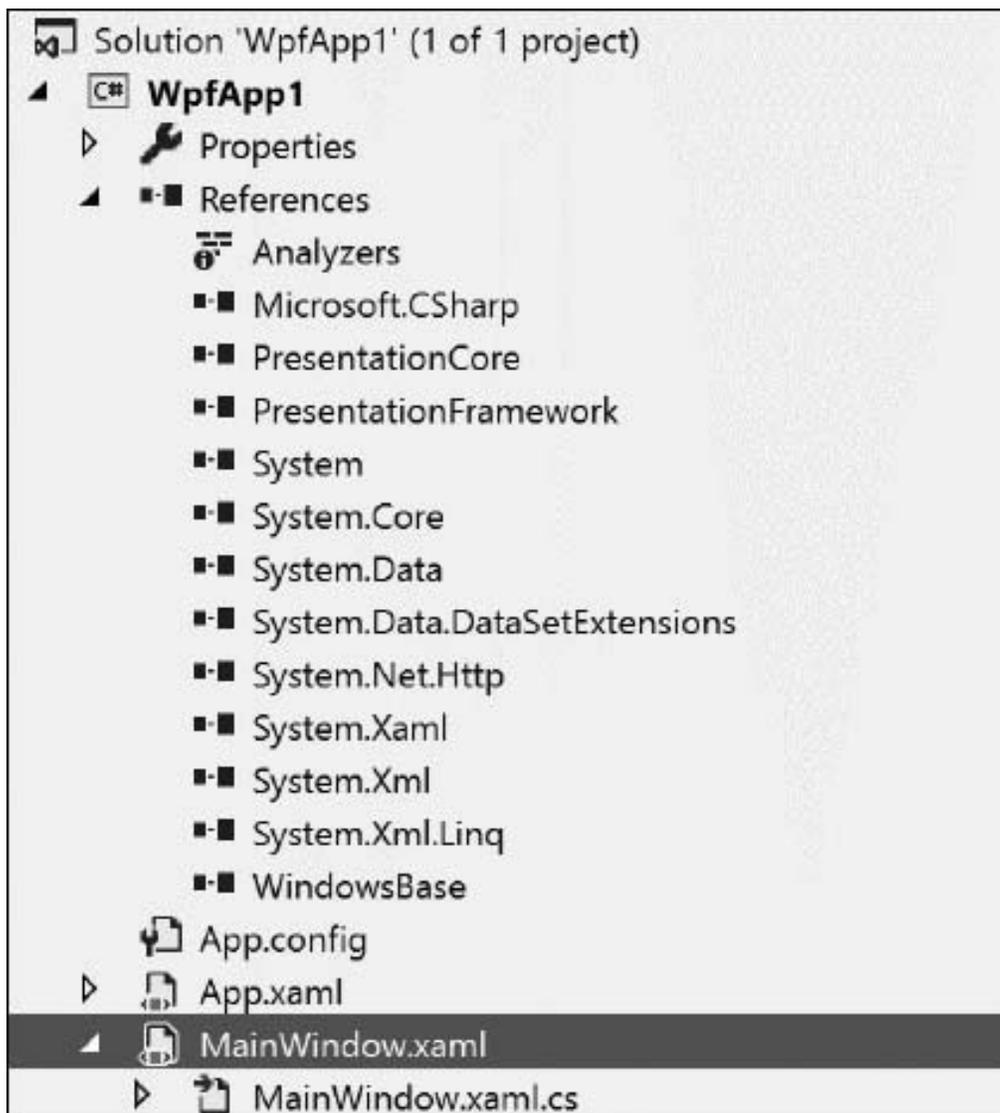


Figura 15.7 – Riferimenti di un progetto WPF con il .NET Framework.

Se invece abbiamo scelto di utilizzare .NET Core, il progetto creato sarà della nuova tipologia, cioè un XML che possiamo modificare anche live, contenente le indicazioni del SDK da usare. Il contenuto del file csproj ottenuto è visibile nell'[esempio 15.6](#).

Esempio 15.6 - CSPROJ

```
<Project Sdk="Microsoft.NET.Sdk.WindowsDesktop">
  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>netcoreapp3.0</TargetFramework>
    <UseWPF>>true</UseWPF>
  </PropertyGroup>
</Project>
```

Tolta questa iniziale differenza di progetto, le possibilità offerte da WPF sono identiche su entrambi i runtime. Proseguendo nell'analisi del progetto, quello che ritroviamo è un approccio del tutto simile a quanto già visto per le altre tipologie di applicazioni. Il file `Application.xaml` rappresenta l'entry point dell'applicazione, nel quale posizionare le risorse e dove viene indicata la vista iniziale da caricare, come è possibile notare nell'[esempio 15.7](#).

Esempio 15.7 - XAML

```
<Application x:Class="Application"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  StartupUri="MainWindow.xaml">
  <Application.Resources>
  </Application.Resources>
</Application>
```

L'attributo `startupUri` indica la finestra da avviare al lancio dell'applicazione. Essendo un'applicazione desktop, non disponiamo di pagine ma di finestre, che possiamo gestire come vogliamo. Il template predefinito crea per noi una `MainWindow`, con i consueti pattern XAML e

codice. Di conseguenza, l'elemento radice del file `MainWindow.xaml` è `Window`, ma al suo interno trovano spazio i normali elementi e controlli che accomunano tutte le tipologie delle varie applicazioni. Nell'[esempio 15.8](#) possiamo vedere un pulsante inserito al centro della finestra.

Esempio 15.8 - XAML

```
<Window x:Class="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 HorizontalAlignment="Center"
    VerticalAlignment="Center"
    Content="Pulsante" />
  </Grid>
</Window>
```

A questo punto possiamo avviare il progetto premendo F5 e otterremo la finestra visibile nella [Figura 15.8](#).

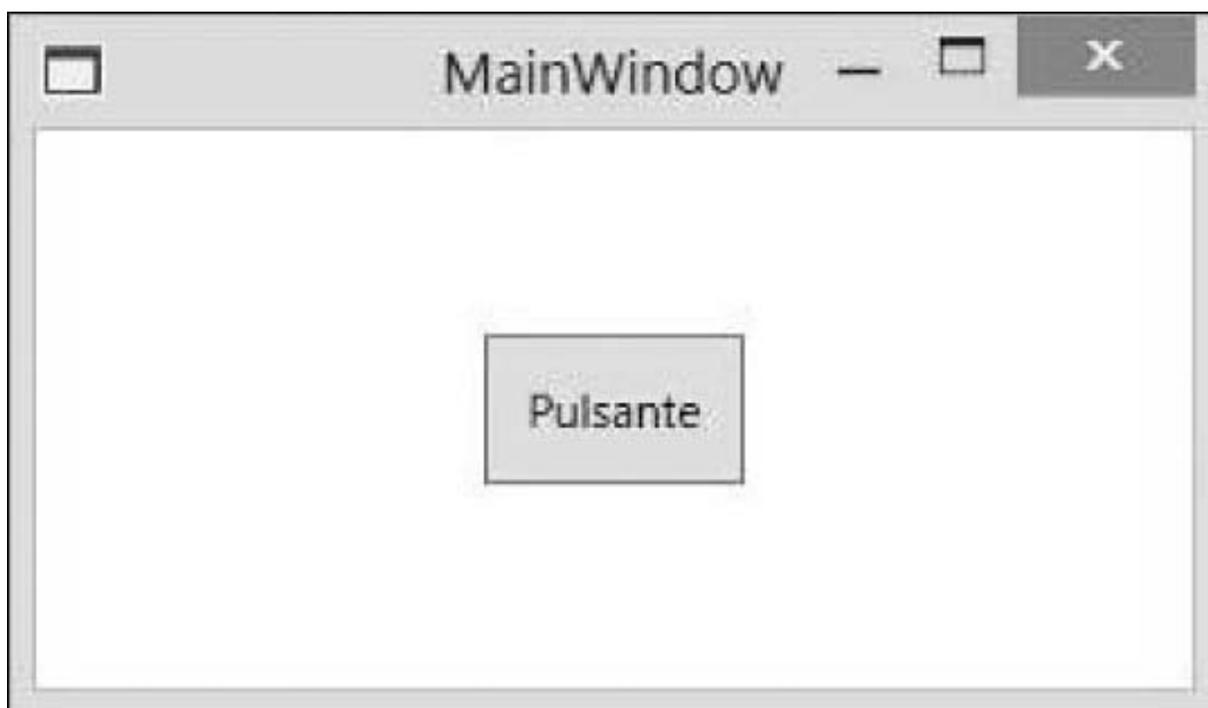


Figura 15.8 – Una semplice applicazione desktop in WPF.

Come possiamo notare, è una finestra desktop che possiamo ingrandire, minimizzare o ridimensionare a piacimento. Il pulsante, inoltre, dispone dello skin di sistema (in questo caso Windows 8), poiché WPF dispone di molteplici stili e template già preconfezionati, che vengono definiti dal sistema operativo e del tema scelto. Di fatto, quindi, non usiamo il rendering di Win32, come già anticipato, ma emuliamo l'interfaccia del sistema con l'engine di WPF.

La cromatura della finestra è l'unico fra gli elementi appartenenti a Win32 che non possiamo personalizzare a piacimento con stili e template, ma solo tramite le proprietà messe a disposizione dal sistema. Le finestre, quindi, sono la spina dorsale delle applicazioni WPF e dunque meritano un approfondimento.

Gestire le finestre

La finestra è quel componente sul quale possiamo agire solo attraverso proprietà che poi agiscono sul sistema operativo, e la sua cromatura non può essere personalizzata. Possiamo cambiarne però le caratteristiche tipiche, quali l'icona, il titolo e altre informazioni sul dimensionamento della finestra. Nell'[esempio 15.9](#) possiamo vedere definite tutte queste proprietà principali.

Esempio 15.9 - XAML

```
<Window x:Class="MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Height="350"
Width="525"
Icon="icon.png"
Title="MainWindow"
ResizeMode="CanResize"
ShowInTaskbar="True"
WindowStartupLocation="CenterScreen"
WindowState="Maximized">
```

Molte proprietà hanno un nome auto esplicativo e possiamo facilmente intuire cosa fanno. Altre, invece, come `ResizeMode` servono per scegliere se dare la possibilità di ridimensionare la finestra; `WindowState` definisce se

avviare massimizzata la finestra; `WindowsStartupLocation` permette di indicare se avviare la finestra al centro dello schermo. Infine, `ShowInTaskbar` (normalmente già a `true`), permette di mostrare – o di non farlo – la finestra nella taskbar di sistema, la quale consente all'utente di passare da un'applicazione a un'altra o di visualizzare un'anteprima della finestra, come è visibile nella [Figura 15.9](#).

In genere si rende visibile solo la finestra principale e si nascondono tutte le finestre figlie. Una caratteristica del mondo desktop, infatti, è quella di gestire l'esperienza dell'utente attraverso altre finestre, in genere modali.

Un'altra proprietà molto interessante è `WindowsStyle`, che permette di scegliere il tipo di cromatura della finestra. Tra i valori più interessanti può assumere `ToolWindow`, che rende più adatta la finestra a essere un pannello di strumenti agganciato alla finestra principale, oppure `None`, che nasconde l'intera cromatura, permettendoci di creare un'esperienza interamente in XAML, anche a schermo pieno.

Sebbene all'interno della finestra possiamo utilizzare tutti i pannelli e i controlli del mondo XAML, è spesso utile dividere logicamente l'esperienza dell'utente, portandola su finestre differenti, da aprire solo quando risulti necessario. Un nuovo cliente, il dettaglio di un ordine o una scelta obbligatoria per l'utente sono alcuni esempi che trovano risposta nella creazione di una nuova finestra. Per farlo dobbiamo innanzitutto aggiungere una `Window` al nostro progetto. La voce `Add Window` sotto il menu `Project` ci permette di farlo facilmente e avere un nuovo XAML da riempire, che per l'occasione chiamiamo `ChildWindow`. Come suggerito in precedenza, impostiamo `ShowTaskbar` a `false` e, come si fa generalmente, diamo una dimensione fissa e non ridimensionabile alla finestra. A questo punto possiamo creare la finestra e mostrarla quando necessario, per esempio premendo sul pulsante creato in precedenza nella finestra principale. Sfruttando il designer possiamo effettuare un doppio click con il mouse sul pulsante, per ottenere lato codice la gestione dell'evento `Click` del `Button`, così come abbiamo illustrato nel [Capitolo 14](#).



Figura 15.9 – Taskbar con l’anteprima della finestra.

Per mostrare la finestra dobbiamo innanzitutto istanziare la classe che la finestra stessa rappresenta. Questo ci permette, tramite costruttore, di passare parametri e informazioni utili alla finestra che andiamo ad aprire. Una volta ottenuta l’istanza, possiamo optare per due metodi: `show` o `ShowDialog`. Il primo mostra la finestra e permette all’utente di passare da una finestra all’altra, mentre il secondo mostra la finestra modale, cioè impedisce all’utente di interagire con altre finestre dell’applicazione, finché la prima finestra non viene chiusa. Nell’[esempio 15.10](#) utilizziamo una finestra modale.

Esempio 15.10

```

public class MainWindow
{
    private void Button_Click(object sender, RoutedEventArgs e)
    {
        // Creo l'istanza
        ChildWindow child = new ChildWindow();
        // Finestra modale
        bool result = child.ShowDialog();
    }
}

```

Un'altra caratteristica del metodo `ShowDialog` è il fatto che non ritorna fino a quando l'utente non ha chiuso la finestra. Quindi il codice successivo, definito dopo la chiamata a `ShowDialog`, viene eseguito solo dopo la chiusura, di cui facoltativamente possiamo conoscere l'esito. Nell'esempio, la variabile booleana `result` ci permette di conoscere se la finestra è stata chiusa premendo `x` sulla cromatura o se, dal punto di vista logico della finestra, è stata chiusa perché le operazioni si sono concluse.

Per dare esito positivo, invece, dobbiamo modificare la `ChildWindow` e posizionare un bottone `OK` o `chiudi` che al `click` valorizzi la proprietà `DialogResult` della finestra, come possiamo vedere nell'[esempio 15.11](#).

Esempio 15.11

```

public class ChildWindow
{
    private void OK_Click(object sender, RoutedEventArgs e)
    {
        this.DialogResult = true;
    }
}

```

La valorizzazione della proprietà scatena anche la chiusura della relativa finestra, permettendo a chi l'ha aperta, di leggere il valore. Consideriamo inoltre che, poiché la finestra è rappresentata da una classe, su di essa possiamo porre proprietà che espongono ulteriori valori, da leggere poi dall'esterno, una volta chiusa la finestra.

Come abbiamo detto WPF è diventato una scelta obbligata per il mondo desktop fino all'arrivo della UWP. Ma adesso come gestire la migrazione di

tutto l'attuale? È possibile modernizzare le nostre applicazioni WPF, e non solo, con le XAML Island.

Le XAML Island

Per anni le applicazioni Windows sono state sviluppate prima con MFC, poi con Windows Form, per passare poi a WPF. Attualmente ci troviamo di fronte a un enorme bacino di applicazioni sviluppate e mantenute da anni.

Con il tempo e con il numero di funzionalità implementate, queste applicazioni diventano ogni giorno più difficili da migrare a nuove tecnologie, dalle quali potrebbero solo trarre grandi benefici prima di tutto dal punto di vista dell'usabilità, della velocità e, non ultimo, del punto di vista della UI, ottenendo sicuramente un aspetto più contemporaneo. Al fine di rendere accessibili queste nuove tecnologie alle applicazioni realizzate con tecnologie precedenti alla piattaforma UWP, Microsoft ha reso disponibili le XAML ISLAND. Queste "isole" di XAML permettono ad applicazioni WPF e Windows Forms di ospitare e utilizzare componenti e controlli tipici della UWP. Questa opportunità, di fatto, rende possibile una parziale migrazione alle nuove tecnologie e ai nuovi framework. Migrazione che, nel corso del tempo, da parziale può diventare totale.

A partire dall'aggiornamento di ottobre di Windows 10 October (SDK 17763) sono state aggiunte una serie di API, come `WindowsXamlManager` e `DesktopWindowXamlSource`. La prima gestisce il framework UWP stesso e lo fa con il metodo `InitializeForCurrentThread`. La seconda rappresenta l'isola vera e propria. Attraverso la proprietà `Content`, lo sviluppatore può istanziare l'isola e gestirne il contenuto.

La classe `DesktopWindowXamlSource` renderizza e interagisce con l'input dell'utente direttamente dall'handler `HWND`, delegandone la gestione, come il `resizing`, allo sviluppatore.

Visto che sia WPF sia Windows Forms già gestiscono l'handler `HWND` in completa autonomia, l'onere per la gestione dell'handler `HWND` per la XAML island risultava essere un enorme freno all'utilizzo. Per questo motivo, Microsoft ha introdotto nel Windows Community Toolkit la classe `WindowsXamlHost`. Questa classe rappresenta a tutti gli effetti un wrapper per le API `WindowsXamlManager` e `DesktopWindowXamlSource` e si fa carico della loro gestione.

Per utilizzare la XAML Island, per esempio in un'applicazione WPF, per prima cosa è necessario aggiungere le seguenti librerie al nostro progetto WPF:

- `System.Runtime.WindowsRuntime`
- `System.Runtime.WindowsRuntime.UI.Xaml`
- `System.Runtime.InteropServices.WindowsRuntime`
- `Windows.Foundation.UniversalApiContract.winmd`
- `Windows.Foundation.FoundationContract.winmd`

In seguito dobbiamo installare via NuGet il pacchetto `Microsoft.Toolkit.Wpf.UI.XamlHost`. A questo punto, dichiariamo il namespace che ospita il wrapper.

Esempio 15.12

```
<Window xmlns:xamlhost="clr-namespace:Microsoft.Toolkit.Wpf.UI.XamlHost; assembly=Microsoft.Toolkit.Wpf.UI.XamlHost">  
  
<xamlhost:WindowsXamlHost x:Name="myUwpButton" InitialTypeName="Windows.UI.Xaml.Controls.Button" />  
</Window>
```

Come possiamo vedere, il wrapper viene aggiunto come qualsiasi altro controllo WPF all'interno della Window. Per specificare il controllo da istanziare, è sufficiente impostare la proprietà `InitialTypeName` con il suo nome completo.

Visto che i controlli UWP non sono disponibili fin da subito, per esempio perchè non sono ancora stati istanziati dopo l'esecuzione del metodo `InitializeComponent`, è necessario recuperare un'istanza solo quando questi controlli vengono effettivamente creati.

Esempio 15.13

```
myUwpButton.ChildChanged += MyUwpButton_ChildChanged;

...

private void MyUwpButton_ChildChanged(object sender, System.EventArgs e)
{
    if (myUwpButton.Child is Windows.UI.Xaml.Controls.Button button)
    {
        button.Content = "Capitolo 15";
        button.Click += (s, args)=>
        {
            MessageBox.Show("Hi from UWP Button!");
        };
    }
}
```

La prima riga di codice registra un handler per il caricamento del controllo rappresentato dal wrapper. Al caricamento del controllo è possibile farne un cast all'istanza precisa e utilizzarne i membri come siamo soliti fare con altre classi.

In questo modo abbiamo inserito, per quanto si tratti di un semplice bottone, un controllo UWP all'interno di un'applicazione WPF.

Migrare applicazioni desktop a .NET Core 3

Le XAML Island consentono di portare l'UI della Universal Windows Platform all'interno di applicazioni WinForms e WPF, ma possiamo fare di più e convertirle per riuscire a utilizzare il .NET Core 3.0. Quanto abbiamo visto finora a livello di XAML è valido tanto per .NET Framework quanto per .NET Core, ma usando quest'ultimo si hanno dei vantaggi, oltre a quelli già citati:

- ❑ **Performance:** i tempi di compilazione e avvio dell'applicazione e dell'esecuzione saranno migliori, e a seconda della tipologia dell'applicazione, saranno percettibili con il semplice utilizzo.
- ❑ **Facilità di test:** .NET Core viene eseguito side-by-side, quindi, contrariamente a quanto accade per .NET Framework, possiamo avere più versioni installate contemporaneamente e fare in modo che ogni applicazione usi la versione più appropriata senza correre il rischio che entrino in conflitto tra di loro.

- ❑ **CLI:** la riga di comando per .NET Core utilizzata per generare progetti, fare il restore dei pacchetti, creare le build, pubblicare i progetti e molto altro.
- ❑ **Self contained deployment:** possiamo pubblicare l'app in modo che porti con sé direttamente il runtime di .NET Core, non obbligando la macchina finale ad avere dei requisiti di installazione. Possiamo inoltre creare un unico exe auto che contenga tutte le nostre dll e quelle del runtime. Grazie al linker, infine, possiamo produrre un exe più piccolo, che includa solo l'IL delle API da noi effettivamente utilizzate.

La migrazione di applicazioni esistenti può creare un percorso complesso, visto che il porting è fortemente influenzato dalla tipologia di librerie utilizzare e dalla grandezza dell'applicazione. Sicuramente il primo passo di questo percorso di migrazione è quello di determinare se la nostra applicazione è una buona candidata, e per questo possiamo utilizzare il **Portability Analyzer**, un'estensione per il Visual Studio liberamente scaricabile dal Marketplace. L'analisi della Solution può richiedere del tempo, ma al termine avremo un report dettagliato sulle incompatibilità con il .NET Core. Molte di queste derivano dall'utilizzo di API non supportate o dipendenti dal sistema operativo. Per supportare questo tipo di API, cioè, è possibile scaricare il **Windows Compatibility Pack**. Come lascia intuire il nome, questa "estensione" al .NET Standard è compatibile unicamente con Windows. Non dobbiamo quindi mai confondere la possibilità di utilizzare .NET Core con la capacità di portare applicazioni desktop su sistemi operativi diversi da Windows. Per realizzare applicazioni cross platform, è necessario utilizzare Xamarin Forms.

Il Windows Compatibility Pack è la soluzione per far compilare applicazioni che usano caratteristiche tipiche della piattaforma Windows. Le oltre 20.000 API Windows-only incluse nel pack, contengono riferimenti a: Code Pages, CodeDom, Configuration, Directory Services, Drawing, ODBC, Permissions, Ports, Windows Access Control Lists (ACL), Windows Communication Foundation (WCF), Windows Cryptography, Windows EventLog, Windows Management Instrumentation (WMI), Windows Performance Counters, Windows Registry, Windows Runtime Caching e Windows Services. Per sfruttare tutte le API rese

disponibili dal Windows Compatibility Pack, è sufficiente installare il pacchetto di NuGet chiamato Microsoft.Windows.Compatibility, e importarlo nelle classi in cui ci sono riferimenti ad API non supportate direttamente da .NET Standard.

Aggiornamento dei progetti

Al termine dell'analisi è consigliabile procedere ad aggiornare la gestione dei pacchetti NuGet. Il **PackageReference** è un nuovo meccanismo per gestire i pacchetti all'interno del file di progetto anziché in un file separato: il packages.config. Di default, la **PackageReference** è usata per i progetti .NET Core, .NET Standard e UWP (se il target è pari o superiore a Windows 10 15063, il Creators Update) con l'eccezione dei progetti UWP C++.

I progetti che invece hanno ancora come target il .NET Framework, come le “vecchie” applicazioni WPF e Windows Forms, mantengono il restore tramite packages.config e, per poter utilizzare la PackageReference, è necessario apportare le seguenti modifiche a fine progetto, modificando il csproj:

Esempio 15.14

```
<PropertyGroup>
  <!-- ... -->
  <RestoreProjectStyle>PackageReference</RestoreProjectStyle>
  <!-- ... -->
</PropertyGroup>
```

Una volta modificato il file di progetto, possiamo migrare ogni singolo pacchetto utilizzato. L'[esempio 15.15](#) riporta la modifica necessaria a far funzionare Newtonsoft.

Esempio 15.15

```
<ItemGroup>
  <!-- ... -->
```

```
<PackageReference Include="Newtonsoft.Json" Version="12.0.1" />
<!-- ... -->
</ItemGroup>
```

Esistono anche tool esterni che automatizzano il meccanismo di conversione, ma intervenire manualmente, soprattutto su progetti complessi, è sicuramente la scelta migliore, specialmente in questo momento in cui l'ide non offre ancora un supporto adeguato.

[Retargeting e migrazione dei file di progetto](#)

Nel caso in cui il progetto da migrare faccia riferimento a una versione precedente al .NET Framework 4.7.2. È opportuno eseguire il retargeting, poiché è molto probabile che esistano delle alternative sul .NET Framework qualora una API non sia ancora specificata in .NET Standard.

Con i progetti .NET Core, Microsoft ha deciso di adottare una nuova tipologia di file di progetto, i cosiddetti SDK-style project. Uno degli aspetti fondamentali è che la gestione predefinita di NuGet è mediante il **PackageReference** ma abbiamo ulteriori vantaggi:

- ❑ I file di progetto sono semplificati e, di conseguenza, più piccoli e leggibili.
- ❑ Edit del file di progetto senza fare l'unload del progetto stesso.
- ❑ Supporto al multi-targeting.

I progetti .NET Standard utilizzano già il nuovo modello di progetto che nella sua forma più semplice può essere simile a quello dell'[esempio 15.16](#).

Esempio 15.16

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>
</Project>
```

Come possiamo notare, il file di progetto si è ridotto soprattutto grazie al fatto che adesso tutti i file sono importati automaticamente e solo quelli da escludere andranno specificati singolarmente, l'opposto di quanto avveniva in precedenza.

Conclusioni

In questo capitolo abbiamo analizzato le caratteristiche principali delle tipologie di applicazione e dei relativi framework che si basano sullo XAML per la definizione del layout. Nello specifico, abbiamo visto come iniziare nello sviluppo per il Desktop e come convertire o sfruttare le caratteristiche del .NET Core 3.0.

Tutto ciò è stato solo un assaggio di queste applicazioni, poiché meritano molto più spazio, ma pensiamo sia stato sufficiente per permetterci di capire quale strumento scegliere, a seconda dell'obiettivo che vogliamo raggiungere. Sicuramente imparare XAML premia per la possibilità di riutilizzo del markup, del layout e delle conoscenze che possiamo sfruttare agevolmente su più piattaforme.

Avendo chiarito questo aspetto, non ci resta che abbandonare lo sviluppo client side per dedicarci al Web, in quanto nel prossimo capitolo parleremo di ASP.NET e delle novità introdotte nell'ultima versione.

Applicazioni web con .NET

Nei capitoli precedenti abbiamo iniziato a capire come C# possa essere applicato con profitto allo sviluppo di applicazioni per Windows, nel caso specifico, utilizzando XAML. Nel corso di questo capitolo tratteremo invece lo sviluppo di applicazioni basate su ASP.NET e su ASP.NET Core, la versione nata specificamente per .NET Core.

ASP.NET rappresenta una delle tecnologie più utilizzate all'interno di .NET ed è alla base della grande diffusione di quest'ultimo. Consente di creare, con un paradigma molto semplice, applicazioni di qualsiasi tipo: combinando bene gli strumenti offerti da .NET, C# e Visual Studio, è possibile creare applicazioni di tutte le qualità e dimensioni, da quelle più semplici, fino alle classiche applicazioni enterprise.

Essendo di fatto destinato a rappresentare la sola interfaccia grafica, ASP.NET sfrutta molte delle tecnologie all'interno di .NET, oltre ovviamente a C# in quanto linguaggio. Cominciamo da una prima, rapida analisi di quello che ci consente di fare, per poi soffermarci su alcune fra le novità dell'ultima versione.

La prima pagina ASP.NET

Qualora fossimo totalmente digiuni di ASP.NET, è utile sottolinearne alcune caratteristiche peculiari. Prima di tutto, ASP.NET supporta due modalità per creare l'interfaccia utente:

- ❑ **ASP.NET Web Forms:** è il modello originale, disponibile solo per .NET Framework, dove ogni pagina viene chiamata Web Forms;

- ❑ **ASP.NET MVC 5 e ASP.NET Core MVC 3:** sono l'implementazione del pattern MVC (Model-View-Controller) per applicazioni ASP.NET e ASP.NET Core.

In passato, il modello maggiormente utilizzato è stato Web Forms, che offre un approccio basato su controlli ed eventi, molto simile a quanto possibile nello sviluppo RAD (Rapid Application Development) con le applicazioni per Windows. Viceversa, ASP.NET MVC e ASP.NET Core MVC, che ne condivide gran parte delle funzionalità, consentono un controllo maggiore in fase di definizione del markup e garantiscono, soprattutto, la possibilità di testare il codice. Se questo aspetto dovesse risultare cruciale, ASP.NET MVC è in grado di garantire questa caratteristica. Inoltre, dato che ASP.NET Core supporta di fatto solo il pattern MVC, è una scelta obbligata qualora si voglia sfruttare il supporto cross-platform di .NET Core: il motivo per il quale WebForms non è stato portato in .NET Core è nella sua difficile separazione in piccoli pezzi e nel legame indissolubile con Windows, IIS e .NET Framework.

All'interno di questo capitolo vi proponiamo una veloce trattazione di entrambi.

Creare un progetto ASP.NET

Non è cambiato molto negli ultimi anni su questo fronte, perché ASP.NET WebForms è una tecnologia considerata matura. Per creare la prima pagina basata su Web Forms, possiamo selezionare, dall'apposita voce all'interno di Visual Studio, un nuovo sito web. È possibile creare un sito web senza progetto, oppure un progetto di tipo sito web. In base a questa scelta, cambia il modello di compilazione e di distribuzione. Nella [Figura 16.1](#) è riportata la schermata mostrata da Visual Studio in fase di creazione di un nuovo progetto. Dopo aver scelto ASP.NET, dovremo scegliere Web Forms dalla maschera successiva.

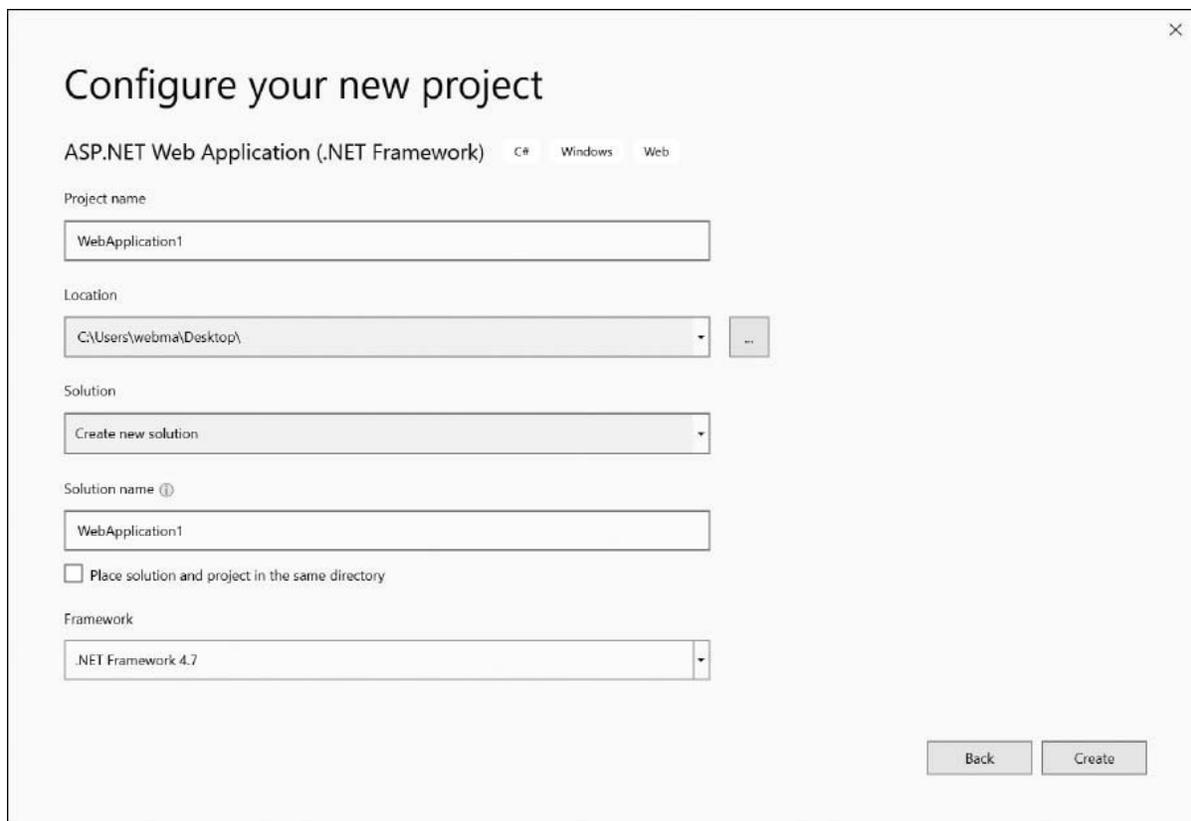


Figura 16.1 – Le possibili scelte in fase di creazione di un progetto con Visual Studio.

Nel caso ottimale per il sito web, per mettere in produzione il sito sarà necessario copiare tutti file all'interno del percorso. In tal caso, è possibile fare una modifica a un singolo file senza necessità di caricare l'intera applicazione, ma applicando le sole modifiche.

Il progetto web, viceversa, ha bisogno di una compilazione per ogni modifica che viene fatta. In fase di deployment (cioè di messa in produzione), sarà necessario copiare tutti i file, con l'esclusione di quelli che hanno estensione `.cs`, il cui contenuto è compilato automaticamente.

In entrambi i casi, la directory `/bin/`, posta sotto la root dell'applicazione, contiene tutti gli assembly che sono utilizzati all'interno dell'applicazione. Nel caso del sito web, vengono poi utilizzati altri tipi di risorse particolari, contenuti nelle directory riportate nella [Tabella 16.1](#).

Tabella 16.1 – Le directory speciali di ASP.NET

Directory	Descrizione
-----------	-------------

/bin/	Contiene gli assembly generati attraverso Visual Studio oppure contenenti oggetti di terze parti.
/App_Code/	È una directory pensata per memorizzare classi in formato sorgente, da compilare al volo insieme all'applicazione. Supporta un solo linguaggio per volta.
/App_Data/	Può contenere file di appoggio, come "txt" o "XML", piuttosto che file di database Access o SQL Server Express. È, più in generale, una directory pensata per contenere file, che sono protetti dal download ma che si possono sfruttare nelle pagine.
/App_Themes/	Include i file legati ai temi, funzionalità introdotta in ASP.NET a partire dalla versione 2.0.
/App_WebReferences/	Include i file generati per l'utilizzo delle reference di web service.
/App_LocalResource/	Contiene i file di risorse, spesso utilizzati per la localizzazione, ma locali alle singole pagine web.
/App_GlobalResource/	È il contenitore delle risorse globali, cui hanno accesso tutte le componenti dell'applicazione.

Tanto che si scelga di utilizzare il sito web, quanto di usare il progetto web, la creazione delle pagine all'interno non subisce particolari complicazioni. Alla fine, il progetto web è preferito all'interno dello sviluppo in team e per applicazioni di cui si voglia gestire in maniera più rigorosa il ciclo di sviluppo. Per utilizzare ASP.NET 4.8, è sufficiente accedere alle proprietà del progetto (o del sito), alla voce *Build* e quindi a *Target Framework*, specificando la versione 4.8. Questa azione ci consente di migrare in un colpo solo eventuali applicazioni esistenti basate su versioni precedenti Web Forms.

Sviluppare con Web Forms

Le pagine ASP.NET sono spesso chiamate Web Form, perché all'interno di queste ultime è presente un tag form particolare, con l'attributo `runat` impostato sul valore "server". Questa caratteristica, applicata ai frammenti di codice HTML, li rende **controlli server**. I controlli server sono istanze di classi che producono un markup e sono in grado di fornire un comportamento.

La semplice pagina ASP.NET è contraddistinta dall'estensione `.aspx` ed è generalmente suddivisa in almeno due parti, una denominata markup e l'altra codice. Spesso il codice è contenuto, per questioni di separazione logica, all'interno di un file separato, che per convenzione ha estensione `.aspx.cs`. In base al tipo di progetto prescelto, come viene spiegato nella parte precedente, il file con il sorgente può essere modificato senza necessità di compilazione.

La classe da cui tutte le pagine derivano è `System.Web.UI.Page`: questo garantisce che tutte si comportino allo stesso modo.

Generalmente, il markup è composto sia da frammenti di HTML sia da server control. Questi ultimi si riconoscono a colpo d'occhio, perché generalmente sono nella forma riportata nell'[esempio 16.1](#).

Esempio 16.1

```
<asp:TextBox id="FirstName" runat="server" />
```

La sintassi particolare che contraddistingue i controlli è di tipo XML, quindi il tag deve essere **well formed**, cioè ben formato. Il prefisso “asp:” è, in genere, riservato ai controlli disponibili all'interno di ASP.NET, mentre il suffisso è il nome vero e proprio della classe che sarà istanziata. I controlli che sono dotati di questo prefisso vengono chiamati **web control**.

Resta possibile aggiungere l'attributo `runat="server"` a qualsiasi frammento di HTML: in questo caso si parla di **HTML Control**. Questi ultimi hanno un modello a oggetti che ricalca quello dei tag dell'HTML a cui si riferiscono, mentre i web control hanno dalla loro parte un modello a oggetti che rende controlli molto diversi nel markup ma, in realtà, molto simili nel modello a oggetti. Questo rappresenta un vantaggio, perché rende più facile un loro eventuale interscambio: per impostare il testo, per esempio, useremo la proprietà `Text`, per impostare un URL di navigazione quella `NavigateUrl`. Questo porta a semplificare l'approccio in fase di creazione.

HTML e i web control possono essere approfonditi su <http://aspit.co/agw> e <http://aspit.co/agx>.

Tanto gli HTML quanto i web control producono alla fine codice HTML, ma lo fanno grazie a un componente particolare, che si chiama **page parser**. Questo componente è in grado di garantire che noi possiamo scrivere il markup nella maniera classica ma che, contestualmente, a runtime i frammenti vengono automaticamente tradotti in codice. A noi resta il vantaggio di poter scrivere l'interfaccia in maniera semplice e veloce, con il risultato di poter programmare i controlli posizionati nella pagina, impostandone le proprietà e, soprattutto, potendone gestire gli eventi.

Gli eventi, ilPostBack e il ViewState

I controlli posizionati all'interno della Web Form sono, a tutti gli effetti, degli oggetti e quindi, come tali, sono dotati di eventi. Per mostrare al meglio questo concetto, componiamo una form come quella dell'[esempio 16.2](#).

Esempio 16.2

```
<form runat="server">
  Inserisci il tuo nome:
  <asp:TextBox id="FirstName" runat="server" /> <br />
  <asp:Button id="ConfirmButton" runat="server" Text="Conferma"
    OnClick="ShowFirstName" />
</form>
```

Se lanciato nel browser, questo snippet produce una schermata simile a quella della [Figura 16.2](#).

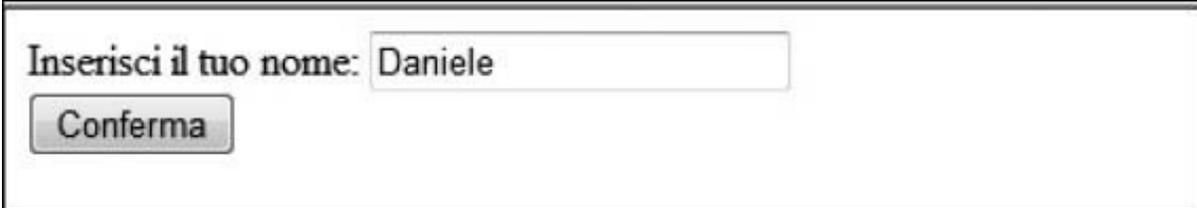
The image shows a browser window displaying a simple web form. On the left, the text "Inserisci il tuo nome:" is followed by a text input field containing the name "Daniele". Below the input field is a button labeled "Conferma". The entire form is enclosed in a thin black border.

Figura 16.2 – La nostra prima pagina.

Quando premiamo sul bottone di conferma, in automatico, viene fatto un invio della form. Quest'azione prende il nome di **PostBack**, perché la form viene nuovamente inviata a sé stessa. Attraverso questa operazione, ASP.NET riceve nuovamente la pagina e capisce che c'è da intercettare l'evento di click, richiamando l'event handler specificato nel markup definito attraverso l'attributo `onClick`, come nell'[esempio 16.3](#).

Esempio 16.3

```
public void ShowFirstName(object sender, EventArgs e)
{
  //... codice
}
```

Come si può notare anche nella [Figura 16.2](#), la `TextBox` ha mantenuto lo stato senza che noi dovessimo effettivamente gestirlo manualmente. Questa caratteristica fa parte dell'infrastruttura di ASP.NET e si basa su un concetto noto come **ViewState**.

Il `ViewState` consente alle pagine di mantenere lo stato dei controlli attraverso i vari `PostBack` e concorre al supporto degli eventi all'interno del sistema. Questi due concetti sono essenziali al funzionamento di una pagina ASP.NET. Nella [Tabella 16.2](#) sono visibili gli eventi della **Web Form**, riportati in ordine di invocazione.

Tabella 16.2 – Gli eventi dalla Web Form, in ordine di invocazione

Evento	Descrizione
<code>PreInit</code>	Si verifica prima dell'inizializzazione della pagina. Serve per impostare master page e theme, caratteristiche che controllano l'aspetto generale delle applicazioni ASP.NET Web Forms.
<code>Init</code>	Si verifica all'inizializzazione della classe rappresentata dalla pagina ed è, di fatto, il primo vero evento.
<code>InitComplete</code>	Si verifica subito dopo l'evento <code>Init</code> .
<code>LoadState</code>	Segna il caricamento dello stato della pagina e dei controlli dal <code>ViewState</code> .
<code>PreLoad</code>	Si verifica prima del caricamento della pagina.
<code>Load</code>	Si verifica al caricamento della pagina, successivamente all'inizializzazione.
<code>LoadComplete</code>	Si verifica subito dopo l'evento <code>Load</code> .
<code>PreRender</code>	Si verifica subito prima del rendering della pagina.
<code>SaveState</code>	Salva lo stato dei controlli all'interno del <code>ViewState</code> .
<code>Render</code>	Si verifica al rendering della pagina e segna la generazione del codice (X)HTML associato.
<code>UnLoad</code>	Si verifica allo scaricamento dell'istanza della pagina. Non corrisponde al <code>Dispose</code> della stessa, perché quest'ultimo è gestito dal Garbage Collector.

È importante sottolineare che il rendering della pagina avviene quando la stessa arriva nello stato di "Render". Non si tratta tanto di un vero e proprio evento come gli altri, quanto di uno stato specifico, che viene sfruttato per far

generare l'output ai controlli. Fino a quel momento, qualsiasi scrittura diretta sullo stream di risposta avrà l'effetto di inserire del testo precedente a quello che sarà generato poi dai controlli.

Interagire con la pagina

Attraverso l'uso dei controlli, diventa possibile racchiudere il flusso logico di una data operazione all'interno di una singola pagina: questo consente di incorporare e tenere uniti tutti i pezzi di un'ipotetica serie di passaggi. Possiamo comporre facilmente il layout, nascondendo a piacimento gli oggetti, attraverso l'uso della proprietà `visible`, che è disponibile su tutti i controlli (perché derivano tutti dalla classe base `Control`, pagina inclusa). Inoltre, ASP.NET è dotato di un controllo specifico, chiamato `wizard`, che consente di strutturare rapidamente una serie di passaggi da far compiere all'utente. In generale, è utile soffermarsi per un attimo sui **controlli di tipo container**. Si tratta di un particolare tipo di controlli che, a differenza degli altri, è in grado di funzionare da contenitore. Questo significa che controlli di questo tipo possono raggruppare altri controlli e, se implementano l'interfaccia `INamingContainer`, assicurano agli stessi che il loro ID sarà generato univocamente. Quando un nostro controllo è contenuto all'interno di un controllo di questo tipo, lo stesso non è direttamente accessibile utilizzando l'ID, come faremmo con un controllo contenuto direttamente nella pagina, ma va ricercato all'interno di quello che è chiamato **albero dei controlli**, che contiene tutta la struttura della pagina, usando il metodo `FindControl` sul contenitore.

Quando dobbiamo semplicemente racchiudere controlli e non vogliamo influenzarne così tanto il comportamento, possiamo optare per `Placeholder`, che offre il vantaggio di raggruppare e nascondere (o visualizzare) un gruppo di controlli, senza agire singolarmente sugli stessi. L'[esempio 16.3](#) può quindi essere adattato per mostrare il risultato dopo il click e nascondere, contestualmente, la form di immissione dati. Il codice necessario si trova nell'[esempio 16.4](#).

Esempio 16.4 - Markup

```
<form runat="server">
  <asp:Placeholder id="EntryForm" runat="server">
    Inserisci il tuo nome:
```

```
<asp:TextBox id="FirstName" runat="server" /> <br />
<asp:Button id="ConfirmButton" runat="server" Text="Conferma"
    OnClick="ShowFirstName" />
</asp:Placeholder>

<asp:Placeholder id="Results" runat="server" Visible="false">
    Il tuo nome è
    <asp:Literal id="SelectedFirstName" runat="server" />
</asp:Placeholder>
</form>
```

Esempio 16.4 - C#

```
public void ShowFirstName(object sender, EventArgs e)
{
    EntryForm.Visible = false;
    Results.Visible = true;
    SelectedFirstName.Text = FirstName.Text;
}
```

Una volta lanciato a video, otterremo un risultato simile a quello della [Figura 16.3](#).

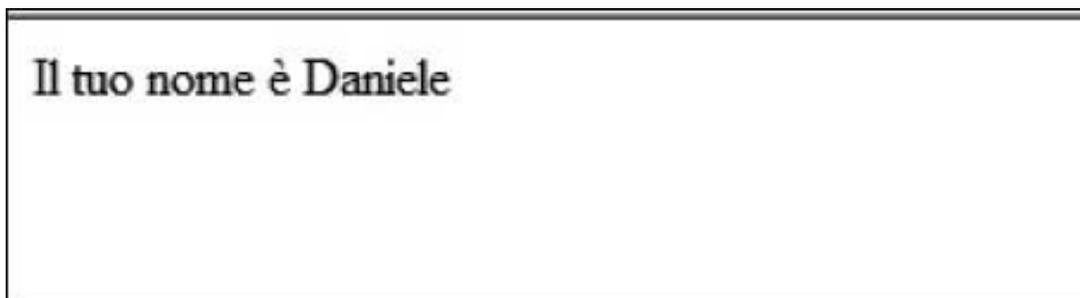


Figura 16.3 – La form prima e dopo il submit.

A questo punto si pone un problema: come fare in modo che la form non possa essere inviata se i campi non sono compilati correttamente? Per questo scenario, ASP.NET dispone di un insieme di controlli che prendono il nome di validator control.

[Validazione delle form](#)

Una delle funzionalità più utilizzate dagli sviluppatori web è la validazione dei dati contenuti all'interno di una form. ASP.NET supporta nativamente questo

scenario grazie ai già citati **validator control**. Si tratta di una famiglia di controlli che vengono associati ai controlli di inserimento e sono in grado di effettuare una convalida tanto lato client quanto lato server. L'uso in tal senso è visibile nell'[esempio 16.5](#).

Esempio 16.5

```
Inserisci il tuo nome:  
<asp:TextBox id="FirstName" runat="server" />  
<asp:RequiredFieldValidator runat="server"  
  ControlToValidate="FirstName"  
  ErrorMessage="*" />
```

La chiave di tutto risiede nella proprietà `ControlToValidate`, che accetta l'ID del controllo da convalidare, sul quale viene applicata la convalida. La proprietà `ErrorMessage`, invece, consente di specificare un messaggio di errore. Come il nome stesso suggerisce, questo controllo verifica che il campo sia effettivamente riempito. Per le altre possibili tipologie di validazione, ASP.NET fornisce adeguati controlli, che abbiamo schematizzato nella [Tabella 16.3](#).

Tabella 16.3 – I controlli di validazione di ASP.NET

Controllo	Descrizione
<code>RequiredFieldValidator</code>	Effettua il controllo più semplice: verifica che ci sia del testo.
<code>RangeValidator</code>	Verifica che il valore del controllo a cui è associato sia compreso tra i valori delle proprietà <code>MinimumValue</code> e <code>MaximumValue</code> per il tipo specificato attraverso <code>Type</code> .
<code>CompareValidator</code>	Può comparare il valore di due controlli, attraverso la proprietà <code>ControlToCompare</code> , oppure rispetto a un valore fisso, con <code>ValueToCompare</code> . La proprietà <code>Type</code> specifica il tipo di valore della convalida, mentre <code>operator</code> la tipologia di operatore da utilizzare.
<code>RegularExpressionValidator</code>	Sfrutta una regular expression, specificata nella proprietà <code>ValidationExpression</code> , per effettuare i controlli di convalida.
<code>ValidationSummary</code>	Mostra un riepilogo dei validator che non hanno passato la convalida, leggendo la proprietà <code>ErrorMessage</code> , se presente, altrimenti sfruttando <code>Text</code> . La proprietà <code>DisplayMode</code> consente di scegliere il tipo di formattazione da dare all'elenco degli errori.

Per essere certi che la convalida sia avvenuta, lato server, dobbiamo richiamare la proprietà `IsValid` della classe `Page`: se dimenticassimo di farlo, l'effetto sarà di non verificare la convalida, dato che quella client-side può essere aggirata disattivando il supporto a JavaScript.

Con la release 4.5 è stato introdotto il supporto alla unobtrusive validation, migliorando l'output generato e utilizzando i più recenti standard. Per attivare questa funzionalità, occorre agire sul `web.config`, aggiungendo il codice visibile nell'[esempio 16.6](#).

Esempio 16.6

```
<add name="ValidationSettings:UnobtrusiveValidationMode" value="WebForms" />
```

Infine, è possibile creare validazioni custom, utilizzando il controllo `CustomValidator`, oppure implementando un proprio controllo derivato dalla classe `BaseValidator`, che è in comune a tutti questi controlli.

[Mantenere il layout con le master page](#)

Nello sviluppo di applicazioni web è essenziale che tutte le pagine mantengano un layout comune. A tal proposito, ASP.NET introduce i concetti di master page e content page.

Una **master page** rappresenta una sorta di template che, oltre a contenuti dinamici e statici, contiene alcune aree rappresentate da altrettanti controlli di tipo `ContentPlaceHolder`. Questi saranno riempiti con i contenuti definiti nelle varie sezioni delle pagine di contenuto associate, che sono chiamate **content page**.

Una master page è un file con estensione `.master` e ha una sintassi del tutto analoga a quella di una normale pagina ma la direttiva `@Page` è sostituita con la direttiva `@Master`. Una content page è una normale Web Form, che contiene unicamente controlli di tipo `Content` e alla quale è associata una master page. Inoltre, per ogni controllo di tipo `Content` della pagina di contenuto, deve esistere un controllo di tipo `ContentPlaceHolder` nella master page. Ciascun controllo `Content` include il contenuto effettivo per ogni placeholder presente nella master page. La corrispondenza tra i due tipi di controllo si basa sul valore delle proprietà `ID` e `ContentPlaceholderID`.

Per esempio, per rappresentare la tipica struttura di un sito, con 3 aree (centrale, destra e sinistra) che possono essere specializzate nelle content page, dovremo usare una master page come quella nell'[esempio 16.7](#).

Non è sempre obbligatorio sovrascrivere i placeholder. Per questo motivo, nella pagina dell'[esempio 16.8](#) vengono soltanto definite la parte centrale e quella di destra. Possiamo notare l'uso dell'attributo `MasterPageFile` sulla direttiva `@Page`, che indica effettivamente il percorso virtuale del file `.master`.

Esempio 16.7

```
<%@ Master Language="C#" %>
...
<form runat="server">
  <div class="left">
    <asp:ContentPlaceHolder ID="Left" runat="server">
      <div>Sinistra</div>
    </asp:ContentPlaceHolder>
  </div>

  <div class="body">
    <asp:ContentPlaceHolder ID="Body" runat="server" />
  </div>

  <div class="right">
    <asp:ContentPlaceHolder ID="Right" runat="server">
      <div>Destra</div>
    </asp:ContentPlaceHolder>
  </div>
</form>
```

Esempio 16.8

```
<%@ Page Language="C#" MasterPageFile="Site.master" %>
<asp:Content ID="Body" ContentPlaceHolderID="Body" runat="server">
  <div>Contenuto locale della pagina</div>
</asp:Content>

<asp:Content ID="Right" ContentPlaceHolderID="Right" runat="server">
  <div>Parte destra personalizzata</div>
</asp:Content>
```

Una volta eseguita questa pagina all'interno del browser, otterremo un effetto simile a quello che possiamo vedere nella [Figura 16.4](#).

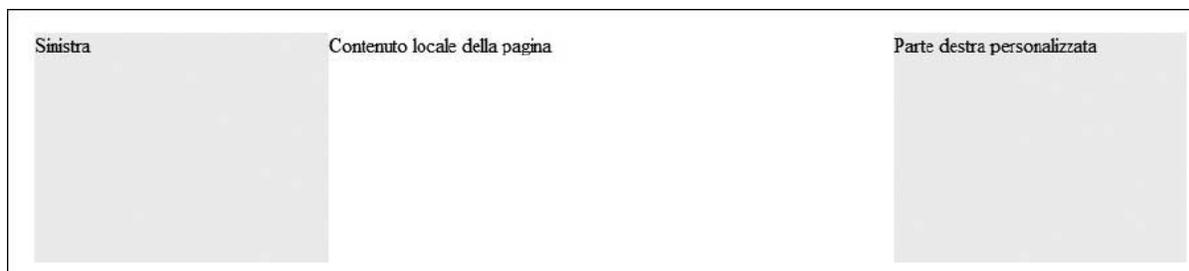


Figura 16.4 – Ecco come appare una content page dopo che la master page è stata applicata.

L'uso di questa tecnica è molto potente e consente, tra l'altro, di annidare anche master page tra loro.

Maggiori informazioni sono disponibili all'indirizzo <http://aspit.co/agy>.

È possibile assegnare una master page anche da codice, programmaticamente, all'interno dell'evento `Pre_Init` mostrato in precedenza.

Visualizzare dati: il data binding

Al giorno d'oggi, i dati in un'applicazione web vengono recuperati dai contesti più disparati, che possono essere un semplice file di testo, un documento XML, un web service o un database relazionale. Per facilitare il recupero dei dati e la loro presentazione all'interno delle applicazioni web mediante un'architettura che permetta di astrarre dal tipo di fonte dati, ASP.NET introduce il concetto di **data binding**. Da un punto di vista tecnico, il data binding è semplicemente l'associazione di una fonte dati a un controllo, garantita dalla possibilità della pagina di distinguere tra creazione e caricamento dei dati nei controlli.

I controlli che supportano la visualizzazione dei dati vengono chiamati **data bound control** o, più semplicemente, data control. Spesso si confonde il data binding con l'uso della proprietà `DataSource`, che i controlli data bound offrono. Questa proprietà accetta un oggetto di tipo `IList`, `IEnumerable` o `ICollection` (e, ovviamente, qualsiasi tipo che implementi una di queste tre interfacce) e, quindi, si presta a garantire la possibilità di essere utilizzata con qualsiasi fonte dati, che non sia esclusivamente un database.

Quello che avviene dietro le quinte, dopo aver associato la sorgente, è molto semplice:

- ❑ viene valutato il contenuto della proprietà `DataSource`;
- ❑ se vi sono elementi all'interno della sorgente, viene effettuato un ciclo che li mostra a video, a seconda della logica che implementa il controllo a cui la sorgente è associata.

Questo meccanismo è del tutto trasparente, perché non dobbiamo fare altro che richiamare il metodo `DataBind` sul controllo al quale abbiamo associato la sorgente. Questo metodo è presente direttamente sulla classe `Control` del namespace `System.Web.UI`, da cui tutti i controlli derivano.

[I list control](#)

Anche se possiamo applicare il data binding a qualsiasi proprietà di un controllo, generalmente si preferisce sfruttare i data control, che spesso sono dotati di template. Ci sono controlli speciali però, come `DropDownList` o `RadioButtonList` che, pur appartenendo a questa famiglia, hanno un layout prefissato. Generalmente, per semplicità, questi vengono definiti list control (perché ripetono liste). La fase di data binding resta però identica. Un modello di list control si può vedere nell'[esempio 16.9](#).

Esempio 16.9

```
<asp:DropDownList ID="DropDownList1"
  DataTextField="CompanyName"
  DataTextFormatString="- {0}"
  DataValueField="CustomerID"
  runat="server" />
```

Esempio 16.9 - C#

```
DropDownList1.DataSource = customers;
DropDownList1.DataBind();
```

Da notare l'uso delle proprietà `DataTextField` e `DataTextFormatString`, che indicano rispettivamente la proprietà da utilizzare per visualizzare e per formattare il dato, mentre `DataValueField` serve a indicare qual è il valore da associare al controllo. Nel codice, invece, possiamo notare come venga assegnata, attraverso la proprietà `DataSource`, la sorgente (che è prelevata grazie a Entity Framework e salvata in una collection locale) e come questa venga effettivamente interrogata all'invocare del metodo `DataBind`. L'esecuzione della pagina è riportata nell'immagine 16.5.

- Alfreds Futterkiste
- Alfreds Futterkiste
- Ana Trujillo Emparedados y helados
- Antonio Moreno Taquería
- Around the Horn
- Berglunds snabbköp
- Blauer See Delikatessen
- Blondesddsl père et fils
- Bólido Comidas preparadas
- Bon app'
- Bottom-Dollar Markets
- B's Beverages
- Cactus Comidas para llevar
- Centro comercial Moctezuma
- Chop-suey Chinese
- Comércio Mineiro
- Consolidated Holdings
- Die Wandernde Kuh
- Drachenblut Delikatessen
- Du monde entier
- Eastern Connection
- Ernst Handel
- Familia Arquibaldo
- FISSA Fabrica Inter. Salchichas S.A.
- Folies gourmandes
- Folk och fä HB
- France restauration
- Franchi S.p.A.
- Frankenversand
- Furia Bacalhau e Frutos do Mar
- Galería del gastrónomo

Figura 16.5 – La DropDownList con la sorgente dati associata.

Anche in presenza di dati prelevati da fonti dati diverse, tutto il codice e il markup visto finora non cambieranno: è questo il punto di forza che sta alla base di questa tecnica.

Utilizzare i template

ASP.NET fornisce diversi tipi di controlli con supporto per i template. Rispetto a quanto offerto, vale la pena citare i seguenti:

- Repeater**: è un controllo molto semplice che, dato un template, lo ripete per tutti gli elementi della sorgente dati;
- GridView**: mostra in forma tabellare la sorgente dati, permettendone la paginazione e l'ordinamento, e la modifica delle righe;
- DetailsView**: mostra in forma tabellare una singola riga recuperata dalla sorgente dati, permettendone l'inserimento e la modifica;
- FormView**: simile al **DetailsView**, permette un layout personalizzato per la rappresentazione dei dati;
- ListView**: consente la visualizzazione con un template personalizzato (sono forniti alcuni template con Visual Studio, supportando paginazione, ordinamento, modifica e inserimento, con maggiore estendibilità rispetto ai controlli esistenti).

La scelta di questi controlli stabilisce le modalità di visualizzazione, inserimento e modifica che vogliamo utilizzare all'interno della pagina. Dal punto di vista delle funzionalità, invece, consentono di sfruttare le medesime caratteristiche, per cui spesso ci sono punti in comune tra controlli diversi, con ovvie differenze dettate dalla diversa resa grafica. In maniera molto semplificata, **Repeater** viene utilizzato quando vogliamo semplicemente mostrare dei dati. **ListView**, invece, ci garantisce la massima flessibilità, **GridView** ci offre una rappresentazione in formato di griglia, e **DetailsView** e **FormView**, invece, la possibilità di agire su un solo elemento alla volta.

I template sono gestiti attraverso una classe di tipo `ITemplate`, un'interfaccia particolare che viene poi implementata da una classe generata al volo attraverso il page parser. Generalmente, questi controlli sono dotati di template specifici per i diversi stati. Per esempio, `ItemTemplate` è il template per il singolo elemento, mentre invece `HeaderTemplate` rappresenta l'intestazione.

Per poter comporre il template e posizionare il contenuto della sorgente dati nei template, dobbiamo utilizzare l'istruzione di binding `<%# ... %>`. Si tratta di una direttiva particolare, che il page parser riconosce e che fa sì che ciò che è contenuto venga invocato insieme all'evento `DataBinding`, che si verifica quando il controllo scatena il data binding. Questi controlli offrono accesso ai dati attraverso l'interfaccia `IDataItemContainer`, che noi spesso utilizziamo grazie alla proprietà `Container` del template, come possiamo vedere nell'[esempio 16.10](#).

Esempio 16.10

```
<asp:Repeater id="CustomerView" runat="server">
  <ItemTemplate>
    <%#Container.DataItem%><br />
  </ItemTemplate>
</asp:Repeater>
```

Il codice precedente rappresenta a video un elenco di elementi. Dato che non è specificato diversamente, viene invocato il metodo `ToString` che, in questo caso, produce la visualizzazione del nome della classe. Per poter accedere alle proprietà, ASP.NET Web Form consente di specificare il tipo messo in binding, senza più dover fare il casting come richiesto nelle versioni precedenti. Il codice necessario a implementare questo scenario è visualizzato nell'[esempio 16.11](#), dove la proprietà `ItemType` indica il tipo da utilizzare nei template di binding.

Esempio 16.11

```
<asp:Repeater runat="server" ItemType="MyModel.Customer">
  <ItemTemplate>
    <%#Item.CustomerName%>
  </ItemTemplate>
</asp:Repeater>
```

Grazie all'uso di questa nuova sintassi, abbiamo acquisito il vantaggio di poter sfruttare l'IntelliSense, evitando di commettere errori e senza la necessità di utilizzare la reflection, come avviene con le versioni precedenti. Grazie alle novità introdotte con questa release, anche i controlli legati alla modifica dei dati hanno la possibilità di sfruttare nuove funzionalità che semplificano gli scenari più diffusi. Per prima cosa, i controlli data source, introdotti con le precedenti versioni, sono deprecati: questo non vuol dire che non possano essere utilizzati, ma significa che ora viene suggerito un nuovo approccio, che è una variante di quanto già abbiamo visto per il binding. Il codice dell'[esempio 16.12](#) mostra come fare.

Esempio 16.12 - griglia.aspx

```
<asp:GridView runat="server"
  SelectMethod="GetCustomers" ItemType="MyModel.Customer">
  ...
</asp:GridView>
```

Esempio 16.12 - griglia.cs.aspx

```
public IQueryable<Customer> GetCustomers([QueryString("n")] string name)
{
    var customers = db.Customers;

    if (!string.IsNullOrEmpty(name)) {
        customers = customers.Where(f => f.Name, Contains(name));
    }

    return customers;
}
```

In particolare, stiamo usando Entity Framework all'interno del codice C#, per farci restituire i clienti. Grazie al fatto che i metodi qui riportati restituiscono un tipo `IQueryable<T>`, il risultato che otteniamo è che il controllo ricostruirà i criteri di ricerca sotto forma di lambda, così che Entity Framework possa produrre l'esatto risultato. In altre parole, potremo implementare paginazioni, ricerche o ordinamenti senza dover fare molto altro e con la certezza che la query che verrà inviata al database sarà creata nella maniera migliore possibile, con i dati effettivamente visualizzati dal controllo come risultato dell'esecuzione della query stessa.

Possiamo vedere la griglia creata nella [Figura 16.6](#).

	<u>CustomerID</u>	<u>CompanyName</u>	<u>ContactName</u>	<u>ContactTitle</u>	<u>Address</u>	<u>City</u>
Edit Delete	ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representative	Obere Str. 57	Berlin
Update Cancel	ANATR	Ana Trujillo Emparedados	Ana Trujillo	Owner	Avda. de la Constitución 2	México D.F.
Edit Delete	ANTON	Antonio Moreno Taqueria	Antonio Moreno	Owner	Mataderos 2312	México D.F.
Edit Delete	AROUT	Around the Horn	Thomas Hardy	Sales Representative	120 Hanover Sq.	London
Edit Delete	BERGS	Berglunds snabbköp	Christina Berglund	Order Administrator	Berguvsvägen 8	Luleå
Edit Delete	BLAUS	Blauer See Delikatessen	Hanna Moos	Sales Representative	Forsterstr. 57	Mannheim
Edit Delete	BLONP	Blondesddsl père et fils	Frédérique Citeaux	Marketing Manager	24, place Kléber	Strasbourg
Edit Delete	BOLID	Bólido Comidas preparadas	Martin Sommer	Owner	C/ Araquil, 67	Madrid
Edit Delete	BONAP	Bon app'	Laurence Lebhan	Owner	12, rue des Bouchers	Marseille
Edit Delete	BOTTM	Bottom-Dollar Markets	Elizabeth Lincoln	Accounting Manager	23 Tsawassen Blvd.	Tsawassen

1 2 3 4 5 6 7 8 9 10

Figura 16.6 – La griglia con i dati dei clienti in binding.

L'uso dell'attributo all'interno dei parametri supportati dal metodo è un'altra novità (mutuata da ASP.NET MVC), che prende il nome di model binding. In pratica, stiamo indicando che la sorgente dei dati è in querystring. Il risultato è che se passiamo alla pagina un valore specifico (per esempio, `griglia.aspx?n=Daniele`), ci ritroveremo la proprietà valorizzata. Possiamo recuperare questi valori dalle sorgenti più disparate, come la form, i cookie o la session.

Lo stesso identico approccio è possibile in fase di aggiornamento o inserimento dei dati. Per esempio, possiamo specificare il metodo che si occuperà di eseguire l'aggiornamento, come nell'[esempio 16.13](#).

Esempio 16.13 - Griglia.aspx

```
<asp:GridView runat="server"
  UpdateMethod="UpdateCustomers" ItemType="MyModel.Customer">
  ...
</asp:GridView>
```

Esempio 16.13 - Griglia.cs.aspx

```
public void UpdateCustomer(Customer c)
{
  ...
}
```

La cosa interessante di questo approccio è che riceveremo, in automatico, i dati caricati all'interno del tipo specificato per la griglia: questo vuol dire che non dovremo preoccuparci di conversioni o altri aspetti e che potremo

semplicemente aggiornare i dati – cosa che, nel caso di Entity Framework, è ancora più semplice fare. Poi basterà fare l’attach al contesto dell’entità e modificarne i dati. All’interno del codice disponibile con il libro è presente un esempio completo, che mostra come si possa implementare facilmente questo scenario con le Web Form in ASP.NET 4.8.

Creare URL per la SEO

ASP.NET supporta nativamente una funzionalità che prende il nome di **URL routing** e che consente di rigirare le richieste su una pagina, a fronte di un URL più lungo.

La SEO (Search Engine Optimization) è una tecnica volta a ottimizzare l’indicizzazione di un sito all’interno dei motori di ricerca. In applicazioni dal contenuto dinamico, è frequente l’uso di parametri inseriti in querystring, per caricare informazioni e generare pagine per ogni specifica richiesta. Questo approccio non è il migliore da adottare, perché non contiene informazioni utili nell’URL. Queste informazioni, come una descrizione breve, possono essere utili tanto ai motori di ricerca quanto all’utente che, da un URL, può capire meglio il contenuto di un indirizzo.

Per sfruttare l’URL routing in ASP.NET, possiamo registrare gli URL da gestire nel `global.asax`, come è visibile nell’[esempio 16.14](#).

Esempio 16.14 – C#

```
public void Application_Start(object sender, EventArgs e)
{
    RouteTable.Routes.Add("ProductsRoute",
        new Route("products/{ProductID}/{ProductName}",
            new PageRouteHandler("~/products.aspx")));
}
```

La route definita rende possibile il fatto che tutte le chiamate effettuate a URL del tipo “products/15/ Libro-ASP.NET”, vengano in realtà rigirate alla pagina specificata, cioè “products.aspx”. All’interno di questa pagina, potremo accedere al valore di uno o più parametri, come mostrato nell’[esempio 16.15](#), tanto da markup, usando una sintassi particolare (che prende il nome di expression builder), quanto da codice.

Esempio 16.15

```
<asp:Literal runat="server" Text="<%%$ RouteValue: ProductName %>" />
```

Esempio 16.15 - C#

```
string productName = Page.RouteData["ProductName"].ToString();
```

Inoltre, possiamo creare un link che sfrutti la route, utilizzando il codice dell'[esempio 16.16](#). Utilizzando questa tecnica, al variare del percorso, in automatico, i link seguiranno la stessa strada, evitandoci di doverli definire manualmente.

Esempio 16.16

```
<asp:HyperLink  
    NavigateUrl="<%%$ RouteUrl: RouteName=ProductsRoute,ProductID=15,  
                ProductName = Libro-ASP.NET %>"  
    runat="server">Libro ASP.NET con ID=15</asp:HyperLink>
```

Questo motore supporta diverse caratteristiche avanzate, come vincoli sui parametri o valori di default per gli stessi.

[Gestione delle aree protette](#)

ASP.NET ha sempre fornito un supporto specifico per aree protette, che però negli ultimi anni si è molto evoluto.

Per quanto riguarda la protezione, sono supportati i due scenari più diffusi, cioè l'autenticazione integrata di Windows e quella attraverso una form. Nel primo caso a fare tutto è il web server, nel secondo dobbiamo definire noi del codice.

Per limitare questa necessità, ASP.NET ha inizialmente introdotto un meccanismo noto come **Membership API**, che consente di definire un provider che fa tutto il lavoro. Per quanto riguarda i ruoli, invece, avevamo a disposizione le **Roles API**. In entrambi i casi si tratta di funzionalità che si basano sul Provider Model Design Pattern, che prevede la definizione di un

provider che implementa concretamente la strategia e di una serie di provider, che possono essere definiti nel `web.config`. I provider, basati su una classe astratta in comune, possono essere scambiati tra loro, modificando il funzionamento interno di queste due funzionalità. ASP.NET supporta inoltre una serie di controlli, chiamati **security control**, che offrono un supporto nativo alle funzionalità di login, creazione utente e recupero password.

È possibile approfondire ulteriormente la questione, che qui richiederebbe una trattazione molto ampia, su <http://aspit.co/agz>.

Di recente, per riflettere i cambiamenti legati a questi ambiti, è stato introdotto un nuovo meccanismo, noto come **ASP.NET Identity**. Più moderno e facilmente estendibile, ASP.NET Identity viene introdotto già con i template predefiniti in fase di creazione di un nuovo progetto e supporta scenari molto più avanzati, come la definizione di un database con schema libero (sfruttando Entity Framework o scrivendo provider specifici), il supporto a login attraverso i social network (Twitter, Facebook, Google e Microsoft Account, senza scrivere codice), l'autenticazione a due fattori e molto altro ancora. ASP.NET Identity può essere approfondito su <http://aspit.co/aqo>.

ASP.NET MVC e ASP.NET Core MVC

ASP.NET MVC è l'alternativa ad ASP.NET Web Forms, che consente di implementare il pattern MVC. Nel caso di ASP.NET Core, è anche l'unica possibilità concreta (per onor del vero, esiste un altro toolkit, denominato ASP.NET Core Web Pages, che però non tratteremo, poiché orientato a pagine web più semplici).

Poiché oggi se si sceglie di sfruttare il pattern MVC ha più senso parlare di ASP.NET Core MVC, piuttosto che di ASP.NET MVC, nel capitolo tratteremo di questo argomento: i concetti trattati, al netto di alcune differenze, restano però validi in entrambi i casi, poiché le due implementazioni si assomigliano molto.

Per creare una nuova applicazione di questo tipo, dobbiamo creare un nuovo progetto all'interno di Visual Studio, come possiamo vedere nell'immagine 16.7.

All'interno di un progetto che fa uso di ASP.NET Core MVC, il lavoro è diviso equamente tra **controller**, che integra la logica sotto forma di codice C#, **view**, che rappresenta l'interfaccia (contiene cioè l'HTML) e **model**, che rappresenta i dati che controller e view si scambiano. Secondo questo pattern,

infatti, controller e view non hanno cognizione l'uno dell'altro e sono slegati tra loro, per favorire la testabilità del codice. Una parte interessante di ASP.NET Core MVC è nel fatto che implementa il concetto di *conventions over configuration* (le convenzioni vincono sulla configurazione). Questo è particolarmente visibile quando diamo un'occhiata al meccanismo di attivazione dei controller e dell'associazione della view al controller stesso.

In particolare, un controller è una classe che deriva dalla classe `Controller` e che ha nel suffisso il nome `Controller` e offre dei metodi particolari, chiamati *action*, che restituiscono un tipo che implementa l'interfaccia `ActionResult`. Possiamo vedere un semplice controller nell'[esempio 16.17](#).

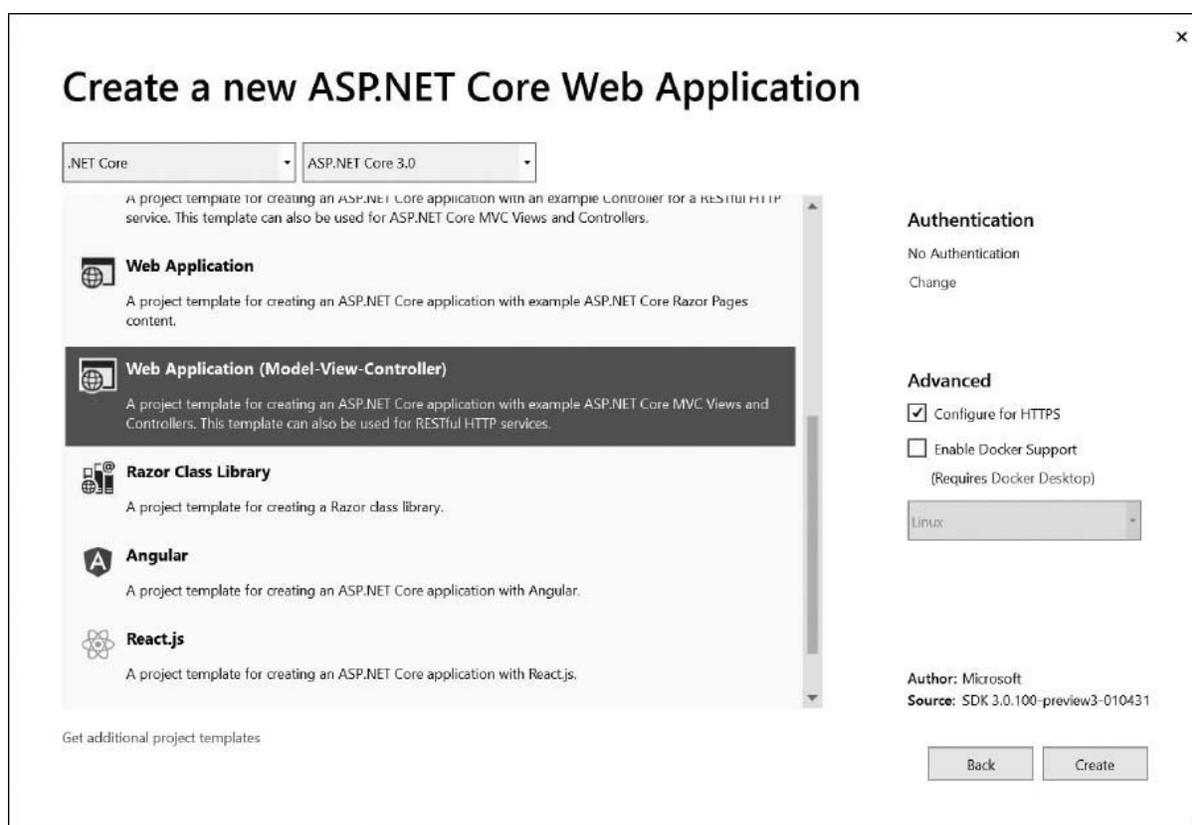


Figura 16.7 – La scelta del progetto con ASP.NET Core MVC.

Esempio 16.17

```
public class HomeController : Controller
{
    ...
    // GET: /Home
    public IActionResult Index()
}
```

```
{
    var customers = db.Customers.OrderBy(f => f.ID).Take(5);
    return View(customers);
}
}
```

Come si richiama questa action? Semplicemente a questo indirizzo: /Home/Index.

Come si può notare, la convenzione indica che il controller è stabilito dal primo pezzo dell'URL, mentre l'action dal secondo. `Index` è anche l'action di default, quindi può essere omessa. Anche ASP.NET MVC utilizza le route di ASP.NET, che possono essere personalizzate agendo sul file `\App_Start\RouteConfig.cs`, mentre ASP.NET Core MVC si basa su ASP.NET Core, che agisce su un nuovo file denominato `Startup.cs`, all'interno del quale vengono definiti i servizi attivabili e anche le route. ASP.NET Core, rispetto ad ASP.NET, è **modulare** e i singoli servizi vanno attivati specificamente: questa caratteristica consente, insieme a investimenti fatti su questo fronte, di avere performance che spesso, in percentuale, sono nell'ordine della doppia cifra, a parità di codice.

A questo punto, abbiamo capito come avere la view: per convenzione, sarà ricercato un file che si chiama `Index.cshtml` sotto il percorso `\Views\Home\`. Anche in questo caso, il nome del controller e dell'action sono utilizzati per comporre dinamicamente il percorso nel quale andare a recuperare la view. Poiché abbiamo passato nel controller un elenco di clienti, la nostra view sarà tipizzata usando il model. Se aggiungiamo una view utilizzando l'apposita voce di Visual Studio, ci verrà proposto in automatico di utilizzare i tipi che abbiamo disponibili, semplificandoci la vita, come viene mostrato nella [Figura 16.8](#).

Add MVC View

View name:

Template:

Model class:

Options:

Create as a partial view

Reference script libraries

Use a layout page:

...

(Leave empty if it is set in a Razor _viewstart file)

Add Cancel

Figura 16.8 – Creare una view tipizzata con ASP.NET Core MVC.

L'[esempio 16.18](#) contiene il codice necessario a mostrare a video i dati recuperati dal database, usando Entity Framework.

Esempio 16.18

```
@model IEnumerable<Customer>
<ul>
@foreach (var item in Model)
{
    <li>@item.Name</li>
}
</ul>
```

Come si può notare, facciamo semplicemente un ciclo e mostriamo a video i dati prelevati dal database: ASP.NET MVC, insomma, non ha il concetto di data binding, perché basta avere una view tipizzata ed estrarre opportunamente le informazioni. La sintassi utilizzata prende il nome di Razor ed è specifica per ASP.NET MVC: non è complessa da imparare, perché ha poche e semplici regole, che sono illustrate su <http://aspit.co/0u>.

Creare form con ASP.NET MVC

Possiamo sfatare subito un mito: ASP.NET MVC e ASP.NET Core MVC vanno benissimo anche per creare maschere di inserimento dati. Anzi, grazie ai wizard integrati all'interno di Visual Studio, diventa semplicissimo, in fase di creazione di un controller, far generare in automatico le action e le view necessarie: basta selezionare il template che fa riferimento a Entity Framework tra le opzioni di scaffolding e poi specificare la classe di modello e quella con il context.

Avremo un controller con codice simile a quello dell'[esempio 16.19](#).

Esempio 16.19

```
public class AdminController : Controller
{
    // GET: /Admin/Edit/5
    public IActionResult Edit(int id = null)
    {
        Customer customer = db.Customers.Find(id);
        if ((customer == null))
        {
            return HttpNotFound();
        }
        return View(customer);
    }

    // POST: /Admin/Edit/5
    [HttpPost()]
    public IActionResult Edit(Customer customer)
    {
        if (ModelState.IsValid)
        {
            db.Entry(customer).State = EntityState.Modified;
            db.SaveChanges();
            return RedirectToAction("Index");
        }

        return View(customer);
    }
}
```

Il codice è interessante perché ci mostra come implementare una form con MVC: il primo metodo, infatti, rappresenta la chiamata che viene fatta per visualizzare la form di inserimento dati, al cui invio verrà invocata la action che risponde solo al metodo `POST`, e che infatti è decorata con l'attributo `HttpPost`. In questo metodo, i dati vengono recuperati, modificati e salvati con Entity Framework: se tutto va a buon fine, saremo inviati alla action di default,

che mostra l'elenco delle informazioni. Ma come si fa a renderizzare una form dentro una view? Occorre implementare un codice simile a quello dell'[esempio 16.20](#).

Esempio 16.20

```
@model Customer

@using (Html.BeginForm())
{
    @Html.ValidationSummary(True)

    @<fieldset>
        <legend>Customer</legend>

        @Html.HiddenFor(model => model.Id)

        <div class="editor-label">
            @Html.LabelFor(model => model.Name)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Name)
            @Html.ValidationMessageFor(model => model.Name)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.City)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.City)
            @Html.ValidationMessageFor(model => model.City)
        </div>
        ...
        <input type="submit" value="Save" />
    </fieldset>
}
```

Possiamo notare l'uso degli helper `LabelFor`, `EditorFor` e `ValidationMessageFor`, che in automatico, andando a lavorare con i tipi delle proprietà del modello, ci danno rispettivamente il titolo, un editor per il tipo che tiene conto di come è fatto e un messaggio di validazione. Queste funzionalità lavorano con le data annotations, introdotte nelle versioni precedenti del .NET Framework e valide anche per .NET Core. Queste sono un modo per annotare e arricchire le nostre classi con informazioni aggiuntive, come la tipologia di campo, se è obbligatorio, oltre che con il modello stesso, che in questo caso è basato su Entity Framework e può avere attributi che includono queste informazioni.

ASP.NET Core MVC, invece, è molto più completo e introduce il concetto dei **Tag Helper**, che consentono di scrivere codice più fedele all'HTML e avere le stesse caratteristiche offerte dagli HTML Helper appena menzionati.

Poiché la trattazione di questi argomenti richiede molto spazio ed esula dallo scopo di questo libro, vi consigliamo di consultare il nostro volume “ASP.NET Core 2 - Guida completa per lo sviluppatore” o di fare riferimento a questo indirizzo: <http://aspit.co/bia>.

In generale, l'intento di questa prima introduzione è più che altro quello di darvi una rapida panoramica su quello che ASP.NET MVC (o ASP.NET Core MVC) consente di fare e permettervi di capire che è molto più semplice di quanto si possa pensare, perché mette HTML e l'HTTP a più stretto contatto con lo sviluppatore, dato che non ci sono artefici costruiti intorno, come nel caso di ASP.NET WebForms. L'esempio allegato al libro contiene una trattazione completa di quanto abbiamo introdotto con questi ultimi esempi, che possono servire come base di partenza per continuare a esplorare ASP.NET.

Per il resto, occorre sottolineare che molte delle nozioni valide per ASP.NET Web Forms (autenticazione e autorizzazione in particolare) valgono anche per ASP.NET MVC, perché, di fatto, entrambi sono basati sullo stesso runtime e hanno accesso alle stesse funzionalità di base.

Con ASP.NET Core, invece, l'unica alternativa possibile è ASP.NET Core MVC: è possibile continuare a utilizzare lo stesso modello di sviluppo, ma cambia il runtime (.NET Core anziché .NET Framework) e le possibilità. In questo modo, saremo in grado, per esempio, di far funzionare le applicazioni basate su ASP.NET MVC anche su Linux e MacOSX, oltre che su Windows. Il nostro consiglio, se state iniziando ora con lo sviluppo web, in un'ottica di investimento in chiave futura, è di scegliere ASP.NET Core MVC – o, in seconda battuta, ASP.NET MVC – rispetto a Web Forms, poiché garantisce un miglior supporto cross-platform e un nuovo runtime ottimizzato per le performance, la scalabilità e il cloud.

Conclusioni

ASP.NET è una tecnologia molto estesa e complessa, per cui questo capitolo ha avuto lo scopo precipuo di presentarne una rapida introduzione. Per approfondimenti specifici, rimandiamo ai libri “ASP.NET 4.5 e ASP.NET MVC 4 in C# e VB – Guida Completa per lo sviluppatore” e “ASP.NET Core 2 - Guida completa per lo sviluppatore”, che potrete trovare in questa stessa collana.

Parte del successo di .NET è da ricercare in ASP.NET, nelle sue varie incarnazioni: la sua forte diffusione ha permesso a tanti sviluppatori di imparare ad apprezzare anche altre tecnologie all'interno del framework, contribuendo a favorirne un ulteriore sviluppo.

Insieme con ASP.NET Web Forms, il supporto per il modello a eventi, il data binding e la gestione delle aree protette rappresentano sicuramente gli argomenti principali da cui partire. D'altro canto, ASP.NET MVC e ASP.NET Core MVC consentono di basarsi al meglio su HTML e HTTP, dandoci il massimo controllo e consentendoci anche la testabilità.

Per ulteriori approfondimenti, vi invitiamo anche a consultare il sito [ASPItalia.com](http://www.aspitalia.com), all'indirizzo <http://www.aspitalia.com/>.

Ora che abbiamo terminato la breve trattazione di ASP.NET, possiamo passare a esaminare una tecnologia che, anche se non strettamente legata al Web, consente di creare una parte essenziale per lo scambio dei dati: i servizi. Nel prossimo capitolo parleremo quindi di applicazioni distribuite e di come strutturare i servizi necessari allo scambio di dati.

Sviluppare servizi web

Nel capitolo precedente abbiamo imparato a realizzare applicazioni web con C# e, nello specifico, attraverso il framework ASP.NET. Abbiamo visto che ci sono due tecnologie che possiamo sfruttare per la realizzazione di pagine dinamiche: WebPages e MVC. Ma il Web non è solo HTML e anzi, negli ultimi anni si sta trasformando sempre più in servizi che espongono API per l'accesso e la manipolazione di dati. **HTTP**, infatti, è un protocollo molto semplice, implementato da qualsiasi dispositivo, framework e linguaggio, e spesso viene sfruttato dalle Single Page Application (SPA), applicativi client sviluppati con framework JavaScript. È perciò interoperabile e dà un controllo completo sulla richiesta, permettendoci di esseri liberi nel suo utilizzo.

Per questo motivo i servizi sono ampiamente sfruttati, soprattutto per la realizzazione di applicazioni mobile, ma sono diffusi anche in ambito enterprise per comunicazioni server to server.

In questo capitolo vogliamo presentarvi la tecnologia di riferimento nell'ambito Microsoft per realizzazione di servizi e in particolare di quelli RESTful. Questi hanno la particolarità di sfruttare appieno il protocollo HTTP impiegando i metodi HTTP come GET, POST, PUT e DELETE (e non solo) rispettivamente per accedere, aggiungere, modificare e cancellare una risorsa rappresentata dal suo URI. Più avanti nel capitolo affronteremo uno standard di nome **OData**, che estende questo concetto definendo delle regole che ci permettono di effettuare query e di navigare tra i dati. Infine, parleremo di come realizzare applicazioni real-time sfruttando i canali bidirezionali di comunicazione e, in particolare, i **WebSocket**.

Partiamo quindi da come realizzare servizi RESTful attraverso ASP.NET.

I servizi RESTful con ASP.NET

WebAPI

Abbiamo visto che ASP.NET è il framework di riferimento per tutto ciò che riguarda il mondo web. Per realizzare servizi usiamo lo stesso motore visto nel capitolo precedente e i concetti di configurazione e di startup sono i medesimi. Con Visual Studio possiamo sfruttare la finestra di dialogo offerta per combinare il run time che vogliamo sfruttare con la versione e, infine, il template dal quale partire. Nella [Figura 17.1](#) è mostrata la selezione del template per un progetto di tipo API.

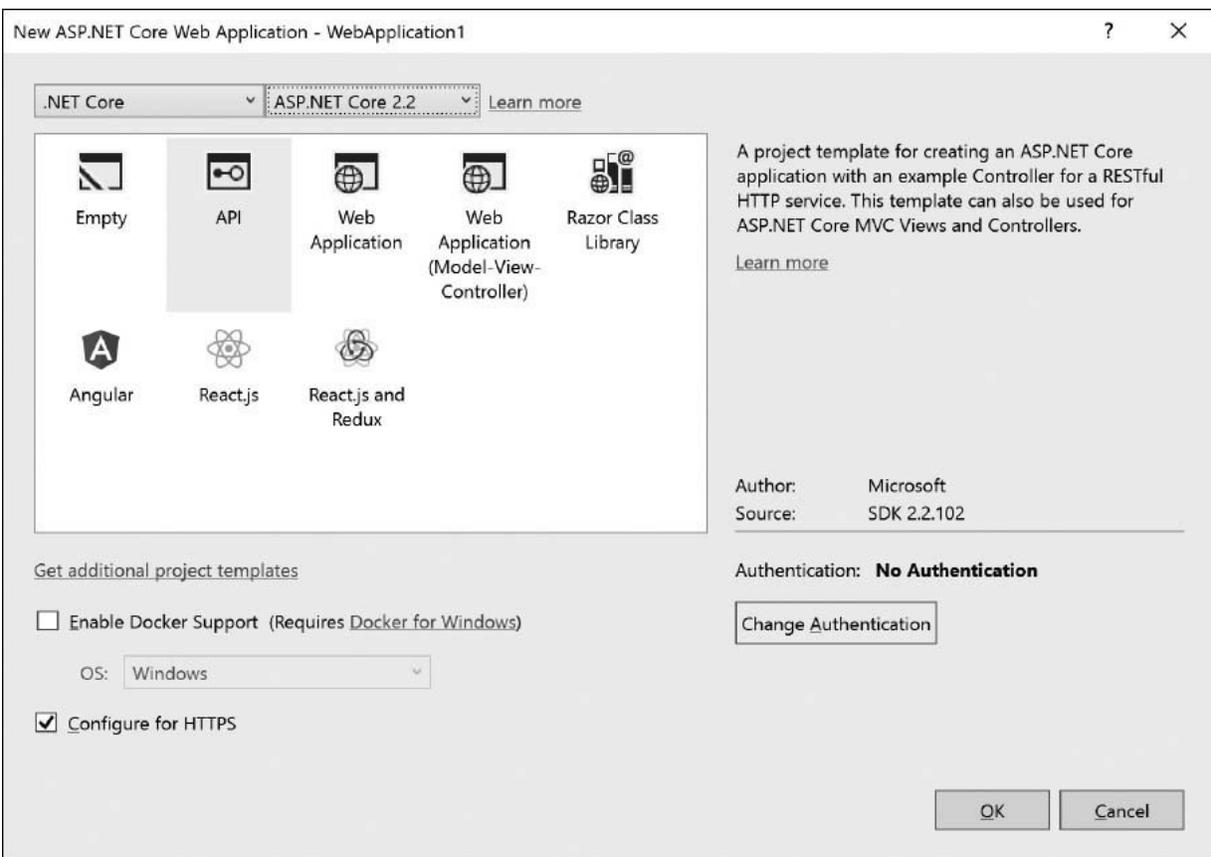


Figura 17.1 – Selezione di un progetto ASP.NET Core 2.2 con un template API.

Il template si distingue da quello Web Application perché non include quelle risorse utili alle pagine. Non è presente una particolare configurazione, perciò possiamo realizzare servizi anche su progetti avviati con un template web o vuoto. Una volta creato il progetto, troviamo una classe di nome `ValuesController` all'interno della cartella `Controllers`. Questa classe è definita come si può vedere nell'[esempio 17.1](#).

Esempio 17.1

```
[Route("api/[controller]")]
[ApiController]
public class ValuesController : ControllerBase
{
    // GET api/values
    [HttpGet]
    public IEnumerable<string> Get()
    {
        return new string[] { "value1", "value2" };
    }

    // GET api/values/5
    [HttpGet("{id}")]
    public string Get(int id)
    {
        return "value";
    }

    // POST api/values
    [HttpPost]
    public void Post([FromBody] string value)
    {
    }

    // PUT api/values/5
    [HttpPut("{id}")]
    public void Put(int id, [FromBody] string value)
    {
    }

    // DELETE api/values/5
    [HttpDelete("{id}")]
    public void Delete(int id)
    {
    }
}
```

Possiamo subito notare che al nome della classe segue il suffisso `Controller` come nel caso di controller ASP.NET per il web. Questo significa che il controller appena definito potrebbe rispondere in base alla

convenzione di routing stabilita a livello generale, quindi all'indirizzo `/values`. Poiché i servizi REST poggiano prima di tutto sull'identificazione della risorsa, è fondamentale avere il controllo dell'indirizzo al quale un controller risponde. Mediante l'attributo `Route` definito a livello di classe specificiamo quindi un percorso alternativo alla convenzione. Con l'uso della parola chiave `[controller]` indichiamo un placeholder che viene sostituito con il nome del controller stesso, perciò il nostro servizio è di fatto raggiungibile all'indirizzo `api/values`. Infine, la classe base è la medesima per i controller dedicati alle pagine web.

Proseguendo con l'analisi dell'[esempio 17.1](#), possiamo notare varie funzioni, aventi nomi `Get`, `Post`, `Put` e `Delete`. Come possiamo immaginare, grazie ai rispettivi attributi che definiscono il percorso di routing parziale, queste funzioni hanno il compito rispettivo di restituire una risorsa, aggiungerla, aggiornarla e cancellarla. Per capire cosa questo significhi, sfruttiamo uno strumento di nome **Fiddler** (<https://aspit.co/bt2>), il quale permette di intercettare tutte le richieste HTTP, anche quelle fatte dai browser, ma anche di comporre richieste e analizzarne il contenuto. Procediamo quindi eseguendo la nostra applicazione web con F5 e apriamo Fiddler. Nella sezione `composer` possiamo scegliere di utilizzare il metodo HTTP `GET` con l'indirizzo di test <http://localhost:50277/api/values>, come nella [Figura 17.2](#).

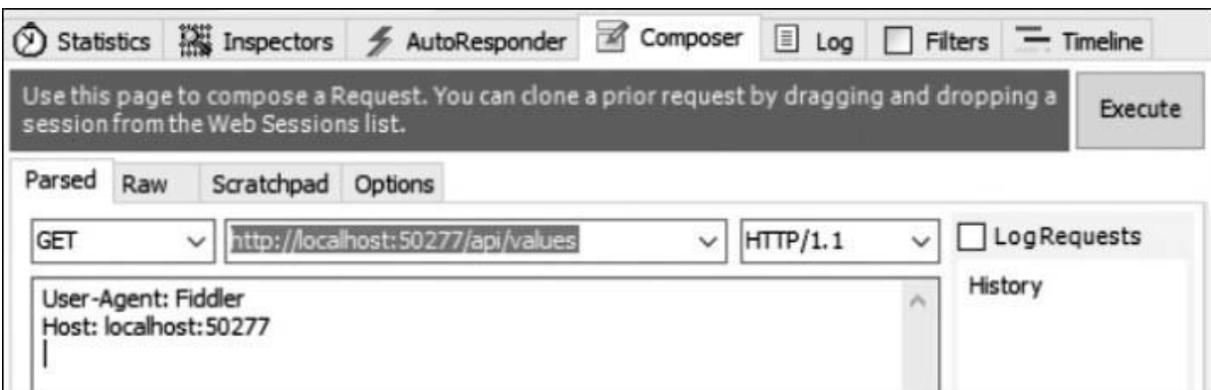


Figura 17.2 – Creazione di una richiesta GET tramite Fiddler.

Premendo il pulsante `Execute` creiamo una richiesta HTTP come quella dell'[esempio 17.2](#).

Esempio 17.2

```
GET http://localhost:50277/api/values HTTP/1.1
User-Agent: Fiddler
Host: localhost:50277
```

La risposta che otteniamo è invece visibile nell'[esempio 17.3](#).

Esempio 17.3

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
Expires: -1
X-Powered-By: ASP.NET
Date: Fri, 1 Feb 2019 14:46:49 GMT
Content-Length: 19

["value1","value2"]
```

Possiamo notare che la risposta ottenuta è serializzata in JSON, un formato molto compatto proveniente da JavaScript e manipolabile ormai da qualsiasi linguaggio. In questo caso quello che otteniamo è un array di stringhe e da questo si evince che la funzione chiamata sulla classe

`ValuesController` è il metodo `Get`, senza parametri. Possiamo effettuare la stessa chiamata inserendo l'indirizzo in un browser, il quale, in modo predefinito, effettua sempre una chiamata GET. Il risultato è la visualizzazione delle stringhe a video.

Se invece eseguiamo la chiamata ad `api/values/5` (o qualsiasi altro numero valido) otteniamo l'invocazione della seconda funzione `Get`, la quale accetta un intero. Questo perché il motore è in grado di distinguere le chiamate a seconda del percorso e dei parametri che passiamo. Il parametro `id` della funzione ha il medesimo nome del parametro indicato nell'attributo `HttpGet`, e viene recuperato dall'indirizzo che segue `api/values/`. Come possiamo facilmente intuire, se cambiamo il metodo HTTP in PUT, POST o DELETE otteniamo l'invocazione delle altre rispettive funzioni. Questo meccanismo è allo stesso tempo semplice, ma potente, poiché sfruttabile da qualsiasi tecnologia, da JavaScript all'interno di una pagina Internet, a un semplice generatore di richieste, come Fiddler.

Quello che fa ASP.NET, quindi, è trasformare le richieste nell'invocazione della rispettiva funzione e tradurre poi la relativa risposta. Trovare il significato e la rispettiva implementazione spetta a noi; l'[esempio 17.1](#), infatti, non esegue alcuna operazione ed è nostro compito invocare query sul database o metodi su servizi applicativi per l'esecuzione materiale della richiesta.

La serializzazione e il model binding

La trasformazione delle richieste e delle risposte è molto potente ed è in grado di gestire anche oggetti complessi. Invece di restituire una stringa, possiamo creare un oggetto più complesso, come nell'[esempio 17.4](#).

Esempio 17.4

```
public Customer Get(int id)
{
    return new Customer
    {
        FirstName = "Pippo",
        LastName = "Pluto",
    };
}
```

Ciò che otteniamo da un'interrogazione al servizio è il JSON visibile nell'[esempio 17.5](#).

Esempio 17.5 - JSON

```
{ "firstName": "Pippo", "lastName": "Pluto" }
```

La serializzazione è affidata a Json.NET, un potente motore open source che è in grado di trasformare in JSON tutte le classi e i membri pubblici, anche di grafi complessi contenenti proprietà a loro volta di tipo complesso.

Un aspetto interessante del motore di ASP.NET sta nel supporto a onorare l'header HTTP `Accept` cioè il supporto al content negotiation. Con esso, chi esegue la richiesta può chiedere in che formato si aspetta la

risposta, se quest'ultimo è supportato dal servizio. Inserendo quindi un header `Accept: application/json` (o omettendolo) otteniamo quindi una serializzazione in JSON, come abbiamo già visto. Con l'header `Accept: text/xml`, invece, indichiamo che vogliamo la risposta serializzata in XML. In questo caso, viene sfruttata la classe `DataContractSerializer` per produrre l'XML visibile nell'[esempio 17.6](#).

Esempio 17.6

```
<Customer
  xmlns="http://schemas.datacontract.org/2004/07/Capitolo17.Controllers">
  <FirstName>Pippo</FirstName>
  <LastName>Pluto</LastName>
</Customer>
```

Questi due formati sono quelli direttamente supportati da ASP.NET, ma possiamo scrivere dei formater personalizzati per aggiungerne altri.

Il supporto a XML è built-in, ma va abilitato in modo esplicito chiamando la funzione `AddXmlDataContractSerializerFormatters` sul builder ottenuto con `AddMvc` che troviamo nel file `Startup.cs`.

Possiamo sfruttare gli oggetti complessi anche come parametri di input nelle nostre funzioni, come nell'[esempio 17.7](#).

Esempio 17.7

```
public void Post(Customer customer)
{
    // TODO: chiamata al db
}
```

In questo caso, il body della richiesta HTTP deve contenere il cliente serializzato in JSON o XML. Per far sì che ASP.NET sappia che motore usare per la creazione dell'oggetto `customer`, esiste un altro header HTTP che lo standard indica di usare per questi scopi: `Content-Type`. Questo header può assumere i medesimi valori di `Accept` e richiede che il servizio

lo supporti. Comunque, grazie ad ASP.NET, non dobbiamo far null'altro che lavorare ad alto livello specificando che oggetto vogliamo. In questo caso il processo che viene invocato si chiama model binding e segue dinamiche molto flessibili per la trasformazione della richiesta verso l'invocazione della funzione. Per esempio, possiamo inserire parametri aggiuntivi alla funzione, come nell'[esempio 17.8](#).

Esempio 17.8

```
public Customer Get(int id, bool withDetails)
{
    // TODO: chiamata
    return new Customer();
}
```

Il motore è in grado di recuperare il valore dalla querystring ed effettuare la conversione del tipo, permettendoci di invocare l'indirizzo `api/customers/5?withDetails=true`.

Così facendo, rendiamo obbligatorio l'utilizzo del parametro `withDetails`, ma possiamo sfruttare i parametri opzionali o i default value per permettere di invocare la funzione anche senza specificare il valore.

Le action e i metodi HTTP

Le funzioni che abbiamo visto finora hanno un nome uguale ai verbi HTTP, ma questo non è necessario. Ciò che determina il verbo e l'indirizzo con il quale raggiungere la funzione è impostato dagli attributi `HttpGet`, `HttpPost`, `HttpPut`, `HttpDelete`, `HttpHead`, `HttpPatch` e `HttpOptions` per marcare le funzioni che rispondono ai rispettivi metodi HTTP, come nell'[esempio 17.9](#).

Esempio 17.9

```
[HttpGet("{id}")]
public Customer FindCustomer(int id)
{
    return new Customer();
}
```

```
[HttpPatch]
public void UpdateCustomer(Customer customer)
{
    // TODO: implementare
}
```

Qualora non bastasse, con l'attributo `AcceptVerbs` possiamo elencare la lista dei metodi HTTP ai quali la funzione risponde.

Non sempre vogliamo offrire un approccio CRUDQ (Create, Read, Update, Delete, Query) al nostro servizio, ma vogliamo esporre API mirate ad azioni specifiche, come la conferma di un ordine, la generazione di un report e così via. In questi casi possiamo sfruttare al massimo l'attributo che ci consente di specificare l'intero indirizzo relativo con il quale raggiungere l'azione, come nell'[esempio 17.10](#).

Esempio 17.10

```
[HttpPut("/api/customers/confirm/{id}-{code}")]
public void Confirm(int id, string code)
{
    // TODO: implementare
}
```

Nel codice è mostrato come possiamo specificare il percorso assoluto anche ignorando il nome del controllo o cambiando il percorso iniziale api stabilito per ogni servizio, semplicemente ponendo lo / (slash) all'inizio. All'interno dell'indirizzo possiamo inoltre indicare dei segnaposti, identificati dal nome contenuto tra graffe. Essi devono necessariamente essere divisi da un separatore e rappresentano i parametri che la funzione si aspetta. Nell'[esempio 17.10](#) richiediamo un `id` intero seguito dal `code` come stringa. Qualora la richiesta venga fatta con un metodo HTTP sbagliato, viene generata automaticamente una risposta `405 Method Not Allowed`. Se invece l'indirizzo è sbagliato, viene generato un `404 Not Found`. Questi status code sono un altro fattore importante che contraddistingue i servizi RESTful, perché rappresentano l'esito dell'operazione

[Le tipologie di risultato delle action](#)

Finora abbiamo visto semplici definizioni di funzioni, le quali restituiscono `void` o un oggetto, primitivo o complesso che sia. Se l'esecuzione del nostro codice va a buon fine, la risposta generata contiene il risultato serializzato, ma anche l'esito dell'operazione: lo status code. Nell'[esempio 17.3](#) abbiamo potuto notare il 200 OK che indica appunto la corretta elaborazione della richiesta. Una funzione che restituisce `void`, invece, determina un 204 No Content, mentre un'eccezione genera un 500 Internal Server Error, perciò in questo modo possiamo decidere come informare chi fa la richiesta dell'esito. Questo però non è sempre sufficiente, perché la nostra funzione può necessitare di effettuare validazioni o di controllare l'esistenza dell'elemento su cui sta lavorando.

In questi casi dobbiamo cambiare la firma della funzione e far restituire un tipo `ActionResult`. Esistono molteplici classi che implementano questa interfaccia a seconda che vogliamo restituire uno specifico status code, con o senza oggetto da serializzare.

Nell'[esempio 17.11](#) possiamo vedere come sfruttare questa interfaccia per rispondere con lo status code adeguato in base all'esito dell'operazione.

Esempio 17.11

```
public ActionResult<Customer> Get(int id)
{
    Customer customer = FindCustomerOnDatabase(id);
    if (customer == null)
    {
        return NotFound(); // Ritorna NotFoundResult
    }
    return Ok(customer); // Ritorna OkNegotiatedContentResult
    // return customer; // Ritorna OkNegotiatedContentResult
}
```

Dal codice notiamo alcune funzioni speciali fornite dalla classe `ControllerBase`, quali `NotFound` e `Ok` rispettivamente per creare una risposta che identifichi un errore 404 o un 200 con l'oggetto della risposta. Sono disponibili diverse funzioni a seconda dello status code che vogliamo generare, come `Redirect`, `BadRequest` o il più generico `StatusCode`. Da notare il fatto che la funzione ora restituisce un `ActionResult<T>`, una particolare classe di tipo `ActionResult` che dispone di operatori impliciti di conversione. Questo rende possibile l'utilizzo delle funzioni non

generiche, come `NotFound` o la restituzione dell'oggetto senza necessariamente invocare la funzione `ok`. L'uso di questo tipo fa sì che i nostri servizi siano tipizzati e facilita la creazione della documentazione degli stessi, supportata da ASP.NET.

Se invece tutto questo non bastasse all'interno del controller abbiamo accesso alle proprietà `Request` e `Response`, rispettivamente per accedere alle informazioni dirette della richiesta e di risposta.

Esempio 17.12

```
public async Task Get()
{
    // Imposto lo status code
    Response.StatusCode = 200;

    // Imposto il tipo di risposta
    Response.ContentType = "text/plain";

    // Header personalizzato
    Response.GetTypedHeaders().CacheControl = new CacheControlHeaderValue
    {
        MaxAge = TimeSpan.FromMinutes(20)
    };

    // Scrittura nel body
    await Response.WriteAsync("hello");
}
```

Nell'[esempio 17.12](#) possiamo vedere come impostare lo `statusCode` e il `ContentType` tramite le omonime proprietà e con `Headers`, invece, avere accesso al dizionario degli header, consentendoci di personalizzare gli aspetti relativi alla cache, al contenuto, ai cookie o alle informazioni personalizzate. Nell'[esempio 17.12](#) impostiamo una cache di 20 minuti che consente al browser o, in generale, al client di evitare di effettuare ulteriori richieste nell'intervallo da noi indicato. Infine, la funzione scrive direttamente nel body di risposta tramite `writeAsync`, il quale genera quanto è visibile nell'[esempio 17.13](#). Data che essa è asincrona per ottimizzare le prestazioni, la funzione `Get` è stata a sua volta trasformata in asincrono e viene invocata in modo trasparente da ASP.NET.

Esempio 17.13

```
HTTP/1.1 200 OK
Cache-Control: max-age=1200
Content-Length: 10
Content-Type: text/plain; charset=utf-16 Date: Fri, 01 Feb 2019 16:04:58 GMT

hello
```

In alternativa alla funzione `WriteAsync`, la funzione `SendFileAsync` permette di rispondere direttamente con un file, mentre la proprietà `Body` dà accesso allo stream di risposta, permettendoci la massima libertà di scrittura della risposta.

Anche la richiesta può essere letta, in modo molto simile, dalla proprietà `Request` e con essa possiamo accedere in modo grezzo alla richiesta, leggere gli header e i byte del contenuto, come nell'[esempio 17.14](#).

Esempio 17.14

```
[HttpPost]
public async Task<IActionResult> Post()
{
    // Header If-Match
    var tag = Request.GetTypedHeaders().IfMatch?.FirstOrDefault()?.Tag;
    // Leggo il contenuto come testo
    var text = await new StreamReader(Request.Body).ReadToEndAsync();

    return Ok();
}
```

Nell'esempio leggiamo un header e, cosa più interessante, leggiamo il contenuto della richiesta come stringa, interpretando i byte del body come caratteri.

Quanto abbiamo visto finora è sufficiente per esporre un servizio web adeguato ai tempi moderni, ma con un po' di esperienza ci accorgeremo che spesso i servizi sono ripetitivi tra loro e la loro implementazione è molto simile. Lo scaffolding di Visual Studio ci viene in aiuto.

[Lo scaffolding delle API](#)

La realizzazione di un servizio è chiaramente a nostra discrezione. Esso può fornire, come abbiamo visto, operazioni specifiche in base alle logiche di business, oppure offrire un approccio più aperto di lettura e manipolazione

dei dati. Se stiamo utilizzando tutte tecnologie Microsoft, molto probabilmente stiamo sfruttando anche **Entity Framework** per facilitarci in questo genere di operazioni.

Lo scaffolding di Visual Studio è l'ideale in questo genere di situazioni, perché data un'entità ci permette di ottenere automaticamente un servizio con tutte le operazioni di CRUDQ effettuate direttamente su Entity Framework. Di fatto è un generatore di codice, dal quale possiamo partire, eventualmente, per apportare modifiche. Quando da Visual Studio sulla cartella `Controllers` aggiungiamo un nuovo controller, ci viene proposta la finestra della [Figura 17.3](#).

Possiamo vedere evidenziata nella figura la voce che genera le azioni attraverso Entity Framework. Se proseguiamo, ci viene di conseguenza chiesto qual è il contesto di Entity Framework da utilizzare e l'entità che vogliamo esporre, oltre al nome del controller. Il tutto ci viene mostrato con alcune combo box che ci permettono di selezionare oggetti creati precedentemente, come è visibile nella [Figura 17.4](#).

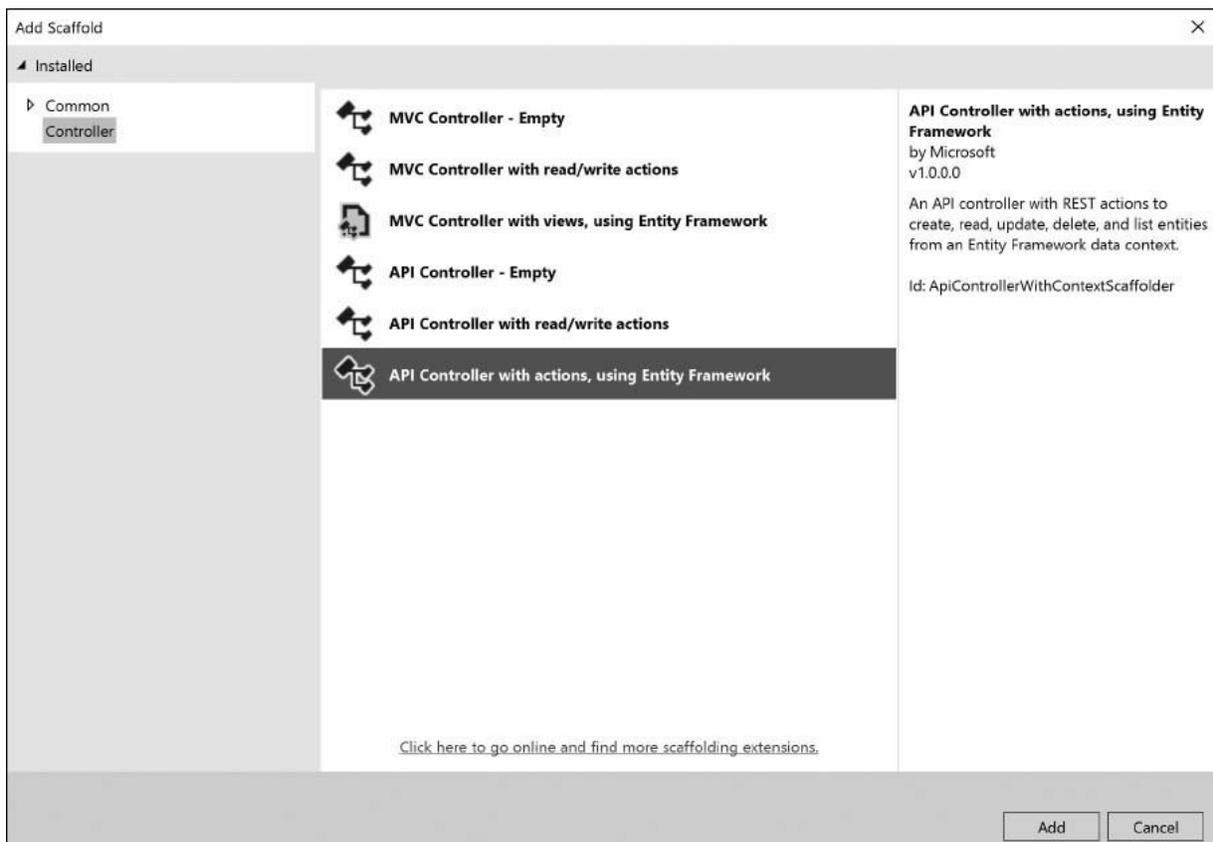


Figura 17.3 – Creazione di un controller API tramite scaffolding in Visual Studio.

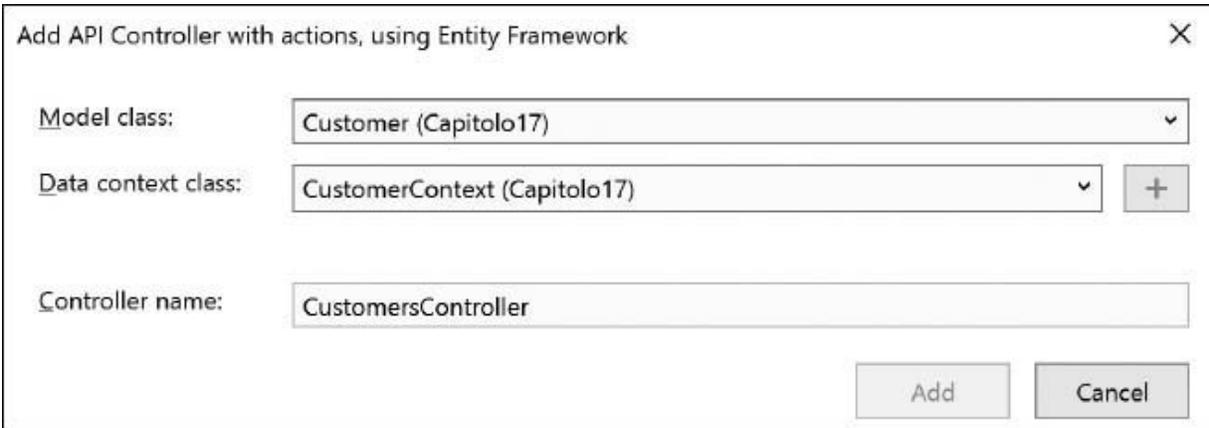


Figura 17.4 – Finestra per la configurazione dello scaffolding tramite Visual Studio.

Confermando la finestra, viene generata una classe `CustomersController`, della quale possiamo vedere un estratto nell'[esempio 17.15](#).

Esempio 17.15

```
[Route("api/[controller]")]
[ApiController]
public class CustomersController : ControllerBase
{
    private readonly CustomerContext _context;

    public CustomersController(CustomerContext context)
    {
        _context = context;
    }

    // GET: api/Customers
    [HttpGet]
    public async Task<ActionResult<IEnumerable<Customer>>> GetCustomers()
    {
        return await _context.Customers.ToListAsync();
    }

    // GET: api/Customers/5
    [HttpGet("{id}")]
    public async Task<ActionResult<Customer>> GetCustomer(int id)
    {
        var customer = await _context.Customers.FindAsync(id);

        if (customer == null)
        {
```

```

        return NotFound();
    }

    return customer;
}

// POST: api/Customers
[HttpPost]
public async Task<ActionResult<Customer>> PostCustomer(Customer customer)
{
    _context.Customers.Add(customer);
    await _context.SaveChangesAsync();

    return CreatedAtAction("GetCustomer", new { id = customer.Id }, customer);
}
}

```

Possiamo notare che vengono generate tutte le operazioni, come nell'[esempio 17.1](#), ma implementate sfruttando Entity Framework. Il contesto viene ottenuto tramite dependency injection e sono sfruttati la proprietà `customers` e il metodo `SaveChangesAsync` per accedere al `DbSet` e apportare i cambiamenti. Per motivi di spazio abbiamo omesso le altre operazioni di `Put` o `Delete`, ma l'implementazione è del tutto simile.

Come per le pagine web che inviano la form, anche nelle nostre API disponiamo della proprietà `ModelState`, che sfruttiamo per capire se la richiesta ricevuta è valida oppure no. L'uso dell'attributo `ApiController` a livello di classe introduce alcuni automatismi come il controllo dello stato, l'uso del body per la deserializzazione e la restituzione di un errore senza neanche chiamare la nostra funzione. Per stabilirlo, il motore di validazione sfrutta le data annotation, quel meccanismo già sfruttato nel [Capitolo 16](#) per mostrare l'HTML di modifica di una proprietà, e inoltre consente a Entity Framework di modellare il database in modo opportuno. Lo stesso serve anche per capire se una proprietà è obbligatoria e quali regole deve rispettare. Nel nostro caso la classe `customer` è così definita.

Esempio 17.16

```

public class Customer
{
    public int Id { get; set; }

    [Required]
    [MaxLength(200)]
    public string FirstName { get; set; }

    [Required]
    [MaxLength(200)]

```

```
public string LastName { get; set; }  
}
```

Di fatto rendiamo obbligatorio `FirstName` e `LastName` e consentiamo un massimo di 200 caratteri, indipendentemente dal formato JSON o XML utilizzato dalla richiesta. In caso contrario, il motore genera un 400 `Bad Request` e risponde con gli errori di validazione. Nell'[esempio 17.17](#) vediamo la risposta HTTP ottenuta in seguito a una richiesta errata.

Esempio 17.17

```
HTTP/1.1 400 Bad Request  
Content-Type: application/json; charset=utf-8  
Date: Sat, 01 Aug 2015 09:53:16 GMT  
Content-Length: 172  
  
{  
  "errors": {  
    "lastName": [  
      "The LastName field is required."  
    ],  
    "firstName": [  
      "The FirstName field is required."  
    ]  
  },  
  "title": "One or more validation errors occurred.",  
  "status": 400  
}
```

Lo scaffolding è sicuramente uno strumento molto comodo, ma la realizzazione di un servizio in questo modo non è sufficiente per scenari reali di utilizzo. L'interrogazione dei dati tramite richiesta GET è limitata al semplice recupero di tutte le entità. Il protocollo **OData** va oltre questa limitazione, definendo alcune specifiche per rendere i servizi più completi.

Supportare il protocollo OData

Lo standard **Open Data Protocol** (OData) è ormai giunto alla versione 4 e ha lo scopo di definire delle regole per consumare servizi principalmente orientati ai dati. Sebbene i servizi RESTful sfruttino le caratteristiche di HTTP per indicare che vogliamo ottenere o modificare una risorsa, non

sono sufficienti per indicare, per esempio, quanti record vogliamo ottenere, se ordinarli, filtrarli e così via.

Dovremmo quindi sfruttare il contenuto di una richiesta per definire un'entità oppure sfruttare la querystring per indicare tutte queste informazioni, e infine documentare il tutto per istruire chi fa la richiesta. Lo standard OData definisce già tutto questo, in modo di avere delle specifiche cross platform e cross language, con il vantaggio di parlare la stessa lingua oltre che, e questo non è poco, avere già framework che sappiamo parlare OData. ASP.NET è uno di questi e in pochi passi ci permette di realizzare servizi che supportino lo standard.

Il primo passo da fare è installare il pacchetto NuGet con ID `Microsoft.AspNetCore.OData` il quale contiene le estensioni necessarie per far funzionare il tutto. Successivamente, dobbiamo modificare il file `Startup.cs`, nel quale vengono configurati tutti gli aspetti dell'applicazione. Dobbiamo prima di tutto registrare le dipendenze all'interno della funzione `ConfigureServices` attraverso l'extension method `AddOData`. Nella funzione `Configure`, invece, dobbiamo arricchire MVC affinché supporti le query option filter, come viene mostrato nell'[esempio 17.18](#).

Esempio 17.18

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddOData();

        services.AddMvc();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        app.UseMvc(m => m.Select().MaxTop(100).Filter().OrderBy()
            .Count().EnableDependencyInjection());
    }
}
```

Con questa semplici istruzioni abilitiamo OData a tutti i servizi del nostro progetto e in particolare lo abilitiamo su tutte le action con l'unico requisito

che queste restituiscano un `IQueryable<T>`. Possiamo trasformare la funzione `GetCustomers` dell'[esempio 17.15](#) come nell'[esempio 17.19](#).

Esempio 17.19

```
[HttpGet]
[Queryable]
public ActionResult<IQueryable<Customer>> GetCustomers()
{
    return _context.Customers
}
```

Questa interfaccia, che Entity Framework supporta, permette di effettuare query LINQ e di tradurle poi in query sul database, con il massimo dell'ottimizzazione. Un altro requisito è quello di marcare con l'attributo `Queryable` le action che vogliamo supportino il query option. In questo modo, quando effettuiamo una richiesta GET possiamo sfruttare tutte le estensioni definite da OData per l'interrogazione, quali `$filter`, `$count`, `$orderby`, `$select`, `$skip`, `$top`. Sono tutti parametri query string che permettono rispettivamente di filtrare i record, ottenere il numero degli elementi, ordinare, selezionare solo alcuni campi, saltare o ottenere solo alcuni record. All'indirizzo <http://aspit.co/a6b> possiamo trovare maggiori dettagli riguardanti le specifiche, ma possiamo vedere subito qualche semplice esempio di richieste GET:

- ❑ `api/customers?$skip=10&$top=5`: recupera 5 record successivi ai primi 10;
- ❑ `api/customers?$filter=indexof(firstName, 'pippo') gt 0`: filtra cercando pippo nella proprietà `FirstName`;
- ❑ `api/customers?$select=FirstName&$orderby=lastName`: seleziona solo la proprietà `FirstName` e ordina il risultato per `LastName`.

Poche parole chiave sono sufficienti per ottenere già molto nell'interrogazione di un servizio, ideale per gli scenari più comuni.

OData, però, fa molto di più e definisce anche altri aspetti. Primo fra tutti è la capacità di operare su più entità, consentendo di navigare, per esempio,

all'interno degli ordini di un cliente o di filtrare in base a proprietà complesse figlie. Il servizio quindi non diventa più dedicato a una singola entità, ma all'intero contesto. Per questo motivo, OData prevede anche la possibilità di esporre dei metadati, utili a capire quali sono le entità esposte e quali proprietà contengono, per poi generare dei proxy per i vari linguaggi. Infine, OData consente anche di effettuare operazioni massive (più Create/Update/Delete) contemporaneamente in una sola richiesta, rendendola di fatto transazionale. Supporta inoltre la possibilità di effettuare merge e patch delle entità.

Anche in questo caso il framework dedicato ad ASP.NET ci viene in aiuto, ma dobbiamo scrivere un po' di codice in più che purtroppo non trova spazio in questo capitolo. Per approfondire il supporto OData rimandiamo a questo indirizzo <http://aspit.co/bsr> dove troviamo più informazioni sulla creazione completa di un endpoint OData. Ad ogni modo le query option rappresentano già un balzo in avanti nella realizzazione di servizi, non richiedono necessariamente l'uso di Entity Framework e sono facilmente utilizzabili da qualunque consumatore.

Possiamo chiudere quindi l'argomento servizi RESTful e passare ad un altro framework disponibile in ASP.NET che copre un'altra esigenza: la creazione di servizi real-time.

Create servizi real-time attraverso SignalR

Come abbiamo detto all'inizio del capitolo, i servizi sono diffusi in qualsiasi tipologia di applicazione. Tramite i client, dalle pagine HTML fino alle app mobile, essi permettono di interrogare e modificare i dati. Il loro ampio uso ha portato a una serie di ottimizzazioni per migliorare il traffico di rete, la loro efficienza e la velocità di comunicazione. Su questo aspetto, sono sempre più i client che necessitano di informazioni in real-time, cioè che giungano il prima possibile al client non appena queste sono disponibili. Notifiche di un social network, informazioni sul traffico, sistemi di chat e monitoraggio di attività sono alcuni degli esempi che siamo ormai abituati ad utilizzare che sfruttano comunicazioni real-time.

Per ottenere questo grado di prestazioni, il protocollo HTTP è stato esteso con un nuovo standard di nome WebSocket, che ha lo scopo di mantenere un canale bidirezionale tra client e server. In questo modo il canale è sempre pronto e permette al client di comunicare al server ma, fatto ancora più importante, permette al server di comunicare al client, una cosa non possibile tramite il normale HTTP. Con il diffondersi di queste esigenze e dell'implementazione di questo protocollo, anche IIS e ASP.NET si sono attrezzati con strumenti che permettono di realizzare scenari real-time. Lo strumento disponibile per farlo con ASP.NET si chiama **SignalR** e permette di creare delle API RPC, cioè che permettono al client e al server di invocare funzioni l'uno con l'altro. Da una parte fornisce l'infrastruttura per ASP.NET per gestire le connessioni e permettere di invocare i client, dall'altra mette a disposizione librerie in più linguaggi, come Java, JavaScript e .NET, le quali gestiscono in autonomia la connessione e la mantengono in caso di interruzioni.

Cosa ancora più importante, SignalR permette di creare servizi real-time sfruttando più trasporti. Il citato WebSocket è il migliore per questo obiettivo, ma non è supportato da tutti i web server e da tutti i client, perciò SignalR si fa carico di utilizzare tecniche alternative per simulare una comunicazione bidirezionale. Queste tecniche sono il Server Sent Event, il Forever Frame e l'AJAX Long Polling e, sostanzialmente, sfruttano l'HTTP o alcune caratteristiche dei browser per cercare di mantenere canali sempre attivi con il server, in modo da reagire non appena succede qualcosa. Ad ogni modo, come abbiamo detto, questi dettagli sono automaticamente gestiti al fine di supportare la totalità dei browser e delle piattaforme, e possiamo semplicemente godere di questo motore.

Per poter iniziare dobbiamo prima di tutto scaricare il pacchetto NuGet con ID `Microsoft.AspNetCore.SignalR` e, successivamente, dobbiamo creare un hub.

L'hub come servizio bidirezionale

L'hub è il punto di incontro tra il client e il server e definisce le API RPC che il server e i client mettono a disposizione. Un hub è sostanzialmente una rappresentazione logica e permette alle connessioni fisiche, mantenute da SignalR, di veicolare messaggi a una specifica operazione del client o del

server. Per questo motivo esiste un hub sul server, ma altrettanto un hub sul client, che è speculare alle operazioni del server. Sul client è presente una sola istanza che permette di invocare operazioni e agganciare gli eventi, mentre sul server l'unica informazione persistente sono le connessioni che creano l'hub, fanno eseguire l'operazione richiesta e lo rilasciano.

Dopo questa breve introduzione quindi, partiamo nella creazione di hub, una classe che ha la particolarità di ereditare da `Hub`, del namespace `Microsoft.AspNetCore.SignalR`. In esso definiamo tutte le operazioni, sotto forma di metodi/funzioni, che vogliamo rendere richiamabili dal client. Nell'[esempio 17.20](#) possiamo vedere una semplice implementazione.

Esempio 17.20

```
public class CustomersHub : Hub
{
    public Task Hello()
    {
        return Clients.All.SendAsync("hello", "my message");
    }
}
```

Il metodo `Hello`, in modo molto simile a quanto facciamo per creare API ASP.NET, rappresenta una funzione invocabile dal client. Questo, di per sé, non è niente di nuovo, dato che anche i servizi RESTful sono così; è interessante invece l'implementazione, poiché viene invocata una funzione `SendAsync` sull'oggetto `Clients.All`. Essa invoca la funzione il cui nome è indicato con il primo parametro e usa i successivi come argomenti della stessa. Questo significa che a sua volta il servizio chiama tutti i client connessi nello stesso momento e inoltra il saluto. SignalR si fa carico automaticamente di sapere quali sono i client connessi e di invocarne il messaggio, il tutto in real-time, senza preoccuparci dello strato di trasporto. `Clients` è una proprietà della classe `Hub` che, come vedremo, ci consente di invocare tutti o alcuni client.

Ci sono situazioni in cui vogliamo richiamare i client in altri contesti, come all'interno di una pagina web o di un servizio. In questi casi possiamo facilmente ottenere un'istanza del contesto di un hub e ritrovarci con la medesima proprietà `Clients`, come viene mostrato nell'[esempio 17.21](#).

Esempio 17.21

```
public IActionResult SayHello(  
    [FromServices] IHubContext<CustomersHub> context)  
{  
    context.Clients.All.SendAsync("hello", "my message");  
    return View();  
}
```

Nell'esempio sfruttiamo la dependency injection per ottenere il contesto dell'hub e con esso invochiamo i client allo stesso modo di quanto fatto nell'[esempio 17.20](#).

Possiamo sostituire questo esempio banale con una definizione di hub più realistica. Sempre in tema di client, l'utente potrebbe richiedere l'esecuzione di un processo lungo, come un'esportazione, e di ricevere una notifica quando il processo termina. L'[esempio 17.22](#) mostra un'ipotetica implementazione.

Esempio 17.22

```
public class CustomersHub : Hub  
{  
    public bool ExportData(int id)  
    {  
        // Avvio in asincrono  
        Task.Run(async () =>  
        {  
            // Simulo operazione lunga  
            Thread.Sleep(3000);  
  
            // Notifico il chiamate dell'avvenuta operazione  
            await Clients.Caller.SendAsync("exportDataReady", "download/" + id);  
        });  
  
        // Esito della richiesta  
        // Rispondo subito senza aspettare l'operazione  
        return true;  
    }  
}
```

Il codice è commentato nelle sue parti più salienti. Da notare che la funzione `ExportData` si completa immediatamente, invocando l'esportazione su un altro thread. Quando l'esportazione è terminata, invochiamo poi `exportDataReady` per notificare il richiedente

dell'esportazione, identificato da un'altra proprietà speciale `Clients Caller`.

Una volta creato l'hub, non ci resta che abilitare a livello di intera applicazione il supporto a SignalR. Per farlo, abbiamo bisogno di configurare i servizi della dependency injection e il middleware sempre nello `startup.cs`, così come mostrato nell'[esempio 17.23](#).

Esempio 17.23

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddSignalR();
    }

    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        app.UseSignalR(route =>
        {
            route.MapHub<CustomersHub>("/customershub");
        });
    }
}
```

Finora abbiamo visto come implementare la parte server e abbiamo parlato solamente del client, perciò è giunto il momento di capire come può interfacciarsi con un hub, invocare un'operazione o ricevere un evento. Abbiamo a disposizione molti linguaggi e in questo libro vogliamo affrontare quello più inerente allo sviluppo con ASP.NET: JavaScript.

Utilizzare SignalR da JavaScript

JavaScript è il linguaggio utilizzato nelle pagine HTML per manipolare il DOM, effettuare richieste AJAX e, in generale, per rendere dinamiche le pagine client side. Associato a un hub di SignalR, JavaScript rende le pagine più fruibili e più moderne, evitando polling da parte del client. Per poterlo usare, dobbiamo usufruire della libreria JavaScript che può essere installata attraverso **npm** e l'istruzione seguente.

Esempio 17.24

```
npm install @aspnet/signalr
```

La libreria supporta script per il browser o per NodeJS. Per i nostri scopi utilizziamo il primo, che possiamo trovare nella cartella `node_modules/@aspnet/signalr/dist/browser`.

Nella pagina HTML dove vogliamo utilizzarlo, è necessario quindi referenziare tale libreria, come viene mostrato nell'[esempio 17.25](#).

Esempio 17.25 - HTML

```
<html>
<head>
  <script type="text/javascript" src="signalr.min.js "></script>
  <script type="text/javascript">
    // inizializzazione canale comunicazione
  </script>
</head>
```

Nel codice precedente è presente dello script, portato direttamente nella stessa posizione di dove risiede il file HTML. Procediamo ora a inizializzare e sfruttare quindi il nostro hub. Abbiamo a disposizione l'oggetto `signalR.HubConnectionBuilder` che ci permette di creare una connessione, intercettare le chiamate dal server o invocare funzioni su di esso.

L'[esempio 17.26](#) mostra come fare tutto questo da JavaScript stampando in console le operazioni.

Esempio 17.26

```
var connection = new signalR.HubConnectionBuilder().
  withUrl("/customersHub").build();

// Intercetto il completamento dell'export
connection.on("exportDataReady", function (uri) {
  // Stampo a console
  console.log("Download disponibile " + uri);
});

// Avvio la connessione
```

```
connection.start().then(function(){
    console.log("Export in corso...");
    // Invio la richiesta di export
    connection.invoke("exportData", 6).catch(function (err) {
        // Mostro l'errore in caso di errore di invocazione
        return console.error(err.toString());
    });
}).catch(function (err) {
    // Mostro l'errore in console in caso di errore di connessione
    return console.error(err.toString());
});
```

Possiamo notare che la funzione invocabile dal server viene impostata prima di avviare tutta la connessione tramite `start`. Poiché il processo è asincrono, la funzione `then` ci permette di conoscere quando il canale di comunicazione è attivo e di invocare l'esportazione. Chiaramente questa parte di codice sarebbe più opportuno che fosse chiamabile dal click di un bottone. La libreria mette a disposizione ogni evento e sfrutta il **pattern Promise** per consentirci di intercettare le avvenute operazioni o gli errori e di reagire di conseguenza. Eseguendo questa pagina possiamo quindi ottenere l'invocazione del metodo di esportazione e, dopo tre secondi, la stampa a console dell'uri per il download.

Possiamo apprezzare da questo piccolo esempio come bastino poche linee di codice per ottenere una comunicazione bidirezionale, che ci permette di ignorare tutte le complicazioni della comunicazione.

Conclusioni

In questo capitolo abbiamo affrontato gli strumenti di ASP.NET dedicati allo sviluppo di servizi moderni, al passo con le esigenze attuali. Siamo partiti da come impiegare ASP.NET per esporre servizi RESTful. Abbiamo visto come possiamo sfruttare a pieno le capacità di HTTP, con i metodi, gli status code e il body delle richieste. Tutto questo viene gestito automaticamente dal framework, che trasforma le richieste mappandole sui nostri oggetti e istradandole alla rispettiva funzione per poi serializzarne il risultato. Abbiamo visto come personalizzare alcuni aspetti di questo motore e come avere il pieno controllo della richiesta e della risposta.

Con lo scaffolding, poi, possiamo farci aiutare da Visual Studio per ottenere la generazione del codice per il CRUDQ di un'entità, basandoci su Entity Framework. Sempre sfruttando questo framework, possiamo usare le estensioni per OData per esporre servizi che rispondano a uno standard per l'interrogazione e la manipolazione di dati.

Infine, abbiamo visto come creare servizi real-time mediante SignalR, un framework che si fa carico di tutte le problematiche di trasporto per metterci a disposizione API-RPC bidirezionali tra client e server. Chiudiamo questi due capitoli dedicati alle tecnologie web e passiamo, nel prossimo capitolo, a una tematica generica relativa alla sicurezza, per capire quali sono gli errori più comuni da non commettere e quali strumenti abbiamo a disposizione per la sicurezza delle informazioni.

La sicurezza nelle applicazioni .NET

Nel corso dei precedenti capitoli abbiamo esaminato le varie funzionalità offerte da .NET e il loro utilizzo nella programmazione con C#. Grazie a tali strumenti abbiamo la possibilità di sviluppare applicazioni in modo semplice ma, nonostante questo, non dobbiamo dimenticare l'uso che può essere fatto del nostro software, sia da parte degli utenti sia dagli altri sviluppatori, nel caso in cui il nostro codice possa essere utilizzato o integrato da altri.

Nel corso degli anni, l'affermazione del personal computer come strumento di lavoro e svago ha aumentato la maturità dell'utente in materia di agilità d'uso dei nostri sistemi. Di contro è cresciuta di pari passo la malizia nel cercare metodi di utilizzo diversi da quelli previsti, aprendo più o meno volontariamente tutta una serie di scenari in cui l'integrità delle nostre applicazioni arriva spesso a essere effettivamente compromessa.

Quello che accade solitamente è che, in qualità di progettisti, teniamo un comportamento corretto nei confronti delle interfacce, dei dati che immettiamo, dei file che carichiamo e così via; questo ci porta a trascurare altre cose che invece possono essere messe in atto da personaggi meno corretti di noi. Il normale utente, infatti, finisce per scavalcare l'utilizzo che abbiamo pensato, ritenendo di poter ottenere di più dal programma, per esempio, immettendo dati quasi *senza senso*, al limite del comprensibile.

Ma c'è di più: l'utente malizioso, inoltre, utilizza volontariamente le proprie conoscenze informatiche per immettere dati, allo scopo di ottenere, per esempio, informazioni cui non dovrebbe avere accesso. Oltre a questo,

non dobbiamo dimenticare che la nostra applicazione risiede su un PC o su un server e, pertanto, potrebbe non essere isolata rispetto alle altre applicazioni, che potrebbero cercare di recuperare dati o risorse private. Anche la possibilità di realizzare applicazioni web le rende, per propria natura, accessibili da molti utenti, con le implicazioni potenziali che abbiamo citato.

Nel corso del capitolo introdurremo quindi gli strumenti messi a disposizione di .NET per cercare di rendere il nostro software il più sicuro possibile.

Progettare applicazioni sicure

Nel contesto che abbiamo introdotto, possiamo dire che il concetto di sicurezza di un'applicazione si estende sia a **livello applicativo**, relativo cioè alle modalità di utilizzo, sia a quello **architetturale**, riferendoci con questo termine al codice che può essere eseguito nei confronti del sistema che ospita l'applicazione stessa.

Negli anni, la cultura della programmazione ha maturato una vera e propria dottrina della sicurezza, incentrata sul principio di security come requisito e non come accessorio alle funzionalità. Questo comporta che, nelle fasi iniziali di progettazione di un software, alcune delle prime scelte architetturali e tecnologiche siano fortemente influenzate dal livello di sicurezza che l'applicazione deve rispettare, più precisamente al **livello di rischio** nel quale il sistema incorre durante il proprio ciclo di vita.

Dobbiamo considerare il concetto di requisito proprio come elemento che deve essere introdotto e seguito in tutto il corso dello sviluppo, scartando la possibilità che la protezione del codice e la sicurezza applicativa possano essere aggiunte con facilità in un secondo momento.

Sicurezza in base all'ambiente e al contesto

Quando dobbiamo rendere un'applicazione sicura, il primo aspetto che prendiamo in considerazione è quello dell'autenticazione (riconoscere l'utente) e dell'autorizzazione (identificare cosa può fare l'utente).

Questi due aspetti sono tutt'altro che banali. Relativamente al primo, ci sono molti punti da chiarire prima di procedere a una scelta sulla

tecnologia. Alcune tipiche domande sono:

- L'applicazione gira su server/clienti aperti all'esterno o solo in locale?
- Gli utenti fanno parte di una rete aziendale, oppure l'applicazione è aperta al mondo e chiunque si può registrare?
- Devo poter autenticare gli utenti tramite i social network come Facebook, Microsoft, Twitter, Google ecc?
- Devo immagazzinare i dati degli utenti in locale o posso affidarmi a un servizio esterno?

In base alle risposte a queste e altre domande, possono emergere quadri differenti che necessitano soluzioni differenti. Passiamo ad analizzare le più comuni.

Applicazione intranet

Un'applicazione Windows o web, destinata a utenti che fanno parte di una rete aziendale, può sfruttare la Windows Authentication, tramite la quale l'utente dell'applicazione è quello autenticato su Windows. Grazie a questo meccanismo, una volta che l'utente si è loggato non deve immettere alcuna credenziale per accedere all'applicazione, con grandi vantaggi per l'usabilità. Inoltre, i ruoli dell'utente possono risiedere su Active Directory e quindi l'applicazione non necessita di memorizzare alcun dato relativo all'autorizzazione sul proprio database.

Applicazione web con ASP.NET Identity

Un'applicazione web aperta a chiunque può sfruttare diversi protocolli di autenticazione e diverse tecnologie. La soluzione più semplice la fornisce ASP.NET Identity. Questo prodotto offre UI, codice e database per gestire tutte le interazioni dell'utente. Le interfacce utente che abbiamo a disposizione sono quelle per il login, la registrazione e il cambio password. Le interfacce, presenti nel package di ASP.NET Identity, vengono integrate

nell'applicazione, e questo significa che l'applicazione è completamente autosufficiente.

L'interazione con il database avviene tramite **Entity Framework**, il che rende possibile estendere lo schema del database per aggiungere altre informazioni all'utente sfruttando le migrazioni.

Oltre alla gestione degli utenti, ASP.NET Identity offre anche la gestione dei ruoli legati agli utenti che ci permette di associare un utente a uno o più ruoli. Essendo questa funzionalità integrata con ASP.NET, possiamo integrarla gratuitamente con il meccanismo di autorizzazione di MVC.

Avere un'applicazione che include anche il login è perfetto in scenari semplici, ma si va incontro a molti problemi in sistemi più complessi.

ASP.NET Identity esiste sia per .NET Framework sia per .NET Core. Tuttavia, il prodotto è diverso a seconda del framework e quindi ne esistono due package differenti.

Applicazione con Single Sign-On proprietario

Quando si sviluppa un sistema molto grande e complesso, si può scegliere di suddividere il sistema in più applicazioni, così da suddividere la complessità in piccoli blocchi. Queste applicazioni possono anche essere sviluppate con tecnologie diverse (ASP.NET, WPF, Java), da fornitori diversi e in tempi diversi. Tuttavia, queste applicazioni condividono la necessità di autenticare l'utente. Se ogni applicazione avesse il proprio sistema di autenticazione (per esempio, basato su ASP.NET Identity per applicazioni web), si creerebbero grossi problemi di interoperabilità, in quanto gli utenti avrebbero un set di credenziali per ogni applicazione e dovrebbero fare diversi login. Inoltre, le applicazioni avrebbero difficoltà a scambiarsi i dati in sicurezza, in quanto non potrebbero validare quale utente sta facendo la richiesta da un'altra applicazione.

In questi casi torna utile avere un sistema di **Single Sign-On** (SSO d'ora in poi) cioè un'applicazione a parte che ha il solo compito di gestire gli utenti rendendo possibile la registrazione, il login, il cambio di password, il reset password e così via. Le varie applicazioni che compongono il sistema non hanno più bisogno di un loro meccanismo di autenticazione ma si affidano al SSO.

I vantaggi principali di avere un SSO sono che le applicazioni possono parlare in sicurezza più facilmente tra di loro conoscendo gli utenti e che gli utenti non devono fare molteplici login, in quanto ne esiste uno unico, valido per tutte le applicazioni.

La creazione di un SSO è un'operazione molto complessa, soprattutto perché occorre trovare un protocollo di comunicazione con le applicazioni per restituire i dati dell'utente loggato, verificare se il login è scaduto e altro ancora. Inoltre, se abbiamo un'applicazione che espone API via REST, dobbiamo anche fare in modo che queste chiamate siano autenticate, cioè che l'utente che ha fatto login sia propagato anche alla API.

Negli ultimi anni, sono emersi due protocolli che sono poi diventati uno standard: **OpenIdConnect** e **OAuth2**. Il primo si occupa della fase di login di un utente e permette di stabilire quali dati dell'utente restituire all'applicazione dopo il login, la durata della sessione di login, dati di contesto e altro ancora. Il secondo si occupa della comunicazione tra applicazioni, quindi entra in gioco quando l'applicazione su cui l'utente ha fatto il login invoca una API REST.

Creare un SSO con ASP.NET non è complesso grazie a **IdentityServer** (<https://www.identityserver.io/>). Si tratta di un prodotto di terze parti che implementa in .NET i suddetti protocolli, astruendo per noi tutte le complicazioni e lasciandoci il solo compito di configurare le applicazioni, il database e poco altro.

IdentityServer offre già delle interfacce pronte per il login, il logout e altre operazioni, ma queste sono completamente personalizzabili secondo le nostre necessità. Inoltre, IdentityServer, internamente, è completamente estendibile, a tal punto che c'è un'estensione ufficiale che utilizza ASP.NET Identity come meccanismo di storage degli utenti. Oltre al login con utenti *proprietary*, IdentityServer permette di integrarsi molto facilmente anche con i social network, offrendo quindi un range enorme di possibilità di login.

Gli SSO coprono esigenze relative alla gestione degli utenti, ma non quelle relative ai loro permessi. Tutta la fase di autorizzazione deve essere sempre gestita dalle singole applicazioni.

Per quanto riguarda le applicazioni che accedono al SSO, Microsoft offre già tutte le librerie necessarie, che astraggono tutte le complessità del protocollo. Noi ci dobbiamo preoccupare solo della fase di configurazione, in quanto il resto viene gestito dalle librerie. Queste librerie esistono non

solo per .NET, ma anche per JavaScript, Java e altri linguaggi. Questa semplicità di utilizzo dei protocolli *OpenIdConnect* e *Oauth2* è una delle cause di forte adozione da parte degli sviluppatori.

Creare un SSO offre innumerevoli vantaggi ma è comunque un'applicazione da mantenere all'interno del nostro sistema. Fortunatamente esistono servizi che ci offrono un sistema di SSO che va solo configurato.

[Applicazione con Single Sign-On di terze parti](#)

L'esigenza di un SSO è andata sempre crescendo e con questa crescita sono arrivati diversi fornitori di servizi di autenticazione. Questi servizi offrono un SSO già sviluppato, all'interno del quale dobbiamo solo configurare quali dati dell'utente vogliamo raccogliere, quali vogliamo esporre, quali siano le applicazioni che devono accedere agli utenti, l'interfaccia grafica, i messaggi via email e, in generale, tutto quello che ha la necessità di essere brandizzato.

Inoltre, questi servizi offrono l'integrazione con i social network, garantendo così una copertura totale dei vari metodi di autenticazione di un utente.

Microsoft offre un ottimo SSO tramite il servizio **AzureAD B2C**. Si tratta di un SSO molto semplice da configurare, grazie al portale di Azure, ed è ottimo in scenari semplici dove non ci sono molte personalizzazioni da fare nei vari flussi di interazione con l'utente. Quando gli scenari diventano più complessi, si può valutare l'adozione di **Auth0**. Questo è un SSO che offre una maggiore versatilità in scenari complessi, ma la cui potenza non è sempre necessaria: quindi, prima di scegliere uno o l'altro (o servizi di ulteriori fornitori), è bene sempre verificare i propri requisiti.

[Applicazione Mobile](#)

Le applicazioni mobile che necessitano di autenticare un utente devono rivolgersi a un sistema di autenticazione esterno. A meno che non si sviluppi un protocollo proprietario, è molto probabile che le applicazioni mobile vadano a loggarsi sfruttando un SSO che utilizza *OpenIdConnect* e *OAuth2*. Anche in questo caso, esistono già librerie pronte per l'uso distribuite da Microsoft tramite NuGet, quindi autenticare l'utente è un'operazione estremamente semplice.

In questa breve introduzione ai meccanismi di autenticazione e autorizzazione abbiamo solo scalfito la superficie. Parlare approfonditamente di tutte le possibilità richiederebbe un libro a parte. Per questo motivo abbiamo preferito semplicemente introdurre questi concetti.

Ora cambiamo decisamente argomento e vediamo come affrontare un altro tema molto importante: la crittografia delle informazioni.

Principi di crittografia

Nei paragrafi precedenti abbiamo illustrato come sia possibile organizzare le classi e, complessivamente, le nostre applicazioni, in modo da ridurre i rischi di sicurezza legati alla violazione degli altri dati presenti sulla macchina. Oltre a quanto analizzato, è usuale avere dei dati che, anche in caso di accesso da parte di altri utenti, non siano direttamente leggibili dagli esseri umani, in quanto **cifrati**.

All'interno del .NET sono disponibili molte classi per gestire dati riservati e rendere il loro immagazzinamento in memoria o il loro scambio più sicuro. Il concetto di crittografia si basa sul principio di modificare la rappresentazione dei dati in modo tale che non siano leggibili direttamente, che solo l'applicazione possa conoscerne il valore reale e che non sia possibile risalire al valore originale, se non in presenza di precise condizioni.

Come vedremo nei paragrafi successivi, molte delle operazioni di crittografia si basano sul principio di modificare i dati in funzione di un preciso valore, detto comunemente **chiave**, valore che, a sua volta, può avere un livello di sicurezza dettato dalla riproducibilità dello stesso e dalla difficoltà con la quale può essere conosciuto da altri utenti oltre al cifratore.

Windows Data Protection

Una delle soluzioni più immediate che abbiamo a disposizione in .NET è rappresentata dalla classe `ProtectedData` del namespace `System.Security.Cryptography`.

Trattandosi di “Windows Data Protection”, questo modello di protezione è accessibile in .NET mediante il pacchetto NuGet

System.Security.Cryptography.ProtectedData

Tale classe implementa la cifratura, basandosi sul servizio di data protection (DPAPI) esposto dal sistema operativo stesso. Con i due metodi statici `Protect` e `Unprotect` possiamo cifrare e decifrare i nostri dati rappresentati sotto forma di array di byte e abbiamo la possibilità di definire un livello di decifrabilità nel contesto di sistema o al solo utente autenticato, grazie al parametro `DataProtectionScope`.

Nel codice dell'[esempio 18.1](#), possiamo vedere come cifrare una semplice stringa; con `DataProtectionScope.CurrentUser` l'algoritmo usa come chiave la password dell'utente.

Esempio 18.1

```
string data = "testo da cifrare";
byte[] dataByte = Encoding.UTF8.GetBytes(this.data);
byte[] entropy = Encoding.UTF8.GetBytes("chiave ulteriore sicurezza");
byte[] dataCripted = ProtectedData.Protect(dataByte, entropy,
DataProtectionScope.CurrentUser);
```

Come possiamo vedere nel codice, è possibile aggiungere un'ulteriore chiave da affiancare a quella derivata dal `DataProtectionScope`. Il livello di sicurezza di questa modalità si basa essenzialmente sulla **complessità della password** scelta dall'utente. L'utilità maggiore di questa classe risponde essenzialmente alle esigenze di una applicazione che memorizzi, anche su file system, delle informazioni alle quali solo l'utente può accedere in chiaro.

Crittografia simmetrica

La semplicità delle Data Protection API isola tuttavia il loro ambito di applicazione all'uso locale di informazioni cifrate e al sistema operativo Windows; qualora abbiamo la necessità di trasferire i dati e, quindi, di proteggerli dalla vista e dalla modifica da parte di utenti non autorizzati, è necessario adottare sistemi di cifratura basati su **chiavi esterne**, a cui solo i legittimi destinatari possano avere accesso. Dobbiamo pensare al concetto di cifratura in un ambito più ampio e non strettamente legato

all'informatica: storicamente, infatti, la crittografia ha sempre avuto un ruolo chiave in ambito **militare**. In quel campo la riservatezza delle informazioni è di vitale importanza per l'esito delle operazioni. È proprio da tale contesto che lo sviluppo della crittografia ha tratto le maggiori spinte evolutive, ancor prima dello sviluppo di sistemi di elaborazione assistiti da calcolatori; agli inizi del novecento sono state sviluppate moderne, per così dire, modalità di codifica e decodifica di messaggi basati su un'unica chiave. Tale tipo di crittografia, detta **simmetrica**, si basa proprio sull'uso della stessa chiave per cifrare e decifrare un certo dato.

IN .NET la classe astratta che descrive le API di cifratura simmetrica è `SymmetricAlgorithm`, (assembly `System.Security.Cryptography.Primitives`) dalla quale deriva una serie di classi ciascuna con la propria implementazione dell'algoritmo, in funzione della lunghezza della chiave, e quindi di sicurezza generale. Possiamo vedere la gerarchia di tali classi nella [Figura 18.1](#).

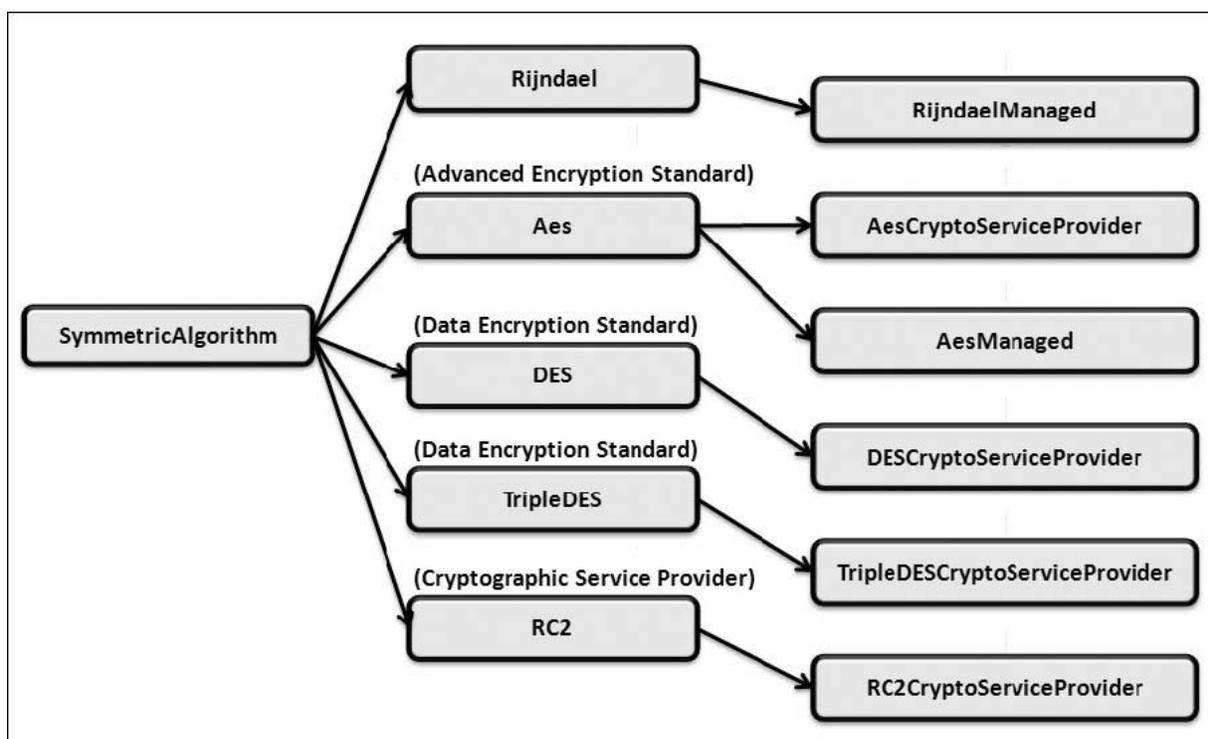


Figura 18.1 – Gerarchia delle classi `SymmetricAlgorithm`.

Ogni classe astratta corrispondente al nome dell'algoritmo sviluppato ha un'implementazione concreta (assembly

System.Security.Cryptography.Algorithms) e, come detto, le classi differiscono tra loro per la lunghezza in bit della chiave; qui di seguito ne troviamo un elenco con i relativi valori:

- ❑ Rijndael: 128, 192 e 256 bit;
- ❑ Aes: 128, 192 e 256 bit;
- ❑ DES: 64 bit;
- ❑ TripleDES: 128 e 192 bit;
- ❑ RC2: da 40 a 128 con step di 8 bit.

Per utilizzare una di queste classi è necessario, quindi, generare una chiave di grandezza corrispondente a quella richiesta dall'algoritmo. Tale chiave, espressa sotto forma di array di byte, può essere generata in diversi modi, così come possiamo vedere nel codice dell'[esempio 18.2](#).

Esempio 18.2

```
//generazione automatica da parte provider
TripleDESCryptoServiceProvider cryptoProvider =
new TripleDESCryptoServiceProvider();
key = cryptoProvider.Key;
vector = cryptoProvider.IV;
//dichiarazione esplicita diretta
this.key = new byte[] { 0xea, 12, 0x43, 0xf5, 0x42, 0x63, 0x16, 0xd6, 6, 0x58,
0x7c, 0x2c, 0xdd, 0x22, 9, 0x16 };

string password = "password per generare chiave";
string salt = "salt per generare chiave";

byte[] passwordByte = Encoding.UTF8.GetBytes(password);
byte[] saltByte = Encoding.UTF8.GetBytes(salt);

this.key = new PasswordDeriveBytes(passwordByte, saltByte).
CryptDeriveKey("TripleDES", "SHA1", 192, this.vector);
```

Possiamo generare una chiave dichiarando direttamente un array di byte, oppure, come possiamo vedere nel codice, possiamo farla generare automaticamente dal provider scelto (nell'esempio TripleDESCryptoServiceProvider).

Inoltre, abbiamo la possibilità di utilizzare la classe `PasswordDeriveBytes`, con la quale possiamo generare una chiave riferendoci a una password arbitraria, combinata a un codice “salt” per ulteriore sicurezza. Internamente, anche questa classe si appoggia a un algoritmo simmetrico di cifratura, che possiamo dichiarare nel metodo `CryptDeriveKey`, generatore della chiave desiderata.

Il vettore identificato con la proprietà `IV` di `SymmetricAlgorithm` concorre alla codifica del messaggio insieme alla chiave segreta. Poiché, a parità di chiave, il risultato della cifratura di un dato è costante, l'utilità del vettore è quella di inserire un'ulteriore elemento di cifratura da variare a ogni specifica codifica, per esempio, in corrispondenza della trasmissione del messaggio attraverso la rete.

Generati la chiave e il vettore per la cifratura, possiamo criptare un messaggio attraverso il metodo `CreateEncryptor` della `SymmetricAlgorithm` scelta e utilizzare tale oggetto per popolare un `CryptoStream` quale stream intermedio a quello usato per gestire il messaggio criptato. Nell'[esempio 18.3](#) abbiamo scritto in un file un semplice testo, associando il `FileStream` del file con il `CryptoStream` popolato dai dati cifrati.

Esempio 18.3

```
byte[] dataByte = Encoding.UTF8.GetBytes("testo da cifrare"));
using (ICryptoTransform encryptor =
    this.cryptoProvider.CreateEncryptor(this.key, this.vector))
{
    using (Stream stream = File.Create(@"d:\encrypted.txt"))
    {
        using (CryptoStream cryptoStream = new CryptoStream(stream,
            encryptor, CryptoStreamMode.Write))
        {
            cryptoStream.Write(dataByte, 0, dataByte.Length);
            cryptoStream.FlushFinalBlock();
        }
    }
}
```

Per procedere alla decrittazione del testo presente nel file è sufficiente creare un'istanza di `ICryptoTransform` attraverso il metodo `CreateDecryptor`, utilizzando chiave e vettore con i quali il dato è stato cifrato. Come possiamo vedere nel codice 18.4, possiamo recuperare il messaggio originale aprendo un `CryptoStream` in lettura, passando l'oggetto `ICryptoTransform` dopo aver associato tale stream al `FileStream` del file cifrato.

Esempio 18.4

```
using (ICryptoTransform encryptor =
    this.cryptoProvider.CreateDecryptor(this.key, this.vector))
{
    using (Stream stream = File.OpenRead(@"d:\encrypted.txt"))
    {
        using (Stream cryptoStream = new CryptoStream(stream,
            encryptor, CryptoStreamMode.Read))
        {
            using (StreamReader reader = new StreamReader(cryptoStream))
            {
                this.tb2.Text = reader.ReadToEnd();
            }
        }
    }
}
```

Per risalire al testo leggibile non ci rimane che aprire uno `StreamReader` con il `CryptoStream` da leggere.

Cablare la chiave di cifratura nel codice non è una buona pratica poiché alcuni tool rendono molto semplice la decompilazione degli assembly. In certi contesti, potrebbe essere utile usare la Data Protection API, per cifrare tale chiave durante il setup dell'applicazione e memorizzarla nel registro di sistema o su file system.

A fronte di una facilità d'implementazione, semplificata dalle classi presenti in .NET, la difficoltà maggiore rimane nello scambiare la chiave dall'autore agli utenti autorizzati, i quali sono responsabili della **segretezza di tale chiave** per mantenere in vigore il sistema di riservatezza dei dati. Come possiamo intuire, questo onere rappresenta un forte vincolo alla robustezza del modello e potrebbe risultare troppo debole per alcune tipologie di dati.

Inoltre, poiché i dati sono cifrati con la medesima chiave posseduta da mittente e destinatario, il sistema simmetrico non permette l'identificazione univoca dell'autore di un messaggio e dell'integrità stessa del dato, che può essere manipolato da un attore intermedio, ovviamente in possesso, più o meno legittimo, della chiave di cifratura.

Per far fronte a tali vincoli è stato sviluppato il modello a chiave asimmetrica, che illustreremo nel prossimo paragrafo.

Crittografia asimmetrica

La cifratura asimmetrica si basa sul principio di elaborazione dei dati attraverso una coppia di chiavi, diverse e non riconducibili l'una all'altra. Tale coppia è costituita da una **chiave pubblica**, che viene impiegata per cifrare i dati, e una **privata** che, in combinazione con quella pubblica, consente la decifratura. In tale modello, quando un mittente deve inviare un messaggio riservato a un destinatario, chiede la sua chiave pubblica e cifra il messaggio secondo uno degli algoritmi disponibili. Alla ricezione del messaggio, solo il destinatario in possesso della chiave privata, accoppiata a quella pubblica con cui è stato cifrato il messaggio, può effettuare la decifratura.

Poiché la cifratura e la decifratura con gli algoritmi asimmetrici è **abbastanza onerosa** in termini di elaborazione macchina, spesso viene utilizzata per cifrare una chiave simmetrica, con cui viene elaborato il dato riservato. La chiave può essere scambiata contestualmente al messaggio cifrato.

Analogamente alla cifratura simmetrica, in .NET abbiamo una serie di classi che implementano algoritmi di cifratura asimmetrica; il più noto è sicuramente **RSA** (sigla che deriva dall'acronimo dei cognomi degli inventori), i cui principi sono stati descritti nel 1977. Nel grafico della [Figura 18.2](#) possiamo trovare uno schema delle classi a nostra disposizione.

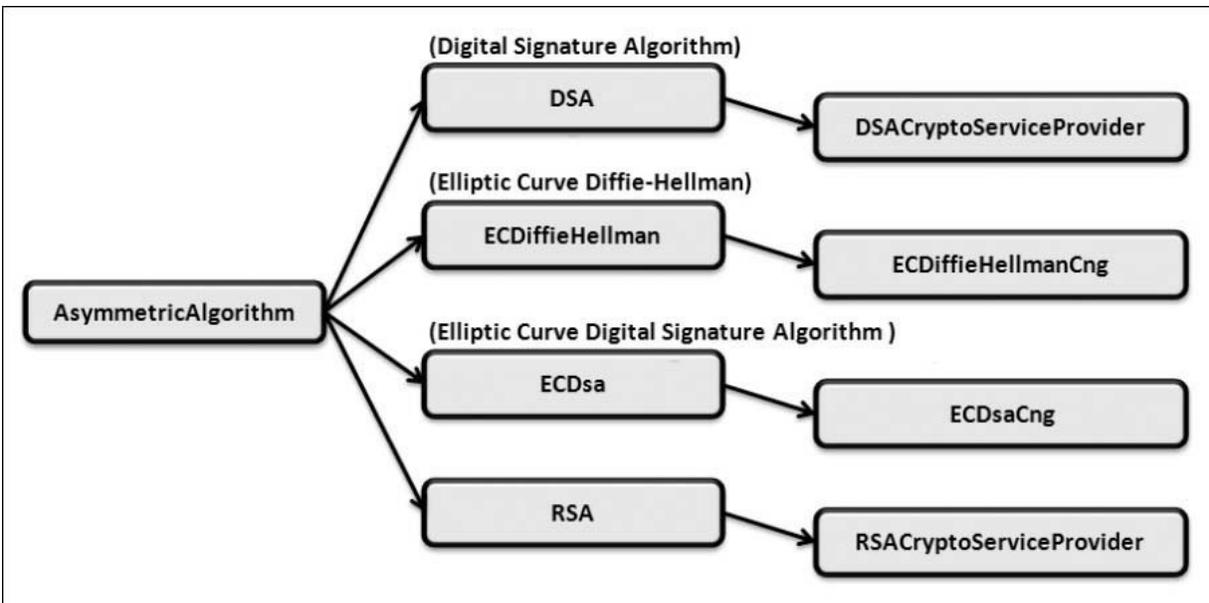


Figura 18.2 – Gerarchia delle classi AsymmetricAlgorithm.

Cerchiamo di comprendere l'utilizzo della cifratura asimmetrica utilizzando l'algoritmo RSA in modo da criptare una chiave da utilizzare nella cifratura simmetrica di un testo, così come abbiamo descritto poco sopra. La prima operazione sarà quella di generare la coppia di chiavi, pubblica e privata. Nell'[esempio 18.5](#) andiamo a scrivere le due chiavi su due file XML.

Esempio 18.5

```

using (RSACryptoServiceProvider asCryptoProvider =
    new RSACryptoServiceProvider())
{
    File.WriteAllText(@"d:\PublicKey.xml",
        asCryptoProvider.ToXmlString(false));
    File.WriteAllText(@"d:\PublicAndPrivate.xml", asCryptoProvider.
        ToXmlString(true));
}
  
```

Grazie al metodo `ToXmlString`, le chiavi vengono rappresentate sotto forma di stringa come infoset XML, in modo da semplificarne la scrittura su file system.

Del metodo `WriteAllText` della classe `File` esiste anche la versione asincrona, chiamata `WriteAllTextAsync`, che permette

scrittura del file senza necessariamente bloccare il thread.

Come abbiamo visto nel paragrafo precedente, andiamo a far creare chiave e vettore al provider di cifratura simmetrica, nell'esempio, TripleDESCryptoServiceProvider. Successivamente, andiamo a leggere la chiave pubblica e, con questa, andiamo a effettuare la cifratura di chiave e vettore, attraverso il metodo Encrypt di RSACryptoServiceProvider.

Infine, cifriamo il messaggio riservato con chiave e vettore originali e l'algoritmo simmetrico desiderato, così come descritto nell'[esempio 18.6](#).

Esempio 18.6

```
//creazione chiave simmetrica per cifratura
TripleDESCryptoServiceProvider cryptoProvider = new
TripleDESCryptoServiceProvider();
byte[] key = cryptoProvider.Key;
byte[] vector = cryptoProvider.IV;

//lettura chiave pubblica per cifratura asimmetrica
string publicKeyOnly = File.ReadAllText(@"d:\PublicKey.xml");

//cifratura asimmetrica della chiave simmetrica
using (RSACryptoServiceProvider asCryptoProvider = new
RSACryptoServiceProvider())
{
    asCryptoProvider.FromXmlString(publicKeyOnly);
    this.keyCrypted = asCryptoProvider.Encrypt(key, true);
    this.vectorCrypted = asCryptoProvider.Encrypt(vector, true);
}

//dati riservati
byte[] dataByte = Encoding.UTF8.GetBytes(this.data);

//cifratura simmetrica con chiave e vettore autogenerati
using (ICryptoTransform encryptor =
    cryptoProvider.CreateEncryptor(key, vector))
{
    using (Stream stream = File.Create(@"d:\encrypted.txt"))
    {
        using (CryptoStream cryptoStream = new CryptoStream(stream,
            encryptor, CryptoStreamMode.Write))
        {
            cryptoStream.Write(dataByte, 0, dataByte.Length);
            cryptoStream.FlushFinalBlock();
        }
    }
}
```

Al momento di voler decifrare il messaggio, dobbiamo decrittare chiave e vettore utilizzando la coppia chiave pubblica e privata con l'algoritmo

asimmetrico, per finire con la decifratura simmetrica del messaggio. Grazie al metodo `FromXmlString` possiamo agevolmente recuperare il valore della coppia di chiavi in nostro possesso ed effettuare la decifratura con il metodo `Decrypt` di `RSACryptoServiceProvider`.

Oltre a `FromXmlString` e `ToXmlString`, abbiamo la possibilità di rappresentare le chiavi pubbliche e private sotto forma di array di byte, grazie ai metodi `ImportCspBlob` e `ExportCspBlob`.

Nell'[esempio 18.7](#) possiamo vedere un'implementazione di tale procedura.

Esempio 18.7

```
byte[] keyDecrypted;
byte[] vectorDecrypted;

//lettura chiave pubblica e privata
string publicPrivate = File.ReadAllText(@"d:\PublicAndPrivate.xml");

//decifratura chiave simmetrica e vettore, con algoritmo asimmetrico
using (RSACryptoServiceProvider asCryptoProvider = new
RSACryptoServiceProvider())
{
    asCryptoProvider.FromXmlString(publicPrivate);
    keyDecrypted = asCryptoProvider.Decrypt(this.keyCrypted, true);
    vectorDecrypted = asCryptoProvider.Decrypt(this.vectorCrypted, true);
}

//decifratura dati riservati con chiave simmetrica decifrata
using (ICryptoTransform encryptor =
    this.cryptoProvider.CreateDecryptor(keyDecrypted,
    vectorDecrypted))
{
    using (Stream stream = File.OpenRead(@"d:\encrypted.txt"))
    {
        using (Stream cryptoStream = new CryptoStream(stream,
            encryptor, CryptoStreamMode.Read))
        {
            using (StreamReader reader = new StreamReader(cryptoStream))
            {
                this.tb3.Text = reader.ReadToEnd();
            }
        }
    }
}
```

Con questo semplice esempio abbiamo illustrato un'implementazione concreta dell'utilizzo della cifratura asimmetrica, nella quale si è cercato di ottimizzare prestazioni ed efficienza.

Come abbiamo compreso, rispetto alla cifratura simmetrica non abbiamo la necessità di scambiare la stessa chiave tra mittente e destinatario: gli attori condividono soltanto la chiave pubblica. In taluni contesti, potrebbe essere necessario aggiungere un ulteriore livello di sicurezza, legando l'autenticità delle chiavi a un **certificato digitale** di sicurezza, rilasciato da un'authority.

Cifratura con algoritmo di hashing

La caratteristica principale delle modalità di cifratura che abbiamo visto finora è quella di poter risalire al valore originale. Con gli algoritmi di hashing, invece, possiamo calcolare un codice di lunghezza fissa in relazione alla **struttura dei byte** di un certo messaggio, sia esso un documento o una semplice stringa.

La **relazione univoca** che si crea tra un messaggio e il proprio hash è dettata dal livello di complessità dell'algoritmo: poiché l'hash ha una lunghezza fissa caratteristica dell'algoritmo stesso, più esteso è l'hash, più difficili sono le possibilità di **collisione** (cioè i casi in cui due dati differenti abbiamo lo stesso hash).

Uno degli utilizzi più comuni degli algoritmi di hashing è l'elaborazione delle **password di autenticazione**, i cui hash vengono memorizzati su database al posto del valore reale immesso dall'utente. In questo caso, per autenticare l'utente, l'applicazione non esegue un confronto diretto tra il valore su database e quello immesso dall'utente ma dello hash del dato proveniente dalla maschera di login. L'aspetto importante di un hash, infatti, è che non è possibile risalire al dato originale, quindi non ci sono rischi sulla manipolazione di tale codice se non attraverso un **dictionary attack**, cioè attraverso un confronto massivo del codice con una serie di codici hash corrispondenti a un esteso dizionario. È proprio per ridurre tali rischi che viene sempre consigliato all'utente di inserire password complesse e di cambiarle periodicamente.

La classe PasswordDeriveBytes, che abbiamo utilizzato negli esempi precedenti, impiega proprio un algoritmo di hashing per creare le chiavi. Inoltre aggiunge due ulteriori livelli di sicurezza, combinando un codice "salt" alla password ed

eseguendo, progressivamente, uno hashing ricorsivo pari al numero di iterazioni desiderato.

Grazie alla relazione univoca che si crea tra un dato e il proprio hash, in una trasmissione possiamo sfruttare il codice per controllare che il dato **non sia stato manipolato** durante una fase intermedia. Questo perché, come abbiamo detto, anche una piccola modifica a un documento causa una diversità nella propria rappresentazione di byte e quindi allo hash corrispondente.

Possiamo sfruttare questa caratteristica, per esempio, durante lo scambio di chiavi di cui abbiamo parlato nei paragrafi precedenti, condividendo anche l'hash di tali chiavi, per assicurarci che queste siano effettivamente corrispondenti all'hash delle chiavi originali, controllando quindi che nessuno si sia interposto nello scambio dati.

Nel .NET troviamo una nutrita serie di classi che implementano algoritmi di hashing con una struttura molto simile a quella che abbiamo visto per la cifratura simmetrica e asimmetrica. Senza entrare nel dettaglio di ciascuna classe, come abbiamo accennato, queste variano in funzione della lunghezza del codice hash elaborato.

La classe più nota è MD5. Essa implementa un algoritmo a 128 bit e quindi genera un hash di 16 byte. Sebbene sia il meno sicuro dal punto di vista delle collisioni, la comodità di questo algoritmo risiede nella rappresentabilità dello hash sotto forma di Guid, trattandosi di un dato di 16 byte.

Questo livello di sicurezza può essere sufficiente per molti contesti. In caso di necessità di una codifica più estesa, possiamo utilizzare le classi SHA160, SHA256, SHA384 o SHA512 che, come possiamo intuire dal nome, implementano algoritmi rispettivamente a 160, 256, 384 e 512 bit. Il loro utilizzo è estremamente semplice: attraverso il metodo `ComputeHash` possiamo passare l'array di byte corrispondenti al dato su cui calcolare l'hash, oppure possiamo passare direttamente lo stream di un documento da elaborare. Nel codice dell'[esempio 18.8](#), possiamo vedere una semplice implementazione che confronta i risultati dei vari algoritmi.

Esempio 18.8

```
byte[] data = Encoding.UTF8.GetBytes(this.tb4.Text);
```

```

byte[] hash1 = MD5.Create().ComputeHash(data);
byte[] hash2 = SHA1.Create().ComputeHash(data);
byte[] hash3 = SHA512.Create().ComputeHash(data);
this.tb5.Text = this.GetHexadecimal(hash1);
this.tb6.Text = this.GetHexadecimal(hash2);
this.tb7.Text = this.GetHexadecimal(hash3);

//rappresentazione esadecimale di un array di byte
public string GetHexadecimal(byte[] hash)
{
    StringBuilder sBuilder = new StringBuilder();

    int i = 0;
    for (i = 0; i <= hash.Length - 1; i++)
    {
        sBuilder.Append(hash[i].ToString("x2"));
    }
    return sBuilder.ToString();
}

```

Nella [Figura 18.3](#) possiamo confrontare visivamente i risultati dell'hashing della parola "p@ssw0rd".

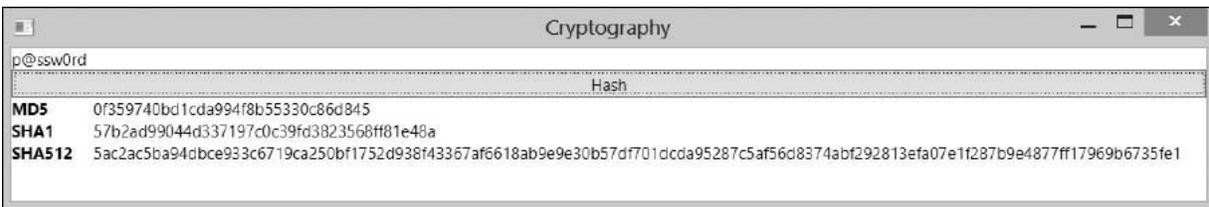


Figura 18.3 – Hashing con vari algoritmi e la loro rappresentazione esadecimale.

Il metodo `ComputeHash` ci restituisce l'hash sotto forma di array di byte; per ottenere una rappresentazione a stringa abbiamo la possibilità di usare il metodo `Convert.ToBase64String` oppure di elaborare i singoli byte dell'hash e convertirli nel loro valore esadecimale, così come abbiamo realizzato nell'[esempio 18.8](#), nel metodo `GetHexadecimal`.

[Firmare gli assembly](#)

Al fine di creare un'identità ben precisa, abbiamo la possibilità di applicare una **firma digitale** ai nostri assembly, in modo da permettere al CLR di verificare l'integrità del codice e bloccarne il caricamento. Il processo di firma passa attraverso la generazione di una **coppia di chiavi, pubblica e**

privata; a differenza della cifratura asimmetrica, di cui abbiamo parlato nei paragrafi precedenti, l'hash dell'assembly viene cifrato con la chiave privata, mentre la chiave pubblica viene inclusa nella libreria stessa.

Al momento del caricamento della libreria, il Framework decifra l'hash dell'assembly con la chiave pubblica inclusa, verifica che l'hash dell'assembly in elaborazione corrisponda con quello decifrato e, in caso di differenze, ne impedisce il caricamento.

La procedura di firma può essere fatta da riga di comando con il tool **sn.exe** incluso nel SDK del .NET Framework, oppure gestita in forma visuale con Visual Studio. Nelle proprietà di un progetto, infatti, troviamo la maschera "Signing", come è possibile vedere nella [Figura 18.4](#).

Con l'apposizione di una firma digitale, all'assembly viene anche attribuito uno **strong name**, che contraddistingue in modo univoco la libreria nei confronti di tutte le altre presenti nel sistema, anche a parità di nome su file system.

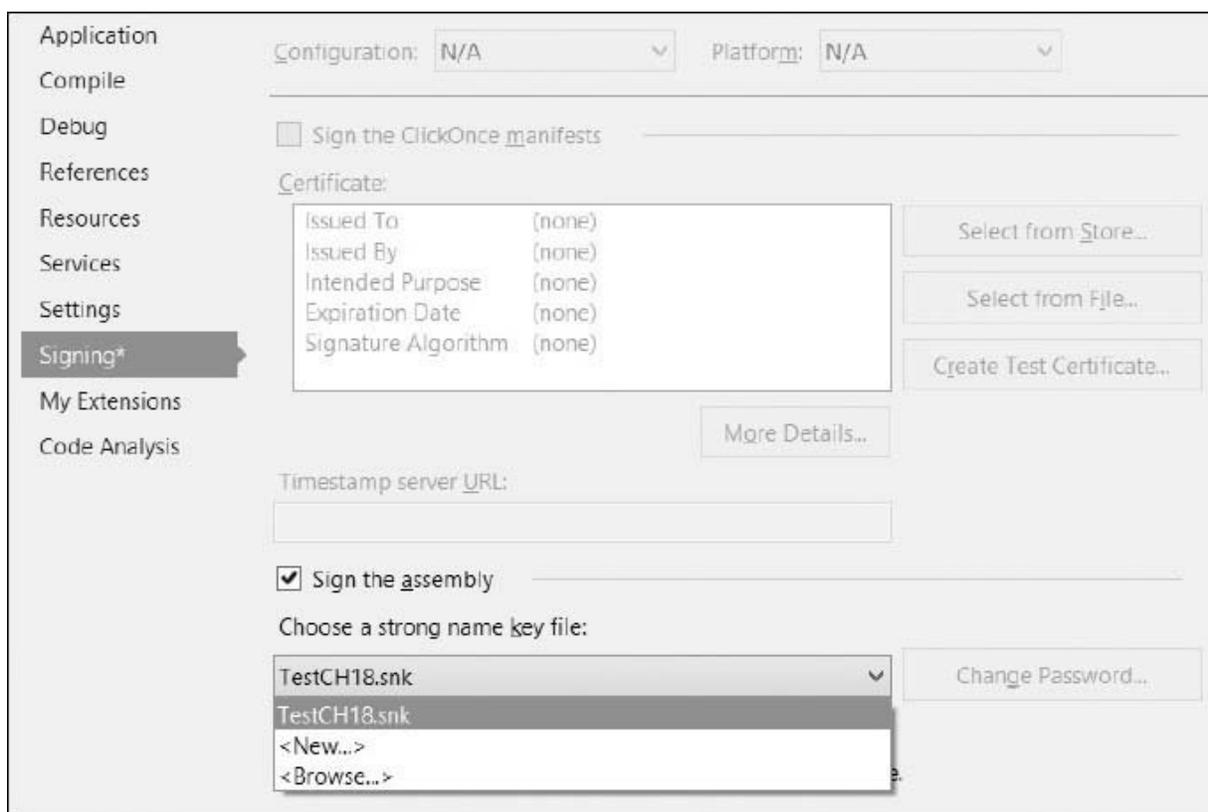


Figura 18.4 – Gestione della firma di un assembly con Visual Studio.

Validazione dei dati immessi dall'utente

Nei paragrafi precedenti abbiamo illustrato i principi di progettazione delle classi e come certi tipi di dati possano essere protetti dall'accesso di utenti non autorizzati. Anche progettando l'applicazione con i criteri più rigidi, possiamo ancora incorrere in violazioni di sicurezza legate all'implementazione delle logiche applicative, in particolare, in quelle legate all'accesso alle basi dati.

Come abbiamo detto all'inizio del capitolo, non dobbiamo trascurare l'uso che l'utente finale può fare della nostra applicazione, ponendo particolare attenzione ai dati che immette nelle maschere, specialmente a quei dati che interagiscono con il database.

Proteggersi da attacchi di tipo SQL Injection

Gli attacchi di SQL Injection si riferiscono a quegli scenari in cui i dati immessi dall'utente confluiscono direttamente nella composizione delle query su database. Senza alcun controllo da parte del programmatore, l'utente potrebbe immettere alcuni codici per andare a creare query inattese e recuperare dati a cui non dovrebbe avere accesso, se non proprio danneggiando l'intera base dati.

Questo scenario si riferisce a un codice scritto come nell'[esempio 18.9](#), nel quale il valore immesso dall'utente nelle `TextBox` viene formattato nella query.

Esempio 18.9

```
'!!!codice insicuro
string query = String.Format("SELECT * FROM Users WHERE Username='{0}'
                              AND Password='{1}'", tbxUsername.Text, tbxPassword.Text);
```

In questo caso, se l'utente inserisce nella `TextBox` il proprio username con la seguente sintassi: "Billy';--" la query che andrà in esecuzione sul database sarà quella dell'[esempio 18.10](#).

Esempio 18.10

```
SELECT * FROM Users WHERE Username='Billy';--
AND Password='valore inutile'
```

Con tale string SQL l'utente avrà una lista dei dati di autenticazione di tutti gli utenti, poiché la parte di codice dopo il carattere "--" non verrebbe elaborata.

A questo fatto e a situazioni simili possiamo far fronte utilizzando l'oggetto `SqlParameter`, in modo da far comporre la query all'oggetto `SqlCommand` di ADO.NET. Possiamo vederne un'implementazione nel codice dell'[esempio 18.11](#).

Esempio 18.11

```
SqlConnection connection = new SqlConnection("stringa di connessione");
string query = "SELECT * FROM Users WHERE Username = @Username";
SqlCommand command = new SqlCommand(query, connection);
SqlParameter param = new SqlParameter("@Username", SqlDbType.NVarChar, 100);
param.Value = tbxUsername.Text;
command.Parameters.Add(param);
SqlDataReader reader = command.ExecuteReader();
```

Nel caso in cui non sia possibile utilizzare le query parametriche, nei contesti in cui si vuole comunque concedere all'utente la possibilità di comporre le proprie query, si deve sempre controllare che i dati immessi siano consoni all'applicazione, effettuando controlli sul tipo con i corrispondenti metodi `TryParse`, disponibili in ogni tipo di valore.

Nonostante possa apparire come una banalità, questo tipo di accorgimento è ignorato da molte applicazioni e siti web, anche importanti, che espongono i dati a visibilità e gestione arbitrarie. È bene ricordare che le vigenti normative sulla privacy e sul trattamento dei dati personali ci

richiedono una particolare attenzione proprio in materia di pubblicazione di dati sensibili, come nell'esempio che abbiamo appena riportato.

Conclusioni

La realizzazione di software sicuro è un argomento sempre più rilevante per la buona **riuscita commerciale** di un prodotto; anche gli utenti stanno maturando la sensibilità a questo tipo di argomento, ponderando il livello di sicurezza come fattore di scelta tra i diversi prodotti presenti sul mercato. Nonostante l'informatica non sia più una scienza sperimentale, le metodologie di sviluppo del software sono ancora abbastanza artigianali, lasciando al singolo programmatore la creatività e l'onere di implementare la logica applicativa e, contemporaneamente, i principi architetturali e quelli di sicurezza.

In questo contesto, .NET ci mette a disposizione delle linee guida e un'infrastruttura di classi utili alla gestione della sicurezza, e la gestione dei dati riservati, con l'implementazione di algoritmi di cifratura simmetrica, asimmetrica e di hashing.

Nel corso del capitolo abbiamo introdotto i principi e l'uso di questi strumenti, cercando di analizzare i casi più frequenti che possiamo incontrare nello sviluppo di applicazioni sicure. Ovviamente l'argomento è molto più vasto di quanto è stato possibile illustrare nel capitolo e, per certi versi, è molto complesso, oltre a essere in costante evoluzione; tuttavia possiamo considerare quanto spiegato come una trattazione propedeutica, tale da rendere più semplice al lettore l'approfondimento dello studio della letteratura specifica, presente anche nella documentazione ufficiale.

Gestione di file e networking

Nei capitoli precedenti abbiamo visto come realizzare diverse tipologie di applicazioni: web, desktop, servizi, e come distribuirli. Spesso può capitare che queste ultime debbano interagire con il file system, per leggere, scrivere o, semplicemente, per eseguire delle ricerche. La prima parte del capitolo è dedicata proprio a questo argomento: vedremo quali sono le classi di .NET che possiamo utilizzare per accedere a file e directory.

In seguito sposteremo la nostra attenzione verso la gestione del registro di Windows, il cui scopo è di memorizzare un gran numero di impostazioni. Capiremo quali sono le modalità per interrogarlo e modificarne il contenuto.

Nella seconda parte, invece, analizzeremo gli strumenti mediante i quali possiamo interagire con le singole tipologie di risorse remote attraverso la rete, per avere così una conoscenza generale di come sia possibile gestire esplicitamente i principali modelli di comunicazione.

Gestione del file system

Con il termine informatico File system si intende il meccanismo che serve a organizzare, recuperare e manipolare le informazioni residenti su un supporto di archiviazione, sia esso un Hard disk, o una memoria esterna.

Sistemi quali Windows, utilizzavano un file system chiamato File Allocation Table (FAT e il suo successore FAT32), il cui scopo, inizialmente, era solo quello di memorizzare informazioni in strutture chiamate file, che potevano essere catalogate all'interno di contenitori detti directory. In seguito, i file system si sono evoluti fino ad arrivare a NTFS, che oggi equipaggia tutte le versioni di Windows ed è in grado di sfruttare supporti a elevata capienza

come quelli odierni e, soprattutto, di fornire servizi aggiuntivi, quali, per esempio, una gestione avanzata della sicurezza e delle permission di accesso alle varie risorse.

In Linux, Unix o macOS, poi, il discorso, grazie alla capacità di .NET Core di essere cross-platform, si complica ulteriormente, con molte più opzioni: a prescindere dal file system effettivamente utilizzato, saranno le librerie di accesso allo stesso che ci astrarranno dalla maggior parte delle differenze tra questi sistemi operativi; a noi, come sviluppatori, non resterà che fare attenzione a ragionare in maniera generica, per esempio facendo buon uso di variabili che impostino i nomi dei file, piuttosto che testando opportunamente il nostro codice su tutte le piattaforme che vogliamo supportare.

Organizziamo le informazioni: Directory e File

Tutti i tipi necessari a gestire file e directory sono raggruppati nel namespace `System.IO` (assembly `System.IO.FileSystem`). In particolare, per quanto riguarda queste ultime, possiamo fare affidamento su due classi: `Directory` e `DirectoryInfo`. La prima espone solo metodi statici e, in tal modo, risulta utile per compiere operazioni saltuarie sulle directory, come crearne o eliminarne una. La seconda, invece, è una classe d'istanza, e il suo uso è pertanto consigliato tutte le volte in cui dobbiamo compiere più operazioni, poiché alcune proprietà e parte del contesto di sicurezza sono valutate solo in fase di creazione della stessa classe; `DirectoryInfo` rappresenta infatti un vero e proprio modello a oggetti di una directory del file system e, come tale, possiamo utilizzarlo, per esempio, come argomento di un metodo, in luogo di una semplice stringa.

Nei prossimi paragrafi vedremo qualche esempio su come creare, spostare, copiare ed eliminare file o directory, mettendo in luce le differenze determinate dall'utilizzo delle due tipologie di oggetti.

Il codice che gira all'interno di .NET Core ha la capacità di essere eseguito su sistemi operativi differenti da Windows: per questo, è importante porre attenzione al fatto che su OS Unix-like, il percorso di file e cartelle è case sensitive e ha un formato differente da quello di Windows. Sono anche differenti alcuni concetti, come quello dei permessi. Raccomandiamo sempre di scrivere codice che

ragioni in modo agnostico rispetto al sistema operativo e rispetti queste differenze.

Creazione di una directory

La creazione di una directory tramite la classe `DirectoryInfo` può essere portata a termine con poche semplici operazioni. Nel codice dell'[esempio 19.1](#), abbiamo creato una nuova istanza della classe `DirectoryInfo`, passando al costruttore una stringa contenente il percorso sul quale vogliamo compiere le nostre operazioni, ossia una directory denominata “Capitolo19” all’interno di “Documenti”.

Esempio 19.1

```
class Program
{
    static void Main(string[] args)
    {
        DirectoryInfo dInfo = new DirectoryInfo(
            Path.Combine(Environment.GetFolderPath(
                Environment.SpecialFolder.MyDocuments), "Capitolo19"));
        try
        {
            dInfo.Create();
            Console.WriteLine("Directory creata correttamente, premi un tasto per spostarla");
            Console.ReadLine();
            if (!Directory.Exists(
                Path.Combine(Environment.GetFolderPath(
                    Environment.SpecialFolder.MyPictures), "Capitolo19")))
            {
                dInfo.MoveTo(Path.Combine(
                    Environment.GetFolderPath(
                        Environment.SpecialFolder.MyPictures), "Capitolo19"));
                Console.WriteLine("Directory spostata correttamente");
            }
            else
            {
                Console.WriteLine("Directory già esistente!");
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine("Si è verificato un errore: {0}", ex.ToString());
        }
        Console.ReadLine();
    }
}
```

A causa delle differenze tra sistemi operativi, non tutti i valori dell'enum `SpecialFolder` hanno una corrispondenza valida. Quando si usano queste informazioni, si raccomanda di verificare che la corrispondenza sia valida. I valori utilizzati nell'esempio corrispondono a scenari supportati su Windows, macOS e Linux.

Invece di comporre il nome completo del percorso, concatenando direttamente le stringhe, abbiamo utilizzato il metodo `Path.Combine`: si tratta di un helper che risulta molto comodo nella manipolazione di questo tipo di stringhe, che ci consente di non doverci preoccupare, per esempio, di includere o verificare la presenza del carattere di separazione dei percorsi.

Una volta ottenuta l'istanza del tipo `DirectoryInfo`, non dobbiamo far altro che richiamarne il metodo `Create` affinché la directory venga fisicamente creata. La [Figura 19.1](#) mostra l'output su console dell'esempio precedente.



Figura 19.1 – L'output della console application.

Nell'[esempio 19.1](#), abbiamo utilizzato il metodo `Environment.GetFolderPath` per recuperare il percorso della directory “Documenti”. Questo metodo accetta un parametro di tipo `SpecialFolder`; si tratta di un enumerato, i cui valori principali sono riassunti nella [Tabella 19.1](#).

Tabella 19.1 – Alcuni dei valori dell'enumeratore `SpecialFolder`

Nome	Significato
------	-------------

ApplicationData	Directory utilizzata per i dati relativi all'applicazione.
Desktop	Directory utilizzata per il Desktop.
ProgramFiles	Directory contenente applicazioni.
MyMusic	Directory contenente musica.
MyPictures	Directory per le immagini.
MyDocuments	Directory con documenti.

È indispensabile comporre i percorsi usando `Environment.GetFolderPath` e l'enumeratore `SpecialFolder`, in modo che possiamo evitare di cablare nel codice il percorso fisico della directory, il quale non varia solo in base alla lingua utilizzata ma potrebbe cambiare tra una versione e l'altra del sistema operativo, dell'ambiente o, addirittura, essere modificato dall'utente.

La classe `Environment` espone il metodo `GetLogicalDrives`, che ci restituisce un array contenente il nome di tutti i drive logici presenti nel sistema. Questo comportamento varia in funzione del sistema operativo.

In un caso come quello appena esaminato, in cui dobbiamo compiere una singola operazione di creazione di una directory, l'uso del tipo statico `Directory` è più immediato e conciso. Come possiamo vedere nell'[esempio 19.2](#), infatti, in questo caso è sufficiente invocare il metodo `CreateDirectory` passando il percorso desiderato.

Esempio 19.2

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            Directory.CreateDirectory(
                Path.Combine(Environment.GetFolderPath(
                    Environment.SpecialFolder.MyDocuments),
                    "Capitolo19"));
            Console.WriteLine("Directory creata correttamente");
        }
        catch (Exception ex)
        {
            Console.WriteLine("Impossibile creare la directory: {0}", ex.
                ToString());
        }
    }
}
```

```
        Console.ReadLine();
    }
}
```

Aggiungendo l'assembly `System.IO.FileSystem.AccessControl`, è possibile utilizzare i metodi `Create` e `CreateDirectory`, passando un parametro del tipo `DirectorySecurity`, così da poter specificare il livello di protezione e le autorizzazioni per la directory appena creata. In questo caso, però, pur rimanendo, il codice .NET Standard non sarà più portatile su sistemi operativi diversi da Windows.

[Eliminare una directory](#)

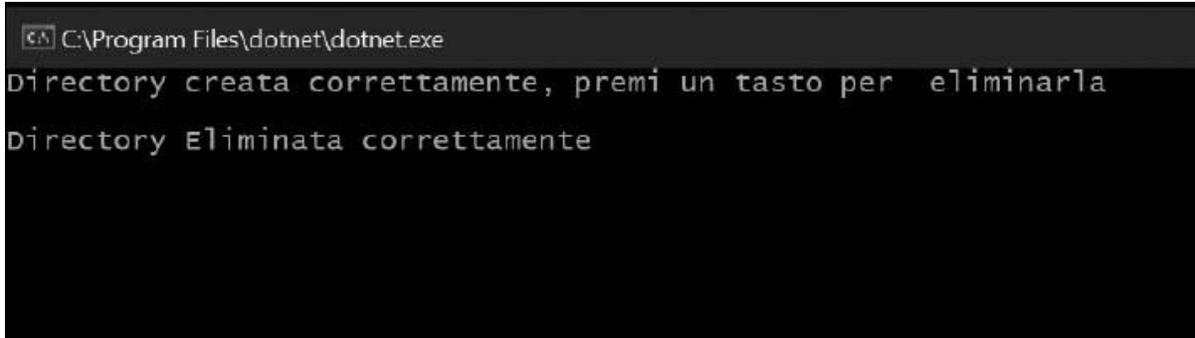
Nel paragrafo precedente abbiamo visto come sia semplice creare una directory; l'eliminazione è assolutamente analoga e non presenta, come possiamo vedere nell'[esempio 19.3](#), particolari difficoltà.

Esempio 19.3

```
class Program
{
    static void Main(string[] args)
    {
        DirectoryInfo dInfo = new DirectoryInfo(
            Path.Combine(Environment.GetFolderPath(
                Environment.SpecialFolder.MyDocuments), "Capitolo19"));
        try
        {
            dInfo.Create();
            Console.WriteLine("Directory creata correttamente, premi un tasto  per eliminarla");
            Console.ReadLine();
            dInfo.Delete();
            Console.WriteLine("Directory Eliminata correttamente");
        }
        catch (Exception ex)
        {
            Console.WriteLine("Si è verificato un errore: {0}", ex.ToString());
        }
        Console.ReadLine();
    }
}
```

Il codice dell'[esempio 19.3](#), infatti, riutilizza interamente il codice che abbiamo visto nel paragrafo precedente e, più precisamente, nell'[esempio 19.1](#), per costruire un'istanza di `DirectoryInfo` tramite la quale creare una directory

su file system. In seguito possiamo procedere alla sua eliminazione, invocando il metodo `Delete`. La [Figura 19.2](#) mostra l'output di questo esempio sulla console.



```
C:\Program Files\dotnet\dotnet.exe
Directory creata correttamente, premi un tasto per eliminarla
Directory Eliminata correttamente
```

Figura 19.2 – L'output della console application.

Il codice dell'[esempio 19.4](#) è analogo a quello dell'[esempio 19.3](#) ma sfrutta la classe `Directory`; esso rappresenta un primo caso in cui si nota il vantaggio della soluzione basata sull'istanza rispetto a questo approccio che sfrutta metodi statici. In questo caso, infatti, siamo costretti a utilizzare una variabile temporanea per memorizzare il percorso della directory, così da passarlo sia al metodo `Create` sia a quello `Delete`.

Esempio 19.4

```
class Program
{
    static void Main(string[] args)
    {
        string myPath = Path.Combine(
            Environment.GetFolderPath(
                Environment.SpecialFolder.MyDocuments), "Capitolo19");
        try
        {
            Directory.CreateDirectory(myPath);
            Console.WriteLine("Directory creata correttamente, premi un tasto per
                eliminarla");
            Console.ReadLine();
            Directory.Delete(myPath);
            Console.WriteLine("Directory Eliminata correttamente");
        }
        catch (Exception ex)
        {
            Console.WriteLine("Impossibile creare la directory: {0}", ex.
                ToString());
        }
        Console.ReadLine();
    }
}
```

```
}
```

Quando si eliminano delle directory con il codice che abbiamo mostrato negli esempi 19.3 e 19.4, proprio come avviene dal prompt dei comandi, dobbiamo prestare attenzione al fatto che, nel caso in cui le directory non siano vuote, viene sollevata una `IOException`.

Nell'[esempio 19.5](#) abbiamo utilizzato il metodo `CreateSubdirectory` per aggiungere una sotto directory alla directory creata tramite `DirectoryInfo`.

Esempio 19.5

```
class Program
{
    static void Main(string[] args)
    {
        DirectoryInfo dInfo = new DirectoryInfo(
            Path.Combine(Environment.GetFolderPath(
                Environment.SpecialFolder.MyDocuments), "Capitolo19"));
        try
        {
            dInfo.Create();
            dInfo.CreateSubdirectory("Esempio 19.5");
            Console.WriteLine("Directory e sotto directory creata correttamente, ←
            premi un tasto per eliminarla");
            Console.ReadLine();
            dInfo.Delete(true);
            Console.WriteLine("Directory Eliminata correttamente");
        }
        catch (Exception ex)
        {
            Console.WriteLine("Si è verificato un errore: {0}", ex.ToString());
        }
        Console.ReadLine();
    }
}
```

L'utilizzo del parametro `True` nel metodo `Delete` ci consente di procedere comunque alla cancellazione. Passando, invece, il valore `False`, si ottiene l'output che possiamo vedere nella [Figura 19.3](#).



```
Microsoft Visual Studio Debug Console
Directory e sotto directory creata correttamente, premi un tasto per eliminarla
Directory Eliminata correttamente
```

Figura 19.3 – L'output della console application quando si tenta di eliminare una directory non vuota.

Nei termini del risultato ottenuto, passare `False` al metodo `Delete(bool recursive)` è del tutto equivalente a richiamare il metodo `Delete()`.

Per quanto riguarda le operazioni di creazione ed eliminazione viste finora, è piuttosto indifferente utilizzare la classe `Directory` o `DirectoryInfo`. Nelle prossime pagine vedremo invece che, per compiere operazioni più complesse quali lo spostamento o la copia, è necessario utilizzarle entrambe secondo le necessità.

Spostare una directory

Possiamo spostare una directory mediante il metodo `Move`, esposto dalla classe `Directory` o il metodo `MoveTo` (`string destDirName`), esposto dalla classe `DirectoryInfo`. Nell'[esempio 19.6](#), dopo aver creato la directory, ne indichiamo il nuovo percorso mediante il metodo `MoveTo`. È importante che, all'interno di quest'ultimo, non esista già una directory con il nome che vogliamo utilizzare; in caso contrario, verrà sollevata un'eccezione e l'operazione sarà abortita.

Esempio 19.6

```
class Program
{
    static void Main(string[] args)
    {
        DirectoryInfo dInfo = new DirectoryInfo(
            Path.Combine(Environment.GetFolderPath(
                Environment.SpecialFolder.MyDocuments), "Capitolo19"));
        try
        {
            dInfo.Create();
            Console.WriteLine("Directory creata correttamente, premi un tasto 
per spostarla");
            Console.ReadLine();
            dInfo.MoveTo(Path.Combine(
                Environment.GetFolderPath(
                    Environment.SpecialFolder.MyPictures), "Capitolo19"));
            Console.WriteLine("Directory spostata correttamente");
        }
        catch (Exception ex)
        {
            Console.WriteLine("Si è verificato un errore: {0}", ex.ToString());
        }
        Console.ReadLine();
    }
}
```

```
}
```

L'[esempio 19.7](#) mostra il medesimo codice dell'esempio precedente, controllando preventivamente l'esistenza della directory destinazione e annullando, eventualmente, l'operazione.

Esempio 19.7

```
class Program
{
    static void Main(string[] args)
    {
        DirectoryInfo dInfo = new DirectoryInfo(
            Path.Combine(Environment.GetFolderPath(
                Environment.SpecialFolder.MyDocuments), "Capitolo19"));
        try
        {
            dInfo.Create();
            Console.WriteLine("Directory creata correttamente, premi un tasto ←
per spostarla");
            Console.ReadLine();
            if (!Directory.Exists(
                Path.Combine(Environment.GetFolderPath(
                    Environment.SpecialFolder.MyPictures), "Capitolo19")))
            {
                dInfo.MoveTo(
                    Path.Combine(Environment.GetFolderPath(
                        Environment.SpecialFolder.MyPictures), "Capitolo19"));
                Console.WriteLine("Directory spostata correttamente");
            }
            else
            {
                Console.WriteLine("Directory già esistente");
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine("Si è verificato un errore: {0}", ex.ToString());
        }
        Console.ReadLine();
    }
}
```

Possiamo verificare l'esistenza di una directory mediante l'utilizzo del metodo `Exists`, esposto dalla classe `Directory`. Esso accetta un parametro di tipo `string`, che rappresenta il percorso della directory di cui verificare l'esistenza.

[Copiare una directory](#)

Come possiamo vedere nell'esempio che segue, copiare una directory richiede qualche riga di codice in più, poiché non esiste un metodo specifico.

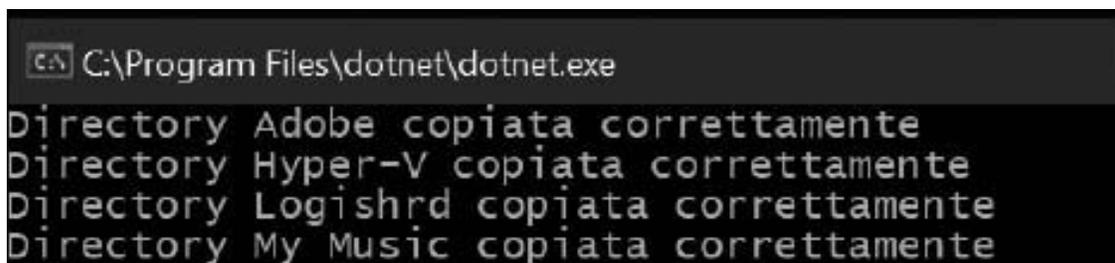
Nell'[esempio 19.8](#), come prima cosa recuperiamo tutte le directory presenti nella directory; a questo scopo utilizziamo il metodo `GetDirectories`, esposto dalla classe `DirectoryInfo`.

Esempio 19.8

```
class Program
{
    static void Main(string[] args)
    {
        DirectoryInfo[] directories = new DirectoryInfo(
            Environment.GetFolderPath(
                Environment.SpecialFolder.MyDocuments)).GetDirectories();
        try
        {
            foreach (DirectoryInfo directoryInfo in directories)
            {
                string copyPath = Path.Combine(
                    Environment.GetFolderPath(
                        Environment.SpecialFolder.MyDocuments),
                    string.Format("{0} copia", directoryInfo.Name));
                Directory.CreateDirectory(copyPath);
                Console.WriteLine(string.Format("Directory {0} copiata ←
                    correttamente",
                        directoryInfo.Name));
                FileInfo[] files = directoryInfo.GetFiles();
                foreach (FileInfo fileInfo in files)
                {
                    fileInfo.CopyTo(Path.Combine(
                        copyPath, string.Format("{0} copia{1} ",
                            Path.GetFileNameWithoutExtension(fileInfo.Name),
                            Path.GetExtension(fileInfo.Name)));
                    Console.WriteLine(string.Format("File {0} copiato correttamente",
                        fileInfo.Name));
                }
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine("Si è verificato un errore: {0}", ex.ToString());
        }
        Console.ReadLine();
    }
}
```

Proseguiamo iterando la collezione di oggetti `DirectoryInfo`, restituita dal metodo `GetDirectories`. Se possiamo accedere alla directory, ne creiamo una nuova con il metodo `CreateDirectory`, alla quale passiamo il nome della directory corrente. Sempre sull'oggetto `DirectoryInfo` corrente, utilizziamo il metodo `GetFiles` per recuperare i file presenti nella directory. Il metodo

GetFiles restituisce una collezione di oggetti FileInfo; su ognuno di essi richiamiamo il metodo CopyTo per eseguire la copia.



```
C:\Program Files\dotnet\dotnet.exe
Directory Adobe copiata correttamente
Directory Hyper-V copiata correttamente
Directory Logishrd copiata correttamente
Directory My Music copiata correttamente
```

Figura 19.4 – L’output della console dopo la copia della directory.

La [Figura 19.4](#) mostra l’output prodotto dalla console application, che abbiamo realizzato nell’[esempio 19.8](#).

Nell’[esempio 19.8](#), per eseguire la copia del file, abbiamo utilizzato un’istanza della classe FileInfo, che espone metodi e proprietà per la gestione dei file. Al metodo CopyTo abbiamo passato una stringa, che è il risultato della concatenazione del percorso della copia della directory, più il nome del file corrente ma sprovvisto di estensione, che abbiamo rimosso mediante l’utilizzo del metodo Path.GetFileNameWithoutExtension, seguito dalla parola “copia” e, infine, l’estensione originale del file, recuperata mediante il metodo Path.GetExtension.

Negli esempi precedenti, abbiamo più volte incontrato la classe Path quando il codice richiedeva la manipolazione del percorso di file e directory. Ma quali sono i vantaggi insiti nell’utilizzo della classe Path quando possiamo utilizzare i numerosi metodi esposti dalla classe string per ottenere gli stessi risultati? La classe Path rappresenta un modo sicuro e testato di manipolare i percorsi, un’operazione delicata che, se non eseguita correttamente, può esporre la nostra applicazione a problemi di sicurezza come, per esempio, problemi di path canonicalization. La [Tabella 19.2](#) riepiloga i metodi più frequentemente utilizzati della classe Path.

Tabella 19.2 – Membri più comunemente utilizzati della classe Path

Nome	Significato
Combine (e overload)	Permette di combinare due o più parti di un percorso, aggiungendo, se necessario, il simbolo /

GetExtension	Recupera l'estensione del file, dato il percorso
GetFileName	Recupera il nome del file con l'estensione, dato il percorso
GetFileNameWithoutExtension	Recupera il nome del file prima dell'estensione
GetInvalidFileNameChars	Ottiene un'array dei caratteri non consentiti nei nomi dei file, utile per controllare che il nome di un file non li contenga
GetRandomFileName	Crea in modo random il nome di un file o di una directory
GetDirectoryName	Recupera il nome della directory dato il percorso

La creazione, l'eliminazione, lo spostamento e la copia non sono le uniche operazioni che è possibile effettuare sul file system. Nel prossimo paragrafo vedremo come eseguire ricerche e filtrare il contenuto di una directory.

Nel caso in cui sia necessario monitorare eventuali modifiche apportate al file system da altre applicazioni in esecuzione nel sistema, .NET mette a disposizione la classe `FileSystemWatcher`, che abbiamo avuto modo di illustrare nel [Capitolo 16](#).

[Eseguire ricerche sul file system](#)

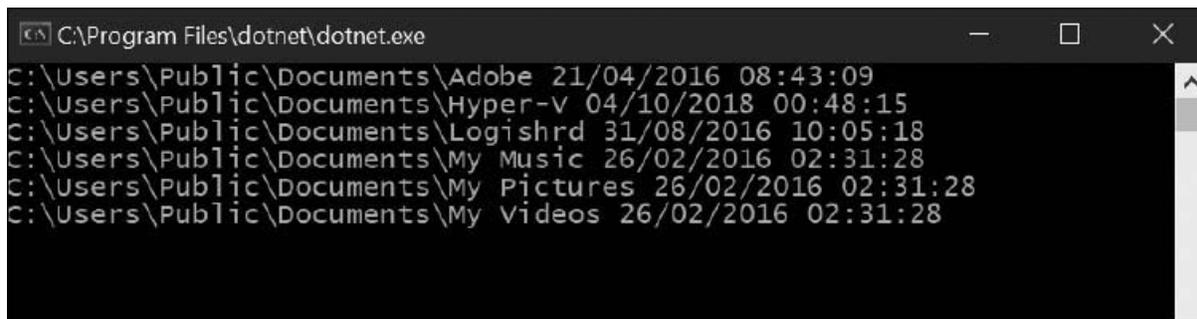
Una necessità piuttosto comune per le applicazioni che lavorano con file e cartelle è quella di effettuare ricerche. Le classi `Directory` e `DirectoryInfo` espongono allo scopo una serie di metodi, che possiamo utilizzare come nell'[esempio 19.9](#).

Esempio 19.9

```
class Program
{
    static void Main(string[] args)
    {
        DirectoryInfo dInfo = new DirectoryInfo(
            Environment.GetFolderPath(
                Environment.SpecialFolder.MyDocuments));
        try
        {
            DirectoryInfo[] directories = dInfo.GetDirectories("*",
                SearchOption.TopDirectoryOnly);
            foreach (DirectoryInfo directory in directories)
            {
                Console.WriteLine("{0} {1}", directory.FullName,
                    directory.LastAccessTime);
            }
        }
    }
}
```

```
catch (Exception ex)
{
    Console.WriteLine("Si è verificato un errore: {0}", ex.ToString());
}
Console.ReadLine();
}
```

L'[esempio 19.9](#) è molto simile ai precedenti: per iniziare, creiamo un'istanza della classe `DirectoryInfo`. In seguito, per ottenere una lista delle directory, utilizziamo il metodo `GetDirectory`, al quale passiamo due argomenti: il primo, del tipo `string`, rappresenta i criteri di ricerca, il secondo, del tipo `SearchOption`, indica di non propagare la ricerca alle sotto directory. Possiamo vedere il risultato della ricerca nella [Figura 19.5](#).



```
C:\Program Files\dotnet\dotnet.exe
C:\Users\Public\Documents\Adobe 21/04/2016 08:43:09
C:\Users\Public\Documents\Hyper-V 04/10/2018 00:48:15
C:\Users\Public\Documents\Logishrd 31/08/2016 10:05:18
C:\Users\Public\Documents\My Music 26/02/2016 02:31:28
C:\Users\Public\Documents\My Pictures 26/02/2016 02:31:28
C:\Users\Public\Documents\My Videos 26/02/2016 02:31:28
```

Figura 19.5 – L'output della console dopo la ricerca.

Nell'[esempio 19.8](#) ci siamo limitati a enumerare solamente le directory. Le modifiche da apportare al codice per recuperare i file sono semplicissime e consistono nell'utilizzare il metodo `GetFiles`. Una delle operazioni più comuni sui file, oltre allo spostamento e la copia, è la creazione e la successiva modifica del suo contenuto. Questi saranno gli argomenti del prossimo paragrafo.

Creare e modificare un file

La gestione dei file, la loro creazione, il loro spostamento o eliminazione, come abbiamo avuto modo di vedere negli esempi precedenti, avvengono mediante il tipo `File` o `FileInfo`. Nell'[esempio 19.10](#) abbiamo utilizzato il metodo `File.Create` per creare un file dal nome "myFile.txt".

Esempio 19.10

```
class Program
{
    static async Task Main(string[] args)
    {
        try
        {
            string myPath = Path.Combine(
                Environment.GetFolderPath(
                    Environment.SpecialFolder.MyDocuments), "myfile.txt");
            using(var myFileStream = File.Create(myPath))
            {
                byte[] info = new UTF8Encoding(true).GetBytes("Un saluto dall'autore
di questo capitolo.");
                await myFileStream.WriteAsync(info, 0, info.Length);
                myFileStream.Close();
                using (StreamReader myStreamReader = File.OpenText(myPath))
                {
                    while (myStreamReader.Peek() >= 0)
                    {
                        Console.WriteLine("Il file contiene: {0}",
                            myStreamReader.ReadLine());
                    }
                }
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine("Si è verificato un errore: {0}", ex.ToString());
        }
        Console.ReadLine();
    }
}
```

Il metodo `File.Create` restituisce un'istanza della classe `FileStream`. Questa classe ci consente di manipolarne il contenuto, accodando del testo tramite il metodo `Write`, dopo averlo **codificato** in formato UTF8.

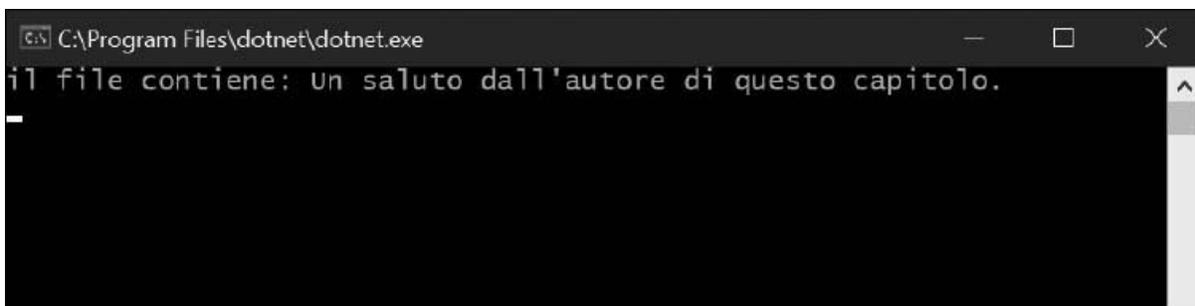


Figura 19.6 – Il contenuto del file, visualizzato nella console application.

La [Figura 19.6](#) mostra il risultato della creazione del file e dell'inserimento del testo. Bisogna prestare attenzione al fatto che, nel caso in cui il file sia già presente, l'esecuzione del metodo `File.Create` ne comporta la sovrascrittura. Pertanto, è sempre meglio verificarne preventivamente l'esistenza tramite il metodo `File.Exists`.

Nell'[esempio 19.10](#) abbiamo utilizzato il tipo `FileStream`. Più in generale, uno stream è un'astrazione che produce o consuma informazioni. Tutti i tipi di stream presentano un medesimo comportamento, ereditato dalla classe base `Stream`. Ciò permette di utilizzare tutti gli stream allo stesso modo, anche se provenienti da fonti differenti.

Al livello più basso, tutti gli stream operano sui byte; noi esseri umani non siamo abituati a lavorare direttamente sui byte e, per questo motivo, .NET definisce diverse classi che sono in grado di convertire un flusso di byte in un flusso di caratteri, permettendoci di leggere e scrivere, per esempio, le informazioni contenute in un file.

È importante porre attenzione alla scelta della directory nella quale salvare i dati della nostra applicazione: esistono directory per il cui accesso in scrittura il sistema operativo richiede di impersonare un utente con privilegi amministrativi. Una possibile alternativa, valida per le applicazioni desktop che girano all'interno di .NET (solo su Windows), può essere quella di utilizzare l'`IsolatedStorage`, ossia una porzione di disco che Windows riserva alle applicazioni.

[L'Isolated Storage di Windows](#)

Quando scegliamo un percorso per memorizzare i dati della nostra applicazione, senza saperlo, stiamo prendendo una decisione delicata. Infatti, non possiamo escludere a priori che quel percorso sia già utilizzato o che sarà utilizzato in futuro. Questa situazione può portare conseguenze difficilmente calcolabili, specialmente se la congruenza delle informazioni è un requisito fondamentale.

Per ovviare a questo inconveniente possiamo utilizzare la classe `IsolatedStorageFile`. Tale classe rappresenta un compartimento stagno, basato sull'identità dell'`Assembly` e dell'utente e perciò non esposto

all'interferenza di altre applicazioni. Naturalmente l'IsolatedStorage non è magica: il supporto di memorizzazione è sempre il disco del nostro computer. Semplificando, quello che fa l'IsolatedStorage è impedire che le informazioni della nostra applicazione siano distrutte da altre. Nel prossimo esempio vedremo come ottenere uno store, creare una directory e aggiungere un nuovo file.

L'[esempio 19.11](#) prende spunto dal precedente 19.10, ma è stato modificato per utilizzare la classe `IsolatedStorageFile`. Questa classe non può essere istanziata direttamente e per ottenere un riferimento valido è necessario utilizzare il metodo statico `GetStore`; in seguito, controlliamo che lo storage contenga una directory dal nome "Documenti" e, in caso negativo, procediamo alla sua creazione mediante il metodo `CreateDirectory`.

Esempio 19.11

```
class Program
{
    async static Task Main(string[] args)
    {
        try
        {
            IsolatedStorageFile isoFile = IsolatedStorageFile.GetStore(
                IsolatedStorageScope.User |
                IsolatedStorageScope.Assembly |
                IsolatedStorageScope.Domain, null);
            if (!isoFile.DirectoryExists("Documenti"))
            {
                isoFile.CreateDirectory("Documenti");
            }
            IsolatedStorageFileStream isoStream =
                isoFile.CreateFile("myFile.txt");
            byte[] info = new UTF8Encoding(true)
                .GetBytes("Un saluto dall'autore di questo capitolo.");
            isoStream.Write(info, 0, info.Length);
            isoStream.Close();
            using(StreamReader mySteamReader = new StreamReader(
                isoFile.OpenFile("myFile.txt", FileMode.Open))
            {
                while (mySteamReader.Peek() >= 0)
                {
                    Console.WriteLine("il file contiene: {0}", mySteamReader.
                        ReadLine());
                }
                mySteamReader.Close();
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine("Si è verificato un errore: {0}", ex.ToString());
        }
        Console.ReadLine();
    }
}
```

```
}  
}
```

Quest'ultimo passo non differisce molto da quanto fatto in precedenza. Lo stesso vale per il file, che possiamo creare mediante l'utilizzo del metodo `CreateFile`, esposto dalla classe `IsolatedStorageFile`.

La lettura avviene in maniera del tutto analoga all'esempio precedente, utilizzando la classe `StreamReader`. Come possiamo notare, però, in questo caso abbiamo costruito sfruttando un particolare tipo di stream, `IsolatedStorageFileStream`, che rappresenta un flusso di byte di un file contenuto nell'`IsolatedStorage` e che possiamo ottenere invocando il metodo `IsolatedStorageFile.OpenFile`. L'output dell'applicazione è il medesimo di quello nella [Figura 19.7](#), ma il comportamento è sensibilmente differente, perché siamo riusciti a scrivere su una porzione del disco appositamente dedicata da Windows per la memorizzazione di dati specifici dell'applicazione.

[Il Registry di Windows](#)

Il Registry è un componente centrale di Windows. In esso sono mantenute, sotto forma di una struttura gerarchica, le informazioni vitali per il funzionamento del sistema operativo e per le applicazioni che vi sono installate.

La sua introduzione risale alla versione 3.0 di Windows e vi sono mantenuti profilo utente, hardware e quant'altro prima veniva memorizzato nei file *.INI.

Lavorare con le informazioni contenute nel Registry è un'operazione delicata. L'eliminazione accidentale di una chiave o una modifica inopportuna possono compromettere il corretto funzionamento del sistema, fino a causarne il blocco. È per questo motivo che, prima di qualsiasi modifica al Registry, è consigliabile fare una copia dello stesso, operazione che possiamo compiere utilizzando l'applicazione `RegEdit.exe`.

Prima di interagire con il registro è indispensabile conoscerne alcuni aspetti. Come abbiamo detto, le informazioni in esso contenute sono organizzate in forma gerarchica; ogni elemento della gerarchia è chiamato chiave (key) e ogni chiave contiene una serie di valori (values).

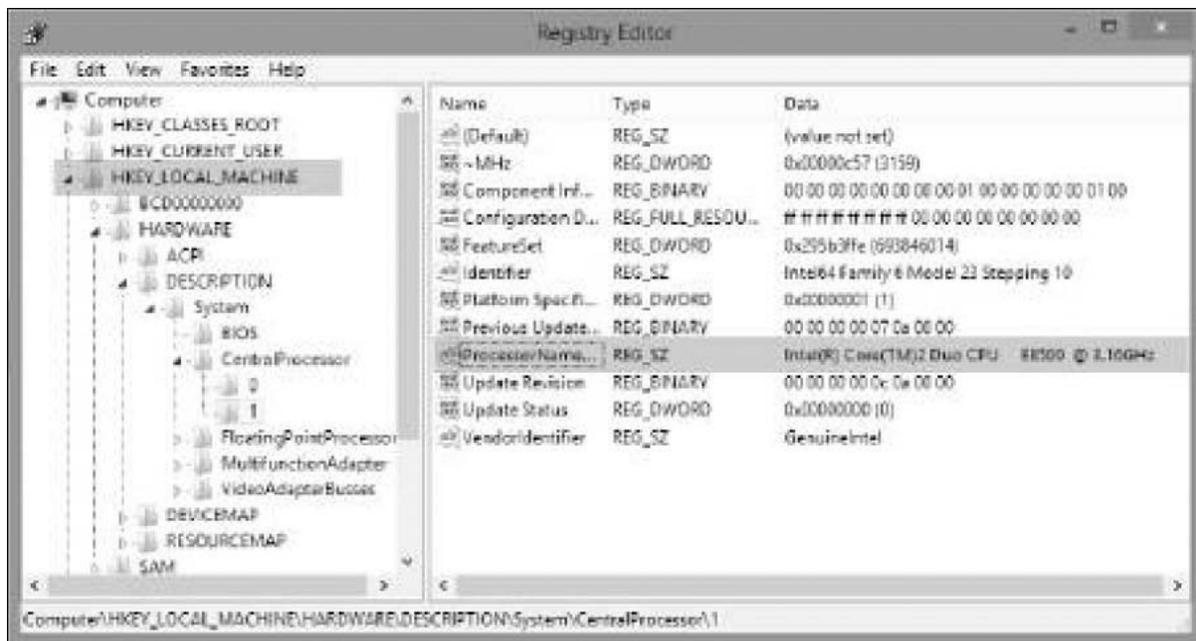


Figura 19.7 – L'applicazione RegEdit.

Nella [Figura 19.7](#) possiamo vedere l'interfaccia dell'applicazione RegEdit. Sulla sinistra troviamo la struttura gerarchica mediante la quale è organizzato il Registry. Sulla destra i dettagli della chiave selezionata.

La gerarchia si compone di un nodo principale, il quale rappresenta il nostro computer e da cinque sotto chiavi:

- HKEY_CLASSES_ROOT contiene le informazioni memorizzate circa le applicazioni registrate e le associazioni con le estensioni dei file;
- HKEY_CURRENT_USER contiene le informazioni relative al profilo utente corrente, linkandole dalla chiave HKEY_USERS;
- HKEY_LOCAL_MACHINE contiene le informazioni comuni a tutti gli utenti come, per esempio, le configurazioni hardware del computer;
- HKEY_USERS contiene gli utenti registrati nel sistema;
- HKEY_CURRENT_CONFIG contiene le informazioni relative al runtime, non persistite su disco e ottenute durante il boot della macchina.

Possiamo eseguire un backup completo del Registry attraverso la voce Export, che troviamo sotto il menu File.

In ambiente .NET possiamo leggere , aggiornare e creare nuove chiavi del Registry mediante la classe RegistryKey, che troviamo nel namespace Microsoft.Win32.

Lo scopo dell'[esempio 19.12](#) è quello di leggere la chiave di registro evidenziata nella [Figura 19.8](#). Il punto di partenza è la chiave LOCAL_MACHINE, cui è possibile accedere utilizzando il corrispondente field statico LocalMachine.

Esempio 19.12

```
class Program
{
    static void Main(string[] args)
    {
        RegistryKey registrykey = Registry.LocalMachine;
        registrykey = registrykey.OpenSubKey("HARDWARE\\\\"DESCRIPTION\\"\\
System\\"\\
CentralProcessor\\"\\0");
        object value = registrykey.GetValue("ProcessorNameString");
        Console.WriteLine("Il tuo processore è:" + value);
        Console.ReadLine();
    }
}
```

Successivamente, tramite il metodo OpenSubKey è possibile addentrarsi nei dettagli, fino a recuperare il valore della chiave ProcessorNameString, che possiamo mostrare a video.

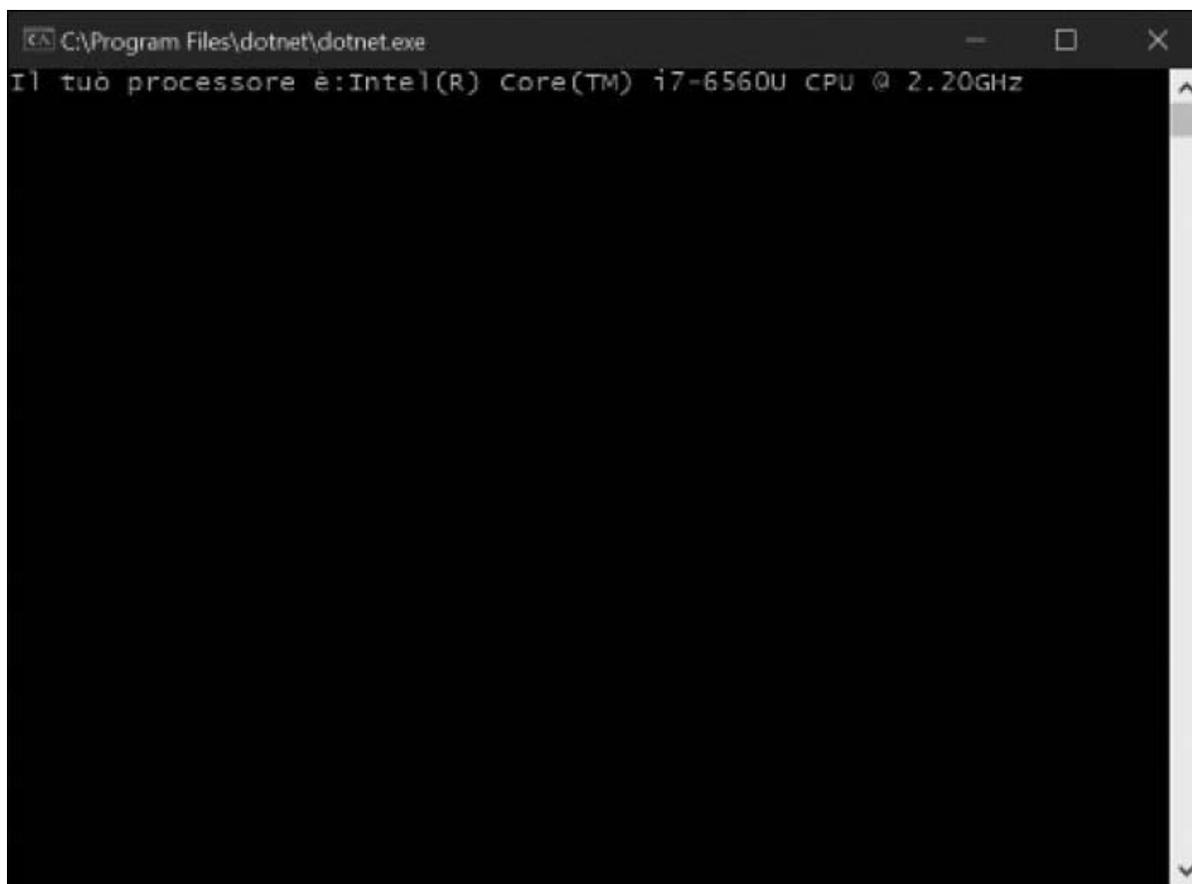


Figura 19.8 – Il tipo di processore installato, visualizzato nella Console Application.

L'immagine 19.8 mostra il risultato dell'[esempio 19.12](#), tramite il quale siamo riusciti a determinare la tipologia del processore installato.

Possiamo inoltre aggiungere e rimuovere una chiave nel registro, utilizzando i metodi `CreateSubKey` e `DeleteSubKey`, ponendo molta attenzione a cosa andiamo a eliminare con quest'ultimo metodo; si tratta infatti di modifiche irreversibili e pertanto, soprattutto utilizzando `DeleteSubKey`, bisogna essere molto accorti.

La creazione di una chiave è semplice come la sua lettura: l'[esempio 19.13](#) mostra come creare una nuova chiave e aggiungere il valore al suo interno.

Per utilizzare le classi che permettono l'accesso al registro, è necessario aggiungere il pacchetto nuget `Microsoft.Win32.Registry`. Questa operazione rende il progetto compatibile solo con i sistemi operativi Windows, limitando di fatto la portabilità della nostra applicazione.

Esempio 19.13

```
class Program
{
    static void Main(string[] args)
    {
        RegistryKey chapter19 = Registry.CurrentUser.CreateSubKey("Capitolo19");
        RegistryKey autor = chapter19.CreateSubKey("Autore");
        autor.SetValue("Name", "Marco");
        Console.WriteLine("Chiave creata con successo");
        Console.ReadLine();
    }
}
```

Analogamente agli esempi precedenti, tramite il field statico `CurrentUser` esposto dalla classe `Registry`, possiamo ottenere un riferimento alla chiave `HKEY_CURRENT_USER`, all'interno della quale abbiamo creato le due nuove chiavi "Capitolo19" e "Autore", utilizzando il metodo `CreateSubKey`. Infine, con il metodo `SetValue` abbiamo inserito un nuovo valore, identificato dalla parola "Name" e con il valore "Marco".

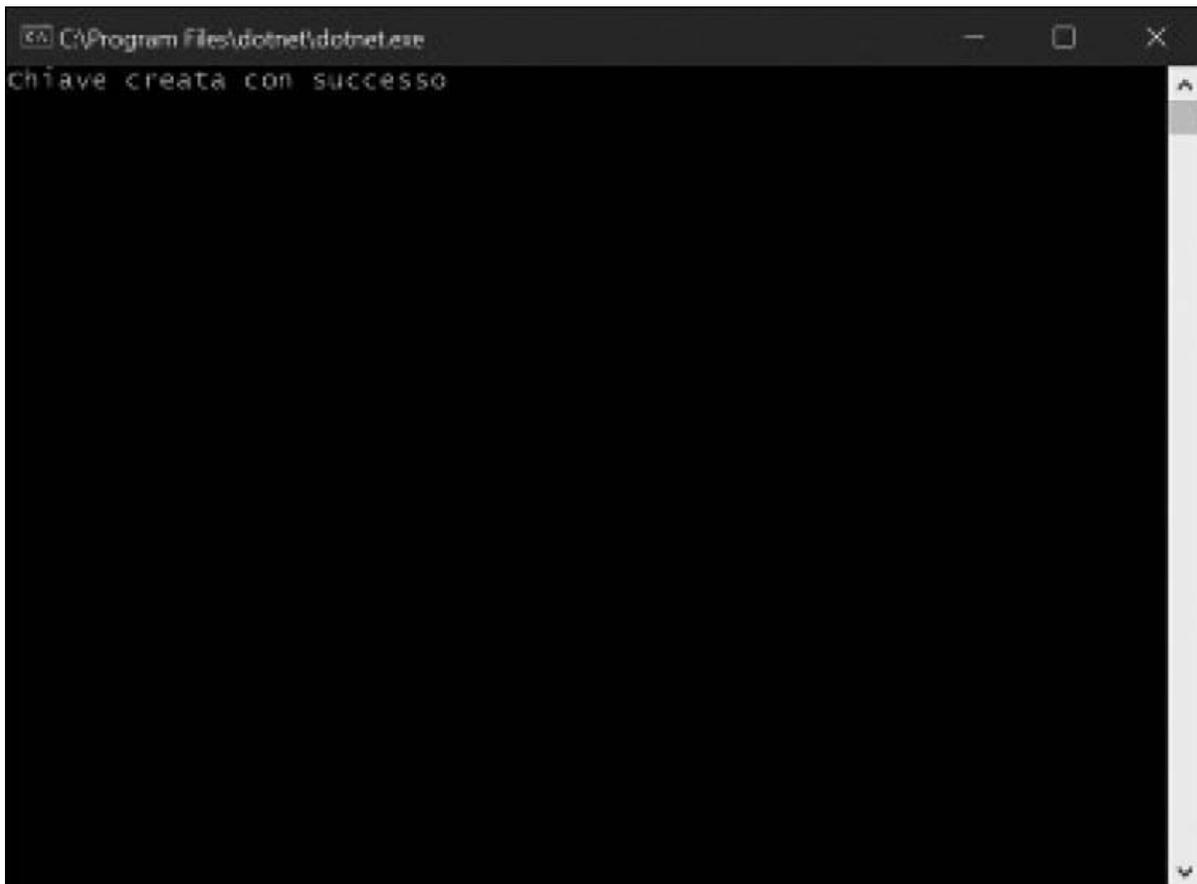


Figura 19.9 – Il risultato dell'[esempio 19.13](#).

La [Figura 19.9](#) mostra la chiave e il valore che sono stati aggiunti al Registry dopo l'esecuzione dell'[esempio 19.13](#).

Il Registry può essere utilizzato nelle applicazioni .NET Core per Windows, per salvare e recuperare le informazioni di configurazione, e offre il vantaggio di mantenere degli store separati per utente, oltre a uno store globale. Bisogna però porre attenzione al fatto che i dati che può contenere hanno dimensione limitata e che il contenuto deve essere espresso in formato stringa o binario (e in quest'ultimo caso, quindi, non leggibile se non con un apposito tool). Quindi, per mantenere la compatibilità e la portabilità su altri sistemi se ne sconsiglia l'uso, a favore di altri sistemi.

Principi di comunicazione di rete

La necessità di mettere in comunicazione più sistemi nasce dall'esigenza di condividere informazioni tra più applicazioni, anche eterogenee o distanti geograficamente. Negli anni settanta, questa esigenza ha portato i ricercatori del **Massachusetts Institute of Technology** (il famoso MIT) a sviluppare un modello di interconnessione “a rete”, di efficienza superiore rispetto a una connessione lineare. In questo modello le informazioni potevano transitare in più percorsi diversi, senza che fosse necessaria la presenza di un sistema specifico.

Le fondamenta di tale architettura si basano sul protocollo di rete **IP** (**Internet Protocol**), che prevede la frammentazione dei dati in più **pacchetti**, al fine di farli transitare all'interno della rete in modo frammentato e autonomo, con il vantaggio di poter essere richiesti nuovamente, in caso di corruzione o perdita.

Nonostante il modello “a rete”, è indispensabile che ogni sistema connesso sia identificabile in maniera univoca rispetto agli altri. Per questo motivo, a ciascuno di essi viene assegnato un **indirizzo IP**, generalmente composto da quattro numeri a tre cifre, separati da un punto.

Per conoscere il proprio indirizzo è sufficiente visualizzare le proprietà di rete, seguendo le procedure del proprio sistema operativo, oppure richiamare “IpConfig” in console, da riga di comando.

Quando un computer non è connesso ad alcuna rete, ha l'indirizzo IP 127.0.0.1. Questo indirizzo viene utilizzato dal sistema operativo proprio per riferirsi a se stesso, in assenza di connessione di rete. Esiste anche un alias, con il nome di localhost.

L'indirizzo del nostro computer all'interno di una rete è univoco, ma solo rispetto agli altri sistemi connessi a tale rete; quando siamo connessi a Internet, il nostro indirizzo non identifica direttamente alcun computer, ma il router che fa da gateway per la connessione.

Architettura a livelli: il modello di trasporto

Possiamo semplificare l'architettura di rete come un modello a quattro "livelli" principali: oltre al livello "fisico", definito dall'hardware, abbiamo già accennato al protocollo di rete IP, il quale costituisce il fondamento, definendo le modalità con cui devono essere organizzati i dati, per rispettare uno dei paradigmi dell'architettura. Il livello sovrastante si occupa di definire la tipologia di scambio dei dati; negli anni, si sono affermati i protocolli TCP e UDP, dei quali parleremo in seguito. Tali protocolli sono specializzati nel definire come i pacchetti debbano essere veicolati e gestiti nelle varie richieste e pertanto possono essere classificati all'interno dei **protocolli di trasporto**.

Infine, il macro-livello sovrastante è costituito dai ben più noti **protocolli applicativi**, come l'HTTP e l'FTP, i quali definiscono le interfacce con le applicazioni, e possiedono caratteristiche specifiche per i tipi di dati e servizi ai quali si rapportano.

Il protocollo HTTP (**Hypertext Transfert Protocol**), per esempio, si basa sul principio di "richiesta e risposta", tipico delle architetture client/server. Sviluppato in ambito Web per la divulgazione di documenti ipertestuali, il client è rappresentato dal browser, il quale invia un messaggio di "richiesta" al server (web), che restituisce i dati attraverso un messaggio di "risposta". Al termine dell'interazione, la connessione viene chiusa automaticamente, senza alcuna gestione dello stato: il protocollo HTTP, infatti, è noto per essere "**stateless**" (senza stato) e la sua diffusione, accoppiata a Internet, ha spinto gli sviluppatori a trovare soluzioni alternative (nei linguaggi stessi) per la gestione dello stato.

Porte e protocolli applicativi standard

Per ogni specifica tipologia di servizi esposti in rete, nati anche in parallelo con i tipi di dati da gestire, è stata sviluppata una moltitudine di protocolli applicativi, spesso creando una duplicazione di funzionalità. Per ricevere contemporaneamente dati eterogenei tra la stessa coppia di host, è stato sviluppato il concetto di **multiplazione** di porte, grazie alla quale ogni protocollo applicativo utilizza una specifica porta (un intero a 16 bit), per distinguere, a livello di trasporto, i propri pacchetti in transito.

Benché del tutto arbitrarie, negli anni si sono consolidate alcune coppie di porte di trasmissione e protocolli applicativi, divenendo in seguito veri e propri **standard**. Nella [Tabella 19.3](#) possiamo leggere un elenco dei protocolli più noti, con le corrispondenti porte di comunicazione.

Tabella 19.3 – Principali protocolli e relative porte di comunicazione

Porta	Protocollo	Utilizzo
21	FTP	Scambio dati
25	SMTP	Invio messaggi e-mail
110	POP3	Ricezione messaggi e-mail
143	IMAP	Ricezione messaggi e-mail
80	HTTP	(HyperText Transfer Protocol) pagine web
119	NNTP	Newsgroup

Tali protocolli sono in costante evoluzione, in parallelo con i cambiamenti e le esigenze tecnologiche che si presentano: lo stesso **IMAP** rappresenta l'evoluzione del **POP3** per la ricezione di posta elettronica, in risposta alla necessità di una gestione più evoluta dei messaggi, anche su dispositivi mobili. Trattandosi di standard di comunicazione, la loro diffusione è molto graduale e diluita nel tempo.

I protocolli TCP e UDP

Dopo una breve introduzione alla comunicazione di rete, parliamo dei due protocolli di trasporto maggiormente utilizzati nelle nostre applicazioni: il **Transmission Control Protocol (TCP)** e lo **User Datagram Protocol (UDP)**.

Come possiamo intuire già dal loro nome, la particolarità del protocollo TCP è il controllo dei dati in transito; nello specifico, tale protocollo prevede che i pacchetti arrivino a destinazione, che non siano corrotti e che arrivino anche nello stesso ordine in cui sono stati inviati. Com'è comprensibile, tale controllo sottopone a un onere percettibile, soprattutto in termini di tempo di trasmissione; per contro, offre una notevole garanzia in materia di qualità della comunicazione. La differenza principale tra il TCP e UDP è proprio il controllo sui dati in transito: in UDP abbiamo una velocità maggiore di comunicazione, proprio per l'assenza di controllo, e i dati vengono inviati sotto forma di **datagrammi**, che il ricevente può ricevere. Tale protocollo trova impiego nelle comunicazioni in cui ha più importanza la **velocità di trasmissione** rispetto alla qualità come, per esempio, nelle trasmissioni **broadcast**, nelle quali è trascurabile perdere alcuni byte, nel complesso della comunicazione.

Il TCP, invece, è molto più adatto nel **trasferimento di file**, per i quali l'attesa è giustificata da un dato completamente valido e utilizzabile dall'applicazione richiedente. Il protocollo HTTP si appoggia a TCP proprio per la necessità che un file **HTML** arrivi al **browser** integro e interpretabile.

I socket e la comunicazione a basso livello

Nella programmazione managed, poniamo la nostra attenzione sul livello più alto di comunicazione, trascurando le modalità di scambio dei pacchetti e concentrandoci sui dati e sulla logica applicativa, mentre il resto delle operazioni a basso livello è gestito dal sistema operativo ed è del tutto trasparente per il programmatore e per l'applicazione stessa.

L'oggetto di livello inferiore, per la gestione delle comunicazioni, è socket, del namespace `System.NET.Sockets`, attraverso il quale abbiamo il controllo capillare di tutte le impostazioni di connessione. Qualora non avessimo la necessità di tale controllo, che comporta alcuni oneri in termini di sviluppo, possiamo utilizzare le seguenti **classi specializzate**, che troviamo già implementate nella BCL del .NET Framework:

- `UdpClient`.
- `TcpClient`.
- `TcpListener`.

Nei paragrafi successivi vedremo come utilizzare queste classi.

Inviare un semplice testo con un client UDP

La classe `UdpClient` consente uno scambio di dati in modalità socket, quindi di basso livello, attraverso il protocollo UDP, senza stato e senza la necessità di attendere la connessione con un host specifico. Nell'[esempio 19.14](#), vediamo come inviare una semplice stringa di testo attraverso un'istanza dell'oggetto `UdpClient`.

Esempio 19.14

```
private async void Button1_Click(object sender, RoutedEventArgs e)
{
    var client = new UdpClient();
    client.Connect("localhost", 8080);
    byte[] sendByte = Encoding.ASCII.GetBytes(TextBox1.Text);
    await client.SendAsync(sendByte, sendByte.Length);
}
```

Per iniziare la connessione, utilizziamo il metodo `Connect` e specifichiamo come parametri sia l'host a cui vogliamo inviare i messaggi sia la porta di comunicazione che useremo: in questo caso inviamo i messaggi alla stessa macchina (`localhost`) attraverso la porta 8080. Questo valore deve essere superiore a 1024, per non andare in conflitto con le porte standard di cui abbiamo parlato, e la porta non deve essere utilizzata da altre applicazioni. L'effettivo invio dei dati avviene attraverso il metodo `SendAsync`, a cui passiamo l'array di byte corrispondenti al testo che vogliamo trasmettere, oltre alla lunghezza dell'array stesso.

Per eseguire questo esempio abbiamo costruito un semplice client WPF, con una `TextBox` in cui possiamo immettere il testo, oltre a un `Button`, di cui gestiamo l'evento `click` con il codice dell'[esempio 19.14](#). Nel corso del capitolo utilizzeremo questo tipo di interfaccia per eseguire gli altri esempi.

Ricevere i messaggi con un mini server UDP

Per ricevere i messaggi possiamo utilizzare la stessa classe `UdpClient` e il metodo `Receive`, non prima di esserci messi in ascolto su una combinazione specifica di indirizzo/porta. Come abbiamo spiegato in precedenza, il protocollo non prevede che sia stabilita una connessione, quindi **la ricezione è**

del tutto svincolata dall'invio e necessita, pertanto, che sia eseguito in continuo. Nell'[esempio 19.15](#), grazie al metodo `ReiceveAsync`, il ciclo `while` può continuare senza che l'interfaccia rimanga bloccata.

Esempio 19.15

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
        this.Loaded += async (o, e) => await ReiceveDataAsync();
    }

    private async void Button1_Click(object sender, RoutedEventArgs e)
    {
        var client = new UdpClient();
        client.Connect("localhost", 8080);
        byte[] sendByte = Encoding.ASCII.GetBytes(this.TextBox1.Text);
        await client.SendAsync(sendByte, sendByte.Length);
    }

    private async Task ReiceveDataAsync()
    {
        var server = new UdpClient(8080);
        while (true)
        {
            var result = await server.ReceiveAsync();
            var reiceveByte = result.Buffer;
            var reiceveString = Encoding.ASCII.GetString(reiceveByte);
            TextBox2.Text += String.Format(" {0}", reiceveString);
        }
    }
}
```

Anche in questo caso abbiamo creato una semplice WPF Application, che abbiamo illustrato nei capitoli precedenti e che chiameremo “Server”: nel costruttore dello `UserControl` principale dichiariamo l'istanza del nuovo `Thread`, il cui metodo `start` inizierà l'esecuzione del metodo `ReiceveData`. In tale metodo, in un ciclo sempre attivo, andiamo a ricevere i dati provenienti da tutti gli host in comunicazione sulla porta 8080, dichiarando un oggetto `IPEndPoint`. Infine, non ci rimane che convertire in stringa i byte ricevuti e stamparli a video, nel `TextBox` inserito appositamente.

Nella [Figura 19.10](#) possiamo vedere l'uso delle applicazioni client e server di esempio.

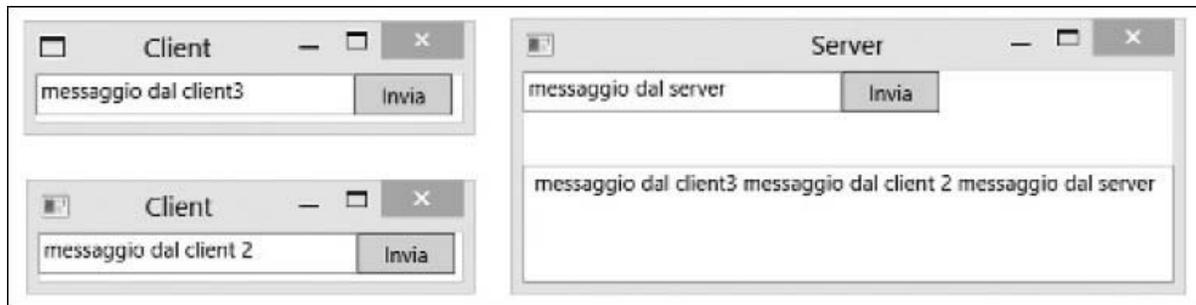


Figura 19.10 – Esempio di scambio dati tra applicazioni con protocollo UDP.

Inviare e ricevere dati con la classe TcpClient

Come abbiamo detto, la natura del protocollo TCP lo rende più indicato per il trasferimento di file e dati, la cui integrità è più importante della velocità di trasmissione. Come per l'UDP, nel .NET Framework esiste una classe socket specializzata per questo protocollo: è la classe `TcpClient`.

A differenza di `UdpClient`, per inizializzare un trasferimento con `TcpClient` è necessario che sia preventivamente stabilita una connessione con un ricevente. Questa operazione può essere eseguita attraverso la classe `TcpListener`, grazie alla quale possiamo attendere e gestire le connessioni provenienti da più client.

Per introdurre le modalità d'uso di queste classi, analizziamo un semplice esempio di trasferimento dati, in modo che più client possano visualizzare su una applicazione server alcune immagini selezionate dall'utente. Come per l'applicazione server dell'[esempio 19.15](#), procediamo alla creazione di una WPF Application e, nel costruttore dell'`UserControl` principale, andiamo ad inizializzare l'oggetto `TcpListener`. L'implementazione è visibile nell'[esempio 19.16](#).

Esempio 19.16

```
public MainWindow()
{
    InitializeComponent();
    this.Loaded += async (o, e) => await ReiceveDataAsync();
}

private async Task ReiceveDataAsync()
{
    var server = new TcpListener(new IPEndPoint(IPAddress.Any, 1234));
    server.Start();
}
```

```

while (true)
{
    var localClient = await server.AcceptTcpClientAsync();
    var netStream = localClient.GetStream();
    if (netStream.CanRead)
    {
        var dataStream = new MemoryStream();
        var dataByte = new byte[1024];
        int i = 0;
        do
        {
            i = await netStream.ReadAsync(dataByte, 0, 1024);
            if (i > 0)
            {
                await dataStream.WriteAsync(dataByte, 0, i);
            }
        }
        while (i > 0);

        dataStream.Seek(0, SeekOrigin.Begin);

        var bmpImage = new BitmapImage();
        bmpImage.BeginInit();
        bmpImage.StreamSource = dataStream;
        bmpImage.EndInit();

        var img = new Image
        {
            Stretch = Stretch.Uniform,
            Source = bmpImage
        };
        StackPanell.Children.Add(img);
    }
    localClient.Close();
    netStream.Close();
}
}

```

Con l'oggetto `IPEndPoint` specifichiamo gli indirizzi e le porte degli host su cui l'oggetto `TcpListener` rimane in ascolto in attesa di possibili connessioni, mentre con `IPAddress.Any` ci apriamo a tutti i potenziali client.

All'interno di un ciclo, con il metodo `Pending` verifichiamo la presenza di una connessione in ingresso: in caso positivo, attiviamo la comunicazione con `AcceptTcpClientAsync`, che restituisce l'oggetto `TcpClient`, mediante il quale possiamo inviare e ricevere i dati. Nell'esempio precedente recuperiamo lo stream di comunicazione, di tipo `NetworkStream`, attraverso il metodo `GetStream`. Trattandosi della parte server del nostro esempio, quella in ricezione, procediamo alla lettura dei dati, salvandoli in un `MemoryStream`, al fine di poterli utilizzare come sorgente dell'oggetto `BitmapImage`, e di mostrarli a video con un controllo di tipo `Image`.

Per realizzare l'esempio, andiamo a creare un'applicazione nella quale daremo la possibilità di scegliere un'immagine con l'oggetto `OpenFileDialog`, e ne invieremo i dati in modo del tutto simile a quanto abbiamo fatto

nell'applicazione server. Possiamo vederne un'implementazione nell'[esempio 19.17](#).

Esempio 19.17

```
private async Task Button1_Click(object sender, RoutedEventArgs e)
{
    using (Stream fileStream = File.OpenRead(this.TextBlock1.Tag.ToString()))
    {
        var client = new TcpClient();
        await client.ConnectAsync("localhost", 1234);
        var netStream = client.GetStream();
        var sendBuffer = new byte[1024];
        int bytesRead = 0;
        do
        {
            bytesRead = await fileStream.ReadAsync(sendBuffer, 0, 1024);
            if (bytesRead > 0)
            {
                netStream.Write(sendBuffer, 0, bytesRead);
            }
        }
        while (bytesRead > 0);
        netStream.Close();
    }
}
```

Dopo aver aperto lo stream dell'immagine presente sul file system, inizializziamo l'oggetto `TcpClient` e apriamo una connessione all'host locale, sulla porta 1234, con il metodo `ConnectAsync`. Proprio per la natura del protocollo, è importante che, al momento della chiamata del metodo `connect`, il server sia già attivo e in ascolto.

Con `GetStream` apriamo il `NetworkStream`, con il quale inviamo i dati al server, e andiamo a scrivere i byte dell'immagine selezionata.

Nella [Figura 19.11](#) possiamo notare come più client possano inviare al server i byte corrispondenti alle immagini selezionate, senza la necessità di trasferire veri e propri file: è un'operazione per la quale abbiamo a disposizione specifici oggetti che analizzeremo in seguito.



Figura 19.11 – Esempio di scambio di dati tra applicazioni con protocollo TCP.

Con queste classi abbiamo visto come sia semplice gestire le comunicazioni a livello basso. Proseguiamo ora con l'analisi degli altri oggetti del .NET Framework.

[Il namespace System.NET](#)

Nella sezione precedente abbiamo descritto le classi del namespace `System.NET.Sockets`, che sono classi di livello basso per la comunicazione di dati. Oltre a queste, nel namespace `System.NET` possiamo trovare una vasta serie di strumenti specializzati nella comunicazione con i **protocolli applicativi**, come HTTP o FTP, che ci sollevano dall'onere della gestione diretta dei byte.

I principali oggetti d'uso più frequente sono rappresentati nella [Figura 19.12](#).

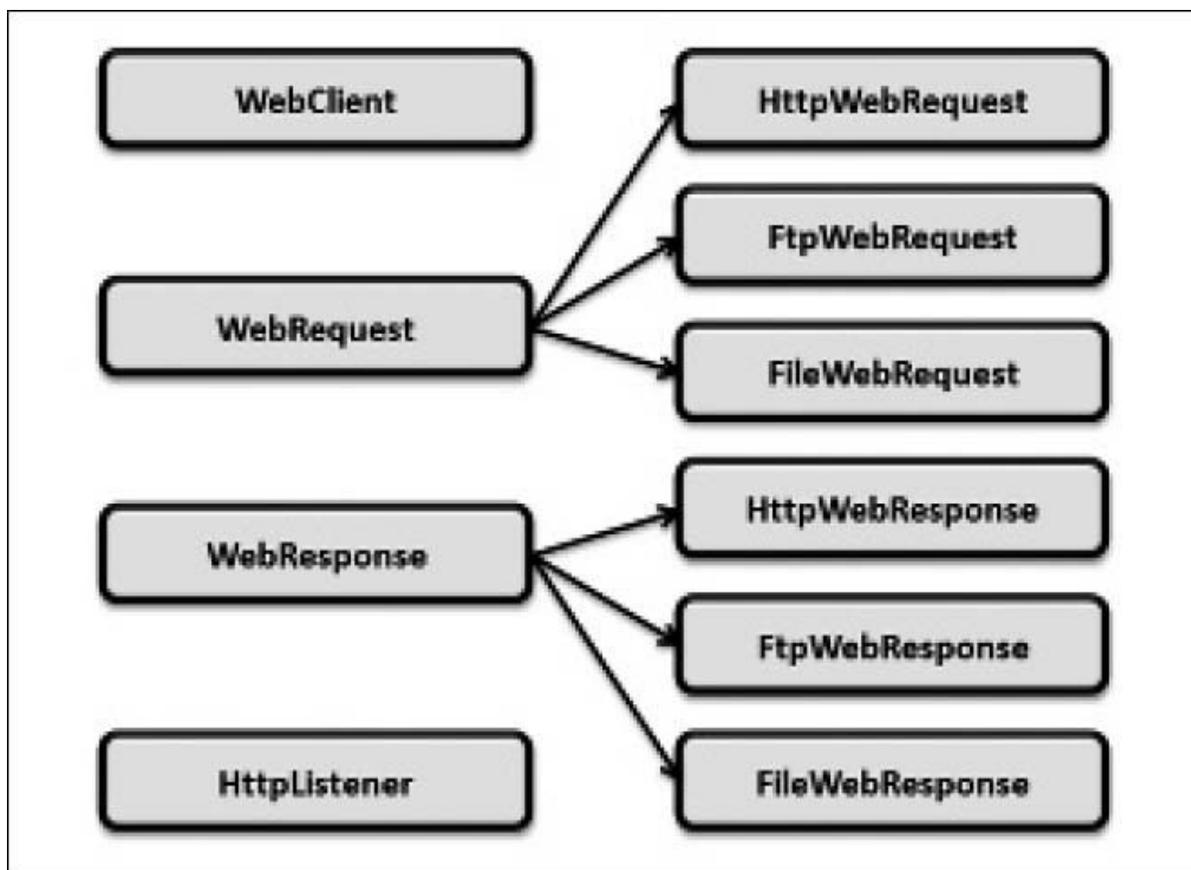


Figura 19.12 – Gerarchia degli oggetti namespace System.NET.

Più avanti analizzeremo le caratteristiche e i modi d'uso di queste classi, così da comprenderne al meglio l'utilità specifica.

[HttpClient: un'evoluta interfaccia HTTP per applicazioni moderne](#)

Oltre alle classi che abbiamo appena visto, nell'assembly `System.NET.Http.dll` abbiamo a disposizione `HttpClient`, una classe che incapsula le funzionalità per consumare servizi REST-ful con grande produttività e soprattutto in modo portabile tra applicazioni eterogenee.

`HttpClient` espone una serie di metodi per le comuni operazioni fatte con i *verb* GET, POST, PUT, DELETE:

- `GetAsync;`

- GetByteArrayAsync;**
- GetStreamAsync;**
- GetStringAsync;**
- PostAsync;**
- PutAsync;**
- SendAsync;**
- DeleteAsync.**

Ciascun metodo dispone di diversi overload e, come possiamo vedere, tali metodi sono tutti asincroni e possono essere invocati con `async/await`. Nell'[esempio 19.22](#) vediamo recuperare dati di un servizio in semplice formato testo utilizzando il metodo `GetStringAsync`.

Esempio 19.22

```
var client = new HttpClient();
try
{
    client.BaseAddress = new Uri("http://localhost:18974/");
    client.MaxResponseContentBufferSize = 1024;
    string data = await client.GetStringAsync("api/products/1");
}
catch (Exception ex)
{
}
finally
{
}
```

Nel codice dell'esempio, possiamo vedere la proprietà `BaseAddress` con cui impostiamo l'`Uri` di base del servizio invocato; con `MaxResponseContentBufferSize`, invece, possiamo definire la grandezza del buffer in lettura, il cui valore predefinito è di due gibabyte.

Solitamente i web service REST si scambiano oggetti nei formati JSON e XML e quindi, in questo caso, potremmo procedere manualmente alla deserializzazione della stringa nell'oggetto noto.

Quando abbiamo bisogno di entrare nel dettaglio della risposta del servizio, possiamo invocare il metodo `GetAsync`, come nell'[esempio 19.23](#).

Anche se la classe `HttpClient` implementa l'interfaccia `IDisposable`, se ne sconsiglia l'uso in tale contesto. Nel caso sia necessario fare chiamate HTTP in modo frequente e ripetuto, è consigliato l'uso della classe `HttpClientFactory`, così da evitare problemi i che si possono verificare, come la `sockets exhaustion`. Maggiori informazioni su questo aspetto sono disponibili su <https://aspit.co/bvw>.

Esempio 19.23

```
var client = new HttpClient();

try
{
    client.BaseAddress = new Uri("http://localhost:18974/");

    using (HttpResponseMessage response =
        await client.GetAsync("api/products/1", cts.Token);)
    {
        log.AppendLine(response.RequestMessage.ToString());
        log.AppendLine(response.Version.ToString());
        log.AppendLine(response.StatusCode.ToString());

        if (response.IsSuccessStatusCode)
        {
            var content = response.Content;
            foreach (var item in content.Headers)
                log.AppendLine(item.Key + " " +
                    item.Value.FirstOrDefault());

            string data = await content.ReadAsStringAsync();
            log.AppendLine(data);
        }
        else
        {
            log.AppendLine(response.ReasonPhrase);
            response.EnsureSuccessStatusCode();
        }
    }
}
catch (TaskCanceledException)
{
}
catch (Exception ex)
{
}
finally
{
    log.AppendLine("-----");
    TextBox2.Text = log.ToString();
}
```

Il metodo `GetAsync` ci dà accesso all'oggetto `HttpResponseMessage`, che espone il contenuto della response con la proprietà `Content`, i suoi `Header` e altre informazioni.

`Content`, di tipo, `HttpContent`, può essere letto con il metodo `ReadAsStringAsync`, che ci restituisce il contenuto del messaggio in formato stringa.

L'oggetto `HttpResponseMessage`, recuperato con questa modalità, permette maggiore controllo sulla response e ci permette di sapere se il servizio ha risposto con uno `statusCode` consono a un'elaborazione andata a buon fine (`IsSuccessStatusCode`) oppure anche di sollevare un'eccezione nel caso contrario, con il metodo `EnsureSuccessStatusCode`.

Nel codice dell'[esempio 19.24](#), vediamo l'uso del metodo `PostAsync` per inviare dei dati in POST al web service.

Esempio 19.24

```
var client = new HttpClient();

client.BaseAddress = new Uri("http://localhost:18974/");
string newData = @"{'Id':22,
                    'Name':'Product22',
                    'Price':22.0,
                    'Category':'Category22'}";
HttpContent contentPost = new StringContent(newData,
                                             Encoding.UTF8, "application/json");
using (HttpResponseMessage response =
        await client.PostAsync("api/products", contentPost);)
{
    if (response.IsSuccessStatusCode)
    {
        Product product = await response.Content.ReadAsStringAsync<Product>();
        log.AppendLine(product.ToString());
    }
}
```

Il metodo `PostAsync` accetta come parametro l'Uri del servizio e, soprattutto, un oggetto che estende la classe astratta `HttpContent`, come oggetto da inviare. Nell'esempio è stato usato `StringContent`, con il quale un oggetto `Product` è stato rappresentato in stringa con formato JSON.

L'utilizzo degli altri metodi non differisce molto da quanto abbiamo visto; a semplificare ulteriormente il codice viene in aiuto una serie di extension method presenti nella libreria `Microsoft.AspNet.WebApi.Client`, installabile attraverso NuGet.

In particolar modo, nell'assembly `System.NET.Http.Formatting.dll`, possiamo trovare `ReadAsStringAsync`, `PostAsJsonAsync`, `PutAsJsonAsync` e tutti i loro overload che riducono gli oneri di serializzazione e deserializzazione dei dati in transito. L'[esempio 19.25](#) è eloquente e mostra la riduzione del codice grazie a due dei suddetti metodi.

Esempio 19.25

```
Product product = await response.Content.ReadAsStringAsync();  
//  
await client.PostAsJsonAsync<Product>("api/products", product, cts.Token);
```

Quasi tutti i metodi di `HttpClient` prevedono un parametro di tipo `CancellationToken` con cui possiamo controllare i task e annullare l'esecuzione del codice successivo. Nel codice allegato sono presenti alcuni esempi.

Scambiare file con il protocollo FTP

Quanto abbiamo detto nel caso di HTTP per `HttpRequest` trova una controparte nelle classi `FtpWebRequest` e `FtpWebResponse`, per gestire le risorse esposte da un server FTP.

Anche `FtpWebRequest` eredita da `WebRequest` e, per questo motivo, troviamo i corrispondenti metodi, relativi al protocollo FTP, che abbiamo visto nel caso dell'accesso e dell'aggiornamento dei dati. Una delle differenze più importanti è la funzionalità che assume la proprietà `Method` nel caso venga utilizzata con il protocollo FTP. Grazie a questa proprietà, infatti, andiamo a specificare le diverse operazioni che possiamo compiere, sia con una singola risorsa remota sia nell'intero contesto di una directory. Possiamo descrivere queste operazioni con i seguenti membri della classe `WebRequestMethods.Ftp`:

- `AppendFile`.
- `DeleteFile`.
- `DownloadFile`.
- `GetDateTimestamp`.

- GetFileSize.
- ListDirectory.
- ListDirectoryDetails.
- MakeDirectory.
- PrintWorkingDirectory.
- RemoveDirectory.
- Rename.
- UploadFile.
- UploadFileWithUniqueName.

I nomi sono auto esplicativi: nell'[esempio 19.26](#) possiamo vedere come utilizzare l'oggetto per eseguire l'upload di un file.

Esempio 19.26

```
using (Stream fileStream = File.OpenRead(this.TextBlock1.Tag.ToString()))
{
    var request = (FtpWebRequest)WebRequest.Create(string.Format(
        "ftp://xxx.xxx.xxx.xxx/{0}", this.TextBlock1.Text));
    request.Method = WebRequestMethods.Ftp.UploadFile;
    request.Credentials = new NetworkCredential("username", "password");
    Stream requestStream = await request.GetRequestStreamAsync();
    byte[] sendBuffer = new byte[1024];
    int bytesRead = 0;
    do
    {
        {
            bytesRead = await fileStream.ReadAsync(sendBuffer, 0, 1024);
            if (bytesRead > 0)
                await requestStream.WriteAsync(sendBuffer, 0, bytesRead);
        }
    } while (bytesRead > 0);
    requestStream.Close();

    this.TextBlock1.Text = (await (FtpWebResponse)request.GetResponseAsync()).
        StatusDescription;
}
```

L'implementazione non differisce molto dalle modalità che abbiamo analizzato per gli altri protocolli. Una particolarità d'uso molto frequente con il protocollo FTP è la possibilità di specificare le credenziali d'accesso. Nell'esempio, usando la proprietà `Credentials` di tipo `ICredential`, abbiamo specificato username e password grazie all'oggetto `NetworkCredential`. In virtù di queste credenziali, il server FTP autorizzerà o non autorizzerà le operazioni che andremo a eseguire.

Conclusioni

In questo capitolo abbiamo inizialmente visto come svolgere le operazioni più comuni su file e directory, quali spostamento, copia o eliminazione. Successivamente abbiamo mostrato come eseguire ricerche e come possiamo utilizzare la classe `FileStream` per interagire con il contenuto di un file.

In seguito abbiamo introdotto il concetto di `IsolatedStorage`, ossia un'alternativa valida e sicura per memorizzare informazioni in una porzione di disco già predisposta allo scopo dal sistema operativo. Abbiamo quindi illustrato i principi di comunicazione di rete e le caratteristiche dei protocolli che si sono evoluti nel tempo, in parallelo allo sviluppo di Internet. Poi abbiamo analizzato i principali strumenti di medio livello, mediante i quali possiamo gestire le comunicazioni socket, lavorando con gli oggetti specifici per i protocolli TCP e UDP.

Successivamente abbiamo affrontato gli esempi più comuni di gestione delle risorse remote, come HTTP e FTP, analizzando le classi principali disponibili nel namespace `System.NET` del .NET Framework.

L'argomento è molto vasto e complesso, come dimostra il numero di classi disponibili nella class library di .NET. Il loro studio richiede un certo tempo e, soprattutto, la comprensione propedeutica dei principi delle interazioni a livello di protocollo. Ne sono stati introdotti i concetti principali e sono stati analizzati, in particolar modo, gli scenari più comuni, che possono rispondere alle esigenze quotidiane delle nostre applicazioni. Attraverso l'uso di questi strumenti diventa possibile implementare il supporto a qualsiasi tipo di protocollo, anche a uno totalmente custom.

In questo capitolo abbiamo visto come il networking ci permetta di comunicare tra applicazioni e/o risorse remote, mentre nel prossimo capitolo, quello finale, vedremo come eseguire il deploy di queste applicazioni,

analizzando le differenze che ci sono tra le varie modalità di distribuzione oggi disponibili.

Configurare e pubblicare un'applicazione .NET

Nel corso del libro abbiamo affrontato tutti quelli che sono i temi principali relativi allo sviluppo di un'applicazione, partendo dal linguaggio e arrivando a creare delle varianti per console, web e desktop con XAML. Tuttavia, nonostante tutti questi concetti siano risultati utili alla costruzione di una vera e propria applicazione, non possono essere messi in pratica e testati effettivamente fino a quando non si arriva alla fase del rilascio, in modo che chiunque possa vedere il prodotto realizzato.

Tipicamente, in passato, distribuire un'applicazione web .NET significava preparare un'ambiente con macchine Windows Server e Internet Information Server (ovvero IIS). Inoltre, era quasi dato per scontato che la stessa versione del .NET Framework utilizzata dal team di sviluppo fosse la stessa disponibile nell'ambiente di produzione, soprattutto considerati i lunghi periodi tra il rilascio di una versione e la successiva. A oggi, però, sono cambiati tanti aspetti:

- ❑ **Tempistiche di rilascio:** i tempi che intercorrono tra la fase di sviluppo e il rilascio in produzione sono notevolmente ridotti per via di pratiche come continuous integration e continuous deployment.
- ❑ **Ambienti multipli:** è fondamentale rilasciare un prodotto che sia verificato e funzionante, possibilmente testato da più team (sviluppo, QA di diversi livelli) ma, soprattutto per via dei tempi ristretti, è bene

testare le applicazioni in ambienti che non siano la produzione (per evitare potenziali problematiche) ma che siano piuttosto simili per effettuare simulazioni di upgrade e downgrade.

- ❑ **L'hardware e il software:** .NET Core, al contrario di .NET Framework, è in grado di eseguire le applicazioni anche in ambienti Linux e macOS, quindi non è più possibile dare per scontata la presenza di Windows Server o di IIS e, inoltre, il framework potrebbe non essere installato fisicamente sulla macchina ma distribuito con l'applicazione stessa.
- ❑ **Tipologia di lavoro:** il team di lavoro moderno e ideale si sta sempre più spostando verso il mondo agile e DevOps, quindi diventa fondamentale avere conoscenze sia di sviluppo software sia di gestione e manutenzione (operations) dell'infrastruttura sulla quale viene eseguito il software; per questo sono emersi temi fondamentali come i container e il cloud.

All'interno di questo capitolo affronteremo proprio il concetto relativo al deployment e al delivery delle applicazioni (prima web e poi desktop) e parleremo nel dettaglio di come le novità introdotte da .NET aiutino a risolvere le problematiche sollevate dagli aspetti evidenziati. Prima di procedere al rilascio di un'applicazione web, però, è lecito domandarsi in quale ambiente l'applicazione stessa verrà distribuita: di fatto, è possibile prevedere un cambio del comportamento sulla specifica dell'ambiente.

Modificare il comportamento secondo l'ambiente di destinazione

Ci possono essere casi in cui il comportamento dell'applicazione può essere diverso a seconda dell'ambiente in cui viene eseguito: l'ambiente di sviluppo (o Development) potrebbe prevedere un logging molto avanzato e continuo su ogni singola chiamata, mentre l'ambiente di produzione potrebbe avere solo un logging sommario per non rallentare l'esecuzione, così come nell'ambiente di controllo qualità si potrebbe aver bisogno di un sistema misto, in cui ha più senso verificare il flusso delle operazioni

effettuate dagli utenti rispetto al sapere il dettaglio di risposta di ogni singola riga di codice. In produzione si vorrà avere la minification o, ancora, ci saranno endpoint differenti in base all'ambiente per richiamare una WebAPI oppure un background service. ASP.NET Core mette a disposizione gli **environment** che permettono proprio di distinguere, a livello di codice, l'ambiente di esecuzione a seconda di quanto viene definito dagli sviluppatori oppure dal team di operations direttamente sulla macchina fisica.

Gli environment di default delle applicazioni ASP.NET sono `Development`, `Staging` e `Production`, che vengono esposti direttamente tramite l'interfaccia `IHostingEnvironment` e, in particolare, tramite la proprietà `EnvironmentName` ma, poiché ci sono scenari che richiedono configurazioni e ambienti molto più complessi, è anche possibile andare a creare ambienti personalizzati.

Esempio 20.1

```
public class MyClass
{
    public MyClass(IHostingEnvironment en)
    {
        // controlliamo l'ambiente
        if (en.IsEnvironment("MyCustomEnvironment"))
        {
            // l'ambiente è identificato, eseguiamo operazioni specifiche...
        }
    }
}
```

Nell'[esempio 20.1](#) si può notare come sia possibile identificare un ambiente personalizzato per poter eseguire le operazioni specifiche relative a quell'environment. Non tutta la configurazione però è strettamente legata ai servizi che devono essere erogati (come, per esempio, il cambio di URL tra ambiente di sviluppo e di produzione), ma può anche essere relativa all'infrastruttura su cui viene eseguita, come per esempio l'uso di Kestrel (un server web cross-platform direttamente integrato in .NET Core) piuttosto che IIS per il web server, l'entrypoint dell'applicazione stessa o, ancora, l'uso di variabili d'ambiente personalizzate. Tutte queste opzioni possono essere specificate all'interno del file di configurazione

launchSettings.json nella cartella Properties del progetto, come mostrato nell'[esempio 20.2](#).

Esempio 20.2

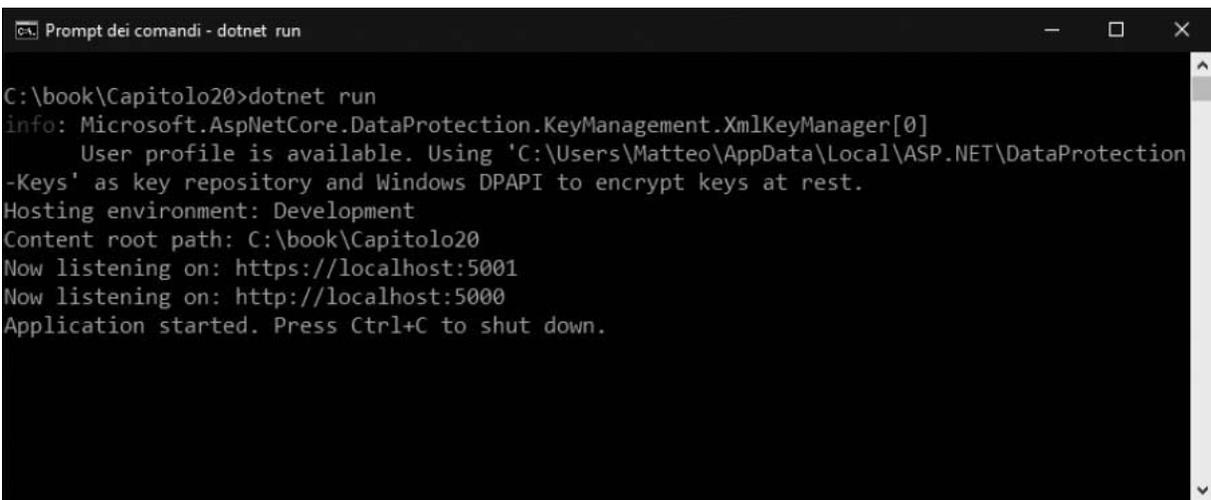
```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:43421",
      "sslPort": 44300
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "Capitolo20": {
      "commandName": "Project",
      "launchBrowser": true,
      "applicationUrl": "https://localhost:5001;http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development",
        "ASPNETCORE_CUSTOM_VALUE": "Custom",
      }
    }
  }
}
```

Tra le variabili d'ambiente esposte dall'[esempio 20.2](#) si può notare la sola ASPNETCORE_ENVIRONMENT che rappresenta proprio l'EnvironmentName di IHostingEnvironment per il profilo "IIS Express", mentre per il profilo "Capitolo20" è stata aggiunta anche una variabile custom.

Durante l'avvio di un'applicazione ASP.NET Core, vengono eseguite una serie di operazioni in sequenza che sono mostrate nella [Figura 20.1](#): in prima istanza viene recuperato il file launchSettings.json per capire quale profilo avviare, quindi il framework legge i valori delle variabili d'ambiente e li sostituisce a quelli eventualmente definiti tramite variabili d'ambiente a livello di sistema; poi viene recuperato il valore dell'environment ed eseguito il Main della classe Program, che avvia il web

server e apre le porte necessarie e, infine, viene caricata la configurazione opportuna.

La variabile d'ambiente `ASPNETCORE_ENVIRONMENT` può essere impostata a diversi livelli, in modo dipendente dal sistema operativo e dalla durata prevista. Per avere un'impostazione temporanea a livello di finestra/esecuzione corrente, su Windows, per esempio, può essere impostata tramite i comandi `setx` e `$Env` rispettivamente per Command Line e PowerShell, mentre per macOS e distribuzioni Linux può essere impostata tramite il comando `export`. Purtroppo, però, l'esecuzione di questi comandi è limitata, quindi in ambienti di produzione o in scenari in cui l'applicazione può andare in crash non è l'ideale, dato che riavviandosi perderebbe l'ambiente precedentemente impostato; per questo è necessario impostare la variabile direttamente a livello di sistema operativo, tramite il `bash_profile` di Linux e macOS, oppure tramite le variabili d'ambiente di Windows, come è bene illustrato nella [Figura 20.2](#).



```
C:\book\Capitolo20>dotnet run
info: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[0]
      User profile is available. Using 'C:\Users\Matteo\AppData\Local\ASP.NET\DataProtection-Keys' as key repository and Windows DPAPI to encrypt keys at rest.
Hosting environment: Development
Content root path: C:\book\Capitolo20
Now listening on: https://localhost:5001
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

Figura 20.1 - È possibile avviare un'applicazione .NET Core anche da console tramite l'esecuzione del comando `dotnet run`. In questo modo è possibile vedere l'esecuzione di tutte le operazioni di startup.

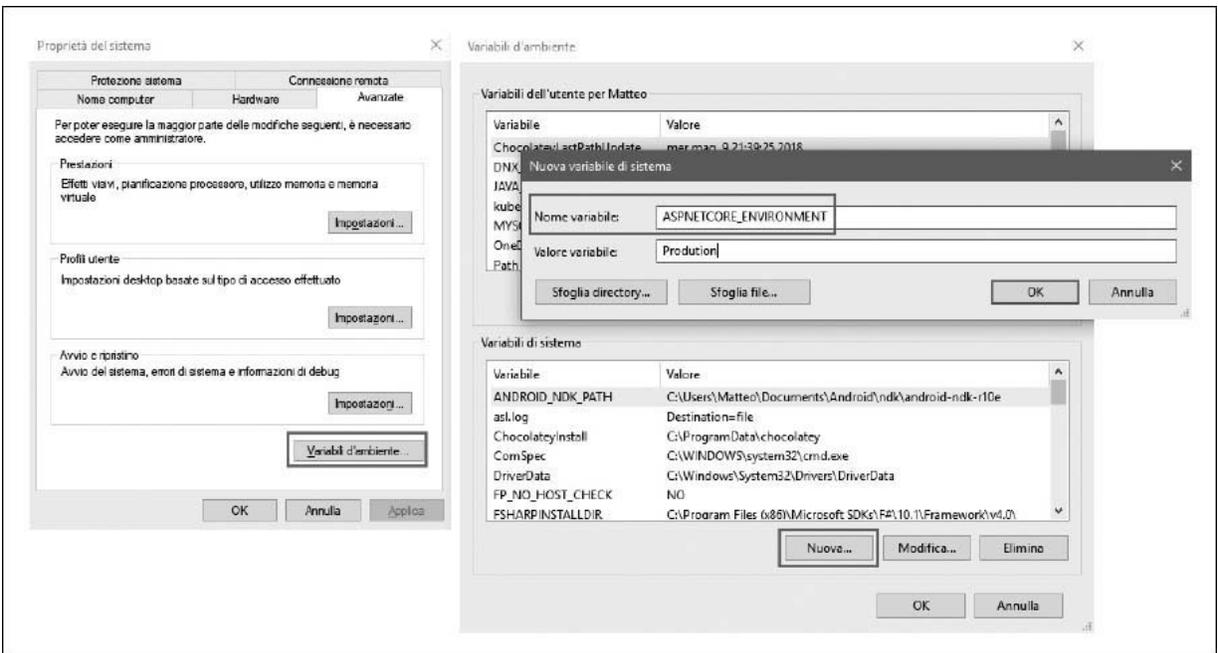


Figura 20.2 - L'impostazione di una variabile di sistema sotto Windows avviene tramite il menu Proprietà di sistema → Variabili d'ambiente → Nuova variabile di sistema.

Qualora l'applicazione venga eseguita in ambiente Windows con IIS, allora è anche possibile specificare l'environment e le altre variabili d'ambiente tramite il `web.config`, come dimostra l'[esempio 20.3](#).

Esempio 20.3

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule"
        resourceType="Unspecified" />
    </handlers>
    <aspNetCore processPath="dotnet"
      arguments=".\Capitolo20.dll"
      stdoutLogEnabled="false"
      stdoutLogFile=".\logs\stdout">
      <environmentVariables>
        <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development"
          />
        <environmentVariable name="ASPNETCORE_CUSTOM_VALUE" value="Custom" />
      </environmentVariables>
    </aspNetCore>
  </system.webServer>
</configuration>
```

```
</configuration>
```

All'interno di un progetto vuoto ASP.NET Core, la prima volta in cui si incontra l'uso della variabile `ASPNETCORE_ENVIRONMENT` è nel metodo `Configure` della classe `Startup`, in cui l'uso della pagina di errore dettagliato e di HSTS vengono aggiunti, rispettivamente, solo per gli ambienti di sviluppo e produzione, come viene illustrato nell'[esempio 20.4](#). Facendo uso della dependency injection sulla variabile di tipo `IHostingEnvironment` è possibile quindi escludere porzioni di codice e cambiare il comportamento dell'applicazione a seconda dell'ambiente che viene eseguito.

Esempio 20.4

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/Error");
        app.UseHsts();
    }

    app.UseMvc();
}
```

Anche la configurazione dell'applicazione stessa può essere specifica per ogni ambiente, viene prelevata dai file `appsettings.{environment-name}.json` e può essere letta all'interno dell'applicazione sia tramite `IConfiguration` sia in maniera fortemente tipizzata. Questo, però, non garantisce alcun tipo di protezione dei dati: infatti tutte le chiavi, come per esempio `connection string` per il database ed eventuali password, sono esposte in chiaro e sono leggibili da chiunque. Pertanto è necessario un sistema basato a secret, come Azure Key Vault, che assicuri un minimo di sicurezza.

Una volta definiti gli ambienti di esecuzione, parametrizzata l'applicazione per poter lavorare con environment diversi e aggiunto il supporto alle secret e ad Azure Key Vault per poter lavorare in sicurezza

con dati che non devono essere esposti all'esterno o vulnerabili tramite la macchina stessa, è il momento di pensare al deployment, ovvero al rilascio in un ambiente dedicato dell'applicazione web creata.

Publicazione e hosting

Durante l'introduzione di questo capitolo abbiamo illustrato come e quanto siano cambiate le pratiche per il rilascio delle applicazioni web per via di diversi fattori quali il tempo, le competenze tecniche e la tipologia di framework. Nonostante tutti questi cambiamenti, però, il flusso in linea generale non cambia e si mantiene su tre aspetti:

- ❑ **Publicazione:** il codice sorgente deve essere compilato e distribuito in una cartella del server.
- ❑ **Scelta del process manager:** la scelta è sempre stata implicita su Windows con IIS per ASP.NET, ma con ASP.NET Core è diventato necessario un process manager anche per gli ambienti macOS e Linux, che sia in grado di riavviare l'applicazione in caso di malfunzionamenti e che riesca a gestire le richieste in ingresso, rimandandole all'applicazione stessa.
- ❑ **Configurazione del reverse proxy:** è una scelta opzionale, da fare in quei casi in cui non si vuole esporre il servizio in modo diretto; così facendo, il reverse proxy prenderà le richieste e le rimanderà a Kestrel in un secondo momento.

La pubblicazione è un processo molto semplice e, di fatto, consiste solamente nel cliccare il tasto destro del mouse sul progetto web e selezionare *Publish*. Allo stesso modo, tramite il comando `dotnet publish` della CLI, per le applicazioni .NET Core, vengono generate DLL, file e dipendenze in una cartella specifica per la pubblicazione (specificata all'interno delle proprietà del progetto), in maniera tale che non venga copiato in alcun modo il codice sorgente originale.

La scelta del web server invece è importante, non solo perché non né si può fare a meno, dato che il sistema in-process serve a rimandare le richieste HTTP verso `HttpContext` all'interno dell'applicazione, ma anche

perché prima di decidere quale web server utilizzare, è necessario capire il sistema operativo di destinazione, ed è una scelta strategica. Poiché ASP.NET Core, al contrario di ASP.NET, è in grado di funzionare cross-platform, Microsoft ha deciso di rilasciarne due implementazioni:

- ❑ **Kestrel**: di default, funziona anche cross-platform.
- ❑ **HTTP.sys**: chiamato in passato **WebListener**, è l'implementazione esclusiva per Windows basata sul kernel driver HTTP.sys e HTTP server API.

Ci sono diversi vantaggi nell'uso di uno, dell'altro, o di un server personalizzato costruito sulla base dell'interfaccia `IServer` ma, in generale, poiché Kestrel è di default e il funzionamento è cross-platform, sarà quello che utilizzeremo in futuro.

L'uso di un reverse proxy, inoltre, non è strettamente necessario, poiché Kestrel è considerato, dalla versione 2 di ASP.NET Core, production-ready ma ci sono comunque decine di casi in cui ha senso utilizzarlo: limitare l'esposizione dell'host, aggiungere un nuovo strato di sicurezza, limitare il numero di chiamate e il traffico generato e semplificare il load balancing sono solo alcuni esempi. Proprio per queste motivazioni, vedremo ora come configurare NGINX come reverse proxy per ambienti Linux e macOS.

Configurazione di NGINX

NGINX è un web server molto leggero, che garantisce ottime prestazioni per via del suo approccio asincrono basato a eventi generati dalle richieste HTTP che arrivano in ingresso ed è ottimo per funzionare come reverse proxy con Kestrel. Inoltre, funziona in ambienti Unix/Linux, BSD (macOS) e Windows, ed è perciò in grado di fornire supporto cross-platform ai nostri applicativi .NET. Per vedere un aspetto diverso e per mantenere una certa semplicità, le demo successive saranno costruite secondo un ambiente macOS con una sola istanza di NGINX non replicata, ma tutto il codice e le configurazioni che vedremo saranno portabili con pochi cambiamenti all'interno degli altri sistemi operativi e delle eventuali repliche.

Considerando che, una volta configurato il reverse proxy, le richieste passeranno prima da NGINX e poi verranno rimandate all'applicazione ASP.NET, è necessario configurare opportunamente gli header di forward, altrimenti, qualora nelle applicazioni web ci siano middleware o logiche che leggono questi valori, come per esempio sistemi di tracing delle request, questi potrebbero non funzionare più correttamente. Questo è particolarmente utile anche in scenari in cui si vuole avere un trace completo delle richieste, per capire potenzialmente che cosa non stia funzionando a livello di routing.

Esempio 20.5

```
public void Configure (IApplicationBuilder app)
{
    app.UseForwardedHeaders(new ForwardedHeadersOptions
    {
        ForwardedHeaders = ForwardedHeaders.XForwardedFor |
                           ForwardedHeaders.XForwardedProto;
    });
    app.UseMvc();
}
```

L'[esempio 20.5](#) mette in evidenza come tramite il metodo `Configure` si possa agire sulla configurazione di `UseForwardedHeaders` per specificare di rimandare gli header `X-Forwarded-For` (che conterrà l'indirizzo IP di origine della richiesta) e `X-Forwarded-Proto` (che conterrà invece il protocollo HTTP o HTTPS).

Una volta preparata l'applicazione a ricevere le richieste, arriva il momento di installare e configurare il web server: per l'installazione si può fare uso del package manager HomeBrew su macOS con il comando `brew install nginx`, piuttosto che di APT per Linux con il comando `apt-get install nginx`. Per testare il funzionamento, è necessario avviare il web server tramite il comando `sudo nginx start` e, se questo è stato installato correttamente, partirà con la sua configurazione di default mostrando una pagina HTML statica al raggiungimento dell'indirizzo <http://localhost:8080>. Una volta verificato il corretto funzionamento del servizio out-of-the-box, è necessario istruirlo per lavorare come proxy e quindi rimandare le chiamate. Per farlo, sarà sufficiente modificare la

configurazione di default creata da NGINX nel file `/etc/nginx/nginx.conf`, come mostrato nell'[esempio 20.6](#).

Esempio 20.6

```
worker_processes 1;

events {
    worker_connections 1024;
}

http {
    include mime.types;
    default_type application/octet-stream;
    keepalive_timeout 65;

    server {
        listen 80;
        server_name localhost;

        location / {
            proxy_pass http://localhost:5000;
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection keep-alive;
            proxy_set_header Host $host;
            proxy_cache_bypass $http_upgrade;
        }
    }
}
```

La maggior parte della configurazione esposta dall'[esempio 20.6](#) non è cambiata rispetto a quanto è previsto di default dal server. I cambiamenti si trovano solamente nella configurazione della porta 80 anziché 8080 e nella configurazione di tutti gli attributi proxy necessari a rimandare la chiamata all'applicazione. In particolare, poiché il server coincide con la macchina stessa, è stato aggiunto `localhost`, ma nulla vieta, una volta configurato SSL, di aggiungere il proprio dominio "[mydomain.com](#)". Le richieste verranno poi passate all'applicazione ASP.NET tramite `proxy_pass` che è stato impostato sempre su `localhost` nella porta 5000 (default di ASP.NET Core). Una volta salvata la configurazione appena impostata, è necessario rilanciare il web server con il comando `sudo nginx restart` e avviare l'applicazione web con il comando `dotnet nome.dll`: se il tutto è stato rappresentato correttamente, l'applicazione sarà raggiungibile sia tramite

<http://localhost:5000>, ovvero tramite Kestrel, sia tramite <http://localhost>, ovvero in reverse proxy da NGINX.

Nonostante il sistema sia funzionante, a tutti gli effetti c'è ancora un'operazione che è stata eseguita manualmente, ovvero l'avvio dell'applicazione web tramite CLI, pertanto non si stanno sfruttando i vantaggi di riavvio automatico visti in precedenza per i process manager. Per farlo, bisogna fare uso di *launchd* (o *systemd* su Linux) per creare e registrare servizi. Ogni servizio, in ambiente macOS, è rappresentato da un file con estensione *.plist* in cui sono specificate tutte le proprietà relative a quello che vogliamo venga fatto in completa autonomia, come è bene illustrato nell'[esempio 20.7](#).

Esempio 20.7

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/
DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>KeepAlive</key>
  <true/>
  <key>Label</key>
  <string>Capitolo20</string>
  <key>ProgramArguments</key>
  <array>
    <string>/usr/local/share/dotnet/dotnet</string>
    <string>Capitolo20.dll</string>
  </array>
  <key>RunAtLoad</key>
  <true/>
  <key>StandardErrorPath</key>
  <string>/tmp/dotnet-api-sterr.log</string>
  <key>StandardOutPath</key>
  <string>/tmp/dotnet-api-stdout.log</string>
  <key>UserName</key>
  <string>root</string>
  <key>WorkingDirectory</key>
  <string>/Users/matteotumiati/Desktop/Capitolo20/bin/Release/
netcoreapp2.2/publish</string>
  <key>EnvironmentVariables</key>
  <dict>
    <key>ASPNETCORE_ENVIRONMENT</key>
    <string>Production</string>
  </dict>
</dict>
</plist>
```

Il file mostrato nell'[esempio 20.7](#) viene salvato all'interno del percorso */Library/LaunchDaemons*, cosicché, una volta registrato il servizio, al riavvio della macchina stessa venga avviato in automatico. All'interno di questo file vengono descritti il nome del servizio con l'attributo `Label`, qual è il comando che deve essere avviato con i suoi parametri (ovvero `dotnet Capitolo20.dll`) con l'attributo `ProgramArguments`, i percorsi per i log di `stdout` e `stderr`, la `working directory` (che corrisponde al percorso in cui ci sarà la DLL che deve essere eseguita, le variabili d'ambiente e, infine, lo `username` che può avviare il servizio (`root` è l'unica opzione disponibile in `LaunchDaemons`). Per avviare il servizio, è necessario lanciare in sequenza i comandi mostrati nell'[esempio 20.8](#).

Esempio 20.8

```
cd /Library/LaunchDaemons
sudo launchctl load -w Capitolo20.plist
sudo launchctl start Capitolo20
```

Una volta registrato il servizio, si può controllare che sia effettivamente partito lanciando il comando `sudo launchctl list`, come viene mostrato nella [Figura 20.3](#).

```
LaunchDaemons -- -bash
/Library/LaunchDaemons -- -bash
/usr/local/etc/nginx -- -bash
~
-      0      com.apple.efilogin-helper
100    0      com.apple.loginwindow
-      0      com.apple.cmio.iOSScreenCaptureAssistant
-      0      com.apple.rapportd
110    0      com.apple.notifyd
-      0      com.apple.ReportPanicService
-      0      com.apple.wifiFirmwareLoader
53     0      com.apple.fsevents
-      0      com.apple.Kerberos.digest-service
-      0      com.apple.systempreferences.cacheAssistant
4885   0      com.apple.automountd
-      0      com.vix.cron
240    0      com.apple.thermald
355    0      com.apple.FileCoordination
4954   0      Capitolo20
-      0      com.apple.eapolcfg_auth
259    0      com.apple.audio.AudioComponentRegistrar
4546   0      com.apple.AssetCacheLocatorService
-      0      com.apple.InstallerDiagnostics.installerdiagd
4389   0      com.apple.softwareupdated
-      0      com.apple.systemadministration.writeconfig
-      0      com.apple.MobileAccessoryUpdater
-      0      com.apple.diskmanagementstartup
-      0      com.apple.cisconnect.cisconnect-reporter
```

Figura 20.3 - Elenco dei servizi registrati come LaunchDaemons in macOS.

Qualora il servizio sia stato registrato correttamente, come mostrato nella [Figura 20.3](#), con un PID associato, allora la configurazione di NGINX come reverse proxy sarà completata, totalmente automatizzata e resiliente in caso di errori o crash applicativi. Per testimoniare il corretto funzionamento, è sufficiente navigare all'indirizzo <http://localhost> e dovremmo vedere nuovamente il sito web ASP.NET, come illustrato nella [Figura 20.4](#).

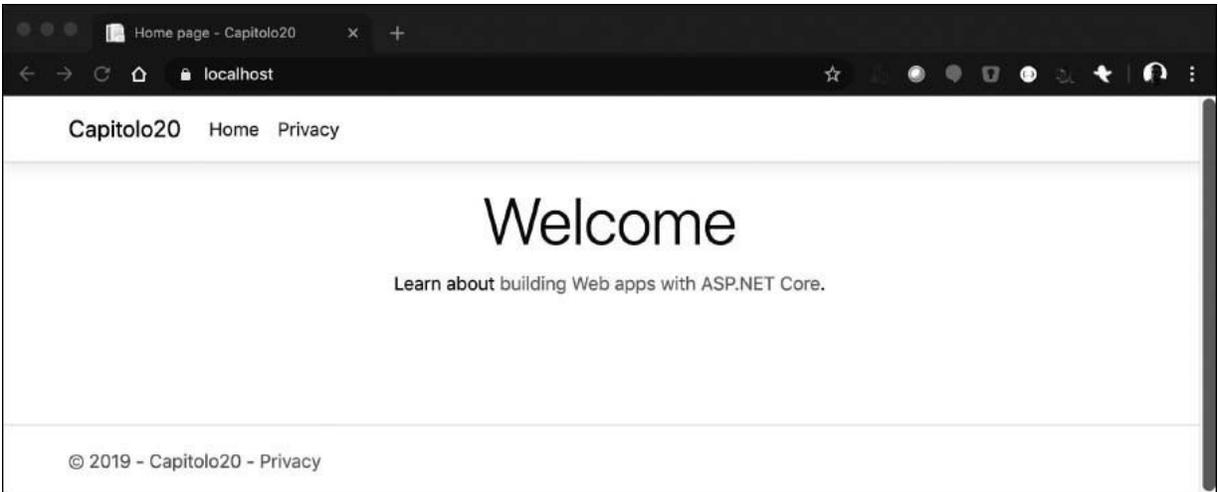


Figura 20.4 - Home page di un'applicazione web ASP.NET esposta in reverse proxy tramite NGINX e il servizio di restart automatico.

Poiché l'applicazione viene avviata in autonomia, non ci sarà a disposizione la console per mostrare i log ma, per via del fatto che sono stati esplicitati nel file del servizio `Capitolo20.plist`, sarà sufficiente lanciare il comando `tail -f /tmp/dotnet-api-stdout.log` per vedere lo stream dello standard output e `tail -f /tmp/dotnet-api-stderr.log` per vedere gli eventuali errori generati.

Nella configurazione di base documentata nell'[esempio 20.6](#), però, manca tutta la parte relativa alla sicurezza stessa del server: pertanto è necessario andare ad aggiungere sul nodo alcuni header fondamentali, come abbiamo illustrato nell'esempio seguente.

Esempio 20.9

```
server {  
    listen *:443 ssl;  
    add_header Strict-Transport-Security "max-age=63072000";  
    add_header X-Frame-Options DENY;  
    add_header X-Content-Type-Options nosniff;  
}
```

Nell'[esempio 20.9](#) si può notare come siano stati aggiunti tre header:

- ❑ **HTTP Strict-Transport-Security:** HSTS è attivo di default in tutti i progetti a partire da ASP.NET Core 2.1 e permette di gestire tutte le richieste inviate dai client su HTTPS; per questo va accompagnato da certificati SSL e limiti di durata tramite `max-age`.
- ❑ **X-Frame-Options:** se impostato con valore `DENY` garantisce protezione dagli attacchi di clickjacking, in quanto non permette l'uso di frame con link a fonti esterne.
- ❑ **X-Content-Type-Options:** se impostato con valore `nosniff` garantisce protezione dagli attacchi di MIME-Type sniffing, in cui il browser potrebbe cambiare l'header Content-Type e interpretare un oggetto di un tipo come di un altro.

La configurazione del server, poiché in questo caso risponderà sulla porta 443 anziché sulla porta 80 come visto in precedenza, può essere affiancata alla configurazione mostrata nell'[esempio 20.10](#) e il sistema risponderà in reverse proxy sempre in modo adeguato, rimandando le richieste su HTTPS. Dal momento che il resto della configurazione del server dipende principalmente dalle proprie esigenze di produzione, non affronteremo temi più avanzati nel corso di questo capitolo, ma rimandiamo alla documentazione ufficiale disponibile su <http://aspit.colbot> e proseguiamo vedendo la configurazione per il web server classico di Windows, ovvero IIS.

Configurazione di IIS

Internet Information Service (IIS) è sicuramente il web server più conosciuto per chi proviene dal mondo Windows e rimane uno dei web server più sicuri al mondo. La sua complessità è notevole, in quanto la sua architettura è formata da diversi moduli componibili fra di loro. Al contrario di quanto abbiamo già visto in precedenza con NGINX, in questa parte del capitolo daremo per scontata l'installazione e la configurazione stessa di IIS poiché dipende dalla versione di Windows (sulla parte server, per esempio, è attiva di default, mentre sulla parte client va attivata tramite le funzionalità aggiuntive) e dai vari ruoli che si vogliono aggiungere, come Windows Authentication o WebSockets. Nonostante l'installazione sia già

data per scontata, questo ancora non consente l'esecuzione delle applicazioni ASP.NET Core out-of-the-box perché .NET Core è un framework relativamente nuovo e IIS non ha le informazioni necessarie per sapere come comportarsi: per questo è necessario procedere all'installazione del **.NET Core Windows Server Hosting bundle**, ovvero di un pacchetto che contiene runtime e librerie relative al framework, in aggiunta al modulo **AspNetCoreModule** di IIS, che è proprio la componente che sa come comunicare con l'applicazione. Il download di questo bundle è possibile tramite l'URL <http://aspit.colboy> e, una volta installato e riavviata la macchina, sarà visibile all'interno dei moduli utilizzabili dal web server, come viene mostrato nella [Figura 20.5](#).

The screenshot shows the 'Moduli' (Modules) configuration page in IIS. It features a title bar with a globe icon and the text 'Moduli'. Below the title is a subtitle: 'Utilizzare questa funzionalità per configurare i moduli di codice nativi e gestiti per l'elaborazione delle richieste effettuate al server Web.' There is a dropdown menu for 'Raggruppa per:' set to 'Nessun raggruppamento'. The main content is a table with four columns: 'Nome', 'Codice', 'Tipo modulo', and 'Tipo voce'. The 'AspNetCoreModule' row is highlighted in grey, indicating it is installed. Other modules listed include AnonymousAuthenticationModule, CustomErrorModule, DefaultDocumentModule, DirectoryListingModule, HttpCacheModule, HttpLoggingModule, ProtocolSupportModule, RequestFilteringModule, StaticCompressionModule, and StaticFileModule.

Nome	Codice	Tipo modulo	Tipo voce
AnonymousAuthenticationModule	%windir%\System32\inetsrv\...	Nativo	Locale
AspNetCoreModule	%SystemRoot%\system32\in...	Nativo	Locale
CustomErrorModule	%windir%\System32\inetsrv\...	Nativo	Locale
DefaultDocumentModule	%windir%\System32\inetsrv\...	Nativo	Locale
DirectoryListingModule	%windir%\System32\inetsrv\...	Nativo	Locale
HttpCacheModule	%windir%\System32\inetsrv\...	Nativo	Locale
HttpLoggingModule	%windir%\System32\inetsrv\l...	Nativo	Locale
ProtocolSupportModule	%windir%\System32\inetsrv\...	Nativo	Locale
RequestFilteringModule	%windir%\System32\inetsrv\...	Nativo	Locale
StaticCompressionModule	%windir%\System32\inetsrv\...	Nativo	Locale
StaticFileModule	%windir%\System32\inetsrv\...	Nativo	Locale

Figura 20.5 - Il modulo `AspNetCoreModule` è visibile solamente quando è stato installato correttamente il .NET Core Windows Server Hosting bundle e il runtime di ASP.NET Core.

La parte importante consiste nel fatto che il modulo `AspNetCoreModule` non solo è in grado di eseguire le applicazioni ASP.NET Core ma permette anche la comunicazione in reverse proxy con Kestrel, senza dover cambiare una riga di codice dell'applicazione poiché il tutto viene già gestito da `IWebHostBuilder` della classe `startup`. Inoltre, dal momento che rimanda tutto il traffico verso Kestrel, questo modulo ci permette anche di gestire

scenari più avanzati relativi a sicurezza, logging e configurazione applicativa e del server stesso.

Una volta completato il setup di base, arriverà il momento di creare la configurazione applicativa: ogni applicazione, per girare, necessita di un suo application pool, ovvero di un sistema che garantisca isolamento tra le applicazioni presenti sullo stesso server, così che se una genera errori oppure va in crash, non è in grado di danneggiare le altre. Si deve quindi procedere e creare un nuovo application pool la cui particolarità, rispetto a quanto ci si è abituati dal passato con le applicazioni ASP.NET, è che ASP.NET Core non ha bisogno del CLR per partire, e pertanto, come dimostra la [Figura 20.6](#), si può impostare sul valore “No managed code”.

Una volta definito l’application pool, è possibile andare ad aggiungere il sito web, cliccando con il tasto destro sul nodo “Sites” e creandone uno nuovo. Alcuni dei parametri sono esposti nella [Figura 20.7](#) mostrata qui di seguito.

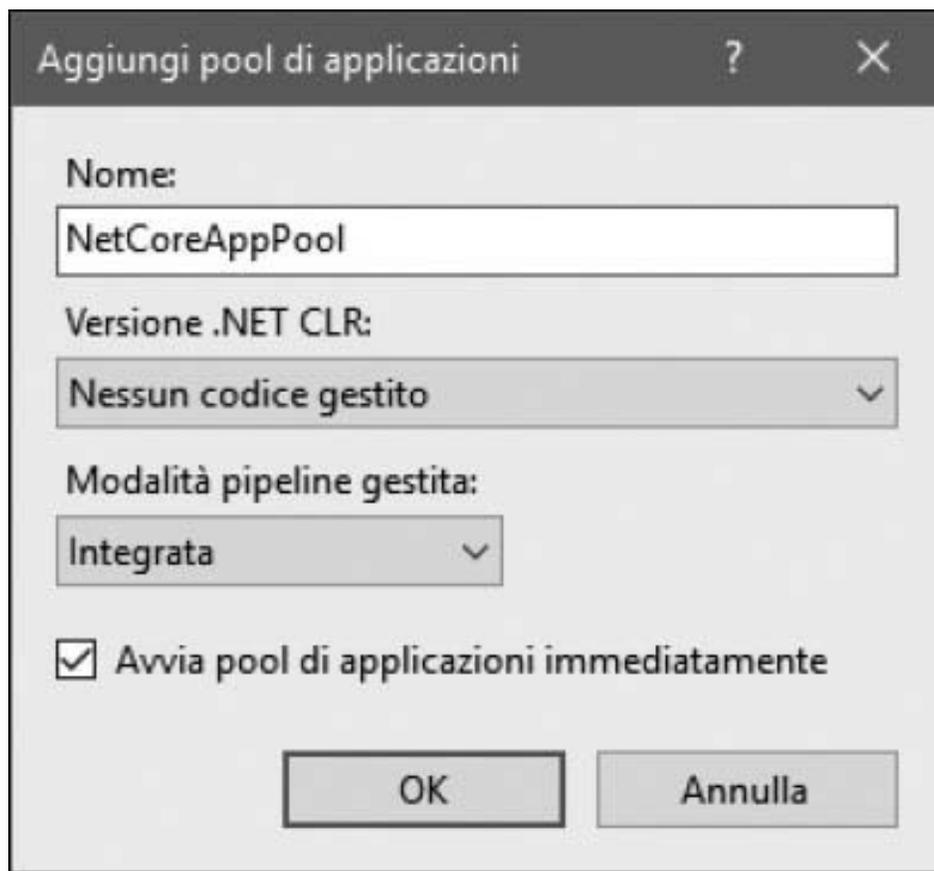


Figura 20.6 - Il nuovo application pool si può configurare con click destro sul nodo degli application pool e va impostato in modalità “No managed code” poiché .NET Core non richiede il Desktop CLR.

Aggiungi sito Web

Nome sito: Pool di applicazioni:

Directory contenuto

Percorso fisico:

Autenticazione pass-through

Binding

Tipo: Indirizzo IP: Porta:

Nome host:

Esempio: www.contoso.com o marketing.contoso.com

Avvia sito Web immediatamente

Figura 20.7 - Configurazione di nuovo sito web all'interno di IIS per ospitare l'applicazione ASP.NET.

Dalla [Figura 20.7](#) si possono notare alcune caratteristiche: il nome del sito è rilevante solamente a livello dell'IIS, il nome dell'host, al contrario, è rilevante quando lo si vuole esporre all'esterno tramite certificati SSL e domini personalizzati, l'application pool selezionato è quello impostato nel passaggio precedente e, infine, il percorso fisico corrisponde alla cartella in cui è stata effettuata la pubblicazione tramite Visual Studio o tramite il comando `dotnet publish`. Andando ad analizzare l'output generato dalla pubblicazione di un'applicazione ASP.NET, troviamo, tra tutti, un file

fondamentale per l'avvio dell'applicazione, ovvero il **web.config** mostrato nell'[esempio 20.10](#), che serve a specificare all'IIS tutti i parametri relativi all'applicazione e all'IIS stesso.

Esempio 20.10

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule"
        resourceType="Unspecified" />
    </handlers>
    <aspNetCore processPath="dotnet" arguments=".\\Capitolo20.dll"
      stdoutLogEnabled="false" stdoutLogFile=".\logs\stdout" />
  </system.webServer>
</configuration>
```

Infatti, nell'[esempio 20.10](#) è evidente che si sta specificando, nel nodo relativo al web server (IIS), che deve essere caricato il modulo `AspNetCoreModule`, proprio quello che è stato installato con il bundle: è grazie a questo che IIS, partendo, sarà in grado di capire che deve caricare il modulo corretto, e quindi avviare i comandi definiti dagli attributi `processPath` e `arguments` del nodo `aspNetCore`, che rappresentano, guarda caso, i comandi di avvio dell'applicazione stessa. La generazione del file `web.config` è automatica sui progetti che hanno impostato come target di progetto `Microsoft.NET.Sdk.Web` e in cui non è già presente un file che sia personalizzato. Questo passaggio è obbligatorio e fondamentale, perché senza l'applicazione stessa non sarebbe in grado di partire: pertanto questo file di configurazione non va mai rimosso. Se tutti i passaggi elencati sono stati eseguiti correttamente, navigando all'indirizzo <http://localhost:8080> (o nella porta definita in fase di setup) saremo nuovamente in grado di visualizzare l'applicazione creata.

La pubblicazione all'interno dei web server che abbiamo affrontato finora è piuttosto semplice ma richiede ancora una cosa che, nel mondo moderno, non è più detto che sia disponibile: il server fisico. Infatti, sempre più spesso si sta cercando di spostare tutti i workload in ambienti cloud, in modo da risparmiare quanto più possibile in termini di costi e tempi di rilascio, ma garantire al tempo stesso una maggiore scalabilità sia

orizzontale sia verticale, praticamente immediata, di migliaia istanze che a oggi non è possibile in data center on-premise. Proprio per via di queste premesse, discuteremo ora di com'è possibile portare la stessa applicazione web anche in un ambiente cloud pubblico di Microsoft Azure e i servizi gestiti.

Publicazione con Microsoft Azure

Microsoft Azure offre, come gli altri cloud vendor, la possibilità di creare servizi e scalarli potenzialmente all'infinito, secondo le proprie esigenze, con una novità per quanto riguarda la fatturazione: al contrario di quanto avviene on-premise in cui il costo dell'hardware e di eventuali servizi necessari alle applicazioni devono essere sostenuti a priori, con l'avvento del cloud i costi sono limitati alle risorse che vengono effettivamente utilizzate. Entrando nel mondo delle applicazioni web, abbiamo già visto come sia possibile configurare un proprio web server e fare l'hosting. Pertanto, se si volesse andare nel cloud, il primo passo sarebbe quello di utilizzare il servizio delle macchine virtuali. Il problema nell'adottare una soluzione di questo tipo è solitamente relativo ai costi che si riflettono su diversi aspetti: manutenzione del sistema operativo, gestione delle patch, configurazione del web server, configurazione della scalabilità delle macchine virtuali, gestione del cluster per mantenere l'alta affidabilità, creazione del load balancing e così via. Una cosa simile si potrebbe dire anche per quanto riguarda i servizi Windows. Analizzando meglio questi punti, è abbastanza evidente che non si trae alcun vantaggio nell'andare nel cloud con una soluzione del genere, perché ci si porterebbero dietro gli stessi problemi dell'on-premise a meno della gestione dell'hardware. Per questo conviene adottare quelli che vengono definiti servizi gestiti (PaaS) in cui tutto l'hardware e il servizio stesso vengono gestiti direttamente da Microsoft, lasciandoci solamente possibilità di personalizzazioni sull'ambiente, senza avere l'accesso alle macchine. Il tutto si traduce in una serie di vantaggi e di riduzioni dei costi, sia da parte del cloud, perché sono coinvolte meno risorse, sia da parte delle aziende, perché sono necessarie meno persone specializzate sull'infrastruttura.

Il servizio gestito in Microsoft Azure che si occupa dell'hosting delle applicazioni web si chiama **App Service** e per utilizzarlo è richiesta

solamente l'attivazione di una sottoscrizione (anche gratuita) che può essere fatta da questo link: <http://aspit.co/boz>. Una volta attivato l'account, tutto il resto delle operazioni può essere fatto direttamente all'interno di Visual Studio tramite una serie di menu guidati: il primo passo è quello di cliccare con il tasto destro sul progetto che si vuole pubblicare e quindi scegliere l'opzione "Publish". Questo mostrerà a schermo un menu come quello evidenziato nella [Figura 20.8](#).

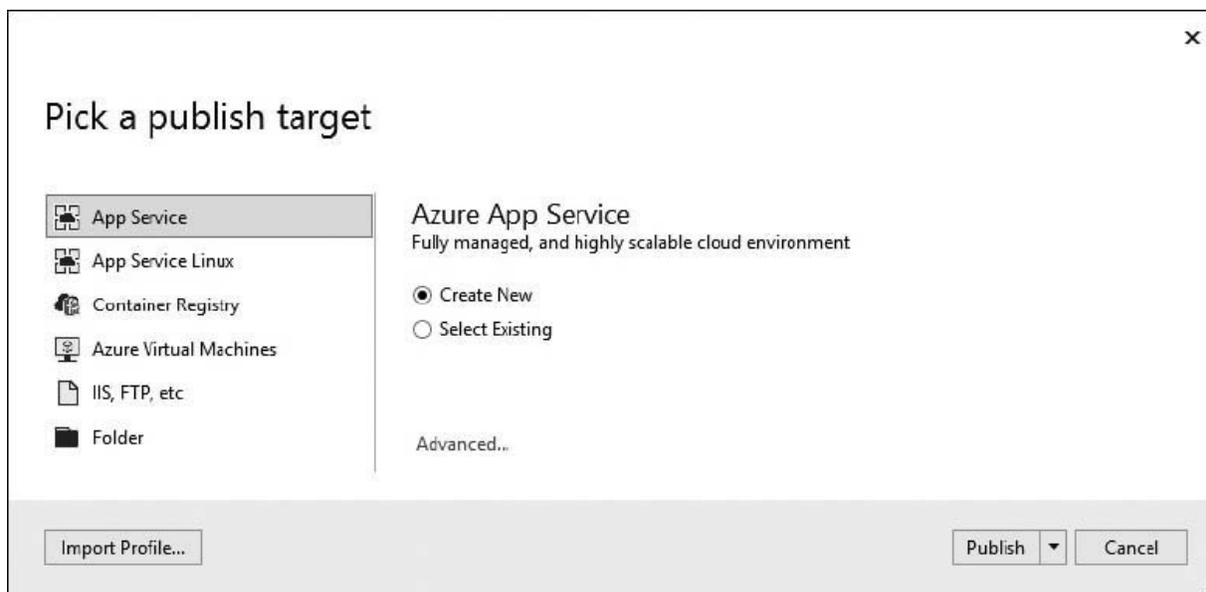


Figura 20.8 - Il menu di pubblicazione di Visual Studio offre diversi target, tra cui gli Azure App Service (Windows o Linux), macchine virtuali, IIS/FTP e cartelle definite in locale.

Come si può notare nella [Figura 20.8](#), sono disponibili diverse opzioni, tra cui la stessa macchina virtuale in cui si dovrà configurare il web server, piuttosto che la pubblicazione tramite IIS, FTP o una cartella, che non fanno altro che richiamare il comando `dotnet publish` (in caso di un'applicazione .NET Core) in un percorso specifico. Tra le opzioni relative al cloud, invece, ci sono due livelli di App Service perché, se si sta sviluppando un'applicazione ASP.NET Core, si potrebbe voler distribuire l'applicazione non solo su Windows ma anche in ambiente Linux, ma discuteremo di come questo venga fatto successivamente parlando dei Docker container. Prima di creare il servizio, dato che al momento ancora da portale non è stato creato, è necessario assicurarsi che le impostazioni di

pubblicazione dell'applicazione stessa siano corrette: pertanto, si può cliccare sul pulsante “Advanced” ed effettuare le modifiche, come viene illustrato nella [Figura 20.9](#).

Il servizio degli App Service prevede già una installazione di .NET Framework e .NET Core, perciò si può sfruttare una tipologia di deployment framework-dependent, come mostrato nella [Figura 20.11](#) ma, nel caso in cui ci sia l'esigenza di un altro runtime, si può comunque fare il deployment di tipo self-contained con .NET Core per portarsi tutte le dipendenze necessarie. Salvate le impostazioni di pubblicazione e cliccate il pulsante “Publish” del menu della [Figura 20.8](#), verrà richiesto di fare il login con l'account Microsoft con il quale è stata creata la sottoscrizione di Azure e quindi verrà mostrato il menu della [Figura 20.10](#), in cui sarà possibile creare la nuova risorsa.

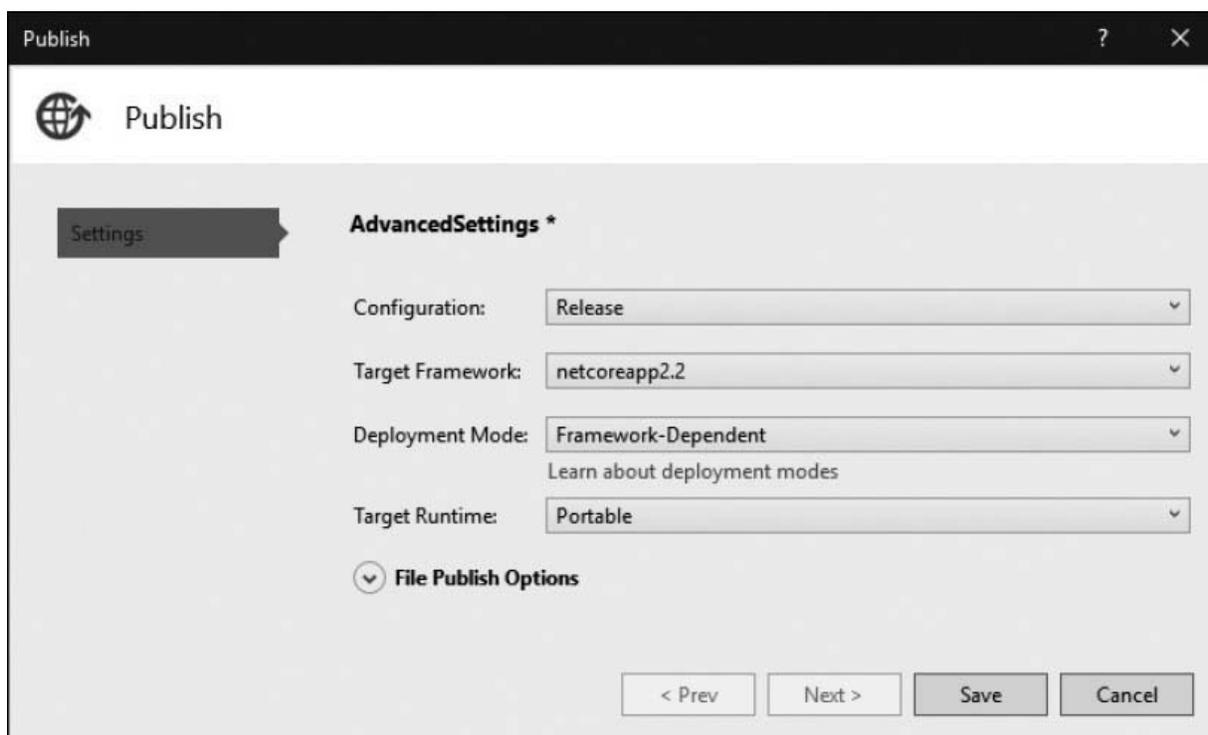


Figura 20.9 - Ogni progetto ASP.NET Core può avere diverse modalità di pubblicazione framework-dependent, che richiede il framework installato sul server, oppure self-contained, che viene pubblicato insieme all'applicazione stessa).

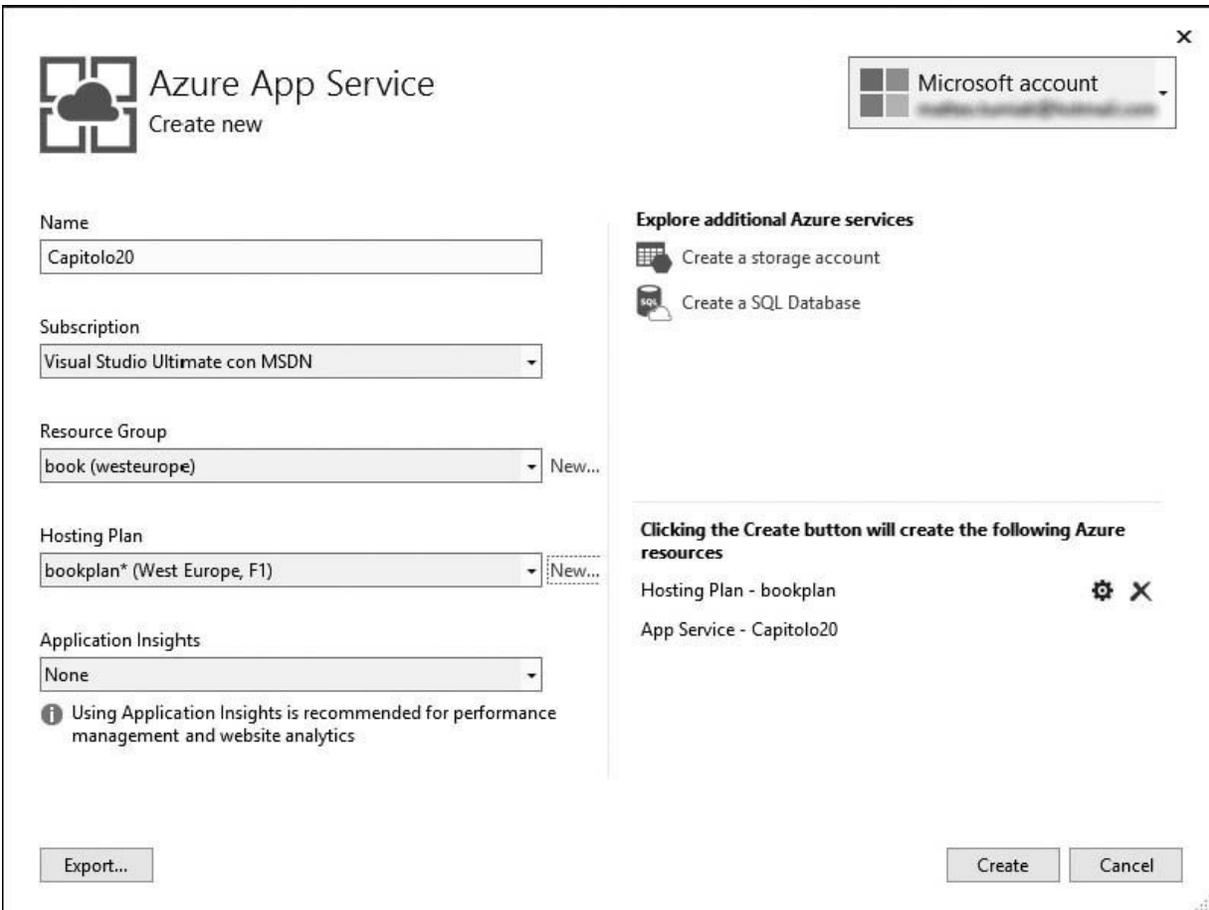


Figura 20.10 - Il menu di creazione di una nuova istanza di Azure App Service permette di aggiungere anche servizi esterni come un Azure SQL Database o un Azure Storage Account per salvare dati.

La creazione di una nuova risorsa di tipo App Service richiede quattro parametri:

- ❑ **Name:** rappresenta il nome DNS che avrà l'applicazione, perciò deve essere unico in tutto il mondo. L'URL completo dell'applicazione sarà <https://{Name}.azurewebsites.net> di default, ma sarà possibile cambiarlo tramite portale con l'aggiunta dei certificati SSL e domini personalizzati.
- ❑ **Subscription:** la sottoscrizione attiva verso la quale verrà effettuata la fatturazione mensile.

- ❑ **Resource Group:** permette di rappresentare un elenco di risorse che sono correlate fra loro (come per esempio un sito web e il relativo database) in un gruppo in cui sono tutte visibili.
- ❑ **Hosting plan:** indica il piano di servizio, ovvero la grandezza della macchina di una singola istanza che eseguirà l'applicazione e la sua posizione geografica.

Eventualmente è anche possibile configurare Application Insights, un sistema di telemetria basato su Microsoft Azure, ma non lo affronteremo nel corso del capitolo e pertanto può essere ignorato. La configurazione del piano di servizio viene fatta tramite un menu secondario, come è visibile nella [Figura 20.11](#).

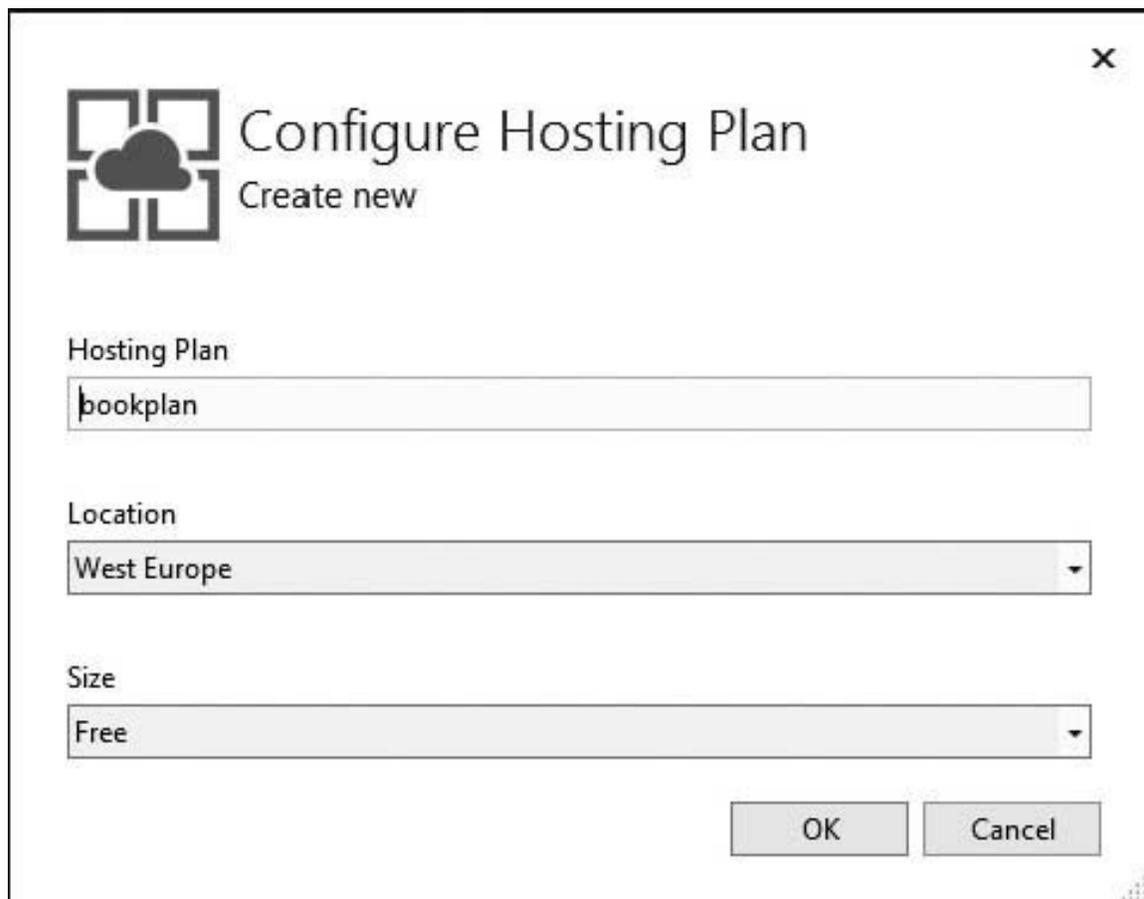


Figura 20.11 - La configurazione del piano di servizio include la scelta del data center e della dimensione della singola istanza dell'App Service.

Una volta configurata la risorsa base per l'hosting dell'applicazione web, è eventualmente possibile configurare il rilascio anche delle risorse collegate come, per esempio, il database. La procedura sarà molto simile, però su Microsoft Azure verrà creata, per esempio, una risorsa gestita di tipo SQL Database, che permetterà di gestire i dati relazionali come se fosse un vero e proprio SQL Server. Cliccando sul pulsante "Create" del menu mostrato nella [Figura 20.10](#), verranno eseguite due operazioni: la prima è la creazione, all'interno della cartella Properties del progetto, di un file chiamato {Name}-WebDeploy.pubxml contenente tutta la configurazione appena effettuata, mentre la seconda è la pubblicazione all'interno del servizio cloud. Quando l'operazione di pubblicazione si concluderà, saremo reindirizzati dal browser al vero e proprio sito web appena creato, come visibile nella [Figura 20.12](#).

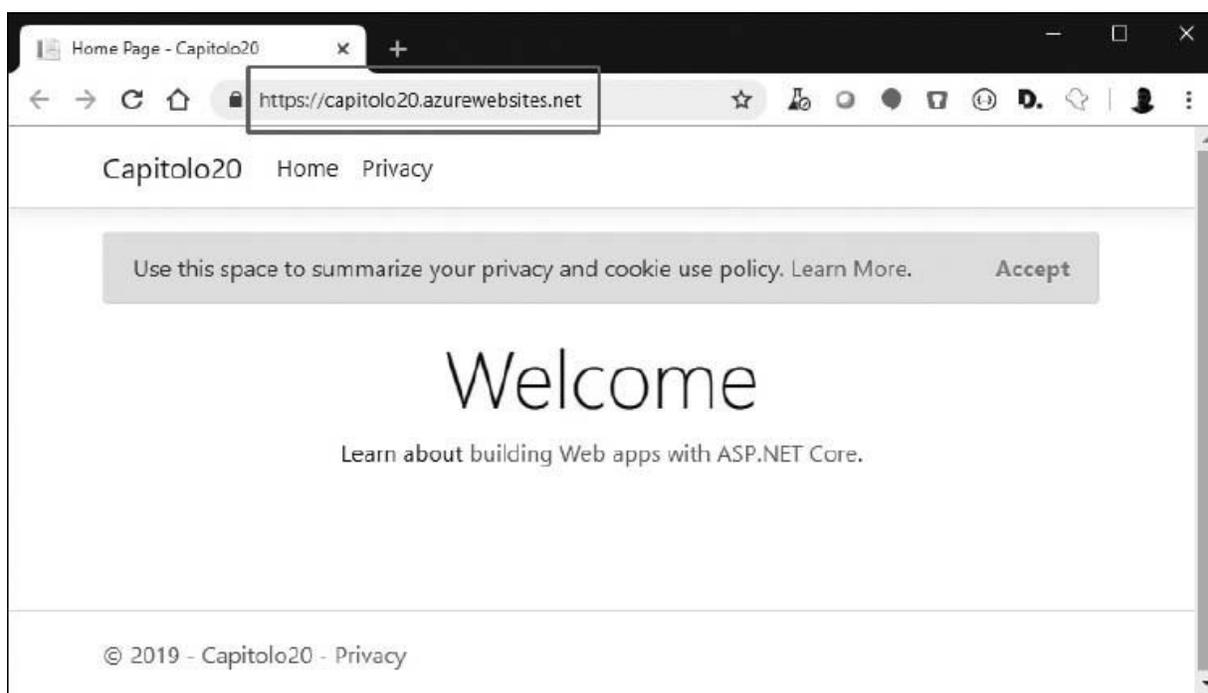


Figura 20.12 - L'applicazione distribuita su App Service è raggiungibile tramite un URL pubblico.

Le pubblicazioni che verranno eseguite successivamente, per via di modifiche all'interno del codice, saranno molto più rapide perché la configurazione è già salvata all'interno del publishing profile e, per via del

meccanismo di deployment scelto, verranno pubblicate solamente le modifiche e non tutta l'applicazione.

Accedendo al portale su <https://portal.azure.com> sarà possibile modificare alcune delle impostazioni del servizio creato, tra cui, per esempio, l'attivazione delle WebSocket per SignalR, piuttosto che HTTP/2 o, ancora, l'impostazione delle variabili d'ambiente come ASPNETCORE_ENVIRONMENT per modificarne la tipologia di esecuzione al volo, come mostrato nella [Figura 20.13](#).

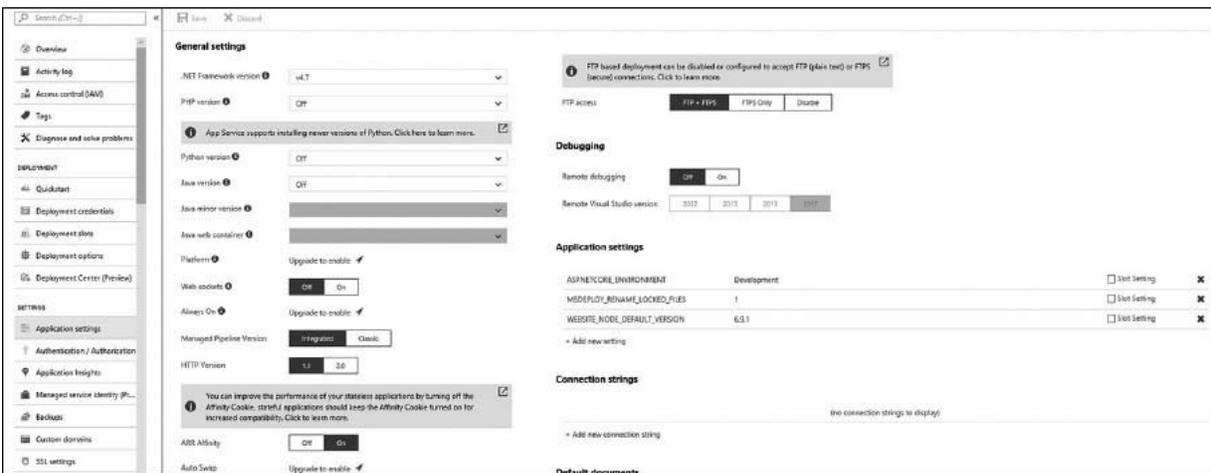


Figura 20.13 - All'interno dell'App Service specificato si possono cambiare le impostazioni relative all'applicazione stessa ma anche all'ambiente sulla quale viene eseguita.

Un aspetto particolarmente interessante degli App Service è che offrono la possibilità di attaccare il debugger di Visual Studio per determinare come mai alcuni problemi, per esempio, si verificano solamente una volta pubblicata l'applicazione. Questa funzionalità si chiama **Remote Debugging** e va attivata in modo esplicito, selezionando opportunamente la versione di Visual Studio che si desidera utilizzare, direttamente all'interno del portale di Azure ed è inoltre necessario che l'applicazione distribuita nell'App Service sia stata compilata in modalità di debug. All'interno di Visual Studio, aprendo la finestra del Cloud Explorer, sarà possibile scegliere tra i nodi *Azure* → *App Service* → *Resource Group* l'applicazione che si deve debuggare e, cliccando con il tasto destro del mouse sopra di

essa, selezionare l'opzione "Attach Debugger", come mostrato nella [Figura 20.14](#).

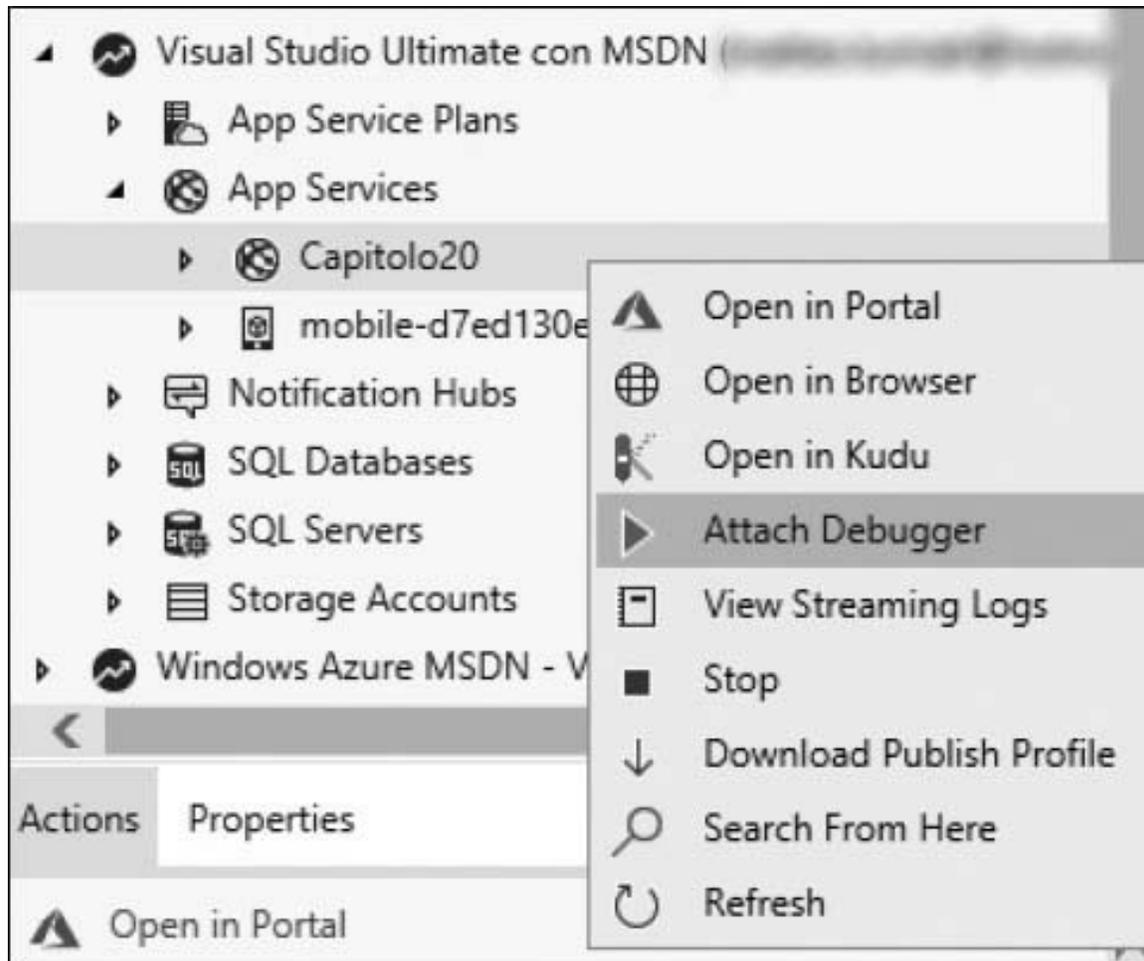


Figura 20.14 - L'opzione "Attach Debugger" è disponibile tramite il menu secondario dal Cloud Explorer sull'istanza di App Service selezionata.

Questa funzionalità è estremamente potente e non deve essere utilizzata in scenari di produzione: ogni qualvolta si incontra, per esempio, un breakpoint all'interno di una action tramite una route specificata dall'utente, tutta l'esecuzione dell'applicazione verrà bloccata fino a quando non verrà eseguito lo step successivo ed eventualmente rilasciato il debugger. Testare in remoto è comunque molto utile perché, essendo il servizio gestito, non è possibile essere a conoscenza dei dettagli implementativi dei server che offrono il servizio stesso. Pertanto, attivando il debugger, si potrebbero

scoprire situazioni non verificabili e non riproducibili nell'ambiente di sviluppo o di staging.

Gli Azure App Service includono molte altre funzionalità, fra cui la personalizzazione dei domini, l'autenticazione, la gestione delle performance e logging tramite Application Insights o la scalabilità automatica ma, poiché non sono il tema centrale del libro, rimandiamo alla documentazione ufficiale su <http://aspit.co/bo0> per approfondire le potenzialità di questi servizi gestiti.

Le applicazioni che si possono creare con il mondo .NET, però, non sono solo web e infatti spesso s'incontra l'esigenza di creare servizi o processi che girano in istanti temporali predeterminati o invocabili on-demand. Per questo in Azure nascono le Azure Functions, un servizio computazionale definito serverless, in quanto l'esecuzione del codice avviene a eventi, senza il bisogno di provisionare (e gestire) alcuna infrastruttura. Alla creazione della risorsa di tipo "Azure Function" in Visual Studio, si aprirà un menu il cui scopo sarà quello di scegliere il trigger per determinare l'invocazione dell'applicazione e la scelta, come si può notare nella [Figura 20.15](#), spazia tra l'invocazione via web (tramite chiamata HTTP con un URL), a variazioni temporali, oppure tramite la lettura di messaggi nelle code (Queue e Service Bus) e così via.

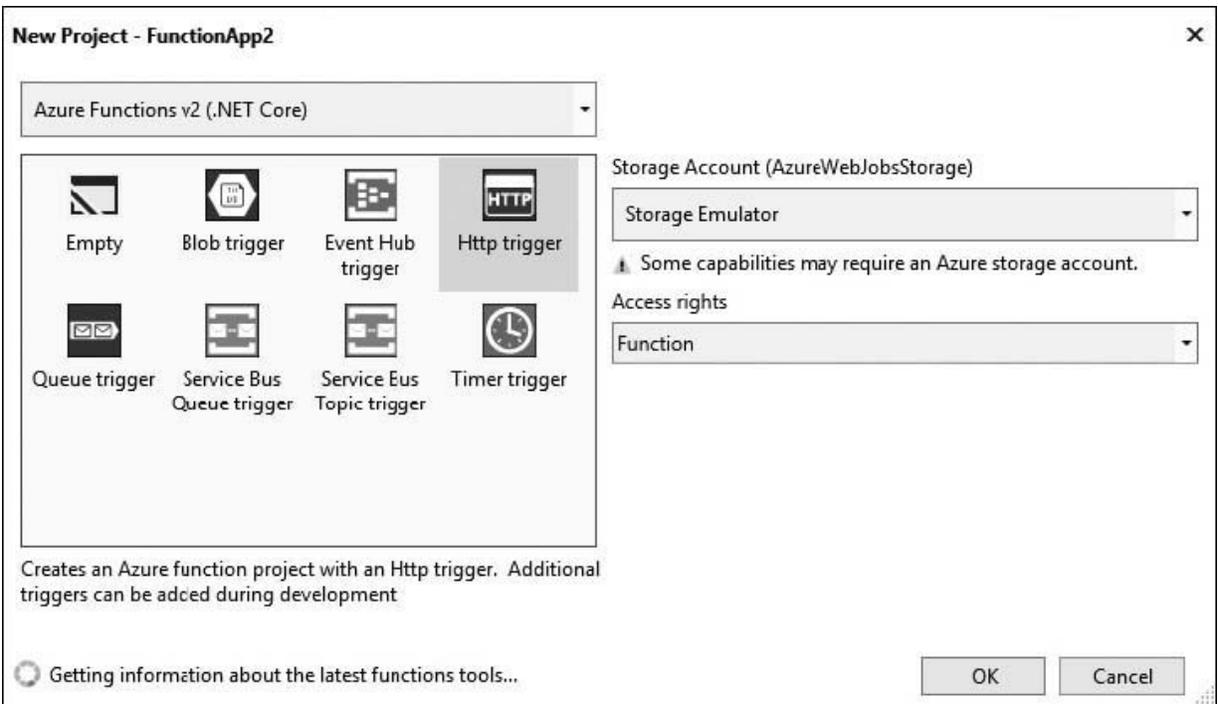


Figura 20.15 - Durante la creazione di un'applicazione di tipo Azure Function, Visual Studio chiederà la tipologia di framework (.NET o .NET Core) e la tipologia di trigger che può scatenare la sua esecuzione. Un account di Azure Storage è obbligatorio per determinati tipi di Azure Functions, poiché è utilizzato per tenere traccia dello stato delle esecuzioni e dei log, mentre la parte di accessi serve per restringere l'esecuzione tramite l'uso di token.

Sempre all'interno dello stesso menu, è possibile scegliere la versione: la prima versione (v1) supporta .NET Framework, mentre la seconda (v2), più moderna, supporta .NET Core. Una volta creato il progetto, noteremo come questo sia una normale console application, in cui scrivere quello che deve avvenire al momento dell'invocazione della funzione stessa.

Esempio 20.11

```
public class Capitolo20
{
    [FunctionName("LogFunction")]
    public void Run([HttpTrigger(AuthorizationLevel.Function, "get")] HttpRequest
    req, ILogger log)
    {
        log.LogInformation("La function è in esecuzione...");
    }
}
```

```
}  
}
```

Avviando il progetto illustrato nell'[esempio 20.11](#), partirà una console application che esporrà un server HTTP a cui inviare le request verso l'endpoint della function, in sola modalità GET, come viene definito dai parametri in ingresso. Cambiando il trigger da `HttpTrigger` a `TimerTrigger`, per esempio, verrebbe cambiata la modalità di esecuzione e l'applicazione stessa si sarebbe eseguita secondo la validazione di una CRON expression.

Lo scopo delle Azure Function, però, è quello di limitare l'esecuzione del codice a una sola “funzione”, per l'appunto, che deve essere più semplice e veloce possibile nell'esecuzione, per poter avere un risparmio sui consumi, anche in termini economici. In generale, soluzioni come quelle che abbiamo affrontato all'interno di questo capitolo funzionano ma non sono l'ideale nei casi in cui si stia sviluppando un sistema complesso orientato ai micro-servizi: in questi casi specifici, infatti, distribuire e orchestrare tutte le versioni, piuttosto che gestire le dipendenze fra i vari servizi, diventa via via più complesso: pertanto c'è bisogno di un sistema che semplifichi ulteriormente il rilascio.

[Pubblicazione con Docker](#)

Uno dei problemi più sentiti da parte degli sviluppatori è che non è possibile lavorare con ambienti completamente isolati e replicabili: infatti, spesso si sente parlare del fenomeno “*it works on my machine*” (letteralmente “funziona sul mio PC”) per via del fatto che l'installazione di una nuova dipendenza, se non comunicata agli altri membri del team di sviluppo o configurata correttamente, potrebbe portare a una instabilità del sistema fino al non funzionamento dello stesso, cosa che in ambienti di produzione non dovrebbe succedere. Mettendosi invece nei panni del team di operation che deve gestire la messa in produzione delle varie applicazioni, il tutto diventa ancora più complicato perché spesso non si hanno gli script giusti per configurare correttamente gli ambienti. Inoltre, per garantire l'affidabilità e la disponibilità del servizio, si finisce spesso col

creare nuove macchine virtuali o fisiche, per fare repliche e hosting di una sola applicazione, e questo implica che non solo ci saranno da gestire gli aggiornamenti applicativi ma che bisognerà anche controllare gli aggiornamenti relativi al sistema operativo, installare patch di sicurezza, riconfigurare il web server allo stesso modo su tutte le macchine e così via. In ambienti in cui ci sono decine e decine di server dedicati, questo è un problema: non è sempre possibile avere script che ricreano la stessa configurazione su tutte le macchine, fare un deployment richiede tanto tempo in relazione al numero dei server e, oltre a questo, ci si ritrova con tanto hardware che in realtà non fa nulla. Proprio per rimediare a questi problemi, si sviluppa il mondo dei container, capeggiato principalmente da aziende come Docker, il cui scopo è proprio quello di avere la virtualizzazione di una sola porzione della macchina virtuale, così da risparmiare sui costi, mantenere alta l'efficienza senza dover gestire il sistema operativo, creando un sistema isolato, il container, in grado di tenere al suo interno tutte le dipendenze, framework e configurazioni di cui il sistema stesso ha bisogno per funzionare.

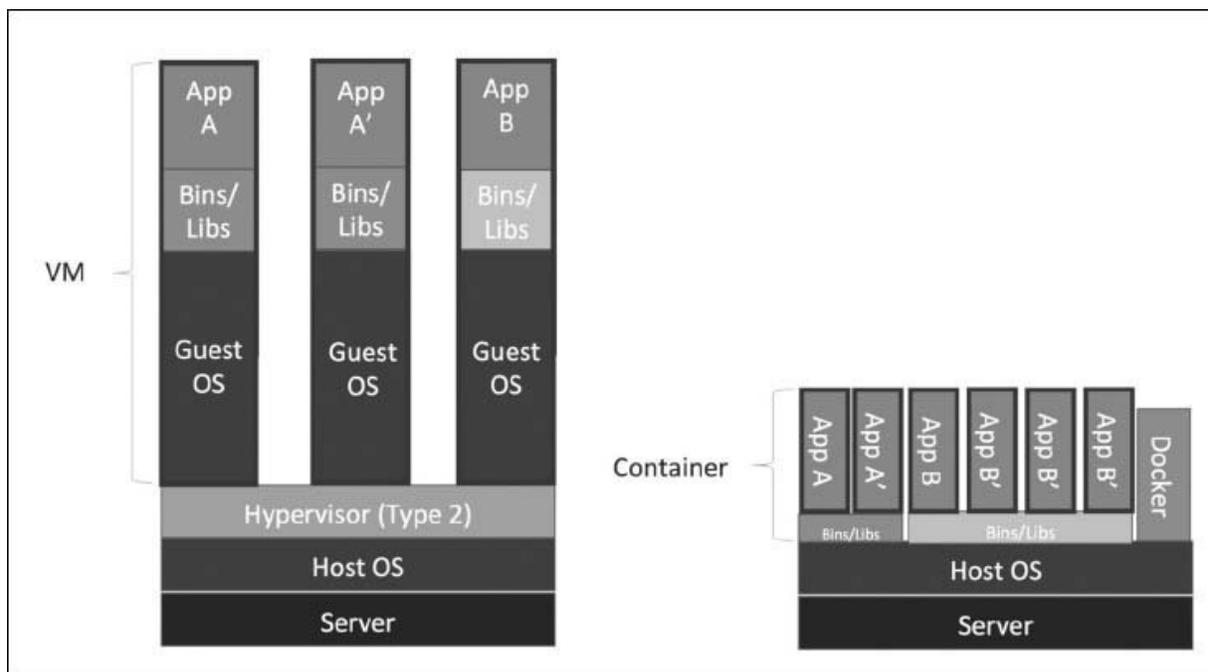


Figura 20.16 - Su una macchina virtuale c'è isolamento tra le applicazioni, ma due istanze della stessa applicazione non possono condividere delle librerie in comune, perciò sono molto grandi, costose, inefficienti e complesse da gestire. I container, invece, vivono in

isolamento e possono condividere le librerie attraverso un host (in questo caso il Docker Host), senza bisogno di un sistema operativo intermedio come nelle machine virtuali.

Come viene dimostrato nella [Figura 20.16](#), i container, non avendo più il sistema operativo delle VM da gestire e avendo la condivisione delle risorse e delle dipendenze, portano a una serie di vantaggi:

- ❑ **Controllo granulare:** l'approccio è orientato ai micro-servizi, anche se può funzionare con qualsiasi tipologia di applicazione (incluse le applicazioni console).
- ❑ **Testing facilitato:** il testing (e il funzionamento) di un'applicazione in locale su container produrrà gli stessi risultati che in un qualsiasi altro ambiente, compreso produzione.
- ❑ **Deployment semplificato:** l'applicazione è pacchettizzata con tutte le sue dipendenze in un solo componente, ovvero il container. In caso di rilascio di una nuova versione, i container possono girare in parallelo, consentendo anche la creazione di scenari di A/B testing.
- ❑ **Avvio rapido:** meno di un secondo, in genere, poiché c'è cache sulle immagini che creano i container.

Considerati tutti i vantaggi di questa nuova tipologia di sviluppo e deployment, Microsoft ha deciso di approcciarla facendo un'integrazione più semplice possibile direttamente all'interno di Visual Studio per i progetti .NET e ASP.NET (non desktop). Nonostante cambi tutta la tecnologia, la semplicità nasce dal fatto che, tramite Visual Studio, il modello di sviluppo non cambia e si continuerà a lavorare come sempre fatto, ma facendo il click con il tasto destro del mouse e selezionando *Add* → *Docker Support*, come mostrato nella [Figura 20.17](#), si avranno tutti i vantaggi illustrati nel paragrafo precedente.

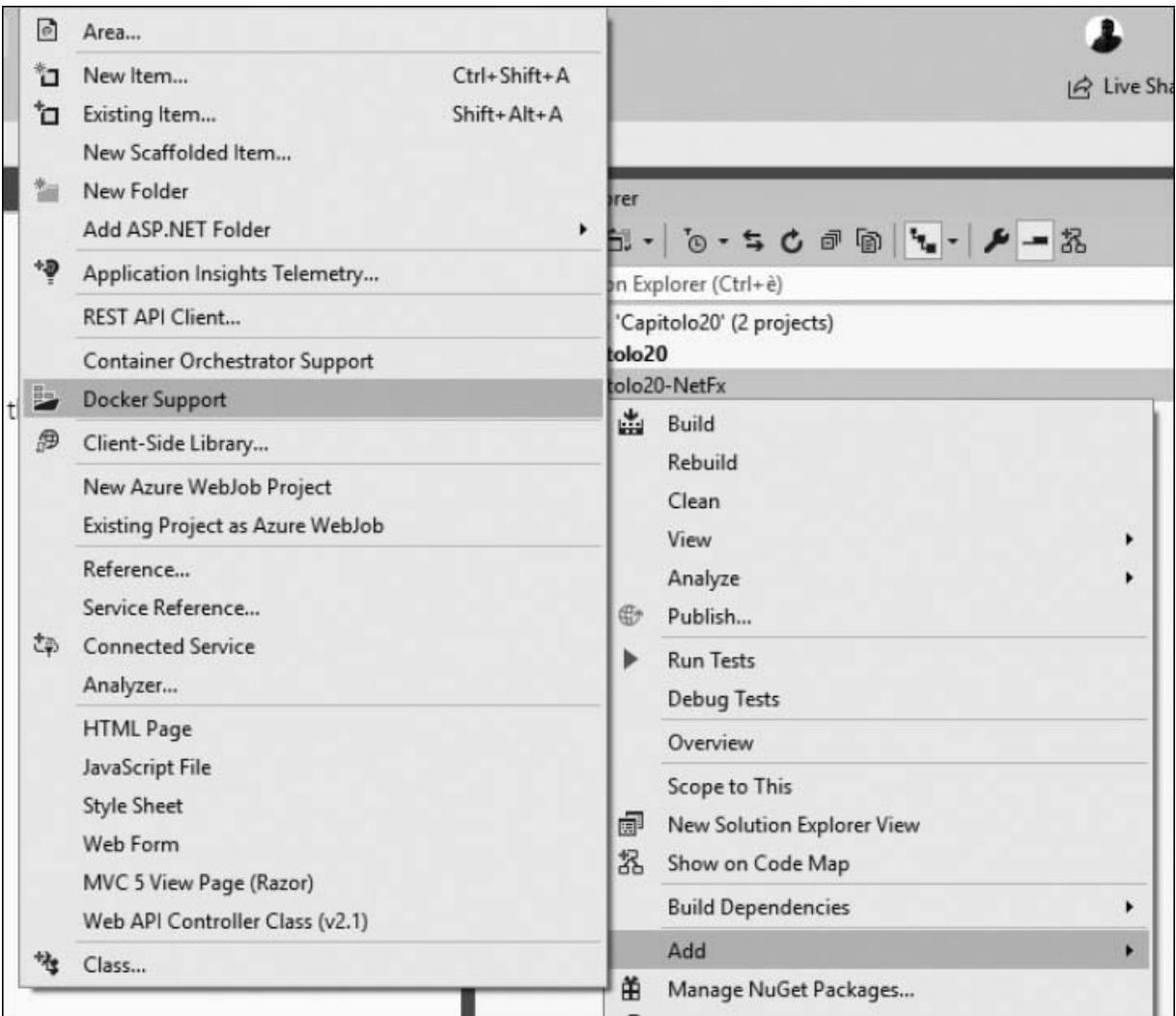


Figura 20.17 - L'aggiunta del supporto a Docker è fatta tramite il tasto destro sul progetto che si vuole containerizzare e selezionando l'opzione Add → Docker Support.

Quello che avviene dopo questa operazione mostrata nella [Figura 20.17](#), è la creazione di un file chiamato `Dockerfile` e in base alla versione di Visual Studio, alla versione e tipologia di framework o al numero di servizi che si vuole collegare, potrebbe avvenire anche la creazione di un progetto chiamato `docker-compose` a livello di solution. Il nuovo progetto (di natura opzionale) contiene una serie di file, tra cui il `.dockerignore` che serve ai sistemi di source control per non fare il commit di alcuni tipi di file relativi a Docker, e il `docker-compose.yml`, un file con sintassi YAML molto simile

al JSON, in cui vengono descritti tutti i servizi che devono essere creati, come dimostra l'[esempio 20.12](#).

Esempio 20.12

```
version: '3.4'

services:
  capitolo20:
    image: capitolo20
    build:
      context: .
      dockerfile: Capitolo20/Dockerfile
```

Nell'[esempio 20.12](#) si può notare come il servizio, in questo caso, sia solo uno, perché il progetto sul quale abbiamo abilitato questa tecnologia è solamente uno, e verrà creato a partire dal file `Dockerfile` contenuto nella sua soluzione. Quello che verrà creato non è il vero e proprio container ma solo un'immagine che avrà il nome specificato nel tag `image`, ovvero "capitolo20": il container non sarà altro che l'esecuzione dell'immagine stessa: per questo, a livello di avvio e scalabilità, è molto più efficiente.

Il `Dockerfile`, ovvero il file di configurazione che specifica a Docker come deve essere costruita l'immagine, è stato creato di default per funzionare su Linux e punta alla versione dell'SDK che abbiamo dichiarato a livello di progetto, in questo caso la versione 2.2 di ASP.NET Core. Pertanto il risultato ottenuto in automatico da Visual Studio sarà qualcosa di simile a quello mostrato nell'[esempio 20.13](#).

Esempio 20.13

```
FROM mcr.microsoft.com/dotnet/core/aspnet:2.2-stretch-slim AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/core/sdk:2.2-stretch AS build
WORKDIR /src
COPY ["Capitolo20/Capitolo20.csproj", "Capitolo20/"]
RUN dotnet restore "Capitolo20/Capitolo20.csproj"
COPY . .
WORKDIR "/src/Capitolo20"
RUN dotnet build "Capitolo20.csproj" -c Release -o /app
```

```
FROM build AS publish
RUN dotnet publish "Capitolo20.csproj" -c Release -o /app

FROM base AS final
WORKDIR /app
COPY --from=publish /app .
ENTRYPOINT ["dotnet", "Capitolo20.dll"]
```

Nell'[esempio 20.13](#) viene mostrata una funzionalità particolare di Docker chiamata multi-staged build e, infatti, si stanno susseguendo quattro fasi distinte dalla parola chiave `FROM` che identifica un nuovo gruppo di operazioni:

- ❑ **Fase 1:** viene costruita una prima immagine, temporanea e nominata *base*, a partire da un'immagine già esistente e pre-configurata che contiene il runtime di ASP.NET Core 2.2. Sebbene non sia obbligatorio, è molto utile per non partire dall'immagine vuota di Docker chiamata `SCRATCH`, in cui bisognerebbe lanciare comandi PowerShell o Bash in sequenza per installare tutto il runtime. Successivamente viene creata una cartella chiamata *app* e quindi vengono aperte le porte 80 e 443 che serviranno a comunicare con l'esterno del container.
- ❑ **Fase 2:** viene costruita una seconda immagine, temporanea e nominata *build*, a partire da un'altra immagine già esistente, che però questa volta contiene l'SDK di .NET Core 2.2; quindi vengono copiati i file di progetto e tutto il codice sorgente e, una volta impostata la working directory nella cartella corretta, viene lanciato il comando `dotnet build` con la configurazione di `Release` per compilare il progetto.
- ❑ **Fase 3:** viene costruita una terza immagine, sempre temporanea e nominata *publish*, a partire dall'immagine *build* creata nella Fase 2, in cui avviene la pubblicazione tramite il comando `dotnet publish` in configurazione di `Release`, in cui viene anche specificato il percorso della cartella di output. Questo passaggio poteva anche essere inglobato con l'immagine generata in precedenza, ma per avere una maggiore separazione tra le varie operazioni è stata suddivisa in un'immagine a parte. Il comando `dotnet publish` è

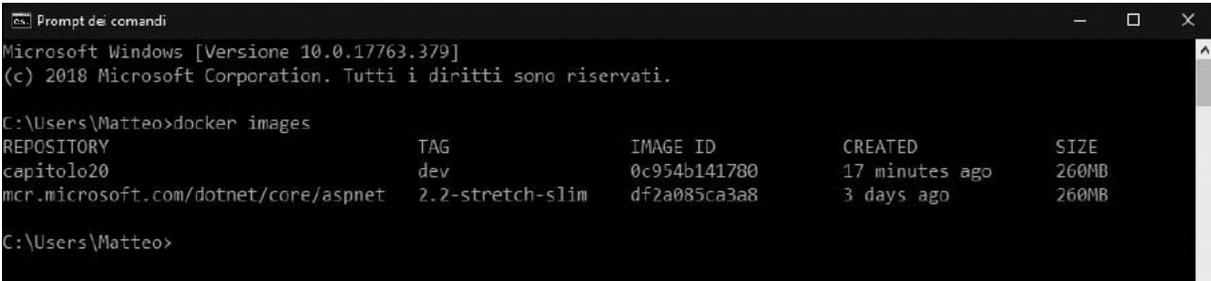
possibile perché, dato che l'immagine di build è accessibile, lo saranno anche i sorgenti e i vari file di progetto in essa contenuti.

- ❑ **Fase 4:** viene costruita una quarta e ultima immagine a partire dall'immagine *base* costruita sul runtime nella Fase 1, in cui avviene la copia dei soli file contenuti nella cartella di pubblicazione dell'immagine costruita nella Fase 3, così da non avere più i sorgenti ma solo le DLL compilate. Quindi viene impostato l'entrypoint che sarà corrispondente al comando `dotnet Capitolo20.dll` che farà partire a tutti gli effetti l'applicazione ASP.NET.

Come possiamo notare, le immagini base dalla quale si è partiti sono due: il runtime e l'SDK. Poiché tutti i processi di build e pubblicazione vengono eseguiti direttamente all'interno del container, data la definizione del `Dockerfile`, c'è bisogno di un'immagine con tutto l'SDK per effettuare queste due operazioni, mentre **quando si deve eseguire l'applicazione è sufficiente il runtime**, che permette anche di risparmiare diverse centinaia di megabyte di spazio. L'operazione di build e pubblicazione all'interno del container stesso non è molto utile quando si lavora in un ambiente in cui c'è a disposizione Visual Studio ma è fondamentale in ambienti in cui non c'è installato .NET. È possibile vedere le immagini scaricate e la loro dimensione su disco lanciando il comando `docker images`, come dimostra la [Figura 20.18](#).

L'immagine generata partendo dal `Dockerfile` dell'[esempio 20.13](#) sarà, come è visibile nella [Figura 20.18](#), di circa 260Mb che, considerando un ambiente Linux, non è poco: ogni volta che si deve rilasciare un aggiornamento, viene potenzialmente spostata tutta l'immagine (anche se non è del tutto vero, dato che viene verificata la cache sui vari strati di cui è composta). Quindi, più sarà grande, più tempo ci metterà a essere trasferita via network, più banda occuperà e più tempo ci metterà a partire in caso di scalabilità. Una delle novità introdotte a partire da .NET Core 2.1 riguarda l'introduzione di nuove immagini basate su Alpine, ovvero una variante di Debian, pensata per essere leggera. Infatti, l'immagine contenente il solo runtime arriva a pesare poco più di 80Mb, che rappresenta una differenza di circa 180Mb rispetto al passato. Tutte le immagini (anche per versioni basate su Windows) che possono essere utilizzate per runtime e SDK sono oggi disponibili nel repository privato di Microsoft chiamato *mcr*, ma sono

ancora visibili all'interno del Docker Hub pubblico di Microsoft, ovvero di un repository in cui sono salvate tutte le immagini precostruite e pronte da utilizzare, a partire da questo indirizzo: <http://aspit.co/bo2>.



```
Prompt dei comandi
Microsoft Windows [Versione 10.0.17763.379]
(c) 2018 Microsoft Corporation. Tutti i diritti sono riservati.

C:\Users\Matteo>docker images
REPOSITORY          TAG          IMAGE ID          CREATED          SIZE
capitolo20          dev         0c954b141780     17 minutes ago  260MB
mcr.microsoft.com/dotnet/core/aspnet  2.2-stretch-slim  df2a085ca3a8     3 days ago     260MB

C:\Users\Matteo>
```

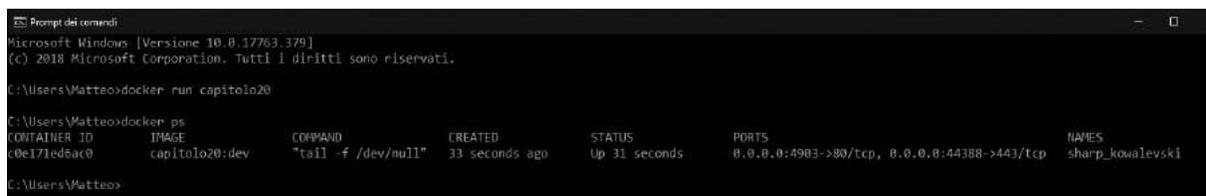
Figura 20.18 - Ogni immagine elencata dal comando `docker images` è composta da un nome, da un tag che ne specifica la versione, da un identificativo che la riferenzia in modo univoco, da una data di creazione e da una dimensione.

Una volta dichiarato come deve essere costruita l'immagine che conterrà quindi l'applicazione e specificate le varie ottimizzazioni come l'uso di Alpine, verrà il momento di eseguirla. Osservando bene Visual Studio, dopo l'aggiunta del supporto a Docker, si noterà come sia stato cambiato il progetto di startup, che non sarà più l'applicazione web stessa ma Docker, aggiunto da Visual Studio. Questo consente a Visual Studio di interagire con Docker e creare le immagini specificate nei Dockerfile a partire (eventualmente) dalle specifiche dichiarate nel `docker-compose.yml` ma non solo: se in Docker è attiva la condivisione del disco, Visual Studio aprirà anche una porta per il debugger, e quindi sarà possibile testare le applicazioni all'interno dei container stessi che, di default, sarebbero isolati. Qualora invece si volesse procedere manualmente, sarà necessario lanciare solo due comandi, come mostrato nell'[esempio 20.14](#).

Esempio 20.14

```
docker build -t capitolo20 -f Capitolo20/Dockerfile .
docker run capitolo20
```

Come viene dimostrato nell'[esempio 20.14](#), infatti, il primo passaggio è quello di costruire l'immagine a partire dal Dockerfile creato in automatico da Visual Studio, con un nome specificato dal parametro `t`. Quindi, una volta creata l'immagine, si potrà effettivamente avviare il container con il suo nome. A questo punto l'applicazione ASP.NET verrà avviata con il comando definito dall'`ENTRYPOINT` del Dockerfile e ci sarà lo startup, come mostrato nella [Figura 20.19](#).



```
Microsoft Windows [Versione 10.0.17763.379]
(c) 2018 Microsoft Corporation. Tutti i diritti sono riservati.

C:\Users\Matteo>docker run capitolo20

C:\Users\Matteo>docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
c0e171ed6ac0      capitolo20:dev     "tail -f /dev/null" 33 seconds ago     Up 31 seconds      0.0.0.0:4983->80/tcp, 0.0.0.0:44388->443/tcp
sharp_kowalevski
```

Figura 20.19 - L'avvio (manuale in questo caso) del container con l'applicazione ASP.NET Core farà partire Kestrel e aprirà le porte 80 e 443, esposte all'interno del container, per fare in modo che sia raggiungibile dall'esterno.

Una volta creata l'immagine e verificato il corretto funzionamento del container, si può procedere alla pubblicazione. Esistono diverse soluzioni che sono in grado di eseguire e gestire container, ma lo strumento più semplice è quello che abbiamo già visto in precedenza, ovvero gli App Service: la versione per Linux è infatti in grado di prendere l'immagine ed eseguirla, lanciando i comandi che abbiamo eseguito manualmente o tramite Visual Studio. Gli App Service sono comodi solo per un caso molto specifico però, ovvero il caso in cui ci siano pochi servizi avviati come container, tutti definiti all'interno del file `docker-compose.yml` e tutti stateless. Nei casi in cui ci sia bisogno di avviare container che devono mantenere dati, piuttosto che container che necessitano di migliaia di istanze, gli App Service non rappresentano la soluzione migliore e, proprio per questo, in Azure sono disponibili altri sistemi come Azure Kubernetes Service (AKS), Service Fabric o Azure Container Instances (ACI) che permettono di mantenere cluster e ne garantiscono l'alta affidabilità e alta disponibilità. Maggiori informazioni su questi servizi sono disponibili su <http://aspit.co/bo8>.

Tutto quello che abbiamo visto riguardo ai container è particolarmente vero sia per le applicazioni .NET Core sia per le classiche applicazioni .NET (sia web sia console), che però necessitano di Windows Server come immagine base per poter essere avviate e quindi non possono godere di grandi vantaggi per quanto riguarda la dimensione dell'immagine o l'alta scalabilità del container. In ogni caso, le applicazioni non sono solo web e console, anzi, sono ancora in gran parte applicazioni desktop e, in quanto tali, seguono principi di pubblicazione differenti da quanto abbiamo visto finora.

Le applicazioni Windows

Quando parliamo di applicazioni Windows, in realtà includiamo diverse tipologie di applicazioni e tecnologie: da WinForms, WPF e UWP (la Universal Windows Platform di cui abbiamo parlato nei capitoli precedenti) fino ad arrivare anche a console application (che abbiamo già visto come distribuire in Azure). Mentre per WinForms e WPF per la distribuzione sono necessari strumenti particolari come ClickOnce (che essendo, fra l'altro, particolarmente datati, non tratteremo all'interno di questo capitolo), per esempio, per la Universal Windows Platform tutto è integrato nel framework e in Visual Studio, per cui pubblicare una nuova applicazione è molto più semplice, intuitivo e veloce.

Esistono tre modalità per la distribuzione delle applicazioni desktop:

- ❑ **Tramite il Microsoft Store:** è un app store pubblico che, in quanto tale, richiede che le applicazioni siano verificate e certificate e che gli sviluppatori abbiano acquistato una licenza annuale per fare la submission delle app.
- ❑ **Tramite web:** l'applicazione può essere scaricata e installata da chiunque, senza passare per le limitazioni imposte dallo store pubblico di Microsoft. La certificazione non verrà eseguita da Microsoft, ma il certificato per la verifica dell'identità dello sviluppatore, poiché l'applicazione proviene da fonti non sicure, rimane indispensabile per la fase di installazione.

- ❑ **In locale:** in questo caso l'applicazione è privata e può essere distribuita a client specifici manualmente o tramite sistemi enterprise come Intune e SCCM.

In base alla modalità di distribuzione scelta, possono essere creati uno o più tipi di pacchetti direttamente da Visual Studio:

- ❑ **Pacchetto dell'applicazione:** contiene l'applicazione per la distribuzione privata o per scopi di test (per esempio, debug remoto) in formato *appx* o *msix*.
- ❑ **Application bundle:** è una tipologia di pacchetto progettata per supportare più architetture (x86, x64 e ARM).
- ❑ **File di upload:** è un file con estensione *appxupload* o *msixupload* che viene creato per la distribuzione all'interno del Microsoft Store. Come nel caso precedente, anche questo file può essere, in realtà, un insieme di più architetture e può contenere simboli.

Prima di creare il pacchetto per distribuire l'applicazione, è però necessario andare a impostare le opzioni di pubblicazione, un po' com'è già stato fatto in precedenza per la pubblicazione in cloud. In questo caso la configurazione dell'applicazione stessa è centralizzata in un file chiamato `Package.appxmanifest` che viene incluso nei pacchetti creati durante la fase di deployment e che viene utilizzato dal sistema operativo per identificare il nome dell'applicazione per il menu "Start", per la ricerca, per impostare le icone e così via.

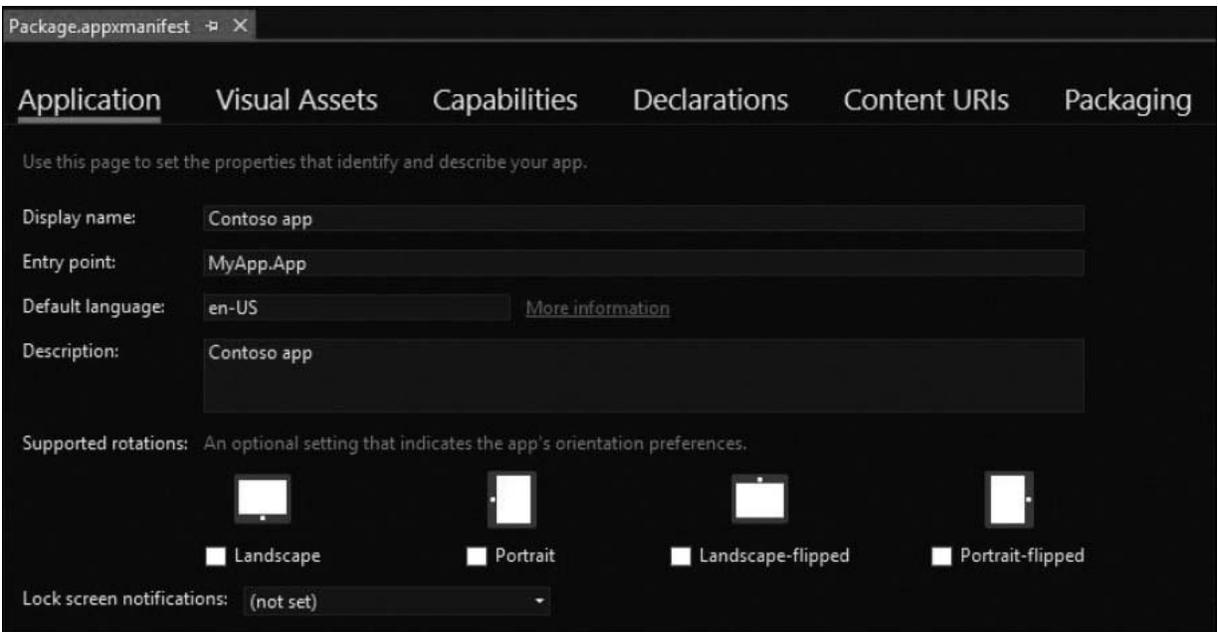


Figura 20.20 - Il file di manifest permette di inserire tutte le capability supportate dall'applicazione, URL per deep linking, informazioni sull'orientamento e lingue supportate dall'applicazione, immagini e icone, piuttosto che informazioni relative allo sviluppatore.

Una volta configurata l'applicazione per la distribuzione, come indicato dalla [Figura 20.20](#), è possibile procedere alla pubblicazione vera e propria tramite il click con il tasto destro sul progetto UWP e cliccando sul menu *Store* → *Create App Package*. Si aprirà quindi una finestra di dialogo che chiederà la funzione di distribuzione, ovvero se siamo interessati per lo store pubblico oppure privato e, una volta scelto, sarà possibile procedere alla creazione del pacchetto, specificando anche le tipologie di architetture supportate, come potete notare nella [Figura 20.21](#).

La pubblicazione all'interno dello store pubblico è un po' più articolata e non verrà affrontata nel dettaglio all'interno del libro per brevità: durante la fase di creazione del pacchetto è necessario collegare l'applicazione in locale con un'applicazione già precedentemente creata all'interno del portale sviluppatori del Microsoft Store, quindi si dovrà caricare il pacchetto e seguire tutto il processo di verifica (sicurezza, validazione dei certificati, uso di API improprie, dimensione delle icone, nomi e descrizioni ecc.) prima di avere l'app disponibile al download. Ulteriori dettagli sulla

pubblicazione per lo store sono disponibili all'interno della documentazione ufficiale su <https://aspit.co/buy>.

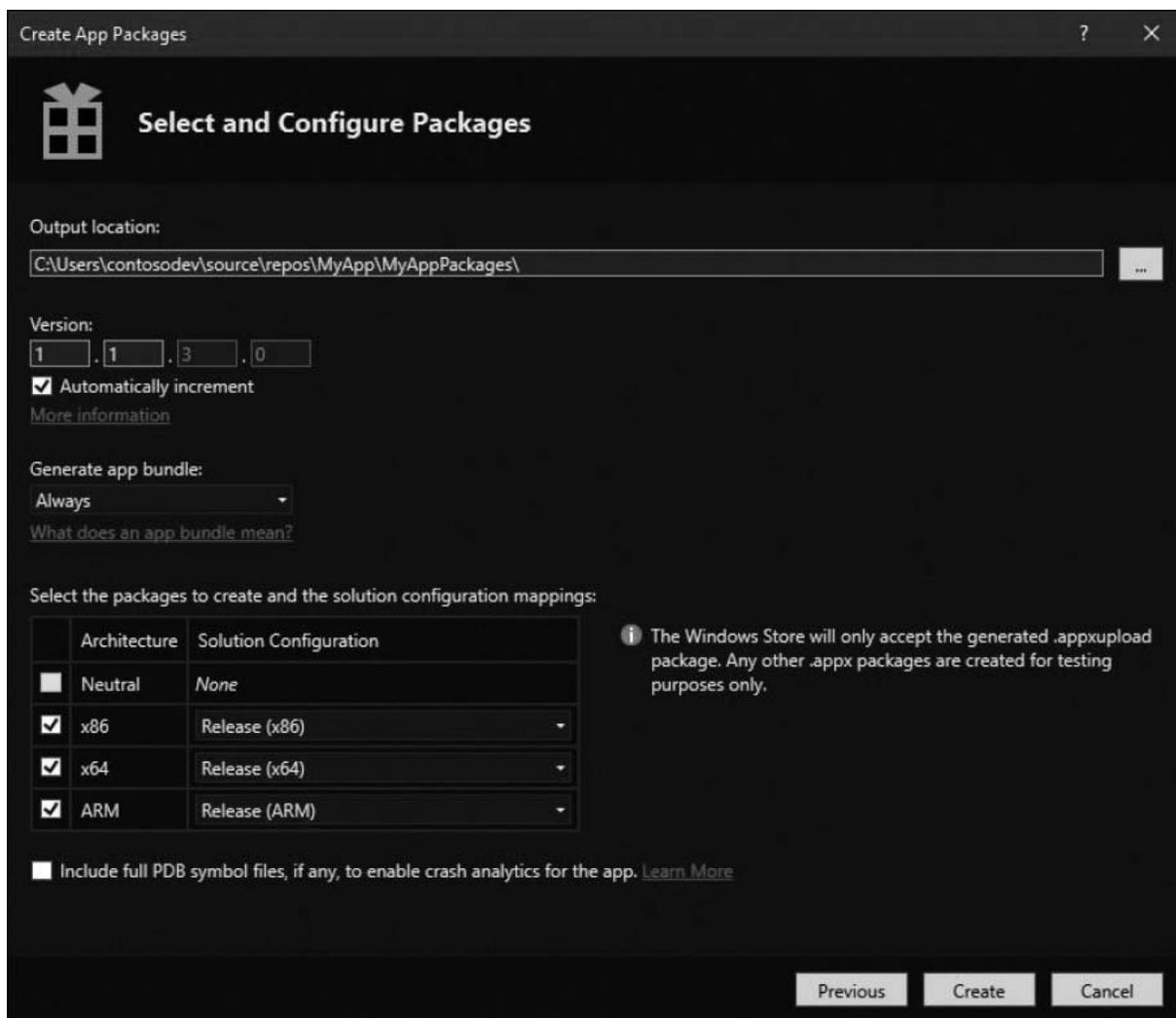


Figura 20.21 - Durante il processo di pubblicazione è possibile specificare la cartella in cui verranno creati i pacchetti, il numero di versione (anche auto-incrementante, comodo se non si usano sistemi automatici di build) e l'architettura.

Il risultato del processo, in ogni caso, consiste in una serie di file che per la distribuzione locale e tramite web sono già pronti all'uso. Per la distribuzione via web è necessario effettuare tre passaggi ulteriori: il primo, è la pubblicazione sul server del pacchetto dell'applicazione, il secondo è la configurazione a livello di webserver del MIME type corrispondente (per

esempio *application/appx* per i file *appx*), e il terzo e ultimo passaggio è quello di creare un URL per effettuare il download, come mostrato nell'[esempio 20.15](#).

Esempio 20.15

```
<html>
<head>
  <meta charset="utf-8" />
  <title>Pagina di installazione</title>
</head>
<body>
  <a href="ms-appinstaller:?source=myAppFile.appxbundle">Installa</a>
</body>
</html>
```

Per lanciare l'applicazione e installarla localmente sarà sufficiente effettuare un doppio click sul file *appx* (o sul relativo bundle) per vedere in esecuzione l'installer previsto da Windows 10 (in sostituzione di ClickOnce).

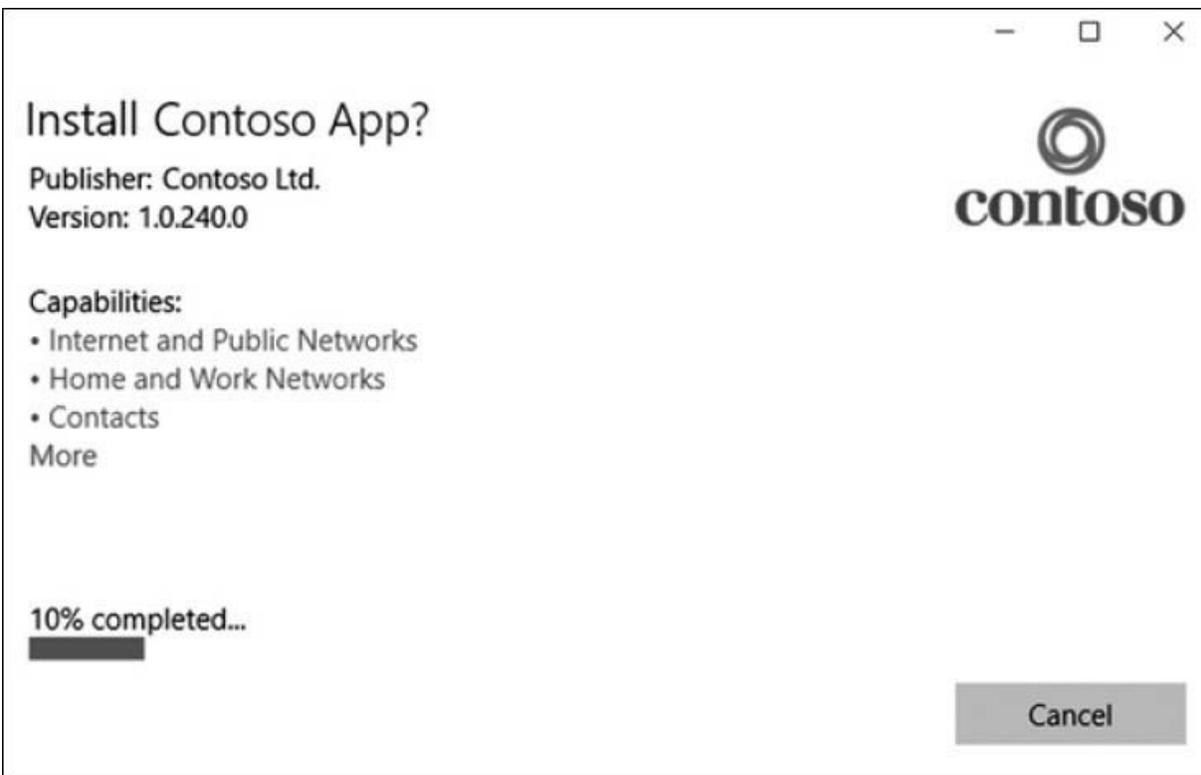


Figura 20.22 - L'installer di Windows 10 mostra informazioni relative al publisher, il numero di versione e le capability specifiche dell'applicazione che si sta cercando di installare.

Come mostrato all'interno della [Figura 20.22](#), all'interno dell'installer vengono elencate tutte le capabilities (ovvero i permessi richiesti all'utente per esecuzioni speciali, come l'accesso, per esempio, alla fotocamera) che sono stati precedentemente impostati nel file di manifest. L'obiettivo è quello di mettere l'utente sempre a conoscenza di cosa può effettuare l'applicazione. Al tempo stesso, gli sviluppatori devono progettare l'applicazione perché possa funzionare anche in quei casi in cui l'utente non dia esplicitamente il permesso per utilizzare determinate funzionalità, pena il crash dall'applicazione a runtime.

Conclusioni

In questo capitolo abbiamo parlato di come stanno evolvendo le esigenze sia dei team di sviluppo sia dei team di operation e di come stanno sempre di più convergendo verso il mondo del DevOps, in cui le competenze sono un insieme di software e infrastruttura, ma non solo: anche le tempistiche dei rilasci e la possibilità di avere più ambienti in cui verificare il funzionamento sono sempre di più punti critici nella crescita di un qualsiasi prodotto. Proprio per risolvere questi problemi abbiamo visto e affrontato più nel dettaglio i concetti legati agli ambienti, integrati nativamente in ASP.NET Core. Sempre in tema ambienti, ma sul lato infrastrutturale, abbiamo visto come sfruttare il web server Kestrel in configurazione di reverse proxy con NGINX e il classico IIS e, in particolare, abbiamo discusso di come questo porti innumerevoli vantaggi in termini di prestazioni, sicurezza e semplicità di gestione. Poiché a oggi i costi dell'hardware sono spesso un problema e considerando anche che la scalabilità è spesso imprevedibile, si è affrontato il tema cloud con Microsoft Azure e il servizio gestito degli App Service, che permettono di fare hosting e offrire scalabilità automatica alle applicazioni web .NET, e delle Azure Functions che permettono l'esecuzione on-demand di processi. Successivamente si è discusso di Docker, che va ad aumentare esponenzialmente questi concetti, portando anche grosse semplificazioni

nella fase di deployment, in cui i tempi sono decisamente più stretti, e infine abbiamo trattato della pubblicazione delle applicazioni desktop e in particolar modo della Universal Windows Platform.

Con questi argomenti, possiamo considerare il nostro viaggio alla scoperta di .NET al termine.

Vi raccomandiamo di non smettere mai di scrivere codice.

Buona programmazione!

Informazioni sul Libro

Aggiornata a .NET Framework 4.8, .NET Core 3 e Visual Studio 2019, questa guida completa a C# 8 è l'espressione corale di un gruppo di sviluppatori che, sin dalla sua prima versione, utilizza questo linguaggio per costruire applicazioni di ogni tipo, da quelle dedicate al web fino a complessi sistemi enterprise.

Il libro, che include le ultime novità introdotte dalle ultime versioni di .NET, tratta le basi del linguaggio fino ai concetti più avanzati, spiega l'uso dell'OOP in C#, per poi passare alle tecnologie più attuali come LINQ, Entity Framework, WPF, Windows 10, ASP.NET e servizi distribuiti.

È il testo ideale sia per chi inizia a programmare con C# sia per chi usa da tempo questo linguaggio e vuole scoprire tutte le novità di C# 8.

PUNTI DI FORZA

- I componenti di .NET Core e .NET Framework
- C# 8: dalle basi alle applicazioni avanzate
- OOP: la programmazione orientata agli oggetti
- Collection e Generics, Delegate ed Eventi
- Eseguire query nel codice con LINQ
- ADO.NET ed Entity Framework, Entity Framework Core
- XML e LINQ to XML
- Applicazioni ASP.NET con C#
- Applicazioni per Windows 10, XAML, WPF e WinForms

Indice

[Contenuti del libro](#)

[Gli autori](#)

[ASPIItalia.com](#)

[Ringraziamenti](#)

[Capitolo 1: Introduzione a .NET](#)

[Cos'è .NET](#)

[.NET = .NET Framework + .NET Core](#)

[.NET Standard](#)

[I componenti di .NET](#)

[Common Language Runtime \(CLR\)](#)

[Il concetto di codice managed](#)

[Common Type System](#)

[Common Language Specification](#)

La Cross-Language Interoperability

[Tipi di valore e tipi di riferimento](#)

[Conversioni tra tipi, boxing e unboxing](#)

[La gestione della memoria: il Garbage Collector](#)

Il concetto di Assembly

Interoperabilità tra .NET e COM

Analisi di .NET Core

Scegliere .NET Core o .NET Framework

.NET 5 e il futuro di .NET

Conclusioni

Capitolo 2: Visual Studio

L'IDE di Visual Studio

[Text Editor, designer e Intellisense](#)

[Toolbox](#)

[Property Editor](#)

[Altre aree dell'IDE](#)

Creare un progetto

[Il multi-targeting di .NET in Visual Studio](#)

[Il concetto di progetto e soluzione](#)

Gestire soluzione e progetto

[Aggiungere un progetto alla soluzione](#)

[Gestione delle referenze](#)

[Gestione di directory nella solution](#)

[Gestione del codice sorgente](#)

Compilare un progetto

[Gestire le configurazioni](#)

[Debug di un progetto](#)

[Usare il debugger](#)

[Breakpoint e watch](#)

[Intellitrace e historical debug](#)

[Refactoring](#)

[Novità di Visual Studio 2019](#)

[Supporto a editorConfig](#)

Conclusioni

Capitolo 3: Introduzione al linguaggio e sintassi di base

Introduzione al linguaggio

Uso dei commenti

Tipi di base

Namespace

Dichiarazione di variabili

Espressioni e operatori

Conversione tra tipi

Array

Enumerazioni

Funzioni e procedure

Istruzioni condizionali

Istruzione if

Istruzione switch

Operatore binario

Operatore null-conditional

Istruzioni iterative

Istruzione while

Istruzione do...while

Istruzione for

Istruzione foreach

Istruzioni di salto

Istruzione break

Istruzione continue

Istruzione return

Istruzione goto

Formattazione di stringhe

Conclusioni

Capitolo 4: Object Oriented Programming

Vantaggi dell'Object Oriented Programming

Principi fondamentali di OOP

Ereditarietà

[Polimorfismo](#)

[Incapsulamento](#)

Classi

[Membri di una classe](#)

[Livelli di accessibilità](#)

[Creazione delle istanze di classe](#)

Classi statiche e parziali

[Classi statiche](#)

[Partial class](#)

Ereditarietà e polimorfismo

Interfacce

Strutture

Overload degli operatori

Tuple

Regole di nomenclatura

Conclusioni

Capitolo 5: Collection e generics

Introduzione alle collection

[La classe ArrayList](#)

[Dizionari in .NET tramite la classe Hashtable](#)

[Le interfacce in System.Collections](#)

[Ulteriori tipologie di collection](#)

I Generics e la tipizzazione forte

[Le collezioni generiche](#)

[La lista nel mondo dei generics: List<T>](#)

[Le interfacce nelle collezioni generiche](#)

[Inizializzazione di una collezione](#)

[Un dizionario fortemente tipizzato: Dictionary<TKey, TValue>](#)

[Una collection con elementi univoci: HashSet<T>](#)

[Altre tipologie di collezioni generiche](#)

[Creazione di tipi generici](#)

[Impostare dei vincoli sul tipo generico](#)

[Un particolare tipo generico: Nullable<T>](#)

[Assegnazioni tra tipi generici: covarianza e controvarianza](#)

[Creazione di interfacce covarianti e controvarianti](#)

Index e Range

Conclusioni

Capitolo 6: Delegate ed eventi

I Delegate in .NET

Definizione e utilizzo di un delegate

Modello a oggetti dei delegate

Combinazione di delegate: la classe MulticastDelegate

Cenni sull'esecuzione asincrona di un delegate

I delegate e i generics

Anonymous method e lambda expression

I delegate come strumento di notifica: gli eventi

Definizione e uso di un evento in un oggetto

Creare eventi personalizzati

Scambiare dati tramite eventi: la classe EventArgs e le sue derivate

Definizione esplicita di eventi

Conclusioni

Capitolo 7: Approfondimenti al linguaggio

Gestione delle eccezioni

Gli errori prima del .NET Framework

Gestione strutturata degli errori tramite le exception

La classe System.Exception

Realizzare eccezioni personalizzate

Lavorare con le eccezioni nel codice

Intercettare le eccezioni

Il blocco finally

L'interfaccia IDisposable e il blocco using

Sollevare eccezioni nel codice e best practice

Gestione e rilancio delle eccezioni

Utilizzo delle InnerException

Considerazioni a livello di prestazioni sull'uso delle Exception

Esplorare i tipi a runtime con Reflection

La classe System.Type

Scrittura di codice dinamico

Realizzazione di codice dinamico con reflection

Codice dinamico tramite la parola chiave dynamic

[Le classi DynamicObject e ExpandableObject](#)

[Codice dichiarativo tramite gli attributi](#)

[Costruire e utilizzare attributi custom: la classe System.Attribute](#)

Il compilatore-as-a-service: Roslyn

[Installazione e primi passi con Roslyn](#)

[Analisi della sintassi](#)

Conclusioni

Capitolo 8: Async, multithreading e codice parallelo

Processi e thread

[La classe System.Threading.Thread](#)

[Passare parametri a un worker thread](#)

[Controllare il flusso di esecuzione di un thread](#)

[Il ThreadPool per applicazioni multithreading](#)

[Asynchronous programming model](#)

[Utilizzo del metodo EndInvoke](#)

[Sincronizzazione tramite IAsyncResult e polling](#)

[Utilizzo di un metodo di callback](#)

Esecuzione parallela con Parallel Extensions

[La Task Parallel Library](#)

[Composizione di task](#)

[Nested task e child task](#)

Programmazione asincrona con async e await

[Eseguire operazioni in parallelo con async e await](#)

[Realizzare metodi asincroni](#)

[Task per esecuzioni sincrone](#)

[Entrypoint asincroni](#)

Concorrenza e thread safety

[Sincronizzare l'accesso alle risorse](#)

[Collezioni con supporto alla concorrenza](#)

Stream asincroni con IAsyncEnumerable

Conclusioni

Capitolo 9: Introduzione a LINQ

1 perché di LINQ

Come funziona LINQ

Introduzione all'esempio del capitolo

Gli extension method di LINQ

La filosofia alla base LINQ

Anatomia di una query.

Gli operatori di restrizione

OfType

Gli operatori di proiezione

Select

SelectMany

Gli operatori di ordinamento

OrderBy, OrderByDescending, ThenBy e ThenByDescending

Reverse

Gli operatori di raggruppamento

Gli operatori di aggregazione

Average, Min, Max, Sum

Count, LongCount

Gli operatori di elemento

Gli operatori di partizionamento

Take e Skip

TakeWhile e SkipWhile

Operatori di insieme

Except

Intersect

Distinct

Union

La Query Syntax

Parallel LINQ

Conclusioni

Capitolo 10: Accedere ai dati con ADO.NET

Architettura di ADO.NET

Il namespace System.Data.Common

Data provider

Il namespace System.Data.SqlTypes

Il namespace System.Data

Lavorare con ADO.NET

Stabilire una connessione

Esecuzione di un comando

Scrivere dati in transazione

Lettura del risultato di una query.

Modalità disconnessa in ADO.NET

Ricerca dati in un DataSet

Conclusioni

Capitolo 11: Accedere ai dati con Entity Framework Core

Cosa è un O/RM

Mappare il modello a oggetti sul database

Disegnare le classi

Creare il contesto

Mapping tramite convenzioni

Mapping tramite API

Mapping tramite data annotation

Generare automaticamente le classi dal database

Istanziare il contesto

Recuperare i dati dal database

Ottimizzare il fetching

Lazy loading

Salvare i dati sul database

Persistere un nuovo oggetto

Persistere le modifiche a un oggetto

Cancellare un oggetto dal database

Manipolare gli oggetti nel change tracker

Funzionalità aggiuntive di Entity Framework

Limitazioni di Entity Framework 6.3 su .NET Core

Conclusioni

Capitolo 12: XML e LINQ to XML

Il supporto a XML con .NET

Gestire l'XML con la classe XmlDocument

Letture e scrittura rapida e leggera

Leggere con XmlReader

Scrivere con XmlWriter

LINQ to XML

Interrogare i nodi con LINQ

Manipolazione dei nodi

Interrogare rapidamente con XPathDocument

Navigare tra i nodi

Modificare i nodi

Trasformare i documenti con XSLT

Conclusioni

Capitolo 13: Introduzione a XAML

L'ambiente di sviluppo

Il markup XAML

La sintassi

La sintassi Object element

La sintassi Property Attribute

La sintassi Property Element

I namespace

Il layout system

Elementi fisici e logici

La disposizione degli elementi

I pannelli

I controlli

Le classi principali: UIElement e FrameworkElement

I controlli

La grafica

I pennelli: il Brush

Le trasformazioni sugli oggetti

Le animazioni

Conclusioni

Capitolo 14: Sviluppare con XAML: concetti avanzati

Definire e riutilizzare le risorse

Creare e gestire gli Style

Modellare il layout con i Template

Personalizzare un controllo con il ControlTemplate

Il data binding

Mostrare le informazioni con il data binding

Scenari master/detail con il data binding

Le fonti dati per il data binding

La formattazione dei dati

Le modalità di data binding

Gestire gli eventi

Conclusioni

Capitolo 15: Usare XAML

Universal Windows Platform

I tool per sviluppare

La prima app per il Windows Store

Applicazioni desktop con Windows Presentation Foundation

Creazione di un progetto

Gestire le finestre

Le XAML Island

Migrare applicazioni desktop a .NET Core 3

Aggiornamento dei progetti

Retargeting e migrazione dei file di progetto

Conclusioni

Capitolo 16: Applicazioni web con .NET

La prima pagina ASP.NET

Creare un progetto ASP.NET

Sviluppare con Web Forms

[Gli eventi, ilPostBack e il ViewState](#)

Interagire con la pagina

[Validazione delle form](#)

[Mantenere il layout con le master page](#)

Visualizzare dati: il data binding

[I list control](#)

[Utilizzare i template](#)

Creare URL per la SEO

Gestione delle aree protette

ASP.NET MVC e ASP.NET Core MVC

Creare form con ASP.NET MVC

Conclusioni

Capitolo 17: Sviluppare servizi web

I servizi RESTful con ASP.NET WebAPI

[La serializzazione e il model binding](#)

[Le action e i metodi HTTP](#)

[Le tipologie di risultato delle action](#)

[Lo scaffolding delle API](#)

Supportare il protocollo OData

Create servizi real-time attraverso SignalR

[L'hub come servizio bidirezionale](#)

[Utilizzare SignalR da JavaScript](#)

Conclusioni

Capitolo 18: La sicurezza nelle applicazioni .NET

Progettare applicazioni sicure

[Sicurezza in base all'ambiente e al contesto](#)

[*Applicazione intranet*](#)

[*Applicazione web con ASP.NET Identity*](#)

[*Applicazione con Single Sign-On proprietario*](#)

[*Applicazione con Single Sign-On di terze parti*](#)

[*Applicazione Mobile*](#)

Principi di crittografia

[Windows Data Protection](#)
[Crittografia simmetrica](#)
[Crittografia asimmetrica](#)
[Cifratura con algoritmo di hashing](#)

Firmare gli assembly

Validazione dei dati immessi dall'utente

[Proteggersi da attacchi di tipo SQL Injection](#)

Conclusioni

Capitolo 19: Gestione di file e networking

Gestione del file system

[Organizziamo le informazioni: Directory e File](#)

[Creazione di una directory](#)

[Eliminare una directory](#)

[Spostare una directory](#)

[Copiare una directory](#)

[Eeguire ricerche sul file system](#)

[Creare e modificare un file](#)

L'Isolated Storage di Windows

Il Registry di Windows

Principi di comunicazione di rete

[Architettura a livelli: il modello di trasporto](#)

[Porte e protocolli applicativi standard](#)

I protocolli TCP e UDP

[I socket e la comunicazione a basso livello](#)

[Inviare un semplice testo con un client UDP](#)

[Ricevere i messaggi con un mini server UDP](#)

[Inviare e ricevere dati con la classe TcpClient](#)

Il namespace System.NET

HttpClient: un'evoluta interfaccia HTTP per applicazioni moderne

Scambiare file con il protocollo FTP

Conclusioni

Capitolo 20: Configurare e pubblicare un'applicazione .NET

Modificare il comportamento secondo l'ambiente di destinazione

Pubblicazione e hosting

[Configurazione di NGINX](#)

[Configurazione di IIS](#)

[Pubblicazione con Microsoft Azure](#)

Pubblicazione con Docker

Le applicazioni Windows

Conclusioni

Informazioni sul Libro