Capitolo 1

Introduzione al linguaggio

C#, pronunciato come "see sharp", è un moderno linguaggio di programmazione, maturo, elegante, flessibile e *general-purpose*, che dalla sua prima implementazione, avvenuta nel lontano 2000, a oggi (release 6.0), è cresciuto in modo esponenziale sia quanto a caratteristiche offerte, sia quanto a popolarità e a numero di programmatori che lo utilizzano proficuamente per sviluppare qualsiasi tipo di applicazioni software come quelle per il Web, quelle per il desktop e dotate di avanzate GUI (Graphical User Interface), quelle per i dispositivi mobile, quelle per creare videogiochi e così via.
Per quanto riguarda il primo punto ne citiamo alcune:

- supporta in modo completo i blocchi costitutivi e il paradigma della *programmazione orientata agli oggetti* (incapsulamento, ereditarietà e polimorfismo);
- è *type-safe*, ossia è stato progettato in modo che sia, per esempio, impossibile leggere una variabile non inizializzata, accedere in lettura o in scrittura a un indice di un array che è oltre il suo limite e così via;
- consente di creare e usare *tipi* e *metodi generici*;
- permette di astrarre in un modo sicuro il concetto di puntatore a funzione attraverso i *tipi delegati* che rappresentano, in breve, riferimenti a metodi con un determinato tipo restituito ed elenco di parametri;
- supporta un approccio strutturato ed estensibile per il rilevamento degli errori software, grazie al potente meccanismo delle *eccezioni*;

In questo capitolo

- **Le origini di C#**
- **C# e il .NET Framework**
- **Cenni sull'architettura di un elaboratore**
- **Paradigmi di programmazione**
- **Concetti introduttivi allo sviluppo in C#**
- **Il primo programma in C#**
- **Compilazione ed esecuzione del codice**
- **Assembly: un cenno contestuale**

- ha un *sistema dei tipi unificato* ovvero tutti i tipi, anche quelli cosiddetti "primitivi" (`int`, `float` e così via) discendono dal tipo `object`, con ciò condividendo un insieme di operazioni in comune.

NOTA

Un'altra caratteristica da non sottovalutare del linguaggio C# riguarda il fatto che è stato standardizzato a livello internazionale da ECMA (European Computer Manufacturers Association) un'associazione fondata nel 1961 avente per scopo lo sviluppo di standard e *technical report* nel settore dell'information technology e delle telecomunicazioni, e da ISO (International Organization for Standardization) un'organizzazione indipendente non governativa, composta da membri provenienti da oltre centosessanta paesi, che produce standard internazionali per svariati settori (agricoltura, informatica, sanità e così via). Abbiamo, infatti, il documento di standard *ECMA-334 C# Language Specification* e il documento di standard *ISO/IEC 23270 Information technology – Programming languages – C#*. In sostanza questa standardizzazione rende possibile l'implementazione di compilatori, aderenti alle specifiche proprie della sintassi e della semantica del linguaggio C#, anche in altri sistemi diversi da Windows come, per esempio, GNU/Linux (ne è un esempio quello fornito attraverso il progetto Mono).

Per quanto riguarda, invece, il secondo punto, possiamo asserire che C# è, in definitiva, divenuto un importante e potente linguaggio di programmazione; lo dimostra anche il fatto che la sua conoscenza sta diventando una competenza molto richiesta nel mondo del lavoro e che il suo rating di popolarità è tra i primi posti, come dimostrato anche dal famoso indicatore di popolarità TIOBE (Figure 1.1 e 1.2).

Apr 2016	Apr 2015	Change	Programming Language	Ratings	Change
1	1		Java	20.846%	+4.80%
2	2		C	13.905%	-1.84%
3	3		C++	5.918%	-1.04%
4	5	^	C#	3.796%	-1.15%
5	8	^	Python	3.330%	+0.64%
6	7	^	PHP	2.994%	-0.02%
7	6	v	JavaScript	2.566%	-0.73%
8	12	^^	Perl	2.524%	+1.18%
9	18	^^	Ruby	2.345%	+1.28%
10	10		Visual Basic .NET	2.273%	+0.15%
11	11		Delphi/Object Pascal	2.214%	+0.75%
12	29	^^	Assembly language	2.193%	+1.54%
13	4	vv	Objective-C	1.711%	-4.18%
14	9	vv	Visual Basic	1.607%	-0.59%
15	24	^^	Swift	1.478%	+0.60%
16	14	v	MATLAB	1.344%	+0.08%
17	17		PL/SQL	1.314%	+0.20%
18	19	^	R	1.266%	+0.24%
19	43	^^	Groovy	1.262%	+0.97%
20	38	^^	D	1.030%	+0.63%

Figura 1.1 Tabella risultati TIOBE (rating aggiornato ad aprile 2016).

Programming Language	2016	2011	2006	2001	1996	1991	1986
Java	1	1	1	3	29	-	-
C	2	2	2	1	1	1	1
C++	3	3	3	2	2	2	7
C#	4	5	6	10	-	-	-
Python	5	6	7	26	17	-	-
PHP	6	4	4	18	-	-	-
Visual Basic .NET	7	189	-	-	-	-	-
JavaScript	8	9	9	8	32	-	-
Perl	9	8	5	4	3	-	-
Objective-C	10	7	43	-	-	-	-
Ada	25	18	15	21	7	7	3
Lisp	26	12	13	15	6	3	2

Figura 1.2 Tabella risultati TIOBE (scostamento di posizione rispetto a determinate annualità aggiornato ad aprile 2016).

NOTA

Il *TIOBE Programming Community index* è un indicatore che misura la popolarità di un linguaggio di programmazione *Touring completo*. Viene aggiornato mensilmente in base al numero dei risultati delle ricerche effettuate con venticinque motori di ricerca di una query avente il seguente pattern `+"<language> programming"`.

Le origini di C#

Nel lontano 1999 un ingegnere del software danese, Anders Hejlsberg, già dipendente Microsoft dal 1996, fondò, con Scott Wiltamuth, Peter Golde, Peter Sollich ed Eric Gunnerson, un team di sviluppo con l'obiettivo di progettare un nuovo linguaggio di programmazione, semplice, moderno, orientato agli oggetti e *type-safe*, che potesse essere da un lato il complemento perfetto per una piattaforma di sviluppo "unificata" che Microsoft stava nello stesso periodo sviluppando e che avremmo poi conosciuto con il nome di .NET Framework, e dall'altro una risposta all'emergente e innovativo linguaggio Java, dell'allora SUN (oggi Oracle), e alla sua piattaforma di sviluppo, caratterizzata, fondamentalmente, da una *virtual machine* e da un corposo set di API (*Application Programming Interface*).

Ricordiamo, infatti, che a quell'epoca, lo sviluppo in ambiente Windows avveniva principalmente utilizzando il C++ con le librerie MFC (*Microsoft Foundation Classes*) e ATL (*Active Template Library*) per la programmazione più *hardcore* e a basso livello, oppure il Visual Basic per la programmazione più semplice e rapida.

In più c'era COM (*Component Object Model*) un modello di programmazione *low-level* che permetteva di costruire librerie di codice riutilizzabili tra diversi linguaggi di programmazione, anche se il suo utilizzo era alquanto difficile a causa delle sua intrinseca e complessa infrastruttura.

NOTA

È giusto precisare che oggi ancora si possono sviluppare applicazioni Windows utilizzando le librerie MFC, il modello COM e così via. Si pensi, per esempio, che per la programmazione di videogiochi a baso livello in Windows si usano il C++ e le potenti librerie DirectX, che sono basate proprio su COM. Tuttavia, il modello di sviluppo predominante e che Microsoft sta spingendo senza sosta è certamente quello che fa uso delle tecnologie proprie del .NET Framework e dei linguaggi di programmazione a esso "collegati" (tipo, per l'appunto, C#).

Tre anni dopo, dunque, verso la fine del 2002, gli sforzi del team capitanato da Anders Hejlsberg produssero la versione 1.0 di questo nuovo linguaggio di programmazione che fu chiamato C# e che gettava le sue radici nella grande famiglia dei linguaggi *C-style* e che per i principali costrutti sintattici (per esempio, quelli che fanno uso delle istruzioni `if`, `for`, `while` e così via) e per la sua sintassi in generale (per esempio, quella che prevede l'uso delle parentesi graffe per delimitare blocchi di codice e il punto e virgola per delimitare la fine di un'istruzione) poteva essere subito compreso e proficuamente utilizzato da chi già utilizzava linguaggi come C, C++ o Java.
Dopo quella prima versione sono seguite, nell'ordine: la versione 1.2 nel 2003; la versione 2.0 nel 2005; la versione 3.0 nel 2007; la versione 4.0 nel 2010; la versione 5.0 nel 2012; la versione 6.0 nel 2015.

NOTA

Il *codename* scelto in origine per questo nuovo linguaggio di programmazione fu Cool (*C like Object Oriented Language*) e i primi file di codice avevano estensione `.cool`. Tuttavia, un apposito *naming committee*, decise che quel nome, così "comune", poteva dare problemi di trademark e decise così di ripensare a un nome più appropriato. Si narra che l'ispirazione del nuovo nome prese origine dal fatto che Cool voleva rappresentare un linguaggio che si poneva come un "incremento" rispetto al C++ il quale era stato, a sua volta, un "incremento" rispetto al C (da questo punto di vista, intatti, `++`, rappresenta il simbolo dell'operatore che incrementa una variabile di 1 unità). Chiaramente utilizzare qualcosa come C++++ non era affatto una buona idea (suonava decisamente male) e così si decise di usare, come suffisso di C, il simbolo *musicale* detto sharp (*diesis* in greco) che nella notazione musicale indica un'alterazione in senso crescente dell'intonazione della nota riferita (scritta alla sua sinistra l'aumenta di un semitono). Però, dato che il reale simbolo di diesis è ♯ (Unicode U+266F), non presente nelle tastiere standard, si decise di usare il simbolo cancelletto # (Unicode U+0023) invece presente (è tuttavia singolare notare come Microsoft, in contesti "grafici", tipo la pubblicità del logo del linguaggio, usi il reale simbolo musicale di diesis ♯). Ecco quindi che il nome completo scelto fu quello che noi oggi conosciamo, ossia C# (in modo alquanto curioso a taluni piace anche vedere il simbolo # come un'unione "in verticale" di due ++ con ciò ricordando l'originale idea di incremento ulteriore rispetto al C++ propria del C++++).

C# e il .NET Framework

Il linguaggio C#, ancorché potente, elegante e flessibile, rimarrebbe un mero strumento "didattico" se non fosse possibile utilizzarlo unitamente a una qualche API ossia a un insieme di *librerie di tipi* che mettono a disposizione funzionalità per l'interfacciamento

a una console, per l'accesso a basi di dati, per la progettazione di GUI (*Graphical User Interface*), per la programmazione concorrente, per la programmazione web, per l'utilizzo di file e via discorrendo.
A tal fine, per C# e per i sistemi Windows, l'API comunemente utilizzata è quella fornita da Microsoft attraverso il .NET Framework il quale è, in breve, un ambiente di esecuzione costituito da un *Common Language Runtime* (CLR) e da, per l'appunto, un'ampia e completa libreria di tipi denominata *.NET Framework Class Library* (FCL).

NOTA

.NET Framework ha iniziato il suo lungo cammino di sviluppo all'incirca nel 1997. Prima che ne uscisse la versione 1.0 nel 2002, assunse diversi altri *codename*, come Project Lightning, Project 42 e Next Generation Windows Services. Questa prima versione era installabile sui sistemi Windows 98, Windows NT 4.0, Windows 2000 e Windows XP; c'era il CLR 1.0 ed era inclusa in Visual Studio .NET. Seguirono: nel 2003 la versione 1.1 inclusa in Visual Studio .NET 2003 con il CLR 1.1; nel 2005 la versione 2.0 inclusa in Visual Studio 2005 con il CLR 2.0; nel 2006 la versione 3.0 inclusa in Expression Blend con il CLR 2.0; nel 2007 la versione 3.5 inclusa in Visual Studio 2008 con il CLR 2.0; nel 2010 la versione 4.0 inclusa in Visual Studio 2010 con il CLR 4; nel 2012 la versione 4.5 inclusa in Visual Studio 2012 con il CLR 4; nel 2013 la versione 4.5.1 inclusa in Visual Studio 2013 con il CLR 4.0; nel 2014 la versione 4.5.2 con il CLR 4; nel 2015 la versione 4.6 inclusa in Visual Studio 2015 con il CLR 4.

In linea generale, per un utente Windows, il .NET Framework è trasparente; infatti, normalmente è già installato. Talune volte, un'applicazione può presentare con il proprio installer la richiesta di installare una specifica versione del .NET Framework necessaria per il funzionamento. Ciò non comporta alcun problema, perché sullo stesso sistema Windows possono coesistere più versioni del .NET Framework.

Obiettivi del .NET Framework

Il .NET Framework è stato progettato e sviluppato da Microsoft al fine di raggiungere in, via principale, i seguenti obiettivi.

- *Garantire il supporto a differenti linguaggi di programmazione*. Le applicazioni .NET possono essere create con qualsiasi linguaggio di programmazione che supporti la compilazione nel cosiddetto *Intermediate Language* (IL) che possa essere eseguito nel *Common Language Runtime* (CLR). Oltre ai linguaggi forniti da Microsoft, come C#, Visual Basic, C++/CLI e F#, abbiamo infatti anche altri linguaggi forniti da terze parti, come IronPython, Silverfrost FTN95, Fortran for Windows, Fujitsu NetCOBOL for .NET, IronRuby, Scala.NET, PerlNET e così via.

TERMINOLOGIA

L'*Intermediate Language* (IL) è quel linguaggio intermedio, indipendente e agnostico rispetto a una particolare piattaforma, prodotto durante la fase di compilazione da un compilatore di un linguaggio .NET aware. Questo codice IL sta, in effetti, tra il codice sorgente di un programma ad alto livello e il codice nativo in linguaggio macchina di una determinata piattaforma hardware. Di fatto il codice IL non è mai codice direttamente eseguibile, ma ha bisogno di uno specifico processo di compilazione e traduzione nel

codice macchina della corrente piattaforma target, effettuato da un apposito *just-in-time compiler* messo a disposizione dall'ambiente di *runtime* (per esempio il CLR) presente nella piattaforma stessa. Possiamo dunque dire che, in definitiva, il codice IL è semanticamente paragonabile, ancorché strutturalmente diverso, al cosiddetto *bytecode* prodotto da un compilatore Java.

- *Permettere l'interoperabilità tra differenti linguaggi di programmazione*. In sostanza questo significa che è possibile scrivere componenti software con differenti linguaggi di programmazione (per esempio C# e F#) e poi unire questi componenti in un unico programma. Questa "indipendenza dal linguaggio" è resa possibile perché i compilatori dei linguaggi espressamente creati per l'ambiente .NET producono sempre lo stesso codice IL che è poi il codice che viene effettivamente processato dal corrente ambiente CLR.
- *Consentire un'indipendenza di piattaforma*. Se un sistema software diverso da Windows fornisce un'implementazione del .NET Framework, allora sarà possibile farvi girare senza modifiche il programma creato in origine per Windows. In definitiva quest'indipendenza da una specifica piattaforma software permette di scrivere i programmi una sola volta, con la certezza che sarà poi possibile eseguirli dappertutto senza alcuna modifica sulla piattaforma software/hardware di destinazione. Per descrivere questa possibilità, talune volte viene usato l'acronimo WORA, *Write Once Run Anywhere*. Nel mondo open source, per esempio, è molto famosa e apprezzata un'implementazione del .NET Framework di Microsoft, definita Mono, che gira, tra gli altri, sui sistemi OS X e GNU/Linux.
- *Fornire a tutti i linguaggi .NET aware un ricco set di API*. Ciò si concretizza mediante la possibilità di utilizzare i tipi di un'estesa libreria di classi che espongono le funzionalità necessarie per l'accesso ai database e ai file, per l'uso dei *thread* e per il networking, per la creazione di sofisticate GUI, per lo sviluppo di moderne applicazioni web, per la manipolazione di dati in XML e così via.
- *Semplificare i problemi di versioning e deployment delle applicazioni*. Grazie agli *assembly* che sono, in breve, l'unità fondamentale di deployment di un'applicazione .NET (per esempio un file `.exe` o un file `.dll`), è infatti possibile porre rimedio al cosiddetto *DLL hell*, ossia a ciò che può capitare quando un'applicazione "A" installa per il proprio funzionamento una versione di una libreria a collegamento dinamico (*Dynamic Link Library*), diciamo la versione 1.0, la quale viene poi aggiornata da una versione 2.0 installata dall'applicazione "B", la cui versione non è però compatibile con la versione 1.0. A questo punto un successivo avvio dell'applicazione A non avverrà con successo a causa del predetto aggiornamento.

CLR (Common Language Runtime)

Il *Common Language Runtime*, in breve CLR, è quella parte fondamentale del .NET Framework che esegue e gestisce le applicazioni create per l'ambiente .NET.
Per dirla in modo più preciso, si tratta del *runtime environment* (ambiente di *runtime*) del .NET Framework, che localizza, carica in memoria, avvia e gestisce la relativa applicazione .NET; in più fornisce ulteriori e fondamentali servizi come la gestione automatica della memoria e il *garbage collection*, la gestione e il coordinamento dei *thread*, la verifica avanzata

della sicurezza del codice e la compilazione e conversione dell'*Intermediate Language* (IL) nel codice nativo (*machine language code*) della piattaforma di esecuzione.

NOTA

Le funzionalità del CLR sono implementate nella libreria `clr.dll` (`mscorwks.dll` per le versioni del .NET Framework precedenti la 4). In più è presente anche una piccola libreria denominata `mscoree.dll` (*Microsoft Common Object Runtime Execution Engine*), la quale, quando viene avviato un programma .NET, esamina il suo file eseguibile al fine di determinare quale versione del CLR utilizzare. Questa informazione viene poi passata alla funzione `CLRCreateInstance` (implementata per l'appunto in `mscoree.dll`), la quale può restituire un'interfaccia come `ICLRMetaHost`; questa, grazie alla funzione `GetRuntime`, crea l'ambiente CLR relativo, che viene, infine, caricato in memoria e si rende disponibile per compiere tutte le operazioni a esso pertinenti e poc'anzi già indicate.

TERMINOLOGIA

Il codice IL prodotto da un compilatore di un linguaggio .NET aware è detto anche *managed code* (codice gestito) perché può essere eseguito e gestito solo nell'alveo della sandbox propria dell'ambiente di *runtime* di .NET (il CLR). Di converso, il codice nativo prodotto, per esempio, da un compilatore C o C++ è detto *unmanaged code* (codice non gestito), perché non vi è alcun ambiente di *runtime* deputato a gestirlo e dunque a offrirgli una robusta infrastruttura di sicurezza e una serie di servizi fondamentali quali, per esempio, quello di gestione automatica della memoria. Quanto detto si può concretizzare facendo il seguente esempio: in C++, che produce codice *unmanaged*, dobbiamo ricordarci di deallocare manualmente dalla memoria ogni oggetto creato, pena la possibilità di incorrere in pericolosi e gravi *memory leak*; in C#, di converso, che produce codice *managed*, non dobbiamo ricordare di deallocare ogni oggetto creato, perché vi provvederà in automatico il sistema di *runtime* di .NET, tramite un sofisticato componente software denominato *garbage collector*. Con ciò si evita la possibilità di incorrere in memory leak e ciò permette di migliorare la robustezza del codice prodotto.

CTS (Common Type System)

Il *Common Type System* (CTS) è una specifica formale che definisce le regole che un ambiente di *runtime*, tipo il CLR, deve seguire per dichiarare, utilizzare e gestire i tipi di dati. In sostanza, la specifica CTS descrive un ricco sistema di tipi unificato che garantisce le seguenti funzionalità.

- Un'integrazione tra linguaggi (*cross-language integration* o *language interoperability*), ossia la possibilità, per esempio, da un linguaggio di programmazione, tipo C# o qualsiasi altro linguaggio .NET, di usare i tipi definiti in un altro linguaggio di programmazione, tipo C++/CLI o qualsiasi altro linguaggio di programmazione .NET.
- Un importante grado di sicurezza rispetto alle operazioni che possono essere compiute con un determinato tipo (*type safety*). Per esempio, dal punto di vista del CLR, effettuare operazioni di somma o sottrazione su tipi interi sarà senza dubbio lecito, mentre le stesse operazioni non saranno lecite se eseguite su tipi booleani.
- L'esecuzione del codice con prestazioni elevate.

- La definizione di un modello orientato agli oggetti, così come è comunemente implementato in molti linguaggi di programmazione;
- La definizione di un set di tipi primitivi, per esempio `bool`, `char`, `string`, `int16`, `int32` e così via, direttamente supportati da un ambiente di *runtime*, tipo il CLR, i quali sono correntemente utilizzati con un linguaggio di programmazione durante lo sviluppo di un'applicazione.

IMPORTANTE

Le regole del CTS sono dettagliate nella "Partizione I Clausola 8 - *Common Type System*" presente sia nella specifica ECMA, denominata *Standard ECMA-335 Common Language Infrastructure (CLI)*, sia nella specifica ISO denominata ISO/IEC 23271 Information technology – Common Language Infrastructure (CLI).

Più in concreto, questa specifica fornisce indicazioni e dettagli sui seguenti argomenti: tipi e valori (sui tipi riferimento, sulle classi, sulle interfacce e così via); locazioni di memoria (assegnamenti, conversioni e così via); membri dei tipi (campi, proprietà, metodi, eventi e così via); denominazione delle entità (validità di un nome, visibilità di un tipo, accessibilità di un membro di un tipo e così via); contratti e firme (contratto di classe, contratto di interfaccia, firme dei tipi, firme dei metodi e così via); assegnamenti (compatibilità e liceità di assegnamenti tra tipi e locazioni); sicurezza rispetto ai tipi e processo di verifica; definitori dei tipi (array, delegati, classi, interfacce, ereditarietà dei tipi e così via); ereditarietà dei membri dei tipi (dei campi, dei metodi, delle proprietà e così via); definizioni dei membri dei tipi (dei campi, dei metodi, delle proprietà e così via).

NOTA

Parlando di CTS, nonostante abbiamo dato molti ragguagli "teorici" su cosa rappresenti, manca ancora una spiegazione sul significato della parola *common* nel suo acronimo. Il sistema dei tipi descritto dalla specifica è *comune* a tutti i linguaggi di programmazione indirizzati a un ambiente di *runtime* compatibile (tipo il CLR) ovvero è un sistema che a basso livello "esprimerà" sempre, per tutti quei linguaggi, gli stessi tipi. Questo significa che, in pratica, è possibile usare, per esempio, un tipo intero in un linguaggio come C# con la sua sintassi e *keyword* richiesta (`int nr = 10;`) e poi usare sempre un tipo intero in un linguaggio come Visual Basic con la sua sintassi e *keyword* (`Dim nr As Integer = 10;`) ed essere certi che dal punto di vista del CTS entrambi rappresenteranno un tipo intero a 32 bit denominato in modo comune e indipendente come `System.Int32`. Detto in altri termini, per il CTS quello che conta è il codice IL emesso, che per entrambi i compilatori (C# e Visual Basic), sarà sempre lo stesso, perché esprimerà un intero a 32 bit di *tipo* `int32`).

CLS (Common Language Specification)

La *Common Language Specification*, in breve CLS, è una specifica formale che definisce un set minimale e comune di regole e funzionalità, sottoinsieme di quello espresso dalla specifica CTS, che i seguenti "attori" devono rispettare e offrire al fine di garantire una reale interazione tra *oggetti* scritti usando linguaggi di programmazione .NET differenti.

- *Consumer*. Rappresenta un linguaggio di programmazione o tool in grado di accedere a tutte le caratteristiche offerte da un *framework* CLS-compliant. Per esempio, un

consumer deve essere in grado di invocare un metodo o delegato, creare un'istanza di un qualsiasi tipo CLS-compliant, accedere ai tipi annidati, istanziare e usare tipi e metodi generici e così via.

- *Extender*. Rappresenta un linguaggio di programmazione o tool che, oltre a supportare le capacità offerte da un consumer, è anche in grado di estendere un *framework* CLS-compliant. Per esempio, un extender deve essere in grado di definire nuovi tipi CLS-compliant, implementare qualsiasi interfaccia CLS-compliant, definire tipi e metodi generici e così via.
- *Framework*. Rappresenta una libreria di codice CLS-compliant sviluppata per essere utilizzata da linguaggi di programmazione e tool che possono essere sia consumer sia extender.

IMPORTANTE

I concetti propedeutici della CLS sono dati nella "Partizione I Clausola 7 - *Common Language Specification*" presente sia nella specifica ECMA, denominata *Standard ECMA-335 Common Language Infrastructure (CLI)*, che nella specifica ISO denominata ISO/IEC 23271 Information technology – Common Language Infrastructure (CLI). Tutte le 48 regole, invece, sono presenti, laddove di interesse, sempre nella Partizione I, ma sono snodate attraverso le Clausole, dalla 7 alla 11, dove quest'ultima ne dà una panoramica d'insieme.

Per comprendere appieno l'importanza delle regole CLS è necessario ricordare che ogni linguaggio di programmazione è in sostanza diverso l'uno dall'altro: ciascuno ha i propri costrutti sintattici e le proprie caratteristiche e funzionalità. Alcuni non fanno differenza se i nomi delle entità definibili sono *case-sensitive*, altri non offrono il supporto ai tipi interi senza segno o all'*overloading* degli operatori e così via.

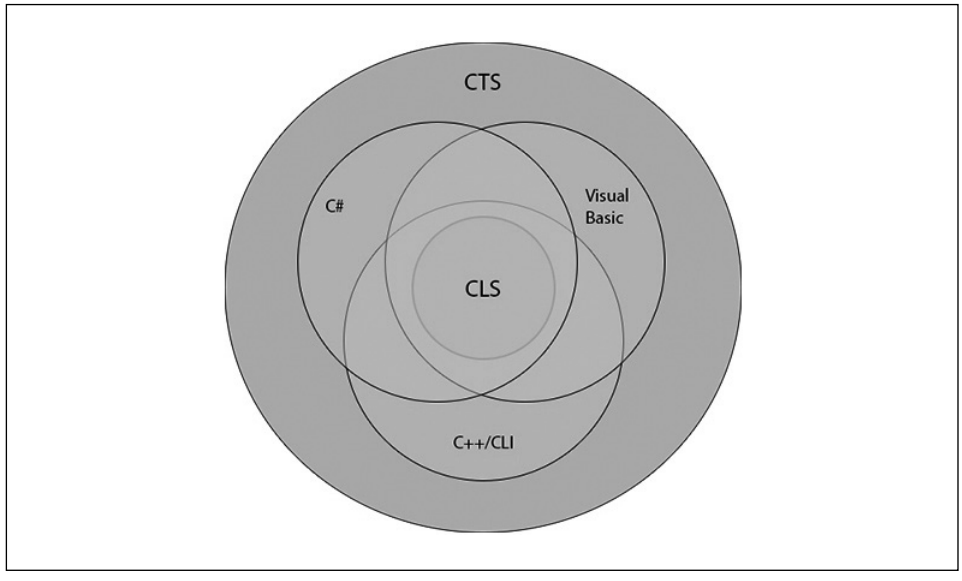

Figura 1.3 C#, Visual Basic e C++/CLI offrono un sottoinsieme delle funzionalità di un CTS e un soprainsieme delle funzionalità di una CLS (non necessariamente lo stesso). Il cerchio concentrico di intersezione rappresenta il sottoinsieme CLS di funzionalità comuni.

Ciò detto, se vogliamo scrivere del codice, cioè tipi e funzionalità che siano utilizzabili da qualsiasi altro linguaggio di programmazione .NET compatible, dobbiamo impiegare solamente quelle funzionalità offerte dal linguaggio di programmazione in uso che sono "comuni" anche agli altri linguaggi di programmazione .NET compatible e che sono, per l'appunto, evidenziate e regolamentate dalla specifica CLS.

TERMINOLOGIA

I linguaggi di programmazione che rispettano le regole della CLS sono definiti come *CLS Compliant Languages*.

Data la Figura 1.3 possiamo asserire che se definiamo un tipo in C#,Visual Basic o C++/CLI (ma è lo stesso per qualsiasi altro linguaggio di programmazione .NET compatible, tipo IronPython, PerlNET e così via) tale tipo potrà da questi essere usato in modo interoperabile (*cross language interoperability*) solo se è implementato con le funzionalità comuni della CLS. Infatti, come mostra la figura, tutto quello che è al di fuori del "cerchio" della CLS rappresenta una funzionalità non comune ma che può essere propria di C#,Visual Basic o C++/CLI.
Mostriamo di seguito alcune regole.

- *CLS Rule 1: CLS rules apply only to those parts of a type that are accessible or visible outside of the defining assembly*. Questa fondamentale regola stabilisce che le regole proprie della CLS si applicano soltanto se un tipo è pubblicamente accessibile, così come i suoi metodi e i suoi campi (la loro definizione deve rispettarle in pieno). Di converso, quindi, le regole della CLS non si applicano alla logica interna utilizzata per la costruzione e definizione di un tipo o metodo. Per esempio, se definiamo un tipo pubblicamente visibile con un metodo altresì pubblico che restituisce un valore intero senza segno allora lo stesso sarà non *CLS compliant* (alcuni linguaggi di programmazione non sono in grado di gestire tipi interi senza segno). Se però nello stesso metodo (nel suo corpo di definizione) utilizziamo un tipo intero senza segno questo non sarà di interesse per la CLS e dunque sarà lecitamente impiegabile;
- *CLS Rule 4: ... Identifiers shall be in the canonical format defined by Unicode Normalization Form C. For CLS purposes, two identifiers are the same if their lowercase mappings (as specified by the Unicode locale-insensitive, one-to-one lowercase mappings) are the same. That is, for two identifiers to be considered different under the CLS they shall differ in more than simply their case...* Questa regola stabilisce che non è *CLS compliant* definire tipi, metodi o campi che hanno lo stesso nome ma differiscono solo in considerazione delle lettere minuscole/maiuscole e ciò perché non tutti i linguaggi di programmazione considerano tale distinzione. Infatti, mentre, per esempio, per un linguaggio come C# è lecito definire due metodi denominati `multiply` e `Multiply`, la stessa definizione non è lecita per un linguaggio come Visual Basic, in quanto il suo compilatore tratta gli identificatori senza distinguere tra maiuscole e minuscole (è *case insensitive*).
- *CLS Rule 17: Unmanaged pointer types are not CLS-compliant*. Questa regola stabilisce che non è *CLS compliant* fornire una sintassi che consenta di definire o accedere a *tipi puntatore*, che sono tipi "particolari", molto comuni in alcuni linguaggi di programmazione come il C e il C++.

TERMINOLOGIA

Un puntatore è definibile come un tipo derivabile da un tipo funzione oppure da un qualsiasi tipo oggetto (per esempio una variabile) denominato tipo referenziato. In pratica rappresenta un oggetto contenente un valore che è un riferimento, un puntamento, verso un altro oggetto che è, per l'appunto, il tipo referenziato. Dunque un puntatore derivato da un tipo referenziato `T` è definibile come un puntatore a `T`. Se abbiamo, per esempio, una variabile di tipo intero contenente il valore `10`, possiamo definire un puntatore verso di essa, ossia possiamo costruire un tipo derivato (*un puntatore a un intero*) in grado di contenere l'indirizzo di memoria nel quale è localizzabile tale variabile e, tramite tale puntatore, compiere sulla variabile delle operazioni di lettura e/o di scrittura.

- *CLS Rule 23: System.Object is CLS-compliant. Any other CLS-compliant class shall inherit from a CLS-compliant class*. Questa regola stabilisce che, con la sola eccezione del tipo `System.Object` (che non eredita da alcun altro tipo), tutti gli altri tipi devono ereditare esattamente da un solo tipo. In pratica un linguaggio di programmazione è *CLS compliant* se non offre un costrutto che consenta di implementare la cosiddetta *ereditarietà multipla*, ossia la possibilità da parte di un tipo di ereditare da due o più tipi (come vedremo poi, l'ereditarietà multipla è consentita solo utilizzando il *tipo interfaccia*).

CLI (Common Language Infrastructure)

La *Common Language Infrastructure* (CLI) è una specifica formale che descrive come sviluppare una complessa e unificata infrastruttura software che permetta alle applicazioni, scritte in un qualsiasi linguaggio di programmazione di alto livello compatibile, di poter essere eseguite in sistemi differenti senza dover essere riscritte e ricompilate per uno di quei sistemi in particolare.

IMPORTANTE

CLI è formalizzata nella specifica ECMA, denominata *Standard ECMA-335 Common Language Infrastructure (CLI)* e nella specifica ISO denominata ISO/IEC 23271 Information technology – Common Language Infrastructure (CLI).

In sostanza questa specifica è di notevole importanza, perché permette a sviluppatori di terze parti di fornire una propria implementazione del .NET Framework di Microsoft che sia capace di operare su sistemi diversi da Windows (per esempio GNU/Linux o OS X) e su processori diversi dall'x86 (per esempio PPC, ARM, Sparc, s390, Alpha e così via). La specifica CLI è suddivisa nelle seguenti parti.

- *Partition I: Concepts and Architecture*. Fornisce una descrizione di massima di una CLI; introduce alle regole della CLS (*Common Language Specification*); descrive e definisce le regole del CTS (*Common Type System*); introduce al meccanismo dei *metadati*, che sono, in breve, dati utilizzati per descrivere e referenziare i tipi definiti dal CTS; descrive un generico ambiente di esecuzione (*runtime engine*), definito come *Virtual Execution System* (VES), per il codice *managed* prodotto (le istruzioni espresse in linguaggio CIL).

TERMINOLOGIA

CIL sta per *Common Intermediate Language* ed è il termine ufficiale utilizzato dalla specifica CLI per indicare l'*Intermediate Language* (IL). In letteratura è molto utilizzato anche il termine MSIL (*Microsoft Intermediate Language*), che identifica anch'esso l'*Intermediate Language* (IL). In pratica, CIL, IL e MSIL hanno tutti lo stesso significato.

- *Partition II: Metadata Definition and Semantics*. Fornisce una descrizione dettagliata, sia semantica che strutturale, dei metadati. Per quanto riguarda la semantica dei metadati, essa è descritta utilizzando la sintassi di un apposito linguaggio assembly definito come *CIL Assembly Language* (ILAsm). Per quanto concerne l'aspetto strutturale dei metadati, essa fornisce indicazioni sul loro formato logico (quali strutture di dati sono utilizzate per memorizzare le informazioni che descrivono) e sul loro layout fisico (come e dove sono memorizzate quelle strutture logiche nell'ambito del relativo file eseguibile).

ATTENZIONE

È importante non confondere il termine ILAsm, che descrive il codice assembly proprio del CIL, dall'assemblatore vero e proprio, che è denominato invece ilasm (*IL assembler*).

- *Partition III: CIL Instruction Set*. Fornisce una descrizione completa del set di istruzioni proprio del *Common Intermediate Language* (CIL).
- *Partition IV: Profiles and Libraries.* Fornisce una descrizione delle librerie standard e dei profili standard che (come minimo) un'implementazione CLI deve fornire. Per questa specifica una libreria rappresenta un insieme di tipi deputati a fornire determinate funzionalità di base a un ambiente CLI e che sono contenuti negli assembly `mscorlib`, `System` e `System.Xml`. Le librerie standard previste sono le seguenti: *Runtime Infrastructure Library* (fornisce tipi e funzionalità per un compilatore per un ambiente CLI); *Base Class Library* o *BCL* (fornisce tipi e funzionalità essenziali per un ambiente CLI come quelle per la manipolazione delle stringhe, per l'accesso ai file, per le collezioni e così via); *Network Library* (fornisce tipi e funzionalità per implementare servizi di networking); *Reflection Library* (fornisce tipi e funzionalità che consentono di utilizzare la *reflection*, un meccanismo attraverso il quale è possibile "ispezionare" e "manipolare" a *runtime* gli elementi di un programma); *XML Library*, (fornisce tipi e funzionalità per il parsing di documenti XML); *Extended Numerics Library* (fornisce tipi e funzionalità per l'aritmetica in virgola mobile in precisione singola, doppia ed estesa); *Extended Array Library* (fornisce tipi e funzionalità per la manipolazione di array multidimensionali e *non-zero based* ossia di array il cui indice iniziale non comincia dalla posizione 0); *Vararg Library* (fornisce tipi e funzionalità che garantiscono il supporto per l'utilizzo degli *argomenti di lunghezza variabile*); *Parallel Library* (fornisce tipi e funzionalità che consentono di effettuare, in modo semplificato, dei loop "paralleli" su una collezione di valori utilizzando più *thread* di esecuzione). Un profilo, invece, è un insieme di librerie raggruppate e deputate a fornire un ben preciso livello di funzionalità. I profili standard previsti sono i seguenti: *Kernel Profile* (contiene i tipi delle librerie Base Class Library e Runtime Infrastructure Library); *Compact Profile*, (contiene i tipi delle librerie Base Class Library, Runtime Infrastructure Library, XML Library, Network Library e Reflection Library). Infine

è importante dire che un'implementazione CLI è definita *conforme* solo se prevede almeno il Kernel Profile e dunque le relative librerie.

NOTA

La Framework Class Library (FCL) fornita dal .NET Framework di Microsoft è un superset delle librerie standard previste da CLI. Essa, infatti, prevede tipi e funzionalità supplementari per lo sviluppo web (ASP.NET), per l'accesso ai database (ADO.NET), per la progettazione di GUI (Windows Forms e WPF) e così via.

- *Partition V: Debug Interchange Format*. Fornisce una descrizione dettagliata sulla struttura dei file portabili CILDB per lo scambio di informazioni di debugging tra un *producer CLI* (per esempio una libreria di tipi) e un *consumer CLI* (per esempio il linguaggio che usa i tipi di una libreria).
- *Partition VI: Annexes.* Fornisce una serie di allegati supplementari alla specifica come, per esempio: l'*Annex C*, che fornisce ragguagli sull'implementazione di un assembler CIL (ilasm); l'*Annex D*, che fornisce informazioni sulle linee guida di progettazione per lo sviluppo di librerie di classi.

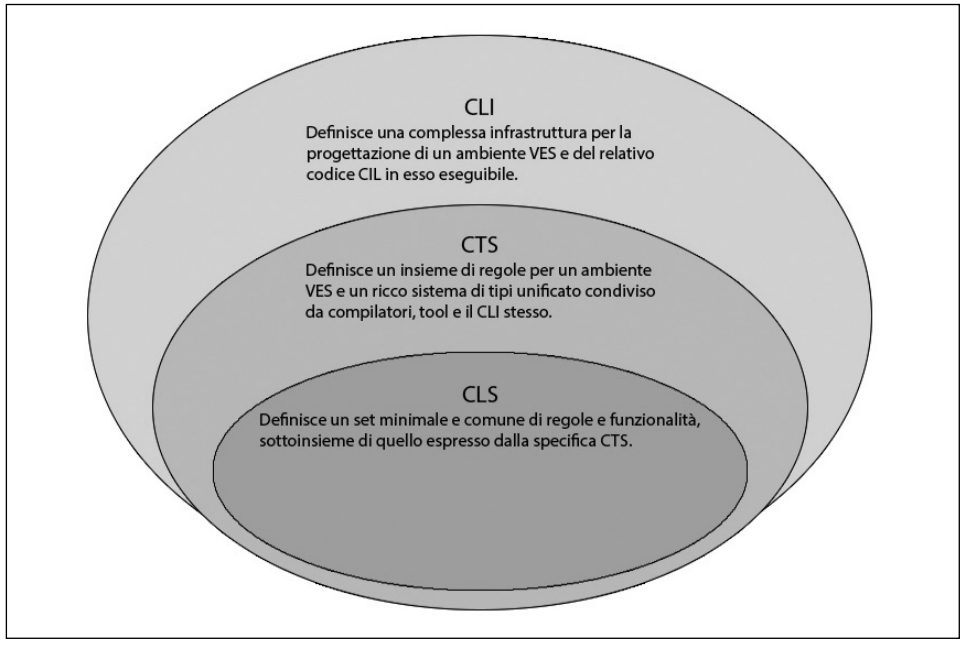

Figura 1.4 Relazione tra CLI, CTS e CLS.

TERMINOLOGIA

La Figura 1.5 evidenzia come sia il .NET Framework che Mono rappresentino delle implementazioni concrete, rispettivamente di Microsoft e Xamarin (prima Novell), di una generica infrastruttura CLI. Al contempo, il .NET CLR e il Mono Runtime, sono le implementazioni pratiche, di Microsoft e di Xamarin, di un generico ambiente VES.

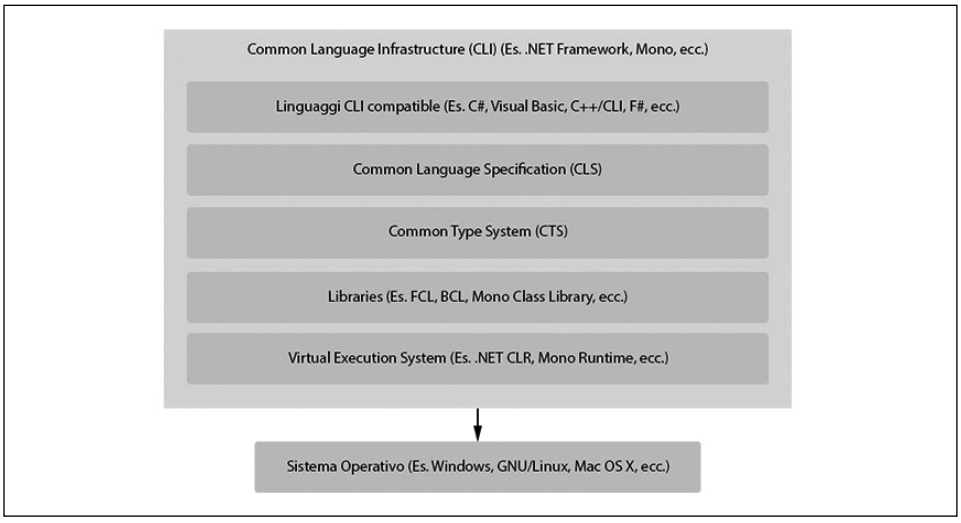

Figura 1.5 Architettura generale di una CLI.

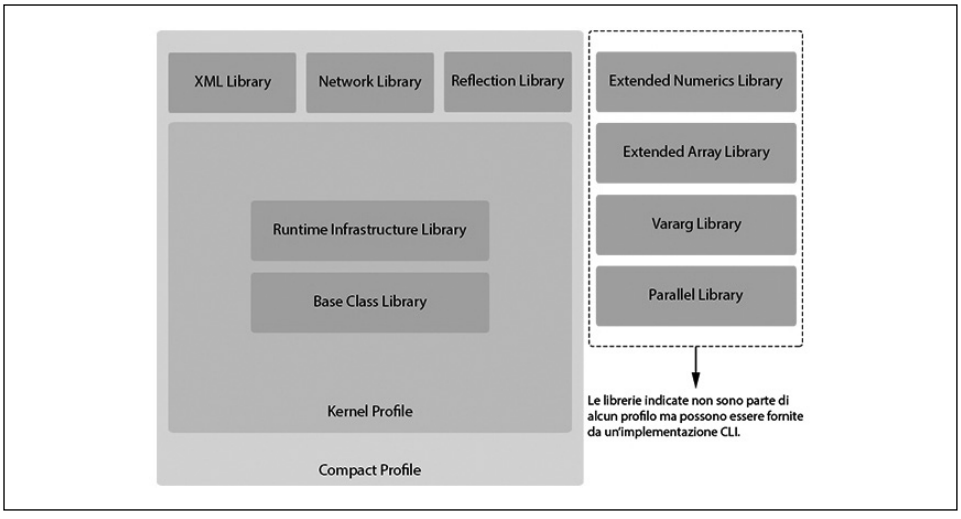

Figura 1.6 Relazione tra profili e librerie.

.NET e il futuro

Microsoft, il 12 novembre 2014, durante l'evento *Connect()*, ha annunciato un cambiamento, che potremmo dire epocale, per quanto riguarda il .NET Framework, concretizzabile riportando la frase di Immo Landwerth, program manager del .NET team: "... *is a huge day for .NET! We're happy to announce that .NET core will be open source, including the runtime as well as the framework libraries...*".

In sostanza questo significa che Microsoft ha intenzione di spingere l'ecosistema .NET, almeno la parte core e quella delle librerie di base, verso una integrazione realmente multi-piattaforma e che prescinda da implementazioni di terze parti (per esempio, Mono).

Infatti, la volontà di rendere open source i pezzi fondamentali del .NET Framework non potrà che favorirne l'adozione su una totalità di piattaforme diverse da Windows (tipo GNU/Linux o OS X) che condivideranno tutte una base di codice comune direttamente "compatibile" con quella ufficiale di Microsoft per il suo sistema operativo. E il fatto di avere una base di codice, comune, open source e compatibile su cui lavorare, con una larga ed estesa comunità di sviluppatori, non potrà che apportare benefici di una maggiore affidabilità e qualità implementativa della piattaforma stessa.

NOTA

Appare evidente che l'apertura di Microsoft al mondo open source del suo cavallo di battaglia .NET sia stata fatta anche per ragioni di diretta e seria competizione con la piattaforma Java. Infatti, Oracle, avrà ora di sicuro un valido competitor multi-piattaforma di cui preoccuparsi e dovrà sicuramente iniziare a porre in atto strategie di miglioramento e avanzamento tecnologico della sua piattaforma Java per stare al passo con la piattaforma .NET. Questa competizione tra due dei massimi "colossi" del mondo IT non potrà che essere un vantaggio per il mondo dello sviluppo software nel complesso, in quanto porterà di sicuro a nuove ed entusiasmanti innovazioni.

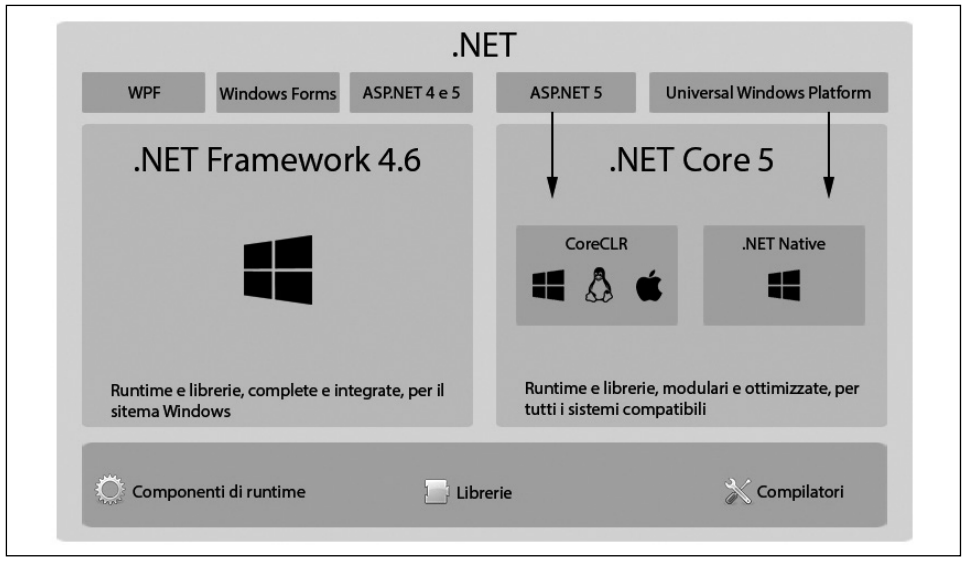

Figura 1.7 Architettura della nuova piattaforma .NET (aggiornata agli inizi del 2016).

La visione architetturale della *nuova* piattaforma .NET è quella presentata dalla Figura 1.7 dove possiamo notare come, in linea generale, .NET è formalmente suddiviso in due macro-aree separate, ciascuna delle quali ha un layer superiore a essa pertinente. Abbiamo, poi, un layer inferiore, comune alle due aree.
La macro area a sinistra è di competenza del .NET Framework, giunto alla versione 4.6, ed è il *framework* di Microsoft *non cross-platform* ed eseguibile solo su sistemi Windows. Il suo layer superiore comprende le fondamentali tecnologie: WPF (Windows Presentation Foundation), *framework* per la progettazione di GUI moderne e sofisticate; Windows

Forms, *framework* utilizzato per la progettazione di GUI e presente già dalla versione 1.0 del .NET Framework, basato, però, su modalità di utilizzo e implementative piuttosto differenti rispetto a WPF; ASP.NET, *framework* per lo sviluppo di applicazioni web.
La macro area a destra è invece di competenza di una nuova tecnologia Microsoft, in via di sviluppo, definita come .NET Core. Si tratta di un'implementazione *cross-platform* (Windows, GNU/Linux, OS X, FreeBSD), multi-architettura (x64, x86, ARM32, ARM64) e open source (il codice sorgente, distribuito in accordo con la licenza MIT License, è disponibile su GitHub) del .NET Framework di Microsoft. Essa è costituita da un ambiente di *runtime*, CoreCLR, e da una libreria di base, CoreFX.

NOTA

GitHub è una piattaforma web che offre come servizio la possibilità di ospitare, principalmente, dei repository di file di codice sorgente che utilizzano Git come *sistema di controllo di versione*. Un sistema di controllo di versione, *Version Control System o VCS*, è un software che consente di registrare nel tempo i cambiamenti apportati a dei file (le loro versioni o revisioni successive), non necessariamente di codice sorgente, e permette poi di compiere una serie di operazioni quali: sottometterne versioni aggiornate, utilizzare una determinata versione; tenere traccia di tutte le modifiche apportate e così via.

.NET Core rappresenta una totale ristrutturazione del .NET Framework, che si basa, in linea generale, su due pilastri fondamentali: quello della *portabilità*, ossia della possibilità concreta di un effettivo *porting* dell'ambiente di *runtime* (CoreCLR) e della libreria di base (CoreFX) su sistemi anche differenti da Windows, e quello della *modularizzazione*, ovvero della possibilità di fornire un package di un'applicazione che contiene anche il CoreCLR e che quindi non dipende da nessuna installazione sul sistema target di un .NET Framework (si possono anche installare diverse applicazioni, *side-by-side*, ciascuna con una propria versione del CoreCLR aggiornabile individualmente). Si possono fornire per un'applicazione solo le librerie CoreFX necessarie (ciò è possibile perché CoreFX è progettato a *componenti* ossia come un set di librerie suddivise per funzionalità, con poche dipendenze e distribuite come package tramite NuGet).

TERMINOLOGIA

La modalità di creazione di pacchetti di applicazioni dipendenti solo da alcune funzionalità della libreria CoreFX effettivamente utilizzata è chiamata *pay-for-play model*.

NuGet

NuGet è il package manager ufficiale di Microsoft utilizzato per produrre o consumare dei file "speciali" che contengono librerie, o tool più in generale, e che sono definiti come *package* (file con estensione `.nupkg`). NuGet è diventato, dal 2010, anno della sua prima apparizione, uno strumento fondamentale per la creazione, ricerca, installazione e aggiornamento di librerie, anche di terze parti, che ruotano attorno al mondo dello sviluppo .NET. Per utilizzare NuGet è necessario installare il relativo programma client che viene distribuito sia come utility a riga di comando (`nuget.exe`) sia come package VSIX (file con estensione `.vsix`) ossia come un'estensione utilizzabile all'interno dell'IDE Visual Studio. In ogni caso è bene tenere presente che NuGet è nato con lo scopo di facilitare l'integrazione di librerie di terze parti nei progetti Visual Studio che in modo automatico

possono essere installate (ne vengono copiati nella soluzione i relativi file, ne vengono aggiunti i corrispondenti riferimenti, viene modificato opportunamente il file `app.config` o `web.config`), disinstallate (ne vengono rimossi dalla soluzione i relativi file, ne vengono rimossi i corrispondenti riferimenti, viene ripristinato il file `app.config` o `web.config`) e aggiornate. L'utility `nuget.exe` invece provvederà solo a scaricare e scompattare un determinato package, ma non interagirà con Visual Studio modificandone un relativo progetto. Starà quindi allo sviluppatore decidere "manualmente" come utilizzare i file di libreria scaricati. Ciò precisato, `nuget.exe` è comunque necessario se si desidera produrre e pubblicare un package, perché è solo per il suo tramite che è possibile compiere le predette operazioni. Comunque, per chi non ama i tool a riga di comando, è possibile anche usare un apposito strumento grafico denominato NuGet Package Explorer. Per chi volesse approfondire l'argomento è disponibile un apposito sito, raggiungibile all'URL `https://www.nuget.org`, il quale, oltre a consentire di scaricare un client NuGet e fornire una copiosa documentazione, consente anche la ricerca e l'upload dei package.

All'interno dell'area .NET Core 5 vediamo la presenza di due sotto-aree: una è quella propria dell'ambiente di *runtime* CoreCLR, condiviso da Windows, Gnu/Linux e OS X; l'altra, denominata .NET Native, a vantaggio del solo sistema Windows, rappresenta una tecnologia di precompilazione grazie alla quale è possibile produrre applicazioni per Windows direttamente in codice nativo.
Ricordiamo che, tipicamente, un'applicazione per il .NET Framework è compilata nel linguaggio intermedio IL, il quale, a *runtime*, è poi tradotto in codice nativo da un compilatore *Just-In-Time* (JIT). Viceversa, con .NET Native, un'applicazione per il .NET Framework è compilata direttamente in codice nativo da un compilatore *Ahead of Time* (AOT) e ciò permette, soprattutto se tale applicazione è indirizzata al settore mobile, di caricarsi più velocemente ed eseguirsi con maggiore performance.

ATTENZIONE

Allo stato attuale la tecnologia .NET Native è utilizzabile per la creazione di applicazioni per Windows 10 e distribuibili nel Windows Store.

Al di sopra dell'area .NET Core 5, invece, troviamo un layer che comprende le tecnologie ASP.NET e Universal Windows Platform, *framework* che consente di creare applicazioni che possono eseguirsi su tutti i dispositivi dove è presente il sistema operativo Windows 10 installato (per esempio, PC, tablet, smartphone, console XBOX e così via) e che sono altresì distribuibili su Windows Store.
Per quanto attiene, infine, al layer inferiore comune e condiviso ad ambedue le macro-aree, esso comprende: *componenti di runtime*, ne fanno parte, fondamentalmente, un nuovo compilatore JIT di codice IL, codename RyuJIT, next-gen e a 64 bit, un *Garbage collector* (GC) migliorato e aggiornato, la tecnologia SIMD (*Single Instruction*, *Multiple Data*) che consente l'effettuazione di operazioni matematiche che si avvalgono di vettori, in parallelo, e dunque con una maggiore velocità computazionale; *librerie*, ne fanno parte un vasto insieme di librerie di base e comuni a entrambe le piattaforme; *compilatori*, ne fa parte la piattaforma di compilazione denominata Roslyn che fornisce l'implementazione dei compilatori per C# e per Visual Basic.
Da questa disamina sul futuro della piattaforma .NET appare evidente come Microsoft si stia impegnando in una complessa e articolata riorganizzazione che, al momento

attuale (inizio 2016), prevede un binario, sul quale corre il treno di .NET Framework esplicitamente pensato per il sistema Windows, e un altro binario sul quale corre il treno di .NET Core, esplicitamente pensato multi-sistema (GNU/Linux, OS X e così via) e open-source.
In ogni caso è prevista una convergenza tra i *framework* nel senso che oltre a condividere una serie di tecnologie fondamentali (librerie, compilatori, domini applicativi e così via) qualsiasi miglioria apportata al .NET Core potrà essere riportata nel .NET Framework e viceversa (in quest'ultimo caso la modifica dovrà essere, chiaramente, *cross-platform*).
In conclusione anche se è difficile dirlo con certezza, data l'alta dinamicità di innovazione legata al settore dello sviluppo software, possiamo provare a ipotizzare un futuro in cui il .NET Framework continuerà la sua evoluzione naturale e ciclica (4.6, 4.7 e così via), con l'aggiornamento anche di API *non cross-platform* e di aspetti strettamente legati al sistema Windows. Esso continuerà dunque a essere utilizzato per lo sviluppo di applicazioni per il sistema Windows che avranno bisogno di sfruttarne tutte le specifiche peculiarità. Il .NET Core, invece, pur continuando la propria evoluzione migliorativa e integrativa, sarà principalmente utilizzato per la costruzione di applicazioni *cross-platform* che necessiteranno di un ambiente di esecuzione snello e ottimizzato e di un sistema che permetterà di fornire solo i servizi strettamente necessari (modularizzazione).

Mono e .NET Core

Mono, come già detto, è un'implementazione open-source e multi-piattaforma del .NET Framework di Microsoft. È costituito dai seguenti componenti: un compilatore C#; un ambiente di *runtime* (*Mono Runtime*); una libreria di base di classi (*Base Class Library*) che fornisce funzionalità essenziali per una qualsiasi applicazione (collezioni, IO, networking e così via); una libreria estesa di classi (*Mono Class Library*) che fornisce funzionalità aggiuntive utili soprattutto per applicazioni eseguibili in ambienti GNU/Linux (Gtk+, LDAP, OpenGL e così via). Esso rappresenta, dal lontano 2004 (anno in cui fece la comparsa la sua prima release), un *framework* con l'obiettivo di costruire applicazioni per il mondo .NET in modo *cross-platform* e, prima dell'avvento di .NET Core, era (ed è tuttora) anche il *framework* di maggior successo e utilizzo. Ora, però, con l'avvento di .NET Core, che si prefigge di raggiungere gli stessi scopi, come potrà la piattaforma Mono relazionarsi con essa? Anche in questo caso dare una risposta sicura ed esaustiva è impresa ardua e foriera di congetture. Proviamo, comunque, a fare un paio di ipotesi: è improbabile che vi sarà una "fusione" del progetto Mono nel progetto .NET Core, in modo da non avere due *fork* separati del .NET Framework (in pratica gli sviluppatori di Mono e quelli del .NET team lavorerebbero insieme per un'unica piattaforma sponsorizzata da Microsoft); più probabilmente entrambi i progetti continuerebbero ad avere la propria indipendenza, ma integrerebbero, in modo bidirezionale, tutte le migliorie e aggiunte (in pratica gli sviluppatori di Mono e quelli del .NET team collaborerebbero insieme "scambiandosi" il codice prodotto).

Cenni sull'architettura di un elaboratore

Un linguaggio di programmazione, indipendentemente dal fatto che sia categorizzato come linguaggio di basso, medio o alto livello, deve sempre far affidamento al sottostante *hardware* per il suo funzionamento e per la produzione del codice eseguibile proprio di una qualsivoglia applicazione.

TERMINOLOGIA

Un linguaggio di programmazione è definibile di *alto livello* se offre un alto livello di astrazione e indipendenza rispetto ai dettagli hardware di un elaboratore. Ciò implica che un programmatore utilizzerà *keyword* e costrutti sintattici di facile comprensione, che gli permetteranno di scrivere un programma in modo relativamente semplificato, con la possibilità di concentrarsi solo sulla logica dell'algoritmo da implementare. I linguaggi di alto livello sono definibili come linguaggi *closer to humans*.
Di converso un linguaggio di programmazione è definibile di *basso livello* quando non offre alcun layer di intermediazione/astrazione rispetto all'hardware da programmare e il programmatore deve non solo avere una profonda conoscenza di tale hardware, ma deve anche lavorare direttamente con esso (registri, memoria e così via). I linguaggi di basso livello sono definibili come linguaggi *closer to computers*.
Infine, un linguaggio di programmazione è definibile di *medio livello* quando mette a disposizione del programmatore sia funzionalità di *alto livello* (per esempio il costrutto di *funzione* o il costrutto di *struttura* o *classe*), che garantiscono una maggiore efficienza e flessibilità nella costruzione dei programmi, sia funzionalità di *basso livello* (come il costrutto di *puntatore*) che garantiscono, al pari dei linguaggi *machine-oriented* come l'assembly, una maggiore efficienza nello sfruttamento diretto dell'hardware sottostante.

Prima, quindi, di addentrarci nello studio sistematico del linguaggio C# appare conveniente delineare alcuni concetti teorici che riguardano la struttura di un generico elaboratore elettronico, soffermandoci in modo più approfondito su due componenti in particolare: la CPU e la memoria centrale.
Ciò si rende opportuno soprattutto quando un linguaggio di programmazione consente di sfruttare a basso livello l'hardware di un sistema target; pertanto, comprendere, seppure a grandi linee, come un computer è progettato e costituito, può sicuramente aiutare a scrivere programmi che *dialogano* con l'hardware sottostante con maggiore consapevolezza dei loro effetti (capire, per esempio, come è strutturata la memoria centrale può sicuramente aiutare a gestirla in modo più sicuro ed efficiente).
In ogni caso, quanto diremo resterà comunque valido anche per un linguaggio di alto livello come C# che, pur non consentendo normalmente di sfruttare direttamente l'hardware sottostante, permette comunque, mediante determinate tecniche, di scrivere il codice di un'applicazione in *modo unsafe* (per esempio quando il codice richiede di dialogare con una DLL *unmanaged* che magari espone funzionalità per la manipolazione diretta dell'hardware oppure della memoria).

TERMINOLOGIA

Il codice che viene eseguito nell'ambito dei confini propri di un sistema di *runtime*, come è per esempio il CLR, ossia sotto il suo controllo, è definito *managed code* (codice gestito). Viceversa, il codice che viene eseguito al di fuori di un sistema di *runtime* è definito *unmanaged code* (codice non gestito). Di conseguenza, una *DLL unmanaged* è una libreria a collegamento dinamico (*Dynamic Link Library*) che espone funzionalità (codice) non gestite e non sotto il controllo di un ambiente di *runtime*. I componenti COM, le interfacce ActiveX e le funzioni delle API Win32 sono esempi calzanti di librerie con codice *unmanaged*.

Il modello di von Neumann

I linguaggi imperativi, come il C#, condividono un modello computazionale che rappresenta un'astrazione del sottostante calcolatore elettronico dove la computazione procede modificando valori memorizzati all'interno di locazioni di memoria. Questo modello è definito come *von Neumann Architecture*, dal nome dello scienziato ungherese John von Neumann che, nel 1945, lo ideò. Tale modello è alla base della progettazione e costruzione dei computer e quindi dei linguaggi imperativi che vi si rifanno e che sono, dunque, astrazioni della macchina di von Neumann (Figura 1.8).

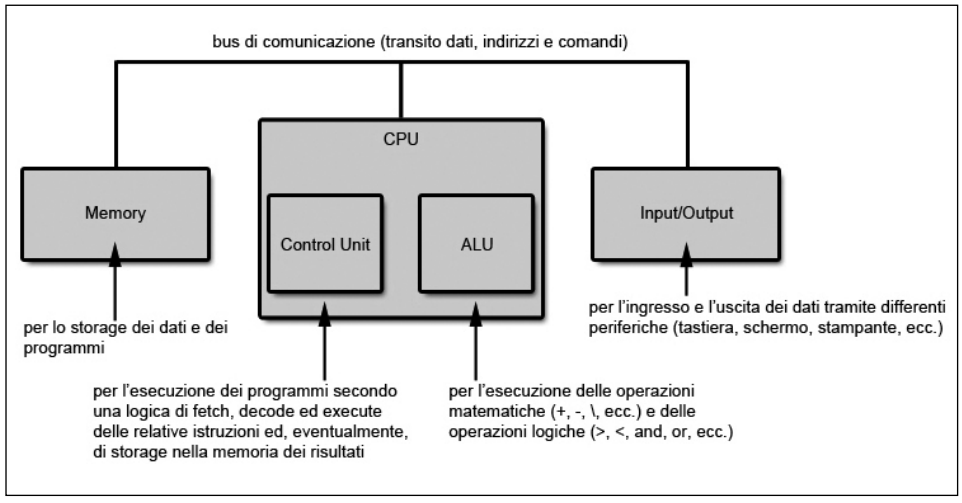

Figura 1.8 Architettura semplificata di un computer basata sul modello di von Neumann.

In sostanza, nel modello di von Neumann un computer è costituito dai seguenti elementi: una CPU (*Central Processing Unit*) per il controllo e l'esecuzione dell'elaborazione, al cui interno si trovano l'ALU (*Arithmetic Logic Unit*) e una *Control Unit*; celle di memoria identificate da un indirizzo numerico atte a ospitare i dati coinvolti nell'elaborazione; dispositivi per l'input e l'output dei dati da e verso l'elaboratore; un bus di comunicazione tra le varie parti per il transito di dati, indirizzi e segnali di controllo. Sia i dati, sia le istruzioni di programmazione sono memorizzati nella memoria. In pratica nel modello di von Neumann abbiamo due elementi caratterizzanti: la memoria, che memorizza le informazioni, e il processore, che fornisce operazioni per modificare il contenuto, ossia lo stato della memoria. In definitiva un computer digitale modellato secondo l'architettura di von Neumann non è altro che un sistema di componenti quali processori, memorie, device di input/output e così via tra di loro interconnessi (*bus oriented*) che cooperano congiuntamente al fine di svolgere i processi computazionali per i quali sono stati progettati.

La CPU

La CPU (*Central Processing Unit*) è il "cervello" di ogni computer ed è deputata principalmente a interpretare ed eseguire le istruzioni elementari che rappresentano i programmi e a effettuare operazioni di coordinamento tra le varie parti di un computer.

Questa unità di elaborazione, dal punto di vista fisico, è un circuito elettronico formato da un elevato numero di transistor (da diverse centinaia di milioni fino ai miliardi delle più potenti CPU) che sono ubicati in un circuito integrato (chip) di ridotte dimensioni (pochi centimetri quadrati).
Dal punto di vista logico, invece, una CPU è formata dalle seguenti unità (Figura 1.9).

- ALU (*Arithmetic Logic Unit*) ossia l'unità aritmetico-logica. È costituita da un insieme di circuiti deputati a effettuare calcoli aritmetici (somme, sottrazioni e così via) e operazioni logiche (boolean AND, boolean OR e così via) sui dati. Il suo funzionamento si può così sintetizzare ponendo, per esempio, un'operazione di somma tra due numeri: preleva da appositi registri di memoria di input gli operandi trasmessi (diciamo `X` e `Y`); effettua l'operazione di somma (`X + Y`); scrive il risultato della somma in un registro di memoria di output. In pratica possiamo vedere una ALU come una sorta di calcolatrice "primitiva", che riceve dati sui quali effettuare calcoli al fine di produrre un risultato.

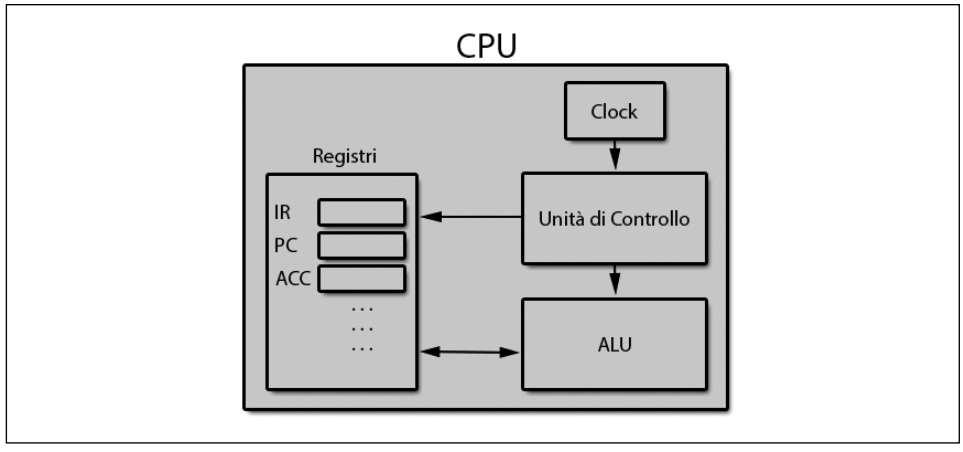

Figura 1.9 Vista logica dell'interno di una CPU.

- *Control Unit* ossia l'unità di controllo. Coordina e dirige le varie parti di un calcolatore in modo da consentire la corretta esecuzione dei programmi. In pratica essa, in una sorta di ciclo infinito, preleva (*fetch*), decodifica (*decode*) ed esegue (*execute*) le istruzioni dei programmi. Attraverso la fase di prelievo acquisisce un'istruzione dalla memoria, la carica nel registro istruzione e rileva la successiva istruzione da prelevare. Attraverso la fase di decodifica interpreta l'istruzione da eseguire. Attraverso la fase di esecuzione esegue ciò che l'istruzione indica. È un'azione di input? Allora incarica l'unità di input di trasferire dei dati nella memoria centrale. È un'azione di output? Allora incarica l'unità di output di trasferire dei dati dalla memoria centrale verso l'esterno. È un'azione di elaborazione dei dati? Allora richiede il trasferimento dei dati nell'ALU, incarica l'ALU di elaborarli e trasferisce il risultato nella memoria centrale. È un'azione di salto? Allora aggiorna il registro *contatore di programma* con l'indirizzo cui saltare. Le operazioni svolte dall'unità di controllo sono regolate da un orologio interno di sistema (*system clock*) che genera segnali o impulsi regolari a una certa frequenza, espressa in *hertz*, che consentono alle varie parti di operare in

modo coordinato e sincronizzato. Maggiori sono questi segnali per secondo, detti anche cicli di clock, maggiore è la quantità di istruzioni per secondo che una CPU può elaborare e quindi la sua velocità.

- *Registers* ossia i registri. Sono unità di memoria estremamente veloci (possono essere letti e scritti ad alta velocità perché sono interni alla CPU) e di una certa dimensione, utilizzati per specifiche funzionalità. Vi sono numerosi registri; quelli più importanti sono l'*Instruction Register* (registro istruzione), che contiene la corrente istruzione che si sta eseguendo, il *Program Counter* (contatore di programma), contiene l'indirizzo della successiva istruzione da eseguire; gli *Accumulator* (accumulatori), contengono, temporaneamente, gli operandi di un'istruzione e alla fine della computazione il risultato dell'operazione eseguita dall'ALU.

La memoria centrale

La memoria centrale, detta anche primaria o principale, è quella parte del computer dove sono memorizzate le istruzioni e i dati dei programmi.
Ha diverse caratteristiche: è *volatile*, perché il contenuto si perde nel momento in cui il calcolatore viene spento; è *veloce*, ossia il suo accesso in lettura/scrittura può avvenire in tempi estremamente ridotti; è ad *accesso casuale* perché il tempo di accesso alle relative celle è indipendente dalla loro posizione e dunque costante per tutte le celle (da questo punto di vista la memoria centrale è definita come RAM, *Random Access Memory*).
L'unità di base di memorizzazione è la cifra binaria o *bit*, che è il composto aplologico delle parole *binary digit*. Un bit può contenere solo due valori: la cifra `0` oppure la cifra `1`; il relativo sistema di numerazione binario richiede, pertanto, solo quei due valori per la codifica dell'informazione digitale.

NOTA

Consultare l'*Appendice B Sistemi numerici: cenni* per un approfondimento sul sistema di numerazione binario.

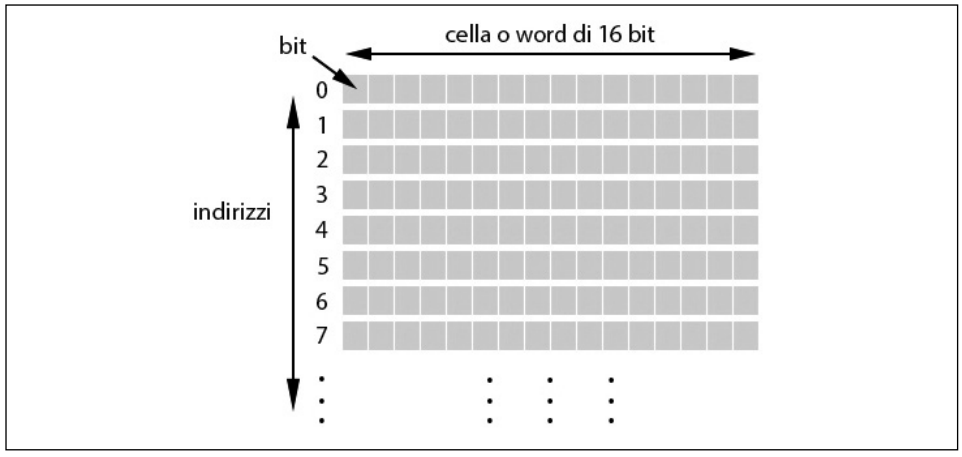

Figura 1.10 Memoria primaria come sequenza di celle.

La memoria primaria, dal punto di visto logico, è rappresentabile (Figura 1.10) come una sequenza di celle o locazioni di memoria, ciascuna delle quali può memorizzare una certa quantità di informazioni. Ogni cella, detta *parola* (*memory word*), ha una dimensione fissa e tale dimensione, espressa in multipli di 8, ne indica la sua lunghezza, ossia il suo numero di bit (possiamo avere, per esempio, word di 8 bit, di 16 bit, di 32 bit e così via). Così se una cella ha k bit allora la stessa può contenere una delle 2^k combinazioni differenti di bit. Inoltre la lunghezza della word indica anche che la quantità di informazioni che un computer durante un'operazione può elaborare, allo stesso tempo e in parallelo.
Altra caratteristica essenziale di una cella di memoria è che essa ha un *indirizzo* ossia un numero binario che ne consente la localizzazione da parte della CPU, al fine di reperire o scrivervi contenuti informativi, anch'essi binari. La quantità di indirizzi referenziabili dipende dal numero di bit di un indirizzo: generalizzando, se un indirizzo ha n bit, allora il massimo numero di celle indirizzabili sarà 2^n.
Per esempio la Figura 1.11 mostra tre differenti layout per una memoria di 160 bit rispettivamente con celle di 8 bit, 16 bit e 32 bit. Nel primo caso sono necessari almeno 5 bit per esprimere 20 indirizzi da 0 a 19 (infatti, 2^5 ne permette fino a 32); nel secondo caso sono necessari almeno 4 bit per esprimere 10 indirizzi da 0 a 9 (infatti, 2^4 ne permette fino a 16); nel terzo caso sono necessari almeno 3 bit per esprimere 5 indirizzi da 0 a 4 (infatti, 2^3 ne permette fino a 8).
In più è importante dire che il numero di bit di un indirizzo è indipendente dal numero di bit per cella: una memoria con 2^{10} celle di 8 bit ciascuna e una memoria con 2^{10} celle di 16 bit ciascuna necessiteranno entrambe di indirizzi di 10 bit.
La dimensione di una memoria è dunque data dal numero di celle per la loro lunghezza in bit. Nei nostri tre casi è sempre di 160 bit (correntemente, un computer moderno avrà una dimensione di 4 Gigabyte di memoria se avrà almeno 2^{32} celle indirizzabili di 1 byte di lunghezza ciascuna).

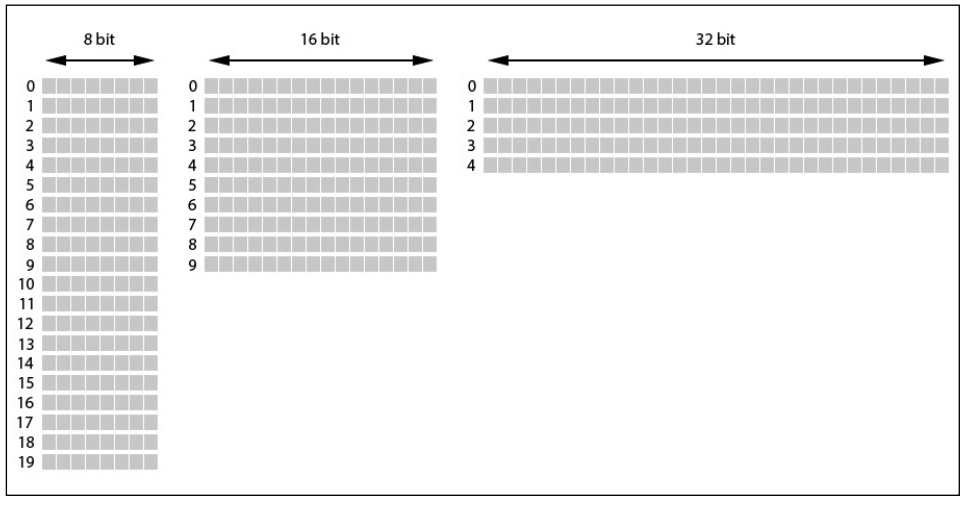

Figura 1.11 Tre differenti layout per una memoria di 160 bit.

Ordinamento dei byte

Quando una word è composta da più di 8 bit (1 byte), e quindi 16 bit (2 byte), 32 bit (4 byte) e così via vi è la necessità di decidere come ordinare o disporre in memoria i relativi byte.
Oggi esistono due modalità di ordinamento largamente utilizzate dai moderni elaboratori elettronici.

- *Big endian* (architetture Motorola 68k, SPARC e così via): data una word, il byte più significativo (*most significant byte*) è memorizzato all'indirizzo più basso (*smallest address*) e a partire da quello, da sinistra a destra (*left-to-right*), sono memorizzati i byte successivi.
- *Little endian* (architetture Intel x86, x86-64 e così via): data una word, il byte meno significativo (*least significant byte*) è memorizzato all'indirizzo più basso (*smallest address*) e a partire da quello, da destra a sinistra (*right-to-left*), sono memorizzati i byte successivi.

> **NOTA**
>
> Questi termini si devono allo scrittore e poeta irlandese Jonathan Swift, che nel suo famoso libro *I Viaggi di Gulliver* prendeva in giro i politici che si facevano guerra per il giusto modo di rompere le uova sode: dalla punta più piccola (little end) oppure dalla punta più grande (big end)? Questo termine fu comunque usato per la prima volta da Danny Cohen, uno scienziato informatico, in un significativo articolo del 1980, dal titolo *On Holy Wars and a Plea for Peace*, rintracciabile al seguente URL: `http://www.ietf.org/rfc/ien/ien137.txt`.

Per comprendere quanto detto, consideriamo come viene memorizzato un numero come 2854123 (binario, 00000000001010111000110011101011) in una word di 32 bit secondo un'architettura big endian e secondo un'architettura little endian (Figura 1.12): nel primo caso il byte più a sinistra (più significativo) è memorizzato nell'indirizzo 0 e poi a seguire, da sinistra a destra, gli altri byte negli indirizzi 1, 2 e 3; nel secondo caso il byte più a destra (meno significativo) è memorizzato nell'indirizzo 0 e poi a seguire, da destra a sinistra, gli atri byte negli indirizzi 1, 2 e 3.

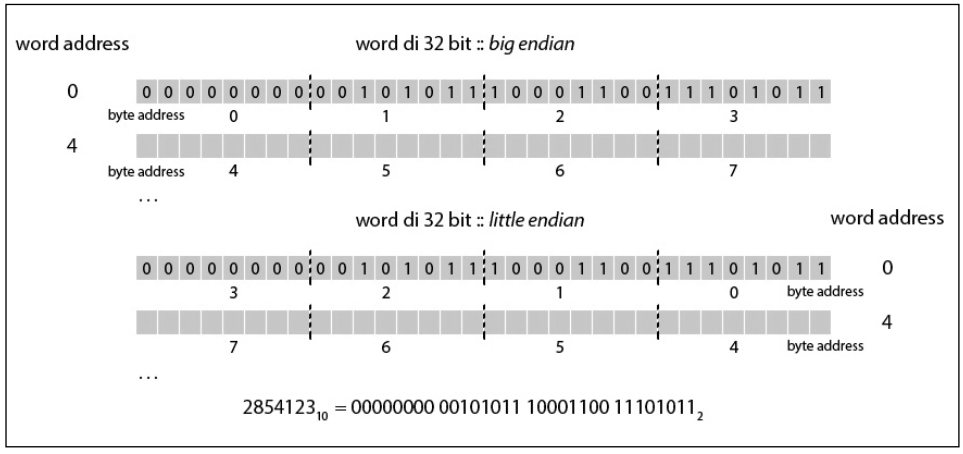

Figura 1.12 Ordinamento della memoria: big endian vs little endian.

TERMINOLOGIA

Se vediamo una word come una sequenza di bit (Figura 1.13) piuttosto che una sequenza di byte possiamo altresì dire che essa avrà un bit più significativo (*most significant bit*) che sarà ubicato al limite sinistro di tale sequenza (*high-order end*) e un bit meno significativo (*least significant bit*) che sarà ubicato al limite destro sempre della stessa sequenza (*low-order end).*

Figura 1.13 Word come sequenza di bit.

In definitiva, se due sistemi adottano questi due metodi di ordinamento in memoria dei byte e si devono scambiare dei dati, vi possono essere dei problemi di congruità tra i dati inviati da un sistema low endian verso un sistema big endian se non sono previsti appositi accorgimenti oppure se non sono forniti idonei meccanismi di conversione.
Per esempio, se un'applicazione scritta su un sistema SPARC memorizza dei dati binari in un file e poi lo stesso file è aperto in lettura su un sistema x86 si avranno dei problemi di congruità, perché il file è stato scritto in modo *endian-dependent*.
Per evitare tale problema si possono adottare vari accorgimenti come, per esempio, quello di scrivere i dati in un formato *neutrale* che prevede file testuali e stringhe oppure adottare idonee routine di conversione (*byte swapping*) che, a seconda del sistema in uso, forniscano la corretta rappresentazione dell'informazione binaria.
In pratica quando si deve scrivere software per diverse piattaforme hardware che hanno sistemi di *endianness* incompatibili, tale software deve essere sempre pensato in modo portabile, non presupponendo mai un particolare ordinamento in memoria dei byte.

Paradigmi di programmazione

Un paradigma o stile di programmazione, indica un determinato modello concettuale e metodologico, offerto in termini concreti da un linguaggio di programmazione, al quale fa riferimento un programmatore per progettare e scrivere un programma informatico e dunque per risolvere un determinato problema algoritmico. Si conoscono molti differenti paradigmi di programmazione, ma quelli che seguono sono i più comuni.

- *Il paradigma procedurale*: l'unità principale di programmazione è, per l'appunto, la procedura o la funzione, che ha lo scopo di manipolare i dati del programma. Questo paradigma è talune volte indicato anche come *imperativo*, perché consente di costruire un programma indicando dei comandi (assegna, chiama una procedura, esegui un loop e così via) che esplicitano quali azioni si devono eseguire, e in quale ordine, per risolvere un determinato compito. Questo paradigma si basa, dunque, su due aspetti di rilevo: il primo è riferito al cambiamento di stato del programma che è causa

delle istruzioni eseguite (si pensi al cambiamento del valore di una variabile in un determinato tempo durante l'esecuzione del programma); il secondo è inerente allo stile di programmazione adottato, che è orientato al "come fare o come risolvere" piuttosto che al "cosa si desidera ottenere o cosa risolvere". Esempi di linguaggi che supportano il paradigma procedurale sono: FORTRAN, COBOL, Pascal, C e così via.

- *Il paradigma a oggetti*: l'unità principale di programmazione è l'oggetto (nei sistemi basati sui prototipi) oppure la classe (nei sistemi basati sulle classi). Questi oggetti, definibili come *virtuali*, sono astrazioni concettuali degli oggetti reali del mondo fisico che intendono modellare. Questi ultimi possono essere oggetti più generali (per esempio un computer) oppure oggetti più specifici, ovvero maggiormente specializzati (per esempio una scheda madre, una scheda video e così via). Noi utilizziamo tali oggetti senza sapere nulla della complessità con cui sono costruiti e comunichiamo con essi attraverso messaggi (sposta il puntatore, digita dei caratteri) e mediante interfacce (mouse, tastiera). Inoltre, essi sono dotati di attributi o caratteristiche (velocità del processore, colore del case e così via) che possono essere letti e, in alcuni casi, modificati. Questi oggetti reali vengono presi come modello per la costruzione di sistemi software a oggetti, dove l'oggetto (o la classe) avrà dei metodi per l'invio di messaggi e degli attributi che rappresenteranno i dati da manipolare. Principi fondamentali di tale paradigma sono i seguenti.
  - L'*incapsulamento*: un meccanismo attraverso il quale i dati e il codice di un oggetto sono protetti da accessi arbitrari (*information hiding*). Per dati e codice intendiamo tutti i membri di una classe, ovvero sia i membri dati (definiti anche, nel gergo della OOP semplicemente come *campi*), sia i membri funzione (definiti anche, nel gergo della OOP semplicemente come *metodi*). La protezione dell'accesso viene effettuata applicando ai membri della classe degli specificatori o modificatori di accesso, definibili come: *pubblico*, con cui si consente l'accesso a un membro di una classe da parte dei metodi di altre classi; *protetto*, con cui si consente l'accesso a un membro di una classe solo da parte di metodi appartenenti alle sue classi derivate; *privato*, con cui un membro di una classe non è accessibile né da metodi di altre classi né da quelli delle sue classi derivate, ma soltanto dai metodi della sua stessa classe.
  - L'*ereditarietà*: un meccanismo attraverso il quale una classe può avere relazioni di ereditarietà nei confronti di altre classi. Per relazione di ereditarietà intendiamo una relazione gerarchica di parentela *padre-figlio*, dove una classe figlio (definita *classe derivata* o *sottoclasse*) deriva da una classe padre (definita *classe base* o *superclasse*) i metodi e le proprietà pubbliche e protette, e dove essa stessa ne definisce di proprie. Con l'ereditarietà si può costruire, di fatto, un modello orientato agli oggetti che in principio è generico e minimale (ha solo classi base) e poi, a mano a mano che se ne presenta l'esigenza, può essere esteso attraverso la creazione di sottomodelli sempre più specializzati (ha anche classi derivate).
  - Il *polimorfismo*: un meccanismo attraverso il quale si può scrivere codice in modo generico ed estendibile grazie al potente concetto che una classe base può riferirsi a tutte le sue classi derivate cambiando, di fatto, la sua *forma*. Ciò si traduce, in pratica, nella possibilità di assegnare a una variabile `A` (istanza di una classe base) il riferimento di una variabile `B` (istanza di una classe derivata da `A`) e, successivamente, riassegnare alla stessa variabile `A` il riferimento di una

variabile `C` (istanza di un'altra classe derivata da `A`). La caratteristica appena indicata consentirà, attraverso il riferimento `A`, di invocare i metodi di `A` che `B` o `C` hanno ridefinito in modo specializzato, con la garanzia che il sistema run-time del linguaggio di programmazione a oggetti saprà sempre a quale classe derivata appartengono. La discriminazione automatica, effettuata dal sistema *runtime* di un tale linguaggio, di quale oggetto (istanza di una classe derivata) è contenuto in una variabile (istanza di una classe base) è effettuata con un meccanismo definito *dynamic binding* (*binding* dinamico).

TERMINOLOGIA

Nel gergo OOP, i membri di una classe sono chiamati in molteplici modi. Per i membri che rappresentato dei "dati" viene usata la seguente terminologia: *campi*, *membri dati*, *membri di dati*, *dati membro*, *variabili membro*. Per i membri che invece rappresentano "funzionalità" possono essere impiegati i seguenti ti termini: *metodi*, *membri funzione*, *membri di funzioni*, *funzioni membro*. Per quanto riguarda il presente testo utilizzeremo, in generale, i termini così come sono stati indicati nel documento di specifica del linguaggio C# ossia per i dati, *membri dati* e per le funzionalità, *membri funzione*.

Esempi di linguaggi che supportano il paradigma a oggetti sono: Java, C#, C++, JavaScript, Smalltalk, Python e così via.

- *Il paradigma funzionale*: l'unità principale di programmazione è la funzione vista in puro senso matematico. Infatti, il flusso esecutivo del codice è guidato da una serie di valutazioni di funzioni che, trasformando i dati che elaborano, conducono alla soluzione di un problema. Gli aspetti rilevanti di questo paradigma sono: nessuna mutabilità di stato (le funzioni sono *side-effect free*, ossia non modificano alcuna variabile); il programmatore non si deve preoccupare dei dettagli implementativi del "come" risolvere un problema, ma piuttosto di "cosa" si vuole ottenere dalla computazione. Esempi di linguaggi che supportano il paradigma funzionale sono: Lisp, Haskell, F#, Erlang e Clojure.
- *Il paradigma logico*: l'unità principale di programmazione è il *predicato logico*. In pratica con questo paradigma il programmatore dichiara solo i "fatti" e le "proprietà" che descrivono il problema da risolvere lasciando al sistema il compito di "inferirne" la soluzione e dunque raggiungerne il "goal" (l'obiettivo). Esempi di linguaggi che supportano il paradigma logico sono: Datalog, Mercury, Prolog, ROOP e così via.

Per esempio, un linguaggio come il C supporta pienamente il paradigma procedurale, dove l'unità principale di astrazione è rappresentata dalla funzione attraverso la quale si manipolano i dati di un programma. Da questo punto di vista, per meglio comprendere quanto detto, ciò si differenzia dai linguaggi di programmazione che sposano il paradigma a oggetti come, per esempio, C#, C++ o Java, perché in quest'ultimo paradigma ci si concentra prima sulla creazione di nuovi tipi di dato (le classi) e poi sui membri funzione e i membri dati a essi relativi. In altre parole, mentre in un linguaggio procedurale come il C la modularità di un programma viene fondamentalmente descritta dalle procedure o funzioni che manipolano i dati, nella programmazione a oggetti la modularità viene descritta dalle classi che incapsulano al loro interno membri funzione e membri dati. Per questa ragione si suole dire che nel mondo a oggetti la dinamica (metodi) è subordinata alla struttura (classi).

TERMINOLOGIA

Nei linguaggi di programmazione si usano termini come funzione (*function*), metodo (*method*), procedura (*procedure*), sottoprogramma (*subprogram*), sottoroutine (*subroutine*) e così via per indicare un *blocco* di codice posto a un determinato indirizzo di memoria che è invocabile, richiamabile, per eseguire le istruzioni che vi si trovano. Dal punto di vista pratico, pertanto, significano tutte la stessa cosa anche se, in letteratura, talune volte sono evidenziate delle differenze soprattutto in base al linguaggio di programmazione che si prende in esame. Per esempio: in Pascal una funzione restituisce un valore mentre una procedura non restituisce nulla; in C una funzione può agire anche come una procedura, mentre il termine metodo non esiste; in C# un metodo è una funzionalità "associata" all'oggetto o alla classe in cui è stato definito ed è anche denominato *membro funzione*.

Concetti introduttivi allo sviluppo in C#

Prima di affrontare lo studio sistematico della programmazione in C# e di descrivere un semplice programma introduttivo, appare opportuno soffermarsi su alcuni concetti propedeutici allo sviluppo C# che sono trasversali al linguaggio e che servono a inquadrarlo meglio nel suo complesso.

Fasi di sviluppo di un programma in C#

Un programma scritto in C# prima di poter essere eseguito passa attraverso le seguenti fasi che ne definiscono un generico *ciclo* operativo.

- *Analisi*. Questa è la fase che sottende alla individuazione delle informazioni preliminari allo sviluppo di un software, le quali possono riguardare la sua fattibilità in senso tecnico ed economico (analisi costi/benefici), il suo dominio applicativo, i suoi requisiti funzionali (cosa il software deve offrire) e così via.
- *Progettazione*. In questa fase si inizia a ideare in modo più concreto come si può sviluppare il software che è stato oggetto della precedente analisi. In pratica il software viene scomposto in moduli e componenti e si definisce la loro interazione e anche il loro contenuto (dettaglio interno). La progettazione indica, pertanto, *come* il software deve essere implementato piuttosto di *cosa* deve fare il software (appannaggio della fase di analisi).

TERMONOLOGIA

La scomposizione di un programma in moduli o componenti segue tipicamente una logica *divide et impera* (dividi e domina o dividi e conquista). In sostanza, un problema viene diviso in vari sotto-problemi i quali sono a loro volta divisi in altri sotto-problemi e questa divisione continua finché i più piccoli sotto-problemi non sono in grado di ottenere la relativa soluzione nel modo più semplice possibile. Dopodiché, tutte le soluzioni semplici trovate vengono combinate per produrre la soluzione generale del problema. Quest'approccio alla soluzione dei problemi è anche detto *top-down* (dall'alto verso il basso) perché si parte dal formulare un problema generale e in modo poco dettagliato (posto, cioè, in cima a una piramide ideale) e poi lo si decompone, rifinisce (*stepwise refinement*), in sotto-problemi più specializzati e maggiormente dettagliati (posti, cioè, in alla base della piramide ideale).

- *Codifica*. Questa è la fase in cui si implementa concretamente il software oggetto della progettazione. In pratica, attraverso l'utilizzo di un editor di testo, si scrivono gli algoritmi, le funzionalità e le istruzioni del programma codificate secondo la sintassi propria del linguaggio C#. Il codice scritto nell'editor è detto codice sorgente (*source code*), e il file prodotto è un file di codice sorgente (*source code file*).
- *Compilazione*. Questa è la fase in cui un apposito programma, il compilatore (*compiler*), traduce, converte il codice sorgente nel relativo file in codice intermedio (*intermediate language code*), ovvero in codice non direttamente eseguibile nel sistema target, ma comprensibile e interpretabile solo dal sistema di *runtime*, attraverso un apposito compilatore Just-In-Time (JIT). Questo codice intermedio viene scritto in un apposito file (*intermediate language code file*). Inoltre, in questa fase, il compilatore si occupa anche di verificare che il codice sorgente sia scevro da errori sintattici che violerebbero le regole di C#; in caso contrario, avvisa l'utente della presenza di tali errori, interrompe la compilazione e non procede alla generazione del codice intermedio.
- *Esecuzione*. Questa è la fase in cui il file contenente il codice intermedio viene caricato nella memoria, convertito in codice nativo per la specifica piattaforma ed è eseguito ovvero produce gli effetti computazionali per i quali è stato progettato e sviluppato. In linea generale, in una shell testuale, per caricare in memoria ed eseguire un programma scritto in C#, è sufficiente digitare nel relativo ambiente di esecuzione il nome del file eseguibile. Lo stesso file eseguibile, in un ambiente dotato di GUI, può invece essere caricato ed eseguito facendo doppio clic sulla sua icona.
- *Test e debug*. Questa è la fase in cui si verifica la correttezza funzionale del programma in esecuzione; se presenta errori o comportamenti non pertinenti con la logica che avrebbe dovuto seguire. A tal fine esistono appositi programmi chiamati *debugger*, che consentono di analizzare il flusso di esecuzione di un programma *step by step*, fermare la sua esecuzione al raggiungimento di un determinato *breakpoint* per esaminarne il corrente stato, rilevare il valore delle variabili in un certo momento e così via.

NOTA

L'Appendice A *Ambienti di sviluppo integrato* contiene un breve tutorial introduttivo sull'utilizzo del debugger integrato nell'IDE Visual Studio e nell'IDE MonoDevelop.

- *Mantenimento*. Questa è la fase in cui un programma che è in produzione, a seguito di richieste od osservazioni degli utilizzatori, può subire modifiche migliorative, per esempio in termini di prestazioni, oppure modifiche integrative che riguardano l'aggiunta di moduli che offrono funzionalità supplementari rispetto a quelle previste in origine.

Il primo programma in C#

Vediamo, attraverso la disamina del Listato 1.1, quali sono gli elementi basilari per strutturare e scrivere un programma C#, ravvisando, comunque, che tutte le informazioni didattiche saranno necessariamente introduttive e, giocoforza, non dettagliate.

Le funzionalità impiegate saranno trattate con dovizia di particolari nei capitoli di pertinenza. Abbiamo, in questa fase, inteso illustrare una struttura di massima degli elementi costitutivi di un programma in C# e fornire una terminologia di base applicabile ai principali costrutti del linguaggio.

Listato 1.1 PrimoProgramma.cs (PrimoProgramma).

```
using System;

namespace LibroCSharp.Capitolo1
{
    /*
    Primo programma in C#
    */
    class PrimoProgramma
    {
        private static int counter = 10;

        private const int multiplicand = 10;
        private const int multiplier = 20;

        static void Main(string[] args)
        {
            // dichiarazione e inizializzazione contestuale
            // di più variabili di diverso tipo
            string text_1 = "Primo programma in C#:",
                   text_2 = " Buon divertimento!";
            int a = 10, b = 20;

            float f;   // dichiarazione
            f = 44.5f; // inizializzazione

            // stampa qualcosa...
            Console.WriteLine("{0}{1}", text_1, text_2);
            Console.WriteLine("Stamperò un test condizionale tra a={0} e b={1}:", a, b);

            if (a < b)   // se a < b stampa quello che segue...
            {
                Console.Write("a < b VERO!");
            }
            else        /* altrimenti stampa quest'altra stringa */
            {
                Console.Write("a > b VERO!");
            }

            Console.Write("\nStamperò un ciclo iterativo, dove leggerò ");
            Console.WriteLine("per 10 volte il valore di a");

            /*
             * ciclo for
            */
            for (int i = 0; i < 10; i++)
```

```
            {
                Console.Write("Passo {0} ", i);
                Console.WriteLine("--> a={0}", a);
            }

            /*
                // esecuzione della moltiplicazione
            */
            Console.WriteLine("Ora eseguirò una moltiplicazione tra {0} e {1}",
                              multiplicand, multiplier);
            int res = mult(multiplicand, multiplier); // invocazione di un metodo
            Console.WriteLine("Il risultato di {0} x {1} è: {2}",
                              multiplicand, multiplier, res);
        }

        /**************************************
         * Metodo: mult                       *
         * Scopo: moltiplicazione di due valori *
         * Parametri: a, b -> int             *
         * Restituisce: int                   *
         **************************************/
        private static int mult(int a, int b)
        {
            return a * b;
        }
    }
}
```

Il programma del Listato 1.1 inizia con una direttiva espressa tramite la *keyword* `using`, che consente di utilizzare tutti i tipi appartenenti al *namespace* `System` senza la necessità di referenziarli o qualificarli in modo completo, ossia senza l'obbligo di far precedere al loro nome la parola `System` e il carattere punto; per esempio, è possibile referenziare la classe `Console`, appartenente al namespace `System`, senza scrivere `System.Console`, ma solo `Console`. In sostanza l'utilizzo di una direttiva `using` con un appropriato namespace fa sì che tutti i tipi contenuti nel dato namespace siano resi disponibili per un uso "facilitato" e meno "prolisso" nella corrente applicazione. Infatti, è bene tenere presente che l'utilizzo della direttiva `using` è opzionale, ossia si può sempre decidere di referenziare un tipo in modo qualificato, anteponendogli sempre il namespace di appartenenza.
La direttiva `using`, come visto, è in stretta connessione con l'astrazione namespace (*spazio dei nomi*), la quale rappresenta, in definitiva, una sorta di "contenitore" o "ambito" al cui interno sono raggruppati dei tipi logicamente correlati. Per esempio, il namespace `System` contiene i tipi fondamentali del linguaggio, classi di uso comune e così via; il namespace `System.IO`, contiene tipi per il supporto di operazioni di input e output; il namespace `System.Threading`, contiene tipi per il supporto alla programmazione concorrente e così via. Un namespace, comunque, non è solo importante per raggruppare logicamente tipi correlati, ma anche perché consente di evitare una possibile *collisione* tra tipi che possono avere lo stesso nome; infatti, dal punto di vista di un CLR, un nome di un tipo non sarà mai referenziato in modo non qualificato (per esempio, `Console`) ma sempre in modo qualificato (per esempio, `System.Console`, se scegliamo di usare il tipo `Console` del namespace `System` fornito dalla libreria FCL di Microsoft; `A.Console`, se scegliamo di usare il

tipo `Console` del namespace `A` fornito da una libreria di *terze parti*; `B.Console`, se scegliamo di usare il tipo `Console` del namespace `B` fornito da una libreria di *terze parti* e così via).
Ciò detto, per il nostro programma creiamo un apposito namespace utilizzando la *keyword* `namespace` cui facciamo seguire il relativo nome (`LibroCSharp.Capitolo1`) e una coppia di parentesi graffe che delimiteranno l'ambito entro il quale saranno posti tutti i tipi correlati al predetto namespace; infatti, tutti i tipi lì dichiarati, come nel nostro caso la classe `PrimoProgramma`, faranno per l'appunto parte del namespace `LibroCSharp.Capitolo1`.
Notiamo poi la definizione di un'istruzione di *commento*: tra il carattere di inizio commento `/*` e il carattere di fine commento `*/`, viene specificato del testo che verrà ignorato dal compilatore e che serve a documentare o chiarire specifiche parti del codice sorgente.
Il testo di questo tipo di commento può anche essere suddiviso su più righe e, infatti, per tale ragione è spesso anche definito come commento *multiriga* (*multiline comment*) oppure, in accordo con la specifica terminologia del linguaggio C#, come commento *delimitato* (*delimited comment*).
In ogni caso non è possibile innestare commenti multiriga all'interno di altri commenti (Snippet 1.1) e ciò perché il marker di chiusura `*/` del commento innestato finisce per "chiudere" il marker di commento iniziale `/*`. In questo modo, quindi, il secondo marker di chiusura commento `*/` rimarrebbe senza un marker di apertura commento con cui "accoppiarsi", inducendo il compilatore a generare un errore del tipo `CS0103 Il nome 'CCCC' non esiste nel contesto corrente`.

Snippet 1.1 Commenti multiriga innestati.

```
...
class Snippet_1_1
{
    static void Main(string[] args)
    {
        /* // commento che innesta
            AAAA
            /* // commento innestato
                BBB
            */
        CCCC
        */
    }
}
...
```

Infine, i commenti multiriga possono essere anche scritti di fianco, a lato di una porzione di codice (*winged comment*), come mostra quello definito dopo l'istruzione `else`, così come essere scritti in *forma* di riquadro di contenimento (*boxed comment*), come mostra quello scritto prima della definizione della funzione `mult`, oppure scrivendo degli asterischi `*` per ogni riga di separazione, come mostra quello posto prima del ciclo `for`.
In ogni caso, il programmatore è libero di scegliere per i commenti la formattazione che più gli aggrada, a condizione, però, che via sia sempre una corrispondenza tra il marcatore di apertura commento `/*` e il marcatore di chiusura commento `*/`.
In C# si può comunque utilizzare anche un secondo tipo di commento, che è indicato tramite l'utilizzo di due caratteri slash `//` cui si fa seguire il testo di commento. Questo

commento, poiché termina automaticamente alla fine della riga, è definito commento *a singola riga* (*single-line-comment*) e ha la caratteristica che può essere anche innestato all'interno di un commento multiriga, come mostra in modo evidente il commento `// esecuzione della moltiplicazione` posto prima dell'invocazione del metodo `Console.WriteLine`. In più, anche con questo tipo di commento, è possibile scrivere commenti multiriga semplicemente scrivendo su ogni riga i caratteri `//` e la porzione di testo di commento. Un esempio di quanto detto è visibile nei primi due commenti, posti subito dopo la parentesi graffa di apertura della funzione `Main`.

IMPORTANTE

Nel Capitolo 15, *Documentazione del codice sorgente*, vedremo che è anche possibile usare un'altra tipologia di commenti, definiti come *documentation comments* (commenti di documentazione), che consentono di documentare il codice sorgente secondo una "sintassi" che fa uso di tag XML.

Dopo il commento che segue la definizione del namespace abbiamo la definizione di una *classe*, effettuata tramite la *keyword* `class`; la classe è denominata `PrimoProgramma` e tra le parentesi graffe aperta e chiusa sono indicati i suoi membri (dati e metodi).
Una classe, ricordiamo, è il *blocco costitutivo* di un qualsiasi programma che segua il paradigma della programmazione orientata agli oggetti; essa definisce una sorta di modello, di tipo, per delle "entità", definite come *oggetti*, che ne rappresentano specifiche e concrete *istanze* o *esempi*; possiamo così dire che mentre, per esempio, una classe `Car` può rappresentare un modello che astrae una generica "macchina", un oggetto `ford` può rappresentare un'istanza concreta, un esempio specifico di quella generica macchina.
La nostra classe `PrimoProgramma` ha dunque tre membri dati, denominati `counter`, `multiplicand` e `multiplier` e due membri funzione denominati `Main` e `mult`.
Il membro dati `counter` è una *variabile*, ovvero una locazione di memoria che può contenere valori che possono cambiare nel tempo. A ogni variabile deve essere associato l'insieme di valori che essa può contenere, tramite la specificazione di un tipo di dato di appartenenza. La variabile `counter` ha infatti associato, tramite la *keyword* `int`, un tipo di dato intero, ovvero potrà solo contenere valori propri del sottoinsieme matematico dei numeri interi (per esempio, `-220`, `45600` e così via).
I membri dati `multiplicand` e `multiplier` rappresentano, invece, delle *costanti* ossia delle locazioni di memoria che contengono valori che non possono cambiare nel tempo; in pratica dopo aver assegnato un valore a una costante, questa potrà essere usata nell'ambito del programma solo in modalità "lettura" (per ottenerne cioè il valore) ma non in modalità "scrittura" (per alterarne cioè il valore).
Il membro funzione `Main`, invece, è il *metodo* principale di un qualsiasi programma in C# laddove per metodo si intende un blocco di codice contenente una o più istruzioni che rappresentano una funzionalità ed eseguono un compito specifico; un metodo `Main` deve essere sempre presente nell'ambito di un programma, perché ne rappresenta l'*entry point* ossia il "punto di ingresso", attraverso il quale viene eseguito.
In pratica il metodo `Main` viene invocato automaticamente dall'ambiente di esecuzione quando il programma viene avviato (*application startup*); poi attraverso il metodo `Main` vengono eseguite le istruzioni in esso contenute e invocati gli altri metodi indicati; da questo punto di vista, possiamo asserire che un programma in C# "deve" essere sempre composto da almeno una classe che contiene il metodo `Main` e "può" definire o utilizzare

una o più classi ausiliarie. Quindi non è altro che una *collezione* di una o più classi, tra di loro interagenti.
Quest'importante e fondamentale metodo può essere definito utilizzando una delle seguenti modalità (Snippet 1.2, 1.3, 1.4 e 1.5).

- Senza restituire un valore e senza parametri formali.

Snippet 1.2 Modalità di definizione di Main; prima definizione.

```
...
class Snippet_1_2
{
    static void Main() { /* ... */ }
}
...
```

- Senza restituire un valore e con un parametro formale di tipo array di stringhe.

Snippet 1.3 Modalità di definizione di Main; seconda definizione.

```
...
class Snippet_1_3
{
    static void Main(string[] args) { /* ... */ }
}
...
```

- Per restituire un valore di tipo intero e senza parametri formali.

Snippet 1.4 Modalità di definizione di Main; terza definizione.

```
...
class Snippet_1_4
{
    static int Main() { return 0; }
}
...
```

- Per restituire un valore di tipo intero e con un parametro formale di tipo array di stringhe.

Snippet 1.5 Modalità di definizione di Main; quarta definizione.

```
...
class Snippet_1_5
{
    static int Main(string[] args) { return 0; }
}
...
```

In pratica quando si utilizzano la prima e la terza definizione si indica che `Main` non desidera accettare ed elaborare gli argomenti forniti dalla riga di comando, mentre quando

si utilizzano la seconda e la quarta definizione si indica che `Main` ha la volontà di accettare ed elaborare gli argomenti forniti dalla relativa shell.
In linea generale possiamo anticipare la nozione che ogni metodo può avere zero o più *parametri*, che rappresentano, quando presenti, delle variabili che saranno riempite con valori (detti *argomenti*) passati al metodo all'atto della sua invocazione.
Ai parametri di un metodo deve, inoltre, essere associato un tipo di dato; infatti, il parametro `args` è dichiarato come di tipo array di stringhe (`string[]`).
Il parametro `args`, dunque, permette al metodo `Main` di ottenere in input gli argomenti eventualmente passati quando si invoca dalla riga di comando il programma che lo contiene.

NOTA

Il nome del parametro formale di `Main` è arbitrario. È pertanto possibile utilizzare un nome diverso da `args`, ma il suo tipo deve sempre essere un array di stringhe (`string[]`).

Ancora, la prima e la seconda definizione esprimono il desiderio di non voler restituire alcun valore (*termination status code*) all'ambiente di esecuzione quando il programma terminerà (*application termination*); in ogni caso all'ambiente di esecuzione sarà restituito in automatico come codice di stato il valore `0` in tre casi: quando il tipo restituito dal metodo `Main` sarà `void`, quando il flusso di esecuzione del programma raggiungerà la parentesi graffa di chiusura `}` che termina il `Main` e quando sarà eseguita un'istruzione `return` senza alcuna espressione.
La terza e quarta definizione esprimono il desiderio di restituire un esplicito valore di tipo intero, che indica all'ambiente di esecuzione un codice di stato di terminazione atto a comunicare una "situazione" di *successo* o *fallimento* dell'elaborazione delle istruzioni del programma.

NOTA

Nella possibilità di far restituire al metodo `Main` un codice di stato di tipo `int` è, con ogni probabilità ravvisabile la volontà di replicare il comportamento delle applicazioni scritte in linguaggio C, dove, tipicamente, all'ambiente di esecuzione si restituisce il valore `0` (identificato dalla macro `EXIT_SUCESS`) per esprimere la terminazione di un programma avvenuta con successo o il valore 1 (identificato dalla macro `EXIT_FAILURE`) per esprimere la terminazione di un programma avvenuta con fallimento.

Infine, indipendentemente dalla modalità di definizione scelta per `Main`, tutte utilizzano la *keyword* `static`, un *modificatore* che stabilisce che tale metodo è un *membro statico di una classe*, piuttosto che un *membro di istanza di un oggetto* e dunque può essere invocato senza creare il relativo oggetto; questa *keyword* è quindi importante, perché permette al metodo `Main` di essere invocato direttamente dall'ambiente di esecuzione e di avviarne il relativo programma.

IMPORTANTE

Di default, se un membro di una classe non ha indicato alcun modificatore di accesso, verrà utilizzato `private`. Questo è quanto avviene per Main. Ciò può far sorgere una domanda. *Se* `Main` *è privato, come fa l'ambiente di esecuzione (il CLR) a invocarlo?* Dal punto di vista del CLR un modificatore di accesso non ha alcun significato e infatti per il CLR ciò che conta è

una particolare direttiva IL, `.entrypoint`, che stabilisce il metodo che rappresenta il punto di ingresso per un'applicazione. In ogni caso, se ricorrono particolari esigenze, è possibile applicare al `Main` il modificatore `public`, al fine di renderlo direttamente utilizzabile da eventuali client (programmi) esterni che hanno necessità di usare le funzionalità della classe nella quale è stato definito.

Abbiamo poi la definizione del membro funzione `mult`, la quale evidenzia come esso restituisca un tipo di dato intero e accetti due argomenti sempre di tipo intero. Nell'ambito del metodo `Main` notiamo, infatti, come il metodo `mult` sia invocato con due argomenti di tipo intero (i valori `10` e `20`) e restituisca un valore di tipo intero assegnato alla variabile `res`. Lo stesso metodo `mult` ha, infatti, una sua implementazione algoritmica che evidenzia come restituisca un valore di tipo intero che è il risultato della moltiplicazione tra i parametri `a` e `b`, sempre di tipo intero.
Per quanto attiene al contenuto del metodo `Main`, notiamo subito una serie di istruzioni che definiscono delle variabili che rappresentano locazioni di memoria modificabili, deputate a contenere un valore di un determinato tipo di dato.
Così gli identificatori `text_1` e `text_2` indicano variabili che possono contenere *caratteri*, gli identificatori `a`, `b` e `res` indicano variabili che possono contenere numeri interi, l'identificatore `f` indica una variabile che può contenere numeri *decimali*, ossia numeri formati da una parte intera e una parte frazionaria separati da un determinato carattere (per il C# tale carattere è il punto).
Una variabile, in linea generale, può essere prima dichiarata e poi inizializzata (è il caso della variabile `f`) oppure può essere dichiarata e inizializzata contestualmente in un'unica istruzione (è il caso delle altre variabili).
A parte le varie istruzioni di stampa su console dei valori espressi dalle rispettive stringhe dei metodi `WriteLine` e `Write` del tipo `Console` notiamo: l'impiego di un'istruzione di selezione doppia `if/else` che valuta se una data espressione è vera o falsa eseguendone, a seconda del risultato, il codice corrispondente; l'impiego di un'istruzione di iterazione `for` che consente di eseguire ciclicamente una serie di istruzioni finché una data espressione è vera.
Pertanto l'istruzione `if/else` valuta se il valore della variabile `a` è minore del valore della variabile `b` e, nel caso, stampa la relativa stringa; altrimenti, in caso di valutazione falsa, stampa l'altra stringa. L'istruzione `for`, invece, stampa su console, per 10 volte informazioni sul valore delle variabili `i` e `a`.

NOTA

In un linguaggio di programmazione a oggetti, come C#, ogni membro funzione "appartiene" alla classe nella quale è stato definito e pertanto per poterlo invocare è sempre necessario usare una particolare sintassi che prevede: se è di tipo *metodo di classe*, l'uso del nome della sua classe, l'operatore punto e il suo nome; se è di tipo *metodo di istanza*, l'uso del nome dell'oggetto istanza della sua classe, l'operatore punto e il suo nome.

Un "assaggio" del metodo WriteLine

Il metodo `WriteLine`, definito nella classe `Console`, la quale è definita nel namespace `System`, il quale è contenuto nell'assembly `mscorlib`, consente di visualizzare sullo standard output (generalmente sul display video) il letterale stringa fornitogli come argomento.

NOTA

`mscorlib.dll` è un assembly di notevole importanza, in quanto contiene namespace e tipi fondamentali per il funzionamento di un qualsiasi programma C#. È talmente essenziale che è referenziato in automatico dal compilatore C#.

NOTA

In origine l'acronimo `mscorlib` stava per *Microsoft's Common Object Runtime Library*, ma poi, dopo il processo di standardizzazione della CLI attuato da ECMA, ha assunto il significato più "generico" di *Multilanguage Standard Common Object Runtime Library*.

Esistono diverse definizioni in *overloading* del metodo `WriteLine`, ma per ora accenneremo solo quella che consente di specificare i seguenti argomenti.

- *Come primo argomento* un letterale stringa definito come stringa di formato composto (*composite format string*). Questa stringa di formato composto è costituita da zero o più istanze di testo fisso (*fixed test*), alternate a uno o più elementi di formato (*format items*), scritti secondo una particolare sintassi che fa uso delle parentesi graffe. Questi elementi di formato rappresentano in sostanza una sorta di segnaposto indicizzati (*indexed placeholders*) ossia "locazioni" all'interno della stringa di formato composto il cui contenuto sarà sostituito, a *runtime*, da un valore che è una rappresentazione di tipo stringa dell'oggetto passato al metodo come argomento successivo.
- *Come argomenti successivi* un elenco di oggetti.

Così nel caso dell'istruzione `Console.WriteLine("Stamperò un test condizionale tra a={0} e b={1}:", a, b);` vista nel Listato 1.1 possiamo dire che l'argomento `"Stamperò un test condizionale tra a={0} e b={1}:"` rappresenta la stringa di formato composto, mentre `a` e `b` rappresentano gli argomenti successivi, ossia gli elenchi di oggetti da formattare.
Nell'ambito della stringa di formato composto avremo che: `Stamperò un test condizionale tra a=` e `b=`, `:`, rappresentano il testo fisso; `{0}` e `{1}` rappresentano gli elementi di formato, laddove i numeri `0` e `1` scritti all'interno delle parentesi graffe sono degli indici, denominati identificatori di parametro (*parameter specifier*), che individuano, rispettivamente, il primo e il secondo oggetto dell'elenco da formattare.
Così, a *runtime*, la stringa `"Stamperò un test condizionale tra a={0} e b={1}:"` sarà visualizzata nel seguente modo: `Stamperò un test condizionale tra a=10 e b=20:` dove appare evidente come il segnaposto `{0}` sia stato sostituito dalla rappresentazione stringa del valore intero proprio della variabile `a` (il primo oggetto dell'elenco) mentre il segnaposto `{1}` è stato sostituito dalla rappresentazione stringa del valore intero proprio della variabile `b` (il secondo oggetto dell'elenco).

TERMINOLOGIA

L'*overloading* (sovraccarico), trattato in modo dettagliato nel Capitolo 6, *Metodi*, è una caratteristica implementativa che consente di definire metodi con lo stesso nome, ma con una diversa *segnatura*. Per segnatura di un metodo si intende quell'insieme di informazioni che lo identificano univocamente fra gli altri metodi della sua stessa classe di appartenenza. Tali informazioni includono il suo nome, il numero, il tipo e l'ordine dei suoi parametri e mai il tipo del valore da esso restituito.

NOTA

Il metodo `WriteLine` avanza in automatico sulla successiva riga di output al termine della visualizzazione della stringa. Si differenzia dal metodo `Write`, sempre utilizzato nel Listato 1.1, perché quest'ultimo non avanza alla riga di output successiva al termine della sua elaborazione. Il metodo `Write` è comunque anch'esso definito nella classe `Console`.

Elementi strutturali di un programma in C#

Un programma in C# è costituito da uno o più file di codice sorgente definiti in modo rigoroso e formale come unità di compilazione (*compilation unit*).
Ogni unità di compilazione è modellata, idealmente, in più parti strutturali e semantiche che sono combinate insieme con lo scopo di formare un programma.
Abbiamo cinque elementi di base che formano la *struttura lessicale* di un file sorgente C#: terminatori di riga, spazi, commenti, token e direttive di pre-elaborazione.
Di questi, solo i token hanno un impatto significativo sulla *struttura sintattica* di un programma C#; infatti, i terminatori di riga, gli spazi e i commenti sono impiegati per separare i token, mentre le direttive di pre-elaborazione sono impiegate, tipicamente, per "saltare", ignorare, determinate sezioni di un file di codice sorgente.
Formano i token gli identificatori, le *keyword*, i letterali (booleani, interi, reali, carattere, stringa e `null`), gli operatori e i segni di punteggiatura.
Per C# il numero di caratteri di spaziatura (spazi, tabulazione, invio e così via) impiegati come separatori tra i token è libero: ogni programmatore può scegliere quello che più gli aggrada, secondo il suo personale stile di scrittura. Tuttavia un token non può essere "diviso" senza causare un errore e cambiarne la semantica. Lo stesso vale per un letterale stringa, con il seguente distinguo: al suo interno è sempre possibile inserire dei caratteri di spazio, ma è un errore separarlo all'interno dell'editor su più righe premendo il tasto Invio.
A partire dagli elementi lessicali si costruiscono le *statement* ossia le istruzioni proprie di un programma C# che sono rappresentate dalle direttive di pre-elaborazione, dalle dichiarazioni, dalle espressioni, dalle selezioni, dalle iterazioni e così via (Figura 1.14).
Quanto alle istruzioni, esse rappresentano azioni o comandi che devono essere eseguiti durante l'esecuzione di un programma.
Infine, per C#, ogni istruzione singola (*single-line statement*) deve terminare con il carattere punto e virgola; due o più istruzioni (*multi-line statement*) possono essere raggruppate insieme a formare un'unica entità sintattica, definita blocco (*statement block*), se incluse tra parentesi graffe, aperta e chiusa.

NOTA

A volte nel codice sorgente si può rilevare la scrittura di una sola istruzione posta in un blocco di istruzioni. Ciò non è sintatticamente errato e viene fatto, comunemente, per ragioni di stile e indentazione.

La Tabella 1.1 mostra tutte le *keyword* del linguaggio, aggiornate alla versione 6.0 di C#; la Tabella 1.2 mostra i segni di punteggiatura e gli operatori utilizzabili; lo Snippet 1.6 evidenzia alcune *keyword*, operatori, identificatori, letterali e così via.

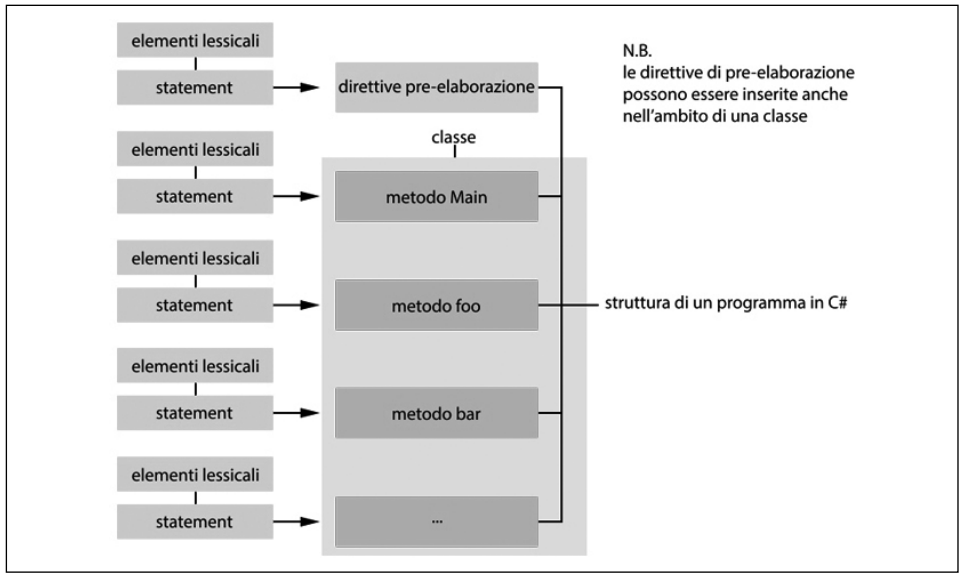

Figura 1.14 Costituzione strutturale di un generico programma in C#.

Tabella 1.1 Keyword del linguaggio C#.

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

Tabella 1.2 Segni di punteggiatura e operatori.

{	}	[	]	(	)	.	,	:	;
+	-	*	/	%	&	|	^	!	~
=	<	>	?	??	::	++	--	&&	||
->	==	!=	<=	>=	+=	-=	*=	/=	%=
&=	|=	^=	<<	<<=	>>	>>=	=>		

TERMINOLOGIA

Una *keyword* (*parola chiave*) è una sequenza di caratteri riservata, che non è permesso impiegare come identificatore a meno che sia preceduta dal simbolo @.

TERMINOLOGIA

Gli operatori sono simboli utilizzati nell'ambito di espressioni, atti a descrivere operazioni a uno o più operandi. Così, un'espressione come `x * z` utilizza l'operatore * per moltiplicare l'operando `x` per l'operando `z`. I segni di punteggiatura, invece, sono impiegati per eseguire raggruppamenti e separazioni. Ne elenchiamo alcuni rispetto alla Tabella 1.2: `{ } [ ] ( ) . , : ;`. Per esempio, i segni `{ }` servono per raggruppare le istruzioni in un blocco; il segno `,` serve per separare più variabili in una singola dichiarazione e così via.

Snippet 1.6 Esempi di keyword, operatori, identificatori, letterali e così via.

```
...
class Snippet_1_6
{
    // static, void, string -> keyword
    // Main, args           -> identificatori
    // []                   -> operatore
    // ()                   -> segno di punteggiatura di raggruppamento
    static void Main(string[] args)
    // {                    -> segno di punteggiatura di raggruppamento
    {
        // int                   -> keyword
        // number, temp, status  -> identificatori
        // ,                     -> segno di punteggiatura di separazione
        // ;                     -> segno di punteggiatura di separazione
        int number, temp, status;

        // int, float, char -> keyword
        // a, f, c          -> identificatori
        // =                -> operatore
        // 100              -> letterale intero
        // 120.78f          -> letterale reale
        // 'A'              -> letterale carattere
        // ;                -> segno di punteggiatura di separazione
        int a = 100;
        float f = 120.78f;
        char c = 'A';

        // string       -> keyword
        // name         -> identificatore
        // =            -> operatore
        // "Pellegrino" -> letterale stringa
        // ;            -> segno di punteggiatura di separazione
        string name = "Pellegrino";
        // }            -> segno di punteggiatura di raggruppamento
    }
}
...
```

ATTENZIONE

Alcuni simboli, a seconda del contesto, possono essere trattati come segni di punteggiatura oppure come operatori. Per esempio, le parentesi tonde quando sono usate nell'ambito della definizione di un metodo agiscono come segni di punteggiatura, per raggruppare i parametri; ma quando sono impiegate nell'ambito di utilizzo di un metodo agiscono come un operatore di invocazione.

Compilazione ed esecuzione del codice

Dopo aver scritto il programma del Listato 1.1, con un qualunque editor di testo o con un IDE di preferenza (per noi l'IDE sarà Visual Studio oppure MonoDevelop), vediamo come eseguirne la compilazione che, lo ricordiamo, è quel procedimento mediante il quale un compilatore C# legge un file sorgente (nel nostro caso `PrimoProgramma.cs`) e lo trasforma in un file (per esempio `PrimoProgramma.exe`) che conterrà istruzioni scritte nel linguaggio IL pronte per essere elaborate da un CLR, tramite un apposito compilatore Just-In-Time, che le trasformerà in linguaggio macchina del sistema hardware di riferimento.

NOTA

Per un dettaglio sul modo in cui eseguire la compilazione ed esecuzione di un programma C#, sia in ambiente GNU/Linux che in ambiente Windows o OS X, ma con l'ausilio di un IDE (Visual Studio o MonoDevelop) consultare l'Appendice A, *Ambienti integrati di sviluppo*.

Prima di vedere come utilizzare manualmente un compilatore C# (Shell 1.1 e 1.2) verifichiamo che esso sia disponibile da una shell.

- In ambiente GNU/Linux o OS X digitiamo da una shell il comando `mcs --version` e verifichiamo che appaia qualcosa come `Mono C# compiler version 4.0.3.0`. In caso contrario verifichiamo di aver installato correttamente Mono (consultare a tal fine l'Appendice A, *Ambienti integrati di sviluppo*).
- In ambiente Windows apriamo il prompt dei comandi per gli sviluppatori (in Windows 10 premiamo il tasto *Windows* e digitiamo la sequenza di caratteri *prompt dei comandi per...* finché nell'area dei risultati della ricerca dal menu *Start* non apparirà la scritta *Prompt dei comandi per gli sviluppatori per VS2015* e facciamo clic qui per aprirlo); digitiamo poi `csc -?` e verifichiamo che appaia una lista delle opzioni di compilazione. In caso contrario verifichiamo di aver installato correttamente Visual Studio (consultare a tal fine l'Appendice A, *Ambienti integrati di sviluppo*) il quale installa, se necessario, anche il .NET Framework SDK (*Software Development Kit*) che contiene tutti gli strumenti essenziali per la creazione di applicazioni .NET (compilatori, utility varie, codice di esempio, librerie, documentazione e così via). In ogni caso è comunque possibile installare separatamente il .NET Framework SDK, se per esempio si sceglie di non usare Visual Studio, compiendo le seguenti azioni: puntare il browser all'URL `https://dev.windows.com/it-it/downloads`; scorrere la pagina risultante finché non appare il link *Scarica la versione autonoma di Windows SDK per Windows 10*; fare clic sul link e confermare il download del file `sdksetup`.

exe; avviare quindi l'installazione dell'SDK dal file `sdksetup.exe` seguendo le istruzioni che compariranno a video.

NOTA

Il Windows Software Development Kit (SDK) per Windows 10 contiene librerie e tool essenziali per lo sviluppo di applicazioni compatibili con i sistemi operativi Windows (Windows 10, Windows 8.1, Windows 8, Windows 7, Windows Vista, Windows Server 2012, Windows Server 2008 R2 e Windows Server 2008). Tra la moltitudine degli strumenti di sviluppo presenti è incluso il .NET Framework 4.6 SDK.

Shell 1.1 Invocazione del comando di compilazione (GNU/Linux e OS X).

```
[thp@localhost MY_C#_SOURCES]$ mcs PrimoProgramma.cs -out:$HOME/MY_C#_EXE/PrimoProgramma.exe
```

Shell 1.2 Invocazione del comando di compilazione (Windows).

```
C:\MY_C#_SOURCES> csc PrimoProgramma.cs -out:\MY_C#_EXE\PrimoProgramma.exe
```

NOTA

L'opzione di compilazione `-out` è utilizzata per specificare il nome dell'assembly da creare. Se si omette, il compilatore creerà in automatico nella directory corrente un file con lo stesso nome del file sorgente che contiene il metodo `Main` e con estensione `.exe`.

Dopo la fase di compilazione possiamo avviare, eseguire, il file prodotto (Shell 1.3 e 1.4) digitando da shell il suo nome (nel nostro caso `PrimoProgramma.exe`) che ne mostrerà il relativo output (Output 1.1).

Shell 1.3 Avvio del programma (GNU/Linux e OS X).

```
[thp@localhost MY_C#_EXE]$ mono ./PrimoProgramma.exe
```

Shell 1.4 Avvio del programma (Windows).

```
C:\MY_C#_EXE>PrimoProgramma.exe
```

Output 1.1 Esecuzione di Shell 1.3 o di Shell 1.4.

```
Primo programma in C#: Buon divertimento!
Stamperò un test condizionale tra a=10 e b=20:
a < b VERO!
Stamperò un ciclo iterativo, dove leggerò per 10 volte il valore di a
Passo 0 --> a=10
Passo 1 --> a=10
Passo 2 --> a=10
Passo 3 --> a=10
Passo 4 --> a=10
Passo 5 --> a=10
Passo 6 --> a=10
Passo 7 --> a=10
Passo 8 --> a=10
Passo 9 --> a=10
```

```
Ora eseguirò una moltiplicazione tra 10 e 20
Il risultato di 10 x 20 è: 200
```

Elenchiamo, infine, altre utili e comuni opzioni di compilazione.

- `-target:exe`, dichiara che il file eseguibile sarà un programma che girerà in una console ovvero sarà di "tipo" CUI (*Console User Interface*). È un'opzione usata di default e quindi omettibile.
- `-target:winexe`, dichiara che il file eseguibile sarà un programma che girerà in una finestra, ovvero sarà di "tipo" GUI (*Graphical User Interface*).
- `-target:library`, dichiara che il file ottenuto sarà una libreria di codice, ossia una DLL. Il file avrà estensione `.dll`.
- `-reference:`*`assembly`*, dichiara una DLL da referenziare in modo che un programma possa utilizzarne i relativi tipi. Per esempio, per usare il tipo `Bitmap` appartenente al namespace `System.Drawing` ne dobbiamo referenziare il relativo assembly `System.Drawing` nel seguente modo: `-reference:System.Drawing.dll`.

> **NOTA**
>
> Il compilatore `csc` di Microsoft referenzia in automatico la maggior parte delle librerie del *framework* di uso comune, tra cui la citata `System.Drawing`. Il compilatore `mcs` di Mono, invece, referenzia in automatico solo le librerie `mscorlib.dll`, `System.dll` e `System.Xml.dll`. Se vogliamo utilizzare il tipo `Bitmap` in un nostro programma, in ambiente .NET potremo non usare l'opzione `-reference:System.Drawing.dll` e in ambiente Mono dovremo usare l'opzione `-reference:System.Drawing.dll`.

Problemi di compilazione ed esecuzione?

Elenchiamo alcuni problemi che si potrebbero incontrare durante la fase di compilazione o di esecuzione del programma appena esaminato.

- Il comando di compilazione `csc o mcs` è inesistente? Verificare che sotto Windows il .NET Framework SDK sia stato correttamente installato, così come il Mono Framework per la corrente distribuzione di GNU/Linux scelta oppure per il sistema OS X.
- Il compilatore `csc o mcs` non trova il file `PrimoProgramma.cs`? Verificare che la directory corrente sia `C:\MY_C#_SOURCES` (per Windows) oppure `~/MY_C#_SOURCES` (per GNU/Linux o OS X, laddove in quest'ultimo caso il carattere tilde ~ è sostituito dalla directory dell'utente corrente, che per noi sarà `/home/thp` oppure `/Users/thp`).
- Il file `PrimoProgramma.exe non viene trovato`? Verificare che la directory corrente sia `C:\MY_C#_EXE` (per Windows) oppure `~/MY_C#_EXE` (per GNU/Linux o OS X, laddove, anche in questo caso, il carattere tilde ~ sarà sostituito dalla directory dell'utente corrente che per noi sarà sempre `/home/thp` oppure `/Users/thp`).

La Figura 1.15 mostra in dettaglio cosa accade, dopo aver creato con un qualsiasi editor di testo un file contenente codice scritto obbedendo alla sintassi del linguaggio C#, dalla fase di compilazione fino alla fase di avvio del relativo file prodotto, che conterrà le istruzioni eseguibili del programma.

- Il compilatore C# verifica la correttezza sintattica del codice sorgente scritto all'interno di un file `.cs`. Se il processo di verifica va a buon fine, converte quindi il codice sorgente presente nel file `.cs` in un apposito codice di linguaggio intermedio IL, scrivendolo in un altro file, assembly, e avente tipicamente estensione `.exe`.
- L'ambiente di *runtime* (CLR), tramite un *class loader*, carica in memoria i tipi utilizzati nel programma e poi avvia un apposito compilatore Just-In-Time, il quale converte il codice di linguaggio intermedio IL in codice di linguaggio macchina, ossia nel codice nativo della corrente piattaforma di esecuzione.
- L'ambiente di *runtime* (CLR) manda in esecuzione il codice nativo prodotto sulla piattaforma (in pratica la CPU preleva, decodifica ed esegue le istruzioni che compongono il programma).

NOTA

Un compilatore JIT converte, traduce, a *runtime*, il codice IL in codice nativo. Tuttavia questo processo di conversione è effettuato *just in time*, ossia "appena in tempo per l'esecuzione", quando cioè essa è effettivamente necessaria. Avviene, in pratica, quanto segue: quando il flusso di esecuzione del codice raggiunge un punto in cui si invoca un metodo, il compilatore JIT ne traduce il codice IL in codice macchina e poi esegue tale codice. Successivamente, se il programma invoca di nuovo lo stesso metodo, questo non verrà ritradotto, ma verrà subito eseguito. Al contempo i metodi non invocati non vengono mai tradotti. Il processo descritto evidenzia come un compilatore JIT sia in effetti efficiente e performante, perché non traduce, in blocco, tutto il codice IL in codice macchina senza che vi sia una reale necessità. Tuttavia, in alcune circostanze, per migliorare le prestazioni di un'applicazione, si può anche usare un particolare tool del .NET Framework, `Ngen.exe` (*Native Image Generator*), che effettua una compilazione anticipata (*ahead-of-time compilation*) di un assembly, ossia ne compila tutto il codice IL in codice nativo prima di eseguire l'applicazione; questa compilazione avviene *at install time* quando cioè l'applicazione è installata su una piattaforma di destinazione.

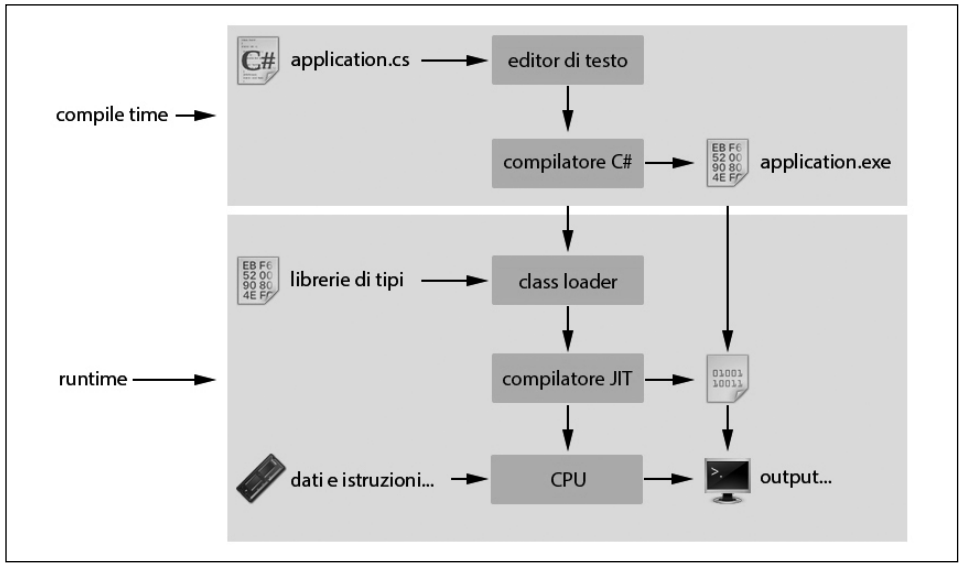

Figura 1.15 Flusso completo di generazione ed esecuzione di un programma per .NET.

Assembly: un cenno contestuale

Prima di continuare la studio del linguaggio C# appare opportuno soffermarci, seppur brevemente, su un argomento di notevole importanza, gli *assembly*, i quali rappresentano un altro "mattone" fondamentale dell'infrastruttura di .NET (gli assembly saranno comunque trattati con dettaglio nel Capitolo 13, *Assembly*).
Un assembly è in sostanza un file binario `.exe` o `.dll` prodotto da un compilatore C# durante la fase di compilazione che ha al suo interno alcuni elementi fondamentali: il codice IL; dei metadati che descrivono in grande dettaglio le caratteristiche dei tipi utilizzati; altri metadati, contenuti nel cosiddetto manifesto dell'assembly (*assembly manifest*), che descrivono l'assembly stesso (nome, numero di versione, informazioni sulla lingua, elenco di tutti i file in esso contenuti, elenco di tutti gli assembly riferiti e così via); un insieme di risorse (per esempio dei file di immagine `.jpg` o `.bmp`, un file XML e così via).
Per comprendere in modo compiuto quanto detto, "dissezioniamo" l'assembly `PrimoProgramm.exe` esplorandone le sue parti grazie al tool `ildasm.exe` (*Intermediate Language Disassembler*) fornito con l'SDK del .NET Framework.

- Apriamo il prompt dei comandi per gli sviluppatori e lanciamo il comando `ildasm.exe` il quale avvierà la relativa applicazione (Figura 1.16).

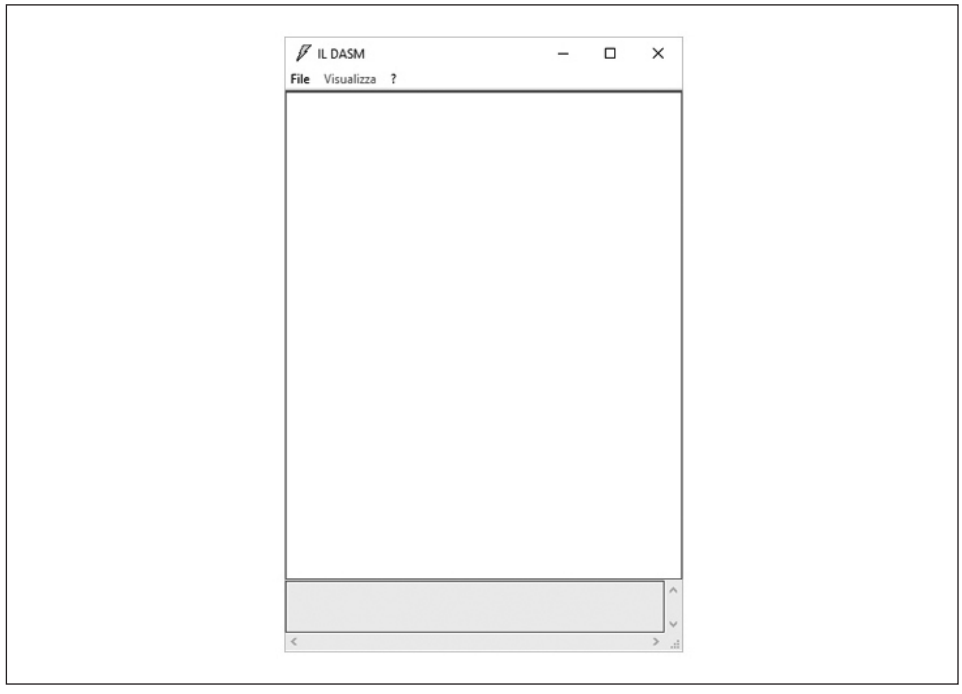

Figura 1.16 Finestra principale dell'applicazione IL Disassembler.

- Dalla finestra dell'applicazione attiviamo il menu *File > Apri* e navighiamo dove abbiamo prodotto il file `PrimoProgramma.exe` (dovrebbe trovarsi nella directory `MY_C#_EXE`), quindi selezioniamolo e apriamolo.
- Nell'area principale dell'applicazione, nell'espansione della vista ad albero di `PrimoProgramma.exe`, facciamo doppio clic sulla voce *MANIFEST* per far apparire una finestra che mostrerà i metadati propri dell'assembly (Figura 1.17).

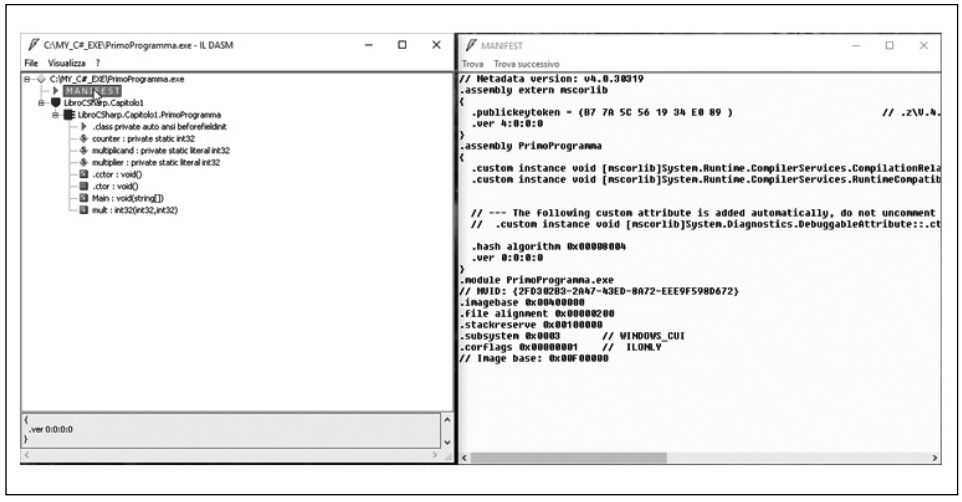

Figura 1.17 Manifest dell'assembly PrimoProgramma.exe.

- Dalla finestra attiva dell'applicazione digitiamo Ctrl+M per far apparire una finestra che mostrerà i metadati dei tipi dell'assembly (Figura 1.18).
- Nell'area principale dell'applicazione, nell'espansione della vista ad albero di `PrimoProgramma.exe`, facciamo doppio clic sulla voce *LibroCSharp.Capitolo1* e anche sulla voce *LibroCSharp.Capitolo1.PrimoProgramma*; poi facciamo doppio clic sulla voce *mult: int32(int32,int32)* per far apparire una finestra che mostrerà il codice assembly proprio dell'IL (Figura 1.19).

IMPORTANTE

È fondamentale sottolineare che il codice IL visionato nel tool `ildasm.exe` è un codice IL in formato *umanamente* leggibile, ossia è il codice espresso in linguaggio assembly proprio dell'IL. Il codice IL prodotto dal compilatore C#, invece, è un codice IL in formato non *umanamente* leggibile, ossia ne è una sua rappresentazione binaria. Quanto detto rende possibile scrivere un intero programma utilizzando solo le istruzioni del linguaggio assembly proprio dell'IL e infatti Microsoft mette a disposizione uno specifico tool, `Ilasm.exe` (*Intermediate Language Assembler*) fornito sempre con l'SDK del .NET Framework, ossia un apposito assembler che permette di generare il file eseguibile (l'assembly).

ATTENZIONE

Non confondere il termine *assembly* proprio del linguaggio di programmazione assembly con il termine *assembly* proprio di un file `.exe` o `.dll` prodotto da un compilatore C#.

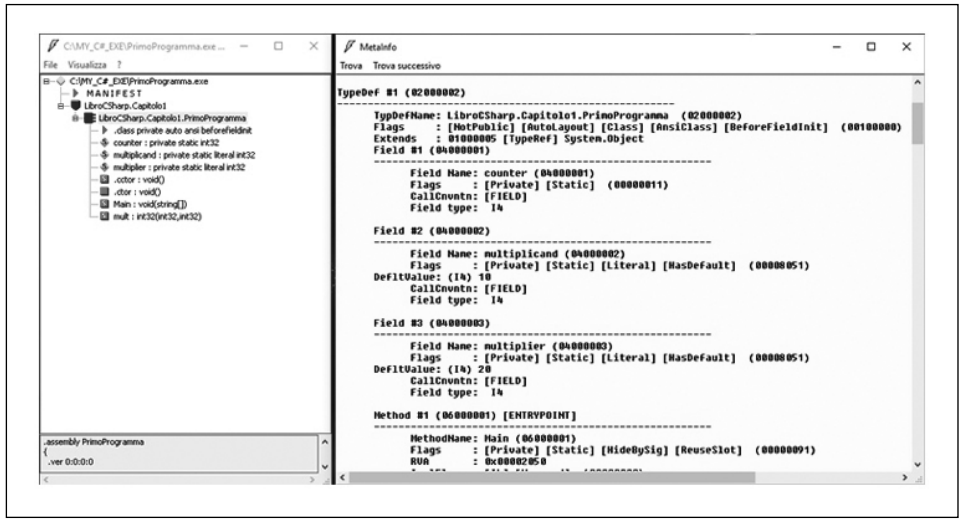

Figura 1.18 Metadati dei tipi dell'assembly PrimoProgramm.exe.

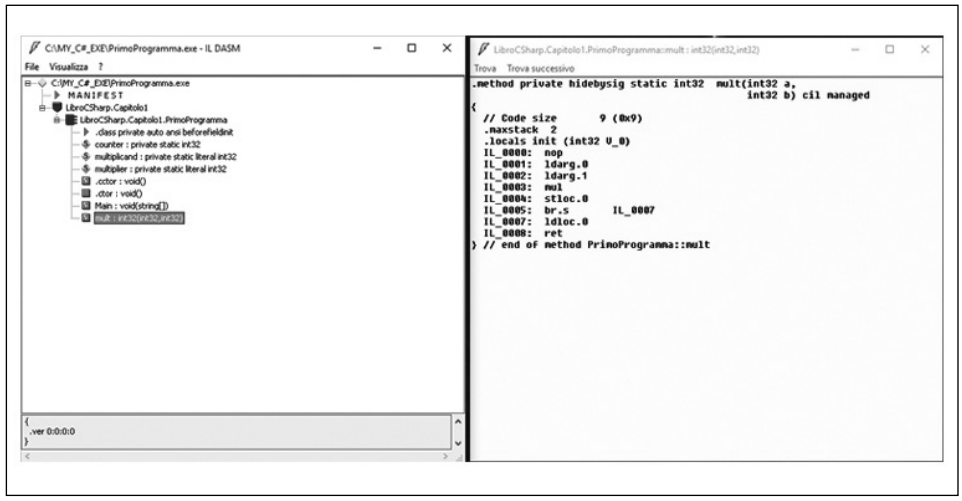

Figura 1.19 Assembly IL del metodo mult.

A questo punto, con le "nuove" e più complete informazioni in nostro possesso, possiamo rielaborare la Figura 1.15 nella Figura 1.20, evidenziando come l'`application.exe` prodotta dal compilatore C# sia di fatto un assembly con il contenuto poc'anzi discusso.

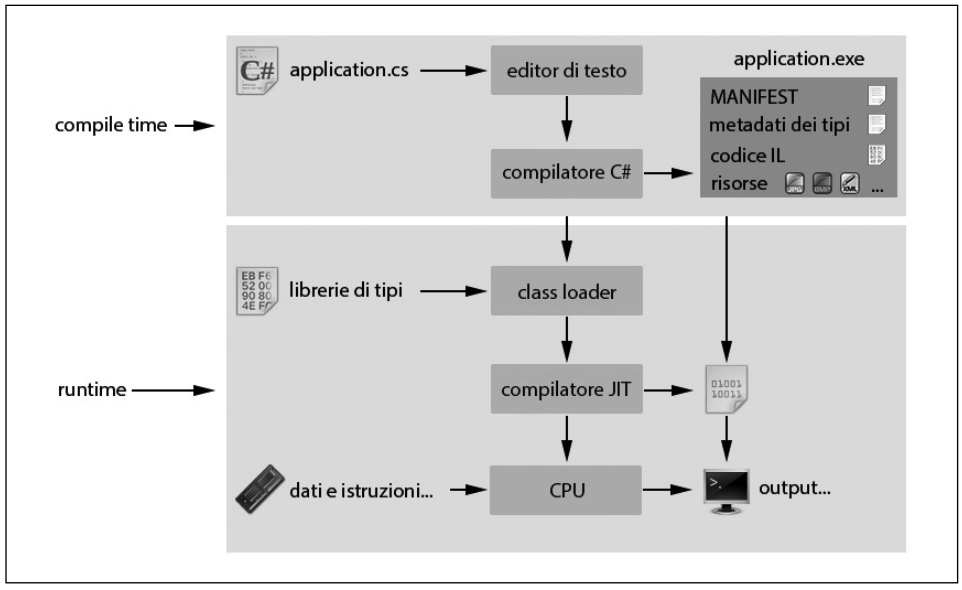

Figura 1.20 Flusso completo di generazione ed esecuzione di un programma per .NET (nuova versione).