

Dal vecchio al nuovo

CAPITOLO 2

Le novità di Visual Basic .NET potrebbero frastornare anche il più esperto programmatore; per questo motivo in questo secondo capitolo vengono passati in rassegna i cambiamenti di base della nuova suite. Si analizzeranno i tipi di dati, le dichiarazioni di variabili, gli operatori e le variazioni fondamentali di sintassi tra Visual Basic 6 e Visual Basic .NET.

Estensioni dei file

Il segnale più chiaro che le cose siano cambiate è che le estensioni dei file in VB.NET sono diverse. In VB.NET questi cambiamenti sono stati fatti per tenere conto del modo in cui i file sono gestiti nel nuovo IDE unificato. In VB6 quando si aggiungeva un secondo o un terzo progetto veniva creato un file di tipo .vbg ovvero Visual Basic Group; la stessa semplicità di lavoro con progetti multipli è stata mantenuta anche in VB.NET: il file .sln è l'equivalente del file tipo .vbg.

Nuove estensioni dei file sorgente

Un secondo grande cambiamento è che non viene più fatta distinzione tra file contenenti classi, form o solo codice. Tutto il codice sorgente è scritto in file con estensione .vb, inclusi controlli utente, componenti, form, classi, moduli, assembly e il sorgente dei Web Form. Sono state aggiunte anche ulteriori estensioni relative ad altri tipi di file. Per esempio i file dei modelli hanno estensione .mdx.



VB.NET supporta l'idioma classe e l'idioma modulo. Si pensi al modulo come a una classe speciale i cui membri sono condivisi; in alcuni linguaggi i moduli sono chiamati classi statiche o classi membro. I membri condivisi esistono a livello di metaclassa ovvero non è necessario creare un'istanza di oggetto per accedere a metodi condivisi. L'idioma modulo è supportato per rendere meno ostico l'utilizzo della suite.

Il nuovo tipo di estensione .vb è indicativo del fatto che i file sorgente sono contenitori di codice nel senso più generico della parola. La nuova sintassi, con nuovi vocaboli o idiomi, è stata

introdotta per fare distinzione tra i vari tipi di entità, come classi e moduli, permettendo la definizione di varie entità nel medesimo file.

Listato 2.1 *Più moduli definiti in un file singolo.*

```
Module Module1

    Sub Test()
        MsgBox("Module1")
    End Sub
End Module

Module Module2

    Sub Test()
        MsgBox("Module2")
    End Sub

End Module
```

Il codice nel Listato 2.1 è definito in un singolo modulo chiamato Module1.vb. Ci si potrebbe aspettare che solo il modulo1 esista nel file Module1.vb. In realtà non esiste un legame tra il nome del file e il nome del modulo in VB.NET. Una convenzione che potrebbe essere utile è quella di nominare il file col nome del modulo più importante al suo interno; questo naturalmente è solo un suggerimento che potrebbe rendere più leggibile il codice.

Come si può notare nel Listato 2.1 due metodi hanno lo stesso nome. Si potrebbe dire che module1 e Module2 incapsulano un metodo Test. Se non avete familiarità con il concetto di incapsulamento fate riferimento al Capitolo 7.

Tornando ai moduli: i moduli sono classi che condividono membri. La condivisione è spiegata con maggiori dettagli nel Capitolo 11, per adesso basti ricordare che per accedere a un membro di classe basta utilizzare come prefisso al nome del membro il nome del modulo seguito da un punto. Se si scrive la seguente linea di codice

```
Module1.Test()
```

sarà mostrata a video una finestra di messaggio con il testo “Module 1” e la linea successiva

```
Module2.Test()
```

mostrerà il testo “Module 2”. Poiché Test è definito in due moduli, chiamando Test senza il prefisso potrebbe causare errori in fase di compilazione dovuti alla dichiarazione ambigua. Generalmente si può accedere a un membro di modulo senza specificare il prefisso, nome modulo più punto, ma questo potrebbe causare un errore di compilazione.

Namespace

Un altro notevole cambiamento introdotto è l'utilizzo dei namespace. Il Capitolo 6 ne spiega il significato. Per ora si pensi ai namespace come a una sorta di organizzazione di classi.

Più un software diviene complesso più alto deve essere il livello di astrazione per organizzare il codice. I tipi di dati vengono definiti per contenere informazioni. Procedure vengono definite per organizzare e gestire le linee di codice. Le classi vengono definite per organizzare procedure e dati, ed ora abbiamo i namespace per organizzare le classi.

Quando si crea un progetto VB .NET, un namespace di default viene definito con lo stesso nome del progetto. Il namespace per il progetto corrente può essere visto nel pannello Generale all'interno della pagina delle proprietà del progetto. Per utilizzare una classe, una procedura o un oggetto definito in un namespace ci sono due vie. Scrivere il percorso completo dell'entità o aggiungere una dichiarazione Imports nella lista delle importazioni nella pagina delle proprietà.



Quando cercate informazioni nel file di aiuto, parte delle informazioni fornite è il namespace che contiene l'entità su cui chiedete informazioni. Ad esempio, se cercate il metodo Debug.WriteLine, otterrete informazioni sul namespace System.Diagnostics. Per utilizzare il codice nella diagnostica, aggiungete una dichiarazione Imports all'inizio del modulo o nella pagina delle proprietà.

Vediamo un esempio di una dichiarazione Imports per System.Diagnostics:

```
Imports System.Diagnostics
```

Dal Capitolo 1, saprete che diversi namespace di base vengono aggiunti a un tipo di progetto. Ad esempio, la nostra applicazione Windows importa System.Windows.Forms. System.Windows.Forms si riferisce comunemente a WinForms.

Riferimenti

Se aprite Esplora Soluzioni, troverete una sezione Riferimenti, semanticamente identica a quella di Visual Basic 6. I passi per l'aggiunta di un riferimento in VB .NET sono sostanzialmente simili ai passi per l'aggiunta di un riferimento in VB6.

1. Aprite Esplora soluzioni.
2. Trovate la sezione *Riferimenti del progetto*.
3. Fate clic con il pulsante destro per mostrare il menu di scelta rapida *Riferimenti* e fate clic su *Aggiungi riferimento*.

Potete aggiungere assembly .NET, oggetti COM o riferimenti a progetti dipendenti nella soluzione corrente. Nel Capitolo 1 avete visto come aggiungere un riferimento di progetto in ClassLibraryDemo.sln. Per creare applicazioni interdipendenti sviluppabili simultaneamente, accessibili le une dalle altre, si utilizza la finestra di dialogo *Aggiungi riferimento*.

Dichiarazione di opzioni

Visual Basic 6 supporta numerose specifiche di opzione:

- L'opzione `Explicit` richiede al programmatore di dichiarare tutte le variabili.

- L'opzione `Base [0 ; 1]` assegna l'indice per il limite inferiore della dimensione dei vettori.
- L'opzione `Compare [Binary|Text|Database]` indica il comportamento di default nella comparazione di stringhe.
- L'opzione `Private` indica che il codice del modulo era privato all'applicazione di hosting.

Alcune di queste specifiche di opzione sono state introdotte per la prima volta in Visual Basic.NET.

Opzione Explicit

L'opzione `Explicit` è stata rimandata in Visual Basic .NET. Potete impostare nel progetto il modo `Option Explicit` a `On` o a `Off` nella voce *Generazione* della *Pagina delle proprietà*, in modo tale da non dover definire tale proprietà localmente nei moduli.

Il valore di default a livello di progetto è `Option Explicit On`. La ragione è semplice: potrebbe creare problemi avere dichiarazioni ambigue di variabili.

I limiti di ambito sono definiti da blocchi di codice come procedure, moduli, classi e applicazioni. È possibile che nella stessa applicazione si abbiano due variabili o procedure con lo stesso nome. Senza regole di ambito, incorreremmo in conflitti di nome.

L'opzione `Explicit On` vi aiuta a evitare il caso in cui dichiarate implicitamente una variabile ed esista una variabile con lo stesso nome in un ambito più largo. Il vostro codice finirebbe con il modificare quella variabile involontariamente.

Un secondo problema in VB6: le variabili implicite erano di tipo `Variant`. I tipi di dati `Variant` non sono molto efficienti e sono stati progettati per implementare COM, non come uno strumento di programmazione generale.

La dichiarazione esplicita di una variabile significa che la variabile è dichiarata utilizzando la dichiarazione `Dim` specificando un tipo di dati. La dichiarazione implicita di una variabile avviene quando si introduce una variabile senza una dichiarazione `Dim`. Quando `Option Explicit` è `On`, si dovranno dichiarare tutte le variabili con la parola chiave `Dim` prima di utilizzarle. Modificando `Module1` del Listato 2.1, si può vedere nel Listato 2.2 cosa cambia nel comportamento del codice in base al valore della proprietà `Option Explicit`.

Listato 2.2 *Dimostrazione dell'uso di Option Explicit.*

```

1: Option Explicit Off
2:
3: Module Module1
4:
5:     Sub Test()
6:         Message = "Module1"
7:         MsgBox(Message)
8:     End Sub
9:
10: End Module

```



Vogliamo che in fase di compilazione del nostro codice ci venga mostrato qualsiasi tipo di errore. L'opzione `Explicit On` risponde a questa finalità.

Il codice nel Listato 2.2 funziona. La variabile `Message` è dichiarata implicitamente e inizializzata alla stringa "Module1". Se la linea 1 è cambiata in `Option Explicit On`, il codice nel Listato 2.2 causerà un errore di compilazione. Per dichiarare la variabile `Message` come una stringa e inizializzarla, ridefinite la linea 6 come segue:

```
Dim Message As String = "Module1"
```

Ora `Message` è dichiarata esplicitamente come una stringa ed è inizializzata. Una singola riga di codice per dichiarazione e inizializzazione è una caratteristica introdotta da Visual Basic .NET: per maggiori dettagli vedere il paragrafo "Dichiarazione di variabili".

Visual Basic .NET non permette il tipo di dati `Variant`. Con dichiarazioni implicite otterrete una variabile di tipo `Object`. Dichiarare un tipo specifico, significa permettere al compilatore di aiutarvi il più possibile. Se in effetti si ha bisogno di un tipo generico, si deve dichiarare una variabile del tipo `Object`. Nel paragrafo "Tipi di dati" saranno trattati più dettagliatamente i tipi `Object`.

Gran parte della letteratura disponibile indica che le variabili dichiarate implicitamente saranno tipi `Object`. In Visual Basic .NET tutto è un oggetto; tuttavia, attualmente, sembra che il compilatore esegua qualche tipo di conversione basato sul valore assegnato alla variabile implicitamente dichiarata. Se rivedete il Listato 2.2 e sostituite la riga di codice 7 con `MsgBox(Message.GetType() :t..Name :et.)` con `Option Explicit Off`, l'applicazione riferirà che `Message` è una variabile di tipo `String`.

Tuttavia se avete inizializzato `Message` a un valore intero, ad esempio `Message = 1` e avete richiesto informazioni sul tipo di variabile con `Option Explicit Off`, il tipo riferito sarebbe `Int32`. `GetType` fornisce informazione sul contenuto della variabile nel momento in cui è chiamato.

Opzione Compare

Il valore di default è `Option Compare Binary`. Un confronto `Binary` è anche del tipo `case-sensitive`. La scelta alternativa è `Option Compare Text`. Questa dichiarazione è settata a `Binary` a livello di progetto; può essere cambiata a livello di progetto ed essere così valida per tutti i moduli. `Option Compare Databases` non è più supportata in VB .NET.

Opzione Strict

L'opzione `Strict` è una nuova direttiva del compilatore. L'opzione `Strict Off` è scelta come default. L'impostazione da preferire è `Option Strict On`, ma questa impostazione ha alcune implicazioni sul comportamento del codice. È facile capire perché sia consigliato lo stato `Option Strict On`: segue una dimostrazione dei problemi che si potrebbero incontrare lasciando `Option Strict Off`. Per impostare `Option Strict On`, basta aprire la Pagina delle proprietà del progetto, scegliere la voce *Generazione* e cambiare l'impostazione predefinita di compilazio-

46 Capitolo 2

ne `Option Strict` in `On`.

Quando `Option Strict` è `On`, nessuna conversione implicita del tipo è permessa dal compilatore. Il Listato 2.3 mostra una procedura che prova a convertire in modo implicito un tipo `long` in `integer`.

Listato 2.3 *Codice che prova a convertire in modo implicito un long in un integer.*

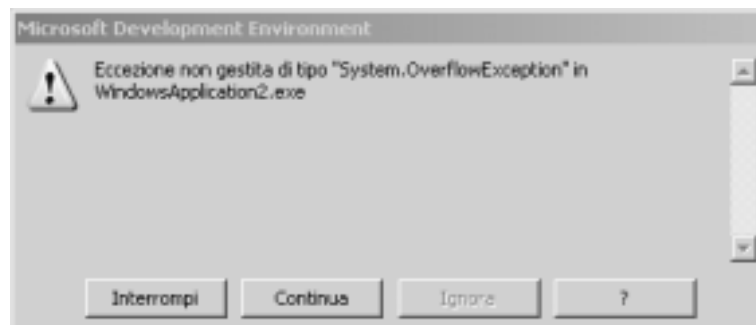
```
1: Sub TestStrictMode()
2:   Dim I As Integer = 5
3:   Dim L As Long = 100
4:   I = L
5: End Sub
```

Se `Option Strict On` è `On`, il codice in Listato 2.3 se eseguito riferirà che `Option Strict` vieta la conversione della variabile `L`, da `long` a `integer`.

Cosa avverrebbe se `Long L` contenesse un numero più grande di quanto l'`integer` non possa contenere e `Option Strict` fosse `Off`? Il risultato sarebbe un errore in fase di esecuzione piuttosto che in fase di compilazione. Modificando la linea 3 del Listato 2.3 e inizializzando `L` a 100.000.000.000, con `Option Strict Off`, nell'esecuzione del programma si otterrà una segnalazione del tipo di quella mostrata in Figura 2.1.

Figura 2.1

Un `OverflowException` è causato quando un numero troppo grande di tipo `long` viene assegnato ad una variabile di tipo `integer`.



Se vi piace la dichiarazione variabile implicita e preferite cercare da soli fastidiosi bug, fatelo. Provate però a selezionare `Option Explicit` e `Option Strict` e ricompilate. Queste due opzioni possono aiutarvi a evidenziare errori in fase di compilazione.

L'opzione `Strict` permette solo conversioni del tipo ampliamento. Con `Option Strict On` potete assegnare un `integer` a un `Long`, questa operazione è corretta perché il `Long` ha un numero sufficiente di bit per immagazzinare il valore dell'intero ma ovviamente non è vero il contrario. Con `Option Strict On`, non è possibile assegnare un `Long` a un tipo `integer` – conversione di tipo restrittivo – senza un esplicito tipo `cast`. Un tipo `cast` è modo di indicare programmaticamente che si è consapevoli del rischio di overflow, ma che si crede che il valore del

Long in questa situazione sarà compatibile con il numero di bit disponibili in un Integer. Torneremo sull'argomento.

L'opzione `Strict On` vieta il late binding, operazioni su tipi oggetto diverse da uguaglianza, disuguaglianza, `typeof...Is` e `test Is`. Vi è chiesto di dichiarare il tipo di una variabile utilizzando la clausola `As` anche con `Option Strict On`.

Opzione Private

L'opzione `Private` non è più supportata in Visual Basic .NET. L'opzione `Private` impediva l'utilizzo del codice fuori dai vostri moduli VB6, ma VB .NET non ne ha bisogno.

Visual Basic .NET supporta specificatori di accesso. È possibile aggiungere specificatori di accesso private a membri di un modulo per impedire che il codice sia utilizzato all'esterno di quei membri. Potete consultare il Capitolo 7 per saperne di più sugli specificatori di accesso e su come nascondere informazioni.

Opzione Base

L'opzione `Base` non è più supportata. L'opzione `Base` in VB6 indicava se i vettori sono iniziati a 0 o 1. Ad esempio:

```
DIM MyArray (10) as integer
```

In VB6 si avrebbe un vettore di 10 elementi interi se `Option Base` fosse posto a 1 e un vettore di 11 elementi interi con indici che vanno da 0 a 10 se `Option Base` fosse posto a 0. Visual Basic .NET si comporta sempre come se `Option Base` fosse posto a 0.

Probabilmente una scelta di questo tipo inizialmente avrebbe creato confusione. Le versioni beta 1 e beta 2 di VB .NET supportavano vettori con indici che andavano da 0 a n-1, dove n è il numero di elementi del vettore. Nell'aprile 2001, Microsoft ha fatto alcune concessioni agli sviluppatori VB6, tra cui vettori di n+1 elementi con indici da 0 a n, in pratica come se fosse sempre `Option Base` uguale a 0.

Tipi di dati

Parte del Common Language Runtime è il Common Type System (CTS). Per garantire compatibilità tra i vari linguaggi, .NET definisce un sistema di tipi che descrive il modo in cui i tipi sono immagazzinati e i valori permessi per uno specifico tipo in modo tale da assicurare che i dati possano essere utilizzati indistintamente dal linguaggio. Per funzionare nel .NET Framework, i linguaggi hanno bisogno di utilizzare tipi di dati conformi con il CTS.

Tipi CTS racchiudono classi, interfacce e tipi di valore. I tipi in .NET possono avere metodi, proprietà, campi ed eventi. Questo paragrafo descrive i cambiamenti dei tipi esistenti e l'introduzione di nuovi tipi di dati. Altre informazioni saranno presentate nel Capitolo 14.

La Tabella 2.1 mostra il nome dei diversi tipi di dati, il namespace in cui essi sono definiti e l'intervallo accettabile di valori per ogni tipo.

Tabella 2.1 *Tipi di dati CTS che sostituiscono tipi di dati VB6.*

<i>Nome</i>	<i>Namespace</i>	<i>Byte</i>	<i>Range</i>
Boolean	System.Boolean	2	True o False
Byte	System.Byte	1	Da 0 a 255 senza segno
Char	System.Char	2	Da 0 a 65535 senza segno
Date	System.DateTime	8	Dal 1° gennaio dell'anno 1 d.C. al 31 dicembre 9999
Decimal	System.Decimal	16	Approssimativamente da $\pm 7.9 \times 10^{28}$ senza punto decimale fino a ± 7.9 con 28 decimali alla destra del punto decimale; il più piccolo numero è $\pm 1 \times 10^{-29}$ (vedere il paragrafo sui numeri decimali per valori più specifici)
Double	System.Double	8	Approssimativamente da $-1.7E308$ a $-4.9E-324$ per valori negativi; da $4.9E-324$ a $1.7E308$ per valori positivi (vedere più avanti nel capitolo per maggiori informazioni)
Integer	System.Int32	4	Da -2,147,483,648 a 2,147,483,647
Long	System.Int64	8	Da -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807
Object	System.Object	4	Può contenere qualunque tipo; sostituisce Variant
Short	System.Int16	2	Da -32,768 a 32,768
Single	System.Single	4	Approssimativamente da $-3.4E38$ a $-1.4E-45$ per valori negativi; da $1.4E-45$ a $3.4E38$ per valori positivi (vedere più avanti nel capitolo per maggiori informazioni)
String	System.String	Platform-	Da 0 a 2 miliardi di caratteri Unicode
User	System.ValueType Sum	Platform-dependent	Somma del valore dei tipi di dati definiti nel tipo user

Abituarsi ai nuovi valori dei tipi di dati sarà molto più facile che pensare ai tipi di dati come oggetti. Come detto in apertura di paragrafo, tipi di dati VB .NET possono contenere metodi, campi, proprietà ed eventi. I campi sono membri di dati privati, che di solito contengono un valore di proprietà. Potete consultare il Capitolo 7 per saperne di più sui campi.

Ad esempio, il seguente codice mostra il valore massimo di un intero nella finestra Output:

```
(Debug.WriteLine(Integer.MaxValue))
```

Integer è il tipo e MaxValue è un attributo del tipo Integer. I membri in comune sono equivalenti a membri statici C++ o membri di classe Pascal (consultate il Capitolo 11 per saperne

di più). È importante notare che `Integer` è un tipo globale; benché sia utilizzato come un tipo primitivo, è una struttura e ha membri propri.

Nei paragrafi che seguono scopriremo i nuovi tipi di dati e le loro potenzialità.

Tipi di dati Object

Il tipo `Variant` originariamente è stato realizzato per supportare una programmazione COM. I tipi `Variant` aggiungono un alto costo gestionale alle applicazioni e talvolta causano la scrittura di un codice ambiguo. In VB .NET il tipo `Variant` è stato sostituito con la classe `Object`.

`Object` è il nome della classe di origine di tutte le classi e le strutture del Common Language Runtime. (Il costrutto `Structure` sostituisce il costrutto `Type`. Per ulteriori informazioni consultare il Capitolo 5.) Il risultato è che tutte le classi in .NET avranno comportamenti di base comuni e voi avrete in caso di bisogno un tipo generico su cui fare affidamento. La Tabella 2.2 contiene i metodi introdotti nella classe `Object`.

Tabella 2.2 *Membri della classe `Object` ereditati da tutte le classi e strutture.*

<i>Membri</i>	<i>Descrizione</i>
<code>Equals</code>	Testa il codice hash per determinare l'uguaglianza
<code>GetHashCode</code>	Ritorna il codice hash che rappresenta il valore dell'oggetto
<code>GetType</code>	Ritorna il tipo di classe
<code>ReferenceEquals</code>	Ritorna un valore Boolean che indica se due riferimenti a un oggetto siano riferiti o no allo stesso oggetto
<code>ToString</code>	Ritorna una stringa che rappresenta un oggetto



È importante osservare che tutti questi metodi di membro in nessun modo suggeriscono che operatori comuni, come `=`, siano sostituiti da metodi `overloading`. Gli operatori comuni esistono ancora e li utilizzerete per operazioni quotidiane. (più avanti in questo capitolo troverete ulteriori informazioni).

Tutti i metodi elencati nella Tabella 2.2 sono metodi di esempio, tranne il metodo `ReferenceEquals`. Avrete bisogno di una classe di esempio per invocare metodi di esempio. `ReferenceEquals` è un metodo comune, che può essere invocato utilizzando una classe o un'istanza.

Equals

Il membro `Equals` restituisce un valore Boolean il quale indica se il valore dell'istanza chiamante è equivalente al valore dell'istanza dell'argomento passato gli. Il metodo `Equals` è utilizzato per verificare il valore del codice hash per determinare l'uguaglianza di due oggetti (traducendo in codice il discorso precedente, `J.Equals(I)` è `True`).

GetHashCode

`GetHashCode` è definito a livello `Object` e restituisce un codice `Hash` che rappresenta il valore matematico dell'oggetto. Una funzione `Hash` ritorna lo stesso valore per due diversi oggetti se i due oggetti rappresentano lo stesso valore. I codici `Hash` sono utilizzati per agevolare l'immagazzinamento di dati in tipi di dati astratti facendo riferimento a una tabella di `Hash`. Assicurando una distribuzione casuale di valori il codice `Hash` può essere usato per indicizzare elementi nella tabella `Hash` in maniera veloce.



Un'altra caratteristica interessante di .NET è il metodo `GetHashCode`. Cosa rappresenta tale metodo? Non è semplice trovare un riferimento specifico per spiegare tale metodo, ma è facile capire che verificare un'uguaglianza basata su un codice di `Hash` è molto più veloce che verificare l'uguaglianza membro per membro, cosa che risulterebbe assai noiosa.

```
Object1.Field1 = Object2.Field2
```

```
Object1.Field2 = Object2.Field2
```

Algoritmi di `Hash` possono essere scritti genericamente, verificando che tutti gli oggetti abbiano un metodo `GetHashCode`.

La documentazione di aiuto suggerisce che i codici di hash dovrebbero essere basati su un campo costante e invariabile all'interno di una classe. Tutti gli oggetti dello stesso tipo che hanno lo stesso codice hash sono da considerarsi uguali.

GetType

`GetType` restituisce la classe o la metaclassa, del tipo invocato. Il seguente esempio illustra tale concetto.

```
Dim I As Integer
(MsgBox(I.GetType.Name))
```

`I.GetType` restituisce un oggetto del tipo `Integer`, che è un tipo `Int32`. La proprietà `Name` è definita nella classe `Type`. Nell'esempio il message box mostrerebbe `Int32`, il tipo CLR di un intero. Un oggetto è un'istanza della classe stessa. Questo è essenziale per sostenere `Run Time Type Information (RTTI)`. `RTTI` determina la capacità di una struttura orientata agli oggetti di determinare le classi di oggetti in esecuzione. Ciò garantisce grande flessibilità. Un esempio eccellente di un sistema di tipo dinamico è dimostrato nel Capitolo 6.

GetTypeCode

`GetTypeCode` restituisce il tipo di codice di un oggetto. `I.GetTypeCode`, dove `I` è un `Integer`, restituisce il valore 9, che è la costante `TypeCode.Int32`.

ToString

Il metodo `ToString` restituisce una stringa come rappresentazione di un oggetto. (chiamare `ToString` su un `Integer` produrrebbe una stringa equivalente al valore dell'`Integer`). Questo metodo al livello più basso della gerarchia di una classe è utile perché assicura che tutti i tipi possano essere rappresentati da una stringa. Questo rende particolarmente facile realizzare procedure come `MsgBox`, che si aspetta un parametro di tipo testo.

Tipi integrali

Ci sono quattro tipi integrali in VB .NET: `Short`, `Integer`, `Long` e `Byte`. `Byte`, `Integer` e `Long` esistevano anche in VB6 mentre `Short` è stato introdotto in VB .NET. Il numero di byte utilizzati per conservare `Integer` e `Long` è stato raddoppiato in VB .NET da 16 a 32 e da 32 a 64, rispettivamente. Il numero di valori accettabili per un dato tipo è una funzione di permutazione del numero di bit. Un intero a 32 bit è così capace di immagazzinare 232 possibili valori; separando il numero dei possibili valori tra valori negativi e valori positivi e permettendo lo zero si otterrà una gamma di valori che va da -2.147.483.648 a 2.147.483.647.

I tipi integrali rappresentano solo numeri interi. Gli interi sono utili specialmente per operazioni di indicizzazione, come ad esempio cicli `For...Next`. I tipi `Integer` sono classi `ValueType`, discendenti dirette del tipo `Object`. Tutti i tipi sono classi in .NET, benché ci si aspetti che `ValueType` funzioni come un tipo di dati nativo. I tipi derivati da `ValueType` sono citati genericamente come tipi di valore, ciò implica una differenza fra tipi come `Integer` e tipi di riferimento come un pulsante di controllo.

La differenza fra tipi di riferimento e tipi di valore è che i tipi di riferimento portano con sé informazioni in fase di esecuzione mentre i tipi di valore no. Potete consultare i Capitoli 6 e 12 per ulteriori informazioni. Quando il CLR ha bisogno di trattare un tipo valore come un tipo riferimento, ad esempio quando si richiedono informazioni per un tipo valore, si verifica un'operazione chiamata `Boxing`. `Boxing`, chiamata dopo l'istruzione `Intermediate Language (IL)`, è un processo che crea un oggetto pila e copia il valore del tipo valore nell'oggetto pila. Questo permette al CLR di trattare il tipo valore come un tipo riferimento. Quando il tipo riferimento non è più necessario, è eseguito il processo inverso, chiamato `Unboxing`.

Le differenze fra tipi valore e tipi riferimento fanno sì che i tipi riferimento siano altrettanto facili da utilizzare ma che abbiano tutti i vantaggi che comporta l'essere oggetti. Per fortuna tutti i `Boxing` e `Unboxing` si verificano dietro le quinte.



I membri pubblici sono membri accessibili dal codice al di fuori di restrizioni di classe o di struttura. I membri protetti sono accessibili alla classe o alle sottoclassi che li contengono. Consultare il Capitolo 7 per saperne di più su specificatori di accesso pubblici e protetti.

Per determinare i membri di un `Integer`, cercate il tipo CTS `Int32` e le sue classi genitrici nei file di aiuto. I campi pubblici di un tipo `Integer` sono il `MaxValue` e il `MinValue`. Il metodo comune pubblico di un `Integer` è il metodo `Parse`. I metodi pubblici sono quelli ereditati da `Object` ma si possono estendere anche metodi protetti, come `Finalize` e `MemberwiseClone`, creando un nuovo tipo da `Object`.

MinValue e MaxValue

`MinValue` e `MaxValue` in tipi integrali sono campi condivisi e di conseguenza accessibili senza la creazione di un tipo. Ad esempio, il codice seguente descrive il funzionamento di entrambi i membri utilizzando il tipo `Integer`:

```
Dim Test As Boolean
Test = ( Integer.MinValue < 100000 ) and ( 100000 < Integer.MaxValue )
```

Test sarà valutato `True`. In VB6 Test sarebbe valutato `False`, ma in VB .NET invece 100.000 perché il tipo `Integer` è a 32 bit.



La presenza di metodi in una sezione specifica non implica che tali metodi non esistano anche in altre classi. I metodi come `MinValue`, `MaxValue` e `Parse` non sono limitati a classi integrali.

Metodo condiviso `Parse`

`Parse` è un metodo condiviso che converte la rappresentazione di stringa di un numero al suo equivalente integrale.

```
Dim Number As String
Number = "4"
Debug.WriteLine(Integer.Parse(Number))
```

In tale codice viene dichiarata una variabile, `Number`, come un tipo stringa. La stringa "4" è assegnata alla variabile, `Integer.Parse(Number)` ritorna la stringa come un `Integer`.

Tipi non integrali

I tipi non integrali sono numeri a virgola mobile, compresi i tipi `Single`, `Double` e `Decimal`. I numeri a virgola mobile supportano una gamma più ampia di valori e sono utili per operazioni matematiche che prevedono numeri frazionari. I tipi `Single` e `Double` rappresentano una gamma più grande di valori del tipo `Decimal`, ma `Single` e `Double` sono soggetti a errori di arrotondamento. Il tipo `Decimal` è adatto per calcoli finanziari in cui meno si tollerano arrotondamenti.

Si può utilizzare una notazione a virgola mobile o esponenziale quando vengono assegnati valori a variabili `Single` o `Double`. La notazione a virgola mobile è nel formato `whole.part`: ad esempio la notazione 3.14159. La notazione esponenziale o scientifica utilizza il formato mantissa-e-esponente. Ad esempio, `3e2` è equivalente a 300. (La mantissa è la parte di valore, di solito normalizzata in una cifra significativa singola e l'esponente è il moltiplicatore di potenza.)

Se si inizializza un numero `Decimal` con un valore maggiore del valore del `Long` intero più grande, si dovrà aggiungere il suffisso `D` alla fine del valore iniziale. Ad esempio,

```
Dim Number As Decimal = 9223372036854775808D
```

assegna 9 quintillioni e spiccioli alla variabile `Number`.

Divisione a virgola mobile

I numeri a virgola mobile sono stati modificati per essere conformi allo standard sancito dall'IEEE (Institute of Electrical and Electronics Engineers). Il risultato è che si ottengono alcuni nuovi comportamenti quando si va ad operare con un'aritmetica a virgola mobile.



L'operazione $1/0$ ritorna infinito e $\text{math.Sqrt}(-1)$ restituisce Not a Number (NaN) per numeri a virgola mobile. Le divisioni tra interi ($1\backslash 0$ e $-1\backslash 0$) causano invece un'eccezione del tipo *divisione per zero*.

Il risultato di una divisione per zero è il valore infinito. Per esempio il codice seguente mostra un message box che contiene la parola infinito:

```
Dim D As Double = 0
MsgBox( 5 / D )
```

Provate a calcolare la radice quadrata di -1 otterrete il valore NaN (Not a Number). Il codice seguente mostra un esempio:

```
MsgBox( Math.Sqrt(-1))
```

Visual Basic 6 richiedeva la scrittura di un error handler per catturare problemi come la divisione per zero. In Visual Basic .NET, potrete usare istanze di metodi o operatori di tipo non integrale per testare questi valori.

Membri di tipo non integrale

Vi sono numerosi membri di tipo non integrale. La Tabella 2.3 mostra un elenco dei membri dei tipi `Single` e `Double`.

Tabella 2.3 *Membri del tipo virgola mobile.*

<i>Nome</i>	<i>Descrizione</i>
<i>Campi condivisi</i>	
Epsilon	valore positivo più piccolo maggiore di zero
MaxValue	valore massimo
MinValue	valore minimo
NaN	costante simbolica che rappresenta il valore Not-a-Number (NaN)
NegativeInfinity	valore simbolico (-Infinito) rappresenta infinito negativo
PositiveInfinity	valore simbolico (Infinito) rappresenta infinito positivo
<i>Metodi condivisi</i>	
IsInfinity	ritorna un Boolean che indica se un argomento è infinito (per esempio, <code>Double.IsInfinity(D/0)</code> è True)
IsNaN	ritorna un Boolean che indica se un argomento è NaN
IsNegativeInfinity	test per infinito negativo (per esempio, <code>-1/0</code>)
IsPositiveInfinity	test per infinito positivo (per esempio, <code>1/0</code>)
Parse	converte una stringa in un tipo di classe

(segue)

Tabella 2.3 (continua) *Membri del tipo virgola mobile.*

<i>Nome</i>	<i>Descrizione</i>
<i>Metodi di istanza</i>	
CompareTo	<p>Confronta un'istanza con il riferimento a un argomento; il risultato è <0 se l'istanza è minore dell'argomento, 0 se l'istanza è uguale all'argomento e >0 se l'istanza è maggiore dell'argomento</p>

Tutti i membri elencati nella Tabella 2.3 sono membri pubblici dei tipi `Single` e `Double`.

Tipo decimale non integrale

I numeri decimali sostituiscono il ruolo precedentemente coperto dal tipo `Currency`. Non esiste più la valuta in Visual Basic .NET. Nella Tabella 2.4 sono elencati i membri del tipo `Decimal`. Come si può vedere si tratta di un elenco cospicuo.

Tabella 2.4 *Membri del tipo Decimal.*

<i>Nome</i>	<i>Descrizione</i>
<i>Campi condivisi</i>	
MaxValue	Massimo valore decimale
MinusOne	Rappresenta il numero negativo 1
MinValue	Massimo valore negativo
<i>Campi condivisi</i>	
One	Rappresenta il numero 1
Zero	Rappresenta il numero 0
<i>Metodi condivisi</i>	
Add	Somma valori decimali
Compare	Confronta due argomenti decimali, per esempio <code>Decimal.Compare(D1, D2)</code> . <code>D1 < D2</code> ritorna < 0; <code>D1 = D2</code> ritorna 0, e <code>D1 > D2</code> ritorna > 1.
Divide	Ritorna il decimale risultante da <code>D1</code> diviso per <code>D2</code> (per esempio, <code>Decimal.Divide(D1, D2)</code> , dove <code>D1</code> e <code>D2</code> sono valori decimali)
Equals	Determina se due valori sono uguali usando il codice hash
Floor	Arrotonda il decimale al numero minore (per esempio, <code>Decimal.Floor(5.7) = 5</code>)

FromOACurrency	Converte un tipo Currency in Decimal
GetBits	Decimale rappresentato come vettore binario di elementi Int32 (per esempio, <code>Decimal.GetBits(3).GetValue(0) = 3</code>)
Multiply	Svolge un operazione di prodotto decimale con due argomenti
Negate	Nega il valore dell'argomento decimale
Parse	Converte una stringa in decimale (vedi Listato 2.4)
Round	Arrotonda un numero decimale (per esempio, <code>Decimal.Round(10.347, 2) = 10.35D</code>)
Subtract	Sottrazione di due argomenti decimali. Ritorna la differenza
ToByte	Converte un decimale nell'equivalente 8-bit ; se il valore è maggiore, <code>Byte.MaxValue</code> provoca un'eccezione
ToDouble	Converte un Decimale in Double (vedi ToByte)
ToInt16	Converte un Decimale in 16-bit Integer (vedi ToByte)
ToInt32	Converte un Decimale in 32-bit Integer (vedi ToByte)
ToInt64	Converte un Decimale in 64-bit Integer (vedi ToByte)

Metodi condivisi

ToOACurrency	Converte un valore decimale in un Currency (Il tipo currency è supportato in Microsoft Office)
ToSByte	Converte un Decimale in signed Byte (vedi ToByte)
ToSingle	Converte un Decimale in Single (vedi ToByte)
ToUInt16	Converte un Decimale in 16-bit unsigned integer (vedi ToByte)
ToUInt32	Converte un Decimale in 32-bit unsigned integer (vedi ToByte)
ToUInt64	Converte un Decimale in 64-bit unsigned integer (vedi ToByte)
Truncate	Ritorna un decimale senza la parte frazionaria

Costruttori

Decimal	Inizializza un nuovo costruttore decimale
---------	---

Metodi di istanza

CompareTo	Confronta istanza e argomento; ritorna -1, 0, e 1 rispettivamente se l'istanza è minore dell'argomento, uguale all'argomento, o maggiore dell'argomento.
-----------	--

Come promesso, il Listato 2.4 mostra come il metodo `Parse` possa essere utilizzato per convertire una stringa formattata nel suo equivalente `Decimal`.

Listato 2.4 *Conversione Decimal.Parse.*

```
Dim Number As String
Number = "100,000,005"
Debug.WriteLine(Decimal.Parse(Number))
```

Eseguendo il Listato 2.4 si scrive 100000005 (centomilionicinque) nella finestra Output. Molti dei metodi come `Parse` sono utilizzati comunemente in modo polimorfo. Ciò significa che, quando è chiamato `Parse`, questo si comporta in base al tipo della classe. L'utilizzo dello stesso nome per i metodi che eseguono le stesse operazioni semplifica la semantica del linguaggio. Immaginate se aveste dovuto memorizzare un nome univoco per il funzionamento di `Parse` per ogni tipo di dati. Per saperne di più sul polimorfismo, potete consultare il Capitolo 7.

Tipi Double e variabili Datetime

Visual Basic .NET non conserva variabili di data e ora come numeri in doppia precisione. Il Common Language Specification definisce un tipo `DateTime` distinto dal tipo `Double`.

Visual Basic 6 supporta codice del tipo seguente:

```
Dim D As Double
D = Now
```

Dopo aver eseguito il comando `D = Now`, `D` contiene un numero in doppia precisione. La parte intera del numero rappresenta la data e la parte frazionaria del numero rappresenta l'ora. Visual Basic .NET restituirà l'errore "Use the method `ToOADate`" su un `Date` per convertire un tipo `Date` a un `Double`.

Tipo di dati Char

`Char` è definito in `System.Char`. Il tipo `Char` è stato allargato a 16 bit per supportare caratteri Unicode. I 128 caratteri ASCII sono sufficienti per implementare l'alfabeto inglese, ma lo standard Unicode è stato definito per supportare tutti i caratteri in tutte le altre lingue, tra cui i 5.000 o più caratteri giapponesi Kanji.

Il concetto di un tipo carattere non esiste in VB6. Inoltre, Visual Basic .NET non supporta la notazione `fixed-length String`:

```
Dim S As String * 1 'no longer supported in VB .NET
```

Questa dichiarazione non è gestita in VB.NET. La Tabella 2.5 definisce i membri del tipo `Char`.

Tabella 2.5 *Membri della nuova struttura Char.*

<i>Nome</i>	<i>Descrizione</i>
<i>Campi condivisi</i>	
<code>MaxValue</code>	Costante che rappresenta il massimo valore
<code>MinValue</code>	Costante che rappresenta il minimo valore

Metodi condivisi

<code>CompareTo</code>	Compara istanze e argomenti; ritorna -1, 0, or 1 se l'istanza è minore dell'argomento, uguale all'argomento, o maggiore dell'argomento, rispettivamente
<code>GetNumericValue</code>	Ritorna il valore numerico del carattere Unicode
<code>GetUnicodeCategory</code>	Categoria del carattere Unicode raggruppato con la costante enumerated <code>UnicodeCategory constant</code>
<code>IsControl</code>	Ritorna un Boolean che indica se il carattere è un controllo
<code>IsDigit</code>	Verifica se è un numero
<code>IsLetterOrDigit</code>	Verifica se è una lettera o un numero
<code>IsLower</code>	Verifica se il carattere è minuscolo
<code>IsNumber</code>	Verifica se decimale o esadecimale
<code>IsPunctuation</code>	Verifica se è carattere di punteggiatura
<code>IsSeparator</code>	Verifica se è un separatore
<code>IsSurrogate</code>	Verifica se è un carattere surrogato
<code>IsSymbol</code>	Verifica se è un simbolo
<code>IsUpper</code>	Verifica se è maiuscolo
<code>IsWhiteSpace</code>	Verifica se è uno spazio vuoto
<code>ParseUnicode</code>	Converte il valore di un argomento di tipo stringa in un carattere
<code>ToLower</code>	Converte il carattere Unicode in minuscolo
<code>ToUpper</code>	Converte il carattere Unicode in maiuscolo

Dichiarare variabili di tipo `char` è come dichiarare qualsiasi altra variabile. La sintassi base è:

```
Dim C As Char
```

Campi condivisi e metodi possono essere chiamati con istanze o con riferimenti alla classe. Il codice seguente se eseguito mostrerà a video il carattere A :

```
MsgBox( Char.ToUpper("a"))
```

Tipi String

Il tipo `String` ha subito alcune trasformazioni. Le stringhe ovviamente sono classi in Visual Basic .NET. Non si possono più dichiarare stringhe di lunghezza fissata come si faceva in VB6, ma in compenso ci sono altri considerevoli vantaggi.

Le stringhe possono essere lunghe circa 2 miliardi di caratteri e inoltre il tipo `String` ha un numero elevato di membri che rendono significativamente più facile gestire le stringhe.

Finora il capitolo non contiene molti frammenti di codice per altri tipi, ma le stringhe sono così comuni che si sono inclusi numerosi esempi di dichiarazioni di codice per dimostrare alcune delle nuove capacità di stringa elencate nella Tabella 2.6.

Tabella 2.6 *Membri della classe string in Visual Basic.NET.*

<i>Nomi</i>	<i>Descrizione</i>
<i>Campi condivisi</i>	
Empty	Costante che rappresenta la stringa vuota
<i>Metodi condivisi</i>	
Compare	Confronta due oggetti stringa passati come argomenti
CompareOrdinal	Confronta due stringhe argomenti senza considerare il linguaggio nazionale locale
Concat	Ritorna una nuova stringa ottenuta concatenando insieme una o più stringhe
Copy	Ritorna una stringa copia della stringa argomento
Format	Simile a printf in C;
Intern	Ritorna un riferimento all'istanza della stringa
IsInterned	Recupera il riferimento alla stringa
Join	Inserisce separatori di stringa tra ogni stringa in un vettore
<i>Istanze di proprietà</i>	
Chars	Ritorna il carattere in una certa posizione
Length	Ritorna la lunghezza di una stringa
<i>Istanze di metodi</i>	
Clone	Clona una stringa
CompareTo	Confronta una stringa con una stringa passata come argomento
CopyTo	Copia un numero specificato di caratteri a partire da un offset in una certa posizione di un vettore di caratteri Unicode
EndsWith	Ritorna un Boolean che indica se la stringa termina con la stringa passata come argomento
GetEnumerator	Ritorna un CharEnumerator che permette di iterare operazioni sulla stringa
IndexOf	Ritorna l'indice di una sottostringa
IndexOfAny	Ritorna l'indice della prima occorrenza di ogni carattere nel vettore di caratteri passato come argomento
Insert	Inserisce una stringa in una certa posizione
LastIndexOf	Ritorna l'indice dell'ultima istanza di una stringa o carattere

LastIndexOfAny	Ritorna l'ultima posizione dell'occorrenza di ogni carattere specificato come argomento
PadLeft	Allinea a destra caratteri di riempimento
PadRight	Allinea a sinistra i caratteri di riempimento o caratteri indicati
Remove	Rimuove un numero di caratteri indicato partendo da una posizione
Replace	Sostituisce una sottostringa con un'altra
Split	Separa gli elementi di una stringa in un vettore quando incontra un carattere di delimitazione; è l'opposto di join
StartsWith	Valore Boolean che indica se la stringa inizia con la stringa argomento
SubString	Copia una sottostringa in una stringa
ToCharArray	Copia i caratteri della stringa dentro un vettore di caratteri Unicode
ToLower	Ritorna la copia in minuscolo della stringa
ToUpper	Ritorna la copia in maiuscolo della stringa
Trim	Rimuove gli spazi vuoti da una stringa
TrimEnd	Rimuove gli spazi vuoti a destra di una stringa
TrimStart	Rimuove gli spazi vuoti a sinistra di una stringa

Il Listato 2.5 contiene frammenti di codice che dimostrano l'uso di alcuni metodi della classe String.

Listato 2.5 *Esempio di membri della classe string.*

```

1: Sub TestStringMethods()
2:
3:     Dim S As String = _
4:     "Welcome to Valhalla Tower Material Defender"
5:     Dim T As String = S.Clone()
6:
7:     Debug.WriteLine(S.ToLower())
8:     Debug.WriteLine(S.ToUpper())
9:     Debug.WriteLine(T.EndsWith("Defender"))
10:    Debug.WriteLine(S.Chars(5))
11:    Debug.WriteLine(String.Concat("Hello", " ", "World"))
12:    Debug.WriteLine(S.Substring(5, 10))
13:
14:    Debug.WriteLine(String.Format( _
15:        "Abraham Lincoln was born {0:s}", "February 12"))
16:
17:    Dim Enumerator As CharEnumerator = S.GetEnumerator
18:
19:    While (Enumerator.MoveNext())
20:        Debug.WriteLine(Enumerator.Current())
21:    End While
22:
23:    Dim R() As String = {"2", "12", "1966"}
24:    MsgBox(S.Join("/", R))
26: End Sub

```

La maggior parte del codice vi apparirà comprensibile. Le linee 3 e 4 dimostrano la nuova forma di dichiarazione e inizializzazione associate delle variabili in VB .NET. Il modo di accedere ai membri potrebbe creare confusione; occorre ricordare che nel codice d'esempio si utilizza la notazione `object.member` per istanze di membri e `class.member` per membri in comune.

Le linee dalla 17 alla 21 mostrano un `Enumerator`. Se avete già lavorato con `Iterators`, ad esempio in C++, il codice vi apparirà comprensibile. Il vantaggio di un `Enumerator` è che il codice per l'iterazione è identico per qualunque di tipo che supporti la classe `Enumerator`.

La linea di codice 23 può sembrare un po' strana. Si tratta di una dichiarazione e inizializzazione di un vettore di stringhe. Ulteriori esempi sull'utilizzo di membri della classe `String` saranno presenti in tutto il libro.

Tipo Boolean

Il tipo `Boolean` ha subito nel tempo varie modifiche per arrivare al suo stato attuale. In VB6, valori `Boolean` hanno utilizzato 0 per `False` e -1 per `True` come valori integrali sottostanti. VB .NET è stato originariamente progettato per utilizzare 0 e 1 rispettivamente per `False` e `True`.

I valori 0 e 1 sono quelli supportati dal CLR ma, come concessione agli sviluppatori VB6, i valori del tipo `Boolean` sono stati convertiti a 0 e a -1. Per funzionare correttamente con i valori CLR i `Boolean` devono essere convertiti internamente a 0 e 1 ma quando voi valutate i `Boolean` come interi, prendete in considerazione i valori 0 e -1.

Per evitare complicazioni, gli operatori `Boolean` stavano per essere convertiti a operatori logici piuttosto che operatori logici e operatori di confronto bit a bit, e le valutazioni `Boolean` stavano per essere cortocircuitate. Tutti questi cambiamenti programmati sono stati abrogati intorno all'aprile 2001. In definitiva il comportamento del tipo `Boolean` risulta molto simile al comportamento in VB6.

Il risultato finale è che se utilizzate sempre `Boolean True` o `False`, il valore sottostante non avrà nessun impatto negativo sul vostro codice e il valore `Boolean` avrà un significato semantico più grande di un valore `Integer-as-Boolean`.

Operatori booleani

Gli operatori booleani includono `And`, `Or`, `Not` e `Xor`. `AndAlso` e `OrElse` sono stati aggiunti per supportare valutazioni di tipo cortocircuito in VB .NET. Ne riparleremo tra breve.

Visual Basic .NET esegue le stesse operazioni logiche e valutazioni bit a bit con `And`, con `Or`, `Not` e `XOR` come era in VB6. Se gli operandi sono di tipo `Boolean`, gli operatori `Boolean` eseguono una valutazione logica. Se gli operandi sono integrali, è eseguita una valutazione bit a bit. Il listato 2.6 dimostra diverse valutazioni logiche e bit a bit.

Listato 2.6 *Valutazioni logiche e bit a bit in VB .NET.*

```
1: Sub TestBooleans()  
2:  
3:   Dim B As Boolean  
4:   B = False Or True
```

```
5:   Debug.WriteLine(B)
6:
7:   B = False Xor False
8:   Debug.WriteLine(B)
9:
10:  B = False And True
11:  Debug.WriteLine(B)
12:
13:  B = Not True
14:  Debug.WriteLine(B)
15:
16:  Dim I As Integer
17:
18:  I = 3 Or 4
19:  Debug.WriteLine(I)
20:
21:  I = 2 And 4
22:  Debug.WriteLine(I)
23:
24:  I = 3 Xor 3
25:  Debug.WriteLine(I)
26:
27:  I = Not 5
28:  Debug.WriteLine(I)
29:
30: End Sub
```

La linea 5 scrive `True` alla finestra `Output`. La linea 8 scrive `False` alla finestra `Output`; le valutazioni logiche `Xor` sono `True` solo quando gli operandi non coincidono. La prova logica sulla linea 10 fornisce `False` in quanto un operatore `False` in `And` con qualsiasi altro operatore è sempre `False`. La negazione di `True` è `False`; la linea 14 scrive `False` alla finestra `Output`. Perché la linea 18 valuta due interi, l'operazione sarà bit a bit. Gli ultimi quattro bit per 3 sono 0011 e gli ultimi quattro bit per 4 sono 0100. Il risultato di 0100 Or 0011 è 0111 o 7. La linea 19 scrive 7 alla finestra `Output`. 2 e 4 non hanno alcun bit in comune, la riga 22 andrà a scrivere 0. Tutti i pezzi sono uguali sulla linea 24, e questo è esattamente il modo per ottenere 0 in una valutazione bit a bit. La negazione di 5 (avendo tutti 0 tranne gli ultimi quattro bit che sono 0101) mette a 1 tutti i primi 28 bit e 1010 per gli ultimi quattro bit (ricordate che in VB .NET gli interi sono a 32 bit).

Valutazioni di tipo Boolean complete

Visual Basic .NET compie una valutazione completa delle espressioni booleane senza l'utilizzo degli operatori `AndAlso` e `OrElse`. Una valutazione completa significa che tutti gli operandi dell'espressione sono valutati al fine di determinare il risultato. Prendiamo in esame la dichiarazione `False and anything`: restituisce sempre un valore `False`. Non ci sono ragioni pratiche per valutare l'operando a destra (*rhs*, *right hand side*); la non valutazione è detta cortocircuito. Ma cosa potrebbe succedere se l'operando *rhs* fosse il risultato di una funzione? Se per esempio la funzione aggiornasse un database? La risposta è che l'utilizzo del cortocircuito non aggiornerebbe il database. Il frammento di codice che segue mostra questo scenario

```
Function UpdateDatabase() AS Boolean
    ' update database
    return Passed
End Function
If( BoolVal And UpdateDatabase()) Then
```

62 Capitolo 2

Il codice valuta l'And tra BoolVal e UpdateDatabase. Se la variabile BoolVal fosse False e venisse utilizzato il cortocircuito UpdateDatabase non sarebbe mai chiamata. Ecco un altro caso.

```
Function LogIn() As Boolean
    ' Login and return connected state
    return IsConnected()
End Function

If( LoggedIn() Or LogIn() ) Then
    ' Process
```

Allo stesso modo con l'operatore Or se fosse circuitato e qualunque valore la prima funzione restituisse il risultato sarebbe il valore in questione e LogIn() non verrebbe mai valutata. Scrivere codice dipendente dal cortocircuito o dalla valutazione completa porta in strade diverse, per questo motivo è auspicabile che si utilizzino il meno possibile costrutti che dipendano dal tipo di valutazione delle espressioni boolean.

Tipo DateTime

Le variabili di questo tipo sono state riviste. Il tipo DateTime non viene più salvato come Double, ma è diventato un tipo a tutti gli effetti. VB6 conservava questa tipologia di valori come parte frazionaria di un Double, in questo modo .5, per esempio, era equivalente a #12:00:00 PM#. Quando si vogliono utilizzare variabili di tipo data e ora in Visual Basic .NET si dovrà dichiarare una variabile di tipo DateTime. Alcuni metodi possono garantire la compatibilità all'indietro, ma questo tipo ora è una sottoclasse della classe ValueType e possiede nuovi metodi e attributi. La Tabella 2.7 mostra la lista dei membri pubblici del tipo DateTime. Nel Listato 2.7 sono riportati alcuni esempi di dichiarazioni che spiegano le nuove capacità di questo tipo.

L'ambiente .NET supporta l'overloading di operatori per tipi come DateTime ma purtroppo Visual Basic .NET in particolare no.



Tabella 2.7 I tipi DateTime sono tipi di valore in VB .NET, non Doubles.

Nome	Descrizione
<i>Campi condivisi</i>	
MaxValue	Limite superiore di una data; 31 dicembre 9999 (Y10K bug)
MinValue	Limite inferiore; 1° gennaio 1 d.C.
<i>Proprietà condivise</i>	
Now	Data e ora corrente
Today	Data odierna
UTCNow	Ora corrente espressa come UTC (Universal Coordinated Time o Greenwich Mean Time [GMT])

Metodi condivisi

Compare	Confronta due date; $t1 < t2$ restituisce -1; $t1 = t2$ restituisce 0; $t1 > t2$ restituisce 1
DaysInMonth	Ha come parametri l'anno e il mese e restituisce il numero di giorni del mese
FromFileTime	Restituisce data e ora di sistema uguali al file timestamp
FromOADate	Converte date VB6, o date OLE Automation a DateTime (esempio, DateTime.FromOADate(0.5) è 12:00:00 PM)
IsLeapYear	Restituisce un boolean che indica se l'anno passato come parametro è bisestile
Parse	Converte una data in formato stringa a DateTime
ParseExact	Converte un parametro stringa a DateTime utilizzando IFormatProvider.System.Globalization. DateTimeFormatInfo implementa l'interfaccia IFormatProvider

Costruttore

DateTime	Inizializza un'istanza di oggetto DateTime
----------	--

Proprietà di Istanza

Date	Restituisce il valore data di un'istanza
Day	Restituisce il valore giorno del mese di un'istanza
DayOfWeek	Crea un'istanza giorno del mese
DayOfYear	Restituisce il numero di giorno dell'anno
Hour	Restituisce l'ora di un'istanza
Millisecond	Restituisce i millisecondi di un'istanza
Minute	Restituisce i minuti di un'istanza
Month	Restituisce il mese di un'istanza
Second	Restituisce i secondi di un'istanza
Ticks	Restituisce il numero di 100-nanosecondi (ticks) dal 1/1/0001
TimeOfDay	L'ora del giorno
Year	L'anno

Metodi di Istanza

Add	Aggiunge argomenti TimeSpan al valore di un'istanza
AddDays	Aggiunge giorni a un'istanza
AddHours	Aggiunge ore a un'istanza

(segue)

Tabella 2.7 (continua) *I tipi DateTime sono tipi di valore in VB .NET, non Doubles.*

<i>Nome</i>	<i>Descrizione</i>
AddMilliseconds	Aggiunge millisecondi a un'istanza
AddMinutes	Aggiunge minuti a un'istanza
AddMonths	Aggiunge mesi a un'istanza
AddSeconds	Aggiunge secondi a un'istanza
AddTicks	Aggiunge ticks a un'istanza
AddYears	Aggiunge anni a un'istanza
GetDateTimeFormats	Restituisce DateTime in formato String
Subtract	Sottrae tempo da un'istanza
ToFileTime	Converte DateTime al local system file time
ToLocalTime	Converte UTC al local time
ToLongDateTime	Converte la data in String
ToLongTimeString	Converte l'ora in String
ToOADate	Converts la data in una data compatibile OLE Automation (Double)
ToShortDateString	Converte la data a String
ToShortTimeString	Converts l'ora a String
ToTimeString	Converte data e ora a String
ToUniversalTime	Converte la data in UTC DateTime

Il tipo `DateTime` misura il tempo dalle 12:00 AM del 01/01/0001 del calendario Gregoriano in Ticks (100 nanosecondi). Il Listato 2.7 mostra alcune delle capacità di base del tipo `DateTime`; nel resto del libro troverete numerosi altri esempi.

Listato 2.7 *Utilizzo del nuovo tipo DateTime.*

```

1: Sub TestDateTime()
2:
3:   Debug.WriteLine(DateTime.FromOADate(0.5))
4:   Debug.WriteLine(DateTime.Parse("12:00:00 PM"))
5:   Dim Provider As New System.Globalization.DateTimeFormatInfo()
6:   Debug.WriteLine(Provider.AMDesignator())
7:   Debug.WriteLine(DateTime.ParseExact("12:42", "hh:mm", Provider))
8:
9:   Dim D As DateTime
10:  D = Now()
11:
12:  Debug.WriteLine(D.ToUniversalTime())
13:

```

```
14: Debug.WriteLine("UTC-LocalTime=" + D.ToUniversalTime(). _
15:     Subtract.ToString())
16:
17: Debug.WriteLine("Ticks since 12:00AM January 1, 1 CE=" _
18:     + Now().Ticks())
19:
20: End Sub
```

La linea 3 converte il `Double` 0.5 in 12:00:00 PM. Sia chiaro che tale novità è solo di Visual Basic .NET; Office XP utilizza ancora VBA e di conseguenza utilizza `Doubles` per contenere tipi data e ora. Nella linea 3 `FromOADate` converte una data OLE Automation in una `DateTime`. La linea 4 converte una stringa in `DateTime`; ricordate che `DateTime` è una classe e non un'istanza. La linea 4 mostra l'utilizzo di un metodo condiviso. La linea 5 crea una istanza della `DateTimeFormatInfo` che implementa l'interfaccia `IFormatProvider`. La linea 6 usa l'oggetto `Provider` per stampare `AMDesignator` ovvero AM. La linea 7 passa `Provider` come argomento di `ParseExact`.

Dichiarazione di variabili

La dichiarazione di variabili ha subito alcuni utili cambiamenti in Visual Basic .NET. In VB6 non si potevano dichiarare più variabili per uno stesso tipo; qualora si dichiarasse una lista di variabili tutte erano di tipo `Variant` eccetto l'ultima. VB6 non supporta la dichiarazione e l'inizializzazione contemporaneamente. Di seguito si analizzano i differenti stili di dichiarazione.

Dichiarare e inizializzare variabili singole

Visual Basic .NET supporta la dichiarazione e l'inizializzazione di variabili nella stessa istruzione. Nel Listato 2.8 sono riportate alcune dichiarazioni in VB6 seguite dalle corrispondenti in Visual Basic .NET.

Listato 2.8 Istruzioni di dichiarazione e inizializzazione in VB6.

```
1: Private Sub Command1_Click()
2:
3:     Dim I As Integer
4:     I = 5
5:     Dim S As String
6:     S = "Jello Mold"
7:
8:     Dim D As Date
9:     D = Now
10:
11:     Dim F As Double
12:     F = D
13:
14:     Dim S1 As String: S1 = "Some More Text"
15:
16:     Debug.Print S1
17:
18: End Sub
```

Come si può evincere dal Listato 2.8, ogni dichiarazione di variabile è seguita da una inizializzazione in VB6. Il Listato 2.9 mostra quanto il codice Visual Basic .NET sia più conciso.

Listato 2.9 *Il codice VB6 del Listato 2.8, riscritto per VB .NET.*

```

1: Private Sub button4_Click(ByVal sender As System.Object, _
2:     ByVal e As System.EventArgs) Handles button4.Click
3:
4:     Dim I As Integer = 5
5:     Dim S As String = "Jello Mold"
6:     Dim D As Date = Now()
7:     Dim F As Double = D.ToOADate()
8:     Dim S1 As String = "Some More Text"
9:     Debug.WriteLine(S1)
10:
11: End Sub

```

La linea 1 evidenzia la gestione degli eventi in particolare del button-click. L'argomento EventHandler è trattato in dettaglio nel Capitolo 8.



Un ottimale stile di programmazione prevede l'inizializzazione di una variabile al momento della dichiarazione.

Il codice dalla linea 4 alla 9 risponde allo stesso compito del Listato 2.8 scritto in VB6. Sebbene sia possibile dividere dichiarazione e inizializzazione anche in Visual Basic .NET, a meno che non ci siano particolari motivazioni è meglio eseguire le due operazioni in un'unica istruzione.

Dichiarazione di più variabili contemporaneamente

Come detto, in VB6 la dichiarazione multipla non produce i risultati che ci si aspetterebbero. In aggiunta all'abbandono dei tipi Variant, Visual Basic .NET supporta la dichiarazione multipla di variabili. Seguono due listati con dichiarazioni: nel primo in VB6 e nel secondo in Visual Basic .NET.

Listato 2.10 *Codice VB6 che dichiara più variabili in una singola istruzione.*

```

1: Private Sub Command2_Click()
2:
3:     ' Dichiarazione multipla di variabili in VB6
4:     Dim I, J, K As Integer
5:
6:     I = 5
7:     J = "Oops!"
8:     K = 15
9:
10: End Sub

```

Nel Listato 2.10, I e J potrebbero sembrare variabili di tipo `Integer` ma in realtà sono `Variant`. Solo K è un `Integer`. Come si può vedere nella riga 7 si può assegnare a J una stringa, cosa che in realtà non avremmo voluto, la cosa auspicabile era che il compilatore segnalasse l'errore. In realtà questo non succede perché J è di tipo `Variant`. Nel Listato 2.9 è riportato il codice Visual Basic .NET equivalente.

Listato 2.11 *Dichiarazione multipla di variabili in VB .NET.*

```

1: Option Strict On
2:
3: Module Module3
4:
5:     Sub MultipleVariables()
6:
7:         ' Dichiarazione multipla di variabili in VB6
8:         Dim I, J, K As Integer
9:
10:        I = 5
11:        J = "Oops!" ' Causa errore di compilazione
12:        K = 15
13:
14:    End Sub
15:
16: End Module

```



Il codice nel Listato 2.11 intenzionalmente non può essere compilato. Il compilatore segnalerà che la dichiarazione `Option Strict` non permette una conversione implicita da `String` a `Integer`. Questo implica che I, J e K sono di tipo `Integer` mentre nel Listato 2.10 in VB6 solo K era di tipo `Integer`.

Se si vuole che il compilatore individui il numero più alto possibile di errori così da ridurre sensibilmente il numero di errori da catturare a runtime è auspicabile che si imposti su `On` sia `Option Explicit` che `Option Strict`. Scrivendo codice esplicito si abilita il compilatore a svolgere molto lavoro al nostro posto. Ricordate dunque che `Option Explicit` deve essere impostato su `On` per far sì che gli assegnamenti di `String` a `Integer` siano rilevati; al contrario questi errori saranno segnalati a runtime con un `InvalidCastException`.

Inizializzazioni multiple non consentite

Sebbene Visual Basic .NET supporti la dichiarazione multipla di variabili non è possibile eseguirne l'inizializzazione multipla. Il frammento di codice che segue

```
Dim I , J , K As Integer = 3, 4, 5
```

non può essere eseguito senza generare errore, nemmeno utilizzando qualche altra sintassi.

Definizione di costanti

Quando si definiscono costanti in Visual Basic .NET con la direttiva `Option Strict On` è obbligatorio includere il tipo di dato. In VB6 il tipo di dato era determinato sulla base del

primo assegnamento; Visual Basic .NET determinerà il tipo di dato in automatico solo se non è presente la dichiarazione `As` e `Option Strict` è `Off`. Il primo frammento di codice mostra una dichiarazione di costante in VB6 e il secondo il corrispondente codice in Visual Basic .NET.

```
' VB6 Code
Const ADate = #12:00:00 AM#
Debug.Print ADate
```

Segue il codice VB .NET equivalente:

```
' VB.NET Code
Const ADate As DateTime = #12:00:00 AM#
Debug.WriteLine(ADate)
```

La differenza più grande è la presenza del tipo di dati `As DateTime` in Visual Basic .NET. Notate che per inizializzare una costante sia VB6 che Visual Basic .NET utilizzano una costante: il valore non può derivare da una funzione.

Istanziare oggetti

Per una discussione esauriente sui principi della programmazione a oggetti potete fare riferimento al Capitolo 7. Questo paragrafo contiene una breve introduzione dell'argomento: si esaminano alcuni semplici concetti prima di parlare della creazione di oggetti.

Una classe è la descrizione di un'entità. Le classi indicano quali metodi, proprietà e campi si possono trovare in istanze di un certo tipo. Una metaclassa è una variabile di un tipo classe. Le metaclassi sono restituite dal metodo polimorfico `GetType()` introdotto con la classe `Object`. Quando si utilizza una classe come se fosse un oggetto ci si riferisce alla classe come metaclassa. Istanza e oggetto sono sinonimi. Un'istanza, o oggetto, si crea dichiarando una variabile il cui tipo sia una classe; di conseguenza viene allocata memoria per la variabile; questa operazione è detta istanziare un oggetto o creare un'istanza di oggetto. Gli oggetti si creano usando la dichiarazione `New`. La procedura è simile a quella utilizzata in VB6; tuttavia le classi in Visual Basic .NET hanno costruttori. Un costruttore è un metodo il cui compito è inizializzare l'oggetto. Gli oggetti in Visual Basic .NET vengono creati con codice simile al seguente:

```
Dim List As New Collection()
```

Il frammento di codice dichiara e inizializza un'istanza della classe `Collection`. La classe è `Collection` e l'oggetto è `List`. Si notino le parentesi tonde dopo il nome della classe, o meglio del costruttore che appunto è una funzione e vuole le parentesi.

I costruttori parametrizzati sono costruttori che accettano parametri. In molti linguaggi il costruttore ha lo stesso nome della classe, in altri ha nomi particolari per convenzione. In Visual Basic .NET i parametri sono passati tra le parentesi che seguono il nome della classe. Per esempio, il tipo `DateTime` è un `ValueType` che a sua volta è una sottoclasse della classe `Object`. Quindi è possibile utilizzare l'overloading del costruttore per gli oggetti `DateTime` come si può notare di seguito.

```
Dim MyDate As New DateTime(1966, 2, 12)
Debug.WriteLine(MyDate)
```

Si possono notare sia la parola `New` che i parametri passati al costruttore `DateTime`. In questo caso `DateTime` possiede ben 7 costruttori di tipo `overloading`. Un metodo letteralmente 'sovraccaricato' è un metodo nella stessa classe che ha lo stesso nome ma si distingue dagli altri per i parametri passati. L'`overloading` non era supportato in VB6.

Liste di inizializzatori

Il suggerimento di assegnare valori iniziali alle variabili vale anche per tipi di dati complessi. Nel caso delle classi il compito è svolto dai costruttori. Anche le strutture possono avere costruttori per cui il problema si risolve agilmente.

La dichiarazione e l'inizializzazione di vettori è leggermente più complessa che per una variabile tipo `Integer`. Le due operazioni però possono essere fatte in una singola istruzione in Visual Basic .NET. Vediamo come:

```
Dim MyArray() As Integer = {1, 2, 3, 4}
```

Il vettore è dichiarato senza limiti e di tipo `Integer`. La lista di inizializzazione costruisce il vettore per elementi: 1, 2, 3, 4. Non si possono specificare valori di inizializzazione se nella dichiarazione è specificata la dimensione. Per esempio, la dichiarazione seguente non è valida:

```
Dim MyArray(4) As Integer = {1, 2, 3, 4}
```

I vettori `Array` sono classi definite nel namespace `System.Array`.

Operatori

Gli operatori sono simboli speciali come `+`, `/`, `Mod`, e `AndOr` che eseguono operazioni aritmetiche, bit a bit, comparazioni, concatenamenti e operazioni logiche. Ogni operatore ha un determinato numero e tipo di operandi o dati.

Sono stati aggiunti molti operatori, alcuni di questi svolgono operazioni sui bit o logiche, altri permettono forme abbreviate delle operazioni aritmetiche. Alcuni operatori di assegnamento sono stati presi in prestito dal C++. Nella Tabella 2.8 è riportato un elenco degli operatori disponibili in Visual Basic .NET.

Tabella 2.8 *Operatori disponibili in Visual Basic .NET.*

<i>Azione</i>	<i>Simbolo</i>	<i>Descrizione</i>
<i>Operatori binari</i>		
Esponenziale	<code>^</code>	Scritto <code>x^y</code> ; eleva <code>x</code> alla potenza <code>y</code>
Sottrazione	<code>-</code>	<code>x-y</code> ; esegue la sottrazione e come operatore unario la negazione
Moltiplicazione	<code>*</code>	<code>x*y</code> ; esegue la moltiplicazione

(segue)

Tabella 2.8 (continua) *Operatori disponibili in Visual Basic .NET.*

<i>Azione</i>	<i>Simbolo</i>	<i>Descrizione</i>
Divisione Floating-point	/	x/y ; divisione floating-point restituisce un Double, Single se entrambi gli operandi sono Byte, Integer, or Single; Decimal se entrambi gli operandi sono Decimal
Divisione Integer	\	$x \setminus y$; divisione tra interi
Divisione Modulo	Mod	$x \text{ Mod } y$; restituisce il resto della divisione di x per y
Somma	+	$x + y$; somma di x e y
Assegnamento	=	$x = y$; assegna il valore di y a x
Esponenziale e assegnamento	^=	$x \wedge y$; eleva x alla potenza y e assegna il risultato a x
Moltiplicazione e assegnamento	*=	$x * y$; moltiplicazione di x e y e assegna il risultato a x
Divisione Floating-point e assegnamento	$x /= y$	divisione floating-point di x per y e assegnamento a x
Divisione Integer e assegnamento	\=	$x \setminus y$; divisione di x per y e assegnamento del risultato a x
Somma e assegnamento	+=	$x += y$; somma di x e y e assegnamento del risultato a x
Sottrazione e assegnamento	-=	$x -= y$; sottrazione di x da y e assegnamento del risultato a x
Concatenazione e assegnamento	&=	$x \&= y$, dove x e y sono stringhe; concatenazione di x e y e assegnamento a x
Uguaglianza	=	$x = y$; testa l'uguaglianza
Disuguaglianza	<>	$x <> y$; teste la diversità
Minore di	<	$x < y$; testa se x è minore di y
Maggiore di	>	$x > y$; testa se x è maggiore di y
Minore o uguale	<=	$x <= y$; testa se x è minore o uguale a y
Maggiore o uguale	>=	$x >= y$; testa se x è maggiore o uguale a y
Like	Like	string Like pattern; compara l'argomento string all'argomento pattern.
Is	Is	object1 is object2; testa se object1 e object2 fanno riferimento allo stesso oggetto
Concatenazione	&	string1 & any_expression; converte l'argomento a dx in stringa e lo concatena al sx

Concatenazione	+	string1 + string2; concatena due stringhe; da errore se un argomento non è stringa
And	And	Bool1 And bool2 o x And y; logico And per Booleans e bit And per integrali
Or	Or	Bool1 Or bool2 e x Or y; logico Or per Booleans e bit Or per integrali
Xor	Xor	Bool1 Xor bool2 o x Xor y; logico Or-esclusivo per Booleans e bit Or-esclusivo per integrali
AndAlso	AndAlso	And cortocircuitato
OrElse	OrElse	Or cortocircuitato
<i>Operatori unari</i>		
AddressOf	AddressOf	Restituisce l'indirizzo di una procedura; per delegates in VB .NET
GetType	GetType	GetType(any); restituisce il tipo runtime
Not	Unario	Not bool or Not x; negazione logica per Booleans e negazione bit per integrali

Molti di questi operatori sono noti e molti svolgono nuovi compiti. Vediamo di seguito una delucidazione sull'argomento. Tutti gli operatori della forma `operando=` compiono due operazioni. `AddressOf` assume nuova importanza in Visual Basic .NET. In VB6 questo operando è utilizzato per passare l'indirizzo di una procedura a una API che necessita di una procedura `Callback`. Una `Callback` è semplicemente una procedura richiamata tramite il suo indirizzo. In Visual Basic .NET si possono creare `callback` personalizzate e utilizzarle nel programma, sono supportate direttamente dai `Delegates`. Si veda il Capitolo 9 per maggiori dettagli.

Funzioni per la conversione dei tipi

Visual Basic .NET include moltissime funzioni per la conversione di tipi. Avrete notato in precedenza l'introduzione del metodo `ToString`, metodo della classe `Object`. Poiché ogni classe e struttura come detto deriva dalla classe `Object`, ogni tipo implementerà un metodo `ToString`. Ogni tipo possiede anche un metodo che prende un argomento di tipo stringa e lo converte in un tipo specifico. Per esempio, nel tipo `DateTime`, il metodo `Parse` converte una stringa a `DateTime`. Le due dichiarazioni seguenti mostrano la conversione da stringa a `DateTime` e viceversa.

```
Dim ADate As DateTime = DateTime.Parse("12:00:00 AM")
Dim AString As String = ADate.ToString()
```

Molti dei tipi di conversione di VB6 sono stati riportati anche in Visual Basic .NET; alcuni sono stati rimpiazzati per supportare anche altri tipi. In Visual Basic .NET il tipo degli argomenti nelle conversioni è un oggetto; ciò significa che si possono utilizzare argomenti

tipo sottoclassi o qualunque oggetto che abbia senso per quel tipo specifico di conversione. La Tabella 2.9 mostra un elenco delle funzioni di conversione VB6 e delle corrispondenti in Visual Basic .NET.

Tabella 2.9 *Funzioni di conversione di VB6 e loro sostituzioni in VB .NET.*

<i>VB6</i>	<i>VB .NET</i>	<i>Descrizione</i>	<i>Intervallo</i>
CBool	CBool	Converte Objet in Boolean	Stringa valida o espressione numerica
CByte	CByte	Converte Objet in Byte	Da 0 a 255
None	CChar	Converte Objet in Char	Da 0 a 65535
CCur	CDec	Converte Objet in Decimal	Vedere CDec
CDate	CDate	Converte Objet in DateTime	Qualunque data o ora valida
Cdbl	Cdbl	Converte Objet in Double	Qualunque valore Double valido (vedere Tabella 2.3)
CDec	CDec	Converte Objet in Decimal	Qualunque valore Decimal valido (vedere Tabella 2.3)
CInt	CInt	Converte Objet in Integer	Qualunque valore Integer valido (vedere Tabella 2.3)
CLng	CLng	Converte Objet in Long	Qualunque valore Long valido (vedere Tabella 2.3)
CSng	CSng	Converte Objet in Single	Qualunque valore Single valido
CStr	CStr	Converte Objet in String	Qualunque String valida
CVar	CObj	Converte Objet in Object	Qualunque valore
----	CShort	Converte Objet in Short	Qualunque Short (vedere Tabella 2.3)

Come si può notare molte funzioni VB6 sono state riprese e altre sono state aggiunte.

Cambiamento degli ambiti di variabili in VB .NET

Visual Basic .NET supporta gli ambiti di variabile a livello di blocco. Generalmente è utile dichiarare le variabili con validità negli ambiti più stretti possibili. Questa regola si può riassumere così: “Non usate variabili globali”. In VB6 una variabile definita in un ciclo For...Next è visibile anche nell’ambito contenente il ciclo, in Visual Basic .NET non è accessibile dall’esterno del ciclo. Il Listato 2.12 mostra il concetto di ambito in VB6 e il Listato 2.13 mostra la restrizione a livello di blocco in Visual Basic .NET.

Listato 2.12 *L’ambito in VB6 e limitato alle procedure.*

```
1: Private Sub Command6_Click()
2:   Dim I As Integer
3:   For I = 1 To 100
```

```
4: Dim D As Integer
5: D = D + 1
6: Next I
7: MsgBox D
8: End Sub
```

In VB6, la linea 7 mostra il valore 100 mettendo in evidenza che D ha validità in tutta la procedura anche se definito nel ciclo For...Next. In Visual Basic .NET l'ambito di validità di D sarebbe il blocco per cui la linea 7 provocherebbe un errore.

Listato 2.13 VB .NET supporta ambito di validità a livello di blocco.

```
Sub BlockScope()
    Dim I, D As Integer
    For I = 1 To 100
        D = D + 1
    Next I
    MsgBox
End Sub
```

Per rendere D accessibile anche fuori dal blocco la variabile deve essere definita fuori dal ciclo For...Next. Ogni variabile definita in un ambito è visibile negli ambiti interni ma non in quelli più esterni all'ambito di definizione. Per esempio la variabile D del listato 2.13 è visibile nell'ambito For...Next che è interno, ma non all'esterno della procedura.

Dichiarazioni per il controllo del flusso

Alcune delle istruzioni del controllo del flusso di esecuzione sono state riviste. L'istruzione GoSub non è più supportata. La dichiarazione Call può essere utilizzata ma non è più necessaria ed è destinata a scomparire nelle prossime versioni. On Gosub e On Goto non sono più supportate. On Error GoTo è stata mantenuta ma è preferibile l'utilizzo della gestione degli errori Exception Handling,; per maggiori informazioni si veda più avanti nel capitolo. La dichiarazione While...Wend è stata mantenuta ma con un cambio di semantica, Wend è stato sostituito da End While. Nel listato seguente vi è un esempio dell'utilizzo del costrutto.

Listato 2.14 While...End While sostituisce While...Wend.

```
1: Sub WhileEndWhileTest()
2:
3:     Dim I As Integer = 1
4:     While (I < 10)
5:         I += 1
6:         Debug.WriteLine(I)
7:     End While
8:
9: End Sub
```

Il comportamento del ciclo non è cambiato. Esegue zero o più volte il corpo delle istruzioni; quando si digita While in automatico l'IDE aggiunge per noi End While completando il blocco.

Vettori e collezioni

I vettori “Array” non hanno subito cambiamenti sostanziali. Si dichiarano come segue:

```
Dim A(10) As Type
```

Il vettore A contiene n+1(11) elementi indicizzabili da 0 a n (10). Si possono utilizzare istruzioni per la gestione degli indici Lbound e Ubound. È preferibile l'utilizzo dei due metodi Array.GetLowerBound e Array.GetUpperBound. Il tipo di dati Collection è definito per mantenere la compatibilità con VB6.

Sono stati aggiunti anche altri tipi di dati astratti come vedremo in seguito.

Vettori con indici definiti

VB6 permetteva la dichiarazione degli estremi degli indici come si può notare nel codice VB6 che segue

```
Dim I(3 To 5) As Integer
```

In Visual Basic .NET questa possibilità è stata eliminata.

Vettori a N dimensioni

In Visual Basic .NET si possono definire vettori multidimensionali. Per dichiarare vettori di questo tipo si devono esprimere le dimensioni come una serie di numeri delimitati da virgole. Ricordate che ogni dimensione contiene n+1 elementi.

Visual Basic .NET supporta vettori fino a 60 dimensioni. Un esempio di dichiarazione di vettore bidimensionale.

```
Dim Doubles(10, 10) As Double
```

Nel codice si dichiara un vettore di Double di dimensione 11x11. I vettori sono sottoclassi di System.Array implementando alcune caratteristiche che ne permettono una facile gestione. Il Listato 2.15 mostra l'indicizzazione di un vettore bidimensionale.

Listato 2.15 *Indicizzazione di un vettore bidimensionale.*

```
1: Sub TestArray()
2:   Dim Doubles(10, 10) As Double
3:
4:   Dim I, J As Integer
5:   For I = Doubles.GetLowerBound(0) To Doubles.GetUpperBound(0)
6:
7:     For J = Doubles.GetLowerBound(1) To Doubles.GetUpperBound(1)
8:
9:       Doubles(I, J) = I * J
10:      Debug.WriteLine(Doubles(I, J))
11:
12:    Next
13:  Next
14:
15: End Sub
```

Il vettore bidimensionale nel listato è utilizzato in un ciclo For; per mezzo dei nuovi metodi `GetLowerBound` e `GetUpperBound` (linee 5 e 7) è possibile determinare il limite inferiore e superiore rispettivamente del vettore, è preferibile l'utilizzo dei metodi della classe `Array` all'utilizzo delle vecchie funzioni `Lbound` e `Ubound`.

Ridimensionamento dei vettori

Come in VB6 i vettori possono essere ridimensionati dinamicamente tramite `ReDim`. Per preservare i dati preesistenti basta includere la dichiarazione `Preserve`. Questa parola indica di mantenere invariati i dati relativi alla dimensione più a destra. Il codice che segue

```
ReDim Preserve Doubles(10, 100)
```

esegue il ridimensionamento della colonna a destra (che ora ha dimensione 100) mantenendo invariati i dati contenuti nelle prime dieci colonne. La dichiarazione `Preserve` va posta dopo `ReDim` ma ricordate che è possibile variare solo la dimensione più a destra del vettore. La dichiarazione seguente

```
ReDim Preserve Doubles(5, 100)
```

non è valida e restituisce un errore in esecuzione `ArrayTypeMismatchException`.

Funzioni che restituiscono vettori

In Visual Basic .NET una funzione può restituire vettori, come in VB6. Per farlo, basta definire una funzione che abbia come tipo restituito un vettore di qualche tipo e assegnare la variabile vettore nella dichiarazione `Return`.

Listato 2.16 *Restituzione di un vettore da parte di una funzione.*

```
1: Function GetArray() As Byte()  
2:  
3:     Dim Bytes() As Byte = {0, 1, 2, 3, 4, 5}  
4:     Return Bytes  
5:  
6: End Function  
7:  
8: Sub TestArray()  
9:     Dim I As Integer  
10:    Dim Bytes() As Byte = GetArray()  
11:  
12:    For I = Bytes.GetLowerBound(0) To Bytes.GetUpperBound(0)  
13:        Debug.WriteLine(Bytes(I))  
14:    Next  
15: End Sub
```

Alla linea 10 viene richiamata la funzione `GetArray` che restituisce un vettore utilizzato per inizializzare una variabile locale `Bytes`. Al solito i metodi `GetLowerBound` e `GetUpperBound` sono utilizzati per calcolare il limite inferiore e superiore del vettore. Vi sono anche due nuove caratteristiche: nella linea 3 è mostrato il metodo di assegnamento diretto tramite enumerazione e nella linea 4 l'utilizzo della funzione `Return`.

I vettori sono sottoclassi della classe `System.Array`

Nella Tabella 2.10 sono riportati e descritti i membri ereditati dalla classe `Array`.

Tabella 2.10 *Membri della classe `Array` ereditati dai vettori.*

<i>Nome</i>	<i>Descrizione</i>
<i>Metodi condivisi</i>	
<code>BinarySearch</code>	Si usa per cercare elementi in un vettore monodimensionale ordinato
<code>Clear</code>	Imposta un intervallo di elementi a <code>Nothing</code>
<code>Copy</code>	Copia un intervallo di elementi da un vettore a un altro
<code>CreateInstance</code>	Crea un vettore di dimensioni e tipo specificato
<code>IndexOf</code>	Cerca in un vettore monodimensionale, restituendo l'indice della prima corrispondenza
<code>LastIndexOf</code>	Trova l'indice dell'ultima istanza di una corrispondenza
<code>Reverse</code>	Inverte l'ordine di un vettore
<code>Sort</code>	Ordina un elemento
<i>Proprietà</i>	
<code>IsFixedSize</code>	Restituisce <code>False</code> a meno che non ci sia overloading di una sottoclasse
<code>IsReadOnly</code>	Restituisce <code>False</code> a meno che non ci sia overloading di una sottoclasse
<code>IsSynchronized</code>	Indica se un vettore è thread-safe
<code>Length</code>	Restituisce il numero di elementi in un vettore
<code>Rank</code>	Restituisce il numero di dimensioni di un vettore
<code>SynchRoot</code>	Restituisce una versione sincronizzata del vettore per un accesso thread-safe
<i>Metodi</i>	
<code>Clone</code>	Restituisce una copia di un vettore; oggetti contenuti sono riferiti non copiati
<code>CopyTo</code>	Copia elementi di un vettore monodimensionale in una posizione specifica di un altro vettore
<code>GetEnumerator</code>	Restituisce un <code>IEnumerator</code> (si veda il Listato 2.5 per un esempio di <code>Enumerator</code>)
<code>GetLength</code>	Restituisce il numero di elementi nella dimensione specificata
<code>GetLowerBound</code>	Restituisce il limite inferiore della dimensione specificata
<code>GetUpperBound</code>	Restituisce il limite superiore della dimensione specificata
<code>GetValue</code>	Restituisce il valore indicato dall'argomento
<code>Initialize</code>	Initializza elementi del vettore chiamando il costruttore di default
<code>SetValue</code>	Imposta il valore di un vettore indicato dall'indice passato come argomento

Metodi protetti e metodi ereditati dagli oggetti non sono riportati. Ne esistono e si possono vedere nell'Help.

Tipi di dati astratti

Visual Basic .NET definisce tipi di dati astratti che offrono varie possibilità di gestione dei dati (ADT): `ArrayList`, `BitArray`, `Dictionary`, `HashTable`, `Queue`, `SortedList`, `Stack`, e `StringCollection`. Questi tipi sono troppo numerosi per essere discussi in questo capitolo per cui verranno incorporati nel codice di alcuni esempi che troveremo per darne una spiegazione dettagliata.

Ecco alcune brevi definizioni:

- Il tipo `Collection` non fa parte del CLR ma è stato incorporato nel namespace `Microsoft.VisualBasic` per aiutare i programmatori VB6.
- Il tipo `ArrayList` è un vettore intelligente con capacità integrate di gestione dinamica.
- `BitArrays` rappresenta i bit come vettore di `True` e `False`.
- Un `Dictionary` fornisce una classe astratta per coppie (`Chiave, Valore`), il registro ne è un esempio.
- `HashTable` immagazzina elementi in modo da renderne l'accesso veloce e conveniente.
- `Queue` è una coda di dati ADT del tipo FIFO.
- `Stack` è una pila di tipo LIFO.
- `StringCollection` è un vettore specifico per contenere stringhe.

Il Listato 2.17 mostra le capacità di base del tipo di dati `Stack` che supporta operazioni di tipo `Push/Pop`, si aggiunge un elemento in cima alla pila `Push` e si prende un elemento dalla cima `Pop`.

Listato 2.17 *Capacità di base dello Stack.*

```
1: Sub DemoStack()  
2:   Dim MyStack As New Stack()  
3:  
4:   Dim I As Integer  
5:   For I = 102 To 65 Step -1  
6:     MyStack.Push(I)  
7:   Next  
8:  
9:   While (MyStack.Count > 0)  
10:    Debug.WriteLine(MyStack.Pop())  
11:  End While  
12:  
13: End Sub
```

La linea 2 crea un istanza di `Stack`, definita in `System.Collections`. Nel ciclo `For` i numeri dal 102 al 65 sono messi nella pila. L'operazione `Push` ha come argomento un `Object` così da poter immagazzinare dati di qualunque tipo. Il ciclo `While` esegue operazioni di `Pop` sugli elementi

del vettore. Si può notare come il codice produca in uscita una lista di elementi dal 65 al 102, inversa all'ordine di inserimento, come atteso dal funzionamento LIFO.

Gestione degli errori strutturata

Visual Basic .NET introduce il concetto dell'`Exception Handling`, per migliorare la capacità di gestione degli errori, eccezioni in fase di esecuzione. Questo sistema di gestione è molto più efficiente del classico `On Error GoTo` ed è stato introdotto per rendere più robuste le applicazioni. Sono stati introdotti i costrutti `Try...Catch` che intercetta le eccezioni e `Try...Finally` per proteggere le risorse. Nel Capitolo 3 l'argomento verrà trattato in modo esaustivo.

Uso di parole riservate

Visual Basic .NET permette l'utilizzo di parole riservate nel codice. Se accidentalmente si utilizza una parola riservata, come per esempio `Enum`, come nome di variabile, l'IDE pone parentesi quadre attorno al nome permettendone l'utilizzo. Naturalmente è un pessimo stile di programmazione, fortemente sconsigliabile.

Compatibilità tra VB6 e VB .NET

Non è facile certamente migrare un'applicazione da VB6 a Visual Basic .NET. Il namespace `Microsoft.VisualBasic` include elementi rilevanti per garantire la compatibilità tra i due linguaggi. Se si apre un progetto VB6 con Visual Basic .NET il wizard di migrazione viene avviato e utilizza gli oggetti di `Microsoft.VisualBasic` per renderlo compatibile. In realtà conviene riscrivere il codice; due sono i vantaggi tangibili: si imparerà l'utilizzo di Visual Basic .NET e si renderà il codice più snello ed efficiente. Nell'Appendice A si trovano riferimenti alle caratteristiche che sono state riviste, sostituite o rimosse in Visual Basic .NET.