8

Matrici

Le **matrici** sono molto importanti per la **programmazione** e la loro conoscenza è **essenziale** per strutturare meglio un **programma complesso**.

Le matrici sono fondamentali in ogni linguaggio di programmazione ed è per questo che il concetto di matrice ha fatto la sua comparsa nella programmazione già dai primi linguaggi, molto tempo fa.
Una delle caratteristiche principali che rendono le matrici così interessanti è la velocità: infatti, i dati delle matrici sono memorizzati nella RAM che è una memoria velocissima. Recuperare dati dalla RAM è un'operazione più veloce, rispetto al recupero dal disco fisso, poiché in quest'ultimo caso c'è un rallentamento dovuto alle operazioni meccaniche del disco: soprattutto, il posizionamento delle testine e la rotazione del disco, per fare in modo che sotto la testina passino i dati desiderati. Inoltre, se i dati sono distribuiti su più tracce, i tempi di attesa aumentano ulteriormente.
In questo capitolo, esamineremo alcune tecniche per la definizione e la gestione di determinati tipi di matrici.

Le matrici

Il termine italiano matrice equivale al termine inglese **array**.
In questa sede non tratteremo la matrice dal punto di vista rigorosamente matematico, ci basterà sapere che una matrice:

- deve essere dichiarata e, quindi, assegnata a una variabile;
- può avere una o più dimensioni, fino a 32: se ha una unica dimensione è un

vettore (matrice unidimensionale). Nella dichiarazione della matrice devono essere indicate espressamente il numero di dimensioni e il numero di elementi che formano ciascuna dimensione;

- può essere vista come un insieme di celle ordinate, ognuna delle quali può contenere un dato elementare di un certo tipo o un oggetto, nel senso della programmazione orientata agli oggetti;
- permette di fare riferimento all'elemento contenuto in una determinata posizione mediante un riferimento all'elemento *i*-esimo, nel caso di una matrice unidimensionale, o all'elemento espresso con un indice per ciascuna delle dimensioni della matrice stessa. Per esempio, è tipico il caso di una matrice bidimensionale (due dimensioni), come una tabella, dove un elemento qualsiasi ha un riferimento dato da un numero di riga e da un numero di colonna della matrice;
- il tipo di dato degli elementi contenuti nella matrice è lo stesso per tutti gli elementi ed è definito in base al tipo di dato stabilito con la dichiarazione della matrice. Per esempio, una matrice di tipo `Integer` conterrà solo elementi `Integer`.

Dichiarazione di matrici

Possiamo dichiarare una matrice come indicato nel seguente esempio:

```
Dim giorni(100) As String
```

In questo caso, definiamo una variabile array di nome `dipendenti`, di tipo `String` e con un numero di elementi che va dall'indice `0` all'indice `100` (cioè 101 elementi). Non assegniamo alcun valore alle varie posizioni dell'array, ma semplicemente creiamo il "contenitore" di elementi del tipo indicato.
Per assegnare un elenco di valori all'array, possiamo utilizzare anche la seguente sintassi:

```
' Esempio: 08.01
Dim giorni() As String
giorni = New String() {"Lunedì", "Martedì",
         "Mercoledì", "Giovedì", "Venerdì",
         "Sabato", "Domenica"}
```

Nel primo esempio dichiariamo una variabile giorni come matrice di stringhe, senza specificare il numero di elementi dell'array e senza assegnare alcun valore agli elementi.

Con la seconda istruzione assegniamo alla variabile giorni un nuovo array di stringhe con il contemporaneo inserimento dei nomi dei giorni della settimana. Non precisiamo il numero di elementi da cui deve essere composto l'array perché l'informazione è già implicita nel numero di elementi specificati.
Avremmo potuto anche assegnare direttamente i valori dei singoli elementi direttamente nella dichiarazione dell'array:

```
Dim giorni2() As String = {"Lunedì", "Martedì",
            "Mercoledì", "Giovedì", "Venerdì",
            "Sabato", "Domenica"}
```

Una forma semplificata di definizione di un'array con assegnazione dei valori, è quella che prevede l'utilizzo di una importante caratteristica: l'**inferenza** di tipo. Per esempio, un array di numeri interi può essere dichiarato in forma estesa, in questo modo:

```
Dim Array1() As Integer = {1, 2, 3, 4, 5}
```

Con l'inferenza, l'istruzione può diventare così:

```
Dim Array2() = {1, 2, 3, 4, 5}
```

La stessa cosa si può fare con un array di stringhe o di qualsiasi altro tipo:

```
Dim strArray() = {"Dieci", "Venti", "Trenta"}
```

Attenzione! Ci sono dei casi in cui la definizione non è valida, cioè quando si vogliono inserire dati di tipo diverso nello stesso array. Questa istruzione, per esempio, provocherà un errore perché non è possibile capire se gli elementi devono essere numeri o stringhe:

```
Dim varArray = {10, "Dieci"}
```

Naturalmente, presumiamo sempre che Option Strict sia impostato a On, per un maggiore controllo sui tipi di dati. Nel caso in cui fosse impostato a Off, infatti, l'istruzione sopra indicata sarebbe accettata e il tipo di dato dell'array sarebbe impostato a Object (fortemente sconsigliato!).

Nel prossimo esempio, dichiariamo una variabile dipendenti di tipo stringa, specificando un numero di elementi da zero a cento (quindi, 101 elementi).

Il numero segnalato, infatti, non indica il numero totale di elementi definiti, ma l'indice massimo. Tenendo conto che il primo elemento ha sempre un indice pari a zero, il numero di elementi dell'array è pari al numero di indice massimo dell'array più uno.

```
' Esempio: 08.02
Dim dipendenti(100) As String
dipendenti(0) = "Mario Rossi"
dipendenti(1) = "Luigi Neri"
dipendenti(2) = "Giuseppe Verdi"
dipendenti(100) = "Flavio Marrone"
' dipendenti(101) = "Lucio Bianchi" ' <-- Errore!
```

L'ultima istruzione, trasformata in commento per mezzo dell'apice singolo iniziale (e quindi non eseguibile), solleva un'eccezione poiché fa riferimento a un indice di colonna non valido (superiore al numero di indice massimo della relativa dimensione). Questo ulteriore esempio definisce una matrice bidimensionale di elementi del tipo `Double`. Nel caso specifico, la matrice è formata da sei righe (da zero a cinque) e da nove colonne (da zero a otto).

```
' Esempio: 08.03
Dim matrice(5, 8) As Double
matrice(0, 0) = 1.5
matrice(0, 1) = 2.0
matrice(0, 2) = 2.5
matrice(1, 0) = 0.5
matrice(2, 0) = 0.3
' matrice(4, 9) = 0.0     ' <-- Errore!
```

Le considerazioni fatte prima valgono anche in questo caso: se l'ultima istruzione venisse eseguita, otterremmo un errore, perché la seconda dimensione dell'array non arriva all'indice 9.
Seguono alcune istruzioni di assegnazione di valori `Double` a singoli elementi della matrice, sempre specificando gli indici di riga e di colonna.
Per specificare i valori da inserire già nella dichiarazione dell'array a più dimensioni, si può utilizzare la seguente sintassi:

```
Dim matrice2(,) As Integer = {{0, 1, 2},
                              {10, 11, 12}}
Console.WriteLine(matrice(1, 2))
Console.ReadLine()
```

In questo caso sono state dichiarate due dimensioni, cioè due righe e tre colonne. Se vogliamo estrarre l'elemento `(1, 2)`, troviamo che corrisponde al valore `12`: infatti, si tratta dell'elemento di riga uno (quindi, il gruppo `{10, 11, 12}`) e di colonna due, proprio il valore `12`.

È possibile assegnare un valore a una determinata posizione della matrice indicando tra parentesi l'indice dell'elemento corrispondente:

```
' Esempio: 08.04
Dim totali(0 To 20) As Integer
totali(0) = 100
totali(20) = 150
' totali(21) = 0     ' <-- Errore!
```

In questo esempio è stata dichiarata una matrice di ventuno elementi di tipo `Integer`, da zero a venti. Notate che il riferimento al numero di elementi definiti è specificato con un intervallo di validità. Il valore minimo, zero, non è però modificabile.
L'indicazione del valore minimo (sempre zero) può essere utile per una maggiore chiarezza del codice.
L'indice dell'array è molto utile, soprattutto, quando abbiamo a che fare con molti elementi e non possiamo scrivere, per ognuno, un'istruzione di assegnazione e un'istruzione per leggerne il contenuto.
Infatti, in questi casi si utilizza un ciclo, cioè un'istruzione per scorrere tutti o parte degli elementi contenuti nella matrice.

Potrete incontrare le istruzioni di ciclo nel capitolo relativo alle spiegazioni sul linguaggio di programmazione.

Naturalmente, nel caso di matrici bidimensionali o multidimensionali occorre indicare tra parentesi tutti gli indici, separati da virgola:

```
Dim totali2(20, 10) As Integer
totali2(0, 5) = 100
totali2(20, 1) = 150
```

Alcuni esempi nell'uso delle matrici

Il primo esempio è il seguente:

```
' Esempio: 08.05
Dim i As Integer             ' variabile di ciclo
Dim out As String = ""       ' risultato da visualizzare
Dim arrMesi() As String = {"Gennaio", _
    "Febbraio", "Marzo", "Aprile", "Maggio", _
    "Giugno", "Luglio", "Agosto", "Settembre", _
    "Ottobre", "Novembre", "Dicembre"}
For i = 0 To 11
```

```
    out &= arrMesi(i)
    ' aggiunge un ritorno a capo:
    out &= Environment.NewLine
Next
Console.WriteLine(out)
Console.ReadLine()
```

Le variabili `i` e **out**, definite nelle prime due istruzioni, sono rispettivamente la variabile di incremento del ciclo e la variabile stringa che conterrà il risultato da visualizzare.

```
Dim arrMesi() As String = {"Gennaio", _
    "Febbraio", "Marzo", "Aprile", "Maggio", _
    "Giugno", "Luglio", "Agosto", "Settembre", _
    "Ottobre", "Novembre", "Dicembre"}
```

Questa dichiarazione definisce una matrice di nome `arrMesi` e le assegna il nome di tutti i mesi dell'anno.

```
For i = 0 To 11
```

L'istruzione `For` costituisce l'inizio del ciclo che scandisce tutti gli elementi della matrice, da `0` a `11`.

```
    out &= arrMesi(i)
```

L'operatore di assegnamento `&=` accoda al contenuto della variabile `out` la stringa contenuta nell'elemento `i`-*esimo* della matrice `arrMesi`.

```
    out &= Environment.NewLine
```

Il metodo `NewLine` dell'oggetto `Environment` serve ad aggiungere un carattere di ritorno carrello (ASCII 13 = carriage return) e un carattere di avanzamento linea (ASCII 10 = line feed).

```
Next
```

Con l'istruzione `Next` chiudiamo il blocco `For`. A ogni iterazione del ciclo, finché la variabile `i` non assume un valore superiore al valore massimo stabilito (in questo caso 11), `i` viene incrementato di un'unità e l'esecuzione prosegue dalla riga che contiene l'istruzione `For` per eseguire una nuova iterazione. Alla fine, quando `i` assume il valore superiore al limite superiore (in questo caso `12`), il ciclo `For` termina e l'esecuzione passa all'istruzione successiva.

```
Console.WriteLine(out)
Console.ReadLine()
```

Infine, utilizziamo l'oggetto `Console` per visualizzare l'intero risultato contenuto nella variabile `out`. `WriteLine` scrive quanto indicato all'interno delle parentesi tonde e avanza di una riga.
L'ultima istruzione permette di fermare l'esecuzione del programma in attesa della pressione del tasto **Invio** da parte dell'utente. Se non inseriamo questa istruzione, vedremo aprirsi e chiudersi in rapida successione la finestra della console, senza avere la possibilità di leggerne il contenuto.
Di seguito presentiamo un semplice programma per visualizzare i numeri di un'estrazione del Lotto.

Questo programma ha l'obiettivo di mostrare l'utilizzo delle matrici e delle variabili di ciclo. È più importante capirne il senso generale che le singole istruzioni.

```
' Esempio: 08.06
Module EstrazioneLotto
   Sub Main()
      Dim ruote() As String = {"NAZIONALE   ", _
                               "BARI        ", _
                               "CAGLIARI    ", _
                               "FIRENZE     ", _
                               "GENOVA      ", _
                               "MILANO      ", _
                               "NAPOLI      ", _
                               "PALERMO     ", _
                               "ROMA        ", _
                               "TORINO      ", _
                               "VENEZIA     "}
      ' variabili di ciclo: i (riga), j (colonna)
      Dim i As Integer
      Dim j As Integer
      Dim out As String = ""   ' risultato
      ' generatore numeri pseudo-casuali:
      Dim generator As New Random
      Dim arrEstrazioneLotto(10, 4) As Byte
      For i = 0 To 10
         For j = 0 To 4
            ' estrazione dei 5 numeri:
            arrEstrazioneLotto(i, j) = _
               CType(generator.Next(1, 90), Byte)
         Next
      Next
      For i = 0 To 10
         out &= ruote(i) & "   "
         For j = 0 To 4
            ' prepara stampa di 5 numeri:
```

```
            out &= (arrEstrazioneLotto(i, _
               j).ToString("00")) & " "
         Next
         out &= Environment.NewLine
      Next
      Console.WriteLine(out)   ' scrive risultato
      Console.ReadLine()
   End Sub
End Module
```

Il programma non controlla se nella stessa Ruota viene estratto un numero già uscito. È una semplificazione che abbiamo adottato per non complicare troppo il programma di esempio, poiché dobbiamo ancora vedere molte altre istruzioni del linguaggio Visual Basic.

Analizziamo ora il significato delle istruzioni.

```
Dim ruote() As String = {"NAZIONALE   ", _
                         "BARI        ", _
                         "CAGLIARI    ", _
                         "FIRENZE     ", _
                         "GENOVA      ", _
                         "MILANO      ", _
                         "NAPOLI      ", _
                         "PALERMO     ", _
                         "ROMA        ", _
                         "TORINO      ", _
                         "VENEZIA     "}
```

Questa istruzione definisce una matrice di stringhe associate a una variabile di nome `ruote`. Notate che sono stati aggiunti degli spazi ai nomi delle ruote per poter allineare correttamente i numeri estratti durante la visualizzazione nella console.

```
Dim i As Integer
Dim j As Integer
Dim out As String = ""
```

Queste tre istruzioni definiscono le due variabili di ciclo necessarie alla preparazione del risultato con tutti i numeri estratti che sarà memorizzato nella variabile `out`.

```
Dim generator As New Random
```

`Random` è un oggetto che genera numeri pseudo-casuali, cioè fornisce una sequenza di numeri che soddisfano dei requisiti statistici di casualità. Non è come avere una

vera sequenza di numeri casuali, perché in presenza di un identico stato del sistema genera la stessa sequenza di numeri. Quest'ultima, però, è sufficientemente casuale data l'impossibilità di riprodurre facilmente un identico stato del sistema.

```
Dim arrEstrazioneLotto(10, 4) As Byte
```

Tale istruzione definisce una matrice di `Byte` composta da undici righe per cinque colonne. Le righe rappresentano le dieci ruote più la ruota nazionale, mentre le colonne rappresentano i cinque numeri estratti.
Il formato `Byte`, in questo caso, è appropriato perché `i` numeri estratti hanno un intervallo di validità limitato da `1` a `90`.

```
For i = 0 To 10
```

Con questa istruzione iniziamo il ciclo più esterno, selezionando le righe della matrice da zero a dieci. Questo ciclo esterno, pertanto, viene eseguito undici volte.

```
    For j = 0 To 4
```

Utilizziamo la variabile `j`, invece, per iniziare il ciclo più interno, selezionando le righe della matrice da zero a quattro. Questo ciclo interno, quindi, è eseguito 55 volte (11 x 5).

```
        arrEstrazioneLotto(i, j) = _
            CType(generator.Next(1, 90), Byte)
```

Qui, finalmente, utilizziamo l'oggetto `Generator` per estrarre realmente i numeri. Notate che abbiamo definito un intervallo di validità dei numeri, da un minimo di `1` a un massimo di `90`.
Con il metodo `generator.Next(1, 90)` otteniamo un numero intero che viene poi convertito nel tipo `Byte` con la funzione `CType` e poi assegnato alla `i`-*esima* riga e `j`-*esima* colonna della matrice.

```
    Next
Next
```

`Next`, semplicemente, incrementa la variabile di ciclo corrispondente: il primo `Next` incrementa la variabile più interna (`j`), mentre il secondo `Next` la variabile più esterna (`i`).
Nel caso in cui ci siano più cicli nidificati e vogliamo chiarire meglio quale ciclo viene di volta in volta chiuso, è possibile indicare anche la variabile di ciclo in questo modo:

```
    Next j
Next i
```

Al termine di questo primo blocco sono stati estratti tutti i 55 numeri e associati ciascuno a un diverso elemento della matrice.

```
For i = 0 To 10
```

Con questa istruzione inizia nuovamente un ciclo che attraversa l'intera matrice per estrarre i numeri da visualizzare.

```
        out &= ruote(i)
```

Prima di iniziare l'esplorazione dei numeri estratti di ciascuna ruota, accodiamo (`&=`) il nome della ruota alla variabile `out` che utilizzeremo per visualizzare i risultati.

```
    For j = 0 To 4
```

Qui, ancora una volta, iniziamo un ciclo interno per esplorare tutte le colonne dell'attuale riga della matrice.

```
        out &= (arrEstrazioneLotto(i, _
            j).ToString("00")) & " "
```

Leggiamo il numero contenuto nell'elemento `(i, j)` della matrice, poi lo convertiamo in stringa con il metodo `ToString`. Notate che `ToString` specifica anche il formato con cui vogliamo ottenere il numero: tale formato in questo caso è necessario per poter allineare correttamente i numeri in fase di visualizzazione del risultato.

```
    Next
```

È terminato il ciclo più interno.

```
    out &= Environment.NewLine
```

Anche in questo esempio avanziamo di una riga con il metodo `NewLine` dell'oggetto `Environment`.

```
Next
```

È terminato anche il ciclo più esterno.

```
Console.WriteLine(out)
Console.ReadLine()
```

Infine, utilizziamo l'oggetto `Console` per visualizzare l'intero risultato contenuto nella variabile `out`. `WriteLine` scrive quanto indicato all'interno delle parentesi tonde e avanza di una riga.

Ordinamento di una matrice unidimensionale

Per ordinare una matrice unidimensionale è possibile utilizzare il metodo `Sort` della classe `Array`:

```
Array.Sort(nomeArray)
```

Riprendiamo, per esempio, l'array dei nomi dei mesi e proviamo a ordinarlo alfabeticamente:

```
' Esempio: 08.07
Dim ruote() As String = {"NAZIONALE", "BARI", _
               "CAGLIARI", "FIRENZE", "GENOVA", _
               "MILANO", "NAPOLI", "PALERMO", _
               "ROMA", "TORINO", "VENEZIA"}
Array.Sort(ruote)
For i As Integer = 0 To 10
    Console.WriteLine(ruote(i))
Next
Console.ReadLine()
```

Il risultato è mostrato nella Figura 8.1.

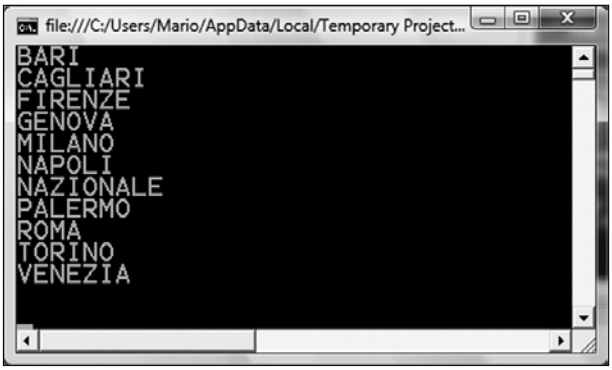

Figura 8.1 - Matrice ordinata di stringhe.

L'ordinamento alfabetico è insensibile alle lettere maiuscole/minuscole e, quindi, l'ordinamento sarà eseguito sempre correttamente.
Naturalmente, l'ordinamento deve adattarsi al tipo di dato con cui è stata dichiarata la matrice e quindi, nel caso di una matrice di numeri interi, le regole di ordinamento cambiano. Possiamo verificare che l'ordinamento sia eseguito correttamente anche in questo caso:

```
' Esempio: 08.08
Dim ruote() As Integer = _
    {15, 81, 77, 44, 12, 45, 24, _
```

```
     34, 76, 44, 1, 5, 7, 2, 5}
Array.Sort(ruote)
For i As Integer = 0 To 14
    Console.WriteLine(ruote(i))
Next
Console.ReadLine()
```

Mostriamo l'output di questo esempio nella Figura 8.2.

Figura 8.2 - Matrice ordinata di numeri interi.

L'interfaccia IComparable

Finché ordiniamo elementi che hanno un tipo di dato elementare (`Char`, `String`, `Integer`, `Double`) anche l'ordinamento è elementare.
Talvolta, però, è necessario ordinare anche degli oggetti e questa non è un'operazione elementare. Gli oggetti, infatti, non possono essere ordinati con le regole classiche che sono applicate a stringhe e a numeri, perché in realtà gli oggetti espongono tipicamente molteplici attributi: si pensi per esempio, a un oggetto che rappresenta un'auto. Questa ha numerosi attributi: marca, modello, cilindrata, numero di porte, peso, portata, capacità del serbatoio, consumi, colore, dimensioni, accessori, allestimento interno, dati anagrafici del proprietario e altro. Quale di questi attributi deve essere utilizzato per un ordinamento corretto? Né Visual Basic né il Framework .NET sottostante possono saperlo.
Per risolvere tale incertezza dobbiamo tenere presente che il metodo `Sort` di una matrice unidimensionale utilizza l'implementazione dell'interfaccia `IComparable` di ciascun elemento dell'oggetto `Array` per ordinare gli elementi.
Questo fatto è molto importante, perché rende possibile definire un criterio di ordinamento specifico per ciascuna nostra classe.

Per fare un esempio concreto di questa tecnica per ordinare un array di oggetti personalizzati, si crei un nuovo progetto di tipo `Console` e si inserisca il seguente codice in un file di classe di nome `Auto`.

```
' Esempio: 08.09
Public Class Auto
      Implements IComparable(Of Auto)

   Private m_cilindrata As Integer
   Private m_targa As String

   Public Sub New(ByVal pCilindrata _
         As Integer, ByVal pTarga As String)
      m_cilindrata = pCilindrata
      m_targa = pTarga
   End Sub

   Public Overloads Function CompareTo(ByVal _
        obj As Auto) As Integer Implements _
        IComparable(Of Auto).CompareTo

        If TypeOf obj Is Auto Then
            Dim temp As Auto = CType(obj, Auto)
            Return _
          m_cilindrata.CompareTo(temp.cilindrata)
        End If

        Throw New _
           ArgumentException( _
               "Oggetto di tipo " & _
               "diverso da Auto")
    End Function

    ReadOnly Property cilindrata() As Integer
        Get
            cilindrata = m_cilindrata
            Exit Property
        End Get
    End Property

    ReadOnly Property targa() As String
        Get
            targa = m_targa
            Exit Property
        End Get
    End Property
End Class
```

All'interno del file di modulo, invece, si inserisca il seguente codice:

```
' Esempio: 08.09 (segue)
Public Module Modulo
    Sub Main()
        Dim i As Integer     ' variabile di ciclo
        Dim risp As String = ""   ' per messaggio
```

```
        Dim arr(3) As Auto
        arr(0) = New Auto(1600, "AZ417DD")
        arr(1) = New Auto(1000, "DE934AS")
        arr(2) = New Auto(1800, "AB123FF")
        arr(3) = New Auto(600, "MI43488A")

        Array.Sort(arr)       ' ordinamento array

        For i = 0 To 3
            ' preparazione messaggio ordinamento
            ' per cilindrata
            risp &= arr(i).cilindrata & _
               Environment.NewLine
        Next
        Console.WriteLine(risp)
        Console.ReadLine()
    End Sub
End Module
```

Infine, si avvii il programma premendo il tasto **F5**. A questo punto apparirà il seguente messaggio, indicando che l'ordinamento è stato eseguito:

```
600
1000
1600
1800
```

Analizziamo in modo più approfondito le caratteristiche di questa implementazione, per comprendere del tutto il funzionamento della classe.
Innanzitutto, notiamo che la definizione della classe è seguita subito dopo dalla definizione dell'interfaccia che la classe `Auto` implementa:

```
Implements IComparable(Of Auto)
```

Anticipiamo, in questa sede, un argomento che riguarda la programmazione orientata agli oggetti. Non è fondamentale che il lettore, inesperto di programmazione a oggetti, comprenda a fondo il significato di questa parte perché sarà trattata in modo più organico nel capitolo specifico. È, comunque, un'opportunità per iniziare a familiarizzare con alcuni esempi di programmazione orientata agli oggetti.

Esistono diverse varianti dell'interfaccia `IComparable`, ma quella che abbiamo mostrato è la nuova variante introdotta con il Framework .NET 2.0. Infatti, nell'indice di MSDN Library, fornito con Visual Studio 2010, tale interfaccia è inserita alla voce

`IComparable(Of T).CompareTo (metodo)`. L'utilizzo del metodo `CompareTo` ci permette di confrontare l'oggetto corrente con un oggetto diverso ma dello stesso tipo.
Il termine `T` del parametro `(Of T)` è stato sostituito con il nome della classe personalizzata da utilizzare `(Of Auto)`.
Dichiariamo le due variabili di istanza private, `m_cilindrata` e `m_targa`, rispettivamente di tipo `Integer` e di tipo `String`.

Il prefisso `m_` prima del nome delle variabili è una convenzione adottata da molti programmatori per indicare che la variabile stessa è una variabile privata e, quindi, non visibile all'esterno della classe. Le proprietà, invece, normalmente non hanno un prefisso, perché in questo caso deve prevalere la leggibilità del nome delle proprietà pubbliche rispetto alle necessità di autodocumentazione del codice.

Subito dopo definiamo il costruttore dell'oggetto che prende come parametri i valori da assegnare alle variabili private. Il costruttore viene eseguito sempre al momento della creazione di un oggetto e serve, quindi, a inizializzare l'oggetto stesso, ponendolo in uno stato ben definito e indipendente da quello di altri oggetti della stessa classe.
Implementiamo, quindi, l'interfaccia `IComparable`, definendo la `Function CompareTo` come segue:

```
Public Overloads Function _
    CompareTo(ByVal obj As Auto) _
    As Integer _
    Implements IComparable(Of Auto).CompareTo
```

Anche qui abbiamo usato il nome della classe da utilizzare (`Auto`).
Il test `If TypeOf obj Is Auto` verifica se l'oggetto è effettivamente di tipo `Auto`: nel caso in cui non lo fosse, otterremmo un'eccezione. Decidiamo invece di sollevare un'eccezione di tipo `ArgumentException` e con una descrizione personalizzata:

```
Throw New ArgumentException("Oggetto di tipo " _
                "diverso da Auto")
```

Se il test va a buon fine, invece, viene eseguito il codice all'interno del blocco `If .. End If` che dichiara una variabile interna con la conversione dell'oggetto dal tipo generico `Object` al tipo specifico `Auto`. Al metodo `CompareTo` viene passato il contenuto della variabile privata che contiene il valore su cui viene effettivamente eseguito l'ordinamento.

Il resto del codice della classe definisce due proprietà a sola lettura (`ReadOnly`) utilizzate per leggere il contenuto delle variabili private `m_cilindrata` e `m_targa`.
Principio fondamentale: le variabili interne devono essere sempre nascoste e non esposte all'esterno, per impedire che possano essere manipolate direttamente in modo incontrollato. Questa regola fa parte del principio di **incapsulamento dei dati**. Torneremo su questo aspetto nel capitolo specifico sulla programmazione orientata agli oggetti.
Nel modulo viene dichiarata una matrice di quattro elementi, popolata con altrettanti riferimenti a oggetti di tipo `Auto`, ciascuno dei quali riceve i due parametri previsti: cilindrata e targa.
Il metodo `Sort` invocato sulla classe `Array` ordina effettivamente l'array passato come parametro.
Nell'ultima operazione viene costruito il messaggio da visualizzare per verificare che l'ordinamento sia stato eseguito correttamente.

Ordinamento con diverso criterio

Proviamo ora ad apportare una piccola modifica: vogliamo che l'ordinamento avvenga non più per cilindrata ma per targa.
Per ottenere questo risultato dobbiamo modificare soltanto due istruzioni:

1. nella classe `Auto` dobbiamo sostituire l'istruzione

   ```
   Return _
       m_cilindrata.CompareTo(temp.cilindrata)
   ```

   con

   ```
   Return _
       m_targa.CompareTo(temp.targa)
   ```

2. nel modulo dobbiamo invece sostituire l'istruzione

   ```
   risp &= arr(i).cilindrata Environment.NewLine
   ```

   con

   ```
   risp &= arr(i).targa & Environment.NewLine
   ```

Il risultato finale sarà il seguente:

```
AB123FF
AZ417DD
DE934AS
MI43488A
```

Ordinamento parametrizzato

Come ulteriore esempio, proviamo a rendere l'ordinamento ancora più flessibile. Vogliamo, infatti, definire diversi tipi di ordinamento che siano utilizzabili in qualsiasi momento: per cilindrata, per targa o per marca.
Il codice modificato per l'oggetto `Auto` è il seguente:

```
' Esempio: 08.10
Public Class Auto
  Implements IComparable(Of Auto)

  Private m_tipoOrdinamento As Integer
  Private m_cilindrata As Integer
  Private m_targa As String
  Private m_marca As String

  Public Sub New(ByVal pCilindrata As Integer,
                 ByVal pTarga As String,
                 ByVal pMarca As String)
    m_cilindrata = pCilindrata
    m_targa = pTarga
    m_marca = pMarca
  End Sub

  Public Overloads Function _
     CompareTo(ByVal obj As Auto) _
     As Integer _
     Implements IComparable(Of Auto).CompareTo

    If TypeOf obj Is Auto Then
      Dim temp As Auto = CType(obj, Auto)
      Select Case m_tipoOrdinamento
        Case 1
          Return _
            m_cilindrata.CompareTo(temp.cilindrata)
        Case 2
          Return _
            m_targa.CompareTo(temp.targa)
        Case 3
          Return _
            m_marca.CompareTo(temp.marca)
      End Select
    End If

    Throw New ArgumentException("Oggetto " & _
                  "di tipo diverso da Auto")
  End Function

  ReadOnly Property cilindrata() As Integer
    Get
      cilindrata = m_cilindrata
      Exit Property
    End Get
  End Property
```

```
  ReadOnly Property targa() As String
    Get
      targa = m_targa
      Exit Property
    End Get
  End Property

  ReadOnly Property marca() As String
    Get
      marca = m_marca
      Exit Property
    End Get
  End Property

  WriteOnly Property tipoOrdinamento() As String
    Set(ByVal value As String)
      ' 1 = cilindrata
      ' 2 = targa
      ' 3 = marca
      m_tipoOrdinamento = value
      Exit Property
    End Set
  End Property
End Class
```

Come si può notare, le modifiche apportate al codice, oltre all'aggiunta di una nuova proprietà ("marca"), non sono molte:

1. è stata aggiunta una variabile privata `m_tipoOrdinamento` per memorizzare il tipo di ordinamento desiderato;
2. nella `Function CompareTo` è stata inserita un'istruzione `Select Case` per eseguire un confronto differenziato (per cilindrata, per targa o per marca) e per restituire il relativo risultato;
3. è stata aggiunta una proprietà a sola scrittura, `tipoOrdinamento()`, per impostare il tipo di ordinamento che dovrà essere eseguito su ogni singola istanza della classe `Auto`.

Il codice modificato del modulo, invece, è il seguente:

```
' Esempio: 08.10 (segue)
Public Module Modulo
  Sub Main()
    Dim i As Integer    ' variabile di ciclo
    Dim risp As String = ""    ' messaggio

    Dim arr(3) As Auto
    arr(0) = New Auto(1600, "AZ417DD", "FIAT")
    arr(1) = New Auto(1000, "DE934AS", "OPEL")
    arr(2) = New Auto(1800, "AB123FF", "BMW")
    arr(3) = New Auto(1200, "MI43488A", "AUDI")
```

```
    arr(0).tipoOrdinamento = 1
    arr(1).tipoOrdinamento = 1
    arr(2).tipoOrdinamento = 1
    arr(3).tipoOrdinamento = 1
    Array.Sort(arr)       ' ordinamento array

    For i = 0 To 3
      ' preparazione messaggio
      ' ordinamento per cilindrata
      risp &= arr(i).cilindrata & " / " &
         arr(i).targa & " / " & arr(i).marca &
         Environment.NewLine
    Next
    Console.WriteLine(risp)

    arr(0).tipoOrdinamento = 2
    arr(1).tipoOrdinamento = 2
    arr(2).tipoOrdinamento = 2
    arr(3).tipoOrdinamento = 2
    Array.Sort(arr)       ' ordinamento array

    risp = ""
    For i = 0 To 3
      ' preparazione messaggio
      ' ordinamento per targa
      risp &= arr(i).cilindrata & " / " &
         arr(i).targa & " / " & arr(i).marca &
         Environment.NewLine
    Next
    Console.WriteLine(risp)

    arr(0).tipoOrdinamento = 3
    arr(1).tipoOrdinamento = 3
    arr(2).tipoOrdinamento = 3
    arr(3).tipoOrdinamento = 3
    Array.Sort(arr)       ' ordinamento array

    risp = ""
    For i = 0 To 3
      ' preparazione messaggio
      ' ordinamento per marca
      risp &= arr(i).cilindrata & " / " &
         arr(i).targa & " / " & arr(i).marca &
         Environment.NewLine
    Next
    Console.WriteLine(risp)
    Console.ReadLine()
  End Sub
End Module
```

Anche in questa parte di codice non abbiamo apportato troppe modifiche:

1. aggiunte le istruzioni `arr(x).tipoOrdinamento = n` prima dell'esecuzione dell'ordinamento;
2. nell'ultima parte abbiamo aggiunto tre visualizzazioni di altrettanti diversi ordinamenti: per cilindrata, per targa e per marca.

Il risultato di questa sequenza di istruzioni è il seguente:

```
1000 / DE934AS / OPEL
1200 / MI43488A / AUDI
1600 / AZ417DD / FIAT
1800 / AB123FF / BMW

1800 / AB123FF / BMW
1600 / AZ417DD / FIAT
1000 / DE934AS / OPEL
1200 / MI43488A / AUDI

1200 / MI43488A / AUDI
1800 / AB123FF / BMW
1600 / AZ417DD / FIAT
1000 / DE934AS / OPEL
```

Si vede chiaramente dal risultato che il primo gruppo è ordinato per cilindrata, il secondo per targa e il terzo per marca, esattamente come ci aspettavamo.

Matrici rettangolari (o regolari)

Una matrice è rettangolare se ciascuna dimensione ha lo stesso numero di elementi: per esempio, la Figura 8.3 rappresenta una matrice bidimensionale rettangolare, tre righe per tre colonne, mentre nella Figura 8.4 possiamo vedere una matrice tridimensionale regolare, tre righe per tre colonne in tre piani di profondità.

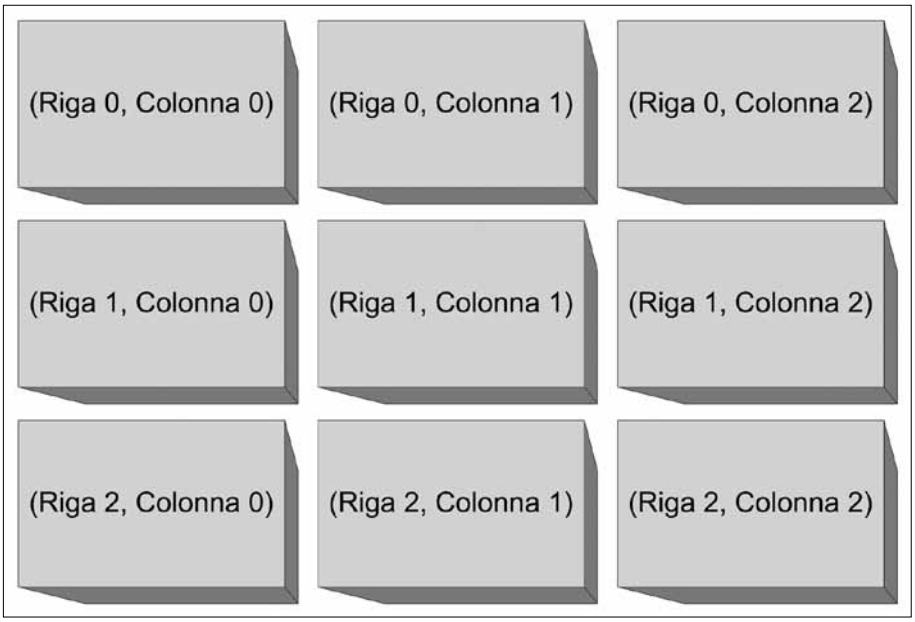

Figura 8.3 - Matrice bidimensionale regolare.

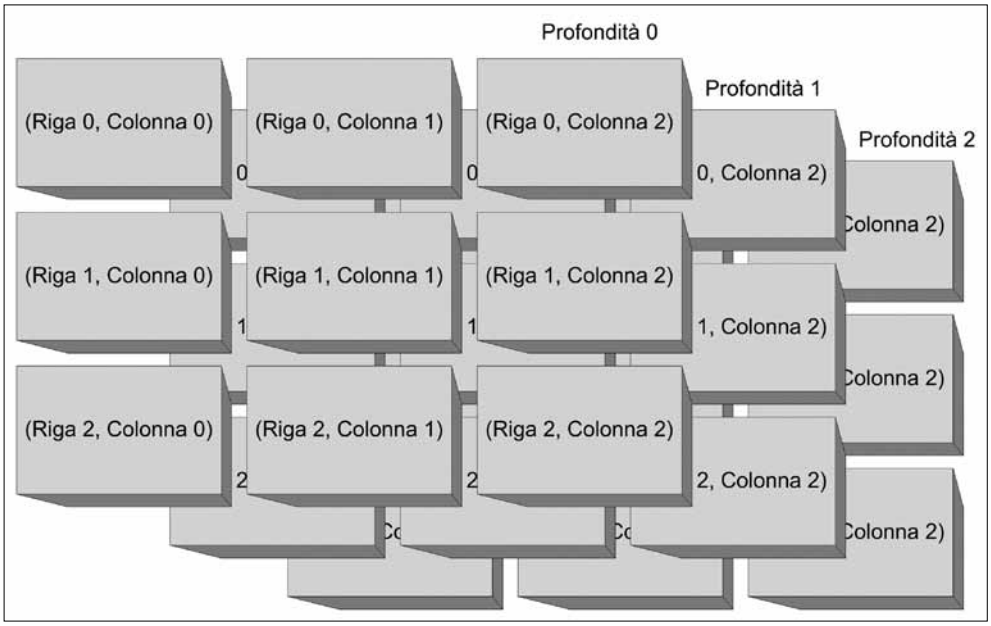

Figura 8.4 - Matrice tridimensionale regolare.

Una precisazione: le dimensioni non devono essere necessariamente uguali tra loro, come negli esempi presentati. Per esempio, possiamo avere senza problemi una matrice bidimensionale tre per quattro, una tridimensionale otto per tre per due. L'importante è che ciascuna dimensione abbia lo stesso numero di elementi: per esempio, in una matrice tre per quattro, le tre righe devono avere tutte quattro colonne.
Da questi esempi emerge anche una caratteristica importante: l'indice di riga, di colonna o di profondità inizia sempre da zero.
Un esempio di dichiarazione di un array bidimensionale è il seguente:

```
Dim array2dim = {{1, 2}, {3, 4}, {5, 6}}
```

Matrici multidimensionali

Abbiamo già mostrato alcuni esempi di matrici bidimensionali, ma ora andiamo oltre: infatti, il termine multidimensionale indica che tali matrici hanno più dimensioni, sottintendendo che le dimensioni sono almeno tre.
Per capire meglio cosa intendiamo per multidimensionalità, dobbiamo pensare alle classiche dimensioni del mondo reale, svincolando il concetto dall'informatica.
Come abbiamo affermato in precedenza, una matrice bidimensionale può essere rappresentata da una tabella, per esempio, di una rubrica telefonica: ogni riga con-

tiene i dati di un diverso contatto, mentre le colonne contengono i singoli dati, come nome, telefono, cellulare ecc.
La matrice tridimensionale è soltanto un po' più complicata, perché abbiamo anche nel mondo reale degli oggetti concreti tridimensionali: prima di tutto è tridimensionale lo spazio in cui ci muoviamo (si pensi alle coordinate spaziali `x`, `y` e `z`). Un altro esempio è un archivio di documenti: ogni persona potrebbe avere una cartella, all'interno della quale c'è un unico foglio costituito da una tabella bidimensionale. Questa è una struttura tridimensionale.
Già dalla quarta dimensione inizia a essere difficile avere una rappresentazione reale. Di solito, nel mondo reale, si aggiunge la dimensione tempo: un oggetto in movimento ha una rappresentazione istantanea di coordinate `x`, `y` e `z`, per ogni istante di tempo `t`. Per riprendere l'esempio dell'archivio, possiamo aggiungere più fogli in ogni cartella, ciascuno dei quali si riferisce a un determinato periodo di tempo (per esempio, `anno`, `trimestre` o `mese`). Se vogliamo accedere a un certo dato dobbiamo prima identificare la cartella che ci interessa, poi il foglio (fattore tempo), tra i tanti contenuti nella cartella, e infine l'incrocio tra righe e colonne della tabella contenuta nel foglio.
Tornando agli aspetti più informatici, possiamo avere due tipi di matrici multidimensionali: le matrici rettangolari (dette anche regolari) e le matrici irregolari.

Esaminare e manipolare matrici

Possiamo esaminare le caratteristiche di una qualsiasi matrice attraverso l'uso di funzioni sempre disponibili, perché definite all'interno delle matrici stesse. Non dobbiamo, quindi, preoccuparci di crearle appositamente.
Per capire come si utilizzano alcune di queste funzioni vogliamo mostrarvi un esempio che dovrebbe chiarire molti dubbi.

```
' Esempio: 08.11
Module Module1
    Private NumeroCercato As Integer = 0
    Sub Main()
        Dim arr(,) As Integer = _
            {{0, 1, 1, 2, 3, 5, 8, 13, 21, 34}, _
            {1, 2, 3, 5, 7, 11, 13, 17, 19, 23}}
        Dim arr2() As Integer = _
            {1, 2, 3, 5, 7, 11, 13, 17, 19, 23}
        Dim arr3() As Integer = _
            {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, _
             14, 16, 99}

        ' Vari metodi
        Console.WriteLine("Numero elementi: " & _
            arr.Length)
        Console.WriteLine("Rango: " & arr.Rank)
        Console.WriteLine("Lunghezza " & _
```

```
    "dimensione 0: " & _
    arr.GetLength(0))
Console.WriteLine("Lunghezza " & _
    "dimensione 1: " & _
    arr.GetLength(1))
Console.WriteLine("Limiti " & _
    "dimensione 0: ({0}, {1})", _
    arr.GetLowerBound(0), arr.GetUpperBound(0))
Console.WriteLine("Limiti " & _
    "dimensione 1: ({0}, {1})", _
    arr.GetLowerBound(1), _
    arr.GetUpperBound(1))

' Metodo "Exists"
NumeroCercato = 444
Console.WriteLine("Exists " & _
    NumeroCercato & ":" & _
    Array.Exists(arr2, _
    AddressOf NumeroUguale))
NumeroCercato = 5
Console.WriteLine("Exists " & _
    NumeroCercato & ": " & _
    Array.Exists(arr2, _
    AddressOf NumeroUguale))
' Metodo "Find"
NumeroCercato = 7
Console.WriteLine("Find numero " & _
    "successivo a " & _
    NumeroCercato & ": " & _
    Array.Find(arr2, _
    AddressOf TrovaSuccessivo))
NumeroCercato = 13
Console.WriteLine("Find numero " & _
    "successivo a " & _
    NumeroCercato & ": " & _
    Array.Find(arr2, _
    AddressOf TrovaSuccessivo))

' Metodo "FindLast"
NumeroCercato = 7
Console.WriteLine("Find numero " & _
    "precedente a " & NumeroCercato & _
    ": " & Array.FindLast(arr2, _
    AddressOf TrovaPrecedente))
NumeroCercato = 13
Console.WriteLine("Find numero " & _
    "precedente a " & _
    NumeroCercato & ": " & _
    Array.FindLast(arr2, _
    AddressOf TrovaPrecedente))

' Metodo "FindAll" per numeri dispari
' e numeri pari
Console.WriteLine("Trova tutti i " & _
    "numeri dispari: ")
Dim subArray() As Integer = _
```

```
            Array.FindAll(arr3, _
            AddressOf NumeriDispari)
        For Each numero As Integer In subArray
            Console.WriteLine(numero)
        Next

        Console.WriteLine("Trova tutti i " & _
            "numeri pari: ")
        subArray = Array.FindAll(arr3, _
            AddressOf NumeriPari)
        For Each numero As Integer In subArray
            Console.WriteLine(numero)
        Next

        ' Metodo "Copy"
        Console.WriteLine("Copia 3 numeri " & _
            "dall'elemento " & _
            "4 del primo array a partire " & _
            "dall'elemento 0 " & _
            "del secondo array: ")
        Dim arr4(4) As Integer
        Array.Copy(arr3, 4, arr4, 0, 3)
        For Each numero As Integer In arr4
            Console.WriteLine(numero)
        Next

        Console.ReadLine()
    End Sub
    Private Function NumeroUguale(ByVal n _
            As Integer) As Boolean
        If n = NumeroCercato Then
            Return True
        Else
            Return False
        End If
    End Function
    Private Function TrovaSuccessivo(ByVal n _
            As Integer) As Boolean
        If n > NumeroCercato Then
            Return True
        Else
            Return False
        End If
    End Function
    Private Function TrovaPrecedente(ByVal n _
            As Integer) As Boolean
        If n < NumeroCercato Then
            Return True
        Else
            Return False
        End If
    End Function
    Private Function NumeriDispari(ByVal n _
            As Integer) As Boolean
        If (n Mod 2) <> 0 Then
            Return True
        Else
```

```
            Return False
        End If
    End Function
    Private Function NumeriPari(ByVal n _
            As Integer) As Boolean
        If (n Mod 2) = 0 Then
            Return True
        Else
            Return False
        End If
    End Function
End Module
```

I risultati visualizzati nella console sono i seguenti:

```
Numero elementi: 20
Rango: 2
Lunghezza dimensione 0: 2
Lunghezza dimensione 1: 10
Limiti dimensione 0: (0, 1)
Limiti dimensione 1: (0, 9)
Exists 444:False
Exists 5: True
Find numero successivo a 7: 11
Find numero successivo a 13: 17
Find numero precedente a 7: 5
Find numero precedente a 13: 11
Trova tutti i numeri dispari:
1
3
5
7
9
99
Trova tutti i numeri pari:
2
4
6
8
10
12
14
16
Copia 3 numeri dall'elemento 4 del primo array a partire dall'elemento 0 del
secondo array:
5
6
7
0
0
```

Vedremo ora in dettaglio come funzionano tutte le parti di questo esempio.

```
Private NumeroCercato As Integer = 0
```

Questa istruzione, come abbiamo visto in altre occasioni, dichiara una variabile privata di nome `NumeroCercato`, del tipo `Integer` e le assegna un valore nullo.
La variabile ci servirà in varie occasioni per passare un valore di confronto ai vari metodi che creeremo nell'esempio.

```
Sub Main()
    Dim arr(,) As Integer = _
        {{0, 1, 1, 2, 3, 5, 8, 13, 21, 34}, _
        {1, 2, 3, 5, 7, 11, 13, 17, 19, 23}}
    Dim arr2() As Integer = _
        {1, 2, 3, 5, 7, 11, 13, 17, 19, 23}
    Dim arr3() As Integer = _
        {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, _
         14, 16, 99}
```

Le prime istruzioni del metodo `Main`, da cui viene avviato il programma, dichiarano tre array di tipo `Integer` con contenuti diversi: l'array `arr` è un array bidimensionale formato da due sequenze di numeri, cioè i primi dieci dei cosiddetti "numeri di Fibonacci" e i primi dieci numeri primi; l'array `arr2` è un array unidimensionale formato ancora dai primi dieci numeri primi, mentre `arr3` è un array unidimensionale contenente alcuni numeri in una sequenza non determinata.

```
    Console.WriteLine("Numero elementi: " & _
       arr.Length)
```

`Length` restituisce il numero totale di elementi contenuti in tutte le dimensioni dell'array `arr`.

```
    Console.WriteLine("Rango: " & arr.Rank)
```

`Rank` restituisce la dimensione della più grande matrice quadrata contenuta nell'array `arr`.

```
    Console.WriteLine("Lunghezza " & _
        dimensione 0: " & arr.GetLength(0))
    Console.WriteLine("Lunghezza " & _
        dimensione 1: " & arr.GetLength(1))
```

`GetLength` restituisce il numero di elementi contenuti nella dimensione specificata.

```
    Console.WriteLine("Limiti " & _
        dimensione 0: ({0}, {1})", _
        arr.GetLowerBound(0), _
        arr.GetUpperBound(0))
    Console.WriteLine("Limiti " & _
        dimensione 1: ({0}, {1})", _
        arr.GetLowerBound(1), _
        arr.GetUpperBound(1))
```

`GetLowerBound` restituisce il limite inferiore della matrice (tipicamente zero), mentre con `GetUpperBound` otteniamo il limite superiore.

```
    ' Metodo "Exists"
    NumeroCercato = 444
    Console.WriteLine("Exists " & _
        NumeroCercato & ":" & _
        Array.Exists(arr2, _
        AddressOf NumeroUguale))
    NumeroCercato = 5
    Console.WriteLine("Exists " & _
        NumeroCercato & ": " & _
        Array.Exists(arr2, _
        AddressOf NumeroUguale))
```

Questo frammento di codice mostra come sia possibile verificare l'esistenza di un elemento determinato nella matrice. Nel primo caso il numero non viene trovato mentre nel secondo caso la ricerca ha esito positivo. `Array.Exists` è un metodo generico che utilizza un **delegato** a un metodo che restituisce un valore booleano. Se l'elemento corrisponde con il valore cercato, il delegato restituisce `True`; in caso contrario restituisce `False`. Di seguito riportiamo il codice del delegato:

```
Private Function NumeroUguale(ByVal n _
        As Integer) As Boolean
    If n = NumeroCercato Then
        Return True
    Else
        Return False
    End If
End Function
```

Mostriamo ora il codice per cercare il numero memorizzato nell'elemento successivo della matrice, rispetto alla posizione del numero indicato:

```
    ' Metodo "Find"
    NumeroCercato = 7
    Console.WriteLine("Find numero " & _
        "successivo a " & _
        NumeroCercato & ": " & Array.Find(arr2, _
        AddressOf TrovaSuccessivo))
    NumeroCercato = 13
    Console.WriteLine("Find numero " & _
        "successivo a " & _
        NumeroCercato & ": " & Array.Find(arr2, _
        AddressOf TrovaSuccessivo))
```

Anche in questo caso viene utilizzato un delegato per l'esecuzione del metodo `Array.Find`. Di seguito, presentiamo il codice del delegato `TrovaSuccessivo`:

```
Private Function TrovaSuccessivo(ByVal n _
```

```
        As Integer) As Boolean
    If n > NumeroCercato Then
        Return True
    Else
        Return False
    End If
End Function
```

Questo codice restituisce `True` quando viene confrontato il primo valore maggiore del numero indicato.
Il metodo `Array.FindLast` esegue la stessa operazione, ma partendo dall'elemento finale dell'array e risalendo verso l'inizio. Il risultato è che viene restituito `True` quando viene confrontato il primo valore inferiore al valore indicato:

```
    ' Metodo "FindLast"
    NumeroCercato = 7
    Console.WriteLine("Find numero " & _
        "precedente a " & NumeroCercato & _
        ": " Array.FindLast(arr2, _
        AddressOf TrovaPrecedente))
    NumeroCercato = 13
    Console.WriteLine("Find numero " & _
        "precedente a " & NumeroCercato & _
        ": " Array.FindLast(arr2, _
        AddressOf TrovaPrecedente))
```

Il codice del delegato `TrovaPrecedente` è quasi identico a quello del delegato `TrovaSuccessivo`, infatti, cambia solamente l'operatore di confronto:

```
Private Function TrovaPrecedente(ByVal n _
        As Integer) As Boolean
    If n < NumeroCercato Then
        Return True
    Else
        Return False
    End If
End Function
```

Ecco un'implementazione interessante per la ricerca di tutti i numeri dispari contenuti nella matrice:

```
    ' Metodo "FindAll" per numeri dispari _
    ' e numeri pari
    Console.WriteLine("Trova tutti i " & _
            "numeri dispari: ")
    Dim subArray() As Integer = _
        Array.FindAll(arr3, _
        AddressOf NumeriDispari)
    For Each numero As Integer In subArray
        Console.WriteLine(numero)
    Next
```

A questo punto diventa quasi inutile sottolineare che il metodo `Array.FindAll` estrae tutti i valori per i quali il delegato `NumeriDispari` restituisce `True`. I valori estratti sono inseriti in una matrice di appoggio che viene poi utilizzata per visualizzarli nella console. Ecco il codice del semplice delegato `NumeriDispari`:

```
Private Function NumeriDispari(ByVal n _
        As Integer) As Boolean
    If (n Mod 2) <> 0 Then
        Return True
    Else
        Return False
    End If
End Function
```

Anche in questo caso cambia solo la condizione di confronto: l'operatore `Mod` estrae il resto della divisione tra l'elemento corrente della matrice e il numero due; il resto deve essere diverso da zero, altrimenti si tratterebbe di un numero pari.
L'operazione di estrazione di tutti i numeri pari è esattamente simmetrica:

```
    Console.WriteLine("Trova tutti i " & _
        "numeri pari: ")
    subArray = Array.FindAll(arr3, _
        AddressOf NumeriPari)
    For Each numero As Integer In subArray
        Console.WriteLine(numero)
    Next
```

così come il delegato `NumeriPari`:

```
Private Function NumeriPari(ByVal n As Integer) As Boolean
    If (n Mod 2) = 0 Then
        Return True
    Else
        Return False
    End If
End Function
```

L'ultimo metodo che abbiamo presentato in questo codice di esempio è il metodo `Array.Copy`, che permette di copiare un sottoinsieme della matrice in un'altra matrice:

```
    ' Metodo "Copy"
    Console.WriteLine("Copia 3 numeri " & _
        "dall'elemento 4 del primo array " & _
        "a partire dall'elemento 0 " & _
        "del secondo array: ")
    Dim arr4(4) As Integer
    Array.Copy(arr3, 4, arr4, 0, 3)
    For Each numero As Integer In arr4
        Console.WriteLine(numero)
    Next
```

L'esempio è sufficientemente esplicativo, quindi, non riteniamo necessario fare ulteriori commenti.
Talvolta, può essere necessario eliminare o azzerare un sottoinsieme di elementi di una matrice. Quest'operazione può essere eseguita utilizzando il metodo `Clear`:

```
Array.Clear(nomeArray, _
    elementoIniziale, numeroElementi)
```

In base al tipo di dato con cui abbiamo definito la matrice, gli elementi indicati (`numeroElementi`), a partire da `elementoIniziale`, sono impostati a zero, a `Null` o a `False`.

Matrici irregolari (jagged array)

Sono chiamate matrici irregolari quelle matrici che hanno un diverso numero di elementi per una certa dimensione (Figura 8.5).

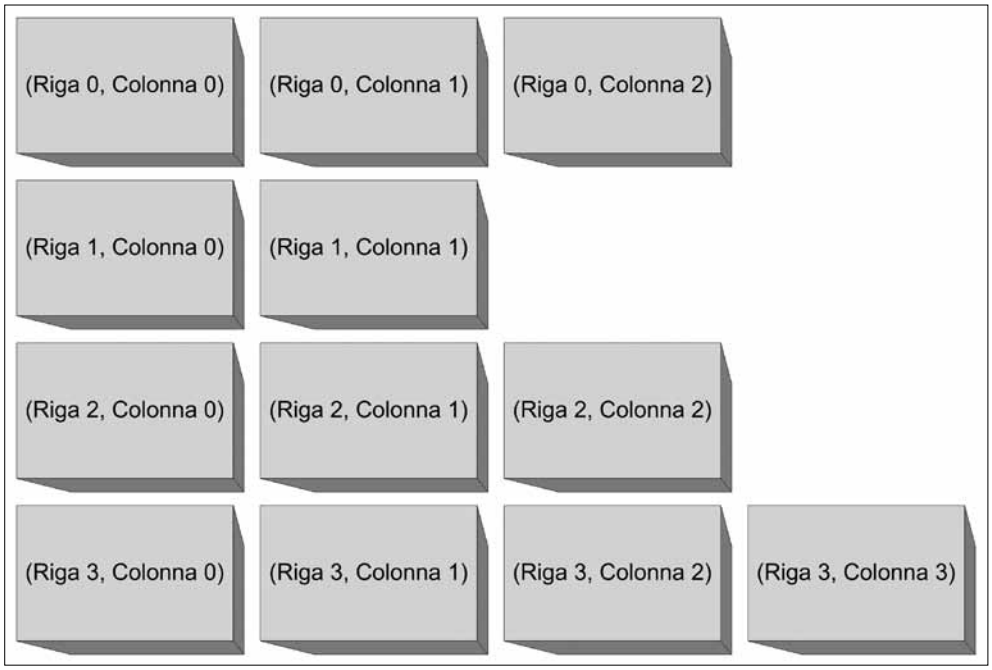

Figura 8.5 - Una matrice irregolare.

Possiamo definire una matrice irregolare in un modo molto semplice: prima definiamo una matrice unidimensionale. In ogni elemento di tale matrice, poi, inseriamo il riferimento ad altre matrici di lunghezza variabile.

Nell'esempio che presentiamo ora, definiamo la matrice irregolare della Figura 8.5:

```
' Esempio: 08.12
Module Module1
  Sub Main()

    Dim mat(3) As Array    ' definisce 4 righe
    ' (da 0 a 3)
    Dim mat0(2) As String  ' array di 4 elementi
    ' nella riga 0
    Dim mat1(1) As String  ' array di 2 elementi
    ' nella riga 1
    Dim mat2(2) As String  ' array di 3 elementi
    ' nella riga 2
    Dim mat3(3) As String  ' array di 2 elementi
    ' nella riga 3
    mat(0) = mat0          ' assegna array mat0 alla
    ' riga 0 di mat
    mat(1) = mat1          ' assegna array mat1 alla
    ' riga 1 di mat
    mat(2) = mat2          ' assegna array mat2 alla
    ' riga 2 di mat
    mat(3) = mat3          ' assegna array mat3 alla
    ' riga 3 di mat
    Dim i As Integer       ' variabile di ciclo
    ' esterno (righe)
    Dim j As Integer       ' variabile di ciclo
    ' interno (colonne)
    Dim risp As String = "" ' variabile per
    ' composizione messaggio
    For i = 0 To mat.GetUpperBound(0)
      ' ciclo esterno (righe)
      For j = 0 To mat(i).GetUpperBound(0)
        ' ciclo interno (colonne)
        ' imposta un valore:
        mat(i).SetValue(i & "-" & j, j)
      Next
    Next
    For i = 0 To mat.GetUpperBound(0)
      For j = 0 To mat(i).GetUpperBound(0)
        risp &= mat(i).GetValue(j) & _
            "  " ' legge un valore
      Next
      ' ritorno a capo:
      risp &= Environment.NewLine
    Next
    Console.WriteLine(risp)
    Console.ReadLine()

  End Sub
End Module
```

Il risultato è il seguente:

```
0-0  0-1  0-2
```

```
1-0  1-1
2-0  2-1  2-2
3-0  3-1  3-2  3-3
```

perfettamente corrispondente alla Figura 8.5.
Un modo più semplice di dichiarare tale matrice, in Visual Basic 2010, è il seguente:

```
Dim array2dim = {{"0-0", "0-1", "0-2"},
                 {"1-0", "1-1"},
                 {"2-0", "2-1", "2-2"},
                 {"3-0", "3-1", "3-2", "3-3"}}
```

Domande ed esercizi

Domanda 8.1 Le matrici servono solo in problemi matematici.
A - Vero;
B - Falso.

Domanda 8.2 Se abbiamo la seguente istruzione che definisce un array:

```
Dim giorni(100) As Integer
```

per aggiungere un ulteriore elemento alla fine dell'array posso scrivere questa istruzione:

```
giorni(101) = 12345
```

A - Falso;
B - Vero.

Domanda 8.3 Posso utilizzare Option Base 1 se non voglio che l'array abbia l'elemento 0 (zero).
A - Falso;
B - Vero.

Domanda 8.4 Per ordinare gli elementi di un array è necessario implementare una procedura di ordinamento con un algoritmo a scelta (Quick Sort, Bubble Sort...).
A - Falso;
B - Vero.

Risposte e soluzioni degli esercizi

Risposta 8.1

La risposta corretta è la B: le matrici possono contenere dati di vari tipi e anche riferimenti a oggetti, quindi, il loro ambito di utilizzo va oltre i soli problemi di carattere matematico.

Risposta 8.2

La risposta corretta è la A: non si possono aggiungere elementi a un array già definito. Eventualmente, può essere utilizzata l'istruzione `ReDim`, oppure si può creare un nuovo array della dimensione desiderata e copiare tutti gli elementi dall'array originario al nuovo array.

Risposta 8.3

La risposta corretta è la A: nel .NET Framework il primo indice degli array è sempre 0 (zero). L'istruzione `Option Base 1`, infatti, non è presente in VB.NET.

Risposta 8.4

La risposta corretta è la A: è sufficiente utilizzare il metodo statico `Array.Sort`. Se vogliamo, possiamo comunque implementare un diverso ordinamento, usando l'interfaccia `IComparable`.

Conclusioni

Questo capitolo ha dimostrato l'utilità delle matrici in una vasta gamma di applicazioni. Spesso, alcune tecniche prese in esame permettono di eseguire operazioni complesse con una sola istruzione, mettendo in risalto, ancora una volta, la potenza che il Framework .NET e i suoi linguaggi mettono a disposizione dello sviluppatore.