Sommario

1 Introduzione

Il primo ed il secondo capitolo introducono l'ambiente .NET ed il linguaggio C#.

Viene focalizzata l'attenzione sui costrutti di base come la selezione ed il ciclo, consentendo di scrivere le prime applicazioni console.

Nei successivi due capitoli verranno introdotti i concetti di tipo, variabile ed espressione. Seguendo la metodologia di quelli precedenti, non vengono illustrati tutti i tipi presenti nel linguaggio, ma solo quelli necessari alla comprensione del testo ed al prosieguo del corso.

Un argomento, che è naturale prosecuzione ed evoluzione di quello precedente, riguarda l'analisi del tipo strutturato Array: verrà introdotto il concetto di collezione e verrà fornita una breve descrizione dei metodi più comuni per accedere agli array oltre, ovviamente, al meccanismo che ne regola la creazione e la dichiarazione.

Il quinto capitolo riprende i concetti introdotti nei primi capitoli, approfondendo i costrutti. Si parte dal concetto di variabile locale, si prosegue analizzando gli operatori di incremento e decremento nei cicli iterativi e viene introdotto il costrutto di iterazione con controllo in coda, mentre per quanto riguarda la selezione si parla del costrutto di selezione multipla.

Subito dopo si comincia ad addentrarsi nella programmazione "avanzata" in C# introducendo i metodi ed i concetti ad essi correlati, partendo dalla definizione al prototipo fino a giungere al corpo.Tra gli altri concetti illustrati vi sono le definizioni dei campi di classe e, molto importante, del concetto di sovrapposizione dei contesti di validità. Si parla anche della chiamata (invocazione) di un metodo e di come passare un parametro ed ottenere valori di ritorno, sia essi siano tipi semplici o strutturati.

La parte "propedeutica" di questa parte sul linguaggi C#, si conclude approfondendo i tipi di dato array e string, chiarendo alcuni concetti

Per il tipo string si spiegano alcuni metodi di più frequente utilizzo e l'uso di variabili string come collezione di caratteri.

Viene dato anche un certo spazio al tipo di dato object, mattone fondamentale della programmazione in linguaggio C# e vengono introdotti concetti quali boxing ed unboxing.

1

Introduzione ai programmi C#

1 Salve, Mondo

Per tradizione, quando ci si accinge ad apprendere un nuovo linguaggio di programmazione il primo programma che viene realizzato è il canonico "Salve mondo", e cioè un programma che si limita a visualizzare un messaggio sullo schermo.

In C# un simile programma si presenta così:

```
using System;

class Program
{
    static void Main()
    {
        Console.WriteLine("Salve, mondo!");
    }
}
```

Il programma produce sullo schermo il seguente output:

```
Salve, mondo!
```

Ecco una rapida spiegazione dei singoli elementi che lo compongono.

```
using System;
```

Mediante la parola chiave `using` si dichiara in quali *namespaces* ricercare le classi a cui si accede nel programma; ciò consente di omettere il nome del *namespace* di appartenenza davanti alla classe. La maggior parte delle classi utilizzate nei programmi presentati appartengono al *namespace* `System` (che dunque, benché non sia obbligatorio, dovrebbe essere dichiarato).

```
class Programm { }
```

`Programm` è il nome della **classe principale**; l'aggettivo principale significa semplicemente che essa contiene il metodo `Main()`. Tutti gli esempi presentati nel testo contengono la sola classe principale. Il nome della classe può essere scelto liberamente dal programmatore; Program è il nome predefinito stabilito dall'ambiente di sviluppo.

```
static void Main() { }
```

`Main()` il metodo che viene automaticamente eseguito al momento del lancio del programma. Dunque, le prime istruzioni che vengono eseguite sono contenute all'interno del metodo `Main()`. Per il momento, occorre sorvolare sul significato delle parole chiave `static` e `void`; si ricordi soltanto che il metodo `Main()` dev'essere preceduto da queste parole.

Poiché il linguaggio C# fa distinzione tra lettere maiuscole e minuscole, occorre fare attenzione al rispetto del "case" delle medesime. Ad esempio, `Main()` dev'essere scritto con la "M" maiuscola e il resto minuscolo, altrimenti non sarà riconosciuto.

Le parole chiave del linguaggio, negli esempi evidenziate in neretto, devono essere scritte completamente in minuscolo.

```
Console.WriteLine();
```

`WriteLine()` è un metodo appartenente alla classe `Console` (La quale appartiene al *namespace* `System`). `WriteLine()` svolge il compito di scrivere sullo schermo l'argomento (o gli argomenti) che vengono specificati tra le parentesi tonde.

```
"Salve, mondo!"
```

`"Salve, mondo!"` è una **costante letterale stringa**. Rappresenta il messaggio che si intende visualizzare sullo schermo e dunque viene passata come argomento al metodo `WriteLine()`.

1.1 Struttura generale di un programma C#

L'esempio proposto non è rappresentativo di un programma realistico, ma fornisce già delle indicazioni. Primo: ogni programma è strutturato in base a uno "scheletro" predefinito, che nella sua forma più semplice è il seguente:

```
using System;
class nome-classe
{
    static void Main()
    {
        // qui stanno le istruzioni del programma
    }
}
```

L'ambiente di sviluppo genera automaticamente lo scheletro del programma nel momento della creazione di un nuovo progetto; questo può differire in alcuni dettagli da quello proposto.

Secondo, in C# molti elementi (come classi, metodi, ma non solo) presentano una sintassi simile:

```
nome
{
      // ciò sta qui dentro appartiene a nome
}
```

dove con *nome* si intende in questo caso una classe o un metodo. La coppia di parentesi graffe assume il nome di **blocco**; esse designano i limiti dell'elemento al quale si riferiscono.

In questo contesto il termine blocco può anche essere definito come **corpo**; e dunque: corpo della classe, corpo del metodo, ecc.

Terzo, per una convenzione stilistica, chiamata **indentazione del codice sorgente**, tutto ciò che sta dentro a un blocco viene spostato sulla destra di un numero di spazi fisso rispetto al margine allineato alle parentesi graffe di apertura e chiusura del blocco.

Quarto, ogni istruzione deve terminare con il carattere il punto-e-virgola. Questa regola grammaticale consente una notevole libertà sullo stile di scrittura del codice, della quale però non si dovrebbe abusare. Ad esempio, il programma potrebbe essere scritto nel seguente modo (assolutamente sconsigliato!):

```
static void Main()
{
   Console.
WriteLine
("Salve, mondo!");
}
```

Poiché tutti i programmi hanno la stessa fisionomia di base, e poiché le istruzioni del programma sono contenute nel metodo `Main()`[1], d'ora in avanti ci si limiterà a presentare soltanto tale metodo, assumendo implicitamente che il programma completo contenga anche il resto.

2 Calcolo dell'area e del perimetro di un rettangolo

Il problema del calcolo dell'area e del perimetro di un rettangolo, dati la base e l'altezza, è di facile soluzione. Un'implementazione in C# è la seguente.

```
static void Main()
{
   double b, h, a, p;  // qui sono dichiarate le variabili
   b = 2;              // implementa asserzione d'ingresso
   h = 4;              // implementa asserzione d'ingresso
   a = b * h;
   p = (b + h) * 2;
// la successiva istruzione implementa l'asserzione d'uscita
   Console.WriteLine("Area = {0}   Perimetro = {1}", a,b);
}
```

Il programma produce il seguente output:

```
Area = 8   Perimetro = 12
```

Esaminiamo una per una le righe del codice sorgente.

```
double b, h, a, p;
```

Questa rappresenta **un'istruzione di dichiarazione** (più semplicemente: **dichiarazione**) delle variabili utilizzate nel metodo. Essa crea nella memoria lo spazio necessario per contenere tali variabili. Le dichiarazioni fanno parte del **codice sorgente dichiarativo** del programma.

La parola chiave `double` designa il tipo di tutte e quattro le variabili. Il tipo `double` equivale al tipo reale, con l'importante differenza che `double` è un tipo finito, sia nella grandezza dei numeri rappresentabili sia nella precisione (numero di cifre significative).

La dichiarazione di una variabile deve precedere il suo impiego.

```
b = 2;
h = 4;
```

Entrambe sono due **istruzioni di assegnazione** e, come le istruzione successive, appartengono al **codice esecutivo** del programma. In questo esempio hanno lo scopo di fornire dei valori alla base e all'altezza del rettangolo, valori ai quali verrà applicato l'algoritmo.

[1] Ciò è vero soltanto per i programmi presentati nei primi cinque capitoli.

```
a = b * h;
p = (b + h) * 2;
```

Queste istruzioni di assegnazione rappresentano l'algoritmo vero e proprio; esse calcolano i dati di uscita e cioè il risultato del problema.

```
Console.WriteLine("Area = {0}   Perimetro = {1}", a, b);
```

Questa istruzione, attraverso la chiamata al metodo `WriteLine()` della classe `Console`, visualizza i dati precedentemente calcolati, comunicando all'utente il risultato del problema.

In questo esempio, al metodo `WriteLine()` vengono passati tre argomenti, separati da virgole. Inoltre, il primo argomento, una stringa, contiene al proprio interno delle combinazioni di caratteri – `{0}` e `{1}` – che fungono da segnaposti. Questa modalità d'uso di `WriteLine()` consente di visualizzare messaggi che rappresentano una combinazione di più argomenti, in genere parti testuali e numeriche.

Nell'esempio, il primo argomento del metodo, chiamato **stringa di formattazione**, specifica, mediante i segnaposti `{0}` e `{1}` in quale posizione, all'interno del testo visualizzato, dovranno essere inseriti i valori della variabile a (designata dal segnaposto `{0}`) e della variabile b (designata dal segnaposto `{1}`).

Mescolate alle istruzioni vi sono delle scritte precedute dal simbolo "`//`". Queste rappresentano dei **commenti al codice sorgente**. Un commento non fa parte del programma, non viene cioè preso in considerazione dal compilatore nella produzione del programma eseguibile; esso ha lo scopo di chiarire il significato di alcune istruzioni o parti di programma. I commenti rappresentano un ausilio al programmatore per aumentare la comprensibilità del codice sorgente.

Negli esempi presentati, saranno utilizzati per fornire alcune spiegazioni direttamente nel codice sorgente del programma.

3 Area e perimetro del rettangolo: seconda versione

L'implementazione del precedente algoritmo è corretta, ma di nessuna utilità. Infatti, il programma calcola sempre gli stessi valori di uscita e cioè 8 per l'area e 12 per il perimetro, poiché la base e l'altezza del rettangolo sono fisse e codificate direttamente nel codice sorgente attraverso le istruzioni `b = 2` e `h = 4`. Naturalmente vorremmo che il programma fosse in grado di calcolare area e perimetro di un rettangolo qualsiasi, consentendo all'utente di inserire i valori della base e dell'altezza.

Ciò si ottiene mediante il metodo `ReadLine()` della classe `Console`, l'uso di una variabile stringa, della classe `Convert` e infine un po' di codice sorgente in più.

```
static void Main()
{
    double b, h, a, p;
    string tmp;
    Console.WriteLine("Inserire i valori dei coefficienti 'a' e 'b'");
// le successive 4 istruzioni implementano l'asserzione di ingresso
    tmp = Console.ReadLine();
    b = Convert.ToDouble(tmp);
    tmp = Console.ReadLine();
    h = Convert.ToDouble(tmp);
// --------------------------- <b e h sono forniti>
```

```
    a = b * h;
    p = (b + h) * 2;

// la successiva istruzione implementa l'asserzione di uscita
    Console.WriteLine("Area = {0}   Perimetro = {1}", a, b);
}
```

L'esecuzione di questo programma determina quanto segue.

1) mediante il metodo `WriteLine()` il programma visualizza un messaggio informativo sui dati richiesti; quindi attende che l'utente digiti dalla tastiera il valore della base e confermi l'operazione premendo il tasto INVIO. Quest'ultimo comportamento viene ottenuto mediante il metodo `ReadLine()` e viene ripetuto anche per l'altezza.

2) dopo che l'utente ha inserito base e altezza, il programma calcola l'area e il perimetro e visualizza il risultato.

Vi sono diverse novità rispetto all'implementazione precedente, e saranno esaminate una per una.

```
string tmp;
```

Rappresenta la dichiarazione di una variabile di tipo stringa. L'uso di questa variabile è legato al modo di funzionare del metodo `ReadLine()`. Il nome `tmp`, diversamente dal nome `b` per base e `h` per altezza, è volutamente generico. La variabile non svolge alcun ruolo nell'algoritmo, ma serve soltanto come ausilio per l'acquisizione del valore della base prima e dell'altezza poi.

```
tmp = Console.ReadLine();
```

Il metodo `ReadLine()` sospende temporaneamente l'esecuzione del programma, attendendo che l'utente digiti dalla tastiera e prema il tasto INVIO; dopodiché, la sequenza di caratteri digitati dall'utente viene memorizzata nella variabile stringa `tmp`. Il programma riprende quindi la propria esecuzione.

La particolarità del metodo `ReadLine()` è che la sequenza di caratteri acquisita dalla tastiera viene sempre prodotta sotto forma di valore stringa, anche se si tratta di numeri. Per questo motivo è necessario dichiarare un'apposita variabile stringa da usare temporaneamente per memorizzare il valore introdotto dall'utente.

Se l'utente non digita nulla e preme subito INVIO, `ReadLine()` ritorna una stringa vuota, che non contiene cioè alcun carattere.

`ReadLine()` svolge dunque la funzione di ingresso dati consentendo all'utente di specificare un valore dalla tastiera per memorizzarlo in una variabile.

```
b = Convert.ToDouble(tmp);
h = Convert.ToDouble(tmp);
```

I dati inseriti dall'utente, devono essere convertiti dalla rappresentazione in forma di stringa a una rappresentazione numerica, necessaria per svolgere i calcoli. Ciò si ottiene attraverso la classe `Convert`, che espone numerosi metodi di conversione, tra cui `ToDouble()`. Il metodo `ToDouble()` richiede un argomento di tipo stringa e produce come risultato un valore equivalente di tipo `double`.

2

Costrutti di base

1 Selezione

Si consideri il problema di implementare un programma per il calcolo delle radici reali di una equazione di 2° grado, dati i suoi coefficienti. Esistono diverse possibilità, la più semplice delle quali consiste nel calcolo immediato delle radici X1 e X2 senza verificare che i valori A, B e C rendano possibile tale calcolo nell'insieme dei numeri reali. Ecco il programma:

```
static void Main()
{
    double a, b, c, x1, x2;
    string tmp;
    Console.WriteLine("Inserire i coefficienti 'a', 'b', 'c'");
// le successive istruzioni implementano l'asserzione di ingresso
    tmp = Console.ReadLine();
    a = Convert.ToDouble(tmp);
    tmp = Console.ReadLine();
    b = Convert.ToDouble(tmp);
    tmp = Console.ReadLine();
    c = Convert.ToDouble(tmp);
// -------------------------- <a, b, c sono fornite>
    x1 = (-b - Math.Sqrt(b*b - 4*a*c)) / (2*a);
    x2 = (-b + Math.Sqrt(b*b - 4*a*c)) / (2*a);
// la successiva istruzione implementa l'asserzione d'uscita
    Console.WriteLine("X1 = {0}   X2 =  {1}", x1, x2);
}
```

Esempio 2-1 Programma per la soluzione di un equazione di 2° grado.,

In presenza dei seguenti dati: `a: 2, b: 6; c: 0;` il programma produce correttamente il seguente output:

```
X1 = 7,5   X2 = 4,5
```

Come già detto, questa soluzione ha il limite di non considerare l'eventualità di un delta (termine sotto radice) negativo. Infatti, se l'utente inserisce i valori: `a: 2, b: 6; c: 6;` il programma produce il seguente output:

```
X1 = Non un numero   X2 =  Non un numero
```

Infatti, il metodo `Sqrt()`, nel calcolare la radice quadrata di un valore negativo (non ottenibile nell'insieme dei reali) produce come risultato un valore speciale, chiamato NaN (Not a Number), il

quale indica, appunto, un valore inesistente, che il metodo WriteLine() traduce automaticamente nella scritta "Non un numero".

Una versione migliore del programma dovrebbe procedere prima al calcolo del delta e, *soltanto dopo aver verificato che quest'ultimo sia positivo,* al calcolo le radici dell'equazione. Una simile soluzione utilizza uno **schema di selezione** il quale, in base al verificarsi di una condizione devia il flusso di esecuzione in uno dei due rami dello schema.

In C#, la selezione viene implementata attraverso i costrutti di controllo `if() else` e `if()`.

1.1 Costrutto if() else

Il costrutto di controllo `if() else` presenta la seguente sintassi:

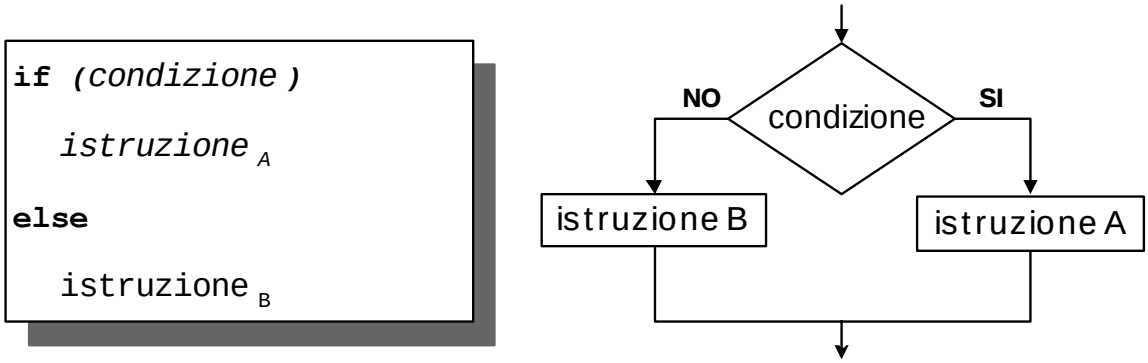
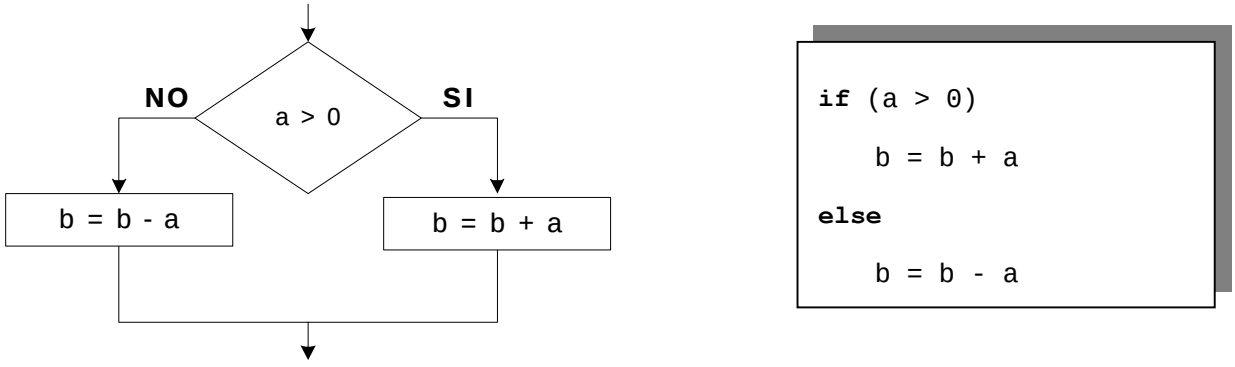


Figura 2-1 Sintassi del costrutto if() else e schema a blocchi equivalente.

Il costrutto eseguito nel seguente modo:

1) viene valutata la condizione tra parentesi;
2) se è vera, viene eseguita l'istruzione immediatamente successiva (*istruzione*$_A$),
3) altrimenti viene eseguita l'istruzione che segue `else` (*istruzione*$_B$).
4) in entrambi i casi l'esecuzione prosegue con l'istruzione successiva al costrutto.

Ad esempio, il frammento di algoritmo a sinistra viene tradotto nel codice a destra:



1.2 Costrutto if()

Il costrutto di controllo `if()` rappresenta una specializzazione del costrutto `if() else`; rispetto a quest'ultimo non comprende la parte `else`.

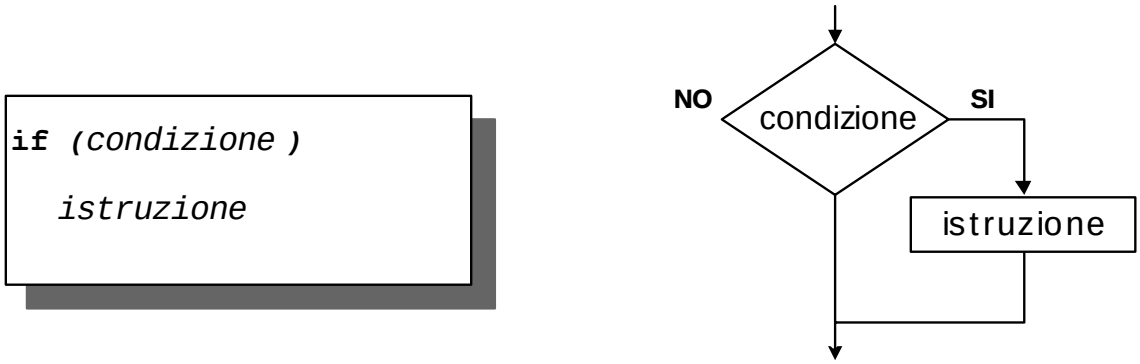


Figura 2-2 Sintassi del costrutto if() e schema a blocchi equivalente.

Ciò si traduce tra l'alternativa di eseguire un'istruzione o non eseguirne nessuna e passare immediatamente all'istruzione successiva al costrutto. Il costrutto `if()` implementa dunque la cosiddetta **sequenza condizionata**.

1.3 Uso del costrutto if() else

Con il costrutto `if() else` diventa possibile implementare un programma per la risoluzione di un'equazione di 2° grado che verifichi il delta dell'equazione prima di procedere al calcolo delle radici.

```
static void Main()
{
    double a, b, c, delta, x1, x2;
    string tmp;
    Console.WriteLine("Inserire i coefficienti 'a', 'b', 'c'");
// le successive istruzioni implementano l'asserzione di ingresso
    tmp = Console.ReadLine();
    a = Convert.ToDouble(tmp);
    tmp = Console.ReadLine();
    b = Convert.ToDouble(tmp);
    tmp = Console.ReadLine();
    c = Convert.ToDouble(tmp);
//------------------------------ <a, b, c sono forniti>
    delta = b*b - 4*a*c;
    if (delta >= 0)
    {
        x1 = (-b - Math.Sqrt(delta)) / (2*a);
        x2 = (-b + Math.Sqrt(delta)) / (2*a);
        Console.WriteLine("X1 = {0}   X2 =  {1}", x1, x2);
    }
    else
        Console.WriteLine("Non esistono radici reali");
}
```

Si possono fare alcune osservazioni sulla sintassi del costrutto `if() else` (e dunque anche sulla sua variante `if()`). La sintassi richiede che dopo la parte `if` e dopo la parte `else` sia specificata una sola istruzione. Il costrutto rappresenta quindi un'implementazione particolare dello schema di

selezione, il quale prevede sia nel ramo SI che in quello NO una **sequenza**, e cioè un elenco di una o più istruzioni.

Nell'esempio, dopo la parte `if (delta >=0)` segue un **blocco**, e cioè una coppia di parentesi graffe che racchiude una sequenza di istruzioni. Ebbene, un blocco implementa per l'appunto una sequenza, che sarà trattata nel prossimo paragrafo.

2 Sequenza: Blocco

Il termine blocco è già stato introdotto nel primo capitolo, indicato come elemento che designa il contenuto – corpo – di una classe o di un metodo.

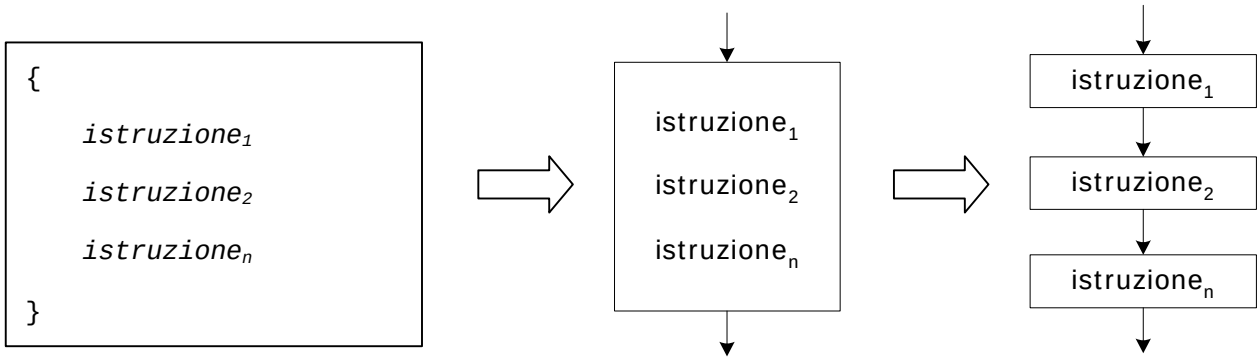


Figura 2-3 Sintassi del costrutto blocco e schema a blocchi equivalente.

E' necessario l'uso di un blocco quando l'algoritmo impone l'esecuzione di due o più istruzioni ma la sintassi del linguaggio richiede la presenza di una sola istruzione. E' questo il caso dei costrutti `if()` e `if() else`. Infatti, sia dopo la parte `if` che dopo la parte `else` la sintassi di C# ammette una sola istruzione.

Ad esempio, il seguente frammento di codice è valido nelle intenzioni ma errato nella sintassi:

```
if (a > 0)
    b = b + a;
    Console.WriteLine("b = {0}", b);
else
    Console.WriteLine("b = {0}", b);
```

Infatti si vorrebbe che le istruzioni:

```
b = b + a;
Console.WriteLine("b = {0}", b);
```

appartenessero entrambe alla parte `if` e dunque fossero eseguite insieme qualora la condizione `a > 0` risultasse vera. La sintassi del linguaggio, però, impone che venga considerata appartenente alla parte `if` soltanto l'istruzione immediatamente successiva. In questo caso l'uso di un blocco consente di rispettare i requisiti sintattici del linguaggio e allo stesso tempo di specificare come parte di una `if` o di una `else` una sequenza arbitrariamente lunga di istruzioni.

Un blocco può contenere anche un'istruzione soltanto, o addirittura nessuna istruzione (blocco vuoto). Dunque scrivere:

```
if (a < 0)
{
    Console.WriteLine("valore ass di a = {0}", -a);
```

```
}
else
{
    Console.WriteLine("valore ass di a = {0}", a);
}
```

è corretto, ed equivale a scrivere:

```
if (a < 0)
    Console.WriteLine("valore ass di a = {0}", -a);
else
    Console.WriteLine("valore ass di a = {0}", a);
```

Infatti, un blocco che contiene una sola istruzione è funzionalmente equivalente all'istruzione stessa.

E' inoltre corretto, anche se non molto appropriato, scrivere:

```
if (a < 0)
{
}
else
    Console.WriteLine("valore ass di a = {0}", a);
```

In questo caso, se la condizione `a < 0` risulta vera, non viene eseguita alcuna istruzione, poiché la parte `if` è seguita da un blocco vuoto.

3 Iterazione, o ciclo iterativo

Affrontiamo adesso un altro problema: trovare tutti i divisori di un numero dato, convenzionalmente chiamato N.

La soluzione consiste nello "scorrere" tutti i numeri naturali da 1 fino a N diviso 2, calcolando il resto della divisione tra N e il numero in esame; se tale resto è zero, il numero è un divisore di N, altrimenti no.

L'algoritmo presenta uno **schema di iterazione**, o **ciclo iterativo**, al cui interno una variabile X assume tutti i valori dell'intervallo che va da 1 a (N diviso 2). Quando il valore di X supera il limite superiore dell'intervallo il ciclo termina, e con esso l'algoritmo.

In C# esistono tre costrutti che implementano l'iterazione, qui ne tratteremo due: `while()` e `for()`.

3.1 Costrutto while()

Il costrutto `while()` presenta la seguente sintassi:

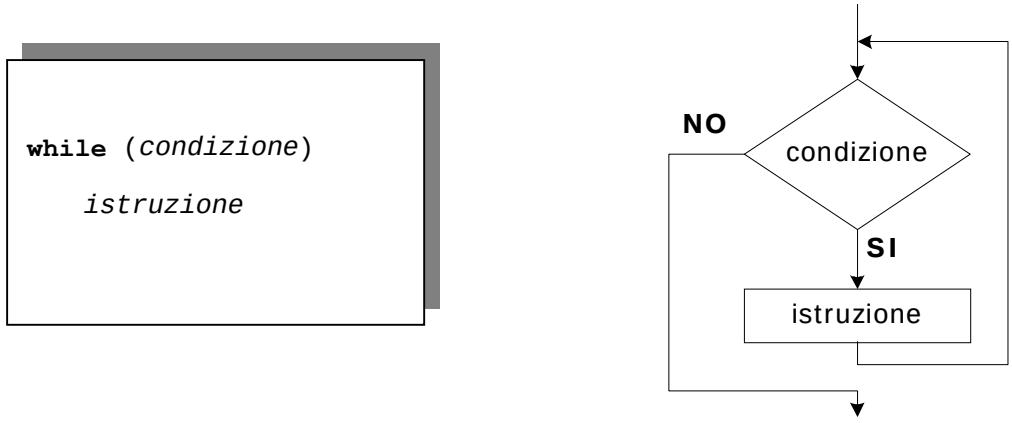


Figura 2-4 Sintassi del costrutto while() e schema a blocchi equivalente.

Viene eseguito nel seguente modo:

1) viene valutata la condizione chiusa tra parentesi, chiamata anche **condizione di controllo del ciclo** o semplicemente **controllo del ciclo**;

2) se la condizione è vera, viene eseguita l'istruzione immediatamente successiva, chiamata **corpo del ciclo**, dopodiché si ritorna al punto a);

3) se invece la condizione è falsa il ciclo termina.

3.2 Uso del costrutto while()

Il seguente codice mostra l'impiego del ciclo while per il calcolo dei divisori di un numero N:

```
static void Main()
{
    int n, x, r;
    string tmp;
    Console.WriteLine("Inserire il numero del quale trovare i divisori");
    tmp = Console.ReadLine();
    n = Convert.ToInt32(tmp);
// ---------------------------<n è fornito>
    x = 1;
    while (x <= n / 2)
    {
        r = n % x;
        if (r == 0)
            Console.WriteLine("{0} è un divisore di {1}", r, n);
        x = x + 1;
    }
}
```

Se l'utente inserisce il valore `10`, il programma produce sullo schermo:

```
1 è un divisore di 10
2 è un divisore di 10
5 è un divisore di 10
```

Nota bene: se l'utente inserisce 1 il programma non produce alcun output. Ciò dipende dal fatto che il ciclo `while()` rappresenta un ciclo iterativo con **controllo in testa**, dove per ogni iterazione viene prima verificata la condizione *e soltanto dopo, se questa risulta vera, viene eseguito il corpo del ciclo*. Naturalmente, se la condizione risulta subito falsa il corpo del ciclo non viene mai eseguito.

Ed è ciò che accade se N è uguale a 1, infatti: 1 non è inferiore o uguale alla metà di 1.

In questo programma il tipo delle variabili è `int`, che corrisponde al tipo intero. L'argomento dei tipi di dati viene affrontato nel capitolo successivo, per ora è importante sottolineare il fatto che per memorizzare il contenuto della stringa `tmp` nella variabile `n` viene fatto uso del metodo `ToInt32()` della classe `Convert`.

3.3 Costrutto for()

Supponiamo di voler realizzare un programma che implementi l'algoritmo che sommi i numeri naturali compresi tra 1 e un numero dato. Occorrerà naturalmente utilizzare un ciclo iterativo, come il `while()`:

```
static void Main()
{
    int n, somma, x;
    string tmp;
    Console.WriteLine("Inserire il numero N ");
    tmp = Console.ReadLine();
    n = Convert.ToInt32(tmp);
// ---------------------- <n è fornito>
    somma = 0;
    x = 1;
    while (x <= n)
    {
        somma = somma + x;
        x = x + 1;
    }
    Console.WriteLine("La somma è: {0}", somma);
}
```

Esempio 2-2 Programma che somma i numeri naturali fino a un numero dato.

Si nota immediatamente una forte similitudine tra questo programma e l'esempio precedente; in entrambi una variabile assume il ruolo di **contatore**, e cioè:

1) la variabile viene impostata a un valore iniziale;
2) viene incrementata ad ogni iterazione del ciclo;
3) raggiunge un valore finale che determina la fine del ciclo.

In entrambi i casi il ciclo `while()` svolge la funzione di reiterare per un certo numero di volte una determinata operazione; per ottenere questo viene utilizzata una variabile il cui scopo è appunto quello di contare quante volte è stata eseguita l'operazione.

E' questo un uso molto comune dei cicli iterativi; per questo motivo C# mette a disposizione un secondo costrutto di controllo che implementa l'iterazione, il ciclo `for()`. Questo è funzionalmente equivalente al ciclo `while()`, ma caratterizzato da una sintassi che facilita la realizzazione di cicli iterativi che svolgono un conteggio.

```
for (inizializ.ne_opz; condizione_opz; incremento_opz)
    istruzione
```

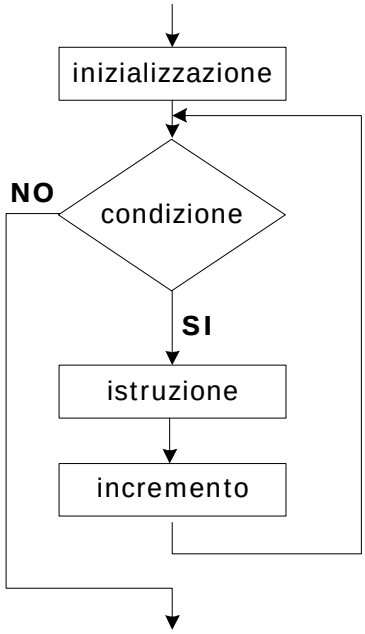


Figura 2-5 Sintassi del costrutto for() e schema a blocchi equivalente.

Il ciclo `for()` consente di specificare all'interno delle parentesi gli elementi tipici di un ciclo iterativo che svolge un conteggio, e cioè:

- inizializzazione della variabile contatore;
4) condizione di controllo del ciclo;
5) incremento della variabile contatore.

E' importante comprendere che il ciclo `for()` non aggiunge niente di sostanziale al ciclo `while()`, infatti lo schema a destra in fugura può essere realizzato anche nel seguente modo:

```
inizializzazione
while (condizione)
{
    istruzione
    incremento
}
```

Figura 2-6 Struttura di un ciclo while() funzionalmente uguale a un ciclo for().

Semplicemente, il ciclo `for()` ha una maggiore compattezza sintattica.

3.4 Uso del costrutto for()

Di seguito viene mostrata una nuova implementazione del programma che calcola i divisori di N, questa volta basata sull'impiego di un ciclo `for()`.

```
static void Main()
{
    int n, somma, x;
    string tmp;
```

```
    tmp = Console.ReadLine();
    n = Convert.ToInt32(tmp);
// ---------------------- <n è fornito>
    somma = 0;
    for (x=1; x <= n; x = x + 1)
        somma = somma + x;
    Console.WriteLine("La somma è: {0}", somma);
}
```

3.5 Cicli for() che non definiscono inizializzazione, condizione o incremento

La sintassi del ciclo `for()` stabilisce che le parti di *`inizializzazione`*, *`condizione`* e *`incremento`* sono opzionali e che dunque possono essere omesse. Ad esempio, il seguente codice:

```
int i = 0;
for (; i < 10; i = i + 1)     // omissione della parte inizializzazione
    istruzione;
```

è formalmente corretto. Come lo è:

```
int i;
for (i = 0; i < 10;)          // omissione della parte incremento
{
    istruzione;
    i = i + 1;
}
```

Infine, è lecito anche scrivere:

```
for (;;)          // omesse tutte e tre le parti
    istruzione;
```

Nell'ultima forma, il ciclo `for()` reitererà indefinitamente; esso rappresenta dunque ciò che si dice un **loop infinito**, poiché la condizione, non esistendo nemmeno, non potrà mai diventare falsa.

> Nota bene: non esiste qualcosa di analogo per il ciclo `while()`. Dunque, è sbagliato scrivere:
>
> ```
> while()
>   istruzione
> ```

3.6 Condizione di controllo dei cicli iterativi

La condizione di controllo dei cicli iterativi `while()` e `for()` viene spesso definita **condizione di fine del ciclo**, poiché è mediante essa che viene stabilito quando il ciclo deve terminare. La definizione è po' fuorviante; se infatti si osserva lo schema a blocchi equivalente ai costrutti `while()` e `for()` si vede che l'interpretazione è la seguente:

fintantoché la condizione è vera il ciclo prosegue; nel momento in cui diventa falsa il ciclo termina.

La condizione di controllo di un ciclo, dunque, più che condizione di terminazione dovrebbe essere definita in modo più appropriato "condizione di proseguimento"; poiché finché è vera determina il proseguimento del ciclo iterativo.

4 Schema di selezione multipla: costrutto switch()

La **selezione multipla** rappresenta una variazione del concetto di selezione. Laddove quest'ultima determina l'esecuzione di una tra due alternative possibili, la selezione multipla consente la scelta tra un numero arbitrario di alternative.

Dal punto di vista schematico la selezione multipla può essere così rappresentata:

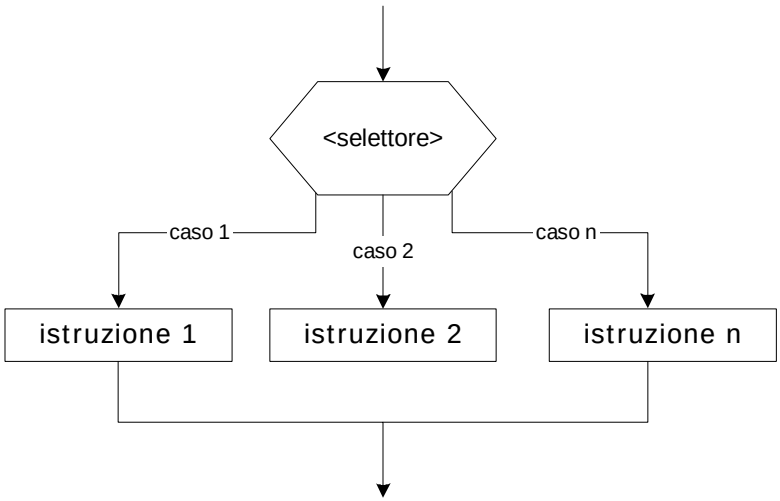


Figura 2-7 Rappresentazione dello schema di selezione multipla.

Lo schema di seleziona multipla si legge nel seguente modo:

1) viene valutata l'espressione **selettore** e il suo valore viene confrontato con delle costanti, nello schema rappresentate da $caso_1$, $caso_2$ e $caso_n$;

2) viene eseguita l'istruzione associata alla costante il cui valore è uguale a quello del selettore;

3) infine l'esecuzione procede con l'istruzione successiva al costrutto.

Se nessuna delle costanti è uguale al valore del selettore non viene eseguita nessuna istruzione e il flusso di esecuzione procede immediatamente con l'istruzione successiva al costrutto.

4.1 Costrutto switch()

Il costrutto `switch()` fornisce una particolare implementazione dello schema di selezione multipla. Esso assume la seguente sintassi, qui presentata in una forma semplificata:

```
switch(selettore)
{
    case costante: istruzioni; break;
    case costante: istruzioni; break;
    .
    .
    default: istruzioni-default; break;
}
```

L'espressione `selettore` può appartenere ai tipi `int`, `bool`, `string` o `char`, ma non `double`. Il selettore viene confrontato con ognuna delle **costanti case**, che devono appartenere allo stesso tipo. Alla prima corrispondenza trovata viene eseguita l'istruzione o le istruzioni associate; dopodiché l'esecuzione procede con l'istruzione successiva al costrutto. Nel caso non via sia alcuna

corrispondenza tra il valore di *selettore* e una delle costanti case specificate vi sono due possibilità:

1) se non è stata specificata la parte **default**: l'esecuzione procede immediatamente all'istruzione successiva al costrutto,

2) altrimenti viene eseguita l'istruzione o le istruzioni specificate nella parte default.

La parte default consente di indicare le istruzioni da eseguire se nessuna delle alternative specificate corrisponde al valore del selettore.

Il costrutto `switch()`può essere tradotto in una serie di `if() else` in cascata, come mostrato dalla seguente sintassi:

```
if (selettore == costante1)
  istruzione1;
else
if (selettore == costante2)
  istruzione2;
else
...
if (selettore == costante3)
  istruzione3;
else opz
  istruzione-default;
```

In realtà lo `switch()` è meno flessibile, per due ragioni:

- il tipo ammesso dell'espressione *selettore* rientra in un sotto insieme ristretto dei tipi offerti dal linguaggio;

- tra l'espressione *selettore* e le costanti case avviene sempre un confronto per uguaglianza; non vi è modo per stabilire un diverso tipo di corrispondenza, come ad esempio: maggiore di, minore di, eccetera.

4.2 Uso del costrutto switch()

Il costrutto `switch()` è particolarmente adatto quando il valore dell'espressione da testare ricade in un insieme ristretto di valori possibili, per ognuno dei quali è previsto un determinato compito da svolgere. Si consideri come esempio il seguente problema:

Si vuole realizzare un menù di scelta che consenta di visualizzare la mansione svolta da un dipendente una volta inserito il suo codice di mansione. Seguono i codice di mansione con le relative descrizioni:

"AA" – Apprendista

"O1" – Operaio di 1° livello

"O2" – Operaio di 2° livello

"IM" – Impiegato

"CP" – Capo reparto

Ecco l'implementazione del programma:

```
static void Main()
{
   bool uscita;
```

```
    string scelta
    string mansione = "";

    uscita = false;
    while (!uscita)
    {
        Console.WriteLine("Selezionare la mansione da svolgere");
        Console.WriteLine();
        Console.WriteLine("AA - Apprendista");
        Console.WriteLine("O1 - Operaio di 1° livello");
        Console.WriteLine("O2 - Operaio di 2° livello");
        Console.WriteLine("IM - Impiegato");
        Console.WriteLine("CP - Capo reparto");
        Console.WriteLine("XX - Uscita");
        scelta = Console.ReadLine();
        mansione = "";
        switch (scelta)
        {
            case "AA": mansione = "Apprendista"; break;
            case "O1": mansione = "Operaio di 1° livello"; break;
            case "O2": mansione = "Operaio di 2° livello"; break;
            case "IM": mansione = "Impiegato"; break;
            case "CP": mansione = "Capo reparto"; break;
            case "XX": uscita = true; break;
            default: mansione = "ERRORE: mansione sconosciuta"; break;
        }
        if (!uscita)
        {
            Console.WriteLine("Mansione selezionata:{0} ", mansione);
        }
        Console.WriteLine();
    }
}
```

Poiché l'input dell'utente può assumere uno tra sei possibili valori (compreso "XX" che identifica la scelta di terminate il programma), l'uso del costrutto `switch()` risulta più elegante e compatto di una sequenza di sei if().

Nota bene: il programma prevede che l'utente possa inserire un codice inesistente. Tale possibilità viene gestita nella parte default del costrutto `switch()`, che viene eseguita ogni qual volta non esiste corrispondenza tra il codice del dipendente e almeno una delle costanti case.

4.3 Associare un'istruzione a più costanti case

La sintassi del costrutto `switch()` prevede la possibilità che più costanti case facciano riferimento alla stessa istruzione. Questoscenario ciò assume la seguente forma:

```
switch(selettore)
{
    case costante1:
    case costante2:
```

```
                    case costante3: istruzioni; break;
                    .
                    .
                }
```

La possibilità di avere delle costanti case vuote è utile quando tra l'insieme dei valori che può assumere il selettore vi è un sotto insieme (o più di uno) che richiede l'esecuzione dello stesso compito.

Per chiarire il concetto, si consideri una variazione al problema precedente:

La tabella dei codici delle mansioni è stata modificata nel seguente modo:

"AP" – Apprendista (vecchio codice: "AA")

"IP" – Impiegato (vecchio codice: "IM")

Si desidera che il programma tenga conto delle modifiche, ma anche che resti compatibile con i vecchi codici.

Ecco la nuova implementazione:

```
static void Main()
{
    bool uscita;
    string scelta
    string mansione = "";

    uscita = false;
    while (!uscita)
    {
        Console.WriteLine("Selezionare la mansione da svolgere");
        Console.WriteLine();
        Console.WriteLine("AA - Apprendista");
        Console.WriteLine("O1 - Operaio di 1° livello");
        Console.WriteLine("O2 - Operaio di 2° livello");
        Console.WriteLine("IM - Impiegato");
        Console.WriteLine("CP - Capo reparto");
        Console.WriteLine("XX - Uscita");
        scelta = Console.ReadLine();
        mansione = "";
        switch (scelta)
        {
            case "AA" :
            case "AP" : mansione = "Apprendista"; break;
            case "O1": mansione = "Operaio di 1° livello"; break;
            case "O2": mansione = "Operaio di 2° livello"; break;
            case "IM" :
            case "IP" : mansione = "Impiegato"; break;
            case "CP": mansione = "Capo reparto"; break;
            case "XX": uscita = true; break;
            default: mansione = "ERRORE: mansione sconosciuta"; break;
        }
        if (!uscita)
```

```
        {
            Console.WriteLine("Mansione selezionata:{0} ", mansione);
        }
        Console.WriteLine();
    }
}
```

Le costanti `"AA"` e `"AP"` sono associate alla stessa istruzione, discorso analogo vale per le costanti `"IM"` e `"IP"`. Quando nell'esame delle costanti case viene trovata una corrispondenza con una costante vuota l'esame continua fino alla prima costante associata a una o più istruzioni.

3

Tipi, variabili ed espressioni

1 Concetto di Tipo

Il concetto intuitivo di **tipo di dato** è noto a tutti e fa riferimento alla natura e al significato che diamo dell'informazione. Ad esempio, il nome di una persona è un'informazione di natura diversa rispetto alla sua età. Entrambi si scrivono mediante sequenze di simboli, ma tali sequenze assumono due significati diversi. Le lettere che compongono un nome non sono soggette ad alcuna interpretazione, semplicemente identificano, mediante la loro combinazione, un certo individuo. Le cifre che compongono un'età esprimono un valore quantitativo preciso e delle relazioni altrettanto precise; ad esempio: un uomo di 25 anni è più giovane di un uomo di 75. E ancora, il colore dei capelli di una persona si esprime con una combinazione di lettere (rossi, biondi, scuri, eccetera). Come per i nomi, tali combinazioni non assumono significati particolari al di là della caratteristica (colore dei capelli) che identificano. Nonostante ciò, nessuno compilerebbe mai un elenco fatto per metà di colori di capelli e per metà di nomi di persona; tutti i e due i tipi sono contraddistinti da combinazioni di lettere, ma quelle combinazioni hanno significati diversi tra loro: sono tipi diversi di informazioni. Infine, un numero di telefono è rappresentato da una combinazione di cifre (perlomeno in Italia) esattamente come un'età, ma assume un significato diverso. Un numero di telefono non designa una quantità di qualcosa; non si confrontano tra loro numeri di telefono, se non per stabilire se sono uguali oppure no.

In conclusione, il concetto che abbiamo di tipo di dato è essenzialmente semantico, si riferisce cioè al significato che assumono le informazioni di quel tipo. Nei linguaggi di programmazione, e C# non fa eccezione, le cose sono abbastanza diverse. In essi il concetto di tipo di dato si richiama alle modalità di:

1) **memorizzazione e codifica**: come viene codificato in memoria il dato e quanta memoria occupa; qual è, se è definito, l'intervallo di valori che il dato può assumere;

2) **rappresentazione**: in che modo vengono rappresentati i valori costanti, chiamati **letterali**;

3) **elaborazione**: l'insieme delle operazioni che si possono effettuare con il dato;

4) **relazioni con altri tipi di dati**: compatibilità e conversioni che possono essere effettuate da e verso altri tipi di dati.

C# stabilisce delle precise regole formali che caratterizzano completamente i punti sopra elencati per ogni tipo di dato, al di là del significato che valori di quel tipo possono assumere nella logica del programma.

2 Tipi di dati numerici: int e double

C# mette a disposizione vari tipi di dati numerici; di questi saranno per il momento analizzati il tipo `int` e `double`. Prima di tutto, però, occorre rispondere a un quesito: perché avere più tipi numerici? Non sarebbe sufficiente un tipo soltanto? La risposta è: sarebbe sufficiente, ma sarebbe anche inefficiente. Il perché sta nella codifica, nell'occupazione di memoria e nell'efficienza delle operazioni.

2.1 Il tipo double

Il tipo `double` corrisponde all'insieme dei numeri reali e viene definito un tipo **virgola-mobile** (floating-point), termine che ne designa la modalità di codifica in memoria. Un valore appartenente al tipo `double` occupa 8 byte di memoria e rientra nel seguente **intervallo di variazione** (qui espresso attraverso valori positivi):

valore più piccolo: (circa) 5.0×10^{-324}

valore più grande: (circa) 1.7×10^{308}

Ciò significa, ad esempio, che il numero: 1.0×10^{-400} non può essere rappresentato attraverso il tipo `double`: è un numero troppo vicino allo zero.[2]

Il tipo `double` possiede una **precisione** di 16 cifre. Per precisione si intende il numero di cifre realmente necessarie per distinguere un numero da un altro di grandezza equivalente (cifre significative). Ciò corrisponde al numero di cifre decimali dopo che il numero è stato ricondotto alla forma:

0.<cifre decimali> x 10 <esponente>

Ad esempio:

1.23	**contiene 3 cifre significative:**	$0.\underline{123} \times 10^{1}$
0.0012	**contiene 2 cifre significative:**	$0.\underline{12} \times 10^{-2}$
0.1200	**contiene 2 cifre significative:**	$0.\underline{12} \times 10^{0}$
12340000	**contiene 4 cifre significative:**	$0.\underline{1234} \times 10^{8}$

Dunque, il seguente numero:

12345678912345678

che contiene 17 cifre diverse da zero viene ricondotto alla forma:

12345678912345670 che corrisponde a: $0.1234567891234567 \times 10^{17}$

forma che non rappresenta l'ultima cifra, e cioè 8.

Rispetto al tipo `int`, il numero di byte di memoria occupata e il costo computazionale[3] legato alle operazioni con valori `double` rendono questo tipo appropriato soltanto quando la natura del dato da

[2] Il tipo `double` rappresenta i valori al di fuori del proprio intervallo di variazione mediante particolari codifiche. Ad esempio, tutti i valori troppo vicini allo zero per essere rappresentati vengono approssimati a zero.

[3] Per **costo computazionale** di una operazione si intende, approssimativamente, il tempo impiegato dalla CPU per compierla. Un'operazione tra due valori di tipo `double` possiede un costo computazionale molto superiore a quello della stessa operazione eseguita tra due valori di tipo `int`, indipendentemente dalla grandezza dei valori in gioco.

rappresentare, la sua semantica, è effettivamente di tipo reale. Ad esempio: volumi, altezze, pesi, valori monetari, ma non valori cardinali (numerosità di un insieme) o codici numerici.

2.2 Il tipo int

Il tipo `int` corrisponde all'insieme dei numeri interi con segno. Un valore di tipo `int` occupa 4 byte di memoria e rientra nel seguente intervallo di variazione:

```
valore più piccolo:     -2147483648

valore più grande:      2147483647
```

Ciò significa, ad esempio, che il numero 10000000000 non può essere rappresentato attraverso il tipo `int`, poiché è un numero troppo grande.

E' appropriato l'uso del tipo `int` ogni qualvolta la semantica del dato da rappresentare è effettivamente di tipo intero, poiché questo garantisce una minore occupazione di memoria e un minor costo computazionale. Vi sono inoltre alcune operazioni, come la divisione a quoziente intero e l'accesso con indice a un array, che richiedono espressamente valori di tipo `int`.

2.3 Costanti letterali numeriche

Il tipo di una variabile viene specificato nella sua dichiarazione e prescinde dai valori che ad essa saranno assegnati. Ad esempio, nel seguente codice:

```
double x;
int a = 10;
x = a;
```

la variabile `x` è dichiarata di tipo `double`; il fatto che ad essa venga assegnato il valore di una variabile `int` è assolutamente irrilevante. `x` può contenere valori come 10, 100000, 0.223333 o 1.0 X 10^{100}, continuerà ad occupare 8 byte di memoria e le operazioni effettuate con essa avranno sempre il medesimo costo computazionale.

Ciò detto, cos'è che definisce il tipo di una costante letterale? Si ricorda che una **costante letterale**, o semplicemente costante, rappresenta un valore espresso dalla stessa notazione usata per descriverlo. Ebbene, il tipo di una costante è per l'appunto definito dalla notazione usata per descriverlo. Ad esempio, il valore `1` è una costante di tipo intero. Il valore `1.0`, perfettamente equivalente al precedente, è una costante di tipo `double`. Sono valori identici, ma di tipo diverso.

Per definizione, un numero rappresenta una costante intera se non possiede decimali e rientra nell'intervallo di variazione del tipo `int`. Per le costanti `double` esistono invece varie forme di rappresentazione, elencate nella Tabella 3-1.

La forma a virgola mobile consente di specificare valori molto grandi (o molto piccoli) attraverso una notazione compatta. L'uso del suffisso '`D`' o '`d`' consente di indicare esplicitamente che il letterale è di tipo `double`. Ciò risulta utile per distinguere un letterale `double` da altri tipi di letterali reali (qui non trattati), oppure per usare notazioni del tipo `10D` al posto di `10.0`, quando si desidera forzare un valore intero al tipo `double`.

Tabella 3-1 Esempi di costanti letterali double.

RAPPRESENTAZIONE	ESEMPI		EQUIVALGONO RISPETTIVAMENTE A	
forma a virgola fissa:	1.23	2000.0	0.123×10^{1}	0.2×10^{4}
forma a virgola mobile	1.2E10	–1.2e3	0.12×10^{11}	-0.12×10^{4}
uso del suffisso 'D' o 'd'	1D	1E10d	0.1×10^{1}	0.1×10^{11}

3 Tipi di operatori e conversioni numeriche

La diversità tra i tipi `int` e `double` non riguarda soltanto l'occupazione di memoria, l'intervallo di variazione e l'efficienza computazionale. Si consideri ad esempio la seguente istruzione:

```
Console.WriteLine("5/2 = {0}   5/2 = {1}", 5/2, 5.0/2.0);
```

Viene eseguito per due volte lo stesso calcolo, ma i risultati non sono esattamente quelli che ci si potrebbe attendere. Infatti viene prodotto sullo schermo:

```
5/2 = 2  5/2 = 2,5
```

Sembra, e in effetti è così, che siano state eseguite due diverse forme di divisione. Nel primo caso una divisione a quoziente intero (divisione intera), nel secondo caso una divisione con la virgola (divisione reale).

Il concetto di tipo, dunque, non si applica soltanto ai dati, ma anche agli operatori che operano su di essi:

esistono operatori aritmetici distinti, `int` e `double`, che si applicano rispettivamente a valori `int` e `double`, e producono come risultato valori `int` o `double`.

I simboli utilizzati per gli operatori sono gli stessi (+, –, *, /, eccetera), ma in realtà essi vengono tradotti in tipi diversi di operazioni, in base al tipo degli operandi.

3.1 Conversioni tra tipi numerici

Si consideri ora un terzo calcolo.

```
Console.WriteLine("5/2 ={0}", 5.0/2);
```

Quale divisione viene effettuata, tra reale e intera, considerato che gli operandi non sono entrambi `double` né entrambi `int`? La risposta è: viene effettuata la divisione reale. Infatti viene visualizzato.

```
5/2 = 2,5
```

Esiste dunque una regola che si applica quando i due operandi di un operatore non sono dello stesso tipo e precisamente:

se uno dei due operandi è di tipo `double`, indipendentemente dal tipo dell'altro operando, viene sempre invocato l'operatore di tipo `double`.

Poiché, però, l'operatore `double` richiede entrambi gli operandi di tipo `double`, l'operando intero viene **convertito implicitamente** (in modo automatico) in un `double` prima che sia applicato

l'operatore. Convenzionalmente, si dice che l'operando intero viene **promosso** a un tipo più grande, poiché viene convertito in un tipo che possiede un intervallo di variazione maggiore.

3.2 Compatibilità tra i tipi int e double

Vi è dunque una certa compatibilità tra i tipi `int` e `double`, poiché in alcune situazioni possono essere utilizzati contemporaneamente senza che vengano segnalati degli errori formali o prodotti dei calcoli errati. Non sempre è così però. Si consideri il seguente frammento di codice:

```
int a;
a = 2.4;
```

In questa istruzione si tenta di assegnare un valore `double` a una variabile di tipo `int`; in un certo senso, si tenta di "far entrare" un valore di tipo "grande" in una variabile di tipo "piccolo". Ebbene, in questo caso C# segnala un errore formale, poiché un simile assegnazione porterebbe a una perdita di informazione, in quanto il tipo `int` non è in grado di memorizzare i decimali. Grossolanamente parlando: il grande non può essere convertito nel piccolo.

Attenzione, è estremamente importante comprendere che l'impossibilità di convertire da grande a piccolo non è una questione di valori ma di tipi. Un secondo esempio chiarirà meglio il concetto:

```
int a;
double x;
x = 1;
a = x;
```

Dato che `x` contiene il valore 1, ci si può chiedere se sia corretta l'assegnazione `a = x`. La risposta è no. Il fatto che `x` contenga un valore senza la virgola è irrilevante; il compilatore, nel verificare se una assegnazione rispetti le regole, si limita a verificare i tipi e non i valori. E poiché una variabile `double` *potrebbe* contenere un valore non codificabile nel tipo `int`, per sicurezza proibisce tutte le assegnazioni da `double` a `int`, dato che potrebbero portare a una perdita di informazione.

Questa proibizione viene designata dalla regola:

non esiste una conversione implicita da `double` a `int`.

Alla luce di quanto detto è lecito chiedersi se sia corretto scrivere:

```
int a;
double x;
a = 1;
x = a;
```

La risposta è sì. Infatti sappiamo già che esiste una conversione implicita da `int` a `double`, e cioè da piccolo a grande. Tale conversione è permessa dal linguaggio poiché, non porterà mai a una perdita di informazione, qualunque sia il valore da convertire.

4 Tipo string

Il tipo `string` consente la rappresentazione di informazioni alfanumeriche e cioè sequenze di caratteri, che possono rappresentare nomi, indirizzi, codici fiscali, numeri di telefono, eccetera.

Si può immaginare un valore di tipo `string` – detto anche **stringa** – come una sequenza, o collezione, di caratteri, ognuno dei quali è codificato mediante il codice Unicode. La memoria

occupata da una stringa non è fissa come per i tipi `int` e `double`, ma dipende dal numero di caratteri, ognuno dei quali occupa due byte.

Ad esempio, la stringa `"Salve, Mondo!"` è memorizzata nel seguente modo[4]:

Figura 3-1 Rappresentazione in memoria della stringa "Salve, Mondo!".

> In realtà, in memoria non vengono codificati i caratteri (cosa impossibile) ma i corrispondenti valori numerici tratti dalla tabella dei codici Unicode.

4.1 Costanti letterali stringa

Una costante letterale stringa viene designata racchiudendo i caratteri che la formano all'interno di una coppia di virgolette. Dunque, letterali stringa sono:

```
"Salve, mondo!"   "ciao come va!"   "Bill Gates"      "055/12345678"
```

E' importante comprendere che è la notazione usata a denotare un letterale stringa e non il valore letterale in sé. Ciò detto, i seguenti letterali:

```
"123"      "0,25"     "10,5E-10"          "-100000"
```

sono costanti stringa e non costanti numeriche. Infatti sarebbe un errore scrivere:

```
int a;
a = "10";      // errore: "10" è una stringa!
```

poiché `"10"` è un letterale stringa e non si può assegnare una stringa a una variabile di tipo `int`.

Stringhe vuote

Nell'ambito delle stringhe esiste il concetto di valore vuoto, che corrisponde a una stringa che non contiene alcun carattere. Essa viene rappresentata sempre mediante la coppia di virgolette, con le virgolette finali subito adiacenti a quelle iniziali: `""`.

E' importante non confondere la stringa vuota con una stringa che contiene uno o più spazi spazio, come `" "`. Visivamente sono molto simili, ma rappresentano due letterali stringa diversi.

4.2 Funzione delle stringhe nella programmazione

Il tipo `string` svolge due funzioni fondamentali nella programmazione:

- supporto alla programmazione;
- rappresentazione ed elaborazione di informazioni di carattere alfanumerico;

Cominciamo ad esaminare la prima funzione, della quale abbiamo già visto alcuni esempi.

[4] La cella apparentemente vuota dopo la virgola memorizza il carattere spazio, al quale, per definizione, non corrisponde un simbolo visibile. In realtà, esistono delle notazioni appropriate per specificare tale carattere; saranno studiate contestualmente all'introduzione del tipo `char`.

4.3 Stringhe come supporto alla programmazione

Anche in programmi che trattano dati di natura numerica esiste comunque la necessità di tradurre tali dati in forma di stringa o viceversa.

Si consideri ad esempio un programma che dato il lato di un quadrato ne calcoli l'area.

```
static void Main()
{
    double lato, area;
    string tmp;
    Console.WriteLine("Inserire il valore del lato");
    tmp = Console.ReadLine();
    lato = Convert.ToDouble(tmp);
    area = lato * lato;
    Console.WriteLine("Area = {0}", area);
}
```

In questo esempio, l'uso della variabile stringa `tmp` non è un requisito dell'algoritmo, che necessita delle sole variabili `lato` e `area`. La stringa è però un requisito del programma, poiché i valori inseriti dall'utente vengono prodotti dal metodo `ReadLine()` sotto forma di rappresentazione alfanumerica, e cioè di stringa. Questa dev'essere convertita in forma numerica attraverso il metodo `ToDouble()` della classe `Convert`, poiché non si possono fare calcoli con le stringhe, anche se contengono sequenze di cifre. Inoltre, anche per visualizzare il risultato viene fatto uso di una stringa, e precisamente della costante letterale `"Area = {0}"`.

4.4 Stringhe per la rappresentazione ed elaborazione di informazioni di carattere alfanumerico

Non sono molti i programmi che elaborano soltanto numeri, poiché la maggior parte delle informazioni del mondo reale è di natura alfanumerica.

A titolo di esempio viene mostrato un programma che chiede all'utente di inserire nome e password, verificando la correttezza della seconda con un valore di riferimento codificato come costante stringa.

```
static void Main()
{
    string nome;
    string password;
    Console.WriteLine("Immettere nome utente:");
    nome = Console.ReadLine();      // chiede il nome dell'utente
    password = "";                  // per forzare l'entrata nel ciclo
    while (password != "matrix")
    {
        Console.WriteLine("Immettere password:");
        password = Console.ReadLine();    // continua a richiedere la password
        if (password != "matrix")
            Console.WriteLine("Password non riconosciuta:");
    }
    Console.WriteLine("Accesso al sistema confermato");
}
```

Il programma funziona nel seguente modo:

1) all'utente viene chiesto il proprio "nome utente";

2) l'esecuzione entra in un ciclo `while()`, all'interno del quale viene richiesta la password; il ciclo termina solo se l'utente inserisce quella giusta e cioè `"matrix"`;

Nell'esempio, le due variabili `nome` e `password` e la costante stringa `"matrix"` svolgono un ruolo di rappresentazione ed elaborazione delle informazioni necessarie al programma e non di semplice supporto per l'esecuzione di altre elaborazioni.

5 Tipo bool

Nonostante non sia mai apparso in forma esplicita, gli esempi presentati finora hanno già fatto uso del tipo `bool`, o **tipo booleano**: esso, infatti, è il tipo delle **espressioni condizionali**, chiamate anche **espressioni booleane**, usate nelle `if()` e nei cicli iterativi.

Nel seguente codice:

```
if (a > b)
    Console.WriteLine("a è maggiore di b");
else
    Console.WriteLine("a non è maggiore di b");
```

la valutazione dell'espressione booleana `a > b` produce un valore di tipo `bool`, che può essere soltanto **vero** o **falso**, in base al fatto che il valore contenuto in `a` sia o no maggiore del valore contenuto in `b`.

Un valore di tipo booleano occupa un byte di memoria e la sua elaborazione ha un costo computazionale molto basso. Ovviamente, come per tutti i tipi di dati, possono essere dichiarate variabili di tipo `bool`.

5.1 Costanti letterali booleane

C# esprime i due possibili valori che appartengono al tipo `bool` mediante le costanti: **`true`** (vero) e **`false`** (falso).

5.2 Funzione del tipo bool nella programmazione

Il tipo `bool` è spesso usato implicitamente, poiché qualsiasi costrutto di controllo fa uso di espressioni booleane. D'altra parte, in molti problemi è appropriato l'uso di variabili `bool`, sia come supporto alla programmazione sia per la rappresentazione di dati che, per loro natura, possono assumere due valori soltanto.

Segue un esempio d'uso di una variabile `bool` come supporto alla programmazione. Il seguente programma svolge il compito di riprodurre sullo schermo le parole digitate dall'utente, fin quando questo non digita la parola `"fine"`.

```
static void Main()
{
    bool fineCiclo;
    string parola;
    Console.WriteLine("Inserire la parola 'fine' per terminare");
    fineCiclo = false
    while (fineCiclo == false)
```

```
    {
        parola = Console.ReadLine();
        if (parola == "fine")
            fineCiclo = true
        else
            Console.WriteLine("La parola inserita è: {0}", parola);
    }
}
```

La variabile `fineCiclo` svolge la funzione di controllo del ciclo `while()`. Finché il suo valore è `false`, il programma continua a richiedere all'utente di inserire parole da tastiera. Quando l'utente digita la parola `"fine"`, la variabile viene impostata a `true`, quindi la successiva valutazione dell'espressione booleana `fineCiclo == false` produce come risultato `false`; il ciclo termina, e con esso il programma.

Nota bene: il precedente problema può essere risolto in modo più efficiente facendo uso della sola variabile `parola` e senza l'ausilio del costrutto `if() else`.

6 Variabili

Le variabili sono gli oggetti che contengono i dati elaborati dal programma. La maggior parte delle istruzioni di un programma consiste nell'assegnare il valore di un'espressione a una variabile. Ad ogni variabile viene riservato uno spazio di memoria della dimensione appropriata in base al suo tipo.

6.1 Dichiarazione di una variabile

L'istruzione di dichiarazione definisce il nome e il tipo di una variabile, oltre a determinare anche la sua creazione in memoria[5]. Assume la seguente forma:

tipo nome-var$_1$, nome-var$_2$, ... nome-var$_n$;

Ad esempio, le dichiarazioni:

```
int a;
double x;
```

determinano la creazione in memoria di due oggetti, identificati dai nomi `a` e `x`:

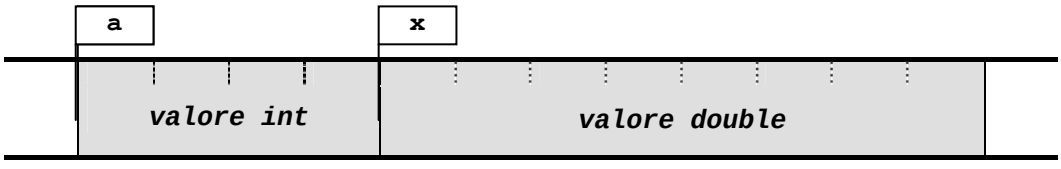


Figura 3-2 Rappresentazione in memoria delle variabili a e x.

Qualsiasi riferimento ad `a` e `x` in una espressione consiste nel riferimento al loro valore, che ovviamente dipende da cosa c'è memorizzato nella zona di memoria ad esse riservata.

[5] Ciò è vero soltanto per le variabili appartenenti alla classe dei tipi valore. Più avanti nel testo, contestualmente all'introduzione delle variabili array, sarà sottolineato come non tutti gli oggetti vengono allocati in memoria già durante la dichiarazione.

Come stabilito dalla sintassi, all'interno della stessa istruzione possono essere dichiarate più variabili; in questo caso, esse appartengono al medesimo tipo. Ad esempio, l'istruzione:

```
int a, b, c;
```

determina la creazione in memoria di tre oggetti di tipo `int`:

Il linguaggio non impone requisiti sull'ordine delle dichiarazioni o sulla loro collocazione all'interno del codice sorgente, a parte uno:

La dichiarazione di una variabile deve precedere il suo uso.

Dunque, il seguente codice è scorretto:

```
int a;
a = 1;
b = a;        // errore: b non è stata ancora dichiarata!
int b;
```

Mentre il seguente codice è corretto:

```
int a;
a = 1;
int b;
b = a;
```

Un secondo requisito riguarda l'impossibilità di dichiarare due volte la stessa variabile, o comunque due variabili di tipo diverso ma con lo stesso nome.

Dunque, è un errore scrivere:

```
int a, b;
double x, y;
string a;     // errore: a è già stata dichiarata!
```

Il tipo delle variabili è in questo caso irrilevante, ciò che ha importanza è il nome.

I precedenti esempi utilizzano spesso dichiarazioni di più variabili nella stessa istruzione. E' una pratica formalmente corretta, ma sconsigliata, poiché diminuisce la leggibilità del codice. E' stata più volte utilizzata nel testo soltanto allo scopo di ottenere un codice più compatto.

6.2 Assegnazione di un valore ad una variabile

Il modo più comune per modificare il valore di una variabile è mediante un'istruzione di assegnazione. Questa assume sempre la forma:

variabile = *espressione*;

e viene eseguita nel seguente modo:

1) viene innanzitutto valutata *espressione* e cioè calcolato il suo valore. Durante questa fase vengono eseguite tutte le conversioni che sono necessarie in modo che ci sia identità di tipo tra operandi e operatori;

2) il valore prodotto viene posto nella zona di memoria riservata a *variabile*; anche in questo caso se il tipo del valore non coincide con il tipo della variabile avviene una conversione, e cioè il valore viene ricodificato nel tipo della variabile prima di essere memorizzato in essa.

Ad esempio, si ipotizzi la seguente situazione iniziale:

Figura 3-3 Rappresentazione in memoria di una variabile int e una variabile double.

L'assegnazione:

```
x = a + 10;
```

produce le seguenti operazioni:

1) viene valutata l'espressione che sta a destra dell'operatore di assegnazione. Questa equivale al valore di a più la costante 10.
2) poiché il valore ottenuto è di tipo `int`, viene promosso al tipo `double`, e cioè il tipo di `x`;
3) il valore viene memorizzato nella zona di memoria riservata ad `x`.

Il risultato finale è mostrato in figura:

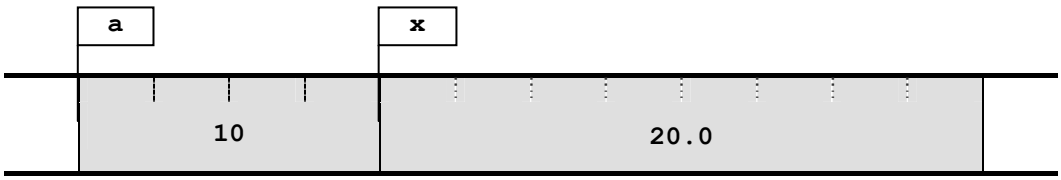


Figura 3-4 Rappresentazione delle variabili a e x dopo l'assegnazione.

Compatibilità tra espressione e variabile nell'assegnazione

In un'istruzione di assegnazione, il tipo dell'espressione deve essere compatibile con il tipo della variabile. Il termine compatibile è specificato dalle seguenti regole:

Il tipo <TE> di un'espressione si dice compatibile con il tipo <TV> di una variabile quando:

1) **<TE> e <TV> sono lo stesso tipo oppure, se sono diversi,**
2) **<TE> è convertibile implicitamente nel tipo <TV>.**

E' importante ricordare che è ammessa una conversione implicita da un tipo ad un altro quando esiste la garanzia che non vi possa essere alcuna perdita di informazione. E' dunque corretto scrivere:

```
int a = 10;
double x = a;     // ok: a è implicitamente convertibile nel tipo double
```

Mentre è sbagliato scrivere:

```
double x = 10;
int a = x;     // errore: x non è implicitamente convertibile nel tipo int
a = "100";     // errore: "100" non è implicitamente convertibile nel tipo int
```

6.3 Assegnazione composta

L'operatore = viene chiamato operatore di **assegnazione semplice**; il linguaggio mette infatti a disposizione un insieme di operatori chiamati di **assegnazione composta**, poiché combinano due operazioni in una: l'assegnazione più un'operazione che dipende dal tipo di operatore.

Un'assegnazione composta assume la sintassi:

variabile op= espressione;

che viene tradotta in:

variabile = variabile *op* (*espressione)*;

dove *op* indica il tipo di operazione da comporre con l'assegnazione.

Ad esempio, il seguente codice:

```
int i;
double x;
x = 10;
i += 10;
x *= i;
x /= i + 10;
```

equivale a:

```
int i;
double x;
x = 10;
i = i + 10;
x = x * i;
x = x / (i + 10)
```

ma ha il vantaggio di utilizzare una sintassi più compatta e una maggiore efficienza.

E' importante comprendere che un operatore di assegnazione composta rappresenta sintatticamente un singolo operatore e non l'unione di due operatori. Per questo motivo non vi devono essere spazi tra i due simboli. E' dunque sbagliato scrivere:

```
i + = 10;      // errore: tra + e = c'è uno spazio!
```

6.4 Valore iniziale di una variabile

Il **valore iniziale** di una variabile è il valore che essa contiene dopo la dichiarazione ma prima che abbia subito un'assegnazione. Ebbene, per le variabili dichiarate in un blocco (attualmente le uniche che stiamo considerando), questo è del tutto casuale. Si dice infatti che la variabile si trova in uno **stato iniziale indefinito**, detto anche **non assegnato**. Per contro, una variabile che abbia subito almeno una assegnazione si trova in uno stato **definito**.

Il concetto di stato iniziale di una variabile è molto importante poiché il linguaggio proibisce di utilizzare una variabile con stato indefinito:

```
double x, y, z;
z = 10;
x = z;         // ok: z aveva già un valore, era cioè definita
x = y;         // errore: lo stato iniziale di y è indefinito!
```

L'istruzione `x = y;` benché appaia corretta, viene segnalata come errata, poiché anche senza eseguire il programma è chiaro che a `z` non è stato assegnato alcun valore e dunque il suo stato non

può che essere indefinito. Poiché non avrebbe senso assegnare ad x un valore casuale; C# lo proibisce.

In conclusione, la dichiarazione di una variabile determina semplicemente l'allocazione dello spazio in memoria, ma nessun'altra operazione. Prima che vi sia stata almeno un'assegnazione alla variabile, il contenuto di tale spazio è casuale e pertanto la variabile non può essere usata in una espressione.

6.5 Inizializzazione di una variabile

Quello delle variabili non definite è un problema comune della programmazione, per questo motivo C# dà al programmatore la possibilità di specificare il valore iniziale di una variabile già durante la sua dichiarazione. In questo caso la dichiarazione assume la forma:

tipo nome-variabile = valore-iniziale;

dove *valore-iniziale* assume anche il nome di **inizializzatore** della variabile.

Ecco alcuni esempi di dichiarazione con inizializzazione:

```
double x = 0, y = 0;
int a = 1
bool flag = false;
string nome = "paperino";
```

L'inizializzazione di una variabile garantisce che il suo stato iniziale sia definito, ma non è obbligatoria, né sempre desiderabile; essa dipende infatti dall'uso che viene fatto della variabile. Si riconsideri l'Esempio 2.2, il quale calcola la somma dei numeri naturali che vanno da 1 a un numero dato.

```
static void Main()
{
    int somma  = 0,
    int n = 0,    // inizializzazione inutile.
    int x;
    string tmp;
    Console.WriteLine("Inserire il numero N ");
    tmp = Console.ReadLine();
    n = Convert.ToInt32(tmp);
// ----------------------- <n è fornito>
    for (x = 1; x <= n; x = x + 1)
        somma = somma + x;
    Console.WriteLine("La somma è: {0}", somma);
}
```

Sia n che somma sono state inizializzate a zero, ma soltanto l'inizializzazione di somma è appropriata. La variabile n, infatti, acquisisce il proprio valore dall'utente e dunque non ha alcun bisogno di essere inizializzata. Farlo ugualmente non rappresenta un errore, ma introduce un'operazione superflua nel programma e ne diminuisce la comprensibilità, poiché induce a pensare che n necessiti di essere inizializzata quando in realtà non è affatto vero.

La variabile somma, d'altro canto, svolge il ruolo di **accumulatore**; per definizione, un accumulatore deve avere un valore iniziale, che nel caso di una sommatoria è ovviamente zero. L'inizializzazione di somma, dunque, garantisce che il suo stato iniziale sia definito e appropriato.

L'inizializzatore di una variabile non dev'essere per forza un valore costante; infatti, l'unico requisito imposto dal linguaggio è che:

quando l'istruzione di dichiarazione viene eseguita, l'inizializzatore dev'essere a sua volta definito.

Ovviamente, una costante è definita per sua natura, ma può esserlo anche una variabile. Ad esempio, nel seguente codice la variabile `a` funge da inizializzatore per la variabile `b`:

```
int a = 10;
int b = a;        // ok: a è definita
```

Mentre è sbagliato scrivere:

```
int a;
int b = a;        // errore: a non è definita!
```

7 Costanti simboliche

C# offre la possibilità di associare un identificatore a una costante letterale per facilitare la leggibilità del programma; tale identificatore assume il nome di **costante simbolica**.

La dichiarazione di una costante simbolica assume la seguente sintassi:

```
const  tipo  nome-costante = valore-costante;
```

Ad esempio:

```
const int maxNumeroAlunni = 30;
const double rapportoMedio = 2.2;
const string password = "matrix";
```

Come si vede, a parte l'uso della parola chiave `const`, la dichiarazione di una costante simbolica è praticamente identica a quella di una variabile inizializzata. Resta il fatto che la prima è una costante e dunque è formalmente sbagliato scrivere:

```
const int maxAltezza = 10;
maxAltezza = 20;  // errore: il valore di un costante non può essere modificato!
```

Inoltre l'espressione associata a una costante simbolica dev'essere a sua volta una costante (letterale o simbolica non ha importanza). Dunque, il seguente codice è formalmente sbagliato:

```
int a = 20;
int b = a;    // ok: è corretto inizializzare b con a, poiché a è definita
const int maxAltezza = b;     // errore!
```

Il fatto che `b` sia già definita è irrilevante: non può essere usata una espressione contenente variabili per stabilire il valore di una costante simbolica.

7.1 Uso delle costanti simboliche

Le costanti favoriscono la realizzazione di programmi più comprensibili e facili da modificare. Si dovrebbe fare uso di costanti simboliche ogni qual volta che:

1) una costante letterale assume un preciso significato nella logica del programma;

2) una particolare costante letterale compare più volte nel codice sorgente, sempre con lo stesso significato.

Come esempio si consideri il seguente problema:

Data la massa di due corpi e la loro distanza, calcolare la forza reciproca di attrazione gravitazionale.

La soluzione consiste nell'applicare la legge di gravitazione universale:

$$F = 6{,}673 \cdot 10^{-11} \cdot \frac{M_1 \cdot M_2}{r^2}$$

dove M_1 e M_2 sono la masse dei due corpi ed r la distanza tra i medesimi; infine, la costante $6{,}673*10^{-11}$ è la costante di gravitazione universale, denotata convenzionalmente dalla lettera G.

Ecco l'implementazione del programma (alcuni dettagli sono stati omessi):

```
static void Main()
{
    const double G = 6.673E11  // costante di gravitazione universale
    double m1, m2, r, f;
    string tmp;

// ... qui m1, m2, r ricevono i loro valori dall'utente

    f = G * (m1 * m2) / (r * r);
    Console.WriteLine("La forza di attrazione è: {0}", f);
}
```

L'uso del nome G al posto del corrispondente valore letterale rende il programma più comprensibile, poiché mentre G (nell'ambito della fisica) è un simbolo convenzionale e dotato di un preciso significato, il valore 6.673E11 è un numero al quale è difficile dare immediatamente un significato.

L'uso di costanti simboliche ha senso quando il corrispondente valore letterale possiede di per sé un significato; ciò non significa dunque che ogni valore letterale debba essere sostituito da una costante simbolica.

Ad esempio, sarebbe inutile (e anche stupido) scrivere qualcosa del tipo:

```
const double unMezzo = 0.5;
double x;
double y = 20;
x = y * unMezzo;
```

L'identificatore unMezzo non è certo più esplicativo di 0.5.

E' invece appropriato scrivere qualcosa del tipo:

```
const double gravitaTerra = 9.81;              // acc. gravitazionale terrestre;
const double gravitaLuna = gravitaTerra / 6;   // acc. gravitazionale lunare
double massa, pesoNellaLuna, pesoNellaTerra;

// ... qui la variabile massa riceve il proprio valore dall'utente

pesoNellaTerra = massa * gravitaTerra;
pesoNellaLuna = massa * gravitaLuna;
```

8 Espressioni

Un'espressione è rappresentata da:

una combinazione di uno o più valori (variabili e/o costanti e/o altre espressioni) e di zero o più operatori che operano con essi. Un'espressione denota un valore, il quale appartiene a un determinato tipo, chiamato tipo dell'espressione.

Data questa definizione, se ne conclude che anche una variabile o una costante designano, da sole, un'espressione. Ecco alcuni esempi:

```
int a, b;
a = 10;       // 10 è un'espressione composta da una sola costante
b = a;        // a è una espressione composta da una sola variabile
b = a + 10    // a + 10 è un'espressione rappresentata dal valore prodotto
              // dall'operatore + sugli operandi a e 10
```

8.1 Tipo di un'espressione

Il risultato di un'espressione appartiene sempre ad un determinato tipo, il quale dipende:

unicamente dai tipi dei valori che vi compaiono e dai tipi degli operatori che operano con essi.

8.2 Operatori

Un operatore è un simbolo che designa un'operazione prodotta su uno o più argomenti, detti operandi. Tale operazione produce sempre un risultato di un determinato tipo.

Per chiarire le due regole appena introdotte, consideriamo il seguente frammento di codice allo scopo di determinare il tipo delle espressioni che vi compaiono.:

```
double x, y;
int a, b;
// qui a e b ricevono dei valori
x = a / b * 2;
y = a / b * x;
```

Le espressioni presenti nelle ultime due istruzioni sono scomponibili come segue:

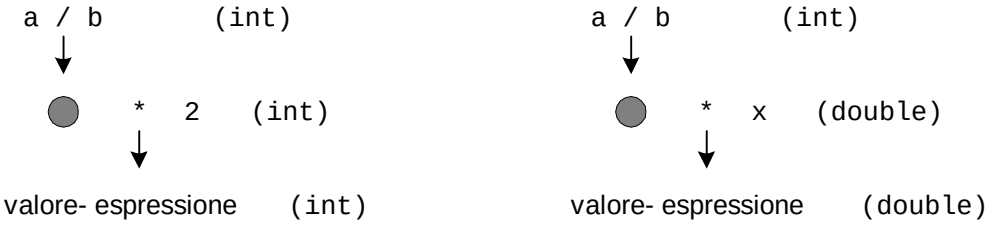


Figura 3-5 Esempio di scomposizione di una espressione nelle singole operazioni svolte.

Accanto ad ogni op erazione è specificato il tipo del valore risultante; esso può essere dedotto dalle regole del linguaggio sugli operatori. In tutto ciò non hanno importanza i valori di `a`, `b` e `x`, poiché il tipo dell'operatore invocato, e dunque il tipo del risultato prodotto, dipende unicamente dal tipo degli operandi e non dai loro valori.

Le regole del linguaggio sui tipi degli operatori e sul tipo delle espressioni in generale portano a conseguenze interessanti e importanti da comprendere, come mostra il seguente codice:

```
double x
int a, b, c;
a = 10;
x = 2;
b = a * 2;     // ok
c = a * x;     // impossibile assegnare un valore double a una variabile int!
```

Le due espressioni, `a * 2` e `a * x`, producono lo stesso valore, e cioè 20, ma questo è di tipo `int` per la prima espressione e di tipo `double` per la seconda. Per la regola che non si può assegnare un valore di tipo `double` a una variabile `int`, l'ultima istruzione è errata.

8.3 Precedenza, associatività e "arità" degli operatori

"Arità" di un operatore

L'arità di un operatore indica il numero di argomenti di cui necessita. Un operatore può essere:

- unario (arità uno); necessita di un solo argomento;
- binario (arità due); necessita di due argomenti;
- ternario (arità tre); necessita di tre argomenti.

Precedenza degli operatori

La regola di precedenza degli operatori indica, in presenza di un'espressione con più operazioni, l'ordine di esecuzione delle stesse. Consideriamo il seguente codice:

```
int a =10;
int b = 20;
int c;
c = a + b * 10;       // viene eseguito a + (b * 10)
```

L'operatore di moltiplicazione ha la precedenza sull'operatore di somma e dunque, in assenza di parentesi, viene eseguito per primo. Tutti gli operatori del linguaggio sono organizzati secondo un preciso ordine di precedenza.

Associatività di un operatore

L'associatività riguarda soltanto gli operatori binari o ternari e indica quale operando viene valutato per primo; può essere:

- associatività a sinistra: l'operazione viene valutata da sinistra a destra;
- associatività a destra: l'operazione viene valutata da destra a sinistra.

Ad esempio, nell'istruzione:

```
int a = 42 - 10 - 6;
```

l'associatività a sinistra dell'operatore di sottrazione determina la traduzione dell'espressione in:

```
(42 - 10) - 6
```

che produce 26 come risultato finale. Se l'operatore di sottrazione fosse associativo a destra, l'espressione sarebbe tradotta in:

```
42 – (10 – 6)
```

e produrrebbe 38 come risultato finale.

(L'argomento è trattato in modo approfondito nell'Appendice A)

8.4 Uso delle espressioni

Data la definizione precedentemente data di espressione ne deriva che:

all'interno di un'istruzione può essere usata un'espressione di tipo <T> ovunque le regole del linguaggio ammettano l'uso di un valore dello stesso tipo.

Ciò significa, per esempio, che ovunque sia ammesso un numero intero è ammessa anche un'espressione di tipo `int`, arbitrariamente complessa. Ecco un esempio dimostrativo:

```
int a, b, c, d;
// qui b, c ricevono i loro valori
a = Math.Sqrt(b);    // ok, uso banale del metodo Math
a = Math.Sqrt(b * c / 100 * d + b / c – 100 * c);          // sempre ok!
```

Il linguaggio impone un requisito particolare sull'uso di un'espressione e cioè che il valore da essa prodotto sia effettivamente usato in qualche modo! La valutazione di un'espressione deve cioè produrre un qualche effetto sull'esecuzione del programma.

Ad esempio, il seguente codice, benché formalmente corretto (ma logicamente stupido) provoca un errore di compilazione:

```
int a = 10;
int b = 20;
a + b;     // errore: cosa viene fatto del valore di a + b?
```

L'istruzione `a + b` implica la valutazione di un'espressione, ma non l'uso del valore calcolato, e ciò rappresenta un errore formale.

9 Espressioni booleane, o codizionali

La definizione di espressione data in precedenza è perfettamente valida anche per le espressioni booleane, le quali sono le uniche che possono apparire tra le parentesi dei costrutti di controllo; infatti, solo le espressione booleane sono in grado di esprimere una condizione, il cui valore può essere vero (`true`) o falso (`false`)[6].

9.1 Espressioni booleane semplici e composte

Negli esempi forniti finora sono state considerate espressioni booleane semplici, espressioni, cioè, che esprimono una condizione nella forma:

$$espressione_1 \quad operatore\text{-}relazionale \quad espressione_2$$

laddove i valori delle due espressioni vengono confrontati tra loro mediante uno degli operatori relazionali, elencati nella tabella a pagina successiva.

Tabella 3-2 Elenco Operatori relazionali.

[6] Altre espressioni possono comparire ma solo come parte di una espressione booleana.

OPERATORE	CONFRONTO EFFETTUATO
==	UGUALE
!=	DIVERSO
>	MAGGIORE
<	MINORE
>=	MAGGIORE O UGUALE
<=	MINORE O UGUALE

Espressioni semplici sono, ad esempio:

```
a > 10        (a + b) == (d + a)        Math.Sqrt(a) <= 2
```

Ma come un'espressione qualsiasi può essere composta da un numero arbitrario di sotto espressioni, così un'espressione booleana composta rappresenta la combinazione di più espressioni booleane semplici, connesse dagli **operatori logici condizionali**, o **connettivi**.

Segue l'elenco degli operatori logici condizionali ed il risultato che producono:

Tabella 3-3 Elenco operatori logici condizionali.

OPERATORE LOGICO	CONNETTIVO CORRISPONDENTE	RISULTATO
&&	e	`true` se entrambi gli operandi sono `true`, `false` altrimenti
||	o	`true` se uno o entrambi gli operandi sono `true`, `false` altrimenti
!	NON (negazione)	`true` se l'operando è `false`, `false` se l'operando è `true`

Un'espressione booleana composta consente la combinazione di più condizioni nella stessa espressione allo scopo di ottenere un codice più compatto ed efficiente. Essa assume la forma:

condizione$_1$ *operatore-logico* condizione$_2$

Ad esempio, si ipotizzi di voler stabilire se un numero intero X inserito dall'utente cade all'interno di un intervallo A e B. Ecco una possibile soluzione:

```
static void Main()
{
    double a, b, x;
    string tmp;
    Console.WriteLine("Inserire gli estremi 'A' e 'B' dell'intervallo");
    tmp = Console.ReadLine();
    a = Convert.ToDouble(tmp);
    tmp = Console.ReadLine();
    b = Convert.ToDouble(tmp);
    Console.WriteLine("Inserire il valore da verificare");
    tmp = Console.ReadLine();
```

```
    x = Convert.ToDouble(tmp);
    // -----------  <a, b, x sono forniti>
    if (x >=a && x <= b)
        Console.WriteLine("Il numero {0} cade nell'intervallo {1} – {2}", x, a,
                          b);
    else
        Console.WriteLine("Il numero {0} non cade nell'intervallo {1} – {2}", x,
                          a, b);
}
```

La verifica della condizione *x-cade-nell'intervallo* viene effettuata attraverso l'espressione booleana composta:

```
x >=a && x <= b
```

Questa si legge:

`x` è maggiore o uguale al limite inferiore `a` "e" `x` è minore o uguale del limite superiore `b`.

9.2 Valutazione delle espressioni booleane composte

Nel determinare il risultato di un'espressione booleana composta viene applicata la cosiddetta forma di **valutazione breve**. Per comprendere di cosa si tratta si consideri l'espressione booleana:

```
a > b && a > c
```

il cui risultato è `true` se `a` risulta maggiore sia di `b` che di `c`.

L'espressione viene valutata nel seguente modo:

Prima viene valutata la sotto espressione `a > b`:

1) se risulta `false`, per la presenza del connettivo `&&` il valore dell'espressione composta è sicuramente `false` e dunque non c'è bisogno di valutare `a > c`, che infatti non viene valutata;

2) se risulta `true`, allora viene valutata anche l'espressione `a > c`, la quale determina il valore finale dell'espressione composta;

In sostanza, il valore di una sotto espressione viene calcolato soltanto se il risultato finale non è già stato determinato dal valore delle sotto espressioni calcolate in precedenza.

Ordine di composizione delle espressioni semplici

Questa forma di valutazione delle espressioni composte produce in alcuni casi delle conseguenze importanti, determinate dall'ordine in cui le espressioni semplici vengono specificate. Ad esempio, si consideri il seguente frammento di codice, che verifica se una variabile di nome `dividendo` sia divisibile da una variabile di nome `divisiore`:

```
int dividendo, divisore
...
if (dividendo % divisore == 0 && divisore != 0)
    Console.WriteLine("{0} è divisibile per {1}", dividendo, divisore);
```

L'espressione booleana dentro la `if()` verifica che:

1) la variabile `dividendo` sia multipla della variabile `divisore`, (resto della divisione tra `dividendo` e `divisore` uguale a zero), e che

2) la divisione sia effettivamente eseguibile (variabile `divisore` diversa da zero).

L'implementazione è scorretta, infatti prima è necessario verificare che `divisore` sia diversa da zero, e dopo, soltanto se ciò è vero, è corretto eseguire la divisione e verificare che il suo resto sia zero. Dunque la condizione deve essere riscritta nel seguente modo:

```
if (divisore != 0 && dividendo % divisore == 0)
    Console.WriteLine("Dividendo = {0}", dividendo);
```

Con questa impostazione, la condizione:

```
dividendo % divisore == 0
```

viene calcolata solo se la condizione:

```
divisore != 0
```

è vera (e quindi la divisione è effettivamente possibile).

Espressioni booleane composte in cui la possibilità di una reale valutazione di una sotto espressione dipende dal risultato dell'altra (o delle altre) non sono così rare. In questo caso è necessario scrivere le sotto espressioni secondo l'ordine appropriato.

9.3 Assegnazione di espressioni booleane

Comunemente, un'espressione booleana compare come condizione in un costrutto di controllo con lo scopo di determinare l'andamento del flusso di esecuzione, ma non è l'unico uso possibile. Come qualsiasi altra espressione, anche un'espressione booleana può essere assegnata a una variabile, ovviamente dello stesso tipo. Ad esempio:

```
double x, y;
    // ... x e y ricevono dei valori
bool b = (x > y);
```

la variabile booleana `b` riceve il valore dell'espressione booleana `x > y`, per l'occasione evidenziata in neretto.

Nel precedente codice, l'uso di parentesi ha il solo scopo di aumentare la comprensibilità del codice e non rappresenta affatto un requisito formale. Nel costrutti di controllo, le condizioni sono tra parentesi perché queste sono richieste dai costrutti e non perché siano parte delle espressioni booleane.

Quest'uso un po' inconsueto delle espressioni booelane possiede la propria utilità, infatti capita spesso di dover impostare una variabile bool in base al risultato di un'espressione booleana, come ad esempio nel codice che segue:

```
bool autenticato;
string password;
password = Console.ReadLine();
if (password == "matrix")
    autenticato = true;
else
    autenticato = false;
```

In sostanza, il codice assegna alla variabile `autenticato` il valore della condizione usata nella `if()`. E allora tanto vale farlo direttamente, scrivendo:

```
bool autenticato = (password == "matrix");
```

10 Espressioni di tipo string

Anche sulle stringhe, benché non rappresentino dei valori nel senso che diamo di solito a questo, termine, possono essere eseguite delle operazioni.

10.1 Confrontare due stringhe

Due stringhe possono essere confrontate tra loro mediante gli operatori relazionali `==` e `!=`. In questo caso, due stringhe sono considerate uguali soltanto se hanno la stessa lunghezza e tutti i caratteri corrispondenti sono uguali. Ad esempio:

- "amore" e "amorevole" sono stringhe diverse, poiché la seconda contiene dei caratteri che non sono presenti nella prima;
- "Amore" e "amore" sono stringhe diverse, poiché viene fatta distinzione tra maiuscole e minuscole; (si dice che il confronto è **case sensitive**, e cioè sensibile al **case** delle lettere).

> Sia il tipo `string` che altre classi definisco dei metodi di confronto che offrono un maggior controllo al programmatore, come ad esempio quello di poter stabilire se considerare o meno la differenza tra maiuscole e minuscole durante l'operazione di confronto.

10.2 Concatenare stringhe

Due stringhe possono essere concatenate mediante l'operatore di **concatenazione**, rappresentato dal simbolo '+'. Il risultato è rappresentato da una terza stringa contenente i caratteri della seconda attaccati a quelli della prima. Naturalmente, il prodotto della concatenazione di due stringhe può essere a sua volta concatenato con un'altra stringa, e così via.

Come esempio, si consideri il seguente codice:

```
string risultato, s1, s2, s3;
s1 = "Stanlio";
s2 = " & ";
s3 = "Ollio";
risultato = s1 + s2 + s3;
```

`s1 + s2 + s3` rappresenta un'espressione di tipo `string`. Il valore di questa espressione è la stringa risultante dalla concatenazione delle variabili `s1`, `s2` e `s3` nell'ordine in cui compaiono nell'espressione, e cioè:

```
"Stanlio & Ollio"
```

E' importante comprendere che l'operatore '+' applicato alle stringhe rappresenta sempre l'operazione di concatenazione e non di somma. Quindi, ad esempio, il seguente codice:

```
string risultato, s1, s2;
s1 = "100";
s2 = "300";
risultato = s1 + s2;
Console.WriteLine(risultato);
```

produce sullo schermo:

```
100300
```

e non:

```
400
```

Un altro aspetto su cui fare attenzione è che nell'operazione di concatenazione gli spazi vengono trattati né più né meno come gli altri caratteri. Ad esempio, il seguente codice:

```
string risultato, s1, s2;
s1 = "ciao        ";
s2 = "mamma";
risultato = s1 + s2;
Console.WriteLine(risultato);
```

produce sullo schermo:

```
ciao        mamma
```

e non:

```
ciaomamma
```

11 Conversioni da tipi numerici a tipo string

Poiché le stringhe rivestono una grande importanza nella rappresentazione e nella elaborazione dei dati di un programma, C# consente la conversione di valori numerici in valori stringa corrispondenti.

11.1 Conversione automatica da tipo numero a stringa

Esiste una situazione particolare nella quale il linguaggio effettua una conversione automatica di un valore numerico nella sua rappresentazione in forma di stringa. Ad esempio, il linguaggio ammette il seguente codice:

```
string s;
double altezza = 1.83;
s = "Altezza in metri = " + altezza;  // sembra errato, ma invece va bene!
Console.WriteLine(s);
```

L'espressione

```
Altezza in metri = " + altezza
```

appare formalmente sbagliata, poiché è una somma tra una stringa e una variabile `double`, e cioè un'operazione priva di senso. Ma l'espressione non designa affatto una somma ma un concatenazione di stringhe! Nell'interpretare l'espressione, C# applica la seguente regola:

quando uno dei due operandi dell'operatore + è una stringa, l'altro operando viene automaticamente convertito in stringa; successivamente viene invocato l'operatore di concatenazione.

Dunque, l'espressione si trasforma in:

```
"Altezza in metri = " + "1,83"
```

E' di fondamentale importanza non confondere mai la concatenazione con la somma, e le stringhe che rappresentano numeri con valori numerici. Ad esempio, consideriamo il seguente codice:

```
string s = "1000";
int peso = 2000;
string risultato = s + peso
Console.WriteLine("Peso in chilogrammi = {0}", risultato);
```

Non ci si aspetti di vedere sullo schermo:

```
Peso in chilogrammi = 3000
```

L'espressione `peso + s` è un'espressione stringa, e l'operatore '+' realmente applicato è quello di concatenazione e non di somma. Sullo schermo apparirà:

```
Peso in chilogrammi = 20001000
```

11.2 Conversione di valori in stringa mediante il metodo ToString()

L'operatore di concatenazione è l'unico che ammette una conversione automatica tra valori numerici e stringa; negli altri casi è necessario invocare esplicitamente il metodo `ToString()`, appartenente alla classe `Convert`. Ad esempio, per assegnare una espressione numerica ad una variabile stringa si può scrivere:

```
double x = 10;
string str = Convert.ToString(x);
```

Sarebbe invece scorretto scrivere:

```
double x = 10;
string str = x;      // errore: non esiste conversione da double a string!
```

4

Array

1 Collezioni di dati

La possibilità di gestire collezioni di dati è indispensabile per la realizzazione di programmi realistici. Nel mondo reale, infatti, i dati si presentano spesso in forma di insiemi, di collezioni appunto. Si pensi alle paghe degli impiegati di un'azienda, all'elenco dei titoli dei libri di una biblioteca, a un indirizzario, eccetera. Ognuno di questi esempi è caratterizzato dal fatto che:

1) esiste un tipo di dato da gestire: una paga, un titolo, un indirizzo;
2) esiste un numero arbitrario di valori di questo tipo;
3) il programma deve poter memorizzare tutti i valori e poter accedere indifferentemente a ognuno di essi.

Gli elementi del linguaggio studiati finora non consentono di gestire collezioni di dati; infatti, l'unica forma di memorizzazione studiata è quella della variabile semplice, e cioè un oggetto che consente di elaborare un solo valore per volta. Si rendono dunque indispensabili delle variabili che siano in grado di accogliere più valori contemporaneamente e che consentano l'accesso a ognuno di essi. Con il termine collezione ci si riferisce appunto a tale classe di variabili, definite anche **variabili strutturate** perché in possesso di una struttura in grado di memorizzare più valori contemporaneamente.

Una collezione è caratterizzata da:

1) il tipo di dati che la compongono;
2) dalla sua organizzazione interna;
3) dalla modalità di accesso ai singoli dati;
4) dalla possibilità di poter aggiungere o togliere dati.

Dei molti tipi di collezioni che mette a disposizione C#, qui sarà analizzato il tipo più semplice e cioè l'array.

1.1 Array

Un array rappresenta una collezione:

1) di **dati omogenei**, e cioè tutti dello stesso tipo;
2) **statica**: dopo che è stata creata, e cioè allocata in memoria, non è più possibile aggiungere o togliere elementi
3) che consente l'accesso ai singoli valori della collezione attraverso degli **indici** di tipo `int`.

Il tipo array si può presentare attraverso diverse forme di organizzazione dei dati; esistono cioè:

- **array unidimensionali**: i dati sono organizzati come in un elenco;
- **array multidimensionali**, i dati sono organizzati in forma di tabella (due dimensioni) cubo (tre dimensioni), ipercubo (quattro dimensioni), eccetera. Di questa categoria saranno presi in considerazione soltanto gli array bidimensionali.

La differenza tra array di diverse dimensioni sta nell'organizzazione dei dati; i concetti spiegati nei paragrafi successivi, che fanno riferimento ad array unidimensionali, sono validi per qualsiasi tipo di array.

2 Array unidimensionali: vettori

Una variabile semplice può essere vista come un contenitore, etichettato da un nome, il cui contenuto rappresenta il valore della variabile. Analogamente, un array unidimensionale, chiamato anche **vettore**, può essere visto come una fila di contenitori adiacenti, numerati (etichettati) a partire da zero, ognuno dei quali contiene un singolo valore.

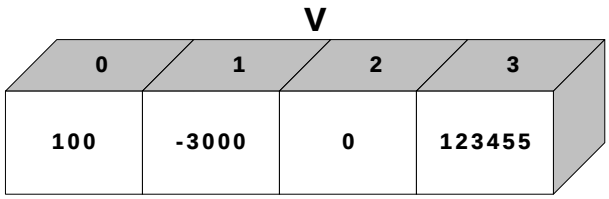


Figura 4-1 Rappresentazione schematica di un vettore.

I contenitori sono chiamati **elementi** del vettore, e ognuno di essi è sostanzialmente identico a una variabile semplice.

2.1 Accesso agli elementi di un vettore

Le operazioni consentite sugli elementi sono le stesse consentite su una variabile, e cioè la modifica del valore e l'utilizzo del medesimo in una espressione. La differenza tra gli elementi di un vettore e le variabili semplici risiede nella modalità di accesso agli stessi. Per riferirsi a un elemento di un vettore occorre infatti specificare sia il nome del vettore che la posizione, chiamata **indice**, dell'elemento. Ciò lo si ottiene con la seguente sintassi:

nome-vettore[*indice-elemento*]

che fa uso dell'operatore **accesso a elemento**, rappresentato dalla coppia di parentesi quadre.

Ad esempio, considerato il vettore di nome `v` rappresentato in figura, per assegnare il valore 10 all'elemento di indice zero si scrive:

```
v[0] = 10;
```

Oppure, per assegnare alla variabile a il valore contenuto nell'elemento di indice 1, si scrive:

```
a = v[1];
```

L'indice dell'elemento deve ssere di tipo `int`. Dunque è scorretto scrivere:

```
double x = 0;
v[x] = 10;
```

indipendentemente dal fatto che `x` contenga un numero con o senza la virgola. (Dal punto di vista formale, sono i tipi che hanno importanza e non i valori.)

E' importante comprendere che la modifica dei valori memorizzati in un vettore avviene sempre attraverso l'accesso ai suoi elementi. E' dunque scorretto scrivere qualcosa del tipo:

```
v = 1000;
```

poiché il nome del vettore, da solo, non fa riferimento a nessun elemento.

2.2 Dichiarazione e creazione di un vettore

La dichiarazione di una variabile semplice, oltre a definirne nome e tipo, determina anche l'allocazione in memoria dello spazio necessario in base al tipo della variabile. La creazione di un vettore, più in generale di un array, funziona in modo diverso, infatti:

la fase di dichiarazione della variabile è separata dalla fase di creazione vera e propria del vettore.

In altre parole, la dichiarazione di una variabile vettore si limita a definire il nome e il tipo degli elementi, ma non produce allocazione di memoria per gli stessi. Dichiarazione e creazion sono fasi concettualmente ed operativamente distinte.

Dichiarazione di un vettore

La dichiarazione di un vettore assume la seguente sintassi:

tipo`[]` *nome-vettore*`;`

Ad esempio:

```
int[] v;
```

La coppia di parentesi quadre indica che `v` si riferisce a un vettore di interi e non a una variabile semplice di tipo `int`. E' fondamentale comprendere che la precedente dichiarazione stabilisce semplicemente che `v` è una variabile array di tipo `int`, ma non crea lo spazio necessario per memorizzare gli elementi del vettore. Il seguente codice è infatti scorretto:

```
int[] v;
v[2] = 10;  //errore: l'elemento v[2] non esiste ancora!
```

poiché il vettore `v` è dichiarato ma non creato, e quindi è impossibile fare riferimento ai suoi elementi, in quanto non esistono ancora.

Creazione di un vettore

L'operazione di creazione di un vettore viene effettuata tramite l'operatore `new`. Assume la seguente sintassi:

nome-vettore `=` **`new`** *tipo*`[`*numero-elementi*`];`

Ad esempio:

```
v = new int[5];
```

La precedente istruzione produce l'allocazione in memoria di un vettore di 5 elementi, ognuno dei quali è di tipo `int`. Successivamente, mediante l'operatore di assegnazione, l'indirizzo di memoria del vettore viene memorizzato nella variabile `v`. L'espressione specificata tra parentesi quadre, che indica il numero di elementi del vettore, dev'essere di tipo `int`.

Nota bene, il numero degli elementi può essere stabilito anche mediante una variabile. Il seguente frammento di codice mostra come creare un vettore di stringhe la cui lunghezza è stabilita dall'utente:

```
int numStudenti;
string tmp;
string[] studenti;
Console.WriteLine("Inserisci il numero degli studenti:");
tmp = Console.ReadLine();
numStudenti = Convert.ToInt32(tmp);
studenti = new string[numStudenti];
...
```

2.3 Variabili e oggetti vettore

La variabile utilizzata per gestire un vettore ed il vettore stesso sono due oggetti distinti. Questa particolare caratteristica dei vettori (e degli array in generale) sarà affrontata in più avanti nel testo; per il momento ci limitiamo a sottolineare la fondamentale differenza tra le due categorie di variabili che abbiamo incontrato finora:

- **variabili valore**: sono le variabili semplici. Per esse la dichiarazione della variabile implica la sua immediata creazione.
- **variabili riferimento**, alla quale appartengono le variabili array. Per esse, la dichiarazione della variabile e la creazione dell'oggetto alla quale si riferisce rappresentano due fasi distinte, come distinti sono i due oggetti in memoria.

2.4 Valore iniziale degli elementi di un vettore

La creazione di un vettore si distingue dalla di creazione di una variabile di tipo `int` o `double` anche per un secondo fattore. Il valore iniziale degli elementi appena creati non è affatto casuale, ma è impostato al **valore predefinito** (detto anche **valore di default**) stabilito per il tipo del vettore.

Ogni tipo di dato definisce un proprio valore di default, che rappresenta il valore assegnato a una variabile di quel tipo nel caso venga inizializzata automaticamente dal linguaggio. (Finora, a parte appunto gli elementi di un array, non sono state considerate situazioni nelle quali le variabili vengono inizializzate automaticamente durante la dichiarazione.) Seguono i valori predefiniti per i tipi più comuni.

Tabella 4-1 Valori di default per i tipi di dati finora introdotti.

TIPO	VALORE DEFAULT
`int` o `double`	`0`
`string`	`""` (stringa vuota)
`bool`	`False`

In conclusione, gli elementi di un vettore non si troveranno mai nello stato iniziale indefinito.

2.5 Numerazione degli elementi di un vettore

Una caratteristica importante dei vettori, che può comportare alcuni errori di programmazione, è il sistema di numerazione usato per gli elementi, sistema che parte da zero e non da uno.

Ad esempio, la creazione di un vettore `int` di 5 elementi produce la seguente situazione:

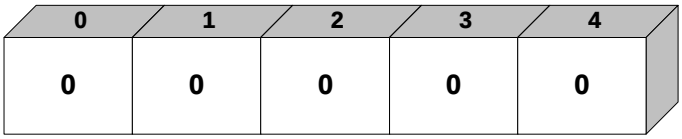


Figura 4-2 Rappresentazione di un vettore int di cinque elementi.

Il primo elemento ha indice 0, il secondo elemento ha indice 1, e così via, fino all'ultimo elemento, che ha indice 4. Dunque, un vettore di N elementi ha un **intervallo di variazione degli indici** che va da 0 a N -1.

Dimenticarsi la precedente regola è spesso fonte di errori, che possono essere sintetizzati dal seguente frammento di codice, nel quale si richiede all'utente di inserire tre parole, che vengono memorizzate in un vettore di tipo `string`.

```
string[] parole;
parole = new string[3];
for (i = 1; i <= 3; i = i + 1)   // inizializzazione e condizione errate!
    parole[i] = Console.ReadLine();
```

Ecco quale dovrebbe essere il contenuto del vettore dopo l'inserimento delle parole `"pippo"`, `"pluto"`, `"topolino"`:

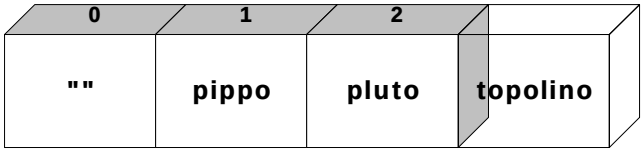


Figura 4-3 Ipotetico contenuto del vettore dopo l'inserimento delle parole.

Nel primo elemento non viene memorizzato niente, poiché il valore iniziale di `i` è 1, e cioè l'indice del secondo elemento. Il ciclo procede fintantoché `i` è minore o uguale 3. Dunque, nell'ultima iterazione, l'istruzione:

```
parole[i] = Console.ReadLine();
```

equivale a:

```
parole[3] = Console.ReadLine();
```

Ma l'elemento di indice 3 non esiste affatto!

In realtà il precedente codice produce un errore di esecuzione, poiché il linguaggio proibisce l'accesso ad un elemento con un indice che è al di fuori dell'intervallo di variazione degli indici.

2.6 Esempi d'uso di vettori

Gli array in generale (e i vettori in particolare) sono assolutamente indispensabili nella programmazione. Tutti i programmi realistici fanno ampio uso di questo tipo di struttura dati; ma esistono anche problemi apparentemente banali che pure richiedono espressamente l'uso di array.

Esempio n° 1

Si vuole consentire all'utente di digitare un numero arbitrario di parole; il programma dovrà successivamente riprodurre le parole sullo schermo, ma nell'ordine inverso.

Poiché il programma deve visualizzare le parole solo dopo che tutte sono state inserite, è necessario che queste siano memorizzate da qualche parte prima di essere successivamente visualizzate; è necessario collezionarle.

Ecco l'implementazione del programma:

```
static void Main()
{
    string tmp;
    int numParole, i;
    string[] parole;                    // dichiara il vettore
    Console.WriteLine("Inserire il numero di parole");
    tmp = Console.ReadLine();
    numParole = Convert.ToInt32(tmp);
    parole = new string[numParole];     // crea il vettore
    Console.WriteLine("Inserire le parole");

// ciclo iterativo che richiede le parole all'utente e le memorizza nel vettore
    for (i = 0; i < numParole; i = i + 1)
        parole[i] = Console.ReadLine();

// ciclo iterativo che visualizza le parole memorizzate nel vettore
    for (i = numParole-1; i >= 0; i = i - 1)
        Console.WriteLine(parole[i]);
}
```

Il programma funziona nel seguente modo:

1) viene richiesto all'utente il numero di parole che intende digitare;
2) viene creato un vettore di stringhe contenente un numero di elementi equivalente al numero di parole da memorizzare;
3) all'interno di un ciclo vengono richieste le parole all'utente, le quali sono memorizzate nel vettore;
4) mediante un secondo ciclo vengono visualizzate le parole in ordine inverso; ciò si ottiene facendo partire il contatore del ciclo da `numParole - 1` e decrementandolo fino a `0`.

Esempio n° 2

Si vuole consentire all'utente di digitare un numero dato di valori numerici; successivamente, in base alla scelta dell'utente, si vuole che il programma calcoli o la sommatoria o il prodotto cumulativo dei numeri inseriti.

Anche in questo caso la fase di inserimento dei dati e quella della loro elaborazione sono distinte; si rende dunque necessario che tutti i dati siano memorizzati prima che su di essi siano eseguite le operazioni richieste dal problema.

Ecco l'implementazione del programma:

```
static void Main()
{
    string tmp, tipoCalcolo;
    int numDati, i;
    double[] dati;
    double risultato;         // il risultato del calcolo scelto dall'utente
    Console.WriteLine("Inserire il numero di valori da elaborare");
    tmp = Console.ReadLine();
    numDati = Convert.ToInt32(tmp);
    dati = new double[numDati];
    Console.WriteLine("Inserire i valori");
// ciclo iterativo che richiede i dati all'utente e li memorizza nel vettore
    for (i = 0; i < numDati; i = i + 1)
    {
        tmp = Console.ReadLine();
        dati[i] = Convert.ToInt32(tmp);
    }
// chiede all'utente quale calcolo effettuare
    Console.WriteLine("Tipo di calcolo: 'sommatoria' o 'prodotto'?");
    tipoCalcolo = Console.ReadLine();
    if (tipoCalcolo == "sommatoria")      // l'utente ha scelto la sommatoria
    {
        risultato = 0;
        for (i = 0; i < numDati; i = i + 1)
            risultato = risultato + dati[i];
    }
    else          // l'utente ha scelto il prodotto cumulativo
    {
        risultato = 1;
        for (i = 0; i < numDati; i = i + 1)
            risultato = risultato * dati[i];
    }

    Console.WriteLine("Risultato finale: {0}", risultato);
}
```

Il programma funziona così:

1) viene richiesto all'utente il numero di valori da inserire;
2) viene creato un vettore di `double` contenente un numero di elementi equivalente al numero di valori da memorizzare;
3) all'interno di un ciclo vengono richiesti all'utente i valori numerici, che vengono memorizzati nel vettore dopo l'appropriata conversione da `string` a `double`;
4) viene chiesto all'utente quale tipo di calcolo effettuare;
5) se l'utente digita "sommatoria", mediante un ciclo viene calcolata la sommatoria;
6) altrimenti, sempre mediante un ciclo viene calcolato il prodotto cumulativo.

Considerazioni sugli esempi presentati

I programmi implementati, pur molto semplici, dimostrano ugualmente la potenza espressiva dei vettori. Essa è dovuta fondamentalmente a due caratteristiche degli array:

1) l'omogeneità degli elementi, e cioè il fatto che siano dello stesso tipo;
2) la possibilità, attraverso un indice, di accedere indifferentemente a qualsiasi elemento.

Queste, unite all'impiego dei cicli iterativi, consentono di elaborare tutti gli elementi del vettore in modo semplice e compatto.

2.7 Creare un vettore durante la dichiarazione

La dichiarazione di una variabile di tipo array rappresenta una fase distinta dalla creazione dell'array stesso; nulla impedisce, però, di riunire le due fasi all'interno della medesima istruzione. In questo caso la dichiarazione assume la sintassi:

```
tipo[] nome-vettore = new tipo[numero-elementi];
```

Ad esempio:

```
int[] v = new int[10];
```

questa istruzione dichiara la variabile array `v` e crea immediatamente il vettore corrispondente. Tra le due forme di dichiarazione-creazione, questa è preferibile, poiché più compatta.

2.8 Inizializzare gli elementi di un vettore

Così com'è possibile specificare il valore iniziale di una variabile semplice, allo stesso modo è possibile inizializzare gli elementi di un vettore durante la fase di creazione. L'inizializzazione può assumere due sintassi; una estesa, l'altra compatta.

Inizializzazione mediante la sintassi estesa.

Essa assume la forma:

$$nome\text{-}vettore = \textbf{new}\ tipo[num\text{-}elementi_{opz}]\ \{val_1, val_2 \ldots val_n\};$$

Ad esempio:

```
int[] v = new int[] {1,–20, 100, 22};
```

Questa istruzione produce la creazione di un vettore di 4 elementi i cui valori sono quelli specificati tra le parentesi graffe:

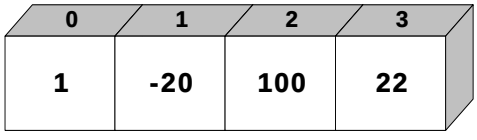


Figura 4-4 Rappresentazione del vettore dopo l'inizializzazione degli elementi

Nell'esempio precedente è stato omesso il numero degli elementi del vettore; in questo caso viene creato un vettore che contiene tanti elementi quanti sono i valori specificati tra parentesi graffe. Ma è possibile specificare esplicitamente la lunghezza desiderata; ad esempio:

```
int[] v = new int[5] {–100,–200, 123, 400, 5};
```

Detto ciò, rappresenta un errore specificare un numero di valori superiore o inferiore alla lunghezza dichiarata, come nel seguente codice:

```
int[] v = new int[3] {–100,–200, 5, 10};        // errore!
```

L'inizializzazione mediante la sintassi estesa non deve necessariamente aver luogo nella dichiarazione; è dunque ammesso scrivere:

```
int[] v;
...
v = new int[3] {–100,–200, 5};
```

Inizializzazione mediante la sintassi compatta.

Assume la forma:

tipo[] *nome-vettore* = {val_1, val_1 ... val_1};

Ad esempio:

```
int[] v = {1,–20, 100, 22}
```

Nell sintassi compatta l'operatore new viene invocato in modo automatico dal linguaggio. Questa forma, d'altra parte, non consente di indicare esplicitamente la lunghezza del vettore.

Diversamente dalla sintassi estesa, non è ammesso inizializzare un vettore mediante la sintassi compatta separatamente dalla sua dichiarazione. E' dunque errato scrivere:

```
int[] v;
v = {–100,–200, 5};  // errore!
```

3 Array bidimensionali: matrici

Tra matrici e vettori la differenza risiede nell'organizzazione degli elementi; quelli del vettore sono organizzati a mo' di lista, quelli della matrice lo sono a mo' di tabella e dunque sono individuati da due indici, l'indice di riga e quello di colonna:

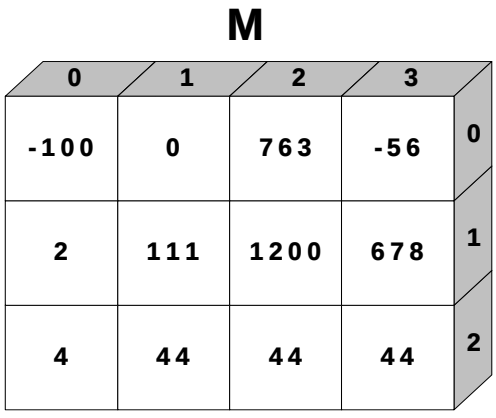


Figura 4-5 Rappresentazione schematica di una matrice.

Come si nota dalla figura, anche nelle matrici gli elementi sono numerati a partire da zero.

3.1 Accesso agli elementi di una matrice

L'accesso agli elementi di una matrice lo si ottiene specificando i due indici dell'elemento; prima l'indice di riga poi quello di colonna, separati da virgola. La sintassi è:

```
nome-matrice[indice-riga, indice-colonna]
```

Ad esempio, considerata la matrice di nome `m` sopra rappresentata, per assegnare il valore 10 all'elemento di riga uno e colonna zero occorre scrivere:

```
m[1, 0] = 10;
```

Oppure, per assegnare alla variabile `a` il valore contenuto nel quarto elemento della terza riga, occorre scrivere:

```
a = m[2, 3];
```

3.2 Dichiarazione e creazione di una matrice

Non esiste differenza sostanziale nella dichiarazione e nella creazione di una matrice rispetto a un vettore. La distinzione sta nel fatto che una matrice possiede una dimensione in più.

Dichiarazione di una matrice

Assume la sintassi:

```
tipo[,] nome-matrice;
```

Ad esempio:

```
int[,] m;
```

L'elemento rilevante rispetto alla dichiarazione di un vettore è rappresentato dalla virgola posta tra le parentesi quadre;essa indica che `m` è una variabile array a due dimensioni.

Creazione di una matrice

Nell'operazione di creazione di una matrice occorre specificare sia il numero di righe che il numero di colonne:

```
nome-matrice = new tipo[numero-righe, numero-colonne];
```

Ad esempio:

```
m = new int[5, 3];
```

La precedente istruzione produce l'allocazione in memoria di una matrice di tipo `int` di 5 righe per 3 colonne. Naturalmente possibile fondere dichiarazione e creazione nella stessa istruzione:

```
int[,] m = new int[5, 3];
```

3.3 Inizializzare gli elementi di una matrice

Nell'inizializzare gli elementi di una matrice occorre fornire un inizializzatore per ogni riga, all'interno del quale saranno specificati i valori iniziali degli elementi della riga. Ciò si ottiene attraverso la seguente sintassi:

nome-matrice = **new** *tipo*[*num-righe*$_{opz,}$ *num-col*$_{opz}$] {*riga*$_1$, *riga*$_{2,}$ ... *riga*$_n$};

dove:

riga$_1$ equivale a: {*val*$_1$, *val*$_2$, ... *val*$_n$}

riga$_2$ equivale a: {*val*$_1$, *val*$_2$, ... *val*$_n$}

riga$_n$ equivale a: {*val*$_1$, *val*$_2$, ... *val*$_n$}

Ad esempio:

```
int[,] m = new int[3,2] {{3,–2}, {0, 10}, {–34,100}};
```

Lo stesso risultato si può ottenere attraverso la sintassi compatta, che non richiede di specificare l'operatore new e le dimensioni della matrice:

```
int[,] m = {{3,–2}, {0, 10}, {–34,100}};
```

Per una maggior leggibilità può essere conveniente scrivere il codice di inizializzare in più righe di programma, come nel seguente esempio:

```
int[,] m = new int[3,2]
{
    {3,–2},
    {0, 10},
    {–34,100}
};
```

E' importante comprendere che il codice suddetto, pur occupando sei righe di programma, rappresenta comunque una singola istruzione.

3.4 Esempi d'uso di matrici

Le matrici sono ideali quando si tratta di elaborare dati omogenei che si presentano, o comunque possono essere organizzati, in forma tabellare.

Esempio n° 1

Date le temperature massime giornaliere di ogni settimana del mese, si vuole consentire all'utente di calcolare la temperatura media settimanale per una settimana a scelta. Per semplicità si consideri un mese di 28 giorni.

I dati – le temperature – sono chiaramente organizzati in due dimensioni:

- le 7 temperature massime giornaliere di ogni settimana;
- le 4 settimane del mese;

In questo caso la matrice è l'ideale per rappresentarli e consentire la loro elaborazione.

Ecco l'implementazione del programma:

```
static void Main()
```

```
{
    string tmp;
    int settimana, i;
    double tempMedia = 0;
    double[,] temperature = new double[4, 7];   // dichiara e crea la matrice

    // ... qui vengono immesse le temperature nella matrice

    Console.WriteLine("Scegliere per quale settimana calcolare la temp. media");
    tmp = Console.ReadLine();
    settimana = Convert.ToInt32(tmp);
// ciclo iterativo che calcola la temperatura media settimanale
    for (i = 0; i < 7; i = i + 1)
        tempMedia = tempMedia + temperature[settimana–1, i];
    tempMedia = tempMedia / 7;
    Console.WriteLine("Temperatura media = {0}", tempMedia);
}
```

Esempio 4-1 Programma per il calcolo della temperatura media settimanale.

Il programma (nel quale è stato omesso il codice di input dei dati) funziona nel seguente modo:

1) presupposto che nella matrice siano già memorizzate le temperature, viene richiesto all'utente il numero della settimana (1 – 4) della quale calcolare la media;

2) all'interno di un ciclo viene eseguita la sommatoria delle temperature relative all'indice di settimana fornito. Successivamente viene calcola la media.

Considerazioni sul programma

La matrice rispecchia la seguente organizzazione: ogni riga rappresenta una settimana. E' questa una scelta arbitraria, e la scelta opposta, ogni colonna una settimana, sarebbe andata altrettanto bene. L'importante è che le dimensioni specificate nella fase di creazione della matrice siano coerenti con gli indici usati in fase di elaborazione. In altre parole, se l'istruzione:

```
tempMedia = tempMedia + temperature[settimana-1, i];
```

fosse scritta scambiando l'indice di riga con quello di colonna:

```
tempMedia = tempMedia + temperature[i, settimana-1];
```

la sua esecuzione determinerebbe il tentativo di accesso a un elemento inesistente della matrice, nella fattispecie il primo elemento della quinta riga (riga inesistente).

La seconda considerazione riguarda l'organizzazione dei dati, che privilegia l'uso di una matrice. E' importante comprendere che, in molti casi, il modo in cui i dati sono organizzati non dipende tanto dalla loro natura in sé, ma dal tipo di elaborazione richiesta su di essi. Ad esempio, se invece di dover calcolare la temperatura media di ogni settimana si volesse ottenere la temperatura media dell'intero mese, la memorizzazione dei dati in un vettore di 28 elementi faciliterebbe l'implementazione del programma rispetto all'uso di una matrice.

Esempio n° 2

Siano date le ore di straordinario settimanali dei 10 dipendenti di un'azienda. Si calcoli il corrispettivo in busta paga, considerata in 25 € la retribuzione per ogni ora di straordinario effettuata.

Si ipotizza che i dati – il numero di ore giornaliere di straordinario – siano memorizzati nelle cinque colonne di una matrice, corrispondenti ai giorni lavorativi della settimana. Ogni riga farà riferimento a un dipendente. Per completezza si può ipotizzare anche che i nomi dei dipendenti siano memorizzati in un vettore di stringhe; ad ogni elemento del vettore corrisponderà la riga della matrice che memorizza gli straordinari.

La figura mostra lo schema di memorizzazione dei dati, considerando soltanto tre dipendenti.

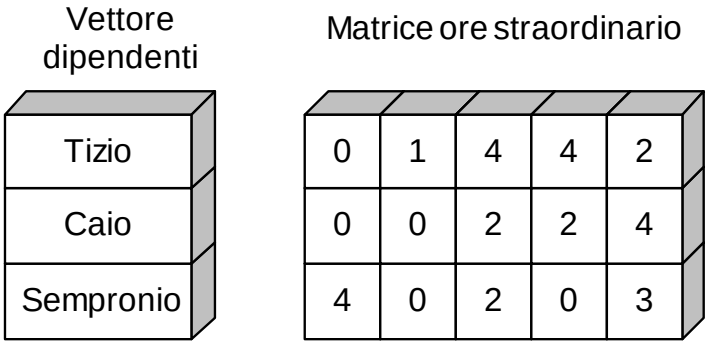


Figura 4-6 Rappresentazione ipotetica dei dati del problema.

Segue l'implementazione del programma:

```
static void Main()
{
    const double PAGAORARIA = 25000;
    int inDip, giorno;
    double corrispettivo;
    string[] nomi = new string[10];
    double[,] ore = new double[10, 5];

// ... qui vengono immesse le ore di straordinario nella matrice e i nomi
// ... dei dipendenti dell'azienda


// calcolo dei corrispettivi in busta paga per ogni dipendente
    for (indDip = 0; indDip < 10; indDip = indDip + 1)
    {
        corrispettivo = 0;
        for (giorno = 0; giorno < 5; giorno = giorno + 1)
            corrispettivo = corrispettivo + ore[indDip, giorno] * PAGAORARIA;
        Console.WriteLine("Al dipendente: {0} corrispondono: {1}", nomi[indDip],
                          corrispettivo);
    }
}
```

Il programma funziona nel seguente modo:

1) all'interno del ciclo `for()` principale vengono esaminate le righe della matrice;

2) per ogni riga, mediante un ciclo `for()` secondario viene calcolato il corrispettivo dovuto al dipendente;

Considerazioni sul programma

Questo programma rappresenta un classico esempio dove, mediante due cicli `for()` nidificati (uno all'interno dell'altro), vengono esaminati tutti gli elementi di una matrice. In questo caso il ciclo principale scandisce la matrice per righe, mentre quello secondario (all'interno del principale) scandisce le colonne di ogni riga.

Se il problema avesse richiesto il calcolo delle ore totali giornaliere di straordinario svolte da tutti i dipendenti, la scansione sarebbe stata prima per colonne (giorno lavorativo) e quindi per righe (somma delle ore di straordinario svolte da tutti i dipendenti).

4 Operazioni permesse sugli array

Diversamente da quanto accade per le variabili semplici, il linguaggio C# non implementa direttamente operazioni tra array. Ad esempio, nell'ambito della matematica, su vettori e matrici (entità analoghe a quelle che stiamo considerando) sono definite operazioni come somma, moltiplicazione, calcolo dell'inverso, eccetera.

Ad esempio, di seguito viene mostrata la somma tra due vettori in ambito matematico:

$$\begin{Bmatrix} 1 \\ 2 \\ 3 \end{Bmatrix} + \begin{Bmatrix} 3 \\ -2 \\ 1 \end{Bmatrix} = \begin{Bmatrix} 4 \\ 0 \\ 4 \end{Bmatrix}$$

Ebbene, il linguaggio C# non ammette affatto qualcosa di analogo:

```
int[] v1 = {1, 2, 3};
int[] v2 = {3, -2, 1};
int[] v3 = new int[3];
v3 = v1 + v2;                  // errore!
```

D'altra parte, il .NET Framework fornisce un insieme di metodi che eseguono operazioni di copia, ordinamento, inversione degli elementi, eccetera. Inoltre, altre operazioni possono essere implementate dal programmatore.

Il discorso cambia, però, quando si parla di variabili array; in questo caso il linguaggio ammette l'uso degli operatori di assegnazione e confronto per uguaglianza. Come vedremo, tali operazioni hanno un significato diverso da quello che normalmente attribuiamo loro.

4.1 Operazione di assegnazione tra variabili array

In C# è possibile assegnazione una variabile array ad un'altra, ma la semantica dell'operatore di assegnazione – il tipo di azione che svolge – è diversa da quella dell'assegnazione che conosciamo. Ciò, in realtà, non è dovuto all'operatore in sé ma alla differenza tra un array e una variabile `int`, `double`, eccetera.

Si ricordi che una variabile array rappresenta un oggetto distinto dall'array in sé. Visivamente, la relazione tra i due oggetti può essere così rappresentata:

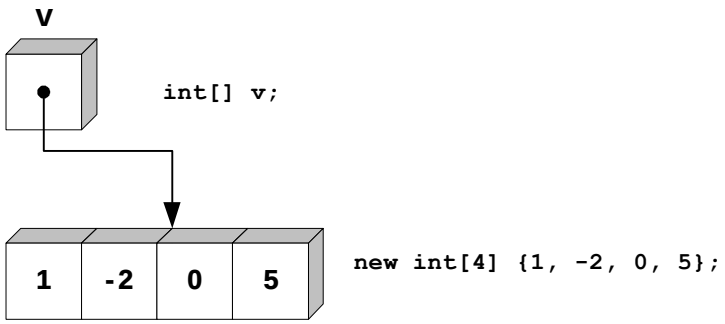


Figura 4-7 Rappresentazione schematica del legame tra variabile array e oggetto array.

La precedente situazione è prodotta attraverso le istruzioni:

```
int[] v;
v = new int[4] {1, –2, 0, 5};
```

La variabile `v` è dunque un **riferimento** all'array, e cioè un tramite per accedere ai suoi elementi, ma non dev'essere confusa con l'oggetto array.

Si consideri ora il seguente codice, che dichiara due variabili array di tipo intero, crea un array di quattro elementi e lo associa alla prima variabile:

```
int[] v;
int[] v1;
v = new int[4] {1, –2, 0, 5};
v1 = v;
```

L'ultima istruzione, `v1 = v;` funziona come qualsiasi altra assegnazione e determina la memorizzazione in `v1` del contenuto di `v`. Ma cosa contiene la variabile `v`? Non contiene un array, ma soltanto un riferimento ad esso. Dunque, ciò che viene memorizzato è un riferimento a un array già esistente. Dopo l'assegnazione, entrambe le variabili si riferiscono allo stesso oggetto array.

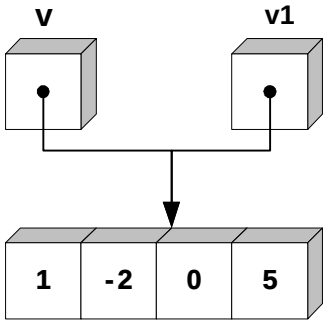


Figura 4-8 Rappresentazione di un doppio riferimento a un oggetto array.

L'assegnazione tra variabili array viene convenzionalmente definita **copia superficiale**, poiché si limita a copiare il riferimento e non l'array stesso; ciò, in contrapposizione alla **copia profonda**, con la quale si intende la copia dell'array vero e proprio, e cioè dei suoi elementi. Ebbene, per assegnare effettivamente un array ad un altro, per eseguire cioè una copia profonda, occorre scrivere del codice che esegua la copia elemento per elemento[7]:

```
int[] v;
v = new int[4] {1, –2, 0, 5};
int[] v1 = new int[4];       // occorre creare il secondo array!
```

```
int i;
for (i = 0; i < 4;i = i + 1)
    v1[i] = v[i];
```

Da notare che prima di eseguire la copia degli alementi è necessario creare il secondo array.

4.2 Regole di compatibilità nell'assegnazione di variabili array

Nell'assegnazione tra due variabili array, il linguaggio richiede che siano dello stesso tipo e non semplicemente compatibili, come avviene con le variabili semplici[8].

Il codice che segue mostra le conseguenze di questa regola:

```
int a = 10;
double x = a;        // ok: il tipo int è compatibile verso il tipo double
int[] vi = {1,2 -2, 4};
double[] vd = new double[4];
vd = vi       // errore!
```

La seconda assegnazione è formalmente sbagliata, poiché le due variabili array sono di tipo diverso. Per lo stesso motivo ne segue che è scorretto anche il codice seguente:

```
double[] vd;
vd = new int[10];    // errore!
```

4.3 Regole di compatibilità nell'assegnazione tra elementi di array

Per l'assegnazione tra gli elementi di array valgono le stesse regole applicate alle variabili semplici; ciò significa che è richiesta la semplice compatibilità, come mostra il seguente codice, nel quale gli elementi di un array di `int` vengono assegnati agli elementi di un array di `double`:

```
int[] vi;
vi = {1, –2, 0, 5};
double[] vd = new int[4];
int i;
for (i = 0; i < 4;i = i + 1)
    vd[i] = vi[i];            // ok: assegnazione tra elementi compatibili
```

4.4 Confronto tra variabili array

Come per l'assegnazione, anche nel confronto tra variabili array la semantica è diversa da quella applicata alle variabili semplici. E anche in questo caso, la differenza non dipende dall'operatore == ma dal modello di memorizzazione degli array.

Confrontando tra due variabili array si stabilisce semplicemente se referenziano lo stesso array oppure no. In tale operazione, gli elementi dei due array non vengono coinvolti affatto.

Il seguente codice lo dimostra:

```
int[] v = {10, 20, 30};
int[] v1 = {10, 20, 30};
if (v == v1)
    Console.WriteLine("v e v1 sono uguali");
else
```

[7] In realtà esiste un metodo per la copia di array che produce lo stesso risultato, ma questo argomento sarà trattato nella parte relativa alle collezioni.

[8] In realtà, questa affermazione è esatta soltanto nell'ambito dei tipi di dati che sono stati trattati.

```
    Console.WriteLine("v e v1 sono diversi");
```

Il codice produce sullo schermo:

```
v e v1 sono diversi
```

poiché `v` e `v1` referenziano array distinti, anche se contenenti esattamente gli stessi valori.

D'altra parte, il seguente codice fornisce la controprova assegnado la variabili `v1` alla variabile `v`:

```
int[] v = {10, 20, 30};
int[] v1 = v
v[2] = -100;
if (v == v1)
    Console.WriteLine("v e v1 sono uguali");
else
    Console.WriteLine("v e v1 sono diversi");
```

Il codice produce sullo schermo:

```
v e v1 sono uguali
```

poiché `v` e `v1` referenziano lo stesso array. Da notare come non abbia alcuna importanza che il terzo elemento di `v` sia stato modificato dopo l'istruzione di assegnazione tra le due variabili.

In conclusione, vale la semplice regola:

due variabili array sono uguali se fanno riferimento allo stesso oggetto array.

Se si desidera verificare se due array contengono gli stessi elementi, è necessario scrivere il codice appropriato, che confronti ogni elemento del primo array con quello corrispondente del secondo.

5

Approfondimento sui costrutti

1 Variabili dichiarate all'interno dei costrutti

Nell'ambito del linguaggio C# esiste un concetto molto importante che riguarda la **visibilità**, o **campo di azione**, di un identificatore. Tale termine designa la parte di programma all'interno della quale si può fare riferimento all'identificatore, sia esso un nome di variabile, di classe, di metodo, eccetera.

L'aspetto della visibilità degli identificatori è molto vasto; qui ci si limiterà a introdurlo in relazione alla possibilità di definire variabili che siano utilizzabili soltanto all'interno dei costrutti nei quali sono dichiarate.

1.1 Variabile locale a un blocco

Si consideri il seguente problema: date due variabili intere, `a` e `b`, si vuole garantire che il valore di `a` sia maggiore o uguale a quello di `b`; se così non è, di dovranno scambiare i valori tra le due variabili. Ecco un frammento di codice che risolve il problema:

```
int a, b;
int tmp;
// ... qui a e b ricevono i loro valori
if (b > a)        // se vero i valori devono essere scambiati
{
    tmp = a;
    a = b;
    b = tmp;
}
```

Per scambiare i valori delle due variabili viene fatto uso della variabile di ausilio `tmp`, che ha lo scopo di fungere da "deposito" temporaneo del valore di `a`. Tale variabile svolge una funzione di supporto, limitata al blocco che contiene le istruzioni di scambio dei valori (evidenziato in grigio). Poiché `tmp` svolge la propria funzione all'interno del blocco e non c'è alcuna necessità di fare riferimento ad essa nel restante codice, sarebbe desiderabile restringere il suo campo di azione al blocco stesso, proibendo esplicitamente il suo uso nella restante parte del programma.

Ciò si ottiene dichiarando la variabile all'interno del blocco:

```
int a, b;
// ... qui a e b ricevono i loro valori
if (b > a)        // i valori devono essere scambiati
{
    int tmp = a;  // tmp è ora locale al blocco
    a = b;        // e non può essere usata al
    b = tmp;      // di fuori di esso!
```

```
}
...
tmp = 0;   // errore: si tenta di usare tmp fuori dal suo campo d'azione!
```

Dichiarando `tmp` all'interno del blocco si circoscrive il suo campo d'azione, e cioè la parte di codice nella quale può essere usata. Ciò fornisce alcuni vantaggi, tra i quali:

- evitare di "mescolare" le dichiarazioni delle variabili necessarie all'algoritmo con quelle delle variabili di supporto che svolgono la propria funzione in una parte circoscritta del codice;
- consentire di usare nuovamente l'identificatore `tmp` all'interno di un altro blocco.

Un secondo esempio chiarirà meglio entrambi i concetti. Si supponga, all'interno dello stesso programma, di dover eseguire un secondo compito del tutto analogo al precedente, questa volta in relazione a due variabili `double`, di nome `x` e `y`. Senza la possibilità di dichiarare variabili all'interno di un blocco è necessario scrivere un codice simile:

```
int a, b;
int tmp;
double x, y;
double tmp1;
// ... qui a, b, x e y  ricevono i loro valori
if (b > a)        // i valori devono essere scambiati
{
    tmp = a;
    a = b;
    b = tmp;
}
if (y > x)        // i valori devono essere scambiati
{
    tmp1 = x;
    x = y;
    y = tmp1;
}
...
```

Ebbene, per svolgere lo stesso compito, che è quello di deposito temporaneo di un valore da scambiare, è necessario dichiarare due variabili, (`tmp` e `tmp1`), poiché sono diversi i tipi in gioco (`a` e `b` sono `int`, `x` e `y` sono `double`). La soluzione che fa uso di variabili locali è senz'altro migliore:

```
int a, b;
double x, y;
// ... qui a, b, x e y  ricevono i loro valori
if (b > a)        // i valori devono essere scambiati
{
    int tmp = a;
    a = b;                          BLOCCO 1
    b = tmp;
}
if (y > x)        // i valori devono essere scambiati
{
    double tmp = x;
                                    BLOCCO 2
```

```
    x = y;
    y = tmp;
}
...
```

Le dichiarazioni iniziali si limitano ora alle sole variabili necessarie all'algoritmo. Inoltre, per svolgere entrambi i compiti viene usato lo stesso nome di variabile che, nei due blocchi, fa riferimento a variabili di tipo diverso.

1.2 Conflitto di nomi

Si ha un conflitto di nomi quando un identificatore può essere virtualmente associato a due entità diverse (variabili, metodi, eccetera). In questo caso o, attraverso le regole del linguaggio, è possibile stabilire senza ambiguità a quale entità faccia riferimento oppure si ha un errore. Tra le regole del linguaggio, la più importante afferma:

all'interno di un blocco, un identificatore deve sempre fare riferimento alla stessa entità[9].

Nell'esempio precedente, nonostante l'identificatore `tmp` venga usato per designare due diverse variabili, non esiste ambiguità, poiché tali variabili appartengono a blocchi diversi e dunque hanno campi di azione distinti. In altre parole, all'interno del `BLOCCO 1` esiste solo la variabile `tmp` di tipo `int`, mentre nel `BLOCCO 2` esiste solo la variabile `tmp` di tipo `double`: in un dato momento esiste una sola variabile `tmp`!

La situazione è invece diversa nel seguente codice:

```
static void Main()
{
    int a, b;
    string tmp;
    // ... qui a e b ricevono i loro valori
    if (b > a)        // i valori devono essere scambiati
    {
        int tmp = a;          // errore: esiste già una variabile di nome tmp!
        a = b;
        b = tmp;
    }
}
```

In questo frammento di codice, nel blocco esterno (che corrisponde al corpo del metodo `Main()`) viene dichiarata la variabile stringa `tmp`; in tutto il blocco, e dunque anche nei blocchi in esso contenuti, qualsiasi riferimento all'identificatore `tmp` indica tale variabile. Il tentativo di dichiarare una nuova variabile di nome `tmp` nel blocco interno produce un errore, poiché crea un conflitto di nomi che il linguaggio non è in grado di risolvere.

Si badi bene che lo stesso errore sarebbe stato prodotto dal codice seguente:

```
static void Main()
{
    int a, b;
    // ... qui a e b ricevono i loro valori
    if (b > a)        // i valori devono essere scambiati
```

[9] Ciò non è vero in generale, ma lo è in relazione al livello di studio di linguaggio che stiamo facendo.

```
    {
        int tmp = a;
        a = b;
        b = tmp;
    }
    string tmp; // errore: esiste già una variabile di nome tmp!
    ...
}
```

Il fatto che la variabile `tmp` di tipo stringa sia dichiarata dopo il blocco interno è del tutto irrilevante:

un blocco introduce un campo d'azione e all'interno di questo due entità distinte non possono avere lo stesso nome.

1.3 Contatore locale al ciclo for()

La possibilità di dichiarare variabili il cui campo d'azione sia ristretto a una determinata parte di codice è molto utile nel ciclo iterativo `for()`. Infatti, la sintassi di questo costrutto ammette nella parte di inizializzazione la dichiarazione della variabile contatore del ciclo[10].

Dunque, la sintassi:

for (*inizializzazione*$_{opz}$; *condizione*$_{opz}$; *incremento*$_{opz}$)
istruzione;

diventa:

for (*dichiarazione-inizializzazione*$_{opz}$; *condizione*$_{opz}$; *incremento*$_{opz}$)
istruzione;

Segue una nuova versione dell'Esempio 2.2, che sfrutta questa possibilità:

```
static void Main()
{
    int n, somma;
    string tmp;
    tmp = Console.ReadLine();
    n = Convert.ToInt32(tmp);
// ----------------------- <n è fornito>
    somma = 0;
    for (int x = 1; x <= n; x = x + 1)
        somma = somma + x;
    Console.WriteLine("La somma è: {0}", somma);
}
```

Il campo d'azione di una variabile dichiarata nella parte inizializzazione di un ciclo `for()` coincide con il ciclo stesso; dunque, può essere fatto riferimento ad essa soltanto nel controllo e nel corpo del ciclo.

Esattamente come per le variabili locali ai blocchi, se esiste la dichiarazione di una variabile con lo stesso nome nel blocco che contiene il ciclo (tipicamente il corpo del metodo) si ha un errore:

```
int i;
// qui ci sono altre dichiarazioni;
```

[10] La sintassi, in realtà, ammette la dichiarazione di più variabili. Tale uso è però poco comune e dunque non sarà trattato.

```
for (int i = 0; i < 10; i = i + 1) // errore: doppia dichiarazione di i!
    // qui c'è il corpo del ciclo
```

La dichiarazione locale della variabile contatore di un ciclo `for()`, oltre ai vantaggi già sottolineati in precedenza, favorisce la scrittura di codice esente da errori piuttosto insidiosi. Si consideri la seguente situazione: si vuole visualizzare il contenuto di una matrice intera di 10 righe per 5 colonne. Ecco un frammento di codice che svolge tale compito:

```
int[,] m = new int[10, 5];
int i;
// ... qui vengono inseriti i valori nella matrice
for (i = 0; i < 10; i = i + 1)
    for (i = 0; i < 5; i = i + 1)
        Console.WriteLine(m[i, i]);
```

Il codice è formalmente corretto ma contiene un errore che produce l'effetto di fare entrare il programma in un loop infinito: il flusso di esecuzione non esce mai dal ciclo `for()` esterno e dunque il programma non si fermerà mai. L'errore è dovuto all'uso dello stessa variabile come contatore dei due cicli; uso consentito, poiché `i` è dichiarata all'inizio del blocco che contiene entrambi i cicli. Un errore simile può essere evitato dichiarando le variabili contatore localmente ai cicli. Infatti, scrivendo, sempre erroneamente:

```
int[,] m = new int[10, 5];
// ... qui vengono memorizzati i valori nella matrice
for (int i = 0; i < 10; i = i + 1)
    for (int i = 0; i < 5; i = i + 1)     // errore: doppia dichiarazione!
        Console.WriteLine(m[i,i]);
```

si ottiene questa volta un messaggio di errore del compilatore, il quale ci informa che tentiamo di dichiarare due volte la stessa variabile. Con questa informazione, correggere l'errore diventa semplice.

> L'esempio precedente, prima ancora di ricordarci l'utilità di dichiarare localmente le variabili contatore, ci insegna ad usare nomi maggiormente autoesplicativi per le stesse. Infatti, nomi come `i`, `j`, `k`, `z`, eccetera, abbondantemente usati nel testo per esigenze di compattezza, sono troppo generici e nella maggior parte dei casi non rappresentano una buona scelta, poiché non forniscono indizi sull'uso che viene fatto del contatore, e dunque favoriscono la presenza di errori nel codice.
>
> Resta il fatto, comunque, che esistono situazioni nelle quali ci sono variabili che svolgono una funzione di supporto, che hanno necessariamente nomi generici. Dichiarare localmente tali variabili diventa ancora più importante.

2 Ottimizzazione dei cicli: uso degli operatori di incremento e decremento

C# mette a disposizione due operatori la cui funzione è quella di incrementare (o decrementare) di uno la variabile alla quale sono applicati. Benché il loro uso non sia affatto circoscritto ai cicli iterativi, è proprio all'interno di essi che tali operatori trovano il loro impiego più comune. Nella maggior parte dei cicli, infatti, una variabile contatore viene incrementata o decrementata di una unità ad ogni iterazione.

Entrambi gli operatori assumono due forme:

- **forma postfissa**: l'operatore segue la variabile alla quale viene applicato;
- **forma prefissa**: l'operatore precede la variabili alla quale viene applicato.

Per l'uso che faremo di entrambi gli operatori, le due forme sono equivalenti.[11]

2.1 Operatore di incremento

La sintassi d'uso dell'operatore incremento è:

```
variabile++;      (forma postfissa)
++variabile;      (forma prefissa)
```

In entrambe le forme, viene incrementata di uno *variabile*, la quale deve essere di tipo intero.

L'operatore incremento dovrebbe essere usato al posto delle istruzioni che appaiono nella forma:

```
variabile = variabile + 1
```

poiché, oltre ad avere una sintassi più compatta, viene tradotto in un codice eseguibile più efficiente. Un esempio d'uso di questo operatore in un ciclo `for()` è il seguente:

```
int[] v;
v = new int[4] {1, –2, 0, 5};
int[] v1 = new int[4];
for(int i = 0; i < 4;i++)
    v1[i] = v[i];
```

Lo stesso esempio, realizzato attraverso un ciclo `while()`:

```
int[] v;
v = new int[4] {1, –2, 0, 5};
int[] v1 = new int[4];
int i = 0;
while(i < 4)
{
    v1[i] = v[i];
    i++;
}
```

2.2 Operatore di decremento

La sintassi d'uso dell'operatore decremento è:

```
variabile--;      (forma posfissa)
--variabile;      (forma prefissa)
```

In entrambe le forme, viene decrementata di uno *variabile*, la quale dev'essere di tipo intero. L'operatore decremento dovrebbe essere usato al posto delle istruzioni che appaiono nella forma:

```
variabile = variabile – 1
```

[11] In generale, le due forme sono sempre equivalenti per l'effetto che producono sulla variabile. Non lo sono per un'altra questione, più sottile, che qui non viene trattata.

Un esempio d'uso di questo operatore in un ciclo `for()` è:

```
int[] v = {1, –2, 0, 5};
for (int i = 4; i >= 0;i--)
    Console.WriteLine("Elemento {0} = {1}", i, v[i]);
```

3 Ciclo iterativo do while()

I cicli iterativi `while()` e `for()` rappresentano, pur in forme sintatticamente diverse, l'implementazione del medesimo schema di iterazione: l'iterazione con controllo in testa. C# mette a disposizione un costrutto anche per l'implementazione dello schema di iterazione con controllo in coda, nel quale prima viene eseguito il corpo del ciclo e soltanto dopo viene valutata la condizione che determina o meno l'uscita dallo stesso.

Il ciclo `do while()` presenta la seguente sintassi:

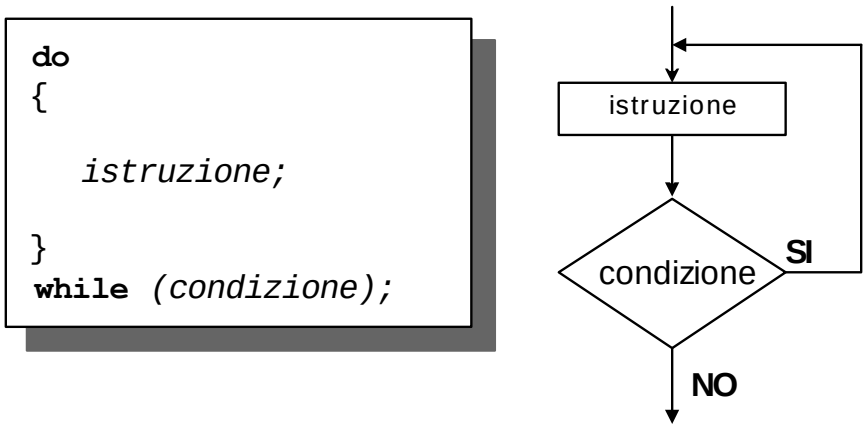


Figura 5-1 Sintassi del costrutto do while() e schema a blocchi equivalente.

Il ciclo e viene eseguito nel seguente modo:

1) viene eseguita l'istruzione o le istruzioni presenti all'interno del blocco;
2) viene quindi valutata la condizione; se risulta vera, il ciclo viene reiterato, altrimenti termina.

Da notare che diversamente dai cicli `while()` e `for()`, il ciclo `do while()` richiede che il corpo del ciclo sia sempre costituito da un blocco.

3.1 Uso del ciclo do while()

Il ciclo `do while()` si distingue dagli altri cicli perché garantisce almeno un'iterazione, poiché quand'anche la condizione fosse subito falsa, il corpo del ciclo sarebbe già stato eseguito. Normalmente, questa non è una caratteristica desiderabile, poiché nella maggior parte dei casi l'esecuzione del corpo del ciclo dev'essere subordinata al fatto che la condizione sia vera. Ciò può portare a un codice potenzialmente errato. Si consideri ad esempio il seguente problema:

Acquisiti dall'utente due valori interi positivi che rappresentano la base e l'esponente di una potenza, si vuole calcolare la potenza medesima. Si ipotizzi per la base un valore maggiore di zero.

La soluzione è semplice: moltiplicare la base tante volte quante ne indica l'esponente. Ecco l'implementazione:

```
static void Main()
{
    int i, base, esponente, potenza = 1;
    string tmp;
    Console.WriteLine("Inserire 'base' ed 'esponente'");
    tmp = Console.ReadLine();
    base = Convert.ToInt32(tmp);
    tmp = Console.ReadLine();
    esponente = Convert.ToInt32(tmp);
    i=1;
    do
    {
        potenza = potenza * base
        i++;
    }
    while (i <= esponente);
    Console.WriteLine("La potenza è: {0}", potenza);
}
```

Il programma funziona correttamente eccetto che per un caso, e cioè quando l'esponente vale zero. Poiché la potenza di qualsiasi numero elevato a zero è sempre uno, nel caso di esponente zero il corpo del ciclo iterativo non dovrebbe essere eseguito. Ma un ciclo `do while()` garantisce almeno un'iterazione e in questo caso ciò rappresenta un errore.

In un problema come questo è corretto utilizzare un ciclo `while()` o `for()`:

```
...
i = 1
while (i <= esponente);
{
    potenza = potenza * base
    i++;
}
...
```

Se l'esponente è zero, la condizione `i <= esponente` è subito falsa e il corpo del ciclo, correttamente, non viene mai eseguito.

Situazioni nelle quali risulta appropriato l'uso del ciclo do while()

L'esempio precedente dimostra che normalmente si dovrebbero preferire i cicli `while()` e `for()` al ciclo `do while()`. Esistono però situazioni nelle quali l'uso del `do while()` si presta in modo del tutto naturale. Si consideri ad esempio il seguente problema:

Si vuole consentire all'utente di inserire una sequenza arbitrariamente lunga di interi positivi, da memorizzare in un vettore. L'immissione dei dati avrà termine quando l'utente inserirà un valore negativo, anch'esso da memorizzare nel vettore.

Ecco l'implementazione:

```
static void Main()
{
```

```
    const int numeroMassimoValori = 100;
    int[] dati = new int[numeroMassimoValori];
    int i=0;
    Console.WriteLine("Inserire i valori della sequenza");
    do
    {
        string tmp = Console.ReadLine();
        dati[i] = Convert.ToInt32(tmp);
        i++;
    }
    while (dati[i-1] > 0);    // nota: i-1 punta all'ultimo valore inserito
}
```

In questo caso l'uso del ciclo `do while()` è appropriato, perché la natura del problema impone l'acquisizione di almeno un valore e dunque l'esecuzione, almeno una volta, del corpo del ciclo.

In questo caso, la soluzione implementata con un ciclo `while()` risulta meno naturale:

```
static void Main()
{
    int[] dati = new int[100];
    int i=0;
    string tmp = Console.ReadLine();
    dati[i] = Convert.ToInt32(tmp);
    while (dati[i] > 0)
    {
        i++;
        tmp = Console.ReadLine();
        dati[i] = Convert.ToInt32(tmp);
    }
}
```

poiché rende necessario duplicare le istruzioni che costituiscono il corpo del ciclo, allo scopo di garantire l'immissione di almeno un valore.

3.2 Costrutto do while() e punto-e-virgola di fine istruzione

Il ciclo iterativo `do while()` è l'unico, tra i costrutti di controllo, che richiede come requisito formale il punto e virgola dopo la parentesi di chiusura del controllo del ciclo.

Ad esempio, il seguente codice è sbagliato:

```
int i = 0;
do
{
    i++
}
while (i < 10)    // errore: manca il punto e virgola!
```

Questa particolarità del ciclo `do while()` può provocare qualche problema di chiarezza nel codice, poiché è abbastanza facile confondere la parte terminale del ciclo `do while()` con la parte iniziale del ciclo `while()`.

Si osservi ad esempio il seguente frammento di codice:

```
int i = 0;
```

```
do
{
    i++
}
while (i < 10);       // ok: qui ci vuole il punto e virgola
...
i = 0;
while (i < 10)        // ok: qui non ci vuole il punto e virgola
    i++;
```

Apparentemente, due istruzioni identiche seguono due regole diverse relativamente al punto e virgola, ma non è così. Infatti, nel primo caso l'istruzione `while (i < 10)` rappresenta il controllo di un ciclo `do while()`, mentre nel secondo caso rappresenta il controllo di un ciclo `while()`.

4 Ciclo iterativo foreach()

Il ciclo iterativo `foreach()` è un costrutto messo a disposizione dal linguaggio con il preciso intento di semplificare l'accesso agli elementi di una collezione. Esso nasconde le complicazioni legate all'inizializzazione e all'incremento di una variabile indice, nonché all'uso dell'operatore di accesso a elemento, fornendo al loro posto un meccanismo estremamente semplice e compatto.

Il ciclo `foreach()` assume la sintassi mostrata in figura, che può essere approssimativamente tradotta nello schema a blocchi a destra:

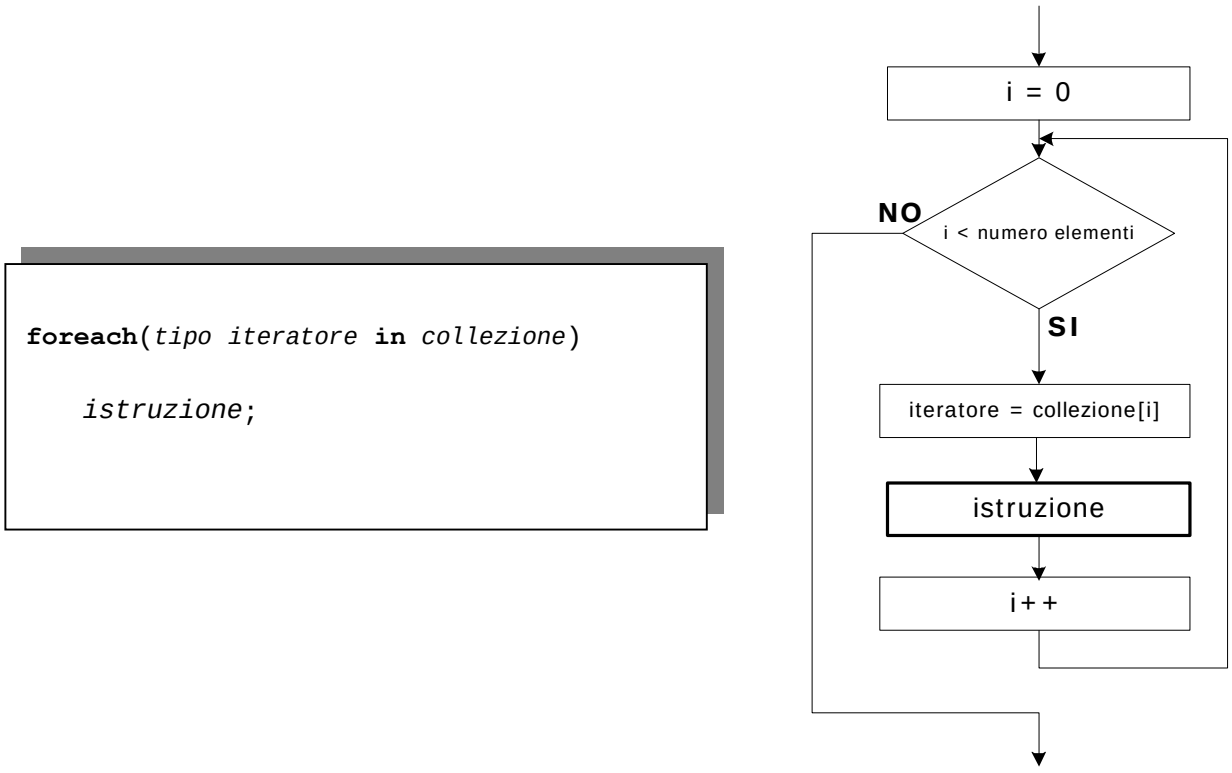


Figura 5-2 Sintassi del costrutto `foreach()` e schema a blocchi equivalente.

Il costrutto funziona nel seguente modo:

1) viene dichiarata e inizializzata una variabile indice nascosta, la cui funzione è quella di puntare agli elementi della collezione;

2) per ogni iterazione del ciclo, alla variabile *iteratore* viene assegnato l'iesimo elemento della collezione, puntato dalla variabile indice nascosta;

3) il ciclo termina dopo che alla variabile *iteratore* è stato assegnato anche l'ultimo elemento della collezione.

In sostanza la variabile *iteratore* assume il valore di ogni elemento della collezione.

Vi sono tre requisiti molto importanti da considerare riguardo la variabile *iteratore*:

1) dev'essere dello stesso tipo degli elementi della collezione[12];

2) è proibito modificare il suo valore;

3) essendo dichiarata localmente al ciclo, il suo campo d'azione è ristretto al ciclo stesso e dunque non può essere usata fuori di esso.

4.1 Uso del ciclo foreach()

Il ciclo `foreach()` è l'ideale quando si deve accedere a tutti gli elementi di una collezione senza la necessità di modificarli. Si consideri ad esempio di voler visualizzare i nomi dei dipendenti di un'azienda memorizzati in un vettore:

```
string[] dipendenti = new string[10];
// ... qui i nomi dei dipendenti vengono memorizzati nel vettore
foreach (string nome in dipendenti)
   Console.WriteLine(nome);
```

Il precedente codice può essere reimplementato in modo del tutto equivalente facendo uso del ciclo `for()`:

```
string[] dipendenti = new string [10];
// ... qui i nomi dipendenti vengono memorizzati nel vettore
for (int i = 0; i < 10; i++)
   Console.WriteLine(dipendenti[i]);
```

Come si vede, l'implementazione tramite `foreach()` risulta molto più semplice, poiché non c'è alcun bisogno di considerare indici, inizializzazioni, incrementi, condizioni e accesso a elementi; tutto il lavoro viene svolto in modo trasparente dal linguaggio.

Uso del foreach() con gli array bidimensionali

Il ciclo `foreach()` è in grado di trattare una matrice come se fosse un vettore, scorrendo prima tutti gli elementi della riga 0, poi quelli della riga 1, e così via, fino all'ultimo elemento dell'ultima riga. Ad esempio, si consideri il problema di calcolare la somma di tutti gli elementi di una matrice:

```
double[,] m = new double[10, 5];
// ... qui la matrice m riceve i propri valori
```

[12] Ciò non è propriamente esatto, poiché il linguaggio consente di dichiarare variabili *iteratore* di tipo diverso purché sia possibile eseguire una conversione tra tipo dell'elemento e tipo della variabile. Questa possibilità riguarda un uso del ciclo `foreach()` che qui non viene trattato.

```
double somma = 0;
foreach (double valore in m)
    somma = somma + valore;
Console.WriteLine("La somma è: {0}", somma);
```

La soluzione di un simile problema attraverso un ciclo `for()` risulta meno compatta:

```
double[,] m = new double[10, 5];
// ... qui la matrice m riceve i propri valori
double somma = 0;
for (int r = 0; r < 10; r++)
    for (int c = 0; c < 5; c++)
        somma = somma + m[r,c];
Console.WriteLine("La somma è: {0}", somma);
```

4.2 Limiti e pregi del ciclo foreach()

Il limite principale è naturalmente costituito dal suo campo d'applicazione, che è quello dell'accesso agli elementi di una collezione; i cicli `for()`, `while()` e `do while()`, per contro, sono costrutti utilizzabili ovunque vi sia l'esigenza di reiterare l'esecuzione di un determinato compito. Un secondo limite, più specifico, riguarda l'accesso agli elementi, che può essere solo in lettura; la variabile iteratore, infatti, non può essere oggetto di un'assegnazione. Un terzo limite, che è poi la caratteristica del `foreach()`, riguarda l'impossibilità di elaborare soltanto un sotto insieme degli elementi della collezione. La sintassi prevede infatti che la variabile iteratore acceda a tutti gli elementi della collezione; non esiste la possibilità di impostare una particolare condizione di uscita, oppure gestire un indice che determini l'accesso ad alcuni elementi soltanto.

Ad esempio, un problema del tipo: "visualizzare tutti gli elementi di indice pari di un vettore" non si presta in modo particolare ad essere trattato con un `foreach()`, benché ovviamente si possa farlo con l'ausilio di una variabile indice, al costo però di complicare il codice:

```
int[] v = new int[10];
// ... qui il vettore v riceve i propri valori
int i = 0;
foreach (int valore in v)
{
    if (i % 2 == 0)
        Console.WriteLine(valore);
    i++
}
```

6

Introduzione ai metodi

1 Premessa

Tutti i programmi considerati finora sono caratterizzati dall'esistenza di un solo metodo, il metodo principale `Main()`, contenente tutte le istruzioni del programma. E' questa un'eccezione, non la regola, dovuta al carattere dimostrativo degli esempi presentati. Qualsiasi programma realistico svolge solitamente molti compiti e implementa vari algoritmi, e nonostante il linguaggio non imponga alcun requisito sulla collocazione e sulla organizzazione delle istruzioni, di norma ogni algoritmo viene implementato attraverso un metodo diverso.

La realizzazione e l'uso di metodi rappresenta un aspetto fondamentale in qualsiasi linguaggio di programmazione. Anche i semplici esempi mostrati finora hanno fatto uso di metodi: `ToDouble()`, e `ToInt32()` della classe `Convert`, `WriteLine()` e `ReadLine()` della classe `Console`, `Sqrt()` e `Pow()` della classe `Math`. In questo capitolo saranno introdotti i concetti fondamentali che stanno alla base della realizzazioni di metodi, i quali saranno visti come strumento per organizzare in modo logico i compiti che deve svolgere il programma. Nel capitolo i metodi successivo saranno caratterizzati con maggior dettaglio e saranno mostrati altri ambiti applicazione, tra cui quello, estremamente importante, di estendere le funzionalità messe a disposizione da .NET.

1.1 Convenzioni usate nel presentare gli esempi

Resta implicita l'organizzazione generale del programma mostrata nel primo capitolo:

```
using System;
class Program
{
    static void Main()
    {
        ...
    }
}
```

D'ora in avanti, alla classe `Program` si aggiungono i metodi implementati.

2 Salve, Mondo!, Arrivederci mondo!

Si consideri l'idea di realizzare una nuova versione del programma "Salve, Mondo!". Il nuovo programma dovrà visualizzare il canonico messaggio di benvenuto, restare in attesa finché l'utente non preme INVIO e visualizzare un messaggio di commiato.

Ecco l'implementazione:

```
class Program
```

```
{
    static void SalutoIniziale()
    {
        Console.WriteLine("Salve, mondo!");
    }

    static void SalutoFinale()
    {
        Console.WriteLine("Arrivederci, mondo!");
    }

    static void Main()
    {
        SalutoIniziale()
        string tmp = Console.ReadLine();
        SalutoFinale();
    }
}
```

DEFINIZIONE

DEFINIZIONE

INVOCAZIONE

INVOCAZIONE

Il programma, oltre al metodo principale `Main()`, presenta altri due metodi, `SalutoIniziale()` e `SalutoFinale()`, il cui compito è semplicemente quello di eseguire il metodo `WriteLine()` della classe `Console` con l'appropriato messaggio da visualizzare.

Per entrambi si possono individuare:

- la **definizione**: qui il metodo viene descritto, viene cioè definito il suo **prototipo** e le istruzioni che lo compongono. Un metodo può essere definito una volta soltanto all'interno della classe.[13]
- **invocazione**, o **chiamata**: qui il metodo viene eseguito; vengono cioè eseguite le istruzioni che lo compongono. Un metodo può essere invocato quante volte si vuole.

3 Definizione di un metodo

La definizione di un metodo rappresenta qualcosa di analogo alla dichiarazione di una variabile, benché le due siano entità completamente distinte. Definire un metodo significa creare una nuova funzionalità della classe alla quale il metodo appartiene, così come dichiarare una variabile significa creare un nuovo oggetto allo scopo di memorizzare informazioni.

La definizione assume la seguente forma generale:

```
modificatori_opz tipo-ritorno nome-metodo (lista-parametri_opz)
{
     // qui stanno le istruzioni del metodo
}
```

ma per il momento sarà considerata la seguente forma semplificata:

```
static void nome-metodo ()
```

[13] Questa affermazione non deve essere presa alla lettere, poiché C# consente di definire più metodi aventi lo stesso nome, ma aventi un in diverso insieme di parametri. E' questo un aspetto che qui non viene trattato.

```
{
    // qui stanno le istruzioni del metodo
}
```

Nella definizione si possono individuare due parti:

- il **prototipo**, o **intestazione** del metodo;
- il **corpo del metodo**, e cioè il blocco che contiene le istruzioni.

3.1 Prototipo di un metodo

Nella definizione di un metodo, il prototipo è rappresentato dalla prima riga; esso caratterizza un metodo come il tipo e il nome caratterizzano una variabile. Il prototipo descrive:

- il tipo del valore ritornato dal metodo, o la parola chiave `void` se il metodo non produce alcun valore;
- il nome del metodo;
- tra parentesi, la lista dei parametri che sono necessari al metodo per funzionare, oppure niente se il metodo non necessita di parametri.

Ad esempio, nella definizione del metodo `SalutoIniziale()`, il prototipo è rappresentato da[14]:

```
void SalutoIniziale()
```

Per il momento saranno considerati soltanto quei metodi che non producono alcun valore di ritorno e non necessitano di parametri.

3.2 Corpo di un metodo

Il corpo definisce le istruzioni che sono eseguite all'atto dell'invocazione del metodo ed è rappresentato da un blocco. Ad esempio, nella definizione del metodo `SalutoFinale()` il corpo del metodo è rappresentato da:

```
{
      Console.WriteLine("Salve, mondo!");
}
```

All'interno del corpo di un metodo sono ammesse dichiarazioni di variabili e qualsiasi istruzione esecutiva. Non è ammesso invece definire un metodo all'interno di un altro metodo, né usare direttive `using`, né tantomeno definire classi.

3.3 Ordine di definizione dei metodi

Non esiste un ordine prestabilito nella definizione dei metodi né, come avviene per le variabili locali di un blocco, è necessario che la definizione di un metodo preceda la sua chiamata.

Ad esempio, il seguente programma è formalmente corretto e funzionante:

```
class Program
{
    static void Main()
```

[14] Del prototipo di un metodo non fa parte la parola chiave `static`.

```
    {
        SalutoIniziale()
        string tmp = Console.ReadLine();
        SalutoFinale();
    }

    static void SalutoFinale()
    {
        Console.WriteLine("Arrivederci, mondo!");
    }
    static void SalutoIniziale()
    {
        Console.WriteLine("Salve, mondo!");
    }

}
```

Come si vede, istruzioni di invocazione ai metodi `SalutoIniziale()` e `SalutoFinale()` precedono la definizione di entrambi; ebbene, ciò è del tutto irrilevante per la correttezza formale e il buon funzionamento del programma.

4 Invocazione di un metodo

Sempre riferendoci per analogie al concetto di variabile, l'invocazione di un metodo rappresenta qualcosa di analogo all'uso di una variabile. In altre parole, un metodo svolge la funzione per la quale è stato definito soltanto all'atto dell'invocazione. Un metodo definito ma mai invocato non produce alcuna influenza sull'esecuzione del programma, esattamente come una variabile dichiarata ma mai utilizzata. L'invocazione di un metodo assume la seguente sintassi generale:

nome-metodo (*lista-argomenti*$_{opz}$)

ma per il momento si farà riferimento alla seguente sintassi semplificata:

nome-metodo ()

L'invocazione di un metodo può avvenire sia come parte di un'espressione:

```
double x = Math.Sqrt(2) * 2;        // invocazione del metodo Sqrt()
```

Sia come unico elemento dell'istruzione:

```
Console.WriteLine("Salve, Mondo!");       // invocazione del metodo WriteLine()
```

nel qual caso si parla di **istruzione di chiamata**.

Per ora sarà considerato soltanto questo tipo di invocazione, utilizzata due volte nel programma di esempio:

```
SalutoIniziale();
```

e:

```
SalutoFinale();
```

4.1 Alterazione del flusso di esecuzione provocato dall'invocazione di un metodo

L'invocazione di un metodo altera il flusso di esecuzione delle istruzioni, che normalmente segue l'ordine di elencazione delle stesse ed è governato dai costrutti di controllo. All'atto dell'invocazione, il flusso viene trasferito alla prima istruzione che appartiene al metodo invocato. Da questo momento, questo segue l'ordine delle istruzioni elencate nel metodo, fino al termine dello stesso. Dopo che è stata eseguita l'ultima istruzione del metodo invocato, il flusso di esecuzione viene trasferito all'istruzione successiva a quella di invocazione.

In questo stato di cose esistono due ruoli distinti:

- il **metodo chiamante** che rappresenta il metodo all'interno del quale si trova l'istruzione di invocazione del
- **metodo chiamato**, che rappresenta il metodo che viene eseguito all'atto dell'invocazione.

Lo schema mostrato a pagina successiva, che si riferisce al precedente programma di esempio, chiarisce il concetto. Le frecce indicano l'andamento del flusso di esecuzione del programma, che inizia e termina con la prima e ultima istruzione del metodo `Main()`. L'invocazione di un metodo, ad esempio `SalutoIniziale()`, determina il trasferimento del flusso di esecuzione dal metodo `Main()` al metodo chiamato, il quale ne avrà il controllo fino a quando terminerà, trasferendolo nuovamente al metodo che lo aveva chiamato, e cioè `Main()`. Nonostante ciò non sia rappresentato nello schema, un metodo che assume il controllo del flusso di esecuzione può a sua volta trasferirlo a un altro metodo, e così via.

Nel seguente codice, ad esempio:

```
static void SalutoFinale()
{
    SalutoIniziale();              // chiamata di SalutoIniziale();
    Console.WriteLine("Arrivederci, mondo!");
}
```

Il metodo `SalutoFinale()` rappresenta il metodo chiamante, mentre `SalutoIniziale()` il metodo chiamato.

In ogni caso, vale sempre la regola:

la fine dell'esecuzione del metodo chiamato determina sempre il ritorno del flusso di esecuzione al metodo chiamante.

Ad ogni invocazione, dunque, deve prima o poi corrispondere un ritorno. Se ciò non avviene può significare soltanto due cose:

- l'esecuzione del programma è stata interrotta prima della sua normale terminazione;
- l'esecuzione del programma è entrata in un loop infinito e non terminerà senza un intervento dall'esterno.

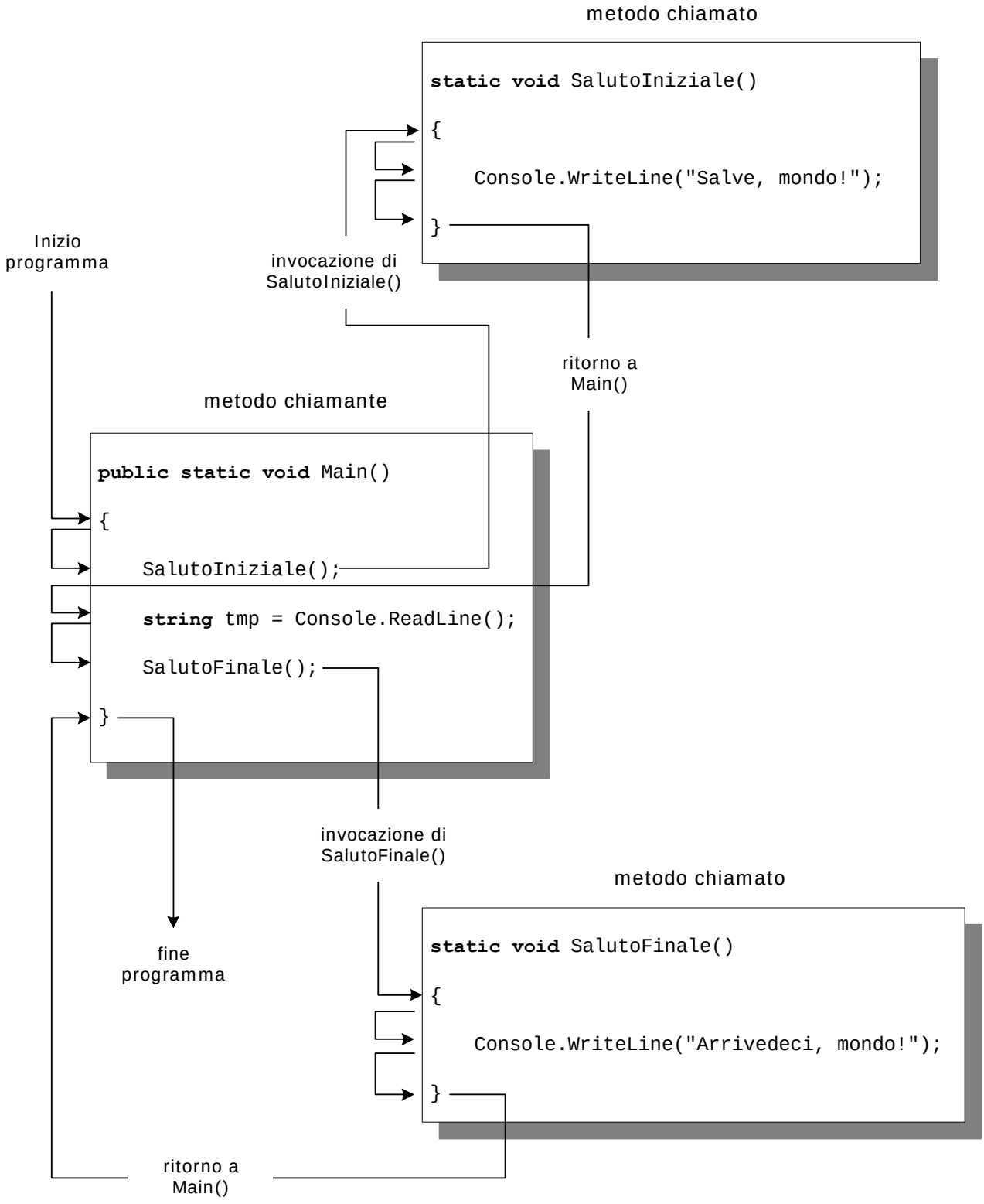


Figura 6-1 Andamento del flusso di esecuzione durante l'invocazione di un metodo.

5 Uso dei metodi

L'uso più convenzionale che viene fatto dei metodi è quello di suddividere logicamente i compiti che deve svolgere il programma allo scopo di semplificarlo. Per un esempio più concreto di quello precedente si riconsideri il problema relativo all'Esempio 4.1, il cui testo viene qui riportato:

Date le temperature massime giornaliere di ogni settimana del mese, si vuole consentire all'utente di calcolare la temperatura media settimanale per una settimana a scelta. Per semplicità si consideri un mese di 28 giorni.

Analizzando il testo del problema si possono individuare quattro compiti distinti:

1) acquisizione delle temperature (questo compito nella precedente implementazione era stato dato per assunto);
2) scelta della settimana per la quale calcolare la temperatura media;
3) calcolo della temperatura media;
4) visualizzazione del risultato;

Si decide di implementare i compiti più complessi, 1) e 3), attraverso dei metodi:

```
class Program
{
    static void Main()
    {
        string tmp;
        int settimana;
        double tempMedia = 0;
        double[,] temperature = new double[4, 7];

        AcquisisciTemperature();
        Console.WriteLine("Inserire n° della settimana [1-4]");
        tmp = Console.ReadLine();
        settimana = Convert.ToInt32(tmp);

        CalcolaTemperaturaMedia();

        Console.WriteLine("Temperatura media = {0}", tempMedia);
    }

    static void AcquisisciTemperature()
    {
        for (int indSettimana = 0; indSettimana < 4; indSettimana++)
        {
            for (int giorno = 0; giorno < 7; giorno++)
            {
                Console.Write("Settimana {0} - Temp. {1}° giorno:",
                              indSettimana+1, giorno+1);
                string tmp = Console.ReadLine();
                temperature[indSettimana, giorno] = Convert.ToDouble(tmp);
```

```
            }
        }
    }

    static void CalcolaTemperaturaMedia()
    {
        double tempMedia = 0;
        int settimana;
        for (int giorno = 0; giorno < 7; giorno++)
            tempMedia = tempMedia + temperature[settimana-1, giorno];
        tempMedia = tempMedia / 7;
    }
}
```

Esempio 6-1 Programma che calcola la temperatura media settimanale.

Il metodo `Main()` rispecchia la logica del programma. Esso ha infatti il compito di creare la matrice `temperature`, invocare il metodo di acquisizione delle temperature inserite dall'utente, chiedere all'utente quale settimana elaborare, invocare il metodo di calcolo della temperatura media e infine visualizzare il risultato.

L'implementazione dei compiti relativamente più complessi, acquisire i dati e calcolare la media, è demandata a due metodi distinti; ciò semplifica il metodo `Main()`, ma anche tutto il programma nel suo insieme, poiché ognuno dei tre metodi raggiunge così un livello modesto di complessità. Inoltre, se riunissimo tutto il codice nel solo metodo `Main()` perderebbe in chiarezza, poiché sarebbe più difficile comprendere la logica generale del programma.

In tutto questo c'è solo un problema: il programma, così come è stato scritto, non sarà nemmeno compilato! Non è un problema di logica sbagliata o di istruzioni scritte scorrettamente; il problema sta tutto nelle variabili e nel loro campo d'azione. Tutte le variabili utilizzate, infatti, sono dichiarate localmente nei corpi dei tre metodi e dunque sono utilizzabili solo all'interno di essi! Ciò produce sia errori formali che di cattivo funzionamento. Eccone due esempi:

1) la matrice `temperature` è dichiarata e creata nel metodo `Main()`; ciò significa che qualsiasi riferimento ad essa al di fuori di tale metodo rappresenta un errore formale: al di fuori di `Main()` la variabile `temperature` semplicemente non esiste;

2) esistono dichiarazioni e riferimenti alla variabile `tempMedia` sia nel metodo `Main()` che nel metodo `CalcolaTemperaturaMedia()`; ebbene, sono riferimenti a oggetti completamente distinti! Qualsiasi modifica a una delle due non si riflette nell'altra e viceversa.

Per quanto riguarda il punto 2), il programma è formalmente corretto, ma funzionalmente sbagliato. Infatti, il metodo `CalcolaTemperaturaMedia()` esegue il calcolo correttamente ma memorizza il risultato in una variabile che non esiste al di fuori di esso. Analogamente, `Main()` visualizza il valore di `tempMedia` dopo aver correttamente invocato il metodo `CalcolaTemperaturaMedia()`, ma ciò che viene realmente visualizzato è il valore della variabile `tempMedia` dichiarata localmente a `Main()`. Questa variabile è inizializzata a zero e tale resterà per tutta la durata del programma.

In conclusione, poiché l'uso delle variabili `temperature`, `tempMedia` e `settimana` è condiviso da più metodi, le stesse non possono essere dichiarate localmente a nessuno di essi. Occorre un modo per estendere il loro campo d'azione a tutti i metodi che ne fanno uso.

6 Campi di classe

C# consente di dichiarare variabili a livello di classe; queste chiamate **campi di classe** o **campi membro**, sono utilizzabili in qualsiasi metodo che appartenga alla classe.

Si consideri il seguente esempio:

```
class Program
{
    static int a, b;
    static double x;
    static void Main()
    {
        a = 10;
        b = 20;
        MetodoQualsiasi();
        Console.WriteLine("x = {0}", x);  // sarà visualizzato: x = 200,0
    }
    static void MetodoQualsiasi()
    {
        x = a * b;
    }
}
```

In questo codice, qualsiasi riferimento alle variabili `a`, `b`, `x`, rappresenta un riferimento ai tre campi di classe omonimi.[15] I metodi `Main()` e `MetodoQualsiasi()` condividono l'uso delle tre variabili, le quali possono essere modificate in un metodo e utilizzate nell'altro e viceversa.

Il campo d'azione di un campo membro è dunque la classe stessa; si dice anche che il campo ha una **visibilità di classe**, poiché può essere usato in qualunque parte della classe.

E' ora possibile reimplementare l'Esempio 6.1, dichiarando le variabili `temperature`, `tempMedia` e `settimana` come campi membro:

```
class Program
{
    static int settimana;
    static double tempMedia;
    static double[,] temperature;

    static void Main()
    {
        string tmp;
        temperature = new double[4, 7];

        AcquisisciTemperature();

        Console.WriteLine("Inserire n° della settimana [1-4]");
        tmp = Console.ReadLine();
        settimana = Convert.ToInt32(tmp);
```

[15] Esattamente come per i metodi, anche per i campi di classe è necessario usare nella dichiarazione la parola chiave `static`. Ciò non rappresenta un requisito imposto dal linguaggio, ma dipende dall'uso che viene fatto in questa trattazione di classi, metodi e campi di classe.

```
        CalcolaTemperaturaMedia();

        Console.WriteLine("Temperatura media = {0}", tempMedia);
    }

    static void AcquisisciTemperature()
    {
        for (int indSettimana = 0; indSettimana < 4; indSettimana++)
        {
            for (int giorno = 0; giorno < 7; giorno++)
            {
                Console.Write("Settimana {0} - Temp. {1}° giorno:",
                              indSettimana+1, giorno+1);
                string tmp = Console.ReadLine();
                temperature[indSettimana, giorno] = Convert.ToDouble(tmp);
            }
        }
    }

    static void CalcolaTemperaturaMedia()
    {
        for (int giorno = 0; giorno < 7; giorno++)
            tempMedia = tempMedia + temperature[settimana–1, giorno];
        tempMedia = tempMedia / 7;
    }
}
```

Esempio 6-2 Programma che calcola la temperatura media settimanale.

6.1 Valori iniziali dei campi di classe

Apparentemente, la nuova implementazione contiene ancora un errore. Infatti, nel metodo `CalcolaTemperaturaMedia()`, oltre ad essere stata rimossa la dichiarazione della variabile `tempMedia` (ora campo membro) è stata cancellata anche l'istruzione `tempMedia = 0`, necessaria perché venga calcolata correttamente la sommatoria delle temperature settimanali. In realtà il programma funziona correttamente e ciò perché:

i campi membro, contrariamente alle variabili locali, sono inizializzati automaticamente al loro valore predefinito.

Ritornando all'Esempio 6.2, il valore iniziale di `tempMedia`, benché non sia specificato in modo esplicito, è quello giusto e cioè zero.

L'esempio relativo a `tempMedia` è appropriato per spiegare l'inizializzazione automatica dei campi di classe, ma rappresenta anche un esempio di cattiva programmazione. Infatti, esaminando il codice del metodo `CalcolaTemperaturaMedia()` non c'è niente che fornisca indizi sul valore iniziale di `tempMedia`. Così facendo, il programmatore produce un codice poco leggibile e assume implicitamente che nessuna modifica futura del programma cambierà il valore iniziale di `tempMedia` prima che venga chiamato `CalcolaTemperaturaMedia()`.

Inizializzazione dei campi di classe

Benché le variabili campi di classe siano inizializzate automaticamente, nulla impedisce di fornire loro un valore iniziale diverso da quello di default. La sintassi da usare è identica a quella usata per inizializzare le variabili locali, come dimostra il seguente codice:

```
class Program
{
    static int a =10;
    static double x = 1;
    static bool flag = true;
    static string parola = "topolino";
    ...
}
```

6.2 Ordine di dichiarazione dei campi dei classe

C# non impone un ordine particolare nella dichiarazione dei campi membro; né impone che la dichiarazione di un campo debba precedere il suo uso, come è necessario che sia per le variabili locali.

Ad esempio, il seguente codice è corretto:

```
class Program
{
    static void Main()
    {
        Console.WriteLine("x: {0}", x);
    }
    static double x = 1.0;
}
```

Benché sembra che la variabile `x` sia usata prima della sua dichiarazione, in realtà, nel momento in cui `Main()` viene eseguito, la variabile `x` è già stata creata e inizializzata.

Vale dunque la regola:

i campi membro vengono allocati in memoria e inizializzati prima che qualsiasi metodo della classe sia eseguito.

7 Campi di classe e variabili locali a confronto

Nel confrontare i campi di classe con le variabili locali c'è da affrontare una questione molto importante: C# consente di dichiarare una variabile locale che ha lo stesso nome di un campo di classe? E se sì, in che modo vengono interpretati i riferimenti a quella variabile?

7.1 Campi di azione di variabili omonime

Si consideri il seguente frammento di codice:

```
class Program
{
    static double x = 10;
```

```
    static void Main()
    {
        double x = 20.0
        Console.WriteLine("x: {0}", x);
    }
}
```

Le domande da porsi su di esso sono due:

- è formalmente corretto dichiarare due volte `x`, sia come campo membro che come variabile locale?
- Se è corretto, cosa viene visualizzato sullo schermo, 10 o 20?

Le risposte sono:

- Sì, è formalmente corretto.
- Viene visualizzato 20.

La questione trova spiegazione nel concetto di campo d'azione. Il linguaggio impone che all'interno di un campo d'azione due riferimenti allo stesso nome siano riferimenti allo stesso oggetto. Per questo motivo, ad esempio, è proibito dichiarare due variabili con lo stesso nome all'interno di un metodo. Ma il linguaggio ammette che all'interno di campi di azione distinti si possa fare riferimento a variabili che abbiano lo stesso nome.

Nel caso in questione, il campo d'azione del campo membro `x` è la classe, mentre il campo di azione della variabile locale `x` è il metodo `Main()`: i due sono campi distinti, anche se il primo comprende il secondo. Lo schema mostrato chiarisce la situazione.

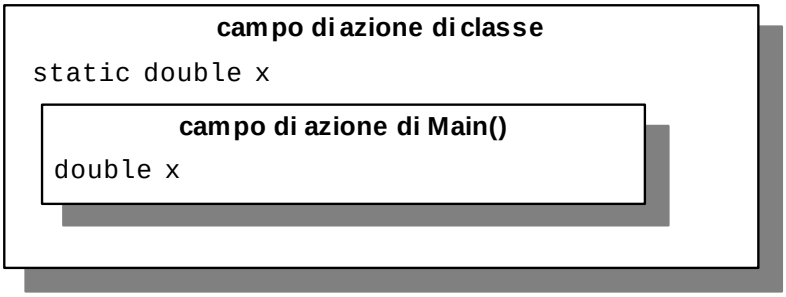


Figura 6-2 Rappresentazione schematica dei campi di azione.

I due campi di azione si sovrappongono nel senso che il campo d'azione di `Main()` è compreso in quello di classe. Poiché in entrambi i campi è dichiarata una variabile di nome `x`, per interpretare un riferimento che è potenzialmente compatibile con due oggetti diversi, C# usa una regola molto semplice:

il campo d'azione locale "nasconde" il campo d'azione di classe.

In sostanza, qualsiasi uso di `x` all'interno di `Main()` si riferisce alla variabile locale e non al campo di classe: la `x` variabile locale "rende invisibile" la `x` campo di classe. Diversamente, al di fuori di `Main()` esiste solo la variabile `x` campo di classe.

Una nuova versione del codice precedente lo dimostra:

```
class Program
{
    static double x = 10.0;
```

```
    static void Main()
    {
        double x = 20.0
        Console.WriteLine("x: {0}", x);
        MetodoQualsiasi();
    }
    static void MetodoQualsiasi()
    {
        Console.WriteLine("x: {0}", x);
    }
}
```

Utilizzando `x`, `MetodoQualsiasi()` fa riferimento al campo di classe e non alla variabile locale dichiarata in `Main()`. Infatti, l'esecuzione del programma produce sullo schermo:

```
x: 20
x: 10
```

All'interno di `MetodoQualsiasi()` non vi è alcun confitto di nomi, poiché esiste un solo oggetto che può essere associato al nome `x`.

8 Terminazione anticipata del metodo

Di norma, l'esecuzione di un metodo termina dopo che è stata eseguita l'ultima istruzione. Questa affermazione è corretta ma incompleta. Il linguaggio mette infatti a disposizione un'istruzione la cui esecuzione determina l'immediata terminazione del metodo: l'istruzione `return`.

8.1 Istruzione return

L'istruzione `return` assume la sintassi generale:

return *espressione-di-ritorno*$_{opz}$;

che per il momento verrà trattata nella forma semplificata:

```
return;
```

Questa istruzione altera il normale flusso di esecuzione del programma, poiché provoca l'uscita immediata dal corpo del metodo, e dunque il ritorno al metodo chiamante. L'esecuzione di un'istruzione `return` collocata all'interno del metodo `Main()` determina la terminazione del programma.

8.2 Uso dell'istruzione return

L'istruzione `return` può essere usata sia allo scopo di realizzare un codice più leggibile e intuitivo, sia per gestire situazioni eccezionali, che necessitano della terminazione immediata del metodo.

Si consideri nuovamente l'Esempio 6.2 e si immagini, all'interno del metodo `CalcolaTemperaturaMedia()`, di voler effettuare un controllo preventivo sulla variabile `settimana`. Il controllo ha lo scopo di verificare che il valore rientri nell'intervallo 1 – 4; se così non è il metodo deve terminare la propria esecuzione, impostando il valore della temperatura media a un valore convenzionale che identifichi uno stato di errore del programma, ad esempio: -1000.

Ecco la nuova implementazione del metodo:

```
static void CalcolaTemperaturaMedia()
{
    if (settimana < 1 || settimana > 4)
    {
        tempMedia = -1000;     // errore: media non calcolabile!
        return;                // fine anticipata del metodo
    }
    double tempMedia = 0;
    for (int giorno = 0; giorno < 7; giorno++)
        tempMedia = tempMedia + temperature[settimana-1, giorno];
    tempMedia = tempMedia / 7;
}
```

Come si intuisce, l'istruzione `return` fa sempre parte di un costrutto di selezione, `if() else`, `if()` o `switch()`. Ciò non rappresenta un requisito formale del linguaggio, infatti non c'è nulla che impedisca di scrivere qualcosa del tipo:

```
static void MetodoQualsiasi()
{
    double x, y = 10;
    ...
    return;
    x = y * 10;       // questa istruzione non verrà mai eseguita!
}
```

Ma in questo caso tutte le istruzioni che seguono `return` non possono essere eseguite. Ciò, pur non rappresentando un errore formale sarà egualmente segnalato dal compilatore mediante un avvertimento che indica l'esistenza di un codice potenzialmente scorretto.

8.3 Terminazioni multiple

Non esiste alcun limite al numero di istruzioni `return` che possono comparire nel corpo di un metodo, e dunque non c'è limite ai "punti" di terminazione del metodo stesso.

In altre parole, è consentito scrivere qualcosa come:

```
static void MetodoQualsiasi()
{
    ...
    if (condizione)
        return;
    ...
    if (condizione)
        return;
    ...
}
```

In linea di massima comunque, un metodo dovrebbe avere uno o al massimo due punti di terminazione; in caso contrario, il codice potrebbe diventare di difficile comprensione e maggiormente soggetto ad errori.

7

Metodi: parametri e valori di ritorno

1 Premessa

I metodi aumentano notevolmente la potenza espressiva di un linguaggio e quindi la capacità applicativa dei programmi, consentendo di implementare singolarmente i vari algoritmi che rappresentano la soluzione di un problema. D'altra parte, l'uso di metodi introduce il problema di articolare correttamente la loro esecuzione e di garantire che le elaborazioni dei dati, ora condivisi tra parti di codice logicamente e fisicamente separate, avvengano nel giusto ordine e nel modo appropriato. Ciò può essere definito un "problema di comunicazione tra i metodi"; ognuno di essi non solo deve funzionare correttamente, ma deve anche coordinare il proprio funzionamento con quello degli altri. In questo senso si possono distinguere due aspetti:

- l'aspetto di **sincronizzazione**, che riguarda il corretto ordine di esecuzione dei metodi;
- l'aspetto di **accesso ai dati condivisi**, il quale implica che la modifica e l'utilizzo delle variabili condivise tra i metodi, i campi di classe, rispettino un "contratto" comune tra gli stessi.

Due esempi relativi all'Esempio 6.2b aiuteranno a comprendere la questione.

Il metodo `CalcolaTemperaturaMedia()` assume che la matrice `temperature` sia già stata creata e riempita con i valori inseriti dall'utente e che sia già stata specificata la settimana da elaborare. Se il metodo, di per sé funzionante, venisse invocato prima del metodo `AcquisisciTemperature()` o del codice che richiede all'utente la settimana da elaborare, produrrebbe un risultato errato.

Sempre nel metodo `CalcolaTemperaturaMedia()` esiste un presupposto che riguarda la settimana da elaborare; e cioè che l'utente specifichi un numero nell'intervallo 1 – 4. Per questo motivo, alla variabile indice `settimana` viene sottratto uno, poiché le righe di una matrice partono dall'indice zero. Qualora il presupposto non venisse rispettato, il metodo provocherebbe l'interruzione del programma.

La strategia ottimale per affrontare il problema della comunicazione tra i metodi sta soprattutto in una attenta progettazione del programma; esiste in ogni caso una regola generale da seguire:

realizzare metodi che siano il più possibile indipendenti gli uni dagli altri.

Ciò si ottiene riducendo al minimo il numero di dati condivisi e facendo sì che il corretto funzionamento di un metodo dipenda dal numero minimo di presupposti possibile. Con il tipo di metodi studiati finora – metodi che non producono valori di ritorno e non accettano argomenti – applicare fino in fondo questa importante regola è semplicemente impossibile.

2 Problema della comunicazione tra metodi

Si immagini che uno dei compiti da svolgere all'interno di un programma sia quello del calcolo della radice ennesima di un valore maggiore di zero. Tale calcolo si ottiene mediante la formula:

$$\sqrt[n]{x} = \exp(1/n \cdot \ln(x))$$

e cioè: la radice ennesima di un numero è uguale all'esponenziale di 1/n per il logaritmo naturale del numero; dove "n" è l'indice della radice.

Si decide di implementare il calcolo attraverso un metodo, che potrà essere invocato ogni qual volta si renda necessario estrarre la radice ennesima da un valore.

Una soluzione potrebbe essere la seguente:

```
class Program
{
    double numeroRad, risultatoRad;
    int indiceRad;
    ...

    static void Main()
    {
        // ... qui ci sono altre istruzioni
        numeroRad = 27;
        indiceRad = 3;
        RadiceEnnesima()
        Console.WriteLine("La radice {0} di {1} è: {3}", indiceRad, numeroRad,
                          risultatoRad);
        ...
    }

    static void RadiceEnnesima ()
    {
        risultatoRad = Math.Exp(1/indiceRad * Math.Ln(numeroRad));
    }
}
```

Esempio 7-1 Programma che calcola la radice ennesima di un valore.

Il programma funziona correttamente e infatti produce sullo schermo:

```
La radice 3 di 27 è: 3
```

L'implementazione del metodo è però del tutto insoddisfacente, per i seguenti motivi:

a) l'invocazione del metodo è estremamente macchinosa; prima di eseguirlo, infatti, occorre assegnare i valori di ingresso alle variabili `numeroRad` e `indiceRad`;

b) il codice risultante è poco comprensibile. Infatti, nell'utilizzo del metodo non è evidente alcun collegamento tra le variabili `indiceRad`, `numeroRad` e `risultatoRad` e il metodo stesso;

3) tra il metodo e il codice chiamante esiste un rigido "contratto" sull'uso delle variabili, che devono rispettare dei nomi ben precisi. Ciò porta a molti svantaggi; ad esempio, se in qualche punto del programma fosse necessario calcolare:

$$y = \sqrt[ind]{x}$$

si dovrebbe scrivere il seguente codice:

```
numeroRad = x;
indiceRad = ind;
RadiceEnnesima();
y = risultatoRad;
```

4) i nomi `indiceRad`, `numeroRad` e `risultatoRad` sono riservati al metodo `RadiceEnnesima()`, e quindi non possono essere usati per altri scopi.

Ci sono troppi limiti; sarebbe desiderabile poter utilizzare il metodo `RadiceEnnesima()` nello stesso modo in cui, ad esempio, si usa il metodo `Sqrt()` della classe `Math`. E cioè poter scrivere, ad esempio:

```
y = RadiceEnnesima(x, ind);
```

oppure:

```
numero = RadiceEnnesima(27, 3);
```

La soluzione sta nell'utilizzo di un diverso modello di comunicazione tra i metodi, che non fa uso di campi membro, ma di **parametri**, **argomenti** e **valori di ritorno**.

3 Parametri e argomenti di ingresso

L'uso di parametri consente di definire metodi la cui realizzazione debba rispettare i soli requisiti dell'algoritmo implementato e non dipenda da altri presupposti stabiliti nel programma. Il prototipo di un metodo che definisce dei parametri assume la seguente sintassi:

```
void nome-metodo (lista-parametri)
```

dove *`lista-parametri`* è definita nel seguente modo:

`tipo param`$_1$*`, tipo param`*$_2$*`, ... tipo`* `param`$_n$

e in pratica ricalca la sintassi di dichiarazione delle variabili, con due differenze:

1) non sono ammesse dichiarazione multiple, ad esempio: `int a, b;`

2) non possono essere specificati dei valori iniziali.

3.1 Invocazione di un metodo con parametri

L'invocazione di un metodo con parametri deve rispettare la seguente regola:

la lista degli argomenti specificati nell'invocazione deve corrispondere nel numero e nel tipo alla lista dei parametri definiti nel prototipo del metodo.

La chiamata assume la seguente sintassi:

nome-metodo (*lista-argomenti*)

dove la `lista-argomenti` definisce una lista di espressioni separate da vigola. Durante la chiamata, il valore di ogni argomento viene assegnato al parametro corrispondente.

Si consideri il seguente programma; oltre a `Main()` definisce il metodo `MetodoQualsiasi()`, il quale dichiara due parametri `int`. `Main()` contiene un'istruzione di invocazione a `MetodoQualsiasi()`, nella quale vengono passati due parametri: la variabile `z` e la costante 3.

```
class Program
{
    static void MetodoQualsiasi(int a, int b)
    {
        ...
    }
    static void Main()
    {
        int z = 1;
        MetodoQualsiasi(z, 3);
    }
}
```

Lo schema mostratoi di seguito riassume cosa avviene durante l'istruzione di chiamata al metodo:

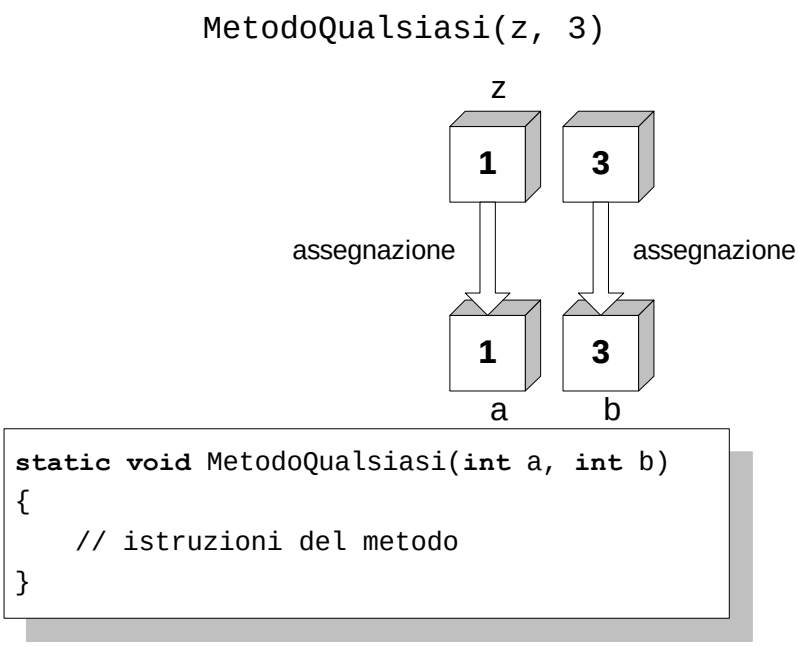


Figura 7-1 Schematizzazione del passaggio degli argomenti ai corrispondenti parametri.

Nella comunicazione tra metodo chiamante e metodo chiamato, i parametri consentono al metodo chiamante di comunicare – fornire in ingresso – i dati al metodo chiamato; per questo motivo sono convenzionalmente chiamati **parametri di ingresso**. Come vedremo più avanti, esistono parametri che svolgono una funzione diversa, per i quali il linguaggio impone l'uso di parole chiave appropriate.

3.2 Compatibilità tra argomenti e parametri

Nella chiamata di un metodo è richiesta la corrispondenza tra la lista degli argomenti e la lista dei parametri definiti dal metodo stesso. Il tipo di corrispondenza richiesta tra è identica a quella necessaria tra l'espressione e la variabile in un'istruzione di assegnazione. Infatti, come è stato già detto, l'invocazione del metodo determina l'assegnazione degli argomenti ai corrispondenti parametri. Tale assegnazione viene anche definita **passaggio per valore degli argomenti**, poiché al parametro viene appunto assegnato il valore dell'argomento.

Ad esempio, nel seguente codice:

```
class Program
{
    static void MetodoQualsiasi(int a, double x, string s)
    {
        ...
    }
    static void Main()
    {
        double z = 10;
        string tmp = "prova";
        MetodoQualsiasi(10, z, tmp);
    }
}
```

l'invocazione di `MetodoQualsiasi()` viene tradotta in:

```
int a = 10;
double x = z;
string s = tmp;
esecuzione del corpo del metodo
```

Poiché gli argomenti vengono assegnati ai parametri, la seguente invocazione è corretta:

```
MetodoQualsisi(10, 10, "10");
```

Essa viene tradotta in:

```
int a = 10;
double x = 10;        //ok: 10 viene convertito in 10.0
string s = "10";
esecuzione del corpo del metodo
```

Vengono cioè applicate le stesse regole di conversione applicate nelle assegnazioni quando il tipo dell'espressione non coincide con il tipo della variabile.

Per lo stesso motivo è invece sbagliato scrivere:

```
MetodoQualsisi(10.0, 10, 10);
```

Poiché viene tradotto in:

```
int a = 10.0;     // errore: conversione da double a int!
double x = 10;    // ok: 10 viene convertito in 10.0
string s = 10;    // errore: conversione da int a string!
esecuzione del corpo del metodo
```

Non esiste una conversione implicita dal tipo `double` al tipo `int`, né è ammessa una conversione da `int` a `string` nell'assegnazione (si ricordi che tale conversione è invece ammessa nell'operatore di concatenazione di stringhe).

3.3 Parametri di ingresso e variabili locali di un metodo

Nel corpo del metodo, i parametri sono a tutti gli effetti considerati delle variabili locali, con due differenze fondamentali:

1) nella dichiarazione di un parametro non può essere specificato un valore iniziale;

2) un parametro di ingresso viene considerato dal linguaggio come inizialmente definito e quindi può essere utilizzato senza che sia stato oggetto di un'assegnazione.

Entrambe le regole sono una ovvia conseguenza della funzione svolta dai parametri di ingresso e cioè quella di ricevere il valore degli argomenti specificati nell'invocazione del metodo.

E' dunque formalmente sbagliato scrivere:

```
static void MetodoQualsiasi(int a = 10)  // errore!
{
    ...
}
```

poiché il parametro a riceverà il proprio valore durante l'invocazione del metodo.

E' invece corretto scrivere:

```
static void MetodoQualsiasi(int a)
{
    int b = a;    // ok: a viene considerata definita
    ...
}
```

perché il meccanismo di chiamata del metodo garantisce che ad a sia assegnato un valore prima che il corpo del metodo venga eseguito.

Conflitto di nomi tra parametri e variabili locali

Essendo i parametri considerati come delle variabili locali al metodo, non sono ammessi due o più parametri con lo stesso nome, come non sono ammessi parametri con lo stesso nome di variabili locali. Il seguente codice, ad esempio, è scorretto:

```
static void MetodoQualsiasi(double x)
{
    int x;     // errore: due variabili x!
    int a;
    ...
}
```

3.4 Modifica dei parametri di ingresso

Essendo i parametri di ingresso sostanzialmente delle variabili locali al metodo,

la modifica del loro valore non influenza affatto il valore dell'argomento corrispondente.

La comprensione di questa regola aiuta a comprendere anche situazioni potenzialmente ambigue come la seguente:

```
class Program
{
    static void MetodoQualsiasi (int a)
    {
        a = 10;
        ...
    }

    static void Main()
    {
        int a = 1;
        MetodoQualsiasi(a);
        Console.WriteLine(a);
    }
}
```

Nell'esempio, `MetodoQualsiasi()` definisce un parametro di nome `a`. `Main()` definisce a sua volta una variabile locale con lo stesso nome, che usa come argomento nell'invocazione di `MetodoQualsiasi()`. Esiste dunque un conflitto di nomi? E se no, l'assegnazione del valore 10 al parametro `a` influenza il valore della variabile `a` dichiarata nel metodo `Main()`?

Le risposte sono:

a) no, non esiste alcun conflitto di nomi, poiché il parametro `a` di `MetodoQualsiasi()` e la variabile locale `a` di `Main()` appartengono a campi di azioni distinti;

b) no, la modifica del parametro `a` non determina alcuna modifica alla variabile locale `a` di `Main()`, poiché le due sono a tutti gli effetti variabili distinte.

Dunque, non esiste alcun legame particolare tra i parametri di ingresso di un metodo e gli argomenti specificati nella sua invocazione, se non che i tipi di ogni coppia argomento-parametro devono rispettare le regole di compatibilità per l'assegnazione. E' quindi del tutto irrilevante che il nome di un argomento coincida con il nome del parametro corrispondente, esattamente com'è irrilevante che, ad esempio, variabili locali di metodi distinti abbiano lo stesso nome.

4 Metodi che ritornano un valore

Negli esempi presentati finora, la comunicazione tra metodo chiamante `Main()` e metodi chiamati avviene in una sola direzione, tramite i parametri di ingresso: `Main()` invoca i metodi passando loro gli argomenti necessari; dopodiché questi svolgono il proprio compito, ritrasferendo infine il controllo a `Main()` senza che vi sia altro scambio di informazioni.

In moltissimi casi esiste la necessità di una comunicazione in entrambe le direzioni. E' ciò che avviene, per esempio, invocando il metodo `Sqrt()`: all'atto dell'invocazione il metodo riceve come argomento il valore dal quale estrarre la radice quadrata; al termine della propria esecuzione, `Sqrt()` ritorna al metodo chiamante il risultato della propria elaborazione.

Si consente al metodo chiamato di ritornare un valore al metodo chiamante facendo seguire all'istruzione di terminazione `return` l'espressione da ritornare:

```
return espressione
```

In questo caso, però, il prototipo del metodo deve definire un **tipo di ritorno** al posto della parola chiave **void** (che ha appunto il significato di "nessun valore ritornato"):

tipo-ritorno nome-metodo (*lista-parametri*$_{opz}$)

Il tipo di ritorno ed il tipo del valore effettivamente ritornato devono essere compatibili secondo le stesse regole applicate all'assegnazione.

4.1 Esempi di metodi che ritornano un valore

Riconsideriamo nuovamente l'Esempio 7.1, nel quale si richiede di realizzare un metodo che calcoli la radice ennesima di un valore, secondo la formula:

$$\sqrt[n]{x} = \exp(1/n \cdot \ln(x))$$

La possibilità di implementaree un metodo con parametri e un valore di ritorno consente di fornire una soluzione molto più elegante e facilmente utilizzabile.

```
class Program
{
    // qui sono dichiarare altre variabili necessarie al programma

    static void Main()
    {
        ...
        double x = RadiceEnnesima(27, 3)
        Console.WriteLine("La radice 3 di 27 è: {0}", x);
        ...
    }

    static double RadiceEnnesima (double radicando, int indiceRad)
    {
        return Math.Exp(1/indiceRad * Math.Ln(radicando));
    }
}
```

Il prototipo del metodo:

```
double RadiceEnnesima(double radicando, int indiceRad)
```

definisce due parametri di ingresso e il tipo dell'espressione di ritorno. Il corpo del metodo:

```
{
  return Math.Exp(1/indiceRad * Math.Ln(radicando));
}
```

contiene un'unica istruzione che calcola il valore della radice e lo ritorna al metodo chiamante. Il metodo `Main()` contiene una chiamata a RadiceEnnesima();quando quest'ultimo ritorna per effetto dell'istruzione, avviene quanto segue:

1) l'espressione `Math.Exp(1/indiceRad * Math.Ln(radicando))` viene valutata;

2) il valore ottenuto, se necessario, viene convertito nel tipo di ritorno specificato nel prototipo del metodo (ma non è questo il caso);

3) il valore viene ritornato al metodo chiamante e utilizzato nell'istruzione di invocazione (in questo caso viene assegnato alla variabile `x`.

L'ultimo punto è molto importante, poiché implica che l'invocazione di un metodo che ritorna un valore è perfettamente equivalente al valore stesso. In termini formali:

ovunque, in una espressione, è ammissibile un valore di tipo <T>, è anche ammissibile l'invocazione di un metodo il cui tipo di ritorno sia <T>.

In alcune situazioni è possibile sfruttare la precedente regola per scrivere un codice più compatto. Ad esempio, finora ogni qual volta si è reso necessario acquisire dall'utente un valore numerico è stato fatto uso di una variabile stringa di supporto, poiché il metodo `ReadLine()` produce un valore stringa, come nel seguente esempio:

```
string tmp = Console.ReadLine();
double x = Convert.ToDouble(tmp)
```

Ma è possibile passare direttamente il valore prodotto da `ReadLine()` come argomento al metodo `ToDouble()` della classe `Convert`:

```
double x = Convert.ToDouble(Console.ReadLine());
```

Il codice risultante è forse meno comprensibile ma certamente più compatto.

4.2 Ignorare il valore di ritorno di un metodo

Se un metodo ritorna un valore, di norma dev'essere invocato in un espressione, poiché il valore ritornato dovrebbe essere utilizzato, o come argomento di un operatore o di un altro metodo, oppure assegnato a una variabile. In realtà questo non rappresenta un requisito; il linguaggio consente che il valore ritornato da un metodo venga semplicemente ignorato, come se il metodo definisse `void` come tipo di ritorno.

Ad esempio:

```
class Program
{
    static void Main()
    {
        int a = Quadrato(10);  // invocazione convenzionale di Quadrato()
        Quadrato(10);          // invocazione non convenzionale di Quadrato()
    }

    static int Quadrato(int a)
    {
        return a*a;
    }
}
```

Di solito, ignorare il valore ritornato da un metodo rappresenta un cattivo esempio di programmazione, o addirittura un un errore, ma esistono alcune situazioni in cui questa scelta è coerente con l'uso che si intende fare del metodo. Eccone un esempio:

```
class Program
{
```

```
    static void Main()
    {
        ...
        Console.ReadLine(); // "stoppa" l'esecuzione del programma
    }
}
```

In questo caso, il metodo `ReadLine()` non viene invocato per acquisire un valore ma al solo scopo di fermare l'esecuzione del programma fino a quando l'utente non preme il tasto INVIO.

5 Passaggio per riferimento degli argomenti: parametri ref e out

La possibilità di utilizzare parametri e/o valore di ritorno estende notevolmente le potenzialità applicative dei metodi, consentendo l'utilizzo di un modello di comunicazione più sicuro, potente e generalizzabile. Tale modello, però, non copre completamente le esigenze della programmazione con i metodi, poiché presenta un limite importante: funziona in una sola direzione soltanto. Tramite i parametri di ingresso il metodo chiamante passa dei valori al metodo chiamato. Tramite il valore di ritorno accade l'inverso. In alcuni scenari, serve un modello di comunicazione più potente, che possa funzionare in entrambe le direzioni..

5.1 Parametri e argomenti ref

I parametri `ref` (*reference*, riferimento) consentono al metodo chiamante e al metodo chiamato di condividere la stessa variabile, pur facendo riferimento a due oggetti apparentemente distinti.

Definire un parametro come `ref` fa sì che qualsiasi modifica effettuata su di esso si rifletta sull'argomento corrispondente. Un parametro `ref` rappresenta dunque un alias dell'argomento: è come se tutte le istruzioni che coinvolgono il parametro, in realtà coinvolgessero l'argomento.

Nella definizione di un metodo è possibile dichiarare uno o più parametri `ref` mediante la seguente sintassi:

tipo-ritorno nome-metodo (**ref** *tipo param*$_1$, **ref** *tipo param*$_2$, ...)

Nell'invocazione del metodo, gli argomenti `ref` corrispondenti devono essere preceduti dall'appropriata parola chiave:

nome-metodo (**ref** *arg*$_1$, **ref** *arg*$_2$, ...)

Ad esempio, il seguente metodo definisce un parametro di ingresso `a` e un parametro `ref b`:

```
void Accumula(double a, ref double b)
{
    b = b + a
}
```

5.2 Invocazione di un metodo che definisce parametri ref

L'invocazione di un metodo che definisce uno o più parametri `ref` rispecchia la consueta sintassi, con la differenza che gli argomenti corrispondenti ai parametri `ref` devono essere prefissati dall'appropriata parola chiave. Ciò che cambia radicalemente è il modello di passaggio dei

parametri. Il seguente codice mostra l'invocazione al metodo `Accumula()` precedentemente definito.

```
static void Main()
{
    double salario = 1000;
    double straordinario = 200;
    Accumula(straordinario, ref salario)
    Console.WriteLine("Salario = {0}", salario);
}
```

Il programma produce come output:

```
Salario = 1200
```

Lo schema in figura mostra cosa avviene durante l'invocazione del metodo:

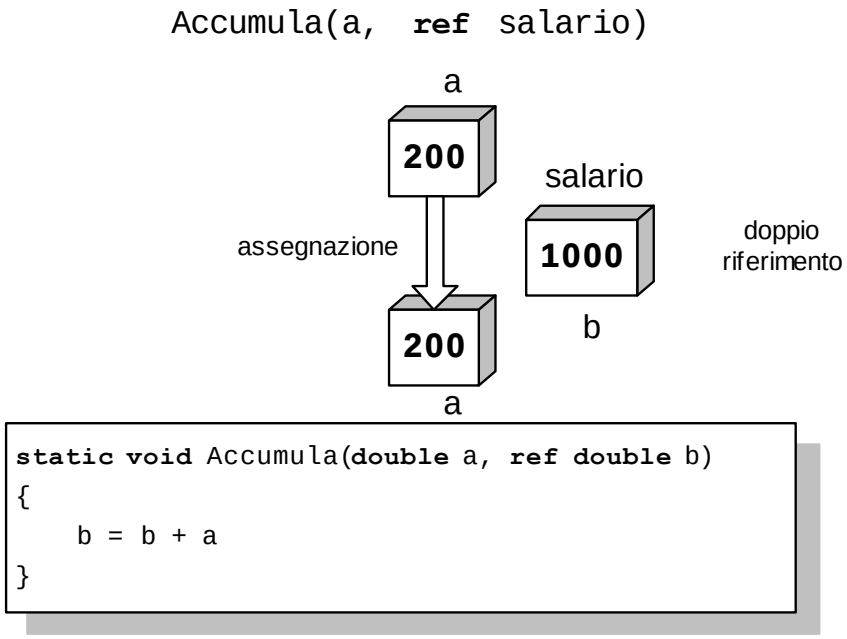


Figura 7-2 Schema del passaggio di un argomento ref a un parametro ref .

L'argomento `a` viene assegnato al parametro `a` (si ricorda che un'eventuale sinonimia tra argomento e parametro è irrilevante). Siamo in presenza di un passaggio per valore, nel quale argomento e parametro rappresentano a tutti gli effetti variabili distinte. Al contrario, il parametro `b` diventa un alias dell'argomento `salario`. Non viene eseguita nessuna assegnazione: `b` e `salario` rappresentano a tutti gli effetti la stessa variabile e dunque referenziano la stessa zona di memoria: siamo dunque in presenza di un **passaggio per riferimento** dell'argomento.

A questo punto si comprende come, nel metodo `Accumula()`, la modifica al parametro `b` si rifletta nella modifica dell'argomento `salario`: parametro `ref` e argomento `ref` corrispondente sono la stessa variabile.

5.3 Requisiti nell'uso di parametri e argomenti ref

Il linguaggio impone dei requisiti sull'uso di parametri `ref`:

- **L'argomento `ref` dev'essere una variabile.**

Poiché un parametro `ref` non riceve un valore ma il riferimento ad una zona di memoria, l'argomento corrispondente deve essere una variabile, considerato che solo le variabili referenziano una zona di memoria.

- **L'argomento `ref` dev'essere dello stesso tipo del parametro `ref` corrispondente.**

Essendo il parametro `ref` un alias dell'argomento, entrambi devono essere dello stesso tipo, poiché è proibito (e insensato) considerare lo stesso oggetto contemporaneamente appartenente a due tipi diversi.

- **E' necessario che un argomento `ref` si trovi nello stato definito.**

La funzione di un parametro `ref` è quella fornire un alias ad una variabile esistente. Per evitare venga usato un parametro ancora privo di valore, il linguaggio richiede che l'argomento `ref` abbia già subito almeno un'assegnazione prima che sia invocato il metodo.

- **Un parametro `ref` viene considerato come inizialmente definito e pertanto non è obbligatorio che subisca un'assegnazione.**

Il linguaggio non impone di assegnare un valore ad un parametro `ref`, poiché l'argomento corrispondente si trova già nello stato definito.

In relazione a questi requisiti, tutt le chiamate a `MetodoQualsiasi()` presenti nel seguente codice sono formalmente errate:

```
class Program
{
    static void MetodoQualsiasi (ref double a)
    {
        a = a + 1;
    }

    static void Main()
    {
        int z= 10;
        double x;

        MetodoQualsiasi1(ref 3); // errore: 3 non è una variabile!
        MetodoQualsiasi1(ref z); // errore: z è del tipo sbagliato!
        MetodoQualsiasi1(ref x); // errore: x non si trova nello stato definito!

    }
}
```

5.4 Uso di argomenti e parametri ref

L'uso di parametri `ref` è appropriato quando il metodo chiamato deve modificare il valore degli argomenti. Come esempio poniamo l'ipotesi di voler realizzare un metodo che consenta di scambiare il contenuto di due variabili di tipo `double`. Ecco l'implementazione

```
class Program
{
    static void Main()
```

```
    {
        double var1 = 4;
        double var2 = 8;

        Scambia (ref var1, ref var2);
        Console.WriteLine("var1: {0}  ;   var2: {1}", var1, var2);
    }

    static void Scambia (ref double x, ref double y)
    {
        double tmp = x;
        x = y;
        y = tmp;
    }
}
```

Nell'invocazione del metodo `Scambia()`, i parametri `ref x` e `y` diventano alias degli argomenti `var1` e `var2` e dunque è proprio tra queste variabili che avviene lo scambio.

5.5 Parametri e argomenti out

I parametri out (*output*, uscita) consentono al metodo chiamato di utilizzare i parametri come mezzo per comunicare dei risultati al metodo chiamante. I parametri `out` usano lo stesso modello di comunicazione dei parametri `ref`: tra argomenti e parametri avviene un **passaggio per riferimento** e dunque un parametro `out` rappresenta un alias dell'argomento `out` corrispondente.

Nella definizione di un metodo è possibile dichiarare uno o più parametri `out` mediante la seguente sintassi:

tipo-ritorno nome-metodo (**out** *tipo* $param_1$, **out** *tipo* $param_2$, ...)

Nell'invocazione del metodo, gli argomenti `out` corrispondenti devono essere preceduti dall'appropriata parola chiave:

nome-metodo (**out** arg_1, **out** arg_2, ...)

5.6 Uso di argomenti e parametri out

Nonostante pametri out e `ref` utilizzino lo stesso meccanismo di passaggio, rivestono un ruolo distinto nella comunicazione tra metodo chiamante e metodo chiamato. I parametri `ref` consentono di condividere una variabile tra i due metodi, i parametri `out` consentono al metodo chiamato di comnunicare dei dati al metodo chiamante: nel secondo caso la comunicazione si svolge in una sola direzione. I parametri `out`, dunque, svolgono la funzione inversa dei parametri di ingresso.

Per comprendere l'utilità dei parametri `out` riconsideriamo il problema della soluzione di un equazione di 2° grado, già affrontato nel Capitolo 2. Si vuole implementare la soluzione mediante un metodo;il quale dovrà ricevere i coefficienti a,b,c dell'equazione e dovrà produrre la radici x1, x2 che rappresentano la soluzione.

Poiché il metodo deve produrre in uscita due valori non è possibile utilizzare il valore di ritorno come meccanismo di restituzione del risultato: si rende necessario l'uso di parametri `out`.

```
class Program
{
    static void Main()
```

```
    {
        double radice1, radice2;
        double a, b, c;
        // ... qui a, b, c ricevono i loro valori
        EquazioneSecondoGrado(a, b, c, out radice1, out radice2);
        Console.WriteLine("X1 = {0}   X2 =  {1}", radice1, radice2);
    }

    static void EquazioneSecondoGrado (double a, double b, double c,
                                       out double x1, out double x2)
    {
        x1 = (-b - Math.Sqrt(b*b - 4*a*c)) / (2*a);
        x2 = (-b + Math.Sqrt(b*b - 4*a*c)) / (2*a);
    }
}
```

Esempio 7-2 Programma che risolve un'equazione di 2° grado.

Il metodo `EquazioneSecondoGrado()` definisce tre parametri di ingresso, per ricevere i coefficienti, e due parametri `out`, per restituire il risultato. Durante l'invocazione del metodo:

a) le variabili locali `a`, `b`, `c` vengono assegnate ai parametri di ingresso, `a`, `b`, `c`;

b) il parametri `out` `x1` e `x2` assumono il ruolo di alias delle variabili `radice1` e `radice2`;

5.7 Requisiti nell'uso di parametri e argomenti out

Il linguaggio impone dei requisiti anche per l'uso di parametri `out`, due dei quali sono gli stessi dei parametri `ref`:

- **L'argomento `out`dev'essere una variabile.**

Analogamente a quanto avviene con i parametri `ref`, poiché i parametri `out` rappresentano l'alias di una variabile, non è possibile che l'argomento sia un valore.

- **L'argomento `out` dev'essere dello stesso tipo del parametro `out` corrispondente.**

Ovvia conseguenza della regola precedente: parametro e argomento sono di fatto la stessa variabile e dunque non possono appartenere a tipi diversi.

- **Non è necessario che un argomento `out` si trovi nello stato definito.**

La funzione di un parametro `out` è quella di consentire al metodo chiamato di comunicare un risultato al metodo chiamante; il fatto che l'argomento abbia già un valore o meno è del tutto irrilevante.

- **Un parametro `out` viene considerato come inizialmente non definito.**

Questa regola deriva logicamente dalla precedente. Poiché non è possibile fare affidamento sul fatto che l'argomento possieda già un valore, è proibito usare un parametro `out` in un'espressione prima che abbiano subito almeno un'assegnazione.

- **Un parametro `out` deve subire almeno un'assegnazione prima che il metodo termini.**

Lo scopo di un parametro `out` è quello di restituire un valore al metodo chiamante; per questo motivo è necessario che subisca almeno un'assegnazione altrimenti resterebbe nello stato non definito.

5.8 Metodi che definiscono parametri out e valore di ritorno

Sia i parametri `out` che il valore di ritorno di un metodo possono essere impiegati per restituire dei dati al metodo chiamante; ciò non significa, però, che un meccanismo debba escludere l'altro. Esistono casi nei quali l'uso di entrambe le modalità di comunicazione consente di gestire in modo ideal situazioni particolari.

Si riconsideri nuovamente il metodo che risolve l'equazione di 2° grado. L'attuale implementazione non verifica l'ipotesi di un delta dell'equazione negativo; il problema non è implementare la verifica, ma stabilire un modo per comunicare il suo esito al metodo chiamante.

In sostanza, il metodo `EquazioneSecondoGrado()` dovrebbe restituire tre valori: il primo indica il successo o meno dell'operazione e gli altri due rappresentano le radici dell'equazione. Una soluzione potrebbe essere quella di definire tre parametri `out`, due per le radici e il terzo per comunicare lo stato dell'operazione. Sarebbe una soluzione accettabile, ma poco elegante. La scelta migliore è quella di tenere distinte la modalità di comunicazione del successo dell'operazione con quella di restituzione delle radici: la prima realizzata mediante il tipo di ritorno, la seconda mediante due parametri `out`.

```
class Program
{
    static void Main()
    {
        double radice1, radice2;
        double a, b, c;
        bool ok;
        // ... qui a, b, c ricevono i loro valori
        ok = EquazioneSecondoGrado (a, b, c, out radice1, out radice2);
        if (ok == true)
            Console.WriteLine("X1 = {0}   X2 =  {1}", radice1, radice2);
        else
            Console.WriteLine("Radici non calcolabili");
        }
    }

    static bool EquazioneSecondoGrado (double a, double b, double c,
                                            out double x1, out double x2)
    {
        double delta = b*b - 4*a*c;
        if (delta < 0)
        {
            x1 = 0;      // è necessario assegnare un valore ai parametri out
            x2 = 0;
            return false;
        }
        x1 = -b - Math.Sqrt(delta) / (2*a);
        x2 = -b + Math.Sqrt(delta) / (2*a);
        return true;
```

```
    }

}
```

Esempio 7-3 Programma che risolve un'equazione di 2° grado.

Il metodo `EquazioneSecondoGrado()` ritorna un valore `bool` per indicare il successo o meno del calcolo delle radici, che vengono restituite mediante i parametri `x1` e `x2`.

> La soluzione è valida ma non ottimale, poiché nulla impedisce al metodo chiamante di ignorare il valore di ritorno e utilizzare comunque le due radici a prescindere dalla loro validità. Esistono due approcci distinti per risolvere questo problema, i quali richiedono però elementi del linguaggio che non sono ancora stati affrontati.

5.9 parametri ref e parametri out a confronto

Parametri `ref` e `out` usano lo stesso modello di comunicazione e quindi possono essere usati in modo intercambiabile. Resta il fatto che l'uso dei due tipi di parametri è soggetto requisiti diversi, poiché diversi sono i ruoli che svolgono nella comunicazione tra metodi.

Per comprendere meglio il concetto riconsideriamo il metodo di soluzione dell'equazione di 2° grado. Attualmente `EquazioneSecondoGrado()` definisce due parametric `out` per restituire le due radici, ma funzionerebbe correttamente anche se utilizzasse parametri `ref`.

```
static bool EquazioneSecondoGrado (double a, double b, double c,
                                   ref double x1, ref double x2)
{
    double delta = b*b - 4*a*c;
    if (delta < 0)
    {
        //x1 = 0;     queste istruzioni non sono più necessarie!
        //x2 = 0;
        return false;
    }
    x1 = -b - Math.Sqrt(delta) / (2*a);
    x2 = -b + Math.Sqrt(delta) / (2*a);
    return true;
}
```

Nonostante ciò, questa versione del metodo porterebbe con se diversi problemi.

Innanzitutto vi è un problema di leggibilità del codice: i parametri `x1` e `x2` hanno lo scopo di restituire dei risultati al metodo chiamante, ma essendo dichiarati `ref` sembra che debbano possedere un valore prima dell'invocazione, come se ciò fosse necessario al buon funzionamento del metodo. In sostanza, così facendo si fornisce un indizio fuorviante sul ruolo svolto dai paremetri.

Collegato a questo esiste un problema pratico: il metodo chiamante è costretto a valorizzare gli argomenti `ref` prima di invocare `EquazioneSecondoGrado()`, benché ciò non sia affatto necessario per il corretto funzionamento di quest'ultimo.

Infine, utilizzare parametri `ref` aumenta la possibilità di introdurre insidiosi bug nel codice. Ad esempio, la versione sotto riportata del metodo `()` contiene un errore, poiché non calcola la prima radice.

```
static bool EquazioneSecondoGrado (double a, double b, double c,
                                   ref double x1, ref double x2)
```

```
{
    double delta = b*b - 4*a*c;
    if (delta < 0)
        return false;

    // qui manca il calcolo della prima radice!
    x2 = -b + Math.Sqrt(delta) / (2*a);
    return true;
}
```

Purtroppo il compilatore non è in grado di rilevare l'errore, poiché non esiste un requisito che imponga di assegnare un valore a un parametro `ref`. Ma un requisito simile esiste per i parametri `out`!

In conclusione, per stabilire se definire un parametro come `out` o `ref` ci si deve interrogare sul ruolo che svolge nella comunicazione tra metodo chiamante e metodo chiamato. Se il parametro ha soltanto la funzione di restituire un valore, deve essere dichiarato `out`, altrimenti `ref`.

6 Array come argomenti di un metodo

Qualsiasi tipo di variabile può apparire nella lista dei parametri di un metodo; alcuni tipi, però, per loro natura si comportano diversamente dagli altri. E' questo il caso degli array.

La differenza di comportamento, ancora una volta, sta nella distinzione che esiste tra variabile array e oggetto array, distinzione che non c'è in una variabile semplice. Nei paragrafi che seguono, l'argomento sarà trattato facendo principalmente riferimento agli array unidimensionali; resta sottinteso che l'uso di parametri array di più dimensioni non comporta alcuna differenza. Inoltre, la trattazione sarà limitata all'esame di parametri array di ingresso, poiché è questo il modello di comunicazione più usato nel caso di parametri array.

6.1 Dichiarazione di un parametro array

La definizione di un parametro array assume l'identica sintassi della dichiarazione di una variabile array:

tipo`[]` *nome-parametro*

Ad esempio, il metodo che segue definisce come parametro un vettore di tipo `int`:

```
static void MetodoQualsiasi(int[] v)
{
    ...
}
```

6.2 Uso di parametri array

Diversamente dagli array dichiarati come variabili locali o campi di classe, un parametro array non necessita di essere creato, poiché fa già riferimento ad un array esistente, rappresentato dall'argomento. Chiariamo il concetto con un esempio.

Dato un vettore contenente le altezze in centimetri di un numero arbitrario di persone, si vuole calcolare l'altezza media delle stesse.

Si decide di implementare il calcolo dell'altezza media attraverso un metodo:

```
class Program
{
    static void Main()
    {
        double altezzaMedia;
        int numPersone = Convert.ToInt32(Console.ReadLine());
        double[] altezze = new double[numPersone];

        // ... qui vengono inserite nel vettore le altezze delle persone

        altezzaMedia = CalcolaAltezzaMedia(altezze, numPersone);
        Console.WriteLine("L'altezza media è: {0}", altezzaMedia)
    }

    static double CalcolaAltezzaMedia (double[] v, int n)
    {
        double media = 0;
        for (int i = 0; i < n; i++)
            media = media + v[i];
        return media / n;
    }
}
```

Nel metodo `CalcolaAltezzaMedia()`, alla definizione del parametro array `v` non segue la creazione dell'oggetto array corrispondente. Infatti, lo scopo del parametro è quello di referenziare un array già esistente, che viene passato come argomento nel momento dell'invocazione del metodo. Ovviamente, è necessario che l'argomento (`altezze`, in questo caso) sia già stato creato in prima di effettuare la chiamata al metodo.

Consideriamo adesso l'istruzione di invocazione:

```
altezzaMedia = CalcolaAltezzaMedia(altezze, numPersone);
```

Questa viene tradotta in:

```
int[] v = altezze;      // assegnazione tra variabili array
int n = numPersone   ;
esecuzione del corpo del metodo
```

L'assegnazione dell'argomento `altezze` al parametro `v` produce in memoria la situazione mostrata in figura, nella quale il parametro `v` referenzia lo stesso oggetto array referenziato dalla variabile array `altezze`:

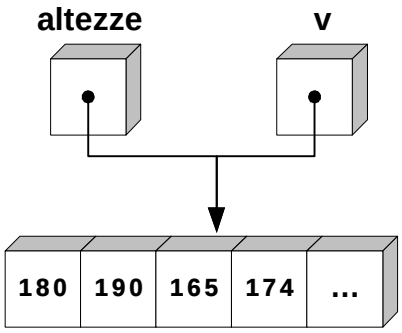


Figura 7-3 Rappresentazione schematica del passaggio tra argomento e parametro array.

In conclusione, attraverso il parametro v diventa possibile accedere agli elementi dell'argomento `altezze`, nonostante questo sia stato creato al di fuori del metodo.

6.3 Modifica degli elementi di un parametro array

Dato che parametro e argomento corrispondente referenziano lo stesso oggetto, ne consegue che attraverso un parametro array è possibile modificare gli elementi di un array creato fuori dal metodo.

Facciamo un esempio: si vuole realizzare un metodo che esegua la moltiplicazione di un vettore per un numero. Il metodo viene così implementato:

```
class Program
{
    static void Main()
    {
        double[] valori = {1, 2, 3};
        MoltiplicaPerNumero(valori, 10, 3);
        foreach (double valore in valori)
            Console.WriteLine(valore);
    }

    static void MoltiplicaPerNumero (double[] v, double numero, int numValori)
    {
        for (int i = 0; i < numValori; i++)
            v[i] = v[i] * numero;
    }
}
```

Il precedente programma produce come risultati:

```
10

20

30
```

e dimostra che il vettore, creato in `Main()`, è stato modificato all'interno del metodo `MoltiplicaPerNumero()`. Apparentemente, l'esempio sembra violare la regola che dice che la modifica di un parametro di input non si riflette nel corrispondente argomento. In realtà la regola è salva; infatti, arraverso un parametro è possibile modificare gli elementi dell'array passato come argomento, ma non l'argomento stesso!

Il seguente codice lo dimostra:

```
class Program
{
    static void Main()
    {
        double[] valori = {10, -2};
        MetodoQualsiasi (valori);
        foreach (double valore in valori)
            Console.WriteLine(valore);
    }

    static void MetodoQualsiasi (double[] v)
    {
        v = new double[5] {10, 20, 30, 40, 50}
    }
}
```

`MetodoQualsiasi()` non si limita a modificare gli elementi di `v` ma crea un nuovo array e lo associa al parametro. Ebbene, tale modifica non influisce affatto sulla variabile `valori`, che rappresenta l'argomento corrispondente. Infatti, l'esecuzione del programma produce sullo schermo:

```
10
-2
```

e dimostra che assegnare un nuovo oggetto ad un parametro array non produce alcuna conseguenza sull'argomento corrispondente.

6.4 Compatibilità tra argomenti e parametri

Anche nell'ambito della compatibilità tra tipo degli argomenti e tipo dei parametri, il tipo array rispetta i requisiti del modello di comunicazione di passaggio per valore, viene cioè applicata la stessa regola usata per l'assegnazione. Si ricorda, pero, che per il tipo array, tale regola richiede l'identità dei tipi e non la semplice compatibilità.

Il seguente codice mostra un esempio nel quale tale regola non viene rispettata:

```
class Program
{
    static void Main()
    {
        int[] valori = {10, -2};
        MetodoQualsiasi (valori);      // errore: array int e array double non
                                           sono compatibili!
        ...
    }
    static void MetodoQualsiasi (double[] v)
    {
        ...
    }
}
```

8

Approfondimento sui tipi di dati

Lo scopo di questo capitolo è quello di affrontare nuovamente i concetti di tipo, oggetto e valore, secondo una prospettiva più fedele alla realtà del linguaggio C#.

Vedremo che il linguaggio C# non concepisce l'idea di tipo solo come qualcosa che designa le caratteristiche dell'informazione, ma anche che definisce dei metodi, delle costanti e delle **proprietà** che supportano il programmatore nella elaborazione di oggetti del tipo in questione.

Approfondiremo l'argomento sul modello di memoria impiegato dal linguaggio per l'effettiva implementazione. Se pur soltanto in modo introduttivo, è già stato sottolineato come, ad esempio, il tipo array sia gestito in modo diverso dai tipi predefiniti `int`, `double` e `bool`, poiché le variabili array non memorizzano affatto un array, ma un riferimento ad esso.

Verrà introdotto il tipo `object`, che fa da denominatore comune a tutti i tipi di dati definiti dal linguaggio. Il suo impiego si rende necessario in quelle situazioni nelle quali non è possibile definire a priori il tipo dell'informazione da elaborare e dunque l'implementazione dell'algoritmo dev'essere compatibile con più tipi di dati.

Infine, saranno introdotti i tipi generici, o *generics*, i quali forniscono un potente meccanismo per la implementazione di algoritmi indipendenti dal tipo dei dati che vengono elaborati. I tipi generici sono ampiamente utilizzati nell'implementazione di collezioni di dati, argomento della terza parte del volume.

1 Metodi e costanti dei tipi predefiniti

Ogni tipo di dato definisce dei metodi che aumentano la potenza espressiva e le capacità applicative del linguaggio. Saranno esaminati i due metodi messi a disposizione da tutti i tipi semplici, `int`, `double`, `bool`.

1.1 Costanti simboliche definite dal tipo int

Il tipo `int` definisce due costanti simboliche alle quali si può accedere premettendo ad esse il nome del tipo:

`int.`*`nome-costante`*

Le due costanti sono `MinValue` e `MaxValue` e rappresentano rispettivamente il più piccolo e il più grande valore intero.

Ad esempio, se ad una variabile è necessario assegnare un valore che sia sicuramente più piccolo di un insieme di valori interi si può scrivere:

```
int minimo = int.MinValue;
```

Discorso analogo vale per `MaxValue`.

1.2 Costanti simboliche definite dal tipo double

Anche il tipo `double` definisce delle costanti simboliche, tra cue `MinValue` e `MaxValue` che hanno lo stessa funzione delle omologhe del tipo `int`.

Seguono le altre costanti.

Costante epsilon

`Epsilon` rappresenta il più piccolo valore positivo rappresentabile. `Epsilon` riveste un ruolo importante quando diventa necessario eseguire un confronto di uguaglianza tra due valori `double`. Per capirne il motivo si consideri il seguente codice:

```
double x1, x2;
x1 = 10 * 3 / 3;      // risultato 10
x2 = 10 / 3 * 3;      // risultato ancora 10?
if (x1 == x2)
    Console.WriteLine("x1 e x2 sono uguali");
else
    Console.WriteLine("x1 e x2 sono diversi");
```

Eseguendolo ci si aspetta ovviamente di ottenere:

```
x1 e x2 sono uguali
```

ma non è affatto così. Il problema dipende dal fatto che il tipo `double` rappresenta un'approssimazione del tipo reale. Ciò significa che alcuni calcoli possono produrre dei risultati approssimati e dunque inesatti. Nell'esempio, il valore dell'espressione `10/3*3` è soltanto "approssimativamente" uguale al valore dell'espressione `10*3/3`, e ciò determina il fallimento del confronto per uguaglianza. La costante `Epsilon` ci aiuta a risolvere il problema:

```
double x1, x2;
x1 = 10 * 3 / 3;
x2 = 10 / 3 * 3;
if (Math.Abs(x1 - x2) <= double.Epsilon)
    Console.WriteLine("x1 e x2 sono uguali");
else
    Console.WriteLine("x1 e x2 sono diversi");
```

L'espressione booleana:

```
Math.Abs(x1 - x2) <= double.Epsilon
```

verifica se la differenza in valore assoluto tra `x1` e `x2` è minore o uguale a `Epsilon`. Se ciò è vero (ed è il caso dell'esempio), significa che `x1` e `x2`, pur non essendolo, possono essere considerati uguali.

> Ciò detto, laddove possibile, in una espressione `double` è preferibile mettere prima le moltiplicazioni e poi le divisioni.

Costanti NegativeInfinity e PositiveInfinity

Queste costanti equivalgono a valori che si ottengono quando un'espressione `double` produce un valore infinito, negativo o positivo.

Ad esempio, il seguente codice produce un infinito positivo:

```
double x = 0;
double y = 10 / x;   // equivale a PositiveInfinity
```

Costante NaN

La costante `NaN` designa un valore `double` non valido, frutto di un'espressione il cui valore non può essere calcolato. Un esempio classico è il seguente:

```
double y = Math.Sqrt(-2);     // produce un NaN
```

La costante NaN può essere utile per impostare un modo esplicito il valore di una variabile qualora sia il frutto di un calcolo che non ppuò essere eseguito.

A tal proposito riconsideriamo il metodo `EquazioneSecondoGrado()` dell'Esempio 7.3. L'attuale versione non è l'ideale, poiché in caso di delta negativo imposta arbitrariamente a zero i valori delle radici. Se nel metodo chiamante ci si dimentica di verificare il valore di ritorno di `EquazioneSecondoGrado()`, sarà impossibile stabilire se eventuali radici uguali a zero siano il risultato atteso o siano il frutto di un'equazione impossibile. L'approccio corretto è utilizzare la costante `NaN`:

```
static bool EquazioneSecondoGrado (double a, double b, double c,
                                   out double x1, out double x2)
{
    double delta = b*b - 4*a*c;
    if (delta < 0)
    {
        x1 = double.NaN; // è necessario assegnare un valore ai parametri out
        x2 = double.NaN;
        return false;
    }
    x1 = -b - Math.Sqrt(delta) / (2*a);
    x2 = -b + Math.Sqrt(delta) / (2*a);
    return true;
}
```

Dopo questa modifica è impossibile che il metodo chiamante interpreti scorrettamente il valore delle radici.

> `NaN`, `PositiveInfinity` e `NegativeInfinity` sono utili poiché i calcoli con valori `double` non provocano mai un eccezione e dunque è confrontando il valore di un'espressione double con queste costanti che si è in grado di stabilire se il risultato di un calcolo è valido oppure no. (Vedi il paragrafo 1.6.)

1.3 Metodi statici e metodi di istanza

Parlando di metodi esposti da un tipo di dato si rende necessario distinguere tra metodi che sono definiti **statici** e metodi che sono definiti **di istanza**, poiché diversa è la modalità del loro impiego. Un metodo statico è accessibile soltanto attraverso l'identificatore di tipo che lo definisce; l'invocazione di un metodo statico assume sempre la forma:

$$nome\text{-}tipo.nome\text{-}metodo(lista\text{-}argomenti_{opz})$$

I metodi statici sono analoghi a tutti i metodi che abbiamo utilizzato finora, appartenenti alla classi `Console`, `Convert` e `Math`.

Un metodo di istanza, d'altro canto, è accessibile soltanto attraverso un oggetto (istanza) di un tipo. La sua invocazione assume la forma:

oggetto.*nome-metodo*(*lista-argomenti*$_{opz}$)

Nota bene che è stato utilizzato il termine oggetto e non variabile; infatti i metodi di istanza possono essere usati anche attraverso una costante e in generale una qualsiasi espressione del tipo che definisce il metodo.

1.4 Metodo statico Parse()

Il metodo statico `Parse()` è definito da tutti i tipi semplici e svolge una funzione analoga a quella svolta dai metodi `ToXXX()` della classe `Convert`: converte un dato espresso in forma di stringa nell'equivalente valore definito dal tipo attraverso il quale il metodo viene invocato. L'invocazione assume la sintassi:

tipo.Parse(*stringa*)

Ad esempio, il seguente codice:

```
int a = int.Parse("10");            // converte la stringa "10" in 10
double x = double.Parse("20,10");   // converte la stringa "20,10" in 20.1
```

equivale a:

```
int a = Convert.ToInt32("10");
double x = Convert.ToDouble("20,1");
```

Nel caso in cui l'argomento stringa non rappresenti una valore ammissibile per il tipo utilizzato, l'esecuzione del programma viene interrotta da un'eccezione. Ciò è ad esempio il caso di:

```
int a = int.Parse("10A");     // errore di esecuzione!
```

1.5 Metodo statico TryParse()

Il metodo `TryParse()` ha una funzione del tutto analoga al metodo `Parse()`, ma con un'importante differenza. L'invocazione del metodo assume la seguente sintassi:

tipo.TryParse(*stringa, out risultato)*)

Il metodo tenta di convertire il primo argomento nel tipo utilizzato, memorizzando il valore risultante nel secondo argomento. Se le conversione riesce, il metodo ritorna `true`, altrimenti ritorna `false`.

Diversamente dal modo di operare del metodo `Parse()`, `TryParse()` non produce mai un errore di esecuzione e utilizza il valore di ritorno per stabilire la buona riuscita o meno della conversione.

`TryParse()` è molto utile nella conversione dei valori inseriti dall'utente, poiché consente facilmente di verificare la loro aderenza al tipo richiesto. Ad esempio, il seguente frammento di codice richiede in input un valore intero, rifutando il valore inserito fintantoché non rispetta il vincolo suddetto.

```
...
int altezza;
bool inputOk;
do
{

    Console.WriteLine("Inserisci l'altezza: ");
```

```
    string tmp = Console.ReadLine();
    inputOk = int.TryParse(tmp, out altezza);
}
while(inputOk == false);
```

1.6 Metodi statici che verificano la validità di un valore double

Come abbiamo visto nel paragrafo 1.2, una espressione `double` può produrre un valore non valido, identificato da una tra le costanti `NaN`, `PositiveInfinity` e `NegativeInfiny`. Il problema è che non si può semplicemente confrontare un valore double con queste costanti per sapere se è valido oppure no: il confronto darebbe comunque esito negativo. Per questo motivo il tipo `double` definisce quattro metodi statici che consentono di stabilire la validità di un valore: `IsNaN()`, `IsInfinity()`, `IsPositiveInfinity()` e `IsNegativeInfinity()`. Tutti accettano come argomento una valore `double` e ritornano un `valore` bool che indica se l'argomento appartiene alla categoria designata dal nome del metodo.

Ad esempio, il seguente codice mostra come testare il risultato di una divisione:

```
double x, y;
//... qui x riceve il proprio valore
y = 10 / x;
if (double.IsInfinity(y))
    Console.WriteLine("Divisione per zero");
```

1.7 Metodo di istanza ToString()

Il metodo `ToString()` svolge la funzione inversa del metodo `Parse()` e cioè converte un valore nel suo equivalente in formato stringa, analogamente a quanto fa il metodo `ToString()` della classe `Convert`.. Essendo un metodo di istanza, deve essere invocato attraverso il valore da convertire, secondo la forma:

$$oggetto.\texttt{ToString}(formato_{opz})$$

Il metodo è utilizzabile con valori di tipo qualsiasi. Ad esempio, il seguente codice:

```
int a = 10;
double x = 20.1;
string sa = a.ToString();
string sx = x.ToString();
```

equivale a:

```
int a = 10;
double x = 20.1;
string sa = Convert.ToString(a);
string sx = Convert.ToString(x);
```

Il metodo elabora il valore contenuto nell'oggetto attraverso il quale viene eseguito (nell'esempio, le variabili `a` e `x`). Come è stato già detto, il termine oggetto non designa necessariamente una variabile; infatti, è perfettamente lecito (anche se di dubbia utilità) scrivere:

```
double x = 100;
int a = 10;
string s = (x * 100 / a).ToString();
```

L'istruzione determina la valutazione dell'espressione `x * 100 / a` e la successiva conversione del valore risultante nell'equivalente stringa, che è `"1000"`.

Uso di stringhe formato

Esiste una versione del metodo `ToString()` che accetta come argomento uno **specificatore di formato**. Questo consente di intervenire sulla forma assunta dalla stringa risultante. Ad esempio, la seguente istruzione:

```
string s = 10.ToString("#.00");
```

determina la conversione in stringa del valore `10` in base al formato `"#.00"`; la stringa risultante è `"10,00"`.

2 Approfondimenti sugli array

In questo paragrafo approfondiremo alcuni aspetti sugli array che per semplicità abbiamo trascurato nel Capitolo 4. Non esauriremo comunque l'argomento, che sarà ripreso nella parte dedicata alle collezioni.

2.1 Stato iniziale e inizializzazione di una variabile array

Nel quarto capitolo è stata sottolineata la distinzione tra una variabile array e l'array stesso. Una variabile array rappresenta un riferimento all'array vero e proprio. Dichiarando ad esempio:

```
int[] v;
```

viene creata una variabile array ma non un oggetto array, la cui creazione avviene mediante l'invocazione dell'operatore `new`:

```
v = new int[10];
```

Ora, una domanda molto importante è: in che stato si trova la variabile `v` prima di essere associata a un array mediante l'operatore `new`? La risposta sta nelle regole che definiscono lo stato iniziale di qualsiasi altra variabile, e cioè:

- se la variabile è locale, il suo stato iniziale è indefinito;
- se la variabile è un campo di classe, il suo stato iniziale è quello predefinito, che nel caso in questione è rappresentato dalla costante `null`.

La costante `null` esprime il fatto che la variabile array non referenzia nessun oggetto array.

Inizializzazione di una variabile array

Considerato che esiste una costante per designare uno stato iniziale nullo di una variabile array, tale costante può essere usata esplicitamente per fornire un valore iniziale alle variabili array locali. Il codice:

```
int[] v = null;
```

dichiara una variabile array e la inizializza con un valore nullo. Nota bene, dopo una simile dichiarazione, la variabile `v` si trova in uno stato definito, anche se non referenzia alcun array.

L'utilità della costante `null` appare per il momento poco chiara. Comunque, esistono delle situazioni nelle quali è necessario inizializzare le variabili array con il valore `null`.

2.2 Lunghezza di un array: proprietà Length

La lunghezza di un array è rappresentata dal numero degli elementi che esso contiene, ottenuto moltiplicando il numero delle dimensioni per il numero degli elementi specificati in fase di creazione per ogni dimensione. Per il momento sarà fatto riferimento agli array unidimensionali, per i quali il numero degli elementi coincide con il valore specificato in fase di creazione.

Si può conoscere la lunghezza di un vettore attraverso la proprietà `Length` (lunghezza) del medesimo, usando la seguente sintassi:

oggetto-array`.Length`

In generale, si può pensare a una **proprietà** come a un particolare tipo di variabile campo di classe. Esattamente come per i metodi, esistono due tipi di proprietà: le proprietà statiche e quelle di istanza. Una proprietà statica è accessibile soltanto attraverso l'identificatore di tipo che la definisce; dunque:

nome-tipo.nome-proprietà

Una proprietà di istanza è accessibile soltanto attraverso un oggetto; dunque e cioè:

oggetto.nome-proprietà

Il seguente frammento di codice crea un vettore di 10 elementi e ne mostra la lunghezza visualizzando il valore della proprietà `Lenght`

```
int[] v = new int[10];        // array di 10 elementi
int numElementi = v.Length
Console.WriteLine("v contiene {0} elementi", numElementi);
```

E' importante comprendere che la proprietà `Length`, come tutte le proprietà e i metodi di istanza, esiste soltanto in relazione a un oggetto precedentemente creato. Ad esempio, il seguente codice è formalmente scorretto:

```
int[] v;
int numElementi = v.Length    // errore: v non referenzia nessun oggetto!
```

poiché si tenta di accedere alla proprietà di un oggetto che ancora non è stato creato.

Uso della proprietà Length

La possibilità di conoscere la lunghezza di un array è estremamente utile. Si consideri ad esempio il seguente problema, già affrontato in un capitolo precedente:

Dato un vettore contenente le altezze in centimetri di un numero arbitrario di persone, si vuole calcolare l'altezza media delle stesse.

Il metodo `CalcolaMediaAltezze()` necessita di un parametro che specifichi il numero di elementi contenuti nell'array del quale calcolare la media. Poiché però la proprietà `Length` ci fornisce già questa informazione, l'uso di un parametro aggiuntivo è superfluo:

```
class Program
{
    static void Main()
    {
        double altezzaMedia;
```

```
        int numPersone = Convert.ToInt32(Console.ReadLine());
        double[] altezze = new double[numPersone];

        // ... qui vengono inserite nel vettore le altezze delle persone

        altezzaMedia = CalcolaAltezzaMedia(altezze);
        Console.WriteLine("L'altezza media è: {0}", altezzaMedia)
    }

    static double CalcolaAltezzaMedia (double [] v)
    {
        double media = 0;
        foreach (double altezza in v)
            media = media + altezza;
        return media / v.Length;

    }
}
```

Il vantaggio dell'uso della proprietà `Length` è evidente:

- non è più necessaria una variabile per memorizzare il numero degli elementi dell'array;
- semplifica il prototipo, e dunque anche l'utilizzo, dei metodi che elaborano array, eliminando la necessità di un parametro aggiuntivo che ne specifichi la lunghezza.

2.3 Array multidimensionali: lunghezza delle singole dimensioni

Anche gli array multidimensionali espongono la proprietà `Length`, ma per essi questa possiede una minor utilità, poiché equivale al numero totale degli elementi. Ad esempio:

```
int[,] m = new int[10,5];          // matrice di 50 elementi
int numElementi = m.Length
Console.WriteLine("m contiene {0} elementi ", numElementi);
```

questo codice produce sullo schermo:

```
m contiene 50 elementi
```

Di solito, ciò che è veramente utile conoscere di un array multidimensionale è il numero degli elementi relativamente alle singole dimensioni; per una matrice, il numero di righe e di colonne. A questo scopo esiste il metodo di istanza `GetLength()`; esso richiede come unico parametro l'indice della dimensione che si vuole indagare e produce come risultato il numero di elementi. La sua invocazione assume la forma:

oggetto-array`.GetLength(`*indice-dimensione*`)`

dove `indice-`*`dimensione`* viene interpretato come segue:

`0` → prima dimensione (righe);

`1` → seconda dimensione (colonne);

`2` → terza dimensione; eccetera.

Ad esempio:

```
int[,] m = new int[10,5];          // matrice di 10 x 5 elementi
int numRighe = m.GetLength(0);
int numColonne = m.GetLength(1);
Console.WriteLine("m contiene {0} righe e {1} colonne", numRighe, numColonne);
```

questo codice produce sullo schermo:

```
m contiene 10 righe e 5 colonne
```

3 Tipo string

Il tipo `string` è implementato dal linguaggio attraverso la classe `String`. Il termine `string` (con la lettere minuscola) non è altro che un sinonimo del termine `String` (con la lettera maiuscola). La classe `String` espone sia proprietà che metodi; specialmente questi ultimi sono ampiamente utilizzati nella maggioe parte dei programmi.

3.1 Stringhe come collezioni di caratteri

Le stringhe sono oggetti particolari, possono essere considerati sia come singoli valori che come vettori di caratteri. Questa ambivalenza può generare confusione; infatti le stringhe, come i vettori, sono variabili strutturate, ma nelle operazioni di assegnazione e confronto mostrano i comportamenti tipici delle variabili semplici.

In memoria, una stringa è gestita come un vettore di caratteri. Ad esempio, la stringa:

```
string s = "pippo";
```

può essere schematizzata nel seguente modo:

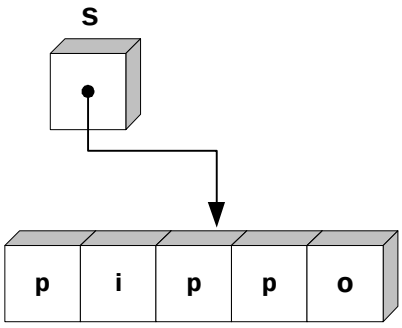


Figura 8-1 Rappresentazione dell'oggetto stringa s.

Una variabile stringa, come una variabile vettore, contiene in realtà un riferimento alla stringa vera e propria. La differenza tra le stringhe e gli array è che le prime "nascondono" la loro natura, consentendo al programmatore di utilizzarle come se fossero variabili semplici.

Un oggetto stringa resta comunque una collezione di caratteri e può essere impiegato come tale; infatti, benché raramente sia necessario, è possibile accedere ai singoli caratteri di una stringa nello stesso modo in cui si accede agli elementi di un vettore:

oggetto-stringa[*indice-carattere*]

Ad esempio, il seguente codice:

```
string s = "pippo";
Console.WriteLine("Il terzo carattere è: {0}", s[2]);
```

produce sullo schermo:

```
Il terzo carattere di s è: p
```

D'altra parte, a differenza degli array, l'accesso ai caratteri di una stringa è permesso solo in lettura. E' dunque sbagliato scrivere:

```
string s = "pippo";
s[1] = 'a';       // errore: proibito assegnare un valore al singolo carattere!
```

3.2 Lunghezza di una stringa: proprietà Length

Si può conoscere la lunghezza di una stringa attraverso la proprietà di istanza `Length`, il cui uso è perfettamente analogo a questo già visto con gli array. Ad esempio:

```
string s = "questa è una stringa";
int numCaratteri = s.Length;
Console.WriteLine("s contiene {0} caratteri", numCaratteri);
```

Il precedente codice visualizzerà sullo schermo:

```
s contiene 20 caratteri
```

3.3 Confrontare due stringhe: metodo Compare()

Diversamente dai tipi `int`, `double`, `char` e `bool`, gli unici operatori relazionali utilizzabili con le stringhe sono `==` e `!=`, i quali consentono di stabilire se due stringhe sono esattamente uguali o diverse per almeno un carattere (compresi gli spazi finali e la distinzione tra lettere maiuscole e minuscole). Ciò non significa che sia impossibile confrontare due stringhe in altro modo, ad esempio per stabilire se una stringa è minore, maggiore o uguale ad un'altra. Questo si può ottenere mediante il metodo statico `Compare()`. La sua invocazione assume la forma:[16]

```
string.Compare(stringaA, stringaB)
```

e opera un confronto carattere per carattere delle due stringhe, chiamato confronto alfanumerico. Il metodo ritorna un valore intero in accordo alle seguenti regole:

- -1: se `stringaA` è **minore** di `stringaB`;
- 0: se `stringaA` è **uguale** a `stringaB`;
- 1: se `stringaA` è **maggiore** di `stringaB`.

Ad esempio, il seguente codice:

```
int cfr = string.Compare("superman", "spider man");
switch (cfr)
{
    case -1: Console.WriteLine("spider man è più grande di superman"); break;
    case 0: Console.WriteLine("spider man è uguale a superman"); break;
    case 1: Console.WriteLine("superman è più grande di spider man"); break;
```

[16] In realtà il metodo `Compare()`, come moltissimi altri metodi, è fornito in più versioni, che si differenziano per il numero degli argomenti e il comportamento prodotto.

```
}
```

produce sullo schermo:

```
superman è più grande di spider man
```

poiché la stringa `"superman"` è lessicograficamente maggiore della stringa `"spider man"`.

Confronto alfanumerico

Il confronto alfanumerico tra due stringhe su basa sostanzialmente sul confronto dei corrispondenti caratteri sulla base del loro valore Unicode. Ciò significa, ad esempio, che la stringa "137" è minore della stringa "82", perché il carattere '1' viene confrontato con il carattere '8', e il primo ha un valore Unicode inferiore al secondo.

Le cose vanno diversamente quando vengono confrontate due lettere dell'alfabeto; in questo caso `Compare()` non fa distinzione tra lettere minuscole e lettere maiuscole, nonostante abbiano codici Unicode diversi, soltanto, però, se le due stringhe sono diverse; se sono uguali vengono prese in considerazione le differenze di case e le lettere maiuscole vengono considerate più grandi delle lettere minuscole corrispondenti. Si parla in questo caso di confronto *case sensitive*, e cioè sensibile al case delle lettere. Facciamo alcuni esempi:

- "zorro" è **maggiore** di "SUPERMAN";
- "ZORRO" è **maggiore** di "zorro", poiché le parole sono uguali e 'Z' viene considerata maggiore di 'z';
- "Amore" è **minore** di "amorevole", poiché le parole sono diverse e la prima è una sottostringa (è contenuta in) della seconda;

L'ultimo esempio è significativo. "Amore" e "amorevole" sono uguali soltanto per i primi cinque caratteri e dunque non viene fatta distinzione tra "A" ed "a". D'altra parte la seconda stringa è più lunga della prima e quindi risulta maggiore.

> Esiste comunque il metodo statico `CompareOrdinal()` che tratta le lettere allo stesso modo di tutti gli altri caratteri, e cioè basando il confronto unicamente sul loro valore Unicode. Per tale metodo, le lettere miuscole hanno un valore maggiore delle maiuscole corrispondenti. Ne consegue che, ad esempio, `CompareOrdinal()` considera la stringa "Amore" più piccola della stringa "amore".

3.4 Inizializzazione di una stringa

Poiché una variabile stringa rappresenta un riferimento a un oggetto, ad essa può essere assegnato un valore nullo esattamente come ad una variabile array. Ad esempio, il seguente codice:

```
string s = null;
```

dichiara e inizializza una variabile stringa. Dopo questa istruzione, la variabile s possiede un valore (`null` appunto) ma non referenzia nessun oggetto stringa.

Stringhe nulle e stringhe vuote

Attenzione, una stringa nulla è diversa da una una stringa vuota, infatti scrivere:

```
string s = null;
```

invece di

```
string s = "";
```

non è affatto la stessa cosa. Nel primo caso la variabile `s` non referenzia alcun oggetto, mentre nel secondo referenzia un oggetto stringa che non contiene caratteri.

La differenza diventa evidente nel codice che segue:

```
string s = "";
string s1 = null;
int a = s.Length;       // ok: ad a viene assegnato zero.
int b = s1.Length;      // errore: s1 non referenzia nessun oggetto!
```

L quarta istruzione produce un errore di esecuzione perché `s1` non fa riferimento ad alcun oggetto, ed è ovviamente errato accedere alla lunghezza di un oggetto che non esiste nemmeno.

Dal punto di vista del confronto lessicografico tra due stringhe, una stringa vuota è più piccola di qualsiasi altra stringa che contenga almeno un carattere, mentre una stringa nulla è più piccola di qualsiasi altra stringa (vuota o meno) eccetto un'altra stringa nulla. Dunque, il seguente codice:

```
string s1 = "";
string s2 = null;
int cfr = string.Compare(s1, s2);
Console.WriteLine(cfr);
```

produce come risultato 1, poiché la stringa `""` è maggiore di `null`.

4 Il tipo char

Ogni elemento di una stringa appartiene al tipo `char`, il quale appartiene a sua volta alla categoria dei tipi valore. Un valore di tipo `char` occupa due byte di memoria ed è codificato attraverso un numero appartenente al codice Unicode.

4.1 Costanti letterali char

Sotto sono mostrate tre delle quattro notazioni esistenti per esprimere una costante letterale `char`; tutte fanno uso di una coppia di apici singoli per delimitare la costante:

```
'carattere'              forma n° 1

'\xcodice-carattere'     forma n° 2

'\sequenza-di-escape'    forma n° 3
```

Ecco alcuni esempi:

```
char c1= 'A';          // carattere A espresso nella notazione 1
char c2 = '\x41'       // carattere A espresso nella notazione 2
char c3 = '\n';        // carattere "nuova riga" espresso nella notazione 3
```

La notazione n° 2 consente di esprimere un carattere specificando direttamente il suo codice Unicode in formato esadecimale. In genere viene usata quando è necessario esprimere caratteri che non hanno un equivalente simbolico. Nella notazione n° 3, la barra retroversa determina l'inizio di una **sequenza di escare**: il carattere che segue la barra non viene interpretato letteralmente ma assume un significato particolare, che dipende dal carattere stesso. La sequenza `'\n'`, ad esempio,

significa **nuova linea** e determina lo spostamento a capo del cursore testo se utilizzata nei metodi `Write()` e `WriteLine()`.

4.2 Valore iniziale di una variabile carattere

Come per tutte le altre variabili, il valore iniziale di una variabile carattere dipende dal fatto che sia una variabile locale o un campo di classe. Nel primo caso il suo stato iniziale è indefinito, nel secondo caso il valore iniziale è il carattere corrispondente al codice zero: `'\x0'` (il quale non deve essere confuso con il carattere `'0'`).

4.3 Compatibilità tra il tipo char e gli altri tipi di dati

Il tipo `char` presenta una forma particolare di compatibilità verso gli altri tipi. Esso può essere usato in tutte le espressioni nelle quali è ammissibile il tipo `int`; esiste cioè una conversione implicita da `char` a `int`. In questo caso, il valore che esprime un carattere è semplicemente quello del suo codice Unicode. Questa possibilità consente di scrivere espressioni a tutta prima piuttosto stravaganti:

```
int a = 'C' - 'A';      // a = codice Unicode di 'C' meno codice Unicode di 'A'
Console.WriteLine("Differenza tra i codici di 'C' e 'A': {0}", a);
```

Questo frammento produce sullo schermo:

```
Differenza tra i codici di 'C' e 'A': 2
```

essendo il codice di `'A'` uguale a 65 e quello di `'C'` uguale a 67.

Per contro, non esiste affatto una conversione da `int`, o qualsiasi altro tipo, al tipo `char`. E' dunque scorretto scrivere:

```
char c = 65;      // errore: non ammesso da int a char!
```

Compatibilità con il tipo string

Benché il tipo `char` sia il tipo degli elementi di una stringa, tra i due tipi non esiste una forte compatibilità. Non esiste per l'assegnazione; è dunque scorretto scrivere:

```
string s = 'A';   // errore: non si può assegnare un carattere a una stringa!
char c = "A";     // errore: non si può assegnare una stringa a un carattere!
```

> Occorre dunque fare particolarmente attenzione alla notazione usata: gli apici delimitano una costante carattere, mentre le virgolette una costante stringa: le due notazioni si somigliano ma designano costanti di tipo diverso.

Esiste invece la possibilità di concatenare caratteri a una stringa. Ad esempio, il seguente codice:

```
string s = "pippo e ";
s = s + 'c';
```

produce la stringa `"pippo e c"`

> Alcuni metodi del tipo `string`, qui non trattati, sono in grado di elaborare array di caratteri, in pratica trattandoli come se fossero stringhe.

Anche il tipo `char`, come tutti i tipi semplici, espone i metodi `Parse()` e `ToString()`. Il seguente codice mostra come effettuare una conversione da `string` a `char` e viceversa.

```
char ch = char.Parse("A");          // a ch viene assegnato 'A';
string st = ch.ToString();          // a st viene assegnato "A";
```

4.4 Operazioni con i caratteri

Poiché nelle espressioni i caratteri vengono considerati come valori interi, per essi sono ammissibili tutte le operazioni consentite per i valori di tipo `int`, comprese le operazioni di confronto mediante gli operatori relazionali. Anche in esse vengono considerati i codici numerici dei caratteri e ciò determina, a volte in modo poco intuitivo, il risultato dell'operazione. Ad esempio, il risultato del confronto:

```
'A' == 'a'
```

è `false`, poiché esso viene tradotto nel confronto tra i codici numerici:

```
65 == 97
```

Allo stesso modo, il risultato del confronto:

```
'Z' < 'a'
```

è `true`, poiché esso viene tradotto nel confronto tra i codici numerici:

```
90 < 97
```

> Si ricorda che nel confronto alfanumerico tra due stringhe effettuato mediante il metodo `Compare()`, le cose vanno diversamente per quanto riguarda le lettere dell'alfabeto; in questo caso 'Z' è maggiore di 'A', indipendentemente dal case delle lettere.

4.5 Costanti stringa verbatim

Le varie forme di rappresentazione delle costanti `char` restano valide anche quando il carattere in questione è parte di una stringa. Si consideri la seguente istruzione:

```
Console.WriteLine("c:\documenti\capitolo 1");
```

Il letterale stringa che funge da argomento al metodo `WriteLine()` contiene al proprio interno due sequenze di caratteri (`'\d'` e `'\c'`) che il compilatore tenta di interpretare come sequenze di escape, poiché cominciano con il carattere `'\'`. Poiché però `'\d'` e `'\c'` non sono sequenze di escape valide viene prodotto un errore formale.

L'intero problema nasce dal fatto che, casualmente, la stringa da visualizzare contiene occorrenze del carattere `'\'`, il quale determina l'interpretazione del carattere successivo come una sequenza di escape. Una prima soluzione sta proprio nell'uso delle sequenze di escape; tra queste esiste infatti la sequenza `'\\'` che viene interpretata e tradotta nel carattere `'\'`. Dunque l'istruzione:

```
Console.WriteLine("c:\\documenti\\capitolo 1");
```

viene compilata correttamente e produce il risultato desiderato:

```
c:\documenti\capitolo 1
```

Esiste comunque una seconda soluzione, senz'altro più soddisfacente: l'uso di **stringhe verbatim**. Una costante stringa verbatim è designata dal carattere `@`, secondo la sintassi:

`@letterale-stringa`

Il termine verbatim può essere tradotto in: "alla lettera". I caratteri contenuti in una stringa verbatim non subiscono infatti alcuna forma di interpretazione, ma vengono elaborati così come sono. Dunque, l'istruzione:

```
Console.WriteLine(@"c:\documenti\capitolo 1");
```

produce correttamente:

```
c:\documenti\capitolo 1
```

5 Tipi valore e tipi riferimento

Esiste una sostanziale differenza nella modalità di memorizzazione di variabili `int`, `double`, `bool` e `char` da una parte e di array e stringhe dall'altra. Il perché risiede nell'appartenenza di qualsiasi tipo di dato a una o l'altra delle seguenti categorie: **tipi valore** e **tipi riferimento**.

Queste designano il modello di memorizzazione impiegato per gli oggetti di un determinato tipo.

5.1 Tipi valore

All'interno della categoria del tipi valore, il contenuto – il valore – delle variabili è accessibile direttamente attraverso il loro nome. In altre parole, il nome della variabile rappresenta l'indirizzo di memoria nel quale è memorizzato il valore.

Si immagini la memoria del computer come un lungo nastro, diviso in celle – i byte – numerate a partire da zero; la posizione di un byte all'interno del nastro rappresenta dunque l'indirizzo del medesimo[17]. Il contenuto di una variabile occupa un certo numero di byte consecutivi, che dipende dal tipo. Il compilatore traduce il nome della variabile nell'indirizzo del primo byte di questa sequenza. Ad esempio, le dichiarazioni:

```
int a = 100;
double x = 200;
```

producono in memoria la seguente situazione (gli indirizzi sono stati scelti in modo puramente arbitrario):

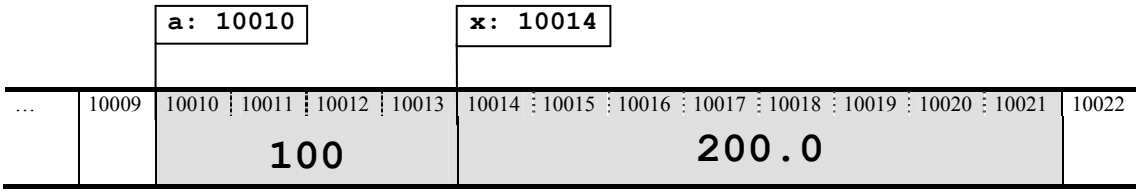


Figura 8-2 Rappresentazione in memoria di una variabile int e una variabile double.

Da un punto di vista schematico, le variabili il cui tipo appartiene alla categoria dei tipi valore possono essere così rappresentate:

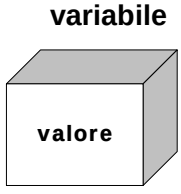


Figura 8-3 Rappresentazione schematica di una variabile di tipo valore.

Assegnazione di oggetti appartenenti ai tipi valore

[17] La metafora del nastro è appropriata ma non dev'essere presa alla lettera. Nella pratica, le memoria e lo "spazio di indirizzamento" dei byte vengono gestiti in modo altamente sofisticato da parte del sistema operativo e di .NET.

L'assegnazione di un valore a una variabile determina la copia dei byte che rappresentano il valore nella zona di memoria indirizzata dal nome della variabile. L'operazione viene eventualmente preceduta nella conversione del valore nel tipo della variabile.

Ad esempio, nel seguente codice:

```
int b = 20;
int a = b;
```

l'assegnazione `a = b` produce in memoria la seguente situazione:

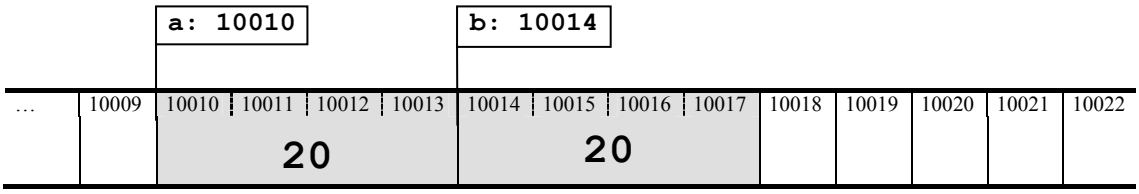


Figura 8-4 Rappresentazione in memoria di una variabile int e una variabile double.

5.2 Tipi riferimento

All'interno della categoria dei tipi riferimento esiste una distinzione tra la variabile e l'oggetto che viene referenziato attraverso di essa. Anche in questo modello di memoria il nome di una variabile rappresenta un indirizzo, ma diversamente da quanto accade per i tipi valore

il contenuto memorizzato a quell'indirizzo non è il valore della variabile, ma un secondo indirizzo che punta alla zona di memoria che contiene effettivamente il valore.

Ad esempio, l' istruzione:

```
int[] v = new int[3] = {-10, -20, -30};
```

produce in memoria la seguente situazione:

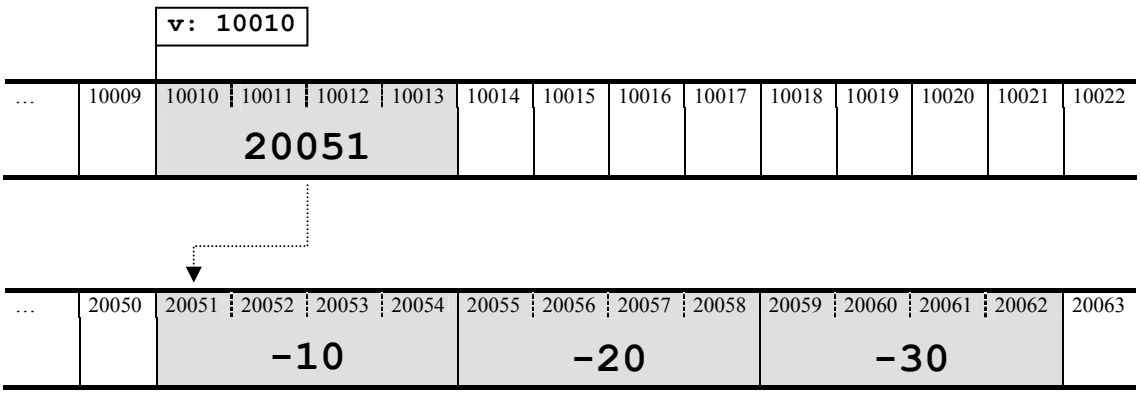


Figura 8-5 Rappresentazione in memoria di una variabile e di un oggetto array.

La variabile `v` non contiene il vettore ma un indirizzo di memoria che punta al vettore vero e proprio. Da un punto di vista schematico, le variabili il cui tipo appartiene alla categoria dei tipi riferimento possono essere così rappresentate:

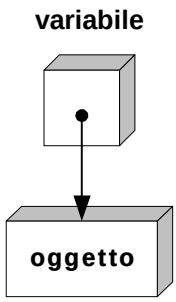


Figura 8-6 Rappresentazione schematica di una variabile di tipo riferimento.

Risultato prodotto dall'assegnazione di oggetti appartenenti ai tipi riferimento

Nel caso dei tipi riferimento l'assegnazione produce sempre la copia di un riferimento e non dell'oggetto referenziato; produce cioè la copia dei byte che rappresentano l'indirizzo della zona di memoria contenente l'oggetto referenziato.

Ad esempio, il seguente codice:

```
int[] v = new int[3] = {-10, -20, -30};
int[] v1 = v;           // copia di un riferimento
```

produce in memoria la seguente situazione:

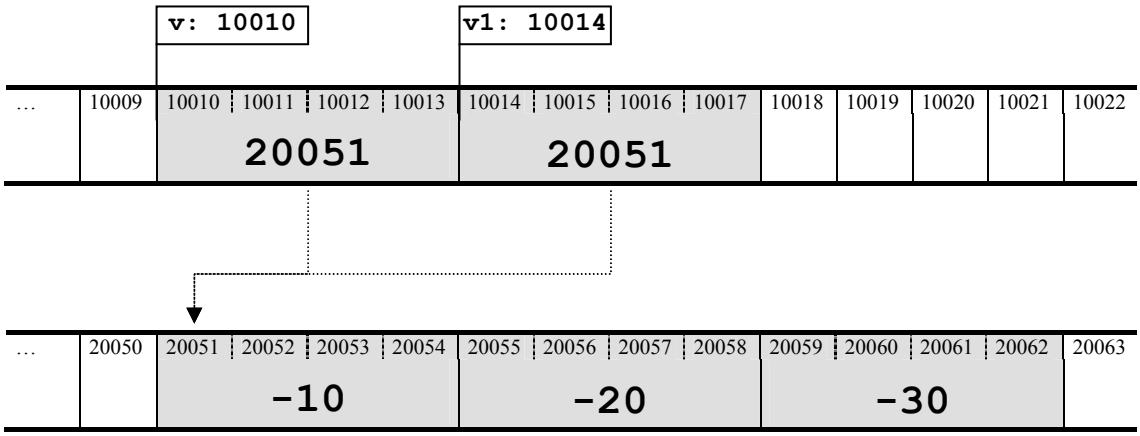


Figura 8-7 Rappresentazione in memoria di una variabile e di un oggetto array.

Dopo l'assegnazione, le variabili `v` e `v1` contengono l'identico indirizzo di memoria, il quale rappresenta un riferimento allo stesso vettore.

5.3 Conclusioni sui due modelli di memorizzazione

Non è molto importante focalizzare l'attenzione sulle idee di byte, indirizzo di memoria, eccetera; ciò che veramente conta è cogliere la fondamentale differenza tra tipi valore e tipi riferimento, e cioè che:

mentre una variabile di tipo valore contiene l'oggetto che rappresenta l'informazione, una variabile di tipo riferimento contiene un riferimento all'oggetto.

E' questa distinzione a determinare la diversità di comportamento nelle assegnazioni, nei confronti, eccetera, tra le due categorie di variabili.

6 Conversioni esplicite: operatore di cast

In alcune situazioni il linguaggio ammette che valori di tipo diverso compaiano nella stessa espressione, oppure che in un'istruzione di assegnazione il tipo dell'espressione sia diverso dal tipo della variabile. In questi casi, e quando è ammissibile, il linguaggio effettua delle conversioni implicite. Una conversione implicita modifica il tipo di un valore ma non modifica il valore stesso. E' dunque una conversione sicura, poiché garantisce che nella trasformazione da un tipo a un altro non vi sarà perdita di informazione.

In relazione ai tipi introdotti, le uniche conversioni implicite permesse sono:

- da `int` a `double`;
- da `char` a `int`;

Esistono situazioni, comunque, nelle quali è appropriato effettuare una conversione anche se questa non è sicura, cioè anche se può produrre una modifica del valore e una perdita di informazione. A questo scopo il linguaggio mette a disposizione l'**operatore di cast**, la cui notazione assume la seguente forma:

(*tipo*) *espressione*

L'operatore è composto da una coppia di parentesi al cui interno è specificato il tipo verso il quale si intende effettuare la conversione; il tipo specificato è definito **tipo destinazione** del cast. Ad esempio, nel seguente codice:

```
double x = 10.7;
int a = (int) x;
```

viene applicato l'operatore di cast alla variabile `x`, il cui valore viene "forzatamente" convertito in un valore `int`, che successivamente viene assegnato alla variabile `a`.

Una conversione mediante cast viene definita **conversione esplicita** poiché il programmatore indica in modo esplicito il tipo destinazione della conversione. Il risultato di una conversione esplicita dipende sia dai tipi che dai valori effettivamente coinvolti. Nell'esempio precedente, la conversione di `x` al tipo `int` produce il valore 10, poiché la parte decimale del numero viene troncata. Ma non sempre una conversione esplicita determina una perdita di informazione o, al contrario, non sempre il valore risultate è dello stesso ordine di grandezza del valore da convertire.

Ad esempio, il seguente codice contiene due conversioni esplicite da `double` a `int`:

```
double x = 10;
double x1 = 1E12;
int a = (int) x;      // nessuna perdita di informazione
int a1 = (int) x1;    // completa perdita di informazione
```

Nella prima, il valore `double` 10 viene convertito nel valore `int` 10: cambia il tipo ma non il valore. Nella seconda conversione, invece, il valore `double` 1E12 (mille miliardi) viene convertito nel valore `int` -727379968: si ha cioè una totale perdita di informazione.

Il motivo di questa differenza di risultato è facilmente comprensibile. Nell'eseguire una conversione, il linguaggio "tenta" di mantenere il valore invariato o comunque di limitare la perdita alla precisione, salvando l'ordine di grandezza, ma non sempre questo è possibile, e ciò accade quando il valore da convertire supera i limiti dell'intervallo di variazione del tipo di destinazione. In questo caso, il valore risultante è praticamente casuale.

In realtà non è che il linguaggio, vista l'impossibilità di convertire il valore in modo appropriato, ne generi uno a caso. La procedura di conversione resta la medesima, la differenza sta nel fatto che tale procedura, applicata a valori che superano l'intervallo di variazione del tipo destinazione, produce risultati apparentemente casuali.

Le conversioni esplicite non riguardano soltanto valori di natura numerica. Ad esempio, il seguente codice:

```
char c = (char) 65;
```

converte il valore intero 65 nell'equivalente valore carattere, e cioè `'A'`. In questo caso la conversione non determina alcuna forma di elaborazione sul valore, poiché i caratteri vengono appunto codificati mediante dei numeri interi.

6.1 Priorità nell'applicazione dell'operatore di cast

L'operatore di cast ha la precedenza sulla maggior parte degli operatori, in pratica su tutti quelli che abbiamo considerato finora. Ciò ha conseguenze notevoli nella modalità di valutazione di un'espressione. Ad esempio, nel seguente codice l'intenzione è quella di calcolare un'espressione `double` (`x * x1`) e di assegnarla a una variabile `int` dopo aver effettuato una conversione esplicita:

```
double x;
double x1;
...
int a = (int) x * x1;   // errore formale!
```

Ma l'espressione viene valutata in modo diverso da quello desiderato. Infatti, l'operatore di cast `(int)` ha precedenza sull'operatore `*` e dunque viene applicato alla sola variabile `x`. Il risultato, di tipo `int`, viene quindi moltiplicato per `x1`. Sappiamo che un intero per un `double` produce ancora un `double`; dunque l'intera espressione è di tipo `double` e quindi non può essere assegnata a una variabile intera.

Il modo corretto di procedere è forzare il giusto ordine di valutazione degli operatori usando le parentesi:

```
double x;
double x1;
...
int a = (int) (x * x1);       // ok
```

Adesso, prima viene eseguito il prodotto tra `x` e `x1` e dopo viene convertito il risultato nel tipo `int`, il quale può dunque essere assegnato alla variable `a`.

(La tabella sulla precedenza degli operatori è trattata nell'Appendice A)

6.2 Conversioni non ammissibili

Poiché nelle conversioni esplicite è il programmatore a indicare il tipo destinazione si potrebbe supporre che sia ammessa qualsiasi forma di conversione, ma non è affatto così. Il linguaggio stabilisce che alcune conversioni non sono comunque ammesse. Le regole secondo le quale ciò avviene sono diverse; ci limiteremo ad esporle in forma generale.

E' ammessa la conversione esplicita da un tipo <TO> a un tipo <TD> se:

c) <TO> e <TD> sono della stessa natura. Ciò implica che sono ammesse tutte le conversioni esplicite tra tipi numerici;

d) <TO> e <TD> sono di natura diversa, ma hanno la stessa modalità di codifica in memoria. Questa regola riguarda i tipi di dati la cui codifica è basata su un valore numerico intero, come ad esempio i tipi `int` e `char`.

Al di fuori di tali regole, non è di norma ammessa nessuna forma di conversione esplicita. Dunque non è ammesso convertire:

1) da `bool` a qualsiasi tipo di dato; o da qualsiasi tipo di dato a `bool`;
2) da `string` a qualsiasi tipo di dato; o da qualsiasi tipo di dato a `string`[18];
3) da un tipo array a qualsiasi tipo array diverso dal primo.

> Esiste un tipo di dato particolare, il tipo `object`, che non è soggetto a queste restrizioni, e che sarà trattato successivamente.

Detto questo, le conversioni presenti nel seguente codice sono tutte scorrette, e come tali sono segnalate durante la fase di compilazione:

```
double x = 100;
int i = 10;
bool b = true;
int[] v = {10, 20, 30}
double[] vx;
string s = "ciao";

x = (double) s;      // errore!
i = (int) s;         // errore!
s = (string) i;      // errore!
s = (string) x;      // errore!
b = (bool) i;        // errore!
i = (int) b;         // errore!
vx = (double[]) v;   // errore!
```

6.3 Uso delle conversioni esplicite

L'uso dell'operatore cast è molto comune nell'ambito di programmi realistici, lo è di meno in relazione al livello introduttivo che stiamo qui considerando. In sostanza, perlomeno per quanto riguarda i tipi predefiniti, si rende necessaria una conversione esplicita quando il tipo, ma non il valore, di una espressione non è appropriato rispetto all'uso che se ne deve fare. Ad esempio, si supponga che il risultato di un determinato calcolo debba essere assegnato a una variabile, necessariamente intera, da usare come indice in un vettore. Si supponga inoltre che al calcolo partecipi una variabile di tipo `double`; in questo caso, il tipo del risultato è `double`, e come tale non può essere implicitamente convertito in un intero: è necessaria una conversione esplicita.

[18] Questa restrizione non riguarda l'operatore di concatenazione +. Tale operatore, infatti, accetta qualsiasi tipo di dato, purché almeno uno dei due operandi sia di tipo stringa.

7 Il tipo object

Esiste un aspetto fondamentale che caratterizza tutti i tipi di dati del .NET, essi non rappresentano delle entità completamente separate, ma fanno parte di una gerarchia, all'interno della quale esistono delle precise relazioni. Convenzionalmente, tale gerarchia viene definita **modello ad oggetti** di .NET.

Le caratteristiche principali del modello ad oggetti sono affrontate in un altro volume; qui ci limitiamo ad introdurre il tipo di dato che sta alla base della gerarchia e che, in un certo senso, fa da denominatore comune a tutti i tipi di dati: il tipo `object`.

`object` è un tipo particolare, poiché, diversamente dagli altri tipi, non codifica informazioni; il suo scopo è infatti quello di:

- definire i metodi comuni a tutti i tipi di dati (tra i quali il metodo `ToString()`, che abbiamo già incontrato);
- consentire l'uso di variabili che siano in grado di memorizzare valori di tipo qualsiasi.

Una variabile di tipo `object` si comporta come un contenutore in grado di memorizzare valori di tipo qualsiasi; ciò consente di implementare degli algoritmi indipendenti dal tipo di dati che devono elaborare. L'argomento è piuttosto complesso, pertanto qui ci limitiamo a fornire un esempio di dimostrativo rappresentato dal seguente frammento di codice:

```
static void Visualizza (object[] v)
 {
     for (int i = 0; i < v.Length; i++)
     {
         string s = v[i].ToString();
         Console.WriteLine(s);
     }
 }

static void Main()
{

    object[] valori = {"pippo", 10, 2.3, true };
    Visualizza (valori);
    return;
}
```

Il programma produce il seguente output:

```
Pippo

10

2,3

true
```

In `Main()` viene inzializzato un vettore di `object` con quattro valori di tipo diverso (`string`, `int`, `double` e `bool`). Quindi viene invocato il metodo `Visualizza()`, che riceve il vettore come argomento. `Visualizza()` definisce come parametro un vettore di tipo `object` e itera su di esso invocando il metodo `ToString()` per ogni elemento e visualizzando successivamente la stringa ottenuta.

Qui avviene la cosa interessante. Per ogni elemento viene invocato il metodo `ToString()` del tipo appropriato, nonostante nel codice non esista niente che faccia riferimento al tipo effettivo dei valori memorizzati nel vettore. In sostanza, `Visualizza()` elabora il vettore indipendentemente dal tipo effettivo dei valori in esso memorizzati.

La caratteristica di fungere da contenitore per valori di tipo diverso non appartiene al solo tipo `object` ma è un elemento cardine della programmazione object oriented; essa consente di implementare degli algoritmi in grado di elaborare i dati pur senza conoscerne il tipo effettivo.

Come vedremo nel successivo paragrafo, mediante i *generics* è possibile ottenere un risultato analogo pur utilizzando un diverso meccanismo.

8 Tipi generici (*Generics*)

Una delle questioni centrali della programmazione è la facoltà di scrivere algoritmi in grado di processare i dati indipendentemente dal loro tipo. Ciò consente di implementare un determinato procedimento una sola volta e di poterlo applicare a dati di tipo diverso. Il paradigma di programmazione object oriented è in grado di rispondere soltanto in parte a tale esigenza, per questo motivo alcuni linguaggi, C# compreso, forniscono un meccanimo alternativo sviluppato ad hoc: i tipi generici, o *generics*.

Mediante i *generics* è possibile definire dei nuovi tipi (chiamati **tipi aperti –** *open types*) senza stabilire la natura delle informazioni che codificano, ma soltanto le operazioni ammissibili su di esse. Questa funzionalità è largamente sfruttata nell'implementazione di collezioni generiche, le quali sono in grado di memorizzare elementi di tipo qualsiasi. (Le collezioni in generale e le collezioni generiche in particolare sono trattate nella seconda parte del volume.)

Lo studio dei tipi generici va oltre lo sopo di questo testo; qui ci limiteremo a presentare un semplice esempio che ne sfrutta le potenzialità e che riguarda l'altro grande ambito di impiego dei *generics*: i metodi generici.

8.1 Definizione di metodi generici

Nel Capitolo 7 è stato presentato un metodo che implementa lo scambio dei valori tra due variabili `double`, metodo del quale riportiamo il codice:

```
static void Scambia (ref double x, ref double y)
{
    double tmp = x;
    x = y;
    y = tmp;
}
```

Lo scambio del contenuto di due variabili rappresenta un tipico esempio di procedimento completamente indipendente dal tipo dei dati processati. Ma il linguaggio impone che la sua implementazione faccia riferimento ad un tipo specifico, che nell'esempio è `double`. Ciò rappresenta ovviamente un grosso limite, poiché, se si desidera utilizzare l'agoritmo di scambio con dati di diverso tipo, siamo costretti a implementare un metodo per ogni tipo di dato utilizzato.

L'impiego dei tipi generici, più precisamente dei metodi generici, fornisce una soluzione ideale al problema:

```
static void Scambia<T>(ref T a, ref T b)
{
    T tmp = a;
```

```
    a = b;
    b = tmp;
}
```

Il codice del nuovo metodo è perfettamente identico a quello del metodo `Scambia()` originale, poiché identico è il procedimento; la grande differenza consiste nel fatto che `Scambia<>()` non fa riferimento a nessun tipo in particolare ma ad un generico tipo T.

Attraverso le parentesi angolari, un metodo generico introduce uno o più identificatori che designano genericamente dei tipi (*type paramater*). Nella lista parametri e nel corpo del metodo è possibile utilizzare questi identificatori come normali tipi di dati[19].

8.2 Uso di metodi generici

La definizione di un metodo generico descrive un procedimento senza stabilire il tipo dei dati ai quali viene applicato. E' nell'uso del metodo che i tipi concreti vengono finalmente specificati. Il seguente codice mostra come applicare il metodo `Scambia<>()` sia a variabile `double` che `int`.

```
class Program
{
    static void Scambia<T>(ref T a, ref T b)
    {
        T tmp = a;
        a = b;
        b = tmp;
    }

    static void Main()
     {
        double x = 10;
        double y = 20;
        Scambia<double>(ref x, ref y);
        ...
        int a = 1;
        int b = 2;
        Scambia<int>(ref a, ref b);
        ...

     }
}
```

Nelle istruzioni di invocazione, tra parentesi angolari viene specificato il tipo concreto che dovrà essere utilizzato nell'esecuzione del metodo. Il tipo specificato sostituira l'identificatore di tipo (T, nell'esempio) utilizzato nella definizione del metodo.

[19] Nella realtà le cose sono un po' più complicate, poiché esistono dei vincoli sulle operazioni che è possibili eseguire, ma il concetto di fondo resta valido.

9

Strutture ed Enumeratori

Nei programmi considerati finora i dati sono stati rappresentati mediante oggetti appartenenti ai tipi predefiniti: `int`, `double`, `string`, eccetera, o array degli stessi: `int[]`, `double[]`, `string[]`, eccetera. L'impiego dei tipi predefiniti può essere sufficiente in programmi semplici, ma diventa una forte limitazione quando si devono affrontare problemi realistici.

Vediamo un esempio. Si vuole memorizzare i dati dei dipendenti di un'azienda; di ogni dipendente si desidera rappresentare: nominativo, codice mansione, ore di straordinario.

E' richiesta dunque l'elaborazione di un elenco di oggetti – i dipendenti – ognuno dei quali è caratterizzato da tre attributi: nominativo, codice mansione, ore straordinario. Una soluzione abbastanza plausibile per la memorizzazione dei dati prevede tre vettori, uno per i nominativi, uno per i codici di mansione e il terzo per le ore di straordinario. Questa situazione può essere così schematizzata:

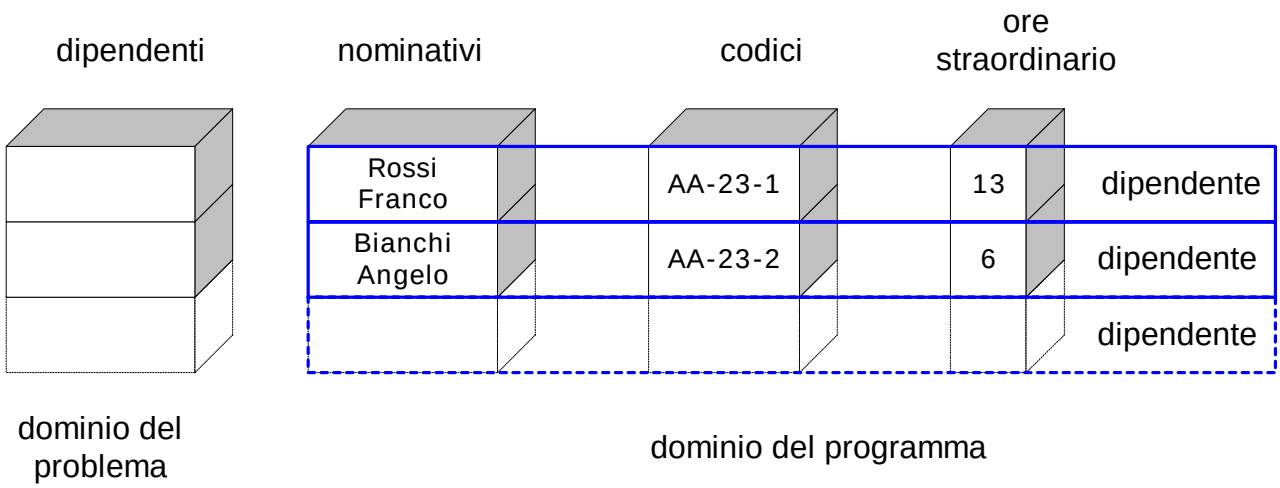


Figura 9-1 Dominio del problema e sua rappresentazione nel programma.

Ogni dipendente rappresenta un oggetto ben definito, ma nella sua rappresentazione all'interno del programma, questa unità viene persa. Il programma definisce tre vettori all'interno dei quali sono "disperse" le informazioni sui dipendenti; sarebbe invece desiderabile poter rappresentare ogni dipendente come un oggetto ben definito, mantenendo la possibilità di elaborare agli attributi che lo caratterizzano.

Facciamo un secondo esempio, completamente diverso. In un programma che gestisce una agenda di appuntamenti si desidera stabilire una codifica per i giorni della settimana. Un modo ovvio per farlo è usare dei numeri interi: 1 per il lunedì, 2 per il martedì, eccetera. Una simile soluzione è senz'altro efficiente, ma produce un codice poco leggibile. Si può migliorarlo usando delle costanti simboliche; ad esempio:

```
const int LUNEDI = 1;
const int MARTEDI = 2;
...
```

Ma resta il fatto che il tipo delle variabili utilizzate per elaborare i giorni della settimana è `int`; non esiste cioè un tipo `GiornoSettimana`, che caratterizzi le variabili appartenenti ad esso come oggetti che rappresentano un giorno della settimana.

Come per qualsiasi linguaggio di programmazione orientato agli oggetti, anche in C# la possibilità di definire nuovi tipi di dati rappresenta un aspetto fondamentale della programmazione e consente di ottenere una rappresentazione accurata delle informazioni che caratterizzano il problema. Poiché la programmazione *object oriented* viene trattata in un altro volume, in questo capitolo ci limitiamo a trattare in modo introduttivo due elementi del linguaggio che consentono al programmatore di definire e utilizzare nuovi tipi di dati: **tipi struttura** ed **enumeratori**

1 Tipi struttura

Gli aspetti che caratterizzano un tipo di dato sono memorizzazione, codifica, rappresentazione delle costanti, insieme delle operazioni ammissibili, relazioni (conversione da e per) con altri tipi. Un **tipo struttura** consente di definire un nuovo tipo di dato e di stabilire un sotto insieme degli aspetti sopra citati. Qui ci limiteremo all'aspetto della memorizzazione, tralasciando gli altri.

1.1 Definizione di un tipo struttura

I tipi struttura vengono anche definiti **aggregati**, poiché un oggetto struttura è composto da uno o più oggetti: appartenenti ai tipi predefiniti, ad altri tipi struttura, a enumeratori, eccetera. Definire una struttura significa dunque dichiarare le variabili che compongono gli oggetti appartenenti ad essa. La definizione assume la seguente forma[20]:

```
struct nome-struttura
{
    public tipo nome-variabile1;
    public tipo nome-variabile1;
    public tipo nome-variabilen;
    ...
}
```

Analogamente alle variabili dichiarate a livello di classe, anche le variabili dichiarate in un tipo struttura sono convenzionalmente **campi membro**.

Ecco un esempio di tipo struttura:

```
struct Persona
{
   public string Nome;
   public string Cognome;
   public int Eta;
}
```

Una simile definizione introduce nel programma un nuovo tipo di dato, `Persona` appunto. Ogni variabile di tipo `Persona` è composta internamente da tra campi e cioè da tre variabili distinte:

[20] In realtà questa forma è ipersemplificata rispetto a quella generale. Sono stati tralasciati tutti gli aspetti non inerenti al livello di trattazione utilizzato nel paragrafo.

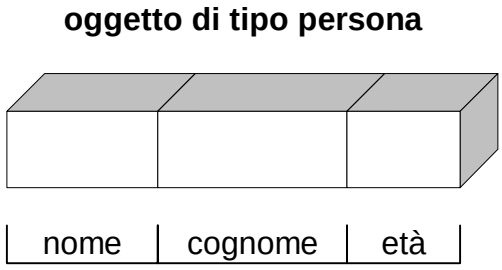


Figura 9-2 Rappresentazione di un oggetto di tipo Persona.

Collocazione della definizione di un tipo struttura

Negli esempi considerati finora, sia il codice dichiarativo che esecutivo è sempre stato collocato all'interno della classe principale (e unica) del programma. I tipi struttura possono essere definiti sia all'interno che all'esterno della classe, e considerato il tipo dei programmi finora realizzati tra le due scelte non esiste alcuna differenza.

1.2 Dichiarazione di variabili struttura

Definire nuovi tipi non produce alcuna influenza sull'esecuzione del programma. Un tipo rappresenta soltanto un progetto, una descrizione delle caratteristiche che hanno gli oggetti appartenenti ad esso. E' dichiarando e utilizzando oggetti appartenenti al tipo che tale progetto viene applicato. La dichiarazione di una variabile struttura assume la stessa forma di qualsiasi altra dichiarazione di variabile:

nome-tipo nome-variabile;

Dato il precedente tipo `Persona`, ecco alcuni esempi di dichiarazioni.

```
Persona p;
Persona persona;
Persona[] elencoDipendenti;
```

Naturalmente, una variabile struttura può essere dichiarata anche a livello di classe:

```
static Persona dipendente;
```

In tutti i casi, la semplice dichiarazione produce l'allocazione in memoria dell'oggetto, che non necessita di essere creato mediante l'operatore `new`, e ciò perché i tipi struttura appartengono alla categoria dei tipi valore.

Valori iniziali di variabili struttura

Lo stato iniziale di una variabile struttura è definito dai valori iniziali dei suoi campi. Questi, a loro volta, sono stabiliti in base alle stesse regole del linguaggio applicate alle variabili appartenenti ai tipi valore, e cioè:

- se la variabile struttura è locale, il valore iniziale dei campi è indefinito;
- se la variabile è a livello di classe il valore iniziale dei campi è quello di default stabilito dal tipo di appartenenza.

Anche le variabili struttura possono essere inizializzate in fase di dichiarazione. D'altra parte, poiché la modalità di inizializzazione è diversa da quella utilizzata per le variabili appartenenti ai tipi predefiniti e richiede un uso *object oriented* del linguaggio non sarà trattata in questa sede.

1.3 Uso di variabili struttura

L'uso di una variabile struttura può avvenire in due forme diverse:

- viene usata come qualsiasi variabile semplice di tipo `int`, `double`, eccetera, ad esempio come oggetto di un'assegnazione o come argomento di un metodo;
- viene utilizzata per accedere ai suoi campi.

Segue un frammento di codice che mostra entrambe le forme:

```
Persona fisico;
fisico.Nome = "Enrico";
fisico.Cognome = "Fermi";                          // accesso ai campi
fisico.Eta = 34;

Persona fisico2 = fisico;                          // assegnazione variabile

fisico2.Nome = "Albert";
fisico2.Cognome = "Einstein";                      // accesso ai campi
Console.WriteLine("Nome: {0}  Cognome: {1}  Eta: {2}",
                  fisico2.Nome, fisico2.Cognome, fisico2.Eta);
```

Il frammento è diviso in tre parti. Nella prima e nella terza parte vengono assegnati dei valori ai campi della variabile `fisico`. Come si vede, la sintassi da usare per l'accesso a un campo è:

`variabile-struttura.nome-campo`

Nella seconda parte, alla variabile `fisico2` viene assegnata la variabile `fisico`. L'esecuzione di questa istruzione produce una copia di tutti i campi di `fisico` nei corrispondenti campi di `fisico2`.

2 Esempio d'uso dei tipi struttura

L'uso dei tipi struttura è appropriato ogni qual volta vi sono gruppi di dati, logicamente correlati tra loro, che caratterizzano un determinato soggetto. Nel codice di esempio precedente, i dati da elaborare sono due stringhe e un intero; questi non sono indipendenti tra loro, ma rappresentano gli attributi – nome, cognome ed età – di un individuo. In una situazione simile, nulla ci impedisce di memorizzare i dati indipendentemente, ad esempio creando un elenco di persone mediante l'impiego di tre vettori o di una matrice, ma così facendo andrebbe persa la corrispondenza tra i dati del problema e la loro rappresentazione nel programma.

Come esempio di applicazione dei tipi struttura, prenderemo nuovamente in considerazione il problema presentato nell'introduzione del capitolo.

2.1 Gestione delle ore di straordinario dei dipendenti di una azienda

Lo scopo del programma è calcolare il totale delle ore di straordinario dei dipendenti di un'azienda e visualizzare un prospetto che mostri, per ogni dipendente, nominativo, codice mansione e ore di straordinario lavorate.

L'aspetto che qui ci interessa è quello della rappresentazione dei dipendenti. Una soluzione senz'altro migliore di quella accennata nell'introduzione è rappresentata dall'implementazione di un tipo struttura, `Dipendente`, che definisce gli attributi caratterizzanti ogni dipendente.

Segue un programma dimostrativo. Del metodo `Main()` è mostrata solo una parte del codice di creazione del vettore che memorizza i dipendenti e di assegnazione dei valori ai campi di ogni dipendente.

using System; Esempio 9.1

```
struct Dipendente
{
    public string Nominativo;
    public string Codice;
    public int Ore;
}

class Program
{
    static void Main()
    {
        Dipendente[] dipendenti = new Dipendente[3];
        Dipendenti[0].Nominativo = "Tizio";
        Dipendenti[0].Codice = "AA-23-1";
        Dipendenti[0].Ore = 12;
        // le precedenti 3 istruzioni sono da ripetere per i dipendenti 1 e 2

        VisualizzaProspetto();

    }

    static void VisualizzaProspetto(Dipendente[] dipendenti)
    {
        Console.WriteLine("Prospetto delle ore di straordinario\n");
        int oreTotali = 0;
        foreach(Dipendente dip in dipendenti)
        {
            oreTotali += dip.Ore;
            Console.WriteLine("{0,-20} {1,-10} {2}", dip.Nominativo, dip.Codice,
                                                     dip.Ore);
        }
        Console.WriteLine("Ore totali di straordinario: {0}", oreTotali);

    }
}
```

Il tipo `Dipendente` si usa come qualsiasi tipo predefinito, e ciò significa ad esempio che è possibile utilizzare variabili iteratore di tipo `Dipendente` in cicli `foreach()`, oppure dichiarare vettori di oggetti `Dipendente`, eccetera.

Continuando il confronto con i tipi predefiniti, si può notare che il codice non contiene espressioni con variabili di tipo `Dipendente` né conversioni da/per variabili di altro tipo; ciò, al di là del fatto che non è richiesto dal problema, non rappresenta un limitazione del linguaggio, ma è dovuto al carattere introduttivo di questo capitolo. I tipi struttura, infatti, ammettono la definizione

di operatori e metodi di conversione, i quali consentono la manipolazione di variabili appartenenti ai tipi suddetti nello stesso in cui è possibile manipolare variabili appartenenti ai tipi, `int`, `double`, `char` o `string`.

3 Tipi e variabili enumeratore

Un tipo enumeratore consente di definire un insieme di identificatori associandoli a dei valori interi. L'uso di tipi e variabili enumeratore migliora la qualità del codice, poiché consente di fare riferimento a dei nomi descrittivi in luogo di numeri, rendendo così il programma più leggibile.

3.1 Definizione di un tipo enumeratore

La definizione di un enumeratore assume la seguente sintassi generale:

```
modificatori_opz enum nome-enumeratore :tipo-sottostante_opz
{
     identificatore_1 = inizializzatore_opz,
     identificatore_2 = inizializzatore_opz,
     identificatore = inizializzatore_opz n
}
```

ma qui faremo riferimento alla seguente forma semplificata:

```
enum nome-enumeratore
{
     identificatore_1,
     identificatore_2,
     identificatore_n
}
```

Ad ogni identificatore che fa parte di un enumeratore il linguaggio assegna un valore intero progressivo partendo da 0. Ecco un esempio di tipo enumeratore:

```
enum Sesso
{
    Maschile,
    Femminile
}
```

Esso introduce un nuovo tipo di dato, `Sesso` appunto, che definisce due costanti, `Maschile` e `Femminile`, chiamate **costanti enumeratore**; ad esse vengono associati i valori 0 e 1. Nel codice sorgente è possibile fare riferimento ai due identificatori premettendo il nome dell'enumeratore, secondo la sintassi:

```
tipo-enumeratore.identificatore
```

Un tipo enumeratore può essere definito sia nel corpo della classe principale (come di qualsiasi altra classe) sia al di fuori essa. Ancora una volta, considerata la struttura generale dei programmi finora realizzati, tra le due scelte non esiste alcuna differenza.

3.2 Dichiarazione di variabili enumeratore

La dichiarazione di una variabile enumeratore assume la stessa forma di qualsiasi altra dichiarazione di variabile:

nome-tipo nome-variabile;

Dato il precedente tipo `Sesso`, ecco alcuni esempi di dichiarazioni.

```
Sesso s;
Sesso sesso;
Sesso[] sessoDipendenti;
```

Naturalmente, una variabile enumeratore può essere dichiarata anche a livello di classe:

```
static Sesso s;
```

La semplice dichiarazione produce l'immediata allocazione in memoria dell'oggetto, poiché i tipi enumeratore appartengono alla categoria dei tipi valore. Una variabile enumeratore memorizza in realtà un numero intero e dunque occupa 4 byte di memoria.

Valori iniziali e inizializzazione di variabili enumeratore

Il valore iniziale di una variabile enumeratore è stabilito in base alle regole del linguaggio applicate alle variabili appartenenti ai tipi valore, e cioè:

- se la variabile è locale, il valore iniziale è indefinito;
- se la variabile è un campo di classe il valore iniziale è quello di default stabilito dal tipo di appartenenza, e cioè la costante enumeratore corrispondente al valore 0.

Le variabili enumeratore possono essere inizializzate con valori costanti in fase di dichiarazione per fornire un valore diverso da quello di default nel caso di un campo di classe, o per fornire un valore iniziale a una variabile locale. Ad esempio:

```
Sesso s = Sesso.Femminile;
```

Da notare che è possibile fornire dei valori iniziali anche a un array di elementi enumeratori:

```
Sesso[] persone = {Sesso.Femminile, Sesso.Femminile, Sesso.Maschile};
```

3.3 Uso di variabili enumeratore

Le variabili enumeratore memorizzano un valore intero e quindi, in alcune circostanze, possono essere manipolate come variabili intere. D'altra parte i tipi enumeratori restano distinti dal tipi `int` e ciò è dimostrato dal fatto che non sono ammesse conversioni implicite da/per tipi enumeratori e tipo `int`, né sono ammesse tutte le operazioni possibili con il tipo `int`, come ad esempio moltiplicazione e divisione.

> Mediante l'operatore di cast è comunque possibile effettuare delle conversioni esplicite tra il tipo `int` e un qualsiasi tipo enumeratore.

Le operazioni più comuni effettuate su variabili enumeratore sono l'assegnazione di una costante e il confronto per uguaglianza con una o più costanti definite dal tipo di appartenenza. Ad esempio, il seguente frammento di codice, dopo aver inizializzato un vettore di elementi di tipo `Sesso`, conta quanti di essi memorizzano il valore `Femminile` e quanti il valore `Maschile`:

```
Sesso[] persone = {Sesso.Femminile, Sesso.Maschile, Sesso.Maschile};
int donne = 0;
```

```
int uomini = 0;
for(int i = 0; i < persone.Length; i++)
    if (persone[i] == Sesso.Femminile)
        donne++;
    else
        uomini++;

Console.WriteLine("Ci sono {0} donne e {1} uomini", donne, uomini);
```

Naturalmente il ciclo `for()` può essere sostituito da un più compatto `foreach()`:

```
...
foreach(Sesso s in persone)
    if (s == Sesso.Femminile)
        donne++;
    else
        uomini++;
...
```

Poiché, dal punto di vista della memorizzazione, i tipi enumeratori sono equivalenti al tipo `int`, il codice precedente viene così tradotto dal linguaggio:

```
int[] persone = {1, 0, 0};
int donne = 0;
int uomini = 0;
foreach(int s in persone)
    if (s == 1)
        donne++;
    else
        uomini++;
Console.WriteLine("Ci sono {0} donne e {1} uomini", donne, uomini);
```

Dove sta dunque la differenza? La risposta è nella maggior comprensibilità e facilità di modifica del codice. L'uso di identificatori come `Femminile` e `Maschile` uniti al nome del tipo `Sesso` rende esplicito il ruolo di queste costanti nonché delle variabili appartenenti al tipo in questione.

3.4 Conversione da enumeratore a stringa

I valori appartenenti agli enumeratori, siano essi costanti o variabili, possono essere convertiti in stringa mediante il metodo `ToString()`; il risultato della conversione è una stringa contenente il nome dell'identificatore associato al valore convertito. Ad esempio, l'output prodotto dal seguente codice:

```
Sesso s = Sesso.Femminile;
Console.WriteLine(s.ToString());               // conversione variabile;
Console.WriteLine(Sesso.Maschile.ToString());  // conversione costante
```

è:

```
Femminile
Maschile
```

3.5 Ottenere i nomi di tutte le costanti definite da un tipo enumeratore

Un'altra funzionalità fornita dagli enumeratori consente di ottenere l'elenco dei nomi associati alle costanti definite da un tipo enumeratore. Allo scopo esiste il metodo `GetNames()`, che rispecchia la seguente sintassi:

```
Enum.GetNames(typeof(tipo-enumeratore))
```

Il metodo ritorna un vettore di stringhe contenente i nomi degli identificatori. Ad esempio, il seguente codice:

```
string[] nomi = Enum.GetNames(typeof(Sesso));
foreach(string nome in nomi)
    Console.WriteLine(nome);
```

produce sullo schermo:

```
Maschile
Femminile
```

3.6 Conversione da stringa a variabile enumeratore

Anche per gli enumeratori esiste il metodo `Parse()`, anche se il suo impiego non è altrettanto immediato di quello utilizzato con i tipi `int`, `double`, ecc. Segue la sintassi di chiamata del metodo, che richiede l'uso sia dell'operatore `typeof` che dell'operatore di cast per indicare il tipo enumeratore della variabile che memorizzerà il risultato:

```
(tipo-enumeratore) Enum.Parse(typeof(tipo-enumeratore), stringa)
```

Ad esempio, il seguente codice memorizza nella variabile `s` la costante enumeratore corrispondente alla stringa `"Maschile"`:

```
Sesso s = (Sesso) Enum.Parse(typeof(Sesso), "Maschile");
```

Se la stringa specificata come argomento non corrisponde a nessuna delle costanti del tipo enumeratore specificato, viene prodotto un errore di esecuzione.

Da notare che il metodo `Parse()` distingue tra maiuscole e minuscole e dunque, ad esempio, specificare la stringa "maschile" piuttosto che "Maschile" non è affatto la stessa cosa: nel primo caso si ottiene un errore di esecuzione. In questo senso, esiste una versione del metodo che consente di ignorare la differenza di case tra le lettere, passando come terzo argomento il valore `false`.

Ad esempio:

```
Sesso s = (Sesso) Enum.Parse(typeof(Sesso), "maschile", false);
```

4 Esempio d'uso dei tipi enumeratore

L'uso di tipi enumeratori è appropriato quando i dati da rappresentare variano all'interno di un intervallo limitato di cui ogni valore possiede un significato preciso, tanto che per questo motivo può essere identificato da un nome. Ciò è ancor più vero quando per l'intervallo in questione non esiste un corrispondente intervallo numerico che abbia un significato.

Ad esempio, codificare il sesso di una persona con i numeri 0 (maschile) e 1 (femminile) rappresenta una scelta puramente arbitraria; altri due numeri sarebbero altrettanto adatti (o inadatti). In questo caso, la definizione di un tipo enumeratore rappresenta praticamente un requisito.

Il discorso cambia per i mesi dell'anno, poiché essi seguono un ordine preciso e oltre che con i nomi sono indicati (da tutti) mediante i numeri che vanno da 1 a 12. Anche in questo caso può essere appropriato l'uso di un tipo enumeratore, ma ciò dipende dal tipo di operazioni che è necessario svolgere.

4.1 Gestione di un elenco anagrafico

Come esempio d'uso degli enumeratori considereremo l'ipotesi di dover memorizzare alcuni dati anagrafici di un elenco di persone. Tali dati sono: nominativo, domicilio, sesso e stato civile. Per rappresentare accuratamente i dati del problema occorre definire tre nuovi tipi:

- un tipo struttura che definisce i quattro attributi sopra elencati;
- un tipo enumeratore che rappresenta il sesso;
- un tipo enumeratore che rappresenta lo stato civile.

Per quanto riguarda lo stato civile decidiamo di indentificare la condizione di una persona indipendentemente dal suo sesso e quindi di definire il seguente tipo enumeratore:

```
enum StatoCivile
{
    Celibe_nubile,
    Coniugato_a,
    Vedovo_a,
    Separato_a,
    Divorziato_a,
    Tutelato_a,
    Minore
}
```

Il programma che segue si limita a memorizzare i dati di alcune persone in un vettore e quindi a visualizzare l'elenco.

```
using System;                                                    Esempio 9.1

struct Persona
{
    public string Nominativo;
    public string Domicilio;
    public Sesso Sesso;
    public StatoCivile StatoCivile;
}
enum StatoCivile
{
    celibe_nubile,
    coniugato_a,
    vedovo_a,
    separato_a,
    divorziato_a,
    tutelato_a,
    minore
}
```

```
enum Sesso
{
    Maschile,
    Femminile
}
class Program
{
    static void Main()
    {
        Persona[] elenco = new Persona[3];
        elenco[0].Nominativo = "Bianchi Aldo";
        elenco[0].Domicilio = "Via della Fortezza, 34";
        elenco[0].Sesso = Sesso.Maschile;
        elenco[0].StatoCivile = StatoCivile.Vedovo_a;
        // le precedenti 4 istruzioni sono da ripetere per le persone 1 e 2

        VisualizzaElenco();

    }

    static void VisualizzaElenco(Persona[] elenco)
    {
        Console.WriteLine("Elenco anagrafico\n");
        foreach(Persona persona in elenco)
        {
            string strSesso = SessoToString(persona.Sesso);
            string strStatoCivile = StatoCivileToString(persona.Sesso,
                                persona.StatoCivile);
            Console.WriteLine("{0,-20} {1,-25} [{2}] {3}", persona.Nominativo,
                                                    persona.Domicilio,
                                                   strSesso, strStatoCivile);
        }

    }

    static string SessoToString(Sesso sesso)
    {
        if (sesso == Sesso.Femminile)
            return "F";
        else
            return "M";
    }

    static string StatoCivileToString(Sesso sesso, StatoCivile sc)
    {
        string tmp = "";
        switch (sc)
```

```
        {
            case StatoCivile.Coniugato_a: tmp = "CONIUGAT"; break;
            case StatoCivile.Divorziato_a: tmp = "DIVORZIAT"; break;
            case StatoCivile.Separato_a: tmp = "SEPARAT"; break;
            case StatoCivile.Tutelato_a: tmp = "TUTELAT"; break;
            case StatoCivile.Vedovo_a: tmp = "VEDOV"; break;
            case StatoCivile.Minore: return "MINORE";
            case StatoCivile.Celibe_nubile:
                if (sesso == Sesso.Femminile)
                    return "NUBILE";
                else
                    return "CELIBE";

        }
        if (sesso == Sesso.Femminile)
            return tmp + "A";
        else
            return tmp + "O";

    }
}
```

Il programma contiene due metodi per la conversione in stringa di valori di tipo `Sesso` e `StatoCivile`. Ciò è dovuto al fatto che sulla base delle scelte effettuate non esiste una corrispondenza esatta tra il nome dato alle costanti enumeratore e la rispettiva rappresentazione testuale. Nella realtà, questa corrispondenza non sempre è possibile, o comunque desiderabile.

Un secondo aspetto riguarda i nomi dei campi sesso e stato civile utilizzati nel tipo struttura `Persona` (righe evidenziate in grigio). Ebbene, questi coincidono con i nomi dei rispettivi tipi di appartenenza, creando apparentemente un conflitto. In realtà il linguaggio è in grado di desumere dal contesto di un'istruzione se un identificatore fa riferimento a un tipo o a una variabile. Nonostante ciò possa provocare qualche problema di leggibilità del codice, è questa una prassi largamente utilizzata dai progettisti di .NET ed è quindi conveniente adottarla.