

Giorno 2

Comprendere i programmi C#

Applicazioni C#

La prima parte della lezione di oggi verte su una semplice applicazione C#. Servendovi del Listato 2.1, comincerete a conoscere alcune delle parti chiave di un'applicazione C#.

Listato 2.1 app.cs. Esempio Applicazione C#

```
1: // app.cs. Un'applicazione C# campione
2: // Non preoccupatevi di capire tutto in
3: // questo listato. Comanderete meglio più avanti!
4: //-----
5:
6: using System;
7:
8: class sample
9: {
10:     public static void Main()
11:     {
12:         //Dichiara le variabili
13:
14:         int radius = 4;
15:         const double PI = 3.14159;
16:         double area;
17:
18:         //Fa i calcoli
19:
20:         area = PI * radius * radius;
21:
22:         //Stampa i risultati
23:
24:         Console.WriteLine("Radius = {0}, PI = {1}", radius, PI );
25:         Console.WriteLine("The area is {0}", area);
26:     }
27: }
```

Dovreste inserire questo listato nell'editor e usare quindi il vostro compilatore per creare il programma. Potete salvare il programma come `app.cs`; quando lo compilate, inserite la seguente stringa nel prompt di comando:

```
csc app.cs
```

In alternativa, se state usando un editor visivo, dovreste essere in grado di selezionare un compilatore dalle opzioni di menu.



Ricordate di non inserire il numero delle righe o i due punti quando state inserendo il listato visto sopra. La numerazione è una convenzione utilizzata in questo libro per facilitare l'individuazione delle righe nel corso della trattazione e dell'analisi dei listati.

Quando eseguite il programma, ottenete il seguente risultato:

```
Radius = 4, PI = 3.14159  
The area is 50.3344
```

Come potete osservare, vengono visualizzati i valori di un raggio, di PI e l'area di un cerchio basato su questi due valori. Nei paragrafi seguenti imparerete a utilizzare alcune delle numerose parti di questo programma. Non preoccupatevi di comprendere tutto.

Commenti

Le prime quattro righe del Listato 2.1 sono commenti impiegati per inserire nel programma informazioni a voi utili e ignorate dal compilatore. Perché dovreste volere inserire informazioni che il compilatore ignorerà? Vi sono una serie di ragioni.

I commenti vengono usati spesso per fornire informazioni descrittive riguardanti il vostro listato (per esempio, informazioni di identificazione). Inserendo i commenti, potete documentare i risultati di un listato. Infatti, anche se inizialmente sapete esattamente di che cosa tratta il listato, poiché lo avete scritto, in un secondo momento potreste non ricordarlo più. Se, poi, il vostro listato viene utilizzato da altri, i commenti saranno loro utili per comprendere la funzione del codice. I commenti possono essere usati anche per fornire la cronologia di revisione di un listato. In C# vi sono tre tipi di commenti:

- di una sola riga;
- di più righe;
- di documentazione.



I commenti vengono rimossi dal compilatore, quindi non si arreca alcun danno lasciandoli nel programma. Nel dubbio, è preferibile includere i commenti.

Commenti di una riga

Il Listato 2.1 utilizza commenti di una riga in ognuna delle righe da 1 a 4. Le righe 12, 18 e 22 contengono anche commenti di una riga e hanno il seguente formato:

```
// testo del commento
```

I due slash indicano che sta iniziando un commento. Da quel punto fino alla fine della riga corrente, ogni elemento viene trattato come un commento.

Non è obbligatorio che un commento di una riga debba cominciare all'inizio della riga. Potete, in realtà, scrivere il codice C# sulla stessa riga dei commenti; tuttavia dopo il doppio slash, il resto della riga è un commento.

Commenti di più righe

Il Listato 2.1 non contiene alcun commento di più righe, ma a volte potreste averne bisogno. In questo caso, potete iniziare ogni riga con due slash (come nelle righe 1-4 del listato) o usare più righe.

I commenti di più righe vengono creati con un contrassegno di inizio e fine. Per iniziare, inserite uno slash seguito da un asterisco:

```
/*
```

Tutto quello che segue quel contrassegno è un commento fino a quando non inserite un contrassegno finale, che è un asterisco seguito da uno slash:

```
*/
```

La riga che segue è un commento:

```
/* questo è un commento */
```

Anche la seguente:

```
/* questo è
un commento che
è su
una serie di
righe */
```

Potete inoltre inserire questo commento nel modo che segue:

```
// questo è
// un commento che
// è su
// una serie di
// righe
```

Con i commenti su più righe potete commentare una sezione di un listato codice semplicemente aggiungendo /* e */. Qualsiasi elemento appaia tra /* e */ viene ignorato dal compilatore come un commento.



Non potete annidare commenti multiriga, cioè posizionare un commento di più righe dentro un altro. Per esempio, questo è un errore:

```
/* Inizio di un commento...
   /* con un altro commento annidato */
*/
```

Commenti di documentazione

C# possiede uno speciale tipo di commento che vi consente di creare automaticamente documentazione esterna.

Tali commenti sono identificati con tre slash invece dei due usati per i commenti a riga singola; questi ultimi vengono impiegati anche per i tag di stile XML (*Extensible Markup Language*). XML è uno standard usato per marcare i dati. Sebbene possa essere impiegato qualsiasi tag XML valido, i tag più usati per C# sono <c>, <code>, <example>, <exception>, <list>, <para>, <param>, <paramref>, <permission>, <remarks>, <returns>, <see>, <seealso>, <summary>, e <value>.

Questi commenti si trovano nei vostri listati di codice. Il Listato 2.2 fornisce un esempio del loro impiego. Potete compilare questo listato dal momento che possedete quelli precedenti: consultate la lezione del Giorno 1, se avete bisogno di un ripasso.

Listato 2.2 xmlapp.cs. Usando i Commenti XML

```
1:  // xmlapp.cs. Un'applicazione C# campione che usa
2:  //          documentazione XML
3:  //-----
4:
5:  /// <summary>
6:  /// Questo è un sommario che descrive la classe .</summary>
7:  /// <remarks>
8:  /// Questo è un commento più lungo che può essere utilizzato per
9:  /// descrivere la classe .</remarks>
10: class MyApp
11: {
12:     /// <summary>
13:     /// Il punto di entrata dell'applicazione.
14:     /// </summary>
15:     /// <param name="args"> Un elenco di argomenti a linea di comando
16:     </param>
17:     public static void Main(string[] args)
18:     {
19:         System.Console.WriteLine("An XML Documented Program");
20:     }
```

Quando compilate ed eseguite il Listato 2.2, ottenete il seguente risultato:

```
An XML Documented Program
```

Per ottenere la documentazione XML, dovete compilare questo listato in maniera diversa da come lo avete fatto precedentemente e aggiungere il parametro /doc, quando compilate riga per riga.

Se compilate riga per riga, inserite:

```
csc /doc:xmlfile xmlapp.cs
```

In questo modo, otterrete esattamente lo stesso risultato del programma che avete eseguito precedentemente. Il file, però, è denominato `xmlfile` e contiene la documentazione in XML. Potete, comunque, sostituire `xmlfile` con qualsiasi altro nome. Per esempio, nel Listato 2.2 il file XML è:

```
<?xml version="1.0"?>
<doc>
  <assembly>
    <name>xmlapp</name>
  </assembly>
  <members>
    <member name="T:myApp">
      <summary>
        Questo è un sommario che descrive la classe .</summary>
      <remarks>
        Questo è un commento più lungo che può essere usato per
        descrivere la classe .</remarks>
    </member>
    <member name="M:myApp.Main(System.String[])">
      <summary>
        Il punto di entrata dell'applicazione.
      </summary>
      <param name="args"> Un elenco di argomenti a linea di comando
    </param>
    </member>
  </members>
</doc>
```



Se state usando uno strumento quale Visual Studio .NET, dovete controllare la documentazione o il sistema di supporto per imparare a generare la documentazione XML. Dovreste, comunque, essere sempre in grado di compilare i vostri programmi dalla riga di comando.

Parti fondamentali di un'applicazione C#

Un linguaggio di programmazione è composto da un gruppo di parole unite assieme. Un programma è la formattazione e l'uso di esse secondo un determinato metodo. Le parti di un linguaggio C# comprendono:

- spazio bianco;
- parole chiave C#;
- letterali;
- indentificatori.

Spazio bianco

TERMINE

Se osservate il Listato 2.1, potete constatare che è stato formattato in modo che il codice sia allineato e quindi agevole da leggere. Gli spazi vuoti inseriti nel listato sono denominati *spazi bianchi*, ciò vuol dire che su carta bianca non li vedrete. Si tratta di spazi, tab, interruzioni di riga e invii a capo.

Il compilatore ignora quasi sempre lo spazio bianco. Per questo motivo potete aggiungere quanti spazi, tab, interruzioni di riga e invii a capo volete. Per esempio, considerate la riga 14 del Listato 2.1:

```
int radius = 4;
```

È ben formattata e presenta un singolo spazio tra gli elementi, ma potrebbe avere anche dello spazio supplementare:

```
int      radius      =      4      ;
```

Questa riga con spazi extra viene eseguita nello stesso modo di quella precedente. Infatti, quando il compilatore esegue il programma, lo spazio bianco extra viene rimosso. Potreste inoltre formattare questo codice disponendolo su più righe:

```
int
radius
=
4
;
```

Sebbene sia poco leggibile, risulta ugualmente valido. Poiché lo spazio bianco viene generalmente ignorato, potete utilizzarlo liberamente per formattare il vostro codice rendendolo più leggibile.

Il compilatore fa un'unica eccezione per quanto riguarda lo spazio bianco, ed è il testo fra virgolette; in questo caso, lo spazio bianco viene conservato poiché il testo deve essere utilizzato esattamente come si presenta. Nel Listato 2.1, le righe 24 e 25 contengono testo virgolettato.



Usate lo spazio bianco per rendere più leggibile il vostro codice.

Parole chiave C#

Come indicato nella lezione del Giorno 1, le parole chiave sono termini specifici che hanno un significato determinato e quindi formano un linguaggio. Il linguaggio C# possiede una serie di parole chiave elencate nella Tabella 1.1.

Quando programmate in C#, le parole chiave hanno un significato specifico, che imparerete leggendo questo libro; per tale motivo sono riservate e non potete personalizzarle. Se confrontate la Tabella 1.1 al Listato 2.1 o a qualsiasi altro listato in questo libro, vedrete che contiene soprattutto parole chiave.

Letterali

TERMINE

I *letterali* sono valori stabiliti in maniera inequivocabile (*hard-coded*). Per esempio, i numeri 4 e 3,14159 sono entrambi letterali; anche il testo tra virgolette è letterale. La lezione del Giorno 3 si occupa in modo approfondito dei letterali e del loro uso.

Identificatori

TERMINE

Oltre alle parole chiave C# e ai letterali, altre parole vengono impiegate nei programmi C# e sono esattamente gli *identificatori*. Nel Listato 2.1 vi sono una serie di identificatori, incluso `System` nella riga 6, `sample` nella riga 8, `radius` nella riga 14, `PI` nella riga 15, `area` nella riga 16 e `PI`, `radius` e `area` nella riga 22.

Struttura di un'applicazione C#

Parole e proposizioni vengono usate per comporre delle frasi e le frasi per comporre dei paragrafi. Allo stesso modo, lo spazio bianco, le parole chiave, i letterali e gli identificatori vengono combinati insieme per creare espressioni e istruzioni che, a loro volta, costituiscono un programma.

Espressioni e istruzioni C#

TERMINE

Le *espressioni*, come le proposizioni, sono frammenti di codice costituiti da parole chiave:

```
PI = 3.14159
```

```
PI * radius * radius
```

Le *istruzioni*, di solito, terminano con un carattere di punteggiatura, un punto e virgola (;). Nel Listato 2.1, le righe 14, 15 e 16 sono esempi di istruzioni.

Istruzione vuota

Come sottolineato precedentemente, le istruzioni solitamente terminano con un punto e virgola. In realtà, potete posizionare un punto e virgola da solo su una riga. Questa istruzione non ha alcun effetto sul programma: viene, infatti, considerata un'affermazione vuota.

Potreste chiedervi che bisogno ci sia di includere un'istruzione che non comporta alcuna operazione; nel corso del libro, soprattutto nel Giorno 5, scoprirete quanto possa essere preziosa un'istruzione vuota.

Analisi del Listato 2.1

ANALISI

Vale la pena di analizzare dettagliatamente il Listato 2.1, ora che sono stati trattati diversi concetti importanti. I paragrafi seguenti prendono in esame ogni riga del Listato 2.1.

Righe 1-4: commenti

Come avete avuto modo di apprendere, le righe 1-4 contengono commenti che saranno ignorati dal compilatore, ma risultano particolarmente utili per chi utilizza il codice sorgente.

Righe 5, 7, 13, 17, 21 e 23: spazio bianco

La riga 5 è vuota: come indicato precedentemente, è uno spazio bianco che sarà ignorato dal compilatore. Viene inclusa per rendere più facile la lettura del listato. Le righe 7, 13, 17, 21 e 23 sono ugualmente vuote. Potete rimuovere queste righe dal vostro file sorgente senza creare alcun problema all'esecuzione del programma.

Riga 6: istruzione using

La riga 6 è un'istruzione che contiene la parola chiave `using` e un letterale `System`. Come tutte le istruzioni, termina con un punto e virgola. La parola chiave `using` è utilizzata per ridurre la quantità di codice del vostro listato. Di solito, la parola chiave `using` viene impiegata con i namespace. Per saperne di più, si veda il Giorno 6.

Riga 8: dichiarazione di classe

C# è un linguaggio di programmazione orientato agli oggetti (OOP, *Object-Oriented Programming*). I linguaggi orientati agli oggetti usano le classi per dichiarare oggetti. Questo programma definisce una classe denominata `sample`. Verrà fornita una panoramica generale sulle classi, più avanti, nella lezione odierna, ma l'argomento verrà trattato approfonditamente nella lezione del Giorno 6.

Righe 9, 11, 26 e 27: caratteri di punteggiatura

La riga 9 contiene una parentesi aperta, correlata alla parentesi chiusa della riga 27; lo stesso vale per la parentesi aperta della riga 11 e quella chiusa della riga 26. Questi gruppi di parentesi contengono e organizzano blocchi di codice. Imparando i differenti comandi, nei prossimi quattro giorni, avrete modo di vedere come si usano le parentesi.

Riga 10: Main ()

TERMINE

Il computer ha bisogno di sapere dove lanciare un programma. I programmi C# iniziano l'esecuzione dalla funzione `Main ()`, come nella riga 10. Una *funzione* è una porzione di codice che può venire eseguita richiamando il nome della funzione. Il Giorno 7 fornisce maggiori dettagli a riguardo. La funzione `Main ()` viene utilizzata quale punto di partenza.



L'impiego delle lettere maiuscole o minuscole nelle parole costituenti il vostro codice, è critico nelle applicazioni C#. `Main ()` ha solo la lettera M maiuscola. I programmatori C++ e C devono essere consapevoli che `main ()` (tutto minuscolo) non funziona in C#.

Righe 14, 15 e 16: dichiarazioni

Le righe 14, 15 e 16 contengono affermazioni usate per creare identificatori che conserveranno le informazioni. Questi vengono usati in un secondo momento per effettuare i calcoli. La riga 14 dichiara un identificatore per conservare il valore di un raggio. Il letterale `4` è assegnato a questo identificatore. La riga 15 crea un identificatore per memorizzare il valore dell'identificatore `PI`, che viene riempito con il valore di `3.14159`. La riga 16 dichiara un identificatore a cui non è attribuito alcun valore.

Riga 20: l'istruzione Assignment

La riga 20 contiene un'istruzione che moltiplica l'identificatore `PI` per il raggio alla seconda. Il risultato di questa espressione viene quindi assegnato all'identificatore `area`.

Riga 24 e 25: richiamo di funzioni

Le righe 24 e 25 sono le espressioni maggiormente complesse di questo listato. Queste due righe richiamano una routine predefinita che stampa le informazioni nella postazione (schermo). Le funzioni predefinite verranno trattate in seguito nella lezione odierna.

Programmazione orientata agli oggetti

Come abbiamo già detto precedentemente, C# è considerato un linguaggio orientato agli oggetti. I prossimi paragrafi trattano in modo sommario gli oggetti e le operazioni necessarie per creare un linguaggio orientato agli oggetti. Approfondirete come questi concetti vengono applicati a C# proseguendo nella lettura del libro.

Concetti orientati agli oggetti

Come creare un linguaggio orientato agli oggetti? Nel Giorno 1 si è fatto accenno a tre concetti che sono generalmente associati a questo linguaggio:

- incapsulamento
- polimorfismo
- ereditarietà

Ne esiste anche un quarto, tipico di un linguaggio orientato agli oggetti: il riuso.

Incapsulamento

Il concetto di incapsulamento consiste nel creare pacchetti contenenti ciò che vi è necessario; quindi, potete creare un oggetto (o pacchetto) come un cerchio, che fa qualsiasi cosa vogliate, compreso tenere traccia di tutto quello che riguarda il cerchio, come il raggio e il centro.

Con l'incapsulamento di un cerchio, fate in modo che l'utente non venga a conoscenza del funzionamento del cerchio; ciò consente di garantire protezione ai lavori interni all'oggetto in questione.

Polimorfismo

Il polimorfismo è la capacità di assumere molte forme e può essere applicato a due (se non a più) aree di una programmazione orientata agli oggetti. Prima di tutto, significa che potete istanziare un oggetto o una routine passandogli parametri diversi, ottenendo comunque lo stesso risultato. Se, per esempio, lavorate con un cerchio e volete nominare un oggetto tondo per ottenere la sua area, potreste utilizzare tre punti o un punto singolo e il raggio. In entrambe le soluzioni, otterreste lo stesso risultato. In un linguaggio di procedura quale C, invece, sarebbero necessarie due routine con due nomi diversi per ottenere l'area. In C# avete sempre due routine; tuttavia, potete attribuire loro lo stesso nome. Qualsiasi programma creato da voi o da altri, richiamerà semplicemente la routine cerchio e passerà la vostra informazione. Automaticamente il programma cerchio determina quale delle due routine usare. Verrà impiegata la routine corretta, in base alle informazioni passate. Gli utenti che nominano la routine, non devono preoccuparsi di quale impiegare.

La lezione del Giorno 11 fornisce maggiori informazioni a riguardo.

Ereditarietà

L'ereditarietà è il concetto orientato agli oggetti più complicato da comprendere. Creare un cerchio può risultare interessante, ma ottenere una sfera lo è ancora di più. Una sfera è solo un particolare tipo di cerchio, con l'unica differenza che possiede una terza dimensione. Infatti, usando il cerchio per creare la vostra sfera, questa erediterà le proprietà del cerchio.

Riutilizzo

Una delle ragioni chiave che giustifica la scelta di un linguaggio orientato agli oggetti, è la riutilizzo. Quando create una classe, potete riutilizzarla per creare molti oggetti. Usando l'ereditarietà e alcune delle caratteristiche descritte preceden-

temente, potete creare routine da utilizzare in vari programmi e in diversi modi. Incapsulando la funzionalità, ottenete routine testate e funzionanti.

Oggetti e classi

TERMINE

Ora che conoscete i concetti di un linguaggio orientato agli oggetti, è importante comprendere la differenza tra classe e oggetto. La *classe* è una definizione per l'oggetto che verrà creato. Detto in parole povere, le classi sono definizioni usate per creare oggetti.

Per descrivere le classi, spesso si ricorre alla metafora della formina per biscotti: non è un biscotto e non è commestibile, ma è semplicemente un attrezzo che viene impiegato per fare dolci.

Allo stesso modo, una classe può essere usata per creare vari oggetti. Per esempio, se create un programma per disegnare cerchi, vi servirete di una classe cerchio per creare molti oggetti cerchio.

Inoltre, potete avere molte altre classi: per esempio nome, card, applicazione, punto e cerchio.



Le classi e gli oggetti vengono trattati più dettagliatamente a partire dal Giorno 6. La lezione di oggi vi offre una panoramica generale sui concetti di orientamento agli oggetti.

Visualizzare le informazioni di base

Le routine per visualizzare le informazioni sono due:

- `system.Console.WriteLine()`
- `system.Console.Write()`

Queste due routine visualizzano informazioni sullo schermo; entrambe stampano le informazioni nella stessa maniera con un'unica piccola differenza: la routine `WriteLine()` le scrive su una nuova riga, mentre `Write()` non esegue tale operazione. Le informazioni che andate a visualizzare sullo schermo sono scritte tra parentesi. Se state visualizzando del testo, comprendete il testo tra parentesi e all'interno delle virgolette. Per esempio, il listato seguente visualizza il testo "Hello World":

```
System.Console.WriteLine("Hello World");
```



Questa routine l'avete utilizzata nella lezione del Giorno 1.

Inoltre, visualizza `Hello World` sullo schermo. I seguenti esempi illustrano altro testo che viene visualizzato:

```
System.Console.WriteLine("This is a line of text");
System.Console.WriteLine("This is a second line of text");
```

Se eseguite entrambe le righe consecutivamente, apparirà questo testo:

```
This is a line of text
This is a second line of text
```

Ora considerate le seguenti due righe:

```
System.Console.WriteLine("Hello ");
System.Console.WriteLine("World!");
```

Se vengono eseguite consecutivamente, che cosa verrà visualizzato? Se pensate che questo sia il risultato:

```
Hello World!
```

siete in errore! Infatti, quelle due righe stampano il seguente risultato:

```
Hello
World!
```

Notate che le due parole si trovano su righe differenti. Se eseguite due righe usando la routine `Write()`, otterrete il risultato desiderato.

```
Hello World!
```

Come potete osservare, la differenza tra le due routine è che `WriteLine()` passa automaticamente a una nuova riga, una volta che il suo testo viene visualizzato, mentre `Write()` non lo fa. Il Listato 2.3 mostra le due routine in azione.

Listato 2.3 `display.cs`. Usando `WriteLine()` e `Write()`

```
1: // display.cs. Stampare con WriteLine e Write
2: //-----
3:
4: class display
5: {
6:     public static void Main()
7:     {
8:         System.Console.WriteLine("First WriteLine Line");
9:         System.Console.WriteLine("Second WriteLine Line");
10:
11:         System.Console.Write("First Write Line");
12:         System.Console.Write("Second Write Line");
13:
14:         // Passare parametri
15:         System.Console.WriteLine("\nWriteLine: Parameter = {0}", 123 );
16:
17:         System.Console.Write("Write: Parameter = {0}", 456);
18:     }
19: }
```

Per compilare il Listato 2.3 dalla riga di comando, inserite:

```
csc display.cs
```

Se state usando uno strumento di sviluppo integrato, potete selezionare l'opzione di compilazione.

```
First WriteLine Line
Second WriteLine Line
First Write LineSecond Write Line
WriteLine: Parameter = 123
Write: Parameter = 456
```

ANALISI

Il Listato 2.3 utilizza la routine `System.Console.WriteLine()` nelle righe 8 e 9 per stampare due parti di testo. Ognuna di queste stampe è su una riga diversa. Le righe 11 e 12 mostrano la routine `System.Console.Write()`. Queste due righe stampano sulla stessa riga. Non vi è un *return line feed* dopo la stampa. Le righe 15 e 17 mostrano ognuna di queste routine con l'uso di un parametro.

Stampa di informazioni aggiuntive

Oltre a stampare del testo tra virgolette, potete passare i valori che devono essere stampati all'interno del testo. Considerate il seguente esempio:

```
System.Console.WriteLine("The following is a number: {0}", 456);
```

Il risultato è:

```
The following is a number: 456
```

Come potete osservare, lo `{0}` viene sostituito dal valore che segue il testo virgolettato. Il formato è:

```
System.Console.WriteLine("Text", value);
```

dove `Text` rappresenta praticamente qualsiasi tipo di testo vogliate visualizzare. `{0}` è un marcatore per un valore (le parentesi indicano appunto che è un marcatore). `0` è un indicatore per usare il primo oggetto che segue le virgolette. Una virgola separa il testo dal valore da posizionare nel marcatore.

Potete possedere più di un marcatore in una stampa: a ognuno viene attribuito un numero, in sequenza. Per stampare due valori, usate il seguente listato:

```
System.Console.Write("Value 1 is {0} and value 2 is {1}", 123, "Brad");
```

che restituisce:

```
Value 1 is 123 and value 2 is Brad
```

Imparerete di più su queste routine proseguendo nella lettura del volume.



Il primo marcatore ha come valore 0 e non 1.



Potete osservare un testo particolare nella riga 15 del Listato 2.3. \n non è un errore, ma indica che si dovrebbe iniziare una nuova riga prima di stampare le informazioni che seguono. Per saperne di più, si veda il Giorno 3.

Domande e risposte

D. Qual è la differenza tra “basato sui componenti” e “orientato agli oggetti”?

R. C# è basato sui componenti. Uno sviluppo di questo tipo può essere considerato l'estensione di una programmazione orientata agli oggetti. Un componente è semplicemente una parte di codice che esegue una specifica attività. La programmazione basata sui componenti consiste nel creare, appunto, dei componenti per poi riutilizzarli. Potete collegarli per costruire applicazioni.

D. Quali altri linguaggi sono considerati orientati agli oggetti?

R. C++, Java e SmallTalk. Anche Microsoft Visual Basic .NET può essere usato per una programmazione orientata agli oggetti. Esistono altri linguaggi, ma questi sono i più diffusi.

D. La composizione è un termine che riguarda la programmazione a oggetti?

R. Molti confondono ereditarietà con composizione; tuttavia si tratta di due concetti differenti. Con la composizione, un oggetto può venir utilizzato all'interno di un altro. La Figura 2.1 mostra una composizione di molti cerchi, differente dalla sfera dell'esempio precedente, in quanto la sfera non è composta da un cerchio, ma è una sua estensione. Quindi, la composizione avviene quando una classe (oggetto) ne contiene un'altra. L'ereditarietà avviene quando una classe (oggetto) è un'espansione di un'altra.

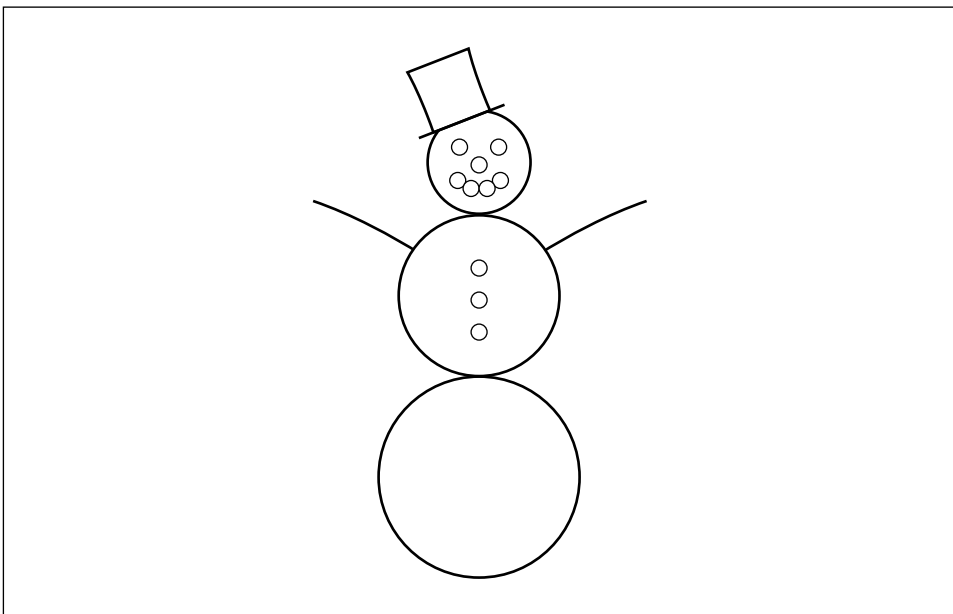


Figura 2.1

Un uomo di neve composto di cerchi.

Verifica

Questo paragrafo fornisce quiz ed esercizi per consolidare la comprensione dei temi trattati nel corso della lezione. Le risposte sono riportate nell'Appendice A.

Quiz

1. Quali sono i tre tipi di commenti che potete utilizzare in un programma C#?
2. In quale modo vengono inseriti questi tre tipi di commenti in un programma C#?
3. Quale impatto ha uno spazio bianco su un programma C#?
4. Quali delle seguenti sono parole chiave?

field, cast, as, object, throw, baseball, catch, football, fumble, basketball

5. Qual è la funzione di un letterale?
6. Quale di queste affermazioni è vera?
 - Le espressioni sono composte da istruzioni.
 - Le istruzioni sono composte da espressioni.
 - Espressioni e istruzioni non hanno niente a che vedere l'una con l'altra.
7. Che cos'è un'istruzione vuota?
8. Quali sono i concetti chiave di programmazione orientata agli oggetti?
9. Qual è la differenza tra `WriteLine()` e `Write()`?
10. Quale elemento viene utilizzato come marcatore, quando si stampa un valore con `WriteLine()` e `Write()`?

Esercizi

1. Inserite e compilate il seguente programma: `listit.cs`. Ricordate di non inserire i numeri di riga.

```

1: // ListIT.cs. Programma per stampare un listato con numeri di
↳riga
2: //-----
3:
4: using System;
5: using System.IO;
6:
7: class ListIT
8: {
9:     public static void Main(string[] args)
10:    {
11:        try
12:        {
13:
14:            int ctr=0;
15:            if (args.Length <= 0 )

```

```
16:         {
17:             Console.WriteLine("Format: ListIT filename");
18:             return;
19:         }
20:     else
21:     {
22:         FileStream f = new FileStream(args[0],
↳ FileMode.Open);
23:         try
24:         {
25:             StreamReader t = new StreamReader(f);
26:             string line;
27:             while ((line = t.ReadLine()) != null)
28:             {
29:                 ctr++;
30:                 Console.WriteLine("{0}: {1}", ctr, line);
31:             }
32:             f.Close();
33:         }
34:         finally
35:         {
36:             f.Close();
37:         }
38:     }
39: }
40: catch (System.IO.FileNotFoundException)
41: {
42:     Console.WriteLine ("ListIT could not find the
↳ file {0}", args[0]);
43: }
44:
45: catch (Exception e)
46: {
47:     Console.WriteLine("Exception: {0}\n\n", e);
48: }
49: }
50: }
```

2. Eseguite il programma inserito nell'Esercizio 1. Che cosa accade quando lo eseguite? Ora lanciatelo una seconda volta, inserendo la riga di comando che segue:

```
listit listit.cs
```

Qual è il risultato di quest'ulteriore operazione?

3. Inserite, compilate e applicate il programma che segue e analizzatene gli effetti.

```
1: // ex0203.cs - Esercizio 3 per il Giorno 2
2: //-----
3:
4: class Exercise2
5: {
6:     public static void Main()
```

```
7:      {
8:          int x = 0;
9:
10:         for( x = 1; x <= 10; x++ )
11:             {
12:                 System.Console.Write("{0:D3} ", x);
13:             }
14:     }
15: }
```

4. Il programma riportato di seguito ha un problema. Inserirlo nel vostro editor e compilatelo. Quali sono le righe che generano messaggi di errore?

```
1: // bugbust.cs
2: //-----
3:
4: class bugbust
5: {
6:     public static void Main()
7:     {
8:         System.Console.WriteLine("\nA fun number is {1}", 123 );
9:     }
10: }
```

5. Compilate la riga di codice che stampa il vostro nome sullo schermo.

