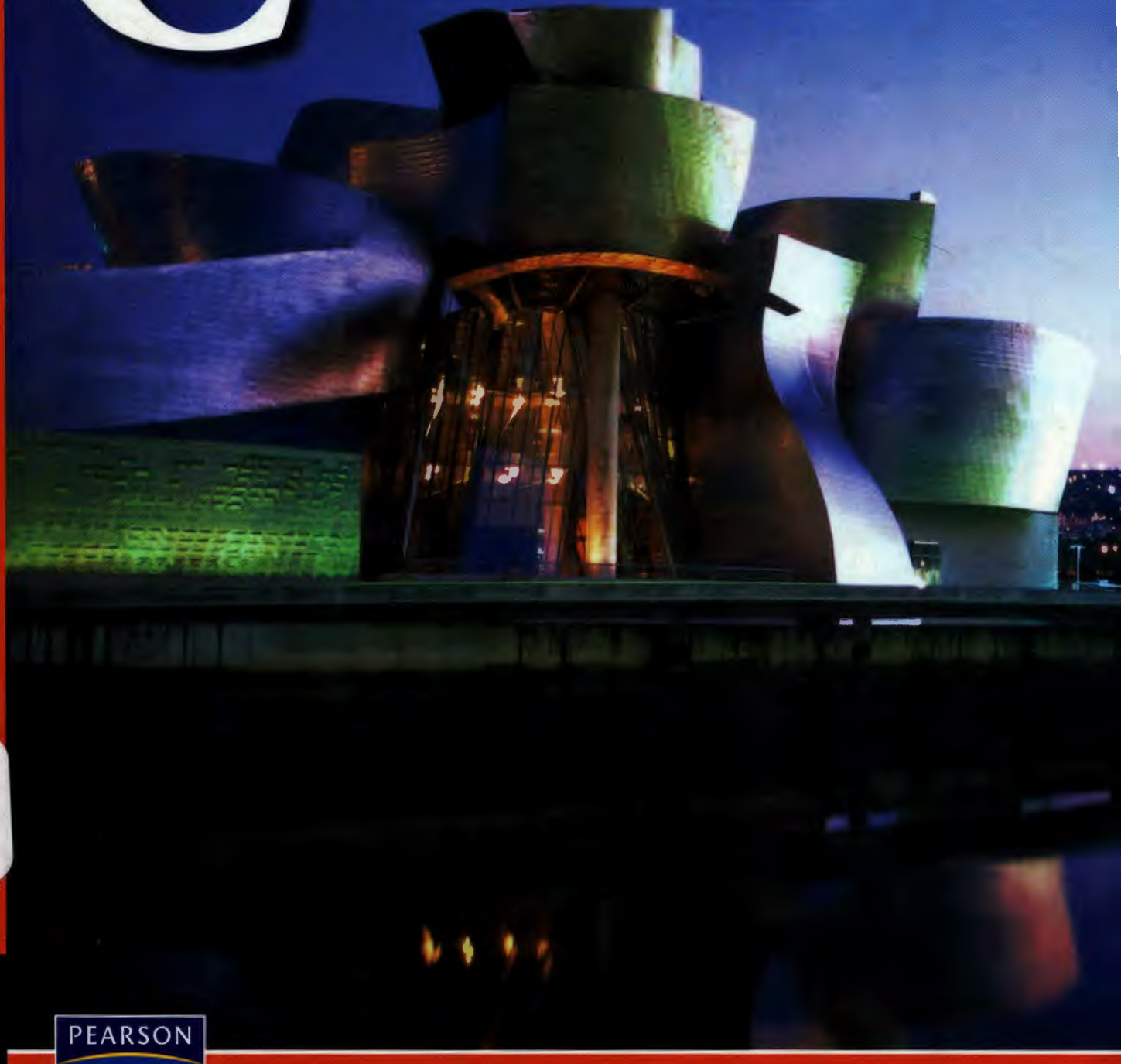
Referencia bibliográfica

Deitel, Harvey M. y Deitel, Paul J. (2007). *Cómo programar en C#.* (2a. ed). México: Pearson Educación. Pp. 118-123, 145-164.

C#

Segunda edición

CÓMO PROGRAMAR



PEARSON Prentice Hall

HARVEY M. DEITEL y PAUL J. DEITEL

SEGUNDA EDICIÓN

CÓMO PROGRAMAR EN C#

Harvey M. Deitel
Deitel & Associates, Inc.

Paul J. Deitel
Deitel & Associates, Inc.

TRADUCCIÓN

Alfonso Vidal Romero Elizondo
Ingeniero en Electrónica y Comunicación
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Monterrey

REVISIÓN TÉCNICA

César Coutiño Gómez
Director del Departamento de Computación
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Ciudad de México

José Martín Molina Espinosa
Profesor investigador
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Ciudad de México

Rafael Lozano Espinosa
Profesor del Departamento de Computación
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Ciudad de México

Sergio Fuenlabrada Velázquez
Profesor investigador
Unidad Profesional Interdisciplinaria de Ingeniería, Ciencias Sociales y Administrativas
Instituto Politécnico Nacional

Mario Alberto Sesma Martínez
Profesor investigador
Unidad Profesional Interdisciplinaria de Ingeniería, Ciencias Sociales y Administrativas
Instituto Politécnico Nacional

Edna Martha Miranda Chávez
Profesora investigadora
Unidad Profesional Interdisciplinaria de Ingeniería, Ciencias Sociales y Administrativas
Instituto Politécnico Nacional

PEARSON Educación

México • Argentina • Brasil • Colombia • Costa Rica • Chile • Ecuador
España • Guatemala • Panamá • Perú • Puerto Rico • Uruguay • Venezuela

SEGUNDA EDICIÓN

CÓMO PROGRAMAR EN C#

Harvey M. Deitel
Deitel & Associates, Inc.

Paul J. Deitel
Deitel & Associates, Inc.

TRADUCCIÓN

Alfonso Vidal Romero Elizondo
Ingeniero en Electrónica y Comunicación
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Monterrey

REVISIÓN TÉCNICA

César Coutiño Gómez
Director del Departamento de Computación
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Ciudad de México

José Martín Molina Espinosa
Profesor investigador
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Ciudad de México

Rafael Lozano Espinosa
Profesor del Departamento de Computación
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Ciudad de México

Sergio Fuenlabrada Velázquez
Profesor investigador
Unidad Profesional Interdisciplinaria de Ingeniería, Ciencias Sociales y Administrativas
Instituto Politécnico Nacional

Mario Alberto Sesma Martínez
Profesor investigador
Unidad Profesional Interdisciplinaria de Ingeniería, Ciencias Sociales y Administrativas
Instituto Politécnico Nacional

Edna Martha Miranda Chávez
Profesora investigadora
Unidad Profesional Interdisciplinaria de Ingeniería, Ciencias Sociales y Administrativas
Instituto Politécnico Nacional

PEARSON Educación

México • Argentina • Brasil • Colombia • Costa Rica • Chile • Ecuador
España • Guatemala • Panamá • Perú • Puerto Rico • Uruguay • Venezuela

Datos de catalogación bibliográfica

Deitel, Harvey M. y Paul J. Deitel

Cómo programar en C#. Segunda edición

PEARSON EDUCACIÓN, México 2007
ISBN: 978-970-26-1056-4
Área: Computación

Formato: 20 × 25.5 cm Páginas: 1080

Edición en español

Editor: Luis Miguel Cruz Castillo
e-mail: luis.cruz@pearsoned.com
Editor de desarrollo: Bernardino Gutiérrez Hernández
Supervisor de producción: Enrique Trejo Hernández

SEGUNDA EDICIÓN, 2007

Cámara Nacional de la Industria Editorial Mexicana. Reg. Núm. 1031.

ISBN 10: 970-26-1056-7
ISBN 13: 978-970-26-1056-4

Impreso en México. *Printed in Mexico.*
1 2 3 4 5 6 7 8 9 0 - 10 09 08 07

Contenido

si una condición es verdadera, o ignora la acción si la condición es falsa. La instrucción ***if...else*** realiza una acción si una condición es verdadera o realiza una acción distinta si la condición es falsa. La instrucción `switch` (capítulo 6) realiza una de varias acciones distintas, dependiendo del valor de una expresión.

A la instrucción `if` se le llama ***instrucción de selección simple*** debido a que selecciona o ignora una acción individual (o, como pronto veremos, un grupo individual de acciones). A la instrucción `if...else` se le llama ***instrucción de selección doble*** debido a que selecciona una de dos acciones distintas (o grupos de acciones). A la instrucción `switch` se le llama ***instrucción de selección múltiple*** debido a que selecciona una de varias acciones distintas (o grupos de acciones).

Estructuras de repetición en C#

C# cuenta con cuatro estructuras de repetición, que de aquí en adelante denominaremos ***instrucciones de repetición*** (también se les conoce como ***instrucciones de iteración*** o ***ciclos***). Las instrucciones de repetición permiten a las aplicaciones ejecutar instrucciones repetidas veces, dependiendo del valor de una ***condición de continuación de ciclo***. Las instrucciones de repetición son: `while`, `do...while`, `for` y `foreach` (en el capítulo 6 presentamos las instrucciones `do...while` y `for`; y en el 8 hablaremos sobre la instrucción `foreach`). Las instrucciones `while`, `for` y `foreach` realizan la acción (o grupo de acciones) contenida en su cuerpo cero o más veces; si la condición de continuación de ciclo es falsa al principio, no se ejecutará la acción (o grupo de acciones). La instrucción `do...while` realiza la acción (o grupo de acciones) contenida en su cuerpo una o más veces.

Las palabras `if`, `else`, `switch`, `while`, `do`, `for` y `foreach` son palabras clave de C# y no pueden utilizarse como identificadores, como en los nombres de las variables. En la figura 3.2 aparece una lista completa de palabras clave de C#.

Resumen de las instrucciones de control en C#

C# sólo cuenta con tres tipos de instrucciones de control estructuradas: la instrucción de secuencia, la de selección (tres tipos) y la de repetición (cuatro tipos). Toda aplicación se forma mediante la combinación de todas las instrucciones de secuencia, de selección y de repetición que sean apropiadas para el algoritmo que implemente la aplicación. Al igual que con la instrucción de secuencia de la figura 5.1, podemos modelar cada instrucción de control como un diagrama de actividad. Cada diagrama contiene un estado inicial y uno final, los cuales representan el punto de entrada y el punto de salida de una instrucción de control, en forma respectiva. Las ***instrucciones de control de una entrada/una salida*** facilitan la creación de aplicaciones; las instrucciones de control se "conectan" una a otra mediante la conexión del punto de salida de una con el punto de entrada de la siguiente. Este procedimiento es similar a la forma en la que un niño apila bloques de construcción, por lo cual le llamamos ***apilamiento de estructuras de control***. En breve aprenderemos que sólo hay una manera alternativa en la que pueden conectarse las instrucciones de control: ***anidamiento de instrucciones de control***, en el cual una estructura de control aparece dentro de otra. Por lo tanto, los algoritmos en las aplicaciones en C# se crean a partir de sólo tres tipos de instrucciones de control estructuradas, que se combinan en sólo dos formas. Ésta es la esencia de la simpleza.

5.3 Instrucción de selección simple `if`

Las aplicaciones utilizan instrucciones de selección para elegir entre los cursos alternativos de acción. Por ejemplo, suponga que la calificación para aprobar un examen es de 60. La instrucción

```
if ( calif >= 60 )
   Console.WriteLine( "Aprobado" );
```

determina si la condición `calif >= 60` es verdadera o falsa. Si la condición es verdadera se imprime `"Aprobado"` y se ejecuta la siguiente instrucción en la secuencia. Si la condición es falsa, no se imprime ningún mensaje y se ejecuta la siguiente instrucción en la secuencia.

La figura 5.2 ilustra la instrucción `if` de selección simple. Este diagrama de actividad contiene lo que tal vez sea el símbolo más importante en un diagrama de actividad: el rombo, o ***símbolo de decisión***, el cual indica que se tomará una decisión. El flujo de trabajo continuará a lo largo de una ruta determinada por las ***condiciones de guardia*** asociadas del símbolo, que pueden ser verdaderas o falsas. Cada flecha de transición que emerge de un símbolo de decisión tiene una condición de guardia (especificada entre corchetes, enseguida de la flecha de transición). Si una condición de guardia es verdadera, el flujo de trabajo entra al estado de acción al que apunta la flecha de

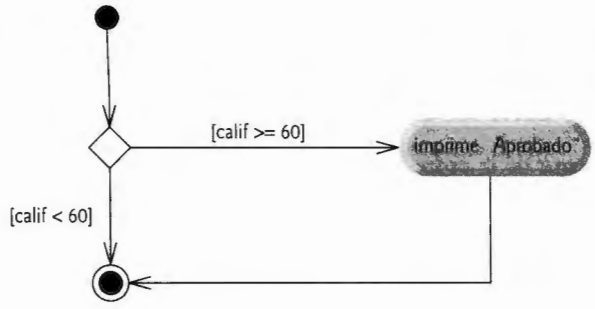


Figura 5.2 | Diagrama de actividad de UML para la instrucción de selección simple `if`.

transición. En la figura 5.2, si la calificación es mayor o igual a 60, la aplicación imprime "Aprobado" y después pasa al estado final de esta actividad. Si la calificación es menor a 60, la aplicación pasa de inmediato al estado final, sin mostrar un mensaje.

La instrucción `if` es una instrucción de control de una entrada/una salida. En breve veremos que los diagramas de actividad para el resto de las instrucciones de control también contienen estados iniciales, flechas de transición, estados de acción que indican las acciones a realizar y símbolos de decisión (con sus condiciones de guardia asociadas) que indican las decisiones a tomar, y los estados finales.

5.4 Instrucción de selección doble `if...else`

La instrucción de selección simple `if` realiza una acción indicada sólo cuando la condición es verdadera; en caso contrario se omite la acción. La instrucción de selección doble `if...else` nos permite especificar una acción a realizar cuando la condición es verdadera y una acción distinta cuando la condición es falsa. Por ejemplo, la instrucción

```
if ( calif >= 60 )
   Console.WriteLine( "Aprobado" );
else
   Console.WriteLine( "Reprobado" );
```

imprime `"Aprobado"` si la calificación es mayor o igual a 60, pero imprime `"Reprobado"` si es menor a 60. En cualquier caso, después de realizar la impresión se ejecuta la siguiente instrucción en la secuencia.

La figura 5.3 ilustra el flujo de control en la instrucción `if...else`. Una vez más, los símbolos en el diagrama de actividad de UML (además del estado inicial, las flechas de transición y el estado final) representan los estados de acción y una decisión.

Operador condicional (?:)

C# cuenta con el *operador condicional* (*?:*), que puede utilizarse en lugar de una instrucción `if...else`. Éste es el único *operador ternario* en C#; es decir, que utiliza tres operandos. En conjunto, los operandos y los símbolos `?:` forman una *expresión condicional*. El primer operando (a la izquierda del ?) es una expresión *booleana*; es decir,

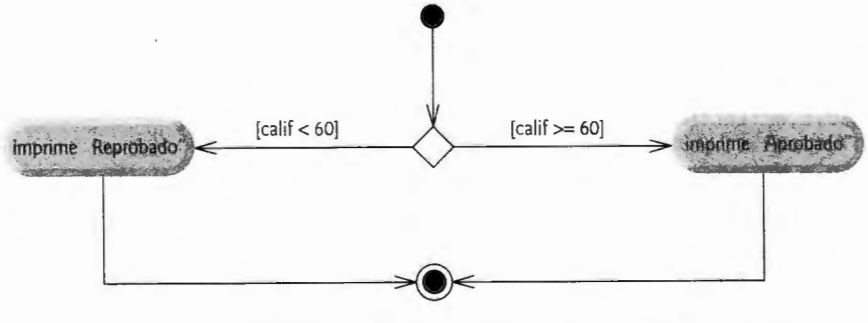


Figura 5.3 | Diagrama de actividad de UML para la instrucción de selección doble `if...else`.

una expresión que se evalúa como un valor tipo bool: ***true*** (verdadero) o ***false*** (falso). El segundo operando (entre el ? y el :) es el valor de la expresión condicional si la expresión booleana es true y el tercer operando (a la derecha del :) es el valor de la expresión condicional si la expresión booleana es false. Por ejemplo, la instrucción

```
Console.WriteLine( calif >= 60 ? "Aprobado" : "Reprobado" );
```

imprime el valor del argumento de la expresión condicional de WriteLine. La expresión condicional en esta instrucción se evalúa como la cadena "Aprobado" si la expresión booleana calif >= 60 es true (verdadera) y se evalúa como la cadena "Reprobado" si la expresión condicional es false (falsa). Por lo tanto, esta instrucción con el operador condicional realiza en esencia la misma función que la instrucción if...else que mostramos antes en esta sección. Más adelante veremos que las expresiones condicionales pueden utilizarse en algunas situaciones en las que no se pueden utilizar las instrucciones if...else.

Buena práctica de programación 5.1

Las expresiones condicionales son más difíciles de leer que las instrucciones if...else, por lo que deben utilizarse sólo para sustituir instrucciones if...else simples que elijan uno de dos valores.

Buena práctica de programación 5.2

Cuando una expresión condicional se encuentra dentro de una expresión más grande, es una buena práctica colocarla entre paréntesis para mejorar la legibilidad. Al agregar los paréntesis también se pueden evitar los problemas de precedencia de operadores que podrían provocar errores de sintaxis.

Instrucciones if...else anidadas

Una aplicación puede evaluar varios casos colocando instrucciones if...else dentro de otras instrucciones if...else, para crear las ***instrucciones if...else anidadas.*** Por ejemplo, la siguiente instrucción if...else anidada imprime A para las calificaciones de los exámenes que sean mayores o iguales a 90, B para el rango de 80 a 89, C para el rango de 70 a 79, D para el rango de 60 a 69 y F para todas las demás:

```
if ( calif >= 90 )
   Console.WriteLine( "A" );
else
   if ( calif >= 80 )
      Console.WriteLine( "B" );
   else
      if ( calif >= 70 )
         Console.WriteLine( "C" );
      else
         if ( calif >= 60 )
            Console.WriteLine( "D" );
         else
            Console.WriteLine( "F" );
```

Si calif es mayor o igual a 90, las primeras cuatro condiciones serán verdaderas pero sólo se ejecutará la instrucción en la parte if de la primera instrucción if...else. Una vez que se ejecuta esa instrucción, se omite la parte else de la instrucción if...else más "externa". La mayoría de los programadores de C# prefiere escribir la anterior instrucción if...else de la siguiente manera:

```
if ( calif >= 90 )
   Console.WriteLine( "A" );
else if ( calif >= 80 )
   Console.WriteLine( "B" );
else if ( calif >= 70 )
   Console.WriteLine( "C" );
else if ( calif >= 60 )
   Console.WriteLine( "D" );
else
   Console.WriteLine( "F" );
```

Las dos formas son idénticas, excepto por el espaciado y la sangría, que el compilador ignora. La segunda forma es más popular, ya que evita usar mucha sangría hacia la derecha del código; dicha sangría a menudo deja poco espacio en una línea de código, forzando a que las líneas se dividan y empeorando la legibilidad.

Problema del `else` *suelto*

El compilador de C# siempre asocia un `else` con el `if` que le precede inmediatamente, a menos que se le indique otra cosa mediante la colocación de llaves ({ y }). Este comportamiento puede ocasionar lo que se conoce como el *problema del* `else` *suelto*. Por ejemplo,

```
if ( x > 5 )
   if ( y > 5 )
      Console.WriteLine( "x e y son > 5" );
else
   Console.WriteLine( "x es <= 5" );
```

parece indicar que si `x` es mayor que 5, la instrucción `if` anidada determina si `y` es también mayor que 5. De ser así, se produce como resultado la cadena "`x e y son > 5`". De lo contrario, parece ser que si `x` no es mayor que 5, la instrucción `else` que es parte del `if...else` produce como resultado la cadena "`x es <= 5`".

¡Cuidado! Esta instrucción `if...else` anidada no se ejecuta como parece ser. El compilador en realidad interpreta la instrucción así:

```
if ( x > 5 )
   if ( y > 5 )
      Console.WriteLine( "x e y son > 5" );
   else
      Console.WriteLine( "x es <= 5" );
```

en donde el cuerpo del primer `if` es un `if...else` anidado. La instrucción `if` más externa evalúa si `x` es mayor que 5. De ser así, la ejecución continúa evaluando si `y` es también mayor que 5. Si la segunda condición es verdadera, se muestra la cadena apropiada ("`x e y son > 5`"). No obstante, si la segunda condición es falsa se muestra la cadena "`x es <= 5`", aun y cuando sabemos que `x` es mayor que 5.

Para forzar a que la instrucción `if...else` anidada se ejecute como se tenía pensado originalmente, debemos escribirla de la siguiente manera:

```
if ( x > 5 )
{
   if ( y > 5 )
      Console.WriteLine( "x e y son > 5" );
}
else
   Console.WriteLine( "x es <= 5" );
```

Las llaves ({}) indican al compilador que la segunda instrucción `if` se encuentra en el cuerpo del primer `if` y que el `else` está asociado con el *primer* `if`.

Bloques

Por lo general, la instrucción `if` espera sólo una instrucción en su cuerpo. Para incluir varias instrucciones en el cuerpo de un `if` (o en el cuerpo del `else` en una instrucción `if...else`), encierre las instrucciones entre llaves ({ y }). A un conjunto de instrucciones contenidas dentro de un par de llaves se le llama ***bloque***. Un bloque puede colocarse en cualquier parte de un programa en donde pueda colocarse una sola instrucción.

El siguiente ejemplo incluye un bloque en la parte `else` de una instrucción `if...else`:

```
if ( calif >= 60 )
    Console.WriteLine ( "Aprobado" );
else
{
    Console.WriteLine( "Reprobado" );
    Console.WriteLine( "Debe tomar este curso otra vez." );
}
```

En este caso, si `calif` es menor a 60, la aplicación ejecuta ambas instrucciones en el cuerpo del `else` e imprime

```
Reprobado.
Debe tomar este curso otra vez.
```

Observe las llaves que rodean a las dos instrucciones en la cláusula `else`. Estas llaves son importantes. Sin ellas, la instrucción

```
Console.WriteLine( "Debe tomar este curso otra vez." );
```

estaría fuera del cuerpo de la parte `else` de la instrucción `if...else` y se ejecutaría sin importar que la calificación fuera menor que 60.

Buena práctica de programación 5.3

Colocar siempre las llaves en una instrucción `if...else` (o cualquier estructura de control) ayuda a evitar que se omitan de manera accidental, en especial cuando se agregan posteriormente instrucciones a la parte `if` o `else`. Para evitar que esto suceda, algunos programadores prefieren escribir la llave inicial y la final de los bloques antes de escribir las instrucciones individuales dentro de ellas.

Así como un bloque puede colocarse en cualquier parte en donde pueda colocarse una sola instrucción, también es posible tener una instrucción vacía. En la sección 3.9 vimos que la instrucción vacía se representa colocando un punto y coma (;) en donde normalmente iría una instrucción.

Error común de programación 5.1

Colocar un punto y coma después de la condición en una instrucción `if` o `if...else` produce un error lógico en las instrucciones `if` de selección simple y un error de sintaxis en las instrucciones `if...else` de selección doble (cuando la parte del `if` contiene una instrucción en el cuerpo).

5.5 Instrucción de repetición `while`

Una ***instrucción de repetición*** le permite especificar que una aplicación debe repetir una acción mientras cierta condición sea verdadera. Como un ejemplo de la ***instrucción de repetición `while`*** de C#, considere un segmento de código diseñado para encontrar la primera potencia de 3 que sea mayor que 100. Suponga que la variable `int` llamada `producto` se inicializa con 3. Cuando termina de ejecutarse la siguiente instrucción `while`, `producto` contiene el resultado:

```
int producto = 3;

while ( producto <= 100 )
   producto = 3 * producto;
```

Cuando esta instrucción `while` comienza a ejecutarse, el valor de la variable `producto` es 3. Cada repetición de la instrucción `while` multiplica `producto` por 3, por lo que `producto` toma los subsiguientes valores 9, 27, 81 y 243 en forma sucesiva. Cuando la variable `producto` se convierte en 243, la condición de la instrucción `while` (`producto <= 100`) se vuelve falsa. Esto termina la repetición, por lo que el valor final de `producto` es 243. En este punto, la ejecución de la aplicación continúa con la siguiente instrucción después de la instrucción `while`.

Error común de programación 5.2

Si no se proporciona, en el cuerpo de una instrucción `while`, una acción que ocasione que en algún momento la condición del `while` se torne falsa, por lo general se producirá un error lógico conocido como **ciclo infinito**, *en el que el ciclo nunca terminará.*

El diagrama de actividad de UML de la figura 5.4 muestra el flujo de control que corresponde a la instrucción `while` anterior. Este diagrama también introduce el ***símbolo de fusión***. UML representa tanto al símbolo de fusión como al de decisión como rombos. El símbolo de fusión une dos flujos de actividad en uno solo. En este diagrama, el símbolo de fusión une las transiciones del estado inicial y del estado de acción, de manera que ambas

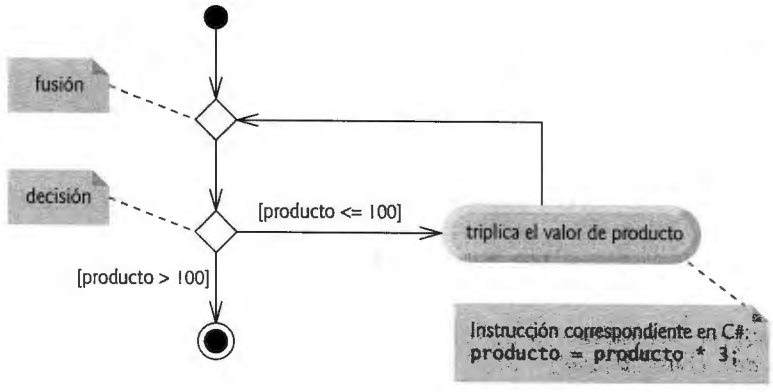


Figura 5.4 | Diagrama de actividad de UML para la instrucción de repetición `while`.

fluyan en la decisión que determina si el ciclo debe empezar a ejecutarse (o seguir ejecutándose). Los símbolos de decisión y de fusión pueden diferenciarse por el número de flechas de transición "entrantes" y "salientes". Un símbolo de decisión tiene una flecha de transición que apunta hacia el rombo y dos o más que apuntan hacia fuera del rombo, para indicar las posibles transiciones desde ese punto. Cada flecha de transición que apunta hacia fuera de un símbolo de decisión tiene una condición de guardia junto a ella. Un símbolo de fusión tiene dos o más flechas de transición que apuntan hacia el rombo y sólo una flecha de transición que apunta hacia fuera del rombo, para indicar múltiples flujos de actividad que se fusionan para continuar la actividad. Ninguna de las flechas de transición asociadas con un símbolo de fusión tiene condiciones de guardia.

La figura 5.4 muestra con claridad la repetición de la instrucción `while` que vimos antes en esta sección. La flecha de transición que emerge del estado de acción apunta de vuelta a la fusión, desde la cual el flujo del programa se dirige nuevamente hacia la decisión que se evalúa al principio de cada repetición del ciclo. El ciclo continúa ejecutándose hasta que la condición de guardia `producto > 100` se torna verdadera. Entonces, termina la instrucción `while` (llega a su estado final) y el control pasa a la siguiente instrucción en la secuencia de la aplicación.

5.6 Cómo formular algoritmos: repetición controlada por un contador

Para ilustrar cómo se desarrollan los algoritmos, modificamos la clase `LibroCalificaciones` del capítulo 4 para resolver dos variaciones de un problema que promedia las calificaciones de los estudiantes. Considere el siguiente enunciado del problema:

> *A una clase de 10 estudiantes se les aplicó un examen. Las calificaciones (enteros en el rango de 0 a 100) de este examen están disponibles para su análisis. Determine el promedio de la clase en el examen.*

El promedio de la clase es igual a la suma de las calificaciones, dividida entre el número de estudiantes. El algoritmo para resolver este problema en una computadora debe recibir como entrada cada calificación, llevar el registro del total de todas las calificaciones introducidas, realizar el cálculo del promedio e imprimir el resultado.

Emplearemos la ***repetición controlada por un contador*** para introducir las calificaciones, una a la vez. Esta técnica utiliza una variable llamada ***contador*** (o ***variable de control***) para controlar el número de veces que se ejecutará un conjunto de instrucciones. En este ejemplo, la repetición termina cuando el contador excede a 10. Esta sección presenta una versión de la clase `LibroCalificaciones` (figura 5.5) que implementa el algoritmo en un método de C#. Después se presenta una aplicación (figura 5.6) que demuestra el algoritmo en acción.

Implementación de la repetición controlada por un contador en la clase `LibroCalificaciones`

La clase `LibroCalificaciones` (figura 5.5) contiene un constructor (líneas 11-14) que asigna un valor a la variable de instancia `nombreCurso` (declarada en la línea 8) de la clase, mediante el uso de la propiedad `NombreCurso`. Las líneas 17-27 y 30-35 declaran la propiedad `NombreCurso` y el método `MostrarMensaje`, en forma respectiva.

```
1  // Fig. 6.1: ContadorWhile.cs
2  // Repetición controlada por un contador con la instrucción de repetición while.
3  using System;
4
5  public class ContadorWhile
6  {
7     public static void Main( string[] args )
8     {
9        int contador = 1; // declara e inicializa la variable de control
10
11       while ( contador <= 10 ) // condición de continuación de ciclo
12       {
13          Console.Write( "{0}  ", contador );
14          contador++; // incrementa la variable de control
15       } // fin de while
16
17       Console.WriteLine(); // imprime en pantalla una nueva línea
18    } // fin de Main
19 } // fin de la clase ContadorWhile
```

```
1  2  3  4  5  6  7  8  9  10
```

Figura 6.1 | Repetición controlada por un contador con la instrucción de repetición `while`.

Error común de programación 6.1

Como los valores de punto flotante pueden ser aproximados, el control de ciclos con variables de punto flotante puede producir valores imprecisos del contador y pruebas que terminen en forma imprecisa.

Tip de prevención de errores 6.1

Controle los ciclos de conteo con enteros.

Buena práctica de programación 6.1

Coloque líneas en blanco por encima y por debajo de las instrucciones de control de repetición y selección, y aplique sangrías a los cuerpos de las instrucciones para mejorar la legibilidad.

La aplicación en la figura 6.1 puede hacerse más concisa si se inicializa contador en 0 en la línea 9 y se incrementa `contador` en la condición del `while` con el operador de incremento prefijo, como se muestra a continuación:

```
while ( ++contador <= 10 ) // condición de continuación de ciclo
   Console.Write( "{0}  ", contador );
```

Este código ahorra una instrucción (y elimina la necesidad de tener llaves alrededor del cuerpo del ciclo), ya que la condición del `while` realiza el incremento antes de evaluar la condición (en la sección 5.10 vimos que la precedencia de ++ es mayor que la de <=). Codificar de esa manera condensada podría afectar la legibilidad del código, su depuración, modificación y mantenimiento.

Observación de ingeniería de software 6.1

"Mantener las cosas simples" es un buen consejo para la mayoría del código que usted escribirá.

6.3 Instrucción de repetición for

La sección 6.2 presentó los aspectos esenciales de la repetición controlada por un contador. La instrucción `while` puede utilizarse para implementar cualquier ciclo controlado por un contador. C# también cuenta con la

instrucción de repetición for, que especifica los elementos de la repetición controlada por un contador en una sola línea de código. La figura 6.2 reimplementa la aplicación de la figura 6.1, usando la instrucción `for`.

El método `Main` de la aplicación opera de la siguiente manera: cuando la instrucción `for` (líneas 11-12) empieza a ejecutar, la variable de control `contador` se declara e inicializa en `1` (en la sección 6.2 vimos que los primeros dos elementos de la repetición controlada por un contador son la variable de control y su valor inicial). Después, la aplicación verifica la condición de continuación de ciclo, `contador <= 10`, la cual se encuentra entre los dos signos de punto y coma requeridos. Como el valor inicial de `contador` es `1`, al principio la condición es verdadera. Por lo tanto, la instrucción del cuerpo (línea 12) muestra en pantalla el valor de la variable de control `contador`, que es `1`. Después de ejecutar el cuerpo del ciclo, la aplicación incrementa a `contador` en la expresión `contador++`, la cual aparece a la derecha del segundo signo de punto y coma. Luego, la prueba de continuación de ciclo se ejecuta de nuevo para determinar si la aplicación debe continuar con la siguiente iteración del ciclo. En este punto, el valor de la variable de control es `2`, por lo que la condición sigue siendo verdadera (el valor final no se excede), y la aplicación ejecuta la instrucción del cuerpo otra vez (es decir, la siguiente iteración del ciclo). Este proceso continúa hasta que se muestran en pantalla los números del 1 al 10 y el valor de `contador` se vuelve `11`, con lo cual falla la prueba de continuación de ciclo y termina la repetición (después de 10 repeticiones del cuerpo del ciclo en la línea 12). Después la aplicación ejecuta la primera instrucción después del `for`; en este caso, la línea 14.

Observe que la figura 6.2 utiliza (en la línea 11) la condición de continuación de ciclo `contador <= 10`. Si especificara por error `contador < 10` como la condición, el ciclo sólo iteraría nueve veces: un error lógico común, conocido como ***error de desplazamiento por 1***.

Error común de programación 6.2

Utilizar un operador relacional incorrecto o un valor final incorrecto de un contador de ciclo en la condición de continuación de ciclo de una instrucción de repetición produce un error de desplazamiento por uno.

Buena práctica de programación 6.2

Utilizar el valor final en la condición de una instrucción `while` *o* `for` *con el operador relacional <= nos ayuda a evitar los errores de desplazamiento por uno. Para un ciclo que imprime los valores del 1 al 10, la condición de continuación de ciclo debe ser* `contador <= 10`*, en vez de* `contador < 10` *(lo cual produce un error de desplazamiento por uno) o* `contador < 11` *(que es correcto). Muchos programadores prefieren el llamado conteo con base cero, en el cual para contar 10 veces,* `contador` *se inicializaría a cero y la prueba de continuación de ciclo sería* `contador < 10`.

```
1  // Fig. 6.2: ContadorFor.cs
2  // Repetición controlada por un contador con la instrucción de repetición for.
3  using System;
4
5  public class ContadorFor
6  {
7     public static void Main( string[] args )
8     {
9        // el encabezado de la instrucción for incluye la inicialización,
10       // la condición de continuación de ciclo y el incremento
11       for ( int contador = 1; contador <= 10; contador++ )
12          Console.Write( "{0}  ", contador );
13
14       Console.WriteLine(); // imprime en pantalla una nueva línea
15    } // fin de Main
16 } // fin de la clase ContadorFor
```

```
1  2  3  4  5  6  7  8  9  10
```

Figura 6.2 | Repetición controlada por un contador, con la instrucción de repetición `for`.

La figura 6.3 analiza con más detalle la instrucción `for` de la figura 6.2. A la primera línea del `for` (incluyendo la palabra clave `for` y todo lo que está entre paréntesis después del `for`), la línea 11 en la figura 6.2, se le conoce como ***encabezado de la instrucción for***, o simplemente ***encabezado del for***. Observe que el encabezado del `for` "se encarga de todo": especifica cada uno de los elementos necesarios para la repetición controlada por un contador con una variable de control. Si hay más de una instrucción en el cuerpo del `for`, se requieren llaves para definir el cuerpo del ciclo.

El formato general de la instrucción `for` es:

```
for ( inicialización; condiciónDeContinuaciónDeCiclo; incremento )
   instrucción
```

en donde la expresión *inicialización* nombra la variable de control del ciclo y proporciona su valor inicial, la *condiciónDeContinuaciónDeCiclo* es la condición que determina si deben continuar las iteraciones y el *incremento* modifica el valor de la variable de control (ya sea un incremento o un decremento), de manera que la condición de continuación de ciclo se vuelva falsa en un momento dado. Los dos signos de punto y coma en el encabezado del `for` son requeridos. Observe que no incluimos un punto y coma después de *instrucción*, ya que se asume que éste se incluye de antemano en la noción de una *instrucción*.

Error común de programación 6.3

Utilizar comas en vez de los dos signos de punto y coma requeridos en el encabezado de una instrucción `for` es un error de sintaxis.

En la mayoría de los casos, la instrucción `for` puede representarse con una instrucción `while` equivalente, de la siguiente manera:

```
inicialización;
while ( condiciónDeContinuaciónDeCiclo )
{
   instrucción
   incremento;
}
```

En la sección 6.7 veremos un caso para el cual no se puede representar una instrucción `for` con una instrucción `while` equivalente.

Por lo general, las instrucciones `for` se utilizan para la repetición controlada por un contador y las instrucciones `while` se utilizan para la repetición controlada por un centinela. No obstante, `while` y `for` pueden utilizarse para cualquiera de los dos tipos de repetición.

Si la expresión de *inicialización* en el encabezado del `for` declara la variable de control (es decir, si el tipo de la variable de control se especifica antes del nombre de la variable, como en la figura 6.2), la variable de control puede utilizarse sólo en esa instrucción `for`; no existirá fuera de esta instrucción. Este uso restringido del nombre de la

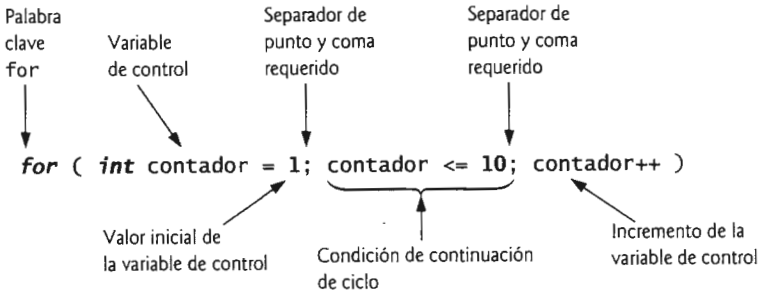


Figura 6.3 | Componentes del encabezado de la instrucción `for`.

variable de control se conoce como el ***alcance*** de la variable; y define en dónde puede utilizarse en una aplicación. Por ejemplo, una variable local sólo puede utilizarse en el método que declara a esa variable, y sólo a partir del punto de declaración, hasta el final del método. En el capítulo 7, Métodos: un análisis más detallado analizaremos el concepto de alcance.

Error común de programación 6.4

Si la variable de control de una instrucción `for` *se declara en la sección de inicialización del encabezado del* `for`*, y se utiliza fuera del cuerpo del* `for` *se produce un error de compilación.*

Las tres expresiones en un encabezado `for` son opcionales. Si se omite la *condiciónDeContinuaciónDeCiclo*, C# asume que esta condición siempre será verdadera, con lo cual se crea un ciclo infinito. No puede omitir la expresión de *inicialización* si la aplicación inicializa la variable de control antes del ciclo; en este caso, el alcance de la variable no se limitará al ciclo. Puede omitir la expresión de *incremento* si la aplicación calcula el incremento mediante instrucciones dentro del cuerpo del ciclo, o si no se necesita un incremento. La expresión de incremento en un `for` actúa como si fuera una instrucción independiente al final del cuerpo del `for`. Por lo tanto, las expresiones

```
contador = contador + 1
contador += 1
++contador
contador++
```

son expresiones de incremento equivalentes en una instrucción `for`. Muchos programadores prefieren `contador++`, ya que es concisa y además un ciclo `for` evalúa su expresión de incremento después de la ejecución de su cuerpo; por lo cual, la forma de incremento postfijo parece más natural. En este caso, la variable que se incrementa no aparece en una expresión más grande, por lo que los operadores de incremento prefijo y postfijo tienen el mismo efecto.

Tip de rendimiento 6.1

Hay una ligera ventaja de rendimiento al utilizar el operador de incremento prefijo, pero si elije el operador de incremento postfijo debido a que parece ser más natural (como en el encabezado de un `for`*), los compiladores con optimización generarán código MSIL que utilice la forma más eficiente de todas maneras.*

Buena práctica de programación 6.3

En muchos casos, los operadores de incremento prefijo y postfijo se utilizan para sumar 1 a una variable en una instrucción por sí sola. En estos casos el efecto es idéntico, sólo que el operador de incremento prefijo tiene una ligera ventaja de rendimiento. Debido a que el compilador, por lo general, optimiza el código que usted escribe para ayudarlo a obtener el mejor rendimiento, puede usar cualquiera de los dos operadores (prefijo o postfijo) con el que se sienta más cómodo en estas situaciones.

Error común de programación 6.5

Colocar un punto y coma inmediatamente a la derecha del paréntesis derecho del encabezado de un `for` *convierte el cuerpo de ese* `for` *en una instrucción vacía. Por lo general esto es un error lógico.*

Tip de prevención de errores 6.2

Los ciclos infinitos ocurren cuando la condición de continuación de ciclo en una instrucción de repetición nunca se vuelve falsa (false)*. Para evitar esta situación en un ciclo controlado por un contador, debe asegurarse que la variable de control se incremente (o decremente) durante cada iteración del ciclo. En un ciclo controlado por un centinela, asegúrese que el valor centinela se introduzca en algún momento dado.*

Las porciones correspondientes a la inicialización, la condición de continuación de ciclo y el incremento de una instrucción `for` pueden contener expresiones aritméticas. Por ejemplo, suponga que `x = 2` y `y = 10`; si `x` y `y` y no se modifican en el cuerpo del ciclo, entonces la instrucción

```
for ( int j = x; j <= 4 * x * y; j += y / x )
```

es equivalente a la instrucción

```
for ( int j = 2; j <= 80; j += 5 )
```

El incremento de una instrucción `for` también puede ser negativo, en cuyo caso sería un decremento y el ciclo contaría en orden descendente.

Si al principio la condición de continuación de ciclo es `false`, la aplicación no ejecutará el cuerpo de la instrucción `for`, sino que la ejecución continuará con la instrucción después del `for`.

Con frecuencia, las aplicaciones muestran en pantalla el valor de la variable de control o lo utilizan en cálculos dentro del cuerpo del ciclo, pero este uso no es obligatorio. Por lo general, la variable de control se utiliza para controlar la repetición sin que se le mencione dentro del cuerpo del `for`.

Tip de prevención de errores 6.3

Aunque el valor de la variable de control puede cambiarse en el cuerpo de un ciclo `for`, evite hacerlo ya que esta práctica puede llevarlo a cometer errores sutiles.

El diagrama de actividad de UML de la instrucción `for` es similar al de la instrucción `while` (figura 5.4). La figura 6.4 muestra el diagrama de actividad de la instrucción `for` de la figura 6.2. El diagrama hace evidente que la inicialización ocurre sólo una vez antes de evaluar la condición de continuación de ciclo por primera vez, y que el incremento ocurre cada vez que se realiza una iteración y después de ejecutarse la instrucción del cuerpo.

6.4 Ejemplos acerca del uso de la instrucción for

Los siguientes ejemplos muestran técnicas para variar la variable de control en una instrucción `for`. En cada caso escribimos el encabezado apropiado para el `for`. Observe el cambio en el operador relacional para los ciclos en los que se decrementa la variable de control.

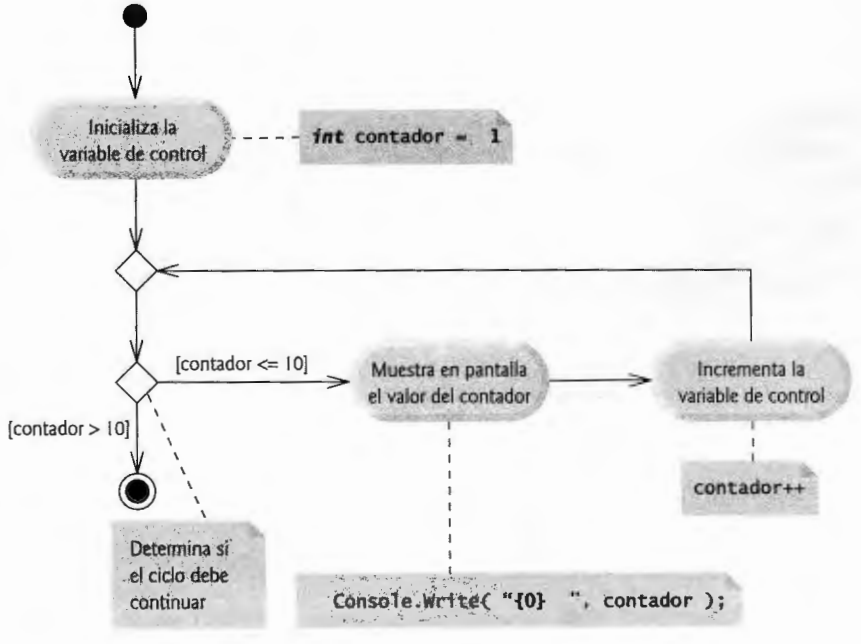


Figura 6.4 | Diagrama de actividad de UML para la instrucción `for` de la figura 6.2.

a) Modificar la variable de control de 1 a 100, en incrementos de 1.

```
for ( int i = 1;  i <= 100;  i++ )
```

b) Modificar la variable de control de 100 a 1, en decrementos de 1.

```
for ( int i = 100;  i >= 1;  i-- )
```

c) Modificar la variable de control de 7 a 77, en incrementos de 7.

```
for ( int i = 7;  i <= 77;  i += 7 )
```

d) Modificar la variable de control de 20 a 2, en decrementos de 2.

```
for ( int i = 20;  i >= 2;  i -= 2 )
```

e) Modificar la variable de control sobre la siguiente secuencia de valores: 2, 5, 8, 11, 14, 17, 20.

```
for ( int i = 2;  i <= 20;  i += 3 )
```

f) Modificar la variable de control sobre la siguiente secuencia de valores: 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.

```
for ( int i = 99;  i >= 0;  i -= 11 )
```

Error común de programación 6.6

Si no se utiliza el operador relacional apropiado en la condición de continuación de ciclo de un ciclo que cuente en orden descendente (es decir, que utilice `i <= 1` en vez de `i >= 1` en un ciclo que cuente hasta 1 en orden descendente), se produce un error lógico.

Aplicación: suma de los enteros pares del 2 al 20

Ahora consideraremos dos aplicaciones de ejemplo que demuestran los usos simples del `for`. La aplicación en la figura 6.5 utiliza una instrucción `for` para sumar los enteros pares del 2 al 20 y almacena el resultado en una variable `int` llamada `total`.

```
1  // Fig. 6.5: Suma.cs
2  // Suma de enteros con la instrucción for.
3  using System;
4
5  public class Suma
6  {
7     public static void Main( string[] args )
8     {
9        int total = 0; // inicializa el total
10
11       // suma los enteros pares del 2 al 20
12       for ( int numero = 2; numero <= 20; numero += 2 )
13          total += numero;
14
15       Console.WriteLine( "La suma es {0}", total ); // muestra los resultados
16    } // fin de Main
17 } // fin de la clase Suma
```

```
La suma es 110
```

Figura 6.5 | Suma de enteros con la instrucción `for`.

Las expresiones de *inicialización* e *incremento* pueden ser listas separadas por comas de expresiones que nos permitan utilizar varias expresiones de inicialización, o varias expresiones de incremento. Por ejemplo, el cuerpo de la instrucción `for` en las líneas 12-13 de la figura 6.5 podría mezclarse con la porción del incremento del encabezado `for` mediante el uso de una coma, como se muestra a continuación:

```
for ( int numero = 2; numero <= 20; total += numero, numero += 2 )
   ; // instrucción vacía
```

Buena práctica de programación 6.4

Limite el tamaño de los encabezados de las instrucciones de control a una sola línea, si es posible.

Buena práctica de programación 6.5

Sólo coloque expresiones que involucren a las variables de control en las secciones de inicialización e incremento de una instrucción `for`. La manipulación de otras variables debe aparecer ya sea antes del ciclo (si se ejecutan sólo una vez, como las instrucciones de inicialización) o en el cuerpo del ciclo (si se ejecutan una vez por cada iteración del ciclo, como las instrucciones de incremento o decremento).

Aplicación: cálculos del interés compuesto

La siguiente aplicación utiliza la instrucción `for` para calcular el interés compuesto. Considere el siguiente problema:

> *Una persona invierte $1,000 en una cuenta de ahorros que genera un 5% de interés anual compuesto. Suponiendo que todo el interés se deja para depósito, calcule e imprima el monto de dinero en la cuenta al final de cada año, durante 10 años. Utilice la siguiente fórmula para determinar los montos:*
>
> $a = p(1 + r)^n$
>
> *en donde*
>
> p es el monto original invertido (es decir, el capital)
> r es la tasa anual de interés (por ejemplo, utilice 0.05 para el 5%)
> n es el número de años
> a es el monto depositado al final del n-ésimo año.

Este problema implica el uso de un ciclo que realice el cálculo indicado para cada uno de los 10 años que permanecerá el dinero depositado. La solución es la aplicación que se muestra en la figura 6.6. Las líneas 9-11 en el método `Main` declaran las variables `decimal` llamadas `monto` y `capital`, y la variable `double` llamada `tasa`. Las líneas 10-11 también inicializan `capital` con `1000` (es decir, $1000.00) y `tasa` con `0.05`. C# asigna a las constantes de números reales como `0.05` el tipo `double`. De manera similar, C# asigna a las constantes de números enteros como `7` y `1000` el tipo `int`. Cuando `capital` se inicializa con `1000`, el valor `1000` de tipo `int` se promueve al tipo `decimal` de manera implícita; no se requiere una conversión.

La línea 14 imprime en pantalla los encabezados para las dos columnas de resultado de la aplicación. La primera columna muestra el año y la segunda columna muestra el monto depositado al final de cada año. Observe que utilizamos el elemento de formato `{1, 20}` para mostrar en pantalla el objeto `string` `"Monto depositado"`. El entero `20` después de la coma indica que el valor a imprimir debe mostrarse con una ***anchura de campo*** de 20; esto es, `WriteLine` debe mostrar el valor con al menos 20 posiciones de caracteres. Si el valor a imprimir tiene una anchura menor a 20 posiciones de caracteres (en este ejemplo son 17 caracteres), el valor se ***justifica a la derecha*** en el campo de manera predeterminada (en este caso, se colocan tres espacios en blanco antes del valor). Si el valor `año` a imprimir tuviera una anchura mayor a cuatro posiciones de caracteres, la anchura del campo se extendería a la derecha para dar cabida a todo el valor; esto desplazaría al campo `monto` a la derecha, con lo que se desacomodarían las columnas ordenadas de nuestros resultados tabulares. Para indicar que el resultado debe ***justificarse a la izquierda***, sólo hay que usar una anchura de campo negativa.

La instrucción `for` (líneas 17-25) ejecuta su cuerpo 10 veces, con lo cual la variable de control `anio` varía de `1` a `10`, en incrementos de `1`. Este ciclo termina cuando la variable de control `anio` se vuelve `11` (observe que `anio` representa a la n en el enunciado del problema).

```
1  // Fig. 6.6: Interes.cs
2  // Cálculos del interés compuesto con for.
3  using System;
4
5  public class Interes
6  {
7     public static void Main( string[] args )
8     {
9        decimal monto; // monto depositado al final de cada año
10       decimal capital = 1000; // monto inicial antes de los intereses
11       double tasa = 0.05; // tasa de interés
12
13       // muestra en pantalla los encabezados
14       Console.WriteLine( "{0}{1,20}", "Año", "Monto depositado" );
15
16       // calcula el monto depositado para cada uno de los 10 años
17       for ( int anio = 1; anio <= 10; anio++ )
18       {
19          // calcula el nuevo monto para el año especificado
20          monto = capital *
21             ( ( decimal ) Math.Pow( 1.0 + tasa, anio ) );
22
23          // muestra el año y el monto
24          Console.WriteLine( "{0,4}{1,20:C}", anio, monto );
25       } // fin de for
26    } // fin de Main
27 } // fin de la clase Interes
```

```
Año   Monto depositado
  1          $1,050.00
  2          $1,102.50
  3          $1,157.63
  4          $1,215.51
  5          $1,276.28
  6          $1,340.10
  7          $1,407.10
  8          $1,477.46
  9          $1,551.33
 10          $1,628.89
```

Figura 6.6 | Cálculos del interés compuesto con `for`.

Las clases proporcionan métodos que realizan tareas comunes sobre los objetos. De hecho, la mayoría de los métodos a llamar debe pertenecer a un objeto específico. Por ejemplo, en la figura 4.2, llamamos al método `MostrarMensaje` del objeto `miLibroCalificaciones` para imprimir en pantalla un saludo. Muchas clases también cuentan con métodos que realizan tareas comunes y no necesitan pertenecer a un objeto para llamarlos. Dichos métodos se denominan ***métodos* `static`**. Por ejemplo, C# no incluye un operador de exponenciación, por lo que los diseñadores de la clase `Math` definieron el método `static` llamado `Pow` para elevar un valor a una potencia. Para llamar a un método `static` debe especificar el nombre de la clase, seguido del operador punto (`.`) y el nombre del método, como en

NombreClase.nombreMétodo (*argumentos*)

Observe que los métodos `Write` y `WriteLine` de `Console` son métodos `static`. En el capítulo 7 aprenderá a implementar métodos `static` en sus propias clases.

Utilizamos el método `static` `Pow` de la clase `Math` para realizar el cálculo del interés compuesto en la figura 6.6. `Math.Pow(x, y)` calcula el valor de x elevado a la y-ésima potencia. El método recibe dos argumentos `double`

y devuelve un valor `double`. Las líneas 20-21 realizan el cálculo $a = p(1 + r)^n$, en donde a es el `monto`, p es el `capital`, r es la `tasa` y n es el `anio`. Observe que en este cálculo necesitamos multiplicar un valor `double` (`principal`) por un valor `double` (el valor de retorno de `Math.Pow`). C# no convierte implícitamente un tipo `double` en un tipo `decimal`, o viceversa, debido a la posible pérdida de información en cualquiera de las conversiones, por lo que la línea 21 contiene un operador de conversión (`decimal`) que convierte explícitamente el valor de retorno `double` de `Math.Pow` en un `decimal`.

Después de cada cálculo, la línea 24 imprime en pantalla el año y el monto depositado al final de ese año. El año se imprime en una anchura de campo de tres caracteres (según lo especificado por `{0, 4}`). El monto se imprime como un valor de moneda con el elemento de formato `{1, 20:C}`. El número `20` en el elemento de formato indica que el valor debe imprimirse justificado a la derecha, con una anchura de campo de 20 caracteres. El especificador de formato `C` especifica que el número deberá tener el formato de moneda.

Observe que declaramos las variables `monto` y `capital` de tipo `decimal` en vez de `double`. Recuerde que en la sección 4.10 introdujimos el tipo `decimal` para los cálculos monetarios. También utilizamos `decimal` en la figura 6.6 para este propósito. Tal vez tenga curiosidad acerca de por qué hacemos esto. Estamos tratando con partes fraccionales de dólares y, por ende, necesitamos un tipo que permita puntos decimales en sus valores. Por desgracia, los números de punto flotante de tipo `double` (o `float`) pueden provocar problemas en los cálculos monetarios. Dos montos, en dólares, tipo `double` almacenados en la máquina podrían ser 14.234 (que por lo general se redondea a 14.23 para fines de mostrarlo en pantalla) y 18.673 (que por lo general se redondea a 18.67 para fines de mostrarlo en pantalla). Al sumar estos montos, producen una suma interna de 32.907, que por lo general se redondea a 32.91 para fines de mostrarlo en pantalla. Por lo tanto, sus resultados podrían aparecer como

```
  14.23
+ 18.67
-------
  32.91
```

pero una persona que sume los números individuales, como se muestran, esperaría que la suma fuera de 32.90. ¡Ya ha sido advertido!

Buena práctica de programación 6.6

No utilice variables de tipo `double` (o `float`) para realizar cálculos monetarios precisos; use mejor el tipo `decimal`. La imprecisión de los números de punto flotante puede provocar errores que resulten en valores monetarios incorrectos.

Observe que el cuerpo de la instrucción `for` contiene el cálculo `1.0 + tasa`, el cual aparece como argumento para el método `Math.Pow`. De hecho, este cálculo produce el mismo resultado cada vez que se realiza una iteración en el ciclo, por lo que repetir el cálculo en todas las iteraciones del ciclo es un desperdicio.

Tip de rendimiento 6.2

*En los ciclos, evite cálculos para los cuales el resultado nunca cambie; dichos cálculos por lo general deben colocarse antes del ciclo. [*Nota: *los compiladores con optimización comúnmente colocan dichos cálculos fuera de los ciclos en el código compilado.]*

6.5 Instrucción de repetición do...while

La ***instrucción de repetición do...while*** es similar a la instrucción `while`. En el `while`, la aplicación evalúa la condición de continuación de ciclo al principio, antes de ejecutar el cuerpo del ciclo. Si la condición es falsa, el cuerpo nunca se ejecuta. La instrucción `do...while` evalúa la condición de continuación de ciclo *después* de ejecutar el cuerpo; por lo tanto, éste siempre se ejecuta cuando menos una vez. Cuando la instrucción `do...while` termina, la ejecución continúa con la siguiente instrucción en secuencia. La figura 6.7 utiliza un `do...while` (líneas 11-15) para imprimir en pantalla los números del 1 al 10.

La línea 9 declara e inicializa la variable de control `contador`. Al momento de entrar a la instrucción `do...while`, la línea 13 imprime el valor de `contador` y la 14 incrementa esta variable. Después la aplicación evalúa la condición de continuación de ciclo en la parte inferior del ciclo (línea 15). Si la condición es verdadera, el ciclo

continúa a partir de la primera instrucción del cuerpo en el do...while (línea 13). Si la condición es falsa el ciclo termina, y la aplicación continúa con la siguiente instrucción después del ciclo.

La figura 6.8 contiene el diagrama de actividad de UML para la instrucción do...while. Este diagrama hace ver que la condición de continuación de ciclo no se evalúa sino hasta después de que el ciclo ejecuta su estado de acción, por lo menos una vez. Compare este diagrama de actividad con el de la instrucción while (figura 5.4). No es necesario utilizar llaves en la instrucción de repetición do...while si sólo hay una instrucción en el cuerpo. No obstante, la mayoría de los programadores incluye las llaves para evitar confusión entre las instrucciones while y do...while. Por ejemplo,

```
while ( condición )
```

es por lo general la primera línea de una instrucción while. Una instrucción do...while sin llaves alrededor de un cuerpo con una sola instrucción aparece así:

```
do
   instrucción
while ( condición );
```

lo cual puede ser confuso. El lector podría malinterpretar la última línea [while(*condición*);] como una instrucción while que contiene una instrucción vacía (el punto y coma por sí solo). Por ende, una instrucción do...while con una sola instrucción en su cuerpo se escribe generalmente así:

```
do
{
   instrucción
} while ( condición );
```

Tip de prevención de errores 6.4

Siempre incluya las llaves en una instrucción do...while, *aun y cuando no sean necesarias. Esto ayuda a eliminar la ambigüedad entre las instrucciones* while *y* do...while *que contienen sólo una instrucción.*

```
1  // Fig. 6.7: PruebaDoWhile.cs
2  // La instrucción de repetición do...while.
3  using System;
4
5  public class PruebaDoWhile
6  {
7     public static void Main( string[] args )
8     {
9        int contador = 1; // inicializa contador
10
11       do
12       {
13          Console.Write( "{0}  ", contador );
14          contador++;
15       } while ( contador <= 10 ); // fin de do...while
16
17       Console.WriteLine(); // imprime en pantalla una nueva línea
18    } // fin de Main
19 } // fin de la clase PruebaDoWhile
```

```
1  2  3  4  5  6  7  8  9  10
```

Figura 6.7 | La instrucción de repetición do...while.

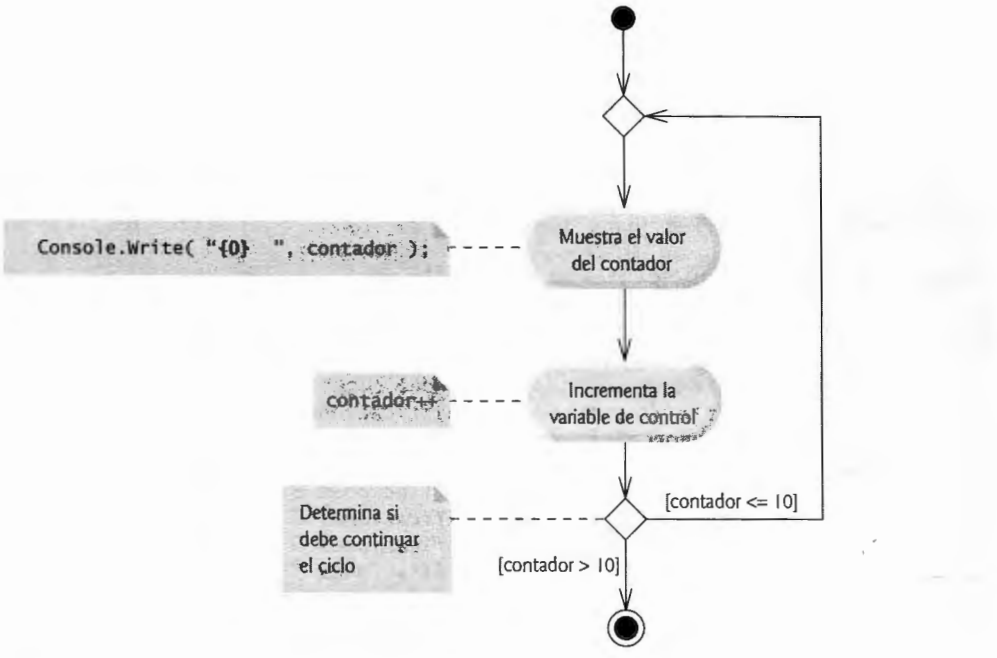


Figura 6.8 | Diagrama de actividad de UML de la instrucción de repetición do...while.

6.6 Instrucción de selección múltiple switch

En el capítulo 5 vimos la instrucción de selección simple `if` y la instrucción de selección doble `if...else`. C# cuenta con la instrucción ***de selección múltiple*** **`switch`** para realizar distintas acciones, con base en los posibles valores de una expresión. Cada acción se asocia con el valor de una ***expresión constante integral*** o con una ***expresión constante de cadena*** que la variable o expresión en la que se basa el `switch` puede asumir. Una expresión constante integral es cualquier expresión que implica constantes tipo carácter y enteras, que se evalúa como un valor entero (es decir, los valores de tipo `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong` y `char`). Una expresión constante de cadena es cualquier expresión compuesta por literales de cadena, que siempre produce el mismo objeto `string`.

Clase `LibroCalificaciones` con la instrucción `switch` para contar las calificaciones A, B, C, D y F

La figura 6.9 contiene una versión mejorada de la clase `LibroCalificaciones` que presentamos en el capítulo 4, y desarrollamos un poco más en el capítulo 5. La versión de la clase que presentamos ahora no sólo calcula el promedio de un conjunto de calificaciones numéricas introducidas por el usuario, sino que utiliza una instrucción `switch` para determinar si cada calificación es el equivalente de A, B, C, D o F, y para incrementar el contador de la calificación apropiada. La clase también imprime en pantalla un resumen del número de estudiantes que recibieron cada calificación. La figura 6.10 muestra la entrada y la salida de ejemplo de la aplicación `PruebaLibroCalificaciones`, que utiliza la clase `LibroCalificaciones` para procesar un conjunto de calificaciones.

Al igual que las versiones anteriores de la clase, `LibroCalificaciones` (figura 6.9) declara la variable de instancia `nombreCurso` (línea 7), la propiedad `NombreCurso` (líneas 24-34) para acceder a `nombreCurso` y el método `MostrarMensaje` (líneas 37-42) para mostrar un mensaje de bienvenida al usuario. La clase también contiene un constructor (líneas 18-21) que inicializa el nombre del curso.

La clase `LibroCalificaciones` también declara las variables de instancia `total` (línea 8) y `contadorCalif` (línea 9), que llevan el registro de la suma de las calificaciones introducidas por el usuario y el número de calificaciones introducidas, respectivamente. Las líneas 10-14 declaran las variables contador para cada categoría de calificaciones. La clase `LibroCalificaciones` mantiene a `total`, `contadorCalif` y a los cinco contadores

de las letras de calificación como variables de instancia, de manera que estas variables puedan utilizarse o modificarse en cualquiera de los métodos de la clase. Observe que el constructor de la clase (líneas 18-21) establece sólo el nombre del curso; las siete variables de instancia restantes son de tipo `int` y se inicializan con 0, de manera predeterminada.

```
1  // Fig. 6.9: LibroCalificaciones.cs
2  // Clase LibroCalificaciones que utiliza la instrucción switch para contar las
      calificaciones A, B, C, D y F.
3  using System;
4
5  public class LibroCalificaciones
6  {
7     private string nombreCurso; // nombre del curso que representa este LibroCalificaciones
8     private int total; // suma de las calificaciones
9     private int contadorCalif; // número de calificaciones introducidas
10    private int contA; // cuenta de calificaciones A
11    private int contB; // cuenta de calificaciones B
12    private int contC; // cuenta de calificaciones C
13    private int contD; // cuenta de calificaciones D
14    private int contF; // cuenta de calificaciones F
15
16    // el constructor inicializa nombreCurso;
17    // las variables de instancia int se inicializan en 0 de manera predeterminada
18    public LibroCalificaciones( string nombre )
19    {
20       NombreCurso = nombre; // inicializa nombreCurso
21    } // fin del constructor
22
23    // propiedad que obtiene (get) y establecer (set) el nombre del curso
24    public string NombreCurso
25    {
26       get
27       {
28          return nombreCurso;
29       } // fin de get
30       set
31       {
32          nombreCurso = value;
33       } // fin de set
34    } // fin de la propiedad NombreCurso
35
36    // muestra un mensaje de bienvenida al usuario de LibroCalificaciones
37    public void MostrarMensaje()
38    {
39       // NombreCurso obtiene el nombre del curso
40       Console.WriteLine( "Bienvenido al libro de calificaciones para\n{0}!\n",
41          NombreCurso );
42    } // fin del método MostrarMensaje
43
44    // recibe como entrada un número arbitrario de calificaciones del usuario
45    public void IntroducirCalif()
46    {
47       int calificacion; // calificación introducida por el usuario
48       string entrada; // texto introducido por el usuario
49
```

Figura 6.9 | La clase `LibroCalificaciones`, que utiliza una instrucción `switch` para contar las calificaciones A, B, C, D y F. (Parte 1 de 3).

```
50          Console.WriteLine( "{0}\n{1}",
51             "Escriba las calificaciones enteras en el rango de 0 a 100.",
52             "Escriba <Ctrl> z y oprima Intro para terminar la captura:" );
53
54          entrada = Console.ReadLine(); // lee la entrada del usuario
55
56          // itera hasta que el usuario escriba el indicador de fin de archivo (<Ctrl> z)
57          while ( entrada != null )
58          {
59             calificacion = Convert.ToInt32( entrada ); // lee la calificación del usuario
60             total += calificacion; // suma calificacion a total
61             contadorCalif++; // incrementa el número de calificaciones
62
63             // llama al método para incrementar el contador apropiado
64             IncrementarContadorLetraCalif( calificacion );
65
66             entrada = Console.ReadLine(); // lee entrada del usuario
67          } // fin de while
68       } // fin del método IntroducirCalif
69
70       // suma 1 al contador apropiado para la calificación especificada
71       private void IncrementarContadorLetraCalif( int calificacion )
72       {
73          // determina cuál calificación se introdujo
74          switch ( calificacion / 10 )
75          {
76             case 9: // la calificación estaba en los 90s
77             case 10: // la calificación fue 100
78                contA++; // incrementa contA
79                break; // necesario para salir de switch
80             case 8: // la calificación estaba entre 80 y 89
81                contB++; // incrementa contB
82                break; // sale del switch
83             case 7: // la calificación estaba entre 70 y 79
84                contC++; // incrementa contC
85                break; // sale del switch
86             case 6: // la calificación estaba entre 60 y 69
87                contD++; // incrementa contD
88                break; // sale del switch
89             default: // la calificación fue menor de 60
90                contF++; // incrementa contF
91                break; // sale del switch
92          } // fin de switch
93       } // fin del método IncrementarContadorLetraCalif
94
95       // muestra un reporte con base en las calificaciones introducidas por el usuario
96       public void MostrarReporteCalif()
97       {
98          Console.WriteLine( "\nReporte de calificaciones:" );
99
100         // si el usuario introdujo cuando menos una calificación...
101         if ( contadorCalif != 0 )
102         {
103            // calcula el promedio de todas las calificaciones introducidas
104            double promedio = ( double ) total / contadorCalif;
105
106            // imprime resumen de resultados
```

Figura 6.9 | La clase `LibroCalificaciones`, que utiliza una instrucción `switch` para contar las calificaciones A, B, C, D y F. (Parte 2 de 3).

```
107          Console.WriteLine( "El total de las {0} calificaciones introducidas es {1}",
108             contadorCalif, total );
109          Console.WriteLine( "El promedio de la clase es {0:F2}", promedio );
110          Console.WriteLine( "{0}A: {1}\nB: {2}\nC: {3}\nD: {4}\nF: {5}",
111             "Número de estudiantes que recibieron cada calificación:\n",
112             contA, // muestra el número de calificaciones A
113             contB, // muestra el número de calificaciones B
114             contC, // muestra el número de calificaciones C
115             contD, // muestra el número de calificaciones D
116             contF ); // muestra el número de calificaciones F
117       } // fin de if
118       else // no se introdujeron calificaciones, por lo que se imprime el mensaje apropiado
119          Console.WriteLine( "No se introdujeron calificaciones" );
120    } // fin del método MostrarReporteCalif
121 } // fin de la clase LibroCalificaciones
```

Figura 6.9 | La clase `LibroCalificaciones`, que utiliza una instrucción `switch` para contar las calificaciones A, B, C, D y F. (Parte 3 de 3).

La clase `LibroCalificaciones` contiene tres métodos adicionales: `IntroducirCalif`, `IncrementarContadorLetraCalif` y `MostrarReporteCalif`. El método `IntroducirCalif` (líneas 45-68) lee un número arbitrario de calificaciones enteras del usuario mediante el uso de la repetición controlada por un centinela, y actualiza las variables de instancia `total` y `contadorCalif`. El método `IntroducirCalif` llama al método `IncrementarContadorLetraCalif` (líneas 71-93) para actualizar el contador de letra de calificación apropiado para cada calificación introducida. La clase `LibroCalificaciones` también contiene el método `MostrarReporteCalif` (líneas 96-120), el cual imprime en pantalla un reporte que contiene el total de todas las calificaciones introducidas, el promedio de las mismas y el número de estudiantes que recibieron cada letra de calificación. Vamos a examinar estos métodos con más detalle.

Las líneas 47-48 en el método `IntroducirCalif` declaran las variables `calificacion` y `entrada`, las cuales almacenan la entrada del usuario, primero como un objeto `string` (en la variable `entrada`) y después lo convierten en un `int` para almacenarlo en la variable `calificacion`. Las líneas 50-52 piden al usuario que introduzca calificaciones enteras y que escriba *<Ctrl> z* y después oprima *Intro* para terminar la captura. La notación *<Ctrl> z* significa que hay que oprimir al mismo tiempo la tecla *Ctrl* y la tecla *z* cuando se escribe en una ventana de Símbolo del sistema. *<Ctrl> z* es la secuencia de teclas de Windows para escribir el ***indicador de fin de archivo***. Ésta es una manera de informar a una aplicación que no hay más datos a introducir. (El indicador de fin de archivo es una combinación de pulsación de teclas independiente del sistema. En muchos sistemas diferentes de Windows, el indicador de fin de archivo se introduce escribiendo *<Ctrl> d*.) En el capítulo 18, Archivos y flujos, veremos cómo se utiliza el indicador de fin de archivo cuando una aplicación lee su entrada desde un archivo. [*Nota:* por lo general, Windows muestra los caracteres `^Z` en una ventana de Símbolo del sistema cuando se escribe el indicador de fin de archivo, como se muestra en la salida de la figura 6.10.]

La línea 54 utiliza el método `ReadLine` para obtener la primera línea que introdujo el usuario y almacenarla en la variable `entrada`. La instrucción `while` (líneas 57-67) procesa esta entrada del usuario. La condición en la línea 57 comprueba si el valor de `entrada` es una referencia `null`. El método `ReadLine` de la clase `Console` sólo devolverá `null` si el usuario escribió un indicador de fin de archivo. Mientras que no se haya escrito el indicador de fin de archivo, `entrada` no contendrá una referencia `null`, y la condición pasará.

La línea 59 convierte el objeto `cadena` que contiene `entrada` en un tipo `int`. La línea 60 suma `calificacion` a `total`. La línea 61 incrementa `contadorCalif`. El método `MostrarReporteCalif` de la clase utiliza estas variables para calcular el promedio de las calificaciones. La línea 64 llama al método `IncrementarContadorLetraCalif` de la clase (declarado en las líneas 71-93) para incrementar el contador de letra de calificación apropiado, con base en la calificación numérica introducida.

El método `IncrementarContadorLetraCalif` contiene una instrucción `switch` (líneas 74-92) que determina cuál contador se debe incrementar. En este ejemplo, suponemos que el usuario introduce una calificación válida en el rango de 0 a 100. Una calificación en el rango de 90 a 100 representa la A, de 80 a 89 la B, de 70 a 79 la C, de 60 a 69 la D y de 0 a 59 la F. La instrucción `switch` consiste en un bloque que contiene una secuencia de

etiquetas **case** y una *etiqueta* **default** opcional. Estas etiquetas se utilizan en este ejemplo para determinar cuál contador a incrementar con base en la calificación.

Cuando el flujo de control llega al `switch`, la aplicación evalúa la expresión entre paréntesis (`calificacion / 10`) que va después de la palabra clave `switch`. A esto se le conoce como ***expresión del switch***. La aplicación compara el valor de la expresión del `switch` con cada etiqueta `case`. La expresión del `switch` en la línea 74 realiza una división entera que trunca la parte fraccional del resultado. Así, cuando dividimos cualquier valor en el rango de 0 a 100 entre 10, el resultado siempre es un valor de 0 a 10. Utilizamos varios de estos valores en nuestras etiquetas `case`. Por ejemplo, si el usuario introduce el entero 85, la expresión del `switch` se evalúa como el valor `int` 8. El `switch` compara el 8 con cada `case`. Si hay una concordancia (`case 8:` en la línea 80), la aplicación ejecuta las instrucciones para ese `case`. Para el entero 8, la línea 81 incrementa a `contB`, ya que una calificación dentro del rango de 80 a 89 es B. La ***instrucción break*** (línea 82) hace que el control del programa continúe con la primera instrucción después del `switch`; en esta aplicación, llegamos al final del cuerpo del método `IncrementarContadorLetraCalif`, por lo que el control regresa a la línea 66 en el método `IntroducirCalif` (la primera línea después de la llamada a `IncrementarContadorLetraCalif`). Esta línea utiliza el método `ReadLine` para leer la siguiente línea introducida por el usuario y la asigna a la variable `entrada`. La línea 67 marca el final del cuerpo del ciclo `while` que recibe como entrada las calificaciones (líneas 57-67), por lo que el control fluye a la condición del `while` (línea 57) para determinar si el ciclo debe continuar ejecutándose, con base en el valor que se acaba de asignar a la variable `entrada`.

Las etiquetas `case` en nuestra instrucción `switch` evalúan explícitamente los valores 10, 9, 8, 7 y 6. Observe las etiquetas `case` en las líneas 76-77, que evalúan los valores 9 y 10 (las cuales representan la calificación A). Al listar las etiquetas `case` de consecutiva, sin instrucciones entre ellas, se permite a los `case` ejecutar el mismo conjunto de instrucciones; cuando la expresión del `switch` se evalúa como 9 o 10, se ejecutan las instrucciones en las líneas 78-79. La instrucción `switch` no cuenta con un mecanismo para evaluar rangos de valores, por lo que todos los valores a evaluar deben listarse en una etiqueta `case` separada. Observe que cada `case` puede tener varias instrucciones. La instrucción `switch` difiere de otras instrucciones de control en cuanto a que no requiere llaves alrededor de varias instrucciones en cada `case`.

En C, C++ y en muchos otros lenguajes de programación que utilizan la instrucción `switch`, no se requiere la instrucción `break` al final de un `case`. Sin instrucciones `break`, cada vez que ocurre una concordancia en el `switch`, se ejecutan las instrucciones para ese `case` y los `case` subsiguientes hasta encontrar una instrucción `break` o el final del `switch`. A menudo a esto se le conoce como que las etiquetas `case` "se pasarían" hacia las instrucciones en las etiquetas `case` subsiguientes. Por lo general esto produce errores lógicos cuando se olvida la instrucción `break`. Por esta razón, C# tiene una regla de "no pasar" las etiquetas `case` en un `switch`: una vez que se ejecutan las instrucciones en un `case`, es obligatorio incluir una instrucción que termine esa etiqueta `case`, como `break`, `return` o `throw` (en el capítulo 12 hablaremos sobre la instrucción `throw`).

Error común de programación 6.7

Olvidar una instrucción `break` *cuando es necesaria en una instrucción* `switch` *es un error de sintaxis.*

Si no ocurre una concordancia entre el valor de la expresión del `switch` y una etiqueta `case`, se ejecutan las instrucciones después de la etiqueta `default` (líneas 90-91). Utilizamos la etiqueta `default` en este ejemplo para procesar todos los valores de la expresión del `switch` que sean menores de 6; esto es, todas las calificaciones de reprobado. Si no ocurre una concordancia y la instrucción `switch` no contiene una etiqueta `default`, el control del programa simplemente continúa con la primera instrucción después de la instrucción `switch`.

Clase `PruebaLibroCalificaciones` *para demostrar la clase* `LibroCalificaciones`

La clase `PruebaLibroCalificaciones` (figura 6.10) crea un objeto `LibroCalificaciones` (líneas 10-11). La línea 13 invoca el método `MostrarMensaje` del objeto para imprimir en pantalla un mensaje de bienvenida para el usuario. La línea 14 invoca el método `IntroducirCalif` del objeto para leer un conjunto de calificaciones del usuario y llevar el registro de la suma de todas las calificaciones introducidas y el número de calificaciones. Recuerde que el método `IntroducirCalif` también llama al método `IncrementarContadorLetraCalif` para llevar el registro del número de estudiantes que recibieron cada letra de calificación. La línea 15 invoca el método `MostrarReporteCalif` de la clase `LibroCalificaciones`, que imprime en pantalla un reporte con base en las calificaciones introducidas. La línea 101 de la clase `LibroCalificaciones` (figura 6.9) determina si el usuario introdujo

cuando menos una calificación; esto evita la división entre cero. De ser así, la línea 104 calcula el promedio de las calificaciones. Entonces, las líneas 107-116 imprimen en pantalla el total de todas las calificaciones, el promedio de la clase y el número de estudiantes que recibieron cada letra de calificación. Si no se introdujeron calificaciones, la línea 119 imprime en pantalla un mensaje apropiado. Los resultados en la figura 6.10 muestran un reporte de calificaciones de ejemplo, con base en 9 calificaciones.

Observe que la clase `PruebaLibroCalificaciones` (figura 6.10) no llama directamente al método `IncrementarContadorLetraCalif` de `LibroCalificaciones` (líneas 71-93 de la figura 6.9). Este método lo utiliza exclusivamente el método `IntroducirCalif` de la clase `LibroCalificaciones` para actualizar el contador de la calificación de letra apropiado, a medida que el usuario introduce cada nueva calificación. El método

```
1  // Fig. 6.10: PruebaLibroCalificaciones.cs
2  // Crea el objeto LibroCalificaciones, recibe como entrada las calificaciones y muestra
      un reporte.
3
4  public class PruebaLibroCalificaciones
5  {
6     public static void Main( string[] args )
7     {
8        // crea el objeto miLibroCalificaciones tipo LibroCalificaciones y
9        // pasa el nombre del curso al constructor
10       LibroCalificaciones miLibroCalificaciones = new LibroCalificaciones(
11          "CS101 Introducción a la programación en C#" );
12
13       miLibroCalificaciones.MostrarMensaje(); // muestra mensaje de bienvenida
14       miLibroCalificaciones.IntroducirCalif(); // lee calificaciones del usuario
15       miLibroCalificaciones.MostrarReporteCalif(); // muestra un reporte basado en las
         calificaciones
16    } // fin de Main
17 } // fin de la clase PruebaLibroCalificaciones
```

```
Bienvenido al libro de calificaciones para
CS101 Introducción a la programación en C#!

Escriba las calificaciones enteras en el rango de 0 a 100.
Escriba <Ctrl> z y oprima Intro para terminar la captura:
99
92
45
100
57
63
76
14
92
^Z

Reporte de calificaciones:
El total de las 9 calificaciones introducidas es 638
El promedio de la clase es 70.89
Número de estudiantes que recibieron cada calificación:
A: 4
B: 0
C: 1
D: 1
F: 3
```

Figura 6.10 | Crear el objeto `LibroCalificaciones`, recibir como entrada las calificaciones y mostrar en pantalla el reporte de calificaciones.

`IncrementarContadorLetraCalif` existe únicamente para dar soporte a las operaciones de los demás métodos de la clase `LibroCalificaciones`, por lo que se declara como `private`. En el capítulo 4 vimos que los métodos que se declaran con el modificador de acceso `private` pueden llamarse sólo por otros métodos de la clase en la que están declarados los métodos `private`. Dichos métodos comúnmente se conocen como ***métodos utilitarios*** o ***métodos ayudantes***, debido a que sólo pueden llamarse mediante otros métodos de esa clase y se utilizan para dar soporte a la operación de esos métodos.

Diagrama de actividad de UML de la instrucción switch

La figura 6.11 muestra el diagrama de actividad de UML para la instrucción `switch` general. Cada conjunto de instrucciones después de una etiqueta `case` debe terminar su ejecución en una instrucción `break` o `return` para terminar la instrucción `switch` después de procesar el `case`. Por lo general, se utilizan instrucciones `break`. La figura 6.11 enfatiza esto al incluir instrucciones `break` en el diagrama de actividad. El diagrama hace ver que la instrucción `break` al final de un `case` ocasiona que el control salga de la instrucción `switch` de inmediato.

Observación de ingeniería de software 6.2

Es conveniente que proporcione una etiqueta `default` en las instrucciones `switch`. Los casos que no se evalúen en forma explícita en un `switch` que carezca de una etiqueta label se ignoran. Al incluir una etiqueta `default` usted puede enfocarse en la necesidad de procesar las condiciones excepcionales.

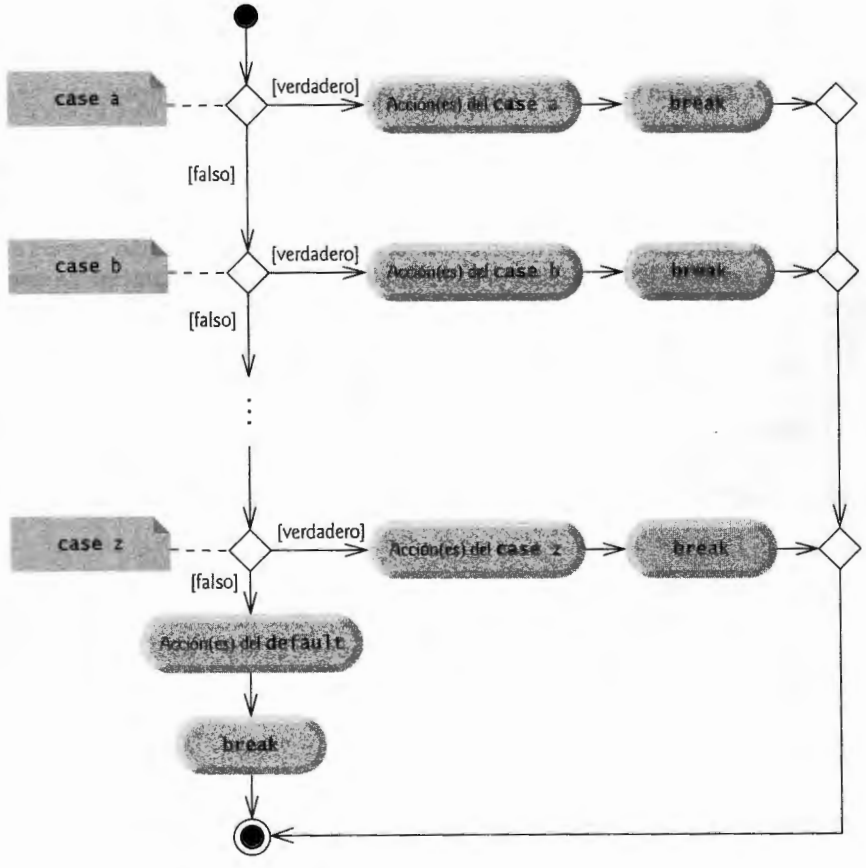


Figura 6.11 | Diagrama de actividad de UML de la instrucción `switch` de selección múltiple con instrucciones `break`.

Buena práctica de programación 6.7

Aunque cada `case` *y la etiqueta* `default` *en una instrucción* `switch` *pueden ocurrir en cualquier orden, es conveniente colocar la etiqueta* `default` *al último para mejorar la legibilidad.*

Cuando utilice la instrucción `switch`, recuerde que la expresión después de cada `case` sólo puede ser una expresión constante integral o una expresión constante de cadena; esto es, cualquier combinación de constantes que se evalúe como un valor constante de tipo integral o `string`. Una constante entera es tan solo un valor entero (por ejemplo, –7, 0 o 221). Además, puede utilizar ***constantes tipo carácter***: caracteres específicos entre comillas sencillas, como `'A'`, `'7'` o `'$'`, que representan los valores enteros de los caracteres. (En el apéndice D, Conjunto de caracteres ASCII, se muestran los valores enteros de los caracteres en el conjunto de caracteres ASCII, que es un subconjunto del conjunto de caracteres Unicote utilizado por C#.) Una constante `string` es una secuencia de caracteres entre comillas dobles, tal como `"¡Bienvenido a la programación en C#!"`.

La expresión en cada `case` también puede ser una ***constante***: una variable que contiene un valor que no cambia durante toda la aplicación. Dicha variable se declara mediante la palabra clave `const` (la cual se describe en el capítulo 7, Métodos: un análisis más detallado). Por otro lado, C# tiene una característica conocida como enumeraciones, que también presentaremos en el capítulo 7. Las constantes de enumeración también pueden utilizarse en etiquetas `case`. En el capítulo 11, Polimorfismo, interfaces y sobrecarga de operadores, presentaremos una manera más elegante de implementar la lógica del `switch`; utilizaremos una técnica llamada polimorfismo para crear aplicaciones que a menudo son más legibles, fáciles de mantener y extender que las aplicaciones que utilizan lógica de `switch`.

6.7 Instrucciones `break` y `continue`

Además de las instrucciones de selección y repetición, C# cuenta con las instrucciones `break` y `continue` para alterar el flujo de control. En la sección anterior mostramos cómo puede utilizarse la instrucción `break` para terminar la ejecución de una instrucción `switch`. En esta sección veremos cómo utilizar `break` para terminar cualquier instrucción de repetición.

Instrucción `break`

Cuando la instrucción `break` se ejecuta en una instrucción `while`, `for`, `do...while`, `switch` o `foreach`, ocasiona la salida inmediata de esa instrucción. La ejecución continúa con la primera instrucción después de la instrucción de control. Los usos comunes de la instrucción `break` son para escapar anticipadamente de una instrucción de repetición, o para omitir el resto de una instrucción `switch` (como en la figura 6.9). La figura 6.12 demuestra el uso de una instrucción `break` para salir de un ciclo `for`.

Cuando la instrucción `if` anidada en la línea 13 dentro de la instrucción `for` (líneas 11-17) determina que `conteo` es 5, se ejecuta la instrucción `break` en la línea 14. Esto termina la instrucción `for` y la aplicación continúa a la línea 19 (inmediatamente después de la instrucción `for`), que muestra un mensaje indicando el valor de la variable de control cuando terminó el ciclo. El ciclo ejecuta su cuerpo por completo sólo cuatro veces en vez de 10, debido a la instrucción `break`.

Instrucción `continue`

Cuando la ***instrucción `continue`*** se ejecuta en una instrucción `while`, `for`, `do...while` o `foreach`, omite las instrucciones restantes en el cuerpo del ciclo y continúa con la siguiente iteración del ciclo. En las instrucciones `while` y `do...while`, la aplicación evalúa la prueba de continuación de ciclo justo después de que se ejecuta la instrucción `continue`. En una instrucción `for` se ejecuta la expresión de incremento y después la aplicación evalúa la prueba de continuación de ciclo.

La figura 6.13 utiliza la instrucción `continue` en un ciclo `for` para omitir la instrucción de la línea 14 cuando la instrucción `if` anidada (línea 11) determina que el valor de `cuenta` es 5. Cuando se ejecuta la instrucción `continue`, el control del programa continúa con el incremento de la variable de control en la instrucción `for` (línea 9).

En la sección 6.3 declaramos que la instrucción `while` puede utilizarse, en la mayoría de los casos, en lugar de `for`. La única excepción ocurre cuando la expresión de incremento en el `while` va después de una instrucción `continue`. En este caso, el incremento no se ejecuta antes de que la aplicación evalúe la condición de continuación de repetición, por lo que el `while` no se ejecuta de la misma manera que el `for`.

```
1  // Fig. 6.12: PruebaBreak.cs
2  // Instrucción break para salir de una instrucción for.
3  using System;
4
5  public class PruebaBreak
6  {
7     public static void Main( string[] args )
8     {
9        int cuenta; // variable de control; también se usa después de que termina el ciclo
10
11       for ( cuenta = 1; cuenta <= 10; cuenta++ ) // itera 10 veces
12       {
13          if ( cuenta == 5 ) // si cuenta es 5,
14             break; // termina el ciclo
15
16          Console.Write( "{0} ", cuenta );
17       } // fin de for
18
19       Console.WriteLine( "\nSalió del ciclo cuando cuenta = {0}", cuenta );
20    } // fin de Main
21 } // fin de la clase PruebaBreak
```

```
1 2 3 4
Salió del ciclo cuando cuenta = 5
```

Figura 6.12 | Instrucción break para salir de una instrucción for.

Observación de ingeniería de software 6.3

Algunos programadores sienten que las instrucciones break *y* continue *violan la programación estructurada. Ya que pueden lograrse los mismos efectos con las técnicas de programación estructurada, estos programadores prefieren no utilizar instrucciones* break *o* continue.

```
1  // Fig. 6.13: PruebaContinue.cs
2  // Instrucción continue para terminar una iteración de una instrucción for.
3  using System;
4
5  public class PruebaContinue
6  {
7     public static void Main( string[] args )
8     {
9        for ( int cuenta = 1; cuenta <= 10; cuenta++ ) // itera 10 veces
10       {
11          if ( cuenta == 5 ) // si cuenta es 5,
12             continue; // omite el código restante en el ciclo
13
14          Console.Write( "{0} ", cuenta );
15       } // fin de for
16
17       Console.WriteLine( "\nSe usó continue para omitir imprimir el 5" );
18    } // fin de Main
19 } // fin de la clase PruebaContinue
```

```
1 2 3 4 6 7 8 9 10
Se usó continue para omitir imprimir el 5
```

Figura 6.13 | Instrucción continue para terminar una iteración de una instrucción for.

Observación de ingeniería de software 6.4

Existe una tensión entre lograr la ingeniería de software de calidad y lograr el software con mejor desempeño. A menudo, una de estas metas se logra a expensas de la otra. Para todas las situaciones excepto las que demanden el mayor rendimiento, aplique la siguiente regla empírica: primero, asegúrese de que su código sea simple y correcto; después hágalo rápido y pequeño, pero sólo si es necesario.

6.8 Operadores lógicos

Cada una de las instrucciones `if`, `if...else`, `while`, `do...while` y `for` requieren una condición para determinar cómo continuar con el flujo de control de una aplicación. Hasta ahora sólo hemos estudiado las ***condiciones simples***, como `cuenta <= 10`, `numero != valorCentinela` y `total > 1000`. Las condiciones simples se expresan en términos de los operadores relacionales >, <, >= y <=, y los operadores de igualdad == y !=. Cada expresión evalúa sólo una condición. Para evaluar condiciones múltiples en el proceso de tomar una decisión, ejecutamos estas pruebas en instrucciones separadas o en instrucciones `if` o `if...else` separadas. Algunas veces, las instrucciones de control requieren condiciones más complejas para determinar el flujo de control de una aplicación.

C# cuenta con los ***operadores lógicos*** para que usted pueda formar condiciones más complejas, al combinar las condiciones simples. Los operadores lógicos son && (AND condicional), || (OR condicional), & (AND lógico booleano), | (OR inclusivo lógico booleano), ^ (OR exclusivo lógico booleano) y ! (negación lógica).

Operador AND (&&) condicional

Suponga que deseamos asegurar en cierto punto de una aplicación que dos condiciones son verdaderas, antes de elegir cierta ruta de ejecución. En este caso, podemos utilizar el operador && (***AND condicional***) de la siguiente manera:

```
if ( genero == FEMENINO && edad >= 65 )
   mujeresMayores++;
```

Esta instrucción `if` contiene dos condiciones simples. La condición `genero == FEMENINO` compara la variable `genero` con la constante `FEMENINO`. Por ejemplo, esto podría evaluarse para determinar si una persona es mujer. La condición `edad >= 65` podría evaluarse para determinar si una persona es un ciudadano mayor. La instrucción `if` considera la condición combinada

```
genero == FEMENINO && edad >= 65
```

lo cual es verdadero si, y sólo si *ambas* condiciones son verdaderas. Si la condición combinada es verdadera, el cuerpo de la instrucción `if` incrementa a `mujeresMayores` en 1. Si una o ambas condiciones simples son falsas, la aplicación omite el incremento. Algunos programadores consideran que la condición combinada anterior es más legible si se agregan paréntesis redundantes, por ejemplo:

```
( genero == FEMENINO && edad >= 65 )
```

La tabla de la figura 6.14 sintetiza el uso del operador &&. Esta tabla muestra las cuatro combinaciones posibles de valores `false` y `true` para *expresión1* y *expresión2*. A dichas tablas se les conoce como ***tablas de verdad***. C# evalúa todas las expresiones que incluyen operadores relacionales, de igualdad o lógicos como valores `bool`, los cuales pueden ser `true` o `false`.

Operador OR condicional (||)

Ahora suponga que deseamos asegurar que *cualquiera* o *ambas* condiciones sean verdaderas antes de elegir cierta ruta de ejecución. En este caso, utilizamos el operador || (***OR condicional***), como se muestra en el siguiente segmento de una aplicación:

```
if ( ( promedioSemestre >= 90 ) || ( examenFinal >= 90 ) )
   Console.WriteLine ( "La calificación del estudiante es A" );
```

Esta instrucción también contiene dos condiciones simples. La condición `promedioSemestre >= 90` se evalúa para determinar si el estudiante merece una A en el curso debido a que tuvo un sólido rendimiento a lo largo del semestre. La condición `examenFinal >= 90` se evalúa para determinar si el estudiante merece una A en el