Con lo visto hasta ahora, ya tenemos una idea de la sintaxis de C# . Hemos visto cómo C# permite escribir instrucciones de una forma que es a menudo mucho más concisa que el código VB correspondiente. También hemos visto cómo C# genera todo el código en el fichero fuente, a diferencia de VB, donde gran parte del código de fondo está oculto – algo que hace su código más simple, a costa de reducir la flexibilidad en cuanto a los tipos de aplicaciones que se pueden crear. También hemos visto por primera vez los conceptos que están detrás de la herencia.

Sin embargo, lo que no hemos visto todavía es un ejemplo real de código que pueda escribirse fácilmente en C# y que sea sumamente difícil de escribir en VB. Esto es lo que vamos a ver en el próximo ejemplo, en el que escribiremos un par de clases que ilustran las cosas que podemos hacer con la herencia.

Ejemplo: Empleados y Gerentes

Para este ejemplo, vamos a asumir que estamos desarrollando una aplicación que realiza un proceso sobre datos relativos a los empleados de la compañía. Realmente no vamos a preocuparnos por qué clase de proceso se trata – estamos más interesados en el hecho de que será bastante útil escribir una clase C# (o un módulo de clase VB) que represente a los empleados. Supondremos que esta clase formará parte de un paquete de software que podremos vender a varias compañías, para ayudarles a controlar el pago de los salarios, etc.

El módulo de clase Employee de VB

El siguiente código representa nuestro esfuerzo por crear un módulo de clase `Employee` en VB. El módulo de clase exporta dos propiedades públicas: `EmployeeName` y `Salary`, así como un método público, `GetMonthlyPayment()`, que devuelve la cantidad que la compañía tiene que pagar al empleado cada mes. Esto no es lo mismo que el salario, en parte porque se supone que el salario es anual, y en parte porque queremos permitir la posibilidad de que otros pagos puedan ser añadidos a lo que la compañía paga a sus empleados (como las pagas extraordinarias relacionadas con el rendimiento):

```
'variables locales para almacenar los valores de las propiedades

Private mStrEmployeeName As String 'copia local
Private mCurSalary As Currency 'copia local
Public Property Let Salary(ByVal curData As Currency)
   mCurSalary = curData
End Property

Public Property Get Salary() As Currency
   Salary = mCurSalary
End Property

Public Property Get EmployeeName() As String
   EmployeeName = mStrEmployeeName
End Property

Public Sub Create(sEmployeeName As String, curSalary As Currency)
   mStrEmployeeName = sEmployeeName
   mCurSalary = curSalary
End Sub

Public Function GetMonthlyPayment() As Currency
   GetMonthlyPayment = mCurSalary/12
End Function
```

En la vida real probablemente escribiríamos algo más complejo que esto, pero esta clase será suficiente para ilustrar los conceptos que queremos cubrir. De hecho, ya tenemos un problema con este módulo de clase VB: la mayoría de los nombres de las personas no cambian muy a menudo, por lo que hemos hecho la propiedad `EmployeeName` de sólo lectura. Pero de todas formas tenemos que asignar el nombre la primera vez. Para ello hemos tenido que agregar un método `Create` que asigna el nombre y el salario. Eso significa que el proceso de crear un objeto `Employee` va a ser parecido a esto:

```
Dim Britney As Employee
Set Britney = New Employee
Britney.Create "Britney Spears", 20000
```

Esto funcionará, pero es confuso. El problema al inicializar el objeto `Employee` es que, aunque VB ofrece los métodos `Class_Load` y `Class_Initialize` para este propósito, `Class_Load` no puede recibir ningún parámetro. Esto significa que a través de él no podremos hacer ninguna inicialización particular a una instancia de `Employee` – de manera que tenemos que escribir un método de inicialización separado, `Create`, y esperar que todos los que escriban código cliente de este módulo se acuerden de llamarlo. Esta solución es poco natural, porque no tiene ningún sentido tener un objeto `Employee` que no tenga un nombre y un salario asignado – pero eso es exactamente lo que tenemos en el código anterior, en el breve instante entre la creación del objeto `Britney` y la inicialización del objeto. Siempre que se recuerde llamar a `Create`, esto funcionará bien, pero aquí siempre habrá una fuente potencial de errores.

En C# la situación es completamente diferente. En C# podemos proporcionar parámetros a los constructores (el equivalente C# al método `Class_Load`). Todo lo que tenemos que hacer es asegurarnos de que cuando definamos nuestra clase `Employee` en C#, el constructor reciba dos parámetros, `Name` y `Salary`. Entonces podremos escribir:

```
Employee Britney = new Employee("Britney Spears", 20000.00M);
```

que es mucho más limpio y menos propenso a errores. A propósito, observe la 'M' añadida al salario. Esto se debe a que el equivalente C# del tipo `Currency` de VB se llama `decimal`, y una 'M' añadida a un número en C# indica que queremos que el número sea interpretado como un `decimal`. No es obligatorio proporcionarlo, pero constituye una útil verificación adicional en tiempo de compilación.

La clase Employee en C#

Teniendo presente los comentarios anteriores, podemos presentar nuestra primera definición de la versión C# de `Employee` (observe que presentamos simplemente la definición de la clase, no la definición del espacio de nombres que la contiene):

```
class Employee
{
   private readonly string name;
   private decimal salary;
   public Employee(string name, decimal salary)
   {
      this.name = name;
      this.salary = salary;
   }
   public string Name
   {
      get
      {
         return name;
```

```
            }
        }
        public virtual decimal Salary
        {
            get
            {
                return salary;
            }
            set
            {
                salary = value;
            }
        }
        public decimal GetMonthlyPayment()
        {
            return salary/12;
        }
        public override string ToString()
        {
            return "Name: " + name + ", Salary: $" + salary.ToString();
        }
    }
```

Analizando este código, primero vemos un par de variables privadas, los campos miembros que se corresponden con las variables miembros en el módulo de clase VB. El campo `name` está marcado como `readonly`. Pronto veremos qué significa exactamente esto. A grandes rasgos, esto asegura que este campo deberá asignarse cuando el objeto `Employee` sea creado y que no podrá modificarse después. En C# no es usual utilizar la notación húngara para los nombres de variables, por lo que simplemente se llaman `name` y `salary`, en lugar de `mStrEmployeeName`. La notación húngara no es más que un convenio de nombres mediante el cual prefijamos los nombres de variables con letras que indican su tipo (`mStr`, `mCur`, etc.). Hoy en día ha dejado de ser considerada importante, porque los editores son más sofisticados y pueden proporcionar información automática sobre los tipos de datos; por ello no se recomienda utilizar la notación húngara en los programas C#.

También hay un constructor en nuestra clase `Employee`, un par de propiedades, `Name` y `Salary`, y dos métodos, `GetMonthlyPayment()` y `ToString()`. Los examinaremos todos a continuación.

> *A propósito, observe que los nombres de las propiedades `Name` y `Salary` sólo difieren en las mayúsculas y minúsculas de los nombres de sus campos correspondientes. Esto no es un problema, porque C# distingue mayúsculas de minúsculas. La manera en que hemos nombrado las propiedades y los campos se corresponde con la convención usual en C# y muestra cómo podemos realmente aprovecharnos de la distinción entre mayúsculas y minúsculas.*

El constructor de Employee

Después de las declaraciones de campos en el código anterior, verá un 'método' que tiene el mismo nombre que la clase, `Employee()`, lo cual nos dice que se trata de un constructor. Sin embargo, aunque este constructor reciba parámetros, hace la misma cosa que el método `Create` en la versión VB: utiliza los parámetros para inicializar los campos miembros.

```
        public Employee(string name, decimal salary)
        {
            this.name = name;
            this.salary = salary;
        }
```

Aquí hay un problema sintáctico potencial, porque los nombres obvios para los parámetros son los mismos que los nombres de los campos – `name` y `salary`. Pero hemos resuelto esto utilizando la referencia `this` para

marcar los campos. Podríamos dar nombres diferentes a los parámetros, pero la manera en que lo hemos hecho es bastante clara, y permite que los parámetros mantengan los nombres simples y obvios que se corresponden con sus significados. También es la manera convencional de implementar esta situación en C#.

Podemos ahora explicar el significado preciso del modificador `readonly` en el campo `name`:

```
        private readonly string name;
```

Si un campo es marcado como `readonly`, el único lugar en que se le podrá asignar valor es en el constructor de la clase. El compilador generará un error si encuentra en alguna parte un intento de modificar el valor de una variable `readonly` que no sea en un constructor. Esto proporciona una buena manera de garantizar que una variable no se modifique una vez que ha sido asignada. No es posible hacer algo así en VB, porque VB no ofrece constructores que reciban parámetros, por lo que las variables de nivel de clase en VB tienen que ser inicializadas mediante métodos o propiedades que se llamen después que el objeto haya sido instanciado.

A propósito, este constructor no sólo nos permite proporcionar parámetros para inicializar un objeto `Employee`: nos obliga a hacerlo. Si intentáramos escribir algo así:

```
      Employee Britney = new Employee();   // erróneo
```

este código no compilaría. El compilador generaría un error porque en C# siempre se debe llamar a un constructor cuando un nuevo objeto es creado. Sin embargo, no hemos proporcionado ningún parámetro y el único constructor disponible requiere dos parámetros. Por consiguiente, simplemente no es posible crear un objeto `Employee` sin proporcionar parámetros. Esto proporciona una buena garantía contra errores causados por objetos `Employee` no inicializados.

Es posible dotar a una clase de más de un constructor, de modo que sea posible escoger qué conjunto de parámetros se desea pasar cuando se crea un nuevo objeto de esa clase. Veremos cómo hacer esto más adelante en este capítulo. Para esta clase en particular, nuestro constructor es bastante adecuado.

Propiedades de Employee

Ahora veremos las propiedades `Name` y `Salary`. La sintaxis C# para declarar una propiedad es muy diferente de la sintaxis VB correspondiente, pero los principios básicos son los mismos. Tenemos que definir dos métodos de acceso para leer y escribir los valores de la propiedad. En VB, éstos se tratan sintácticamente como métodos, pero en C# declaramos la propiedad como un todo, y se definen los métodos de acceso dentro de la definición de la propiedad:

```
        public decimal Salary
        {
           get
           {
              return salary;
           }
           set
           {
              salary = value;
           }
        }
```

En VB, el compilador sabe que estamos definiendo una propiedad porque utilizamos la palabra reservada `Property`. En C#, esta información es transmitida por el hecho de que el nombre del miembro está seguido inmediatamente por una llave abierta. Si estuviéramos definiendo un método, sería un paréntesis abierto,

señalizando el inicio de la lista de parámetros, y para un campo sería un punto y coma, señalando el final de la definición.

Otro aspecto a destacar es que las definiciones de los métodos de acceso de lectura y escritura no tienen lista de parámetros. No tiene sentido – sabemos, por el hecho de que `Salary` es de tipo `decimal`, que el método de acceso de lectura devolverá un `decimal` y no recibirá ningún parámetro, mientras que el método de acceso de escritura devolverá `void` y aceptará un parámetro de tipo `decimal`. Para el método de acceso de escritura, este parámetro no se declara explícitamente, pero el compilador siempre interpreta que la palabra reservada `value` se refiere a él.

> *Una vez más, la sintaxis para definir propiedades muestra cómo la sintaxis C# es más compacta y puede ahorrarnos bastante tecleo.*

Al igual que en VB, si desea hacer una propiedad de sólo lectura, simplemente omita el método de acceso de escritura, como hemos hecho con la propiedad `Name`:

```
public string Name
{
   get
   {
      return name;
   }
}
```

Métodos de Employee

La clase también incluye dos métodos: `GetMonthlySalary()` y `ToString()`.

`GetMonthlySalary()` necesita poco comentario, ya que hemos cubierto la mayor parte de la sintaxis más relevante de C#. Simplemente recibe el salario, lo divide entre 12 para convertirlo de anual en mensual y devuelve el resultado:

```
public decimal GetMonthlyPayment()
{
   return salary/12;
}
```

Lo único nuevo aquí con relación a la sintaxis es la instrucción `return`. En VB, especificamos el valor a devolver asignando el resultado a una pseudo-variable que tiene el mismo nombre que la función:

```
GetMonthlyPayment = mCurSalary/12
```

En C# logramos el mismo resultado añadiendo el resultado a devolver a la instrucción `return`, que también especifica que queremos salir de la función, por lo que la instrucción:

```
return salary/12;
```

es realmente equivalente al siguiente código VB:

```
GetMonthlyPayment = mCurSalary/12
Exit Function
```

`ToString()` es ligeramente más interesante. En la mayoría de los casos, cuando escribe una clase C#, es una buena idea asociarle un método `ToString()` que pueda utilizarse para tener una vista rápida del contenido de un objeto. Como se ha mencionado antes, `ToString()` siempre está disponible, porque todas las clases lo heredan de `System.Object`. Sin embargo, la versión de `System.Object` simplemente visualiza el nombre de la clase – no los datos de la instancia de la clase. Microsoft ya ha redefinido este método para todos los tipos de datos numéricos (`int`, `float`, etc.), para visualizar el valor de la variable, y sería bastante útil hacer lo mismo en nuestras clases. Aún cuando no fuera por ninguna otra razón, podría ser una manera útil de visualizar el contenido de un objeto durante la puesta a punto:

```
public override string ToString()
{
   return "Name: " + name + ", Salary: $" + salary.ToString();
}
```

Nuestra redefinición simplemente visualiza el nombre y el salario del empleado. Otro nuevo elemento sintáctico es que hemos declarado específicamente el método con el modificador `override`. C# exige que los métodos redefinidos sean marcados explícitamente de esta manera, y producirá un error de compilación si no lo hace. Esto elimina el riesgo de errores potenciales, como por ejemplo que se redefina un método sin darse cuenta de ello, quizás por no haberse percatado de que un método con ese nombre ya estaba presente en la clase base.

Ahora hemos terminado la codificación de la clase `Employee` tanto en VB como en C# – y hasta ahora, aunque hemos visto cierta dificultad a la hora de construir e inicializar una instancia de `Employee` en la versión VB, ambos lenguajes han resuelto bastante bien nuestros requisitos. Sin embargo, uno de los objetivos de este apéndice es mostrar por qué C# puede ser más poderoso que VB6 en algunas situaciones. De manera que es hora de que mostrar algún código C# que sea muy difícil o imposible de escribir en VB6. Eso es lo que va a ocurrir a partir de ahora. Vamos a comenzar por añadir algunas características a la versión C# de nuestro ejemplo que va a dejar atrás y sin posibilidad de alcanzarnos, a la versión VB. Comencemos con un campo y una propiedad estáticas.

Miembros estáticos

Hemos mencionado en varias ocasiones que en C# las clases pueden tener métodos especiales, conocidos como métodos estáticos, que pueden ser llamados sin necesidad de instanciar un objeto. Estos métodos no tienen contrapartida alguna en VB. De hecho, no sólo los métodos pueden ser estáticos, sino también los campos, las propiedades, o cualquier otro miembro de la clase.

> **El término `static` tiene en C# un significado muy diferente de su significado en VB.**

Para ilustrar cómo trabajan los miembros estáticos y para qué pueden utilizarse, imagine que quisiéramos que nuestra clase `Employee` permitiera la recuperación del nombre de la compañía para la que cada empleado trabaja. Hay una diferencia importante entre el nombre de la compañía y el nombre del empleado, en cuanto a que cada objeto `Employee` representa un empleado diferente, y por consiguiente debe almacenar un nombre de empleado diferente. Esa es la conducta usual de las variables en los módulos de clase VB, y la conducta predefinida para los campos en C#. Sin embargo, si su organización ha comprado el software que contiene la clase `Employee` para su uso interno, obviamente todos los empleados tendrán el mismo nombre de compañía. Sería un despilfarro almacenar el nombre de la compañía separadamente para cada empleado. Sólo se estaría duplicando la cadena innecesariamente. En cambio, lo que queremos justamente es guardar el nombre de la compañía una vez, y que cada objeto `Employee` tenga acceso al mismo valor. Así es cómo trabaja un campo estático. Declaremos ese campo, `companyName`:

```
class Employee
{
   private string name;
   private decimal salary;
   private static readonly string companyName;
```

En este código, simplemente hemos declarado un nuevo campo, pero al marcarlo como `static` estamos diciendo al compilador que almacene esa variable una sola vez, sin importar cuántos objetos `Employee` se creen. En un sentido real, este campo estático está asociado con la clase como un todo, y no con un objeto específico de la clase.

También hemos declarado este campo como de sólo lectura. Esto tiene sentido porque, al igual que el nombre del empleado, el nombre de la compañía no debe modificarse una vez que el programa se esté ejecutando.

Por supuesto, solamente declarar este campo no es suficiente. También tenemos que asegurarnos de que se inicialice con los datos correctos. ¿Dónde debemos hacer eso? Claramente, no en nuestro constructor – el constructor se llamará cada vez que creemos un objeto `Employee`, mientras que queremos inicializar `companyName` una sola vez. La respuesta es que C# ofrece otra construcción para este propósito, conocida como **constructor estático**. El constructor estático actúa como cualquier otro constructor, pero trabaja para la clase en conjunto, y no para objetos particulares. Si define un constructor estático para una clase, éste se ejecutará solamente una vez. El sistema no garantiza cuándo exactamente este constructor se ejecutará, pero sí que será antes de que cualquier código cliente intente acceder a la clase por primera vez. Normalmente esto ocurrirá cuando el programa se inicie. Agreguemos un constructor estático a la clase `Employee`:

```
   static Employee()
   {
      companyName = "Wrox Press Pop Stars";
   }
```

Como de costumbre, identificamos al constructor porque tiene el mismo nombre de la clase. Éste además se identifica como `static`, por lo tanto se trata del constructor estático. No es necesario marcarlo como `public` o `private`, porque no será llamado por ningún otro código C#, sino por el motor de ejecución de .NET. De manera que, sólo para el constructor estático, no necesitamos ningún modificador de acceso.

En nuestro ejemplo, hemos implementado un constructor estático trivial que lleva "cableado" el nombre de la compañía. Más realista sería leer ese dato de una entrada del Registro o de un fichero, o conectarnos a una base de datos para averiguar el nombre de la compañía. A propósito, debido a que el campo `companyName` se ha declarado como estático y de sólo lectura, el constructor estático es el único lugar en el que podemos asignarle legalmente un valor. Nos queda una última cosa por hacer: definir una propiedad pública que nos permita acceder al nombre de la compañía.

```
   public static string CompanyName
   {
      get
      {
         return companyName;
      }
   }
```

La propiedad `CompanyName` también ha sido declarada como estática, y ahora podemos ver la importancia real de un método o propiedad estático: un método o propiedad puede declararse como estático sólo si accede únicamente a campos estáticos, y no accede a ningún dato que esté asociado con un objeto particular.

Como ya hemos visto, la sintaxis para llamar a los miembros estáticos de una clase desde fuera de ella es ligeramente diferente a la utilizada para otros miembros: debido a que un miembro estático se asocia con la clase y no con un objeto, debemos utilizar el nombre de la clase, en lugar del nombre de una variable, para acceder a él:

```
string Company = Employee.CompanyName;
```

El concepto de miembro estático es muy poderoso, y proporciona un mecanismo muy útil para que una clase pueda implementar cualquier funcionalidad que sea la misma para todos los objetos de esa clase. La única manera en que puede lograr algo así en VB es definiendo variables globales. Sin embargo, si hace esto, las variables globales tienen la desventaja de que no están asociadas directamente a ninguna clase – lo que también puede provocar conflictos de nombres.

Otros ejemplos donde podría utilizar miembros de clase estáticos son:

- Podríamos implementar una propiedad `MaximumLength` para nuestra clase `Employee`, o para cualquier otra clase que contenga un nombre, que permitiría especificar la longitud máxima del nombre.
- En C#, la mayoría de los tipos de datos numéricos tienen propiedades estáticas que indican sus valores máximos posibles. Por ejemplo, para averiguar los valores máximos que pueden almacenarse en un `int` y un `float`, se podría escribir:

```
int MaxIntValue = int.MaxValue;
float MaxFloatValue = float.MaxValue;
```

Herencia

Analizaremos ahora con más detalle cómo funciona la herencia de implementación, utilizando un ejemplo. Vamos a suponer que un año después de que hemos implantado nuestro paquete de software, llega la hora de ofrecer la próxima versión. Un aspecto sobre el que nuestros clientes han hecho algún comentario es que algunos de sus empleados realmente son gerentes (managers), y los gerentes normalmente cobran pagas extraordinarias relacionadas con los beneficios, además de los salarios regulares. Esto significa que el método `GetMonthlyPayment()` no ofrece la información completa para los gerentes. El significado práctico de esto es que hay necesidad de añadir alguna manera de procesar también los gerentes.

Para los propósitos de nuestro ejemplo, supondremos que la paga extraordinaria es una constante que puede especificarse al crear el gerente. No queremos complicarnos ahora haciendo cálculos relacionados con los beneficios.

Si estuviéramos codificando en VB, ¿cómo actualizaríamos nuestro software? Hay dos posibles enfoques, pero ambos tienen serias desventajas.

- Podríamos escribir una nueva clase, `Manager`.
- Podríamos modificar la clase `Employee`.

Escribir una nueva clase es probablemente el enfoque que daría menos cantidad de trabajo, ya que probablemente comencemos copiando y pegando todo el código del módulo de clase `Employee` y modificando después nuestra copia del código. El problema es que `Employee` y `Manager` tienen una gran parte de código en común, como es el código relacionado con las propiedades `Name`, `CompanyName`, y `Salary`. Y tener el mismo código duplicado es peligroso. ¿Qué pasa si, en algún momento futuro, surge

alguna razón por la que hay que modificar el código? Algún pobre programador tendrá que recordar que hay que hacer exactamente los mismos cambios en ambas clases. Esto es sumamente propenso a errores. Otro problema es que habrá dos clases no relacionadas con las que el código cliente tendrá que tratar, lo cual hará más difícil utilizar las clases `Employee` y `Manager` a las personas que desarrollan el código (aunque pudiera evitarse esto encapsulando las propiedades comunes en una interfaz y haciendo que `Employee` y `Manager` implementen esa interfaz).

Una alternativa ligeramente diferente sería escribir una clase `Manager`, y poner `Employee` dentro de ella como una variable de alcance en la clase. Esto resuelve el problema de la duplicación del código, pero todavía nos deja dos objetos separados y una sintaxis confusa e indirecta para llamar a los métodos y propiedades de `Employee` (por ejemplo, `objManager.objEmployee.Name`, etc.).

Si optamos por modificar el módulo de clase `Employee`, entonces añadiríamos un campo adicional, probablemente un `Boolean`, que indicara si el empleado es un gerente o no. Entonces, en las partes pertinentes del código, comprobaríamos ese valor lógico en una instrucción `If` para determinar qué hacer. Esto resolvería el problema de tener dos clases no relacionadas – regresamos a una sola clase. Sin embargo, aquí se introduce una nueva dificultad: hemos dicho que hemos decidido, al cabo de un año o más, añadir soporte para los gerentes. Eso significa que el módulo de clase `Employee` ha sido puesto a punto, y se sabe que está funcionando correctamente. ¿Queremos realmente comenzar a bucear y a descomponer en pedazos un código que ya funciona, con todo el riesgo asociado de introducir nuevos errores?

Repentinamente, hemos alcanzado un punto en el que VB no puede ofrecer ninguna solución satisfactoria. Debido al título de esta sección, no se sorprenderá al saber que C# ofrece una forma de sortear este problema mediante la herencia.

Ya hemos visto que la herencia implica añadir o sustituir características de las clases. En nuestro ejemplo anterior, nuestra clase `SquareRootForm` añade varias cosas a la clase base .NET `System.Windows.Forms.Form`: controles para el acceso a los campos miembros del formularios y un manejador de eventos. En el ejemplo `Employee` vamos a mostrar cómo añadir y reemplazar características de una clase base: definiremos una clase `Manager` derivada de `Employee`. Añadiremos un campo y una propiedad que representan la paga extraordinaria, y reemplazaremos el método `GetMonthlyPayment()` (en aras de la integridad, redefiniremos también `ToString()`, para que visualice la paga extraordinaria junto con el nombre y el salario). Esto significa que tendremos una clase separada. Pero no tenemos que duplicar ningún código, y no tendremos que hacer ningún cambio a la clase `Employee`. Podría pensar que todavía tenemos el problema de las dos clases diferentes – lo que hará más difícil escribir el código cliente, pero veremos que C# tiene también una respuesta para esto.

Heredando de la clase Employee

Antes de definir la clase `Manager`, necesitamos hacer un pequeño cambio a `Employee`:

```
public virtual decimal GetMonthlyPayment()
{
   return salary/12;
}
```

El cambio que hemos hecho consiste en hacer `virtual` el método `GetMonthlyPayment()`. A grandes rasgos, ésta es la forma en que se dice en C# que éste es un método que, en principio, puede ser redefinido.

> *Podría pensar que esto significa que estamos modificando la clase base, lo cual derrumba nuestro argumento acerca de que no es necesario cambiar la clase base. Sin embargo, añadir la palabra reservada `virtual` realmente no es el tipo de cambio mayor que conlleva el*

riesgo de nuevos errores – con el enfoque de VB, teníamos que volver a escribir las implementaciones de varios métodos. Además, cuando Ud. escribe clases en C# planifica de antemano que métodos son candidatos para ser redefinidos. Si éste fuera un ejemplo de la vida real, `GetMonthlyPayment()` seguramente habría sido declarado `virtual` desde el primer momento, de manera que realmente sólo habríamos tenido que añadir la clase `Manager`, sin hacer ningún cambio a la clase `Employee`.

La clase Manager

Ahora podemos definir la clase `Manager`:

```
   class Manager : Employee
   {
      private decimal bonus;
      public Manager(string name, decimal salary, decimal bonus)
         :   base(name, salary)
      {
         this.bonus = bonus;
      }
      public Manager(string name, decimal salary)
         :   this(name, salary, 100000M)
      {
      }

      public decimal Bonus
      {
       get
         {
            return bonus;
         }
      }

      public override string ToString()
      {
         return base.ToString() + ", bonus: " + bonus;
      }

      public override decimal GetMonthlyPayment()
      {
         return base.GetMonthlyPayment() + bonus/12;
      }
   }
}
```

Además de la implementación casi completa de la clase `Employee` que hemos heredado, `Manager` contiene los siguientes miembros:

- Un campo, `bonus`, que se utilizará para guardar la paga extraordinaria del gerente, y una propiedad correspondiente.
- Una redefinición del método `GetMonthlyPayment()`, así como una nueva redefinición de `ToString()`.
- Dos constructores.

El campo `bonus` y la propiedad `Bonus` correspondiente no necesitan ninguna discusión adicional. Sin embargo, veremos en detalle los métodos redefinidos y los nuevos constructores, ya que éstos ilustrarán importantes características del lenguaje C#.

Redefinición de métodos

Nuestra redefinición de `GetMonthlyPayment()` es bastante simple. Observe que lo hemos marcado con la palabra reservada `override`, para indicarle al compilador que estamos redefiniendo un método de la clase base, como hicimos con `Employee.ToString()`:

```
public override decimal GetMonthlyPayment()
{
   return base.GetMonthlyPayment() + bonus/12;
}
```

Nuestra redefinición también contiene una llamada a la versión de la clase base de este método. Para ello se utiliza otra palabra reservada del lenguaje, `base`. `base` trabaja de la misma manera que `this`, con la excepción de que indica específicamente que queremos tomar un método, propiedad, etc., de la definición de la clase base. Si hubiéramos querido, podríamos alternativamente haber implementado nuestra redefinición de `GetMonthlyPayment()` de la siguiente forma:

```
public override decimal GetMonthlyPayment()
{
   return (Salary + bonus)/12;
}
```

pero hemos querido mostrar la palabra reservada `base`, y por ello hemos elegido hacerlo de esta manera. A propósito, una cosa que no podríamos hacer es lo siguiente:

```
public override decimal GetMonthlyPayment()
{
   return (salary + bonus)/12;   // ¡ERROR!
}
```

Este código parece casi exactamente la versión anterior, excepto que estamos accediendo directamente al campo del salario, en lugar de hacerlo mediante la propiedad `Salary`. Ud. puede pensar que esta es la solución más eficaz, porque estamos ahorrándonos una llamada a método. El problema es que el compilador producirá un error, porque el campo `salary` se ha declarado como `private`. Eso significa que nadie fuera de la propia clase `Employee` puede tener acceso a este campo. Incluso las clases derivadas NO tienen conocimiento de los campos `private` de sus clases base.

Si desea que las clases derivadas (pero no otras clases no relacionadas) puedan acceder a un campo de la clase base, C# proporciona un nivel de protección alternativo, `protected`:

```
protected decimal salary;   // podíamos haber hecho esto
```

Si un miembro de una clase se declara como `protected`, sólo es visible dentro de esa clase y en las clases derivadas. Sin embargo, en general se aconseja que mantenga todos los campos como privados, exactamente por la misma razón que se aconseja mantener todas las variables privadas en los módulos de clase VB: porque ocultando la implementación de una clase (o un módulo de clase) está haciendo más fácil el mantenimiento futuro de esa clase. Normalmente, se utilizará el modificador `protected` para las propiedades y métodos que están previstos solamente para permitir que las clases derivadas accedan a ciertas características de la

definición de la clase base.

Los constructores de Manager

Tenemos que añadir por lo menos un constructor a la clase `Manager`, por dos razones:

- Ahora hay una pieza adicional de información, la paga extraordinaria del gerente, que tenemos que especificar al crear una instancia de `Manager`.
- A diferencia de los métodos, las propiedades y los campos, los constructores NO son heredados por clases derivadas.

De hecho, hemos añadido dos constructores. Esto se debe a que hemos decidido que la paga extraordinaria del gerente normalmente tiene como valor predefinido $100.000, si no se especifica otra explícitamente. En VB podemos especificar valores por defecto para los parámetros de métodos, pero C# no permite hacerlo directamente. En cambio, C# ofrece una técnica más poderosa que permite lograr el mismo efecto: la sobrecarga de métodos. Los dos constructores definidos aquí nos permitirán ilustrar esta técnica.

El primer constructor de `Manager` recibe tres parámetros:

```
public Manager(string name, decimal salary, decimal bonus)
   :   base(name, salary)
{
   this.bonus = bonus;
}
```

La primera cosa que observamos en este constructor es una llamada al constructor de la clase base, que utiliza una sintaxis ligeramente rara. Esta sintaxis se conoce como **inicializador de constructor**. A cualquier constructor se le permite llamar a otro constructor antes de que se ejecute. Esta llamada se hace mediante un inicializador de constructor, utilizando la sintaxis mostrada anteriormente. Un constructor puede llamar a cualquier otro constructor de la misma clase, o a un constructor de la clase base. Esto podría parecer restrictivo, pero se hace por buenas razones, en términos de imponer una arquitectura bien diseñada en los constructores. Estos problemas se discuten en el Capítulo 5. La sintaxis para el inicializador del constructor requiere dos puntos, seguidos por las palabras reservadas `base` o `this`, para especificar a qué clase pertenece este segundo constructor, seguido por los parámetros que le pasamos al segundo constructor.

El constructor mostrado recibe tres parámetros. Sin embargo, dos de ellos, `name` y `salary`, realmente están allí para inicializar los campos de la clase base en `Employee`. Estos parámetros realmente son responsabilidad de la clase `Employee`, y no de la clase `Manager`, por lo cual lo que hacemos es simplemente pasarlos al constructor de `Employee` para que se encargue de ellos – eso es lo se logra mediante la llamada `base` (`name`, `salary`). Como hemos visto antes, el constructor de `Employee` utilizará estos parámetros para inicializar los campos `name` y `salary`. Finalmente, tomamos el parámetro de la paga extraordinaria, que es responsabilidad de la clase `Manager`, y lo utilizamos para inicializar el campo correspondiente. El segundo constructor de `Manager` que hemos proporcionado utiliza también una lista de inicialización de constructor:

```
public Manager(string name, decimal salary)
   :   this(name, salary, 100000M)
{
}
```

En este caso, lo que ocurre es que asignamos al tercer parámetro el valor predefinido, y pasamos todo al constructor de tres parámetros. Por supuesto, a su vez, el constructor de tres parámetros llamará al constructor de la clase base para que se encargue de los parámetros `name` y `salary`. Podría preguntarse por qué no hemos utilizado la siguiente forma alternativa de implementar el constructor de dos parámetros:

```
public Manager(string name, decimal salary)
   :   base(name, salary)   // no tan buena
{
   this.bonus = 100000M;
}
```

La razón es que esto involucra alguna potencial duplicación de código: los dos constructores, cada uno por separado, inicializan el campo de la paga extraordinaria, y esto podría causar problemas en el futuro en caso de que ambos constructores necesiten ser modificados separadamente, como por ejemplo si en alguna versión futura de `Manager` modificamos la forma de almacenar la paga extraordinaria. En general, tanto en C# como en VB, se debe evitar duplicar código siempre que sea posible. Por esta razón, la implementación anterior del constructor de dos parámetros es más recomendable.

Sobrecarga de métodos

El hecho de que hemos proporcionado dos constructores para la clase `Manager` ilustra el principio de la sobrecarga de métodos en C#. La sobrecarga de métodos significa que una clase puede tener más de un método con el mismo nombre, siempre que las signaturas de esas variantes sean diferentes. Hemos ilustrado la sobrecarga para los constructores, pero exactamente los mismos principios se aplican para todos los métodos.

> **No confunda los términos *sobrecarga de métodos* (overload) y *redefinición de métodos* (override). A pesar del parecido de los términos, son conceptos diferentes y completamente no relacionados.**

Cuando el compilador encuentra una llamada a un método que ha sido sobrecargado, examina los parámetros que se le está intentando pasar, con vistas a resolver qué variante del método debe ser llamada. En el caso de la creación de un objeto `Manager`, debido a que un constructor recibe tres parámetros y el otro sólo dos, lo primero que el compilador examinará es la cantidad de parámetros. Por lo tanto, si escribe:

```
Manager SomeManager = new Manager ("Name", 300000.00M);
```

el compilador hará que se instancie un objeto `Manager` llamando al constructor de dos parámetros – lo cual significa que a la paga extraordinaria se le asignará su valor predefinido, 100000.00M. Si por el contrario escribe esto:

```
Manager SomeManager = new Manager ("Name", 300000.00M, 50000.00M);
```

el compilador hará que se llame al constructor de tres parámetros, por lo que a la paga extraordinaria se le asignará el valor indicado, 50000.00M. Si hay varias sobrecargas disponibles, pero el compilador es incapaz de encontrar una adecuada, producirá un error de compilación. Por ejemplo, si escribiera:

```
Manager SomeManager = new Manager (100, 300000.00M, 50000.00M);   // ¡ERROR!
```

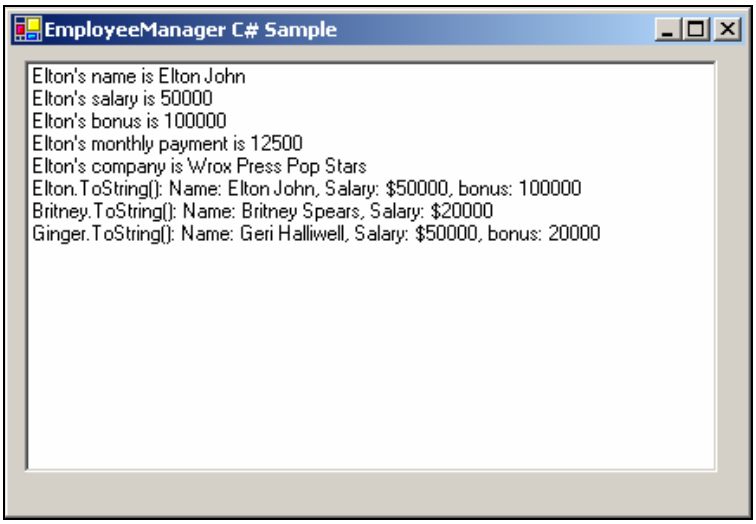
Obtendrá un error de compilación, porque los dos constructores disponibles de `Manager` requieren como primer parámetro una cadena y no un valor numérico. El compilador de C# puede realizar automáticamente algunas conversiones entre diferentes tipos numéricos, pero no convertirá automáticamente un valor numérico a una cadena.

Finalmente, señalaremos que, como hemos comentado antes, C# no permite asignar valores por defecto a los

parámetros de métodos de la forma que lo hace VB. Sin embargo, es muy fácil lograr el mismo efecto utilizando la sobrecarga de métodos, como hemos hecho en este ejemplo. La técnica usual es que las sobrecargas que tienen menos parámetros proporcionen valores predefinidos para los parámetros restantes y entonces llamen a otras sobrecargas.

Utilización de las clases Employee y Manager

Ahora que hemos completado la definición de las clases `Employee` y `Manager`, podemos escribir algún código que las utilice. De hecho, si descarga el código fuente de este proyecto del sitio web de Wrox Press, encontrará que hemos definido estas dos clases como parte de un proyecto de formulario Windows, bastante similar al ejemplo `SquareRoot`. Sin embargo, en este caso el formulario principal tiene sólo un control – un cuadro de lista. Utilizamos el constructor de la clase del formulario principal (llamada `MainForm`) para crear un par de instancias de los objetos `Employee` y `Manager`, y visualizar los datos de esos objetos en el cuadro de lista. La salida del programa es similar a la siguiente:



El código utilizado para generar estos resultados es:

```
public MainForm()
{
   InitializeComponent();
   Employee Britney = new Employee("Britney Spears", 20000.00M);
   Employee Elton = new Manager("Elton John", 50000.00M);
   Manager Ginger = new Manager("Geri Halliwell", 50000.00M,
                                20000.00M);
   this.listBox1.Items.Add("Elton's name is $" + Elton.Name);
   this.listBox1.Items.Add("Elton's salary is $" + Elton.Salary);
   this.listBox1.Items.Add("Elton's bonus is " +
                                ((Manager)Elton).Bonus);
   this.listBox1.Items.Add("Elton's monthly payment is $" +
                                Elton.GetMonthlyPayment());
   this.listBox1.Items.Add("Elton's company is " +
                                Employee.CompanyName);
   this.listBox1.Items.Add("Elton.ToString(): " + Elton.ToString());
   this.listBox1.Items.Add("Britney.ToString(): " +
```

```
                                        Britney.ToString());
            this.listBox1.Items.Add("Ginger.ToString(): " + Ginger.ToString());
        }
```

Este código se explica por sí mismo a partir de lo que hemos aprendido de C# hasta ahora, con la excepción de un pequeño detalle: a uno de los objetos `Manager`, `Elton`, se hace referencia mediante un `Employee`, en lugar de un `Manager`. A continuación explicaremos cómo esto funciona.

Referencias a clases derivadas

Veamos más de cerca la instrucción en la que se hace referencia a un objeto de la clase `Manager` mediante una variable declarada como referencia a `Employee`:

```
        Employee Elton = new Manager("Elton John", 50000.00M);
```

Esta es una sintaxis C# absolutamente legal. La regla es bastante simple: si declara una referencia a un tipo `B`, se permite que esa referencia haga referencia a una instancia de `B`, o a una instancia de cualquier clase **derivada** de `B`. Esto funciona debido a que cualquier clase derivada de `B` también implementa los métodos, propiedades, etc. que `B` implementa. Por eso que en el ejemplo anterior han funcionado los accesos a `Elton.Name`, `Elton.Salary` y `Elton.GetMonthlyPayment()`. Esto es correcto. El hecho de que `Employee` implementa todos estos miembros garantiza que cualquier clase derivada de `Employee` también lo hará. Por lo que no importa si una referencia apunta a una clase derivada – también podemos utilizar la referencia para acceder a cualquier miembro de la clase en la que la referencia se define con la total seguridad de que ese miembro existe en la clase derivada.

Por otra parte, observe la sintaxis que hemos utilizado para acceder a la propiedad `Bonus` de `Elton`: `((Manager)Elton).Bonus`. En este caso, necesitamos convertir explícitamente a `Elton` a una referencia a `Manager`, porque `Bonus` no es implementada por `Employee`. El compilador sabe esto, y generaría un error de compilación si intentásemos acceder a `Bonus` a través de una referencia a `Employee`. Esa línea de código realmente es una forma abreviada de escribir lo siguiente:

```
        Manager ManagerElton = (Manager) Elton;
        this.listBox1.Items.Add("Elton's bonus is " + ManagerElton.Bonus);
```

Al igual que en VB, la transformación entre tipos de datos en C# es conocida como conversión (cast). Podemos ver en el código anterior que la sintaxis para la conversión consiste en poner el nombre del tipo de datos de destino entre paréntesis delante del nombre de la variable que se desea convertir. Por supuesto, en primer lugar, el objeto al que se hace referencia debe ser del tipo adecuado. Si con los datos de este ejemplo escribiéramos:

```
        Manager ManagerBritney = (Manager) Britney;
```

el código compilaría correctamente, pero al ejecutarlo obtendríamos un error, porque el motor de ejecución .NET vería que `Britney` es una instancia de `Employee`, no de `Manager`. Se permite que las referencias hagan referencia a instancias de las clases derivadas, pero no a instancias de clases bases de su tipo estático. No se permite que una referencia `Manager` a haga referencia a un objeto `Employee` (no podría permitirse, porque si lo hiciéramos, ¿qué pasaría si intentáramos acceder a la propiedad `Bonus` a través de esa referencia?).

A propósito, no hemos dicho nada sobre lo que significa que se produzca un error en tiempo de ejecución. C# ofrece un mecanismo muy sofisticado para esto, conocido como excepciones, que mencionaremos brevemente

más adelante en este apéndice.

Debido a que VB no soporta la herencia de implementación, no hay ningún paralelo directo en VB para el soporte de C# a las referencias a objetos de las clases derivadas. Sin embargo, hay alguna similitud con el hecho de que en VB se puede declarar una referencia a una interfaz, y entonces no importa a qué tipo de objeto la interfaz hace referencia, siempre que el objeto en cuestión implemente la interfaz. Si estuviéramos escribiendo las clases `Employee` y `Manager` en VB, podríamos haber definido una interfaz `IEmployee` que los dos módulos de clase implementarían, y entonces acceder a las características de `Employee` mediante esta interfaz.

Arrays de Objetos

Un beneficio importante de tener referencias capaces de hacer referencia a instancias de las clases derivadas es que podemos formar arrays de referencias a objetos, donde los diferentes objetos del array pueden ser de tipos diferentes. Esto es similar a la situación en Visual Basic, donde podríamos formar arrays de referencias a interfaces, sin preocuparnos por el hecho de que estas referencias pueden estar implementadas por objetos de clases completamente diferentes.

No hemos visto todavía cómo C# trata los arrays, por lo que nos arriesgaremos y volveremos a escribir el código de prueba para las clases `Employee` y `Manager`, de manera que construya un array de referencias a objetos. Esto código revisado también puede descargarse del sitio web Wrox Press como el ejemplo `EmployeeManagerWithArrays`. Este es el nuevo código:

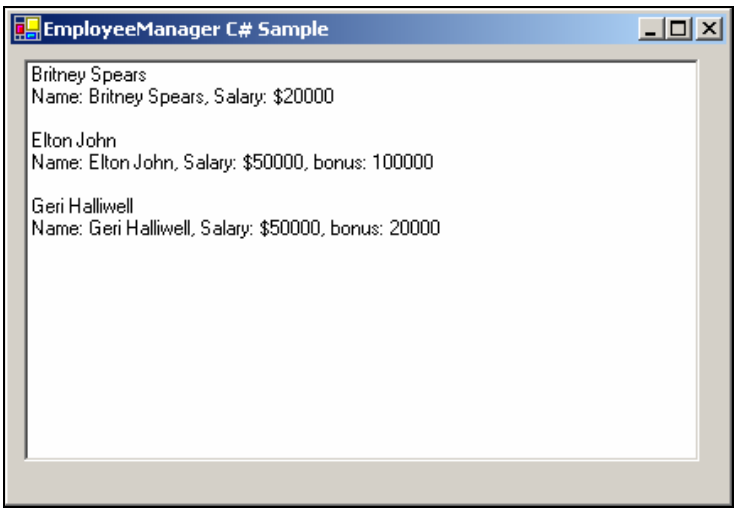
```
public MainForm()
{
   InitializeComponent();

   Employee Britney = new Employee("Britney Spears", 20000.00M);
   Employee Elton = new Manager("Elton John", 50000.00M);
   Manager Ginger = new Manager("Geri Halliwell", 50000.00M,
                                20000.00M);

   Employee[] Employees = new  Employee[3];
   Employees[0] = Britney;
   Employees[1] = Elton;
   Employees[2] = Ginger;

   for (int I=0 ; I<3 ; I++)
   {
      this.listBox1.Items.Add(Employees[I].Name);
      this.listBox1.Items.Add(Employees[I].ToString());
      this.listBox1.Items.Add("");
   }
}
```

Aquí simplemente accedemos a la propiedad `Name` y al método `ToString()` de cada elemento del array. Al ejecutar este código se producen los siguientes resultados:



Aquí podemos ver que C# utiliza corchetes para los arrays. Esto significa que, a diferencia de VB, no hay ningún peligro de confusión sobre si estamos hablando de un array o de una llamada a un método. La sintaxis para declarar un array es similar a ésta:

```
Employee[] Employees = new Employee[3];
```

Como puede Ud. ver, se declara un array de variables de cierto tipo poniendo corchetes detrás del nombre del tipo. Un array en C# siempre es un objeto-referencia (aún cuando sus elementos sean de tipos simples como `int` o `double`), de manera que realmente hay dos fases: declarar la referencia e instanciar el array. Para hacer esto más claro, podríamos dividir la línea de código anterior en dos de la siguiente forma:

```
Employee[] Employees;
Employees = new Employee[3];
```

No hay ninguna diferencia entre lo que estamos haciendo aquí y la forma en que se instancian los objetos, con la excepción de que estamos utilizando corchetes para indicar que se trata de un array. Observe también que el tamaño del array se establece cuando se instancia el objeto – la propia referencia no contiene detalles del tamaño del array – sólo cuántas dimensiones tiene. Las dimensiones se indican mediante comas en la declaración del array; por ejemplo, si quisiéramos declarar un array bidimensional 3x4 de valores `double`, escribiríamos esto:

```
double [,] DoubleArray = new double[3,4];
```

Hay otras variantes sintácticas ligeramente diferentes para declarar arrays, pero nos adheriremos a ésta. Una vez que tenemos instanciado el array, simplemente asignamos valores a sus elementos de la manera usual. Observe, sin embargo, que una diferencia entre C# y VB es que en C# los índices de un array siempre comienzan en 0. En VB tenemos la posibilidad de elegir entre 0 y 1, utilizando la instrucción `Option Base`. También en VB es posible especificar el límite inferior que se desee para un array. Pero esto realmente no ofrece ningún beneficio adicional y puede afectar el rendimiento, porque significa que siempre que se acceda a un elemento de un array, el código deberá hacer una verificación adicional contra el límite inferior de ese array. C# no soporta el cambio de la base de un array de esta manera.

En el código anterior, una vez que hemos inicializado los elementos del array, sólo lo recorremos. La extraña apariencia de la sintaxis del bucle `for` puede causarle algún dolor de cabeza – lo examinaremos pronto.

Observe que como el array se ha declarado como un array de `Employee`, sólo podemos acceder a aquellos miembros de cada objeto que están definidos para la clase `Employee`. Si quisiéramos acceder a la propiedad `Bonus` de cualquier objeto del array, tendríamos que convertir la referencia correspondiente a una referencia a `Manager`, lo cual significaría verificar primero si el objeto realmente es un `Manager`. Eso no es difícil de hacer, pero está más allá del alcance de este apéndice.

Por otro lado, aunque estemos utilizando referencias a `Employee`, siempre se utilizará la versión correcta de `ToString()`. Si uno de los objetos a los que estamos haciendo referencia es un objeto `Manager`, cuando llamemos a `ToString()`, la versión de `ToString()` definida en la clase `Manager` será la que se ejecute para ese objeto. En eso radica la belleza de la redefinición de métodos en C#. Se puede reemplazar un método en las clases derivadas y tener la certeza total de que, no importa mediante qué tipo de referencia se acceda a ese objeto, siempre se ejecutará el método correcto.

El bucle for

Ahora echemos un vistazo a esa extraña sintaxis del bucle `for`. Lo que estamos haciendo es el equivalente C# de este código VB:

```
Integer I
For I = 1 To 3
   listBox1.Items.Add "Detalles del Empleado"
Next
```

La idea del bucle `For` en VB es que Ud. comienza inicializando una variable (la variable de control del bucle), y cada vez que ejecuta una iteración se añade algo a esa variable de control, hasta que se sobrepase un valor final. Esto es bastante útil, pero no ofrece al programador casi ninguna flexibilidad con relación a cómo el bucle funciona. Aunque se puede cambiar el valor de incremento o incluso hacer el incremento negativo utilizando la palabra reservada `Step`, el bucle trabaja siempre contando, y la condición de salida del bucle siempre es que la variable haya llegado a algún valor mínimo o máximo.

El bucle `for` de C# generaliza este concepto. La idea básica del bucle `for` en C# es que al inicio del bucle se hace algo, al llegar al final de una iteración se hace algo para prepararse para pasar a la siguiente, y para determinar cuándo el bucle debe terminar se realiza alguna comprobación. Comparando las versiones Visual Basic y C# de la instrucción `for`, la situación es la siguiente:

	VB	**C#**
Al inicio del bucle	Inicializar la variable de control del bucle	Hacer algo
Para verificar si debe terminar el bucle	¿Ha excedido la variable de control del bucle un cierto valor?	Verificar alguna condición
Al final de cada iteración	Incrementar la variable de control del bucle	Hacer algo

Esto puede parecer un poco vago, ¡pero ofrece una gran flexibilidad! Por ejemplo, en C#, en lugar de añadir una cierta cantidad a la variable de control del bucle en cada iteración, se podría multiplicar su valor por algún número. O en lugar de añadir una cantidad fija, se podría añadir un valor que haya sido leído de un fichero y

que cambia con cada iteración. La comprobación no tiene que limitarse a verificar el valor de la variable de control del bucle – podría ser, por ejemplo, verificar si se ha alcanzado el final de un fichero. Lo que esta flexibilidad garantiza es que, mediante la elección conveniente de las acciones de inicio, verificación y acción final de cada iteración, el bucle `for` puede realizar muy eficientemente la misma tarea que cualquiera de las instrucciones de bucle de VB – `For`, `Foreach`, `Do` y `While`, o alternativamente, el bucle puede trabajar de alguna manera exótica, para la que no hay ningún equivalente en VB. El bucle `for` de C# realmente le da total libertad para controlarlo de cualquier manera que sea la más adecuada para la tarea a la que se enfrenta.

> *Debemos señalar, sin embargo, que C# también ofrece bucles* `foreach`*,* `do` *y* `do...while`*, para las situaciones en que Ud. las pueda necesitar.*

Veamos alguna sintaxis concreta. Recuerde que la versión C# del bucle `for` anterior es la siguiente:

```
for (int I=0 ; I<3 ; I++)
{
   this.listBox1.Items.Add(Employees[I].Name);
   this.listBox1.Items.Add(Employees[I].ToString());
   this.listBox1.Items.Add("");
}
```

Como puede ver, la instrucción `for` incluye tres elementos diferentes entre paréntesis. Estos elementos están separados por puntos y comas:

- El primer elemento es la acción que se realiza a la entrada del bucle para inicializarlo. En este caso declaramos e inicializamos la variable de control del bucle.
- El segundo elemento es la condición que se evaluará para determinar si el bucle debe terminar. En este caso, la condición es que `I` debe ser menor que 3. El bucle continuará siempre que esta condición sea `true` y terminará cuando la condición se evalúe a `false`. La condición se evaluará al principio de cada iteración, por lo que si resulta `false` al inicio, la instrucción dentro del bucle no se ejecutará ni una sola vez.
- El tercer elemento es la instrucción que se ejecutará al final de cada iteración del bucle, como preparación para la siguiente iteración. Los bucles en Visual Basic siempre trabajan incrementando algún número, y esto es exactamente lo que hacemos en este caso.

La sintaxis parece poco familiar, pero una vez que se haya acostumbrado a ella podrá utilizar el bucle `for` de formas muy poderosas. Por ejemplo, suponga que queremos visualizar todas las potencias de 2 que sean menores de 4000 en un cuadro de lista. Podríamos escribir esto:

```
for (int I = 2 ; I<4000 ; I*=2)
   listBox1.Items.Add(I.ToString);
```

Este tipo de efecto se puede lograr en VB, pero es ligeramente más difícil – para este bucle particular, probablemente optaríamos por un bucle `while` en VB.

Otras características de C#

Ahora ya hemos terminado de ver los ejemplos. En el resto del apéndice se examinarán brevemente algunas características de C# que Ud. deberá tener en cuenta al hacer la transición de VB a C# y que no hemos discutido todavía – en particular, alguno de los conceptos de C# que tienen relación con los tipos de datos y operadores.

Los tipos de datos

Como ya hemos indicado, los tipos de datos disponibles en C# difieren en varios detalles de los disponibles en Visual Basic. No sólo eso, sino que todos los tipos de datos en C# tienen características que normalmente se asociarían con un objeto. Por ejemplo, como hemos visto, cada tipo, incluso los tipos simples como `int` y `float`, soportan llamadas a métodos (a propósito, esta característica no afecta el rendimiento).

Aunque los tipos disponibles en C# son ligeramente diferentes de los tipos de VB, la mayoría de los tipos con que está familiarizado en VB tienen equivalentes directos en C#. Por ejemplo, en lugar del `Double` de VB, C# ofrece el tipo `double`. Donde VB tiene `Date`, C# incluye una clase base .NET, `DateTime`, que implementa una gran cantidad de métodos y propiedades para permitirle extraer o asignar la fecha utilizando diferentes formatos.

Una excepción, sin embargo, es `Variant`, para el que no hay ningún equivalente directo en C#. El tipo `Variant` de VB es un tipo muy genérico que, en cierta medida, sólo existe para dar soporte a los lenguajes de script que no disponen de ningún otro tipo de datos. La filosofía de C#, sin embargo, es que el lenguaje garantice un fuerte control de tipos. La idea es que si en cada punto del programa Ud. tiene que indicar explícitamente los tipos de datos a los que está haciendo referencia, se elimina una gran fuente de errores de tiempo de ejecución. Debido a esto, un tipo `Variant` no es muy apropiado en C#. No obstante, puede haber situaciones en las que Ud. necesita hacer referencia a una variable sin indicar de qué tipo es la variable, y para esos casos C# tiene el tipo `object`. El tipo `object` de C# es muy parecido al `Object` de VB. Sin embargo, en VB, `Object` se refiere específicamente a un objeto COM, y por consiguiente sólo puede utilizarse para hacer referencia a objetos, lo cual, en términos de VB, significa tipos-referencia. No es posible utilizar una referencia a un objeto de VB, por ejemplo, para hacer referencia a un `Integer` o a un `Single`. En C#, en cambio, puede utilizarse `object` para hacer referencia a cualquier tipo de datos .NET, y debido a que en C# todos los tipos de datos son tipos de datos .NET, esto significa que se puede convertir cualquier cosa, legítimamente, en un objeto, incluso los `int`, `float`, y todos los tipos de datos predefinidos. En este sentido, la clase `object` en C# realiza un papel similar a `Variant` en VB.

Tipos-valor y tipos-referencia

En Visual Basic hay una clara distinción entre los tipos-valor y los tipos-referencia. Entre los tipos-valor se incluyen la mayoría de los tipos de datos predefinidos: `Integer`, `Single`, `Double` e incluso `Variant` (aunque, estrictamente hablando, las variables `Variant` también pueden contener referencias). Entre los tipos-referencia se incluyen los objetos, incluyendo los módulos de clase que Ud. defina, y los objetos ActiveX. Como habrá visto en los ejemplos de este apéndice, C# también hace la distinción entre los tipos-valor y los tipos-referencia. Sin embargo, C# permite un poco más de flexibilidad, en la medida que permite, al definir una clase, especificar que esa clase debe ser un tipo-valor. Esto se hace declarando la clase como una estructura (`struct`). En lo que concierne a C#, una estructura es básicamente un tipo especial de clase que se representa como valor y no como referencia. El coste asociado a instanciar y destruir estructuras una vez que se ha terminado de trabajar con ellas es menor que el asociado a instanciar y destruir clases. Sin embargo, C# restringe en cierta medida las características soportadas por las estructuras. En particular, no es posible derivar clases u otras estructuras de estructuras. El razonamiento es que las estructuras están previstas para ser usadas para definir objetos muy ligeros y simples, para los cuales la herencia no es muy apropiada. De hecho, todas las clases predefinidas de C#, como `int`, `long`, `float`, `double`, son realmente estructuras de .NET, por lo que podemos aplicarles métodos como `ToString()`. El tipo de datos `string`, sin embargo, es un tipo-referencia y es realmente una clase.

Operadores

Tenemos que decir un par de frases sobre los operadores en C#, porque trabajan de una manera algo diferente a la de los operadores de VB, y esto puede sorprenderlo si está acostumbrado a la manera en que los operadores funcionan en VB. En VB hay realmente dos tipos de operadores:

- El operador de asignación, =, que asigna valores a las variables.
- Todos los demás, como +, -,*, y /, que devuelven algún valor.

Hay una distinción importante en que ninguno de los operadores, aparte de =, produce ningún efecto colateral, en términos de la modificación de algún valor. Por otra parte, = asigna un valor pero no devuelve nada. No hay ningún operador que haga ambas cosas.

En C# esta categorización simplemente no existe. La regla en C# es que **todos** los operadores devuelven un valor, y algunos operadores tienen el efecto adicional de también asignar algún valor a una variable. De hecho ya hemos visto un ejemplo de esto, cuando examinamos el operador de suma y asignación, +=:

```
int A=5, B=15;
A += B;   // realiza una operación aritmética Y asigna el resultado (20) a A
```

De hecho, += devuelve un valor y asigna un valor. Devuelve el nuevo valor que se ha asignado. Por eso podríamos escribir:

```
int A=5, B=15;
int C = (A+=B);
```

Esto tendrá como resultado que a ambas variables, `A` y `C`, se les asignará el valor 20. El operador de asignación, =, también devuelve un valor: devuelve el valor que se le ha asignado a la variable que está en el lado izquierdo de la expresión. Esto significa que se puede escribir código como éste:

```
C = (A = B);
```

Este código asigna a `A` el valor de `B`, y también asigna a `C` ese mismo valor. También puede escribir esta instrucción más simplemente como:

```
C = A = B;
```

Un uso común de este tipo de sintaxis es evaluar alguna condición dentro una instrucción `if`, y simultáneamente asignar el resultado de la evaluación a una variable de tipo `bool` (el equivalente C# del `Boolean` de VB), de manera que podamos reutilizar ese valor después:

```
// se asume que X y Y son variables ya inicializadas

bool B;
if ( B = (X==Y) )
   DoSomething();
```

Este código parece confuso al principio, pero es bastante lógico. Veámoslo por partes. Lo primero que hará el ordenador es evaluar la condición `X==Y`. Dependiendo de si `X` e `Y` contienen los mismos datos, esta evaluación devolverá `true` o `false`, y este valor se asignará a `B`. Sin embargo, como el operador de asignación también devuelve el valor que se asigna, la expresión completa `B = (X==Y)` también devolverá ese mismo valor (`true` o `false`). Este valor devuelto será utilizado por la cláusula `if` para determinar si debe ejecutar la llamada a `DoSomething()`. El resultado de este código es que la condición `X==Y` se evalúa para determinar si las instrucciones condicionales deben ejecutarse, y al mismo tiempo se almacena el resultado de esta evaluación en la variable `B`.

El operador ternario

No tenemos espacio en este apéndice para revisar todos los operadores disponibles en C#. Éstos se describen detalladamente en los Capítulos 3-6. Sin embargo, mencionaremos el operador ternario (también conocido como operador condicional) porque tiene una sintaxis muy poco usual. El operador ternario se forma con los símbolos `?` y `:`. Actúa sobre tres operandos, y es realmente equivalente a una instrucción `If` de VB. Sintácticamente se utiliza de esta forma:

```
// B, X e Y son variables ya inicializadas. B es de tipo bool.

B ? X : Y
```

Cuando se ejecuta el operador, la primera expresión – la que está antes del signo `?` – se evalúa. Si el resultado es `true`, entonces se devolverá como valor de la expresión el resultado de evaluar la segunda expresión; si la primera expresión se evalúa a `false`, se devolverá el resultado de evaluar la tercera expresión. Esto proporciona una sintaxis sumamente compacta para asignar condicionalmente el valor de una variable. Por ejemplo, podríamos escribir:

```
int Z = (X==Y) ? 5 : 8;
```

que produciría exactamente el mismo efecto que:

```
int Z;
if (X == Y)
   Z = 5;
else
   Z = 8;
```

Resumen

En este apéndice hemos presentado una breve introducción a C# desde el punto de vista de la comparación con Visual Basic. Hemos encontrado varias diferencias en la sintaxis. En general, la sintaxis de C# permite expresar la mayoría de las instrucciones de una manera más compacta. También hemos encontrado muchas similitudes entre los lenguajes – por ejemplo en el uso de las clases (o módulos de clase de VB), de los tipos-valor y los tipos-referencia, y en muchas de las estructuras sintácticas. Sin embargo, también hemos visto cómo C# soporta muchas características poderosas, particularmente aquellas relacionadas con la herencia y la programación orientada a objetos clásica, que no están disponibles en VB.

Hacer el tránsito de VB a C# requiere un poco de aprendizaje, pero bien vale la pena, porque la metodología de C# permite crear no sólo cualquier tipo de aplicación que se podría crear en VB, sino también una amplia gama de otras aplicaciones que serían difíciles o imposibles de implementar, con un diseño bien estructurado y mantenible, en VB. Con C# también obtendrá todos los beneficios adicionales asociados con el uso de la Plataforma .NET.