

Argentina: \$5,90 (recargo al interior\$0,20) México: \$30

USERS

Microsoft®

Curso teórico y práctico de programación

Desarrollador .net

Con toda la potencia
de **Visual Basic .NET** y **C#**

La mejor forma de aprender
a programar desde cero



Basado en el programa
Desarrollador Cinco Estrellas
de Microsoft

3

Visual Basic

Funciones propias y primeros
ejercicios prácticos

C# desde cero

Sintaxis básica, sintaxis de clase
y funciones propias



ISBN 978-987-1347-43-8



00003



9 789871 347438

Incluye CD-ROM #3



RedUSERS

COMUNIDAD DE TECNOLOGIA



EL SITIO Nº1 DE TECNOLOGIA

Noticias al instante // Entrevistas y coberturas exclusivas //
Análisis y opinión de los máximos referentes // Reviews de
productos // Trucos para mejorar la productividad //
Regístrate, participa, y comparte tus opiniones



SUSCRIBITE

SIN CARGO A CUALQUIERA
DE NUESTROS NEWSLETTERS
Y RECIBÍ EN TU CORREO
ELECTRÓNICO TODA LA
INFORMACIÓN DEL UNIVERSO
TECNOLÓGICO ACTUALIZADA
AL INSTANTE



INGRESÁ A
redusers.com/suscribirse-al-newsletter
¡Y REGÍSTRATE YA!

www.reduserspremium.blogspot.com.ar



Foros



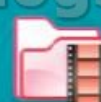
Encuestas



Tutoriales



Agenda de eventos



Videos



¡Y mucho más!



redusers.com

Seguinos en:



www.facebook.com/redusers



www.twitter.com/redusers



www.youtube.com/redusersvideos



Procedimientos y funciones

Aprenderemos a utilizar herramientas que nos permitirán agilizar nuestra práctica de programador.

Cuando escribimos programas, normalmente nos encontramos con porciones de código que se repiten. En estos casos, resulta una buena práctica escribirlas una sola vez y, luego, referenciarlas en cada ocasión en que sean necesarias. En el mundo de la programación estructurada, estas porciones de código se denominan **procedimientos** o **funciones**. Si bien la motivación de ambos es la misma, tienen entre sí semánticas diferentes. Los procedimientos están destinados a realizar tareas que no necesiten devolver nada a quien las indicó (por ejemplo, el resultado es imprimir algo en pantalla), en tanto que las funciones devuelven un valor a quien las llama (por ejemplo, realizar un cálculo sobre la base de datos proporcionados y brindar el resultado).

Procedimientos

En Visual Basic .NET, los procedimientos se implementan usando la palabra clave **Sub**. Cada uno posee un nombre y un bloque de código que será el código por ejecutar cuando éste sea invocado. El bloque de código se delimita por las palabras **Sub** y **End Sub**. Dentro del procedimiento podemos escribir cualquier sentencia que necesitemos, incluso, llamar a otros procedimientos. Un procedimiento se llama, simplemente, escribiendo su nombre. De manera opcional, se puede utilizar la palabra clave **Call** más el **Nombre-DelMetodo**, pero no es obligatorio hacerlo

(esto se mantiene así por compatibilidad con versiones anteriores del lenguaje). Veamos un ejemplo de procedimiento. Supongamos que estamos trabajando en una aplicación de consola con varias salidas por pantalla y que queremos dividir las distintas salidas con una secuencia de 80 guiones.

Para lograrlo, escribimos un procedimiento que imprima los guiones y, luego, lo invocamos cada vez que sea necesario:

```
Sub ImprimirGuiones()  
  For i as Integer=1 To 80  
    Console.Write("-")  
  Next  
End Sub
```

Luego, en el código de la aplicación, podemos usar este procedimiento cuantas veces precisemos, escribiendo las siguientes líneas:

```
ImprimirGuiones()  
Console.WriteLine(debe)  
Console.WriteLine(haber)  
ImprimirGuiones()  
Console.WriteLine(saldo)  
ImprimirGuiones()
```

Cada procedimiento posee un nombre y un bloque de código que será el que se deba ejecutar cuando éste sea invocado.

Los parámetros permiten que un procedimiento se comporte cada vez de manera diferente.

Parámetros

Los procedimientos como el visto anteriormente son útiles, pero muchas veces necesitamos que uno de ellos haga distintas cosas dependiendo del contexto donde se lo esté utilizando. Por ejemplo, siguiendo con el procedimiento `ImprimirGuiones`, quizás haya situaciones en las que tengamos que imprimir los guiones, pero no exactamente 80. Entonces surge un planteo interesante: ¿escribimos un nuevo procedimiento que imprima la cantidad de guiones necesaria?, ¿o modificamos el que ya tenemos para indicar cuántos guiones queremos imprimir? Obviamente, la primera alternativa no es la mejor, ya que estaríamos duplicando código y, además, solucionando un caso particular, que luego quizá no nos sirva otra vez y tengamos que escribir otro método más. La segunda alternativa, en cambio, es practicable y nos permitirá ahorrar código.

La herramienta con la que contamos para indicarle al procedimiento cuántos guiones queremos imprimir son los parámetros. Un parámetro

es un valor que podemos pasarle al procedimiento, y que éste recibe como si fuese una variable. Al ver el parámetro como una variable, el procedimiento puede utilizarlo para alterar su comportamiento.

Modifiquemos el procedimiento `ImprimirGuiones` para que acepte un parámetro que le indique la cantidad de guiones que se deben imprimir:

```
Sub ImprimirGuiones(ByVal cantidad As Integer)
    For i As Integer=1 To cantidad
        Console.WriteLine("-")
    Next
End Sub
```

En este ejemplo, el procedimiento recibe un parámetro que representa la cantidad de caracteres por imprimir, y lo utiliza como límite superior del ciclo `For`. Modifiquemos el ejemplo de uso para imprimir separadores de distinta longitud:

```
ImprimirGuiones(80)
Console.WriteLine(debe)
Console.WriteLine(haber)
ImprimirGuiones(40)
Console.WriteLine(saldo)
ImprimirGuiones(80)
```

Parámetros por valor y por referencia

Visual Basic .NET brinda dos formas de pasaje de parámetros a un procedimiento o función: por valor o por referencia. Cuando se pasa un parámetro por valor, lo que se le pasa al procedimiento es una copia del valor original. Si el procedimiento modifica el valor del parámetro, sólo afectará a la copia. Cuando se pasa un valor por referencia, lo que en realidad se está pasando es la dirección de memoria donde se almacena el valor. Si se pasa un parámetro por referencia, cualquier modificación que el procedimiento o función haga, se hará sobre el valor original, ya que se tiene la dirección de memoria



FIGURA 001 | Mediante el uso de parámetros, podemos variar el comportamiento de los procedimientos.



donde está. Los parámetros por referencia son una herramienta para hacer que un procedimiento se comunique con quien lo invocó, dado que puede devolver valores mediante los parámetros por referencia (también conocidos como parámetros de salida).

En Visual Basic .NET los parámetros por valor se especifican mediante el modificador **ByVal**, mientras que aquellos por referencia son especificados a través del modificador **ByRef**. Podemos ilustrar este concepto con un ejemplo simple. Supongamos que queremos escribir un procedimiento al que le pasemos un nombre de persona, y nos devuelva, en un parámetro, un saludo para ella. Esto se hace de la siguiente manera:

```
Sub Saludar(ByVal nombre As String, ByRef
saludo As String)
    Saludo = "Hola " & nombre
End Sub
```

Los parámetros permiten devolver valores a quien llama a un procedimiento o función, pero no se recomienda su uso.

En este caso, como el parámetro "saludo" es por referencia, al ser modificado dentro del procedimiento, también cambiará su valor original.

Funciones

Las funciones son bloques de código que siempre devuelven un valor a quien las llama. En Visual Basic .NET las funciones se definen mediante la palabra clave **Function**. Además, luego de la lista de parámetros (si

Tipos de parámetros

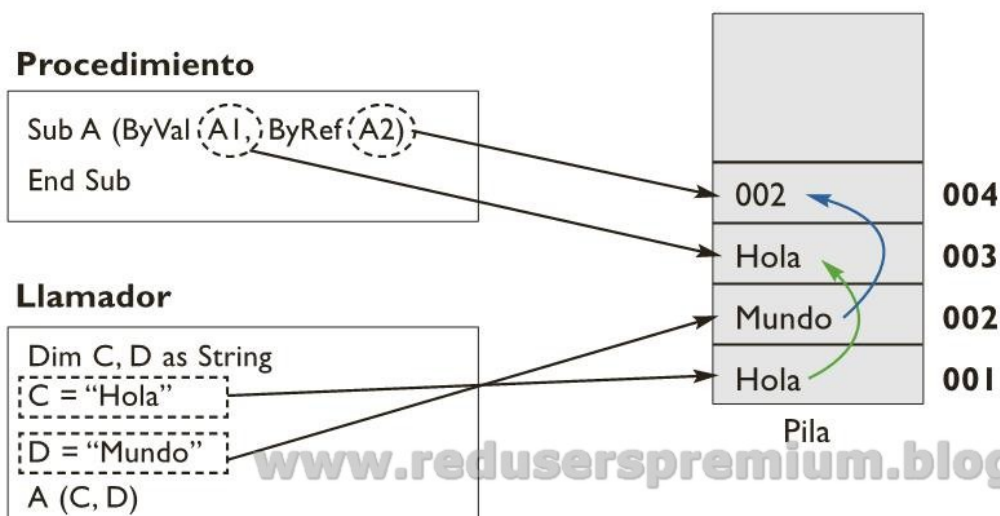


FIGURA 002 | Cuando se pasa un parámetro por valor, en la pila se coloca una copia, en tanto que si es por referencia, se coloca la dirección de memoria del valor original.

Hay que tener en cuenta que los parámetros opcionales deben estar siempre al final de la lista de parámetros.

es que tiene), se debe indicar el tipo de dato del valor devuelto, mediante la palabra **As** seguida de un nombre de tipo. Otra diferencia importante con los procedimientos es que las funciones deben incluir la instrucción **Return** al menos una vez. Ésta se encarga, básicamente, de devolver el valor de la función al llamador. Veamos un ejemplo para ilustrar el uso y la escritura de las funciones. Supongamos que queremos escribir una función que reciba dos valores y devuelva la diferencia entre el mayor y el menor de ellos. El código será el siguiente:

```
Function SumaEjemplo(ByVal a As Integer,
ByVal b as Integer) As Integer
    If a > b Then
        Return a - b
    Else
        Return b - a
    End Function
```

Como podemos ver, en este ejemplo la función recibe dos parámetros de tipo **Integer**

y devuelve un valor del mismo tipo. También podemos ver que hay dos instrucciones **Return**. Es importante saber que podemos usar todas las instrucciones **Return** que queramos o necesitemos, pero debemos tener siempre, por lo menos, una.

Parámetros opcionales

A veces ocurre que ya tenemos un procedimiento escrito y se nos presenta una situación en la que puede servirnos, pero necesitaríamos pasarle un parámetro más para resolver ese caso en particular. El problema es que deberemos buscar todos los lugares donde utilizamos el procedimiento para agregar el parámetro en la llamada. Además, como el nuevo parámetro se usa para resolver un problema que antes no teníamos, no sabremos qué valor pasarle. Para resolver esto, Visual Basic .NET nos da la posibilidad de crear parámetros opcionales, que no es necesario pasar.

Para declarar un parámetro como opcional, recurrimos a la palabra clave **Optional**, pero hay que tener en cuenta que los parámetros opcionales deben estar siempre al final de la lista de parámetros. Esto es importante, porque si queremos agregar un parámetro más, éste deberá ser también opcional o insertarse antes del primero de este tipo. Para terminar de comprender bien cómo hay que utilizarlo, veamos un breve ejemplo referido al uso de parámetros opcionales:

```
Sub Procedimiento(ByVal a As Integer, Optional
ByVal b as Integer)
    ' Hacer algo con los parámetros
End Sub
' . . . . .
'Ejemplo de uso:
Procedimiento(3,5)
Procedimiento(3)
```

§ La instrucción Return

Esta instrucción corta la ejecución y sale de la función. Es importante recordar esto, porque en caso de que haya que hacer alguna otra tarea dentro de la función, deberemos hacerla antes del **Return**; de lo contrario, nunca se ejecutará.



Sintaxis de clases

Los siguientes constructores nos permitirán escribir aplicaciones orientadas a objetos desde Visual Basic.

Una de las grandes diferencias sintácticas y semánticas entre Visual Basic .NET y su predecesor es que VB.NET brinda soporte completo para programación orientada a objetos. Si bien en su versión 6 intentó incluir soporte de orientación a objetos, recién con la llegada de .NET ésta se hizo más sólida y formal. A continuación, veremos los constructores que ofrece Visual Basic .NET para escribir aplicaciones orientadas a objetos.

Visual Basic .NET tiene soporte completo para programación orientada a objetos.

Creación de clases

Los dos conceptos más importantes y usados en la programación orientada a objetos son las **clases** y los **objetos**:

- Las clases son abstracciones de objetos de la realidad que se quieren modelar.
- Los objetos son instancias particulares de las clases; cada uno pertenece a una clase (agrupados por propiedades comunes) pero tiene identidad propia.

La definición de una clase está comprendida por la asignación de un nombre de clase utilizado para agrupar sus métodos y propiedades. En Visual Basic .NET, la forma de definir una clase es a través de las palabras clave `Class` y `End Class`:

```
Class NombreDeLaClase
    'Aquí van los métodos y propiedades
    ' . . .
End Class
```

Todo el código (métodos o propiedades con diferentes modificadores de alcance) será par-

te de la clase y estará disponible para ser usado por sus instancias. Las clases son consideradas como tipos de datos, en el sentido de que desde el punto de vista sintáctico, las instancias son variables cuyo tipo de dato es la clase a la que pertenecen.

Para crear una instancia de una clase, debemos definir el objeto como si fuese una variable, utilizando la palabra clave `Dim`. Sin embargo, a diferencia de los tipos básicos, para usar un objeto, además de declararlo, es preciso

§ Modificadores de alcance

Los modificadores de alcance definen el alcance o visibilidad de cada elemento de una clase. En Visual Basic .NET hay varios modificadores de este tipo, de los cuales los más conocidos son:

- **Public**: Todo lo definido como `Public` será accesible para cualquiera que use la clase, y dentro de la clase misma también.
- **Private**: Los definidos como `Private` sólo serán visibles dentro de la clase; nadie más tendrá acceso a ellos.
- **Protected**: Un elemento definido de este modo será visible dentro de la clase y dentro de las clases que hereden de ella.

En Visual Basic .NET podemos crear variables de instancia tal como si fuesen variables comunes.

crear la instancia, utilizando la palabra reservada **New**, de esta manera:

```
Dim objeto As Clase 'Definimos el objeto
objeto = New Clase() 'Creamos la instancia
```

Semánticamente, el operador **New** reserva un espacio en memoria para guardar el nuevo objeto y llamar a un método especial de la clase, denominado **Constructor**.

Toda clase hereda automáticamente de una clase del framework .NET llamada **Object**, que tiene un constructor utilizado en la mayoría de los casos. Sin embargo, cuando definimos una clase, tal vez necesitemos realizar alguna acción interna a ella al momento de crear una instancia. Para hacerlo, definimos un constructor propio, usando la palabra clave **New** como nombre de método, y cuando creamos una nueva instancia de un objeto de clase **Ejemplo**, se invocará el método **New** creado:

```
Class Ejemplo
Sub New()
' Aquí podemos hacer alguna acción
End Sub
End Class
```

Variables de instancia

Un objeto puede contener variables que determinen su estado. En Visual Basic .NET podemos crear variables de instancia tal como si fuesen variables comunes, con la diferencia de que

podemos reemplazar la palabra clave **Dim** por un modificador de alcance. Si utilizamos **Dim**, la variable tendrá un alcance privado. Supongamos que estamos modelando cuentas bancarias. Una cuenta bancaria tiene un saldo, que podemos mantener como variable de instancia, de la siguiente forma:

```
Class CuentaBancaria
Private saldo As Double
End Class
```

En el ejemplo, la variable **saldo** es una variable de instancia y, en este caso, es privada a la clase.

Propiedades

Los objetos tienen propiedades. Por ejemplo, si estamos modelando figuras geométricas, podemos identificar propiedades tales como la cantidad de lados y la longitud de cada uno de ellos. En una clase, debemos modelar las propiedades. Al hacerlo, estamos diciendo que las instancias de la clase contarán con esas propiedades, pero aún no les damos valor. Luego, cuando creamos instancias, cada una tendrá sus propios valores para las propiedades. En Visual Basic .NET, las propiedades de las clases se definen utilizando la palabra clave **Property**. A esta altura, el lector puede pensar que las propiedades y las variables de instancia son lo mismo, pero no es así. La diferencia está en que, al escribir una propiedad, podemos escribir código para cuando ésta recibe el valor y para cuando alguien consulta el valor, con lo cual podremos hacer validaciones o cálculos. Además, uno de los objetivos de la programación orientada a objetos es lograr un alto grado de encapsulamiento para facilitar el mantenimiento del código. Se entiende por encapsulamiento el hecho de ocultar la implementación real de partes de la clase a quien la usa.



Para ilustrar el uso de propiedades, supongamos que queremos que quien usa la clase CuentaBancaria pueda conocer y modificar el saldo, que actualmente está como una variable privada de instancia. Para esto, debemos escribir una propiedad con alcance público que devuelve y asigne valor a la variable de instancia. El código, entonces, quedará así:

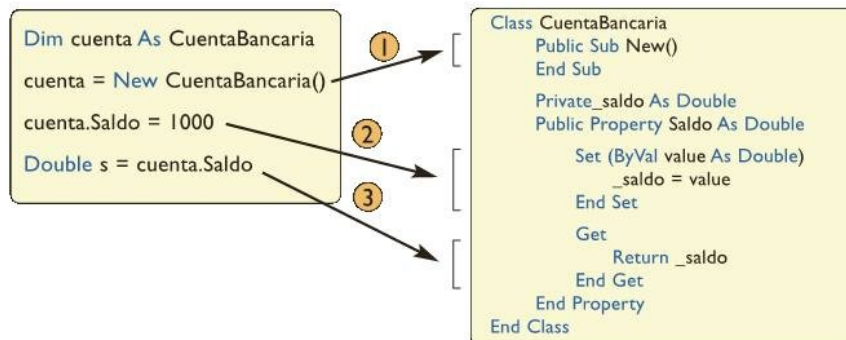
```
Class CuentaBancaria
Private _saldo As Double
Public Property Saldo As Double
Get
Return _saldo
End Get
Set(ByVal value As Double)
_saldo = value
End Set
```

En Visual Basic .NET, las propiedades de las clases se definen utilizando la palabra clave Property.

```
End Property
End Class
```

Analicemos un poco este código de ejemplo. Las palabras **Public Property Saldo As Double** indican que se está definiendo una propiedad pública llamada en este caso “Saldo”, de tipo Double. A continuación, viene lo que se denomina **getter**, que es la porción de código que se ejecutará cuando alguien lea el valor de la propiedad.

Invocar el setter y el getter



- 1 Cuando creamos una instancia de clase, se ejecuta el método New, llamado Constructor.
- 2 Cuando asignamos valor a la propiedad se invoca el setter de la clase. El parámetro del setter recibe automáticamente el valor que se encuentra a la derecha del operador de asignación.
- 3 Cuando consultamos el valor de la propiedad, se invoca el getter, que actúa como una función devolviendo el valor de la propiedad.

FIGURA 003 | Cuando alguien lee el valor de la propiedad, se invoca el getter; y cuando se asigna un valor, se invoca el setter.

Esta porción de código se comporta como una función, que debe tener, al menos, una instrucción Return para devolver el valor de la propiedad.

De manera similar, sigue una porción de código llamada **setter**, que indica el código por ejecutar cuando alguien asigne valor a la propiedad. El setter debe tener un parámetro del mismo tipo de la propiedad.

Desde el lado del cliente de la clase, el acceso a la propiedad es totalmente transparente y se puede ver como si fuese una variable de instancia normal:

```
Dim cuenta As CuentaBancaria
cuenta = New CuentaBancaria()
cuenta.Saldo = 1000
Console.WriteLine(cuenta.Saldo)
```

Las propiedades no siempre deben tener un getter y un setter. Si queremos que los clientes de la clase sólo conozcan el valor de la propiedad pero no puedan modificarlo, podemos hacerla de sólo lectura. Para lograrlo, agregamos el modificador **ReadOnly** antes de la palabra **Property** y, además, omitimos el setter. Para ilustrar este caso, modificamos la clase CuentaBancaria para no guardar el saldo en una variable de instancia, sino que queremos calcularlo como la diferencia entre el haber y el debe de la cuenta, que son variables de instancia. Además, no queremos que el saldo sea modificado.

Nuestra clase quedará así:

```
Class CuentaBancaria
    Private _debe, _haber As Double

    Public ReadOnly Property Saldo As Double
        Get
            Return _haber - _debe
        End Get
    End Property
End Class
```

En el ejemplo, el getter se ocupa de hacer el cálculo del saldo y de devolverlo como valor de la propiedad. Como una propiedad puede ser de sólo lectura, también puede ser de sólo escritura, mediante el modificador **Writeonly** y omitiendo el getter. Así, el cliente de la clase podrá asignarle valores a la propiedad, pero no podrá consultarla.

Métodos

Los objetos se caracterizan por su estado y por su comportamiento. El estado está dado por las propiedades y variables de instancia, mientras que el comportamiento está dado por los métodos. Los métodos son los procedimientos y funciones que un objeto puede realizar y que pueden modificar su estado. En Visual Basic .NET los métodos se escriben como cualquier procedimiento o función, y su visibilidad depende de los modificadores de alcance empleados. Continuando con la clase CuentaBancaria que hemos definido, ya que el saldo nos ha quedado como una propiedad de sólo lectura, la única forma de modificarlo es mediante depósitos y extracciones, que serán sendos métodos de la clase y que cambiarán el valor de las variables de instancia (*_debe* y *_haber*). El código es el siguiente.

Los métodos se escriben como cualquier procedimiento o función, y su visibilidad depende de los modificadores de alcance.



```
Class CuentaBancaria
  Private _debe, _haber As Double

  Public Readonly Property Saldo As Double
    Get
      Return _haber - _debe
    End Get
  End Property

  Public Sub Depositar(monto As Double)
    _haber = _haber + monto
  End Sub

  Public Sub Extraer(monto As Double)
    _debe = _debe + monto
  End Sub
End Class
```

La herencia permite especificar una relación entre dos clases, mediante la cual la clase heredera recibe los atributos y el comportamiento de su clase padre. La herencia puede ser vista como una relación “es un”. Por ejemplo, un triángulo es una figura, un cuadrado es una figura; por lo tanto, podemos decir que el triángulo y el cuadrado heredan de figura. En Visual Basic .NET la herencia se especifica utilizando la palabra clave **Inherits**, seguida del nombre de la clase de la que se hereda. Para continuar con el ejemplo de la figura geométrica, podemos escribir el siguiente código. En este ejemplo, estamos especificando que la clase Triangulo hereda de la clase Figura.

```
Public Class Figura
  Public Lados As Integer
End Class

Public Class Triangulo
  Inherits Figura

  'Código de la clase triángulo
End Class
```

Herencia

Una de las características más útiles e interesantes de la programación orientada a objetos es la reutilización de código mediante la herencia.

Jerarquía de objetos

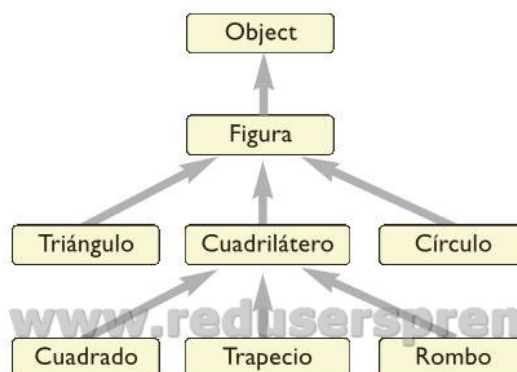


FIGURA 004 | Mediante la herencia se establecen jerarquías de objetos, que pueden verse gráficamente mediante un árbol similar a los árboles genealógicos.

LA HERENCIA PUEDE SER VISTA COMO UNA RELACIÓN “ES UN”. POR EJEMPLO, UN TRIÁNGULO ES UNA FIGURA, UN CUADRADO ES UNA FIGURA; POR LO TANTO, PODEMOS DECIR QUE EL TRIÁNGULO Y EL CUADRADO HEREDAN DE FIGURAS.

También podemos decir que la clase Triangulo es una “subclase” de Figura. Como Triangulo hereda de Figura, contiene también el atributo Lados, con lo cual ya hemos logrado reutilizar parte del código. Si creamos una instancia de la clase Triangulo, podremos escribir lo siguiente:

```
Dim miTriangulo As New Triangulo()  
miTriangulo.Lados = 3
```

Redefinición de métodos

La herencia no sería muy valiosa si cada clase derivada quedara atada al comportamiento de la clase padre. En cualquier lenguaje orientado a objetos, una clase hija debe poder redefinir el comportamiento de su padre, para adaptarlo a sus propias necesidades y características. En Visual Basic .NET la forma de redefinir y rescribir métodos es mediante la palabra clave **Overrides**, pero para que un método pueda ser redefinido, en la clase base debe contener la palabra **Overridable** (que se puede rescribir).

La técnica consiste en identificar aquellos métodos que será necesario rescribir en las clases hijas y colocarles la palabra clave **Overridable**. Luego, cada clase hija puede redefinir estos métodos (no es obligatorio hacerlo), proveyendo el código necesario para actuar de acuerdo con sus necesidades. Para ilustrar este concepto, imaginemos que tenemos una clase Mamifero con un método

llamado EmitirSonido. Como el sonido que se emita dependerá de cada tipo de mamífero en particular, podemos hacer que cada clase derivada tenga su propia implementación del método, imprimiendo en pantalla la onomatopeya del sonido. Vamos a verlo en código:

```
Public Class Mamifero  
    Public Overridable Sub EmitirSonido()  
    End Sub  
End Class
```

```
Public Class Perro  
    Inherits Mamifero  
    Public Overrides Sub EmitirSonido()  
        Console.WriteLine("Guau...")  
    End Sub  
End Class
```

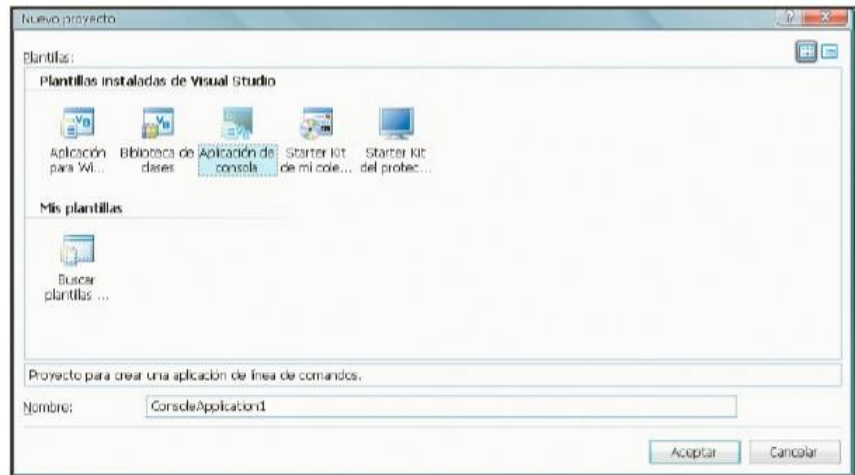
```
Public Class Gato  
    Inherits Mamifero  
    Public Overrides Sub EmitirSonido()  
        Console.WriteLine("Miau...")  
    End Sub  
End Class
```

Ejercicios

Antes de pasar a algunos ejercicios resueltos para afirmar los conceptos estudiados, veamos una serie de instrucciones y herramientas que vamos a necesitar.



FIGURA 005 | Para crear un nuevo proyecto de consola, elegimos la plantilla correspondiente cuando Visual Studio lo solicite.



Aplicaciones de consola

Para los ejercicios, vamos a hacer unas sencillas aplicaciones de consola. Éstas son pequeños programas con una interfaz muy simple (sin ventanas), muy similar a un programa de DOS, pero basadas en la consola del sistema operativo Windows 2000 o superior.

Para crear una aplicación de consola con Visual Studio 2005, debemos ir al menú **Archivo/Nuevo/Proyecto**. Al hacerlo, se desplegará una ventana donde tenemos que elegir el tipo de proyecto que queremos hacer, que será “Aplicación de Consola”. Por último, en la parte inferior de esta ventana, ingresamos el nombre que le queremos dar a nuestro proyecto.

La clase Console

El framework .NET nos provee de una clase llamada **Console**, con métodos para acceder a la consola en este tipo de aplicaciones. Los dos métodos que más vamos a usar son **WriteLine** y **ReadLine**. El primero nos permite escribir mensajes en la consola, mientras que el segundo sirve para leer una cadena de texto que el usuario ingresó en ella.

En algunos ejercicios, necesitaremos interpretar el texto leído con el método **ReadLine** como un número entero, para lo cual utilizaremos el método **Parse** del tipo de dato **Integer**, haciendo

Integer.Parse. Este método recibe una cadena de texto y devuelve el valor numérico adecuado, o dispara una excepción si el texto no corresponde a un entero válido.

El método Main

Toda aplicación de consola debe tener un método llamado **Main**, que es el punto de entrada de la aplicación. Al momento de ejecutar una aplicación de consola, el primer método que se ejecuta es **Main**; es decir que ahí debemos colocar todo el código y las llamadas que necesitemos hacer al comienzo de la aplicación. Al crear una nueva aplicación de consola con Visual Studio 2005, automáticamente se crea un método **Main** vacío, para que podamos empezar a trabajar.

☺ Una buena práctica

A medida que avanzamos en la lectura de todos estos nuevos conceptos, es conveniente ir realizando los diferentes ejemplos en la consola, y probar las distintas variantes para comprender, realmente, cuál es el efecto que se obtiene en cada caso. Ésta es una práctica imprescindible para internalizar cada uno de los temas tratados.

PRACTICA01

Ejercicios prácticos

A continuación, desarrollaremos algunos ejemplos prácticos en los que aplicaremos los conceptos ya aprendidos.

Veamos nuestro primer ejemplo utilizando el entorno de Visual Basic. En este caso, vamos a escribir una aplicación que muestre por pantalla los números impares entre el 1 y el 20, usando un ciclo **For** pero sin usar **Step**. Para realizar este ejercicio, vamos a escribir un ciclo **For** que cuente desde 1 hasta 20, y como no podemos usar el **Step** para ir de dos en dos sobre los números impares, vamos a usar un **If** para determinar si el valor de la variable contadora del **For** es par o impar. Como todos los números pares son divisibles por 2, para saber si un número es impar, podemos preguntar si no es divisible por 2; es decir, si el resto de dividirlo por 2 es 1.

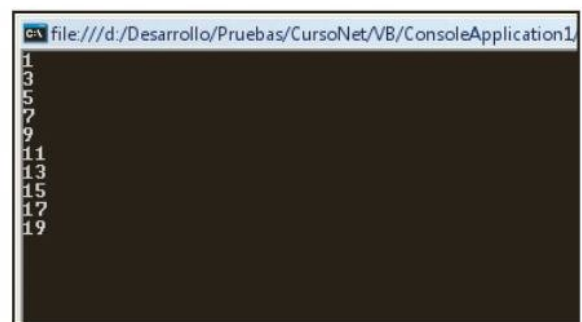
Vayamos entonces al código. Dentro del método **Main** creado cuando generamos el proyecto con Visual Studio, escribimos el código:

```
Dim i As Integer
For i = 1 To 20
    If i Mod 2 = 1 Then
        Console.WriteLine(i)
    End If
Next
```

Recordemos que el operador **Mod** devuelve el resto de dividir el primer operando por el

segundo (con división entera, por supuesto). Al ejecutar la aplicación (presionando <F5>), veremos que se ejecuta pero se cierra inmediatamente, sin que alcancemos a ver los resultados. Esto es así porque el IDE crea la consola para ejecutar la aplicación, y cuando ésta termina, la consola se cierra en forma automática. Para evitarlo, y poder ver la salida de nuestro programa, agregamos una llamada a **Console.ReadKey()** luego de la palabra **Next**. Esto hará que el programa se quede esperando la presión de una tecla.

Si presionamos cualquiera, la aplicación terminará, pero si ejecutamos la aplicación, veremos la consola con los números impares entre el 1 y el 19.



```
file:///d:/Desarrollo/Pruebas/CursoNet/VB/ConsoleApplication1/
1
3
5
7
9
11
13
15
17
19
```

FIGURA 006 | Vemos cómo la aplicación de consola se queda en espera, mostrando los resultados obtenidos.

Console.ReadKey() hará que el programa se quede esperando la presión de una tecla.

El mes correspondiente

Veamos otro ejemplo. En este caso, vamos a escribir una aplicación que pida un número de mes y muestre el nombre correspondiente. Repetimos la operación hasta que el usuario ingrese un cero como número de mes.



Para concretar este ejercicio, debemos utilizar dos de las estructuras de control que hemos aprendido: **While** para repetir hasta que el usuario ingrese un cero, y **Select Case** para seleccionar el nombre del mes según el número ingresado. En este ejercicio necesitamos usar el método `ReadLine` de la clase `Console`, para leer el valor ingresado por el usuario y asignarlo a una variable de tipo entero. En una nueva aplicación de consola, agregamos el siguiente código dentro del método `Main`:

```
Dim mes As Integer = 1
Dim nombreMes As String = ""
While mes <> 0
    Console.Write("Ingrese el número de mes: ")
    mes = Console.ReadLine()
    If mes <> 0 Then
        Select Case mes
            Case 1
                nombreMes = "Enero"
            Case 2
                nombreMes = "Febrero"
            Case 3
                nombreMes = "Marzo"
            Case 4
                nombreMes = "Abril"
            Case 5
                nombreMes = "Mayo"
            Case 6
                nombreMes = "Junio"
            Case 7
                nombreMes = "Julio"
            Case 8
                nombreMes = "Agosto"
            Case 9
                nombreMes = "Septiembre"
            Case 10
                nombreMes = "Octubre"
            Case 11
                nombreMes = "Noviembre"
            Case 12
```

```
                nombreMes = "Diciembre"
            End Select
        End If
    End While
```

Al final, se usa el método `WriteLine`, que no habíamos empleado aún, con un parámetro adicional al texto que queremos imprimir. Esta sintaxis implica que el texto `{0}` se reemplazará por el valor del segundo parámetro dentro de la cadena por imprimir. Si tenemos otro valor que queremos imprimir, podemos agregar el parámetro y referenciarlo como `{1}` dentro de la cadena.

Ahora veamos una variante del ejercicio anterior, escribiendo una función que devuelva el nombre del mes que se le pasa como parámetro. Simplemente, debemos crear una función de tipo `String`, con un parámetro por valor y de tipo `Integer`, colocar dentro de ella el `Select Case` que escribimos en el ejercicio anterior y reemplazar el `Select Case` por el llamado a la función. El encabezado de la función puede ser éste:

```
Function ObtenerNombreMes(ByVal mes As Integer) As String
```

No debemos olvidarnos de colocar la instrucción `Return` al final de la función, porque, de no hacerlo, la función devolverá una cadena vacía.

Podemos asignar un valor o una variable a otra, sin hacer la conversión explícita de tipos.

Reemplazando el Select por la llamada a la función, el código del procedimiento Main queda:

```
Sub Main()
    Dim mes As Integer = 1
    Dim nombreMes As String = ""
    While mes <> 0
        Console.WriteLine("Ingrese el número de mes: ")
        mes = Console.ReadLine()
        If mes <> 0 Then
            nombreMes = ObtenerNombreMes(mes)
            Console.WriteLine("El mes ingresado es {0}", nombreMes)
        End If
    End While
    Console.ReadKey()
End Sub
```

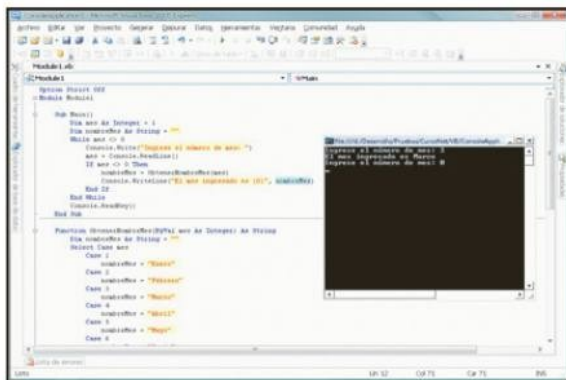


FIGURA 007 | Reemplazando el Select Case por una función, nuestro programa se sigue comportando como debe, pero el código es más claro.

Números fraccionarios

En este caso, veremos cómo crear una clase para representar números fraccionarios. La clase deberá tener dos variables de instancia (privadas) para mantener el valor del numerador y del denominador de la fracción, además de sendas propiedades para hacer públicas las variables de instancia.

La forma más sencilla de agregar una nueva clase es utilizando la plantilla que nos brinda Visual Studio.

La forma más sencilla de agregar una nueva clase es utilizando la plantilla que nos brinda Visual Studio. Para esto, basta con hacer clic con el botón derecho sobre el icono de proyecto en el Explorador de Soluciones, seleccionar la opción Agregar y, luego, Clase. Visual Studio nos pedirá que escribamos el nombre de la clase, y después de aceptar, agregará un nuevo archivo al proyecto, con la definición (aunque vacía por ahora) de la nueva clase:

```
Public Class Fraccion
End Class
```

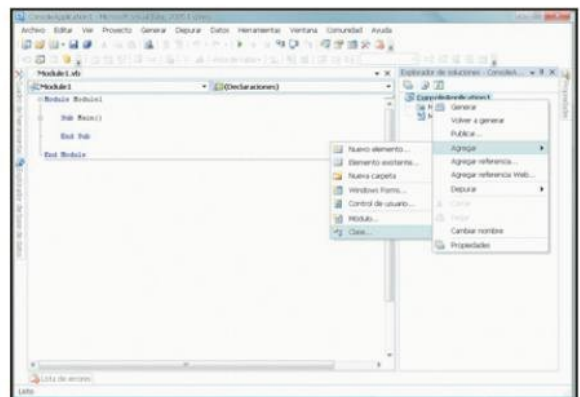


FIGURA 008 | Visual Studio nos brinda una plantilla para agregar una nueva clase.

Si repasamos los conceptos aprendidos sobre creación de clases, recordaremos que la palabra clave fundamental aquí es Class.

A continuación, debemos agregar las variables de instancia para el numerador y el denominador,



además de las propiedades para exponerlas públicamente. En este caso, sólo mostraremos cómo escribir el código para el numerador, y el lector deberá completar el ejercicio agregando el denominador. Hasta ahora, el código de la clase queda de este modo:

```
Public Class Fraccion
    Private _numerador As Integer

    Public Property Numerador() As Integer
        Get
            Return _numerador
        End Get
        Set(ByVal value As Integer)
            _numerador = value
        End Set
    End Property
End Class
```

Ahora bien, tratemos de crear una nueva clase para representar números fraccionarios, como la del ejercicio anterior, y que tenga un constructor que pida al usuario que ingrese en la consola los valores para el numerador y el denominador. Para cumplir con este objetivo, vamos a escribir realmente muy poco código, ya que usaremos la herencia para aprovechar la clase escrita en el Ejercicio 4. Con Visual Studio creamos una nueva clase llamada FraccionConConsola y la hacemos heredar de la clase Fraccion, mediante la palabra clave **Inherits**:

```
Public Class FraccionConConsola
    Inherits Fraccion

End Class
```

Con esto, como vimos antes, estamos heredando las características (propiedades y comportamiento) de la clase Fraccion, es decir que la cla-

Para asociar variables con propiedades podemos nombrarlas igual, colocando un guión al nombre de la variable.

se FraccionConConsola tiene una propiedad llamada Numerador y otra llamada Denominador. Lo que nos queda por hacer, entonces, es crear el constructor como pide el enunciado del ejercicio. Recordemos que el constructor es un método especial llamado New.

Ejercicios para seguir practicando

Ejercicio 1: Modificar la clase Fraccion del último ejercicio para agregar una propiedad de sólo lectura que devuelva el valor real de la fracción (la división real entre el numerador y el denominador).

Ejercicio 2: Modificar la clase Fraccion para validar que el denominador no sea 0 y, así, evitar errores de división por 0 al usar la propiedad definida en el Ejercicio 1. Si el valor que se le asigna es 0, deberá mostrar un mensaje de error en la consola, asignarle 1 al denominador y 0 al numerador. Como ayuda, el setter de la propiedad puede contener todo el código que se desee.

Ejercicio 3: Modificar la clase Fraccion otra vez, agregando un método (Función) llamado Multiplicar, que reciba como parámetro una instancia de la clase Fraccion y devuelva como resultado un nuevo número fraccionario que sea el resultado de multiplicar la instancia actual por la que se recibe como parámetro. Para multiplicar dos fracciones, hay que multiplicar los numeradores y los denominadores (por ejemplo, $\frac{1}{2} * \frac{3}{4}$ es $\frac{3}{8}$).

Ejercicios para seguir practicando

Ejercicio 4: Escribir una clase Empleado con una propiedad llamada Nombre y un método “redefinible” llamado HacerTarea. Escribir dos clases derivadas de Empleado, llamadas Obrero y Gerente. Redefinir el método HacerTarea de la clase Obrero para que muestre en la consola el texto “Trabajar”. Redefinir el método HacerTarea de la clase Gerente para que muestre por pantalla el texto “Mandar”.

Ejercicio 5: Siguiendo con el Ejercicio 4, en el módulo donde está el método Main, escribir un método llamado MostrarTrabajo, que reciba como parámetro una instancia de la clase Empleado e invoque el método HacerTarea del objeto que recibe. Por último, en el método Main, crear una instancia de la clase Obrero y otra de la clase Gerente, e invocar el método MostrarTrabajo con ambas. Se puede consultar el libro *Introducción a la programación* para repasar conceptos de herencia y polimorfismo.

Ejercicio 6: Redefinir el método ToString de la clase Empleado para que muestre el nombre por pantalla. Cabe recordar que este método pertenece a la clase Object, de la que todas las demás clases heredan por defecto (aunque no se haga explícitamente mediante la palabra Inherits). Dentro del método Main, crear una nueva instancia de la clase Obrero llamada obrero1, asignar valor a la propiedad Nombre y escribir la siguiente línea:

```
Console.WriteLine(obrero1)
```

Ejecutar la aplicación y analizar los resultados. ¿Qué se mostró en la consola?

Escribamos el código del constructor para pedir los valores:

```
Sub New()  
    Console.WriteLine("Ingrese el numerador: ")  
    Numerador = Console.ReadLine()  
    Console.WriteLine("Ingrese el denominador: ")  
    Denominador = Console.ReadLine()  
End Sub
```

Asignamos valor a las variables de instancia mediante las propiedades porque las variables de instancia están como privadas, y con ese alcance sólo la clase que las contiene puede acceder a ellas.

Lo que estamos haciendo en el constructor es mostrar un par de carteles al usuario para que ingrese los valores y asignarlos a las variables privadas, pero haciendo uso de las propiedades de la clase Fraccion.

Esto es todo lo que necesitamos hacer para este ejercicio. Mediante este ejemplo, podemos ver algo de la potencia que nos ofrece la herencia. Si no hubiésemos heredado de la clase Fraccion, deberíamos haber escrito otra vez el código de las propiedades y de las variables de instancia.

Es conveniente a esta altura del curso repasar los conceptos desarrollados en el libro *Introducción a la programación*, para reforzar los temas más complejos.



Microsoft Visual C# 2005

C# se está perfilando como “el lenguaje” de desarrollo de Microsoft .NET. Conozcamos el poder que nos ofrece.

Hasta ahora conocimos el entorno de Visual Studio .NET 2005, sus principales características, componentes y modo de funcionamiento. Ahora llega el turno de C# 2005.

El lenguaje C#

C# fue creado por Microsoft con el propósito de ser el mejor lenguaje de programación que exista para escribir aplicaciones destinadas a la plataforma .NET. Combina la facilidad de desarrollo propia de Visual Basic con el poderío del lenguaje C++, un lenguaje con el cual se ha escrito la mayor parte de la historia del software y de los sistemas operativos de todos los tiempos. En líneas generales, podemos decir que es un lenguaje de programación orientado a objetos simple y poderoso. Una muestra de esto es que las aplicaciones desarrolladas en él podrán funcionar tanto en Windows como en dispositivos móviles, ya sea un teléfono celular o una PDA

Características fundamentales

Lo primero que debemos saber es que podremos crear una gran diversidad de programas utilizando este lenguaje: desde aplicaciones de consola, aplicaciones para Windows o aplicaciones Web, hasta software para dispositivos móviles, drivers y librerías para Windows. C# ha evolucionado notablemente desde sus primeras versiones, al incorporar funcionalidades que mejoran y facilitan la escritura de código, la seguridad de tipos y el manejo automático de memoria.

Sintaxis básica del lenguaje

La sintaxis de C# es muy parecida a la de C, C++ y Java. Para el diseño de este lenguaje, Microsoft decidió modificar o añadir únicamente aquellas cosas que en C no tienen una relativa equivalencia. Por ejemplo, un dato para tener en cuenta es que en C# todas las instrucciones y declaraciones deben terminar con “;” (punto y coma), salvo cuando se abra un bloque de código. A diferencia de lo que sucede en Visual Basic, una instrucción que sea muy larga puede ponerse en varias líneas, sin ingresar ningún tipo de signo especial al final de cada una.

El compilador comprende que todo forma parte de la misma instrucción, hasta que encuentra el punto y coma:

```
A = Metodo(argumento1, argumento2, argumento3
, argumento4, argumento5, argumento6
, argumento7, argumento8);
```

Diferencias de sintaxis con Visual Basic

En el siguiente fragmento de código, veremos las diferencias de una misma función escrita en Visual Basic y en C#:

C# es un lenguaje de programación orientado a objetos simple y poderoso.

FUNCIÓN ESCRITA EN VISUAL BASIC

```
Public Function NumeroMayorQueCinco(
    numer as integer) as boolean
    'Comparo si el parámetro numer contiene
    un valor mayor a 5
    if numer > 5 then

        NumeroMayorQueCinco = true

    end if
End Function
```

FUNCIÓN ESCRITA EN C#

```
public bool NumeroMayorQueCinco(int numer)
{
    //Comparo si el parámetro numer contiene
    un valor mayor a 5
    if (numer > 5)
    {
        return true;
    }
    return false;
}
```

En la función escrita en Visual Basic .NET, encontramos dos bloques de código. El primero es el contexto que crea la función, es decir, la línea donde se declara la función del tipo public (**Public Function NumeroMa-**

yorQueCinco), la línea donde termina dicha función (**End Function**). El otro bloque corresponde al contexto If, compuesto por la única línea que hay entre el principio de dicho contexto (**If numer > 5 then**) y la que indica su final (**End If**). Este tipo de bloque en Visual Basic que marca el principio y el fin de la función y el código contenido hace que la legibilidad sea clara y sencilla.

La misma función escrita en C# marca que los bloques están claramente delimitados por llaves, { y }, ambos del mismo modo. Detrás de la línea en donde se declara el método, no está el punto y coma, igual que en la línea del If, lo cual indica que la llave de apertura del bloque correspondiente se podría haber escrito a continuación, y no en la línea siguiente.

Para comentar el código utilizamos //, a diferencia de Visual Basic .NET, donde se usa una comilla simple.

Operadores

C# proporciona un conjunto de operadores, símbolos que especifican las operaciones que deben realizarse en una expresión. Por lo general, en las enumeraciones se permiten las operaciones de tipos integrales, como ==, !=, <, >, <=, >=, binary +, binary -, ^, &, |, ~, ++, — y sizeof().

Veamos algunos ejemplos de los operadores más comunes:

```
Console.WriteLine(6 + 9); // operador de suma
// concatenación de cadenas
Console.WriteLine("2" + "7");
// concatenación de cadenas
Console.WriteLine(2.0 + "9");
// operador de comparación
Console.WriteLine(8 > 12);
// operador de división
```

§ ¿Qué es una DLL?

Una DLL (*Dynamic Link Library*) es un conjunto de funciones y/o clases que pueden ser accedidas y utilizadas por otros programas en tiempo de ejecución. Estas librerías pueden crearse desde C# o desde otros lenguajes.



```
Console.WriteLine(14 / 2);  
// operador de multiplicación  
Console.WriteLine(5 * 6);
```

Éstos son los resultados obtenidos a partir del uso de los operadores:

```
15  
27  
29  
False  
7  
30
```

Variables

Muchos de los lenguajes orientados a objetos proporcionan los tipos agrupándolos de dos formas: los tipos primitivos del lenguaje, como números o cadenas, y el resto de tipos creados

En C# todas las instrucciones y declaraciones deben terminar con “;” (punto y coma).

a partir de clases. C# cuenta con un sistema de tipos unificado, el CTS (*Common Type System*), que proporciona todos los tipos de datos como clases derivadas de la clase de base `System.Object`. El framework .NET divide los tipos en dos grandes grupos: los tipos valor y los tipos referencia. Cuando se declara una variable que es de un tipo valor, se está reservando un espacio de memoria en la pila para que almacene los datos reales que contiene esa variable. Por ejemplo, en la declaración:

```
int num =10;
```

Tabla 2 | Resumen del sistema de tipos

Alias C#	Descripción	Valores aceptados
<code>object</code>	Clase base de todos los tipos del CTS	Cualquier objeto
<code>string</code>	Cadenas de caracteres	Cualquier cadena
<code>sbyte</code>	Byte con signo	Desde -128 hasta 127
<code>byte</code>	Byte sin signo	Desde 0 hasta 255
<code>short</code>	Enteros de 2 bytes con signo	Desde -32.768 hasta 32.767
<code>ushort</code>	Enteros de 2 bytes sin signo	Desde 0 hasta 65.535
<code>int</code>	Enteros de 4 bytes con signo	Desde -2.147.483.648 hasta 2.147.483.647
<code>uint</code>	Enteros de 4 bytes sin signo	Desde 0 hasta 4.294.967.295
<code>long</code>	Enteros de 8 bytes con signo	Desde -9.223.372.036.854.775.808 hasta 9.223.372.036.854.775.807
<code>ulong</code>	Enteros de 8 bytes sin signo	Desde 0 hasta 18.446.744.073.709.551.615
<code>char</code>	Caracteres Unicode de 2 bytes	Desde 0 hasta 65.535
<code>float</code>	Valor de coma flotante de 4 bytes	Desde 1,5E-45 hasta 3,4E+38
<code>double</code>	Valor de coma flotante de 8 bytes	Desde 5E-324 hasta 1,7E+308
<code>bool</code>	Verdadero/falso	true o false
<code>decimal</code>	Valor de coma flotante de 16 bytes (tiene 28-29 dígitos de precisión)	Desde 1E-28 hasta 7,9E+28

Cuando se declara una variable que es de tipo valor, se está reservando un espacio de memoria en la pila para que almacene los datos que contiene.

Cuando queremos declarar una variable de uno de estos tipos en C#, tenemos que colocar el alias que le corresponde; luego, el nombre de la variable y, por último, opcionalmente, le asignamos su valor.

C# distingue entre mayúsculas y minúsculas, lo que se conoce como *case sensitive*. Esto significa que **PERSONA**, **Persona** y **persona** constituyen tres tipos de datos distintos. Un tipo que admite valores de coma flotante admite valores con un número de decimales que no está fijado previamente; es decir, números enteros o con un decimal, o con dos, o con diez. Decimos que la coma es flotante porque no está siempre en la misma posición con respecto al número de decimales (el separador decimal en el código siempre es el punto).

```
double num=10.75;
```

```
double num=10.7508;
```

Las cadenas son una consecución de caracteres —ya sean numéricos, alfabéticos o alfanuméricos— y son definidas de la siguiente manera:

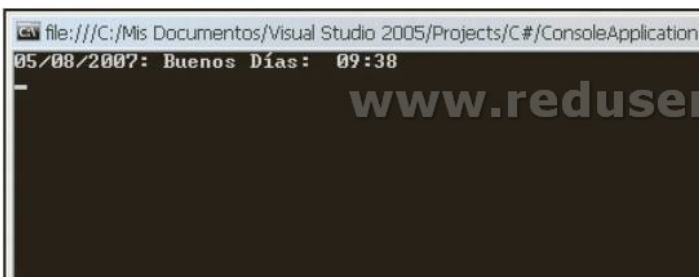


FIGURA 009 | Ejemplo con variables.

```
string palabra1 = "05/08/2007: Buenos";
string palabra2 = " Días:";
string palabra3 = " 09:38";
string saludo = (palabra1 + palabra2
+ palabra3);

Console.WriteLine(saludo);
Console.ReadKey();
```

Arrays

Un array es una estructura de datos que contiene variables del mismo tipo. Se declara de la siguiente manera:

```
TipoDeDato [] NombreDelArray;
```

El siguiente ejemplo muestra cómo declarar diferentes tipos de arrays:

```
static void Main(string[] args)
{
    // Declara un array de una dimension
    con lugar para 3 elementos
    int [] arrayDeNumeros = new int[3];
    arrayDeNumeros[0] = 6;
    arrayDeNumeros[1] = 35;
    arrayDeNumeros[2] = 37;

    // El mismo array, se puede llenar durante
    la declaracion del mismo
    int[] array2DeNumeros = { 6, 35, 37 };

    // o de esta manera
    int[] array3DeNumeros = new int[] { 6,
    35, 37 };

    // mostrar los elementos del array
    for (int i = 0; i < arrayDeNumeros.Length;
    i++) {
        Console.WriteLine(i);
    }

    // Es posible utilizar foreach para iterar
```



```
// sobre los elementos de un array
foreach (int a in array2DeNumeros) {
    Console.WriteLine ( a );
}

Console.Read ();
}
```

C# distingue entre mayúsculas y minúsculas, lo que se conoce como *case sensitive*.

Sentencias de control

En C# disponemos de dos tipos de sentencias de control: de bifurcación condicional y de iteración. La primera consiste en evaluar una expresión ejecutando un bloque de código si la expresión es verdadera; de lo contrario, un segundo bloque de código puede ser ejecutado. Tenemos, entonces, dos elementos para manejar la bifurcación condicional en C#: **if** y **switch**.

```
{
// bloque de código que se ejecuta cuando a es
MAYOR que b
}
else
{
// bloque para a MENOR que b
}
```

Es posible utilizar los siguientes operadores:

>, <, >=, <=, != y ==.

El fragmento de código que se muestra a continuación ejemplifica cómo hacerlo:

Sentencia if

```
// utilizacion de if
if (a > b)
```

```
if (a > b)
{
// si a es mayor que b
}
```

Construcción else if

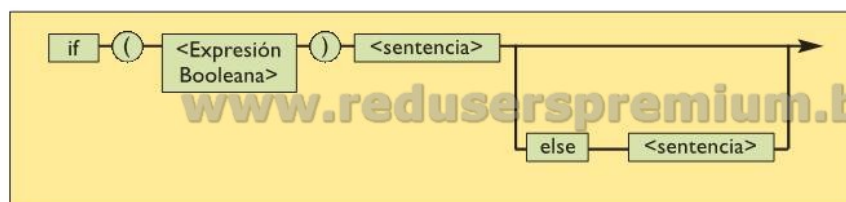
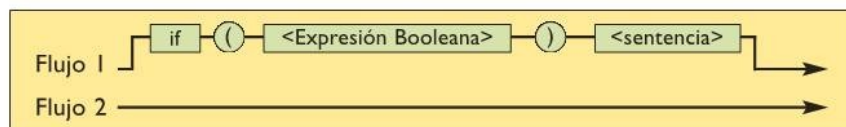


FIGURA 010 | Podemos notar que else if es una construcción que se utiliza para realizar múltiples evaluaciones.


```

else if (a < b)
{
// si a es menor que b
}
else if (a<=b)
{
// si a es menor o igual a b
}
else if (a >= b)
{
// si a es mayor o igual a b
}
else if (a == b)
{
// si a es igual a b
}
else if (a != b)
{
// a es distinto que b
}
else
{
// alguna opcion no contemplada :)
}

```

Sentencia switch

En ocasiones, necesitaremos comparar una variable con una cantidad de valores posibles, a fin de determinar el rumbo por seguir. Para hacerlo, utilizamos el **switch**:

```

switch (a)
{
case 1:

```

```

// codigo si a == 1
break;
case 2:
// codigo si a == 2
break;
case 3: case 4: case 5:
// codigo si a == 3 o 4 o 5
break;
default:
// codigo si a tiene un valor distinto a
1,2,3,4 o 5.
break;
}

```

Con esto podemos decir que **switch** es comparable con la sentencia **Select Case**, vista anteriormente en Visual Basic .NET.

La bifurcación ocurre en el switch sin evaluar todas las opciones, mientras que en un bloque **if-else if** todas las opciones son evaluadas. Por este motivo, si sabemos que la variable que vamos a evaluar tomará sólo uno de los valores posibles (en forma excluyente), y que la cantidad de valores por evaluar es más grande que tres (regla de dedo), utilizar switch en vez de bloques if-else if es más eficiente en términos de procesamiento.

Sentencia de bucle

En algunas oportunidades, necesitaremos que determinada porción de código de un programa se repita una cierta cantidad de veces o

SI LA VARIABLE QUE VAMOS A EVALUAR TOMARÁ SÓLO UNO DE LOS VALORES POSIBLES (EN FORMA EXCLUYENTE), Y LA CANTIDAD DE VALORES POR EVALUAR ES MÁS GRANDE QUE TRES, ES ACONSEJABLE UTILIZAR SWITCH EN VEZ DE BLOQUES IF ELSE.



hasta que ocurra una condición determinada. C# cuenta con varias sentencias para que esto se cumpla.

La sentencia while

Esta sentencia, que significa **mientras**, indica que una sentencia o grupo de sentencias se repetirá en tanto una condición asociada a ella sea verdadera.

A continuación, veamos un ejemplo de código en el cual, mientras no se ingrese un texto determinado, el programa no terminará:

```
string Palabra = "";  
while (Palabra != "terminar")  
{  
    Console.WriteLine("Ingrese una palabra:");  
    Palabra = Console.In.ReadLine();  
}  
Console.WriteLine("Palabra correcta para finalizar");
```

Analizando el código, vemos en la primera línea que declaramos una variable del tipo cadena, y la inicializamos. En esta variable se almacenará el texto que ingresemos por teclado cuando probemos el sistema.

Luego, iniciamos la sentencia **while**, y evaluamos la expresión encerrada entre paréntesis en dicha sentencia, que en este caso sería: "mientras la variable **Palabra** sea distinta de **"terminar"**... Si al evaluar la expresión obtenemos como resultado **Verdadero**, entonces entra en el cuerpo del bucle **while**; de otra manera, saltaremos a la línea siguiente después del cuerpo.

Dentro del **while** enviamos a la salida de la consola el texto **"Ingrese una palabra:"**, y el programa vuelve a capturar en la variable **Palabra** el texto que se ingresa en el teclado.

Así, cada vez que ingresemos un texto en la salida de la consola, éste será analizado, evaluando la expresión **Palabra != salir**. Cuando la

condición del texto que hayamos ingresado dé como resultado **Falso**, el sistema finalizará.

La sentencia do

Con la sentencia **while** vimos que el bloque de código que se debe repetir no será ejecutado nunca si la expresión es falsa la primera vez. En algunas ocasiones, seguramente necesitaremos que se ejecute dicho código al menos una vez, para luego hacer una evaluación de la expresión.

Para estos casos existe la sentencia **do**.

```
String Palabra;  
do  
{  
    Console.Write("Ingrese la palabra\n");  
    Palabra = Console.In.ReadLine();  
} while (Palabra != "terminar");  
Console.Write ("Es la palabra correcta");
```

La sentencia while, que significa mientras, indica que una sentencia o grupo de sentencias se repetirá mientras una condición asociada a ella sea verdadera.

A diferencia de la instrucción **while**, un bucle **do-while** se ejecuta una vez antes de que se evalúe la expresión condicional.

En cualquier punto dentro del bloque **do-while**, se puede salir del bucle utilizando la instrucción **break**. Se puede pasar directamente a la instrucción de evaluación de la expresión **while** utilizando la instrucción **continue**; si la expresión se evalúa como **true**, la ejecución continúa en la primera instrucción del bucle. Si se evalúa como **false**, la ejecución continúa en la primera instrucción detrás del bucle **do-while**.

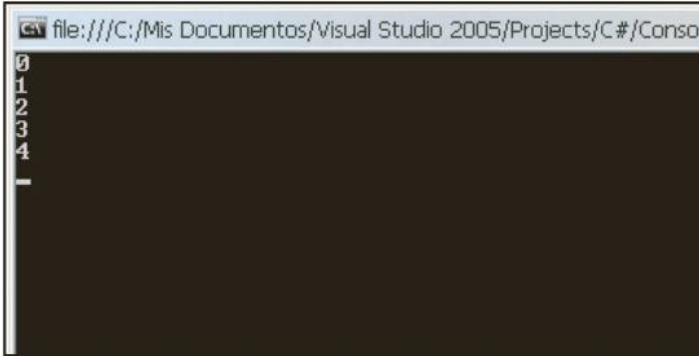


FIGURA 011 | El resultado de la sentencia for.

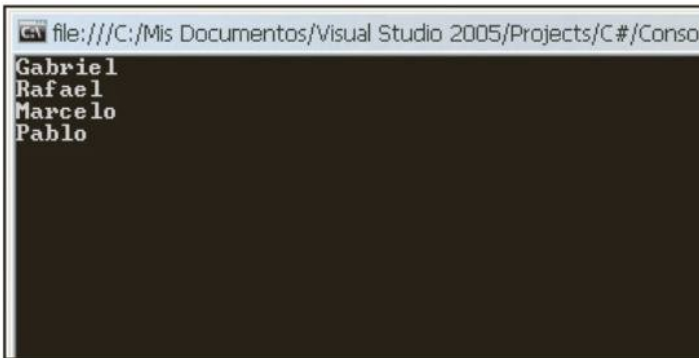


FIGURA 012 | El resultado de la sentencia foreach.

La sentencia for

El bucle **for** ejecuta una instrucción o un bloque de instrucciones repetidamente hasta que una determinada expresión se evalúa como **false**. El bucle **for** es útil para recorrer matrices

! La condición de terminación del bucle

La expresión booleana del bucle “mientras” (**while**) debería ser verdadera en ciertas circunstancias y luego, en algún momento, tornarse falsa. Si la expresión **NUNCA** fuese verdadera, jamás se ingresaría en el cuerpo del bucle. Si la expresión **SIEMPRE** fuese verdadera, nuestro programa jamás se terminaría, ya que nunca saldríamos del bucle.

en iteración y para procesar de manera secuencial. En el ejemplo siguiente el valor de **i** se imprimirá en pantalla mientras se incrementa de a uno:

```
{
    for (int i = 0; i < 5; i++)
        Console.WriteLine(i);
    Console.ReadKey();
}
```

Analizando el código, vemos que se evalúa el valor inicial de la variable **i**. A continuación, mientras el valor de **i** es menor o igual que 5, la condición se evalúa como **true**, se ejecuta la instrucción **Console.WriteLine** y se vuelve a evaluar **i**. Cuando **i** es mayor que 5, la condición se convierte en **false**, y el control se transfiere fuera del bucle.

La sentencia foreach

Ciertas colecciones de objetos utilizan una forma de estructurar la colección que permite la extracción de cada uno de los elementos de manera controlada desde la clase contenedora. Esto permite mantener el encapsulamiento dentro de la clase para evitar que el cliente conozca los detalles internos a la implementación. Por ejemplo, el siguiente fragmento de código consulta un array utilizando **foreach**:

```
string[] nombres = { "Gabriel", "Rafael",
    "Marcelo", "Pablo" };

foreach (string s in nombres) {
    Console.WriteLine ( "{0}", s );
}
```

Es posible construir colecciones que puedan ser recorridas con la sentencia **foreach**. Éstas podrían entregar la lista ordenada alfabéticamente, por ejemplo.

USERS



CURSOS.REDUSERS.COM

CURSOS INTENSIVOS



Los temas más importantes del universo de la tecnología desarrollados con la mayor profundidad y con un despliegue visual de alto impacto: Explicaciones teóricas, procedimientos paso a paso, videotutoriales, infografías y muchos recursos mas.

Brinda las habilidades necesarias para planificar, instalar y administrar redes de computadoras de forma profesional. Basada principalmente en tecnologías Cisco, es una obra actual, que busca cubrir la necesidad creciente de formar profesionales.

- ▶ 25 Fascículos
- ▶ 600 Páginas
- ▶ 3 CDs / 1 Libro



- ▶ 25 Fascículos
- ▶ 600 Páginas
- ▶ 4 CDs

Curso para dominar las principales herramientas del paquete Adobe CS3 y conocer los mejores secretos para diseñar de manera profesional. Ideal para quienes se desempeñan en diseño, publicidad, productos gráficos o sitios web.

Obra teórica y práctica que brinda las habilidades necesarias para convertirse en un profesional en composición, animación y VFX (efectos especiales).

- ▶ 25 Fascículos
- ▶ 600 Páginas
- ▶ 2 CDs / 1 DVD / 1 Libro



- ▶ 26 Fascículos
- ▶ 600 Páginas
- ▶ 2 DVDs / 2 Libros

Obra ideal para ingresar en el apasionante universo del diseño web y utilizar Internet para una profesión rentable. Elaborada por los máximos referentes en el área, con infografías y explicaciones muy didácticas.

www.reduserspremium.blogspot.com.ar

Llegamos a todo el mundo con OCA * y DHL **

✉ usershop@redusers.com ☎ +54 (011) 4110-8700

usershop.redusers.com.ar

** Válido en todo el mundo excepto Argentina. * Sólo válido para la Republica Argentina

Argentina: \$5,90 (recargo al interior\$0,20) México: \$30

USERS

Microsoft®

Curso teórico y práctico de programación

Desarrollador .net

Con toda la potencia
de **Visual Basic .NET** y **C#**

La mejor forma de aprender
a programar desde cero



Basado en el programa
Desarrollador Cinco Estrellas
de Microsoft

3

Visual Basic

Funciones propias y primeros
ejercicios prácticos

C# desde cero

Sintaxis básica, sintaxis de clase
y funciones propias



ISBN 978-987-1347-43-8



00003



9 789871 347438

Incluye CD-ROM #3

