Unidad

2

DIPLOMATURA EN PROGRAMACION .NET

Capítulo 2

Visibilidad de Variables y Expresiones

En Este Capítulo

- Las palabras claves this y Me
- Variables y visibilidad
- Los métodos estáticos y su visibilidad
- Declaración de constantes
- Operadores
- Operadores unarios y ternarios
- Promoción de expresiones en las conversiones de tipo
- Concatenación de Strings
- Conversión automática de tipos primitivos en objetos: autoboxing
- Convirtiendo Strings a valores numéricos

Las palabras claves `this` y `Me`

Estas palabras clave tienen como finalidad almacenar la referencia del objeto que se encuentra en ese momento en ejecución y se utiliza para encontrar los elementos que pertenecen a éste. Cada vez que se invoca a un método que pertenece a un objeto, este valor se almacena en el stack cuando se reserva espacio en memoria para el mismo. Es por esta referencia que se encuentran las variables de instancia que pertenecen a cada objeto.

Sólo los elementos de un objeto, métodos y atributos, poseen asociada una referencia del tipo `this` ó `Me` (C# o VB), por lo tanto se la puede utilizar para resolver ambigüedades.

Normalmente, dentro del cuerpo de un método de un objeto se puede referir directamente a las variables miembros del objeto. Sin embargo, algunas veces es necesario evitar algún tipo de ambigüedad acerca del nombre de un atributo y uno o más de los argumentos del método que tengan el mismo nombre.

Por ejemplo, si una clase tiene declaradas variables de instancia que tienen el mismo nombre que los argumentos de un método, utilizando `this` ó `Me` (C# o VB), se resuelve la ambigüedad. Esto sucede porque los métodos poseen parámetros y variables locales que se alojan en el stack, por lo tanto, nunca podrán tener estos elementos un `this` ó `Me` (C# o VB) asociado.

Ejemplo

C#

```
namespace ambiguedad
{
    public class Ejemplo
    {
        private int atrib1;
        private char atrib2;

        public Ejemplo(int a) { atrib1 = a; }
        public Ejemplo(char b) { atrib2 = b; }
        public Ejemplo(char atrib2, int atrib1)
        {
            this.atrib1 = atrib1;
            this.atrib2 = atrib2;
        }

         public void metodo()
        {
            //[sentencias;]
            atrib2 = 'a';
        }
         public int Atrib1
         {
             get
             {
                 return atrib1;
             }
```

```
            set
            {
                atrib1 = value;
            }
        }
        public char Atrib2
        {
            get
            {
                return atrib2;
            }
            set
            {
                atrib2 = value;
            }
        }
    }
}
```

VB

```
Public Class Ejemplo
    Private _atrib1 As Integer
    Private _atrib2 As Char

    Public Sub New(a As Integer)
        _atrib1 = a
    End Sub

    Public Sub New(b As Char)
        _atrib2 = b
    End Sub
    Public Sub New(_atrib2 As Char, _atrib1 As Integer)
        Me._atrib1 = _atrib1
        Me._atrib2 = _atrib2
    End Sub

    Public Property Atrib1() As Integer
        Get
            Return _atrib1
        End Get
        Set(ByVal value As Integer)
            _atrib1 = value
        End Set
    End Property

    Public Property Atrib12() As String
        Get
            Return _atrib2
        End Get
        Set(ByVal value As String)
            _atrib2 = value
        End Set
    End Property
End Class
```

En el ejemplo anterior se muestra la resolución de visibilidad dentro del constructor de la clase. Si no se utilizará la palabra clave `this` ó `Me` (C# o VB), y como respecto de la visibilidad siempre tiene prioridad los elementos de visibilidad local, no se podrían diferenciar los parámetros (locales) de los atributos del objeto.

Variables y visibilidad

Las variables locales son:

- Definidas dentro de un método (se las llama locales, automáticas, temporarias o de stack)
- Se crean cuando se ejecuta un método y se destruyen cuando este termina
- Pueden almacenar valores o referencias
- Las variables que almacenan referencias siempre se deben inicializar con un operador **new** antes de utilizarse

La duda que se presenta siempre a todo programador es como "saben" dos objetos del mismo tipo cuando se están ejecutando cuáles son sus variables de instancia y cuáles no le corresponden. En realidad, lo que sucede por detrás, es que el CLR asigna a todo método que fue definido en una clase un primer parámetro oculto (`this` ó `Me` -C# o VB-) que tiene almacenada la referencia del objeto que se esta ejecutando actualmente. ¿Cómo puede saber eso? La respuesta es simple: la variable de referencia que se utiliza con la notación de punto almacena dicha referencia a memoria, por lo tanto, lo único que hace es incluirla como primer parámetro por defecto de todo método a ejecutarse de un objeto.

Ejemplo

C#

```
namespace visibilidades
{
    class Visibilidad
    {
        private int i = 1;

        public void primerMetodo()
        {
            int i = 4, j = 5;
            this.i = i + j;
            segundoMetodo(7);
        }

        public void segundoMetodo(int i)
        {
            int j = 8;
            this.i = i + j;
        }

        static void Main(string[] args)
        {
            Visibilidad visibilidad1 = new Visibilidad();
            Visibilidad visibilidad2 = new Visibilidad();
            visibilidad1.primerMetodo();
```

```
            visibilidad2.primerMetodo();
        }
    }
}
```

VB

```
Public Class Visibilidad
    Private i As Integer = 1

    Public Sub primerMetodo()
        Dim i As Integer = 4
        Dim j As Integer = 5
        Me.i = i + j
        segundoMetodo(7)
    End Sub
    Public Sub segundoMetodo(i As Integer)
        Dim j As Integer = 8
        Me.i = i + j
    End Sub
End Class
```

Los siguientes gráficos muestran la evolución del stack y el lugar al que se apunta en el heap para cada secuencia de llamados.

Ejemplo

C#

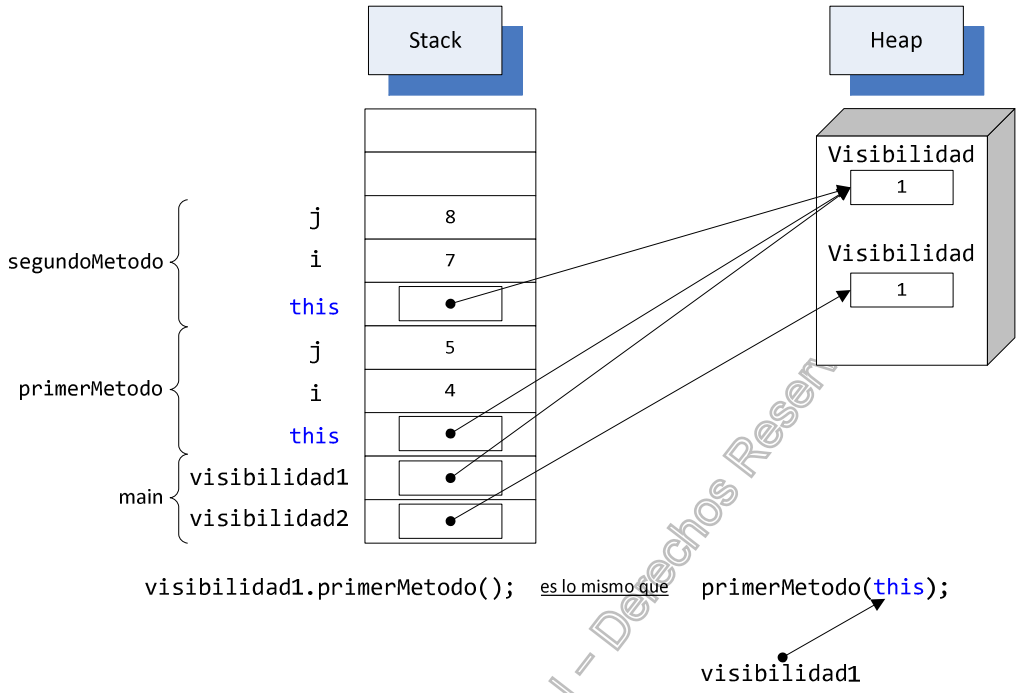
```
            visibilidad1.primerMetodo();
```

VB

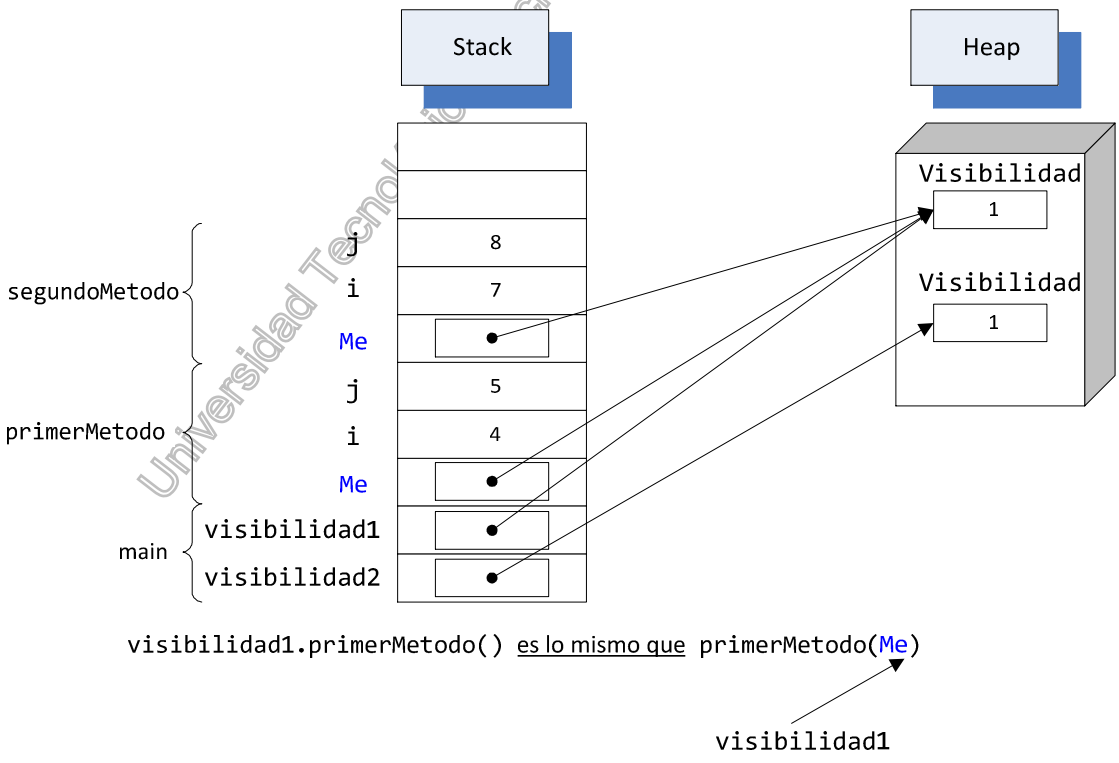
```
            visibilidad1.primerMetodo()
```

Genera

C#



VB



Mientras que

Ejemplo

C#

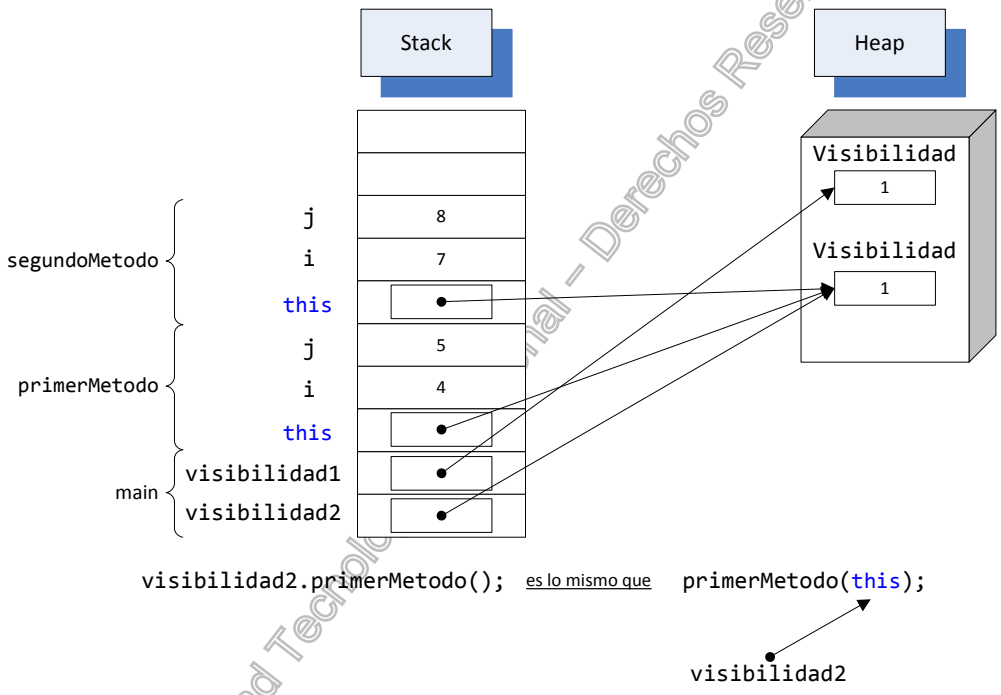
```
visibilidad1.primerMetodo();
```

VB

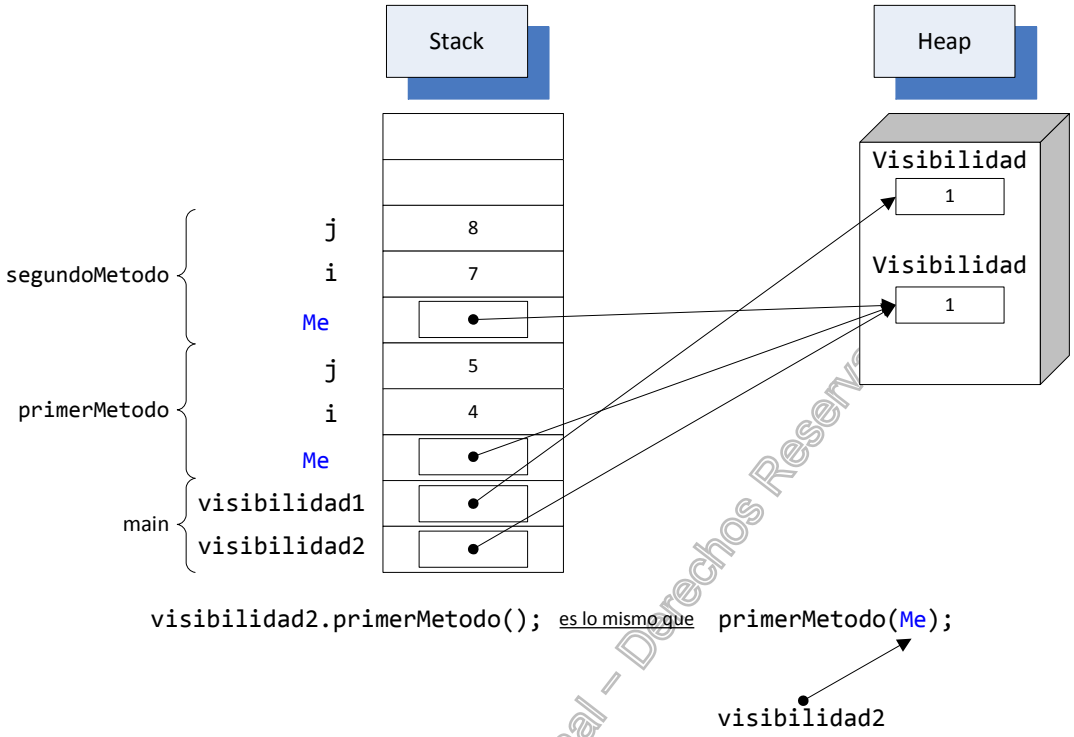
```
visibilidad1.primerMetodo()
```

Genera

C#



VB



Los métodos estáticos y su visibilidad

Este tipo de métodos se diferencian de cualquier otro tipo declarado en una clase. Como se demostró anteriormente los métodos que pertenecen a una clase pasan encubiertamente una referencia `this` ó `Me` (C# o VB) como primer parámetro en cualquier método perteneciente a un objeto y la razón es que el mismo encuentre las variables de instancia que pertenecen a "este" objeto para poder operar con ellas. Esto es cierto salvo para los métodos estáticos

> *Los métodos estáticos se diferencian del resto porque no tienen una referencia* `this` ó `Me` (C# o VB) *asociada*

La principal consecuencia es que no pueden acceder a ningún elemento de un objeto en tiempo de ejecución porque al no tener forma de encontrarlos mediante una referencia, "no sabe que existen".

Es claro que si se le pasa como parámetro una referencia `this` ó `Me` (C# o VB) a un método estático, este podrá acceder a los elementos del objeto limitado siempre por la visibilidad que los mismos posean.

Respecto justamente de la visibilidad, ¿cómo es afectada la visibilidad en un método estático? Como se mencionó en el módulo anterior, las visibilidades están determinadas por los bloques en los que se realizan las declaraciones. Al estar los métodos estáticos definidos dentro de clases, tienen visibilidad dentro de las mismas y para poder accederlos hay que resolver ese alcance.

La forma de resolver visibilidad para acceder un método estático es utilizar el nombre de la clase seguido de la notación de punto. Esto sumado al hecho que no tiene un `this` ó `Me` (C# o VB) asociado permite realizar la siguiente afirmación

> *Todo método estático puede ser accedido mediante el nombre de la clase que lo contiene, seguido de la notación de punto y el nombre del mismo, sin la necesidad de crear un objeto para invocarlo.*

Este hecho tiene importancia a nivel de diseño de clases como se verá posteriormente. Sin embargo ya se ha visto una utilidad de esta característica, el método M**ain**. .Net no puede iniciar un programa declarando un objeto de una clase para acceder al método M**ain** que esta posea. Al ser éste estático, no tiene necesidad de crear ningún objeto, sólo saber la clase en la que se encuentra para resolver la visibilidad.

Nota: En VB no hay necesidad de crear un método estático Main porque se pueden utilizar módulos. Todo procedimiento declarado en un módulo tiene visibilidad global sin necesidad de ser estático. Sin embargo, VB puede comenzar un programa desde un método Main declarado dentro de una clase, a pesar que este no sea su formato por defecto en las plantillas de creación de proyectos que ofrece el Visual Studio.

Declaración de constantes

Para crear un atributo miembro constante en .Net se debe utilizar la palabra clave `const` ó `Const` (C# o VB) en la declaración. La siguiente declaración define una constante llamada AVOGADRO cuyo valor es el número de Avogadro (6.023×10^{23}) y no puede ser cambiado:

Ejemplo

C#

```
namespace constantes
{
    class Avoga
    {
        public const double AVOGADRO = 6.023e23;
    }
}
```

VB

```
Public Class Avoga
    Public Const AVOGADRO As Double = 6.0229999999999998E+23
End Class
```

Por convención, los nombres de los valores constantes se escriben completamente en mayúsculas. Si un programa intenta cambiar una constante, el compilador muestra un mensaje de error similar al siguiente, y no compila el programa.

The left-hand side of an assignment must be a variable, property or indexer

Operadores

Los operadores realizan operaciones entre uno, dos o tres operandos.

Los operadores que requieren un operador se llaman operadores unarios. Por ejemplo, **++** es un operador *unario* que incrementa el valor su operando en uno.

Los operadores que requieren dos operandos se llaman operadores *binarios*. El operador **=** es un operador binario que asigna un valor del operando derecho al operando izquierdo.

Existe un único operador ternario que trabaja con tres operandos y funciona como un condicional y dependiendo de cómo evalúa la condición *devuelve un valor u otro*. Este operador es **?:** ó **IIf** (C# o VB).

Los operadores unarios pueden utilizar la notación de prefijo (C# o VB) o de sufijo (sólo en C#) - también conocidas como pre y post, por ejemplo, un pre incremento de la variable i es ++i, mientras que un post incremento es i++). La notación de prefijo significa que el operador aparece antes de su operando, por ejemplo,

operador operando

La notación de sufijo significa que el operador aparece después de su operando:

operando operador

Todos los operadores binarios tienen la misma notación, es decir aparecen entre los dos operandos:

op1 operador op2

Además de realizar una operación también devuelve un valor. El valor y su tipo dependen del tipo del operador y del tipo de sus operandos. Por ejemplo, los operadores aritméticos (realizan las operaciones de aritmética básica como la suma o la resta) devuelven números, el resultado típico de las operaciones aritméticas.

El tipo de datos devuelto por los operadores aritméticos depende del tipo de sus operandos: si se suma dos enteros, se obtiene un entero. Se dice que una operación evalúa su resultado.

Es muy útil dividir los operadores en las siguientes categorías: aritméticos, relacionales y condicionales, lógicos, de desplazamiento y de asignación.

Operadores aritméticos

Tanto C# como VB soportan varios operadores aritméticos, incluyendo

Operador	Descripción
+	suma

-	resta
*	multiplicación
/	división
/ ó \	División entera
% ó Mod()	módulo

en todos los números enteros y de coma flotante. Por ejemplo, se puede utilizar este código para sumar dos números:

```
sumarEsto + aEsto
```

O este código para calcular el resto de una división:

```
dividirEsto % porEsto
```

Esta tabla resume todas las operaciones aritméticas binarias:

Descripción	VB	C#	Uso	Explicación
Suma	**+**	**+**	op1 + op2	Suma op1 y op2
Resta	-	**-**	p1 - op2	Resta op2 de op1
Multiplicación	*****	*****	op1 * op2	Multiplica op1 y op2
División	**/**	**/**	op1 / op2	Divide op1 por op2
División entera	****	**/**	op1 \ op2 ó op1 / op2	Obtiene el valor entero de dividir op1 por op2
División por módulo (retorna sólo el resto)	**Mod** (también trabaja con punto flotante)	**%**	op1 % op2	Obtiene el resto de dividir op1 por op2
Exponenciación	**^**	n/a	op1 ^ op2	op1 elevado a la op2

Los operadores + y - tienen versiones unarias que seleccionan el signo del operando:

Operador	Uso	Descripción
+	+op	Indica un valor positivo
-	- op	Niega el operando

Además, existen dos operadores de pre y post incremento y decremento aritméticos en C# (los operadores de post no tienen equivalente en VB), ++ que incrementa en uno su operando, y -- que decrementa en uno el valor de su operando.

Operador	Uso	Descripción
++	op ++	Incrementa op en 1; evalúa el valor antes de incrementar
++	++ op	Incrementa op en 1; evalúa el valor después de incrementar
--	op --	Decrementa op en 1; evalúa el valor antes de decrementar

--	-- op	Decrementa op en 1; evalúa el valor después de decrementar

Ejemplo

C#

```
namespace operadores
{
    class OperadoresUnarios
    {
        public void Resultados()
        {
              int a = 4, b = 5, c = 6, d = 0;
              d = ++a - b-- - c; // d = 5 - 5 - 6
              Console.WriteLine("Valor de d: " + d);
        Console.WriteLine("Valor de a: " + a);
        Console.WriteLine("Valor de b: " + b);
        Console.WriteLine("-----------");
        a = 4; b = 5; c = 6; d = 0;
              d = a++ - b-- - c; // d = 4 - 5 - 6
        Console.WriteLine("Valor de d: " + d);
        Console.WriteLine("Valor de a: " + a);
        Console.WriteLine("Valor de b: " + b);
        }
    }
}
```

VB

```
Public Class OperadoresUnarios
    Public Sub Resultados()
        Dim a As Integer = 4
        Dim b As Integer = 5
        Dim c As Integer = 6
        Dim d As Integer = 0

        d = ++a - b - --c ' d = 5 - 5 - 6
        Console.WriteLine("Valor de d: " + d.ToString())
        Console.WriteLine("Valor de a: " + a.ToString())
        Console.WriteLine("Valor de b: " + b.ToString())
        Console.WriteLine("-----------")

    End Sub
End Class
```


Operadores de asignación

Se puede utilizar el operador de asignación **=**, para asignar un valor a otro.

Además del operador de asignación básico, se proporcionan varios operadores de asignación que permiten realizar operaciones aritméticas, lógicas o de bits y una operación de asignación al mismo tiempo.

Específicamente, suponiendo que se quiere añadir un número a una variable y asignar el resultado dentro de ella misma, se puede hacer

```
i = i + 2;
```

Se puede lograr el mismo resultado en la sentencia utilizando el operador **+=**.

```
i += 2;
```

Las dos líneas de código anteriores son equivalentes.

La siguiente tabla lista los operadores de asignación y sus equivalentes:

Descripción	VB	C#	Uso	Explicación
Asignación	=	=	op1 = op2	Asigna op2 a op1
Suma	+=	+=	op1 += op2	op1 = op1 + op2
Resta	-=	-=	op1 -= op2	op1 = op1 - op2
Multiplicación	*=	*=	op1 *= op2	op1 = op1 * op2
División	/=	/=	op1 /= op2	op1 = op1 / op2
División entera	\=	\=	op1 \= op2	op1 = op1 \ op2
Concatenación de cadenas	&=	+=	op1 &= op2 ó op1 += op2	op1 = op1 & op2 ó op1 = op1 + op2
División por módulo (retorna sólo el resto)	n/a	+=	op1 += op2	op1 = op1 + op2
Desplazamiento a izquierda	<<=	<<=	op1 <<= op2	op1 = op1 << op2
Desplazamiento a derecha	>>=	>>=	op1 >>= op2	op1 = op1 >> op2
AND a nivel de bits	n/a	&=	op1 &= op2	op1 = op1 | op2
OR exclusivo	n/a	^=	op1 ^= op2	op1 = op1 ^ op2
OR inclusivo	n/a	|=	op1 |= op2	op1 = op1 | op2
Operador de nulificables	n/a	??	y = x ?? -1	y = x a menos que x sea `null`, en cuyo caso es -1

Operadores relacionales

Los valores relacionales comparan dos valores y determinan la relación entre ellos. Por ejemplo, **!=** devuelve `true` o `True` (C# ó VB) si los dos operandos son distintos.

La siguiente tabla resume los operadores relacionales:

Descripción	VB	C#	Uso	Devuelve true si
Menor	>	>	op1 > op2	op1 es mayor que op2
Menor o igual	>=	>=	op1 >= op2	op1 es mayor o igual que op2
Mayor	<	<	op1 < op2	op1 es menor que op2
Mayor igual	<=	<=	op1 <= op2	op1 es menor o igual que op2
Igual	==	==	op1 == op2	op1 y op2 son iguales
Distinto	<>	!=	op1 != op2	op1 y op2 son distintos

Operadores lógicos o condicionales

Se utilizan para construir expresiones de decisión complejas. Uno de estos operadores es **&&** ó **AndAlso** (C# ó VB) que realiza la operación **Y** lógico o booleano. Por ejemplo, se puede utilizar dos operadores relacionales diferentes y evaluar la totalidad de la expresión uniéndola con **&&** ó **AndAlso** (C# ó VB) para determinar si ambas relaciones son ciertas. La siguiente línea de código utiliza esta técnica para determinar si un valor esta entre dos límites, esto es, para determinar si el índice es mayor que 0 o menor que NUM_ENTRADAS (que se ha definido previamente como un valor constante):

C#

```
0 < index && index < NUM_ENTRADAS
```

VB

```
0 < index AndAlso index < NUM_ENTRADAS
```

Se puede observar que en algunas situaciones, el segundo operando de un operador relacional no será evaluado. Consideremos esta sentencia:

C#

```
((contador > NUM_ ENTRADAS) && (contador!= 1))
```

VB

```
((contador > NUM_ ENTRADAS) AndAlso (contador!= 1))
```

Si **contador** es menor que **NUM_ ENTRADAS** en la parte izquierda del operando **&&** ó **AndAlso** (C# ó VB), la parte derecha no se evalúa. El operador **&&** ó **AndAlso** (C# ó VB) sólo devuelve `true` o `True` si los dos operandos son verdaderos. Por eso, en esta situación se puede determinar el valor de **&&** ó **AndAlso** (C# ó VB) sin evaluar el operador de la derecha y en un caso como este, se ignora el operando de la derecha. Así no se verificará nunca si contador!= 1.

Los operadores condicionales en C# son:

Operador	Uso	Devuelve true si
&&	op1 && op2	op1 y op2 son verdaderos

||	op1 || op2	uno de los dos es verdadero
!	! op	op es falso

El operador **&** se puede utilizar como un sinónimo de && si ambos operandos son bits. Similarmente, | es un sinónimo de || si ambos operandos son bits. En definitiva, estos operadores están diseñados para actuar como sus contrapartidas binarias a nivel de datos regulares, pero en lugar de evaluar todo el operando, lo hace bit a bit entre cada uno de los operandos. Para entender mejor lo afirmado, se realiza la siguiente clasificación.

Ejemplo

C#

```
namespace and
{
    class OperadorAnd
    {
        static void Main(string[] args)
        {
            OperadorAnd op = new OperadorAnd();
            int a = 10, b = 8, c = 6;
            bool verificacion;
            double[] vecPesos = { 3.3999999999999999, 4.5, 8.8000000000000007,
9.9000000000000004, 1.1000000000000001, 1.2, 1.6000000000000001,
0.80000000000000004, 0.98999999999999999, 5.5, 6.7000000000000002 };
            int pos;

            verificacion = a > b && b > c;  // True.
            verificacion = b > a && b > c;  // False. La segunda expresión no se
                                               // evalúa nunca.
            verificacion = a > b && c > b;  // False. La segunda expresión se
                                               // evalúa.

            pos = op.EncontrarValor(vecPesos, 9.9000000000000004);
            Console.WriteLine(pos.ToString());
            Console.ReadKey();
        }

        public int EncontrarValor(double[] vector, double valorBuscado)
        {
            int i = 0;
            while (i <= vector.Length && vector[i] != valorBuscado)
            {
                i += 1;
            }
            return i;
        }
    }
}
```

VB

```
Module Module1

    Sub Main()
        Dim a As Integer = 10
        Dim b As Integer = 8
        Dim c As Integer = 6
        Dim verificacion As Boolean
        Dim vecPesos() As Double = New Double(10) {3.3999999999999999, 4.5,
8.8000000000000007, 9.9000000000000004, 1.1000000000000001, 1.2, 1.6000000000000001,
0.80000000000000004, 0.98999999999999999, 5.5, 6.7000000000000002}
        Dim pos As Integer

        verificacion = a > b AndAlso b > c    ' True.
        verificacion = b > a AndAlso b > c    ' False. La segunda expresión no se
                                                ' evalúa nunca.
        verificacion = a > b AndAlso c > b    ' False. La segunda expresión se
                                                ' evalúa.

        pos = EncontrarValor(vecPesos, 9.9000000000000004)
        Console.WriteLine(pos.ToString())
        Console.ReadKey()
    End Sub

    Public Function EncontrarValor(ByVal vector() As Double, _
                          ByVal valorBuscado As Double) As Integer
        Dim i As Integer = 0
        While i <= UBound(vector) AndAlso vector(i) <> valorBuscado
            ' Si I es mayor que UBound(Arr), SearchValue no se verifica.
            i += 1
        End While
        If i >= UBound(vector) Then i = -1
        Return i
    End Function
End Module
```


Operadores lógicos a nivel de bit entre operandos

Descripción	VB	C#
Negación a nivel de bits	Not	~
OR a nivel de bits	Or	|
AND a nivel de bits	And	&
XOR a nivel de bits	Xor	^

Operadores lógicos a nivel de datos

Descripción	VB	C#
Conjunción	AndAlso	&&
Inclusión	OrElse	||
Negación	Not	!

La siguiente tabla resume una explicación de los operadores a nivel de bit que realizan funciones lógicas para cada uno de los pares de bits de cada operando

C#

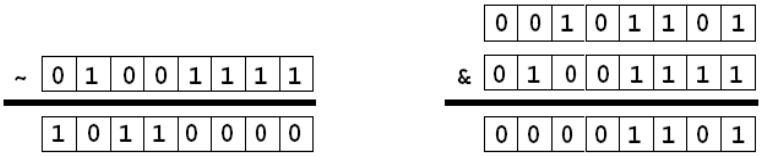
Operador	Operación	Explicación
&	op1 & op2	operador a nivel de bit **Y**
|	op1 | op2	operador a nivel de bit **O**
^	op1 ^ op2	operador a nivel de bit **O excluyente**
~	~ op	operador a nivel de bit **complemento**

VB

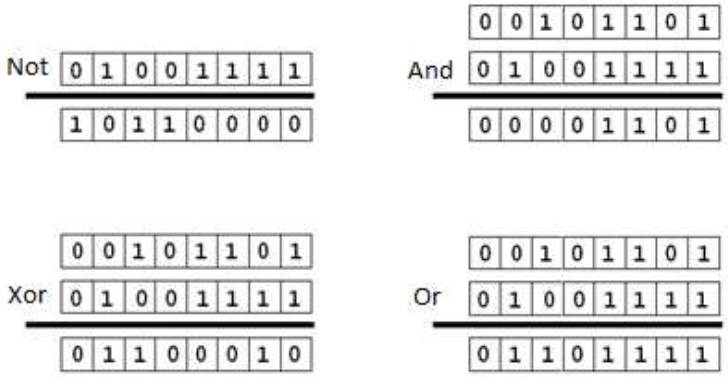
Operador	Operación	Explicación
And	op1 And op2	operador a nivel de bit **Y**
Or	op1 Or op2	operador a nivel de bit **O**
Xor	op1 Xor op2	operador a nivel de bit **O excluyente**
Not	Not op	operador a nivel de bit **complemento**

Ejemplo

C#



VB



La función "**y**" (**and**) activa el bit resultante si los dos operandos son 1.

op1	op2	Resultado
0	0	0
0	1	0
1	0	0
1	1	1

Suponiendo que se quiere evaluar los valores 12 **and** 13:

C#

13 & 12 = 12

VB

13 And 12 = 12

El resultado de esta operación es 12. La causa es que la representación binaria de 12 es 1100 y la de 13 es 1101. La función **and** activa los bits resultantes cuando los bits de los dos operandos son 1, de otra forma el resultado es 0. Entonces si se coloca en línea los dos operandos y se realiza la función **Y**, se puede ver que los dos bits de mayor peso (los dos bits situados más a la izquierda de cada número) son 1 y así el bit resultante de cada uno es 1. Los dos bits de menor peso se evalúan a 0 porque al menos uno de los dos operandos es 0:

	Binario	Decimal
	1 1 0 1	13
&	1 1 0 0	12
	1 1 0 0	12

Ejemplo

C#

```
namespace operadores
{
    class OperadorBitY
    {
        public void Resultados()
        {
            int a = 12, b = 13, c = 0;

            c = b & a;
            Console.WriteLine("a:" + Convert.ToString(a, 2));
            Console.WriteLine("b:" + Convert.ToString(b, 2));
            Console.WriteLine("c = " + Convert.ToString(c, 2));
            Console.WriteLine("c = " + c);
        }
    }
}
```

VB

```
Public Class OperadorBitY
    Public Sub Resultados()
        Dim a As Integer = 12
        Dim b As Integer = 13
        Dim c As Integer = 0

        c = b And a
        Console.WriteLine("a:" + Convert.ToString(a, 2))
        Console.WriteLine("b:" + Convert.ToString(b, 2))
        Console.WriteLine("c = " + Convert.ToString(c, 2))
        Console.WriteLine("c = " + c.ToString())

    End Sub
End Class
```

La salida es

```
a:1100
b:1101
c = 1100
c = 12
```

El operador | ó Or realiza la operación **O inclusivo** y el operador ^ ó Xor realiza la operación **O exclusivo**. **O inclusivo** significa que si uno de los dos operandos es 1 el resultado es 1.

op1	op2	Resultado
0	0	0
0	1	1
1	0	1
1	1	1

O exclusivo significa que si los dos operandos son diferentes el resultado es 1, de otra forma el resultado es 0:

op1	op2	Resultado
0	0	0
0	1	1
1	0	1
1	1	0

Y finalmente el operador **complemento** invierte el valor de cada uno de los bits del operando: si el bit del operando es 1 el resultado es 0 y si el bit del operando es 0 el resultado es 1.

op	Resultado
1	0
0	1

Operadores de desplazamiento

Los operadores de desplazamiento permiten realizar manipulación de los bits en los datos tanto en C# como en VB. Esta tabla resume los operadores lógicos y de desplazamiento disponibles:

Operador	Uso	Descripción
>>	op1 >> op2	desplaza a la derecha op2 bits de op1
<<	op1 << op2	desplaza a la izquierda op2 bits de op1

Los dos operadores de desplazamiento simplemente desplazan los bits del operando de la izquierda el número de posiciones indicadas por el operador de la derecha. Los desplazamientos ocurren en la dirección indicada por el propio operador.

Ejemplo

```
13 >> 1;
```

Desplaza los bits del entero 13 una posición a la derecha. La representación binaria del número 13 es 1101. El resultado de la operación de desplazamiento es 110 o el número 6 en base decimal. Se puede observar que el bit situado más a la derecha (el de menor peso) desaparece. Un desplazamiento a la derecha de un bit es equivalente, pero más eficiente que, dividir el operando de la izquierda por dos y desestimar el resto. Un desplazamiento a la izquierda es equivalente a multiplicar por dos.

Ejemplo

Desplazamiento a derecha:

Operación	Equivale	Resultado
128 >> 1	$128/2^1$	64
256 >> 4	$256/2^4$	16
-256 >> 4	$-256/2^4$	-16

Desplazamiento a izquierda:

Operación	Equivale	Resultado
128 << 1	$128 * 2^1$	256
16 << 2	$16 * 2^2$	64
-16 << 2	$16 * 2^2$	-64

Ejemplo

Operación	Resultado Binario	Resultado Decimal
2423	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 1 1 0 1 1 1	2423
-2423	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 0 0 0 1 0 0 1	-2423
2423 >> 6	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1	37
-2423 >> 6	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 0	-38
2423 << 6	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 1 1 0 1 1 1 0 0 0 0 0 0	155072

Operación	Resultado Binario																															Resultado Decimal
-2423 << 6	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	-155072

Operadores unarios y ternarios

Nota: el operador unario de incremento y decremento cambio su prioridad dependiendo del lugar en el que se encuentre. Cuando se encuentra previo al operando al que afecta es uno de los operadores de mayor prioridad. Sin embargo cuando esta post operando (sólo en C#) pasa a ser el de menor prioridad, *incluyendo la asignación*.

El siguiente ejemplo muestra el comportamiento de los operadores unarios según su prioridad para realizar operaciones y asignaciones.

Ejemplo

C#

```
namespace operadores
{
    class OperadoresUnarios
    {
        public void Resultados()
        {
              int a = 4, b = 5, c = 6, d = 0;
              d = ++a - b-- - c; // d = 5 - 5 - 6
              Console.WriteLine("Valor de d: " + d);
              Console.WriteLine("Valor de a: " + a);
               Console.WriteLine("Valor de b: " + b);
               Console.WriteLine("----------");
               a = 4; b = 5; c = 6; d = 0;
                    d = a++ - b-- - c; // d = 4 - 5 - 6
               Console.WriteLine("Valor de d: " + d);
               Console.WriteLine("Valor de a: " + a);
               Console.WriteLine("Valor de b: " + b);
        }
    }
}
```

VB

```
Public Class OperadoresUnarios
    Public Sub Resultados()
        Dim a As Integer = 4
        Dim b As Integer = 5
        Dim c As Integer = 6
        Dim d As Integer = 0

        d = ++a - b - --c ' d = 5 - 5 - 6
        Console.WriteLine("Valor de d: " + d.ToString())
        Console.WriteLine("Valor de a: " + a.ToString())
        Console.WriteLine("Valor de b: " + b.ToString())
        Console.WriteLine("-----------")
```

```
    End Sub

End Class
```

Notar como se realizan primero las operaciones de pre incremento o decremento para luego resolver el resto de la expresión por medio de la asociatividad explicada. La salida del programa es:

C#

```
Valor de d: -6
Valor de a: 5
Valor de b: 4
-----------
Valor de d: -4
Valor de a: 6
Valor de b: 3
```

VB

Valor de d: -7

Valor de a: 4

Valor de b: 5

El siguiente ejemplo demuestra también las prioridades en las asignaciones para los operadores binarios.

Ejemplo

C#

```
namespace operadores
{
    class OperadoresBinarios
    {
        public void Resultados()
        {
              int a = 4, b = 5, c = 6, d = 0;
              d = a + b * c;
              Console.WriteLine("Valor de d: " + d);
              Console.WriteLine("-----------");
              d += a;
              Console.WriteLine("Valor de d: " + d);
              Console.WriteLine("-----------");
              d -= b * c;
              Console.WriteLine("Valor de d: " + d);
              Console.WriteLine("-----------");
              d %= c;
              Console.WriteLine("Valor de d: " + d);
              Console.WriteLine("-----------");
              d <<= c;
              Console.WriteLine("Valor de d: " + d);
              Console.WriteLine("Valor de c: " + c);
              Console.WriteLine("-----------");
```

```
            d >>= c++;
            Console.WriteLine("Valor de d: " + d);
            Console.WriteLine("Valor de c: " + c);
        }
    }
}
```

VB

```
Public Class OperadoresBinarios
    Public Sub Resultados()
        Dim a As Integer = 4
        Dim b As Integer = 5
        Dim c As Integer = 6
        Dim d As Integer = 0
        d = a + b * c
        Console.WriteLine("Valor de d: " + d.ToString())
        Console.WriteLine("-----------")
        d += a
        Console.WriteLine("Valor de d: " + d.ToString())
        Console.WriteLine("-----------")
        d -= b * c
        Console.WriteLine("Valor de d: " + d.ToString())
        Console.WriteLine("-----------")
        d \= c
        Console.WriteLine("Valor de d: " + d.ToString())
        Console.WriteLine("-----------")
        d <<= c
        Console.WriteLine("Valor de d: " + d.ToString())
        Console.WriteLine("Valor de c: " + c.ToString())
        Console.WriteLine("-----------")
        d >>= c
        Console.WriteLine("Valor de d: " + d.ToString())
        Console.WriteLine("Valor de c: " + c.ToString())
    End Sub
End Class
```

La salida del programa es

C#

```
Valor de d: 34
-----------
Valor de d: 38
-----------
Valor de d: 8
-----------
Valor de d: 2
-----------
Valor de d: 128
Valor de c: 6
-----------
Valor de d: 2
Valor de c: 7
```

VB

```
Valor de d: 34
-----------
Valor de d: 38
-----------
Valor de d: 8
-----------
Valor de d: 1
-----------
Valor de d: 64
Valor de c: 6
-----------
Valor de d: 1
Valor de c: 6
```

Por último se muestra el uso del operador ternario. Este operador debe pensarse como una sentencia condicional en dónde si lo que se afirma en el primer operando es cierto, se retorna el segundo valor, sino, el tercero

Ejemplo

C#

```
namespace operadores
{
    class OperadorTernario
    {
        public void Resultados()
        {
                    int a = 4, b = 5, c = 6, d = 0;
              d = a > b ? c : b;
              Console.WriteLine("Valor de d: " + d);
              d = a < b ? c : b;
              Console.WriteLine("Valor de d: " + d);
              d = a < b ? c : ++b;
              Console.WriteLine("Valor de b: " + b);
              Console.WriteLine("Valor de d: " + d);
              d = a < b ? c : b--;
              Console.WriteLine("Valor de d: " + d);
              Console.WriteLine("Valor de b: " + b);

        }
    }
}
```

VB

```
Public Class OperadorTernario
    Public Sub Resultados()
        Dim a As Integer = 4
        Dim b As Integer = 5
        Dim c As Integer = 6
        Dim d As Integer = 0
```

```
        d = IIf(a > b, c, b)
        Console.WriteLine("Valor de d: " + d.ToString())
        d = IIf(a < b, c, b)
        Console.WriteLine("Valor de d: " + d.ToString())
        d = IIf(a < b, c, ++b)
        Console.WriteLine("Valor de b: " + b.ToString())
        Console.WriteLine("Valor de d: " + d.ToString())
        d = IIf(a < b, c, --b)
        Console.WriteLine("Valor de d: " + d.ToString())
        Console.WriteLine("Valor de b: " + b.ToString())

    End Sub
End Class
```

La salida del programa es:

```
Valor de d: 5
Valor de d: 6
Valor de b: 5
Valor de d: 6
Valor de d: 6
Valor de b: 5
```

Promoción de expresiones en las conversiones de tipo

Las variables se promueven automáticamente a tipos más grandes (como por ejemplo, un `int` a `long` – `Integer` a `Long`). Esto se debe a que .Net permite que si una variable se asigna a otra del mismo tipo primitivo (por ejemplo, dos tipos enteros) automáticamente verifica que el tamaño en donde se va a asignar el valor sea mayor o igual que el mismo.

C#

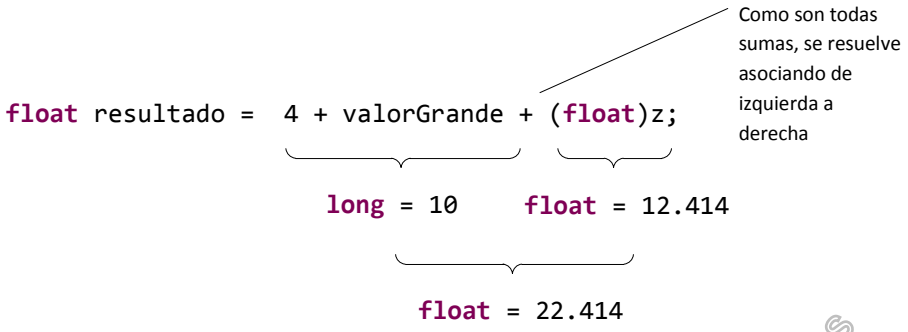
Sin embargo en C#, cuando la variable o valor que se encuentra a la derecha de la asignación es mayor que la que está a izquierda, el lenguaje exige una conversión de tipo explícita (cast) o arroja un error. Por lo tanto, las asignaciones de expresiones son compatibles si el tipo de variable que recibe el valor es al menos del mismo tamaño en bits que el resultado que la expresión arroja

```
long valorGrande = 6; // 6 es un tipo entero, esta bien
//int valorPequeño = 99L; // 99L es un long, esta mal
double z = 12.414F; // 12.414F es float, esta bien
//float z1 = 12.414; // 12.414 es double, esta mal
```

Las promociones en C# ocurren también en expresiones aritméticas como pasos intermedios. Por ejemplo, si se tiene el siguiente código

```
float resultado = 4 + valorGrande + (float)z;
```

La forma en que se realizan los pasos intermedios para resolver la expresión se muestran en el siguiente gráfico.



VB

En VB no se tiene el mismo problema. Si el valor o variable que se encuentra a derecha de la operación es asignable al de la izquierda sin producir un desbordamiento (overflow), la operación se realiza exitosamente sin requerir ninguna declaración específica en el código. En cambio, si el valor a derecha determina un desbordamiento, esto se verifica en tiempo de compilación (en Visual Studio lo determina el intérprete y se marca inmediatamente el error) sin permitir la asignación.

El siguiente ejemplo muestra como prescindir de las conversiones explícitas de tipos primitivos.

```
Module Module1

    Sub Main()
        Dim valorGrande As Long = 6 ' 6 es un tipo entero, esta bien
        Dim valorPequeño As Integer = CLng(999999999) ' 99L es un long, esta mal
        Dim z As Double = 12.4139996F ' 12.414F es float, esta bien
        Dim z1 As Single = 12.414 ' 12.414 es double, esta mal
        Dim resultado As Single = 4 + valorGrande + z

        Console.WriteLine("valorPequeño: " + valorPequeño.ToString())
        Console.WriteLine("Resultado: " + resultado.ToString())
        Console.ReadKey()

    End Sub

End Module
```

Concatenación de Strings

Se pueden concatenar cadenas fácilmente utilizando el operador **+**. Este operador puede crear un nuevo String para resolver una concatenación.

Ejemplo

C#

```
namespace cadenas
{
    class Program
```

```
    {
        static void Main(string[] args)
        {
            String titulo = "Dr.";
            String nombre = "Pedro" + " " + "Ramirez";
            String saludo = titulo + " " + nombre;

            Console.WriteLine(titulo);
            Console.WriteLine(nombre);
            Console.WriteLine(saludo);
            Console.ReadKey();
        }
    }
}
```

VB

```
Module Module1

    Sub Main()
        Dim titulo As String = "Dr."
        Dim nombre As String = "Pedro" & " " & "Ramirez"
        Dim saludo As String = titulo & " " & nombre

        Console.WriteLine(titulo)
        Console.WriteLine(nombre)
        Console.WriteLine(saludo)
        Console.ReadKey()
    End Sub

End Module
```

En una concatenación, si algún elemento no es String, se lo convierte automáticamente en C#, pero se debe tener en cuenta que al menos un elemento debe ser de tipo String. En VB, se puede invocar al método ToString() en cada tipo primitivo

El siguiente fragmento de código concatena tres cadenas

Ejemplo

C#

```
"La entrada tiene " + contador + " caracteres."
```

VB

```
"La entrada tiene " & contador. ToString() & " caracteres."
```

Dos de las cadenas concatenadas son literales: "La entrada tiene " y " caracteres.". El tercer String, el del medio, es realmente un entero que primero se convierte a tipo String y luego se concatena con las otras cadenas.

Conversión automática de tipos primitivos en objetos: autoboxing

Boxing y unboxing es un concepto esencial en el sistema de tipos de .Net. Con Boxing y unboxing se puede vincular tipos por valor y referencia al permitir que cualquier tipo por valor se convierta a un tipo de objeto. Boxing y unboxing permite una visión unificada del sistema de tipos en el que un valor de cualquier tipo en última instancia, puede ser tratado como un objeto. La conversión de un tipo por valor a un tipo por referencia se denomina Boxing. Unboxing es la operación inversa.

.NET proporciona un "sistema de tipos unificado". Todos los tipos - incluyendo los tipos por valor - se derivan del tipo de `Object`. Es posible llamar a métodos de objetos de cualquier valor, incluso de los valores "primitivos" como un entero.

Ejemplo

C#

```
int i = 1;
object o = i; // boxing
int j = (int) o; // unboxing
System.Console.WriteLine("El valor de i: {0}", i);
System.Console.WriteLine("El valor de o: {0}", o);
System.Console.WriteLine(3.ToString());
System.Console.WriteLine(3);
System.Console.ReadKey();
```

VB

```
Dim i As Integer = 1
Dim o As Object = i ' boxing
Dim j As Integer = CInt(o) ' unboxing
System.Console.WriteLine("El valor de i: {0}", i)
System.Console.WriteLine("El valor de o: {0}", o)
System.Console.WriteLine(3.ToString())
System.Console.WriteLine(3)
System.Console.ReadKey()
```


Un valor entero se puede convertir en objeto y volver de nuevo a un entero.

En este ejemplo se muestra tanto el boxing como el unboxing. Cuando una variable de un tipo por valor tiene que ser convertido a un tipo por referencia, un objeto se asigna para contener el valor, y el valor se copia en dentro de él.

Unboxing es todo lo contrario. Cuando a un objeto se convierte su tipo al tipo por valor inicial, el valor se copia fuera del objeto y en una ubicación de almacenamiento apropiada. El siguiente ejemplo muestra el manejo de los espacios de almacenamiento.

Nota: en el ejemplo se utilizan objetos como ArrayList que serán explicados posteriormente

Ejemplo

C#

```
namespace Unboxing
{
    class Program
    {
        static void Main(string[] args)
        {
            // Crear una colección.
            ArrayList fechas = new ArrayList();
            // Crear un objeto y agregarlo a la colección.
            // En tiempo de ejecución se encajona (boxing) al objeto.
            DateTime fec = new DateTime(2008, 1, 1);
            fechas.Add(fec);
            // Recuperar un objeto de la colección y convertirlo a DateTime.
            // El CLR realiza el unboxing del objeto para obtener una copia del
              // heap.
            DateTime copia1 = (DateTime)fechas[0];
            // Modificar copia1 y mostrarlo por pantalla. El resultado es
              // 02/01/2008.
            copia1 = copia1.AddDays(1);
            Console.WriteLine("Fecha en copia1: {0}", copia1.ToShortDateString());
            // Recuperar un objeto nuevamente de la colección.
            // El CLR realiza el unboxing del objeto para obtener otra copia del
              // heap.
            DateTime copia2 = (DateTime)fechas[0];
            // Modificar copia2 y mostrarlo por pantalla. El resultado es
              // 01/01/2008.
            copia2 = copia2.AddMonths(1);
            Console.WriteLine("Fecha en copia2: {0}", copia2.ToShortDateString());
        }
    }
}
```

VB

```
Module Module1

    Sub Main()
        ' Crear una colección.
        Dim fechas As New ArrayList()
        ' Crear un objeto y agregarlo a la colección.
        ' En tiempo de ejecución se encajona (boxing) al objeto.
        Dim fec As New DateTime(2008, 1, 1)
        fechas.Add(fec)
        ' Recuperar un objeto de la colección y convertirlo a DateTime.
        ' El CLR realiza el unboxing del objeto para obtener una copia del heap.
        Dim copia1 As DateTime = CType(fechas(0), DateTime)
        ' Modificar copia1 y mostrarlo por pantalla. El resultado es 02/01/2008.
        copia1 = copia1.AddDays(1)
        Console.WriteLine("Fecha en copia1: {0}", copia1.ToShortDateString())
        ' Recuperar un objeto nuevamente de la colección.
        ' El CLR realiza el unboxing del objeto para obtener otra copia del heap.
        Dim copia2 As DateTime = CType(fechas(0), DateTime)
```

```
        ' Modificar copia2 y mostrarlo por pantalla. El resultado es 01/01/2008.
        copia2 = copia2.AddMonths(1)
        Console.WriteLine("Fecha en copia2: {0}", copia2.ToShortDateString())
    End Sub
End Module
```

Convirtiendo Strings a valores numéricos

Una necesidad común de un desarrollador es convertir cadenas a valores numéricos para luego manipularlos como un dato primitivo (por ejemplo, un `int`, `float`, `double`, etc… en C# o `Integer`, `Single`, `Double`, etc… en VB). Existen numerosas situaciones en la que esta necesidad se presenta (lecturas de archivos, parámetros de línea de comando, etc…), o tan sólo tomar un valor ingresado en el campo de texto de una interfaz gráfica para usuarios.

Todas las conversiones excepto, se realizan con métodos que tienen nombres similares

VB	C#	Método de Conversión
Byte	byte	Byte.Parse(unString)
Short	short	Int16.Parse(unString)
Integer	int	Int32.Parse(unString)
Long	long	Int64.Parse(unString)
Single	float	Single.Parse(unString)
Double	double	Double.Parse(unString)
Boolean	bool	Boolean.Parse(unString)
Decimal	decimal	Decimal.Parse(unString)
Char	char	Char.Parse(unString)

Ejemplo

C#

```
namespace conversionesDeCadenas
{
    class Program
    {
        static void Main(string[] args)
        {
            byte b = Byte.Parse("30");
            short s = Int16.Parse("30000");
            int i = Int32.Parse("3000000");
            long l = Int64.Parse("300000000000000000");
            float f = Single.Parse("3.1416");
            double d = Double.Parse("3.14159265358979323846264338327950288");
            bool bl = Boolean.Parse("true");
            decimal dec = Decimal.Parse("3.14159265358979323846");
            char c = Char.Parse("a");

            Console.WriteLine(b);
            Console.WriteLine(s);
            Console.WriteLine(i);
```

```
            Console.WriteLine(l);
            Console.WriteLine(f);
            Console.WriteLine(d);
            Console.WriteLine(bl);
            Console.WriteLine(dec);
            Console.WriteLine(c);
            Console.ReadKey();
        }
    }
}
```

VB

```
Module Module1

    Sub Main()
        Dim b As Byte = Byte.Parse("30")
        Dim s As Short = Int16.Parse("30000")
        Dim i As Integer = Int32.Parse("3000000")
        Dim l As Long = Int64.Parse("300000000000000000")
        Dim f As Single = Single.Parse("3.1416")
        Dim d As Double = Double.Parse("3.14159265358979323846264338327950288")
        Dim bl As Boolean = Boolean.Parse("True")
        Dim dec As Decimal = Decimal.Parse("3.14159265358979323846")
        Dim c As Char = Char.Parse("a")

        Console.WriteLine(b)
        Console.WriteLine(s)
        Console.WriteLine(i)
        Console.WriteLine(l)
        Console.WriteLine(f)
        Console.WriteLine(d)
        Console.WriteLine(bl)
        Console.WriteLine(dec)
        Console.WriteLine(c)
        Console.ReadKey()

    End Sub

End Module
```

La salida producida es:

```
30
30000
3000000
300000000000000000
31416
3,14159265358979E+35
True
3141592653589793238 46
a
```

Si bien la conversión es simple, se debe tener en cuenta que para cada tipo de datos esta involucrado un único nombre de método en cada clase del tipo correspondiente en el CLR.

Si en cualquiera de los casos mencionados no se puede convertir el tipo String, se produce un error en tiempo de ejecución.

Cabe destacar que este tipo de conversiones se pueden hacer por otros medios (el más popular posiblemente es el método ToString() de la clase Convert). Sin embargo puede ser nemotécnico recordar y saber que cada tipo tiene un método que interpreta cadenas y los convierte a números

El siguiente es un ejemplo integrador de los operadores de desplazamiento a nivel de bits y el método estático de la clase Convert para mostrar números binarios en formato String

Ejemplo

C#

```
namespace operadores
{
    class OperadoresDesplazamiento
    {
        public void Resultados()
        {
            int d = 2423;
            int resultado = d;

            Console.WriteLine("Decimal:" + resultado + "\t\tResultado: " +
              Convert.ToString(resultado, 2));
            resultado = -d;
            Console.WriteLine("Decimal:" + resultado + "\t\tResultado: " +
              Convert.ToString(resultado, 2));
            resultado = d >> 6;
            Console.WriteLine("Decimal:" + resultado + "\t\tResultado: " +
              Convert.ToString(resultado, 2));
            resultado = -d >> 6;
            Console.WriteLine("Decimal:" + resultado + "\t\tResultado: " +
              Convert.ToString(resultado, 2));
            resultado = d << 6;
            Console.WriteLine("Decimal:" + resultado + "\t\tResultado: " +
              Convert.ToString(resultado, 2));
            resultado = -d << 6;
            Console.WriteLine("Decimal:" + resultado + "\t\tResultado: " +
              Convert.ToString(resultado, 2));
        }
    }
}
```

VB

```
Public Class OperadoresDesplazamiento
    Public Sub Resultados()
        Dim d As Integer = 2423
        Dim resultado As Integer = d

        Console.WriteLine("Decimal:" + resultado.ToString() + Chr(9) + Chr(9) + _
              "Resultado: " + Convert.ToString(resultado, 2))
        resultado = -d
```

```
        Console.WriteLine("Decimal:" + resultado.ToString() + Chr(9) + Chr(9) + _
              "Resultado: " + Convert.ToString(resultado, 2))
        resultado = d >> 6
        Console.WriteLine("Decimal:" + resultado.ToString() + Chr(9) + Chr(9) + _
              "Resultado: " + Convert.ToString(resultado, 2))
        resultado = -d >> 6
        Console.WriteLine("Decimal:" + resultado.ToString() + Chr(9) + Chr(9) + _
              "Resultado: " + Convert.ToString(resultado, 2))
        resultado = d << 6
        Console.WriteLine("Decimal:" + resultado.ToString() + Chr(9) + Chr(9) + _
              "Resultado: " + Convert.ToString(resultado, 2))
        resultado = -d << 6
        Console.WriteLine("Decimal:" + resultado.ToString() + Chr(9) + Chr(9) + _
              "Resultado: " + Convert.ToString(resultado, 2))
    End Sub
End Class
```

La salida obtenida es:

C#

Decimal:2423 Resultado: 100101110111
Decimal:-2423 Resultado: 11111111111111111111011010001001
Decimal:37 Resultado: 100101
Decimal:-38 Resultado: 11111111111111111111111111011010
Decimal:155072 Resultado: 100101110111000000
Decimal:-155072 Resultado: 11111111111111011010001001000000

VB

Decimal:2423 Resultado: 100101110111
Decimal:-2423 Resultado: 11111111111111111111011010001001
Decimal:37 Resultado: 100101
Decimal:-38 Resultado: 11111111111111111111111111011010
Decimal:155072 Resultado: 100101110111000000
Decimal:-155072 Resultado: 11111111111111011010001001000000

Nota: Las conversiones a String generan objetos inmutables, esto quiere decir que ***si se le cambia el valor que contienen generan un nuevo objeto*** descartando el anterior para ser recolectado como basura del heap.