C sharp NET/Capítulo 3

CAPÍTULO 3

En el capítulo 2 hemos introducido nuestro primer programa en C#, un programa sencillo el cual incorpora muchos temas que hemos cubierto básicamente y solamente en parte. En esta sección del libro procuraremos ver más a fondo las partes básicas de C#. Nos internaremos más a fondo en la sintaxis y la estructura de C#.

En este capítulo cubriremos lo que son los *tipos*. Hablaremos de tipos básicos o internos y de cómo crear nuevos tipos. También hablaremos en general de la manipulación de datos. Hablaremos sobre condicionales, operadores matemáticos y varios otros temas relacionados. Empecemos entonces nuestro estudio con lo que son tipos.

Tipos

Importancia de los tipos de datos

Los tipos son la base de cualquier programa. Un tipo no es más que un espacio en el que se almacena una información, ya sean números, palabras o tu fecha de nacimiento.

Tipos en C#

C# es un lenguaje de tipeado seguro (o fuertemente tipado) lo cual quiere decir que el programador debe definir a que **tipo** pertenece cada pedazo de información o cada **objeto** que se crea. De esta forma podemos crear objetos de tipo número entero, de tipo cadenas de texto, de tipo ventana, de tipo botones, entre otros. Haciendo esto, C# nos ayudará a mantener nuestro código seguro en donde cada tipo cumple con su función. En todas las operaciones el compilador comprueba los tipos para ver su compatibilidad. Las operaciones no válidas no se compilan. De esta forma se evitan muchos errores y se consigue una mayor fiabilidad. Esto también permite a C# anticipar de antemano la cantidad de recursos del sistema que nuestro programa utilizará haciendo nuestro código seguro y eficiente.

"Los tipos en C# al igual que C++ y Java se clasifican en dos secciones: Tipos básicos o internos y tipos creados por el usuario. Los tipos básicos no son más que alias para tipos predefinidos en la librería base de la plataforma .NET. Así, el tipo **número entero** (que se representa con la palabra clave **int**), no es más que una forma rápida de escribir `System.Int32`.

Dentro de estas dos secciones los tipos del lenguaje C# también son divididos en dos grandes categorías: tipos por valor y tipos por referencia. Existe una tercera categoría de tipos, disponible solo cuando se usa código no seguro: los punteros, que se discutirán más adelante cuando hablemos de los objetos COM.

Los tipos por valor difieren de los tipos por referencia en que las variables de los tipos por valor contienen directamente su valor, mientras que las variables de los tipos por referencia almacenan la dirección donde se encuentran los objetos, es por eso que se las llaman referencias. Más adelante describiremos como funcionan cada una de estas categorías.

Tipos básicos o internos

Los tipos básicos como hemos dicho son espacios predefinidos y categorizados donde se almacena información. En C# tenemos los siguientes tipos internos:

Tabla 3.1 - Tipos básicos

Tipo C#	Nombre para la plataforma .NET	Con signo?	Bytes utilizados	Valores que soporta
bool	System.Boolean	No	1	true o false (verdadero o falso en inglés)
byte	System.Byte	No	1	0 hasta 255
sbyte	System.SByte	Si	1	-128 hasta 127
short	System.Int16	Si	2	-32.768 hasta 32.767
ushort	System.Uint16	No	2	0 hasta 65535
int	System.Int32	Si	4	-2.147.483.648 hasta 2.147.483.647
uint	System.Uint32	No	4	0 hasta 4.394.967.395
long	System.Int64	Si	8	-9.223.372.036.854.775.808 hasta 9.223.372.036.854.775.807
ulong	System.Uint64	No	8	0 hasta 18446744073709551615
float	System.Single	Si	4	Approximadamente ±1.5E-45 hasta ±3.4E38 con 7 cifras significativas
double	System.Double	Si	8	Approximadamente ±5.0E-324 hasta ±1.7E308 con 7 cifras significativas
decimal	System.Decimal	Si	12	Approximadamente ±1.0E-28 hasta ±7.9E28 con 28 ó 29 cifras significativas
char	System.Char		2	Cualquier carácter Unicode (16 bits)

C# tiene una ventaja y característica especial sobre los demás lenguajes de programación modernos y es que cada vez que se crea un objeto de un tipo básico, éstos son mapeados internamente a un tipo primitivo de la plataforma .NET el cual es parte del CLS (Especificación común del lenguaje) lo cual nos permite acceder y hacer uso de estos desde cualquier lenguaje de la plataforma .NET. Es decir si es que creamos un objeto de tipo int (entero) en C#, ese objeto podrá ser usado como tal dentro de J#, JScript, Visual Basic .NET y cualquier otro lenguaje que conforme los requisitos de .NET.

Escogiendo qué tipo usar

A la hora de programar deberéis decidir qué tipo de variables querréis usar. Generalmente esta decisión se basa en el tipo de información que vayáis a usar y en el tamaño de la información. Por ejemplo en nuestro ejemplo 2.2 del capítulo anterior necesitábamos hacer la suma de dos valores numéricos por lo que usamos dos tipos básicos de número entero (usando la palabra clave **int**) los cuales de acuerdo con nuestra tabla 3.1 son números enteros (no pueden llevar valores decimales) y podrán aceptar valores entre -2,147,483,648 y 2,147,483,647 lo cual es más que suficiente para nuestro ejemplo de añadir dos números. En el caso de que necesitáramos hacer uso de números reales (los cuales poseen una parte entera y una parte decimal como el número 10.22) podremos hacer uso del tipo **float**, **double** y **decimal** de acuerdo con el tamaño del número que necesitemos y así cada uno de los tipos tiene su uso y capacidad de acuerdo con la tabla 3.1.

A continuación explicaremos brevemente los tipos más usados en C#:

Enteros

Los tipos que sirven para almacenar números enteros son: byte, sbyte. short, ushort, int, uint, long y ulong. Como se aprecia en la tabla, C# define versiones con y sin signo para tipos con los mismos bytes utilizados. Cada tipo se distingue por la capacidad de almacenaje.

Probablemente el tipo más utilizado es el **int**, púes se utiliza para controlar matrices, indexar arreglos (arrays) además de las operaciones normales con enteros. Además, se trata de un entero de tamaño medio: más pequeño que long y ulong, pero más grande que byte, sbyte, short y ushort.

El siguiente ejemplo muestra la declaración y uso de algunos tipos enteros calculando el número de segundos en una hora, dia y en un año.

Ejemplo 3.1 - utilizando tipos enteros (int)

```
// Ejemplo 3.1 - utilizando tipos enteros (int)
using System;
class Enteros{
   public static void Main()
   {
       int minuto = 60;    //segundos por minuto
       int hora = minuto*60;
       int dia = hora*24;
       long anio = dia*365;
       Console.WriteLine("Segundos en un dia: {0}", dia);
       Console.WriteLine("Segundos en un año: {0}", anio);
   }
}
```

De nuevo hemos usado el método Console.WriteLine para imprimir los resultados por la consola. El identificador {0} dentro de la cadena de texto indica que se sustituye {0} por el primer argumento. si hubiera más de un argumento, se seguiría con {1}, y así sucesivamente. Por ejemplo, las dos líneas que utilizan Console.WriteLine se pueden simplificar así:

```
   Console.WriteLine("En un dia: {0}; en un año: {1}", dia, anio );
```

Tipos de coma flotante

Los tipos de coma flotante sirven para representar a números con parte fraccionaria. La representación por supuesto puede no ser exacta, bien por errores de la máquina, bien porque el número de decimales que se puede alojar es finito.

Existen tres clases de tipos de punto flotante : float, double y decimal. De los dos, el más usado es double, púes es el valor que devuelven la mayoría de las funciones matemáticas de la librería base.

El siguiente ejemplo calcula la raíz cuadrada y el logaritmo de dos:

Ejemplo 3.2 - utilizando tipos flotantes

```
// Ejemplo 3.2 - utilizando tipos flotantes
using System;

class Flotante{
   public static void Main()
   {
       int a = 2;
```

```
        double log2 = Math.Log(a);
        double raiz2 = Math.Sqrt(a);
        Console.WriteLine("El logaritmo de dos es {0}", log2 );
        Console.WriteLine("La raiz de dos es {0}", raiz2 );
    }
 }
```

y la salida será la siguiente:

```
El logaritmo de dos es 0.693147180559945
La raiz de dos es 1.4142135623731
```

si intentamos cambiar el tipo de log2 a otro de menos precisión, como float o int, el compilador protestará. Esto se debe, como hemos dicho a que el valor devuelto por Math.Log() es de tipo double y si se quiere convertir a float, pués se perderán datos. Lo mismo ocurre con la mayoría de los miembros de la clase Math, como Math.Sin(), Math.Tan(), etc.

El tipo decimal

El tipo decimal es un tipo "nuevo" en el sentido de que no tiene equivalente en C/C++. Es muy parecido a los tipo de coma flotante float y double.

En la aritmética de los tipos de coma flotante ordinarios, se pueden producir leves errores de redondeo. El tipo decimal elimina estos errores y puede representar correctamente hasta 28 lugares decimales. Esta capacidad para representar valores decimales sin errores de redondeo lo hace especialmente eficaz para cálculos monetarios.

El tipo bool

El tipo bool sirve para expresar los valores verdadero/falso, que en C# se muestran con las palabras reservadas true y false.

En C#, por ejemplo, una instrucción de condición solo puede estar gobernada por un valor bool, no como en C/C++, que lo puede estar también por un entero. De esta forma se ayuda a eliminar el error tan frecuente en programadores de C/C++ cuando usa "=" en lugar de "==". En definitiva, la inclusión del tipo bool en el lenguaje ayuda a la claridad del código y evita algunos errores muy comunes.

El siguiente ejemplo, muestra algunos usos del tipo bool:

Ejemplo 3.3 - utilizando tipos de decisión bool

```
// Ejemplo 3.1 - utilizando tipos de decisión bool
using System;
class Booleano{
   public static void Main()
   {
       bool b;
       b = true;
       Console.WriteLine("b es {0}", b);
       if(b)
       {
          Console.WriteLine("esto saldrá");
       }
       b = false;
       if(b)
```

```
        {
            Console.WriteLine("esto no saldrá");
        }
        Console.WriteLine("2==2 es {0}", 2==2);
    }
}
```

En la última línea se muestra que el operador "==" también devuelve un valor booleano. El resultado debería ser el siguiente:

```
b es True
esto saldrá
2==2 es True
```

El tipo char

El tipo char permite almacenar un carácter en formato simple, unicode de 16 bits o caracteres de escape. Usando el formato unicode nos garantiza que los acentos se ven de forma adecuada y además permite la representación de otros alfabetos, como el japonés, griego, cirílico, etc. Para introducir un carácter se utilizan comillas simples, de forma que declarar un carácter sigue la estructura

```
char letra1 = 'a'; //formato simple
char letra2 = '\u0041'; //formato Unicode que representa la letra A
char letra3 = '\n'; //formato carácter de escape
```

Para una lista completa de caracteres unicode podréis visitar la siguiente página: http://unicode.coeurlumiere.com/

La siguiente lista contiene los caracteres de escape comunes y su significado:

```
\' apostrofe
\" Comillas
\\ Backslash
\0 Null (nulo)
\a Alerta
\b Retroceso
\f Form feed
\n Línea nueva
\r Retorno del carro
\t Tabulación Horizontal
\v Tabulación Vertical
```

Tipo Cadenas

Los tipos cadena (palabra clave **string**) son tipos que almacenan un grupo de caracteres. En C# los tipos cadena se crean con la palabra clave **string** seguido por el nombre de la variable que deseamos instanciar. Para asignar un valor a este tipo debemos hacerlo entre comillas de la siguiente forma:

```
string miCadena = "Esta es una cadena de caracteres";
```

Debido a que el tipo cadena (string) es uno de los tipos más usados en C#, lo estudiaremos detalladamente más adelante.

Convirtiendo tipos

En nuestros programas muchas veces necesitaremos cambiar de tipo a los objetos que hayamos creado. Esto lo podremos hacer implícitamente o explícitamente. Una conversión de tipos implícita sucede automáticamente, es decir el compilador se hará cargo de esto. Una conversión explicita en cambio se llevará a cabo únicamente cuando nosotros lo especifiquemos. Hay que tomar en cuenta que no siempre podremos hacer una conversión de un tipo hacia otro.

Como regla general las conversiones implícitas se llevan a cabo cuando se desea cambiar **un tipo de menor capacidad hacia un tipo de mayor capacidad de la misma especie**. Por ejemplo si deseamos crear 2 tipos enteros (misma clase) el uno que lleve el tipo short (menor capacidad) y el otro que lleve el tipo int (mayor capacidad) una conversión implicita de short a int se lleva a cabo en el siguiente ejemplo:

```
short corto = 3;
int entero = corto; //compilará sin ningún problema
```

aquí sucede una conversión implícita, el valor de la variable corto (en este caso 3) que es de tipo **short** es asignado a la variable de tipo **int** sin que el compilador nos de ningún problema ya que hará una conversión de **short** a **int** implícitamente por nosotros debido a la regla anteriormente citada.

En el caso que queramos hacer de forma inversa, es decir asignar un valor int a una variable short, estaríamos violando la regla de asignar un tipo de menor capacidad a una variable de tipo de mayor capacidad aunque sean de la misma clase (enteros). Asi el siguiente ejemplo no compilará dándonos un error:

```
int entero = 300;
short corto = entero; //nos dará un error de compilación
```

En estos casos es cuando podremos hacer una conversión explícita. Debido a que la información almacenada en la variable entero de tipo int está también en el rango de capacidad del tipo short y los dos tipos son de la misma clase (enteros) podremos hacer una conversión explicita designando entre paréntesis a que tipo queremos convertir de la siguiente manera:

```
int entero = 300;
short corto = (short) entero; //convertirá la variable entero para que sea del tipo short
```

En el ejemplo anterior, el compilador no nos dará ningún problema. Cada uno de los tipos básicos citados a continuación soportará una conversión implícita o explícita como se lo expresa en la siguiente tabla:

Conversión Implicita (I) Conversión Explícita (E)

Desde/hacia	byte	sbyte	short	ushort	int	uint	long	ulong	float	double	decimal
byte	I	E	I	I	I	I	I	I	I	I	I
sbyte	E	I	I	E	I	E	I	E	I	I	I
short	E	E	I	E	I	E	I	E	I	I	I
ushort	E	E	E	I	I	I	I	I	I	I	I
int	E	E	E	E	I	E	I	E	I	I	I
uint	E	E	E	E	E	I	I	I	I	I	I
long	E	E	E	E	E	E	I	E	I	I	I
ulong	E	E	E	E	E	E	E	I	I	I	I
float	E	E	E	E	E	E	E	E	I	I	E
double	E	E	E	E	E	E	E	E	E	I	E

decimal	E	E	E	E	E	E	E	E	E	E	I

Arreglos

En C# se pueden construir arreglos de prácticamente cualquier tipo de dato. Los arreglos, también llamados vectores o arrays, no son más que una sucesión de datos del mismo tipo. Por ejemplo, el concepto matemático de vector es una sucesión de números y por lo tanto es un arreglo unidimensional. Así, podemos construir arreglos de objetos, de cadenas de texto, y, por supuesto, arreglos de enteros:

```
using System;

class Arreglo{
   public static void Main()
   {
        int[] arr = new int[3];
        arr[0] = 1;
        arr[1] = 2;
        arr[2] = 3;
        Console.WriteLine( arr[1] );
   }
}
```

Para crear un arreglo debemos especificar de qué **tipo** deseamos crear el arreglo seguido por corchetes **[ ]** que es el distintivo del arreglo (en nuestro ejemplo usamos int[]), seguido por la palabra clave new y el tipo y la cantidad de parámetros que tendrá nuestro arreglo. En el ejemplo anterior, por ejemplo, se creó un arreglo **arr** unidimensional con capacidad para 3 enteros (especificado por new int[3]), y luego se le asignó a cada parámetro un entero distinto (nótese que se comienza a contar a partir de 0 y el número del parametro se encuentran entre corchetes).

Existe una forma más corta para declarar el arreglo y asignarle los valores:

```
int[] arr = {1,2,3}; //es exactamente lo mismo que el ejemplo anterior
```

También se pueden crear arreglos bidimensionales (y de la misma forma para más dimensiones). En ese caso la sintaxis para declarar un arreglo bidimensional de enteros será

```
int[,] arr; //declaración de arreglos bidimensionales en C#
```

en contraposición a C/C++, en el que se declararía como

```
int[][] arr; //declaración de arreglos bidimensionales en C/C++
```

De esta forma, un arreglo bidimensional se declararía y utilizaría de la siguiente forma:

```
using System;
class Arreglo2{
  public static void Main()
  {
      int[,] arr = new int[2,2];
      arr[0,0] = 1;
      arr[1,0] = 2;
      arr[0,1] = 3;
      arr[1,1] = 4;
      Console.WriteLine("El valor que posee la variable arr[1,1] es {0}", arr[1,1] );
```

```
  }
}
```

el resultado será:

```
El valor que posee la variable arr[1,1] es 4
```

igual que el ejemplo anterior, podemos declarar todo el arreglo de la siguiente forma:

```
int[,] arr = ;
```

Hablaremos más sobre arreglos más adelante

Identificadores, Variables, Constantes y Enumeraciones

Identificadores

En una aplicación siempre se deben crear variables, constantes, métodos, objetos, etc. Para poder crearlos debemos asignar nombres o identificadores. Estos identificadores deben seguir ciertas reglas:

1. Un identificador DEBE empezar con una letra o un signo _
2. Un identificador NO puede tener espacios en blanco
3. Un identificador NO puede llevar el mismo nombre que una palabra clave

Después de que el identificador empiece con una letra o un simbolo _, el identificador puede tener cualquier cantidad de letras, líneas o números.

Ejemplo:

```
EsteIdentificadorEsValido
_este_también
esteEsOtro1
esteEsOtro2
```

como ejemplos inválidos tenemos:

```
Esto es invalido
123_Otro_inválido
int
```

en los ejemplos que no son válidos, el primero contiene espacios, el segundo empieza con números y el tercero es una palabra clave lo cual no es permitido.

Nota: Es recomendado que las variables siempre empiecen con minusculas y que sigan el patrón llamado *Camello* es decir que el nombre de la variable que tenga varias palabras debe ser formado de la siguiente manera: la primera palabra empezará con minúsculas pero la segunda, tercera, etc palabras estarán unidas a la primera y tendrán su primera letra en mayúsculas ejemplo: **miVariable**. También se ha recomendado que el nombre de Métodos, Clases y demás nombres que necesitamos especificar deberán llevar el mismo formato anterior con la excepción de que la Primera letra deberá ser mayúscula.

Variables

Una variable es el nombre que se le da al espacio donde se almacena la información de los tipos. Las variables pueden llevar cualquier nombre que deseemos, eso sí no se podrá hacer uso de palabras claves de C#. En los ejemplos anteriores hemos usado varias variables con diferentes tipos y diferentes valores.

Para crear una variable debemos especificar a qué tipo pertenece antes del nombre que le vamos a dar. Por ejemplo si deseamos crear una variable que se llame **var** y que sea del tipo entero (**int**) procederemos de la siguiente manera:

```
int var;
```

Una vez creada la variable podemos almacenar la información que deseamos. Para hacerlo utilizamos el símbolo = después del nombre de la variable.

```
var = 10;
```

Hay que tener presente que la variable como su nombre lo indica podrá tomar otros valores. Por ejemplo si deseamos que nuestra variable cambie de valor a 5 hacemos lo que habíamos hecho en el ejemplo anterior pero con el nuevo valor:

```
var = 5;
```

De ahora en adelante la variable var pasa a tener el valor de 5.

Para simplificar el proceso de creación y asignación de valor de una variable podemos especificar estos dos procesos en una misma línea.

```
int var = 10;
```

La primera vez que una variable recibe un valor se llama inicialización de la variable. En C# todas las variables que van a ser utilizadas deben ser inicializadas para que vuestro programa pueda funcionar.

En caso anterior hemos creado la variable **var** e inicializado la variable con el valor 10. Hay que tener cuidado que la creación de una variable se la hace 1 sola vez y la asignación de diferentes valores se puede hacer cuantas veces queramos.

Constantes

Las constantes como su nombre lo indica son variables cuyo valor no puede ser alterado. Éstas se utilizan para definir valores que no cambian con el tiempo. Por ejemplo podemos definir una constante para especificar cuantos segundos hay en una hora de la siguiente forma:

```
const int segPorHora = 3600;
```

Esta constante no podrá ser cambiada a lo largo de nuestro programa. En el caso de que queramos asignarle otro valor, el compilador nos dará un error.

Las constantes deben ser inicializadas en el momento de su creación.

Enumeraciones

Supongamos que estamos diseñando un juego y necesitamos crear una variable para saber cuál es el estado del tanque de combustible de nuestro automóvil. Suponiendo que tenemos 4 niveles en el tanque: lleno, medio, bajo y crítico. ¿Qué tipo de variable podríamos usar para especificar estos estados? Una forma de hacerlo podría ser si especificamos una variable de tipo int (entero) que tome los valores de 1 para lleno, 2 para medio, 3 para bajo y 4 para crítico. Esta alternativa funcionaría pero a la larga nos olvidaremos qué número representaba qué. Una forma muy elegante de solucionar este problema es utilizando el tipo **enum** o enumeraciones. Veamos el siguiente ejemplo

para comprender este concepto:

Ejemplo 3.1 - Control del estado de combustible

```
using System;
namespace Autos
{
class Control
{
  enum tanque
  {
    lleno,
    medio,
    bajo,
    critico,
  }

  static void Main()
  {
    tanque auto1 = tanque.lleno;
    RevisarEstadoTanque(auto1);

    auto1 = tanque.critico;
    RevisarEstadoTanque(auto1);
  }

  static void RevisarEstadoTanque(tanque auto)
  {
    if (auto==tanque.lleno)
          Console.WriteLine ("¡El tanque está lleno!");
    if (auto==tanque.medio)
          Console.WriteLine ("El tanque está por la mitad");
    if (auto==tanque.bajo)
          Console.WriteLine ("¡El tanque está casi vacío!");
    if (auto==tanque.critico)
          Console.WriteLine ("¡Alerta! tu auto se quedó sin combustible");
  }

}

}
```

Este programa sencillo crea una enumeración llamada tanque y dentro de ella crea 4 constantes: lleno, medio, bajo, y critico. Dentro de nuestro programa creamos la variable de **tipo tanque** llamada **auto1** la cual podrá tomar los valores especificados dentro de la enumeración. Cuando asignamos a la variable auto1 el valor de **tanque.lleno** y revisamos el estado del tanque llamando a la función **RevisarEstadoTanque**, podremos comprobar cuál es el estado actual del tanque de combustible. Ésta es una forma muy descriptiva de cómo crear variables que cambien de estado, una solución elegante y sencilla a nuestro problema.

En realidad las constantes dentro de las enumeraciones tienen valores enteros asignados. Estos valores pueden ser inicializados con distintos valores que nosotros deseemos e incluso podemos especificar qué tipo de entero queremos usar. En nuestro ejemplo anterior los valores de la enumeración son valores enteros del tipo **int** el primer elemento dentro de la enumeración tiene asignado el valor de 0, el siguiente el valor 1 y así sucesivamente. Esto sucede cuando no especificamos a qué tipo queremos inicializar nuestra enumeración y tampoco asignamos valores a las constantes. En el siguiente ejemplo podemos ver cómo especificar otro tipo de constantes y otros valores.

Supongamos que queremos especificar cuantos segundos hay en un minuto, cuantos segundos hay en una hora y cuantos segundos hay en 24 horas. Con enumeraciones lo podemos hacer de la siguiente manera:

Ejemplo 3.2 - Enumeraciones

```
 using System;
 namespace Ejemplos
 {
 class Enumeraciones
 {
   enum segundos :uint
   {
     minuto = 60,
     hora = 3600,
     dia = 86400,
   }
   static void Main()
   {
     Console.WriteLine("Existen {0} segundos en 1 minuto, {1} segundos
en 1 hora y {2} segundos en 24 horas",(uint)segundos.minuto,
(uint)segundos.hora, (uint)segundos.dia);
   }
 }
 }
```

El resultado es el siguiente:

```
Existen 60 segundos en 1 minuto, 3600 segundos en 1 hora y 86400 segundos en 24 horas
```

El ejemplo anterior nos muestra otra forma de usar una enumeración. Hay que tener en cuenta que el tipo que va a tener la enumeración se encuentra después del nombre de la enumeración precedido por dos puntos. De esa forma podremos especificar de qué tipo son. Como habíamos dicho anteriormente se podrá utilizar cualquier tipo de la clase **enteros** como byte, sbyte, short, ushort, int, uint, long o ulong. En el caso de que no se especifique a qué tipo pertenece el compilador le dará el tipo int. También se debe tomar en cuenta que los valores de las constantes están asignados con el signo = y están separadas por comas.

Como habéis visto la forma de acceder al valor numérico de las enumeraciones es especificando entre parentesis a qué tipo pertenecen, en este caso (uint). Después de lo cual especificamos el nombre de la enumeración seguido por un punto que separa al nombre de la constante. En el caso de que deseemos desplegar sólo el nombre de la constante y no su valor, se debe omitir el nombre del tipo como: `segundos.hora` sin (uint) al inicio.

En el caso de que solamente especifiquemos algunos valores de las constantes, el compilador asignará el siguiente valor a la siguiente constante. Así por ejemplo:

```
   enum números
   {
```

```
    uno, //toma el valor de 0
    dos, //toma el valor de 1
    diez = 10, //toma el valor de 10
    once, //toma el valor de 11
  }
```

Operadores

Los operadores son símbolos con los cuales C# tomará una acción. Por ejemplo existen operadores matemáticos para sumar, restar, multiplicar y dividir números. Existen también operadores de comparación que analizará si un valor es igual, mayor o menor que otro y operadores de asignación los cuales asignarán nuevos valores a los objetos o variables. A continuación explicaremos un poco más detalladamente los operadores en C#:

Operadores matemáticos

Casi todos los lenguajes de programación soportan operadores matemáticos. Estos operadores se utilizan para realizar operaciones matemáticas sencillas entre números. Entre estos operadores tenemos los de suma, resta, multiplicación, división y módulo (o residuo): +,-,*,/,%, y se los usa de la siguiente manera:

```
using System;

class operadoresMatematicos
{
     public static void Main()
     {
           int a = 7;
           int b = 4;
           int c = a + b;
           int d = a - b;
           int e = a * b;
           int f = a / b;
           int g = a % b;
           Console.WriteLine ("De los números: {0} y {1} la suma es:
{2}, la resta es:{3}, la multiplicación es: {4}, la división es: {5}
con un residuo de: {6}",a,b,c,d,e,f,g);
     }
}
```

Operadores de asignación

Los operadores de asignación son aquellos que sirven para asignar el valor del objeto o variable de la derecha al objeto o variable de la izquierda. Un ejemplo sencillo de este tipo de operadores es la inicialización de variables. Como habíamos visto, para asignar el valor a una variable simplemente utilizamos el símbolo (u operador) igual =

```
int a = 15; //la variable a tomará el valor de 15.
int b = a = 10; //la variable a y la variable b tomarán el valor de 10.
```

Además de estos operadores de asignación sencillos, existen otros operadores de asignación que realizan operaciones matemáticas antes de asignar el valor a la variable u objeto. Entre ellos tenemos: +=, -=, *=, /=, %=, ++, --. Ejemplos:

```
var += 10; // realiza la operación var = var+10;
var -= 10; // realiza la operación var = var-10;
var *= 10; // realiza la operación var = var*10;
var /= 10; // realiza la operación var = var/10;
var++; //realiza la operacion var = var+1; después de procesar esta línea
++var; //realiza la operación var = var+1; antes de procesar esta línea
var--; //realiza la operacion var = var-1; después de procesar esta línea
--var; //realiza la operación var = var-1; antes de procesar esta línea
```

Operadores de comparación

Estos operadores son múy utiles cuando tenemos que cambiar el flujo de nuestro programa. Con ellos podemos comparar si un objeto o variable es igual (==), no es igual (!=), es mayor o igual (>=), es menor o igual (<=), es mayor (>) o es menor (<) que otro objeto. El resultado de esta comparación es de tipo bool es decir verdadero o falso (true o false). Estos operadores se los usa de la siguiente forma:

```
int a = 10;

int b = 20;

bool resp;


resp = (a == b); // compara si a es igual a b y retorna el valor bool false (o falso), tómese en cuenta que a==b es MUY diferente a a=b

resp = (a != b); // compara si a es diferente a b y retorna el valor bool true (o verdadero)

resp = (a <= b); // compara si a es menor o igual a b y retorna el valor bool true (o verdadero)

resp = (a >= b); // compara si a es mayor o igual a b y retorna el valor bool false (o falso)

resp = (a < b); // compara si a es menor a b y retorna el valor bool true (o verdadero)

resp = (a > b); // compara si a es mayor a b y retorna el valor bool false (o falso)
```

Operadores lógicos

Para entender como funcionan los operadores lógicos tenemos que aprender un poco lo que son los números binarios. En esta parte del libro no cubriremos en detalle este extenso tema de los números binarios ni del Algebra que gobierna estos números ni mucho menos de como se comportan las puertas lógicas dentro de un ordenador porque nos tomaría uno o dos libros completos, pero nos gustaría dar un poco de bases de como es que los números binarios forman parte de los operadores lógicos. Toda información que el ordenador opera internamente es representada por números binarios (por unos y ceros que son conocidos también por verdadero y falso), así la letra A el ordenador internamente lo representa en código binaro ASCII como 01000001 que en números "normales" o decimales es 65. Para manipular esta información en unos y ceros, el ordernador tiene operadores lógicos los cuales permiten cambiar la información de una manera que nos convenga. Por medio de estos operadores lógicos el ordenador es capáz de tomar decisiones, procesar cualquier información, hacer complicadas operaciones matemáticas, o en otras palabras, por medio de estos operadores lógicos, el ordenador hace todo lo que vosotros le habéis visto hacer.

Los operadores lógicos más importantes para nuestro estudio en C# son:

AND

Representado por el simbolo **&**. Comprueba si todos los números binarios son 1 (o verdadero) entonces la respuesta es 1 (o verdadero)

OR

Representado por el simbolo | (barra vertical de la tecla del 1). Comprueba si cualquiera de los números binarios es 1 (o verdadero) entonces la respuesta es 1 (o verdadero)

NOT

Representado por el simbolo ~ y !. Invierte la respuesta. En operaciones con tipos bool, el operador ! cambia la variable de verdadero a falso o viceversa, pero en números binarios, el operador ~ cambia cada uno de los unos y ceros por su opuesto, cuando encuentra un uno lo cambia por un cero y viceversa, así por ejemplo si tenemos el número binario 01000001 y aplicamos el operador NOT ~ obtendremos 10111110, pero si tenemos una expresion que se evalua como **true** (o verdadera) y si se aplica el operador !, se obtiene una respuesta false (o falsa). Por ejemplo (!(10==10)) esta expresión tiene como resultado false

XOR

Representado por el simbolo ^. En dos números, comprueba si los dos números binarios son iguales, entonces la respuesta es 0 (o falso).

<<

Desplazar a la izquierda desplaza todos los bits hacia la izquierda introduciendo ceros al final de la derecha y descartando los últimos números. Asi el número 01000001 si se lo desplaza a la izquierda una vez 01000001 << 1, se convierte en 10000010

>>

Al igual que el operador anterior, desplazar a la derecha desplaza todos los bits hacia la derecha introduciendo ceros al final de la izquierda y descartando los últimos números. Asi el número 01000001 si se lo desplaza a la derecha una vez 01000001 >> 1, se convierte en 00100000

Operadores lógicos de unión

En el caso de que deseemos comparar varios valores para saber si todos son verdaderos o si alguno es verdadero podemos usar los operadores lógicos de unión && y ||

```
a && b // esta línea compara si a y b son verdaderos retorna el valor true (o verdadero) si los dos lo son
a || b // esta línea compara si a o b son verdaderos retorna el valor true (o verdadero) si alguno de los dos es
!a     // esta línea compara si a es verdadero retorna falso si lo es y viceversa.
```

a y b pueden representar variables, constantes, números, funciones, expresiones, etc. que den como resultado un valor de decisión (true o false). Asi por ejemplo, el siguiente ejemplo es válido:

```
int a = 0;
int b = 10;
int c = 20;

if ((a <= b) && (c >= b))
  System.Console.WriteLine ("a es menor o igual a b y c es mayor o igual a b");
```

Fuentes y contribuyentes del artículo

C sharp NET/Capítulo 3 *Fuente*: http://es.wikibooks.org/w/index.php?oldid=185566 *Contribuyentes*: Alanbrito2, Davidcanar, Fseoane, LadyInGrey, MarcoAurelio, Servaq, 118 ediciones anónimas

Licencia