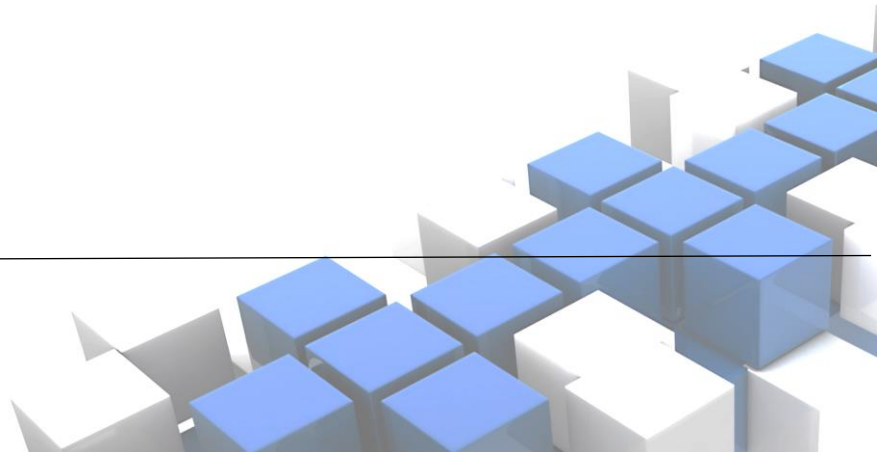


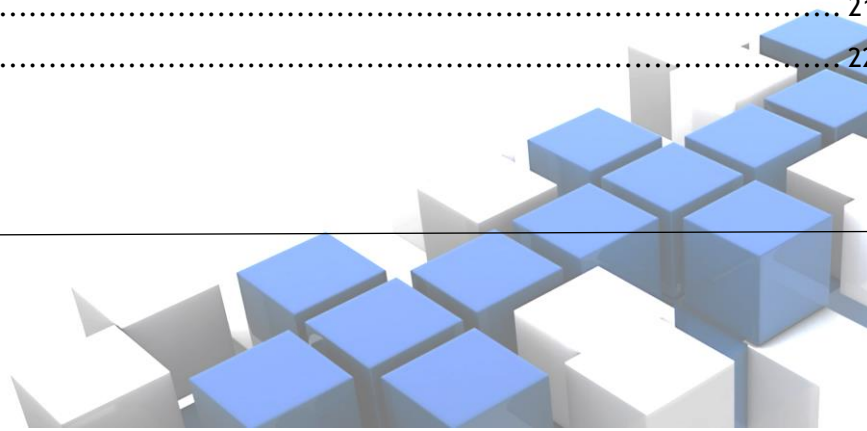
El lenguaje C#

(Revisión rápida y equivalencia con Visual Basic)



Contenido

1. El lenguaje C#	3
1.1 Características:	3
1.2 Estructura de un programa en C#	4
1.3 Tipos de variable. Declaración	6
1.3.1 Tipos básicos, simples o tipos primitivos.	7
1.3.2 Estructuras de datos. Enumerados	9
2 Operadores	12
2.1 Operadores aritméticos.....	12
2.2 Operadores relacionales.....	13
2.3 Operadores lógicos:	14
3 Estructuras condicionales.....	16
3.1 Instrucción if	16
3.2 Instrucción switch	18
4 Estructuras Repetitivas	19
5 Palabras Reservadas C#	21
6 Recursos.....	22



1. El lenguaje C#

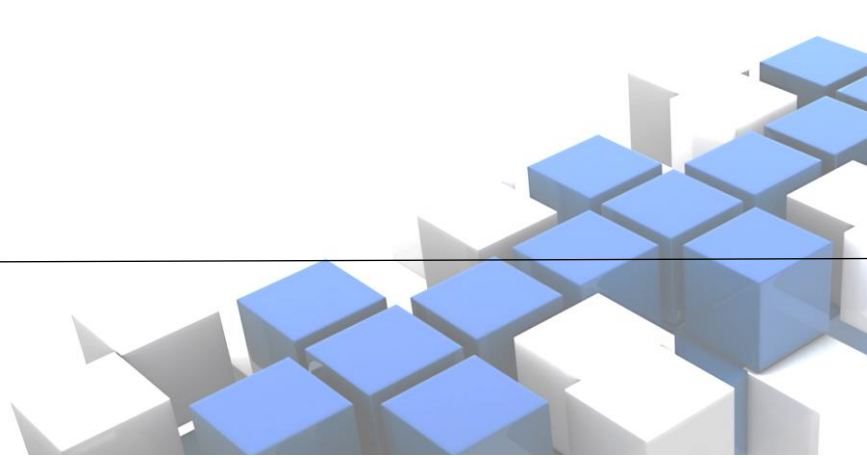
Este presente capítulo está destinado a presentar el lenguaje C# a desarrolladores con experiencia en otros lenguajes que desean desarrollar aplicaciones en este lenguaje y mostrar de una forma sencilla como se reproducen los distintos bloques de programación en su lenguaje habitual en C# (C Sharp).

El lenguaje C# tiene base en el lenguaje C por lo que a aquellos programadores familiarizados o que trabajen con C, C++, Java y JavaScript les resultará muy familiar.

1.1 Características:

El lenguaje C# tiene las siguientes características:

- Orientado a Objetos
- Soporta la encapsulación, herencia y polimorfismo.
- Orientado a componentes
- Recolector de Memoria Libre
- Admite expresiones lambda (que permite crear funciones anónimas)
- Compatible con operaciones asíncronas
- Sistema de tipos unificados (todos los elementos heredan de un único tipo “*Object*” raíz.)
- Admite parámetros por referencia y parámetros valor
- Permite la interoperabilidad entre lenguajes. Permite interactuar con los ensamblados escrito en otros lenguajes.
- Se organiza en espacios de nombres. (librerías)
- Distingue entre mayúsculas y minúsculas

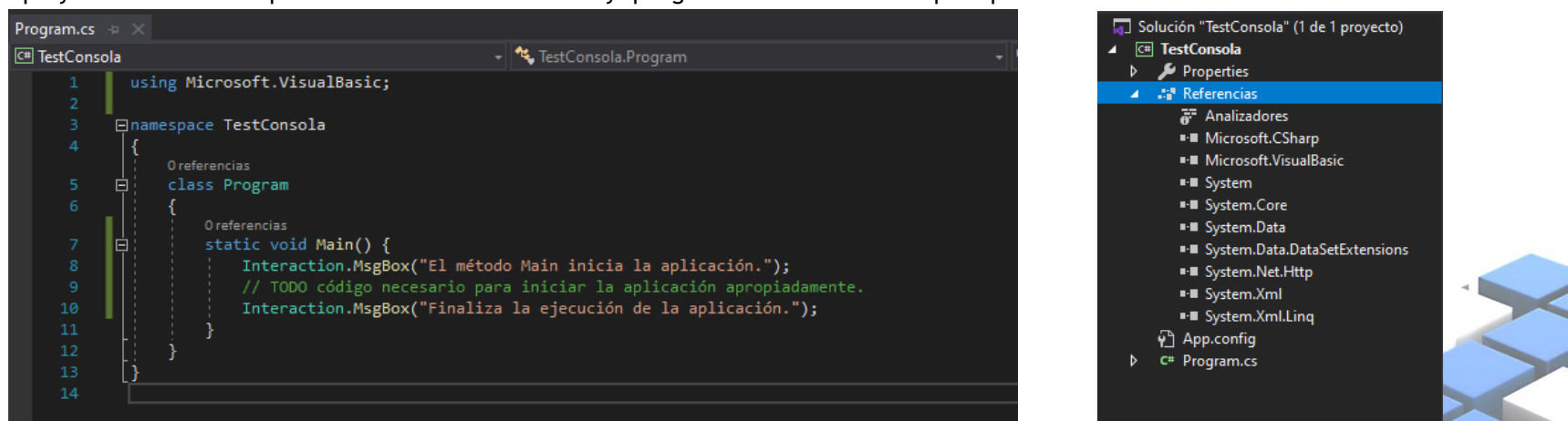


1.2 Estructura de un programa en C#

El presente capítulo tiene por objeto mostrar los paralelismos y diferencias entre un programa escrito en Visual Basic y un programa escrito en C#, como podemos ver en el siguiente ejemplo:

Código Visual Basic	Código C#
<pre>Module mainModule Sub Main () MsgBox("El método Main inicia el programa.") 'TODO código necesario para iniciar la aplicación apropiadamente. MsgBox("Finaliza la ejecución de la aplicación.") End Sub End Module</pre>	<pre>using Microsoft.VisualBasic; namespace TestConsola { static class Program { static void Main() { Interaction.MsgBox("El método Main inicia el programa."); //TODO código necesario para iniciar el programa apropiadamente. Interaction.MsgBox("Finaliza la ejecución del programa."); } } }</pre>

En ambos casos obtendremos el mismo resultado en la ejecución, para probar nuestro ejemplo abriremos Visual Studio 2019 y crearemos un nuevo proyecto de C# .NET que llamaremos “TestConsola” y que guardaremos en una carpeta personal.

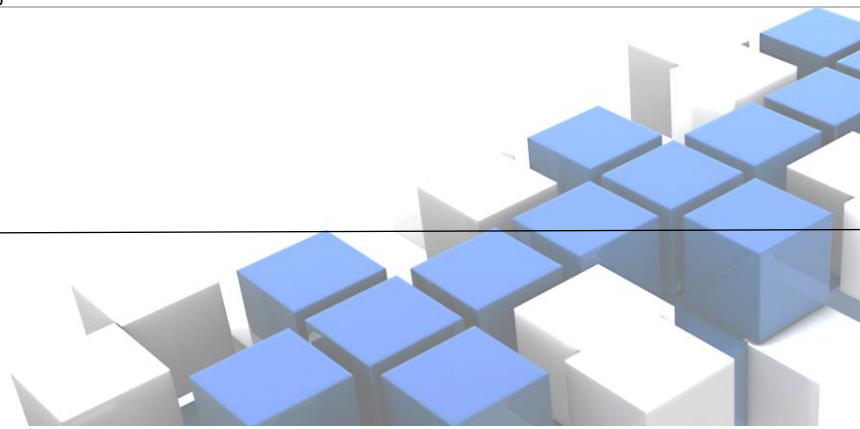


Del presente ejemplo podemos extraer las siguientes conclusiones:

- En C# no existen los módulos, en su lugar existen Clases estáticas
- Los métodos se organizan en nombres de espacios ("TestConsola, VisualBasic, etc)
- Las líneas de código se finalizan mediante el carácter ";".
- No se usan las palabras reservadas de visual basic begin y End en su lugar se abren y cierran llaves "{<instrucciones>}"
- La palabra reservada para importar librerías es "using"
- **C# no "tiene" procedimientos, en su lugar tiene métodos funcionales que no devuelven valor ("void")**

Veamos otro ejemplo. Lo llamaremos TestConsola2

Código Visual Basic	Código C#
<pre>Module mainModule Function Main() As Integer MsgBox("El procedimiento principal está iniciando el programa.") Dim returnValue As Integer = 0 'Inserte la llamada al lugar de inicio apropiado en su código. 'Al regresar, asigne el valor apropiado a returnValue. '0 generalmente significa finalización exitosa. MsgBox("La aplicación terminara con un nivel de error. " & CStr(returnValue) & ".") Return returnValue End Function End Module</pre>	<pre>using Microsoft.VisualBasic; static class mainModule { public static int Main() { Interaction.MsgBox("El procedimiento principal está iniciando el programa."); int returnValue = 0; // Inserte la llamada al lugar de inicio apropiado en su código. // Al regresar, asigne el valor apropiado a returnValue. // 0 generalmente significa finalización exitosa. Interaction.MsgBox("La aplicación terminara con un nivel de error. " + System.Convert.ToString(returnValue) + "."); return returnValue; } }</pre>



1.3 Tipos de variable. Declaración

Como ya hemos visto en el ejemplo anterior la forma de definir una variable es:

<Tipo variable> Nombre [= valor inicial];

Fijémonos en este ejemplo:

Código Visual Basic	Código C#
<pre>Module MainMod public sub main() Dim numero as Integer Dim obj as Object= nothing obj = new MiObjeto() 'TODO hacer cosas con el objeto end sub End Module</pre>	<pre>using System; static class MainMod { public static void main() { int numero; object obj = null; obj = new MiObjeto(); //TODO Hacer cosas con el objeto } }</pre>

Hemos declarado dos variables, la variable “numero” de tipo entero y la variable “obj” de tipo objeto la cual hemos inicializado a valor nulo, hemos de tener en cuenta que en C# las variables no tienen valor inicial por defecto, por lo que cualquier instrucción distinta de la asignación de valor que se efectúe con una variable no inicializada producirá un error en tiempo de compilación. Cuando se declara una variable sin asignación de valor, esta tiene un valor indeterminado.

Para declarar las variables hemos de tener en cuenta las siguientes reglas:

- Los nombres de las variables no pueden comenzar por números, pero si puede formar parte del nombre.
- El nombre de una variable no puede tener espacios.
- El nombre de la variable no puede contener caracteres reservados a los operadores como “+”, “-”, “!”, etc.
- No puede haber dos variables que se llamen igual, aunque sean de distintos tipos, en el mismo ámbito.
- No pueden tener el nombre de palabras reservadas, en eclipse estas palabras se colorean.
- Al distinguir C# entre mayúsculas y minúsculas, una variable A será distinta a otra llamada a.

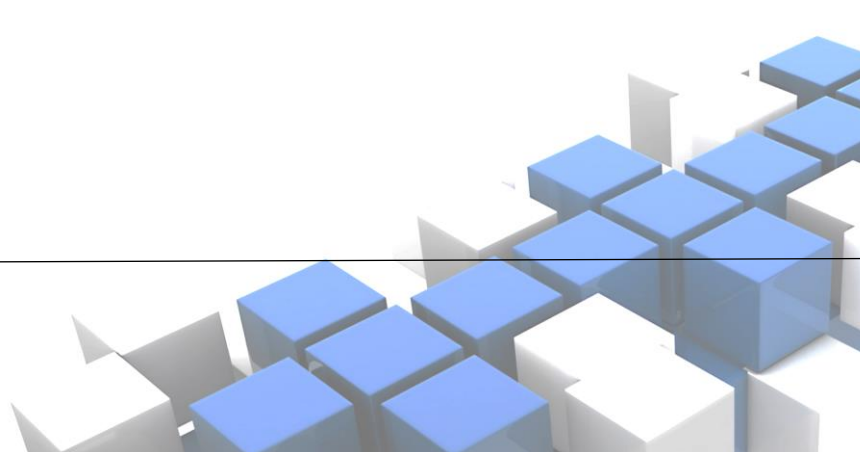


1.3.1 Tipos básicos, simples o tipos primitivos.

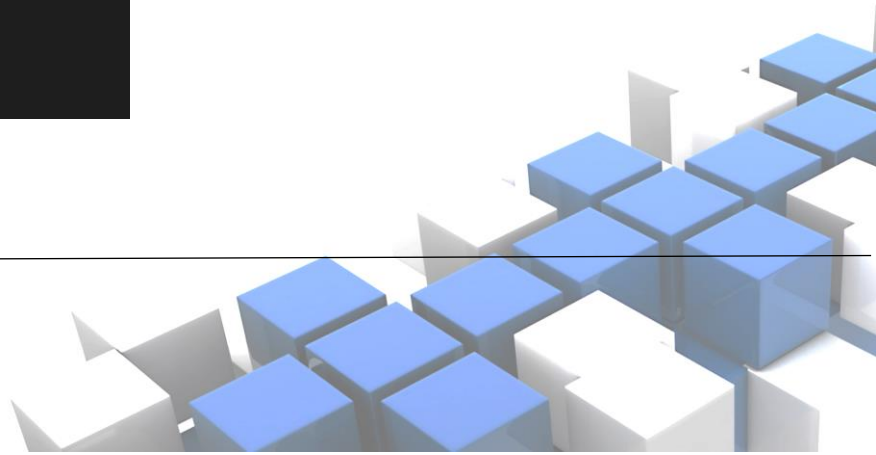
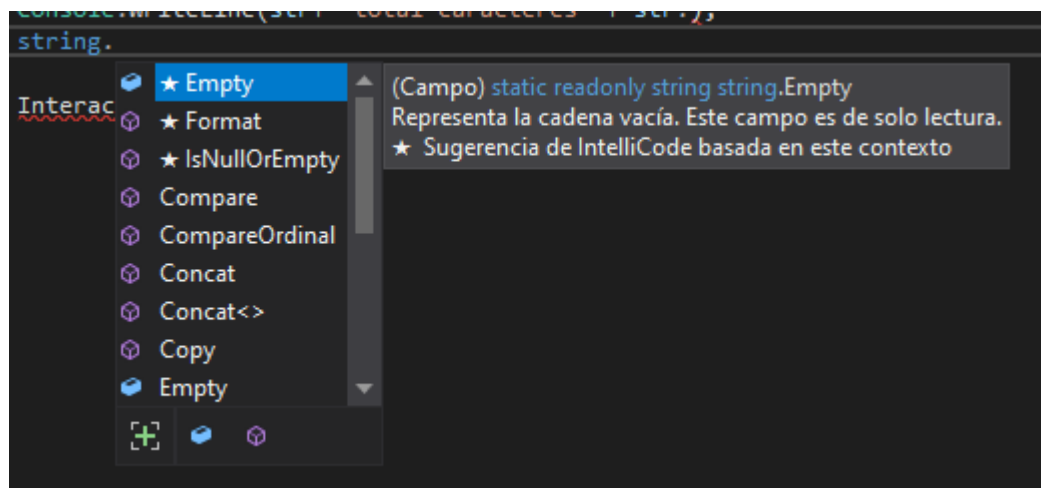
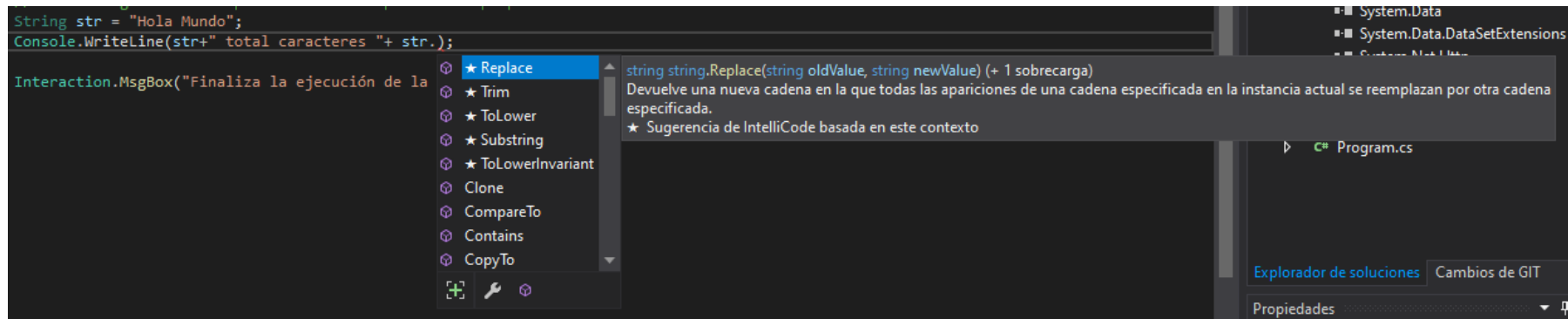
Se denominan de esta manera a los tipos de datos que no necesitan instanciarse (reservar memoria mediante la palabra reservada “new”) y que son comunes por lo general en todos los lenguajes de programación. En el siguiente ejemplo vamos su correspondencia con su declaración en Visual Basic.

Código Visual Basic	Código C#
<pre>Module MainMod Public Sub Main() Dim a As Short Dim b As Integer Dim c As Long Dim d As UShort Dim e As UInteger Dim f As ULong Dim g As String Dim h As Char Dim i As Boolean Dim j As Double Dim k As Decimal Dim l As Single End Sub End Module</pre>	<pre>static class MainMod { public static void Main() { short a; int b; long c; ushort d; uint e; ulong f; string g; char h; bool i; double j; decimal k; float l; } }</pre>

Hay que señalar que el tipo “string”, aunque no es un tipo de simple sino más bien un vector de caracteres, se le incluye en este grupo ya que es una estructura que está presente en todos los lenguajes de programación modernos.



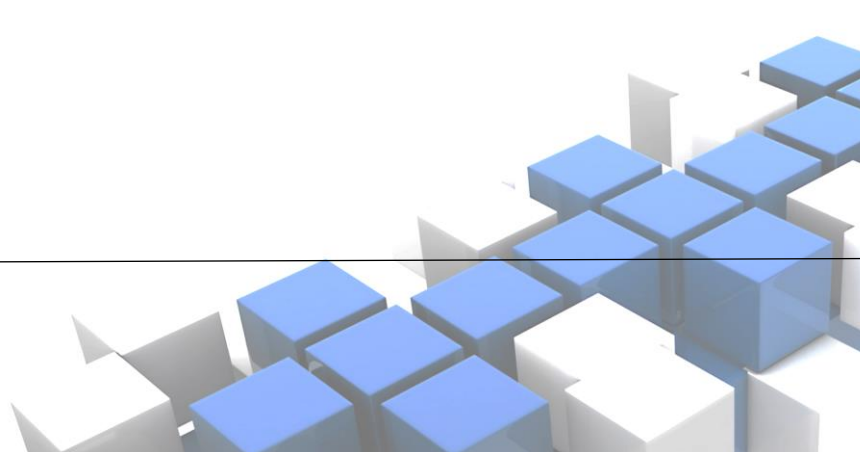
En C# todos los tipos derivan de un tipo común como ya dijimos anteriormente, y los tipos primitivos no van a ser diferentes, cada uno de ellos tiene definido un tipo Objeto con métodos de propósito general que interactúan con el dato y a las que se accede pulsando el carácter “.” junto a la variable o el tipo.



1.3.2 Estructuras de datos. Enumerados

Un *tipo de enumeración* es un [tipo de valor](#) definido por un conjunto de constantes con nombre del tipo [numérico integral](#) subyacente. Para definir un tipo de enumeración, use la palabra clave `enum` y especifique los nombres de `enum`:

Código Visual Basic	Código C#
<pre>Enum DiasSemana lunes martes miercoles jueves viernes sabado domingo End Enum Enum Estaciones primavera = 1 verano = 2 otoño = 3 invierno = 4 End Enum Enum CodigosError as Ushort Ninguno = 0, Desconocido = 1, ConexionPerdida = 100, ErrorLectura = 200 End enum</pre>	<pre>enum DiasSemana { lunes, martes, miercoles, jueves, viernes, sabado, domingo } enum Estaciones { primavera = 1, verano = 2, otoño = 3, invierno = 4 } enum CodigosError : ushort { Ninguno = 0, Desconocido = 1, ConexionPerdida = 100, ErrorLectura = 200 }</pre>



Veamos un ejemplo del uso de tipos enumerados

```
using Microsoft.VisualBasic;
using System;

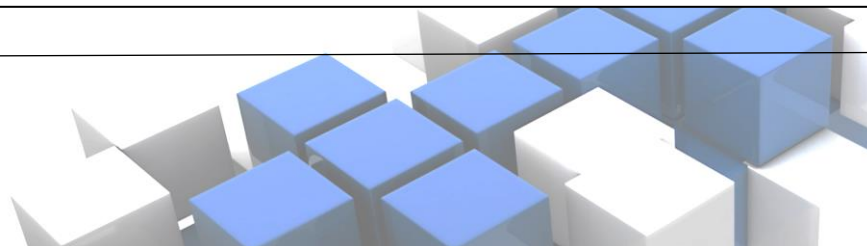
namespace TestConsola
{
    enum DiasSemana
    {
        lunes,
        martes,
        miercoles,
        jueves,
        viernes,
        sabado,
        domingo
    }

    enum Estaciones
    {
        primavera = 1,
        verano = 2,
        otoño = 3,
        invierno = 4
    }

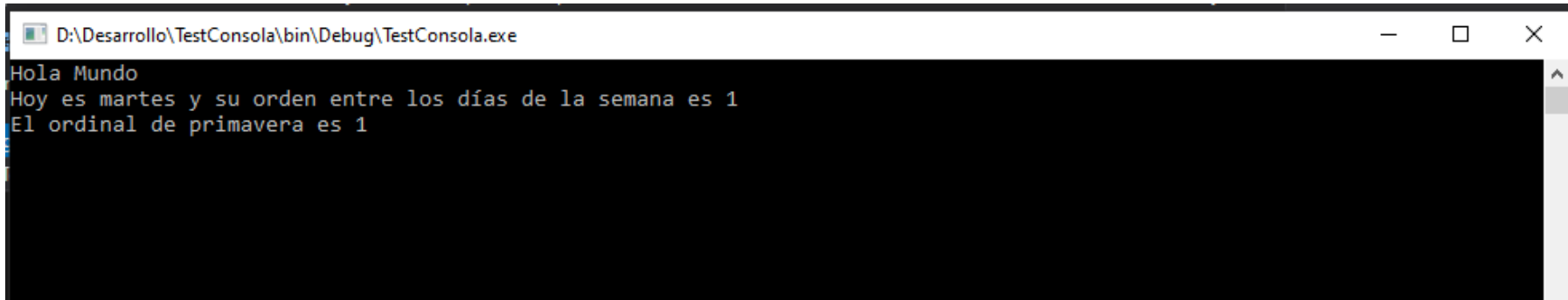
    class Program
    {
        static void Main() {

            Interaction.MsgBox("El método Main inicia la aplicación.");
            // TODO código necesario para iniciar la aplicación apropiadamente.
            string str = "Hola Mundo";
            DiasSemana dia = DiasSemana.martes;
            Estaciones e;
            e = Estaciones.primavera;
            Console.WriteLine(str);
            Console.WriteLine(string.Format("Hoy es {0} y su orden entre los días de la semana es {1}", dia.ToString(), ((short)dia).ToString()));
            Console.WriteLine(string.Format("El ordinal de {0} es {1}", e.ToString(), ((short)e).ToString()));
            Interaction.MsgBox("Finaliza la ejecución de la aplicación.");

        }
    }
}
```



Si ejecutamos la aplicación obtendremos la siguiente salida (obviando los cuadros de texto)

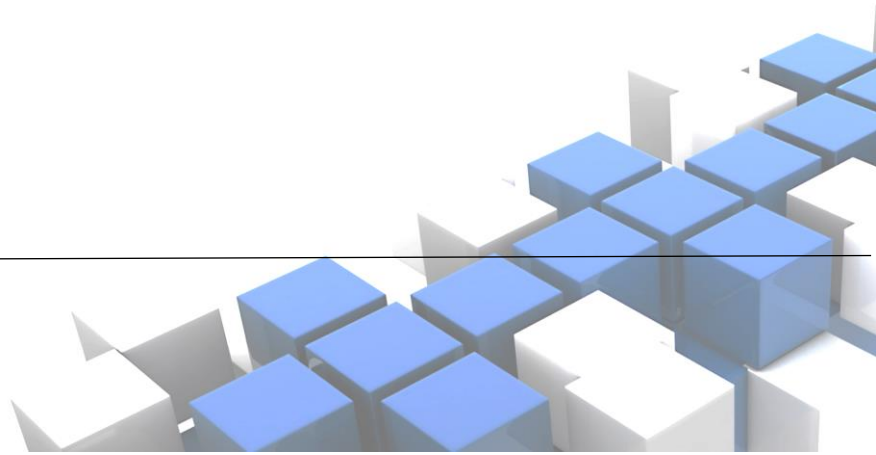


```
D:\Desarrollo\TestConsola\bin\Debug\TestConsola.exe
Hola Mundo
Hoy es martes y su orden entre los días de la semana es 1
El ordinal de primavera es 1
```

Como podemos concluir de acuerdo con la salida, los tipos enumerados inician su orden desde 0, salvo que se especifiquen valores diferente.

Proposición de ejercicio: Crear un proyecto donde definamos un enumerado con los códigos de error establecidos en el ejemplo anterior y mostremos en consola sus valores.

Las estructuras de datos mas complejas, las veremos conjuntamente a las clases.



2 Operadores

C# nos proporciona un conjunto de operadores compatibles, por lo general, con los tipos base del lenguaje, con los que se pueden realizar operaciones básicas con los valores de estas variables. Los operadores se pueden clasificar, entre otros, en los siguientes grupos

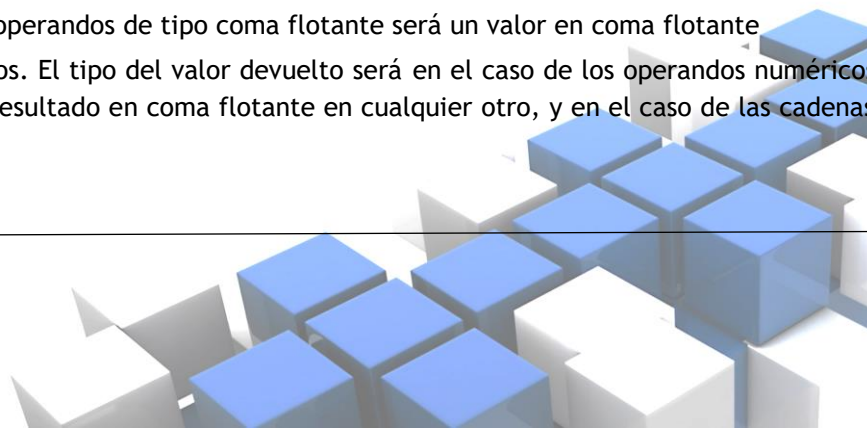
- Operadores aritméticos: Realizan operaciones aritméticas
 - Operadores unarios: incremento, decremento, más y menos
 - Operadores binarios: multiplicación, división, resto, suma y resta
- Operadores relacionales: Comparan operandos numéricos, char y string devolviendo un resultado lógico
- Operadores lógicos: realizan operaciones lógicas.

2.1 Operadores aritméticos

Los operadores aritméticos realizan operaciones con operandos de tipo numérico, ya sean de tipo entero como de tipo de coma flotante. Como caso especial, el operador suma se puede utilizar también para concatenar cadenas.

Los operadores unarios se aplican a un solo operando y son:

- Operador de incremento (++): incrementa su operando en 1. Debe ser una variable, propiedad o indexador numérico.
- Operador de decremento (--): decrementa su operando en 1.
- Operadores unarios más y menos: El operador unario + devuelve el valor de su operando, el operador - calcula la negación numérica del operando.
- Operador de multiplicación (*): calcula el producto de sus operandos
- Operador de división (/): Calcula la división del operador izquierdo entre el derecho, teniendo en cuenta que si ambos operadores son de tipo entero en ese caso el resultado será de tipo entero. Para un resultado en coma flotante utilice los tipos `float`, `double` o `decimal`
- Operador de resto (%): calcula el resto de la división del operador izquierdo con el derecho. En el caso del resto de enteros la operación nos devolverá el siguiente resultado $(a \% b) = a - \frac{a}{b} * b$ el resto de dos operandos de tipo coma flotante será un valor en coma flotante
- Operador de suma (+): devuelve el valor de la suma de los operandos. El tipo del valor devuelto será en el caso de los operandos numéricos de tipo entero si y solo si ambos operandos son enteros, siendo un resultado en coma flotante en cualquier otro, y en el caso de las cadenas de caracteres devolverá la concatenación de dichas cadenas.



- Operador de resta (-): devuelve la diferencia de los dos operandos. Al igual que en el caso anterior el tipo del resultado va a depender del tipo de los operadores, devolviendo únicamente un resultado entero cuando ambos operadores lo son.
- Asignación compuesta (+=, -=, /=, %=): Para un operador binario `op`, una expresión de asignación compuesta con el formato `x op= y` es equivalente a `x = x op y`

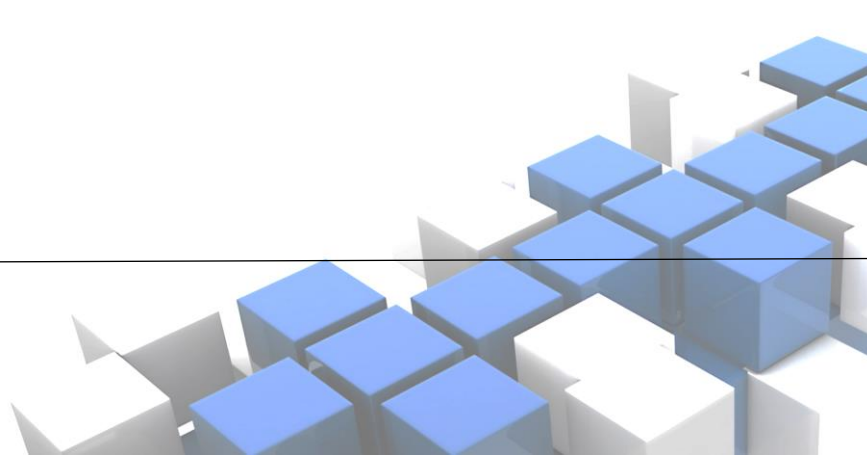
2.2 Operadores relacionales

Los operadores relacionales (menor que “<”), (mayor que “>”), (menor o igual que “<=”) y (mayor o igual que “>=”) comparan sus operandos. Estos operadores se admiten en todos los tipos numéricos enteros y de punto flotante, así como en los tipos `char` y tipo `string`, si bien con los tipos `string` hay que tener cuidado de cómo se realizan estas comparaciones ya que pueden dar resultados inesperados, para evaluar cadenas hay funciones integradas que son más precisas que los operadores relacionales.

Casi todos los operadores relacionales de C# se escriben y funcionan igual que en Visual Basic salvo en el caso de igualdad y la desigualdad. El operador de igualdad en C# es el doble igual (`==`). Devuelve `true` si sus operandos son iguales; en caso contrario, devuelve `false`.

En el caso de las cadenas de caracteres, dos operandos `string` son iguales si ambos son, o bien si las instancias de ambas cadenas tienen la misma longitud y los mismos caracteres en cada posición de caracteres:

El operador de desigualdad es (`!=`) y devuelve `true` si sus operandos son distintos y `false` si son iguales.

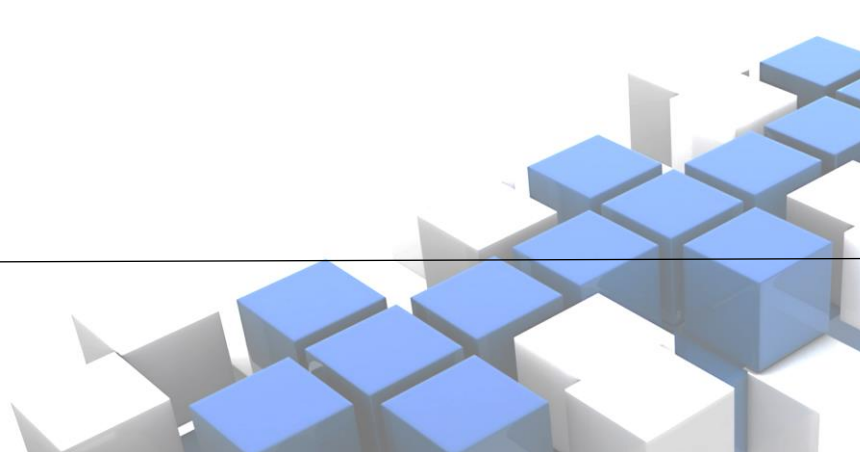


2.3 Operadores lógicos:

Son aquellos operadores que realizan operaciones lógicas con operandos de tipo lógico o booleano. Veamos la comparativa entre un lenguaje y el otro.

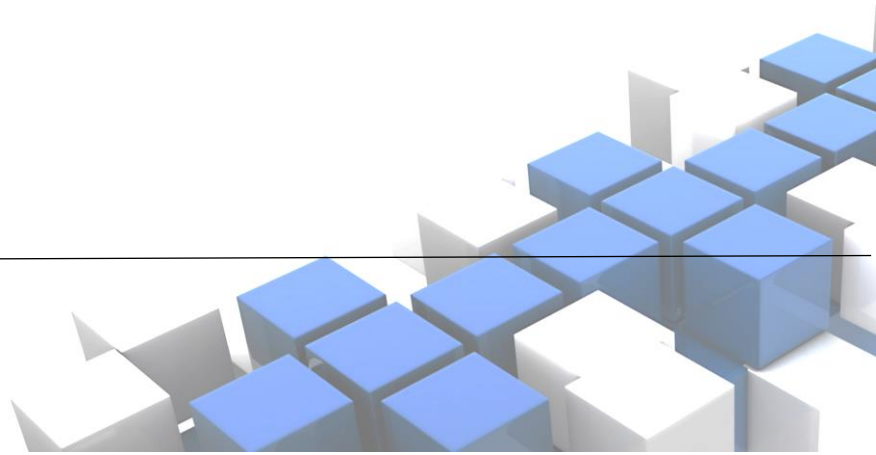
Código Visual Basic	Código C#
<pre>Option Explicit on Module ModMain Public Sub main() Dim i As Integer = 5 Dim resultado As Boolean Console.WriteLine((i >= 0 AndAlso i <= 10)) Console.WriteLine((i >= 0 And i <= 10)) Console.WriteLine((i >= 0 OrElse i <= 10)) Console.WriteLine((i >= 0 Or i <= 10)) Console.WriteLine(Not (i >= 0)) End Sub End Module</pre>	<pre>using System; static class ModMain { public static void main() { int i = 5; bool resultado; Console.WriteLine((i >= 0 && i <= 10)); Console.WriteLine((i >= 0 & i <= 10)); Console.WriteLine((i >= 0 i <= 10)); Console.WriteLine((i >= 0 i <= 10)); Console.WriteLine(!(i >= 0)); } }</pre>

Normalmente en C# vamos a utilizar los operadores de circuito corto ("&&", "||"), es decir, no se evaluará la condición completa cuando ya se sepa el valor del resultado (falso Y lo que sea siempre será falso / verdadero O lo que sea siempre será verdadero).



En el caso de que algunos de los operadores lógicos admitan valores nulos (bool?) la tabla de verdad que se obtiene de la evaluación será la siguiente:

x	y	x&y	x y
true	true	true	true
true	false	false	true
true	null	null	true
false	true	false	true
false	false	false	false
False	null	False	null
null	true	null	true
null	false	False	null
null	null	null	null



3 Estructuras condicionales

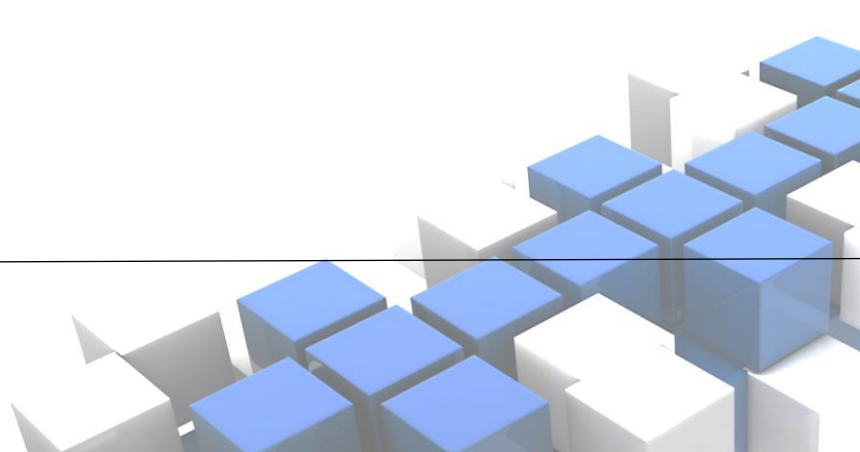
Como ya sabemos, las estructuras condicionales son aquellas que modifican el flujo de la aplicación en función del cumplimiento o no de una condición o conjunto de condiciones. En C# vamos a disponer de dos formas de hacerlo mediante la palabra reservada `if` y la palabra reservada `switch`

3.1 Instrucción if

Veamos las tres formas que puede tener la instrucción if:

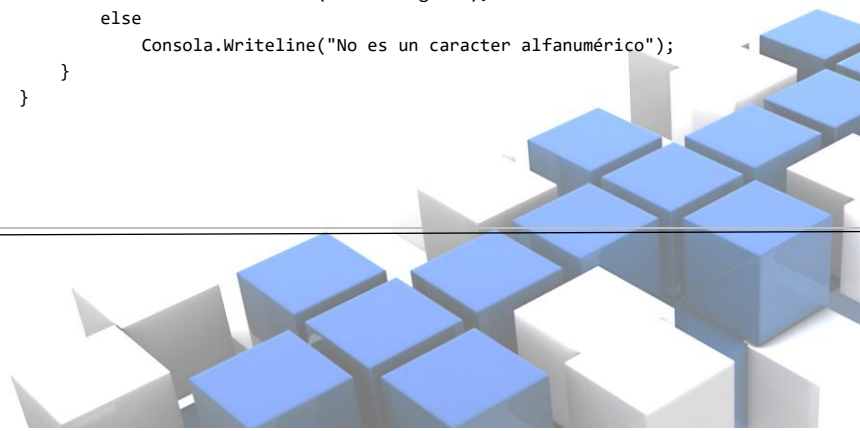
Código Visual Basic	Código C#
<pre>Option Explicit on Module ModMain Public Sub main() Dim temperatura As Single = 20.5 If temperatura <= 0 AndAlso temperatura >= 100 then Console.WriteLine("temperatura incorrecta") End If Consola.WriteLine("La temperatura es " + temperatura) End Sub End Module</pre>	<pre>using System; static class ModMain { public static void main() { float temperatura = 20.5; if (temperatura <= 0 && temperatura >= 100) { Console.WriteLine("temperatura incorrecta"); } Consola.WriteLine("La temperatura es " + temperatura); } }</pre>

En una estructura condicional if cuando se compone de una sola instrucción se pueden omitir los corchetes, por lo general se desaconseja ese uso al menos hasta haber obtenido un buen manejo del lenguaje ya que queda poco legible en caso de necesitar su corrección.



Código Visual Basic	Código C#
<pre>Option Explicit on Module ModMain Public Sub main() Dim temperatura As Single = 20.5 If temperatura <= 0 AndAlso temperatura >= 100 then Console.WriteLine("temperatura incorrecta") Else Consola.WriteLine("La temperatura es " + temperatura) End If End Sub End Module</pre>	<pre>using System; static class ModMain { public static void main() { float temperatura = 20.5; if (temperatura <= 0 && temperatura >= 100) { Console.WriteLine("temperatura incorrecta"); } else { Consola.WriteLine("La temperatura es " + temperatura); } } }</pre>

Código Visual Basic	Código C#
<pre>Option Explicit on Module ModMain Public Sub main() Dim Ch As Char = "T" If Char.IsUpper(ch) then Console.WriteLine("letra mayuscula") Else if Char.IsLower(Ch) then Consola.WriteLine("letra minúscula") Else if Char.IsDigit(Ch) then Consola.WriteLine("es un dígito") Else Consola.WriteLine("No es un caracter alfanumérico") End If End Sub End Module</pre>	<pre>using System; static class ModMain { public static void main() { char Ch = "T"; if (char.IsUpper(ch)) Console.WriteLine("letra mayúscula"); else if (char.IsLower(Ch)) Consola.WriteLine("letra minúscula"); else if (char.IsDigit(Ch)) Consola.WriteLine("es un dígito"); else Consola.WriteLine("No es un caracter alfanumérico"); } }</pre>



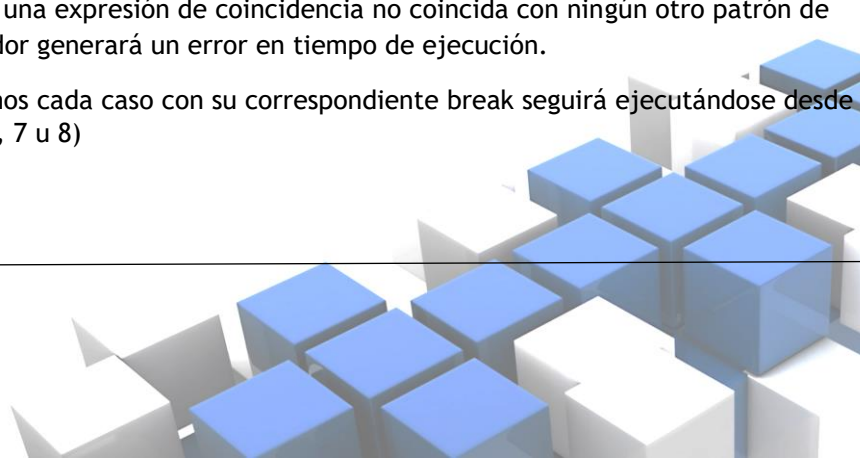
3.2 Instrucción switch

La instrucción `switch` selecciona una lista de instrucciones para ejecutarla en función de la coincidencia de un patrón con una expresión de coincidencia

Código Visual Basic	Código C#
<pre> Option Explicit on Module ModMain Public Sub main() Dim number As Integer = 8 Select Case number Case 1 To 5 Console.WriteLine("Entre 1 y 5, ambos incluidos") Case 6, 7, 8 Console.WriteLine("Entre 6 y 8, ambos incluidos") Case 9 To 10 Console.WriteLine("igual a 9 o 10") Case Else Console.WriteLine("No está incluido entre 1 y 10") End Select End Sub End Module </pre>	<pre> using System; static class ModMain { public static void main() { int number = 8; switch (number) { case number >= 1 && number <= 5: { Console.WriteLine("Entre 1 y 5, ambos incluidos"); break;} case 6: case 7: case 8: { Console.WriteLine("Entre 6 y 8, ambos incluidos"); break;} case number >= 9 && number <= 10: { Console.WriteLine("igual a 9 o 10"); break;} default: { Console.WriteLine("No está incluido entre 1 y 10"); break;} } } } </pre>

El caso `default` especifica las instrucciones que se ejecutarán cuando una expresión de coincidencia no coincida con ningún otro patrón de caso, en el caso de que se produzca algún caso no contemplado, el compilador generará un error en tiempo de ejecución.

La palabra reservada `break` rompe el flujo de la estructura, sino cerramos cada caso con su correspondiente `break` seguirá ejecutándose desde el punto de coincidencia en adelante (como en el caso de que el valor sea 6, 7 u 8)

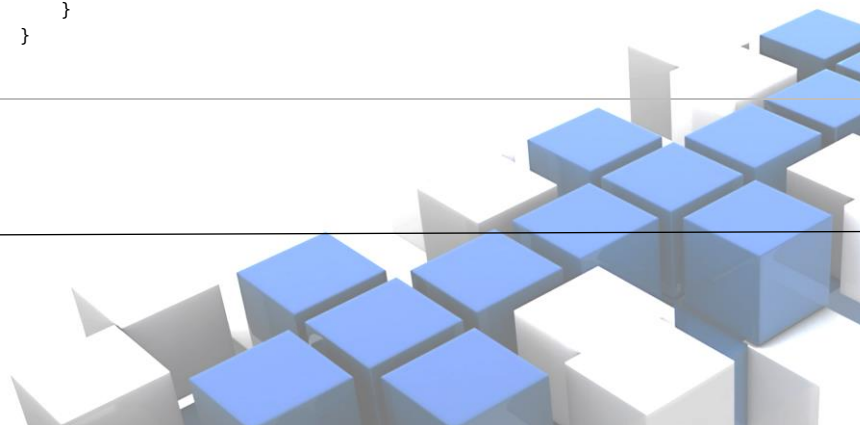


4 Estructuras Repetitivas

Las estructuras repetitivas ejecutan repetidamente una instrucción o bloque de instrucciones siempre que:

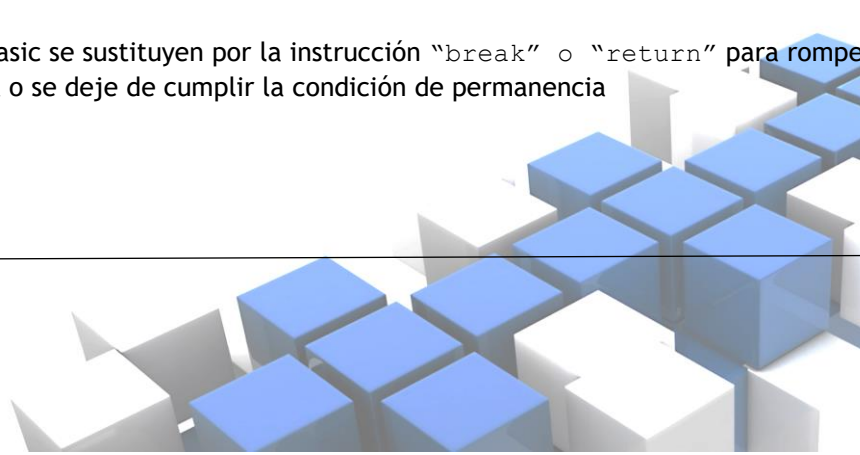
- Una expresión lógica especificada se evalúe como verdadera.
- Una colección de elementos contenga elementos.
- Se ejecute condicionalmente una o varias veces.
- Se ejecute condicionalmente cero o varias veces.

Código Visual Basic	Código C#
<pre> Option Explicit on Module ModMain Public Sub main() Dim i As Integer; Dim lista as new List(Of Integer) from { 0, 1, 1, 2, 3, 5, 8, 13 } for i = 1 to 10 step 1 Console.Write(i) Next For each i in lista Console.Write(i) Next For k as Integer = 0 to lista.length - 1 Console.Write(lista(k)) Next i = 0 while i<=10 Console.Write(i) i=i+1 end while End Sub End Module </pre>	<pre> using System; using System.Collections.Generic; static class ModMain { public static void main() { int i; List<int> lista = new List<int>() { 0, 1, 1, 2, 3, 5, 8, 13 }; for (i = 1; i <= 10; i += 1){ Console.Write(i); } foreach (var j in lista) { Console.Write(j); } for (int k = 0; k < lista.Length(); k++) { Console.Write(lista[k]); } i = 0; while (i <= 10) { Console.Write(i); i++; } } } </pre>



Código Visual Basic	Código C#
<pre> Option Explicit on Module ModMain Public Sub main() Dim i As Integer; Dim lista as new List(Of Integer) from { 0, 1, 1, 2, 3, 5, 8, 13 } i = 0 Do While i<=10 Console.Write(i) i=i+1 Loop i = 0 Do Until i=lista.Length() Console.Write(i) i=i+1 Loop i = 0 Do Console.Write(lista(i)) i=i+1 Loop Until i = Lista.Length() i = 0 Do Console.Write(lista(i)) i=i+1 Loop While i < Lista.Length() End Sub End Module </pre>	<pre> using System; using System.Collections.Generic; static class ModMain { public static void main() { int i; List<int> lista = new List<int>() { 0, 1, 1, 2, 3, 5, 8, 13 }; i = 0; while (i <= 10) { Console.Write(i); i++; } i = 0; while (i != Lista.Length()) { Console.Write(lista[i]); i++; } i = 0; do { Console.Write(lista[i]); i++; } while (i != Lista.Length()); i = 0; do { Console.Write(lista[i]); i = i + 1; } while (i < Lista.Length()); } } </pre>

Las instrucciones “Exit For”, “Exit Do”, “Exit While” de Visual Basic se sustituyen por la instrucción “break” o “return” para romper el bucle antes de que finalice la iteración o se cumpla la condición de salida o se deje de cumplir la condición de permanencia

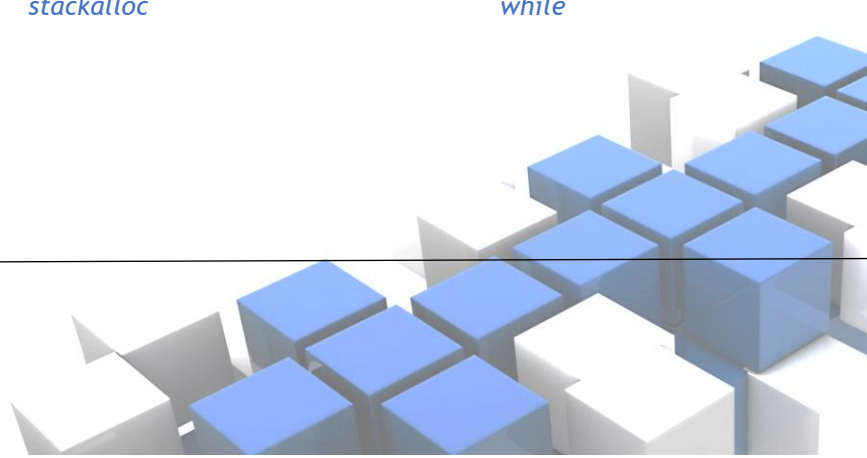


5 Palabras Reservadas C#

Como ya sabemos las palabras reservadas son identificadores predefinidos que tienen un significado especial para el compilador, por lo que nunca se pueden utilizar como identificadores de variables, métodos, propiedades, clases, o cualquier otra definición de usuarios.

La tabla adjunta está vinculada a la documentación sitio web de Microsoft para que podáis consultar cada una de ellas fácilmente.

<i>abstract</i>	<i>event</i>	<i>namespace</i>	<i>static</i>
<i>as</i>	<i>explicit</i>	<i>new</i>	<i>string</i>
<i>base</i>	<i>extern</i>	<i>null</i>	<i>struct</i>
<i>bool</i>	<i>false</i>	<i>object</i>	<i>switch</i>
<i>break</i>	<i>finally</i>	<i>operator</i>	<i>this</i>
<i>byte</i>	<i>fixed</i>	<i>out</i>	<i>throw</i>
<i>case</i>	<i>float</i>	<i>override</i>	<i>true</i>
<i>catch</i>	<i>for</i>	<i>params</i>	<i>try</i>
<i>char</i>	<i>foreach</i>	<i>private</i>	<i>typeof</i>
<i>checked</i>	<i>goto</i>	<i>protected</i>	<i>uint</i>
<i>class</i>	<i>if</i>	<i>public</i>	<i>ulong</i>
<i>const</i>	<i>implicit</i>	<i>readonly</i>	<i>unchecked</i>
<i>continue</i>	<i>in</i>	<i>ref</i>	<i>unsafe</i>
<i>decimal</i>	<i>int</i>	<i>return</i>	<i>ushort</i>
<i>default</i>	<i>interface</i>	<i>sbyte</i>	<i>using</i>
<i>delegate</i>	<i>internal</i>	<i>sealed</i>	<i>virtual</i>
<i>do</i>	<i>is</i>	<i>short</i>	<i>void</i>
<i>double</i>	<i>lock</i>	<i>sizeof</i>	<i>volatile</i>
<i>else</i>	<i>long</i>	<i>stackalloc</i>	<i>while</i>
<i>enum</i>			



6 Recursos

En el siguiente apartado iremos incluyendo recursos útiles para el desarrollo del curso y para la futura realización de aplicaciones

- *Conversor de Código C# a VB y viceversa*
- *Microsoft Visual Basic documentación en línea.*
- *Microsoft C# documentación en línea*
- *Utilidad de compartición de código*

