Elementos Básicos del Lenguaje C#

Prof. Victor Manuel Cordero

Elementos básicos del Lenguaje C#

C# (pronunciado "si sharp" en inglés) es un lenguaje de programación moderno, orientado a objetos y con seguridad de tipos **(*type-safe*)**. C# tiene sus raíces en la familia de lenguajes *C* por lo que resultará inmediatamente familiar a los desarrolladores de *C*, *C++*, *Java* y *JavaScript*.

Sintaxis

- Una de las primeras cosas que hay que tener en cuenta es que C# es un lenguaje con seguridad de tipos (type-safe).

- C# es sensitivo a mayúsculas/minúsculas. El compilador hace distinción entre mayúsculas y minúsculas, esta distinción no solo se aplica a palabras reservadas del lenguaje sino también a **nombres de variables, clases y métodos.**

- La sintaxis de C# es muy parecida a la de Java, C y C++, por ejemplo, las sentencias finalizan con **";"**, los bloques de instrucciones se delimitan con llaves **{ }**, etc.

Estructura de una Aplicación en C#

Los principales conceptos organizativos en C# son:

- ***Programas, Espacios de nombres, Tipos, Miembros, Ensamblados.***

✓ Las aplicaciones se componen de uno o más archivos de código.

✓ Declaran tipos, que contienen miembros y se organizan en espacios de nombres (namespace).

✓ Las clases e interfaces son ejemplos de tipos.

✓ Los campos, los métodos, las propiedades y los eventos son ejemplos de miembros.

✓ Cuando se compilan aplicaciones en C#, se empaquetan físicamente en ensamblados. Normalmente, los ensamblados tienen la extensión de archivo .exe o .dll, dependiendo de si implementan ***aplicaciones*** o ***bibliotecas***, espectivamente.

Elementos básicos del Lenguaje C#

```
using System;
namespace Acme.Collections
{
    public class Stack
    {
        Entry top;
        public void Push(object data)
        {
            top = new Entry(top, data);
        }

        public object Pop()
        {
            if (top == null)
            {
                throw new InvalidOperationException();
            }
            object result = top.data;
            top = top.next;
            return result;
        }
    }

    class Entry
        {
            public Entry next;
            public object data;
            public Entry(Entry next, object data)
            {
                this.next = next;
                this.data = data;
            }
    }
}
```

Espacio de nombres (namespace)

Nombre de Clase (class)
Acme.Collections.Stack (Nombre completo)

La clase contiene varios elementos **(members)**:

- Un campo denominado **top**
- Dos métodos denominados **Push y Pop**
- Una clase anidada denominada **Entry**.

class **Entry** contiene:
- un campo denominado next
- un campo denominado data
- Constructor

Suponiendo que el código fuente se almacene en el archivo **acme.cs** en algún directorio del computador.

https://docs.microsoft.com/es-es/dotnet/csharp/tour-of-csharp/program-structure

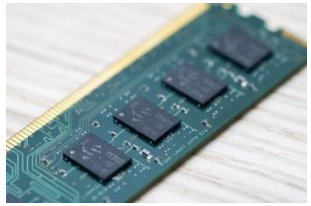
Tipos de datos en C#

Una variable contiene datos de un tipo específico. Cuando declaramos una variable para almacenar datos en una aplicación, debemos elegir un tipo de dato adecuado para esos datos. C# es un lenguaje de tipos seguros "**Type-Safe**", esto significa que el compilador garantiza que los valores almacenados en las variables siempre son del tipo apropiado.

- Existen dos clases de tipos de datos en C#: ***tipos valor* y *tipos referencia*.**

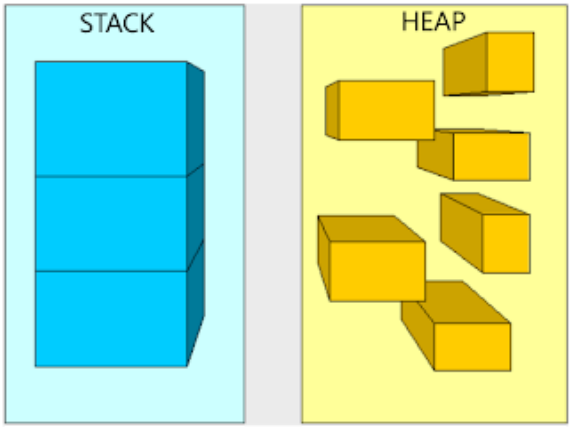
- **Tipos valor,** son variables contienen directamente los datos almacenados reservando un espacio de memoria en la pila (stack).
- **Tipos referencia,** son variables que almacenan referencias (en el stack) a los datos reales. Se almacenan en el monton (heap).

La memoria



Básicamente en la ejecución de una aplicación en .NET, la memoria se divide en dos bloques: **la pila (stack) y el montón (heap)**.

- Tanto el Stack como el Heap actúan en la ejecución del programa y residen en la memoria operativa de la máquina.



- La pila es una estructura en la que los elementos se apilan de modo que el último elemento en entrar es el primero en salir (estructura LIFO, o sea, Last In First Out).
- El montón es un bloque de memoria en el cual no se reserva en un orden determinado como en la pila, sino que se reserva aleatoriamente según se va necesitando. Cuando el programa requiere un bloque del montón, este se sustrae y se retorna un puntero (variable que contiene direcciones de memoria).

Tipos de datos más comunes utilizados en C#

Tipo	Descripción	Tamaño	Rango
int	Números enteros	4 bytes	De -2.147.483.648 a 2.147.483.647
long	Números enteros	8 bytes	De -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
float	Números de punto flotante	4 bytes	De ±1,5 x 10^{-45} a ±3,4 x 10^{38}
double	Números de punto flotante de doble precisión (más precisos).	8 bytes	De ±5,0 × 10^{-324} a ±1,7 × 10^{308}
decimal	Valores de moneda	16 bytes	De ±1,0 x 10^{-28} to ±7,9228 x 10^{28}
char	Un simple carácter Unicode.	16 bits	U+0000 a U+FFFF
bool	Valor booleano	32 bits	True, false
DateTime	Momentos en el tiempo	8 bytes	0:00:00 del 01/01/0001 a 23:59:59 del 12/31/9999
string	Secuencia de caracteres	2 bytes por carácter	N/A

https://learn.microsoft.com/es-es/dotnet/csharp/language-reference/language-specification/types

Sistema común de Tipos en C#

Tipos de Valor

Simples

- Entero con signo: `sbyte, short, int,long`
- Entero sin signo: `byte, ushort, uint, ulong`
- Caracteres Unicode: `char`
- Punto flotante de IEEE: `float, double`
- Decimal de alta precisión: `decimal`
- Booleano: `bool`

Enumeración (Enum)

- Definidos por el usuario con el formato `enum E {...}`

Estructura (struct)

- Tipos definidos por el usuario con el formato `struct S {...}`

Valores NULL

- Extensiones de todos los demás tipos con valor `null`

Sistema común de Tipos en C#

Tipos de Referencia

Clase (class)

- Clase base de todos los demás tipos: `object`
- Tipos definidos por el usuario con el formato `class C {...}`

Interfaz (Interface)

- Definidos por el usuario con el formato `interface I {...}`

Matriz (Arrays)

- Unidimensional y multidimensional; ej, `int[] y int[,]`

Delegados (Delegate)

- Definidos por el usuario con el formato `delegate int D(...)`

https://docs.microsoft.com/es-es/dotnet/csharp/tour-of-csharp/types-and-variables

Expresiones y operadores

Las expresiones son el componente principal de cualquier aplicación C# debido a que estas son construcciones fundamentales que utilizamos para evaluar y manipular datos. Las expresiones son colecciones de operandos y operadores que podemos definir de la siguiente manera:

- Los ***operandos*** son valores por ejemplo números y cadenas. Pueden ser valores constantes (literales), variables, propiedades o valores devueltos por las llamadas a métodos.
- Los ***operadores*** definen operaciones a realizar sobre los operandos, por ejemplo, la suma o la multiplicación, así como para operaciones más avanzadas tales como comparaciones lógicas o la manipulación de datos que constituyen un valor.

No existe un límite en la longitud de las expresiones en las aplicaciones C#, aunque en la práctica estamos limitados por la memoria de la computadora y nuestra paciencia al escribir código. La recomendación es utilizar expresiones cortas. Esto nos facilita ver lo que el código está realizando y facilita también la depuración.

Instrucciones (Statements)

C# contiene en su estructura varios tipos de instrucciones diferentes:

Declaración se usan para declarar variables locales y constantes.

Expresión se usan para evaluar expresiones.

- Invocaciones a métodos
- Creación de objetos con new
- Asignaciones mediante =
- Operaciones de incremento(++) y decremento(--)

Selección para seleccionar una de varias instrucciones posibles (if y switch).

Iteración para ejecutar una instrucción de forma repetida (while, do, for y foreach).

Instrucciones de salto **break, continue, goto, throw, return y yield**

try... catch se usa para detectar excepciones

try... finally se usa para especificar el código de finalización que siempre se ejecuta, tanto si se ha producido una excepción como si no.

using se usa para obtener un recurso, ejecutar una instrucción y, luego, eliminar dicho recurso.

Operadores en C#

Los operadores se combinan con los operandos para formar expresiones. C# proporciona una amplia gama de operadores que podemos utilizar para realizar las operaciones matemáticas y lógicas fundamentales más comunes. Los operadores caen dentro de una de las siguientes tres categorías:

• **Unario**. Este tipo de operador, opera sobre un solo operando. Por ejemplo, podemos utilizar el operador **–** como un operador unario. Para hacer esto, lo colocamos inmediatamente antes de un operando numérico y el operador convierte el valor del operando a su valor actual multiplicado por **-1**.

• **Binario**. Este tipo de operador opera sobre 2 valores. Este es el tipo más común de operador, por ejemplo, *****, el cual multiplica el valor de dos operandos.

• **Ternario**. Existe un solo operador ternario en C#. Este es el operador **? :** que es utilizado en expresiones condicionales.

Operadores en C#

Tipo	Operadores
Aritméticos	+, -, *, /, %
Incremento, decremento	++, --
Comparación	==, !=, <, >, <=, >=, is
Concatenación de cadenas	+
Operaciones lógicas de bits	&, |, ^, !, ~, &&, ||
Indizado (el contador inicia en el elemento 0)	[ ]
Conversiones	( ), as
Asignación	=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=
Rotación de Bits	<<, >>
Información de Tipos de datos	sizeof, typeof
Concatenación y eliminación de Delegados	+, -
Control de excepción de Overflow	checked, unchecked
Apuntadores y direccionamiento en código No seguro (Unsafe code)	*, ->, [ ], &
Condicional (operador ternario)	?:

https://learn.microsoft.com/es-es/dotnet/csharp/language-reference/operators/?WT.mc_id=DT-MVP-21053

Declaración y asignación de Variables/Constantes

Antes de utilizar una variable, debemos declararla. Al declararla podemos especificar su nombre y características. El nombre de la variable es referido como un ***Identificador***.

```
int a;
int b = 2, c = 3;
a = 1;
```

```
const float PI = 3.1415927f;
const int R = 25;
```

```
static void Expressions(string[] args)
{
    int i;
    i = 123;                // Expression statement
    Console.WriteLine(i);   // Expression statement
    i++;                    // Expression statement
    Console.WriteLine(i);   // Expression statement
}
```

Al declarar una variable, debemos elegir un nombre que tenga significado respecto a lo que almacena, de esta forma, el código será más fácil de entender. Debemos también adoptar una convención de nombres y utilizarla.

https://learn.microsoft.com/es-es/dotnet/standard/design-guidelines/naming-guidelines?redirectedfrom=MSDN

Identificadores - Nomenclatura

El término técnico para nombrar alguna variable en un lenguaje de programación, es un identificador. En el lenguaje C# un identificador:

- Puede contener solo letras, digitos del 0 al 9, y el caracter "_".
- El primer carácter de un identificador no puede ser un digito.
- No hay un límite en lo concerniente al número de caracteres que pueden tener los identificadores.
- C# es **case sensitive**
  - ***PersonName*** *y* ***personname*** son 2 indentificadores diferentes.

Identificador	Válido?
outputStream	Si
4you	No
my.work	No
FirstName	Si
_tmp	Si
public	No

Elementos básicos del Lenguaje C#

Las reglas del lenguaje respecto a los nombres de variables son muy amplias y permiten mucha libertad al programador, pero es habitual seguir ciertas normas que facilitan la lectura y el mantenimiento de las aplicaciones en C#.

- Las variables deberían identificarse con letras en minúscula.
- Los nombres de las clases, interfaces y métodos deberían comenzar con mayúsculas. (**Persona, Producto, Figura, Imprimir(), Validar()**).
- Cuando un nombre de variable o método está compuesto de varias palabras se coloca una al lado de la otra, comenzando en mayúscula la primera letra de la palabra, por ejemplo: **BuscarMayor(), NombreApellido**.
- Los nombres de las constantes se escriben en mayúsculas. Ej. **PI, TASA_IVA**.

https://learn.microsoft.com/es-es/dotnet/csharp/fundamentals/coding-style/coding-conventions

Definición automática de tipos en C#

En C# es posible **dejar al compilador la tarea de determinar el tipo de una variable** que hayamos inicializado de manera explícita. El tipo se deduce a partir de la expresión de inicialización. De este modo nos evitamos tener que declarar el tipo correcto, sobre todo en casos en los que no lo tenemos claro al momento.

```
var n = 5; // equivale a int n = 5;
var s = "Hola"; // equivale a string s = "Hola";
```

Por supuesto, se mantienen en vigor las reglas del lenguaje para determinar el tipo de las expresiones:

```
var m = 3 * n; //m es de tipo int
var dosPi = 2 * Math.PI; //dosPi es de tipo double
```

Definición automática de tipos en C#

El mecanismo también funciona para otros tipos.

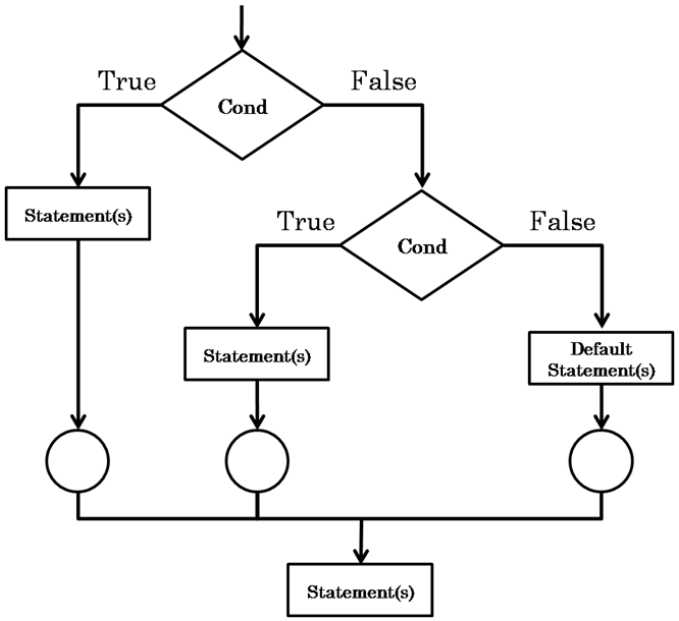
```
var p1 = new Persona("Ana", 24); //p1 es de tipo Persona
var p2 = p1; //p2 también
var enteros = new List(); //enteros es de tipo List
var f1 = DateTime.Now.AddMonths(1); //f1 es de tipo DateTime
```

Los siguientes son ejemplos de usos incorrectos de **var**:

```
var p; //la inicialización es obligatoria
var q = null; //imposible inferir el tipo de q
var z1 = 3, z2 = 5; //no se admiten declaraciones multiples
var r = { 1, 2, 3 }; //un inicializador de array no está permitido
```

Esta característica no tiene nada que ver con la ofrecida en algunos lenguajes dinámicos, donde una variable puede almacenar valores de distintos tipos a lo largo de su tiempo de vida. En C# el compilador simplemente nos libera de la necesidad de especificar el tipo de dato.

Control del flujo de las aplicaciones



Sentencia if

```
if(expresión lógica/relacional){

   Acción a ejecutar;

}
```

```
if (n%d == 0)
   Console.WriteLine(n +" es divisible por "+d);
```

```
static void IfStatement(string[] args)
{
    if (args.Length == 0)
    {
        Console.WriteLine("No arguments");
    }
    else
    {
        Console.WriteLine("One or more arguments");
    }
}
```

Sentencias `if {} else {}`

Las sentencias ***if*** pueden tener clausulas ***else*** asociadas. El bloque ***else*** se ejecuta cuando la expresión de la sentencia ***if*** *no se cumple*.

```
if(expresión lógica/relacional){
   Acción a ejecutar 1;
} else {
   Acción a ejecutar 2;
}
```

```
int Nota;
if (Nota >= 5)
  Console.WriteLine("Aprobado");
else
  Console.WriteLine("Suspenso");
```

```
if(edad>=18){
  Console.WriteLine(“Es mayor de edad”);
} else {
  Console.WriteLine(“Es menor de edad”);
}
```

```
int maximum;
if (value1 < value2) {
  maximum = value2;
}
else {
  maximum = value1;
}
```

Sentencias `if {} else {}` anidadas

Las sentencias ***if*** también pueden tener asociadas sentencias ***else if***. Estas sentencias son evaluadas en el orden en el que aparecen en el código después de la sentencia ***if***. Si una de las cláusulas devuelve ***true***, el bloque de código asociado con esa cláusula es ejecutado y la ejecución deja el bloque entero ***if***.

```
if(expresión lógica/relacional 1){
   Bloque a ejecutar 1;
} else if (expresión lógica/relacional 2){
   Bloque a ejecutar 2;
} else if (expresión lógica/relacional 3){
   Bloque a ejecutar 3;
} else{
   Bloque a ejecutar 4;
}
```

```
if (a == '0')
   Console.WriteLine("zero");
else if (a == '1')
   Console.WriteLine("one");
else if (a == '2')
   Console.WriteLine("two");
else if (a == '3')
   Console.WriteLine("three");
else if (a == '4')
   Console.WriteLine("four");
else
   Console.WriteLine("five+");
```

Instrucción `switch {}`

Es una alternativa de la estructura **IF ELSE IF ELSE** cuando se compara la misma expresión con distintos valores. Su sintaxis es la siguiente:

```
switch(expresion){
  case valor1:
    acción 1;
    break;
  case valor2:
    acción 2;
    break;
  case valor3:
    acción 3;
    break;
  default:
    acción 4;
}
```

```
string Message;
switch (Number)
{
     case 1:
          Message = “Selecciono la opción Insert";
     break;
          case 2:
          Message = “Selecciono la opción Delete";
     break;
     case 3:
          Message = “Selecciono la opción Ver";
     break;
     default:
          Message = “Seleccione una opción correcta";
     break;
}
Console.WriteLine(Message);
```

Podemos ver que en cada sentencia ***case***, tenemos una sentencia ***break***. Esto causa que el control salte hacia el final del ***switch*** después de procesar el bloque de código. Veamos también que existe un bloque llamado ***default***, este bloque de código será ejecutado cuando ninguno de los bloques ***case*** cumpla la condición.

Instrucción `switch {}`

```
string commandName = "start";
switch (commandName) {
     case "start":
     Console.WriteLine("Starting service...");
     StartService();
     break;
     case "stop":
     Console.WriteLine("Stopping service...");
     StopService();
     break;
     default:
     Console.WriteLine(String.Format("Unknown
     command: {0}", commandName));
     break;
}
```

```
int i = 1;
switch (i) {
     case 1:
     case 2:
     Console.WriteLine("One or Two");
     break;
     default: Console.WriteLine("Other");
     break;
}
```

```
public enum State { Active = 1, Inactive = 2 }
State state = State.Active;
switch (state) {
     case State.Active: Console.WriteLine("Active");
     break;
     case State.Inactive: Console.WriteLine("Inactive");
     break;
     default: throw new Exception(String.Format("Unknown state: {0}", state));
}
```

Operador ternario ? (condicional)

```
if (n1 > n2)
  max = n1;
else
  max = n2;
```

Lo podemos escribir así:

```
max = (n1 > n2) ? n1 : n2;
```

Ciclo for {}

Estructura que ejecuta una instrucción un número específico de veces. Cuenta con un valor inicial y final que es el que va a determinar la terminación del ciclo. Su sintaxis es la siguiente:

```
for (inicialización;condición;incremento){
  Instrucción a ejecutar;
}
```

```
for (int n = 0; n < 100; n += 20)
Console.WriteLine("\t" + n + "\t" + n * n );
```

```
for (int j=1;j<4;j++){
   Console.WriteLine("valor de j: "+j);
}
```

```
for (int c = 'A'; c <= 'Z'; c++)
Console.WriteLine(c + " ");
```

```
for (int n = 10; n > 5; n--)
 Console.WriteLine("\t" + n + "\t" + n * n );
```

Normalmente usamos el ciclo **for** cuando conocemos el número de repeticiones y podemos controlarlo usando un contador.

Ciclo `foreach {}`

Aunque un ciclo **for** es fácil de utilizar, no en todos los casos podría ser la mejor solución. Por ejemplo, cuando iteramos sobre una colección o un arreglo, necesitamos conocer cuántos elementos contiene la colección o arreglo. En muchos casos esto es muy sencillo, pero en algunas ocasiones podríamos equivocarnos fácilmente. Por lo tanto, algunas veces es mejor utilizar un ciclo **foreach**.

El ciclo **foreach**, nos permite recorrer los elementos de una colección sin necesidad de conocer cuántos elementos tiene dicha colección.

```
foreach(iterador in coleccion){
  //Instrucción a ejecutar;
}
```

```
foreach (String name in classList)
{
   Console.WriteLine(name);
}
```

```
foreach (int elemento in m)
Console.WriteLine(elemento);
```

Ciclo `while {}`

El bloque de instrucciones se va a ejecutar siempre y cuando la evaluación de su condición se cumpla. En este caso si la condición nunca se hace *falsa* el bloque a ejecutar se realizará indefinidamente. La sintaxis de esta estructura es la siguiente:

```
while (expresión)
{
    //Bloque de código a ejecutar;
}
```

```
//visualizar n asteriscos
int n = 20;
int contador = 0;
while (contador < n)
{
  Console.WriteLine(" * ");
  contador++;
}//fin de while
```

```
while (i <= 3)
{
  Console.WriteLine("valor de i: "+i);
  i = i + 1;
}
```

Ciclo do {}

Un ciclo ***do*** es similar al ciclo ***while*** con la excepción de que el ciclo ***do*** siempre será ejecutado al menos una vez a diferencia del ciclo ***while*** donde si la condición inicial no se cumple, el ciclo nunca se ejecutará. El ciclo ***do*** primero ejecuta el cuerpo del ciclo y después compara la condición.

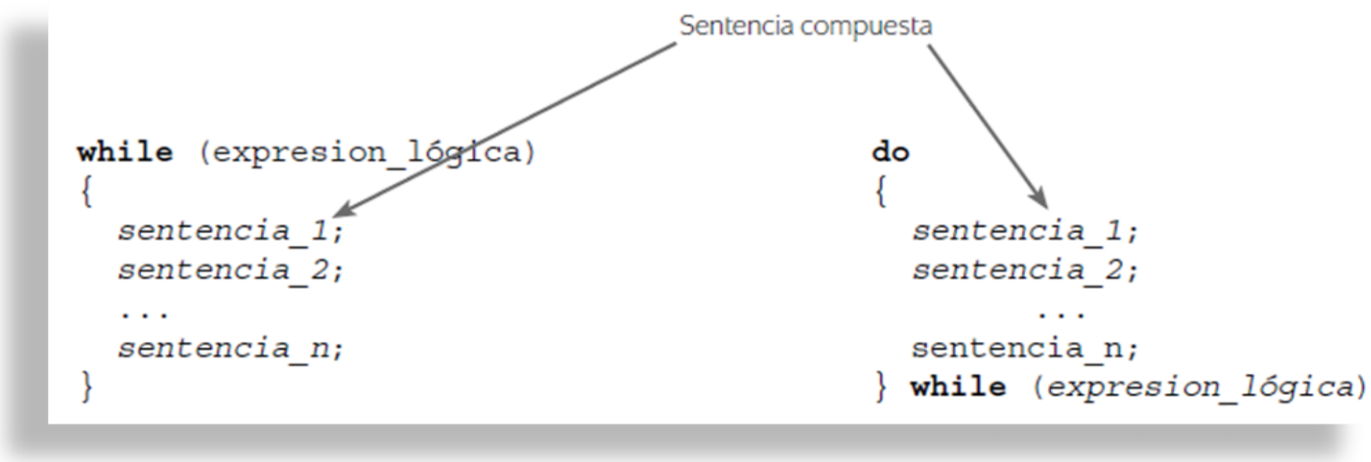
```
do
{
    //Bloque de código a ejecutar;
} while (condición);
```

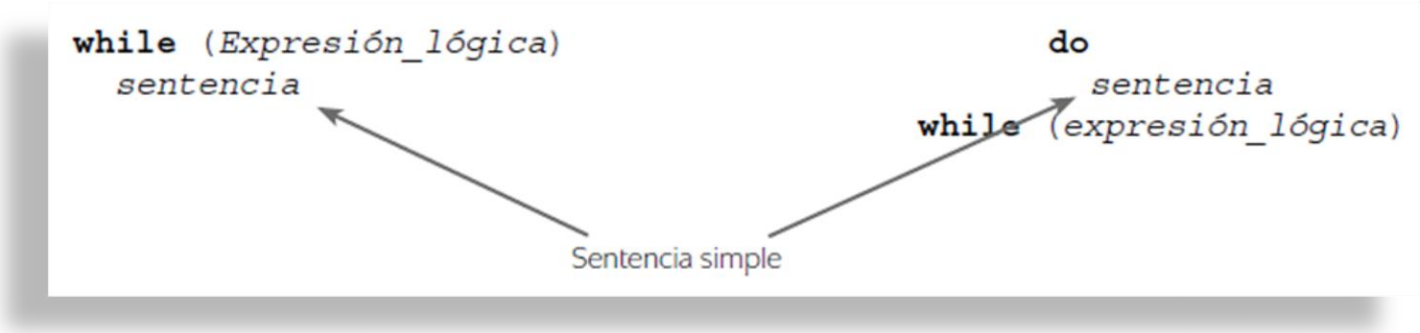
```
do {
  Console.WriteLine("Introduce numero: ");
  numero = Console.ReadLine();
  Console.WriteLine("El numero es: "+numero);
} while (numero > 0);
```

```
char opcion;
do {
     Console.WriteLine("Hola");
     Console.WriteLine("¿Desea otro tipo de saludo?");
     Console.WriteLine("Pulse s para si y n para no,");
     Console.WriteLine("y a continuación pulse intro: ");
     opcion = Console.ReadLine();
} while (opcion == 's' || opcion == 'S');
Console.WriteLine("Adiós");
```

Diferencias entre `while {} y do {} while`

Una sentencia **`do {} while`** es similar a **`while {}`** excepto que el cuerpo del bucle siempre se ejecuta al menos una vez.





Otras instrucciones

```
static void BreakStatement(string[] args)
{
    while (true)
    {
        string s = Console.ReadLine();
        if (string.IsNullOrEmpty(s))
            break;
        Console.WriteLine(s);
    }
}
```

```
static void ContinueStatement(string[] args)
{
    for (int i = 0; i < args.Length; i++)
    {
        if (args[i].StartsWith("/"))
            continue;
        Console.WriteLine(args[i]);
    }
}
```

Otras instrucciones

```
static void GoToStatement(string[] args)
{
    int i = 0;
    goto check;
    loop:
    Console.WriteLine(args[i++]);
    check:
    if (i < args.Length)
        goto loop;
}
```

```
static int Add(int a, int b)
{
    return a + b;
}
static void ReturnStatement(string[] args)
{
   Console.WriteLine(Add(1, 2));
   return;
}
```

Otras instrucciones

```
static void UsingStatement(string[] args)
{
    using (TextWriter w = File.CreateText("test.txt"))
    {
        w.WriteLine("Line one");
        w.WriteLine("Line two");
        w.WriteLine("Line three");
    }
}
```

Elementos básicos del Lenguaje C#

```
static double Divide(double x, double y)
{
 if (y == 0) throw new DivideByZeroException();
 return x / y;
}

static void TryCatch(string[] args)
{
  try
  {

    if (args.Length != 2)
    {
      throw new InvalidOperationException("Two numbers required");
    }
    double x = double.Parse(args[0]);
    double y = double.Parse(args[1]);
    Console.WriteLine(Divide(x, y));

  } catch (InvalidOperationException e) {
     Console.WriteLine(e.Message);
  } finally {
      Console.WriteLine("Good bye!");
  }
}
```

¿Preguntas?