Todo lo que debes saber sobre los Métodos en C#

1 comentario / C# / Por Héctor Pérez

Un método en C# es una sencuencia de enunciados dentro de una unidad lógica en nuestro programa. Son una de las partes escenciales en cualquier aplicación, y proporcionan una forma poderosa de ejecutar operaciones de forma ordenada.

Antes de pasar a analizar todo lo necesario para que te vuelvas un experto en la creación de métodos, quiero agradecer a Benjamín Camacho por la oportunidad de participar en el Segundo Calendario de Adviento de C#, en el enlace podrás encontrar otras fantásticas contribuciones sobre C#. Sin más, ¡pongamos manos a la obra!

Contents [hide]

Video sobre la creación y uso de Métodos en C#

Si quieres aprender de forma visual cómo utilizar Métodos en C#, entonces te dejo el video con la lección de esta publicación, donde se tocan todos los temas:

Partes de un Método en C#

Si deseamos aprender cómo crear un método, debemos de considerar 4 partes a recordar:

- Un nombre significativo
- Los enunciados que se ejecutarán cuando se invoque el método
- Un único tipo de dato de retorno
- Uno ó más datos de entrada, con los que nuestro método puede funcionar (*Opcional*).

Sintaxis de un Método en C#

La sintaxis fundamental para crear un método, es la siguiente:

```
tipoRetorno nombreMetodo (ListaDeParámetros)
{
   //Enunciados que se ejecutarán
}
```

Como puedes apreciar en el código anterior, he escrito cada uno de los elementos de la lista, a continuación, describiré cada uno de los elementos.

tipoRetorno → Tipo de Dato de Retorno: Es el tipo de dato que regresaremos una vez que se lleven a cabo las operaciones dentro del método, puede ser por ejemplo, un tipo de dato cadena (string), entero (int) ó cualquier otro tipo de dato que tengamos a nuestra disposición. Si en dado caso, deseamos que el método no retorne información, podremos utilizar la palabra void.

nombreMetodo → Nombre significativo del método: Este es el nombre con el que invocaremos el método desde otras partes de nuestro programa, se recomienda que tenga un nombre significativo, que denote una acción, es decir, que sea un verbo. Un ejemplo podría ser "DibujarCuadrado", ó "CalcularArea". Cabe destacar, que el nombramiento de los métodos, sigue las mismas reglas que el nombramiento de variables.

ListaDeParametros → Datos de entrada: Estos representan información que podemos pasar a nuestro método, con el fin de que los enunciados dentro de él, puedan trabajar con dicha información. Se tienen que escribir dentro de un par de paréntesis, y la sintaxis para declarar estos parámetros, es: "tipoDeDato" + "nombreParametro". En dado caso de que deseemos declarar más de un parámetro, tendremos que separarlos con una coma (,).

Enunciados que se ejecutarán: Son el conjunto de enunciados que realizrán operaciones dentro de nuestro método cuando éste sea invocado. Se tienen que escribir dentro de las llaves del método ({}).

Ejemplos

Vamos a ver algunos ejemplos prácticos para que quede completamente claro el concepto de métodos:

```
1 void Imprimir()
2 {
3    Console.WriteLine("Hola Mundo!");
4 }
```

En el ejemplo anterior, el método no regresará ningún valor, se llama Imprimir, no acepta ningún valor como parámetro y se encargará de Imprimir un Hola Mundo en la pantalla del usuario.

En dado caso de que queramos que el método, imprima una cadena, pero personalizada, podremos agregar un parámetro, para enviar desde otra parte de nuestro programa una cadena, y así el resultado sea diferente:

```
1 void Imprimir(string nombre)
2 {
3    Console.WriteLine("Hola " + nommbre);
4 }
```

Con esta modificación, el método sigue sin regresar ningún valor, pero hemos agregado a la lista de parámetros, uno que se llama nombre, el cual almacenará el resultado en la variable nombre. Utilizaremos dicho valor más adelante para imprimir en consola, Hola + el valor que enviemos a la variable nombre.

Supongamos que ahora deseamos que nuestro método, aparte de imprimir "Hola Héctor" (suponiendo que ese valor tiene la variable nombre), también deseamos regresar dicha cadena al lugar desde donde fue invocado el método. Esto lo podríamos

lograr, modificando nuestro método, indicando que deseamos regresar una cadena de texto:

```
string Imprimir()
{
   Console.WriteLine("Hola Mundo!");
}
```

Como te habrás dado cuenta, hemos cambiado el void por el string, lo que nos permitirá retornar la cadena de texto que queremos.

Vamos a ver otros ejemplos, que con el conocimiento adquirido, seguro podremos interpretar fácilmente:

```
int Sumar(int numero1, int numero2)
{
    //Enunciados a ejecutar
}
```

En el ejemplo anterior, estamos indicando que deseamos regresar un número entero, el método se llama Sumar (bastante representativo), y recibe 2 parámetros enteros, donde almacenaremos los valores de 2 números enteros, para posteriormente ejecutar una ó más operaciones.

¿Cómo regresar un valor de un método en C#?

Si en dado caso, deseamos regresar un valor, como resultado de la ejecución de uno ó más enunciados de nuestro método, podremos hacerlo a través de la siguiente sintaxis:

```
return ValorARegresar;
```

La palabra return es fundamental para llevar a cabo dicha operación, ya que será la que nos indique que deseamos regresar el valor que le sigue. Tomemos como ejemplo, el código anterior de este punto, y agreguemos un return:

```
int Sumar(int numero1, int numero2)
{
     return numero1 + numero2;
}
```

Hemos agregado la palabra clave return, y finalmente, una operación, la que llevará a cabo la suma de numero1 + numero2. Dicho return, podría regresar el resultado final de un conjunto de operaciones, pero siempre será un único valor.

De igual forma, es importante destacar que una vez invocado el enunciado con return, el código siguiente ya no se ejecutará, por lo que hay que tener cuidado de ver bien dónde lo utilizaremos.

Otro punto importante a denotar, es que podemos utilizar la palabra clave return sin ningún valor adicional, esto podría considerarse un tipo si en dado caso queremos cortar la ejecución de una función en dado caso de que se cumpla una condición, por ejemplo:

```
bool VerificarEdad(int edad)
```

```
{
   if(edad < 18)
   {
      return;
   }
   //Ejecutar enunciados si es mayor de edad
}
```

Métodos de Expresión Corporal (Expression Bodied Methods)

Existen ocasiones, en las que deseamos ejecutar alguna operación sencilla, o bien, una única operación en nuestro código, por lo que no deseamos declarar un método completo, ya que esto podría suponer la creación de mucho código.

Afortunadamente, en C#, contamos con una forma simplificada de escribir dichos métodos, que al igual que un método normal, nos permite retornar un tipo de dato, y a la vez, pasar parámetros.

La definición de estos métodos, es de la siguiente forma:

TipoDatoRetorno NombreMetodo(parámetros) ⇒ EnunciadoAEjecutar

He marcado en negritas el símbolo que representa este tipo de métodos, ya que será el que utilicemos en lugar del par de llaves que se usa normalmente.

Por ejemplo, si deseamos reescribir el método que realiza la suma entre dos valores, que hemos escrito anteriormente, lo haríamos de la siguiente forma:

```
int Sumar(int numero1, int numero2) => return numero1 + numero2;
```

Como puedes observar, aparte del uso del símbolo ⇒, también estamos eliminando el uso de la palabra clave return, ya que en este tipo de expresiones no es necesaria. El valor de la expresión, será automáticamente devuelto.

Por otra parte, si el método no da como resultado el retorno de un valor, entonces, se inferirá que es de tipo void, como en el siguiente ejemplo:

```
void hello() => Console.WriteLine("Hola Mundo!);
```

En realidad, no hay ventaja en cuanto a rendimiento entre utilizar la definición normal de un método, ó bien, un método de expresión corporal, sino que únicamente nos servirán para reducir la cantidad de llaves ({}) en nuestro programa.

¿Cómo invocar un método en C#?

Ya anteriormente hemos visto cómo declarar un método, sin embargo, hay que saber cómo invocarlo desde nuestro programa.

En primer lugar, iniciaremos colocando en Visual Studio, en un proyecto de consola, uno de los métodos definidos anteriormente:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleTests
{
    class Program
    {
        static void Main(string[] args)
        {
        }

        void Imprimir()
        {
            Console.WriteLine("Hola Mundo!");
        }
    }
}
```

Métodos en C#

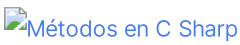
Como puedes apreciar, el método lo hemos definido dentro de una clase, en este caso la clase Program, y debajo del método Main. Vamos a tratar de invocarlo entonces, la sintaxis para una invocación es la siguiente:

1. Si no regresa un valor, y no recibe parámetros:
    1. *NombreMetodo();*
2. Si no regresa un valor, pero sí recibe parámetros:
    1. *NombreMetodo(parámetros);*
3. Si regresa un valor y no recibe parámetros:
    1. *variableParaAlmacenarResultado = NombreMetodo()*
4. Si regresa un valor y recibe parámetros:
    1. *variableParaAlmacenarResultado = NombreMetodo(parámetros)*

Como podemos ver en la imagen, nuestro método no regresa ningún valor (lo vemos porque tenemos un void), y no recibe parámetros (entre paréntesis no hay nada), por lo que procederemos a utilizar la Regla 1 para la invocación:

```
1 Imprimir();
```

Sin embargo, si haces esto, seguramente te salga un error como el siguiente:

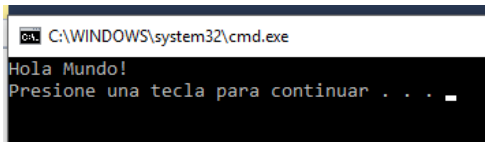


Métodos en C Sharp

Este error, básicamente lo que quiere decir, es que, debido a que tu método Main es estático, no puedes utilizar un método si no tiene también el static (ya veremos en otra entrada este tema). Por lo que procederemos a modificar nuestro método Imprimir para que también sea estático:

```
static void Imprimir()
{
    Console.WriteLine("Hola Mundo!");
}
```

Con lo que el error se corregirá, y podremos ejecutar la app sin problemas:



Implementar Métodos en C#

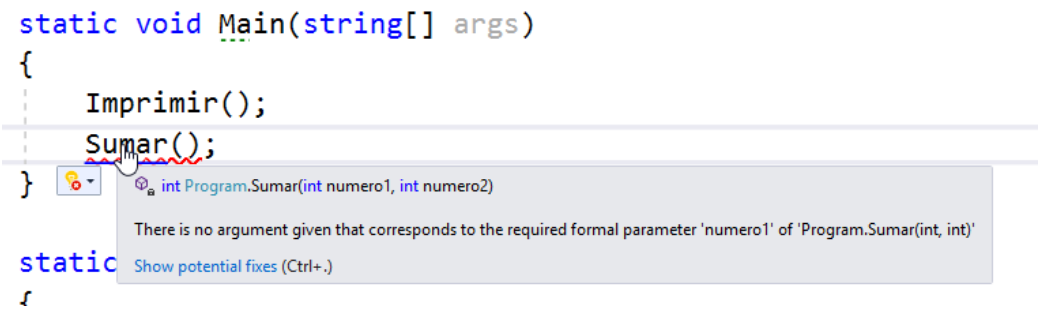
Vamos ahora a ver cómo invocar el método Sumar, que si recuerdas, tenía la siguiente definición (he agregado la palabra Static para evitar líos):

```
static int Sumar(int numero1, int numero2)
{
    return numero1 + numero2;
}
```

Método en C#

Analicemos un poco el método, nos indica que va a regresar un número entero (primer int), y que está en espera de dos parámetros (numero1 y numero2), por lo que procederemos a utilizar la regla 4, sin embargo, vamos a analizar cada uno de los casos para que quede completamente claro:

1.- En dado caso de querer invocar el método como lo hemos hecho con el método imprimir, nos saldrá lo siguiente:



Implementar métodos en C#

Si lees detenidamente el mensaje, nos indica que no estamos pasando los valores o parámetros que el método espera, por lo que tendremos que hacerle caso y pasar 2 números enteros:

```
static void Main(string[] args)
{
    Imprimir();
    Sumar(3,2);
}
```

Implementando un método en C#

El error se ha ido, pero tal vez te quedes con la duda, de porqué no te ha marcado un error, si no hay una variable que almacene el resultado como hemos visto en la regla:

variableParaAlmacenarResultado = NombreMetodo(parámetros)

Esto se debe a que, mientras se ejecute el método sin problema, el almacenamiento del valor es opcional. Por ejemplo, supón que tienes un método que guarde datos en una base de datos, puede que quieras o no almacenar en una variable la cantidad de filas que se han actualizado.

En caso de que quieras almacenar el valor, puedes declarar una variable, que corresponda al tipo de dato devuelto (en nuestro caso int), para llevar a cabo dicha acción:

```
1        static void Main(string[] args)
2        {
3            Imprimir();
4            int resultado = Sumar(3,2);
5            Console.WriteLine(resultado);
6        }
```

Preguntas comunes sobre invocación a métodos en C#

Existen algunas preguntas recurrentes cuando trabajamos con métodos en C#, las cuales enumero a continuación:

¿Puedo enviar otro tipo de dato como parámetro al método?

La respuesta puede variar. Te explico, si el método espera recibir un tipo de dato entero, y le mandas una cadena, el compilador te dirá que es imposible llevar a cabo dicha acción:

Métodos en C#

Como puedes ver, el mensaje es claro: “No se puede convertir de una cadena a un valor entero. Lo mismo ocurre con un tipo de dato, por ejemplo, double, sólo que en este caso, podríamos hacer una conversión para “forzar” un envío de la información:

¿Cómo llamar un método en C#?

En el caso anterior, estaríamos convirtiendo el tipo de dato double, a un tipo de dato entero, por lo que únicamente sería enviada la parte entera, por lo que el resultado del método sería:

Invocar un Método desde C#

¿Cómo regresar múltiples valores de un método?

Hay ocasiones en las que tal vez quieras retornar múltiples valores de un método, tal vez encuentres programadores que te digan que no es posible a menos de que crees una clase. Yo te digo, que sí es posible, a través de una tupla.

Una tupla, en resumidas cuentas, es una colección pequeña de valores. La definición de una tupla, está compuesta por únicamente el tipo de dato de cada uno de los valores que es retornado.

Es importante recalcar, que esta característica, está disponible a partir de la versión de C# 7.0, por lo que tendremos que apuntar a dicho framework como parte de nuestro proyecto:

Regresar múltiples valores de un
método en C#

Una vez hecho esto, podremos regresar más de un valor como parte de un método, tal como en el siguiente ejemplo:

```
1        static (int, int) RegresarValores()
2        {
3            return (1, 2);
4        }
```

Como podrás ver, es casi la misma definición que un método normal, a excepción de que retornamos varios valores.

Para almacenar los valores en variables como parte de nuestro método que invoca al método, declararemos el número de variables correspondientes al número de valores retornados:

```
1            int valor1, valor2;
2            (valor1, valor2) = RegresarValores();
```

Quedando así el resultado final:

Devolver múltiples valores de método
en C#

¿Qué es la sobrecarga de métodos en C#?

Supon que escribes un método para llevar a cabo la impresión de una cadena, como por ejemplo, el siguiente:

```
1        static void Imprimir(string cadena)
2        {
3            Console.WriteLine(cadena);
4        }
```

Ahora, imagina que alguien que quiere usar tu código, te pregunta si hay forma, en la cual, en lugar de pasarle una cadena como parámetro, le pases un número entero (int). Escribamos un nuevo método, cambiando únicamente el tipo de parámetro por un tipo entero, de la siguiente manera:

```
1        static void Imprimir(int numero)
2        {
3            Console.WriteLine(numero);
4        }
```

Como puedes observar en la siguiente imagen, no existe ningún problema por parte del compilador, aunque los 2 métodos se llamen exactamente igual (Imprimir):

Sobrecarga de operadores en C#

A esto se le llama Sobrecarga de un método, al hecho de tener 2 métodos que se llamen igual, pero que tengan un conjunto de parámetros diferentes.

Tal vez te preguntes si aplica lo mismo para el tipo de dato de retorno, pero lo cierto es que si sólo cambias el tipo de dato de retorno, el compilador no estará nada feliz:

Sobrecarga en métodos de C#

Esto significa, que la sobrecarga de métodos sólo aplica para los parámetros que forman parte de un método en C#.

Otro ejemplo del uso de la sobrecarga de métodos, es si tuvieras el primer parámetro igual, pero un segundo parámetro diferente, como en el siguiente ejemplo:

```
1        static void Imprimir(string cadena)
2        {
3            Console.WriteLine(cadena);
4        }
5        static void Imprimir(string cadena, int numero)
6        {
7            Console.WriteLine(numero);
8        }
```

En el código anterior, la definición del método es totalmente válida, por lo que no habrá problemas en el código.

Ejemplos de Sobrecarga de Métodos

Lo que hemos hecho anteriormente, es algo parecido a lo que se hace con el método System.Console.WriteLine(...); el cual, si comenzamos a escribir, veremos que el intellisense nos desplegará un conjunto de opciones:

Definir métodos en C#

Esto quiere decir, que a un Console.WriteLine, podemos pasarle un tipo de dato decimal, double, string, long, etc., por lo que lo siguiente es totalmente válido:

```
1            Console.WriteLine(3);
2            Console.WriteLine(2.3);
3            Console.WriteLine("Hola Mundo");
4            Console.WriteLine('a');
```

Si ingresamos al explorador de objetos, podremos ver todas las definiciones de dicho método:

Método WriteLine en C#

Métodos anidados en C# (También llamados Funciones Locales en C# ó Local Functions en C#)

A partir de C# 7.0, podemos utilizar los llamados métodos anidados ó funciones locales ó Local Functions en C#, los cuales son métodos privados que están anidados dentro de otros métodos en nuestro código.

Esto nos permite encapsular funcionalidad que no queremos exponer a través de nuestro código, ya que este tipo de métodos es únicamente llamable desde el método que lo implementa.

Este tipo de métodos, los podemos declarar y llamar desde:

- Métodos
- Constructores
- Accesores de Propiedades
- Accesores de Eventos
- Métodos Anónimos
- Expresiones Lambda
- Finalizadores
- Otras Funciones locales

Supongamos que deseamos saber si un número ingresado es par o impar, pero con un resultado en formato texto, es decir, que el resultado nos devuelva “Es Par” ó “Es Impar”.

Para llevar a cabo esta tarea, vamos a codificar un método anidado de la siguiente forma:

```
        static string IsPair(string input)
        {
            int validInt = int.Parse(input);
            string isPair = Pair(validInt);
            return isPair;
            string Pair(int dataValue)
            {
                if (dataValue % 2 == 0)
                {
                    return "Es Par";
                }
                else
                {
                    return "Es Impar";
                }
            }
```

```
17        }
```

Como puedes apreciar, dentro del método llamado IsPair, tenemos otro método llamado "Pair", el cual es el que nos permitirá llevar a cabo la operación para saber si un número es par ó no. De la misma forma, se nos devuelve la cadena correspondiente al caso.

Con ello, desde nuestro código en el método Main, podremos invocar al método IsPair, pero no al método Pair:

Métodos Locales en C#

Parámetros opcionales y argumentos con Nombre en C# (Optional Parameters & Named Arguments)

Ya hemos visto anteriormente cómo podemos tener una sobrecarga de operadores para un método.

Ahora bien, ¿Qué pasaría si en dado caso, quisieras tener un método con uno ó más parámetros que sean opcionales para el desarrollador?

Parámetros Opcionales

Empecemos con el ejemplo, supongamos que un restaurante desea implementar un pequeño sistema, que permita a los consumidores calcular el total de una comida con la propina incluida, con el fin de dividor los costos. Codificaríamos un método similar al siguiente:

```
1         static decimal CalculateTip(decimal MealBill, int peopleQty, decimal tipPercentage)
2         {
3             var tip = MealBill / tipPercentage;
4             var total = tip + MealBill;
5             var totalPerPerson = total / peopleQty;
6             return totalPerPerson;
7         }
```

Con esto, podríamos ingresar los siguiente valores al momento de invocar el método:

```
1         static void Main(string[] args)
2         {
3             Console.WriteLine(CalculateTip(500, 2, 10));
4         }
```

Lo que nos retornaría un resultado como el siguiente:

Argumentos Opcionales en C#

Ahora bien, supongamos que el restaurante cuenta con un sistema de aparcado de carros, el cual tiene un costo fijo, y el cual también podremos agregar al cálculo para dividir la cuenta. No todos los clientes llevan carro, por lo que el parámetro es opcional.

La forma para indicar que un parámetro ó argumento es opcional, es asignando un valor desde la firma del método.

Es importante recalcar que los argumentos opcionales, se deben colocar al final de la definición del método, por lo que tendríamos que modificar nuestra definición de la siguiente forma:

```
static decimal CalculateTip(decimal MealBill, int peopleQty, decimal tipPercentage, bool clientHasCar = false)
```

En el ejemplo anterior, indicamos que por default, la variable cientHasCar es igual a falso. Si regresamos al método Main, veremos que no hay ningún error:

Argumentos Opcionales en C#

Esto es, porque en este caso, podríamos o no pasar un valor como parte del método CalculateTip. Ahora, solo faltaría coficiar la funcionalidad en caso de que sí pasáramos un valor como parte del método:

```
static decimal CalculateTip(decimal MealBill, int peopleQty, decimal tipPercentage, bool clientHasCar = false)
{
    var tip = MealBill / tipPercentage;
    var total = tip + MealBill;
    if (clientHasCar)
    {
        total += 20;
    }
    var totalPerPerson = total / peopleQty;
    return totalPerPerson;
}
```

Hemos modificado el método, para que, en dado caso de que el cliente lleve carro, se agreguen 20 pesos al total, con el fin de dividir también el costo de estacionamiento, por lo que podríamos ahora sí, hacer un cálculo en caso de que así lo deseemos:

```
        static void Main(string[] args)
        {
            Console.WriteLine(CalculateTip(500, 2, 10));
            Console.WriteLine(CalculateTip(500, 2, 10, true));
        }
```

De esta forma, el resultado de la ejecución sería la siguiente:

Parámetro opcional en C#

Argumentos con Nombre

Los argumentos con Nombre se refieren al hecho de que puedes invocar los diferentes parámetros de un método, sin necesidad de invocar previamente los argumentos previos.

Iniciemos con algo sencillo, veamos cuál es el nombre del parámetro que ha definido el equipo de C#, para imprimir una cadena con Console.WriteLine:

C# Named Arguments

Como podemos apreciar, el nombre del parámetro es value, por lo que podríamos invocarlo a través de su nombre, de la siguiente forma:

```
        static void Main(string[] args)
        {
            Console.WriteLine(value: "Hola Mundo");
        }
```

Dicho esto, si quisiéramos utilizar de nuevo nuestro método para calcular por partes iguales el costo de una comida, podríamos ocupar el siguiente código:

```
        static void Main(string[] args)
        {
```

```
            Console.WriteLine(CalculateTip(tipPercentage:10, peopleQty:2, MealBill:200));
        }
```

Podemos observar cómo hemos modificado el orden de los parámetros que se tenía por defaul, primero pasando el porcentaje de propina, luego la cantidad de personas, y por último, el total de la cuenta.

Aunque no tiene mucho sentido ocuparlo con parámetros normales, sí que son muy útiles con los parámetros opcionales, ya que nos permitirán especificar únicamente aquellos que necesitemos.

Un ejemplo que es muy visible, es con los métodos de los componentes COM, los cuales tienen una infinidad de parámetros opcionales, específicamente se tenía este problema cuando se requería utilizar una interfaz para trabajar con archivos de Word ó Excel.

Previamente, en C# 3.0 y antes, si deseabamos realizar una tarea, como por ejemplo el uso del método AutoFormat, podíamos tener un código como el siguiente:

```
// In C# 3.0 and earlier versions, you need to supply an argument for
// every parameter. The following call specifies a value for the first
// parameter, and sends a placeholder value for the other six. The
// default values are used for those parameters.
var excelApp = new Microsoft.Office.Interop.Excel.Application();
excelApp.Workbooks.Add();
excelApp.Visible = true;
var myFormat =
    Microsoft.Office.Interop.Excel.XlRangeAutoFormat.xlRangeAutoFormatAccounting1;
excelApp.get_Range("A1", "B4").AutoFormat(myFormat, Type.Missing,
    Type.Missing, Type.Missing, Type.Missing, Type.Missing);
```

Gracias a los parámetros opcionales y named arguments, hoy en día, podemos reescribir la misma funcionalidad con menos código:

```
// The following code shows the same call to AutoFormat in C# 4.0. Only
// the argument for which you want to provide a specific value is listed.
excelApp.Range["A1", "B4"].AutoFormat( Format: myFormat );
```

← Entrada anterior

Entrada siguiente →

1 comentario en “Todo lo que debes saber sobre los Métodos en C#”

123
22 SEPTIEMBRE, 2023 A LAS 9:03 AM

“Sencuencia” de enunciados dentro de una unidad lógica en nuestro programa. Son una de las partes esenciales en cualquier aplicación, y proporcionan una forma poderosa de ejecutar operaciones de forma ordenada, nnms wei pork no sabes hescrivir t3 ahse flata hun profezor neta corrigigamelo ya

Responder

Deja un comentario

Tu dirección de correo electrónico no será publicada. Los campos obligatorios están marcados con *

Escribe aquí...

Nombre*

Correo electrónico*

Web

Guarda mi nombre, correo electrónico y web en este navegador para la próxima vez que comente.

Publicar comentario »