NOTAS DE CLASE
.NET CON C#

/* ****** Jaime E. Montoya M. ****** */

NOTAS DE CLASE
.NET CON C#

```
/**
* Versión 2.0
* Fecha: 2024, semestre 2
* Licencia software: GNU GPL
* Licencia doc: GNU Free Document License (GNU FDL)
*/


namespace CSharp
{
    class Author
    {

        string name = "Jaime E. Montoya M.";

        string profession = "Ingeniero Informático";

        string employment = "Docente y desarrollador";

        string city = "Medellín-Antioquia-Colombia";

        int year = 2024;
    }
}
```

Contenido

Introducción

Este documento es un complemento a las clases presenciales y virtuales, y está basado en la bibliografía del curso, así como de otras fuentes adicionales que se indican a lo largo del texto, además de la experiencia del autor en su función docente en las áreas de programación y como desarrollador en distintas empresas del departamento. No se pretende reemplazar los textos guías con este manual, sino servir de ayuda didáctica y apoyo académico a los estudiantes.

La guía incluye, además de los conceptos teóricos, ejemplos, gráficas, desarrollos en clase, y al final de cada unidad, unas preguntas, ejercicios y una actividad que puede corresponder al entregable de seguimiento acordado desde el inicio de la asignatura. Se espera que estos ejercicios permitan reforzar los conceptos y promover la práctica y el estudio de los conceptos vistos.

Al final de este manual, se indican fuentes y referencias adicionales que el estudiante puede consultar.

Este documento incluye una serie de ejercicios resueltos de programación realizados en clase y propuestos por el autor que se anexan finalizando cada unidad.

Estos ejercicios incluyen algunos conceptos teóricos y prácticos adicionales, consejos, buenas prácticas y ejemplos explicados, divididos de acuerdo a la unidad temática estudiada. Se espera que estos ejercicios permitan reforzar los conceptos y promover la práctica y el estudio de los conceptos vistos.

Los ejemplos se ilustran con el lenguaje C#, y en algunos casos, se muestra el pseudocódigo para abordar algún problema.

Además del lenguaje, también se hace énfasis en lo algorítmico, donde el estudiante podrá poner a prueba sus conocimientos, así como analizar las diferencias entre la teoría y la práctica, esto es, llevar un algoritmo y convertirlo en programa.

Lista de abreviaturas

BD: Base de Datos

CRUD: Create - Read - Update - Delete (Crear/Insertar/Agregar - Leer/Listar/Buscar - Actualizar/Modificar - Eliminar/Borrar)

IDE: Integrated Development Environment (Entorno de Desarrollo Integrado)

IoT: Internet of Things (Internet de las Cosas)

ISO: International Organization for Standardization (Organización Internacional de Estandarización)

SDK: Software Development Kit (Kit de Desarrollo de Software)

SI: Sistema de Información

SO: Sistema Operativo

SQL: Structured Query Language

WWW: World Wide Web

Capítulo 1. Introducción a .NET y al lenguaje C#

.NET (dotnet)

.NET (*dotnet*) es una plataforma de desarrollo de código abierto gratuita y multiplataforma para crear muchos tipos diferentes de aplicaciones. Con .NET se pueden usar varios lenguajes, editores y bibliotecas para programas de diversos tipos: juegos, aplicaciones para la Web, dispositivos móviles y de escritorio, y el Internet de las cosas (IoT), entre otras.

.NET (anteriormente llamado .NET Core, a su vez sucesor de .NET Framework) es un framework gratuito y de código abierto para los sistemas operativos Windows, Linux y macOS. El proyecto es desarrollado principalmente por Microsoft bajo la licencia MIT.

Descargar .NET

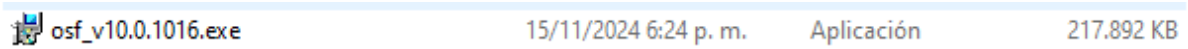
Accedemos a cualquiera de esto enlaces para la descarga del .NET SDK

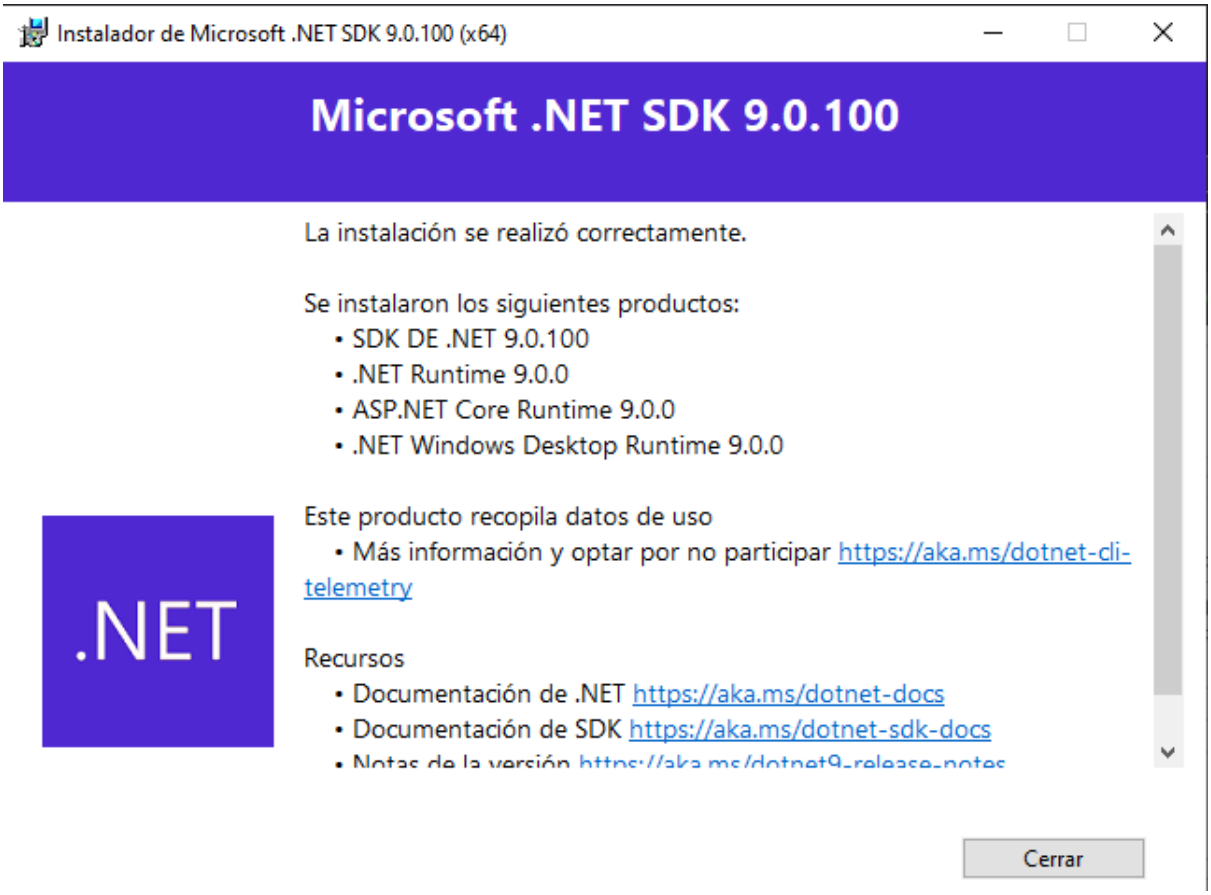
Microsoft .NET 9.0 SDK
https://dotnet.microsoft.com/en-us/download/dotnet/thank-you/sdk-9.0.100-windows-x64-installer

Descargar .NET (Linux, macOS y Windows)

Instalar .NET

Ejecutamos el instalador descargado y seguimos los pasos del asistente en Windows.

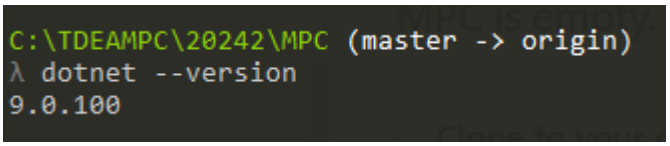




Ver la versión de .NET y C#

Una vez instalado .NET podemos ver su versión desde una terminal

```
>dotnet --version
```



Para ver la versión de C# (C Sharp), incluir la línea "`#error version`" en el código de C#, con distinción de mayúsculas y minúsculas. Esto generará un error de compilador, CS8304, con un mensaje que contiene la versión del compilador y del lenguaje.

Ejemplo

```
#error version
```

```
λ dotnet build
Restauración completada (0,7s)
  HelloWorld error con 2 errores (2,3s)
    C:\Users\posit\Documents\csharp\HelloWorld\Program.cs(12,20): error CS1029: #error: 'version'
    C:\Users\posit\Documents\csharp\HelloWorld\Program.cs(12,20): error CS8304: Versión de compilador:
"4.12.0-3.24523.4 (f3348c2a)". Versión de lenguaje: 13.0.

Compilación error con 2 errores en 4,2s
```

El lenguaje C# (C Sharp)

C# (pronunciado C sharp) es un lenguaje de programación multiparadigma desarrollado y estandarizado por la empresa Microsoft como parte de su plataforma .NET, que después fue aprobado como un estándar por la ECMA (ECMA-334) e ISO (ISO/IEC 23270).

Su sintaxis básica deriva de C/C++ y utiliza el modelo de objetos de la plataforma .NET, similar al de Java, aunque incluye mejoras derivadas de otros lenguajes que lo hacen más “agradable” a la hora de programar.

Crear un proyecto de consola en .NET con C#

Desde una terminal o consola, ingresamos el siguiente comando

Sintaxis

```
>dotnet new console -o nombre-proyecto
```

Ejemplo

```
>dotnet new console -o HelloWorld
```

Una vez creado el proyecto, editamos el archivo `Program.cs`

Capítulo 2. Introducción a la programación con el lenguaje C#

Comentarios

Están las formas para una línea o varias líneas.

Ejemplo

```
// Esto es un comentario de una línea. Aplica desde donde se encuentre

/*
 Comentario de varias líneas.
 Todo lo encerrado entre los símbolos, es un comentario
*/
```

Salida de información

El método `Console.WriteLine()` permite la salida de información en el flujo de salida estándar. Es el equivalente en pseudocódigo a la instrucción `Imprimir`.

Ejemplo

```
Console.WriteLine("¡Hola, esto es C#!");
```

Compilar la aplicación

C# es un lenguaje compilado, por tanto, es necesario compilar la aplicación (programa) antes de ejecutarla. Ubicados dentro de la carpeta del proyecto ejecutamos el siguiente comando para compilar el proyecto:

```
>dotnet build
```

Ejecutar (correr) la aplicación

Una vez compilada la aplicación, .NET se encarga de generar los archivos objeto (ejecutables) con los cuales se puede desplegar ésta. Ubicados dentro de la carpeta del proyecto ejecutamos el siguiente comando.

```
>dotnet run
```

Finalización de instrucciones

Observe que en el ejemplo “Hola Mundo” la instrucción donde se imprime el mensaje finaliza con punto y coma (;). Las sentencias de C# finalizan con el carácter `;` a excepción de los bloques especificados entre llaves, los cuales indican el inicio y fin de construcciones de código específico que se verán más adelante.

Tipos de datos

C# es un lenguaje fuertemente tipado. Las variables y constantes tienen un tipo, al igual que todas las expresiones que se evalúan como un valor. Cada declaración del método especifica un nombre, el tipo y naturaleza (valor, referencia o salida) para cada parámetro de entrada y para el valor devuelto. La biblioteca de clases .NET define tipos numéricos integrados, así como tipos complejos que representan una amplia variedad de construcciones. Entre ellas se incluyen el sistema de archivos, conexiones de red, colecciones y matrices de objetos, y fechas. Los programas de C# típicos usan tipos de la biblioteca de clases, así como tipos definidos por el usuario que modelan los conceptos que son específicos del dominio del problema del programa.

Los tipos de datos presentes en el lenguaje C# son básicamente los mismos de lenguajes que provienen de C/C++; a continuación se muestran los tipos primitivos más comunes:

Tipos de datos numéricos

Tipos numéricos enteros

Palabra clave/tipo de C#	Intervalo	Tamaño	Tipo de .NET

`sbyte`	De -128 a 127	Entero de 8 bits con signo	System.SByte
`byte`	De 0 a 255	Entero de 8 bits sin signo	System.Byte
`short`	De -32 768 a 32 767	Entero de 16 bits con signo	System.Int16
`ushort`	De 0 a 65.535	Entero de 16 bits sin signo	System.UInt16
`int`	De -2.147.483.648 a 2.147.483.647	Entero de 32 bits con signo	System.Int32
`uint`	De 0 a 4.294.967.295	Entero de 32 bits sin signo	System.UInt32
`long`	De -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807	Entero de 64 bits con signo	System.Int64
`ulong`	De 0 a 18.446.744.073.709.551.615	Entero de 64 bits sin signo	System.UInt64
`nint`	Depende de la plataforma (calculada en tiempo de ejecución)	Entero de 64 bits o 32 bits con signo	System.IntPtr
`nuint`	Depende de la plataforma (calculada en tiempo de ejecución)	Entero de 64 bits o 32 bits sin signo	System.UIntPtr

Tipos numéricos de punto flotante

Palabra clave/tipo de C#	Intervalo aproximado	Precisión	Tamaño	Tipo de .NET
`float`	De ±1,5 x 10^{-45} a ±3,4 x 10^{38}	De 6 a 9 dígitos aproximadamente	4 bytes	System.Single
`double`	De ±5,0 × 10^{-324} a ±1,7 × 10^{308}	De 15 a 17 dígitos aproximadamente	8 bytes	System.Double
`decimal`	De ±1,0 x 10^{-28} to ±7,9228 x 10^{28}	28-29 dígitos	16 bytes	System.Decimal

Tipo de dato booleano

La palabra clave de tipo `bool` es un alias para el tipo de estructura de .NET `System.Boolean` que representa un valor booleano que puede ser `true` o `false`.

Constantes booleanas

Se especifican mediante las palabras clave `true` y `false`.

Tipos de datos para el manejo de textos

.NET trata las cadenas de caracteres como objetos de una clase (String), pero también dispone de funcionalidades para tratar cada carácter independiente.

Caracteres

El tipo `char` es usado para almacenar un sólo carácter; dicho carácter debe estar encerrado entre apóstrofos o comillas simples.

Cadenas de caracteres

El tipo `string` es usado para almacenar una secuencia de caracteres; la cadena debe estar encerrada entre comillas dobles.

Literales de cadena

Un texto encerrado entre comillas dobles se conoce como **literal de cadena**.

Secuencias de escape

Las combinaciones de caracteres que constan de una barra diagonal inversa (\) seguidas de una letra o de una combinación de dígitos se denominan "secuencias de escape". Para representar un carácter de nueva línea, comillas simples u otros caracteres en una constante de caracteres, se deben usar secuencias de escape. Una secuencia de escape se considera un carácter individual y por tanto es válida como constante de caracteres.

Las secuencias de escape se suelen utilizar para especificar acciones como retornos de carro y movimientos de tabulación en terminales e impresoras. También se emplean para proporcionar representaciones literales de caracteres no imprimibles y de caracteres que normalmente tienen significados especiales, como las comillas dobles ("). En la tabla siguiente se enumeran las secuencias de escape ANSI y lo que representan, así como su codificación unicode.

Secuencia de escape	Nombre de carácter	Codificación Unicode
\'	Comilla simple	0x0027
\"	Comilla doble	0x0022
\	Barra diagonal inversa	0x005C
\0	Null	0x0000
\a	Alerta	0x0007
\b	Retroceso	0x0008
\f	Avance de página	0x000C
\n	Nueva línea	0x000A

\r	Retorno de carro	0x000D
\t	Tabulación horizontal	0x0009
\v	Tabulación vertical	0x000B
\u	Secuencia de escape Unicode (UTF-16)	`\uHHHH` (intervalo: 0000 - FFFF; ejemplo: `\u00E7` = "ç")
\U	Secuencia de escape Unicode (UTF-32)	`\U00HHHHHH` (intervalo: 000000 - 10FFFF; ejemplo: `\U0001F47D` = "👽")
\x	Secuencia de escape Unicode similar a "\u" excepto con longitud variable	`\xH[H][H][H]` (intervalo: 0 - FFFF; ejemplo: `\x00E7` o `\x0E7` o `\xE7` = "ç")

Ejemplo

```
char c1, c2;
string text1, text2;
string message = System.String.Empty; // Mejor que "" para cadena vacía
text1 = "Hola a todos";
text2 = "Hoy es día \t de estudiar"; //Salida: Hoy es día    de estudiar
c1 = 'A';
c2 = 'x';
```

Cadenas textuales

Las **cadenas textuales** conservan los caracteres y secuencias de escape como parte del texto; éstas se definen con el carácter arroba @ antes del literal de cadena.

Ejemplo

```
string textual;
textual = @"Esta es una ""cadena""; // salida: Esta es una "cadena"
```

Concatenación de cadenas

Es una operación que consiste en unir (junta) cadenas. C# utiliza el operador sobrecargado `+` para concatenar cadenas. Los datos que no son string, son convertidos (casteados) automáticamente a cadenas por el lenguaje.

Ejemplo

El valor de la expresión `num` (que es 3) se concatena con el literal de cadena "`Número = `". para obtener el resultado `número = 3` en la salida.

```
int num = 3;
Console.Write("Número = " + num);
```

Interpolación de cadenas

El carácter **$** identifica un **literal de cadena** como una **cadena interpolada**. Una cadena interpolada es un literal de cadena que puede contener **expresiones de interpolación**. Cuando una cadena interpolada se resuelve en una cadena de resultado, el compilador reemplaza los elementos con expresiones de interpolación por las representaciones de cadena de los resultados de la expresión.

La interpolación de cadenas proporciona una sintaxis más legible y conveniente de dar formato a las cadenas. Es más fácil de leer que el formato compuesto de cadenas.

Ejemplo

El valor de la expresión `num` (que es 3) se interpola con el literal de cadena "`Número = `". para obtener el resultado `número = 3` en la salida.

```
int num = 3;
Console.Write($"Número =  {num}");
```

Nota

Observe como se dispone de dos opciones para realizar la misma tarea con la concatenación y la interpolación. Se recomienda realizar interpolación de cadenas que se ejecuta más rápido que la concatenación.

Declaración de constantes y variables

Constantes

Una **constante** es inmutable a lo largo de la ejecución del programa, por lo que su valor no puede ser cambiado. Para declarar una constante usamos la palabra reservada `const`.

Sintaxis

```
const tipoDato constante = valor;
```

Ejemplo

```
const int x = 100;
const string firstDay = "Domingo";
```

Variables

Una **variable** es un espacio de memoria que muta a lo largo de la ejecución del programa. Para declarar una variable en C# especificamos el tipo de dato seguido de una lista de variables separadas por comas. También se puede inicializar la variable en su declaración

Sintaxis

```
tipoDato lista-variables;
```

Ejemplo

```
double x, y, z;
string text, lastText;
int num = 0;
```

Entrada de información

En la consola o terminal hacemos uso de la clase `Console` y el método `ReadLine()`.

Ejemplo

Leer tres variables por teclado, una cadena, un entero y un real. Aquí se **castea** (convierte) la variable `x` al tipo `int` y `z` al tipo `double`. Por defecto, la entrada por teclado es un `string`.

```
string n = Console.ReadLine();
int x = Int32.Parse(Console.ReadLine());
double z = Convert.ToDouble(Console.ReadLine());
```

Operadores

Los operadores presentes en C# son básicamente los mismos de lenguajes que provienen de C/C++ y que conservan la misma prioridad, entre ellos, están los siguientes.

Operador asimétrico de asignación

Para asignar valores a una variable, C# implementa el signo igual =. En los apartados anteriores se ilustraron casos del uso de este operador.

Sintaxis

```
variable = valor;
```

Ejemplo

```
z = 50;
```

Operadores aritméticos

Se utilizan para realizar operaciones

Operador	Símbolo	Prioridad
Multiplicación	*	Alta
División	/	
Módulo	%	
Suma	+	Baja
Resta	-	

Ejemplo

```
z = x - y;
Console.WriteLine("Resta: " + z);
z = x * y;
Console.WriteLine("Producto: " + z);
```

Operadores relacionales o de comparación

Son operadores binarios que al ser ejecutados efectuando una comparación, devuelven un valor booleano.

Operador	Símbolo	Prioridad
Igual	==	Alta
Diferente	!=	
Mayor que	>	Media
Menor que	<	
Mayor o igual que	>=	Baja
Menor o igual que	<=	

Ejemplo

```
int a, b
bool c;
a = 5;
b = 0;
c = a >= b;
Console.WriteLine("c: " + c);
```

Operadores lógicos

Utilizados como conectivos lógicos; siguen las mismas reglas del álgebra proposicional (booleana)

Operador	Símbolo	Prioridad
Negación	!	Alta

Conjunción	**&&**	Media
Disyunción	**||**	Baja

Ejemplo

```
int a, b
bool c;
a = 5;
b = 0;
c = a >= b && true;
Console.WriteLine("c: " + c);
```

Otros operadores

A nivel aritmético C# implementa otros operadores

Operador	Símbolo	Ejemplo
Incremento y decremento	`++  --`	`a++ //Equivale: a = a + 1`
Asignación compuesta	`+=  -=  *=  /=  %=`	`a-=2 //Equivale: a = a - 2`

Capítulo 3. Estructuras de control

Las estructuras de control presentes en C# son básicamente las mismas de lenguajes que provienen de C/C++ y que conservan la misma sintaxis, entre ellas, están las siguientes.

Estructuras de decisión

C# implementa el condicional y selector múltiple para la toma de decisiones.

Condicional if - else

La sentencia `if else` ejecuta un bloque de instrucciones en función del valor de verdad de una **expresión de comparación**. Los bloques de instrucciones se encierran entre llaves `{}`. El bloque `else` es opcional y se ejecuta si la expresión de comparación es falsa.

Ejemplo

```
if (y != 0) {
    z = x / y;
    Console.WriteLine("Cociente: " + z);
    z = x % y;
    Console.WriteLine("Módulo: " + z);
} else {
    Console.WriteLine("Error, división por 0");
}
```

Condicional if - else if - else

La sentencia `if - else` se puede abreviar cuando hay varios condicionales que deben anidarse; para ello se utiliza la instrucción `if - else if - else`.

Ejemplo

```
if (x > 0) {
    Console.WriteLine("x es positivo");
} else if (x < 0) {
    Console.WriteLine("x es negativo");
} else {
    Console.WriteLine("x es 0");
```

```
}
```

Operador ternario

C# también permite del **condicional ternario**, el cual abrevia la escritura del condicional en el caso particular cuando una variable es afectada tanto en la salida verdadera como en la salida falsa de la estructura de control.

Ejemplo

```
text = z % 2 == 0 ? "z es par" : "z es impar";
Console.WriteLine(text);
```

Selector múltiple switch case

La sentencia `switch` `case`, conocida como **selector múltiple** o **estructura según** en lógica de programación, ejecuta un bloque de instrucciones en función del valor de una variable. Es una alternativa al condicional que permite simplificar las condiciones, también muy útil en la creación de menús y en situaciones donde se depende de un valor de selección. Requiere del uso de la sentencia de bifurcación de control `break` para que no salte al siguiente caso desde donde se encuentra.

Ejemplo

```
switch (x) {
    case 1:
        Console.WriteLine("11");
        break;
    case 2:
        Console.WriteLine("22");
        break;
    case 3:
        Console.WriteLine("33");
        break;
    default:
        Console.WriteLine("Otro");
        break;
}
```

Ciclos o bucles

C# implementa los ciclos while, do while y for para los procesos repetitivos.

Ciclo while

Ejecuta un grupo de instrucciones en función de una expresión de comparación cuyo valor debe ser verdadero. El ciclo finaliza cuando ésta expresión se hace falsa.

Ejemplo

```
text = "";
lastText = "";
while (text != "0") {
    lastText = text;
    Console.Write($"Texto (0 para terminar): ");
    text = Console.ReadLine();
}
Console.Write($"Último texto: {lastText}");
```

Ciclo do while

Similar al ciclo while, con la diferencia que evalúa la condición al final, garantizando al menos una entrada a éste.

Ejemplo

```
do {
    lastText = text;
    Console.Write($"Texto (0 para terminar): ");
    text = Console.ReadLine();
} while (text != "0");
Console.Write($"Último texto: {lastText}");
```

Ciclo for

Es un caso particular del ciclo while que puede usarse cuando se conoce el número de iteraciones que se requieren procesar.

Ejemplo

```
for (int i = 1; i <= 10; i++) {
    Console.Write($"{i}\t");
}
```

Bifurcación de control

C# implementa break, continue, return, goto.

break

La sentencia `break` rompe una estructura de control. Se ilustró en el selector múltiple.

continue

La sentencia `continue` es similar a break, se utiliza en ciclos, y cuando se encuentra, ignora las demás sentencias de estos y continúa con la siguiente iteración.

Ejemplo

```
Console.WriteLine("\n");
while (lastText != "0") {
    Console.Write($"Texto (0 para terminar): ");
    lastText = Console.ReadLine();
    if (lastText == "*") {
        continue;
    }
    Console.Write($"Texto visible sí text es distinto de *: ");
}
```

goto

La instrucción `goto` transfiere el control a una instrucción marcada por una etiqueta. Un planteamiento apropiado del código hace innecesario el uso de este tipo de instrucciones y en lo posible deben evitarse.

Ejemplo

```
while (true) {
    Console.Write("Texto (* para terminar): ");
    lastText = Console.ReadLine();
    if (lastText == "*") {
        goto Terminar;
    }
}
Terminar:
        Console.Write("Ha finalizado el ingreso de datos");
```

return

La sentencia `return` devuelve el control a un punto donde hubo un llamado a un subprograma, acompañado posiblemente de un dato, haciendo que las demás instrucciones sean ignoradas. En la creación de métodos o funciones miembro en las clases se ilustrará el uso de **return**.

Sintaxis

```
return [valor];
```

Capítulo 4. Clases incorporadas. Clases y funciones estáticas

C# dispone de amplias bibliotecas para ayudar en el tratamiento de diversos casos, tales como el manejo de fechas y horas, funciones matemáticas, cadenas de caracteres, etc.; así mismo, como en cualquier lenguaje, también permite crear funciones definidas por el programador.

Clase Math

Proporciona constantes y métodos estáticos para trigonométricos, logarítmicos y otras funciones matemáticas comunes.

Clase String

Proporciona métodos para la manipulación de cadenas de caracteres.

Clases estáticas y sus miembros

Una clase estática es básicamente lo mismo que una clase no estática; la diferencia radica en que no se pueden crear instancias para una clase estática. En otras palabras, no se puede usar el operador `new` para crear una variable del tipo de clase. Dado que no hay ninguna variable de instancia, para tener acceso a los miembros de una clase estática se debe usar el nombre de la clase.

Los métodos (funciones) estáticos no dependen de los datos de la clase y son llamados usando directamente el nombre de la clase cuando se está por fuera de ella.

Ejemplo

```
using System;

namespace HelloWorld
{
    class HelloWorld
    {
        static void Main(string[] args)
        {
```

```
            Console.WriteLine("¡Hola, esto es C#!");
            double x, y, z;

            Console.Write("Número 1: ");
            x = Convert.ToDouble(Console.ReadLine());
            Console.Write("Número 2: ");
            y = Convert.ToDouble(Console.ReadLine());
            power(x, y);
            squareRoot(x);
        }

        static void power(double b, double e)
        {
            if (b == 0 && e == 0) {
                Console.WriteLine("Error, no se puede efectuar la potencia");
            } else {
                Console.WriteLine("Potencia: " + Math.Pow(b, e));
            }
        }

        static void squareRoot(double x)
        {
            if (x >= 0) {
                Console.WriteLine("Raíz cuadrada x: " + Math.Sqrt(x));
            } else {
                Console.WriteLine("Error, no se puede extraer raíz cuadrada");
            }
        }
    }
}
```

Capítulo 5. Excepciones y control de excepciones

Las características de control de excepciones del lenguaje C# ayudan a afrontar cualquier situación inesperada o excepcional que se produce cuando se ejecuta un programa. El control de excepciones usa las palabras clave `try`, `catch` y `finally` para intentar realizar acciones que pueden no completarse correctamente, para controlar errores cuando decide que es razonable hacerlo y para limpiar recursos más adelante. Las excepciones pueden ser generadas por Common Language Runtime (CLR), .NET, bibliotecas de terceros o el código de aplicación. Las excepciones se crean mediante el uso de la palabra clave `throw`.

En muchos casos, una excepción la puede no producir un método al que el código ha llamado directamente, sino otro método más bajo en la pila de llamadas. Cuando se genera una excepción, CLR desenreda la pila, busca un método con un bloque `catch` para el tipo de excepción específico y ejecuta el primer bloque `catch` que encuentra. Si no encuentra ningún bloque `catch` adecuado en cualquier parte de la pila de llamadas, finalizará el proceso y mostrará un mensaje al usuario.

En este ejemplo, un método prueba a hacer la división entre cero y detecta el error. Sin el control de excepciones, este programa finalizaría con un error **DivideByZeroException no controlada**.

```
public class ExceptionTest
{
    static double SafeDivision(double x, double y)
    {
        if (y == 0)
            throw new DivideByZeroException();
        return x / y;
    }

    public static void Main()
    {
        double a = 18, b = 0;
        double result;

        try {
            result = SafeDivision(a, b);
            Console.WriteLine("{0} divided by {1} = {2}", a, b, result);
        } catch (DivideByZeroException) {
            Console.WriteLine("Attempted divide by zero.");
        }
    }
}
```

Capítulo 6. Clases y espacios de nombres[1]

Un tipo que se define como una **`class`**, es un tipo de referencia. Al declarar una variable de un tipo de referencia en tiempo de ejecución, esta contendrá el valor **`null`** hasta que se cree expresamente una instancia de la clase mediante el operador **`new`** o que se le asigne un **objeto** creado en otro lugar.

Cuando se crea el objeto, se asigna suficiente memoria en el montón administrado para ese objeto específico y la variable solo contiene una referencia a la ubicación de dicho objeto. La memoria que un objeto usa reclama la funcionalidad de administración automática de memoria de CLR, lo que se conoce como recolección de elementos no utilizados.

Declarar clases

Las clases se declaran mediante la palabra clave `class` seguida por un identificador único. El nombre de la clase sigue a la palabra clave class. El nombre de la clase debe ser un **nombre de identificador** de C# válido. El resto de la definición es el cuerpo de la clase, donde se definen los datos y el comportamiento. Los campos, las propiedades, los métodos y los eventos de una clase se denominan de manera colectiva **miembros de clase**.

Creación de objetos

Aunque a veces se usan indistintamente, una **clase** y un **objeto** son cosas diferentes. Una clase define un tipo de objeto, pero no es un objeto en sí. Un objeto es una entidad concreta basada en una clase y, a veces, se conoce como una **instancia** de una clase. Se pueden crear objetos usando la palabra clave **`new`** seguida del nombre de la clase.

Espacios de nombres

.NET usa **espacios de nombres** para organizar sus clases. La palabra clave **`using`** se usa para hacer uso de las clases que se encuentren en dicho espacio de nombre.

Declarar sus propios espacios de nombres ayuda a controlar el ámbito de nombres de clase y métodos en proyectos de programación grandes. La palabra clave **`namespace`** permite declarar un espacio de nombres.

Ejemplo

[1] Clases en el sistema de tipos de C#. - C# | Microsoft Learn

Crear la clase `Calculator` dentro del espacio de nombres `HelloWorld`.

Programa C#: Calculator.cs

```
namespace HelloWorld
{
    class Calculator
    {
        private double number1;
        private double number2;

        public void setNumber1(double number)
        {
            this.number1 = number;
        }

        public void setNumber2(double number)
        {
            this.number2 = number;
        }

        public double getNumber1()
        {
            return this.number1;
        }

        public double getNumber2()
        {
            return this.number2;
        }

        public void message()
        {
            Console.WriteLine("Nueva clase");
        }
    }
}
```

Ejemplo

Crear el objeto `calc` de tipo `Calculator` y llamar a uno de sus métodos.

```
Calculator calc = new Calculator();
calc.message();
```

Ejemplo
Crear la clase `Vector` dentro del espacio de nombres `Arrays`.

Programa C#: Vector.cs

```
namespace Arrays
{
    class Vector
    {
        private int n;
        private double[] vec = new double[50];

        public Vector()
        {
            this.n = 0;
        }

        public int getN()
        {
            return this.n;
        }
    }
}
```

Ejemplo
Crear el objeto `vec` de tipo `Vector` y llamar a uno de sus métodos. Aquí se incluye el espacio de nombres donde está la clase `Arrays`.

Programa C#: Program.cs

```
using Arrays;

// ...

Vector vec = new Vector();
Console.Write($"n: {vec.getN()}");
```

Capítulo 7. Arreglos

C# implementa el manejo de arreglos de una dimensión (vectores), dos dimensiones (matrices) o más dimensiones (n-dimensionales, `n > 2`, en teoría, soporta un número arbitrario de dimensiones). Para acceder a sus elementos, se indica el nombre del arreglo y tantos índices enteros como dimensiones tenga el arreglo encerrados entre paréntesis angulares o corchetes `[ ]`.

Ejemplo
Crear una clase para manipular vectores y la respectiva interfaz para usar la clase.

Programa C#: Vector.cs

```
namespace Arrays
{
    class Vector
    {
        private int n;
        private double[] vec = new double[50];

        public Vector()
        {
            this.n = 0;
        }

        public int getN()
        {
            return this.n;
        }

        public void addVector(double datum)
        {
            this.vec[this.n] = datum;
            this.n++;
        }

        public void scrollVector()
        {
            for (int i = 0; i < this.n; i++) {
                Console.Write($"{this.vec[i]}\t");
            }
        }
    }
}
```

Interfaz para llamar la clase (se supone que existe más código asociado)

```
. . .
    static void menuVector()
    {
        string op;
        double datum;
        Vector vec = new();

        do {
            Console.WriteLine("Menú Vectores");
            Console.WriteLine("0. Terminar");
            Console.WriteLine("1. Agregar dato");
            Console.WriteLine("2. Total datos vector");
            Console.WriteLine("3. Mostrar vector");
            Console.Write("Opción: ");
            op = Console.ReadLine();
            Console.Write("\n");
            switch (op) {
                case "0":
                    Console.Write("Programa terminado");
                    break;
                case "1":
                    Console.Write("Ingrese un dato numérico: ");
                    datum = Convert.ToDouble(Console.ReadLine());
                    vec.addVector(datum);
                    break;
                case "2":
                    Console.Write($"Total datos (n): {vec.getN()}");
                    break;
                case "3":
                    vec.scrollVector();
                    break;
                default:
                    Console.Write("Opción no válida");
                    break;
            }
        } while (op != "0");
    }
. . .
```

Capítulo 8. Archivos y directorios

Un **archivo** está compuesto de una colección de datos y que se almacena en un dispositivo de almacenamiento, como un disco duro o una unidad de memoria USB. Los archivos pueden contener información en diferentes formatos, como texto, imágenes, audio o datos binarios[2]. En los ejemplos que se desarrollarán más adelante se trabajará con archivos de texto, también llamados archivos planos.

Un directorio, también conocido como carpeta, es una estructura utilizada para organizar y agrupar archivos de forma jerárquica. Los directorios pueden contener otros directorios (llamados también subdirectorios) y archivos. Esta organización facilita el acceso y la gestión de los datos en el SO.

Rutas de archivos y directorios

En C#, para acceder a un archivo o directorio, necesitamos especificar su ruta, que es la dirección de ubicación dentro del sistema de archivos. Hay dos tipos de rutas comunes: absoluta y relativa.

Rutas absolutas

Especifica la ubicación completa del archivo o directorio desde la raíz del sistema de archivos.

Por ejemplo en Windows:
`C:\Users\Usuario\Documentos\archivo.txt`

En Linux es algo así:
`/home/usuario/documentos/archivo.txt`

Nota
Es importante tener en cuenta que las rutas en Windows utilizan el carácter **** (*backslash*) como separador de directorios, mientras que en sistemas tipo Unix (como Linux y macOS), se utiliza el carácter **/** (*slash*).

[2] Un archivo binario contiene información codificada en formato binario, es decir, como una secuencia de bytes. Los archivos binarios son la base de la informática y se utilizan para almacenar y procesar información en computadoras.

Rutas relativas

Especifica la ubicación del archivo o directorio en relación con el directorio actual del programa. Una ruta relativa puede ser simplemente el nombre del archivo si está en el mismo directorio o incluir .. (punto punto) para indicar un directorio padre.

Por ejemplo:
`datos/archivo.txt`.

En este caso se especifica desde la ubicación actual una subcarpeta llamada `datos` y un archivo `archivo.txt` que se encuentra dentro de ella, indicando una ruta relativa que evita tener que indicar toda la ruta absoluta y que en términos prácticos, es muy útil para especificar subconjuntos de información clasificada en proyectos.

Gestión de directorios y archivos

La librería System.IO contiene las funcionalidades para tratar archivos y carpetas con C#, sin embargo, no es necesario especificar el espacio de nombres para trabajar con éstas.

Manipulación de directorios

Creación de directorios

Para crear un directorio o carpeta, invocamos al método `CreateDirectory()` de la clase `Directory`, para lo cual se crea una instancia de tipo `Directory`. Se debe indicar la ruta (carpeta) donde se creará el directorio. Si la carpeta ya existe, simplemente no sucede nada si se intenta crear de nuevo, así contenga información, aunque un nombre no válido si genera una excepción.

Sintaxis

```
Directory.CreateDirectory(path);
```

Manipulación de archivos

Creación de archivos

Para crear un archivo (en “blanco”), invocamos al método `Create()` de la clase `File`, para lo cual se crea una instancia de tipo `FileStream`. Se debe indicar la ruta (carpeta) donde se creará el archivo. Una vez creado el archivo, es recomendable cerrarlo: `objeto.Close()`. Si el archivo ya existe, lo sobreescribe, con lo cual se pierde lo que hubiese en él.

Sintaxis

```
FileStream fileVariable = File.Create(path);
fileVariable.Close();
```

Ejemplo

Crear una clase que permita la gestión de directorios y archivos.

Programa: Files.cs

```
namespace DataBases.FolderFile
{
    class Files
    {

        public Files()
        {

        }

        public static void CreateFolder(string path)
        {
            try {
                Directory.CreateDirectory(path);
                Console.WriteLine("Carpeta creada exitosamente");
            } catch (Exception ex) {
                Console.WriteLine($"Error al crear la carpeta \n
{ex.Message}");
            }
        }

        public static void CreateFile(string path)
        {
            try {
                FileStream file = File.Create(path);
                file.Close();
```

```
                Console.WriteLine("Archivo creado exitosamente");
            } catch (Exception ex) {
                Console.WriteLine($"Error al crear el archivo
\n{ex.Message}");
            }
        }

    }
}
```

Lectura de archivos

Esta operación consiste en recorrer el archivo por cada una de sus líneas. El método `File.ReadAllLines(path)` permite crear un arreglo unidimensional de cadenas, donde cada posición representa una línea del archivo.

Escritura de archivos

Esta operación consiste en agregar contenido al archivo. El método `File.WriteAllLines(path, data)` permite escribir un arreglo de cadenas sobre un archivo de texto.

Capítulo 9. Acceso a bases de datos con C#

C# permite la conexión con diversos motores de bases de datos, entre ellos, con SQL Server, MySQL y PostgreSQL, entre otros.

Conexión con MySQL

MySQL es un sistema de gestión de bases de datos relacionales de código abierto. Es un gestor de base de datos popular que permite a los usuarios almacenar y acceder a información de manera eficiente utilizando el Lenguaje de Consulta Estructurado (**SQL**).

MySQL está diseñado para ofrecer un rendimiento eficiente incluso con grandes conjuntos de datos y es escalable para adaptarse a las necesidades de usuarios en crecimiento.

MySQL puede descargarse libre y gratuitamente de la página de Oracle, su actual propietario; también se tiene como alternativa a **MariaDB**, un *fork* surgido de MySQL y desarrollado por el creador original de éste. También puede descargarse este gestor de BD que viene incluido en algún servidor de aplicaciones como XAMPP, Wamp u otros, muy populares para el trabajo con PHP y MySQL.

Descargar e instalar el conector MySQL .NET

Adicionamos el paquete NuGet de Oracle MySql.Data para conectar MySQL en la carpeta del proyecto.

```
>dotnet add package MySql.Data
```

Ejemplo
Crear una clase para conectarse a la BD y realizar el CRUD sobre alguna de las tablas.

Programa C#: Connect.cs

```
using MySql.Data.MySqlClient;

namespace DataBases.Connection
{
    class Connect
    {
        private string host;
        private string port;
        private string database;
        private string user;
```

```
        private string pass;
        private MySql.Data.MySqlClient.MySqlConnection? conn;

        public Connect()
        {
            host = "127.0.0.1";
            port = "3307";
            user = "root";
            pass = "";
            database = "academic-ai-001";
        }


        // Connect and disconnect database

        public void OpenConnection()
        {
            string connectionString;
            connectionString =
                $"server={host}; port={port}; uid={user};" +
                $"pwd={pass}; database={database}";
            try {
                conn = new
MySql.Data.MySqlClient.MySqlConnection(connectionString);
                conn.Open();
                Console.WriteLine("Conexión establecida satisfactoriamente");
            } catch (MySql.Data.MySqlClient.MySqlException ex) {
                Console.WriteLine(ex.Message);
            }
        }

        public void CloseConnection()
        {
            conn.Close();
        }


        // CRUD

        public void Create(string code, string firstName, string lastName)
        {
            OpenConnection();
```

```
            MySqlCommand command = new() {
                Connection = conn,
                CommandText =
                    $"INSERT INTO user(code, first_name, last_name)" +
                    $"VALUES('{code}', '{firstName}', '{lastName}')",
            };
            command.ExecuteNonQuery();
            CloseConnection();
        }

        public void Read()
        {
            OpenConnection();
            MySqlCommand command = new() {
                Connection = conn,
                CommandText = @"SELECT * FROM user",
            };

            var reader = command.ExecuteReader();
            Console.WriteLine("\nid\tNombre");
            Console.WriteLine("---------------");
            while (reader.Read()) {
                Console.WriteLine(
                    $"{reader.GetInt32("id")}\t" +
                    $"{reader.GetString("first_name")}"
                );
            }
        }

        public void Update(string code, string firstName, string lastName)
        {
            OpenConnection();
            MySqlCommand command = new() {
                Connection = conn,
                CommandText =
                    $"UPDATE user SET first_name='{firstName}', " +
                    $"last_name='{lastName}' " +
                    $"WHERE code='{code}'"
            };
            command.ExecuteNonQuery();
            CloseConnection();
        }
```

```
        public void Delete(string code)
        {
            OpenConnection();
            MySqlCommand command = new() {
                Connection = conn,
                CommandText =
                    $"DELETE FROM user " +
                    $"WHERE code='{code}'"
            };
            command.ExecuteNonQuery();
            CloseConnection();
        }


        // Getter & Setter

        public void SetHost(string host)
        {
            this.host = host;
        }

        public void SetPort(string port)
        {
            this.port = port;
        }

        public void SetUser(string user)
        {
            this.user = user;
        }

        public void SetPass(string pass)
        {
            this.pass = pass;
        }

        public void SetDatabase(string database)
        {
            this.database = database;
        }
```

```
        public string GetHost()
        {
            return host;
        }

        public string GetPort()
        {
            return port;
        }

        public string GetUser()
        {
            return user;
        }

        public string GetPass()
        {
            return pass;
        }

        public string GetDatabase()
        {
            return database;
        }
    }
}
```

Capítulo 10. Windows Forms con C#

Interfaz de usuario (UI) basada en Windows

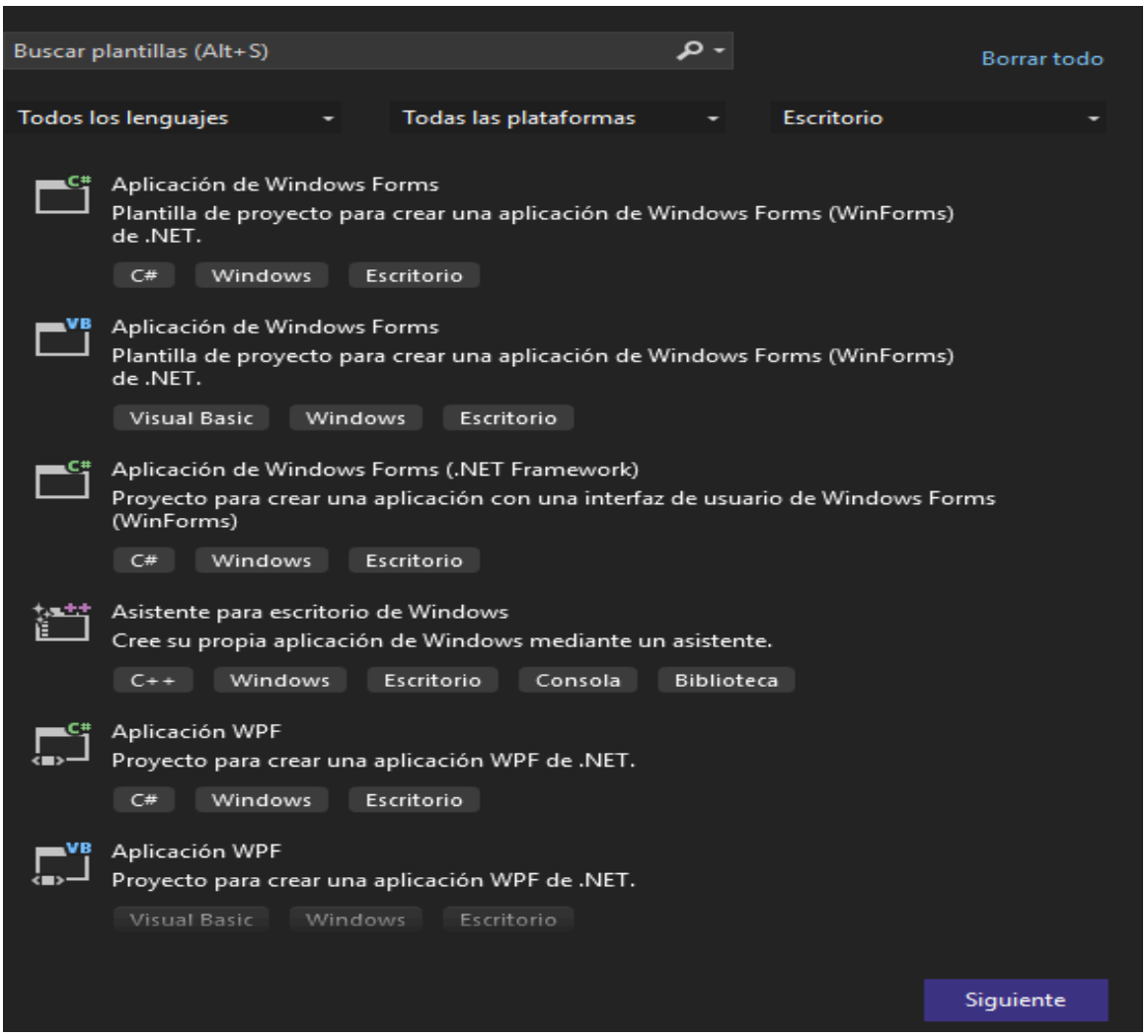
Versión Visual Studio IDE

Microsoft Visual Studio Community 2022 (64 bits) - Current Versión 17.9.6

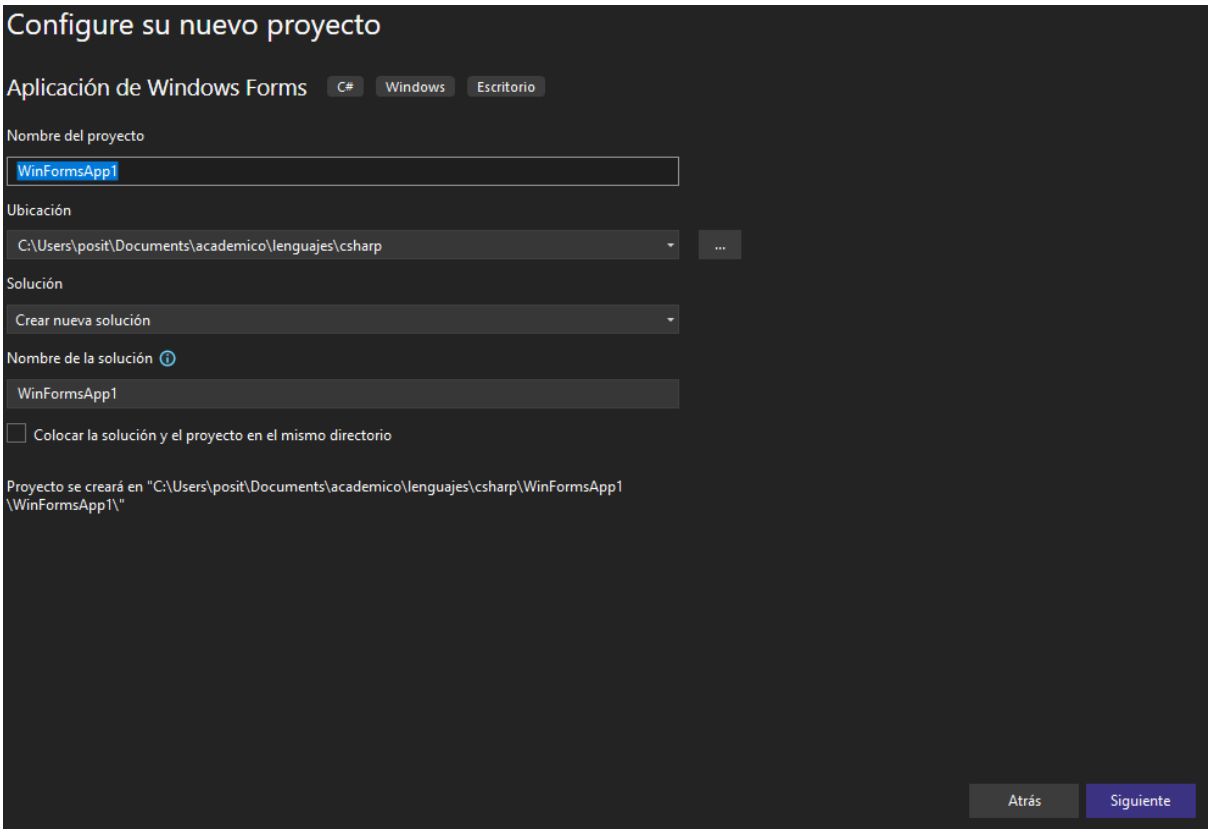
Crear un nuevo proyecto

Vamos a Archivo/Nuevo/proyecto

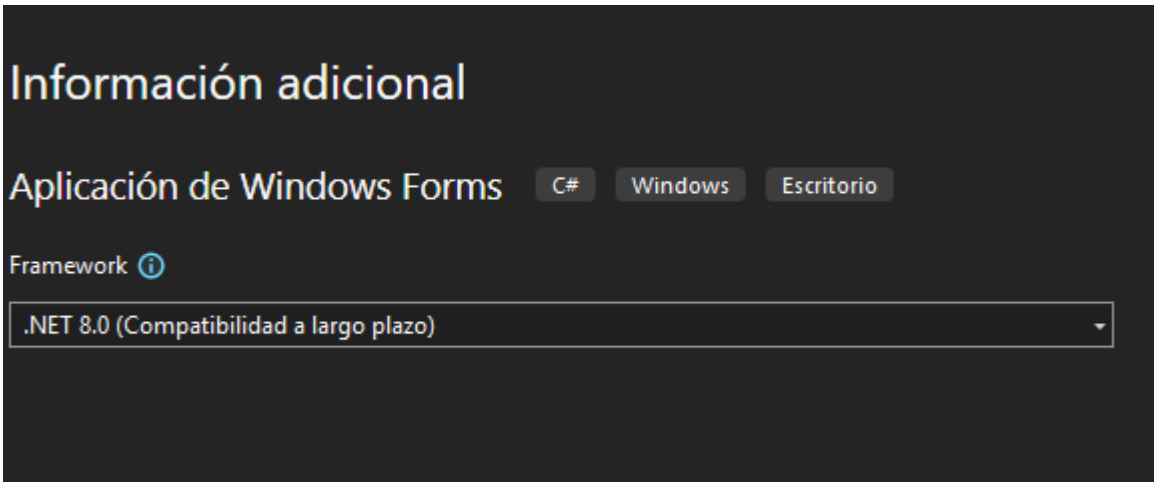
Seleccionamos Aplicación Windows Forms y Siguiente

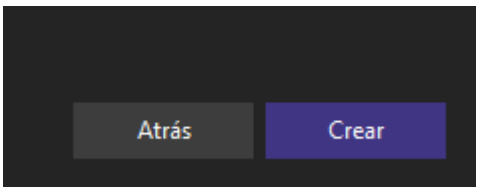


Configuramos el proyecto

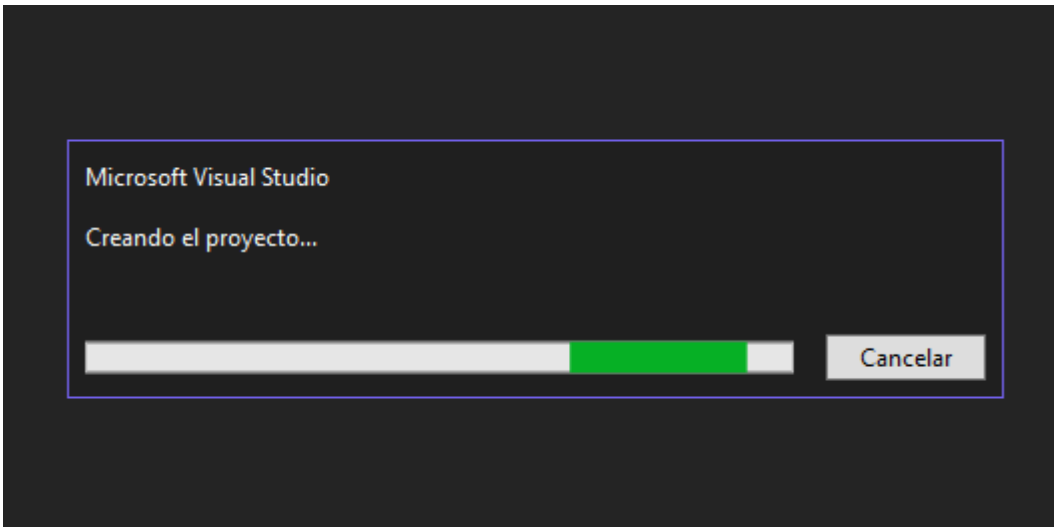


Añadimos información adicional y creamos el proyecto





El IDE muestra el progreso de la creación del proyecto



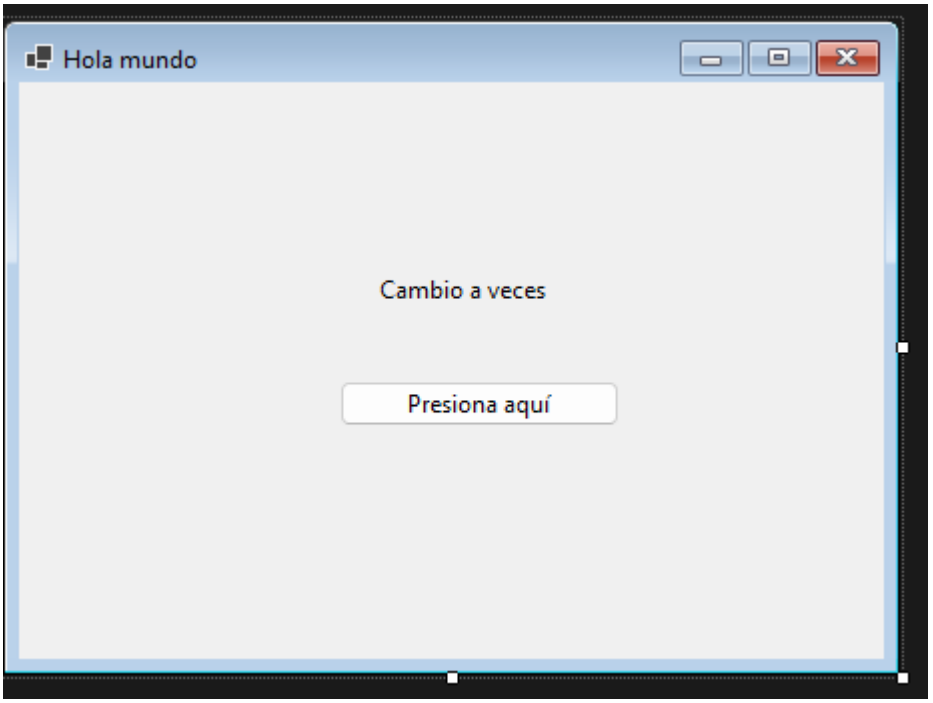
Algunas partes de la interfaz del IDE

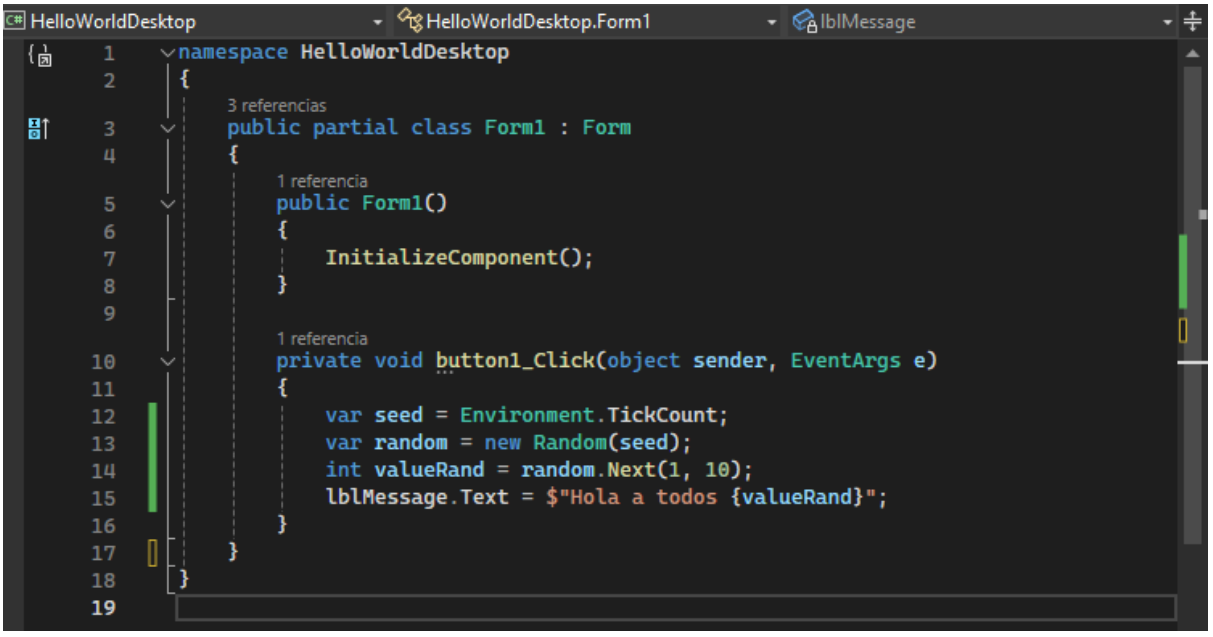
Por el menú Ver se encuentran las distintas ventanas y opciones a visualizar dentro del IDE. Algunas ventanas también se pueden visualizar presionando clic derecho en algunos elementos.

Vista diseño/código

El IDE dispone de una vista interactiva de diseño y código de los eventos asociados a los elementos de los formularios. Las vistas están disponibles desde los menús o las pestañas en la barra de herramientas.

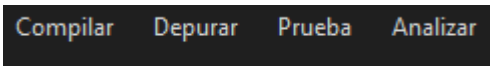
Ejemplo
Diseño de un formulario con una etiqueta (label) y un botón (button)

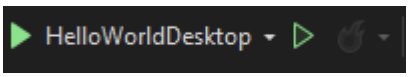




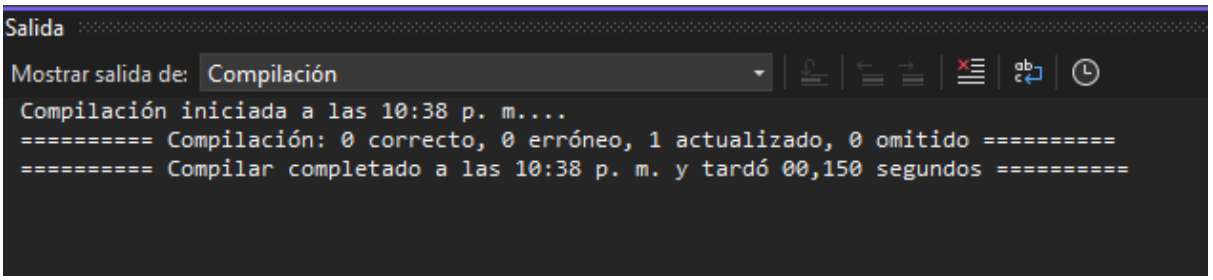
Compilar y ejecutar la aplicación

El IDE de .NET ofrece las opciones de menú, atajos del teclado (Ctrl+F5) o los iconos de la barra de herramientas



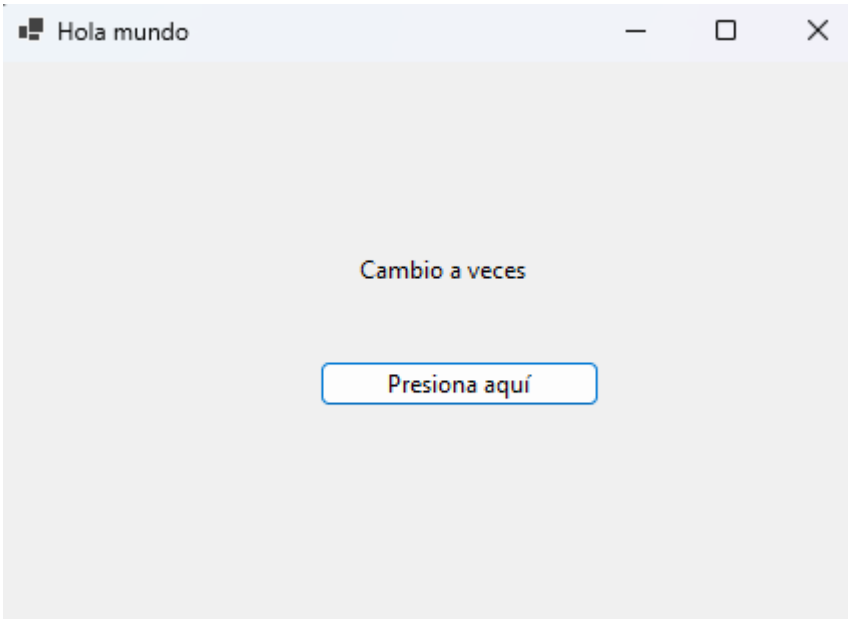


Luego de la compilación

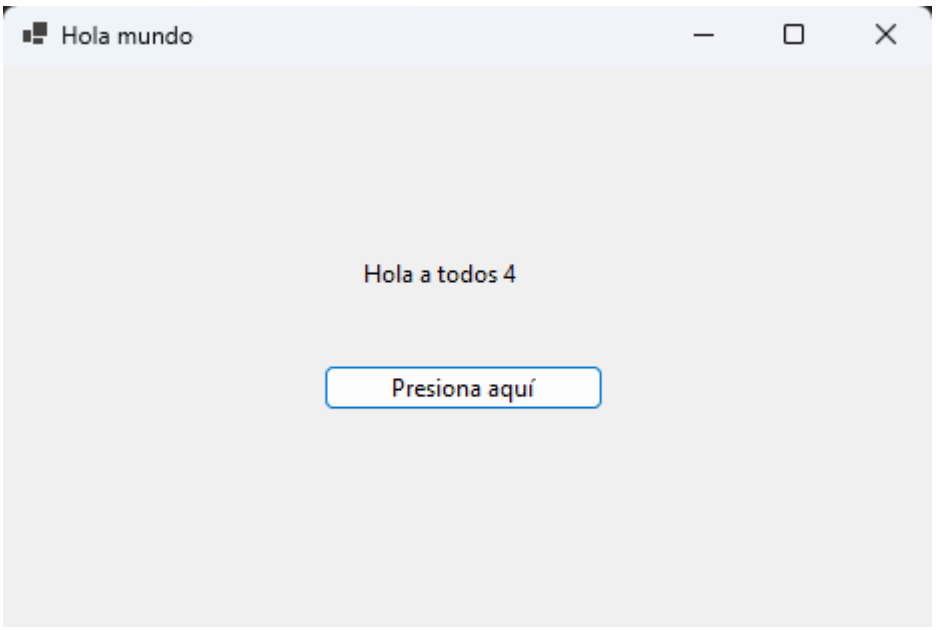


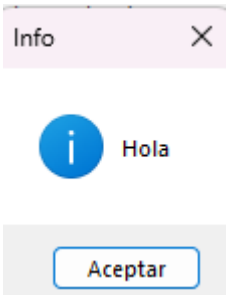
Ejemplo

Para el caso del ejemplo anterior, la salida se puede ver así al ejecutar



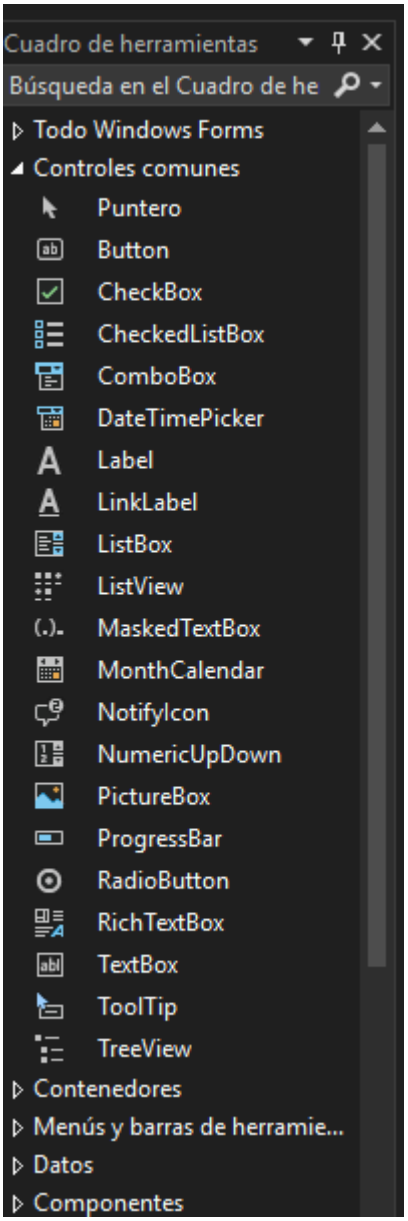
Al presionar clic sobre el botón, la etiqueta cambia a esto:





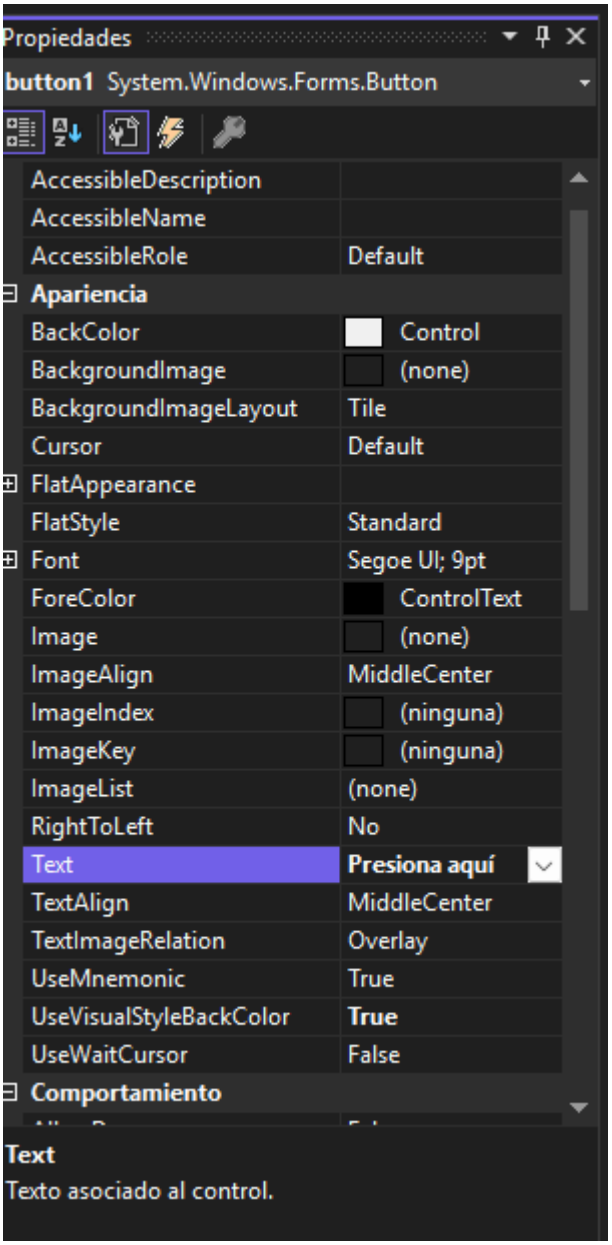
Cuadro de herramientas

Están los distintos controles para Windows que pueden agregarse a un formulario: botones, casillas de verificación, botones de opción, cuadros combinados, etiquetas, cuadros de texto, cuadros de listas, etc.



Propiedades

Muestra las propiedades (atributos) del control seleccionado en el formulario, incluyendo el mismo formulario



Ejemplo

Código del evento clic del botón del formulario

Programa C#: evento clic del botón `button1`

```
private void button1_Click(object sender, EventArgs e)
{
    var seed = Environment.TickCount;
    var random = new Random(seed);
    int valueRand = random.Next(1, 10);
    lblMessage.Text = $"Hola a todos {valueRand}";
    MessageBox.Show("Hola", "Info", MessageBoxButtons.OK,
MessageBoxIcon.Information);
}
```

Formularios MDI

Un formulario **MDI** (*Multiple Document Interface* - Interfaz de Múltiples Documentos) es un formulario que funciona como “administrador” de la aplicación, en otras palabras, es el formulario **principal** y **padre**. Dentro de él se abren los demás formularios hijos, y una vez se cierra éste, se cierra toda la aplicación; los formularios hijos pueden cerrarse y abrirse y la aplicación continúa en ejecución. Una aplicación sólo puede tener un formulario MDI.

Establecer formulario padre

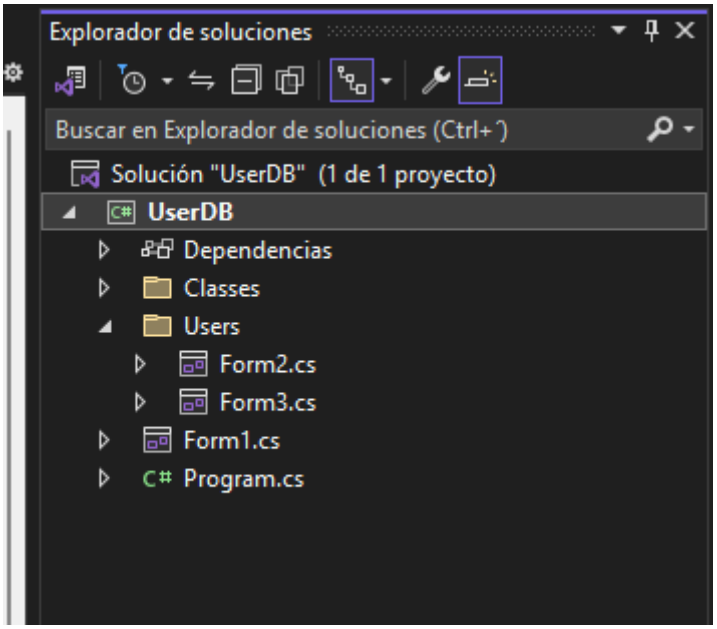
Para establecer un formulario padre (MDI) se habilita la propiedad **IsMdiContainer** para que pueda ser un contenedor de formularios hijos.

Establecer formulario hijo

Para que el formulario hijo se abra dentro de un formulario MDI en WinForms C#, se necesita establecer la propiedad **MdiParent** del formulario hijo al formulario MDI principal.

Agrupar formularios

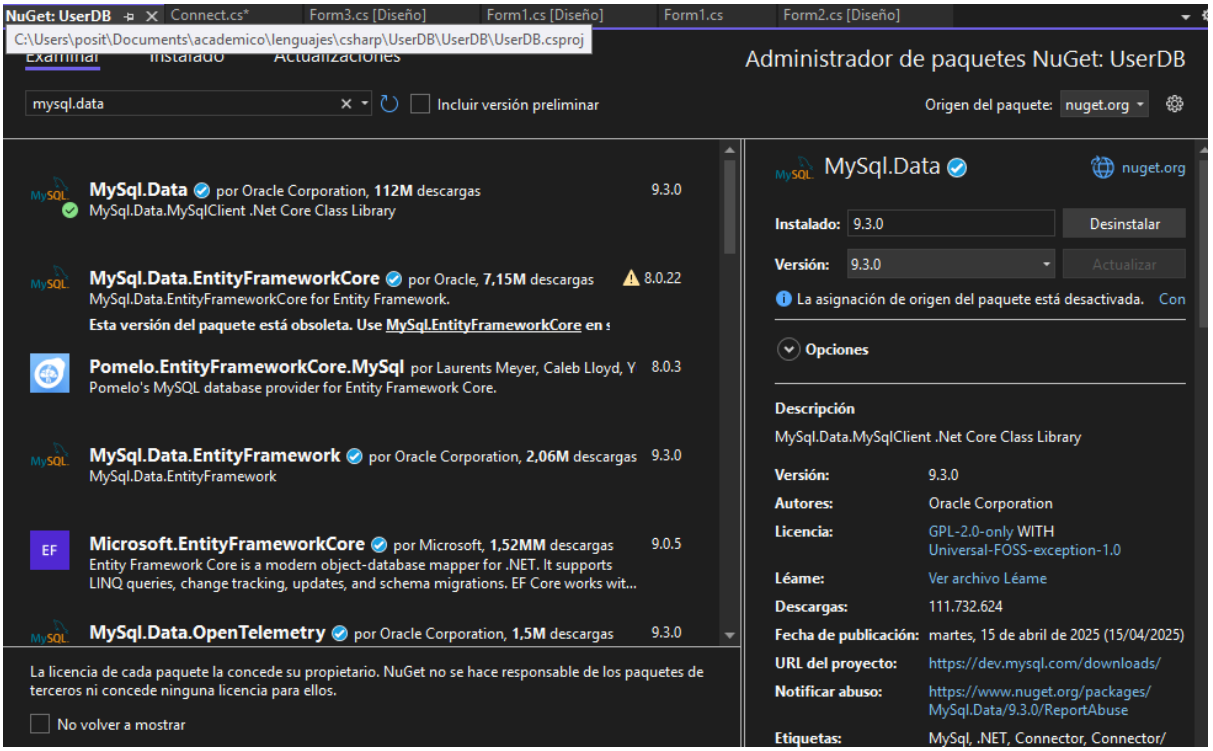
Un proyecto puede contener muchos formularios, por lo que es buena práctica agruparlos en carpetas dentro del proyecto a manera de módulos; así, se puede tener una carpeta usuarios con los formularios para los perfiles y los usuarios del sistema.



Bases de datos en proyectos Windows Forms

Instalar MySql.Data en Visual Studio

Para agregar el paquete **mysql.data** a un proyecto de Windows Forms en C#, primero se debe instalar a través de NuGet, el administrador de paquetes de Visual Studio. Luego, se agrega la referencia al proyecto. Finalmente, se importa la biblioteca en el código C# para usarla.

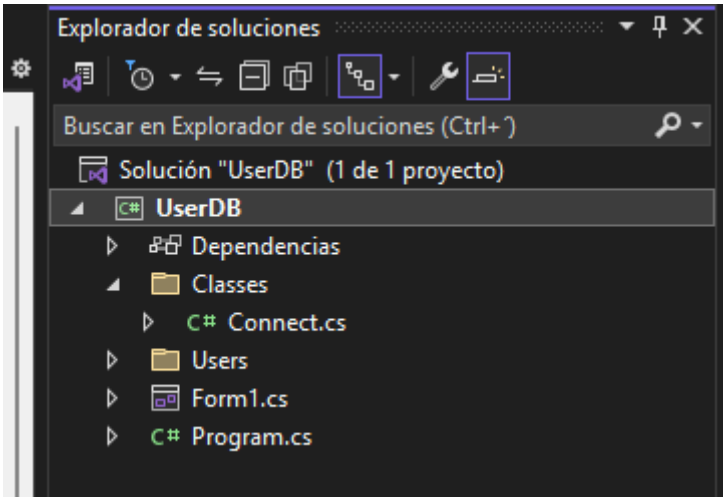


Pasos

1. Instalar el paquete NuGet
    - En el Explorador de soluciones, clic derecho en el nombre del proyecto y seleccionar "Administrar paquetes NuGet".
    - Buscar "MySql.Data" en la pestaña "Buscar" y seleccionar la versión que se necesita.
    - Clic en "Instalar".
2. Agregar la referencia
    - En el Explorador de soluciones, clic derecho en "Referencias" en el proyecto.
    - Seleccionar "Agregar referencia...".
    - En la ventana de diálogo, busca "MySql.Data.MySqlClient" y selecciónalo.
    - Haz clic en "Aceptar".
3. Importar la biblioteca en el código C#:
    - Agregar la siguiente línea al inicio del archivo de código C# donde se usará la conexión a MySQL: `using MySql.Data.MySqlClient;`

Uso de clases para gestionar la base de datos

Podemos crear los modelos para cada caso agrupados en clases dentro de una carpeta que se crea dentro del proyecto.

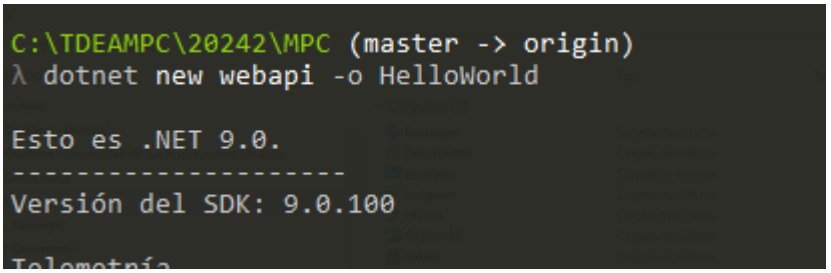

Explorador de soluciones
Buscar en Explorador de soluciones (Ctrl+')
Solución "UserDB" (1 de 1 proyecto)
UserDB
Dependencias
Classes
Connect.cs
Users
Form1.cs
Program.cs

Crear un proyecto web api en C# .NET

Sintaxis

```
>dotnet new webapi -o nombre-proyecto
```

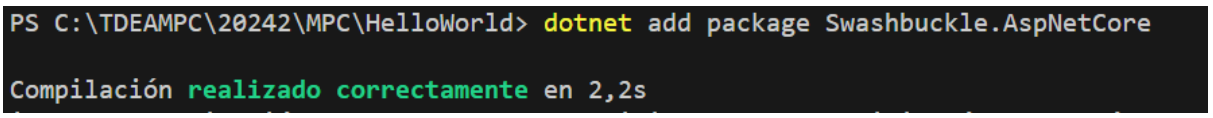
Ejemplo



Agregar paquete para integración de C# con Swagger

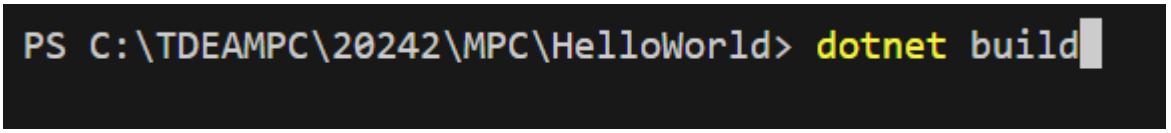
Ubicados dentro de la carpeta del proyecto ejecutamos el siguiente comando.

```
>dotnet add package Swashbuckle.AspNetCore
```



Compilar la aplicación

```
>dotnet build
```



Ejecutar (correr) la aplicación

```
>dotnet run
```

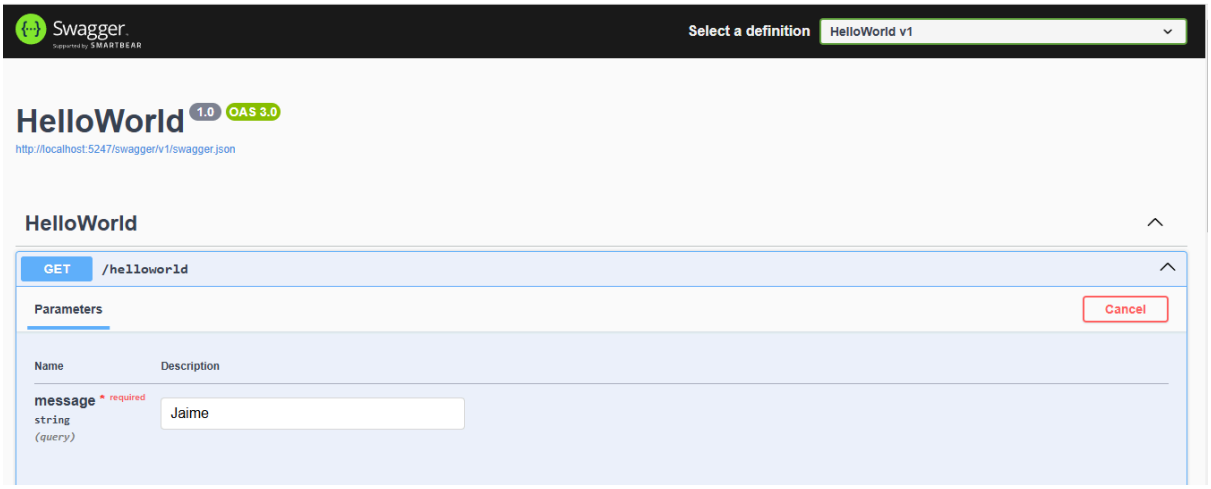
```
PS C:\TDEAMPC\20242\MPC\HelloWorld> dotnet run
Usando la configuración de inicio de C:\TDEAMPC\20242\MPC\HelloWorld\Prope
rties\launchSettings.json...
Compilando...
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5247
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
```

Publicar el proyecto

Una vez se ejecute la aplicación, escribimos en el navegador una URL similar a la que se muestra teniendo presente que el puerto varía de acuerdo al proyecto.

http://localhost:5247/swagger

Esta url nos lleva a esta página, la cual permite verificar el funcionamiento de la api, entre otras funcionalidades de las que dispone **swagger**.



Swagger

Es un conjunto de herramientas de software de código abierto para diseñar, construir, documentar, y utilizar servicios web RESTful. Fue desarrollado por SmartBear Software e

incluye documentación automatizada, generación de código, y generación de casos de prueba.

Swagger es una plataforma de diseño, documentación y testing de APIs que se enfoca en la creación de especificaciones de API utilizando la especificación OpenAPI. Su principal función es definir y describir APIs de manera estandarizada, generando documentación interactiva y permitiendo la prueba y depuración de APIs

dotnet new console -o CifradoTransposicionUnidimensionalCSharp

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
// Learn more about configuring Swagger/OpenAPI at
https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();

app.MapGet("/cifradotransposicionunidimensional", (string Mensaje) =>
{
    return CifradoDecifrado(Mensaje);
})
.WithName("GetCifradoTransposicionUnidimensional")
.WithOpenApi();

app.Run();

static string CifradoDecifrado(string Mensaje) {
    int i;
    int j;
    string mensajeentrada;
```

```
    string mensajematrizorganizadosalida;
    string mensajematrizsalida;
    string mensajematriztranspuestasalida;
    int tamanomensajeentrada;
    // Definir Variables
    // Entrada de Informacion
    //Console.WriteLine("Ingrese Mensaje");
    //mensajeentrada = Console.ReadLine();
    mensajeentrada = Mensaje;
    tamanomensajeentrada = mensajeentrada.Length;
    Console.WriteLine("Mensaje de Entrada: "+mensajeentrada);
    Console.WriteLine("Tamaño Mensaje Entrada: "+tamanomensajeentrada);
    // Dimensionar Estructura mensajeentrada
    string[] matrizmensajeentrada = new string[tamanomensajeentrada];
    string[] matrizmensajetranspuesta = new string[tamanomensajeentrada];
    string[] matrizmensajeorganizado = new string[tamanomensajeentrada];
    // 1. Asignacion de Informacion
    i = 1;
    while (i<=tamanomensajeentrada) {
        matrizmensajeentrada[i-1] = mensajeentrada.Substring(i-1, i-i+1);
        i = i+1;
    }
    // 2. Salida Estructura Mensaje Entrada
    i = 1;
    mensajematrizsalida = "";
    while (i<=tamanomensajeentrada) {
        mensajematrizsalida =
mensajematrizsalida+matrizmensajeentrada[i-1];
        i = i+1;
    }
    Console.WriteLine(mensajematrizsalida);
    // 3. Procesamiento de Informacion
    i = tamanomensajeentrada;
    j = 1;
    while (i!=0) {
        matrizmensajetranspuesta[j-1] = matrizmensajeentrada[i-1];
        i = i-1;
        j = j+1;
    }
    i = 1;
    mensajematriztranspuestasalida = "";
    while (i<=tamanomensajeentrada) {
        mensajematriztranspuestasalida =
mensajematriztranspuestasalida+matrizmensajetranspuesta[i-1];
        i = i+1;
    }
    Console.WriteLine(mensajematriztranspuestasalida);
```

```
    // 4. Organizacion de Informacion
    i = tamanomensajeentrada;
    j = 1;
    while (i!=0) {
        matrizmensajeorganizado[j-1] = matrizmensajetranspuesta[i-1];
        i = i-1;
        j = j+1;
    }
    i = 1;
    mensajematrizorganizadosalida = "";
    while (i<=tamanomensajeentrada) {
        mensajematrizorganizadosalida =
mensajematrizorganizadosalida+matrizmensajeorganizado[i-1];
        i = i+1;
    }
    Console.WriteLine(mensajematrizorganizadosalida);
    // Salida de Informacion
    Console.WriteLine("Mensaje Original: "+mensajeentrada);
    Console.WriteLine("Mensaje Matriz Original: "+mensajematrizsalida);
    Console.WriteLine("Mensaje Matriz Transpuesta:
"+mensajematriztranspuestasalida);
    Console.WriteLine("Mensaje Matriz Organizada:
"+mensajematrizorganizadosalida);

    return "Mensaje Original: "+mensajeentrada + " - " +
            "Mensaje Matriz Original: "+mensajematrizsalida + " - " +
            "Mensaje Matriz Transpuesta: "+mensajematriztranspuestasalida
+ " - " +
            "Mensaje Matriz Organizada: "+mensajematrizorganizadosalida;
}
```

http://localhost:5117/swagger/

Referencias

Sitios para ejecución de código

Try .NET | Runnable .NET code on your site

Online C# Compiler (Editor) - Programiz

.NET y C#

.NET Core - Wikipedia, la enciclopedia libre

C Sharp - Wikipedia, la enciclopedia libre

Aprendizaje para .NET | Microsoft Learn

El sistema de tipos de C# | Microsoft Learn

Tipos numéricos integrales | Microsoft Learn

Tipos numéricos de punto flotante - C# reference | Microsoft Learn

tipo bool - C# reference | Microsoft Learn

C# Data Types

Secuencias de escape | Microsoft Learn

Excepciones

Excepciones y control de excepciones | Microsoft Learn

Cadenas

Cadenas - C# | Microsoft Learn

Archivos en C#

Todo sobre la Manipulación de Archivos y Directorios en C#

Bases de datos con C#

NuGet Gallery | MySql.Data 9.3.0

Windows Forms

Creación de una aplicación de Windows Forms con C# - Visual Studio (Windows) | Microsoft Learn

Llenar nuestro DataGridView (tabla) con MySQL (MariaDB) en C# [03]

Swagger

Swagger (software) - Wikipedia, la enciclopedia libre