Tema 2

Descargar estos apuntes

Índice

Creando aplicaciones en .NET en la actualidad

- Usaremos el kit de desarrollo .NET Core SDK
- Algún **IDE** como Visual Studio Community que me permitirá realizar los desarrollos de forma gratuita. Aunque su instalación es muy pesada y solo está disponible para Windows y Mac.
- Si queremos **trabajar en cualquier SO e incluso On-Line** la mejor opción seria **Visual Studio Code** pues es un editor **Open Source** bastante ligero con una gran cantidad de plugins creados por la *'comunidad'*. Aunque no es tán potente como un IDE, nos permitirá afrontar una gran cantidad de desarrollos en lenguajes y tecnologías diferentes. El uso de un IDE hace transparente para nosotros los comandos y los flujos de trabajo del desarrollo, aunque es interesante tener claro como son los procesos para así gestionar su automatización a través de herramientas de CI/CD.

Creación de un proyecto de consola

La gestión de proyectos se hace con la CLI **dotnet**, así pues para crear un proyecto desde consola usaremos dotnet new y en concreto para crea una aplicación de consola ejecutaremos `dotnet new console -n <nombre>`

```
B:\>dotnet new console -n ejemplo

The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on ejemplo\ejemplo.csproj...
  Determinando los proyectos que se van a restaurar...
  Se ha restaurado B:\ejemplo\ejemplo.csproj (en 100 ms).

Restore succeeded.
```

Tras ejecutar este comando, se creará un proyecto definido en el fichero **ejemplo.csproj** donde, si no especificamos la versión de .NET, nos lo creará con la que hemos instalado y el lenguaje será C# con la última versión posible de esa plataforma.
Como podemos ver, para nuestro ejemplo **la plataforma de destino es .NET Core 3.1 y la versión de C# es la 8.0**

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.1</TargetFramework>
    <LangVersion>8.0</LangVersion>
  </PropertyGroup>

</Project>
```

> Para más información ver la documentación oficial.

Definiendo una solución para asociar nuestro proyecto

Un proyecto debería estar asociado a una solución. Así mismo, una solución podría tener más de un archivo asociado. De hecho, si hubiéramos creado el proyecto de consola con el Visual Studio Community nos crearía por defecto una solución a la que estaría asociado el mismo.

- Con `dotnet new sln -n <nombre>` crearemos una solución.
- Con `dotnet sln <nombre>.sln add <nombre>.csproj` añadiremos una proyecto a una solución.

```
B:\>cd ejemplo
B:\ejemplo>dotnet new sln -n ejemplo
The template "Solution File" was created successfully.

B:\ejemplo>dotnet sln ejemplo.sln add ejemplo.csproj
Se ha agregado el proyecto "ejemplo.csproj" a la solución.
B:\ejemplo>dotnet sln list
Proyectos
---------
ejemplo.csproj
```

Además, siempre es interesante añadir un fichero de metainformación para el **control de versiones de git**.

```
B:\ejemplo>dotnet new gitignore
The template "dotnet gitignore file" was created successfully.
```

También es posible compilar el proyecto y generar el ensamblado con `dotnet build` y ejecutar el programa usando `dotnet run`. Aunque este proceso lo podremos hacer a través de **VS Code** usando la **extensión para C# de OmniSharp**.

```
B:\ejemplo>dotnet build
Microsoft (R) Build Engine versión 16.6.0+5ff7b0c9e para .NET Core
Copyright (C) Microsoft Corporation. Todos los derechos reservados.
  Determinando los proyectos que se van a restaurar...
  ejemplo -> B:\ejemplo\bin\Debug\netcoreapp3.1\ejemplo.dll
Compilación correcta.
    0 Advertencia(s)
    0 Errores
Tiempo transcurrido 00:00:00.64

B:\ejemplo>dotnet run
Hello World!
```

Para más información ver la integración con VS Code.

> **Importante:** Deberemos configurar el parámetros console al valor **integratedTerminal**.

```
"configurations": [
        {
            ...
            "console": "integratedTerminal",
            ...
        },
```

💡 '*Tips*' de teclado para Visual Studio (y otros editores)

A lo largo del curso se darán diferentes tips de uso del teclado y atajos para refactorizar.

Configuraremos los atajos en VS Code como en Visual Studio con `Ctrl+K`, `Ctrl+M` e instalaremos la extensión "*Visual Studio Keymap*".

Acción	Teclado	Acción	Teclado
Salta letra	`→/←`	Salta palabra	`Ctrl+→` o `Ctrl+←`
Salta al inicio	`Ctrl+Inicio`	Salta al final	`Ctrl+Fin`
Selecciona letra	`Shift+→` o `Shift+←`	Selecciona palabra	`Shift+Ctrl+→` o `Shift+Ctrl+←`
Selecciona línea	`Inicio`, `Shift+↓`	Selecciona **columna**	`Shift+Alt+↓` y/o `Ctrl+Alt+→`
Selecciona hasta inicio	`Ctrl+Shift+Inicio`	Selecciona todo	`Ctrl+A` o `Ctrl+Inicio`, `Ctrl+Shift+Fin`
Selecciona hasta fin	`Ctrl+Shift+Fin`	Copiar linea o selección	`Ctrl+C`
Mover línea o selección actual	`Alt+↑` o `Alt+↓`	Pegar	`Ctrl+V`
Cortar linea o selección	`Ctrl+X`	Duplica selección	`Ctrl+D`
Pegar sin formato	`Ctrl+Shift+V`	Des-tabula selección	`Shft+Tab`
Tabula selección	`Tab`	Buscar en todo	`Ctrl+Shift+F`
Buscar	`Ctrl+F`	Buscar y Reemplazar	`Ctrl+H`
Buscar siguiente	`F3`	Formatea Código	`Ctrl+K`, `Ctrl+D`
Buscar y Reemplazar en todo	`Ctrl+Shift+H`	Des-comenta Selección	`Ctrl+K`, `Ctrl+U`
Comenta Selección	`Ctrl+K`, `Ctrl+C`	Ejecuta sin depurar	`Ctrl+F5`
Ir a siguiente error o aviso	`F8`	**Re-nombra identificador**	`Ctrl+R`, `Ctrl+R`
Ejecuta depurando	`F5`	Ir a la definición	`F12`
Ir a la declaración	`Ctrl+F12`	**Ver Información sobrecargas**	`Ctrl+shift+Space`
Ver Sugerencia contextual	`Ctrl+Space`	**Ver Refactorización contextual**	`Ctrl+.`
Muestra información in-line	`Ctrl+K`, `Ctrl+I`		

Estructura de una programa

Además, de estos apuntes. Puedes seguir, de forma complementaria, este Tutorial Básico para aprender las bases de lenguaje.

Nuestro primer programa Hello World

Si abrimos el fichero **Program.cs** que se ha creado junto a nuestro proyecto podrás ver el siguiente código:

```
using System;
namespace ejemplo
{
    class Program
    {
        static void Main()
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

- Una aplicación C# es una colección de clases, estructuras y tipos.
- Todo el código escrito en C# se ha de escribir **dentro** de una definición de clase.

Si TODO son clases ¿Cómo se define una?

- Utilizaremos la palabra reservada **class**
- La definiremos con la siguiente sintaxis...

```
class Identificador
{
  <miembros>
}
```

- Es posible definir varias clases en un archivo, pero lo normal es que haya una por archivo o fichero fuente.

¿Qué es el Main?

- Es el punto de entrada de un programa.
- Sólo puede haber uno.
- Deberemos declararlo de las siguientes maneras, pero siempre con el modificador **static**:

```
static void Main()
static int Main()
static int Main(string[] args)
static void Main(string[] args)
...
```

Espacios de Nombres

- Aunque **lo vamos a ver en más profundidad cuando veamos el concepto de paquete** y librería. Puede ser interesante hacer esta primera introducción.
- Podemos decir que un espacio de nombres es un **identificador** o etiqueta que me ayuda a agrupar y clasificar un conjunto de definiciones de tipos relacionadas.
- En C# se definen de la siguiente manera:

```
namespace Espacio1 // Espacio1 es el identificador o etiqueta del namespace
{
  // Definiciones de tipos dentro del namespace Espacio1
  namespace Espacio11
  {
    // Definiciones de dentro del namespace Espacio1.Espacio11
    class MiTipoEnEspacio11
    {
      ...
    }
  }
}
```

- También podría hacer...

```
namespace Espacio1.Espacio12
{
    // Definiciones de dentro del namespace Espacio1.Espacio12
    class MiTipoEnEspacio12
    {
      ...
    }
  }
}
```

Nombre Completamente Calificado

- Para hacer referencia a un tipo, como por ejemplo una determinada clase, debo indicar la jerarquía de espacios de nombres seguidos por puntos. A esto se le denomina Nombre Completamente Calificado o **NCC**.

```
Espacio1.Espacio11.MiTipoEnEspacio11
```

- Esto permite que diferente librerías puedan dar el mismo nombre a sus tipos o clases y poder distinguir entre una y otra.

```
  System.Console.Writeln("Hola Mundo");
  MisDefiniciones.Console.Escribe("Hola Mundo");
```

Cláusula Using

- Para no tener que poner siempre el NCC cuando esté usando tipos y clases definidas dentro de un determinado namespace, puedo usar la cláusula **using**.

```
using System;
class Program
{
    static void Main()
    {
        Console.WriteLine("Hello World!");
        // en lugar de tener que poner.
        System.Console.WriteLine("Hello World!");
    }
}
```

- Aunque **no es muy común**, podremos definir **alias** para clases repetidas en diferentes namespaces mediante la siguiente sintaxis: `using <alias> = <NCC>;`

```
using System;
using MiConsole = MisDefiniciones.Console;
...
Console.Writeln("Hola Mundo"); // Equivale a System.Console.Writeln("Hola Mundo");
MiConsole.Escribe("Hola Mundo"); // MisDefiniciones.Console.Escribe("Hola Mundo");
```

Salida y Entrada Estándar

Salida Por Consola

- Utilizaremos la clase **Console** definida en el namespace **System**.
- Esta clase solo tiene sentido usarla en aplicaciones de consola.
- Para la salida estándar utilizaremos los métodos Write y WriteLine (añade un salto de línea al final) que además me permiten formatear una salida numérica o de texto con parámetros.

```
System.Console.Write("Total: ");
System.Console.WriteLine(200);
// Salida
// Total: 200<Salto Línea>
```

Formatos compuestos de texto

La sintaxis es: `Console.Write("<texto>{N,M:FormatString}<texto>");`

- **N** indicará el dato a sustituir entre llaves y con el número de orden del parámetro que le corresponde empezando desde 0.
- **M** la alineación. (+ A la derecha y – A la Izquierda)
- **FormatString** indicará como se deben mostrar un dato numérico.
  Podrá tomar los siguientes valores:

Valor	Significado
C	Muestra el número como una unidad monetaria, usando el símbolo y las convenciones de la moneda local.
D	Muestra el número como un entero decimal.
E	Muestra el número usando notación exponencial (científica).
F	Muestra el número como un valor en coma fija.
G	Muestra el número como un valor entero o en coma fija, dependiendo del formato que sea más compacto.
N	Muestra el número con comas incorporadas.
X	Muestra el número utilizando notación hexadecimal.

Ejemplos formato compuesto

1. ```
   Console.WriteLine("\"Justificación a la {0} con anchura 10: {1,-10}\"", "izquierda", 99); `
   // Salida
   // "Justificación a la izquierda con anchura 10: 99      "
   ```

2. ```
   Console.WriteLine("\"Justificación a la {0} con anchura 10: {1,10}\"", "derecha", 99);
   // Salida
   // "Justificación a la derecha con anchura 10:         99"
   ```

3. ```
   Console.WriteLine("Formato entero - {0:D5}", 88);
   // Salida
   // Formato entero - 00088
   ```

4. ```
   Console.WriteLine("Formado exponencial - {0:E}", 888.8);
   // Salida
   // Formado exponencial – 8,888000E+002
   ```

5. ```
   Console.WriteLine("Formato de punto fijo - {0:F3}", 888.8888);
   // Salida
   // Formato de punto fijo – 888,889
   ```

6. ```
   Console.WriteLine("Formato general - {0:G}", 888.8888000);
   // Salida
   // Formato general – 888,8888
   ```

7. ```
   Console.WriteLine("Formato de número - {0:N}", 8888888.8);
   // Salida
   // Formato de número – 8.888.888,80
   ```

8. ```
   Console.WriteLine("Formato hexadecimal - {0:X4} Hex", 88);
   // Salida
   // Formato hexadecimal - 0058 Hex
   ```

Entrada Por Consola

Entrada de texto

Para la entrada estándar utilizaremos los métodos **Read** que lee el siguiente carácter desde stdin (-1 si esta vacía) y **ReadLine** que lee hasta final de línea.

```
string texto = Console.ReadLine();
Console.WriteLine(texto);
```

> **Nota:** Solo funcionará si leemos cadenas.

Entrada de datos otros tipos

Todos los tipos del CTS tienen definido el método tipo `Parse(string texto)` que transforma de cadena al tipo.

```
int entero = int.Parse(Console.ReadLine());
double real = double.Parse(Console.ReadLine());
```

> **Nota:** Si el texto leído no se puede convertir al tipo correspondiente se producirá un error de ejecución y el programa finalizará.

Definición de Identificadores en C#

- Con ellos damos nombre a elementos de un programa, como clases, constantes, variables, métodos, etc…
- Un identificador está formado por letras (a|b|c|...|z|A|B|C|...|Z), y dígitos (0|1|2|3|...|9), la única restricción que se nos pone, es que **el primer carácter empiece por una letra** o signo de subrayado '_'.
- C# Distingue entre mayúsculas y minúsculas.
- Es valido cualquier carácter UNICODE como la 'ñ'.
- No son válidos caracteres propios del lenguaje como "", '#', '?', '@', etc…
- Es importante utilizar una notación o convenio al nombrar, y siempre que haya algún tipo de unidad indicarlo.

> **Resumen:** Deberían ser **sustantivos** significativos, evitando contracciones en identificadores cortos y redundancia dentro del contexto. Buscaremos pues **nombres autodocumentados**.

Nomenclaturas más habituales

- **PascalCasing**
  - Consiste en poner en mayúscula el primer carácter de todas las palabras.
  - Se utiliza para clases, métodos, propiedades, enumeraciones, interfaces, campos constantes y de sólo lectura, espacios de nombres, propiedades, etc…
    Ejemplo: void **I**nicializa**D**atos(…);
- **camelCasing**
  - Consiste en poner en mayúscula el primer carácter de todas las palabras excepto la primera.
  - Se utiliza para variables que definan campos, parámetros y variables miembro privadas.
    Ejemplo: int **n**umero**D**e**P**lantas;

'INCORRECTO'	'CORRECTO'
6Caja	Caja6
tot	TotalUnidades
ventlun	VentasLunes
longitudSalto	longitudSalto_m

Palabras reservadas del lenguaje C#

- Son identificadores o cláusulas propias del lenguaje.
- No se pueden utilizar como identificadores. En realidad si se puede, siempre y cuando las precedamos del carácter '@', pero no deberíamos usarlas nunca.
- Son siempre minúsculas.
- Los **editores de C# normalmente nos las resaltarán** con algún color determinado.

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

Tipos De Datos Básicos

C# es (**type-safe**), esto significa que se garantizan los valores almacenados.

Tipo	Descripción	Bits	Rango de valores	Alias
SByte	Bytes con signo	8	[-128, 127]	sbyte
Byte	Bytes sin signo	8	[0, 255]	**byte**
Int16	Enteros cortos con signo	16	[-32.768, 32.767]	short
UInt16	Enteros cortos sin signo	16	[0, 65.535]	ushort
Int32	Enteros normales	32	[-2.147.483.648, 2.147.483.647]	**int**
UInt32	Enteros normales sin signo	32	[0, 4.294.967.295]	uint
Int64	Enteros largos	64	[-9.223.372.036.854.775.808, 9.223.372.036.854.775.807]	long
UInt64	Enteros largos sin signo	64	[0-18.446.744.073.709.551.615]	**ulong**
Single	Reales con 7 dígitos de precisión	32	[1,5×10-45 - 3,4×1038]	float
Double	Reales de 15-16 dígitos de precisión	64	[5,0×10-324 - 1,7×10308]	**double**
Decimal	Reales de 28-29 cifras decimales	128	[1,0×10-28 - 7,9×1028]	decimal
Boolean	Valores lógicos	32	true, false	**bool**
Char	Caracteres Unicode	16	['\u0000', '\uFFFF']	**char**
String	Cadenas de caracteres (pág 120)	Variable	El permitido por la memoria	**string**
Object	Cualquier objeto	Variable	Cualquier objeto	**object**

Definición de Literales en C#

Entero sin signo: Los literales llevarán el carácter **U** ó u

```
uint variable = 4000000000U;
```

Entero largo y con signo: Los literales llevarán el carácter **L** y U

```
long variable = 4000000000000000L;
ulong variable = 1800000000000000000UL;
```

Hexadecimal: Pondremos el prefijo **0x** al valor en hexadecimal.

```
ushort n = 0x070F; // n = 0000 0111 0000 1111(2
ushort n = 0x_07_0F; // Desde C# 7
```

Binario desde C# 7: Pondremos el prefijo **0b** al valor en binario.

```
ushort n = 0b0000011100001111; // n = 070F(16
ushort n = 0b_0000_0111_0000_1111; // Desde C# 7
```

Real de simple precisión: los literales llevarán el carácter **F** ó f

```
float variable = 333.44f;
float variable = 4.5E-15F;
```

Real de doble precisión: los literales llevarán el carácter **D** ó d

```
double variable = 124325.456859D;
double variable = 4.5e127d;
```

Decimal: Los literales llevarán el carácter **M** ó m

```
decimal variable = 64538756498374657493847594.45M;
```

Literales Booleanos

Son las palabras reservadas `true` y `false`

```
bool estaFraccionado = true;
```

Literales de Carácter

- Podemos inicializarlo a cualquier carácter UNICODE o UTF-8
- Podemos consultar sus valores en UNICODE
- Los literales de carácter se definirán siempre entre comillas simples.
- Podremos representar cualquier literal a través de códigos de escape.

Carácter	Código UNICODE	Código escape
Comilla simple	\u0027	'
Comilla doble	\u0022	"
Carácter nulo	\u0000	\0
Alarma	\u0007	\a
Retroceso	\u0008	\b
Salto de página	\u000C	\f
Retorno de carro	\u000D	\r
Nueva línea	\u000A	\n
Tabulación horizontal	\u0009	\t
Tabulación vertical	\u000B	\v
Barra invertida	\u005C	\

```
char c1 = 'F';
char c2 = '\'';
char c3 = '\u20AC'; \\ €
```

Literales de Cadena

- Los literales se definirán entre comillas dobles "literal"

```
string texto = "Esto es una cadena de caracteres.";
```

- Dentro del literal de la cadena podremos incluir secuencias de escape de carácter, como las que hemos visto anteriormente.

```
string texto = "El precio es 20 \u20AC\nGRACIAS";
```

- Si colocamos el carácter `@` antes del literal de la cadena, se escapará cualquier secuencia de escape que se encuentre..

```
string texto = @"El precio es 20 \u20AC\nGRACIAS";
string texto = "El precio es 20 \\u20AC\\nGRACIAS";
```

- A partir de C# 6 se pueden interpolar cadenas con las reglas de formato que ya hemos visto en la salida por pantalla. Para ello colocaremos el carácter `$` delante de la misma.

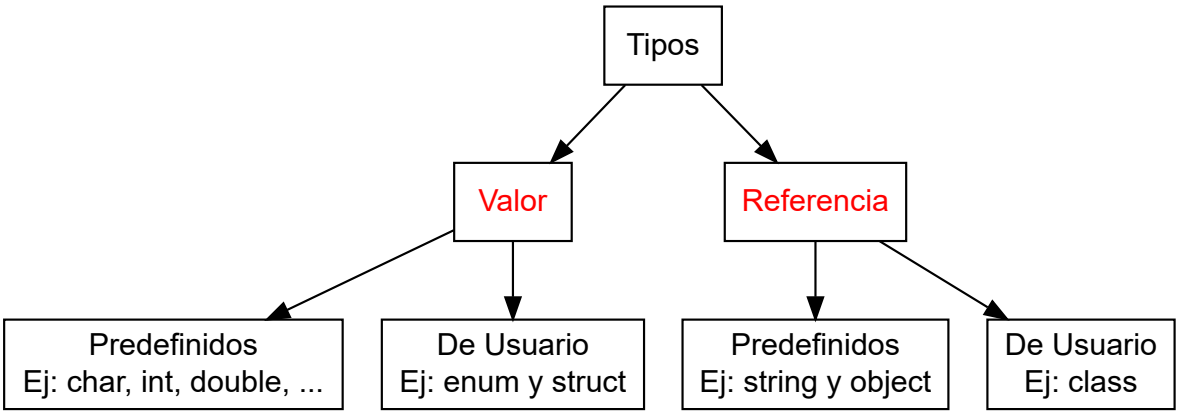
```
string t1 = "derecha";
int n = 99;
string t2 = $"\"Justificación a la {t1} con anchura 10: {n, 10}\"";
// Salida
// "Justificación a la derecha con anchura 10:         99"

int n = 88;
string t = $"Formato entero - {n:D5}";
// Salida
// "Formato entero - 00088"
```

> **Nota:** A partir de ahora usaremos este formato pues, además de hacer más legibles las cadenas, nos evitará errores derivados de una mala indexación en la sustitución.

Definición de Variables

Tipos de Variables



Variables De Tipo Valor

- Contienen sus datos 'directamente'.
- Cada variable tiene su propia copia del dato.
- Las operaciones sobre una no afectan a otra.
- Son Type-Safe, por lo que sabemos que contienen un valor válido.
- Un tipo valor seguido del carácter `?` se convertirá en **anulable** y por tanto se podrá asignar a null
- La palabra reservada `default` se evaluará al valor por defecto para el tipo o a null si es anulable.

Ejemplo Declaración de Tipos Valor

```
// Entero no anulable al que asigno 5 y con valores posibles de 0 a 255
byte a = 5;

// Entero no anulable al que asigno default para byte, que es 0 y con valores posibles de 0 a 255
byte b = default;

// Entero anulable al que asigno default para byte, que ahora es null por ser anulable
// y con valores posibles null o de 0 a 255

byte? c = default;
// Tras la inicialización tendré en memoria: a(5) b(0) c null

// Muestra: a(5) b(0) c()
Console.WriteLine($"a({a}) b({b}) c({c})");

c = a;
// Muestra: a(5) b(0) c(5)
Console.WriteLine($"a({a}) b({b}) c({c})");

c = 255;
// Muestra: a(5) b(0) c(255)
Console.WriteLine($"a({a}) b({b}) c({c})");

// Como 256 no es un valor válido para byte da la vuelta y empieza en 0
c = (byte)(c + 1);
// Muestra: a(5) b(0) c(0)
Console.WriteLine($"a({a}) b({b}) c({c})");

a = null; // Error de compilación
c = null; // Válido
```

Variables De Tipo Referencia

- Almacenan referencias a sus datos, normalmente objetos.
- Dos variables de tipo referencia pueden 'apuntar' a la misma zona de memoria.
- Las operaciones sobre una pueden afectar a la otra, si ambas 'apuntan' a la misma zona de memoria.
- La palabra reservada `default` se evaluará al valor por defecto para el tipo, que será **null** siempre.
- Por defecto **puedo asignarles un valor null** o un objeto del tipo. Pero desde C# 8, si añado en el archivo del proyecto lo siguiente:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    ...
    <Nullable>enable</Nullable>
  </PropertyGroup>
</Project>
```

No podré asignar un valor null a un tipo referencia por defecto.

```
// En proyectos por defecto o anteriores a C# 8
string texto = null; // Será válido.
string? texto = null // Dará un aviso de compilación porque no tiene sentido usar ?

// <Nullable>enable</Nullable> en el proyecto con C# 8
string texto = null; // Dará un aviso de compilación indicándome que estoy asignando un null.
string? texto = null // Será válido porque he marcado el tipo como anulable.
```

Ejemplo Declaración de Tipos Referencia

```
// La variable referencia a un objecto string en memoria, que contiene la cadena hola a -> (hola)
string a = "hola";

// Aviso de compilación sólo si está activado <Nullable>enable</Nullable> pues default vale null y
// con este flag activado no se puede asignar null directamente a un tipo referencia.
string b = default;

// Aviso de compilación sólo si no está activado <Nullable>enable</Nullable>
// ya que en ese caso carece de sentido al ser todos los tipos referencia anulables.
// Se comportará como si hubiéramos hecho string c = default;
string? c = default;

// Tras la inicilización tendré en memoria: a->(hola) b->() c->null

// Muestra: a->(hola) b->() c->()
Console.WriteLine($"a->({a}) b->({b}) c->({c})"); // Muestra: a->(hola) b->() c->()

c = a;
// Muestra: a->(hola) b->() c->a y por tanto c->a->(hola)
Console.WriteLine($"a->({a}) b->({b}) c->a y por tanto c->a->({a})");

c = "Pepe";
// Muestra: a->(hola) b->() c->(Pepe)
Console.WriteLine($"a->({a}) b->({b}) c->({c})");

a = null; // Aviso de compilación sólo si <Nullable>enable</Nullable>
c = null; // Válido en todos los casos.
```

Operadores y Expresiones en C#

Aunque han aparecido muchos operadores nuevos en C# 8 (2019), nosotros vamos a ver los más comunes a la hora de formar expresiones en la mayoría de lenguajes de programación. Muchos otros iremos hablando de ellos a lo largo del curso conforme veamos conceptos que nos permitan aplicarlos.

- **Tipos atendiendo al número de operadores**

Tipo	Ejemplo
Unarios	`(exp.)++`
Binarios	`(exp.1) + (exp.2)`
Ternarios	`(exp.1) ? (exp.2) : (exp.3)`

- **Tipos atendiendo a los valores con los que operan**

Tipo	Ejemplo
Operan con escalares	`(exp. eval. a escalar) * (exp. eval. a escalar)`
Operan con booleanos	`(exp. eval. a bool) && (exp. eval. a bool)`
Operan con bits	`(bits mem. de exp1) & (bits mem. de exp2)`

- **Tipos atendiendo al resultado**

Tipo	Ejemplo
Resultado escalar	`(exp.1) / (exp.2) → Se evalúa a escalar`
Resultado booleano	`(exp.1) >= (exp.2) → Se evalúa a bool`
	`(exp.1) || (exp.2) → Se evalúa a bool`

Operadores Unarios

- **Casting o Conversión Explícita**
  Sintaxis: `(tipo)operando`

```
float a = 2.3F;
double b = a;
long c;

// Casteamos el resultado de la expresión a / b
c = (long)(a / b);
```

  - Forzamos un cambio de un tipo a otro.
  - Normalmente se utiliza para poder realizar algún tipo de operación o forzar el resultado en algún tipo determinado.
  - Si utilizamos conversión **implícita** con el operador '=' el compilador nos avisará, pero con la **explícita** no.
    > **Nota:** Cuidado con las conversiones de mayor a menor rango.

- **Unarios de Pre/Post incremento y decremento en 1**

```
i = ++j; // Equivale ha hacer j=j+1; i=j;
i = j--; // Equivale ha hacer i=j; j=j-1;
```

  - **Post-incremento**: a++; → Se evalúa a y se hace a = a + 1
  - **Post-decremento**: a--; → Se evalúa a y se hace a = a - 1
    Los de Pre-incremento/decremento se evalúan después que los de Post al tener menos prioridad.
  - **Pre-incremento**: ++a; → Se hace a = a + 1 y se evalúa a
  - **Pre-decremento**: --a; → Se hace a = a - 1 y se evalúa a

- **typeof(*<nombre tipo de dato>*)**
  Devuelve un objeto del tipo System.Type que guarda información sobre el tipo de datos sobre lo que lo apliquemos.

```
Type t1 = typeof(double);
Type t2 = typeof(int);
Console.WriteLine(t1);
Console.WriteLine(t2);
```

- **nameof(*<identificador>*)** C# 6
  Devuelve un literal de cadena con el identificador de una variable, ...

```
int total = 4;
Console.WriteLine(total); // Muestra -> 4
Console.WriteLine(nameof(total)); // Muestra -> total
```

- **!(*<expresión booleana>*)**
  Negación lógica, devuelve el valor lógico inverso de una expresión booleana.

```
bool r = !(5 > 7) // r se evaluará a true
```

- **-(*<expresión escalar>*)**
  Aplica el signo negativo al resultado escalar de la derecha.

```
int r = -(5 - 7) // r se evaluará a 2
```

- **+(*<expresión escalar>*)** (No muy común)
  Aplica el signo positivo al resultado escalar de la derecha.

```
int r = +(5 - 7) // r se evaluará a -2
```

- **~(*<expresión>*)** (No muy común)
  Negación de bit. Invierte los bits en memoria de lo que estemos evaluando en la expresión.
  Se obtiene con la combinación (Alt + 126) o (Alt Gr + 4 seguido de espacio)

```
byte r = ~0b10011011; // r se evaluará a 0b01100100;
```

- **checked(*<expresión escalar>*)** (No muy común)
  Detecta condiciones de desbordamiento en un expresión, generando un error durante la ejecución.

```
short d1 = 20000, d2 = 20000;
short miShort = checked((short)(d1 + d2));
Console.WriteLine(miShort);
```

- **unchecked(<expresión escalar>)** (No muy común)
  Ignora condiciones de desbordamiento en un expresión, continuando con la ejecución.

```
short d1 = 20000, d2 = 20000;
short miShort = unchecked((short)(d1 + d2));
Console.WriteLine(miShort);
```

Operadores Binarios

Aritméticos

Operador	Propósito
*	Multiplicación
/	División
%	Módulo
+	Suma
-	Resta

- El `%` requiere que el segundo operador no sea nulo.
  Al contrario que en otros lenguajes pueden ser reales.
- El de división `/` podemos tener varios casos:
  1. Si `int / int = Parte entera resultado`.
  2. Si `float / float = float`
  3. Si `float / int o int / float = float`

Comparación

- Dispondremos de los triviales operadores de comparación `x < y, x > y, x <= y, x >= y` que se evalúan a un valor booleano.
- Con un poco menos de prioridad los de igualdad `x == y, x != y` que también se evalúan a un valor booleano.
  👁 Un **error típico de principiante**, es confundir la asignación `=` con la comparación `==`
- En este apartado también podremos incluir el operador **is**, que me ayudará a preguntar a un identificador si es de un determinado tipo y lo usaremos más adelante al ver POO.
  También se le conoce como operador de reflexión y tendrá la siguiente sintaxis: `<id> is <tipo>`
  Devolverá un booleano indicándome si el operador es del tipo o no.

```
int i = 0;
bool test = i is int;

// test = true;
```

Lógicos

- **AND**: (<expresión booleana>) **&&** (<expresión booleana>)
  Tabla de verdad...

expresión	evaluación
`true && true`	true
`true && false`	false
`false && true`	false
`false && false`	false

- **OR**: (<expresión booleana>) **||** (<expresión booleana>)
  Tiene menos prioridad que el AND
  Tabla de verdad...

expresión	evaluación
`true || true`	true
`true || false`	true
`false || true`	true
`false || false`	false

Operador de uso combinado Null

- Disponible desde **C#7**, también se le conoce como **null coalescing operator**.
  Tradicionalmente no se ha usado, pero recientemente se está incrementando su uso. Podemos encontrarlo con similar funcionamiento en Swift, ES6 o PHP7 y con diferente sintaxis en otros lenguajes.
  Se evalúa de derecha izquierda y tiene la siguiente sintaxis:
  `(tipo o expresión anulable) ?? (tipo o expresión anulable)`

```
int? a = null, b = 5;

Console.WriteLine(a ?? b ?? 3);
```

  En el código de ejemplo hará:
  1. `b ?? 3` pero como **b** no es null se evaluará a su valor **5**.
  2. `a ?? 5` al ser **a null** se evaluará a **5**.

  Si `int? b = null` entonces toda la expresión se evaluaría a **3**;
- Dispondremos de la versión del operador con asignación a partir de **C#8**
  Se evalúa de derecha izquierda y tiene la siguiente sintaxis:
  `(tipo o expresión anulable) ??= (tipo o expresión anulable)`

```
int? a = null, b = 5;

a = a ?? b;
//Equivale a...
a ??= b;
```

Condicional Ternario

- **Sintaxis**: `Condición ? Consecuencia : Alternativa`
- **Condición**: Es una expresión que se evalúa a un booleano.
  - **Consecuencia**: A lo que se evalúa toda la expresión si `Condición` se evalúa a `true`.
  - **Alternativa**: A lo que se evalúa toda la expresión si `Condición` se evalúa a `false`.

> **Importante**: Las expresiones `Consecuencia` y `Alternativa` se deben evaluar al mismo tipo de dato.

```
(a > b) ? a : b; // Si a mayor que b entonces a sino b.
```

- Trataremos de **evitar usarla** o abusar del mismo, si obtenemos **expresiones ofuscadas** o podemos simplificar usando otros operadores.

```
int? a = 3, b = 11, c = null;

int? d = a > 0 && b <= 10 ? ++a * b : c != null ? c : 5;

// Podríamos reescribirla como...
int? d = (a > 2 && b <= 10)
         ? (++a * b)
         : (c ?? 5);
```

Cuadro Resumen con Precedencia y Asociatividad

Orden	Nombre	Asociatividad	Operador
0	Principales	izq. a der.	`x.y, f(x), a[i], x?.y, x?[y], x++, x--, x!, new, typeof, checked, unchecked, default, nameof, delegate, stackalloc`
1	**Unarios**	izq. a der.	`+x, -x, !x, ~x, ++x, --x, ^x, (T)x, await, &x, *x, true and false`
2	**Intervalo** C#8	izq. a der.	`x..y`
3	`switch` **como expresión** C#8	izq. a der.	`x switch { v1 => expr, v2 => expr, _ => expr }`
4	Multiplicación, división, módulo división	izq. a der.	`x * y, x / y, x % y`
5	Suma y concatenación de cadenas, resta	izq. a der.	`x + y, x - y`
6	Desplazamiento de **bits**	izq. a der.	`x << y, x >> y`
7	Comparaciones, is, as	izq. a der.	`x < y, x > y, x <= y, x >= y, is, as`
8	Igualdad, desigualdad	izq. a der.	`x == y, x != y`
9	AND de **bits**	izq. a der.	`x & y`
10	XOR de **bits**	izq. a der.	`x ^ y`
11	OR de **bits**	izq. a der.	`x | y`
12	AND **lógico**	izq. a der.	`x && y`
13	OR **lógico**	izq. a der.	`x || y`
14	Operador de uso combinado de Null	der. a izq.	`x ?? y`
15	Condicional **ternario**	der. a izq.	`c ? t : f`
17	Asignación y Asignación compuesta.	der. a izq.	`x = y, x += y, x -= y, x *= y, x /= y, x %= y, x &= y, x | = y, x ^= y, x <<= y, x >>= y, x ??= y, =>`
18	Evaluación Múltiple	izq. a der.	`x, y, ..., z`

Ejemplos de Precedencia con Operadores Básicos

Ejemplo 1:

Escribe la expresión algorítmica en C# para la siguiente expresión aritmética usando el menor número de paréntesis:

$$r = \frac{a^2}{b-c} + \frac{d-e}{f - \frac{g \cdot h}{j}}$$

```
int r = (a * a / (b - c)) + (d - e) / (f - g * h / j);
```

Ejemplo 2:

Evalúa la expresión **`int a = -(4 * 4 / 2 - (4 * (8 % 2) + 12)) + 8 / 2 % 2;`**
paso a paso teniendo en cuenta la precedencia de los operadores aritméticos.

```
int a = -(4 * 4 / 2 - (4 * (8 % 2) + 12)) + 8 / 2 % 2;
int a = -(4 * 4 / 2 - (4 * 0 + 12)) + 8 / 2 % 2;
int a = -(4 * 4 / 2 - (0 + 12)) + 8 / 2 % 2;
int a = -(4 * 4 / 2 - 12) + 8 / 2 % 2;
int a = -(16 / 2 - 12) + 8 / 2 % 2;
int a = -(8 - 12) + 8 / 2 % 2;
int a = -(-4) + 8 / 2 % 2;
int a = 4 + 8 / 2 % 2;
int a = 4 + 4 % 2;
int a = 4 + 0;
int a = 4;
```

Ejemplo 3:

Evalúa la expresión **`bool a = 35 > 47 && 9 == 9 || 35 != 3 + 2 && 3 >= 3;`**
paso a paso teniendo en cuenta la precedencia de los operadores aritméticos.

```
bool a = 35 > 47 && 9 == 9 || 35 != 3 + 2 && 3 >= 3;
bool a = 35 > 47 && 9 == 9 || 35 != 5 && 3 >= 3;
bool a = false && 9 == 9 || 35 != 5 && 3 >= 3;
bool a = false && 9 == 9 || 35 != 5 && true;
bool a = false && true || 35 != 5 && true;
bool a = false && true || true && true;
bool a = false || true && true;
bool a = false || true;
bool a = true;
```

Ejemplo 4:

- Sean **x, y, z, u, v, t, w** variables que contienen respectivamente los valores **2, 3, 4, 5, 6 y 7**

  ```
  double x=2d, y=3d, z=4d, u=5d, v=6d, t=7d, w;
  ```

- ¿Qué almacenarán después de ejecutar cada una de las siguientes sentencias?
- Realiza una traza creando un tabla donde cada una de las filas sea la expresión que estoy evaluando en ese momento y las columnas el valor de las variable.

Nota: Puedes ver el resultado de evaluar estas expresiones mediante la instrucción:

```
Console.WriteLine($"x={x} - y={y} - z={z} - u={u} - v={v} - t={t} - w={w}");
```

	x	y	z	u	v	t	w
Valor Inicial	2	3	4	5	6	7	-
`x++;`	3	3	4	5	6	7	-
`y = ++z;`	3	5	5	5	6	7	-
`t = v--;`	3	5	5	5	5	6	-
`v = x + (y*=3) / 2;`	3	15	5	5	10,5	6	-
`w = x + y / 2;`	3	15	5	5	10,5	6	10,5