

TEMA 2. LENGUAJE DE PROGRAMACIÓN EN DISEÑO DE INTERFACES

¿Qué es C#?

El lenguaje C# (se pronuncia «ci sharp» en inglés) es un lenguaje de programación multiparadigma (POO y programación funcional), fuertemente tipado y multiplataforma. Para desmitificar esta definición, vamos a ver el significado de los elementos:

- Fuertemente tipado significa que todos los elementos se declaran inicialmente con un tipo determinado, que no se podrá modificar durante toda su vida. Por ejemplo, si declaramos un entero para almacenar en él un valor cualquiera, como la edad de una persona, luego no podremos almacenar en él su nombre en lugar de la cantidad de años porque es una cadena de caracteres, y no un entero. Este funcionamiento es muy distinto del de algunos lenguajes denominados débilmente tipados, como JavaScript, por ejemplo.
- El desarrollo orientado a objetos es una forma específica de escribir código informático siguiendo un paradigma preciso. Basado en clases.
- Programación funcional porque nos permite trabajar con funciones de primera clase, pasar funciones como parametros, devolver funciones como resultado de una función, o asignarle el resultado de una función a una variable...
- Multiplataforma porque el lenguaje permite realizar aplicaciones que no dependen de la plataforma. En efecto, C# se utiliza como un conjunto de herramientas, llamado framework, que le da al desarrollador la posibilidad de crear aplicaciones con tipologías muy diversas. En este sentido, se puede usar en cualquier tipo de entorno (PC, Mac, móvil, TV, Smart Watch, etc.).

La sintaxis

Sentencias de alto nivel: A partir de c#9, han sido creadas para permitirnos tener aplicaciones sencillas, escribiendo estas sentencias directamente en el código, sólo si tenemos un único punto de entrada en un programa.

1. Los identificadores

Los identificadores son los nombres que se asignan a las clases y a sus miembros. Un identificador debe estar compuesto por una única palabra. Ésta debe empezar por una letra o un carácter (`_` o `@`). Los identificadores pueden tener letras mayúsculas o minúsculas, pero

como el lenguaje C# es sensible a mayúsculas/minúsculas, éstas se deben respetar para que la referencia al identificador sea correcta: mildentificador es diferente a Mildentificador.

El estándar en C# para definir las variables es utilizar camel case (estandar de camello)

miEntornoDeTrabajo

2. Las palabras reservadas

Las palabras reservadas son los nombres que reserva el lenguaje C#. Se interpretan por el compilador y, por tanto, no se pueden usar como identificadores. Estas palabras reservadas se diferencian en el editor de texto de Visual Studio utilizando el color azul (con los argumentos de apariencia predeterminados).

Si necesita utilizar una palabra reservada como un identificador para un miembro, es necesario utilizar un prefijo con el nombre del identificador: el carácter @. La siguiente sintaxis es errónea y el compilador no la podrá ejecutar:

```
private bool lock;
```

Si utilizamos el prefijo @ con el miembro lock, el compilador considera que es un identificador y no una palabra reservada:

```
private bool @lock;
```

El carácter @ también puede servir como prefijo de los identificadores que no tienen ningún conflicto con las palabras reservadas, de esta manera @mildentificador se interpretará de la misma manera que mildentificador.

A continuación se muestra una lista de las palabras reservadas del lenguaje C#. Se explicarán, en parte, a lo largo del libro:

abstract	add	as	ascending	async
await	base	bool	break	by
byte	case	catch	char	checked
class	const	continue	decimal	default
delegate	descending	do	double	dynamic
else	equals	enum	event	explicit

extern	false	finally	fixed	float
for	foreach	from	get	global
goto	group	if	implicit	in
int	interfaz	internal	into	is
join	let	lock	long	nameof
namespace	new	null	object	on
operator	orderby	out	override	params
partial	private	protected	public	readonly
ref	remove	return	sbyte	sealed
select	set	short	sizeof	stackalloc
static	string	struct	switch	this
throw	true	try	typeof	uint
ulong	unchecked	unsafe	ushort	using
value	var	virtual	volatile	void
where	while	yield		

3. Las reglas de puntuación

El objetivo de las reglas de puntuación es separar las instrucciones del programa de manera lógica, comprensible por el ser humano e interpretable por el compilador.

Cualquier instrucción debe terminar con un punto y coma ;. Si se olvida al final de la instrucción, el compilador devuelve un error de sintaxis. Sin embargo, la ventaja es poder escribir una instrucción en varias líneas:

```
int i = 5
```

```
    + 2;
```

El punto . después de un identificador permite acceder a los miembros de un objeto. A través de IntelliSense, Visual Studio muestra la lista de miembros disponibles, tan pronto como se añade el punto a un objeto:

```
miObjeto.miPropiedad
```

Las llaves { y } se usan para agrupar varias instrucciones dentro de un bloque de control o de un método. Indican dónde comienzan las instrucciones y dónde terminan:

```
class Program
```

```
{  
  
}
```

Los paréntesis (y) se usan para declarar o para llamar a métodos. Pueden contener argumentos después de la declaración del método. Los argumentos de un método se separan con una coma ,:

```
miObjeto.miMetodo(Parametro1, Parametro2);
```

Los paréntesis también se utilizan para agrupar las instrucciones de la misma manera que para una operación matemática.

Los corchetes [y] permiten acceder a los elementos de un array o, si la clase contiene una propiedad para indexar, a los elementos de una clase. Por ejemplo, si la clase miObjeto es un array de valores de tipo string, para acceder a su primer elemento, la sintaxis sería la siguiente:

```
string s = miObjeto[0];
```

Los elementos de los arrays se indexan comenzando por 0.

4. Los operadores

a. Los operadores de cálculo

Los operadores de cálculo permiten, como en matemáticas, realizar operaciones.

La adición se realiza con el operador +:

```
i = 5 + 2;    // i = 7
```

La sustracción se realiza con el operador -:

```
i = 5 - 2;    // i = 3
```

La multiplicación se realiza con el operador *:

```
i = 5 * 2;    // i = 10
```

La división se realiza con el operador /:

```
i = 6 / 2;    // i = 3
```

El modulo se realiza con el operador %:

```
i = 5 % 2;    // i = 1
```

b. Los operadores de asignación

Los operadores de asignación permiten asignar un valor a una variable. El operador más utilizado es el carácter =:

```
i = x;
```

También es posible realizar una asignación y un cálculo al mismo tiempo, combinando dos operadores:

```
i += 1;
```

usando el operador +=, hay una asignación a la variable de su propio valor, sumando el valor de la derecha del operador. Esta instrucción es equivalente a la siguiente:

```
i = i + 1;
```

La combinación de operadores de cálculo y de asignación es posible para todos los operadores:

```
int i = 5;
```

```
i += 2;    // i = 7
```

```
i -= 2;    // i = 5
```

```
i *= 2;    // i = 10
```

```
i /= 2;    // i = 5
```

```
i %= 2;    // i = 1
```

c. Los operadores de comparación

Los operadores de comparación se utilizan fundamentalmente para tomar decisiones dentro de instrucciones de control.

El operador == determina si dos variables son iguales:

`x == y` // devuelve true si x igual a y

El operador `!=` determina si dos variables son diferentes:

`x != y` // devuelve true si x es diferente de y

El operador `>` determina si la variable de la izquierda es estrictamente superior a la variable de la derecha:

`x > y` // devuelve true si x es superior a y

El operador `>=` determina si la variable de la izquierda es superior o igual a la variable de la derecha:

`x >= y` // devuelve true si x es superior o igual a y

El operador `<` determina si la variable de la izquierda es estrictamente inferior a la variable de la derecha:

`x < y` // devuelve true si x es inferior a y

El operador `<=` determina si la variable de la izquierda es inferior o igual a la variable de la derecha:

`x <= y` // devuelve true si x es inferior o igual a y

El operador (y palabra reservada) `is` permite determinar el tipo de un objeto:

`x is int` // devuelve true si x es del tipo int

El filtrado por motivo permite verificar si un valor corresponde a un motivo. Se trata de utilizar el operador `is` para definir el motivo que puede reemplazarse en instrucciones condicionales:

`o is DateTime d` // devuelve true si o es de tipo DateTime

La variable `o` es automáticamente convertida en el tipo testeado y almacenada en la nueva variable `d`, que se puede utilizar de manera clásica.

También es posible combinar los operadores de comparación con los operadores lógicos. El operador `&&` permite especificar un AND lógico, mientras que el operador `||` especifica un OR lógico. Las diferentes expresiones se pueden combinar con ayuda de los paréntesis para modificar el orden de interpretación: `(x >= 0 || x > 10) && (x <= 1 || x < 25)`

5. La declaración de variables

La declaración de variables se realiza especificando su tipo y posteriormente indicando su identificador. La siguiente instrucción declara una variable llamada `s` de tipo `string`:

`string s;`

La instrucción de declaración termina como cualquier instrucción, es decir, con un punto y coma.

Una variable se puede declarar e inicializar con la misma instrucción:

```
string s = "El valor de mi variable";
```

También es posible declarar e inicializar varias variables en una única instrucción, con la condición de que sean del mismo tipo. Las variables se separan con una coma:

```
bool b1 = true, b2 = false;
```

Una variable también se puede marcar con la palabra reservada `const`, que especifica que el valor de la variable no se puede modificar durante la ejecución. Es una variable en modo de solo lectura:

```
const int i = 0;
```

El ámbito de una variable declarada en el cuerpo de un método se limita a ese método, es decir, se elimina cuando finaliza la ejecución del método. Queda fuera de ámbito.

Las variables declaradas dentro de un bloque condicional o iterativo solo son accesibles dentro de este bloque. Para los bloques iterativos, la variable se elimina al final del bucle y se inicializa de nuevo en la siguiente iteración del bucle.

6. Las instrucciones de control

a. Las instrucciones condicionales

Las instrucciones condicionales permiten ejecutar una porción de código en función de comprobaciones realizadas sobre las variables de la aplicación. Existen dos tipos de estructuras condicionales: los bloques `if` y los bloques `switch`.

`if`, `else` y `else if`

Sintaxis general:

```
if (expresión)
```

```
{
```

```
    instrucciones
```

```
}
```

```
[else if (expresión)
```

```
{
```

```
    instrucciones
```

```
}]
```

```
[else
```

```
{
```

```
    instrucciones
```

```
}]
```

La instrucción if evalúa una expresión booleana y ejecuta el código si esta expresión es verdad (true). Su sintaxis es la siguiente:

```
if (x > 10)
```

```
{
```

```
    // Instrucciones que se ejecutan si x es superior a 10
```

```
}
```

La expresión a evaluar se inserta entre paréntesis después de la palabra reservada if y debe tener un resultado de tipo booleano.

La instrucción else permite integrar el código que se ejecutará si la expresión que se evalúa en la instrucción if es falsa (false):

```
if (x > 10)
```

```
{
```

```
    // Instrucciones que se ejecutan si x es superior a 10
```

```
}
```

```
else
```

```
{
```

```
    // Instrucciones que se ejecutan si x es inferior o igual a 10
```

```
}
```

La instrucción else if permite evaluar una nueva expresión cuando el resultado de la instrucción if es falsa. Puede haber varias instrucciones else if pero un bloque de decisión if siempre debe empezar por una instrucción if, seguida de una o varias instrucciones else if y terminar de manera no obligatoria por una instrucción else:

```
if (x > 10)
```

```
{
```



```

    // Instrucciones que se ejecutan si x es superior a 10
}

else if (x = 10)

{
    // Instrucciones que se ejecutan si x es igual a 10
}

else

{
    // Instrucciones que se ejecutan si x es inferior a 10
}

```

Las instrucciones dentro de la estructura de decisión, se encierran entre llaves. Esto indica el inicio y el fin del bloque. En el caso en el que el bloque solo tenga una línea, se permite omitir las llaves (también válido para las demás instrucciones de control):

```

if (x > 10)

    x -= 10;

else if (x = 10)

    x -= 1;

else

    x += 10;

```

El operador de condición null ?. permite comprobar el valor null cuando se accede a los miembros de una clase. Esto limita el número de instrucciones condicionales y el código resulta más legible. Por ejemplo, para acceder a una propiedad encapsulada en un objeto y asegurarse de que ningún padre es null y, por tanto, que no se producirá ninguna excepción durante su acceso, deberíamos escribir una instrucción de este tipo:

```

string MajorVersion;

if(soft!=null && soft.Version!=null)

{
    MajorVersion = soft.Version.Major;
}

```

Con el nuevo operador, la sintaxis es más sencilla:

```
string MajorVersion = soft?.Version?.Major;
```

Si la variable `soft` es `null`, `soft?.Version` devolverá `null`. Del mismo modo, si la variable `soft?.Version` es `null`, `soft?.Version?.Major` devolverá `null`. En aquellos casos en que las variables no sean `null`, se devolverá el valor de `Major` y se asignará a la variable `MajorVersion`.

Expresión ternaria

Sintaxis general

```
expresión ? [variable si cierto] : [variable si falso]
```

La expresión booleana se evalúa. Si es verdadera, se devuelve el valor de la variable (o el resultado de una expresión o de una función) de la parte izquierda. En caso contrario, se devuelve la variable de la parte derecha. Su sintaxis es la siguiente:

```
int i = 20 ;
```

```
bool b = i >= 20 ? true : false ;
```

```
// b = true
```

La conversión de tipo entre la variable de retorno y el tipo al que se asigna el valor, se realiza de manera automática cuando es posible (en el marco de una herencia de clase, por ejemplo).

switch

Sintaxis general:

```
switch (expresión)
```

```
{
```

```
    case expresión o constante:
```

```
        instrucciones
```

```
        [instrucciones de salto]
```

```
    [default:
```

```
        instrucciones]
```

```
}
```

Una instrucción `switch` se puede considerar como una sucesión de instrucciones `if`, `else if` y `else`.

La instrucción switch evalúa el valor de una variable que se pasa como argumento y ejecuta las instrucciones en función de los posibles valores:

```
switch (i)
{
    case 0:
        i++;
        break;
    case 1:
        i--;
        break;
    default:
        i = 0;
        break;
}
```

La palabra reservada default se debe situar en último lugar en la cadena de evaluación y el código se ejecutará si no se cumple ninguna de las condiciones anteriores. Añadir la instrucción default es opcional.

Cada bloque de instrucciones dentro de una evaluación debe contener la palabra reservada break para salir del bloque switch. Si no está, eventualmente se evaluarán las demás condiciones y potencialmente se ejecutarán. Las demás instrucciones de salto también son válidas dentro de una estructura switch.

Es posible combinar varias evaluaciones si se deben ejecutar las mismas instrucciones:

```
switch (i)
{
    case 0:
    case 1:
    case 2:
        i++;
        break;
}
```

```

    case 3:

        i--;

        break;

}

```

En el contexto de una instrucción switch, se puede usar el filtrado por motivo. Se puede añadir condiciones con la palabra clave when:

```

switch (o)

{

    case int i when i < 0:

        break;

    case int i when i > 0:

        break;

}

```

El programa entra en el bucle de decisión si i es de tipo int y corresponde a la condición. La variable i tiene un ámbito en el bloque correspondiente únicamente.

La decisión también se puede realizar sobre el tipo del objeto únicamente.

La instrucción Switch también puede contener un modelo compuesto de expresiones más o menos complejas o simplemente un tipo (Control en el siguiente ejemplo):

```

Control c = new Label() { Size = new Size(10, 10) };

int resultado = c switch

{

    Control c1 when c1.Size.Height == 0 && c1.Size.Height == 0 => 0,

    Control c2 => c2.Size.Height + c2.Size.Height,

    => -1

};

// resultado = 20

```

La palabra clave default se sustituye en este caso por el signo _ (underscore).

b. Las instrucciones iterativas

Las instrucciones iterativas permiten ejecutar bucles sobre una serie de instrucciones en función de la evaluación de una expresión. Las instrucciones iterativas son los bucles for, while, do while y foreach.

for

Sintaxis general:

for (inicialización; expresión; paso)

```
{  
    instrucciones  
}
```

Un bucle for contiene una serie de instrucciones que se ejecuta varias veces según los argumentos de inicialización, la expresión y de paso de incremento. La inicialización consiste en la declaración y la asignación de una variable de tipo numérico. En cada vuelta del bucle, la variable incrementa el valor del paso y la expresión se evalúa para determinar si las instrucciones se deben ejecutar o no.

```
int total = 0;
```

```
for (int i = 0; i < 4; i++)
```

```
{  
    total += i;  
}
```

```
// total = 6
```

El código anterior declara una variable total y le asigna el valor 0. Un bucle for declara una variable i, asignándole el valor 0. La expresión se evalúa posteriormente y, si es verdadera, se ejecutan las instrucciones. Cuando se han ejecutado todas las instrucciones, la variable i se incrementa. En este caso se suma 1 a la variable i. La expresión se evalúa de nuevo y el bucle continúa hasta que la expresión sea falsa.

La variable declarada en la instrucción for se puede modificar en el cuerpo del bucle:

```
for (int i = 0; i < 4; i++)
```

```
{  
    i++;  
}
```

```
// El cuerpo del bucle for se ejecuta dos veces
```

También es posible definir un paso negativo:

```
for (int i = 10; i > 0; i--)
```

```
{
```

```
while
```

Sintaxis general:

```
while (expresión)
```

```
{
```

```
    instrucciones
```

```
}
```

Un bucle while permite ejecutar una serie de instrucciones mientras que la expresión evaluada sea cierta:

```
int i = 0;
```

```
while (i < 4)
```

```
{
```

```
    i++;
```

```
}
```

```
// i = 3
```

do while

Sintaxis general:

```
do
```

```
{
```

```
    instrucciones
```

```
}
```

```
while (expresión);
```

El bucle do while es parecido al bucle while, en el sentido en que las instrucciones del bucle se ejecutan si la evaluación de la expresión es verdadera. La diferencia reside en que el bucle do while hace la comprobación de la expresión después de la ejecución de las instrucciones. El bucle do while garantiza que las instrucciones se ejecutan al menos una vez:

```
int i = 0;

do

{

    i--;

}

while (i >= 0);

// i = -1
```

foreach

Sintaxis general:

foreach (tipo nombre in object)

```
{

    instrucciones

}
```

Los bucles foreach se usan para recorrer los arrays, las colecciones y todos los tipos enumerados. La instrucción declara un tipo y recorre el objeto que se pasa como argumento para el tipo dado. El bucle se ejecuta mientras haya elementos en el objeto:

```
string s1 = "foreach";
```

```
string s2 = "";
```

```
foreach (char c in s1)
```

```
{

    s2 += c;

}
```

```
// s2 = "foreach"
```

La variable s1 de tipo string se compone de una colección de tipo char. El bucle foreach las toma una por una y las asigna a la variable c de tipo char. A continuación, la variable c se puede usar en el cuerpo del bucle.

c. Las instrucciones de salto

Las instrucciones de salto permiten modificar el flujo de ejecución del programa. Las instrucciones de salto son break, continue, goto, return y throw.

break

La instrucción break permite salir de un bucle o de una instrucción switch sin esperar al fin de la ejecución de todas las instrucciones:

```
for (int i = 0; i < 10; i++)
```

```
{  
    if (i == 5)  
    {  
        break;  
    }  
}
```

// i = 5, el cuerpo del bucle se ha ejecutado 5 veces

continue

La instrucción continue permite parar la ejecución del cuerpo de un bucle y pasar a la siguiente evaluación:

```
for (int i = 0; i < 10; i++)
```

```
{  
    if (i > 5)  
    {  
        continue;  
    }  
}
```

// i = 5, el cuerpo del bucle se ha ejecutado 10 veces

goto

La instrucción goto permite transferir la ejecución a un bloque de instrucciones. Esta instrucción se hereda de los antiguos lenguajes procedimentales, como el basic.

Una instrucción goto está seguida del nombre de una etiqueta y la ejecución del código se transfiere a esta etiqueta. La etiqueta se define en el código utilizando un nombre, seguido del carácter dos puntos ::

```
int i = 0;
```

añadir:

```
i++;
```

```
if (i < 5)
```

```
{
```

```
    goto añadir;
```

```
}
```

```
// i = 5
```

La instrucción goto también se utiliza dentro de las instrucciones switch para pasar de un caso a otro. goto puede hacer referencia a otro caso o al caso por defecto:

```
int i = 0;
```

```
switch (i)
```

```
{
```

```
    case 0:
```

```
        i++;
```

```
        goto 1;
```

```
    case 1:
```

```
        i--;
```

```
        goto default;
```

```
    default:
```

```
        i = 0;
```

```
        break;
```

```
}
```

```
return
```

La instrucción `return` se usa dentro de una función y permite definir su valor de retorno:

```
int Addition(int i, int j)
```

```
{  
    return i + j;  
}
```

`throw`

La instrucción `throw` permite generar excepciones. La ejecución se interrumpe inmediatamente y se pasa la naturaleza del error a la instrucción que llama, dentro de la pila de llamadas, hasta ser interceptada y tratada:

```
throw new Exception();
```

La gestión de las excepciones se estudiará con más en detalle en el capítulo dedicado a la gestión de errores.

7. Los comentarios

El código puede contener comentarios para facilitar su comprensión por parte de otro desarrollador o para su mantenimiento.

Los comentarios comienzan con una doble barra `//`. Pueden estar en una única línea o al final de una instrucción, como en los ejemplos anteriores. Este tipo de comentario es para una sola línea. Un salto de línea implica una nueva instrucción.

Para añadir comentarios en varias líneas, éstos deben comenzar con los caracteres `/*` y terminar con los caracteres `*/`:

```
/* Mi comentario
```

```
en varias
```

```
líneas */
```

El editor de texto de Visual Studio colorea los comentarios en verde (parámetro predeterminado).

Los tipos básicos

Los tipos de datos permiten almacenar valores en la aplicación. Los lenguajes .NET son fuertemente tipados, por lo que no siempre es posible convertir un tipo de datos en un otro. Las conversiones, implícitas o explícitas, permiten convertir los tipos de datos. Esto es posible

porque todos los tipos del Framework .NET provienen del tipo Object, que es el tipo básico del resto de tipos.

1. Los tipos numéricos

Los tipos numéricos se dividen en dos grupos: los enteros y los decimales. Cada uno tiene un conjunto de tipos para representar los datos de la mejor manera, en función de las necesidades.

a. Los enteros

La siguiente tabla resume los tipos enteros disponibles en el Framework .NET:

Tipo .NET	Nombre C#	Descripción	Rango de valores
System.Byte	byte	Entero sin signo de 8 bits	De 0 a 255
System.Int16	short	Entero con signo de 16 bits	De -32 768 a 32 767
System.Int32	int	Entero con signo de 32 bits	De -2 147 483 648 a 2 147 483 647
System.Int64	long	Entero con signo de 64 bits	De -9 223 372 036 854 775 808 a 9 223 372 036 854 775 807
System.SByte	sbyte	Entero con signo de 8 bits	De -128 a 127
System.UInt16	ushort	Entero sin signo de 16 bits	De 0 a 65 535
System.UInt32	uint	Entero sin signo de 32 bits	De 0 a 4 294 967 295

System.UInt64	ulong	Entero sin signo de 64 bits	De 0 a 18 446 744 073 709 551 615
---------------	-------	-----------------------------	-----------------------------------

Un valor se puede asignar a un entero con una notación decimal:

```
int i = 2;    // Notación decimal
```

Se puede usar también la notación hexadecimal y se debe preceder del prefijo 0x:

```
int i = 0x4B;    // Notación hexadecimal equivalente: i = 75;
```

También está a disposición la notación binaria y debe estar precedida del prefijo 0b:

```
int i = 0b1101;    // Notación binaria equivalente: i = 13;
```

Para facilitar la lectura del código, el carácter _ se puede usar como separador:

```
int i = 100_000_000;    // Notación binaria equivalente:
i = 100000000;
```

b. Los decimales

La siguiente tabla resume los tipos decimales disponibles en el Framework .NET:

Tipo .NET	Nombre C#	Descripción	Precisión
System.Single	float	Número con coma flotante de 32 bits	7 cifras significativas
System.Double	double	Número con coma flotante de 64 bits	15 cifras significativas
System.Decimal	decimal	Número con coma flotante de 128 bits	28 cifras significativas

2. Los booleanos

Un booleano es un tipo que permite representar un valor que es true o false. El tipo .NET correspondiente es System.Boolean y su nombre C# es bool.

Es posible asignar a un booleano el resultado de una comparación:

```
byte x = 1;
```

```
bool b = x < 2;    // b tiene el valor true
```

3. Las cadenas de caracteres

El tipo `System.String` (`string`) es un tipo por **referencia** (su variable almacena la dirección de memoria de donde se encuentra la información y no su valor) que representa una serie de tipos `System.Char` (`char`).

Una variable de tipo `char` se asigna con un carácter entre comillas simples:

```
char c = 'a';
```

El tipo `char` representa una instancia de un carácter Unicode de 16 bits. Por lo tanto, es posible asignar un valor a un tipo `char` usando el valor numérico del carácter Unicode, que consiste en proporcionar un carácter en formato hexadecimal de 4 cifras:

```
char c = '\u0061';    // Equivale a: c = 'a';
```

Se asigna una variable de tipo `string` con una cadena de caracteres entre comillas dobles:

```
string s = "Mi cadena";
```

Aunque el tipo `string` es un tipo por referencia, tiene métodos que permiten manipular cadenas. La siguiente tabla presenta los métodos más utilizados:

Método	Descripción
Format	Reemplaza las expresiones de tipo {0}, {1}, {2}, etc. presentes en la cadena por los valores que se pasan como parámetro en la llamada a la función.
Replace	Reemplaza todas las ocurrencias de un carácter en la cadena por otro.
Split	Separa la cadena en varias, en función de un carácter delimitador.
Substring	Devuelve una parte de la cadena.
ToCharArray	Devuelve un array de tipo <code>char</code> a partir de la cadena.
ToLower	Convierte todos los caracteres de la cadena a minúsculas.

ToUpper Convierte todos los caracteres de la cadena a mayúsculas.

Trim Elimina los espacios al inicio y fin de la cadena.

La versión 6 de C# introduce la interpolación de cadenas. En lo sucesivo, para componer una cadena de caracteres, además de la concatenación clásica mediante el operador + y con el método Format de la clase String, podrá utilizar la interpolación escribiendo los nombres de las variables directamente en la cadena (precedida por el carácter \$) delimitándolos por la secuencia { }.

El valor se reemplazará automáticamente en tiempo de ejecución por el valor de la variable:

```
string name = "Lola";  
Console.WriteLine("Bienvenida {name} !"); // Bienvenida Lola !
```

Escapar caracteres de un string

```
string texto = "Mi amigo me dijo una vez: \\'Me caes bien\\'";  
string caracter = "En c# podemos usar \\ para escapar strings";
```

4. Los tipos null

Los tipos por referencia pueden representar valores no existentes (null), a diferencia de lo que sucede con los tipos por valor (int, decimal, float, double, bool y DateTime):

```
string s = null;            // Operación autorizada.  
int i = null;              // Error de compilación.
```

Para representar un valor null en un tipo por valor, es preciso utilizar una construcción específica llamada tipo null. Un tipo que admite valores null se marca con el carácter ?:

```
int? i = null;            // Operación autorizada.
```

De esta manera es posible comprobar si la variable contiene un valor de manera clásica o con su propiedad HasValue:

```
if (i != null)  
{ }
```

Equivale a:

```
if (i.HasValue)  
{ }
```

La conversión de un tipo clásico hacia un tipo null es implícita. El valor se asigna directamente a la propiedad Value del tipo null:

```
int? i = 1;
```

La conversión de un tipo que admite valores nulos en un tipo clásico debe ser explícita. Equivale a asignar la propiedad Value, de tipo null, al tipo clásico. Si la propiedad HasValue es false, y por tanto la variable es null, se genera una excepción:

```
int j = (int)i;    // Equivale a: int j = i.Value;
```

Desde la versión C# 8, es posible hacer que los tipos de referencia no puedan ser nulos por defecto, al igual que un tipo de valor. Para habilitar esta función, debe agregar la siguiente línea en el archivo de código:

```
#nullable enable
```

El objetivo es evitar los errores de tipo NullReferenceException. En el siguiente caso, con la opción no activada, se generará un error durante la ejecución:

```
Stream s = null;  
s.Flush();
```

Con la activación de la opción, aparecerá una advertencia que indicará: Conversión de literal que tiene un valor nulo o de un posible valor nulo en tipo no nulo.

Ya necesitará agregar el carácter ? para descartar esta advertencia:

```
#nullable enable  
Stream? s = null;  
s.Flush();
```

Ya solo tendrá la advertencia para el uso del objeto que podría ser null: Posible eliminación de referencia de una referencia nula.

Luego tendrá que probar si el objeto se instancia antes de poder usarlo:

```
#nullable enable  
Stream? s = null;  
if (s != null)  
    s.Flush();
```

Al agregar la siguiente línea al archivo csproj, todas las advertencias se tratarán como errores (sin excepciones):

```
<TreatWarningsAsErrors>true</TreatWarningsAsErrors>
```

Esto introduciría un gran cambio en su proyecto, ya que todos estos errores de compilación deberían ser manejados.

5. Los tipos DateTime

Tipo de datos que nos permite representar fechas y horas.

```
DateTime ahora = DateTime.Now;

DateTime primerdiadel año = new DateTime(2022,1,1);

DateTime mañana = DateTime.Now.AddDays(1);

Console.WriteLine(mañana.Day);

Console.WriteLine(mañana.DayOfWeek);
```

6. La conversión de tipos

La conversión de tipos se puede realizar de dos maneras diferentes: la conversión implícita, que significa que se hace de manera automática cuando no hay peligro de pérdida de datos, y la conversión explícita, que significa que la conversión y los tipos se deben especificar.

a. La conversión implícita

Si un tipo se puede convertir implícitamente en otro, es posible utilizar el primer tipo, sustituyendo el segundo sin sintaxis específica:

```
int i = 1;
long l = i;
```

Si un método se debe invocar con un argumento de tipo long, será posible invocarlo, pasándole un argumento de tipo int sin generar errores.

La siguiente tabla presenta las conversiones implícitas que C# tiene en cuenta:

Del tipo	Al tipo
byte	short, ushort, int, uint, long, ulong, float, double, decimal
short	int, long, float, double, decimal
int	long, float, double, decimal
long	float, double, decimal
float	double

sbyte	short, int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
uint	long, ulong, float, double, decimal
ulong	float, double, decimal
char	int, uint, long, ulong, float, double, decimal

b. La conversión explícita. Casteos.

Cuando una conversión afecta a tipos que no se pueden convertir implícitamente, hay que convertirlos explícitamente. Esta conversión se hace utilizando una sintaxis especial:

```
long l = 1;
int i = (int)l;
```

Se hace la conversión, dando por hecho que los dos tipos int y long pueden contener el valor 1. Si el tipo de destino de la conversión no puede contener el valor origen, el valor que se usa en la conversión no reflejará la realidad:

```
short s = 300;
byte b = (byte)s;           // b = 44
```

El código se compila y ejecuta sin generar errores, por lo que es necesario estar muy atento a las conversiones explícitas, que pueden ser una fuente de errores.

c. ToString

Cuando queremos convertir cualquier tipo de variable a un tipo String.

Ejemplo:

```
var cantidad1 = 5;
If (cantidad1.ToString()=="5")
```

```
var esCerto = true;
esCerto.ToString();
```

```
var hoy= DateTime.Today.ToString();
var hoy= DateTime.Today.ToString("yyyy-MM-dd");
```

d. Parse

Cuando queremos convertir de String a cualquier otro tipo de datos.

Ejemplo:

```
var cantidadString = "5";  
int cantidad = int.Parse(cantidadString)
```

```
var cantidadDecimalString = "4.9";  
var cantidadDecimal = decimal.Parse(cantidadDecimalString);
```

```
bool.Parse(variableBoolString);  
DateTime.Parse(variableFechaString);
```

Las constantes y las enumeraciones

Las constantes son variables que no se pueden modificar durante la ejecución. Permiten, por una parte, asociar nombres amigables a los valores que se usan frecuentemente en el código y, por otra parte, centralizar un valor de manera que solo se pueda modificar una única vez para toda la aplicación. Las enumeraciones son un conjunto de constantes que facilitan la legibilidad y el mantenimiento del código.

1. Las constantes

La palabra reservada `const` permite definir una constante dentro de una clase:

```
class MiClase  
{  
    const int HorasDelDia = 24;  
}
```

Las constantes pueden ser de cualquier tipo, por valor o string, que representan valores fijos que no se pueden modificar durante la ejecución del programa.

Una constante se puede usar en el código, haciendo referencia a su nombre:

```
int i = HorasDelDia;           // i = 24
```

Según su nivel de acceso, una constante también se puede usar fuera de la clase en la que se define y ser llamada como miembro estático:

```
int i = MiClase.HorasDelDia;    // i = 24
```

2. Las enumeraciones

Las enumeraciones permiten agrupar constantes semánticamente relacionadas:

```
enum DiaSemana
{
    Lunes,
    Martes,
    Miércoles,
    Jueves,
    Viernes,
    Sábado,
    Domingo
}
```

De manera predeterminada, los miembros de una enumeración se enumeran de forma secuencial, empezando por 0. De esta manera, las dos instrucciones siguientes son idénticas:

```
DiaSemana Today = DiaSemana.Lunes;
DiaSemana Today = 0;
```

El valor de los miembros se puede sobrecargar definiendo el valor durante la declaración, de la siguiente manera:

```
enum DiaSemana
{
    Lunes = 1,
    Martes = 2,
    Miércoles = 3,
    Jueves = 4,
    Viernes = 5,
    Sábado = 6,
    Domingo = 7
}
```

El tipo de datos por defecto de las enumeraciones es int. Para definir una enumeración con un tipo de datos diferente, hay que especificarlo durante la declaración, con la siguiente sintaxis:

```
enum DiaSemana: byte
{ ... }
```

Los tipos permitidos son: byte, sbyte, short, ushort, int, uint, long o ulong.

Las enumeraciones necesitan conversiones explícitas para ser asignadas a los tipos básicos:

```
int i = (int)DiaSemana.Lunes;
```

Una particularidad de las enumeraciones es poder combinar varios miembros:

```
DiaSemana FinDeSemana = DiaSemana.Sabado | DiaSemana.Domingo;
```

La combinación de miembros, implica ciertos requisitos previos para una enumeración:

- Por convención, se debe marcar con el atributo Flags. Sigue siendo posible combinar miembros para una enumeración no marcada, pero la llamada del método ToString devolverá un número en lugar de una serie de números. De esta manera, el ejemplo anterior devolverá el valor 7 si el atributo Flags no está presente y Sábado, Domingo en caso contrario.
- Para prevenir cualquier ambigüedad entre los miembros de una enumeración combinable, hay que asignar explícitamente los valores a los miembros. Como regla general, son una sucesión de parejas:

```
[Flags]
enum DiaSemana
{
    Lunes = 1,
    Martes = 2,
    Miércoles = 4,
    Jueves = 8,
    Viernes = 16,
    Sábado = 32,
    Domingo = 64
}
```

También es posible especificar combinaciones de miembros durante la declaración:

```
[Flags]
public enum DiaSemana
{
    Lunes = 1,
    Martes = 2,
    Miércoles = 4,
    Jueves = 8,
    Viernes = 16,
    Sábado = 32,
    Domingo = 64,
    Trabajo = Lunes | Martes | Miércoles | Jueves | Viernes,
    FinDeSemana = Sábado | Domingo,
    Semana = Trabajo | FinDeSemana
}
```

Los operadores lógicos, como & y |, permiten realizar las operaciones de asignación:

```
// Combinación del miembro Domingo
Dias |= DiaSemana.Domingo;
// Eliminación del miembro Domingo
Dias ^= DiaSemana.Domingo;
// Comprobación de la presencia del miembro Sábado
if ((Dias & DiaSemana.Sábado) != 0)
```

```
{  
    // La variable Dias contiene el miembro Sábado  
}
```

Los arrays

Los arrays permiten agrupar series de variables y acceder a ellos usando un índice, empezando en 0. Los arrays pueden tener una o varias dimensiones.

La declaración de un array se hace añadiendo [] al tipo de datos que se almacenará y su inicialización se hace indicando el número máximo de elementos que puede contener:

```
int[] Array;  
Array = new int[10];
```

La declaración e inicialización se pueden hacer en una única instrucción:

```
int[] Array = new int[10];
```

El array declarado en el código anterior podrá contener 10 elementos de tipo int y tendrá un índice comprendido entre 0 y 9.

Es posible inicializar un array sin especificar el límite máximo, sino especificando una serie de valores. Se indican entre llaves y separados por comas:

```
int[] Array = new int[] { 1, 2, 5, 9, 12 };
```

El Framework .NET también tiene en cuenta los arrays multidimensionales. Los arrays rectangulares son arrays en los que cada registro tiene el mismo número de columnas:

```
int[,] Array = new int[2, 2];  
int[,,] Array = new int[5, 3, 2];
```

Los arrays escalares son otro tipo de arrays multidimensionales. A diferencia de lo que sucede con los arrays rectangulares, los arrays escalares tienen registros que pueden tener un número de columnas diferente. Se trata de tener un array de arrays:

```
int[][] Array = new int[2][];  
Array[0] = new int[] { 2, 5 };  
Array[1] = new int[] { 2, 5, 12, 21 };
```

Los valores de los arrays son accesibles gracias al indexador, que permite acceder directamente a la variable en modo lectura y escritura:

```
int[] Array = new int[10];  
Array[0] = 1;  
for (int i = 1; i < Array.Length; i++)
```

```
{  
    Array[i] = Array[i - 1] * 2;  
}
```

Índices

```
var vocales = new char[] { 'a', 'e', 'i', 'o', 'u' };  
var segundaVocal = vocales[1];  
var ultimaVocal = vocales[^1];  
var penúltimaVocal = vocales[^2];
```

Rangos

```
var vocales = new char[] { 'a', 'e', 'i', 'o', 'u' };  
var dosPrimerasVocales = vocales[..2]; //ultimo digito se excluye  
var tresPrimerasVocales = vocales[..3];  
var dosUltimasVocales = vocales[^2..];  
var tresEnMedio = vocales[1..4];  
var tresEnMedio = vocales[1..^1];
```

Las colecciones

Cuando se escribe una aplicación, con frecuencia sucede que hay que manipular una colección de elementos. El framework .NET ofrece un conjunto amplio de tipos de colecciones listos para usar. Sin embargo, algunos tipos se usarán más que otros, por la sencillez de su API o por su aporte en cuestión de rendimiento.

a. La interfaz IEnumerable

Antes de empezar a hablar de las colecciones como tales, hay que saber que una colección que se puede iterar implementa una interfaz específica: `IEnumerable`. Esta interfaz también dispone de una versión genérica, que permite especificar el tipo de datos de la colección entre los símbolos menor y mayor que: `IEnumerable<T>`, donde `T` corresponde al tipo deseado. Así, la colección `List<T>` (que veremos inmediatamente después) implementa la interfaz `IEnumerable<T>`. Podemos decir que una `List<T>` es una `IEnumerable<T>`.

Todas las colecciones que vamos a ver aquí debajo implementan esta interfaz, lo que permite usar el concepto de polimorfismo objeto cuando se quiere crear métodos que usan una colección cualquiera que se puede iterar. Hay que señalar que la interfaz `IEnumerable` solo presenta una API de lectura y no de escritura. Para tener la posibilidad de modificar el contenido de una colección, es preferible usar el tipo final o una interfaz más permisiva.

b. Las tablas

El primer tipo de colección es innegablemente la tabla de elementos. Una tabla corresponde a una serie finita de elementos, es decir, que se conoce con precisión el tamaño de esta colección cuando se instancia. La sintaxis es muy sencilla: solo hay que añadir corchetes después del tipo de elemento que se quiere poner en la tabla. Durante la instanciación, se define el tamaño con ayuda de un entero positivo colocado entre estos mismos corchetes:

```
type[] tab = new type[10];
```

Por ejemplo, si se quiere crear una tabla de diez enteros:

```
int[] tab = new int[10];
```

Cuando se ha creado una tabla, los elementos que se encuentran en el interior están clasificados en una posición, directamente accesible a través de algo llamado un índice.

Atención: un índice de tabla empieza obligatoriamente en 0 y termina en tamaño - 1. Si intenta acceder a un índice que no existe teniendo en cuenta el tamaño de la tabla, se reenviará un error durante la ejecución en el que se le dirá que ha superado el tamaño máximo.

El acceso a un elemento dentro de una tabla permite tanto la lectura como la escritura. Para realizar este acceso mediante el índice, solo hay que añadir los corchetes después de la instancia de la tabla y colocar el valor del índice deseado:

```
tab[0] = 42;  
tab[10] = 10; // aquí, habrá un error de ejecución
```

Una tabla se puede inicializar desde su construcción usando una sintaxis similar al inicializador de objetos (object initializer):

```
int[] tab = new int[3] { 1, 2, 3 }; // se obtiene una tabla con  
3 valores ya definidos
```

Sin saberlo, ya ha usado la tabla porque esta última está oculta detrás de la implementación de la clase string. En efecto, la clase string es, de hecho, una tabla de caracteres. Además, expone al descriptor de acceso mediante índice:

```
string valor = "Hola a todos";  
var c = valor[3]; // aquí, tendremos el carácter 'j' almacenado en  
esta variable
```

El acceso por índice se ha mejorado en C# 8, dando la posibilidad de definir un índice desde el final. Para hacerlo, solo hay que colocar un acento circunflejo como prefijo del valor del índice. La diferencia es que, partiendo del final, es necesario empezar en el índice 1 en lugar de 0. Si hacemos `tab[^0]`, obtenemos un error de ejecución porque esta notación es equivalente a escribir `tab[3]` para una tabla de tres elementos.

```
int[] tab = new int[3] { 1, 2, 3 };  
var ultimo = tab[^1];
```

C# 8 también ha introducido el concepto de rango, que permite extraer una tabla de otra definiendo un límite mínimo (inclusivo) y un límite máximo (exclusivo), separados por dos puntos. Si se omite el límite mínimo, el compilador supone que parte del inicio de la tabla y, si se omite el límite máximo, considera que va hasta el final de la tabla:

```
int[] tab = new int[10] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int[] subTab = tab[1..3]; // contendrá 2 y 3
int[] finDeTab = tab[8..]; // contendrá 9 y 10
int[] inicioDeTabl = tab[..3]; // contendrá 1, 2 y 3
```

Una tabla puede ser multidimensional (para almacenar una matriz, por ejemplo). Para obtener este modo de funcionamiento, hay que usar la coma para separar los índices:

```
int[,] matriz = new int[3,3]; // se obtiene una matriz de 3 por 3
matriz[0,0] = 1; // se almacena 1 en 1ª fila y 1ª columna
```

Pero las tablas tienen un inconveniente: no es fácil redimensionarlas una vez que han sido definidas. Para lograr este objetivo, hay dos posibilidades:

- Se puede crear una tabla nueva del tamaño deseado y copiar la primera, elemento por elemento, en la segunda.
- Se puede usar el método `Resize` en la clase `Array`.

```
int[] pequenaTabla = new int[3];
// se quiere aumentar la tabla a 10
Array.Resize(ref pequenaTabla, 10);
```

Se observa que la llamada al método `Resize` utiliza una palabra clave que todavía no hemos visto. Esta palabra clave se verá más adelante en el libro, en el capítulo Conceptos avanzados.

Afortunadamente, el framework .NET pone a nuestra disposición una clase que permite simplificar este uso, en caso de que la colección deba poder ser aumentada dinámicamente.

c. La lista

Probablemente se trata del tipo de colección más usado; la lista ofrece la ventaja de exponer una colección de elementos y proporcionar una API que permite añadir, modificar o eliminar elementos sin preocuparse por el tamaño. Para declarar una lista, hay que usar la clase `List` y añadir después de su declaración, entre los símbolos menor y mayor que, el tipo de elementos contenidos en la lista:

```
List<type> list = new List<type>();
```

La declaración de una lista también usa una sintaxis específica, los genéricos, que veremos más adelante de manera detallada, en el capítulo Conceptos avanzados.

La lista ofrece ciertas posibilidades idénticas a las tablas:

- Permite una inicialización simplificada: `List<int> list = new List<int> { 1, 2, 3 };`
- Ofrece un descriptor de acceso por índice: `list[2]`.

Esto se debe al hecho de que `List` usa de manera subyacente una tabla para guardar los elementos y hace el trabajo relacionado con la gestión del tamaño en lugar del desarrollador.

Así, al contrario que la tabla, la clase `List` ofrece métodos que permiten gestionar los elementos. Entre todos los posibles, estos son los más útiles:

- El método `add`, que permite añadir un elemento al final de la lista.
- El método `Remove`, que permite eliminar un elemento al final de la lista.
- El método `Insert`, que permite añadir un elemento al índice deseado.

```
List<int> list = new List<int> { 1, 2, 3 };
```

```
list.Add(4); // la lista contiene 4 elementos
```

```
list.Remove(1); // eliminación del valor 1 (no del elemento en
el índice 1). La lista contiene 2, 3 y 4
```

```
list.Insert(0, 42); // inserción del valor 42 al inicio de lista.
La lista contiene 42, 2, 3, 4
```

La clase `List` también ofrece funciones que permiten buscar uno o varios elementos según diferentes criterios, para recuperarlos o simplemente recuperar su(s) índice(s):

- El método `IndexOf`, que permite recuperar el primer índice partiendo del inicio de un elemento dado.
- El método `LastIndexOf`, que permite recuperar el último índice de un elemento dado.
- El método `Find`, que permite recuperar el primer elemento que responde a una prueba lógica.
- El método `FindLast`, que permite recuperar el último elemento que responde a una prueba lógica.
- El método `FindAll`, que permite recuperar todos los elementos que responden a una prueba lógica.
- El método `FindIndex`, que permite recuperar el primer índice de un elemento que responde a una prueba lógica.

- El método `FindLastIndex`, que permite recuperar el último índice de un elemento que responde a una prueba lógica.

```
List<int> list = new List<int> { 1, 2, 3, 1, 2, 1 };
```

```
int indexDebut = list.IndexOf(1); // 0 se almacenará aquí
```

```
int indexFin = list.LastIndexOf(1); // 5 se almacenará aquí
```

```
int superiorA2 = list.Find(e => e > 2); // tendremos el valor 3 aquí,  
y no el índice
```

```
int igualA1 = list.FindLast(e => e == 1);
```

```
var todosSupIgA2 = list.FindAll(e => e >= 2); //tendremos una  
colección de elementos superiores o iguales a 2
```

```
int primer1 = list.FindIndex(e => e == 1); // tendremos 0 porque es  
el primer índice de un número igual a 1
```

```
int ultimo1 = list.FindLastIndex(e => e == 1); //tendremos 5 porque  
es el último índice de un número igual a 1
```

Como se puede ver, la lista aporta una flexibilidad real respecto a la tabla, ya sea a nivel de la búsqueda o a nivel de la gestión del tamaño.

La lista expone la cantidad de elementos que contiene mediante la propiedad `Count`:

```
var tamano = list.Count; // aquí tendremos 6 retomando  
la variable list del bloque de código anterior
```

d. Los diccionarios

Otra colección muy usada, los diccionarios permiten guardar una lista de pares clave/valor. Esto significa que disponemos de una colección que permite asociar una clave con un valor dado. La clave y el valor son tipos libremente elegidos por el desarrollador.

Debido a la implementación interna del diccionario, se recomienda encarecidamente que el tipo de datos utilizado como clave disponga de un algoritmo de hash. El diccionario usa el valor de hash para implementar un almacenamiento y una búsqueda eficaces sobre la base de la clave. Aunque se pueden usar todos los tipos, los tipos primitivos, como `int` o `string`, son candidatos ideales en cuestión de rendimiento.

El framework .NET pone a su disposición esta colección mediante la clase `Dictionary<TKey, TValue>`. Así, para declarar un diccionario que tiene un entero por clave y una cadena de caracteres por valor, se usa la siguiente declaración:

```
var dico = new Dictionary<int, string>();
```

Una vez creado el diccionario, el desarrollador no tiene que preocuparse de la gestión del tamaño ni de la cantidad de elementos, como ya sucedía con la lista. Así, es posible añadir elementos usando el método Add, que toma como parámetros la clave y el valor asociado:

```
dico.Add(42, "Respuesta universal");
```

Sin embargo, hay que prestar atención. La clase Dictionary solo autoriza los duplicados al nivel de la clave: esa operación de adición de una clave por duplicado provocará un error de ejecución. Por eso, antes de cada adición se recomienda llamar al método ContainsKey para comprobar si la clave ya existe y, si fuera necesario, no añadir el elemento, sino actualizarlo.

```
if(!dico.ContainsKey(42))
{
    dico.Add(42, "Respuesta universal");
}
```

La clase Dictionary también ofrece, del mismo modo que la lista, un descriptor de acceso con corchetes que da acceso al valor asociado, tanto en lectura como en escritura. Esta manera de asignar un valor está protegida porque si la clave no existe, la clave y el valor se insertan, mientras que si la clave existe, el valor se actualiza. Pero tenga cuidado: este descriptor de acceso usa el valor de la clave, y no un índice eventual:

```
dico[42] = "Respuesta universal"; // adición si no existe,
actualización si existe
```

En caso de intento de lectura, no obstante, si la clave no existe, la ejecución encontrará un error:

```
var valor = dico[999]; // si la clave 999 no existe, tendremos
un error de ejecución
```

También es un uso bastante habitual asociar un Dictionary con una colección como una lista, por ejemplo:

```
var dico2 = new Dictionary<int, List<string>>>();
if(!dico2.ContainsKey(42))
{
    dico2.Add(42, new List<string> { "Respuesta universal",
    "Crítica de película" });
}
dico2[42].Add("Otra frase");
```

También se puede inicializar una instancia de Dictionary en su creación. Se ha añadido una sintaxis nueva en C# 6 para reutilizar los corchetes. Como comparación, aquí podemos ver las dos maneras de proceder (ambas son válidas):

```
// a partir de C# 3
var dicolnit1 = new Dictionary<int, string>
```

```

{
    { 1, "Uno" },
    { 2, "Dos" }
};
// a partir de C# 6
var dicolnit2 = new Dictionary<int, string>
{
    [1] = "Uno",
    [2] = "Dos"
};

```

La clase Dictionary expone la cantidad de pares clave/valor que contiene mediante la propiedad Count:

```

var tamano = dicolnit1.Count; // aquí, tendremos 2 si nos basamos
en la variable del programa anterior

```

e. Las colecciones algorítmicas

En esta subsección, vamos a ver dos colecciones que tienen más intención algorítmica porque respetan un patrón bien definido. Aquí hablaremos de las pilas (Stack) y de las filas (Queue).

Estas dos colecciones un poco especiales imponen un orden de lectura y de inserción:

- La clase Stack es una pila que funciona en modo **LIFO** (Last In, First Out = último en entrar, primero en salir). Hay que considerar esto como una pila de platos: tomamos el primer plato de arriba, que generalmente es el último añadido.
- La clase Queue es una fila que funciona en modo **FIFO** (First In, First Out = primero en entrar, primero en salir). Hay que considerar esto como una fila de espera: cuanto más pronto entra en la fila, más pronto sale.

El orden es muy importante en estas colecciones y solo se recomienda usarlas cuando hay una auténtica necesidad, y no de manera general, porque el respeto de este orden es restrictivo.

Para crear una pila de enteros nueva, por ejemplo, se usa el siguiente código:

```

var pila = new Stack<int>();

```

Para alimentar la pila, usamos el método Push:

```

pila.Push(1);
pila.Push(2);
pila.Push(3);

```

Al final del código anterior, los valores 1, 2 y 3 están apilados, con el valor 3 en la parte superior de la pila. Para recuperar el último valor apilado, usamos el método Pop:

```
var tres = pila.Pop();
```

Después de llamar al método Pop, el valor leído se elimina automáticamente de la pila, lo que hace que, en el caso del código anterior, nuestra pila solo contenga los valores 1 y 2, con 2 situado ahora en la parte superior de la pila.

No obstante, existe un método que permite leer el último valor de la pila sin retirarlo; se trata de Peek:

```
var dos = pila.Peek();
```

Cuando se ha llamado al código de arriba, el valor 2 se encuentra en la variable llamada dos, pero la pila sigue conteniendo los valores 1 y 2.

La clase Queue funciona de manera similar, pero el orden de acceso es distinto y los métodos se llaman de diferente manera. Así, para crear una Queue nueva, se usa el siguiente código:

```
var fila = new Queue<int>();
```

Para insertar elementos en la fila, usamos el método Enqueue:

```
fila.Enqueue(1);  
fila.Enqueue(2);  
fila.Enqueue(3);
```

Cuando se ejecuta el código anterior, los valores 1, 2 y 3 se añaden a la fila. Para leer el primer valor, usamos el método Dequeue:

```
var uno = fila.Dequeue();
```

Del mismo modo que la clase Stack, la llamada al método Dequeue elimina el elemento en la lectura, lo que hace que la fila solo contenga los valores 2 y 3, donde 2 se ha convertido en el primero de la fila.

Al igual que la clase Stack, la clase Queue expone el método Peek, que permite leer el primer elemento sin retirarlo de la fila:

```
var dos = fila.Peek();
```

Si llamamos al método Pop en una Stack que está vacía o al método Dequeue en una Queue que está vacía, habrá un error de ejecución.

La clase Queue y la clase Stack, ambas exponen la cantidad de elementos que contienen mediante la propiedad Count.

Aunque todavía existen muchas otras colecciones disponibles en el framework .NET, aquí hemos hablado de las más utilizadas. Sin embargo, una colección no sirve de mucho si no se

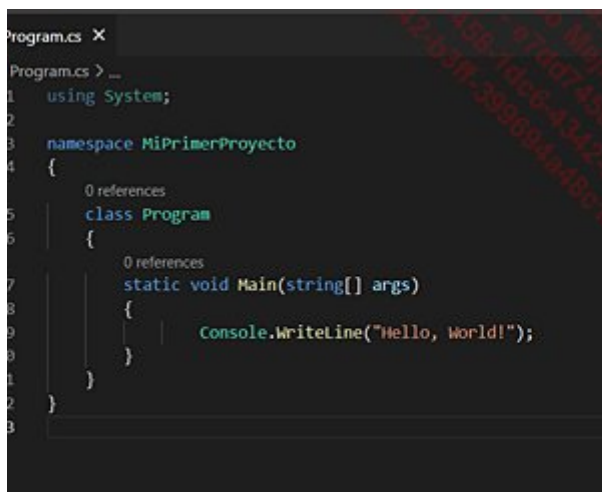
puede iterar (es decir, que no se puede recorrer el conjunto de sus elementos). Esto nos permite abordar la lógica algorítmica del bucle.

Analizar la estructura de un proyecto C#

Un archivo de código C# puede contener instrucciones diversas y variadas. C# 9 ha contribuido en gran medida a simplificar la escritura de programas sencillos.

1. El concepto de bloques

Para poder estudiar el concepto de bloques, vamos a tomar un archivo modelo de lo que era una aplicación de consola antes de la llegada de .NET 6 y de su modelo simplificado. Este es el código:



```
Program.cs X
Program.cs > ...
1 using System;
2
3 namespace MiPrimerProyecto
4 {
5     0 references
6     class Program
7     {
8         0 references
9         static void Main(string[] args)
10        {
11            Console.WriteLine("Hello, World!");
12        }
13    }
14 }
```

Aplicación de consola antes de .NET 6

Por supuesto, esto es más complejo que la versión que hemos conocido, pero contiene conceptos fundamentales sobre el código C#. En todas las explicaciones que siguen, vamos a tener en cuenta este código.

Algunos caracteres aparecen varias veces en un archivo de C# y deberán ser identificados de manera automática. Lo primero que vemos es que hay un sangrado (un desplazamiento de las líneas hacia la derecha) y que este sangrado está vinculado a los caracteres llave («{» y «}»).

De manera bastante sencilla, el lenguaje C# funciona por bloques.

Un bloque es un fragmento de código que tiene su propio contexto y puede contener otros bloques.

Por ejemplo, en nuestro archivo vemos un primer bloque que empieza en la línea 4 y termina en la línea 12. Este bloque contiene otro bloque que empieza en la línea 6 y termina en la línea 11. Y, finalmente, este último también contiene otro bloque que va desde la línea 8 hasta la línea 10. La única instrucción que existe fuera de los bloques es la que aparece en la línea 1, que veremos más adelante.

El denominador común de todos estos bloques es que tienen «un título». Por ejemplo, para el bloque de mayor nivel, la línea 3 define lo que contiene y se puede considerar como «su título». Luego, para el segundo es la línea 5, y la línea 7 para el último.

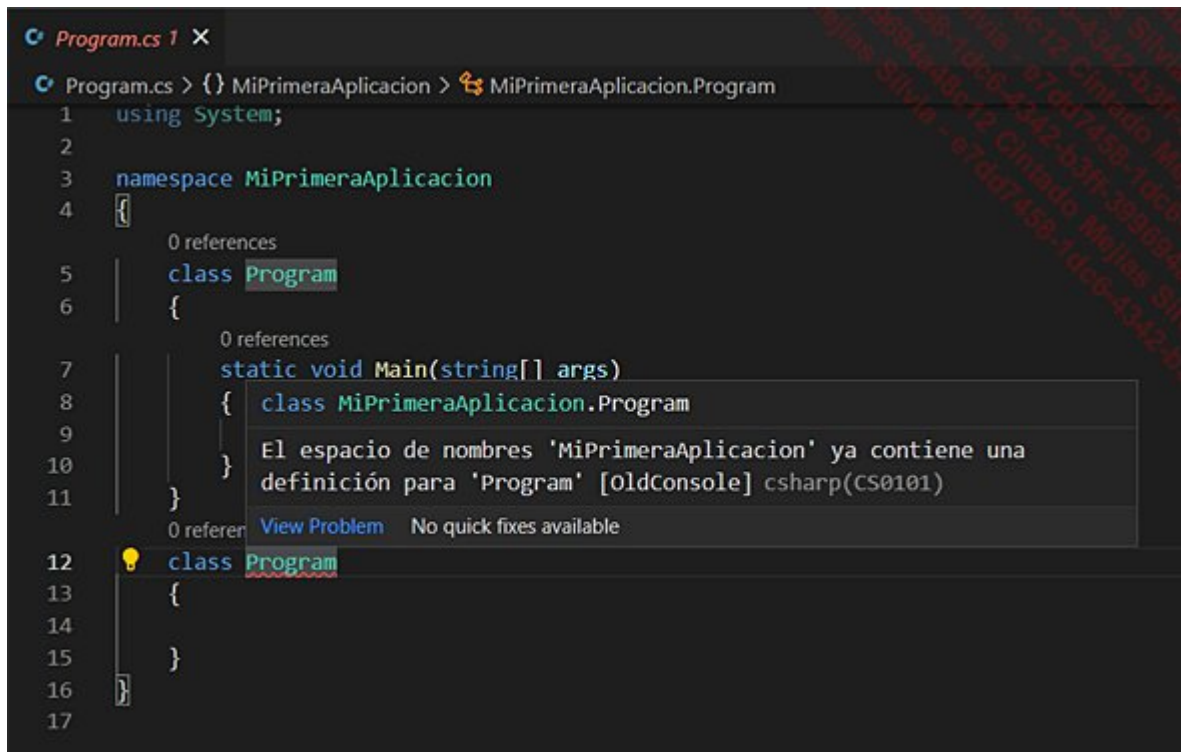
En C# se pueden crear bloques sin un concepto de «título», pero tiene muy poco interés y solamente hará que la lectura sea compleja porque habrá un desplazamiento debido al sangrado.

Otro concepto importante que hay que conocer: lo que está definido dentro de un bloque solo existe dentro de él. Esto no quiere decir que sea imposible ver o usar los elementos definidos dentro de un bloque dado (hablaremos de eso más adelante, cuando tratemos los conceptos de ámbito), pero su definición sólo existe dentro del marco del bloque donde se han creado inicialmente.

De la misma manera, no es posible crear la misma cosa dos veces dentro de un mismo bloque; esto provoca un error de compilación (lo que hace que sea imposible ejecutar la aplicación porque no se puede producir). Para ilustrar de manera concreta este ejemplo, no es posible colocarse entre las líneas 11 y 12 y escribir el siguiente código:

```
class Program
{
}
```

Se subrayará la palabra «Program» en rojo (exactamente como una falta de ortografía en Word). Es posible obtener la causa exacta de este error si se coloca el ratón sobre palabra subrayada, mediante una pequeña ventana emergente, como se puede ver en la captura de pantalla siguiente:



Error cuando la clase Program está definida dos veces en el mismo bloque

Incluso si el mensaje de error puede parecer poco esclarecedor en este momento, indica de manera sencilla que el bloque definido desde la línea 4 hasta la 16 ya contiene una definición completamente idéntica de Program.

2. Significado de los bloques de código

Ahora que hemos adquirido el concepto, vamos a explicar en detalle los bloques principales con su título y significado para comprender mejor cuál es su utilidad.

a. El bloque de espacio de nombres

El primer bloque que aparece en nuestro programa es el del espacio de nombres; aquí se ilustra mediante el código:

```
namespace MiPrimeraAplicacion
{
    ...
}
```

En general, este bloque se declara en el primer nivel porque es el que suele contener a todos los demás. El lenguaje C# ordena y clasifica sus distintos elementos dentro de este concepto de espacio de nombres. Un espacio de nombres se puede considerar como una caja donde se pueden guardar objetos.

La declaración de un espacio de nombres responde de manera sistemática a la siguiente sintaxis:

```
namespace UN_ESPACIO_DE_NOMBRES
{
}
```

En concreto, la instrucción de la línea 3 declara un espacio de nombres llamado «MiPrimeraAplicacion», y todo lo que está entre las llaves después de esta declaración pertenece a este espacio de nombres.

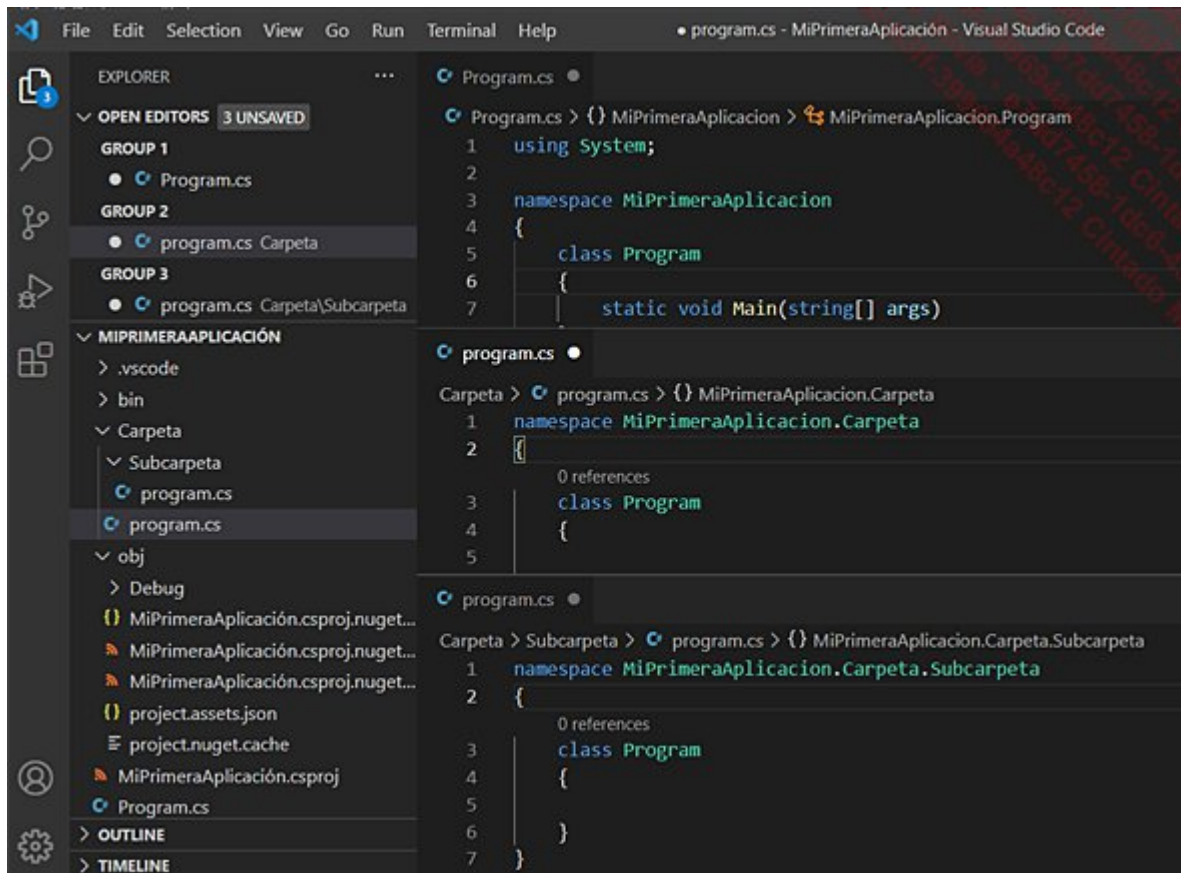
Si nos encontramos fuera de este espacio de nombres, no es posible ver y usar directamente lo que se encuentra en el interior. Primero hay que importar el espacio de nombres. Esta etapa se traduce mediante una instrucción `using` colocada en el encabezado del archivo. Por ejemplo, en nuestro primer archivo de código ya aparece la importación del espacio de nombres `System`, definido en el framework `.NET`. Gracias a esta instrucción, podemos usar todos los objetos que se encuentran directamente dentro de este espacio de nombres. La sintaxis siempre es la misma:

```
using UN_ESPACIO_DE_NOMBRES;
```

Un espacio de nombres también puede contener subespacios de nombres. En concreto, las buenas prácticas quieren que en `C#` el nombre del proyecto sea el espacio de nombres predeterminado (aquí, en nuestro ejemplo, `MiPrimeraAplicacion` es el espacio de nombres raíz porque es el nombre de nuestro proyecto), y cada subcarpeta del proyecto se convierte en un subespacio de nombres donde cada nivel está separado por un punto.

Por ejemplo, si en nuestro proyecto creamos una carpeta que decidimos llamar «Carpeta», cada elemento creado dentro de esta carpeta tiene como espacio de nombres «`MiPrimeraAplicacion.Carpeta`». De la misma manera, si creamos otra carpeta dentro de esta última y la llamamos «`SubCarpeta`», el espacio de nombres de los elementos en este nivel de jerarquía es «`MiPrimeraAplicacion.Carpeta.SubCarpeta`». No se preocupe con la práctica lo hará de manera automática.

Usando este concepto de subespacio de nombres, podemos volver a definir un nombre de elemento que ya existe, siempre y cuando sus espacios de nombres sean distintos. Por ejemplo, podríamos tener un archivo `Program.cs` (incluso si no se recomienda) que contiene la definición presente en la línea 5 dentro del espacio de nombres `Carpeta`, y la misma definición dentro del espacio de nombres `SubCarpeta`, siempre conservando la que existe en la raíz, como se muestra en esta imagen:



Misma definición en varios espacios de nombres distintos

Al inicio del archivo, una instrucción using importa un espacio de nombres. Desde C# 10 ya no es necesario colocar las mismas instrucciones al inicio de cada archivo fuente porque a partir de ahora se puede realizar un using global, es decir, se aplica para todos los archivos del proyecto. Para hacer eso, solo hay que usar la palabra clave global con la instrucción using:

```
using global System;
```

Al hacer esto, el espacio de nombres System está disponible para todos los archivos de código C# dentro del proyecto. Para conservar una cierta claridad, es muy recomendable tener un archivo dedicado a la importación de todos los espacios de nombres globales, con objeto de evitar tener que buscar dónde se ha importado un espacio de nombres de manera global. Para hacerlo, podríamos considerar un archivo de código C#, llamado **Usings.cs**, por ejemplo, situado en la raíz del proyecto, que sólo contendría las instrucciones using globales.

En resumen: el punto esencial que debe recordar es que un espacio de nombres le permite estructurar la aplicación. Gracias a ellos, podrá definir cosas distintas sin que estos últimos entren en conflicto. La convención también quiere que la jerarquía de carpetas en el sistema de archivos tenga una correspondencia con el espacio de nombres dentro del código C#.

b. Definición de una clase

En otro capítulo más adelante, volveremos a hablar más detalladamente del concepto de clase para estudiar los conceptos de programación correspondientes. Por el momento, vamos a ver cómo funciona la declaración de una clase.

En nuestro archivo Program.cs, en la línea 5, tenemos la manera de definir una clase en C#. Relativamente sencilla, la sintaxis es la siguiente:

```
class NOMBRE_DE_LA_CLASE
```

Más tarde estudiaremos otras maneras de declarar una clase pero, por el momento, lo único que hay que saber sobre este concepto es que la clase es un segundo subelemento y que esta última generalmente está «ordenada» dentro de un espacio de nombres.

Cuando se ha definido una clase, la finalidad es almacenar en ella diversas cosas, como datos y comportamientos. Pronto estudiaremos este contenido de manera detallada, pero eso nos lleva directamente al estudio del siguiente bloque.

En resumen: usamos el bloque de tipo clase para definir un conjunto conectado de datos y de comportamientos que tienen un vínculo funcional.

c. Definición de un método

El último bloque presente en nuestro archivo de ejemplo es la definición de método, desde la línea 7 hasta la 9. Un método es un comportamiento que se puede ejecutar y que va a cumplir un conjunto de tareas.

Hay que considerar muchos elementos dentro de la declaración de sintaxis de un método; por eso no vamos a tratar este tema de forma inmediata, sino que simplemente vamos a usar el contenido de este método para comprender cómo se puede escribir código C# que «hace algo». Se puede añadir un conjunto muy significativo de cosas en el interior de un método, y hablaremos de esto justo después de esta sección.

En resumen: usamos el bloque de tipo método para definir un conjunto de instrucciones que se ejecutarán durante el lanzamiento de nuestra aplicación.

3. Declaración «top-level»

C# 9 introdujo una novedad que permite simplificar drásticamente la escritura de programas sencillos: top-level statements. Gracias a esta novedad es posible prescindir de los bloques vistos anteriormente. Se trata de la versión inicial de nuestra aplicación.

Esta novedad se ha convertido en el estándar durante la creación de aplicaciones nuevas (consola como ASP.NET). A pesar del hecho de que muchos elementos hayan «desaparecido» de esta nueva versión, estos existen. El compilador se encarga de generar automáticamente el código necesario.

Hay que considerar que el código que escribimos dentro del archivo Program.cs se encuentra directamente inyectado dentro del método Main, que ya está colocado dentro de la clase Program, todo ello dentro del espacio de nombres del proyecto. Hemos añadido este enfoque para permitir escribir con rapidez un programa sin una sintaxis compleja, al igual que lo permiten otros lenguajes (JavaScript, Python, etc.).

Sin embargo, preste atención: esta clase de archivo sólo se puede obtener una única vez por proyecto y solo para el método Main.