UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

INTRODUCCION A LA PROGRAMACION C#

4 - MÓDULO 2 : VARIABLES Y TIPOS DE DATOS

Universidad Tecnológica Nacional - Facultad Regional Buenos Aires
Medrano 951 (C1179AAQ) C.A.B.A. Tel: (54 11) 4867-7511

UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

UNIDAD: 4

MODULO: 2

PRESENTACIÓN: En esta unidad se explicarán los conceptos de variables y tipos de datos del lenguaje C#.

OBJETIVOS

Que los participantes logren: Comprender los conceptos de esta unidad y aplicar en la practica dichos conceptos.

UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

TEMARIO

Estructura básica de una Clase

Si bien mas adelante profundizaremos los conceptos de clase, su estructura y utilización, a los fines de abordar el tema variables, vamos a definir básicamente una clase y su estructura.

Declarar clases

Las clases se declaran mediante la palabra clave class, como se muestra en el siguiente ejemplo:

```
public class Cliente
{
    // Campos, metodos, propiedades, constructores etc van aca...
}
```

La palabra clave class va precedida del nivel de acceso. Como en este caso se usa public, cualquier usuario puede crear instancias de esta clase. El nombre de la clase sigue a la palabra clave class. El resto de la definición es el cuerpo de la clase, donde se definen los datos y el comportamiento. Los campos, las propiedades, los métodos y los eventos de una clase se denominan de forma colectiva miembros de clase.

Crear objetos

Aunque a veces se usan indistintamente, una clase y un objeto son cosas diferentes. Una clase define un tipo de objeto, pero no es un objeto en sí. Un objeto es una entidad concreta basada en una clase y, a veces, se conoce como una instancia de una clase.

Los objetos se pueden crear usando la palabra clave new, seguida del nombre de la clase en la que se basará el objeto, como en este ejemplo:

```
Cliente cliente = new Cliente();
```

Cuando se crea una instancia de una clase, se vuelve a pasar al programador una referencia al objeto. En el ejemplo anterior, cliente es una referencia a un objeto que se basa en Cliente.

Definición de variables

Una variable puede verse simplemente como un almacén de objetos de un determinado tipo al que se le da un cierto nombre. Por tanto, para definir una variable sólo hay que decir cuál será el nombre que se le dará y cuál será el tipo de datos que podrá almacenar, lo que se
hace con la siguiente sintaxis:

```
<tipoVariable> <nombreVariable>;
```

Una variable puede ser definida dentro de una definición de clase, en cuyo caso se ria un campo. También puede definirse como un variable local a un método, que es una variable definida dentro del código del método a la que sólo puede accederse desde dentro de dicho código. Otra posibilidad es definirla como parámetro de un método, que son variables que almacenan los valores de llamada al método y que, al igual que las variables locales, sólo puede ser accedidas desde código ubicado dentro del método. El siguiente ejemplo muestra como definir variables de todos estos casos:

```
class A
{
      int x, z;
      int y;
      void F(string a, string b)
      {
            Persona p;
      }
}
```

En este ejemplo las variables x , z e y son campos de tipo int , mientras que p es una variable local de tipo Persona y a y b son parámetros de tipo string .

Como se muestra en el ejemplo, si un método toma varios parámetros las definiciones de éstos se separan mediante comas (carácter ,), y si queremos definir varios campos o variables locales (no válido para parámetros) de un mismo tipo podemos incluirlos en una misma definición incluyendo en <nombreVariable> sus nombres separados por comas.

Con la sintaxis de definición de variables anteriormente dada simplemente definimos variables pero no almacenamos ningún objeto inicial en ellas. El compilador dará un valor por defecto a los campos para los que no se indique explícitamente ningún valor según se explica en el siguiente apartado. Sin embargo, a la variables locales no les da ningún valor inicial, pero detecta cualquier intento de leerlas antes de darles valor y produce errores de compilación en esos casos.

Ejemplo de identificadores:

MiVariable

_contador

Contado_

mi_variable_

Ejemplos no validos:

1Variable: error porque comienza con un número.

$Variable: error porque comienza con un signo $.

###Contador###: error porque comienza con un signo #.

default: error porque default es una palabra reservada.

Lo ideal es que el identificador de una variable vaya acorde a su función, es decir un nombre que describa lo mejor posible lo que representa dentro del programa.

Asignación de valores a variables de tipo objetos:

Ya hemos visto que para crear objetos se utiliza el operador new. Por tanto, una forma de asignar un valor a la variable p del ejemplo anterior sería así:.

```
Persona p;
p = new Persona("José", 22, "76543876-A ");
```

Sin embargo, C# también proporciona una sintaxis más sencilla con la que podremos asignar un objeto a una variable en el mismo momento se define. Para ello se la ha de definir usando esta otra notación:

```
<tipoVariable> <nombreVariable> = <valorInicial>;
```

Así por ejemplo, la anterior asignación de valor a la variable p podría reescribirse de esta otra forma más compacta:

```
Persona p = new Persona("José", 22, "76543876-A");
```

La especificación de un valor inicial también combinarse con la definición de múltiples variables separadas por comas en una misma línea. Por ejemplo, las siguientes definiciones son válidas:

```
Persona p1 = new Persona("José", 22, "76543876-A"), p2 = new Persona("Juan", 21, "87654212-S");
```

Y son tratadas por el compilador de forma completamente equivalentes a haberlas declarado como:

```
Persona p1 = new Persona("José", 22, "76543876-A");
Persona p2 = new Persona("Juan", 21, "87654212-S");
```

Tipos de datos básicos

Los tipos de datos básicos son ciertos tipos de datos tan comúnmente utilizados en la escritura de aplicaciones que en C# se ha incluido una sintaxis especial para tratarlos. Por ejemplo, para representar números enteros de 32 bits con signo se utiliza el tipo de dato
System.Int32 definido en la BCL, aunque a la hora de crear un objeto a de este tipo que represente el valor 2 se usa la siguiente sintaxis:

```
System.Int32 a = 2;
```

Como se ve, no se utiliza el operador new para crear objeto System.Int32 , sino que directamente se indica el literal que representa el valor a crear, con lo que la sintaxis necesaria para crear entero de este tipo se reduce considerablemente. Es más, dado lo frecuente que es el uso de este tipo también se ha predefinido en C# el alias (ver columna "Tipo C#" en la siguiente tabla) int para el mismo, por lo que la definición de variable anterior queda así de
compacta:

```
int a = 2;
```

Estructura	Detalles	Bits	Intervalo de valores	Tipo C#
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	$[1{,}5\times10^{-45}$, $3{,}4\times10^{38}]$	float
Double	Reales de 15-16 dígitos de precisión	64	$[5{,}0\times10^{-324}$, $1{,}7\times10^{308}]$	double
Decimal	Reales de 28-29 dígitos de precisión	128	$[1{,}0\times10^{-28}$, $7{,}9\times10^{28}]$	decimal
Boolean	Valores lógicos	32	true, false	bool
Char	Caracteres unicode	16	['\u0000', '\uFFFF']	char
String	Cadenas de caracteres	Variable	El permitido por la memoria	string
Object	Cualquier objeto	Variable	Cualquier objeto	object

Tipos por valor y tipos por referencia

Hay dos clases de tipos en C#: tipos de valor y tipos de referencia. Las variables de tipos de valor contienen directamente los datos, mientras que las variables de los tipos de referencia almacenan referencias a los datos, lo que se conoce como objetos. Con los tipos de referencia, es posible que dos variables hagan referencia al mismo objeto y que, por tanto, las operaciones en una variable afecten al objeto al que hace referencia la otra variable. Con los tipos de valor, cada variable tiene su propia copia de los datos y no es posible que las operaciones en una variable afecten a la otra (excepto en el caso de las variables de parámetro ref y out).

Espacio de nombres o namespace

Para definir un espacio de nombres se utiliza la siguiente sintaxis:

```
namespace <nombreEspacio>
{
<tipos>
}
```

Los tipos que se definan en <tipos> pasarán a considerase pertenecientes al espacio de nombres llamado <nombreEspacio> . Como veremos más adelante, aparte de clases esto tipos pueden ser también interfaces, estructuras, tipos enumerados y delegados. A continuación se muestra un ejemplo en el que definimos una clase de nombre ClaseEjemplo perteneciente a un espacio de nombres llamado EspacioEjemplo :

```
namespace EspacioEjemplo
{
      class ClaseEjemplo
      {
      }
}
```

El verdadero nombre de una clase, al que se denomina nombre completamente calificado, es el nombre que le demos al declararla prefijado por la concatenación de todos los espacios de nombres a los que pertenece ordenados del más externo al más interno y seguido cada uno de ellos por un punto (carácter .) Por ejemplo, el verdadero nombre de la clase ClaseEjemplo antes definida es EspacioEjemplo.ClaseEjemplo .

Si no definimos una clase dentro de una definición de espacio de nombres -como se ha hecho en los ejemplos de temas previos- se considera que ésta pertenece al denominado espacio de nombres global y su nombre completamente calificado coincidirá con el nombre que le demos al definirla.

Aparte de definiciones de tipo, también es posible incluir como miembros de un espacio de nombres a otros espacios de nombres. Es decir, como se muestra el siguiente ejemplo es posible anidar espacios de nombres:

```
namespace EspacioEjemplo
{
      namespace EspacioEjemplo2
      {
            class ClaseEjemplo
            {
            }
      }
}
```

Sentencia using

En principio, si desde código perteneciente a una clase definida en un cierto espacio de nombres se desea hacer referencia a tipos definidos en otros espacios de nombres, se ha de referir a los mismos usando su nombre completamente calificado. Por ejemplo:

```
namespace EspacioEjemplo.EspacioEjemplo2
{
      class ClaseEjemplo
      {
      }
}

class Principal // Pertenece al espacio de nombres global
{
      public static void Main ()
      {
            EspacioEjemplo.EspacioEjemplo2.ClaseEjemplo c = new
                  EspacioEjemplo.EspacioEjemplo2.ClaseEjemplo();
      }
}
```

Como puede resultar muy pesado tener que escribir nombres tan largos en cada referencia a tipos así definidos, en C# se ha incluido un mecanismo de importación de espacios de nombres que usa la siguiente sintaxis:

```
using <espacioNombres>;
```

Este tipo de sentencias siempre ha de aparecer dentro de una definición de espacio de nombres antes que cualquier definición de miembros de la misma y permiten indicar cuáles serán los espacios de nombres que se usarán implícitamente dentro de ese espacio de nombres. A los miembros de los espacios de nombres así importados se les podrá hacer referencia sin tener que usar calificación completa, como muestra la siguiente versión del último ejemplo:

```
using EspacioEjemplo.EspacioEjemplo2;

class Principal // Pertenece al espacio de nombres global
{
      public static void ()
      {
            // EspacioEjemplo.EspacioEjemplo2. está implícito
            ClaseEjemplo c = new ClaseEjemplo();
      }
}
```

Alias

Aún en el caso de que usemos espacios de nombres distintos para diferenciar clases con igual nombre pero procedentes de distintos fabricantes, podrían darse conflictos sin usamos sentencias using para importar los espacios de nombres de dichos fabricantes ya que entonces al hacerse referencia a una de las clases comunes con tan solo su nombre simple el compilador no podrá determinar a cuál de ellas en concreto nos referimos.

Por ejemplo, si tenemos una clase de nombre completamente calificado A.Clase , otra de nombre B.Clase , y hacemos:

```
using A;
using B;

class EjemploConflicto {

      static void Main()
      {
            Clase myClase =new Clase();
      }
}
```

¿Cómo sabrá el compilador si lo que queremos es derivar de A.Clase o de B.Clase ? En realidad el compilador no puede determinarlo y producirá un error informando de que hay una referencia ambigua a Clase .

Para resolver ambigüedades de este tipo podría hacerse referencia a los tipos en conflicto usando siempre sus nombres completamente calificados, pero ello puede llegar a ser muy fatigoso sobre todo si sus nombres son muy largos. Para solucionar los conflictos de nombres sin tener que escribir tanto se ha incluido en C# la posibilidad de definir alias para cualquier tipo de dato, que son sinónimos para los mismos que se definen usando la siguiente sintaxis:

```
using <alias> = <nombreCompletoTipo>;
```

Como cualquier otro using , las definiciones de alias sólo pueden incluirse al principio de las definiciones de espacios de nombres y sólo tienen validez dentro de las mismas.
Definiendo alias distintos para los tipos en conflictos se resuelven los problemas de ambigüedades. Por ejemplo, el problema del ejemplo anterior se podría resolver así:

```
using A;
using B;
using ClaseA = A.Clase;

class EjemploConflicto {

      static void Main()
      {
            Clase myClase =new ClaseA();
      }
}
```

El costo de agregar using

Las sentencias namespace y using son construcciones que se usan en tiempo de compilación.

En tiempo de ejecución las referencias a clases y miembros se hacen por nombre calificado completo.

Tener sentencias using tiene un mínimo impacto en tiempo de compilación porque el compilador tiene que buscar por mas namespaces para encontrar cada clase.

Operadores

Operadores	Descripción	Tipo
(expresión)	Control de precedencia	Primario
objeto.miembro	Acceso a miembro de objeto	Primario
método(argumento, argumento, ...)	Enumeración de argumentos	Primario
array[indice]	Elemento de un array	Primario
var++, var--	Postincremento y postdecremento	Primario
new	Creación de objeto	Primario
typeof	Recuperación de tipo (reflexión)	Primario
sizeof	Recuperación de tamaño	Primario
checked, unchecked	Comprobación de desbordamiento	Primario
+	Operando en forma original	Unitario
-	Cambio de signo	Unitario
!	Not lógico	Unitario
~	Complemento bit a bit	Unitario
++var, --var	Preincremente y predecremento	Unitario
(conversión) var	Conversión de tipos	Unitario
*, /	Multiplicación, división	Binario
%	Resto de división	Binario
+, -	Suma, resta	Binario
<<, >>	Desplazamiento de bits	Binario
<, >, <=, >=, is, ==, !=	Relacionales	Binario
&	AND a nivel de bits	Binario
^	XOR a nivel de bits	Binario
|	OR a nivel de bits	Binario
&&	AND lógico	Binario
||	OR lógico	Binario
? :	QUESTION	Binario
=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=	De asignación	Binario

Palabras reservadas

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

Arreglos

Un arreglo es un tipo especial de variable que es capaz de almacenar en su interior y de manera ordenada uno o varios datos de un determinado tipo.

Para declarar variables de este tipo especial se usa la siguiente sintaxis:

```
<tipoDatos>[] <nombreArray>;
```

Por ejemplo, una tabla que pueda almacenar objetos de tipo int se declara así:

```
int[] tabla;
```

Con esto la tabla creada no almacenaría ningún objeto, sino que valdría null . Si se desea que verdaderamente almacene objetos hay que indicar cuál es el número de objetos que podrá almacenar, lo que puede hacerse usando la siguiente sintaxis al declararla:

```
<tipoDatos>[] <nombreTabla> = new <tipoDatos>[<númeroDatos>];
```

Por ejemplo, una tabla que pueda almacenar 100 objetos de tipo int se declara así:

```
int[] tabla = new int[100];
```

Aunque también sería posible definir el tamaño de la tabla de forma separada a su declaración de este modo:

```
int[] tabla;
tabla = new int[100];
```

Con esta última sintaxis es posible cambiar dinámicamente el número de elementos de una variable tabla sin más que irle asignando nuevas tablas.

Ello no significa que una tabla se pueda redimensionar conservando los elementos que tuviese antes del cambio de tamaño, sino que ocurre todo lo contrario: cuando a una variable tabla se le asigna una tabla de otro tamaño, sus elementos antiguos son sobreescritos por los nuevos.

Si se crea una tabla con la sintaxis hasta ahora explicada todos sus elementos tendrían el valor por defecto de su tipo de dato. Si queremos darles otros valores al declarar la tabla, hemos de indicarlos entre llaves usando esta sintaxis:

```
<tipoDatos>[] <nombreTabla> = new <tipoDatos>[] {<valores>};
```

Ha de especificarse tantos <valores> como número de elementos se desee que tenga la tabla, y si son más de uno se han de separar entre sí mediante comas (,) Nótese que ahora no es necesario indicar el número de elementos de la tabla (aunque puede hacerse) si se desea),sí mediante comas (,) Nótese que ahora no es necesario indicar el número de elementos de la tabla (aunque puede hacerse) si se desea), Ejemplo:

```
int [] misEnteros = new int[] {10, 2, 45, 23, 29};
```

Nótese que ahora no es necesario indicar el número de elementos de la tabla (aunque puede hacerse) si se desea), pues el compilador puede deducirlo del número de valores especificados.

Incluso se puede compactar aún más la sintaxis declarando la tabla así:

```
int[] tabla = {10, 2, 45, 23, 29};
```

También podemos crear tablas cuyo tamaño se pueda establecer dinámicamente a partir del valor de cualquier expresión que produzca un valor de tipo entero. Por ejemplo, para crear una tabla cuyo tamaño sea el valor indicado por una variable de tipo int (luego su valor será de tipo entero) se haría:

```
int i = 5;
...
int[] tablaDinámica = new int[i];
```

A la hora de acceder a los elementos almacenados en una tabla basta indicar entre corchetes, y a continuación de la referencia a la misma, la posición que ocupe en la tabla el elemento al que acceder. Cuando se haga hay que tener en cuenta que en C# las tablas se indexan desde 0, lo que significa que el primer elemento de la tabla ocupará su posición 0, el segundo ocupará la posición 1, y así sucesivamente para el resto de elementos. Por ejemplo, aunque es más ineficiente, la tabla declarada en el último fragmento de código de ejemplo también podría haberse definido así:

```
int[] tabla = new int[4];
tabla[0] = 5;
tabla[1]++; // Por defecto se inicializó a 0, luego ahora el valor
de tabla[1] pasa a ser 1
tabla[2] = tabla[0] - tabla[1]; // tabla[2] pasa a valer 4, pues 5-4
= 1


// El contenido de la tabla será {5,1,4,0}, pues tabla[3] se
inicializó por defecto a 0.
```

Hay que tener cuidado a la hora de acceder a los elementos de una tabla ya que si se especifica una posición superior al número de elementos que pueda almacenar la tabla se producirá una excepción de tipo System.OutOfBoundsException .

Por ahora basta considerar que las excepciones son objetos que informan de situaciones no esperadas (generalmente errores) producidas durante la ejecución de una aplicación. Para evitar este tipo de excepciones puede consultar el valor del campo[1] de sólo lectura Length que está asociado a toda tabla y contiene el número de elementos de la misma. Por ejemplo, para asignar un 7 al último elemento de la tabla anterior se haría:

```
tabla[tabla.Length - 1] = 7;


// Se resta 1 porque tabla.Length devuelve 4 pero el último
/ /elemento de la tabla es tabla[3]
```

Arreglos multidimensionales o Matrices

Una matriz es una tabla cuyos elementos se encuentran organizando una estructura de varias dimensiones. Para definir este tipo de tablas se usa una sintaxis similar a la usada para declarar tablas unidimensionales pero separando las diferentes dimensiones mediante comas (,).

Por ejemplo, una matriz de elementos de tipo int que conste de 12 elementos puede tener sus elementos distribuidos en dos dimensiones formando una estructura 3x4 similar a una matriz de la forma:

1	2	3	4
5	6	7	8
9	10	11	12

Esta matriz se podría declarar así:

```
int[,] matriz = new int[3,4] {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
```

En realidad no es necesario indicar el número de elementos de cada dimensión de la tabla ya que pueden deducirse de los valores explícitamente indicados entre llaves, por lo que la definición anterior es similar a esta:

```
int[,] matriz = new int[,] {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
```

Incluso puede reducirse aún más la sintaxis necesaria quedando tan sólo:

```
int[,] matriz = {{1,2,3,4}, {5,6,7,8},{9,10,11,12}};
```

Si no queremos indicar explícitamente los elementos de la tabla al declararla, podemos obviarlos pero aún así indicar el tamaño de cada dimensión de la tabla (a los elementos se les daría el valor por defecto de su tipo de dato) así:

```
int[,] matriz = new int[3,4];
```

También podemos no especificar ni siquiera el número de elementos de la tabla de esta forma (matrizl contendría ahora null):

```
int[,] matriz;
```

Aunque los ejemplos de tablas multidimensionales hasta ahora mostrados son de tablas de dos dimensiones, en general también es posible crear tablas de cualquier número de dimensiones. Por ejemplo, una tabla que almacene 24 elementos de tipo int y valor 0 en una
estructura tridimensional 3x4x2 se declararía así:

```
int[,,]  matriz = new int[3,4,2];
```

El acceso a los elementos de una tabla multidimensional es muy sencillo: sólo hay que indicar los índices de la posición que ocupe en la estructura multidimensional el elemento al que se desee acceder. Por ejemplo, para incrementar en una unidad el elemento que ocupe la posición (1,3,2) de la tabla anterior se haría (se indexa desde 0):

```
matriz [0,2,1]++;
```

Matrices irregulares (Tablas dentadas o Jagged Arrays)

Una tabla dentada no es más que una tabla cuyos elementos son a su vez tablas, pudiéndose así anidar cualquier número de tablas. Para declarar tablas de este tipo se usa una sintaxis muy similar a la explicada para las tablas unidimensionales solo que ahora se indican tantos corchetes como nivel de anidación se desee.

Por ejemplo, para crear una tabla de tablas de elementos de tipo int formada por dos elementos, uno de los cuales fuese una tabla de elementos de tipo int formada por los elementos de valores 1,2 y el otro fuese una tabla de elementos de tipo int y valores 3,4,5, se puede hacer:

```
int[][] tablaDentada = new int[2][] {new int[] {1,2}, new int[]
{3,4,5}};
```

Como se indica explícitamente cuáles son los elementos de la tabla declarada no hace falta indicar el tamaño de la tabla, por lo que la declaración anterior es equivalente a:

```
int[][] tablaDentada = new int[][] {new int[] {1,2}, new int[]
{3,4,5}};
```

Es más, igual que como se vio con las tablas unidimensionales también es válido hacer:

```
int[][] tablaDentada = {new int[] {1,2}, new int[] {3,4,5}};
```

Si no quisiésemos indicar cuáles son los elementos de las tablas componentes, entonces tendríamos que indicar al menos cuál es el número de elementos que podrán almacenar (se inicializarán con valores por defecto) quedando:

```
int[][] tablaDentada = {new int[2], new int[3]};
```

Si no queremos crear las tablas componentes en el momento de crear la tabla dentada, entonces tendremos que indicar por lo menos cuál es el número de tablas componentes posibles (cada una valdría null), con lo que quedaría:

```
int[][] tablaDentada = new int[2][];
```

Es importante señalar que no es posible especificar todas las dimensiones de una tabla dentada en su definición si no se indica explícitamente el valor inicial de éstas entre llaves. Es decir, esta declaración es incorrecta:

```
int[][] tablaDentada = new int[2][5];
```

Esto se debe a que el tamaño de cada tabla componente puede ser distinto y con la sintaxis anterior no se puede decir cuál es el tamaño de cada una.

Una opción hubiese sido considerar que es 5 para todas como se hace en Java, pero ello no se ha implementado en C# y habría que declarar la tabla de, por ejemplo, esta manera:

```
int[][] tablaDentada = {new int[5], new int[5]);
```

Finalmente, si sólo queremos declarar una variable tabla dentada pero no queremos indicar su número de el

```
int[][] tablaDentada = {new int[5], new int[5]};
```

Finalmente, si sólo queremos declarar una variable tabla dentada pero no queremos indicar su número de elementos, (luego la variable valdría null), entonces basta poner:

```
int[][] tablaDentada;
```

Hay que precisar que aunque en los ejemplos hasta ahora presentes se han escrito ejemplos basados en tablas dentadas de sólo dos niveles de anidación, también es posible crear tablas dentadas de cualquier número de niveles de anidación. Por ejemplo, para una tabla de tablas de tablas de enteros de 2 elementos en la que el primero fuese una tabla dentada formada por dos tablas de 5 enteros y el segundo elemento fuese una tabla dentada formada por una tabla de 4 enteros y otra de 3 se podría definir así:

```
int[][][] tablaDentada = new int[][][] { new int[][] {new int[5],
new int[5]}, new int[][] {new int[4], new int[3]}};
```

A la hora de acceder a los elementos de una tabla dentada lo único que hay que hacer es indicar entre corchetes cuál es el elemento exacto de las tablas componentes al que se desea acceder, indicándose un elemento de cada nivel de anidación entre unos corchetes diferentes pero colocándose todas las parejas de corchetes juntas y ordenadas de la tabla más externa a la más interna.

Por ejemplo, para asignar el valor 10 al elemento cuarto de la tabla que es elemento primero de la tabla que es elemento segundo de la tabla dentada declarada en último lugar se haría:

```
tablaDentada[1][0][3] = 10;
```

La clase System.Array

En realidad, todas las tablas que definamos, sea cual sea el tipo de elementos que contengan, son objetos que derivan de System.Array .

Es decir, van a disponer de todos los miembros que se han definido para esta clase, entre los que son destacables:

- Length : Campo de sólo lectura que informa del número total de elementos que contiene la tabla.

Si la tabla tiene más de una dimensión o nivel de anidación indica el número de elementos de todas sus dimensiones y niveles.

Por ejemplo:

```
int[] tabla = {1,2,3,4};
int[][] tabla2 = {new int[] {1,2}, new int[] {3,4,5}};
int[,] tabla3 = {{1,2},{3,4,5,6}};
Console.WriteLine(tabla.Length); //Imprime 4
Console.WriteLine(tabla2.Length); //Imprime 5
Console.WriteLine(tabla3.Length); //Imprime 6
```

- Rank : Campo de sólo lectura que almacena el número de dimensiones de la tabla.

Obviamente si la tabla no es multidimensional valdrá 1.

Por ejemplo:

```
int[] tabla = {1,2,3,4};
int[][] tabla2 = {new int[] {1,2}, new int[] {3,4,5}};
int[,] tabla3 = {{1,2},{3,4,5,6}};
Console.WriteLine(tabla.Rank); //Imprime 1
Console.WriteLine(tabla2.Rank); //Imprime 1
Console.WriteLine(tabla3.Rank); //Imprime 2
```

- GetLength(int dimensión) : Método que devuelve el número de elementos de la dimensión especificada.

Las dimensiones se indican empezando a contar desde cero, por lo que si quiere obtenerse el número de elementos de la primera dimensión habrá que usar GetLength(0) , si se quiere obtener los de la segunda habrá que usar GetLength(1) , etc.

Por ejemplo:

```
int[,] tabla = {{1,2}, {3,4,5,6}};
Console.WriteLine(tabla.GetLength(0)); // Imprime 2
Console.WriteLine(<gtabla.GetLength(1)); // Imprime 4
```

- CopyTo(Array destino, int posición) : Copia todos los elementos de la tabla sobre la que es aplica en la que se le pasa como primer parámetro a partir de la posición de la misma indicada como segundo parámetro.

Por ejemplo:

```
int[] tabla1 = {1,2,3,4};

int[] tabla2 = {5,6,7,8};

tabla1.CopyTo(tabla2,0);  //  A  partir  de  ahora,  ambas  tablas
contienen {1,2,3,4}
```

Ambas tablas han de ser unidimensionales. Por otro lado, y como es obvio, la tabla de destino ha de ser de un tipo que pueda almacenar los objetos de la tabla fuente, el índice especificado ha de ser válido (mayor o igual que cero y menor que el tamaño de la tabla de destino) y no ha de valer null ninguna. Si no fuese así, saltarían excepciones de diversos tipos informando del error cometido (en la documentación del SDK puede ver cuáles son en concreto)

Aparte de los miembros aquí señalados, de System.Array cuenta con muchos más que permiten realizar tareas tan frecuentes como búsquedas de elementos, ordenaciones, etc.

Cadenas de texto

Una cadena de texto no es más que una secuencia de caracteres Unicode. En C# se representan mediante objetos del tipo tipo de dato llamado string , que no es más que un alias del tipo System.String incluido en la BCL.

Las cadenas de texto suelen crearse a partir literales de cadena o de otras cadenas previamente creadas. Ejemplos de ambos casos se muestran a continuación:

```
string cadena1 = "John Wayne";
string cadena2 = cadena;
```

En el primer caso se ha creado un objeto string que representa a la cadena formada por la secuencia de caracteres John Wayne indicada literalmente (nótese que las comillas dobles entre las que se encierran los literales de cadena no forman parte del contenido de la cadena que representan sino que sólo se usan como delimitadores de la misma).

En el segundo caso la variable cadena2 creada se genera a partir de la variable cadena1 ya existente, por lo que ambas variables apuntarán al mismo objeto en memoria.

Hay que tener en cuenta que el tipo string es un tipo referencia, por lo que en principio la comparación entre objetos de este tipo debería comparar sus direcciones de memoria como pasa con cualquier tipo referencia. Sin embargo, si ejecutamos el siguiente código veremos
que esto no ocurre en el caso de las cadenas:

```
using System;
public class IgualdadCadenas
{
      public static void Main()
      {
            string cadena1 = "John Wayne";
            string cadena2 = String.Copy(cadena1);
            Console.WriteLine(cadena1==cadena2);
      }
}
```

El método Copy() de la clase String usado devuelve una copia del objeto que se le pasa como parámetro. Por tanto, al ser objetos diferentes se almacenarán en posiciones distintas de memoria y al compararlos debería devolverse false como pasa con cualquier tipo referencia.
Sin embargo, si ejecuta el programa verá que lo que se obtiene es precisamente lo contrario: true . Esto se debe a que para hacer para hacer más intuitivo el trabajo con cadenas, en C# se ha modificado el

operador de igualdad para que cuando se aplique entre cadenas se considere que sus operandos son iguales sólo si son lexicográficamente equivalentes y no si referencian al mismo objeto en memoria.

Además, esta comparación se hace teniendo en cuenta la capitalización usada, por lo que "Hola"=="HOLA" ó "Hola"=="hola" devolverán false ya que contienen las mismas letras pero con distinta capitalización.

Si se quisiese comparar cadenas por referencia habría que optar por una de estas dos opciones:

1. compararlas con Object.ReferenceEquals()
2. convertirlas en objects y luego compararlas con == Por ejemplo:

```
Console.WriteLine(Object.RefereceEquals(cadena1, cadena2));
Console.WriteLine( (object) cadena1 == (object) cadena2);
```

Ahora sí que lo que se comparan son las direcciones de los objetos que representan a las cadenas en memoria, por lo que la salida que se mostrará por pantalla es:

```
False
False
```

Hay que señalar una cosa, y es que aunque en principio el siguiente código debería mostrar la misma salida por pantalla que el anterior ya que las cadenas comparadas se deberían corresponder a objetos que aunque sean lexicográficamente equivalentes se almacenan en posiciones diferentes en memoria:

```
using System;
public class IgualdadCadenas2
{
      public static void Main()
      {
            string cadena1 = "Matt Dammon";
            string cadena2 = "Matt Dammon";
            Console.WriteLine(Object.ReferenceEquals(cadena1,
            cadena2));
            Console.WriteLine( ((object) cadena1) == ((object)
            cadena2));
      }
}
```

Si lo ejecutamos veremos que la salida obtenida es justamente la contraria:

```
True
True
```

Esto se debe a que el compilador ha detectado que ambos literales de cadena son lexicográficamente equivalentes y ha decidido que para ahorra memoria lo mejor es almacenar en memoria una única copia de la cadena que representan y hacer que ambas variables apunten a esa copia común.

Al igual que el significado del operador == ha sido especialmente modificado para trabajar con cadenas, lo mismo ocurre con el operador binario + . En este caso, cuando se aplica entre dos cadenas o una cadena y un carácter lo que hace es devolver una nueva cadena con el resultado de concatenar sus operandos.

Así por ejemplo, en el siguiente código las dos variables creadas almacenarán la cadena Hola Mundo :

```
public class Concatenación
{
      public static void Main()
      {
            string cadena = "Hola" + " Mundo";
            string cadena2 = "Hola Mund" + 'o';
      }
}
```

Por otro lado, el acceso a las cadenas se hace de manera similar a como si de tablas de caracteres se tratase: su "campo" Length almacenará el número de caracteres que la forman y para acceder a sus elementos se utiliza el operador [] . Por ejemplo, el siguiente código muestra por pantalla cada carácter de la cadena Hola en una línea diferente:

```
using System;
public class AccesoCadenas
{
      public static void Main()
      {
            string cadena = "Hola";
            Console.WriteLine(cadena[0]);
            Console.WriteLine(cadena[1]);
            Console.WriteLine(cadena[2]);
            Console.WriteLine(cadena[3]);
      }
}
```

Sin embargo, hay que señalar una diferencia importante respecto a la forma en que se accede a las tablas: **las cadenas son inmutables**, lo que significa que no es posible modificar los caracteres que las forman. Esto se debe a que el compilador comparte en memoria las referencias a literales de cadena lexicográficamente equivalentes para así ahorrar memoria, y si se permitiese modificarlos los cambios que se hiciesen a través de una variable a una cadena compartida afectarían al resto de variables que la compartan, lo que podría causar errores difíciles de detectar.

Por tanto, hacer esto es incorrecto:

```
string cadena = "Hola";
cadena[0]="A"; //Error: No se pueden modificar las cadenas
```

Sin embargo, el hecho de que no se puedan modificar las cadenas no significa que no se puedan cambiar los objetos almacenados en las variables de tipo string .Por ejemplo, el siguiente código es válido:

```
String cadena = "Hola";
cadena = "Adios";


// Correcto, pues no se modifica la cadena almacenada en cadena
// sino que se hace que cadena pase a almacenar otro string
// distinto
```

Si se desea trabajar con cadenas modificables puede usarse Sytem.Text.StringBuilder , que funciona de manera similar a string pero permite la modificación de sus cadenas en tanto que estas no se comparten en memoria. Para crear objetos de este tipo basta pasar como parámetro de su constructor el objeto string que contiene la cadena a representar mediante un StringBuilder , y para convertir un StringBuilder en String siempre puede usarse su método ToString() heredado de System.Object . Por ejemplo:

```
using System.Text;
using System;
public class ModificaciónCadenas
{
      public static void Main()
      {
            StringBuilder cadena = new StringBuilder("Pelas");
            String cadenaInmutable;
            cadena[0] = 'V';
            Console.WriteLine(cadena); // Muestra Velas
            cadenaInmutable = cadena.ToString();
            Console.WriteLine(cadenaInmutable); // Muestra Velas
      }
}
```

Aparte de los métodos ya vistos, en la clase System.String se definen muchos otros métodos aplicables a cualquier cadena y que permiten manipularla. Los principales son:

int IndexOf(string subcadena)

Indica cuál es el índice de la primera aparición de la subcadena indicada dentro de la cadena sobre la que se aplica. La búsqueda de dicha subcadena se realiza desde el principio de la cadena, pero es posible indicar en un segundo parámetro opcional de tipo i nt cuál es el índice de la misma a partir del que se desea empezar a buscar. Del mismo modo, la búsqueda acaba al llegar al final de la cadena sobre la que se busca, pero pasando un tercer parámetro opcional de tipo int es posible indicar algún índice anterior donde terminarla.

Nótese que es un método muy útil para saber si una cadena contiene o no alguna subcadena determinada, pues sólo si no la encuentra devuelve un –1 .

int LastIndexOf(string subcadena) :

Funciona de forma similar a IndexOf() sólo que devuelve la posición de la última aparición de la subcadena buscanda en lugar de devolver la de la primera.

string Insert(int posición, string subcadena) :

Devuelve la cadena resultante de insertar la subcadena indicada en la posición especificada de la cadena sobre la que se aplica.

string Remove(int posición, int número) :

Devuelve la cadena resultante de eliminar el número de caracteres indicado que hubiese en la cadena sobre al que se aplica a partir de la posición especificada.

string Replace(string aSustituir, string sustituta) :

Devuelve la cadena resultante de sustituir en la cadena sobre la que se aplica toda aparación de la cadena a sustituir indicada por la cadena sustituta especificada como segundo parámetro.

string Substring(int posición, int número) :

Devuelve la subcadena de la cadena sobre la que se aplica que comienza en la posición indicada y tiene el número de caracteres especificados. Si no se indica dicho número se devuelve la subcadena que va desde la posición indicada hasta el final de la cadena.

string ToUpper() y string ToLower() :

Devuelven, respectivamente, la cadena que resulte de convertir a mayúsculas o minúsculas la cadena sobre la que se aplican.

Es preciso incidir en que aunque hayan métodos de inserción, reemplazo o eliminación de caracteres que puedan dar la sensación de que es posible modificar el contenido de una cadena, en realidad las cadenas son inmutables y dicho métodos lo que hacen es devolver una nueva cadena con el contenido correspondiente a haber efectuado las operaciones de modifiación solicitadas sobre la cadena a la que se aplican.

Por ello, las cadenas sobre las que se aplican quedan intactas como muestra el siguiente ejemplo:

```
Using System;
public class EjemploInmutabilidad
{
      public static void Main()
      {
            string cadena1=”Hola”;
            string cadena2=cadena1.Remove(0,1);
            Console.WriteLine(cadena1);
            Console.WriteLine(cadena2);
      }
}
```

La salida por pantalla de este ejemplo demuestra lo antes dicho, pues es:

Hola
ola

Como se ve, tras el Remove() la cadena1 permance intacta y el contenido de cadena2 es el que debería tener cadena1 si se le hubiese eliminado su primer carácter.

Constantes

Una constante es una variable cuyo valor puede determinar el compilador durante la compilación y puede aplicar optimizaciones derivadas de ello. Para que esto sea posible se ha de cumplir que el valor de una constante no pueda cambiar durante la ejecución, por lo que el compilador informará con un error de todo intento de modificar el valor inicial de una constante. Las constantes se definen como variables normales pero precediendo el nombre de su tipo del modificador const y dándoles siempre un valor inicial al declararlas. O
sea, con esta sintaxis:

```
const <tipoConstante> <nombreConstante> = <valor>;
```

Así, ejemplos de definición de constantes es el siguiente:

```
const int a = 123;
const int b = a + 125;
```

Dadas estas definiciones de constantes, lo que hará el compilador será sustituir en el código generado todas las referencias a las constantes a y b por los valores 123 y 248 respectivamente, por lo que el código generado será más eficiente ya que no incluirá el acceso y cálculo de los valores de a y b . Nótese que puede hacer esto porque en el código se indica explícitamente cual es el valor que siempre tendrá a y, al ser este un valor fijo, puede deducir cuál será el valor que siempre tendrá b . Para que el compilador pueda hacer estos cálculos se ha de cumplir que el valor que se asigne a las constantes en su declaración sea una expresión constante. Por ejemplo, el siguiente código no es válido en tanto que el valor de x no es constante:

```
int x = 123;
// x es una variable normal, no una constante

const int y = x +123;
// Error: x no tiene porqué tener valor constante (aunque aquí lo
// tenga)
```

Debido a la necesidad de que el valor dado a una constante sea precisamente constante, no tiene mucho sentido crear constantes de tipos de datos no básicos, pues a no ser que valgan null sus valores no se pueden determinar durante la compilación sino únicamente tras la ejecución de su constructor. La única excepción a esta regla son los tipos enumerados, cuyos valores se pueden determinar al compilar como se explicará cuando los veamos el tema Enumeraciones

Todas las constantes son implícitamente estáticas, por lo se considera erróneo incluir el modificador static en su definición al no tener sentido hacerlo.

BIBLIOGRAFÍA RECOMENDADA:

https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/types/index

Head First C#, 3rd Edition