UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

INTRODUCCION A LA PROGRAMACION C#

2 - 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

UNIDAD: 2

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 el manejo de estado y comportamiento.

Los campos, las propiedades, los métodos y los eventos de una clase se denominan de forma colectiva **miembros** de clase.

Crear objetos

Una clase y un objeto son cosas diferentes, es fundamental entender esta diferencia.

Una clase define un tipo de objeto. 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 sería 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 valor 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.

Escriba el siguiente código

```
class MyClass
{
  int x, z;
  int y;

 public static void Main()
 {
    int j;
    MyClass myClass =new MyClass();
    System.Console.WriteLine($"j vale {j}");
    System.Console.WriteLine($"campo x vale {myClass.x}");

 }
}
```

Compile y verifique la salida de la compilación. Que sucede ?

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 pa labra 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. Definamos primero la estructura de la clase Persona de la siguiente manera

```
class Persona
{
    public string Nombre {get; set;}
    public int Edad {get; set;}
    public string ID {get; set;}
}
```

Por tanto, una forma de asignar un valor a la variable p del ejemplo anterior sería así:.

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

Se puede instanciar e inicializar un objeto con la sintaxis de inicializadores de C#:

```
            Persona p = new Persona{
                Nombre ="Pepe",
                Edad = 25,
                ID= "12345-A"
            };
```

Este tema lo desarrollaremos mas adelante cuando veamos Objetos en profundidad.

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 el CoreCLR acá, 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.

Por valor:

- Las variables de tipos de valor contienen directamente los datos.
- 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*).

Por referencia:

- Las variables de los tipos de referencia almacenan referencias a los datos, lo que se conoce como objetos.
- 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.

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.

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

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 Main()
    {
        // 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*

```
namespace A
{
    class Clase
    {

    }
}

namespace B
{
    class Clase
    {

    }
}
```

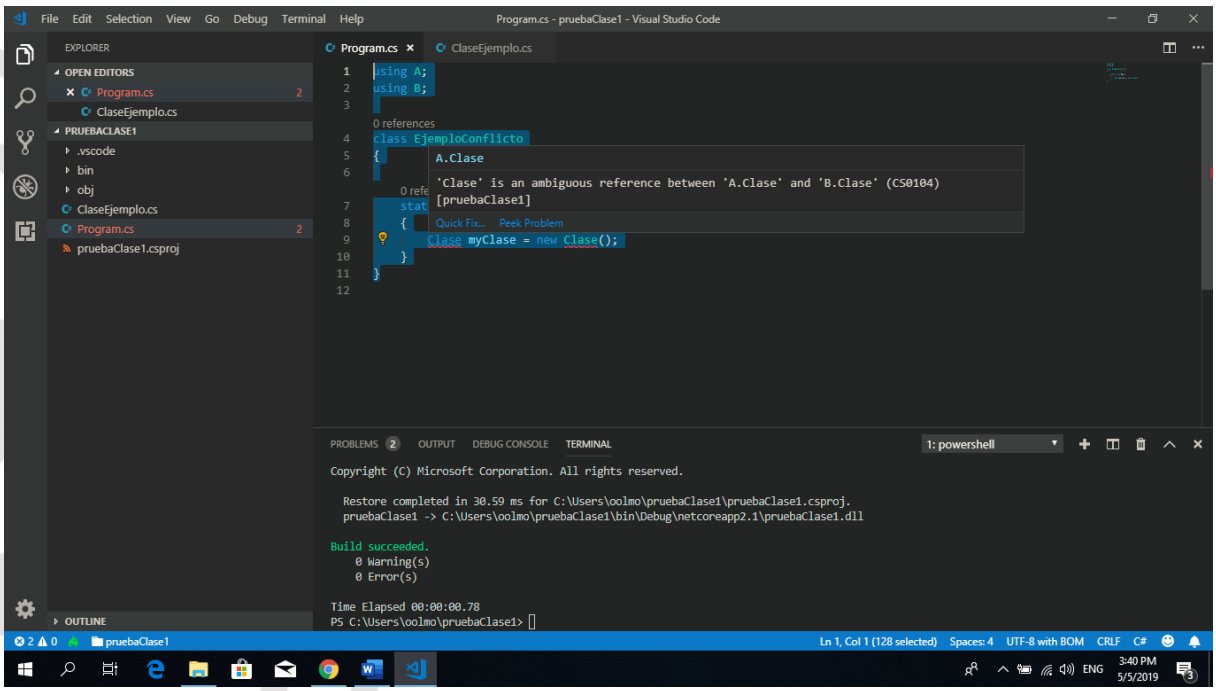
Y hacemos:

```
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()
    {
        ClaseA 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

UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

BIBLIOGRAFÍA RECOMENDADA:

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

Head First C# 3rd Edition Jennifer Greene Andrew Stellman

Fecha	Versión	Observaciones
09/07/2018	1.0	Versión Original
11/07/2018	1.1	Se corrige typo en pagina 23