

Sintaxis de C# .NET

Código básico

Sensible a mayúsculas / minúsculas

Las instrucciones terminan con ;

Los bloques de instrucciones se encierran entre llaves { }

Comentarios:

// comenta la línea

/* ... */ Comenta todo lo contenido entre la apertura y el cierre (multilínea)

Bloque de contenidos plegable:

#region Descripción

#endregion

Tipos de datos predefinidos

C# .NET	.NET Framework	Bytes	Intervalo de valores
bool	System.Boolean	1	true / false (1 / 0)
byte	System.Byte	1	0 a 255
sbyte	System.Sbyte	1	-128 a 127
short	System.Int16	2	-32.768 a 32.767
ushort	System.UInt16	2	0 a 65.535
int	System.Int32	4	-2.147.483.648 a 2.147.483.647 (con signo)
uint	System.UInt32	4	0 a 4.294.967.295 (sin signo)
long	System.Int64	8	-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807 (9,2...E+18) (con signo)
ulong	System.UInt64	8	0 a 18.446.744.073.709.551.615 (1,8...E+19) (sin signo)
float	System.Single	4	-3,4028235E+38 a -1,401298E-45 para valores negativos 1,401298E-45 a 3,4028235E+38 para valores positivos
double	System.Double	8	-1,79769313486231570E+308 a -4,94065645841246544E-324 para los valores negativos 4,94065645841246544E-324 a 1,79769313486231570E+308 para los valores positivos
decimal	System.Decimal	16	0 a +/-79.228.162.514.264.337.593.543.950.335 (+/-7,9...E+28) sin separador decimal; 0 a +/-7,9228162514264337593543950335 con 28 posiciones a la derecha del decimal el número distinto de cero más pequeño es +/-0,00000000000000000000000000000001 (+/-1E-28)
	System.Datetime	8	0:00:00 (medianoche) del 1 de enero de 0001 a 11:59:59 p.m. del 31 de diciembre de 9999
char	System.Char	2	0 a 65.535 (sin signo)
string	System.String	Variable	0 a 2.000 millones de caracteres Unicode aprox.
object	System.Object	Variable	Cualquier tipo puede almacenarse en una variable de tipo object. (4/8 bytes de puntero en plataformas de 32/64 bits + contenido)

Modificadores de accesibilidad

public, internal, private, protected, protected internal

Nota: Los *protected* sólo son válidos para herencia en POO.

Declaración e inicialización de constantes

[*accesibilidad*] const *tipo nombre_constante* = *valor*;

Nota: Sólo pueden ser *public*, *internal* o (por defecto) *private*, aunque privadas no sean útiles.
Se ven a nivel de clase, no de instancia.

Sintaxis de C# .NET

Declaración e inicialización de variables

[*accesibilidad*] [static] [readonly] *tipo* *variable* [= {*valor_inicial* | new *tipo*()}];

Nota: *readonly* sólo permite asignar valor en la declaración y en el constructor de la clase que la contiene.

Algunos tipos, al asignar valor se pueden poner sufijos para especificar el tipo.

Sufijo	Tipo	Sufijo	Tipo
U	unsigned int	F	Float
L	long	D	Double
UL	unsigned long	M	Decimal

Caracteres de escape en cadenas.

Carácter	Nombre	Carácter	Nombre	Carácter	Nombre
\'	Comilla simple	\n	Salto de línea	\0	null
\"	Dobles comillas	\t	Tabulador	\a	Sonido alerta
\\	Backslash	\b	Retroceso		

Para escribir la cadena tal cual queremos que sea, basta con antecederla de una arroba (excepto si hay dobles comillas)
p.e.: string ejemplo = @"C:\Mi directorio" es donde guardo todo"

Arrays

[*accesibilidad*] [static] *tipo*[] *variable* [= new *tipo*[*número_de_elementos*]];

[*accesibilidad*] [static] *tipo*[] *variable* = {*lista_de_valores*};

Nota: Siempre Base 0

Conversiones

Convert. *ToTipo*()

tipo.Parse() y *tipo*.TryParse()

variable.ToString()

(*tipo*) *variable*

variable as *tipo*

Operadores

Unarios: ++, -- (pre y postfijo)

Matemáticos: +, -, *, /, %

Cadena +

Relacionales ==, !=, <, >, <=, >=

Lógicos !, &, |, &&, ||

Asignación =, +=, -=, *=, /=, %=

Ternario (?:)

Tipo is, typeof

Enumeraciones

[*accesibilidad*] enum *enumeración*[: *tipo*] {

elem1 [= *valor*],

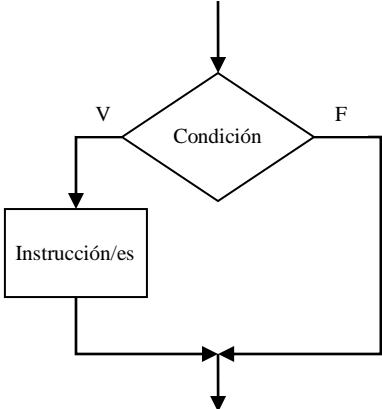
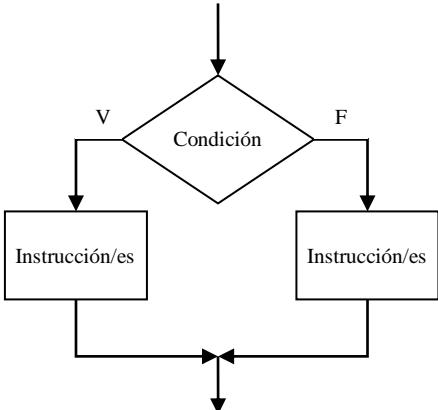
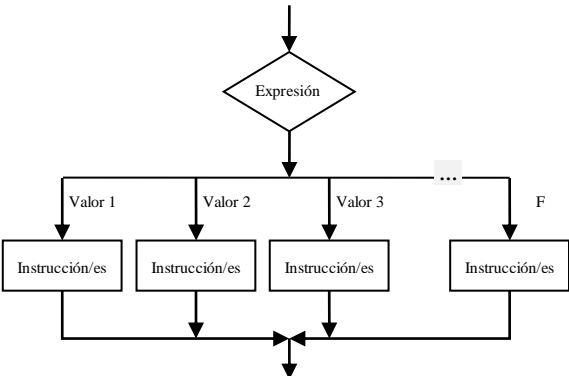
elem2 [= *valor*],

...

}

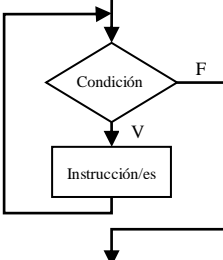
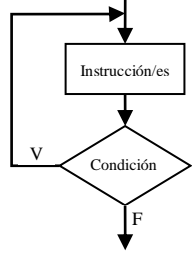
Sintaxis de C# .NET

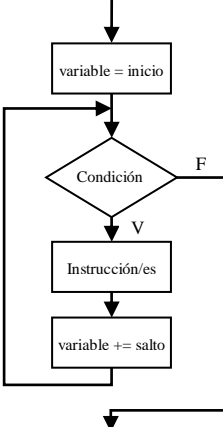
Estructuras condicionales

	<pre>if (condición) { instrucción/es; }</pre>
	<pre>if (condición) { instrucción/es; } else { instrucción/es; }</pre>
	<pre>switch (expresión) { case valor1: instrucción/es; {break goto case valorX throw [[new] excepción] return [valor]} ; [case valor2: case valor3 : instrucción/es; break;] [default: instrucción/es; break;] }</pre>

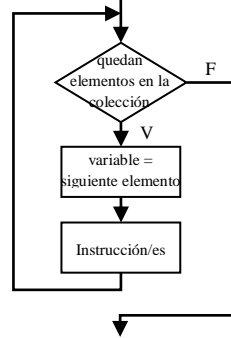
Sintaxis de C# .NET

Bucles

	<pre>while (condición) { instrucción/es; }</pre>		<pre>do { instrucción/es; } while (condición)</pre>
---	--	--	---

	<pre>for ([[tipo] variable = valor_inicia]; [condición]; [incremento_variable_de_control]) { [instrucción/es]; }</pre>
--	--

Recorrido colecciones

	<pre>foreach (tipo variable in colección) { instrucción/es; }</pre>
---	---

Nota: Puede usarse *break* o *continue* para romper la secuencia normal de cualquiera de ellos.

Instrucciones de control

```
using ([tipo] variable [= new tipo()]) {instrucción/es}
```

Nota: Sólo válido para tipos que implementen el interfaz *IDisposable*.

Aunque se pueden declarar las variables fuera, no es recomendable para evitar accesos que den error. Si se declara fuera sólo lleva la variable. Si se declara se puede instanciar en la misma instrucción.

```
goto etiqueta1;
```

...

```
etiqueta1:
```

Nota: Permite salir desestructuradamente de cualquier conjunto de instrucciones (bucles incluidos) aún con varios niveles de anidamiento, pero siempre dentro de una misma función.

Sintaxis de C# .NET

Funciones

```
[accesibilidad] [static] {tipo | void} nombre_función ([lista_argumentos]) {  
    instrucción/es;  
    [return [valor];]  
}
```

Nota: Las funciones de tipo *void* no devuelven nada.

Listas de argumentos

```
[[ref|out]] tipo nombre_argumento [= valor_por_defecto] [, ...]
```

Nota: Los argumentos *ref* tienen que ser inicializados antes de ser pasados.
Los argumentos *out* tienen que ser asignados antes de ser devueltos.
El argumento puede ser un array.
Si se le asigna un valor por defecto, pasa a ser opcional y detrás todos deben ser opcionales.

Paso de un array de parámetros

```
params tipo[ ] nombre_array
```

Nota: Será el último, y tiene que pasarse por valor (ni *ref* ni *out*).
Si no se pasa, el sistema lo interpreta como un array vacío.

Paso de parámetros por nombre

Para llamar a una función y especificar los parámetros desordenados, debemos pasales el nombre delante. Normalmente se usa con parámetros opcionales.

Función (*nombre_parámetro: valor, ...*)

Tratamiento de excepciones

```
try {  
    instrucciones_a_ejecutar;  
}  
catch [(tipo_excepción [variable])] {  
    instrucción/es;  
}  
...  
[finally {  
    instrucción/es;  
}]
```

Lanzamiento de excepciones

```
throw [{instancia_de_excepción | new excepción_tipificada([lista_de_argumentos])}]
```

Nota: *throw* a secas dentro de un bloque *catch* relanza la misma excepción recogida, equivalente a *throw variable*

Sintaxis de C# .NET

Programación Orientada a Objetos

Espacios de Nombres

```
namespace espacio_de_nombres {  
    Clases, espacios_de_nombres  
}
```

Atajos: using [*alias* =]*espacio_de_nombres*;

Estructuras

```
[accesibilidad] struct nombre_estructura {  
    Miembros, propiedades, métodos, eventos, enumeraciones, constructores...  
}
```

Nota: No permiten herencia (heredan de *System.ValueType*, que hereda de *System.Object*), pero sí interfaces.
La accesibilidad no puede ser *protected*, ya que una estructura no permite herencia.
Una estructura comparte el mismo formato que una clase, pero con alguna limitación:
Si se crean constructores, debe tener parámetros, ya que el sistema crea uno sin parámetros.
No pueden implementar destructores.

Definición de clases

```
[accesibilidad] [partial] class nombre_de_clase {  
    Miembros, propiedades, métodos, eventos, enumeraciones, constructores...  
}
```

Nota: Las clases sólo pueden ser *public* o (por defecto) *internal*, salvo si van dentro de otra clase.

Instanciación

```
[accesibilidad] tipo variable = new tipo([lista_de_argumentos]);
```

Constantes, Variables y Miembros

Constantes y Variables: como siempre.

Miembros: Son variables normales pero accesibles desde el exterior (*public* o *internal*)

Propiedades

Formato simple

```
[accesibilidad] tipo nombre_de_propiedad { get; set; }
```

Formato normal

```
[accesibilidad] tipo nombre_de_propiedad {  
    [[accesibilidad] get { instrucción/es; }]  
    [[accesibilidad] set { instrucción/es; }]  
}
```

Nota: Si omitimos bloque *get* o *set* pasan a ser de sólo lectura o sólo escritura.
En el bloque *get* debemos usar un *return* para devolver el dato.
En el bloque *set* el parámetro de entrada se llama obligatoriamente *value*.
Si establecemos accesibilidad, deberá ser para “cerrar” el acceso de unos de los dos, nunca para tratar de dar mayor accesibilidad ni la misma de la propiedad.

Sintaxis de C# .NET

Indizador

```
[accesibilidad] tipo this[tipo_índice índice] {  
    [[accesibilidad] get { instrucción/es; }]  
    [[accesibilidad] set { instrucción/es; }]  
}
```

Nota: Funcionan como propiedades indizadas. Dan acceso a algún array interno que guarde la información.

Métodos

Son funciones normales pero accesibles desde el exterior (*public* o *internal*)

Nota: La sobrecarga de propiedades y métodos en la misma clase se realiza copiando la declaración pero con diferente firma

Eventos

Debemos definir la delegada y luego el evento del tipo de la delegada.

```
[accesibilidad] delegate void DelegadaManejadorEvento ([lista_de_argumentos]);  
[accesibilidad] event DelegadaManejadorEvento nombre_evento;
```

Nota: Aunque no es obligatorio, los argumentos suelen ser un *sender* de tipo *object* y un *e* que herede de *EventArgs*.

Para lanzar el evento (en la clase que lo lanza):

```
if (nombre_evento != null)  
    nombre_evento ([lista_de_argumentos]);
```

Para capturar el evento (en la clase llamante)

```
instancia.nombre_evento += new DelegadaManejadorEvento (método);
```

Constructores

Metodo que se utiliza para inicializar la clase.

```
[accesibilidad] [static] nombre_clase([lista_args]) [:this([lista_args])]  
{ instrucción/es; } Se pueden sobrecargar pero no se pueden sobreescribir.
```

Nota: La accesibilidad de constructor puede *protected*, ya que aunque los constructores no se heredan si lo declaramos *protected* podemos acceder a él con *:base*, y con *private* no podríamos.

Un constructor *static* es único, sin parámetros, se invoca automáticamente la primera vez que se accede la clase y no lleva accesibilidad.

Si hay sobrecarga, con *:this* se puede especificar otro constructor para que se ejecute ANTES (con herencia *:base*)

Destruyores

```
public void dispose()  
{ instrucción/es; }
```

Nota: Este método proviene normalmente de implementar el interface *IDisposable*. Se puede usar conjuntamente con `private void Dispose(bool disposing) {...}` y con `GC.SuppressFinalize(this);`

```
~nombre_clase()  
{ instrucción/es; }
```

Nota: Sólo es recomendable implementarlo si hay objetos de código no administrado (*unmanaged*), por el consumo de recursos.

El compilador lo convierte automáticamente en una sobrecarga del método interno *Finalize*

Miembros, Propiedades y métodos compartidos

Se definen igual, salvo por que deben llevar la palabra clave *static*.

Nota: Los convierte en miembros, propiedades y métodos de clase (*Shared* en VB). No hay equivalente al *static* de VB.

Sintaxis de C# .NET

Herencia

Clases

Definición de clases base

```
[accesibilidad] [{sealed | abstract}] [partial] class nombre_de_clase
{
    Abstract.- No se puede instanciar. Hay que heredar
    Sealed.- No se puede heredar.
    Miembros, propiedades, métodos, eventos, enumeraciones, constructores...
}
```

Nota: Una clase por defecto siempre se puede heredar Si hay un método o propiedad abstracta nos obliga a declarar la clase como abstracta.

Implementación de la Herencia de clases en las clases derivadas

```
[accesibilidad] [{sealed | abstract}] [partial] class nombre_de_clase : clase_base
{
    Una clase solo puede heredar de una clase.
    Pero puede tener múltiples interfaces.
    Miembros, propiedades, métodos, eventos, enumeraciones, constructores...
}
```

Nota: Se debe especificar *abstract* si la clase no implementa todo el código.

Acceso a jerarquía de clases en herencia

`this / base`

En un método `this` apunta a la instancia
En la definición hace referencia a la clase.
Base: hace referencia a la clase de la que hereda.

Nota: En C# no hay equivalencia a *MyClass* de VB.

Propiedades

```
[accesibilidad] [{virtual | abstract}] tipo nombre_de_propiedad
{
    Virtual: Se puede si queremos sobrescribir.
    Abstract: Se debe sobrescribir.

    [get { instrucción/es; }]
    [set { instrucción/es; }]
}
```

Nota: Por defecto, NO se pueden sobrescribir.
Caso de ser declarada como *abstract* NO LLEVA nada más que la cabecera de declaración con *get*; y/o *set*;

Sobreescritura (vía herencia)

```
[accesibilidad] [{sealed} override | new] tipo nombre_de_propiedad
{
    Sealed: No se puede sobrescribir si se heredase.
    Override: Forma correcta de sobrescribirlo.
    New: Forma rara que cuando no esta declarado como virtual se puede sobrescribir.

    [get { instrucción/es; }]
    [set { instrucción/es; }]
}
```

Nota: La accesibilidad de la propiedad y de sus métodos *get* y *set* debe ser la misma en ambos niveles. Para cambiarla deberemos usar *new*.

Con *override* podemos sobrescribir sólo uno de los dos métodos, *get* o *set*, dejando el otro sin tocar. Además *sealed* cierra la posibilidad de sobreescritura.

La palabra clave *new* permite escribir la propiedad desde cero, ocultando la de la clase superior. Evita una advertencia del compilador.

Permite cambiar la visibilidad porque la creamos de nuevo.
Podemos crearlos con *new* para cambiar la visibilidad y que no sean visibles ciertos métodos que no queramos.

Sintaxis de C# .NET

Métodos

```
[accesibilidad] [static] [{virtual | abstract}] {tipo | void} nombre_función ([argumentos])  
{  
    instrucción/es;  
    [return [valor]]; }  

```

Nota: Por defecto, NO se pueden sobrescribir.

Caso de ser declarada como abstract NO LLEVAN nada más que la cabecera de declaración.

Sobreescritura (vía herencia)

```
[accesibilidad] [static] [{sealed | override | new} {tipo | void} nombre_función ([argumentos])  
{  
    instrucción/es;  
    [return [valor]]; }  

```

Nota: La accesibilidad del método debe ser la misma en ambos niveles. Para cambiarla deberemos usar *new*.
sealed cierra la posibilidad de sobreescritura.

La palabra clave *new* sólo evita que el compilador de una advertencia, pero NO oculta los demás métodos como hace *Shadows* en VB

Interfaces

```
[accesibilidad] interface nombre_interface [: nombre_interface_base ]  
  
{  
  
    Cabeceras de propiedades, métodos y eventos  
  
}
```

Nota: Los contenidos NO llevan modificador de accesibilidad ni pueden ser declarados *virtual*.

Implementación (se puede hacer automáticamente con el editor)

```
[accesibilidad] class nombre_clase: interface1 [, interface2...] {  
  
    Implementación de las propiedades, métodos y eventos.  
  
}
```

Nota: Si implementamos varios interfaces que comparten nombres de métodos o propiedades, hay que implementarlos “explícitamente” si queremos diferenciar los métodos o propiedades. Para ello, cada propiedad o método debe llevar delante del nombre, el nombre del Interface y un punto. Se puede hacer directamente desde el editor.

Sintaxis de C# .NET

Colecciones y Diccionarios

Colecciones: Conjunto heterogéneo no limitado.

Propiedades y métodos comunes en las colecciones (a algunas de ellas):

Count, Add(), Clear(), Contains(), IndexOf(), Insert(), Remove(), RemoveAt()

Tipos

Todas las colecciones normales estan dentro de `Sistem.Collections` otras en subespacio `Specialized`.

ArrayList: Colección no ordenada de objetos. Permite acceso secuencial y por índice.

StringCollection: Colección no ordenada de cadenas. Permite acceso secuencial y por índice.

Queue: Colección FIFO de objetos. Métodos especiales (*enqueue, dequeue, peek*)

Stack: Colección LIFO de objetos. Métodos especiales (*push, pop, peek*)

Diccionarios: Conjuntos de pares clave/valor no limitado y heterogéneos (aunque hay excepciones)

Propiedades y métodos comunes en los diccionarios(a muchos de ellos):

Count, Keys, Values, Add(), Clear(), Contains(), Remove()

Tipos

Que son colecciones

Muchos datos. Pocos 10. No se sabe, segun el número la crea de un tipo o de otro.

Hashtable, ListDictionary, HybridDictionary: Diccionarios organizados por el código Hash de la clave. Permite acceso secuencial (no se garantiza el orden) y por clave. No admite claves duplicadas (que generen el mismo hashcode).

OrderedDictionary: Como las anteriores, pero añade métodos de Lista. Mantiene los datos ordenados en el orden de entrada del usuario, pero internamente el sistema la gestiona como una *HashTable*. Permite acceso secuencial (el orden es el generado por la clave) así como por posición y por clave.

StringDictionary: ^{HashCode}HashTable con tipos string para clave y valor.

NameValueCollection: Similar a *StringDictionary*, pero admite valores duplicados de clave.

Devuelve varios valores si tienen la misma clave.

SortedList: Colección de pares Clave/Valor ordenados por clave. Permite acceso secuencial, así como por clave y por posición con los métodos especiales *GetKey(i)* y *GetByIndex(i)*

Particularidades

Ordenación:

Caso de tener que ordenar (manual o automáticamente) elementos no comparables, la clase debe implementar *IComparable*. Si no podemos tocar la clase, o queremos poder cambiar la forma de ordenar, deberemos crear clases que implementen *IComparer* o *IComparer<T>* para pasarlas como parámetro para la ordenación y así poder definir en cada momento la manera de ordenar.

Duplicados y búsquedas:

En todas ellas, cuando se trabaja con posibles duplicados, hay que sopesar posibilidad de sobreescribir los métodos *Equals()* y *GetHashCode()* de la clase o de crear clases comparadoras que implementen *IEqualityComparer* o *IEqualityComparer<T>* para que las instancias sean consideradas iguales en función de sus contenidos, ya que por defecto la comparación se realiza sobre punteros, no sobre contenidos. Esto es necesario también en caso de métodos que buscan elementos en colecciones.

c1.Equals(c2); Debemos sobreescribir el método para que no compare direcciones de punteros sobreescribo el método y comparo Id y Descripción por ejemplo.

Sintaxis de C# .NET

Colecciones y Diccionarios Genéricos: Conjunto homogéneo no limitado.

Una vez definidos los tipos, éstos son invariables.

Nota: Los tipos usados para definir las Genéricas no tienen por qué ser clases. También pueden ser interfaces.

Comparación Colecciones y Diccionarios:

Colecciones originales	Equivalente genérico
ArrayList	List< T >
StringCollection	List< String >
Queue	Queue< T >
Stack	Stack< T >
Hashtable, ListDictionary, HybridDictionary, OrderedDictionary	Dictionary< K, V >
StringDictionary, NameValueCollection	Dictionary< String, String >
SortedList	SortedList< K, V >

Contenidos normales	Contenidos genéricos
DictionaryEntry	KeyValuePair< K, V >

Tipos nuevos

LinkedList< T >: Lista enlazada doble. Contenidos del tipo especificado, en elementos de tipo *LinkedListNode<T>*.

Permite sólo acceso secuencial mediante un *foreach* o enumerador.

Propiedades y Métodos especiales para añadir elementos y para localizar nodos:

Count, First, Last, AddFirst(), AddLast(), AddAfter(), AddBefore(), Remove(), RemoveFirst(), RemoveLast(), Find(), FindLast()

LinkedListNode<T>: Nodos del *LinkedList<T>*

Propiedades para acceso: *Previous, Next, Value*

HashSet<T>: Similar al *List<T>*, pero no admite duplicados. No da error, pero no los añade y NO se puede acceder por índice, sólo secuencialmente con *foreach* o enumerador..

Para pequeñas cantidades de elementos es mejor usar *List<T>*, pero para grandes cantidades éste está más optimizado.

Propiedades y métodos para obtener información e interactuar con los contenidos:

Count, Comparer, Add() Contains(), Remove(), ExceptWith(), IntersectWith(), UnionWith() y algunos más.

SortedSet<T>: Similar al *HashSet<T>* pero ordenada, o bien por el método implementado dentro con *IComparer* o por un comparador pasado al constructor. A las propiedades y métodos del *HashSet<T>* añade las propiedades *Min* y *Max* que devuelven el primer y último elemento de la colección.

Sintaxis de C# .NET

Genéricos de usuario

Clases con tipos parametrizados. Una vez definida los tipos son invariables.

Se definen en la declaración de la clase como “variables”, que se utilizarán dentro de la clase para definir los tipos de los contenidos.

```
[accesibilidad] [partial] class nombre_de_clase <T1, T2, ...>
{
    Miembros, propiedades, métodos, eventos, enumeraciones, constructores... p.e.:
    public T1 Dato1 { get; set; }
    public T2 Dato2 { get; set; }
    public nombre_de_clase (T1 dato1Inicial, T2 dato2Inicial)
    {
        Dato1 = dato1Inicial;
        Dato2 = dato2Inicial;
    }
    ...
}
```

Normalmente las clases genéricas creadas por el usuario se utilizan para crear colecciones, pudiendo usarse los tipos parametrizados para determinar el tipo de los contenidos de la colección y pudiendo así exponer los métodos que nos interese.

Si implementamos *ICollection* o *IDictionary* entonces podremos tener las propiedades y métodos de colecciones y diccionarios, añadiendo así los nuestros.

Sintaxis de C# .NET

Delegados

Son punteros a funciones.

Su declaración es similar a la de un método de un Interface, con lo que no llevan implementación interna. Simplemente definen la firma de la función que se pasará posteriormente al ser instanciadas.

[*accesibilidad*] delegate {*tipo* | void} *nombre_función* ([*argumentos*]);

Nota: Si la delegada o la función a asignar están en otro espacio de nombres (o capa) simplemente añadiremos el espacio de nombres o el *using* correspondiente.

Uso:

Se puede crear una variable e instanciarla, dándole como parámetro el nombre de la función que queremos que se ejecute, llamando luego a la delegada para su ejecución.

Se le puede pasar a dicha variable una expresión Lambda (*ver más adelante*) para su ejecución de la misma manera que la anterior. En ese caso no se instancia, sino que la asignación del código crea implícitamente la instancia. Además, los parámetros que espera recibir la delegada se ponen con un nombre para poder identificarlos en el código posterior al símbolo =>

p.e.:

Declaración

```
delegate float Operar(float ope1, float ope2); //Delegada
float Suma(float x, float y) { return x + y; } //Método a pasar
float Resta(float x, float y) { return x - y; } //Método a pasar
```

Asignación

```
Operar delOperacion = null;
delOperacion = new Operar(Suma); //Método a llamar
delOperacion = new Operar(Resta); //Método a llamar
```

o

```
delOperacion = ((o1, o2) => o1 + o2); //Expresión Lambda
delOperacion = ((o1, o2) => o1 - o2); //Expresión Lambda
```

Llamada

```
Resultado: " + delOperacion(x, y);
```

Sintaxis de C# .NET

Novedades Visual Studio 2008 (Framework 3.5)

~~int x;
x = null;~~

Tipos Nullables

Permite asignar valor null a una variable y provee de propiedades HasValue y Value para su manejo.

[*accesibilidad*] **Nullable**<*tipo*> *variable*;

[*accesibilidad*] **Tipo?** *variable*;

Nota: Sólo aplicable a tipos por valor

OK

```
Nullable<int> i;  
i = null;  
i = 7;
```

Inicializaciones

Junto con la instanciación podemos asignarle valores.

[*accesibilidad*] *tipo variable* = new *tipo*([*argumentos*])

{ *propiedad* = *valor*, *propiedad* = *valor*, ...};

Nota: Si el tipo tiene constructores parametrizados, se pueden usar y asignar el resto con la inicialización.

Inferencia de tipos

Si no se sabe de qué tipo va a ser el contenido de una variable. Define el tipo en la compilación.

var *variable* = *valor_inicial*;

```
foreach ( var x in _____ )  
{  
    x._____  
}
```

Nota: La inferencia de tipos NO funciona a nivel de clase, sólo en variables locales en funciones, etc. Solo a nivel de método.

Los parámetros tampoco.

Tipos anónimos

Es una mezcla entre Inicializaciones e Inferencia:

var *variable* = new { *variable* = *valor*, *variable* = *valor*, ...};

Clases anónimas que instanciamos con los campos que indiquemos.

Los tipos anónimos se generan en tiempo de compilación, como tipos normales, salvo que no tienen nombre y no podemos acceder a ellos para declaraciones, instanciación ni tipos de parámetros, sino que el sistema los usa cuando los necesita.

Métodos extensores

Deben ser métodos estáticos en clases estáticas, y el primer parámetro siempre es el del tipo a extender, precedido por *this*

static class *ClaseExtensiones* {

public **static** *tipo nombre_función*(**this** *tipo_a_extender variable*[, *argumentos*]) {...}

Sintaxis de C# .NET

Expresiones Lambda

Se trata de una forma de pasar código por parámetro, en vez de una delegada. Consta de tres partes:

- Lista de parámetros, que sólo lleva nombres de variables que usaremos dentro de la expresión Lambda para acceder a cada uno de los parámetros de entrada (puede llevar el tipo si el compilador no es capaz de inferirlo).
- El operador de la expresión que separa ambas partes, formado por la unión de los caracteres “=>”
- Código en sí a ejecutar. Toda variable usada debe ser pasada por parámetro o declarada dentro de la propia expresión Lambda. Podemos usar dentro de la expresión variables de instancia y controles si es necesario, siempre que estemos seguros de que se tendrá acceso a ellas donde se ejecute.

Tipos de Expresiones Lambda:

- De expresión

- ✓ Usadas normalmente en métodos de extensión de LINQ (Where, Take...)
- ✓ No necesitan delegada.
- ✓ Pueden no llevar parámetros de entrada.
- ✓ A la derecha del operador Lambda hay una expresión que al ser evaluada devuelve un valor, o una llamada a un método.

(parametro/s) => expresión

Nota: Los paréntesis de los parámetros son opcionales cuando sólo hay un parámetro.

- De instrucciones

- ✓ Usadas para pasar código directamente en vez de crear un método para pasarlo al instanciar una delegada.
- ✓ Necesitan delegada. Sin embargo no hay que instanciar la delegada. La asignación de la expresión Lambda a la variable de la delegada se encarga de instanciarla.
- ✓ Pueden no llevar parámetros de entrada.
- ✓ A la derecha del operador Lambda hay un conjunto de instrucciones encerradas entre llaves.

(parametros) => { instrucción/es; };

Nota: Los paréntesis de los parámetros son opcionales cuando sólo hay un parámetro.