UNIDAD 6

Desarrollo de interfaces basados en XML (.NET WPF)

REQUISITOS PREVIOS

Para poder seguir este tema debéis instalar Visual Studio. Este producto, creado por Microsoft es un completo entorno de desarrollo que permite hacer aplicaciones para múltiples plataformas. Es un producto de pago pero que tiene una versión gratuita (Visual Studio Express), que cumple con mucho las necesidades que tenemos para este curso.

Nosotros vamos a utilizar la versión Visual Studio Express 2012. La razón por la que vamos a usar la versión 2012, en lugar de otras versiones más actuales, es que la instalación de este entorno es más ligera y cumple perfectamente con los objetivos del curso.

Microsoft Visual Studio Express:

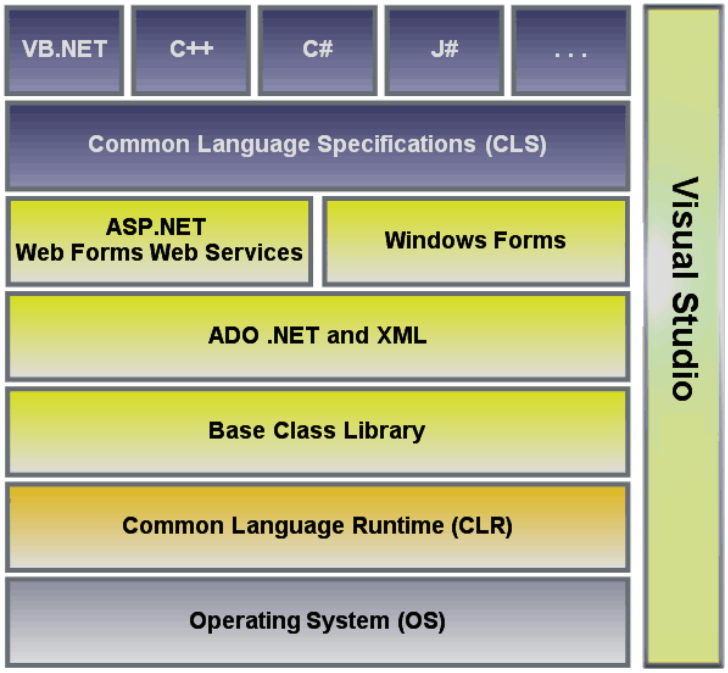
Microsoft Visual Studio Express Edition es un programa de desarrollo en entorno de desarrollo integrado para sistemas operativos Windows desarrollado y distribuido por Microsoft Corporation. Soporta varios lenguajes de programación tales como **Visual C++, Visual C#, Visual J#, ASP.NET y Visual Basic .NET**, aunque actualmente se han desarrollado las extensiones necesarias para muchos otros. Es de carácter gratuito y es proporcionado por la compañía Microsoft Corporation orientándose a principiantes, **estudiantes y aficionados de la programación web y de aplicaciones, ofreciéndose dicha aplicación a partir de la versión 2005 de Microsoft Visual Studio**.
El enlace para descargar Visual Studio Express 2012 es el siguiente:

http://download.microsoft.com/download/A/3/A/A3AE11C4-7B3D-4E1A-9A42-1DA7F122FC0E/VS2012_WDX_ESN.iso

Al ser la versión Express de Visual Studio podemos utilizarla legalmente con fines educativos. La clave para usar el producto es la siguiente: **MMVJ9-FKY74-W449Y-RB79G-8GJGJ**

PLATAFORMA .NET

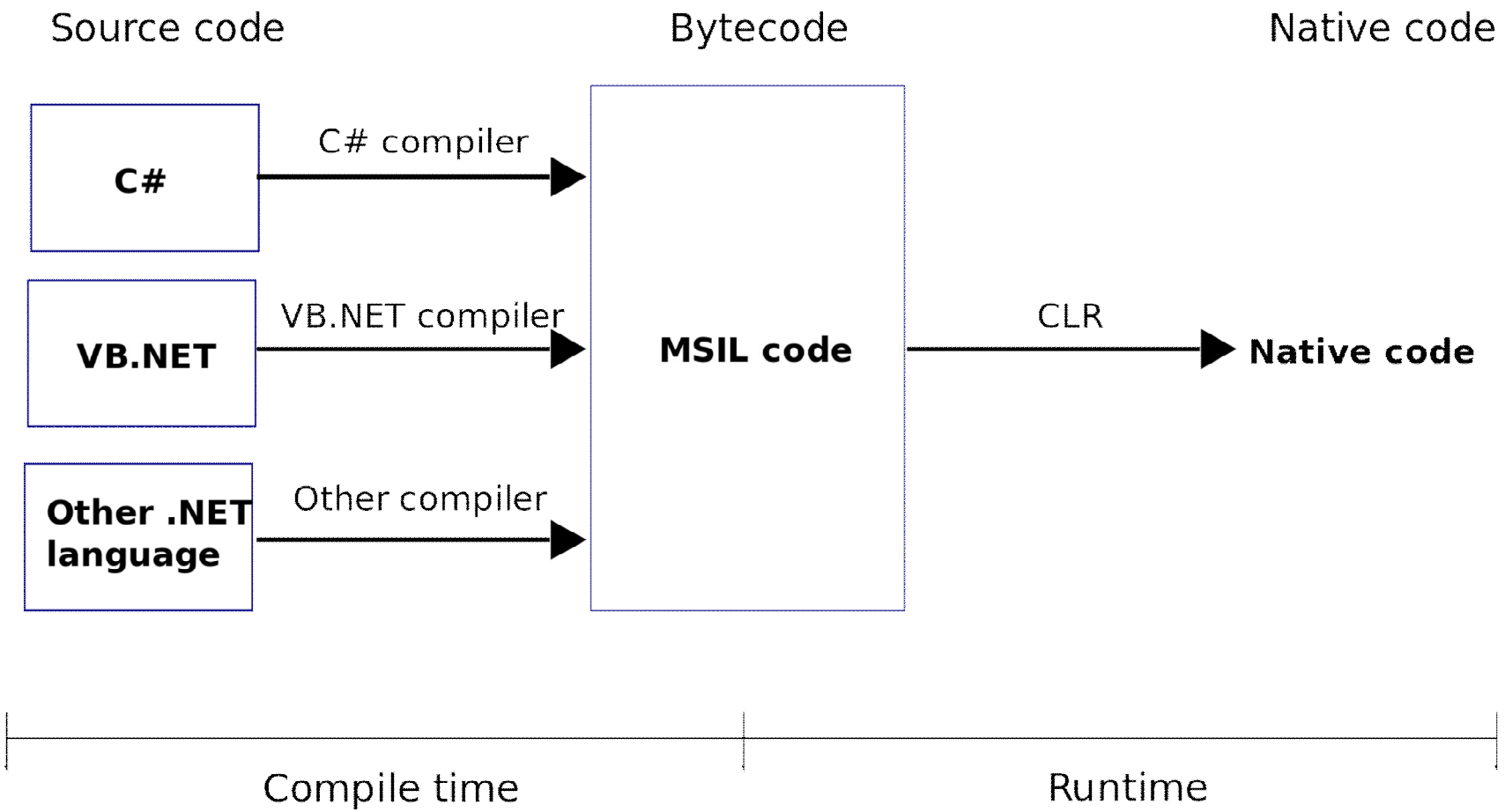
La **plataforma de desarrollo .NET** es un entorno gestionado de ejecución de aplicaciones, lenguajes de programación y compiladores, y permite el desarrollo de todo tipo de funcionalidades: desde programas de consola o servicios Windows, hasta aplicaciones para dispositivos móviles pasando por desarrollos de escritorio o para Internet. La plataforma .NET está formada por una serie de componentes que, en conjunto, permiten la creación de todo tipo de aplicaciones en todo tipo de sistemas operativos y utilizando todo tipo de lenguajes de programación. La introducción de la **plataforma .NET** de Microsoft. .NET es una plataforma de ejecución intermedia multilenguaje, de forma que los programas desarrollados en .NET **no se compilan en lenguaje máquina**, sino en un lenguaje intermedio. El código no se convierte a lenguaje máquina hasta que se ejecuta, de manera que el **código puede ser independiente de la plataforma**. El diagrama siguiente muestra los bloques conceptuales en los que se divide la plataforma .NET:



El **CLR o Common Language Runtime** es la parte de .NET encargada de ejecutar las aplicaciones desarrolladas para la plataforma. Trabaja encima del sistema operativo para aislar a la plataforma de éste. Esto le permite ejecutar aplicaciones .NET multiplataforma.

La plataforma .NET no está atada a un determinado lenguaje de programación cualquier componente creado con uno de estos lenguajes puede ser utilizado de forma transparente desde cualquier otro lenguaje. **CLS (especificación del lenguaje común)** está formada por un conjunto de reglas que deben ser seguidas por las definiciones de tipos de datos. Así, dichos datos pueden interactuar desde una aplicación escrita en un lenguaje determinado con otra aplicación escrita en otro lenguaje diferente.

La **BCL** está formada por bibliotecas o APIs especializadas que pueden ser utilizadas por todos los lenguajes de programación de la plataforma .NET.



Los desarrolladores que usan CLR escriben el código fuente en un lenguaje compatible con .NET, como C# o Visual Basic .NET. En tiempo de compilación, un compilador .NET convierte el código a CIL (Common Intermediate Language). En tiempo de ejecución, el compilador del CLR (Common Language Runtime) convierte el código CIL en código nativo para el sistema operativo.

C# (Visual C#)

El lenguaje C# es un lenguaje **estructurado y orientado a objetos**, **igual que Java**. Al igual que Java, implementa todas las características de un lenguaje orientado a objetos como son las Clases, la herencia, el polimorfismo, encapsulación, sobrecarga, etc.

Diferencias entre C# y Java:

En C# tenemos una facultad que tiene el lenguaje para **ahorrarnos tener que declarar los setters y getters**:

```
public class Persona {
   public String Nombre {get; set;}
}
```

En **C# ya no tenemos paquetes**, tenemos **agrupaciones**. Estas agrupaciones pueden ser de dos tipos:

- **Agrupación lógica**: son los llamados namespace.
- **Agrupación física**: serán los proyectos (estarán en el mismo dll o exe).

Un **destructor** es un método que los programadores tenemos para liberar un objeto de la memoria. Pero tenemos unas diferencias entre los distintos lenguajes:

C#: Se asegura su ejecución, pero **no de un modo determinista** (no sabemos exactamente cuándo se ejecutará). Al menos sabemos que se ejecutará cuando termine el programa.

Java (finalize): No se asegura su ejecución (no determinista). Incluso aunque termines el programa, Oracle no nos asegura que los objetos instanciados se liberen…

C++: Se asegura su ejecución, pero de un **modo determinista**. Tenemos que ser nosotros cuando decirle que acabe.

Por tanto, en C# podemos asegurarnos que nuestro destructor se ejecutará cuando terminemos la ejecución de nuestro programa, no cuando le apetezca al recolector de basura.

A diferencia de Java, en **C# los miembros públicos de una clase deben empezar con mayúscula y los privados por minúscula anteponiendo un guion bajo**.

El famoso **System.out.print de Java**, en C# tenemos una instrucción similar, **Console.WriteLine** o **Console.Write**.

```
public class Persona{
   public string Nombre {get; set;}
   private int _edad;

   public String ImprimirNombre(){
        Console.WriteLine(Nombre);
     }
}
```

C# te permite indicar en los métodos o constructores los parámetros que queramos poner por defecto, es decir, al realizar una llamada a dicho método no tengamos que indicar un valor para un parámetro que sea opcional. Por ejemplo, pensemos en un método que nos permite crear personas, pero que si no le indicamos nada nos creará 100 personas.

```
public static Persona[] CreaPersonas (int n = 100)
{
   string[] names = { "María", "Juan", "Pepe", "Luis", "Carlos", "Miguel", "Cristina" };

   Persona[] listing = new Persona[n];
   Random random = new Random ();
   for (int i = 0; i < n; i++)
      listing [i] = new Persona {
         Nombre= names [random.Next (0, names.Length)],
      };
   return listing;
}
```

Espacios de nombres

Los espacios de nombres se usan mucho en programación de C# de dos maneras. En primer lugar, .NET usa espacios de nombres **para organizar sus clases** de la siguiente manera:

```
System.Console.WriteLine("Hello World!");
```

System es un espacio de nombres y **Console** es una clase de ese espacio de nombres. La palabra clave **using** se puede usar para que no se necesite el nombre completo, como en el ejemplo siguiente:

```
using System;
Console.WriteLine("Hello");
Console.WriteLine("World!");
```

En segundo lugar, declarar nuestros propios espacios de nombres puede ayudarnos a controlar el ámbito de nombres de clase y método en proyectos de programación grandes. Use la palabra clave **namespace** para declarar un espacio de nombres, como en el ejemplo siguiente:

```
namespace SampleNamespace
{
   class SampleClass
   {
      public void SampleMethod()
      {
         System.Console.WriteLine(
           "SampleMethod inside SampleNamespace");
      }
   }
}
```

Los espacios de nombres tienen las propiedades siguientes:

- Organizan proyectos de código de gran tamaño.
- Se delimitan mediante el operador ‘.’.
- La directiva **using** obvia la necesidad de especificar el nombre del espacio de nombres para cada clase.
- El espacio de nombres global es el espacio de nombres "raíz": global::System siempre hará referencia al espacio de nombres System de .NET.

Un programa en C# contiene uno o más espacios de nombres, que quedan definidos por el programador o como parte de una biblioteca de clases previamente escrita.

Por ejemplo, el espacio de nombres System incluye la clase Console, una clase que contiene los métodos para leer y escribir en la ventana de la consola. El espacio de nombres System también contiene múltiples espacios de nombres diferentes, como System.IO y System.Collections.

Sólo .NET Framework tiene más de ochenta espacios de nombres, cada uno con miles de clases: **los espacios de nombres se utilizan para minimizar la confusión que se podría producir entre tipos y métodos con nombres parecidos**.

Si escribimos una clase fuera de una declaración de espacio de nombres, el compilador proporcionará un espacio de nombres predeterminado para esa clase.

El siguiente ejemplo define dos espacios de nombres, cada uno con una clase denominada FileHandling. Al especificar el espacio de nombres, se hace posible diferenciar rápidamente entre las clases y los métodos que contienen.

```
namespace StatisticalData
{
   class FileHandling
   {
      public void Load() {}  // code to load statistical data
   }
}

namespace Images
{
   class FileHandling
   {
      public void Load() {}  // code to load an image file
   }
}

class Program
{
   static void Main()
   {
      StatisticalData.FileHandling data = new StatisticalData.FileHandling();
      data.Load();

      Images.FileHandling image = new Images.FileHandling();
      image.Load();
   }
}
```

Directivas using y espacios de nombres

Cuando se crea una aplicación de consola con Visual C# Express, las primeras líneas en el editor de código contienen directivas using que muestran varios espacios de nombres de .NET Framework.

Cuando crea un programa en Visual C# Express, se crea automáticamente un espacio de nombres.

Para utilizar las clases de otros espacios de nombres en nuestro programa, debemos especificarlos con una directiva **using**. Los espacios de nombres más comúnmente utilizados en .NET Framework se muestran de forma predeterminada cuando crea una nueva aplicación.

Si utilizamos clases de otros espacios de nombres en la biblioteca de clases, debemos agregar una directiva using para ese espacio de nombres al archivo de código fuente.

Cuando el Editor de código detecta que se ha declarado una clase o estructura que no puede encontrar en los espacios de nombres enumerados en las directivas using actuales, sugerirá espacios de nombres que contienen la clase o estructura.

Tipos de datos integrados

byte	0 .. 255
sbyte	-128 .. 127
short	-32,768 .. 32,767
ushort	0 .. 65,535
int	-2,147,483,648 .. 2,147,483,647
uint	0 .. 4,294,967,295
long	-9,223,372,036,854,775,808 .. 9,223,372,036,854,775,807
ulong	0 .. 18,446,744,073,709,551,615
float	-3.402823e38 .. 3.402823e38
double	-1.79769313486232e308 .. 1.79769313486232e308
decimal	-79228162514264337593543950335 .. 79228162514264337593543950335
char	Un carácter Unicode.
string	Una cadena de caracteres Unicode.
bool	True o False.
object	Un objeto.

Los tipos de datos integrados se utilizan dentro de un programa C# de diferentes maneras:

- Como variables:

```
int answer = 42;
string greeting = "Hello, World!";
```

- Como constantes:

```
const int speedLimit = 55;
const double pi = 3.14159265358979323846264338327950;
```

- Como valores devueltos y parámetros:

```
long CalculateSum(int a, int b)
{
   long result = a + b;
   return result;
}
```

Convertir tipos de datos

Los tipos de datos se pueden convertir **implícitamente**, en cuyo caso **el compilador hace automáticamente** la conversión, o **explícitamente** utilizando un operador de conversión explícita, donde **el programador fuerza la conversión** y asume el riesgo de perder información.

```
int i = 0;
double d = 0;

i = 10;
d = i;       // conversión implícita

d = 3.5;
i = (int) d;  // conversión explicita
```

Tipos de referencia y de valor

A diferencia de algunos lenguajes de programación con los que puede estar familiarizado, C# tiene dos variedades de tipos de datos: **valor** y **referencia**.

Cuando se declara una variable utilizando uno de los tipos de datos integrados básicos o una estructura definida por el usuario, éste es un tipo de valor. Una excepción es el tipo de datos **string**, que es un tipo de referencia.
Un tipo de valor almacena su contenido en la memoria asignada en la pila. Por ejemplo, en este caso el valor 42 se almacena en un área de memoria denominada pila.

Profundicemos un poco más. Cuando los datos se pasan a los métodos como parámetros de tipo de valor, se crea una copia de cada parámetro en la pila. Evidentemente, si el parámetro en cuestión es un tipo de datos grande, por ejemplo, una estructura definida por el usuario con muchos elementos, o el método se ejecuta muchas veces, puede afectar el rendimiento.

En estas situaciones, es preferible pasar una referencia al tipo, utilizando la palabra clave ref. Éste es el equivalente de C# de la técnica de C++ de pasar un puntero a una variable en una función. Al igual que con la versión de C++, el método tiene la capacidad de cambiar el contenido de la variable.

```
int AddTen(int number)  // parameter is passed by value
{
   return number + 10;
}

void AddTen(ref int number)  // parameter is passed by reference
{
   number += 10;
```

Matrices y colecciones

El almacenamiento de grupos de elementos de datos relacionados es un requisito básico de casi todas las aplicaciones de software; las dos formas principales de almacenarlos son **matrices** y **colecciones**.

Matrices

Las matrices son **colecciones de objetos del mismo tipo**. Dado que las matrices pueden tener prácticamente cualquier longitud, pueden utilizarse para almacenar miles o incluso millones de objetos, pero el tamaño debe decidirse al crear la matriz.

Se tiene **acceso** a cada elemento de la matriz **mediante un índice**, que es simplemente un número que indica la posición o ranura donde el objeto está almacenado. El primer elemento de la matriz es el elemento 0.

Matrices unidimensionales

Una matriz es una colección indizada de objetos. Una matriz unidimensional de objetos se declara así:

```
type[] arrayName;
```

A menudo, los elementos de la matriz se inicializan al mismo tiempo, de la siguiente manera:

```
int[] array = new int[5];
```

El valor predeterminado de los elementos numéricos de la matriz es cero y los elementos de referencia cambian de forma predeterminada a null, pero los valores se pueden inicializar durante la creación de la matriz de la manera siguiente:

```
int[] array1 = new int[] { 1, 3, 5, 7, 9 };
```

O también: `int[] array2 = {1, 3, 5, 7, 9};`

```
String[] days = {"Sun", "Mon", "Tue", "Wed", "Thr", "Fri", "Sat"};
System.Console.WriteLine(days[0]);  // Outputs "Sun"
```

Matrices multidimensionales

Conceptualmente, una matriz multidimensional con dos dimensiones se asemeja a una cuadrícula. Una matriz multidimensional con tres dimensiones se asemeja a un cubo.

```
//  Declaración Matriz 2 dimensiones
int[,] array2D = new int[2,3];


// Declaración e inicialización Matriz 2 dimensiones
int[,] array2D2 = { {1, 2, 3}, {4, 5, 6} };

// Recorrido de una matriz de dos dimensiones
for (int i=0; i<2; i++)
{
   for (int j=0; j<3; j++)
   {
      System.Console.Write(array2D[i,j]);
   }
   System.Console.WriteLine();
}
```

Colecciones

Una matriz es simplemente una de las muchas opciones para almacenar conjuntos de datos utilizando C#. La opción que seleccionemos depende de varios factores, tales como la forma en que pensamos manipular o tener acceso a los elementos. Por ejemplo, **utilizar una lista suele ser más rápido que utilizar una matriz si debe insertar elementos al principio o en medio de la colección**.

Otros tipos de clases de colección incluyen **mapa, árbol y pila**; cada uno tiene sus propias ventajas. Para obtener más información, mirar **System.Collections** y **System.Collections.Generic**.

En el ejemplo siguiente se muestra cómo usar la clase **List<T>**. Observa que a diferencia de la clase Array, los elementos se pueden insertar en medio de la lista. Este ejemplo restringe los elementos de la lista a fin de que sean cadenas.

```
public class TestCollections
{
   public static void TestList()
   {
      System.Collections.Generic.List<string> sandwich = new System.Collections.Generic.List<string>();

      sandwich.Add("bacon");
      sandwich.Add("tomato");

      sandwich.Insert(1, "lettuce");

      foreach (string ingredient in sandwich)
      {
         System.Console.WriteLine(ingredient);
      }
   }
}
```

Colecciones - List, ArrayList

Las colecciones de clases de C# son un conjunto de clases diseñadas específicamente para agrupar objetos y llevar a cabo tareas con ellos.

Tanto la **list‹T›** como la **ArrayList** y otras clases de la Collections tienen propiedades muy similares a los arrays de C#. Una ventaja fundamental de estas clases sobre los arrays es que puedan crecer y reducir el número de objetos almacenados.

La clase list‹T› esta contenida en **System.Collections.Generic**, mientras que la clase ArrayList figura en el **System.Collections**.

La sintaxis para crear una colección list‹T› es la siguiente: **List‹tipo› nombre = new List‹tipo›();**

ArrayList es un objeto creado de una manera similar, aunque sin el argumento de tipo:

```
ArrayList nombre = new ArrayList ();
```

Con esta sintaxis ahora podemos crear una list‹T› denominada listacolores:

```
using System;
using System.Collections.Generic;

public class Lists
{
   static void Main() {
        List‹string› listacolores = new List‹string›();
   }
}
```

Añadir artículos a las listas

Una vez que una lista se ha creado hay una serie de métodos que pueden ser llamados a realizar tareas en la lista.

En este caso para agregar elementos a la lista de objetos se utiliza el método Add(). Ejemplo:

```
List‹string› ListaColores = new List‹string›();

ListaColores.Add ("Azul");
ListaColores.Add ("Rojo");
ListaColores.Add ("Verde");
ListaColores.Add ("Amarillo");
ListaColores.Add ("Morado");
```

Acceso a los elementos

A los elementos individuales en una lista se puede acceder mediante el índice (teniendo en cuenta que el primer punto del índice es 0, el segundo índice 1 y así sucesivamente). El valor del índice se encuentra entre corchetes tras el nombre de la lista. Por ejemplo, para acceder al segundo punto del objeto ListaColores:

```
Console.WriteLine (ListaColores[1]);
```

Un elemento de la lista puede cambiar su valor de manera similar usando el índice combinado con el operador de asignación. Por ejemplo, para cambiar el color de verde a mamey:

```
ListaColores[2] = "mamey";
```

Todos los elementos de una lista se puede acceder mediante un bucle foreach. Por ejemplo:

```
foreach (string color in ListaColores){
    Console.WriteLine ( color );
}
```

Inserción de artículos en una lista

Anteriormente se utilizó el método Add() para añadir elementos a una lista. El método Add(), sin embargo, sólo añade elementos al final de una lista. A veces es necesario añadir un nuevo tema en una ubicación específica en una lista. **Insert()** es el método para este fin específico. Insert() recibe dos argumentos, un entero que indica el índice de localización de la inserción y el tema que se incluirá en ese lugar. Por ejemplo, para insertar un elemento en la posición 2 de la lista seria:

```
ListaColores.Insert(2, "Blanco");
```

Ordenando Listas

No hay manera de decirle a C# que ordene automáticamente una lista de artículos después de añadir. Si los elementos de una lista deben estar siempre ordenados se debe llamar al método **Sort** una vez se añaden nuevos elementos:

```
ListaColores.Sort();
```

Búsqueda de elementos en una lista

Se proveen un buen número de métodos con las clases listas y la ArrayList. El método más básico es el **Contains()**, en el cual cuando se pide a un objeto ArrayList o List devuelve true si el elemento se encuentra en la lista, o false si no se encuentra.

El **IndexOf()** devuelve el valor de un índice de un elemento de la lista. Por ejemplo, el código siguiente muestra el valor de salida de 2, que es el índice de la posición de "Amarillo" en la cadena:

```
List‹string› ListaColores = new List‹string›();

ListaColores.Add ("Red");
ListaColores.Add ("Green");
ListaColores.Add ("Amarillo");
ListaColores.Add ("Morado");
ListaColores.Add ("Mamey");
Console.WriteLine(ListaColores.IndexOf("Amarillo"));
```

Si el elemento no se encuentra en la lista devuelve -1.

Esta técnica podría utilizarse para reemplazar un valor con otro. Por ejemplo, sin conocer de antemano el valor del índice del "Amarillo" cadena podemos cambiar a "Negro":

```
ListaColores[ListaColores.IndexOf("Amarillo")] = "Negro";
```

El **LastIndexOf()** método devuelve el valor del índice del último elemento de la lista para que coincida con el punto especificado. Esto es particularmente útil cuando la lista contiene elementos duplicados.

Obtener información de una lista

Hay dos miembros de la clase que son útiles para obtener información acerca de una lista de C # o colección de objetos ArrayList. La propiedad **Capacity** puede ser utilizada para identificar el número de artículos que puede almacenar una colección sin tener que cambiar el tamaño.

El propiedad **Count**, por otra parte, identifica cuantos artículos se encuentran actualmente almacenados en la lista. Por razones obvias, Capacity sera siempre superior a Count.

Borrado de elementos

Todos los elementos de una lista podrán ser borrados mediante el método **Clear()**:

```
ListaColores.Clear();
```

Clear() elimina los elementos de la lista y establece la propiedad Count a cero. La propiedad Capacity, sin embargo, sigue siendo la misma.

Estructuras de control

Las estructuras de control se pueden dividir en dos tipos:

- Las **condicionales** (if/else y swich)
- Las repetitivas (**de iteración**) (do/while, for, while, foreach).

En C# podemos encontrar todas estas estructuras de control con la misma funcionalidad que en Java.

if-else

Una instrucción if identifica qué instrucción se debe ejecutar dependiendo del valor de una expresión booleana.

```
// estructura if-else
if (condition)
{
   Instrucciones;
}
else
{
   Instrucciones;
}

// estructura if sin else
if (condition)
{
 Instrucciones;
}
// Next statement in the program.
```

Dentro del If o Else puede constar de una única instrucción o de varias instrucciones entre llaves ({}). Para una única instrucción, las llaves son opcionales. Estas instrucciones puede ser otra instrucción if anidada dentro de la instrucción if.

switch

Switch es una instrucción de selección que elige una sola sección switch para ejecutarla desde una lista de candidatos en función de una coincidencia de patrones con la expresión de coincidencia.

```
switch (caseSwitch)
{
   case 1:
      Console.WriteLine("Case 1");
      break;
   case 2:
      Console.WriteLine("Case 2");
      break;
   default:
      Console.WriteLine("Default case");
      break;
}
```

do

La instrucción do ejecuta una instrucción o un bloque de instrucciones mientras que una expresión booleana especificada se evalúa como true. Como esa expresión se evalúa después de cada ejecución del bucle, un bucle do-while **se ejecuta una o varias veces**. Esto es diferente del bucle while, que se ejecuta cero o varias veces.

```
int n = 0;
do
{
   Console.WriteLine(n);
   n++;
} while (n < 5);
```

Si la expresión se evalúa como true, la ejecución continúa en la primera instrucción del bucle. En caso contrario, la ejecución continúa en la primera instrucción después del bucle.

En cualquier punto del bloque de instrucciones do, se puede salir del bucle mediante la instrucción **break** o podemos ir directamente a la evaluación de la expresión while mediante la instrucción **continue**.

for

La instrucción for ejecuta una instrucción o un bloque de instrucciones mientras una expresión booleana especificada se evalúa como true.

En cualquier punto del bloque de instrucciones for, se puede salir del bucle mediante la instrucción **break**, o bien se puede ir a la siguiente iteración del bucle mediante la instrucción **continue**.

```
for (inicializador; condición; iterador)
body
```

Las instrucciones de la sección **inicializador se ejecutan solo una vez**, antes de entrar en el bucle.

La sección **condición**, si está presente, debe ser una **expresión booleana**. Dicha expresión se evalúa antes de cada iteración del bucle. Si la sección condición no está presente o la expresión booleana se evalúa como true, se ejecutará la siguiente iteración del bucle; en caso contrario, se sale del bucle.

La sección iterador define lo que sucede después de cada iteración del cuerpo del bucle.

```
for (int i = 0; i < 5; i++)
{
   Console.WriteLine(i);
}
```

while

La instrucción while ejecuta una instrucción o un bloque de instrucciones mientras que una expresión booleana especificada se evalúa como true. Como esa expresión se evalúa antes de cada ejecución del bucle, un bucle while **se ejecuta cero o varias veces**. Esto es diferente del bucle do que se ejecuta una o varias veces.

En cualquier punto del bloque de instrucciones while, se puede salir del bucle mediante la instrucción **break**.

Puede ir directamente a la evaluación de la expresión while mediante la instrucción **continue**. int n = 0;

```
while (n < 5)
{
   Console.WriteLine(n);
   n++;
}
```

foreach, in

C# tiene también la estructura foreach, que al igual que en Java, permite recorrer una estructura de datos como un Array o una Lista.

En cualquier punto del bloque de instrucciones foreach, se puede salir del bucle mediante la instrucción **break**, o bien se puede ir a la siguiente iteración del bucle mediante la instrucción **continue**.

```
var fibNumbers = new List<int> { 0, 1, 1, 2, 3, 5, 8, 13 };
int count = 0;
foreach (int element in fibNumbers)
{
   count++;
   Console.WriteLine($"Element #{count}: {element}");
}
Console.WriteLine($"Number of elements: {count}");
```

Programación orientada a Objetos

Todos los lenguajes administrados en el.NET Framework, como Visual Basic y C#, proporcionan compatibilidad completa para la programación orientada a objetos como **encapsulación, herencia y polimorfismo**.

- **Encapsulación** significa que un grupo de propiedades relacionadas, métodos y otros miembros se tratan como una sola unidad u objeto.

- **Herencia** Describe la capacidad de crear nuevas clases basadas en una clase existente.

- **Polimorfismo** significa que puede tener varias clases que se pueden utilizar indistintamente, aunque cada clase implementa las mismas propiedades o métodos de maneras diferentes.

Las clases describen tipos de objetos, mientras que los objetos se crean instanciando las clases.
Las clases se declaran mediante la palabra clave **class**, como se muestra en el ejemplo siguiente:

```
class TestClass
{
   // Methods, properties, fields, events, delegates
   // and nested classes go here.
}
```

Sólo la herencia única se permite en C#. En otras palabras, una clase puede heredar la implementación de una sola clase base. Sin embargo, una clase puede implementar más de una interfaz. La siguiente tabla muestra ejemplos de herencia de clases e implementación de interfaces:

Declaración de una clase

Una clase se declara con la palabra clave **class** seguida del nombre. Como para el namespace, este nombre debe empezar por una letra o por un guión bajo (_). A continuación puede contener letras, cifras y guiones bajos. Evita utilizar caracteres acentuados y utiliza el formato "de tipo camello" (CamelCase). Por ejemplo, si escribimos una clase que emula un lector digital, el nombre en formato "CamelCase" sería LectorDigital. Las primeras letras de las palabras relacionadas se escriben en mayúscula.

Si la clase extiende una clase de base y/o implementa una o varias interfaces, el nombre de la clase está seguido del signo ':' y a continuación el nombre de la clase padre.

Los miembros de la clase (atributos, propiedades y métodos) se definen a continuación entre llaves.

Sintaxis de declaración:

```
visibilidad class NombreClase  [:[ClaseMadre], [Lista
interfaces de base]]
{
  // cuerpo de la clase
}
```

Ejemplo

```
namespace MiPrimerNameSpace
{
  class MiPrimeraClase: SuClaseMadre, Interfaz1, Interfaz2   ...
```

Visibilidad de una clase

Los modificadores de la visibilidad de una clase también son aplicables a los miembros de una clase.

- **public**: la clase puede ser utilizada en cualquier proyecto.
- **internal**: la clase está limitada al proyecto en el cual está definida.
- **private**: la clase sólo puede usarse en el módulo en la que está definida.
- **protected**: la clase sólo puede ser utilizada en una subclase. Es decir sólo se puede utilizar protected para una clase declarada en otra clase.
- **protected internal**: lo mismo que internal + protected.
- **abstract**: no permite crear instancias de esta clase, sólo sirve para ser heredada como clase base. Suelen tener los métodos definidos pero sin ninguna operatividad con lo que se suele escribir estos métodos en las clases derivadas.
- **sealed**: cuando una clase es la última de una jerarquía, por lo que no podrá ser utilizada como base de otra clase.

Las clases que se declara directamente dentro de un espacio de nombres, no anidadas dentro de otras clases, pueden ser **público o interno**. Las clases son **internal de forma predeterminada**.

Los miembros de clase, incluso clases anidadas, pueden ser público, protected internal, protegido, interno, o private.
Los miembros son private de forma predeterminada.

Una clase puede contener declaraciones de los siguientes : **Constructores, Destructores, Constantes, Campos, Métodos, Propiedades, Eventos, Clases, Interfaces, Structs.**

Constructores

Cuando se crea clase se llama a un método constructor. Los constructores tienen el mismo nombre que la clase, e inicializan normalmente los miembros de datos del nuevo objeto.

En el ejemplo siguiente, una clase denominada Taxi se define mediante un constructor simple. Esta clase crea instancias con el operador new. El operador new invoca el constructor Taxi inmediatamente después de asignar la memoria al nuevo objeto.

```
public class Taxi
{
   public bool isInitialized;
   public Taxi()
   {
      isInitialized = true;
   }
}

class TestTaxi
{
   static void Main()
   {
      Taxi t = new Taxi();
      Console.WriteLine(t.isInitialized);
   }
}
```

A menos que la clase sea estática, a las clases sin constructores se les asigna un constructor público predeterminado a través del compilador de C# con el fin de habilitar la creación de instancias de clases.

Destructores

Los destructores se utilizan para destruir instancias de clases.

- Una clase sólo puede tener un destructor.
- Los destructores no se pueden heredar ni sobrecargar.
- No se puede llamar a los destructores. Se invocan automáticamente.
- Un destructor no permite modificadores de acceso ni tiene parámetros.

Por ejemplo, el siguiente código muestra una declaración de un destructor para la clase Car:

```
class Car
{
   ~Car()  // destructor
   {
      // cleanup statements...
   }
}
```

El destructor llama implícitamente al método Finalize sobre la clase base del objeto.

No se deben utilizar destructores vacíos.

Miembros de una clase

Cada clase puede tener diferentes miembros de clase que incluyen propiedades que describen los datos de la clase (atributos), los métodos que definen el comportamiento de la clase y los eventos que proporcionan comunicación entre las distintas clases y objetos.

Cada miembro puede tener un modificador de acceso.

Un modificador de acceso especifica quienes están autorizados a “ver” ese elemento.

Si no se especifica ningún modificador de acceso, se asume que se trata de un elemento “private”.

Public: Accesible a todos los elementos.
Private: Accesible solo a esa misma clase
Protected: Accesible solo a la misma clase y métodos de sus clases derivadas. No accesible desde el exterior.
Internal: Accesible solo a ese ensamblado.
protected internal: Accesible desde el mismo ensamblado, la misma clase y métodos de sus clases derivada.

	Accesible desde ...				
Modificador de acceso	**Clase donde se declaró**	**Subclase (Mismo assembly)**	**Subclase (Distinto Assembly)**	**Externamente (Mismo Assembly)**	**Externamente (Distinto Assembly)**
private	SI	NO	NO	NO	NO
internal	SI	SI	NO	SI	NO
protected	SI	SI	SI	NO	NO
protected internal	SI	SI	SI	SI	NO
public	SI	SI	SI	SI	SI

Declaración de miembros estáticos

Cuando declara un campo de clase estático, todas las instancias de esa clase compartirán ese campo.

Una clase estática es una cuyos miembros son todos estáticos.

```
public class Automobile
{
   public static int NumberOfWheels = 4;
   public static int SizeOfGasTank
   {
      get
      {
         return 15;
      }
   }
   public static void Drive() { }
   public static event EventType RunOutOfGas;

   // Other non-static fields and properties...
}
```

Implementando propiedades get, set

La palabra clave **get** define un método que recupera el valor de la propiedad.

La palabra clave **set** define un método que asigna el valor de la propiedad.

```
class TimePeriod
{
   private double _seconds;
   public double Seconds
   {
      get { return _seconds; }
      set { _seconds = value; }
   }
}
```

Éste es un ejemplo de get y set en una propiedad autoimplementada:

```
class TimePeriod2
{
   public double Hours { get; set; }
}
```

Creación de métodos, sobrecarga y sobreescritura.

Son procedimientos o funciones definidos dentro de una CLASE. Los métodos pueden manejar los campos de la clase incluso si son privados.

La **sobrecarga** es la creación dentro de la clase, de un grupo de métodos que tienen el mismo nombre pero con un número de parámetros distinto y/o bien distintos tipos de datos.

```
public void visualización () {
  MessageBox.Show("Sr. "+Apellido+" "+Nombre+" nacido el "+FechaNac);
}

public void visualización (string idioma) {
  switch (idioma) {
    case "es":  MessageBox.Show("Sr. "+Apellido+" "+Nombre+" nacido el "+FechaNac); break;
    case "en":  MessageBox.Show("Mr. "+Apellido+" "+Nombre+" was born "+FechaNac); break;
  }
}
```

Sabemos que las clases derivadas heredan las propiedades y métodos de su clase base. Se pueden usar sin ninguna modificación, pero sí el método no está adaptado a la nueva clase podemos sobrescribirlo. Para ello utilizamos la palabra reservada **override**.

```
public override void visualización () {
  MessageBox.Show("Sr. "+Apellido+" "+Nombre+" nacido el "+FechaNac+" cobra "+Salario+".-€uros/mes.");
}
```

Ejemplo de método abstracto.

```
 public abstract string EstadoCivil();
  //de este método no hay implementación sólo definición.
```

Herecia

```
class <nombreHija>:<nombrePadre> {
    Código de la clase hija
}
```

WPF vs. WinForms

Para programar **aplicaciones con C# con entorno gráfico** en .NET existen **dos tecnologías**: **Windows Forms y WPF**.
Windows Forms es una tecnología previa a WPF. Podríamos decir que es lo **equivalente a Swing** dentro de la plataforma .NET. Es decir, tenemos un modelo de clases, que nos va a servir para crear una interfaz gráfica. Al igual que en Swing, tendremos clases que representarán a una ventana, un botón, una etiqueta, etc.
Por el contrario, **WPF** da un paso más allá en el desarrollo de aplicaciones gráficas. Con el paso de los años, se ha hecho necesaria la creación de aplicaciones **más ricas en cuanto a contenido gráfico**. WPF, permite diseñar aplicaciones mucho más atractivas, incluyendo de manera sencilla gráficos, vídeos, animaciones, etc. Nació con Windows Vista y hoy en día se utiliza ampliamente en el desarrollo de aplicaciones Windows.
La diferencia más importante entre WinForms y WPF es el hecho de que mientras **Windows Forms es simplemente una capa sobre los controles estándar de Windows** (por ejemplo, un TextBox), **WPF está construido desde cero** y no depende de los controles estándar de Windows en casi todas las situaciones.
Un gran ejemplo de esto es un botón con una imagen y texto en él. Este no es un control estándar de Windows, por lo que WinForms no le ofrece esta posibilidad de inmediato. En su lugar, deberá dibujar la imagen usted mismo, implementar su propio botón que admita imágenes o utilizar un control de terceros.

Con WPF, un botón puede contener cualquier cosa porque es esencialmente un borde con contenido y varios estados (por ejemplo, intacto, suspendido, presionado). El botón WPF es "sin aspecto", al igual que la mayoría de los otros controles WPF, lo que significa que puede contener un rango de otros controles dentro de él. ¿Quieres un botón con una imagen y algo de texto? ¡Simplemente coloca una imagen y un control TextBox dentro del botón y listo! Al igual que en .NET existe una alternativa para crear aplicaciones más ricas visualmente, ¿existe esta alternativa en el mundo Java?. La respuesta es sí, y su nombre es **Java FX**. De esta manera podemos considerar que tenemos por un lado **JavaSwing y Windows Forms** y por otro lado, **WPF y JavaFX**.

Es importante destacar que la diferencia entre Windows Forms y WPF no es tan solo visual. También cambia la forma de programa, facilitando bastante la tarea a los programadores en cuanto se acostumbran al nuevo modo de trabajo. En **WPF (también en JavaFX), se utiliza el lenguaje XML** como parte clave **para la definición del interfaz de usuario**.

De esta manera, definimos el interfaz en XML y posteriormente programamos la funcionalidad con C#.

Concretamente, el lenguaje utilizado para definir la interfaz se llama **XAML** y con él, definiremos todos los componentes que tengamos en cada pantalla, los valores de sus propiedades, etc. Este aspecto es muy importante, ya que conseguimos separar de manera efectiva la vista de la lógica de la aplicación, siendo de esta manera automática la separación de la aplicación por capas.

Vamos a poder trabajar con el concepto de **binding** entre los componentes y fuentes de datos. Todos habéis visto como para cargar en Swing un ComboBox o un JTable, tenemos que rellenar un modelo manualmente, y si los datos cambian, deberíamos de volver a rellenar el modelo para que se muestren los datos. En WPF, al tener el concepto de binding, podemos asociar una estructura de datos con un componente visual. Esto quiere decir que podemos decir por ejemplo, que una lista, se muestre en un ComboBox. Si a lo largo de la ejecución, esta lista cambia porque se le añada o borre un elemento, el componente actualizará automáticamente el componente visual. De alguna manera, nos quitamos de tener que programar toda la parte del modelo y de su actualización, haciendo mucho más sencillas de programar las aplicaciones de entrada y salida de datos.

Ventajas de WPF

- Es más nuevo y, por lo tanto, está más en consonancia con los estándares actuales.
- Microsoft lo está utilizando para muchas aplicaciones nuevas, por ejemplo, Visual Studio.
- Es más flexible, por lo que puede hacer más cosas sin tener que escribir o comprar nuevos controles.
- Cuando necesitemos utilizar controles de terceros, es probable que los desarrolladores de estos controles se enfoquen más en WPF porque es más reciente.
- XAML hace que sea fácil crear y editar su GUI, y permite que el trabajo se divida entre un diseñador (XAML) y un programador (C#, VB.net etc.).
- Enlace de datos (Databinding), le permite obtener una separación más limpia de los datos y el diseño.
- Utiliza la aceleración de hardware para dibujar la GUI, para un mejor rendimiento.
- Permite realizar interfaces de usuario tanto para aplicaciones Windows como para aplicaciones web (Silverlight/XBAP).

Ventajas de WinForms

- Tiene más tiempo por lo que ha sido más utilizado y probado
- Ya hay muchos controles de empresas de terceros que puedes comprar o conseguir gratis
- El diseñador en Visual Studio sigue siendo, en cuanto a la escritura, mejor para WinForms que para WPF, donde tendremos que hacer más trabajo nosotros mismos con WPF