Hada T0: Plataforma .NET y C#

Herramientas Avanzadas para el Desarrollo de Aplicaciones

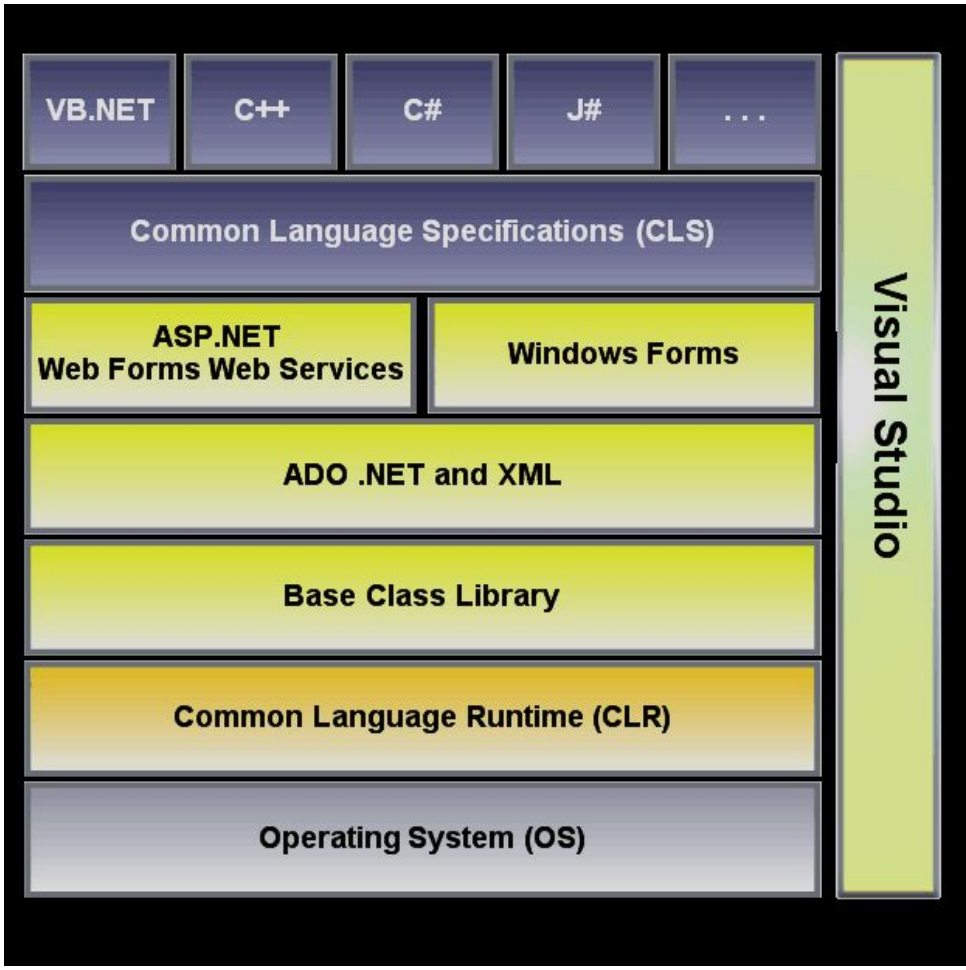
Introducción a .Net

¿Qué es .Net?

Plataforma .NET

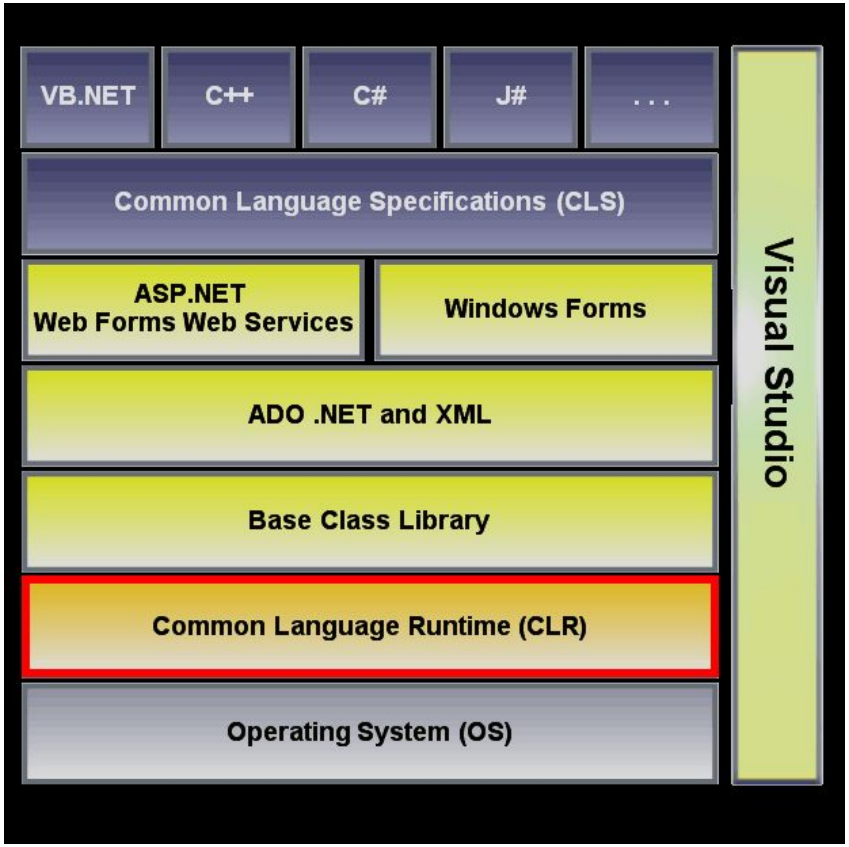
- **la plataforma .NET es un amplio conjunto de bibliotecas de desarrollo**

- Ofrece **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.



CLR

- El **CLR o *Common Language Runtime*** es la parte de .NET encargada de ejecutar las aplicaciones desarrolladas para la plataforma
- El funcionamiento del CLR no es trivial, trabaja encima del sistema operativo para aislar a la plataforma de éste. **Su funcionamiento es muy parecido a una máquina virtual**.
- Esto le permite ejecutar aplicaciones **.NET multiplataforma.** Hoy en día es posible desarrollar aplicaciones .NET para diversas plataformas, como por ejemplo Windows, iOS, Android o Linux.



CLR

- El CLR nos garantiza también **la seguridad de los tipos de datos**, avalando que no se producen errores en la conversión de tipos en la ejecución de una aplicación .NET. Este aspecto y algunos otros vienen regulados por lo que se conoce el ***Common Type System*** **(CTS)** o Sistema Común de Tipos de datos.

- El **CTS** define los tipos de datos de .NET y las construcciones de programación de los lenguajes que el CLR puede utilizar de forma adecuada y correcta.
- En otras palabras, el CTS es lo más parecido a **las reglas de juego que permiten el correcto entendimiento entre diferentes lenguajes de programación** y el propio entorno de ejecución de .NET.

CLR

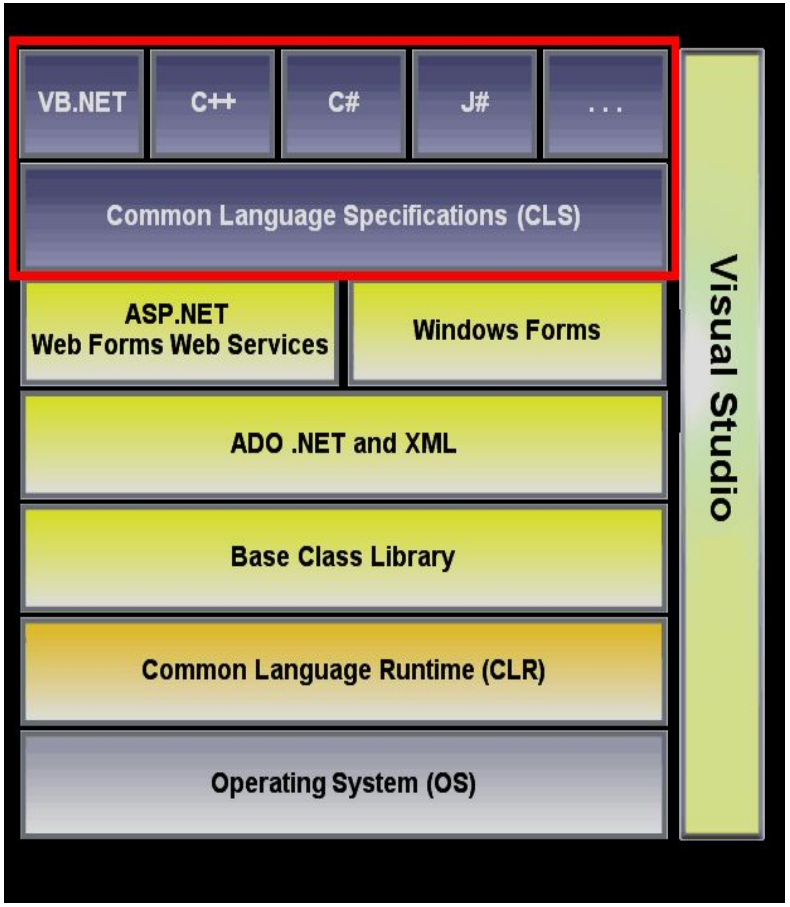
- Otra característica del CLR es la posibilidad de **reutilizar porciones de código escritos en diferentes lenguajes**. Esto es posible gracias a que todo el código, esté escrito en el lenguaje que esté escrito, debe utilizar las mismas "reglas de juego" de las que hablábamos antes, marcadas por el CLR.
- Adicionalmente, **el CLR se encarga también de gestionar la vida de los objetos**, declaraciones y recursos a lo largo de la ejecución de una aplicación .NET.
- Esto se lleva a cabo a través de lo que se conoce como **recolector de basura** o *garbage collector*. *Por lo tanto, a la hora de programar no debemos preocuparnos de reservar espacio de memoria para ejecutar nuestra aplicación .NET. Ni tampoco de liberar los recursos del sistema una vez finaliza la ejecución de la aplicación.* El CLR se encarga de ello y nos exime de esta responsabilidad, facilitando el desarrollo enormemente frente a otros lenguajes "tradicionales" como C/C++.

Ejecución de aplicaciones

El CLR es el encargado de gestionar la ejecución de una aplicación .NET. Debido a esta responsabilidad, **a las aplicaciones de .NET se las conoce como aplicaciones "manejadas"** o aplicaciones **de código gestionado**.

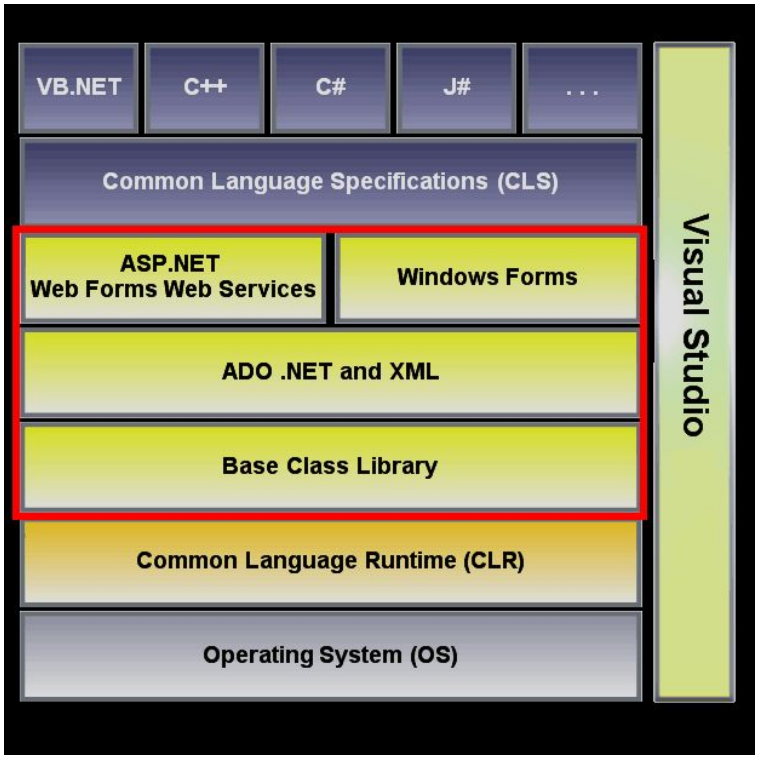
CLS - Common Language Specification

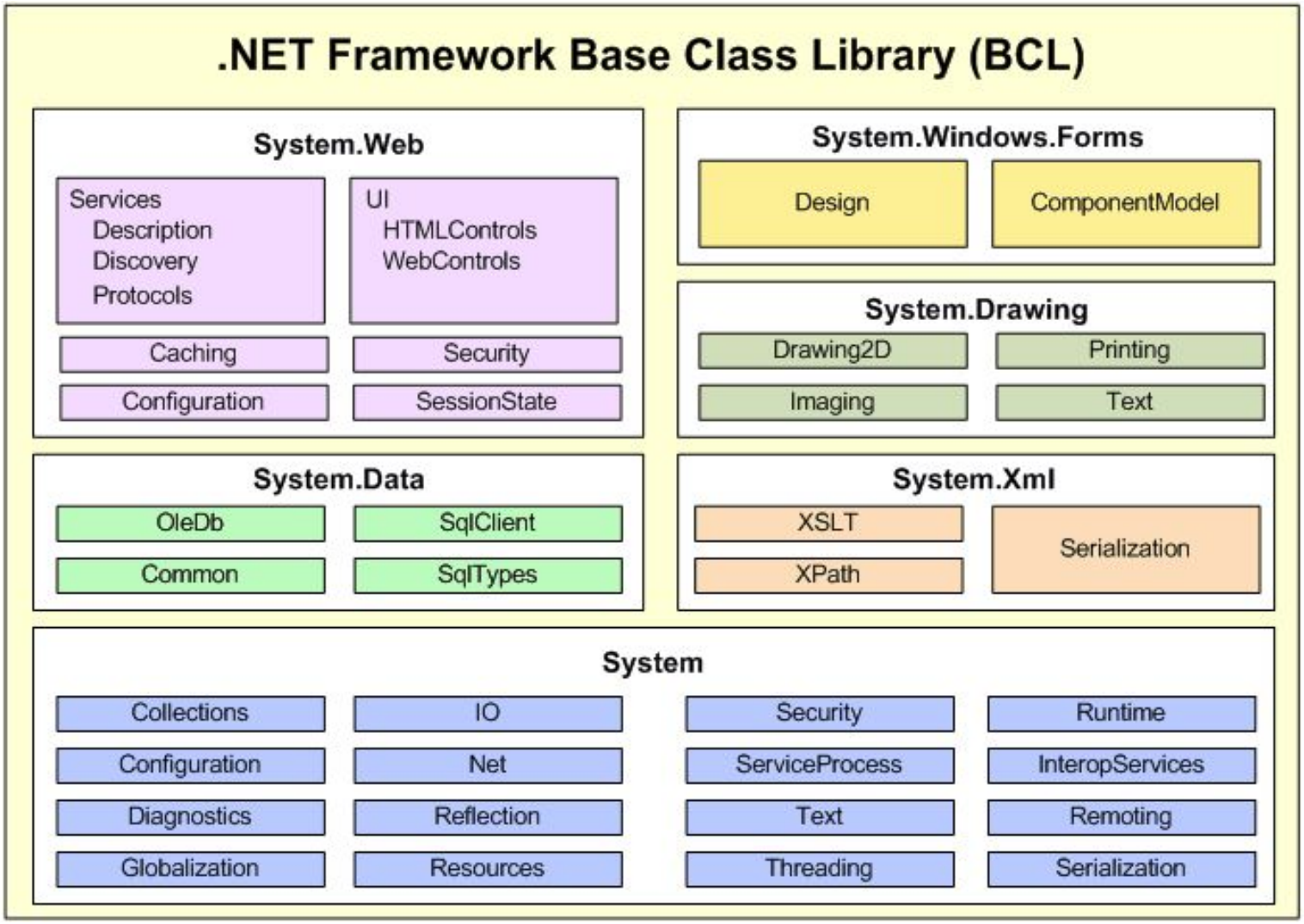
- Al contrario que otros entornos, **la plataforma .NET no está atada a un determinado lenguaje de programación**. En la actualidad existen implementaciones para gran cantidad de lenguajes de programación que permiten escribir aplicaciones para esta plataforma.
- Entre estos lenguajes de programación destacan Visual Basic ó C#,
- La **especificación del lenguaje común o CLS** 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..



Base Class Library

- La BCL está formada por **bibliotecas o APIs especializadas que pueden ser utilizadas por todos los lenguajes** de programación de la plataforma .NET.
- Cada una de estas bibliotecas puede contener a su vez numerosas clases que aglutinan varios métodos y funciones con características concretas.
- De esta manera, podemos encontrar bibliotecas con **funcionalidades para casi cualquier cosa que necesitemos**: enviar correos electrónicos, escribir archivos de texto, acceder a fuentes de datos, manejar información, criptografía, etc...




.NET Framework Base Class Library (BCL)
System.Web
Services
Description
Discovery
Protocols
UI
HTMLControls
WebControls
Caching
Security
Configuration
SessionState
System.Windows.Forms
Design
ComponentModel
System.Drawing
Drawing2D
Printing
Imaging
Text
System.Data
OleDb
SqlClient
Common
SqlTypes
System.Xml
XSLT
XPath
Serialization
System
Collections
IO
Security
Runtime
Configuration
Net
ServiceProcess
InteropServices
Diagnostics
Reflection
Text
Remoting
Globalization
Resources
Threading
Serialization

Introducción a C#

Breve introducción a C# I

- Es un lenguaje de la familia de C, muy parecido a C++ y a Java.
- Es **orientado objetos**.
- Dispone de tipos de valor que admiten valores NULL, enumeraciones, delegados, expresiones lambda y acceso directo a memoria, que no se encuentran en Java.
- Es un lenguaje de especificaciones seguras (type-safe); el compilador de C# garantiza que los valores almacenados en variables son siempre del tipo adecuado.
- C# admite métodos y tipos genéricos, que proporcionan mayor rendimiento y seguridad de tipos.

Breve introducción a C# II

- C# admite los conceptos de encapsulación, herencia y polimorfismo.
- Todas las variables y métodos, incluido el método **Main** (punto de entrada de la aplicación), se encapsulan dentro de definiciones de clase.
- Una clase puede heredar directamente de una sola clase primaria, pero puede implementar cualquier número de interfaces.
- Los métodos que reemplazan a los métodos virtuales en una clase primaria requieren la palabra clave `override` como medio para evitar redefiniciones accidentales.

Breve introducción a C# III

- También dispone de ***propiedades***, que actúan como descriptores de acceso para variables miembro privadas.
- La E/S se hace a través de métodos de la clase `Console`, p.e.: **`Console.WriteLine("Hello World!");`**
- Algunos de los ejemplos de código que verás a continuación pertenecen a msdn.

Hola Mundo en C#

```
// A "Hello World!" program in C#

class Hello {

   static void Main() {

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

   }

}
```

- El método Main puede tener estas signaturas:
  - `static void Main()`
  - `static int Main()`
  - `static void Main(string[] args)`
  - `static int Main(string[] args).`

Espacios de nombres en C# I

- Los espacios de nombres ayudan a controlar el ámbito de clases y nombres de métodos en proyectos de programación grandes.
- Se declaran mediante el uso de la palabra clave **namespace.**

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

Espacios de nombres en C# II

- Los espacios de nombres **organizan proyectos de código de gran tamaño.**
- El operador "." delimita los espacios de nombres.
- La directiva "**using**" hace que no sea necesario especificar el nombre del espacio de nombres para cada símbolo del mismo
- Alias de espacio de nombres:
  - `using Co = Company.Proj.Nested;`
- Sintaxis sencilla para nombres anidados:
  - `namespace N1.N2 { class C3 /* N1.N2.C3 */ {} }`

Matrices en C#

```
class TestArraysClass {
    static void Main() {
        // Declare a single-dimensional array
        int[] array1 = new int[5];

        // Declare and set array element values
        int[] array2 = new int[] { 1, 3, 5, 7, 9 };

        // Alternative syntax
        int[] array3 = { 1, 2, 3, 4, 5, 6 };

        // Declare a two dimensional array
        int[,] multiDimensionalArray1 = new int[2, 3];

        // Declare and set array element values
        int[,] multiDimensionalArray2 = {
           { 1, 2, 3 },
           { 4, 5, 6 }
        };
        ...
```

```
        ....
        // Declare a jagged array
        int[][] jaggedArray = new int[6][];

        // Set the values of the first array
        // in the jagged array structure
        jaggedArray[0] = new int[4] { 1, 2, 3, 4 };
    }
}
```

Cadenas en C#

```
// Declarara una cadena sin inicializarla
string message1;

// Inicializar la cadena a null
string message2 = null;

// También se puede usar System.String en lugar de
// string
System.String greeting = "Hello World!";

// Inicializar con la cadena vacía
// La constante Empty equivale al literal "".
string message3 = System.String.Empty;

// Inicializar con un literal normal
string oldPath = "c:\\Program Files\\Microsoft Visual
Studio 8.0";

// Inicializar con un literal “verbatim”. No es
// necesario escapar carácter ‘\’
string newPath = @"c:\Program Files\Microsoft Visual
Studio 9.0";
```

```
// Declaración de una cadena constante para evitar
// que message4 pueda albergar otra cadena
const string message4 = "You can't get rid of me!";

// Crear una cadena a partir de char* o char[]
char[] letters = { 'A', 'B', 'C' };
string alphabet = new string(letters);

// Las cadena son inmutable
string s1 = "Hola";
string s2 = s1;
s1 += " mundo!";
System.Console.WriteLine(s2); //Salida: “Hola”
```

Clases y propiedades en C#

```
namespace ProgrammingGuide {
    // Class definition.
    public class MyCustomClass {
        // Class members:
        // Property.
        public int Number { get; set; }
        // Method.
        public int Multiply(int num) { return num * Number; }
        // Instance Constructor.
        public MyCustomClass() { Number = 0; }
    }
    // Another class definition. This one contains the Main method, the entry point for the program.
    class Program {
        static void Main(string[] args) {
            // Create an object of type MyCustomClass.
            MyCustomClass myClass = new MyCustomClass();
            // Set the value of a public property.
            myClass.Number = 27;
            // Call a public method.
            int result = myClass.Multiply(4);
        }
    }
}
```

Genéricos en C#

```
// Declare the generic class.
public class GenericList<T> { void Add(T input) { } }
class TestGenericList {
   private class ExampleClass { }
   static void Main() {
      // Declare a list of type int.
      GenericList<int> list1 = new GenericList<int>();
      // Declare a list of type string.
      GenericList<string> list2 = new GenericList<string>();
      // Declare a list of type ExampleClass.
      GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
   }
}
```

Interfaces en C#

```
// Admiten genericidad
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

```
// O no
interface ICarComparable
{
    bool Equals(Car obj);
}
```

Colecciones en C#

- Las colecciones de elementos en C# forman parte de su biblioteca estándar.
- Existen varios tipos de colecciones, las más versátiles emplean genericidad (List<T>, Stack<T>, Queue<T>, etc…).
- Colecciones más usadas:
  - ArrayList: Representa un array de objetos que puede crecer dinámicamente.
  - Hashtable: Representa una colección pares clave/valor organizados por el valor hash code de la clave.
  - Queue: Representa una colección de objetos tipo FIFO.
  - Stack: Representa una colección de objetos tipo LIFO.
- Puedes encontrar más información sobre ellas aquí.

Delegados en C#

- Son punteros a funciones.
- Tienen una sintaxis muy sencilla.
- Se suelen emplear con expresiones lambda.

```
public delegate int PerformCalculation(int x, int y);
```

Expresiones λ en C#

- Una expresión lambda es una función anónima que se puede usar para crear tipos delegados.
- Al utilizar expresiones lambda, se pueden escribir funciones locales que se pueden pasar como argumentos o devolverse como valor de llamadas de función.

```
delegate int del(int i);
static void Main(string[] args) {
    del myDelegate = x => x * x;
    int j = myDelegate(5); //j = 25
}
```

Eventos en C# I

- Cuando ocurre algo *interesante*, los *eventos* permiten que un objeto pueda notificarlo a otros objetos.
- **La clase que envía (o genera) el evento recibe el nombre de *publicador* y las clases que reciben (o controlan) el evento se denominan *suscriptores*.**
- El *publicador* determina el momento en el que se genera un evento; los *suscriptores* determinan la acción que se lleva a cabo en respuesta al evento.
- Un evento puede tener varios suscriptores. Un suscriptor puede controlar varios eventos de varios publicadores.
- Nunca se generan eventos que no tienen suscriptores.

Eventos en C# II

- Los eventos se suelen usar para indicar acciones del usuario, como los clicks de los botones o las selecciones de menú en las interfaces gráficas de usuario.
- Cuando un evento tiene varios suscriptores, los controladores de eventos se invocan síncronamente cuando éste se genera.
- En la biblioteca de clases .NET Framework, los eventos se basan en el delegado EventHandler y en la clase base EventArgs.

Excepciones en C# I

- C# ofrece facilidades para la gestión de errores por medio del manejo de excepciones
- Una excepción es un objeto que se crea cuando se produce una situación de error específica
- Además el objeto contiene información que permite resolver el problema

Excepciones en C# II

```
try
{
    // Codigo de ejecución normal
}
catch
{
    // Gestión de errores
}
finally
{
    // Liberación de recursos (opcional)
}
```