

Tema Introducción a C#

Contenido

1	Framework .NET	3
1.1	Componentes	4
1.2	Microsoft .NET Core	5
1.3	Librerías de clases Base	5
1.4	Extensiones de .NET 3.X	7
1.5	Extensiones a partir de la versión 3.5	10
1.6	Extensiones para .Net 4.X	13
1.7	Extesiones a partir de la versión .NET 4.5	14
2	Microsoft Visual Studio Community. IDE	17
2.1	Instalación	17
2.2	Primer proyecto en C#	21
2.2.1	La estructura del proyecto.....	23
2.2.2	Áreas en el IDE.....	25
2.2.3	Configurando el proyecto	27
3	Lenguaje de programación C#	29
3.1	Introducción	29
3.2	Instrucciones en c#	30
3.2.1	Tipos de instrucciones.....	30
3.2.2	Formato de las instrucciones	31
3.3	Variables en C#	32
3.3.1	Constantes	32
3.4	Lectura y escritura en consola.....	32
3.5	Tipos en C Sharp	33
3.5.1	Tipos numéricos (por valor).....	33
3.5.2	Tipos de carácter y cadena	35
3.5.3	Booleanos.....	36
3.5.4	Los tipos Nullables.	38
3.5.5	Tipo Object (Por referencia)	39
3.5.6	Tipo dynamic.....	39
3.5.7	Tipos indefinidos	39
3.6	Operadores	39
3.7	Ejercicios	41
3.8	Instrucciones de control	41
3.8.1	Instrucción If	42
3.8.2	Instrucción switch	42
3.8.3	While	44
3.8.4	Instrucción Do While	44
3.8.5	For	44
3.8.6	Instrucción foreach	45
3.8.7	Ejercicios	46
3.9	Tipos avanzados en c#	47
3.9.1	Enumeraciones.....	47
3.9.2	Tuplas.....	49

3.9.3	Estructuras	50
3.9.4	Arrays	51
3.10	Cadenas en c#.....	56
3.10.1	El método ToString.....	57
3.10.2	Strings son inmutables.....	59
3.10.3	Caracteres de escape	60
3.10.4	String.Format	61
3.10.5	Elementos de formato.....	62
3.10.6	StringBuilder	65
4	Programación orientada a objetos. Orientacion a objetos en c#	69
4.1	Pilares de la programación orientada a objetos	70
4.1.1	Abstraction.....	71
4.1.2	Encapsulación.....	72
4.1.3	Herencia	73
4.1.4	Polimorfismo	74
4.2	La clase Object en C#.....	74
4.2.1	Ejercicio. Cornell notes	76
4.3	Clases en c#	76
4.3.1	Creando la primera clase	77
4.3.2	Modificando la clase Persona.....	81
4.3.3	Creando un objeto de tipo Persona en el programa principal.	84
4.3.4	Ejercicio practico	84
4.4	Herencia en c#.....	84
4.4.1	Ejercicio.	86
4.5	Ampliación de métodos. Paso por referencia y por valor.....	86
4.5.1	Ejercicio	89
4.6	Polimorfismo y clases abstractas en C#	90
4.6.1	Interfaces.....	92
4.7	Clases e interfaces genéricos	94
4.8	Interfaces útiles en C#.....	98
4.8.1	IComparable	98
4.8.2	Ejercicio	100
4.8.3	ICloneable	100
4.8.4	IFormattable e IEquatable	101
5	Tipos delegados en C#.....	101
5.1.1	Ejercicio	104
5.2	Usando delegados como parámetros o en setters	104
5.2.1	Ejercicio	110
5.2.2	Ejercicio	110
5.2.3	Ejercicio	110
5.3	Delegados predefinidos en C#	110
6	Expresiones lambda en C#. Funciones o métodos anónimos.....	113
6.1	Introducción	113
6.2	Clases anónimas en C#.....	113
6.3	Expresiones lambda en c#.....	114
6.3.1	Ejercicios	117
7	Excepciones.....	117
7.1	La clase Exception.....	119
7.2	Creando nuestras propias excepciones	119

1 Framework .NET

.NET es un framework de Microsoft que hace un énfasis en la transparencia de redes, con independencia de plataforma de hardware y que permite un rápido desarrollo de aplicaciones. Basada en ella, la empresa intenta desarrollar una estrategia horizontal que integre sus productos, desde el sistema operativo hasta las herramientas de mercado.

.NET podría considerarse una respuesta de Microsoft al creciente mercado de los negocios en entornos Web, como competencia a la plataforma Java de Oracle Corporation y a los diversos framework de desarrollo web basados en PHP. Su propuesta es ofrecer una manera rápida y económica, a la vez que segura y robusta, de desarrollar aplicaciones -o como la misma plataforma las denomina, soluciones- permitiendo una integración más rápida y ágil entre empresas y un acceso más simple y universal a todo tipo de información desde cualquier tipo de dispositivo.

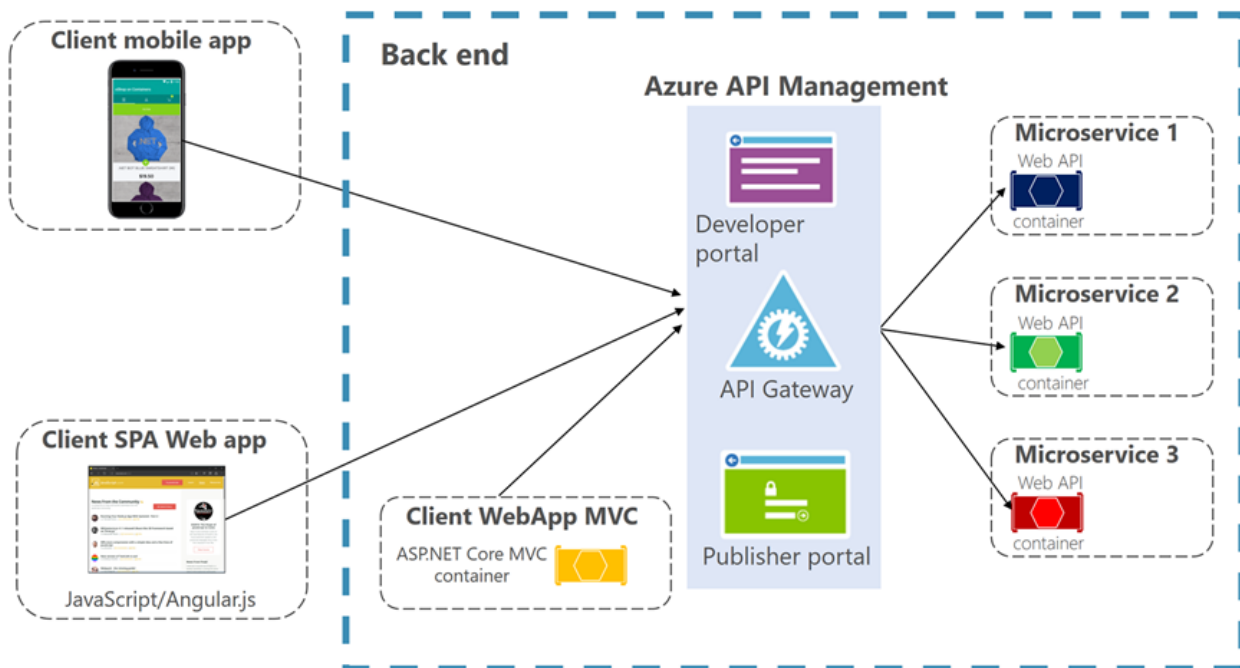
La plataforma .NET de Microsoft es un componente de software que puede ser añadido al sistema operativo Windows. Provee un extenso conjunto de soluciones predefinidas para necesidades generales de la programación de aplicaciones, y administra la ejecución de los programas escritos específicamente con la plataforma. Esta solución es el producto principal en la oferta de Microsoft, y pretende ser utilizada por la mayoría de las aplicaciones creadas para la plataforma Windows.

.NET Framework se incluye en Windows Server 2008, Windows Vista y Windows 7. De igual manera, la versión actual de dicho componente puede ser instalada en Windows XP, y en la familia de sistemas operativos Windows Server 2003. Una versión "reducida" de .NET Framework está disponible para la plataforma Windows Mobile, incluyendo teléfonos inteligentes.

La norma (incluido en ECMA-335, ISO/IEC 23271) que define el conjunto de funciones que debe implementar la biblioteca de clases base (BCL por sus siglas en inglés, tal vez el más importante de los componentes de la plataforma), define un conjunto funcional mínimo que debe implementarse para que el marco de trabajo sea soportado por un sistema operativo. Aunque Microsoft implementó esta norma para su sistema operativo Windows, la publicación de la norma abre la posibilidad de que sea implementada para cualquier otro sistema operativo existente o futuro, permitiendo que las aplicaciones corran sobre la plataforma independientemente del sistema operativo para el cual haya sido implementada.

En resumen es un framework para implementar aplicaciones en múltiples plataformas:

Recommended API Gateway use with Azure API Management for production environments



1.1 Componentes

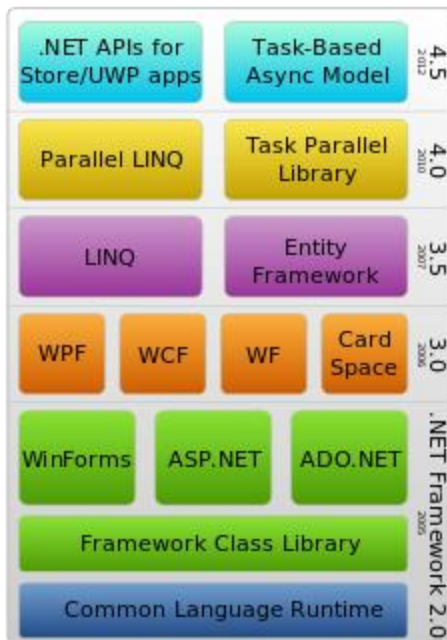
Los principales componentes del marco de trabajo son:

- El conjunto de lenguajes de programación.
- La biblioteca de clases base o BCL.
- El entorno común de ejecución para lenguajes, o CLR (Common Language Runtime) por sus siglas en inglés.

Debido a la publicación de la norma para la infraestructura común de lenguajes (CLI por sus siglas en inglés), el desarrollo de lenguajes se facilita, por lo que el marco de trabajo .NET soporta ya más de 20 lenguajes de programación y es posible desarrollar cualquiera de los tipos de aplicaciones soportados en la plataforma con cualquiera de ellos, lo que elimina las diferencias que existían entre lo que era posible hacer con uno u otro lenguaje.

Algunos de los lenguajes desarrollados para el marco de trabajo .NET son: C#, Visual Basic .NET, Delphi (Object Pascal), C++, F#, J#, Perl, Python, Fortran, Prolog (existen al menos dos implementaciones, el P#1 y el Prolog.NET2), Cobol y PowerBuilder.

En la siguiente figura podemos observar los diferentes niveles del framework:



El CLR es el verdadero núcleo del framework de .NET, entorno de ejecución en el que se cargan las aplicaciones desarrolladas en los distintos lenguajes, ampliando el conjunto de servicios del sistema operativo (W2k y W2003). Permite integrar proyectos en distintos lenguajes soportados por la plataforma .Net, como C++, Visual Basic, C#, entre otros.

La herramienta de desarrollo compila el código fuente de cualquiera de los lenguajes soportados por .NET en un código intermedio, el CIL (Common Intermediate Language) antes conocido como MSIL (Microsoft Intermediate Language), similar al BYTECODE de Java. Para generarlo, el compilador se basa en la especificación CLS (Common Language Specification) que determina las reglas necesarias para crear el código MSIL compatible con el CLR.

Para ejecutarse se necesita un segundo paso, un compilador JIT (Just-In-Time) es el que genera el código máquina real que se ejecuta en la plataforma del cliente. De esta forma se consigue con .NET independencia de la plataforma de hardware. La compilación JIT la realiza el CLR a medida que el programa invoca métodos

1.2 Microsoft .NET Core

Hasta 2015, .NET brindaba únicamente soporte para Windows, además de que su código tenía una licencia patentada. Ello derivó en la creación de implementaciones libres, tales como Mono. No obstante, Mono seguía teniendo limitaciones frente a .NET, sobre todo en lo relacionado con WinForms (herramienta para interfáces gráficas de Windows), además de ciertos problemas de patentes. Por ello, Microsoft decidió liberar parte del framework .NET bajo el nombre de .NET Core. Posteriormente, se la añadió soporte para ASP .NET, ML .NET y WinForms. Se espera que .NET Core reemplace a .NET Framework en un futuro.

1.3 Librerías de clases Base

La Biblioteca de Clases Base se clasifica, en cuatro grupos clave:

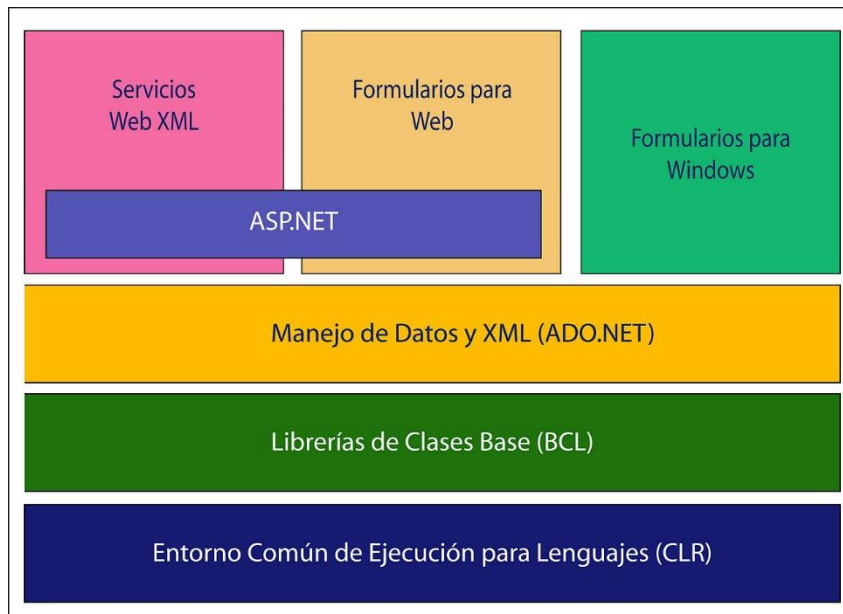
- **ASP.NET y Servicios Web XML:** nos proporciona capacidades para realizar modelos MVC web y servicios Web. ASP.NET es un marco web gratuito para crear excelentes sitios web y aplicaciones web mediante HTML, CSS y JavaScript. También puede crear API Web y usar tecnologías en tiempo real como Sockets Web.
- **Windows Forms:** Windows Forms es un marco de interfaz de usuario para compilar aplicaciones de escritorio de Windows. Proporciona una de las formas más productivas de crear aplicaciones de escritorio basadas en el diseñador visual proporcionado en Visual Studio. Funciones como la colocación de controles visuales mediante arrastrar y colocar facilita la compilación de aplicaciones de escritorio.

Con Windows Forms, puede desarrollar aplicaciones enriquecidas gráficamente que son fáciles de implementar, y actualizar, y con las que se puede trabajar sin conexión o mientras están conectadas a Internet. Las aplicaciones de Windows Forms pueden acceder al hardware local y al sistema de archivos del equipo en el que se ejecutan.

- **ADO.NET:** es un conjunto de clases que exponen servicios de acceso a datos para programadores de .NET Framework. ADO.NET ofrece abundancia de componentes para la creación de aplicaciones de uso compartido de datos distribuidas. Constituye una parte integral de .NET Framework y proporciona acceso a datos relacionales, XML y de aplicaciones. ADO.NET satisface diversas necesidades de desarrollo, como la creación de clientes de base de datos front-end y objetos empresariales de nivel medio que utilizan aplicaciones, herramientas, lenguajes o exploradores de Internet.

Proporciona acceso a datos para SQL Server, XML, y a cualquier base de datos a través de los servicios OLEDB y ODBC.

- **.NET :** Es una herramienta para compilar aplicaciones en múltiples sistemas operativos. Puede crear aplicaciones .NET para muchos sistemas operativos, entre los que se incluyen los siguientes:
 - ✓ Windows
 - ✓ macOS
 - ✓ Linux
 - ✓ Android
 - ✓ iOS
 - ✓ tvOS
 - ✓ watchOS



1.4 Extensiones de .NET 3.X

Distinguimos en este apartado cuatro componentes claves para desarrollar aplicaciones:

- ✓ WPF: para formularios bajo estándar XAML
- ✓ WCF: para comunicaciones cliente servidor y entre aplicaciones.
- ✓ WF: para desarrollar un flujo de trabajo entre actividades
- ✓ Windows Card Space: para autenticación.

WPF permite desarrollar una aplicación que use tanto *marcado* como *código subyacente*, una experiencia que les resultará familiar a **los desarrolladores de ASP.NET**. En general, se usa marcado XAML para implementar la apariencia de una aplicación y lenguajes de programación administrados (código subyacente) para implementar su comportamiento. Esta separación entre la apariencia y el comportamiento tiene las ventajas siguientes:

- ✓ Se reducen los costos de desarrollo y mantenimiento porque el marcado específico de la apariencia no está asociado estrechamente al código específico del comportamiento.
- ✓ La programación es más eficaz porque los diseñadores pueden implementar la apariencia de una aplicación al mismo tiempo que los programadores implementan su comportamiento.
- ✓ La globalización y localización de las aplicaciones WPF se ha simplificado.

Ejemplo de XAML para xamarin:

```
ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
            xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
            xmlns:local="clr-namespace:XamlSamples"
            x:Class="XamlSamples.MainPage">
```

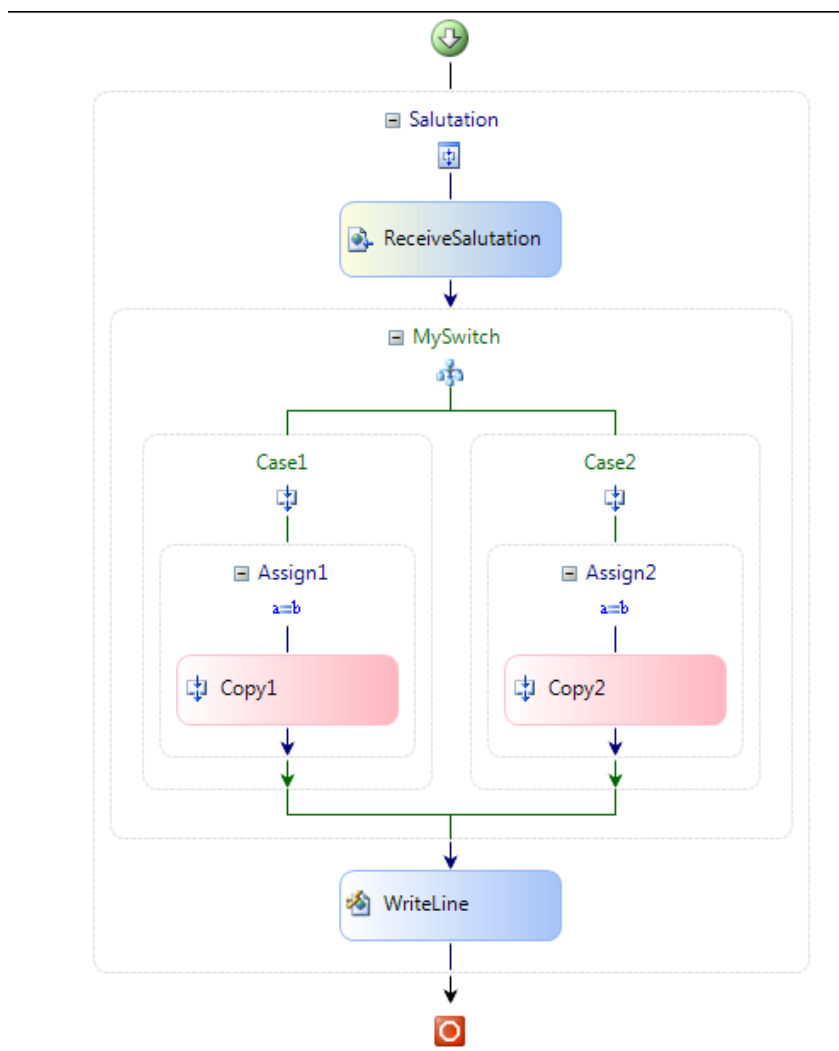
```
<StackLayout>
    <!-- Place new controls here -->
    <Label Text="Welcome to Xamarin Forms!"
          VerticalOptions="Center"
          HorizontalOptions="Center" />
</StackLayout>
```

```
</ContentPage>
```

Windows Communication Foundation (WCF) es un marco para la creación de aplicaciones orientadas a servicios. Con WCF, puede enviar datos como mensajes asincrónicos de un punto de conexión de servicio a otro. Un extremo de servicio puede formar parte de un servicio disponible continuamente hospedado por IIS (Internet Information Server, Servidor web de Microsoft), o puede ser un servicio hospedado en una aplicación. Un extremo puede ser un cliente de un servicio que solicita datos de un extremo de servicio. Los mensajes pueden ser tan simples como un carácter o una palabra que se envía como XML, o tan complejos como una secuencia de datos binarios. A continuación, se indican unos cuantos escenarios de ejemplo:

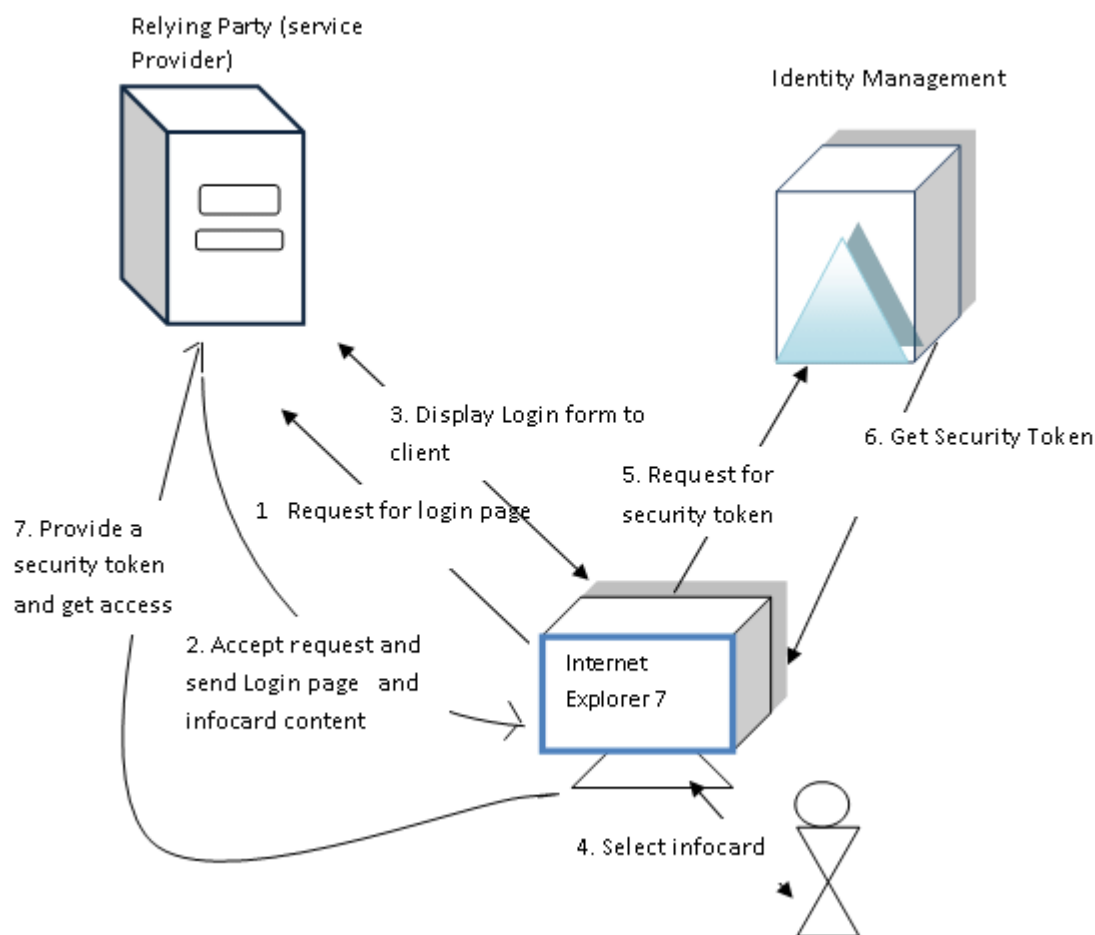
- ✓ Un servicio seguro para procesar transacciones comerciales.
- ✓ Un servicio que proporciona datos actualizados a otras personas, como un informe sobre tráfico u otro servicio de supervisión.
- ✓ Un servicio de chat que permite a dos personas comunicarse o intercambiar datos en tiempo real.
- ✓ Una aplicación de panel que sondea los datos de uno o varios servicios y los muestra en una presentación lógica.
- ✓ Exponer un flujo de trabajo implementado utilizando Windows Workflow Foundation como un servicio WCF.

Windows Workflow Foundation (WF) eleva el nivel de abstracción para desarrollar aplicaciones interactivas de ejecución prolongada. Las unidades de trabajo se encapsulan como actividades. Las actividades se ejecutan en un entorno que proporcione los medios para el control de flujo, el control de excepciones, la propagación de errores, la persistencia de los datos de estado, la carga y descarga de flujos de trabajo en progreso de la memoria, el seguimiento y el flujo de la transacción.



Windows Cardspace (anteriormente conocido como InfoCard) es una nueva tecnología de identificación incluida a partir de Windows Vista que consiste en un metasisistema de identidad que nos permite identificarnos en Internet de forma rápida y segura, dándonos la posibilidad de iniciar sesión en los sitios web compatibles sin tener que dar nuestro nombre de usuario y contraseña.

Cardspace nos posibilita crear tarjetas de identificación basadas en XML con las que podemos identificarnos en aquellos servicios que admitan esta tecnología. Cardspace envía la tarjeta de forma cifrada, el sitio web la reconoce y la asocia a nuestra cuenta de usuario y así iniciamos sesión sin dar nuestra contraseña. Las tarjetas no se envían a ningún sitio que no tenga sus certificados de seguridad al día, y en nuestro computador se guarda un historial de todos los sitios web a los que la tarjeta se ha enviado. De esta manera evitamos problemas de seguridad y privacidad.



1.5 Extensiones a partir de la versión 3.5

LINQ y Entity framework nos van a permitir la creación de consultas a cualquier fuente de datos XML o base de datos, transformando estos a Objetos al estilo ORM.

Language-Integrated Query (LINQ) es el nombre de un conjunto de tecnologías basadas en la integración de capacidades de consulta directamente en el lenguaje C#. Tradicionalmente, las consultas con datos se expresaban como cadenas simples sin comprobación de tipos en tiempo de compilación ni compatibilidad con IntelliSense. Además, tendrá que aprender un lenguaje de consulta diferente para cada tipo de origen de datos: bases de datos SQL, documentos XML, varios servicios web y así sucesivamente. Con LINQ, una consulta es una construcción de lenguaje de primera clase, como clases, métodos y eventos. Escribe consultas en colecciones de objetos fuertemente tipadas con palabras clave del lenguaje y operadores familiares. La familia de tecnologías de LINQ proporciona una experiencia de consulta coherente para objetos (LINQ to Objects), bases de datos relacionales (LINQ to SQL) y XML (LINQ to XML).

Para un desarrollador que escribe consultas, la parte más visible de "lenguaje integrado" de LINQ es la expresión de consulta. Las expresiones de consulta se escriben con una sintaxis de consulta declarativa. Con la sintaxis de consulta, puede realizar operaciones de filtrado, ordenación y agrupamiento en orígenes de datos con el mínimo código. Utilice los mismos patrones de expresión de consulta básica para consultar y transformar datos de bases de

datos SQL, conjuntos de datos de ADO .NET, secuencias y documentos XML y colecciones. NET.

Puede escribir consultas LINQ en C# para bases de datos de SQL Server, documentos XML, conjuntos de datos ADO.NET y cualquier colección de objetos que admita IEnumerable o la interfaz genérica IEnumerable<T>. La compatibilidad con LINQ también se proporciona por terceros para muchos servicios web y otras implementaciones de base de datos.

Ejemplo de consulta LINQ sobre un Enumerable (una lista)

```
// Create a data source by using a collection initializer.
static List<Student> students = new List<Student>
{
    new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores= new List<int>
{97, 92, 81, 60}},
    new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new List<int> {75,
84, 91, 39}},
    new Student {First="Sven", Last="Mortensen", ID=113, Scores= new List<int> {88,
94, 65, 91}},
    new Student {First="Cesar", Last="Garcia", ID=114, Scores= new List<int> {97, 89,
85, 82}},
    new Student {First="Debra", Last="Garcia", ID=115, Scores= new List<int> {35, 72,
91, 70}},
    new Student {First="Fadi", Last="Fakhouri", ID=116, Scores= new List<int> {99,
86, 90, 94}},
    new Student {First="Hanying", Last="Feng", ID=117, Scores= new List<int> {93, 92,
80, 87}},
    new Student {First="Hugo", Last="Garcia", ID=118, Scores= new List<int> {92, 90,
83, 78}},
    new Student {First="Lance", Last="Tucker", ID=119, Scores= new List<int> {68, 79,
88, 92}},
    new Student {First="Terry", Last="Adams", ID=120, Scores= new List<int> {99, 82,
81, 79}},
    new Student {First="Eugene", Last="Zabokritski", ID=121, Scores= new List<int>
{96, 85, 91, 60}},
    new Student {First="Michael", Last="Tucker", ID=122, Scores= new List<int> {94,
92, 91, 91}}
};
```

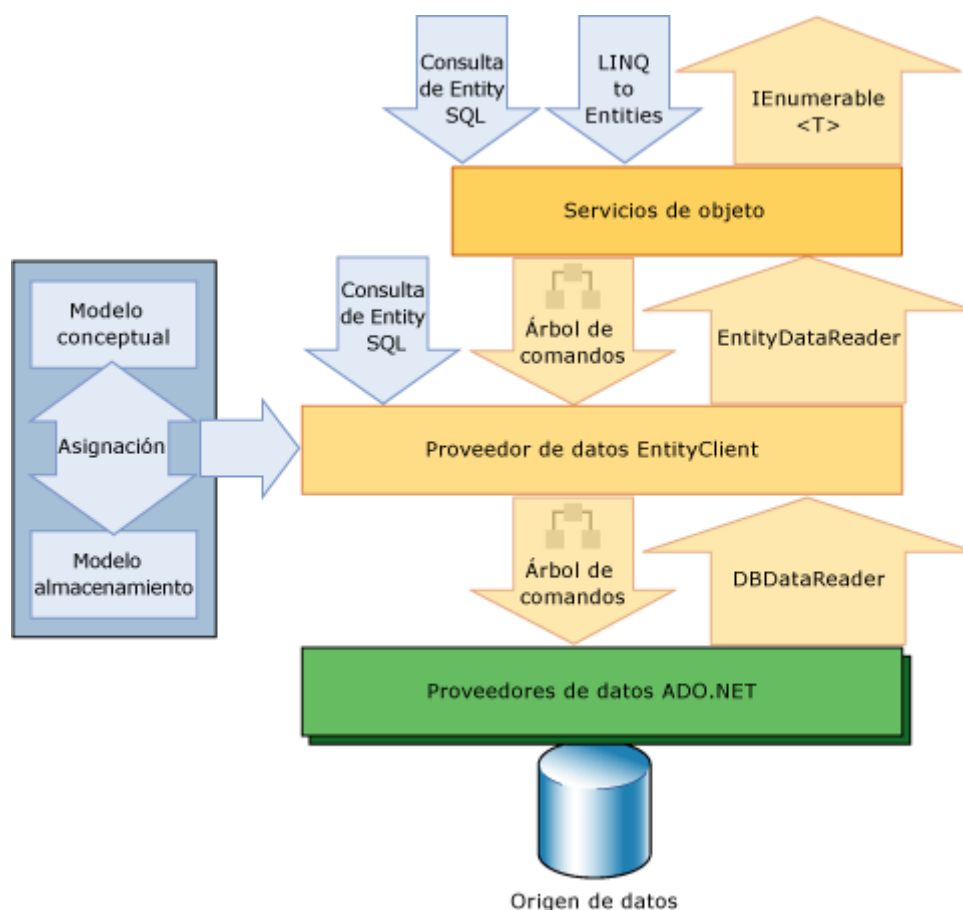
```
IEnumerable<Student> studentQuery =
    from student in students
    where student.Scores[0] > 90
    select student;
```

Se pueden realizar consultas sobre base de datos o documentos XML igualmente

El Entity Framework es un conjunto de tecnologías de ADO.NET que admiten el desarrollo de aplicaciones de software orientadas a datos. Los arquitectos y programadores de aplicaciones orientadas a datos se han enfrentado a la necesidad de lograr dos objetivos muy diferentes. Deben modelar las entidades, las relaciones y la lógica de los problemas

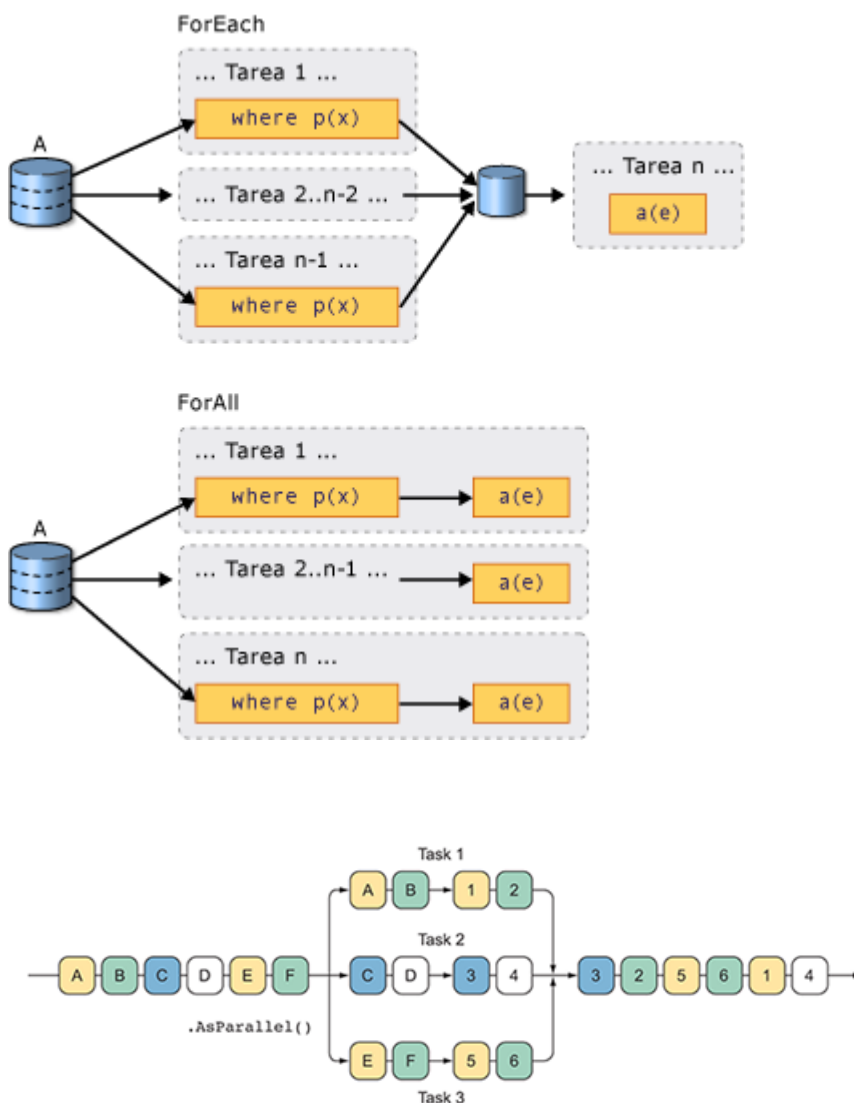
empresariales que resuelven, y también deben trabajar con los motores de datos que se usan para almacenar y recuperar los datos. Los datos pueden abarcar varios sistemas de almacenamiento, cada uno con sus propios protocolos; incluso las aplicaciones que funcionan con un único sistema de almacenamiento deben equilibrar los requisitos del sistema de almacenamiento con respecto a los requisitos de escribir un código de aplicación eficaz y fácil de mantener.

La Entity Framework permite a los desarrolladores trabajar con datos en forma de objetos y propiedades específicos del dominio, como clientes y direcciones de clientes, sin tener que preocuparse por las tablas y columnas de base de datos subyacentes donde se almacenan estos datos. Con Entity Framework, los desarrolladores pueden trabajar en un nivel más alto de abstracción cuando tratan con datos, y pueden crear y mantener aplicaciones orientadas a datos con menos código que en las aplicaciones tradicionales. Dado que el Entity Framework es un componente del .NET Framework, las aplicaciones de Entity Framework se pueden ejecutar en cualquier equipo en el que esté instalado el .NET Framework a partir de la versión 3.5 SP1



1.6 Extensiones para .Net 4.X

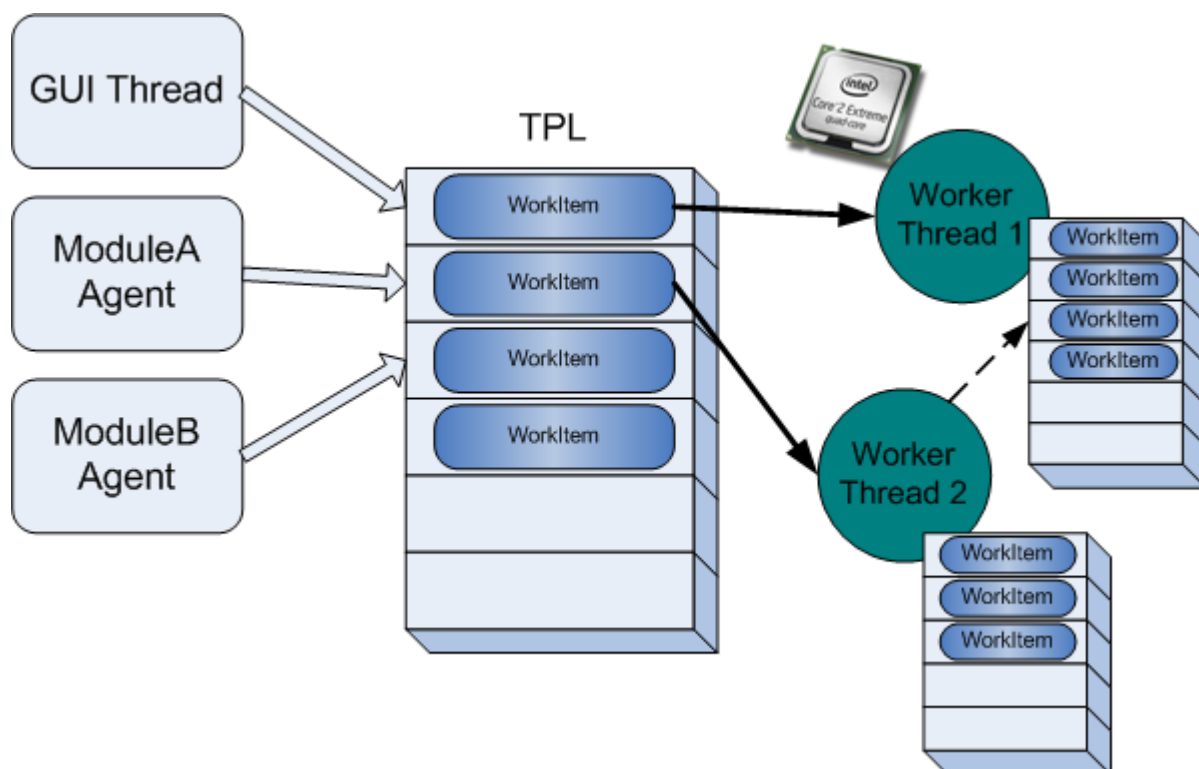
Parallel LINQ (PLINQ) es una implementación en paralelo del patrón Language-Integrated Query (LINQ). PLINQ implementa el conjunto completo de operadores de consulta estándar de LINQ como métodos de extensión para el espacio de nombres System.Linq y tiene operadores adicionales para las operaciones en paralelo. **PLINQ combina la simplicidad y legibilidad de la sintaxis de LINQ con la eficacia de la programación en paralelo.**



La biblioteca TPL (Task Parallel Library, biblioteca de procesamiento paralelo basado en tareas) es un conjunto de API y tipos públicos de los espacios de nombres System.Threading y System.Threading.Tasks. El propósito de la TPL es aumentar la productividad de los desarrolladores **simplificando el proceso de agregar paralelismo y simultaneidad a las aplicaciones**. La TPL escala el grado de simultaneidad de manera dinámica para usar con mayor eficacia todos los procesadores disponibles. Además, la TPL

se encarga de la división del trabajo, la programación de los subprocesos en ThreadPool, la compatibilidad con la cancelación, la administración de los estados y otros detalles de bajo nivel. Al utilizar la TPL, el usuario puede optimizar el rendimiento del código mientras se centra en el trabajo para el que el programa está diseñado.

A partir de .NET Framework 4, la biblioteca TPL es el modo preferido de escribir código multiproceso y en paralelo. Pero no todo el código es adecuado para la paralelización. Por ejemplo, si un bucle realiza solo una cantidad reducida de trabajo en cada iteración o no se ejecuta durante numerosas iteraciones, la sobrecarga de la paralelización puede dar lugar a una ejecución más lenta del código. Además, al igual que cualquier código multiproceso, la paralelización hace que la ejecución del programa sea más compleja. Aunque la TPL simplifica los escenarios de multithreading, recomendamos tener conocimientos básicos sobre conceptos de subprocesamiento, por ejemplo, bloqueos, interbloqueos y condiciones de carrera, para usar la TPL eficazmente.



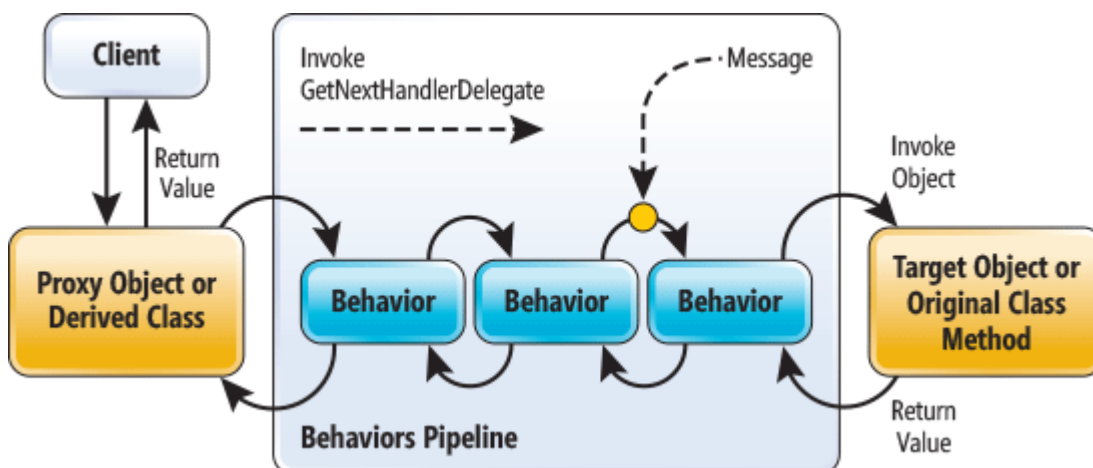
1.7 Extesiones a partir de la versión .NET 4.5

Aplicaciones UWP y API Store: **UWP** es una de las muchas maneras de crear aplicaciones cliente para Windows. Las aplicaciones UWP usan las API de WinRT para **proporcionar características de interfaz de usuario avanzadas y asíncronas eficaces** que son ideales para dispositivos conectados a Internet. La API Store va a proporcionar la posibilidad de añadir estas aplicaciones a la Microsoft Store.

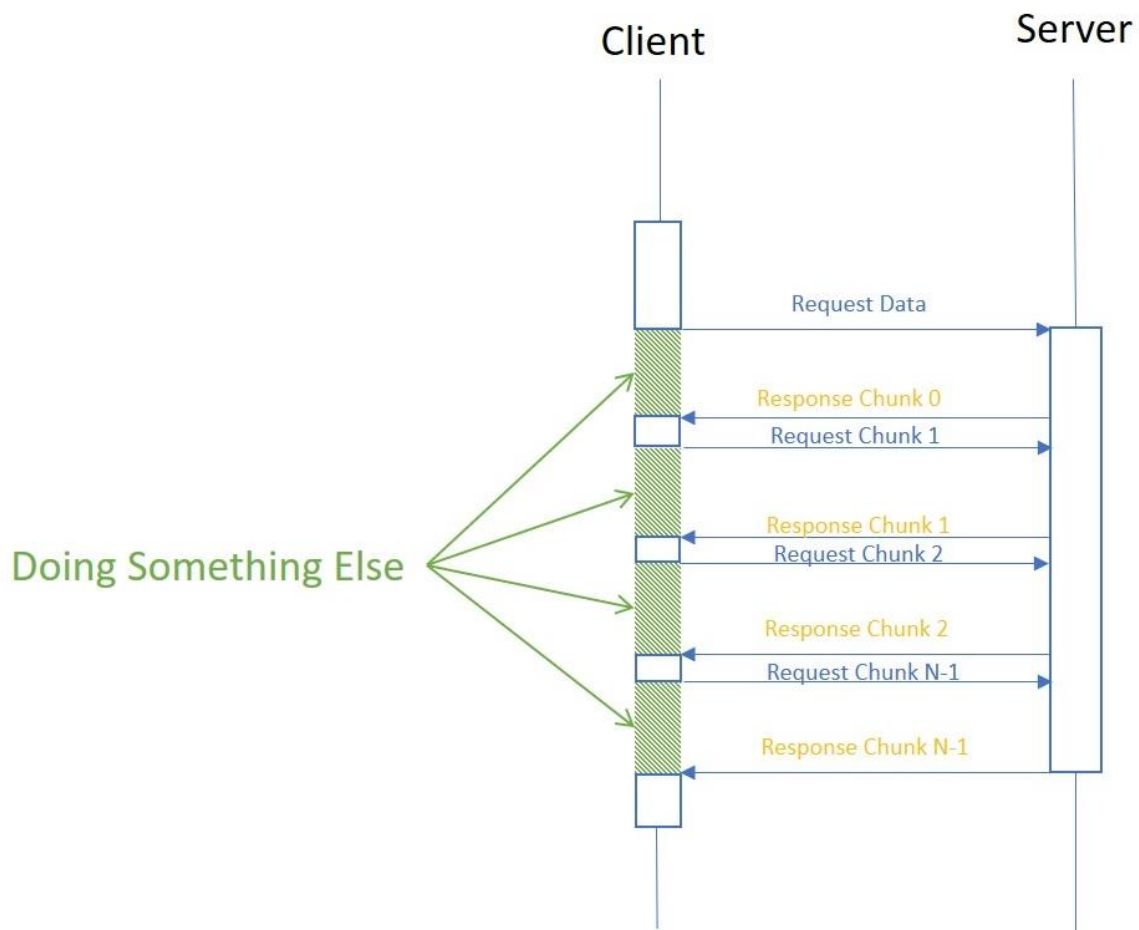
.NET proporciona tres patrones para realizar las operaciones asincrónicas:

- ✓ **Patrón asincrónico basado en tareas (TAP)** , que utiliza un **método único para representar el inicio y la finalización de una operación asincrónica**. TAP Se presentó en .NET Framework 4. Es el enfoque recomendado para la programación asincrónica en .NET. Las palabras clave `async` y `await` en C# y los operadores `Async` y `Await` en Visual Basic agregan compatibilidad de lenguaje para TAP. **Es el modelo que ofrece actualmente Microsoft para el trabajo asíncrono,**
- ✓ El modelo asincrónico basado en eventos (EAP) , que es el patrón heredado basado en eventos para proporcionar el comportamiento asincrónico. Requiere un método con el sufijo `Async`, así como uno o más eventos, tipos de delegado de controlador de eventos y tipos derivados de `EventArgs`. EAP se presentó en .NET Framework 2.0. Ya no se recomienda para nuevo desarrollo.
- ✓ El patrón Modelo de programación asincrónica (APM) (también denominado `AsyncResult`), que es el modelo heredado que usa la interfaz `AsyncResult` para ofrecer un comportamiento asincrónico. En este patrón, las operaciones sincrónicas requieren los métodos `Begin` y `End` (por ejemplo, `BeginWrite` y `EndWrite` para implementar una operación de escritura asincrónica). **Este patrón ya no se recomienda para nuevo desarrollo.**

TAP



Cuando hacemos una petición asíncrona el cliente hace la petición pero no se queda esperando, sigue ejecutando otro código. Cuando recibe la respuesta, entonces procesa esa información recibida desde el servidor.



Asynchronous Sequence Data Pull

Este modelo es de uso extendido en la actualidad en la mayoría de las aplicaciones que se conectan a servidores y requieren de cantidades de datos. Debido a las distancias físicas entre máquinas clientes y servidores y los tiempos de respuesta, para no bloquear las aplicaciones cliente se suele trabajar así. La mayoría de las aplicaciones de vuestros móviles funcionan de esta manera.

2 Microsoft Visual Studio Community. IDE

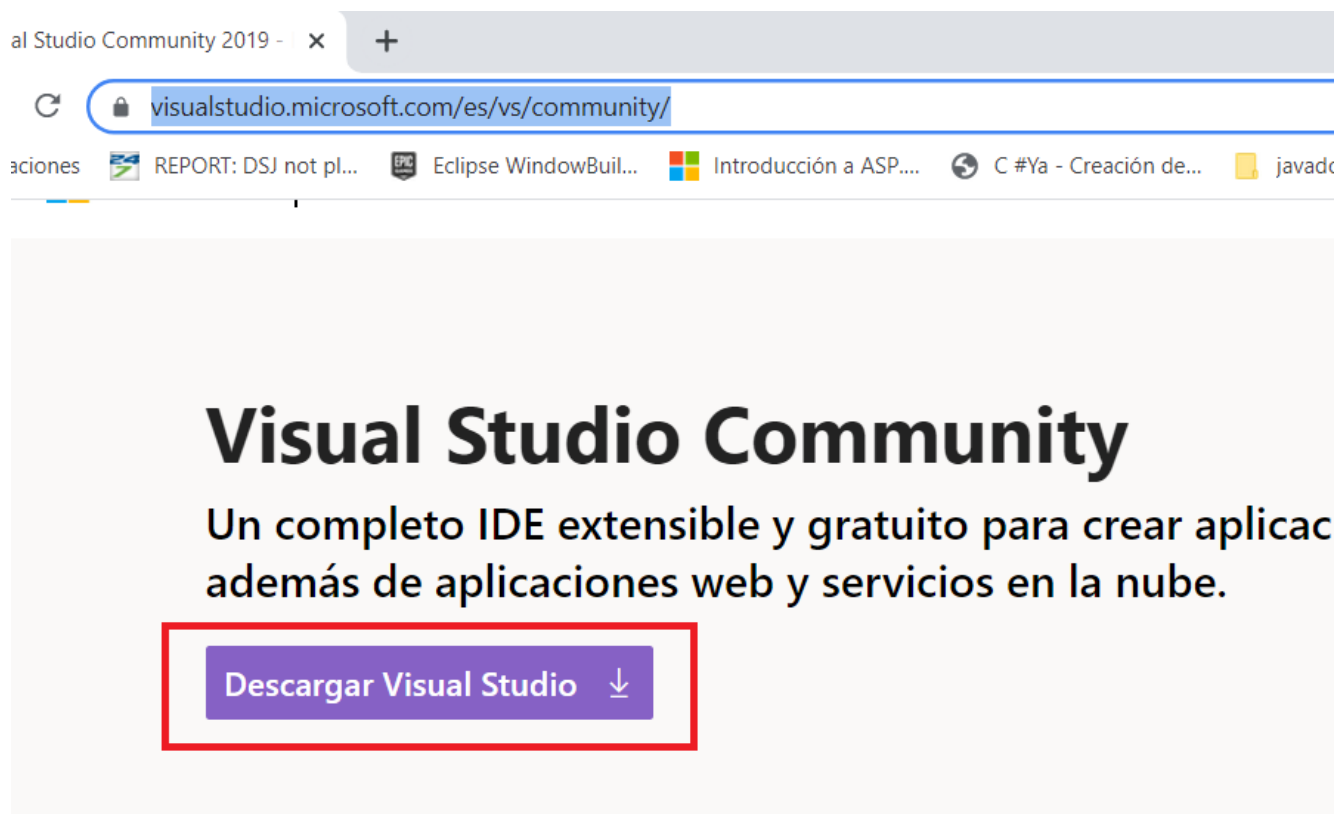
Microsoft Visual Studio es un entorno de desarrollo integrado (IDE, por sus siglas en inglés) para Windows y macOS. Es compatible con múltiples lenguajes de programación, tales como C++, C#, Visual Basic .NET, F#, Java, Python, Ruby y PHP, al igual que entornos de desarrollo web, como ASP.NET MVC, Django, etc., a lo cual hay que sumarle las nuevas capacidades en línea bajo Windows Azure en forma del editor Monaco.


Visual Studio permite a los desarrolladores crear sitios y aplicaciones web, así como servicios web en cualquier entorno compatible con la plataforma .NET (a partir de la versión .NET 2002). Así, se pueden crear aplicaciones que se comuniquen entre estaciones de trabajo, páginas web, dispositivos móviles, dispositivos embebidos y videoconsolas, entre otros.

2.1 Instalación

Podéis descargar el IDE de este enlace o del aula virtual:

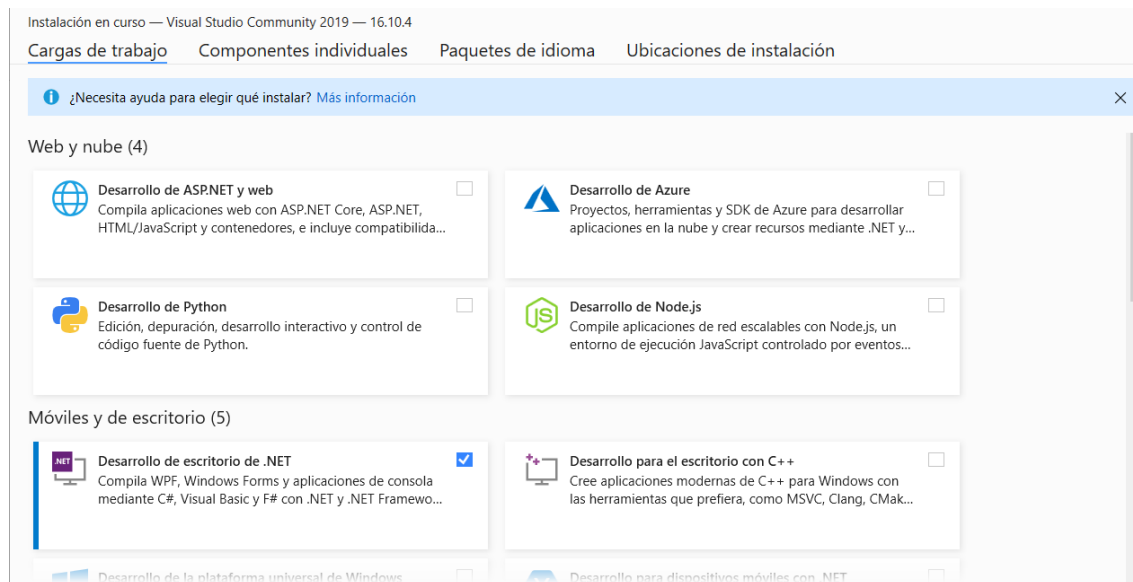
<https://visualstudio.microsoft.com/es/vs/community/>



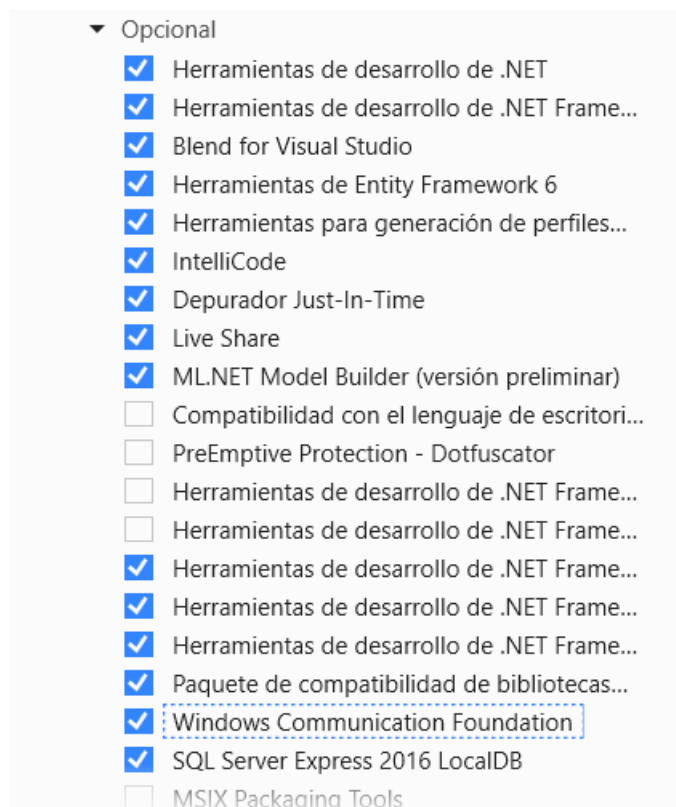
 vs_community__1632842981.1627925327.exe

Pasos para la instalación:

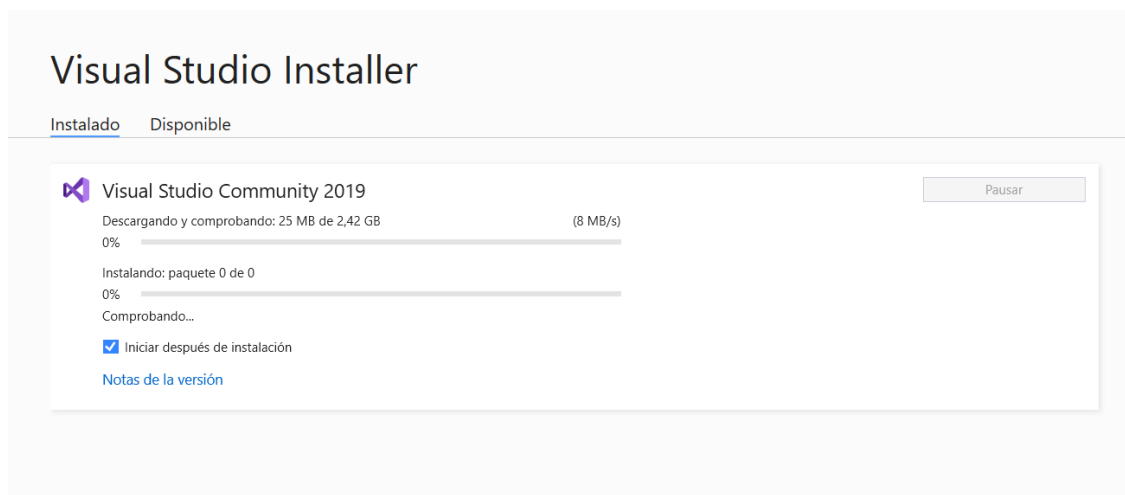
1. Ejecutar el fichero .exe descargado
2. Elegir el paquete de instalación para escritorio de C#



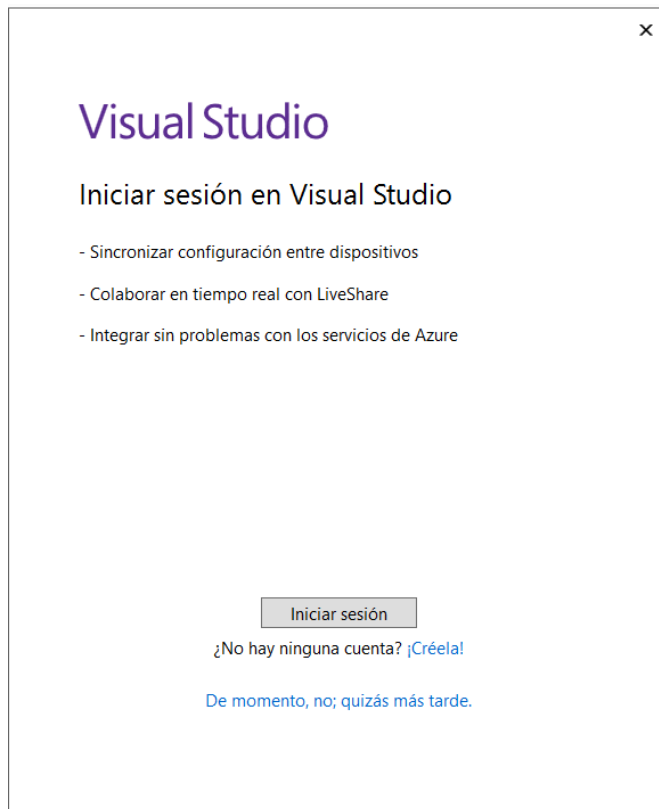
3. Elegir las siguientes librerías:



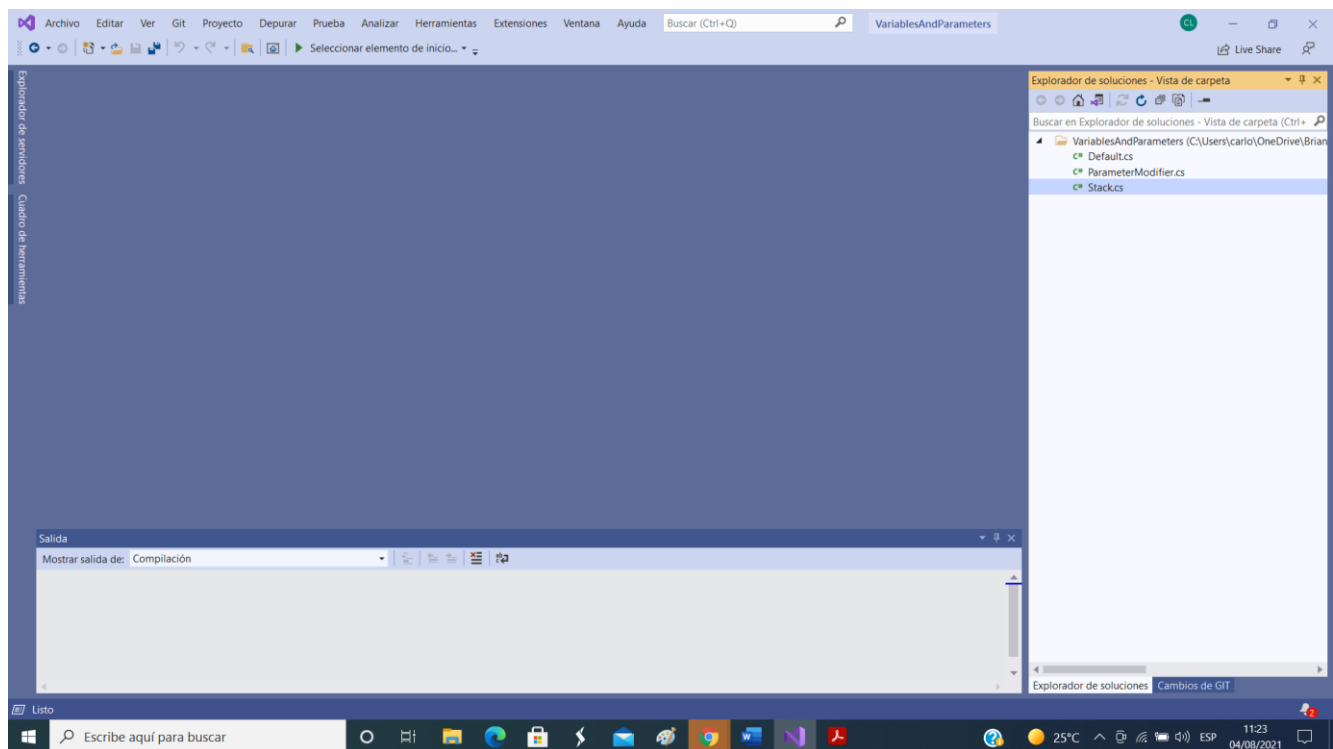
4. Esperar a que termine la instalación:



5. Necesitareis una cuenta de Microsoft para conectaros y hacer login. Pulsad en iniciar sesión. Tenéis dos opciones:
- Usar la cuenta de vuestro ordenador.
 - Usar la cuenta que os proporciona la junta para Teams.



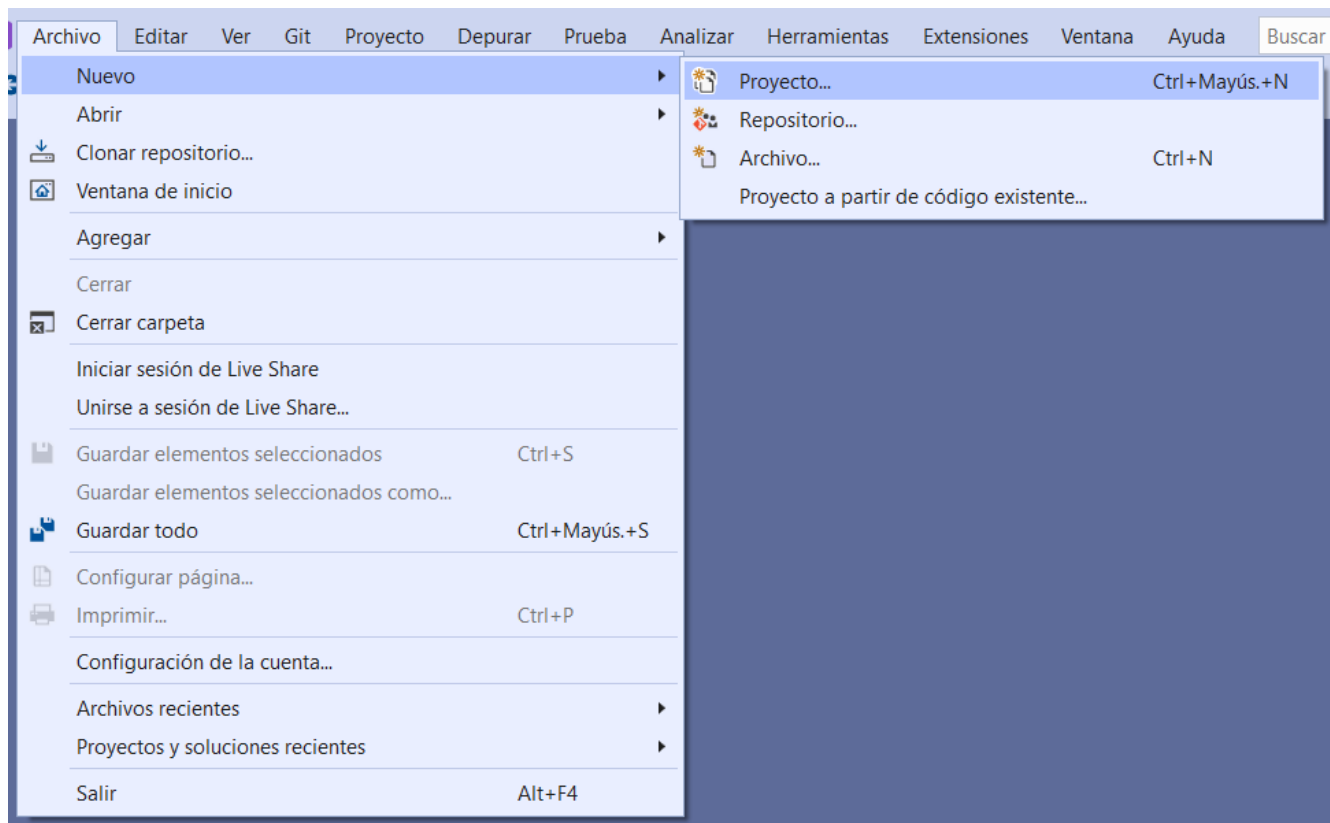
6. Tras realizar el login se os abrirá el entorno de desarrollo



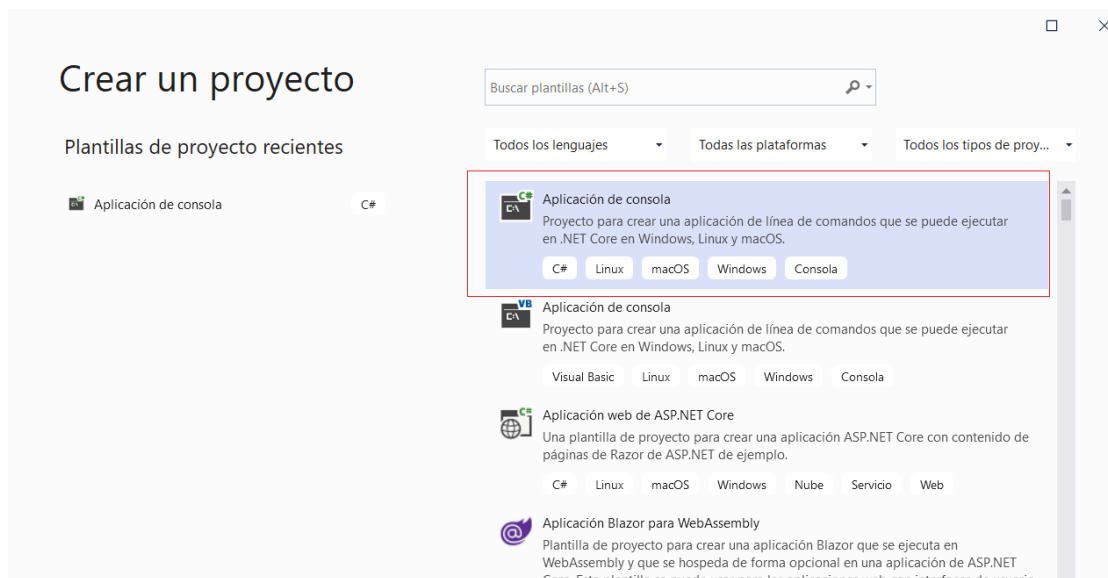
2.2 Primer proyecto en C#

Crearemos un nuevo proyecto MiPrimerProyecto siguiendo los siguientes pasos:

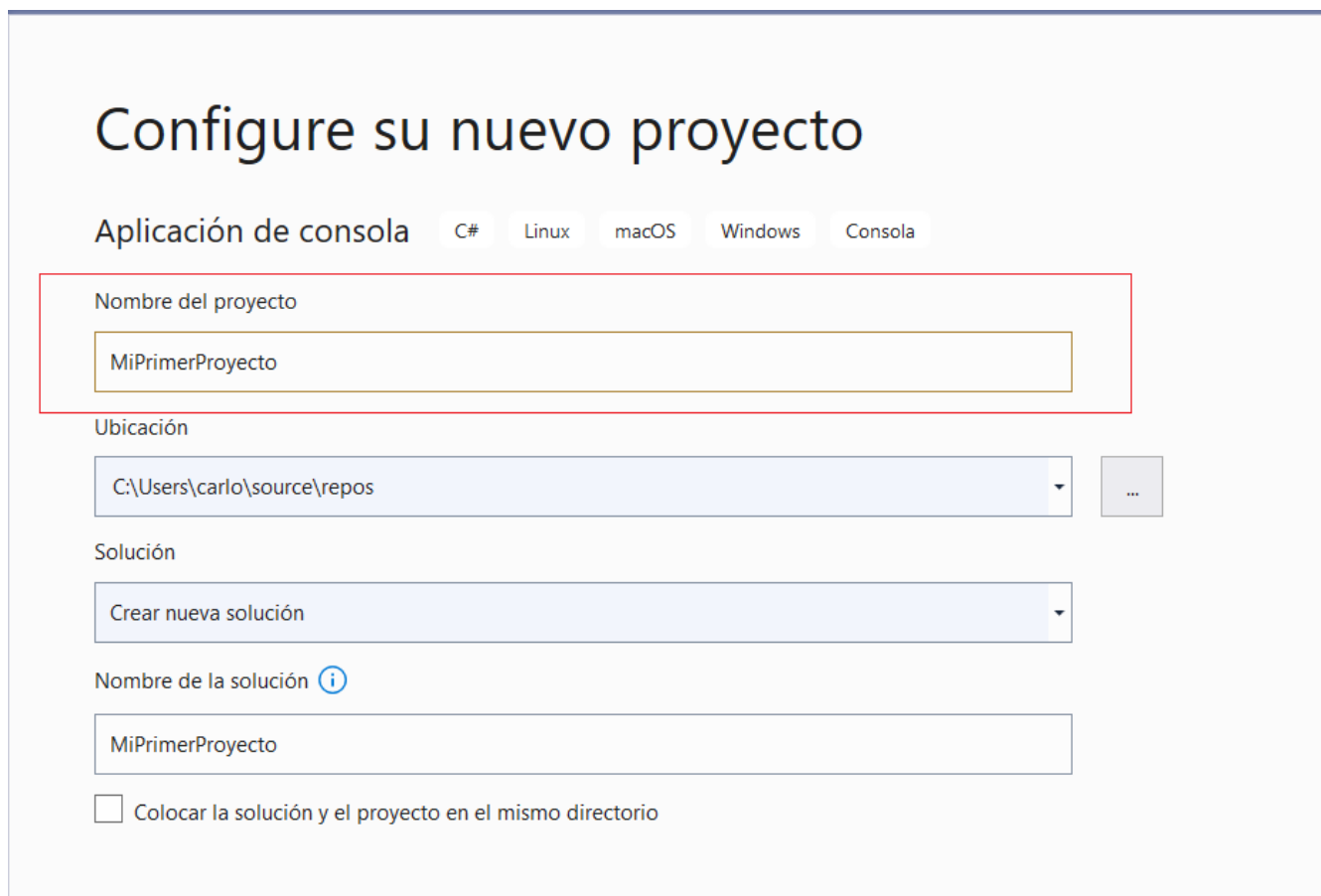
1. En vuestro Visual Studio ir al menú Archivo->Nuevo->Proyecto



2. Seleccionar Aplicación de consola para C#.



3. Nombrarlo como mi primer proyecto



4. Elegimos compatibilidad .NET Core 3.1 para los primeros proyectos y pulsad en el botón crear.

Información adicional

Aplicación de consola

C#

Linux

macOS

Windows

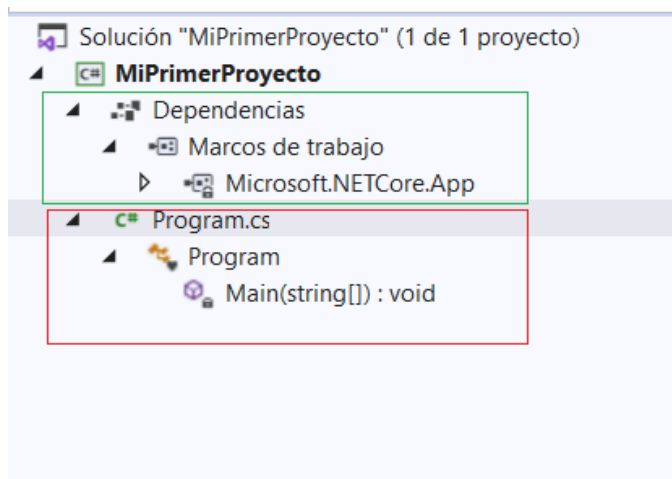
Consola

Plataforma de destino ⓘ

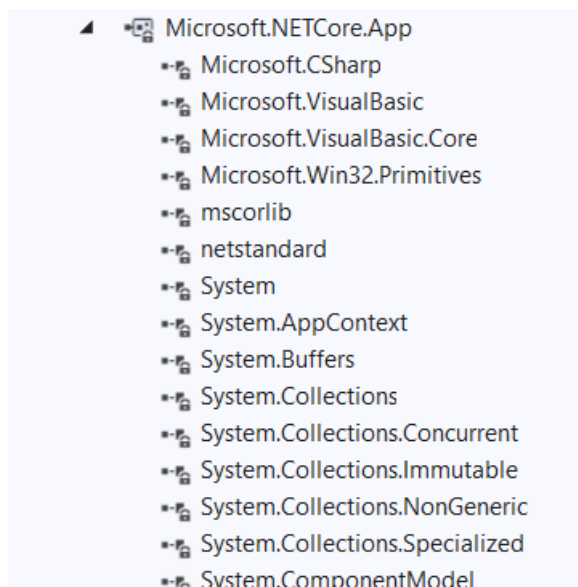
.NET Core 3.1 (Compatibilidad a largo plazo)

2.2.1 La estructura del proyecto

Podéis ver dos partes principales en vuestra estructura de proyecto. Las dependencias y el código, donde en este caso sólo tenemos el programa principal.



Si abris las dependencias os encontrareis todas las librerías del .NET Core añadidas a vuestro proyecto.



El programa principal, en la parte de código es el punto de entrada a la aplicación:

El código del programa principal es el siguiente:

```
using System;

namespace MiPrimerProyecto
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

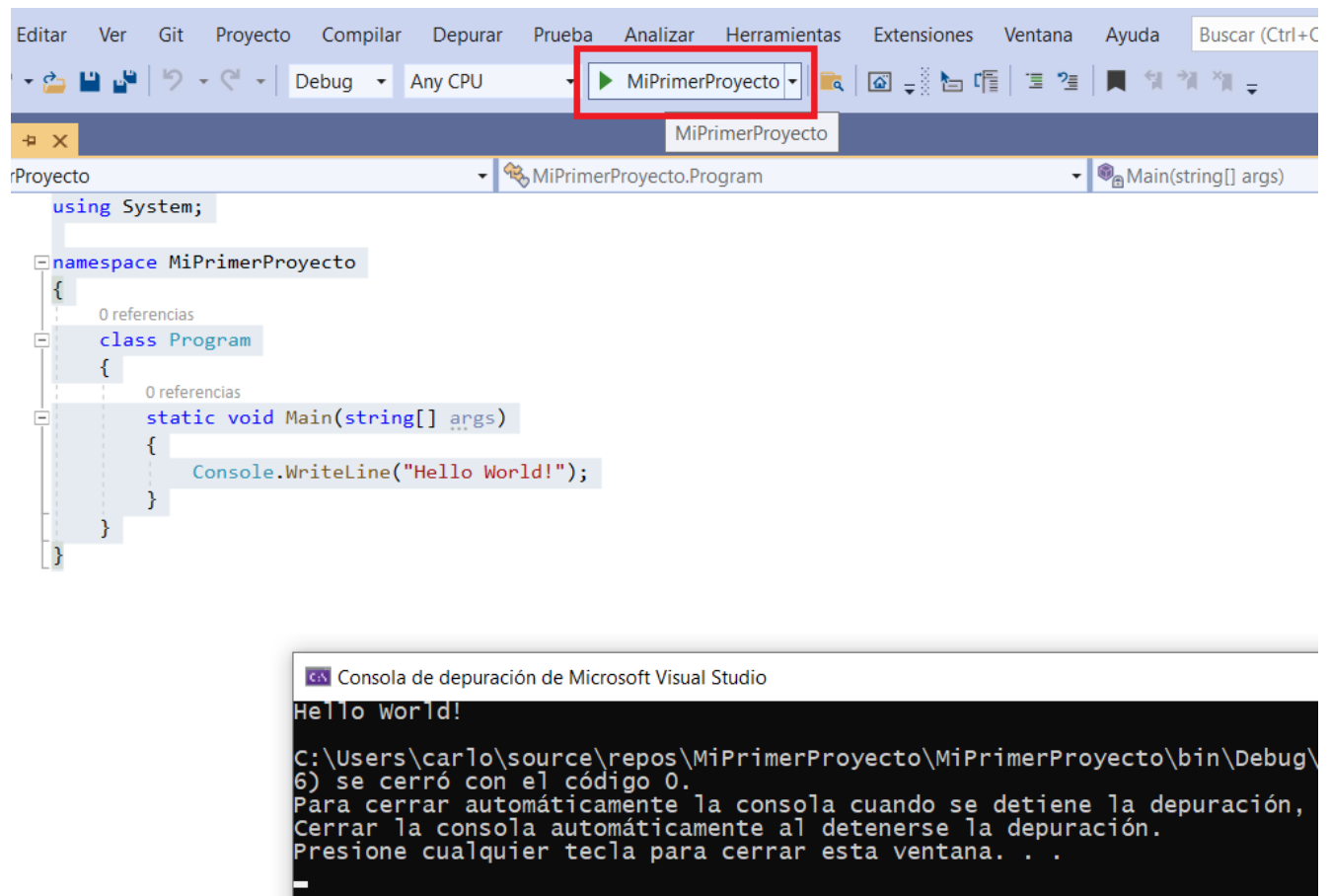
Con `using System` estamos importando todas las clases y subclases del espacio de nombre `System`.

El **namespace** es una manera de asignar un espacio único a mi clase. Para quien sepa Java es como hacer paquetes en Java. De esta manera puedo repetir el nombre de una clase en mi proyecto y diferenciarla de una clase con el mismo nombre que esté en otro namespace o espacio de nombre.

La función estática `Main` es el punto de entrada a la aplicación de consola.

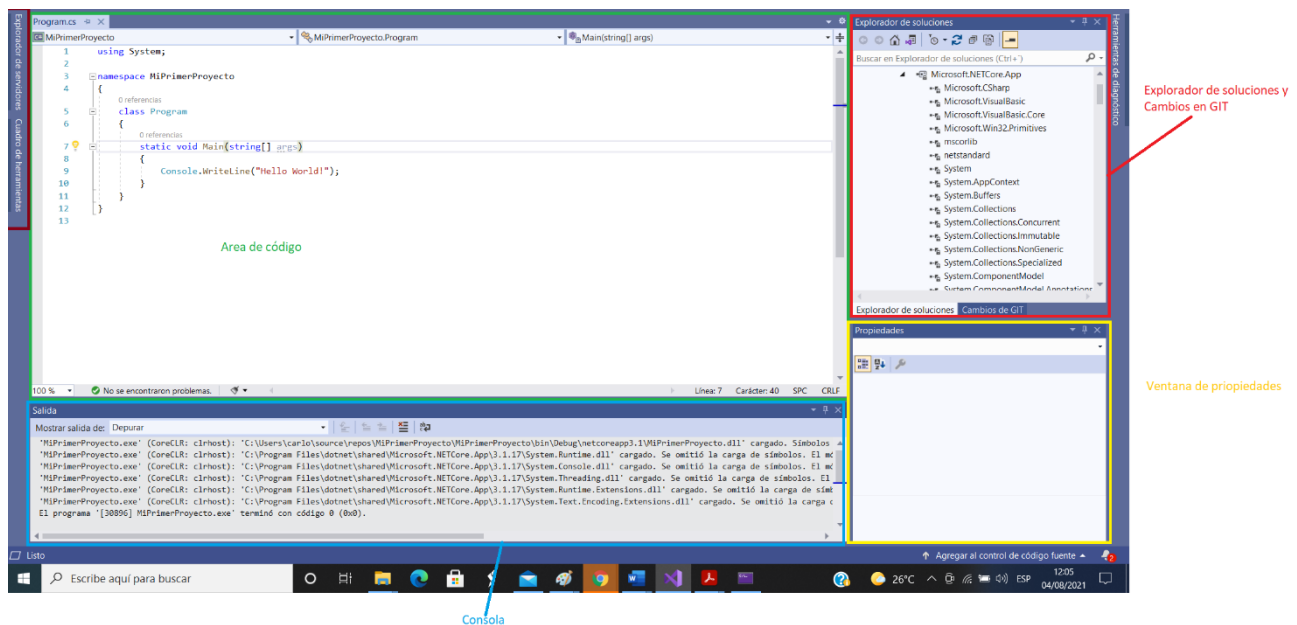
Console.WriteLine es un método estático de la clase Console para escribir en la consola.

Ejecutar el programa y comprobar que funciona.

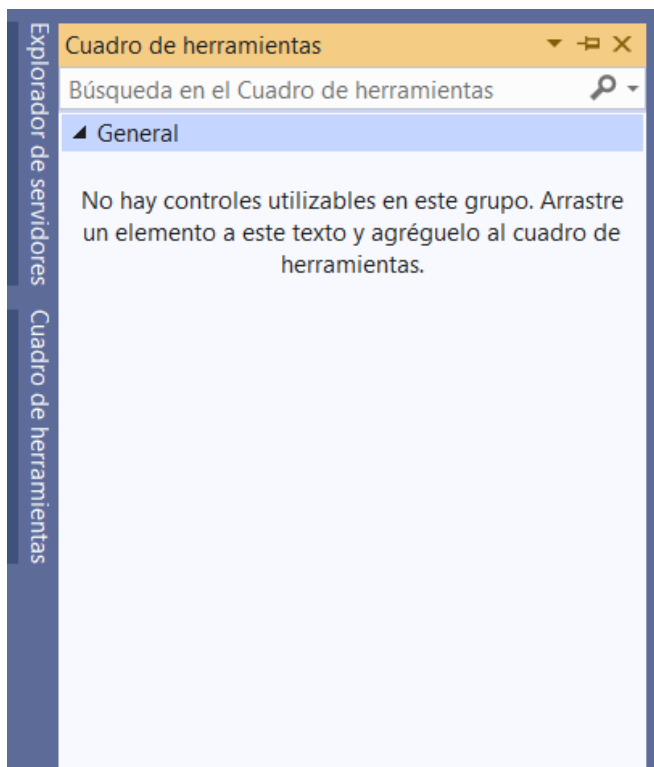


2.2.2 Areas en el IDE

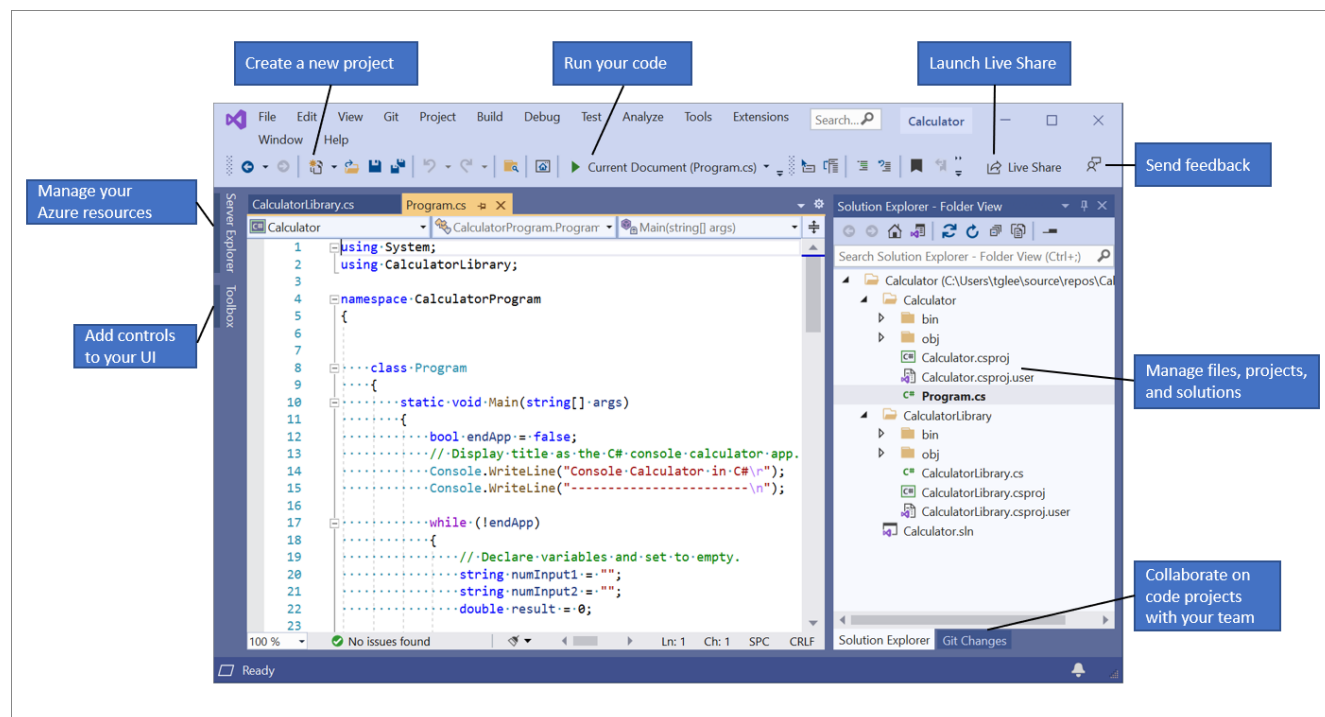
En la siguiente imagen podéis ver las partes en la que se divide el IDE



- Área de código
- Explorador de soluciones y Cambios en Git si hemos activado el control de versiones.
- Área de propiedades
- Área de consola
- En la esquina superior izquierda tenemos dos pestañas:
 - Cuadro de herramientas: para añadir controles a formularios
 - Explorador de Servidores: para manejar los servidores a los que se conecte nuestra aplicación

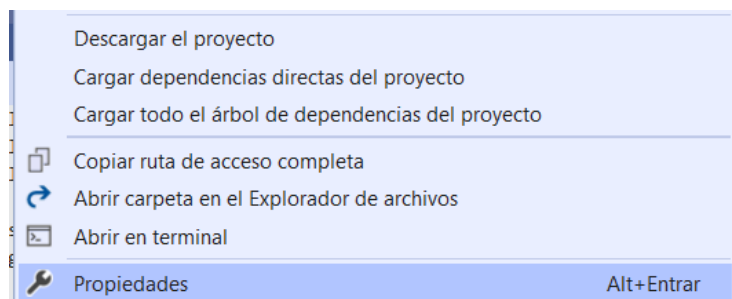
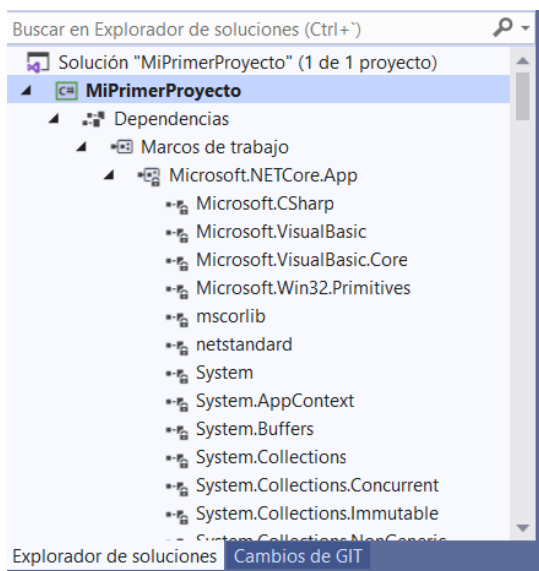


En la siguiente imagen están marcados los botones de la barra de herramientas para operativa básica

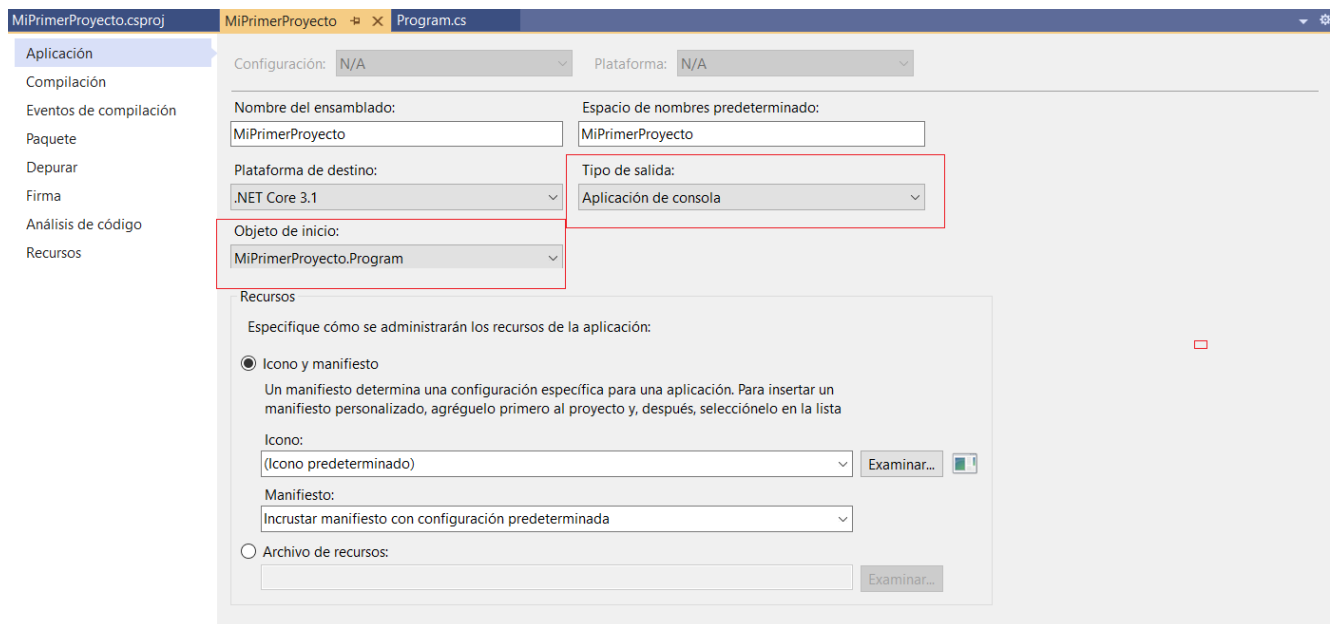


2.2.3 Configurando el proyecto

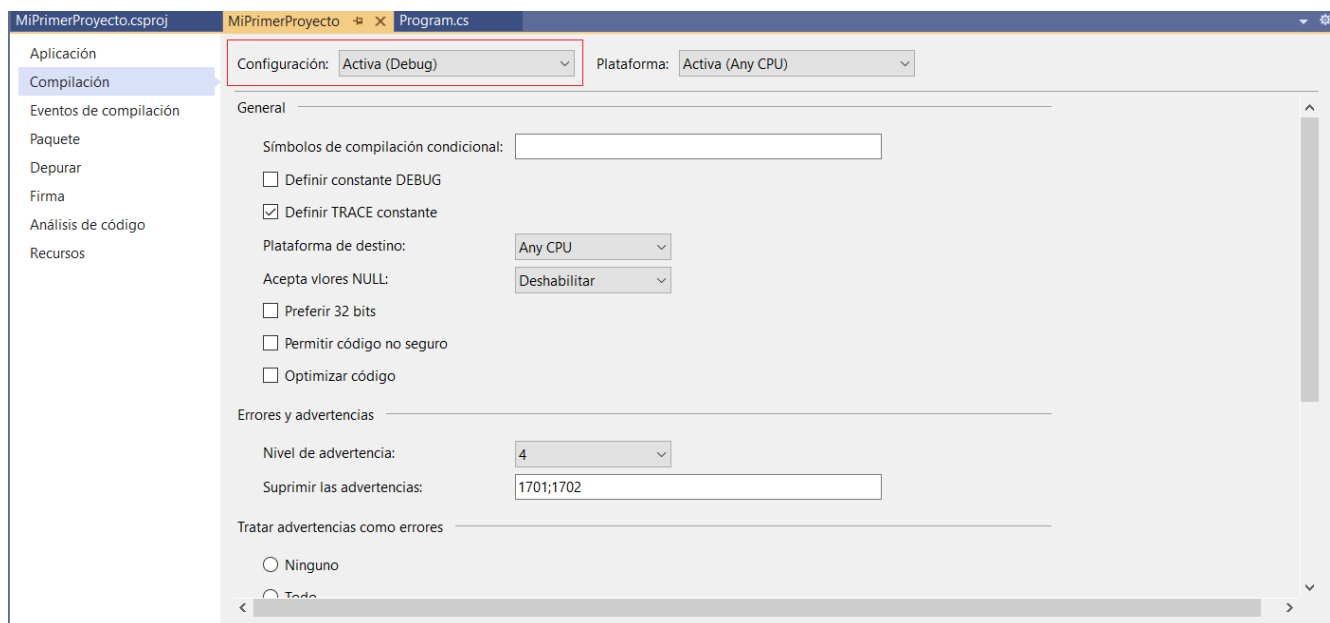
En el explorador de soluciones seleccionar el proyecto y hacer botón derecho con el ratón. Pulsad entonces en propiedades.



Colocar como objeto de Inicio a Program para asegurarnos de que es el punto de entrada.



La depuración debe estar activa igualmente, en la pestaña compilación.



3 Lenguaje de programación C#

3.1 Introducción

"C#" (pronunciado 'si sharp' en inglés) es un lenguaje de programación multiparadigma desarrollado y estandarizado por la empresa Microsoft como parte de su plataforma .NET, que después fue aprobado como un estándar por la ECMA (ECMA-334) e ISO (ISO/IEC 23270). C# es uno de los lenguajes de programación diseñados para la infraestructura de lenguaje común.

Su sintaxis básica deriva de C/C++ y utiliza el modelo de objetos de la plataforma .NET, similar al de Java, aunque incluye mejoras derivadas de otros lenguajes.

El nombre C Sharp fue inspirado por el signo #, el cual se lee como sharp en inglés para notación musical. Es un juego de palabras, pues "C#" significa, musicalmente hablando, "do sostenido", donde el símbolo # indica que una nota (en este caso do, representada por C) debe ser un semitono más alta. Esto es una metáfora de la superioridad de C# sobre su antecesor C++ y a su vez hace alusión a la misma metáfora que se ideó para dar nombre a C++.1 Además, el símbolo # puede ser imaginado como la unión de cuatro símbolos +, continuando así con el sentido de progresión de los lenguajes C.

C# tiene sus raíces en la familia de lenguajes C, y a los programadores de C, C++, Java y JavaScript les resultará familiar inmediatamente. Este paseo proporciona información general de los principales componentes del lenguaje en C# 8 y versiones anteriores. Si quiere explorar el lenguaje a través de ejemplos interactivos, pruebe los tutoriales de introducción a C#.

C# es un lenguaje de programación orientado a componentes, orientado a objetos. C# proporciona construcciones de lenguaje para admitir directamente estos conceptos, por lo que se trata de un lenguaje natural en el que crear y usar componentes de software. Desde su origen, C# ha agregado características para admitir nuevas cargas de trabajo y prácticas de diseño de software emergentes.

Además de los servicios en tiempo de ejecución, .NET también incluye amplias bibliotecas, que admiten muchas cargas de trabajo diferentes. Se organizan en espacios de nombres que proporcionan una gran variedad de funciones útiles para todo, desde la entrada y salida de archivos, la manipulación de cadenas y el análisis de XML hasta los marcos de aplicaciones web y los controles de Windows Forms. En una aplicación de C# típica se usa la biblioteca de clases de .NET de forma extensa para controlar tareas comunes de infraestructura.

3.2 Instrucciones en c#

3.2.1 Tipos de instrucciones

En la tabla siguiente se muestran los distintos tipos de instrucciones de C# y sus palabras clave asociadas, con vínculos a temas que incluyen más información:

TIPOS DE INSTRUCCIONES

Categoría	Palabras clave de C# / notas
Instrucciones de declaración	Una instrucción de declaración introduce una variable o constante nueva. Una declaración de variable puede asignar opcionalmente un valor a la variable. En una declaración de constante, se requiere la asignación.
Instrucciones de expresión	Las instrucciones de expresión que calculan un valor deben almacenar el valor en una variable.
Instrucciones de selección	Las instrucciones de selección permiten crear bifurcaciones a diferentes secciones de código, en función de una o varias condiciones especificadas. Son las siguientes <ul style="list-style-type: none">• If• If-else• Switch- case
Instrucciones de iteración	Las instrucciones de iteración permiten recorrer en bucle colecciones, como matrices, o realizar el mismo conjunto de instrucciones repetidas veces hasta que se cumpla una condición especificada. Son las siguientes <ul style="list-style-type: none">• Do• For• Foreach• While
Instrucciones de salto	Las instrucciones de salto transfieren el control a otra sección de código. Son las siguientes: <ul style="list-style-type: none">• break• continue• default• goto• return• yield
Instrucciones para el control de excepciones	Las instrucciones para el control de excepciones permiten recuperarse correctamente de condiciones excepcionales producidas en tiempo de ejecución. Son las siguientes: <ul style="list-style-type: none">• throw• try-catch• try-finally• try-catch-finally
Checked y	Las instrucciones checked y unchecked permiten especificar si las

TIPOS DE INSTRUCCIONES

Categoría	Palabras clave de C# / notas
unchecked	operaciones numéricas pueden producir un desbordamiento cuando el resultado se almacena en una variable que es demasiado pequeña para contener el valor resultante.
Instrucción await	Si marca un método con el modificador async , puede usar el operador await en el método. Cuando el control alcanza una expresión await en el método asíncrono, el control se devuelve al autor de llamada y el progreso del método se suspende hasta que se completa la tarea esperada. Cuando se completa la tarea, la ejecución puede reanudarse en el método.
Instrucción yield return	Un iterador realiza una iteración personalizada en una colección, como una lista o matriz. Un iterador utiliza la instrucción yield return para devolver cada elemento de uno en uno. Cuando se alcanza una instrucción yield return, se recuerda la ubicación actual en el código. La ejecución se reinicia desde esa ubicación la próxima vez que se llama el iterador.
Instrucción fixed	La instrucción fixed impide que el recolector de elementos no utilizados cambie la ubicación de una variable móvil.
Instrucción lock	La instrucción lock permite limitar el acceso a bloques de código a un solo subproceso de cada vez.
Instrucciones con etiqueta	Puede asignar una etiqueta a una instrucción y, después, usar la palabra clave goto para saltar a la instrucción con etiqueta. (
Instrucción vacía	La instrucción vacía consta únicamente de un punto y coma. No hace nada y se puede usar en lugares en los que se requiere una instrucción, pero no es necesario realizar ninguna acción.

3.2.2 Formato de las instrucciones

Muy similar a C, Javascript o Java , para el que este familiarizado con estos lenguajes.. Tenemos dos tipos básicos de formato de instrucción:

Instrucciones simples: Acaba siempre en “;” y contienen expresiones, operadores, llamadas a métodos y asignaciones.

Ejemplo:

```
cadena = "¿Qué es lo que dice?\rÉl dice \"hola\"";
```

ó

```
area = 3.14 * (radius * radius);
```

Instrucciones de bloque: un conjunto de instrucciones simples entre llaves, ejecutadas en secuencia.

```
{  
    Console.Write(n);  
    n++;  
}
```

3.3 Variables en C#

El nombre de las variables:

- Empiezan por una letra.
- Pueden estar compuestas por caracteres alfanuméricos o el carácter subrayado “_”.
- Se distinguen mayúsculas de minúsculas
- No se pueden usar las palabras reservadas del lenguaje (if,switch, void, ...) a no se que se utilice el caracter arroba “@” precediendo al la variable. Por ejemplo.: @if sí podría ser una variable.

```
int mi_variable_entera=1;
```

3.3.1 Constantes

Las constantes

Valores numéricos o cadenas que no serán modificados durante el funcionamiento de la aplicación. Se usa la palabra predefinida const.

```
const int ValorMax = 100;  
const string Mensaje="Muy grande";  
...  
If (resultado>Valormax)  
    Console.WriteLine(Mensaje);
```

3.4 Lectura y escritura en consola

Para escribir en consola podemos usar los métodos Write de la clase Console como

Console.WriteLine(string)

Console.Write(cualquier tipo)

Para dar formato básico y referenciar variables en WriteLine podemos usar las llaves y el número de la variable en la lista empezando por cero.

```
int a =2;
int b =3;
int res= a+b;
Console.WriteLine("El resultado de sumar {0} e {1} es {2}", a, b, res);
```

Para leer de consola usaremos:

- Para cadenas
String cadena =Console.ReadLine();
- Para carácter individual
char c = Console.Read()

3.5 *Tipos en C Sharp*

Distinguimos dos grandes conjuntos de tipos en C#:

- Tipos por valor: donde la variable almacena el valor directamente. Son los tipos básicos en c#.
- Tipos por referencia: la variable almacena una dirección de memoria donde esta almacenado el tipo. Estructuras y clases predefinidas (de librería) y creadas por el usuario pertenecen a este segundo rango

3.5.1 Tipos numéricos (por valor)

Enteros:

- Con signo
 - **sbyte** (8 bits)
 - **short** (16 bits)
 - **int** (32 bits)
 - **long** (64 bits)
- Sin signo
 - **byte** (8 bits)
 - **ushort** (16 bits)
 - **uint** (32 bits)

- **ulong** (64 bits)

Tipos numéricos decimales (todos son con signo).

- **float** (4 bytes = 32 bits)
- **double** (8 bytes = 64 bits)
- **decimal** (16 bytes = 128 bits)

Estos tipos permiten conversión implícita de menor tamaño a mayor tamaño. En la conversión implícita deben tener o no tener signo, es decir, no puedo convertir un tipo con signo a un tipo sin signo implícitamente. **No se puede realizar conversión explícita de float y double a decimal.**

Para cada tipo simple tenemos su versión .Net con una clase. Por ejemplo: System.Float o System.Int o System.Decimal.

La conversión explícita se realiza con casteo como veremos en el siguiente ejemplo.

```
float fDecimal = 678.89F; // 4 bytes
```

```
decimal deDecimal = 23523523.342M; //16 bytes
```

```
// Para decimal es necesaria la conversion explícita
deDecimal = (decimal)fDecimal;
```

```
using System;
```

```
namespace TiposCsharp
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            sbyte sbByte=-50;
            byte bByte = 245;
            ushort usEnteroCorto = 4566;
            short sEnteroCorto = 31000;
            int iEntero = 3444455;
            long lEnteroLargo = 4543346436;

            lEnteroLargo = bByte;

            sEnteroCorto = sbByte;

            iEntero = usEnteroCorto;
```

```
float fDecimal = 678.89F; // 4 bytes
```

```
double dDecimal = 3634634.789; // 8 bytes
```

```
decimal deDecimal = 23523523.342M; //16 bytes
```

```
//Conversion implícita
dDecimal = fDecimal;
```

```
// Para decimal es necesaria la conversion explicita
deDecimal = (decimal)fDecimal;
```

```
}
}
}
```

3.5.2 Tipos de carácter y cadena

- **char** (2 bytes = 16 bits): código unicode donde se pueden representar hasta 256 caracteres, de los cuales los primeros 128 son los mismos que el juego de caracteres ASCII.
- **String**: es el objeto que utilizamos para representar cadenas.

```
Char Car1 = "A";
String Cad1 = "AulaWeb.com.es";
```

Secuencias de escape

\' = comillas simples	\a = alerta	\r = retorno de carro
\" = comillas dobles	\b = backspace	\t = tabulación horizontal
\\ = barra invertida	\f = salto de página	\v = tabulación vertical
\0 = carácter nulo	\n = salto de línea	

Otra alternativa a la utilización de algunas secuencias de escape la encontramos en el **operador "@"**, que vendría a ser algo parecido a instrucción <pre> de html.

Por ejemplo:

```
cadena = "¿Qué es lo que dice?\rÉl dice \"hola\"";
cadena = @"¿Qué es lo que dice?
Él dice ""hola""";
```

Podéis ver los usos básicos como asignación y concatenación en los siguientes ejemplos. Observar como puede concatenar cadenas y caracteres.

```
"Caracter 1:" +caracter1
```

```

using System;

namespace TiposParaCaracteres
{
    class Program
    {
        {"Character 1:" +character1
        static void Main(string[] args)
        {
            char character1 = 'A';
            String miCadena = "Mi Cadena \\'\'";
            String miCadena2 = "MiCadena 2\\\'\'\'";

            String concat = miCadena + miCadena2;
            character1 = 'B';
            Console.WriteLine("Character 1: {0}", character1);
            Console.WriteLine("Cadena Concat: " + concat);
        }
    }
}

```

3.5.3 Booleanos

La palabra clave de tipo bool es un alias para el tipo de estructura de .NET System.Boolean que representa un valor booleano que puede ser true o false.

Para realizar operaciones lógicas con valores del tipo bool, use operadores lógicos booleanos. El tipo bool es el tipo de resultado de los operadores de comparación e igualdad. Una expresión de tipo bool puede ser: una expresión condicional de control en las instrucciones if, do, while y for, así como en el operador condicional ?.

El valor predeterminado del tipo bool es false.

```

Bool cierto= true;

Boolean EsCierto = true;

Boolean Esfalso = false;

```

Para el NOT tenemos el operador booleano "!". Los operadores tradicionales booleanos || (Or) y &&(And) evalúa el primero operador, y el segundo si hace falta. Por ejemplo:

false && (i<5)

i<5 no se evaluaría por que el resultado de la operación lógica va a ser false, porque el primer operador es false y en un and lógico si el primer operador es false el resultado final es false.

Los operadores booleanos | (Or) y &(and) evalúa los dos operadores independientemente del resultado del primero. Además admiten la opción de evaluar valores nulos, null, un tercer valor de verdad. Existe un nuevo operador en este escenario que es el Or Exclusivo ^.

```
Console.WriteLine("valor de resultado booleano Or Exclusivo nullable:" + (true ^ true));
```

Recordar que para que el Or Exclusivo sea verdadero, devuelva true, sólo un operando debe ser True.

Con el operador interrogación puedo conseguir que el tipo bool sea nullable, que admita nulos, `bool?`.

```
bool? noEsNull = true;
bool? esnull = null;
bool? resultado2 = noEsNull | esnull;

Console.WriteLine("valor de resultado booleano or nullable:" + (noEsNull |
esnull));

Console.WriteLine("valor de resultado booleano and nullable:" + (noEsNull & esnull));
```

Al ejecutar el ejemplo obtendremos que

True | null es True

Y True & null es null.

Hay nuevos valores de verdad, un tercer estado en estos booleanos nullables.

Ejecución en consola:

```
valor de resultado booleano :True
valor de resultado booleano or nullable:True
valor de resultado booleano and nullable:
valor de resultado booleano Or Exclusivo nullable:False
```

El código completo del ejemplo:

```
using System;

namespace Booleanos
{
    class Program
    {
        static void Main(string[] args)
        {
            bool verdadero = true;

            bool falso = false;
            bool resultado = verdadero || falso;
        }
    }
}
```

```

        Console.WriteLine("valor de resultado booleano :{0}", resultado);

        bool? noEsNull = true;
        bool? esnull= null;
        bool? resultado2 = noEsNull | esnull;

        Console.WriteLine("valor de resultado booleano or nullable:" + (noEsNull |
esnull));

        Console.WriteLine("valor de resultado booleano and nullable:" + (noEsNull &
esnull));

        Console.WriteLine("valor de resultado booleano Or Exclusivo nullable:" + (true ^ true));

    }
}
}

```

Tabla de verdad para booleanos que permite NULL

x	y	x e y	x y
true	true	true	true
true	False	false	true
true	null	null	true
False	true	False	true
False	False	False	False
False	null	False	nulo
null	true	null	true
null	False	False	nulo
null	null	null	null

3.5.4 Los tipos Nullables.

para asignar a una variable el valor **Null** (nulo) sólo tenemos que utilizar el carácter “?” después del tipo de variable. Por ejemplo podemos hacer la recuperación de ciertos datos de una base de datos y cuando recuperamos esos datos nos damos cuenta que no tienen ningún valor, por ejemplo campos booleanos, enteros. Para estos casos puede ser interesante no asignar ningún valor a la variable, o mejor dicho asignar el valor **Null** a dicha variable. Luego para saber si una variable tiene valor podemos utilizar la función **Hasvalue**. Si intentáramos utilizar el valor de una variable que no tiene valor (null) nos saltaría una excepción, por lo que a las variables que permitimos tener valor nulo siempre deremos tener la precaución de comprobar antes si tienen algún valor asignado. Lo hemos visto antes con los booleanos

3.5.5 Tipo Object (Por referencia)

En una variable tipo Object podemos almacenar cualquier cosa. en realidad la variable almacena la dirección de memoria donde se encuentra el valor de la variable. Object es la clase base de la que heredan todas las clases de C#, tanto las que importamos de bibliotecas como las que creamos nosotros mismos

3.5.6 Tipo dynamic

A veces puede ocurrir que sólo podemos conocer el tipo de una variable en tiempo de ejecución y no cuando estamos programando o diseñando la aplicación, para este tipo de casos tenemos la palabra reservada **dynamic** para utilizar junto con la variable afectada y la cual no sabemos de que tipo va a ser.

```
public static dynamic operacion (dynamic op1, dynamic op2) {  
    return op1 + op2;  
}
```

3.5.7 Tipos indefinidos

Cuando queremos que sea el compilador el que resuelva un tipo de una variable, además de poder usar Dynamic podemos usar var para declarar una variable, asignarle una expresión y el compilador le dará el tipo.

A esta variable el compilador le dará tipo entero.

```
var num =1;
```

A numd el compilador le dará a la variable tipo float

```
var numd = 1.0;
```

3.6 Operadores

- De asignación. =
- Aritméticos. + - * / % (resto de la división entera)
- Binarios: & (Y binario) | (O binario) ^ (O exclusivo) ~ (negación)
- Comparación: == (igual) != (distinto) < (menor) > (mayor) <= (menor o igual) >= (mayor o igual)

o

igual) >= (mayor o igual) is (compara el tipo de variable con el tipo dado.

```
if (edad is int) { ...}
```

- Concatenación. Podemos concatenar con el símbolo +, o mejor con StringBuilder que realiza la operación más rápida.
 - string cadena = "123";
 - Console.WriteLine (cadena + 456);
- ```
// Visualizará 123456
```

## Lógicos

| Operador | Operación   | Ejemplo              | Resultado                                              |
|----------|-------------|----------------------|--------------------------------------------------------|
| &        | Y lógico    | If (test1)&(test2)   | Cierto si ambos son ciertos                            |
|          | O lógico    | If (test1)   (test2) | Cierto si alguno de los dos es cierto                  |
| ^        | O exclusivo | If (test1)^(test2)   | Cierto si alguno es cierto y el otro NO lo es          |
| !        | Negación    | If ! test            | Invierte el resultado de test                          |
| &&       | Y lógico    | If (test1)&&(test2)  | Igual que &, pero solo evalúa test2 si test1 es cierto |
|          | O lógico    | If (test1)   (test2) | Igual que  , pero solo evalúa test2 si test1 es falso  |

Los repasamos brevemente en el siguiente ejemplo.

```
using System;

namespace Operadores
{
 class Program
 {
 static void Main(string[] args)
 {
 int i = 0;

 float fNumero = 0.0F;

 double resultado = 0.5;

 resultado += fNumero;

 i = i - 3;

 i++;

 --i;
 }
 }
}
```



```

 if (i<0 && resultado>0)
 {
 resultado = resultado * 2;
 Console.WriteLine(" Operador and"+ " El resultado es {0}", resultado);
 }
 }
}

```

### 3.7 Ejercicios

1. Realizar un programa que pida dos números por pantalla y nos devuelva las cuatro operaciones aritméticas básicas dando esta salida por pantalla

PRIMER NÚMERO :5  
SEGUNDO NÚMERO :6

LA SUMA ES 11:  
LA RESTA ES: 5 - 6 = -1  
LA MULTIPLICACIÓN ES: 30  
LA DIVISIÓN ES: 0  
EL RESIDUO ES: 5

2. Realizar un programa que recoja un número de tres cifras por pantalla, y lo invierta. Por ejemplo, si recibimos el número 345, en programa nos devolverá 543 por pantalla.

INGRESE NÚMERO DE TRES CIFRAS :  
345  
NÚMERO INVERTIDO ES: 543

3. Realizar un programa que calcule el sueldo mas bono del trabajador según el numero de hijos. Por cada hijo se añadirán 100 euros a su sueldo mensual

NOMBRE EMPLEADO :Luis  
SUELDO EMPLEADO :1000  
NÚMERO DE HIJOS :2  
RECIBE : 1200,00

### 3.8 Instrucciones de control

Las instrucciones de control son las típicas de cualquier lenguaje de programación, es decir:

- Condicionales o de selección
- Bucles, iterativas o repetitivas

### 3.8.1 Instrucción If

Como casi todos los lenguajes de programación basados en c, permite if, elseif, y else para seleccionar diferentes situaciones en el flujo de control del código

```
bool condition = true;
if (condition) {
 Console.WriteLine("La variable es asignada a verdadero");
}
else if (!condition)
{
 Console.WriteLine("La variable es asignada a falso");
} else
{
 Console.WriteLine("La variable es asignada a null");
}
```

### 3.8.2 Instrucción switch

Podemos elegir con la sentencia switch case, múltiples casos apartir de un entero. Como veremos más adelante las enumeraciones que son de tipo entero pueden usarse como valor de caso para el switch.

```
// SWITCH case

Random rnd = new Random();
int caseSwitch = rnd.Next(1, 4);

switch (caseSwitch)
{
 case 1:
 Console.WriteLine("Caso 1");
 break;
```

```

 case 2:
 case 3:
 Console.WriteLine($"Caso {caseSwitch}");
 break;
 default:
 Console.WriteLine($"Valor inesperado ({caseSwitch})");
 break;
 }

```

En el siguiente ejemplo tenemos el if y el switch funcionando.

```

static void Main(string[] args)
{
 bool condition = true;

 //IF

 if (condition)
 {
 Console.WriteLine("La variable es asignada a verdadero");
 }
 else if (!condition)
 {
 Console.WriteLine("La variable es asignada a falso");
 }
 else
 {
 Console.WriteLine("La variable es asignada a null");
 }

 // SWITCH case

 Random rnd = new Random();
 int caseSwitch = rnd.Next(1, 4);

 switch (caseSwitch)
 {
 case 1:
 Console.WriteLine("Caso 1");
 break;
 case 2:
 case 3:
 Console.WriteLine($"Caso {caseSwitch}");
 break;
 default:
 Console.WriteLine($"Valor inesperado ({caseSwitch})");
 break;
 }
}

```

## Resultado de la ejecución

La variable es asignada a verdadero  
Caso 3

### 3.8.3 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 de un bucle do que se ejecuta una o varias veces.

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

### 3.8.4 Instrucción Do While

El do while entra al menos una vez en el bucle ya que la condición se evalúa después de ejecutar el bucle por primera vez. El bucle se seguirá ejecutando hasta que la condición entre paréntesis sea evaluada a false.

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

### 3.8.5 For

La instrucción for ejecuta una instrucción o un bloque de instrucciones mientras una expresión booleana especificada se evalúa como true. En el ejemplo siguiente se muestra la instrucción for, que ejecuta su cuerpo mientras que un contador entero sea menor que tres:

For (inicializador; condición; incremento/decremento) {

Bloque de instrucciones;

```
...
}
```

```
for (int i = 0; i < 3; i++)
{
 Console.Write(i);
}
```

En el siguiente ejemplo tenéis las tres instrucciones funcionando

```
static void Main(string[] args)
{

 int n = 0;
 while (n < 5)
 {
 Console.Write(n);
 n++;
 }

 n = 0;
 do
 {
 Console.Write(n);
 n++;
 } while (n < 5);

 for (int i = 0; i < 3; i++)
 {
 Console.Write(i);
 }
}
```

### 3.8.6 Instrucción foreach

Esta instrucción que se implementa de manera muy parecida a JavaScript esta especialmente diseñada para recorrer arrays y colecciones en C#. La instrucción foreach ejecuta una instrucción o un bloque de instrucciones para cada elemento de una instancia del tipo que implementa la interfaz System.Collections.IEnumerable. Se declara una variable de tipo elemento de la colección, y el foreach automáticamente carga cada elemento de la colección en esa variable y realiza una iteración.

```
var fibNumbers = new List<int> { 0, 1, 1, 2, 3, 5, 8, 13 };
foreach (int element in fibNumbers)
{
 Console.WriteLine($"{element} ");
}
```

```
}
```

Ejemplo completo:

Debemos usar el namespace `using System.Collections.Generic;` para poder usar la clase List, que es de tipo colección lista.

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

namespace Foreach
{
 class Program
 {
 static void Main(string[] args)
 {
 var fibNumbers = new List<int> { 0, 1, 1, 2, 3, 5, 8, 13 };
 foreach (int element in fibNumbers)
 {
 Console.WriteLine($"{element} ");
 }
 }
 }
}
```

Se realiza una iteración por cada elemento de la lista que se va cargando en la variable element.

Resultado de la ejecución:

```
0 1 1 2 3 5 8 13
```

### 3.8.7 Ejercicios

1. Realizar un programa que recoja dos números por pantalla y nos indique si el primero es mayor, igual o menor que el segundo.

Ejecución:

```
NÚMERO 1 :5
NÚMERO 2 :7
5 ES MENOR QUE 7
```

2. Realizar un programa con un switch que nos diga a partir de un número entre 1 y 7 a que día de la semana corresponde.

Ejecución:

DIAS DE LA SEMANA

Ingrese un numero del 1 al 7 :2

El numero que ingreso corresponde al dia MARTES

3. Realizar un programa que nos calcule la suma y la media de los números introducidos por pantalla.

Ejecución

```
INTRODUZCA UN NÚMERO:5
INTRODUZCA UN NÚMERO:6
INTRODUZCA UN NÚMERO:7
INTRODUZCA UN NÚMERO:8
LA SUMA TOTAL ES : 26
LA MEDIA ARITMÉTICA: 6
```

4. Modificar el programa anterior para indicar un límite de números a introducir
- Ejecución:

```
LÍMITE DE NUMEROS A INTRODUCIR:4
INTRODUZCA UN NÚMERO:5
INTRODUZCA UN NÚMERO:6
INTRODUZCA UN NÚMERO:7
INTRODUZCA UN NÚMERO:8
LA SUMA TOTAL ES : 26
LA MEDIA ARITMÉTICA: 6
```

## 3.9 Tipos avanzados en c#

### 3.9.1 Enumeraciones

Un tipo de enumeración es un tipo de valor definido por un conjunto de constantes con nombre del tipo numérico integral subyacente. Para definir un tipo de enumeración, use la palabra clave enum y especifique los nombres de miembros de enumeración:

```
enum Season
{
 Primavera,
```

```
Verano,
Otoño,
Invierno
}
```

De forma predeterminada, los valores de constante asociados de miembros de enumeración son del tipo int; comienzan con cero y aumentan en uno después del orden del texto de la definición. Puede especificar explícitamente cualquier otro tipo de numérico entero como un tipo subyacente de un tipo de enumeración. También puede especificar explícitamente los valores de constante asociados, como se muestra en el ejemplo siguiente:

```
enum CodigoError : ushort
{
 SinError = 0,
 Desconocido = 1,
 ConexionPerdida = 100,
 ErrorLectura = 200
}
```

Por defecto, cuando escribimos las enumeraciones, escribirá el String con el que hemos definido el tipo. Podemos convertirlo al numérico que nos interesa con un casteo o unbox.

```
Estacion est = Estacion.Otoño;
Console.WriteLine($"El valor numérico de {est} es {(int)est}");
```

Resultado de la ejecución:

El valor numérico de Otoño es 2

## Ejemplo Enumeraciones

```
using System;

namespace Enumeraciones {

 enum Estacion
 {
 Primavera,
 Verano,
 Otoño,
 Invierno
 }

 enum CodigoError : ushort
 {
 SinError = 0,
 Desconocido = 1,
 ConexionPerdida = 100,
 ErrorLectura = 200
 }

 class Program
```



```

{
 static void Main(string[] args)
 {

 Estacion est = Estacion.Otoño;
 Console.WriteLine($"El valor numérico de {est} es {(int)est}");

 CodigoError cError = CodigoError.Desconocido;
 Console.WriteLine($"El valor numérico de {cError} es {(short)cError}");

 }
}

```

### 3.9.2 Tuplas

Disponible en C# 7.0 y versiones posteriores, la característica *tuplas* proporciona una sintaxis concisa para agrupar varios elementos de datos en una estructura de datos ligera. En el siguiente ejemplo se muestra cómo se puede declarar una variable de tupla, inicializarla y acceder a sus miembros de datos. Lo podemos ver en el siguiente ejemplo. Podemos acceder a cada elemento con la propiedad `.Item1` e `.Item2`.

```

(double, int) t1 = (4.5, 3);
Console.WriteLine($"Tupla con elementos {t1.Item1} y {t1.Item2}.");

```

Podemos definir nombres para cada elemento de la tupla.

```

(double Sum, int Count) t2 = (4.5, 3);
Console.WriteLine($"La suma de los {t2.Count} elementos de la tupla is {t2.Sum}.");

```

A partir de C# 7.3, los tipos de tupla admiten operadores de igualdad `==` y `!=`. Para obtener más información, consulte la sección Igualdad de tupla.

Los tipos de tupla son tipos de valores; los elementos de tupla son campos públicos. Esto hace que las tuplas sean tipos de valor mutables.

Puede definir tuplas con un gran número arbitrario de elementos:

```

var t =
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 21, 22, 23, 24, 25, 26);
Console.WriteLine(t.Item26);

```

Ejemplo tuplas

```

using System;

namespace Tuplas
{
 class Program
 {
 static void Main(string[] args)
 {
 (double, int) t1 = (4.5, 3);
 Console.WriteLine($"Tupla con elementos {t1.Item1} y {t1.Item2}.");

 (double Sum, int Count) t2 = (4.5, 3);
 Console.WriteLine($"La suma de los {t2.Count} elementos de la tupla is
{t2.Sum}.");

 var t =
 (1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
 11, 12, 13, 14, 15, 16, 17, 18,
 19, 20, 21, 22, 23, 24, 25, 26);
 Console.WriteLine(t.Item26);
 }
 }
}

```

### 3.9.3 Estructuras

Aparecen en C, como veréis son una versión simplificada de una clase, que puede contener atributos o elementos y métodos. Un tipo de estructura (o tipo struct) es un tipo de valor que puede encapsular datos y funcionalidad relacionada. Para definir un tipo de estructura se usa la palabra clave struct:

Los tipos de estructura tienen semántica de valores. Es decir, una variable de un tipo de estructura contiene una instancia del tipo. De forma predeterminada, los valores de variable se copian al asignar, pasar un argumento a un método o devolver el resultado de un método. En el caso de una variable de tipo de estructura, se copia una instancia del tipo. Para más información, vea Tipos de valor.

Normalmente, los tipos de estructura se usan para diseñar tipos de pequeño tamaño centrados en datos que proporcionan poco o ningún comportamiento. Por ejemplo, en .NET se usan los tipos de estructura para representar un número (entero y real), un valor booleano, un carácter Unicode. Permite usar modificadores de tipo public y private.

Ejemplo completo. Hemos declarado las estructuras dentro de la clase program, fuera del método Main. Representamos primero una dirección con un Struct Direccion. Fijaos que tiene un método `public string getDireccion()`. Después usamos esta estructura en la estructura cliente para representar su dirección, `public Direccion correoPostal;`.

Para acceder a los campos de las estructuras usamos el operador punto '.' sobre la variable definida como tipo estructura `Cliente UnCliente`, si queremos hacer al código del cliente lo

hacemos de esta manera `UnCliente.codigo = 9999`.

```
Cliente UnCliente;
UnCliente.codigo = 9999;
UnCliente.apellido = "Pedraza";
UnCliente.nombre = "Juanjo";
UnCliente.correoPostal.codPostal = 3740;
```

```
using System;
```

```
namespace Estructuras
```

```
{
 class Program
 {
 struct Direccion
 {
 public int codPostal;
 public string calle;
 public string ciudad;
 public string getDireccion()
 {
 return calle + "\r\n " + codPostal + "\t" + ciudad.ToUpper();
 }
 }

 // fijarse en el tipo de dato del campo correoPostal
 struct Cliente
 {
 public int codigo;
 public string apellido;
 public string nombre;
 public string email;
 public Direccion correoPostal;

 };

 static void Main(string[] args)
 {
 Cliente UnCliente;
 UnCliente.codigo = 9999;
 UnCliente.apellido = "Pedraza";
 UnCliente.nombre = "Juanjo";
 UnCliente.correoPostal.codPostal = 3740;
 UnCliente.correoPostal.calle = "C/ Perico Palotes, 33";
 UnCliente.correoPostal.ciudad = "Gata de Gorgos City";

 Console.WriteLine(UnCliente.correoPostal.getDireccion());
 }
 }
}
```

### 3.9.4 Arrays

Puede almacenar varias variables del mismo tipo en una estructura de datos de matriz. Puede declarar una matriz mediante la especificación del tipo de sus elementos. Si quiere que la matriz almacene elementos de cualquier tipo, puede especificar `object` como su tipo. En el

sistema de tipos unificado de C#, todos los tipos, los predefinidos y los definidos por el usuario, los tipos de referencia y los tipos de valores, heredan directa o indirectamente de Object.

Sintaxis:

```
type[] arrayName;
```

Un array tiene las propiedades siguientes:

- Puede ser un array unidimensional, multidimensional o escalonada.
- El número de dimensiones y la longitud de cada dimensión se establecen al crear la instancia de un array. No se pueden cambiar estos valores durante la vigencia de la instancia.
- Los valores predeterminados de los elementos numéricos de arrayz se establecen en cero y los elementos de referencia se establecen en null.
- Una matriz escalonada es un array de arrays y, por consiguiente, sus elementos son tipos de referencia y se inicializan en null.

Declaración de arrays simples. De tres maneras diferentes podemos declarar un array de una dimensión.

```
int[] array1 = new int[5];
```

```
int[] array2 = new int[] { 1, 3, 5, 7, 9 };
```

```
int[] array3 = { 1, 2, 3, 4, 5, 6 };
```

Recorrido de un array con for, dos opciones:

- Con índice y la longitud del array: `for (int i=0; i< array3.Length; i++)`
- Con un foreach: `foreach( int num in array2)`

```
Console.WriteLine("Array de una dimension");

for (int i=0; i< array3.Length; i++)
{
 Console.Write("[{0}] = {1} |", i, array3[i]);
}

Console.WriteLine("Array de una dimension con foreach");
foreach(int num in array2)
```

```

{
 Console.Write(" {0} |", num);
}

```

Para un array multidimensiones podemos declararlo de la siguiente manera, al final siempre indicando sus dos dimensiones:

En el primer caso de manera explicita 2,3.

```
int[,] arrayDosDimensiones = new int[2, 3];
```

En el segundo, através de la asignación de elementos, que también es 2,3.

```
int[,] arrayDosDimensiones2 = { { 1, 2, 3 }, { 4, 5, 6 } };
```

Y para recorrerlo Podemos usar dos bucles for.

```

for (int i = 0; i < 2; i++)
{
 Console.WriteLine("\n");
 for (int j = 0; j < 3; j++)
 {
 arrayDosDimensiones[i, j] = i + j;
 Console.Write("[{0},{1}] = {2} |", i, j, arrayDosDimensiones[i, j]);

 }

}

```

La diferencia entre el array de dos dimensiones como este y el escalonado que vamos a ver ahora es que la segunda dimensión para arrayDosDimensiones que es una matriz, siempre es la misma

arrayDosDimensiones

|   |   |
|---|---|
| 0 | 1 |
| 1 | 2 |
| 2 | 3 |

Sin embargo, los arrays escalonados pueden ser visto como un array de arrays. Y cada array individual puede tener su propia dimensión.

El array almacenado en la posición cero tiene 4 elementos mientras que el almacenado en la posición uno tiene 2.

arrayEscalonado

|   |   |   |   |    |
|---|---|---|---|----|
| 1 | 2 | 3 | 4 |    |
| 0 | 0 |   |   |    |
| 1 | 3 | 5 |   |    |
| 2 | 4 | 6 | 8 | 10 |

En los arrays escalonados sólo declaramos la primera dimensión:

```
int[][] arrayEscalonado = new int[4][];
```

Posteriormente declaramos cada array individualmente:

```
arrayEscalonado[0] = new int[4] { 1, 2, 3, 4 };
arrayEscalonado[1] = new int[2];
arrayEscalonado[2] = new int[3] { 1, 3, 5 };
arrayEscalonado[3] = new int[5] { 2, 4, 6, 8, 10 };
```

Ejemplo de declaración de array escalonado más recorrido:

```
int[][] arrayEscalonado = new int[4][];

// Set the values of the first array in the jagged array structure.
arrayEscalonado[0] = new int[4] { 1, 2, 3, 4 };
arrayEscalonado[1] = new int[2];
arrayEscalonado[2] = new int[3] { 1, 3, 5 };
arrayEscalonado[3] = new int[5] { 2, 4, 6, 8, 10 };

Console.WriteLine("\n\nArrayEscalonado");

for (int i = 0; i < 4; i++)
{
 Console.WriteLine("\nValor para el array posición {0}", i);
 foreach (int num in arrayEscalonado[i])
 {
 Console.Write("| {0} ", num);
 }
}
```

## Ejemplo de arrays

Ejemplo completo

```
using System;
```

```

namespace Arrays
{
 class Program
 {
 static void Main(string[] args)
 {
 int[] array1 = new int[5];

 int[] array2 = new int[] { 1, 3, 5, 7, 9 };

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

 int[,] arrayDosDimensiones = new int[2, 3];

 int[,] arrayDosDimensiones2 = { { 1, 2, 3 }, { 4, 5, 6 } };
 Console.WriteLine("Array de dos dimensiones");
 for (int i = 0; i < 2; i++)
 {
 Console.WriteLine("\n");
 for (int j = 0; j < 3; j++)
 {
 arrayDosDimensiones[i, j] = i + j;
 Console.Write("[{0},{1}] = {2} |", i, j, arrayDosDimensiones[i, j]);

 }

 }

 int[][] arrayEscalonado = new int[4][];

 // Set the values of the first array in the jagged array structure.
 arrayEscalonado[0] = new int[4] { 1, 2, 3, 4 };
 arrayEscalonado[1] = new int[2];
 arrayEscalonado[2] = new int[3] { 1, 3, 5 };
 arrayEscalonado[3] = new int[5] { 2, 4, 6, 8, 10 };

 Console.WriteLine("\n\nArrayEscalonado");

 for (int i = 0; i < 4; i++)
 {
 Console.WriteLine("\nValor para el array posición {0}", i);
 foreach (int num in arrayEscalonado[i])
 {
 Console.Write("| {0} ", num);

 }

 }

 }
 }
}

```

```
}
```

## Ejercicios

1. Rellenar un array de decimales de 50 posiciones usando la clase Random de .NET, y el método

```
Random rnd = new Random();
```

```
rnd.Next(0, 99) // rellena con números entre 0 y 99
```

2. Creando un método estático ordenaBurbuja, ordena al array obtenido anteriormente usando el método de la burbuja
3. Crea un método estático para realizar la búsqueda binaria sobre el array anteriormente generado.

## 3.10 Cadenas en c#

En C#, la palabra clave string es un alias de String. Por lo tanto, String y string son equivalentes, aunque se recomienda usar el alias proporcionado string, ya que funciona incluso sin using System;. La clase String proporciona muchos métodos para crear, manipular y comparar cadenas de forma segura. Además, el lenguaje C# sobrecarga algunos operadores para simplificar las operaciones de cadena comunes.

Hay muchas maneras de inicializar Strings en c#. En el siguiente ejemplo podéis ver varias de ellas:

```
// Declaramos sin inicializar
string cadena1;

// Inicializando a null
string cadena2 = null;

// Cadena vacia sin usar literal "".
string cadenaVacía = System.String.Empty;

// Inicialización normal
string oldPath = "c:\\Program Files\\Microsoft Visual Studio 8.0";

// Inicialización literal, se escribe tal y como aparece con el operador @
string newPath = @"c:\Program Files\Microsoft Visual Studio 9.0";

// Usando System.String
System.String greeting = "Hello World!";
```



```
// Usando variables con tipos implícitos
var temp = "I'm still a strongly-typed System.String!";

// constante.
const string message4 = "You can't get rid of me!";
```

A destacar que en c# se pueden utilizar variables de tipo implícito con el modificador var.

```
var temp = "I'm still a strongly-typed System.String!";
```

La variable temp no toma el tipo String hasta que en tiempo de ejecución el literal es asignado.

Encontrareis en la mayoría de objetos y tipos el método toString para transformar a cadena.

### 3.10.1 El método ToString

La mayoría de los tipos básicos y clases predefinidas tienen el método toString para transformarse a cadena. Vamos a ver unos cuantos ejemplos, que es la manera más gráfica de ver como se realiza la conversión a String.

En los primeros ejemplos transformamos de numérico a cadena con el método ToString

```
double numDouble= 4.6d;
Console.WriteLine("double a String {0}", numDouble.ToString());

int numInteger = 567;

Console.WriteLine("integer a String {0}", numInteger.ToString());

Decimal numDecimal = 56678.889M;

Console.WriteLine("Decinal a String {0}", numDecimal.ToString());

DateTime fecha = DateTime.Now;

Console.WriteLine("Decinal a String {0}", fecha.ToString());

(int nota, string titulo) tupla = (5,"aprobado");

Console.WriteLine("Tupla a String {0}", tupla.ToString());
```

Para enumeraciones el método ToString transforma el valor a una cadena exactamente igual a la definida en la enumeración. En este caso devolverá “AMARILLO”.

```
Colores color = Colores.AMARILLO;

Console.WriteLine("Enumeracion a String {0}", color.ToString());
```

Las estructuras devuelve en su ToString el tipo que son. En este caso devolvería Direccion. Es necesario sobrescribir el método ToString como hemos realizado en el ejemplo, con el modificador override, pues el tipo struct ya tiene el método implementado, `public override string ToString()`

```

struct Direccion
{
 public int codPostal;
 public string calle;
 public string ciudad;
 public string getDireccion()
 {
 return calle + "\r\n " + codPostal + "\t" + ciudad.ToUpper();
 }

 public override string ToString()
 {
 return String.Format("Calle: {0} \r\nCodigo Postal: {1}\t ciudad: {2},", calle,
codPostal, ciudad);
 }
}

 Direccion direccion;

 direccion.calle = "Calle Hermanos Galiano s/n";
 direccion.ciudad = "Guadalajara";
 direccion.codPostal = 19004;

 Console.WriteLine("Estructura a String {0}", direccion.ToString());

```

El ejemplo completo se proporciona a continuación.

```

using System;

namespace ConversionesaString
{
 enum Colores
 {
 ROJO,
 AMARILLO,
 VERDE,
 AZUL
 }

 struct Direccion
 {
 public int codPostal;
 public string calle;
 public string ciudad;
 public string getDireccion()
 {
 return calle + "\r\n " + codPostal + "\t" + ciudad.ToUpper();
 }
 }
}

```

```

 public override string ToString()
 {
 return String.Format("Calle: {0} \r\nCodigo Postal: {1}\t ciudad: {2},", calle,
codPostal, ciudad);
 }
 }

 class Program
 {
 static void Main(string[] args)
 {
 double numDouble= 4.6d;
 Console.WriteLine("double a String {0}", numDouble.ToString());

 int numInteger = 567;

 Console.WriteLine("integer a String {0}", numInteger.ToString());

 Decimal numDecimal = 56678.889M;

 Console.WriteLine("Decinal a String {0}", numDecimal.ToString());

 DateTime fecha = DateTime.Now;

 Console.WriteLine("Decinal a String {0}", fecha.ToString());

 (int nota, string titulo) tupla = (5,"aprobado");

 Console.WriteLine("Tupla a String {0}", tupla.ToString());

 Colores color = Colores.AMARILLO;

 Console.WriteLine("Enumeracion a String {0}", color.ToString());

 Direccion direccion;

 direccion.calle = "Calle Hermanos Galiano s/n";
 direccion.ciudad = "Guadalajara";
 direccion.codPostal = 19004;

 Console.WriteLine("Estructura a String {0}", direccion.ToString());

 }
 }
}

```

### 3.10.2 Los Strings son inmutables

os objetos de cadena son inmutables: no se pueden cambiar después de haberse creado. Todos los métodos String y operadores de C# que parecen modificar una cadena en realidad devuelven los resultados en un nuevo objeto de cadena. En el siguiente ejemplo, cuando el contenido de s1 y s2 se concatena para formar una sola cadena, las dos cadenas originales no se modifican. El operador += crea una nueva cadena que contiene el contenido

combinado. Este nuevo objeto se asigna a la variable s1 y el objeto original que se asignó a s1 se libera para la recolección de elementos no utilizados porque ninguna otra variable contiene una referencia a él.

**Nota:** Tener muy en cuenta esto porque cada vez que asignamos a una variable cadena se crea un Objeto String nuevo. Por tanto no se puede pasar una cadena por referencia.

### 3.10.3 Caracteres de escape

En el siguiente ejemplo se puede observar como se usan los caracteres de escape dentro de una cadena, tando con el slash o barra invertida como con numeración Unicode. Para unicode se añade \u para UTF16 y \U para UTF32 delante del numero \u00C6.

```
//Uso de caracteres de escape

string columns = "Column 1\tColumn 2\tColumn 3";
//Output: Column 1 Column 2 Column 3

string rows = "Row 1\r\nRow 2\r\nRow 3";
/* Output:
Row 1
Row 2
Row 3
*/

string title = "\tThe \u00C6olean Harp\t", by Samuel Taylor Coleridge";
//Output: "The Åolean Harp", by Samuel Taylor Coleridge
```

Tabla con secuencias de escape

| Secuencia de escape | Nombre de carácter     | Codificación Unicode |
|---------------------|------------------------|----------------------|
| \'                  | Comilla simple         | 0x0027               |
| \"                  | Comilla doble          | 0x0022               |
| \\                  | Barra diagonal inversa | 0x005C               |
| \0                  | Null                   | 0x0000               |
| \a                  | Alerta                 | 0x0007               |
| \b                  | Retroceso              | 0x0008               |
| \f                  | Avance de página       | 0x000C               |
| \n                  | Nueva línea            | 0x000A               |
| \r                  | Retorno de carro       | 0x000D               |
| \t                  | Tabulación horizontal  | 0x0009               |
| \v                  | Tabulación vertical    | 0x000B               |

| Secuencia de escape | Nombre de carácter                                                       | Codificación Unicode                                                     |
|---------------------|--------------------------------------------------------------------------|--------------------------------------------------------------------------|
| \u                  | Secuencia de escape Unicode (UTF-16)                                     | \uHHHH (intervalo: 0000 - FFFF; ejemplo: \u00E7 = "ç")                   |
| \U                  | Secuencia de escape Unicode (UTF-32)                                     | \U00HHHHHH (intervalo: 000000 - 10FFFF; ejemplo: \U0001F47D = "🐶")       |
| \x                  | Secuencia de escape Unicode similar a "\u" excepto con longitud variable | \xH[H][H][H] (intervalo: 0 - FFFF; ejemplo: \x00E7 O \x0E7 O \xE7 = "ç") |

### 3.10.4 String.Format

Usamos String.Format si se necesita insertar el valor de un objeto, una variable o una expresión en otra cadena. Por ejemplo, puede insertar el valor de un Decimal valor en una cadena para mostrarlo al usuario como una sola cadena:

```
Decimal precioPorKilo = 17.36m;
String s = String.Format("El precio por kilo es {0} ",
 precioPorKilo);
Console.WriteLine(s);
```

Resultado

El precio por kilo es 17,36

Se puede seguir el índice de un elemento de formato con una cadena de formato para controlar cómo se da formato a un objeto. Por ejemplo, {0:d} aplica la cadena de formato "d" al primer objeto de la lista de objetos, la fecha, y "t" para la hora. A continuación se muestra un ejemplo con un solo objeto y dos elementos de formato.

```
string s2 = String.Format("La fecha {0:d} y hora {0:t}", DateTime.Now);
Console.WriteLine(s2);
```

Resultado:

La fecha 09/08/2021 y hora 13:01

Puede definir el ancho de la cadena que se inserta en la cadena de resultado mediante una sintaxis como {0,12}, que inserta una cadena de 12 caracteres. En este caso, la representación de cadena del primer objeto está alineada a la derecha en el campo de 12 caracteres. Si, sin embargo, la representación de cadena del primer objeto tiene más de 12 caracteres de longitud, se omite el ancho de campo preferido y se inserta la cadena completa en la cadena de resultado.

En el ejemplo siguiente se define un campo de 6 caracteres que contiene la cadena "Año" y algunas cadenas de año, así como un campo de 15 caracteres que contiene la cadena "Poblacion" y algunos datos de población. Tenga en cuenta que los caracteres están alineados a la derecha en el campo.

```
int[] años = { 2013, 2014, 2015 };
int[] poblacion = { 1025632, 1105967, 1148203 };
var sb = new System.Text.StringBuilder();
sb.Append(String.Format("{0,6} {1,15}\n\n", "Año", "Población"));
for (int index = 0; index < años.Length; index++)
 sb.Append(String.Format("{0,6} {1,15:N0}\n", años[index], poblacion[index]));
```

| Año  | Población |
|------|-----------|
| 2013 | 1.025.632 |
| 2014 | 1.105.967 |
| 2015 | 1.148.203 |

Aparece con formato centrado, si quisierais alinear a la izquierda añadid el signo negativo, positivo a la derecha.

```
sb.Append(String.Format("{0,-6} {1,-15}\n\n", "Año", "Población"));
for (int index = 0; index < años.Length; index++)
 sb.Append(String.Format("{0,-6} {1,-15:N0}\n", años[index], poblacion[index]));
```

Si queréis indicar el formato de la parte entera y decimal podéis hacerlo como sigue:

:#.000 -> numero indefinido de cifras parte entera, tres cifras decimales

:000000000.0000000 -> nueve dígitos parte entera y se rellena con ceros a la izquierda, 7 parte decimal

### 3.10.5 Elementos de formato

Un elemento de formato tiene esta sintaxis:

{index[,alignment][:formatString]}

Los corchetes denotan elementos opcionales. Las llaves de apertura y cierre son obligatorias. (Para incluir una llave de apertura o de cierre literal en la cadena de formato, consulte la sección llaves de escape en el artículo formatos compuestos ).

Por ejemplo, un elemento de formato para dar formato a un valor de moneda podría ser similar al siguiente:

Un elemento de formato tiene los siguientes elementos:

- Index: Índice de base cero del argumento cuya representación de cadena se va a incluir en esta posición de la cadena. Si este argumento es null , se incluirá una cadena vacía en esta posición de la cadena.
- Alignment es opcional: Entero con signo que indica la longitud total del campo en el que se inserta el argumento y si está alineado a la derecha (un entero positivo) o alineado a la izquierda (un entero negativo). Si omite alignment, la representación de cadena del argumento correspondiente se inserta en un campo sin espacios iniciales ni finales.

Si el valor de alignment es menor que la longitud del argumento que se va a insertar, se omite alignment y se usa la longitud de la representación de cadena del argumento como el ancho del campo.

- formatString: Opcional. Cadena que especifica el formato de la cadena de resultado del argumento correspondiente. Si omite FormatString, se llama al método sin parámetros del argumento correspondiente ToString para generar su representación de cadena. Si especifica FormatString, el argumento al que hace referencia el elemento de formato debe implementar la IFormattable interfaz. Los tipos que admiten cadenas de formato incluyen:
  - ✓ Todos los tipos integrales y de punto flotante.

<https://docs.microsoft.com/es-es/dotnet/standard/base-types/standard-numeric-format-strings>

Cambiamos de modena cargando el país local y el lenguaje con la clase CultureInfo(), después de transformar el decimal a cadena.

/Elemento de formato

```
string value = String.Format("{0,-10:c}", 126.89m);
Console.WriteLine(value);

decimal dec = 126.89M;

string uk = dec.ToString("C", new CultureInfo("en-GB"));

Console.WriteLine(uk);
```

- ✓ DateTime y DateTimeOffset.

Podéis usar la clase DateTime para trabajar con las fechas. Admite un constructor sin parámetros y el constructor con parámetros que podéis ver en el ejemplo.

```
var date1 = new DateTime(2008, 5, 1, 8, 30, 52);
Console.WriteLine("Fecha sin formatear {0}", date1);

string fechaFormateada = String.Format("La fecha {0:d} y hora {0:t}", date1);
Console.WriteLine("Fecha formateada {0}", fechaFormateada);
```

E igualmente acceder a la fecha y hora de hoy con DateTime.Now, o la fecha de hoy con DateTime.Today, como ya hicimos en el ejemplo inicial.

```
DateTime date1 = DateTime.Now;
DateTime date2 = DateTime.UtcNow;
DateTime date3 = DateTime.Today;
```

- ✓ Todos los tipos de enumeración.

Para las enumeraciones usamos el método `ToString` en dos transformaciones de formato básicas, para forzar la enumeración como cadena `estacion.ToString("G")`; o como número `estacion.ToString("D")`;

```
//Formato para enumeraciones
```

```
Estacion estacion = Estacion.Verano;

string EstacionCadena = estacion.ToString("G");
string EstacionCadenaNumerica = estacion.ToString("D");

Console.WriteLine("Enumeracion cadena: {0}, Enumeracion numérica {1}",
EstacionCadena, EstacionCadenaNumerica);
```

- ✓ valores `TimeSpan`. formato `TimeSpan` personalizado).
- ✓ `GUID`. ( método `Guid.ToString(String)`).

El ejemplo completo se ofrece a continuación

```
using System;
using System.Globalization;

namespace StringFormat
{

 enum Estacion
 {
 Primavera,
 Verano,
 Otoño,
 Invierno
 }

 class Program
 {
 static void Main(string[] args)
 {
 Decimal precioPorKilo = 17.36m;
 String s = String.Format("El precio por kilo es {0} ",
 precioPorKilo);
 Console.WriteLine(s);

 string s2 = String.Format("La fecha {0:d} y hora {0:t}", DateTime.Now);
 Console.WriteLine(s2);
 }
 }
}
```



```

int[] años = { 2013, 2014, 2015 };
int[] poblacion = { 1025632, 1105967, 1148203 };
var sb = new System.Text.StringBuilder();
sb.Append(String.Format("{0,6} {1,15}\n\n", "Año", "Población"));
for (int index = 0; index < años.Length; index++)
 sb.Append(String.Format("{0,6} {1,15:N0}\n", años[index], poblacion[index]));

Console.WriteLine(sb);

//Elemento de formato

string value = String.Format("{0,-10:c}", 126.89m);

Console.WriteLine(value);

decimal dec = 126.89M;

string uk = dec.ToString("C", new CultureInfo("en-GB"));

Console.WriteLine(uk);

// Fecha y hora con DateTime

var date1 = new DateTime(2008, 5, 1, 8, 30, 52);
Console.WriteLine("Fecha sin formatear {0}", date1);

string fechaFormateada = String.Format("La fecha {0:d} y hora {0:t}", date1);

Console.WriteLine("Fecha formateada {0}", fechaFormateada);

//Formato para enumeracions

Estacion estacion = Estacion.Verano;

string EstacionCadena = estacion.ToString("G");
string EstacionCadenaNumerica = estacion.ToString("D");

Console.WriteLine("Enumeracion cadena: {0}, Enumeracion numérica {1}",
EstacionCadena, EstacionCadenaNumerica);

 }
}
}

```

### 3.10.6 StringBuilder

Esta clase representa un objeto de tipo cadena cuyo valor es una secuencia de caracteres mutable. Nos va a permitir construir cadenas pero además, pasar cadenas por referencia.

#### Como usarlo

La propiedad `StringBuilder.Length` indica el número de caracteres que `StringBuilder` contiene el objeto actualmente. Si agrega caracteres al `StringBuilder` objeto, su longitud aumenta hasta que es igual al tamaño de la `StringBuilder.Capacity` propiedad, que define el número de caracteres que puede contener el objeto. Si el número de caracteres agregados hace que la longitud del `StringBuilder` objeto supere su capacidad actual, se asigna una nueva memoria, el valor de la `Capacity` propiedad se duplica, se agregan nuevos caracteres al `StringBuilder` objeto y `Length` se ajusta su propiedad. La memoria adicional para el `StringBuilder` objeto se asigna dinámicamente hasta que alcanza el valor definido por la `StringBuilder.MaxCapacity` propiedad. Cuando se alcanza la capacidad máxima, no se puede asignar más memoria para el `StringBuilder` objeto y, al intentar agregar caracteres o expandirlo más allá de su capacidad máxima, se produce una `ArgumentOutOfRangeException` o `OutOfMemoryException` excepción o.

En el ejemplo siguiente se muestra cómo un `StringBuilder` objeto asigna nueva memoria y aumenta su capacidad dinámicamente a medida que se expande la cadena asignada al objeto. El código crea un `StringBuilder` objeto llamando a su constructor predeterminado (sin parámetros). La capacidad predeterminada de este objeto es de 16 caracteres y su capacidad máxima es superior a 2 mil millones caracteres. Anexando la cadena "Añadimos una oración". da como resultado una nueva asignación de memoria porque la longitud de la cadena (19 caracteres) supera la capacidad predeterminada del `StringBuilder` objeto. La capacidad del objeto se duplica en 32 caracteres, se agrega la nueva cadena y la longitud del objeto ahora es igual a 19 caracteres. A continuación, el código anexa la cadena "Añadimos una oración adicional"

Usamos el espacio de nombre `System.Text` para usar `StringBuilder`. `using System.Text;`

Para añadir caracteres a `StringBuilder` usamos el método `Append` `sb.Append("Añadimos una oración.");`

Para recoger las propiedades del objeto de tipo `StringBuilde`, accedemos a su tipo con `GetType()` y a las propiedades con `GetProperties` `sb.GetType().GetProperties()`. De esta manera las podemos mostrar por pantalla posteriormente.

```
using System;
using System.Text;

namespace EjemploStringBuilder
{
 class Program
 {
 static void Main(string[] args)
 {
 StringBuilder sb = new StringBuilder();
 ShowSBInfo(sb);
 sb.Append("Añadimos una oración.");
 ShowSBInfo(sb);
 for (int ctr = 0; ctr <= 10; ctr++)
 {
 sb.Append("Añadimos una oración adicional.");
 ShowSBInfo(sb);
 }
 }
 }
}
```

```

 }

 private static void ShowSBInfo(StringBuilder sb)
 {
 foreach (var prop in sb.GetType().GetProperties())
 {
 if (prop.GetIndexParameters().Length == 0)
 Console.Write("{0}: {1:N0} ", prop.Name, prop.GetValue(sb));
 }
 Console.WriteLine();
 }
}

```

## Resultado de la ejecución

```

Capacity: 16 MaxCapacity: 2.147.483.647 Length: 0
Capacity: 32 MaxCapacity: 2.147.483.647 Length: 21
Capacity: 64 MaxCapacity: 2.147.483.647 Length: 52
Capacity: 128 MaxCapacity: 2.147.483.647 Length: 83
Capacity: 128 MaxCapacity: 2.147.483.647 Length: 114
Capacity: 256 MaxCapacity: 2.147.483.647 Length: 145
Capacity: 256 MaxCapacity: 2.147.483.647 Length: 176
Capacity: 256 MaxCapacity: 2.147.483.647 Length: 207
Capacity: 256 MaxCapacity: 2.147.483.647 Length: 238
Capacity: 512 MaxCapacity: 2.147.483.647 Length: 269
Capacity: 512 MaxCapacity: 2.147.483.647 Length: 300
Capacity: 512 MaxCapacity: 2.147.483.647 Length: 331
Capacity: 512 MaxCapacity: 2.147.483.647 Length: 362

```

## Constructores para la clase String Builder

### SOBRECARGAS

|                                            |                                                                                                                                                |
|--------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| StringBuilder()                            | Inicializa una nueva instancia de la clase StringBuilder.                                                                                      |
| StringBuilder(Int32)                       | Inicializa una nueva instancia de la clase StringBuilder con la capacidad especificada.                                                        |
| StringBuilder(String)                      | Inicializa una nueva instancia de la clase StringBuilder con la cadena especificada.                                                           |
| StringBuilder(Int32, Int32)                | Inicializa una nueva instancia de la clase StringBuilder que empieza con una capacidad concreta y puede aumentar hasta un máximo especificado. |
| StringBuilder(String, Int32)               | Inicializa una nueva instancia de la clase StringBuilder con la capacidad y la cadena especificadas.                                           |
| StringBuilder(String, Int32, Int32, Int32) | Inicializa una nueva instancia de la clase StringBuilder a partir de la subcadena y la capacidades especificadas.                              |

### Métodos destacables

Append, AppendJoin, AppendLine, Insert: añade elementos a la cadena de caracteres. Inserta cualquier objeto transformado a cadena en la posición especificada.

Clear, Remove: clear deja la cadena vacía quitando todos los caracteres. El método remove elimina desde la posición indicada una subcadena de longitud indicada

CopyTo, Equals: el primero copia la cadena, el segundo compara dos cadenas.

<https://docs.microsoft.com/es-es/dotnet/api/system.text.stringbuilder.append?view=net-5.0>

En el siguiente ejemplo tenéis el uso de algunos de los métodos:

```
using System;
using System.Text;

namespace StringBuilderMetodos
{
 class Program
 {
 static void Main(string[] args)
 {
 StringBuilder stringBuilder = new StringBuilder("Ejemplo inicial");

 stringBuilder.Remove(7, 8);

 Console.WriteLine("Cadena tras remove {0}", stringBuilder);

 if (stringBuilder.Equals(""))
 {
 stringBuilder.Clear();

 Console.WriteLine("Cadena vacía tras clear {0}", stringBuilder);
 }

 stringBuilder.AppendLine("Añadimos línea");

 Console.WriteLine("Cadena tras AppendLine {0}", stringBuilder);

 stringBuilder.AppendJoin(" ", "Añadimos al final con Join");

 Console.WriteLine("Cadena tras AppendJoin {0}", stringBuilder);

 char[] charDest = new char[stringBuilder.Length];
 stringBuilder.CopyTo(0, charDest, charDest.Length);

 Console.WriteLine("Copia la cadena a tipo {0}", charDest);
 }
 }
}
```

## 4 Programación orientada a objetos. Orientacion a objetos en c#

La programación orientada a objetos se basa en cómo, en el mundo real, los objetos a menudo se componen de muchos tipos de objetos más pequeños. Esta capacidad de combinar objetos, sin embargo, es sólo un aspecto muy general de la programación orientada a objetos. La programación orientada a objetos proporciona varios otros conceptos y características para que la creación y el uso de objetos sean más fáciles y flexibles, y la más importante de estas características es la de las clases.

Una clase es una plantilla para varios objetos con características similares. Las clases incorporan todas las características de un conjunto determinado de objetos. Cuando se escribe un programa en un lenguaje orientado a objetos, no se definen objetos reales. Las clases de objetos se definen.

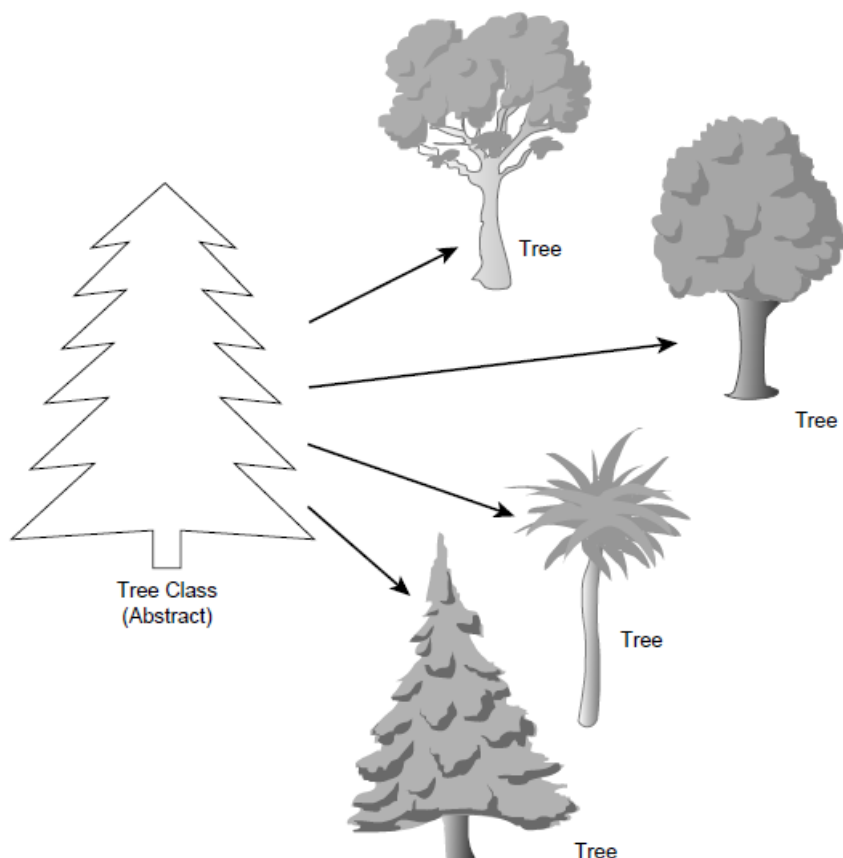
Por ejemplo, podría tener una clase Tree que describa las características de todos los árboles (tiene hojas y raíces, crece, crea clorofila). La clase Tree sirve como un modelo abstracto para el concepto de árbol: para alcanzar y agarrar, o interactuar con, o cortar un árbol, debe tener una instancia concreta de ese árbol. Por supuesto, una vez que tenga una clase de árbol, puede crear muchas instancias diferentes de ese árbol, y cada instancia de árbol diferente puede tener diferentes características (hojas cortas, altas, tupidas, gotas en otoño), mientras sigue comportándose como y siendo inmediatamente reconocible como un árbol.

Una instancia de una clase es otra palabra para un objeto real. Si las clases son una representación abstracta de un objeto, una instancia es su representación concreta. Entonces, ¿cuál es exactamente la diferencia entre una instancia y un objeto? Nada, en realidad. Objeto es el término más general, pero tanto las instancias como los objetos son la representación concreta de una clase.

De hecho, los términos instancian y objeto a menudo se usan indistintamente en el lenguaje de POO (Programación orientada a objetos). Un instancia de un árbol y un objeto de árbol son la misma cosa. En un ejemplo más cercano al tipo de cosas que podría querer hacer en la programación Java, puede crear una clase para el elemento de la interfaz de usuario denominada botón.

La clase Button define las características de un botón (su etiqueta, su tamaño, su apariencia) y cómo se comporta (¿necesita un solo clic o un doble clic para activarlo, cambia de color cuando se hace clic, qué hace cuando se activa?). Una vez definida la clase Button, puede crear fácilmente instancias de ese botón (es decir, objetos de botón) que todos toman las características básicas del botón según lo definido por la clase, pero que pueden tener

diferentes apariencias y comportamientos en función de lo que desea que haga ese botón en particular. Al crear una clase, no tiene que seguir reescribiendo el código para cada botón individual que desee usar en el programa, y puede reutilizar la clase **Button** para crear diferentes tipos de botones a medida que los necesite en este programa y en otros programas.



#### 4.1 Pilares de la programación orientada a objetos

La **programación orientada a objetos** se basa en **cuatro pilares**, conceptos que la diferencian de otros paradigmas de programación. Estos pilares, algunos principios que daremos a conocer en unidades posteriores, reglas y convenciones para la programación orientada a objetos se originan a partir de principios de los años 90.

Las compañías de software y los ingenieros de software buscan código de alta calidad para ofrecer mejores productos de software y reducir el precio del mantenimiento y la fabricación.

En esa búsqueda, Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides escribieron un libro *Design Patterns: Elements of Reusable Object-Oriented Software*. Ha sido influyente en el campo de la ingeniería de software y es considerado como una fuente importante para la

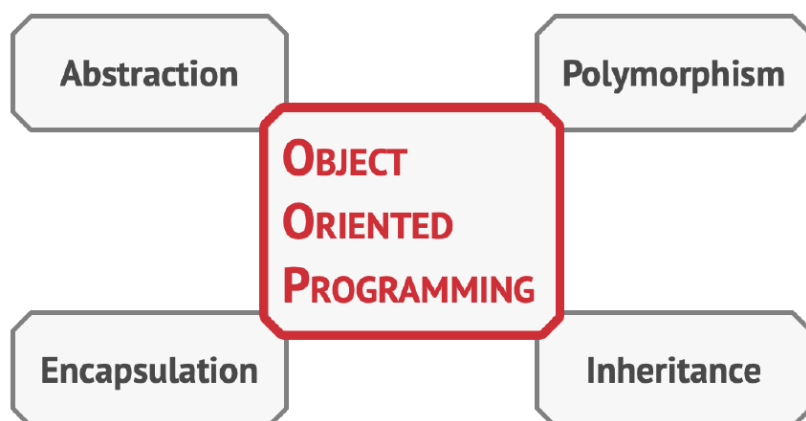
teoría y la práctica del diseño orientado a objetos. Se han vendido más de 500.000 ejemplares en inglés y en otros 13 idiomas.

Asimismo, en los años 90 se celebraron muchas conferencias sobre ingeniería de software, para mejorar la producción de software y sistemas de información. Seguiremos tantas de estas "reglas" que la mayoría de los desarrolladores y académicos reconocidos y admirados en Ciencias de la Computación ha fijado desde ese período.

Los conceptos de OOP nos permiten crear interacciones específicas entre objetos Java. Permiten reutilizar el código sin crear riesgos de seguridad ni hacer que un programa Java sea menos legible.

Aquí están los cuatro principios principales con más detalle.

1. abstracción
2. encapsulación
3. herencia
4. polimorfismo

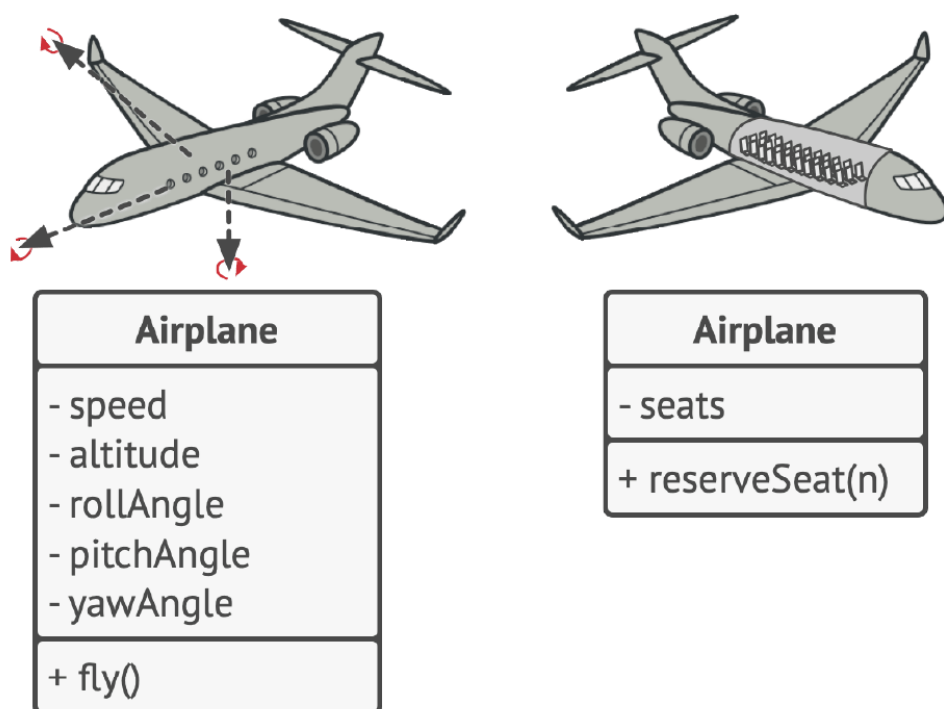


#### 4.1.1 Abstraction

La **abstracción** es un **modelo de un objeto o fenómeno del mundo real**, limitado a un contexto específico, que representa todos los detalles relevantes para este contexto con alta precisión y omite todo lo demás.

La mayoría de las veces, cuando se crea un programa con POO, se da forma a los objetos del programa en función de objetos del mundo real. Sin embargo, **los objetos del programa no representan los originales con una precisión del 100%** (y rara vez se requiere que lo hagan). En su lugar, los objetos solo *modelan* atributos y comportamientos de objetos reales **en un contexto específico**, omitiendo el resto.

Por ejemplo, una clase Airplane probablemente podría existir tanto en un simulador de vuelo como en una aplicación de reserva de vuelos. Pero en el primer caso, albergaría detalles relacionados con el vuelo real, mientras que en la segunda clase solo le importaría el mapa de asientos y qué asientos están disponibles.



#### 4.1.2 Encapsulación

Para arrancar el motor de un coche, sólo tiene que girar una tecla o pulsar un botón. No es necesario conectar cables bajo el capó, girar el cigüeñal y los cilindros, e iniciar el ciclo de potencia del motor. Estos detalles se esconden bajo el capó del coche. Solo tienes una interfaz simple: un interruptor de arranque, un volante y algunos pedales. Esto ilustra cómo cada objeto tiene una interfaz, una parte pública de un objeto, abierta a interacciones con otros objetos.

La **encapsulación** es la característica de cada módulo en nuestro código para **ocultar datos u operaciones que no son necesarias para revelar a otros módulos de nuestro código**. Por ejemplo, suponiendo que tengo una llamada de módulo StoreEmployees que almacena datos de empleados. Otros módulos de nuestra aplicación, como windows (la interfaz gráfica)



utilizarán este módulo para guardar datos. Sin embargo, esta interfaz gráfica no necesita saber si StoreEmployees guarda los datos en un archivo o una base de datos. Esos detalles de implementación están ocultos, encapsulados. Por lo tanto, el Window sólo necesita una función storeEmployeeData(Employee data) para utilizar esta funcionalidad.

La **encapsulación en la programación orientada a objetos** es la **capacidad de un objeto para ocultar partes de su estado y comportamientos de otros objetos, exponiendo sólo una interfaz limitada** al resto del programa.

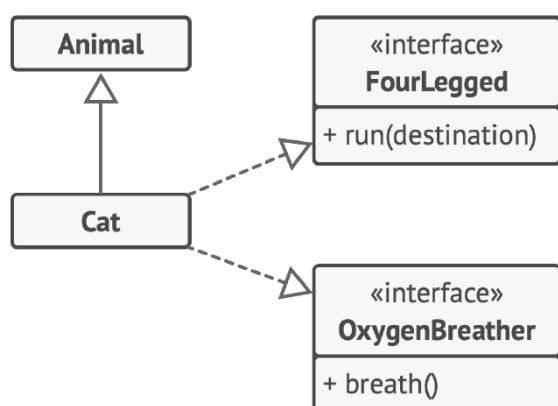
Encapsular algo significa hacerlo **private** y, por lo tanto, accesible solo desde dentro de los métodos de su propia clase.

Hay un modo un poco menos restrictivo llamado **protected** que hace que un miembro de una clase también esté disponible para las subclases. Las interfaces y las clases/métodos abstractos de la mayoría de los lenguajes de programación se basan en los conceptos de abstracción y encapsulación.

### 4.1.3 Herencia

La **herencia** es la **capacidad de construir nuevas clases sobre las existentes**. La principal ventaja de la herencia es la reutilización de código. **Si se desea crear una clase que es ligeramente diferente de una existente, no es necesario duplicar el código**. En su lugar, se extiende la clase existente y se coloca la funcionalidad adicional en una subclase resultante, que hereda los campos y métodos de la superclase.

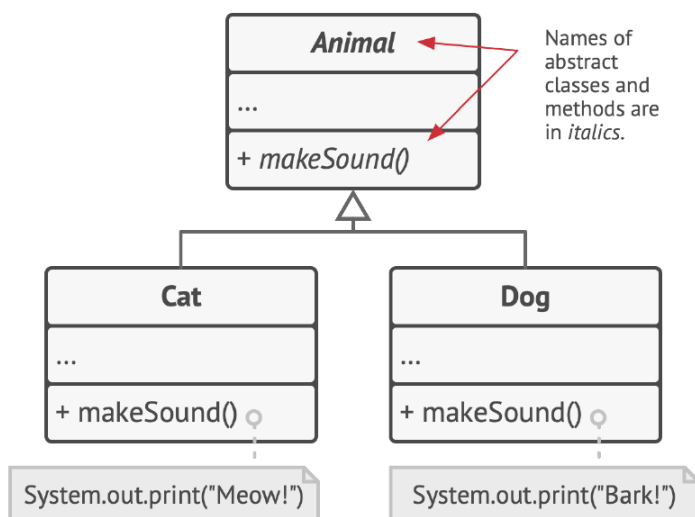
La consecuencia de utilizar la herencia es que las subclases tienen la misma interfaz que su clase primaria. No se puede ocultar un método en una subclase si se declaró en la superclase. Tú también debe implementar todos los métodos abstractos, incluso si no tienen sentido para la subclase.



#### 4.1.4 Polimorfismo

La palabra **polimorfismo** proviene de las palabras griegas para "muchas formas". Un **método polimórfico, por ejemplo, es un método que puede tener diferentes formas**, donde "forma" se puede considerar como tipo o comportamiento.

Veamos algunos ejemplos de animales. La mayoría **de los animales** pueden hacer sonidos. Podemos anticipar que todas las subclases necesitarán reemplazar el método **makeSound** base para que cada subclase pueda emitir el sonido correcto; por lo tanto, podemos declararlo *abstracto* de inmediato. Esto nos permite omitir cualquier implementación predeterminada del método en la superclase, pero forzar a todas las subclases a crear las suyas propias.



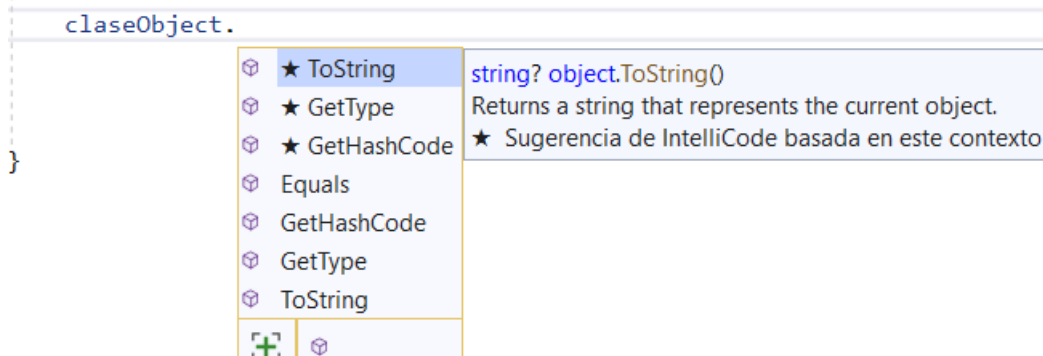
#### 4.2 La clase Object en C#

Es la clase base de la que heredan todas las clases creadas en C#. Admite casteo de todas las clases de la jerarquía de clases de .NET y proporciona servicios de bajo nivel a clases derivadas. Se trata de la clase base fundamental de todas las clases de .NET; es la raíz de la jerarquía de tipos.

Implementa cuatro métodos básicos que son sobrescribibles por todas las subclases de la jerarquía:

- **ToString**: que transforma el objeto a String.
- **GetType**: que devuelve el tipo del objeto, en este caso Object,
- **GetHashCode**: que devuelve o genera un **HashCode** que va a identificar de manera unívoca a ese objeto respecto al resto. Además permite la ordenación y el acceso aleatorio en ciertas colecciones C# como mapas.
- **Equals**: que indica si dos objetos son iguales o no.

Es recomendable sobrescribir estos métodos en nuestro modelo junto con el interface IComparable si vamos a manejar nuestra jerarquía de clases con colecciones.



En el siguiente ejemplo podéis ver el uso de Object y sus métodos. Se puede observar que :

- Equals: Dos variables Objects son iguales si apuntan a la misma referencia. Esto quiere decir que cada objeto creado es único. Por eso Object1 y Objec2 su Equals es false, y Object 1 y Object3 son iguales porque apuntan al mismo objeto.
- GetType: devuelve el tipo del objeto System.Object.
- ToString: devuelve el mismo resultado que GetType.
- GetHashCode: es único para objeto de tipo object creado.

Todos estos métodos son sobrescribibles como realizaremos posteriormente en algunos ejemplos.

## Ejecución

```
Object1 System.Object es igual a Object2 System.Object : False
Object1 System.Object es igual a Object3 System.Object : True
Object1 hash 58225482, Object2 hash 54267293, Object3 hash 58225482
Object1 getType System.Object
```

```
using System;

namespace ClaseObject
{
 class Program
 {
 static void Main(string[] args)
 {
 Object Object1 = new object();
 Object Object2 = new object();
 Object Object3 = Object1;
 }
 }
}
```

```

 Console.WriteLine("Object1 {0} es igual a Object2 {1} : {2}", Object1.ToString(),
Object2.ToString(), Object1.Equals(Object2));
 Console.WriteLine("Object1 {0} es igual a Object3 {1} : {2}", Object1.ToString(),
Object3.ToString(), Object1.Equals(Object3));

 Console.WriteLine("Object1 hash {0}, Object2 hash {1}, Object3 hash {2}",
Object1.GetHashCode(), Object2.GetHashCode(), Object3.GetHashCode());

 Console.WriteLine("Object1 getType {0}", Object1.GetType());
 }
}
}

```

### 4.2.1 Ejercicio. Cornell notes

String como todos los objetos de C# Hereda de Objects. Dada las siguientes variables cadenas

```

String cadena1 = "MiCadena";

String cadena2 = "Otra Cadena";
String cadena3 = "MiCadena";

```

Probar los cuatros métodos que hereda de object y comparar resultados.

## 4.3 Clases en c#

En C# los elementos que definen una clase son:

- Atributos, métodos y constructores, que funcionan de manera parecida a Java y todavía más a C++
- La declaración de una clase comparte aspectos en común con Java y C++:
- La declaración de una clase incluye la definición en implementación ( igual que en Java ).
- Un fichero de código fuente (extensión .cs) puede contener la declaración de varias clases ( igual que en C++ ).

C# añade dos nuevos tipos de declaraciones:

**Propiedades:** Representan características de los objetos que son accedidas como si fueran atributos. Características

- Ocultan el uso de métodos get/set.
- Una propiedad puede representar un atributo calculado.

**Eventos:** Notificaciones que envía un objeto a otros objetos cuando se produce un cambio de estado significativo (basado en el patrón de diseño Observer).

Tanto las propiedades y eventos son el soporte para el Desarrollo de Software basado en Componentes.

Sintaxis general:

```
[atributos] [modificadores] [parcial] class NombreDeLaClase [: clase base] [, interfaz1,
interfaz2, ...] {
 Código de la clase
}
```

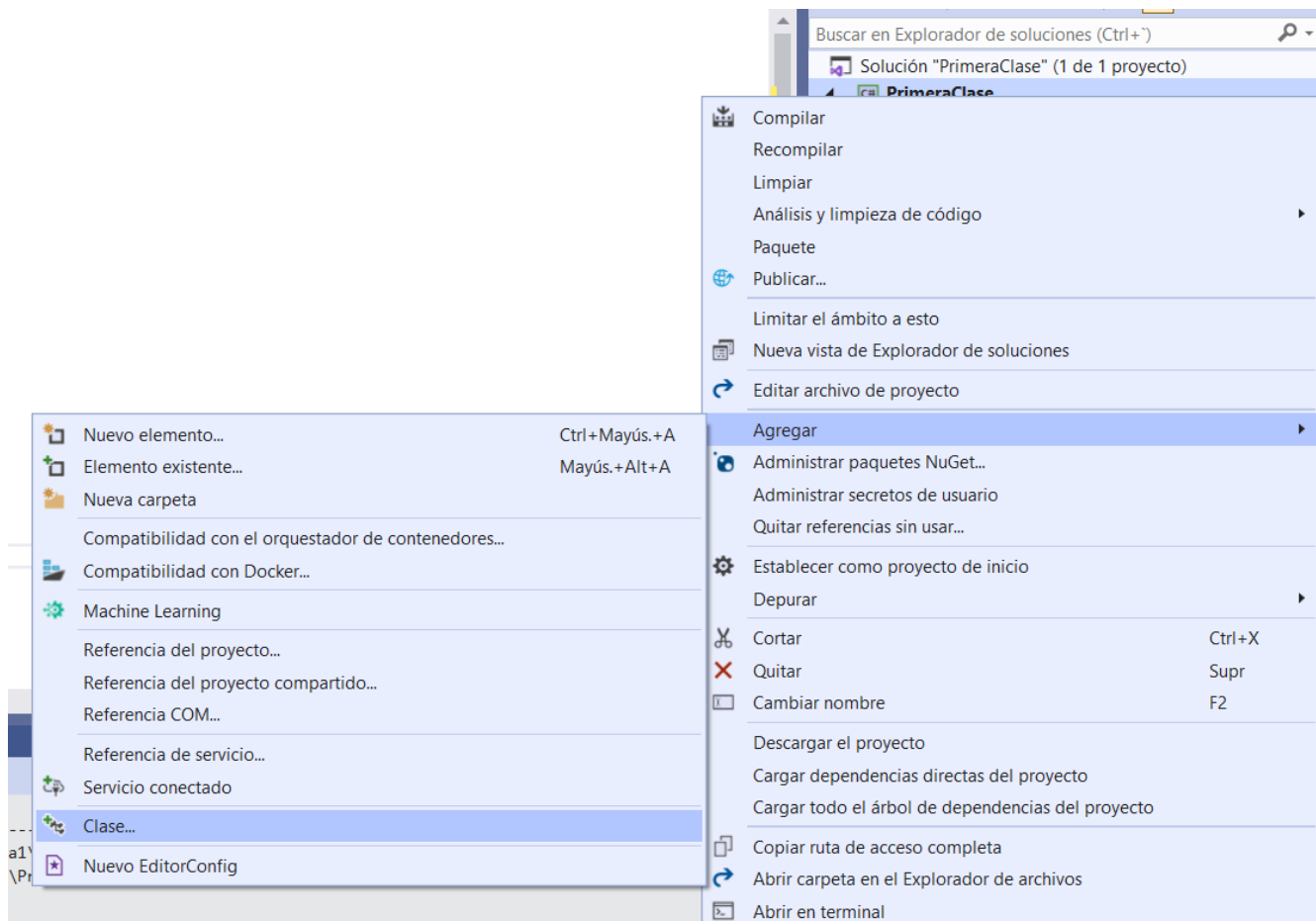
Para determinar la visibilidad el lenguaje cuenta con las siguientes palabras clave:

1. **public**: la clase puede ser utilizada en **cualquier** proyecto.
2. **internal**: la clase está limitada al proyecto en el cual está definida.
3. **private**: la clase sólo puede usarse en el módulo en la que está definida.
4. **protected**: la clase sólo puede ser utilizada en una subclase. Es decir sólo se puede utilizar **protected** para una clase declarada en otra clase.
5. **protected internal**: lo mismo que **internal** + **protected**.
6. **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.
7. **sealed**: cuando una clase es la última de una jerarquía, por lo que no podrá ser utilizada como base de otra clase.

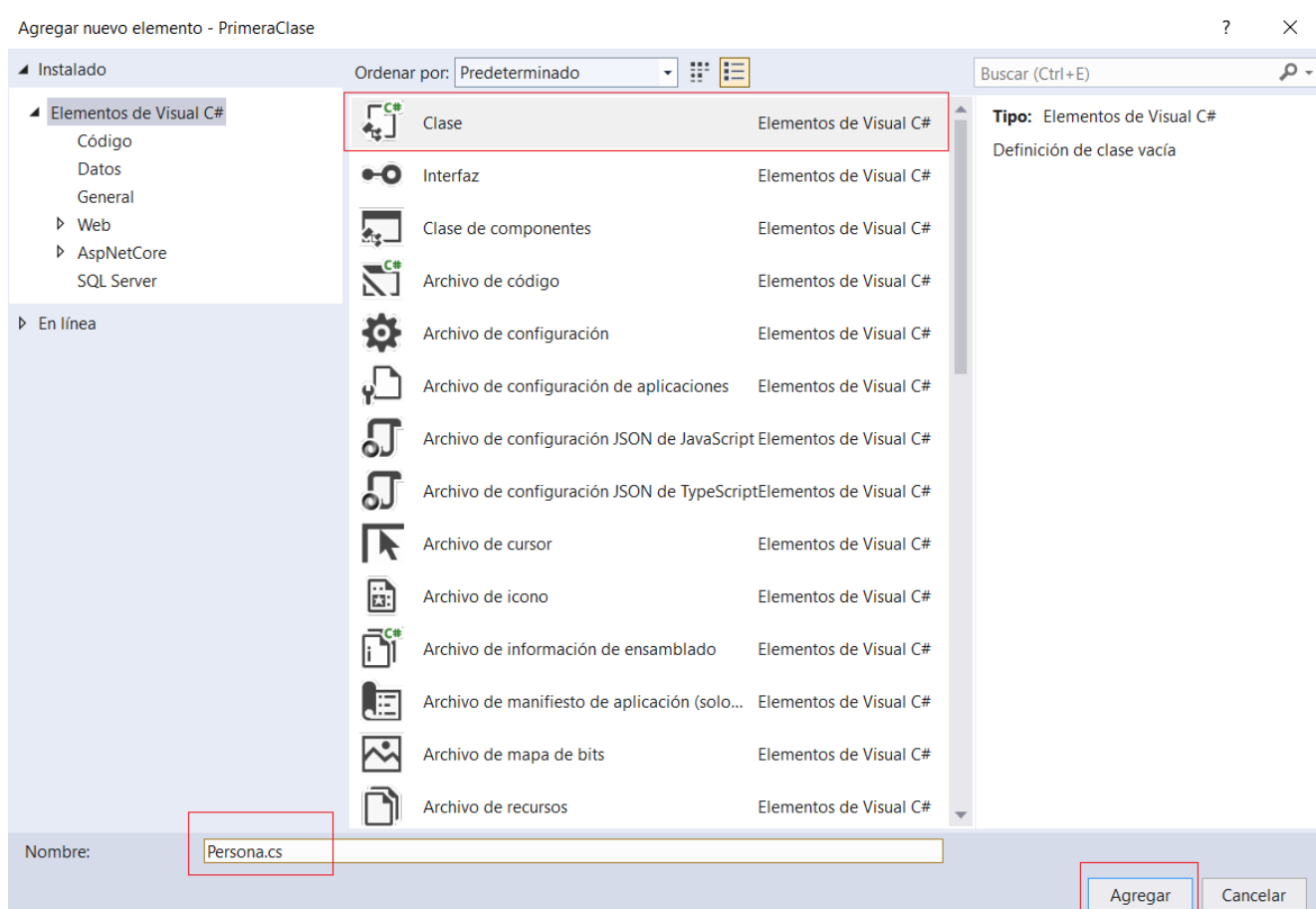
### 4.3.1 Creando la primera clase

Vamos a empezar a crear un primer modelo de clases sencillo en c# para familiarizarnos con el entorno de desarrollo, siguiendo los siguientes pasos:

1. Creamos un proyecto **PrimeraClase** de tipo consola c# en Visual Studio.
2. Añadimos una clase. Para ello, hacemos botón derecho sobre el proyecto en el explorador de soluciones, elegimos el menú agregar y clase.



3. Elegimos clase c#, le damos de nombre Persona.cs al fichero y pulsamos agregar.



4. Sustituir la clase Persona por la que se os proporciona en los apuntes. Damos dos versiones la primera es como se trabaja habitualmente en C#. La segunda se ajusta más a la manera tradicional de realizar propiedades privadas y métodos públicos de orientación a objetos. Usaremos la primera.

## Clases en C# con propiedades public y descriptor.

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

namespace PrimeraClase
{
 public class Persona
 {
 public String Apellido { get; set; }
 public String Nombre { get; set; }
 public DateTime FechaNac { get; set; }

 public Persona()
 {

```

```
}

}
}
```

Normalmente se desarrollan las clases de esta manera en C#. Fijaos en las propiedades. Se define la propiedad como publica y se adjunta a ella los modificadores de acceso get; y set;

```
public String Apellido { get; set; }
```

Con esto estamos permitiendo que se acceda desde cualquier objeto a la propiedad para lectura y escritura. Si creo un objeto de tipo persona

```
Persona p = new Persona();
```

Poder escribir y leer de la propiedad

Escritura (setter)  
p.Apellido="Lopez";

Lectura(getter)

```
String cadena = p.Apellido;
```

## Clases con encapsulación de propiedades

También podría definir mis propiedades como privadas y luego definir un descriptor publico de acceso. De esta manera los descriptors deben tener cuerpo, código.

```
public class Persona
{
 private String elApellido;
 private String elNombre;
 private DateTime laFechaNac;
 public String Apellido
```



```

 {
 get { return elApellido; }
 set { elApellido = value.ToUpper(); }
 }
 public String Nombre
 {
 get { return elNombre; }
 set { elNombre = value.ToLower(); }
 }
 public DateTime FechaNac
 {
 get { return laFechaNac; }
 set
 {
 if (value.Year >= 1900)
 {
 laFechaNac = value;
 }
 }
 }
}

```

Y vamos a analizar el código paso a paso:

- La clase tal cual no tiene constructor añadiremos uno posteriormente. Cuando no hay constructor, igual que en Java, esta disponible el constructor por defecto sin parámetros.
- Para cada propiedad se define un método get y set publico. Las propiedades se mantienen como privadas. Los getters y setters son diferentes a como estáis acostumbrados en Java. Como veis después de la propiedad entre llaves se añaden los dos métodos, get y set. **En C# esto se denomina encapsular una propiedad.**

```
private String elApellido;
```

```

public String Apellido
{
 get { return elApellido; }
 set { elApellido = value.ToUpper(); }
}

```

### 4.3.2 Modificando la clase Persona

Para ver como funcionan las clases en C# vamos a añadirle unos cuantos elementos a esta clase. Un constructor, una propiedad calculada nueva de sólo lectura, y un método ToString. Un operador muy importante en C# al igual que en Java y C++ es el operador **this**. Básicamente hace referencia al objeto actual, a si mismo. De esta manera podemos acceder a métodos y propiedades del objeto en cualquier parte de su propio código. En el siguiente código llamamos al setter de edad desde el constructor **this.Edad**

Añadimos la propiedad privada laEdad. Lo realizamos de esta manera porque la propiedad Edad es autocalculada depende de Fecha de nacimiento. Entonces definimos la propiedad como privada y asignamos un valor en el descriptor público

```
private int laEdad;
```

La hacemos accesible en sólo lectura no definiendo públicamente un set. Fijaos como tiene sólo un set como propiedad.

```
public int Edad
{
 get { return DateTime.Now.Year - FechaNac.Year; }
}
```

Vamos a probar los métodos get/set en el nuevo constructor. El nuevo constructor tiene la siguiente forma:

```
public Persona(String Apellido, String Nombre, DateTime FechaNac)
{
 this.Apellido = Apellido;

 this.Nombre = Nombre;
 this.FechaNac = FechaNac;

 laEdad = this.Edad;
}
```

Estamos usando los métodos públicos set para Apellido Nombre y FechaNac. Como veis los getters y setters en c# se usan como si fueran propiedades públicas. `this.Apellido = Apellido;` es el set para la propiedad privada elApellido.

Como la Edad no tiene set, debemos acceder a la propiedad privada laEdad directamente.

```
private int laEdad;
```

```
laEdad = this.Edad;
```

No podemos hacer una asignación a Edad porque no tiene setter, no tiene descriptor de acceso para escritura.

```
This.Edad = Edad; NO SE PUEDE
```

Pero si os fijais en el método ToString, si puedo acceder al valor de Edad porque tiene un get definido.

```
public override string ToString()
{
 return String.Format("Apellidos: {0}, Nombre: {1}, FechaNac: {2}, Edad: {3}", this.Apellido, this.Nombre, this.FechaNac, this.Edad);
}
```

Código de la clase Persona modificado.

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

namespace PrimeraClase
{
 public class Persona
 {

 public String Apellido { get; set; }
 public String Nombre { get; set; }
 public DateTime FechaNac { get; set; }
 private int laEdad;

 public Persona()
 {

 }

 public Persona(String Apellido, String Nombre, DateTime FechaNac)
 {

 this.Apellido = Apellido;
 this.Nombre = Nombre;
 this.FechaNac= FechaNac;

 laEdad = this.Edad;

 }

 public int Edad
 {

 get { return DateTime.Now.Year - FechaNac.Year; }

 }

 public override string ToString()
 {
```

```

 return String.Format("Apellidos: {0}, Nombre: {1}, FechaNac: {2:d}, Edad: {3} ",
this.Apellido, this.Nombre, this.FechaNac, this.Edad);
 }
}
}

```

### 4.3.3 Creando un objeto de tipo Persona en el programa principal.

En nuestro programa principal vamos a crear el primer objeto de tipo Persona añadiendo este código.

```

static void Main(string[] args)
{
 Persona persona;

 persona = new Persona("Perez Lopez", "Rodolfo", new DateTime(1990,6,12));

 Console.WriteLine("La persona creada tiene las siguientes características: {0}" ,
persona);
}

```

El resultado de la ejecución sería:

La persona creada tiene las siguientes características: Apellidos: PEREZ LOPEZ, Nombre: rodolfo, FechaNac: 12/06/1990, Edad: 31

### 4.3.4 Ejercicio practico

Crear una persona con un constructor sin parámetros modificando las propiedades posteriormente. Realizar los cambios necesarios en la clase Persona para poder crear un objeto sin pasar parámetros al constructor.

## 4.4 Herencia en c#

Como en todos los lenguajes de programación cuando una clase hereda de otra, hereda todos sus métodos y propiedades. Para explicar la herencia en C# vamos a usar el siguiente ejemplo de la clase Empleado que hereda de persona y explicaremos paso a paso las principales características.

Para que una clase herede de otra se usa el operador : seguido del nombre de la clase base.

```
public class Empleado : Persona
```

Para llamar al constructor de la clase base con parámetros se usa el operador : seguido del operador base y los parámetros pasados en la signature de la función.

```
public Empleado(String Apellido, String Nombre, DateTime FechaNac, double elSalario):
base(Apellido, Nombre, FechaNac) {
```

Igualmente se puede usar el operador base para acceder a cualquier método de la clase padre de la que hereda. Observar como en el método ToString llamamos al método ToString de la clase padre con el operador base.

```
public override string ToString()
{
 return String.Format("{0} Salario: {1}", base.ToString(), this.Salario);
}
```

Por último encapsulamos la propiedad salario, como hemos visto anteriormente para persona.

```
public double Salario { get { return salario; } set { salario = value; } }
```

Código de la clase Empleado.

```
public class Empleado : Persona
{

 public double Salario { get; set; }

 public Empleado()
 {

 }

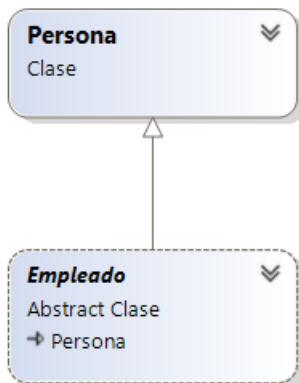
 public Empleado(String Apellido, String Nombre, DateTime FechaNac, double elSalario):
base(Apellido, Nombre, FechaNac) {

 this.Salario = elSalario;

 }

 public override string ToString()
 {
 return String.Format("{0} Salario: {1}", base.ToString(), this.Salario);
 }
}
```

El diagrama de clases asociado :



#### 4.4.1 Ejercicio.

Crear un objeto de tipo Empleado con salario 2000. Elegid los apellidos, nombre y fecha de nacimiento.

### 4.5 Ampliación de métodos. Paso por referencia y por valor

**Parámetros formales:** Son las variables que recibe la función, se crean al definir la función. Los **parámetros formales** son variables locales dentro de la función.

**Parámetros reales:** Son las expresiones que se utilizan en la llamada de la función, sus valores se copiarán en los **parámetros formales**.

Por defecto si pasamos un parámetro a un método que es un tipo primitivo se pasa por valor, es decir, se hace una copia del valor parámetro real, la variable pasada como parámetro, en el parámetro formal, el de dentro de la función.

Cuando pasamos un objeto, el paso es por referencia se pasa su posición de memoria. Si modifico un objeto dentro de una función se modificará el objeto parámetro real. En el siguiente ejemplo. Al modificar el parámetro p en función 2, se modificará el objeto pasado como parámetro real persona `funcion2(persona,numRetorno2)=` cambiando el nombre Juno por Lino.

Sin embargo, `int num` es un tipo básico pasado por valor, aunque le asignemos un 5, no modificará su valor de 1. En el paso por valor se hace una copia del valor original pasado 1.

```
public static bool funcion2(Persona p, int num)
{
```

```

 p.Nombre = "Lino";
 num = 5;
 return true;
 }

```

## Llamada a la función

```

Persona persona = new Persona("Lino", "Lopez");

int numRetorno2=1;

bool retorno2 = funcion2(persona, numRetorno2);

Console.WriteLine(" El valor de numRetorno2 sigue siendo " + numRetorno2);

Console.WriteLine(" El valor retornado por la funcion booleano es " + retorno2);

Console.WriteLine("persona se ha modificado dentro del metodo pues es una direccion
lo que se pasa por referencia " + persona);

```

## Ejecución:

El valor del numRetorno2 sigue siendo 1

El valor retornado por la funcion booleano es True

persona se ha modificado dentro del metodo pues es una direccion lo que se pasa por referencia nombre: Juno apellido: Lopez

Usamos el modificador in para indicar que el método no va a modificar el objeto pasado como parámetro, que se puede pasar de forma segura. En función hemos declarado el parámetro Persona p con el modificador in, para indicar esta situación.

El segundo parámetro formal con modificador out num, aunque es un tipo primitivo y se pasan por valor por defecto al llevar el modificador out delante se esta pasando por referencia. El valor que le demos num lo recogerá la variable que hemos pasado como parámetro real numRetorno. En objetos el out indicará al usuario que el parámetro pasado va a colocar un objeto nuevo, va a hacer un new crear un nuevo objeto y asignarlo a nuestro parámetro, que a su vez pondrá esa dirección en la variable. Se perdería el objeto original pasado como parámetro.

```

public static bool funcion(in Persona p, out int num)
{
 num = 5;
 return true;
}

```

En la llamada en el tipo primitivo tendremos que usar el modificador out igualmente

```
bool retorno = funcion(persona, out numRetorno);
```

Después de la llamada numRetorno tendrá valor 5. El valor que devuelve la función es un booleano, true. De esta manera podemos devolver múltiples valores en un solo método.

```
bool retorno = funcion(persona, out numRetorno);

Console.WriteLine(" El valor del parametro con out devuelto es " + numRetorno);

Console.WriteLine(" El valor retornado por la funcion booleano es " + retorno);

Console.WriteLine(" persona no se ha modificado dentro del metodo " + persona);
```

## Ejecución

El valor del parametro con out devuelto es 5

El valor retornado por la funcion booleano es True

persona no se ha modificado dentro del metodo nombre: Lino apellido: Lopez

## Nota:

Persona dentro del método nombre: Juno apellido: Lopez

Persona parametro no se puede modificar dentro del metodo por modificador in nombre:

Lino apellido: Lopez

El valor del parametro con out devuelto es 5

El valor retornado por la funcion booleano es True

## Ejemplo completo

```
using System;

namespace ModificadorInOut
{
 class Persona
 {
 private string nombre;
 private string apellido;
 public Persona(string nombre, string apellido)
 {
 this.nombre = nombre;
 this.apellido = apellido;
 }

 public string Nombre { get { return this.nombre; } set { this.nombre = value; } }

 public override string ToString()
 {
```



```

 return "nombre: " + nombre + " apellido: " + apellido;
 }
}

class Program
{
 public static bool funcion(in Persona p, out int num)
 {
 num = 5;
 return true;
 }

 public static bool funcion2(Persona p, int num)
 {
 p.Nombre = "Juno";

 num = 5;
 return true;
 }

 static void Main(string[] args)
 {
 Persona persona = new Persona("Lino", "Lopez");
 int numRetorno=1;

 bool retorno = funcion(persona, out numRetorno);

 Console.WriteLine(" El valor del parametro con out devuelto es " + numRetorno);
 Console.WriteLine(" El valor retornado por la funcion booleano es " + retorno);
 Console.WriteLine(" persona no se ha modificado dentro del metodo " + persona);

 int numRetorno2=1;

 bool retorno2 = funcion2(persona, numRetorno);

 Console.WriteLine(" El valor de numRetorno2 sigue siendo " + numRetorno2);
 Console.WriteLine(" El valor retornado por la funcion booleano es " + retorno2);

 Console.WriteLine("persona se ha modificado dentro del metodo pues es una
direccion lo que se pasa por referencia " + persona);
 }
}

```

### 4.5.1 Ejercicio

Crear un método `funcion3(out Persona p, out num)` que cree una nueva persona Perico Palotes, lo asigne a el parámetro, comprobar los resultados, si realmente en la llamada la variable persona contiene un nuevo objeto, diferente al que se paso como parámetro.

```
bool retorno3 = funcion2(persona, numRetorno3);
```

## 4.6 Polimorfismo y clases abstractas en C#

Ya hemos visto que un **método polimórfico**, por ejemplo, es un método que puede tener **diferentes formas**, donde "forma" se puede considerar como tipo o comportamiento. En el ejemplo anterior hemos sobrescrito el método ToString() usando el modificador override, modificando el comportamiento para este método para que añada el salario como parte de la conversión a cadena.

```
public override string ToString()
{
 return String.Format("{0} Salario: {1}", base.ToString(), this.Salario);
}
```

Vamos a indagar un poco más en polimorfismo y clases abstractas en C#, haciendo que la clase Empleado sea abstracta y añadiendo algún método abstracto. De esta manera las clases descendientes que heredan de empleado implementarán cada una su versión de estos métodos abstractos usando la característica de polimorfismo anteriormente referida.

Declaramos la clase como abstracta

```
public abstract class Empleado : Persona
```

Añadimos dos métodos abstractos a implementar por las clases descendientes:

```
public abstract double calculoImpuestos();

public abstract double salarioNeto();
```

La clase Empleado modificada

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

namespace PrimeraClase
{
 public abstract class Empleado : Persona
 {
 private double salario;
```

```

 public Empleado()
 {

 }

 public Empleado(String Apellido, String Nombre, DateTime FechaNac, double elSalario):
 base(Apellido, Nombre, FechaNac) {

 this.Salario = elSalario;

 }

 public abstract double calculoImpuestos();

 public abstract double salarioNeto();

 public double Salario { get { return salario; } set { salario = value; } }

 public override string ToString()
 {
 return String.Format("{0} Salario: {1}", base.ToString(), this.Salario);
 }
}

```

Y creamos una nueva clase que herede de Empleado llamada Administrativo obligada a implementar los dos métodos abstractos, `calculoImpuestos()` y `salarioNeto()` .

```

using System;
using System.Collections.Generic;
using System.Text;

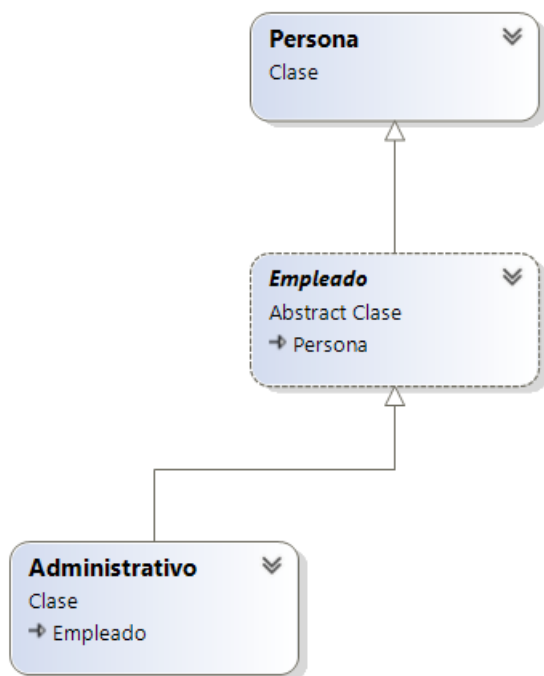
namespace PrimeraClase
{
 class Administrativo : Empleado
 {
 private const double PORCENTAJEIMPUESTOS = 0.10;

 public override double calculoImpuestos()
 {
 return PORCENTAJEIMPUESTOS*this.Salario;
 }

 public override double salarioNeto()
 {
 return this.Salario -this.calculoImpuestos();
 }
 }
}

```

## Diagrama de clases



### 4.6.1 Interfaces

Para declarar un interfaz usamos el modificador interfaz en la declaración y definimos sus métodos abstractos, en este caso calculoBonus() donde no necesitamos usar el modificador abstract.

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

namespace PrimeraClase
{
 public interface IBonus
 {
 public double calculoBonus();
 }
}
```

Creamos una nueva clase directivo que hereda de Empleado e implementa IBonus. Como estamos heredando de una clase e implementando un interfaz usamos el operador ":" y el nombre de la clase y del interfaz separado por comas. Podríamos añadir mas interfaces en esta parte de la declaración: `class Ejecutivo : Empleado, IBonus`.

Implementamos el método abstracto de IBonus calculoBonus `public double calculoBonus()` y los dos métodos abstractos de la clase Empleado, es decir damos una comportamiento propio a cada subclase que generamos sobre los mismos métodos con el mismo interfaz.

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

namespace PrimeraClase {
 class Ejecutivo : Empleado, IBonus
 {
 private const double PORCENTAJEIMPUESTOS = 0.30;

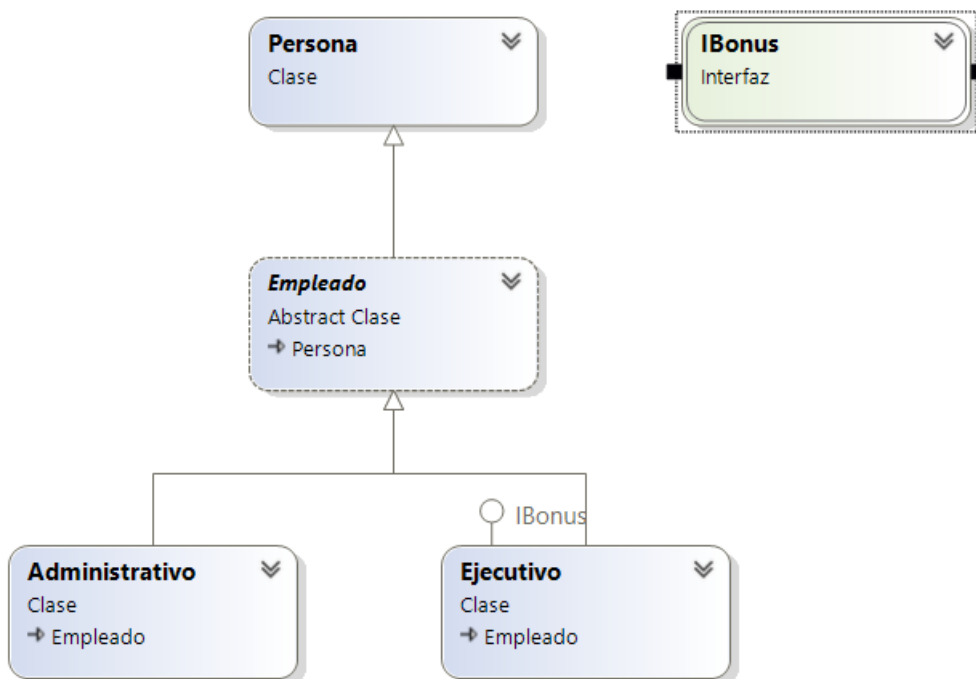
 private const double BONUS = 400;

 public double calculoBonus()
 {
 return BONUS;
 }

 public override double calculoImpuestos()
 {
 return (this.Salario + BONUS) * PORCENTAJEIMPUESTOS;
 }

 public override double salarioNeto()
 {
 return Salario + calculoBonus() - calculoImpuestos();
 }
 }
}
```

Diagrama de clases



## 4.7 Clases e interfaces genéricos

Tanto para clases como para interfaces podemos definir tipos genéricos. Un tipo cuando se define cómo genérico funciona como un parámetro junto con el operador `<>` para pasar un tipo específico como parámetro a la hora de crear un objeto de una clase.

Las clases genéricas encapsulan operaciones que no son específicas de un tipo de datos determinado. El uso más común de las clases genéricas es con colecciones como listas vinculadas, tablas hash, pilas, colas y árboles, entre otros. Las operaciones como la adición y eliminación de elementos de la colección se realizan básicamente de la misma manera independientemente del tipo de datos que se almacenan.

Por ejemplo creamos una clase genérica

```
class BaseNodeGeneric<T> { }
```

Podemos Crear un objeto de tipo `BaseNodeGeneric<int>`

```
BaseNodeGeneric<String> node1 = new BaseNodeGeneric<String>();
```

Una segunda clase que hereda de `BaseNodeGeneric` pero con un tipo específico `int`

```
class Node1 : BaseNodeGeneric<int> { }
```

O podemos definir más de un tipo genérico

```
class BaseNodeMultiple<T, U> { }
```

```
BaseNodeMultiple <String,int> node1 = new BaseNodeGeneric<String,int>();
```

Incluso existen una serie de restricciones para hacer más concretos los tipos genéricos. En el siguiente estamos indicando que T debe implementar Comparable y además debe ser una referencia, un objeto, al que se le puede hacer new ya que debe poseer un constructor sin parámetros público. En este caso obligamos a que el tipo T implemente IComparable. Si escribiéramos una clase , obligaríamos a que el tipo T heredaría de esa clase.

```
class NodeItem<T> where T : System.IComparable<T>, new() { }
```

En la siguiente tabla exponemos todas las restricciones posibles para tipos genéricos.

|                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| where T<br>struct    | El argumento de tipo debe ser un tipo de valor que no acepta valores NULL. Para más información sobre los tipos de valor que admiten un valor NULL, consulte Tipos de valor que admiten un valor NULL. Todos los tipos de valor tienen un constructor sin parámetros accesible, por lo que la restricción struct implica la restricción new() y no se puede combinar con la restricción new(). No puede combinar la restricción struct con la restricción unmanaged. |
| where T<br>class     | El argumento de tipo debe ser un tipo de referencia. Esta restricción se aplica también a cualquier clase, interfaz, delegado o tipo de matriz. En un contexto que admite un valor NULL en C# 8.0 o versiones posteriores, T debe ser un tipo de referencia que no acepte valores NULL.                                                                                                                                                                              |
| where T<br>class?    | El argumento de tipo debe ser un tipo de referencia, que acepte o no valores NULL. Esta restricción se aplica también a cualquier clase, interfaz, delegado o tipo de matriz.                                                                                                                                                                                                                                                                                        |
| where T<br>notnull   | El argumento de tipo debe ser un tipo que no acepta valores NULL. El argumento puede ser un tipo de referencia que no acepta valores NULL en C# 8.0 o posterior, o bien un tipo de valor que no acepta valores NULL.                                                                                                                                                                                                                                                 |
| where T<br>default   | Esta restricción resuelve la ambigüedad cuando es necesario especificar un parámetro de tipo sin restricciones al invalidar un método o proporcionar una implementación de interfaz explícita. La restricción default implica el método base sin la restricción class o struct. Para obtener más información, vea la propuesta de especificación de la restricción default.                                                                                          |
| where T<br>unmanaged | El argumento de tipo debe ser un tipo no administrado que no acepta valores NULL. La restricción unmanaged implica la restricción struct y no se puede combinar con las restricciones struct ni new().                                                                                                                                                                                                                                                               |
| where T<br>new()     | El argumento de tipo debe tener un constructor sin parámetros público. Cuando se usa conjuntamente con otras restricciones, la restricción new() debe                                                                                                                                                                                                                                                                                                                |

|                                                                                                                     |
|---------------------------------------------------------------------------------------------------------------------|
| especificarse en último lugar. La restricción new() no se puede combinar con las restricciones struct ni unmanaged. |
|---------------------------------------------------------------------------------------------------------------------|

Vamos a verlo sobre un ejemplo sencillo, en la clase Persona vamos a hacer que reciba un tipo genérico T, que lo usaremos para el identificador de persona, y que el Identificador sea comparable ya que implementa el interfaz IComparable.

```
public class Persona<T> where T : IComparable
```

El tipo genérico lo Podemos usar para :

Propiedades: `private T id;`

Constructores: `public Persona(String Apellido, String Nombre, DateTime FechaNac, T id)`

Getters y Setters: `public T Id`

Métodos: `public bool compararID(T id)`

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

namespace ClasesEInterfacesGenericos
{
 public class Persona<T> where T : IComparable
 {
 private String elApellido;
 private String elNombre;
 private DateTime laFechaNac;
 private int laEdad;

 private T id;

 public Persona()
 {

 }

 public Persona(String Apellido, String Nombre, DateTime FechaNac, T id)
 {
 this.Apellido = Apellido;
 this.Nombre = Nombre;
 this.FechaNac= FechaNac;
 this.Id = id;
 laEdad = this.Edad;

 }
 }
}
```



```

public String Apellido
{
 get { return elApellido; }
 set { elApellido = value.ToUpper(); }
}
public String Nombre
{
 get { return elNombre; }
 set { elNombre = value.ToLower(); }
}
public DateTime FechaNac
{
 get { return laFechaNac; }
 set
 {
 if (value.Year >= 1900)
 {
 laFechaNac = value;
 }
 }
}

public int Edad
{
 get { return DateTime.Now.Year - FechaNac.Year; }
}

public T Id
{
 get { return this.id; }
 set { this.id = value; }
}

public override string ToString()
{
 return String.Format("Apellidos: {0}, Nombre: {1}, FechaNac: {2:d}, Edad: {3}, id: {4}", this.Apellido, this.Nombre, this.FechaNac, this.Edad, this.Id);
}

public bool compararID(T id)
{
 return this.Equals(id);
}
}
}

```

## 4.8 Interfaces útiles en C#

### 4.8.1 IComparable

Obliga a las clases que lo implementan a sobrescribir el método CompareTo. Muy útil para ordenar colecciones como veremos posteriormente. Es casi obligatorio en nuestro modelo de datos cuando usamos nuestros objetos en colecciones.

Define un método de comparación generalizado específico del tipo que implementa un tipo o una clase de valor con el fin de ordenar sus instancias.

```
public interface IComparable
```

Método:

```
public int CompareTo (object? obj);
```

Un valor que indica el orden relativo de los objetos que se están comparando. El valor devuelto tiene los siguientes significados:

#### DEVOLUCIONES

| Valor         | Significado                                                                        |
|---------------|------------------------------------------------------------------------------------|
| Menor<br>cero | que Esta instancia es anterior a obj en el criterio de ordenación.                 |
| Cero          | Esta instancia se produce en la misma posición del criterio de ordenación que obj. |
| Mayor<br>cero | que Esta instancia sigue a obj en el criterio de ordenación.                       |

Sobre el ejemplo inicial de persona vamos a comparar la persona por Edad. Implementamos IComparable con el tipo predefinido Persona.

```
public class Persona : IComparable<Persona>
```

El interfaz nos obliga a implementar el método CompareTo. Aprovechamos el CompareTo de Edad que ya esta implementado y ordenamos personas por Edad Ascendente.

```
public int CompareTo([AllowNull] Persona other)
{
 return this.Edad.CompareTo(other);
}
```

```

using System;
using System.Collections.Generic;
using System.Diagnostics.CodeAnalysis;
using System.Text;

namespace InterfacesUtiles
{
 public class Persona : IComparable<Persona>
 {

 private String elApellido;
 private String elNombre;
 private DateTime laFechaNac;
 private int laEdad;

 public Persona()
 {

 }

 public Persona(String Apellido, String Nombre, DateTime FechaNac)
 {

 this.Apellido = Apellido;
 this.Nombre = Nombre;
 this.FechaNac = FechaNac;

 laEdad = this.Edad;

 }

 public String Apellido
 {
 get { return elApellido; }
 set { elApellido = value.ToUpper(); }
 }
 public String Nombre
 {
 get { return elNombre; }
 set { elNombre = value.ToLower(); }
 }
 public DateTime FechaNac
 {
 get { return laFechaNac; }
 set
 {
 if (value.Year >= 1900)
 {
 laFechaNac = value;
 }
 }
 }

 public int Edad
 {

```

```

 get { return DateTime.Now.Year - FechaNac.Year; }
 }

 public int CompareTo([AllowNull] Persona other)
 {
 return this.Edad.CompareTo(other);
 }

 public override string ToString()
 {
 return String.Format("Apellidos: {0}, Nombre: {1}, FechaNac: {2:d}, Edad: {3} ",
 this.Apellido, this.Nombre, this.FechaNac, this.Edad);
 }
}

```

### 4.8.2 Ejercicio

Modificar el código anterior para ordenar las personas por Edad pero descendente. Es decir que una persona de menor edad sea mayor que una de mayor edad.

### 4.8.3 ICloneable

Admite la clonación, que crea una nueva instancia de una clase con el mismo valor que una instancia existente.

```
public interface ICloneable
```

#### Método

Crea un nuevo objeto copiado de la instancia actual.

```
public object Clone ();
```

Al ejemplo anterior Podemos añadir, el interfaz ICloneable

```
public class Persona : IComparable<Persona> , ICloneable
```

E Implementar el método clone(), devolviendo un objeto nuevo Persona con los mismo campos que la persona original.

```

 public object Clone()
 {
 return new Persona(this.Nombre, this.Apellido, this.FechaNac);
 }

```

#### 4.8.4 IFormattable e IEquatable

Estos dos interfaces obligan a implementar los métodos Equals(IEquatable) ToString() (IFormattable) que ya hemos visto anteriormente, por lo que no proporcionaremos ejemplos

## 5 Tipos delegados en C#

Un delegado es un tipo que representa referencias a métodos con una lista de parámetros determinada y un tipo de valor devuelto. Cuando se crea una instancia de un delegado, puede asociar su instancia a cualquier método mediante una signatura compatible y un tipo de valor devuelto. Puede invocar (o llamar) al método a través de la instancia del delegado.

Los delegados se utilizan para pasar métodos como argumentos a otros métodos. Los controladores de eventos no son más que métodos que se invocan a través de delegados. Cree un método personalizado y una clase, como un control de Windows, podrá llamar al método cuando se produzca un determinado evento. En el siguiente ejemplo se muestra una declaración de delegado:

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

Esta capacidad de hacer referencia a un método como parámetro hace que los delegados sean idóneos para definir métodos de devolución de llamada. Se puede, por ejemplo, escribir un método que compare dos objetos en la aplicación. Ese método se puede usar en un delegado para un algoritmo de ordenación. Como el código de comparación es independiente de la biblioteca, el método de ordenación puede ser más general.

Un delegado es un tipo que encapsula de forma segura un método, similar a un puntero de función en C y C++. A diferencia de los punteros de función de C, los delegados están orientados a objetos, proporcionan seguridad de tipos y son seguros. El tipo de un delegado se define por el nombre del delegado. En el ejemplo siguiente, se declara un delegado denominado Del que puede encapsular un método que toma una string como argumento y devuelve void:

```
public delegate void Del(string message);
```

Normalmente, un objeto delegado se construye al proporcionar el nombre del método que el delegado encapsulará o con una función anónima. Una vez que se crea una instancia de delegado, el delegado pasará al método una llamada de método realizada al delegado. Los parámetros pasados al delegado por el autor de la llamada se pasan a su vez al método, y el valor devuelto desde el método, si lo hubiera, es devuelto por el delegado al autor de la llamada. Esto se conoce como invocar al delegado. Un delegado con instancias se puede

invocar como si fuera el propio método encapsulado. Por ejemplo:

```
// Create a method for a delegate.
public static void DelegateMethod(string message)
{
 Console.WriteLine(message);
}
```

Y asignarlo a una variable como sigue

```
// Instantiate the delegate.
Del handler = DelegateMethod;

// Call the delegate.
handler("Hello World");
```

Lo vamos a ver en el siguiente ejemplo de forma más clara. Declaramos un tipo delegado Operación que recibirá dos parámetros double y devuelve un double.

```
public delegate double Operacion(double num1, double num2);
```

Creamos unos métodos para operar que también podrían ser no estáticos pero por comodidad los realizamos estáticos, Potencia y división redondeada. Observar como tienen la misma signatura que Operación, reciben dos doubles y devuelven un double. Es obligatorio para poder asignarlos a una variable delegate.

```
private static double Potencia(double num1, double num2)
```

```
private static double DivisionRedondeado(double num1, double num2)
```

Asignamos estos métodos a variables de tipo operación. Estas variables van a contener una referencia a una función.

```
Operacion opPower = Potencia;
```

```
Operacion opDiv = DivisionRedondeado;
```

También podemos usar métodos de clases predefinidas en C# o librerías de tercero, como hacemos con la función Max de la clase Math. Este método no lo hemos creado nosotros.

```
Operacion opMax = Math.Max;
```

Por ultimo para usar estas variables como funciones le pasamos los parámetros sobre los que queremos operar como si fueran una función normal.

```
opPower(num1, num2), opDiv(num1, num2) , Ó opMax(num1, num2)
```

```
using System;
```

```
namespace EjemploDelegadoOperaciones
```

```
{
```

```
 public delegate double Operacion(double num1, double num2);
```

```
 class Program
```

```
 {
```

```
 private static double Potencia(double num1, double num2)
```

```
 {
```

```
 return Math.Pow(num1, num2);
```

```
 }
```

```
 private static double DivisionRedondeado(double num1, double num2)
```

```
 {
```

```
 return Math.Round(num1 / num2);
```

```
 }
```

```
 static void Main(string[] args)
```

```
 {
```

```
 Operacion opPower = Potencia;
```

```
 Operacion opDiv = DivisionRedondeado;
```

```
 Operacion opMax = Math.Max;
```

```
 Console.WriteLine("Escribe un número decimal por pantalla para operar");
```

```
 double num1 = double.Parse(Console.ReadLine());
```

```
 Console.WriteLine("Escribe otro número decimal por pantalla para operar");
```

```
 double num2 = double.Parse(Console.ReadLine());
```

```
 Console.WriteLine("La potencia de {0} y {1} es: {2}", num1, num2, opPower(num1, num2));
```

```
 Console.WriteLine("La division redondeada de {0} y {1} es: {2}", num1, num2, opDiv(num1, num2));
```

```
 Console.WriteLine("El máximo de {0} y {1} es: {2}", num1, num2, opMax(num1, num2));
```

```
 }
```

```
 }
```

```
}
```

### 5.1.1 Ejercicio

Crear un tipo delegador que reciba dos enteros y devuelve, un entero 1, 0 o -1.

Crear dos métodos estáticos :

```
public static int comparaNormal(int n1, int n2)
```

Devolverá 1 si n1 es mayor que n2. Si son iguales devuelve 0, Si n1 es más pequeño que n2 devuelve -1.

```
public static int comparaInversa(int n1, int n2)
```

Devolverá -1 si n1 es mayor que n2. Si son iguales devuelve 0, Si n1 es más pequeño que n2 devuelve 1.

Asignar estos dos métodos a dos variables de tipo delegado. Y realizar un programa que produzca la siguiente salida:

```
Escribe un número por pantalla para comparar
45
Escribe otro número por pantalla para comparar
67
En comparación directa
45 es menor que 67
En comparación inversa
45 es mayor que 67
```

## 5.2 Usando delegados como parámetros o en setters

Desde el momento que una función se convierte en un tipo en si misma, puede ser declarada como una variable, pero además puede ser pasada como parámetro o como propiedad de una clase. Como vamos a ver es muy útil en todos los ámbitos, como por ejemplo vamos a modificar de manera dinámica el comportamiento del método equals para la siguiente clase Persona.

A la clase Persona que usábamos en un ejemplo anterior hemos añadido un delegado que recibe dos personas y devuelve un booleano

```
public delegate bool MiEquals(Persona p1, Persona p2);
```

Hemos añadido al ejemplo de persona anterior la propiedad DNI y una propiedad de tipo delegado mi equals.



```
private string dni;
private int laEdad;
private MiEquals miequals;
```

Para la propiedad miequals asignamos el método `public bool equalsNombreApellido(Persona p1, Persona p2)`, que devuelve true si las dos personas tienen el mismo nombre y apellidos, y false en caso contrario.

Para la propiedad miequals definimos un getter y un setter

```
public MiEquals MiEqual
{

 get { return this.miequals; }
 set { this.miequals = value; }
}
```

Sobreescribimos el método Equals para que dos objetos de tipo persona sean iguales si su nombre y apellidos son iguales. Para eso llamamos a nuestra propiedad delegado que contiene la función miequals nombre y apellidos.

```
public override bool Equals(object obj)
{
 return miequals(this, (Persona)obj);
}
```

Adoptando este enfoque podremos cambiar la propiedad miequals y hacer que el mismo objeto sea susceptible de múltiples comparaciones. En el programa principal del mismo ejemplo hemos definido un método estático que comparará a personas por DNI.

```
private static bool equalsDNI(Persona p1, Persona p2) {
 return (p1.DNI.Equals(p2.DNI));
}
```

Definimos dos personas y realizamos un equals. Recordar que por defecto el equals lo habíamos definido sobre nombre y apellidos

```
Persona persona = new Persona("Perez Lopez", "Rodolfo", new DateTime(1990, 6,
12), "44888555G");
```

```
Persona persona2 = new Persona("Perez Lopez", "Julian", new DateTime(1980, 6, 12),
"44888555G");
```

```
Console.WriteLine("Persona {0} y Persona {1} son iguales por nombre y apellidos:
{2}", persona, persona2, persona.Equals(persona2));
```

Vamos a modificar la propiedad `miEqual` de `persona` para que el método que contenga es la comparación por DNI de `equalsDNI`. Como veis el poder pasar delegados en los setters o en métodos me permite modificar el comportamiento de los objetos de manera dinámica.

```
persona.MiEqual = equalsDNI;
```

```
Console.WriteLine("Persona {0} y Persona {1} son iguales por dni:{2} ", persona,
persona2, persona.Equals(persona2));
```

Después de asignar a `MiEqual` `equalsDNI`, `persona.MiEqual = equalsDNI;` el método `Equals` de `Persona` ha modificado su comportamiento `persona.MiEqual = equalsDNI;`, ahora comparará por DNI.

Otra manera de realizar la misma acción es crear un método propio para comparar dos personas, que recibe una persona y un delegado, el método usado a comparar

```
public bool miComparacion(MiEquals metodo, Persona p)
```

De esta manera podemos llamar a este método desde el programa principal y pasar el método que queramos para comparar dos personas, en este caso DNI otra vez.

```
Console.WriteLine("Persona {0} y Persona {1} son iguales por dni con mi metodo de
comparacion:{2} ",
persona, persona2, persona.miComparacion(equalsDNI, persona2));
```

```
Console.WriteLine("Persona {0} y Persona {1} son iguales por nombre y apellidos con mi método
de comparacion: {2}"
, persona, persona2, persona.miComparacion(persona.equalsNombreApell,
persona2));
```

Importante reseñar aquí que `equalsDNI` es un método de clase, estático. Sin embargo, `persona.equalsNombreApell` es un método de instancia, del objeto `persona`. Estos últimos sólo son asignables si el objeto está instanciado, no antes.

## Ejecución

Persona Apellidos: PEREZ LOPEZ, Nombre: rodolfo, FechaNac: 12/06/1990, Edad: 31 , DNI 44888555G y Persona Apellidos: PEREZ LOPEZ, Nombre: julian, FechaNac: 12/06/1980, Edad: 41 , DNI 44888555G **son iguales por nombre y apellidos: False**  
Persona Apellidos: PEREZ LOPEZ, Nombre: rodolfo, FechaNac: 12/06/1990, Edad: 31 , DNI

44888555G y Persona Apellidos: PEREZ LOPEZ, Nombre: julian, FechaNac: 12/06/1980, Edad: 41 , DNI 44888555G **son iguales por dni:True**

Persona Apellidos: PEREZ LOPEZ, Nombre: rodolfo, FechaNac: 12/06/1990, Edad: 31 , DNI 44888555G y Persona Apellidos: PEREZ LOPEZ, Nombre: julian, FechaNac: 12/06/1980, Edad: 41 , DNI 44888555G **son iguales por dni con mi metodo de comparacion:True**

Persona Apellidos: PEREZ LOPEZ, Nombre: rodolfo, FechaNac: 12/06/1990, Edad: 31 , DNI 44888555G y Persona Apellidos: PEREZ LOPEZ, Nombre: julian, FechaNac: 12/06/1980, Edad: 41 , DNI 44888555G **son iguales por nombre y apellidos con mi método de comparacion: False**

## Ejemplo Persona

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

namespace DelegadoComoParametro
{
 public delegate bool MiEquals(Persona p1, Persona p2);
 public class Persona
 {
 private String elApellido;
 private String elNombre;
 private DateTime laFechaNac;
 private string dni;
 private int laEdad;
 private MiEquals miequals;

 public bool equalsNombreApell(Persona p1, Persona p2)
 {
 return (p1.elNombre.Equals(p2.elNombre) && p1.elApellido.Equals(p2.elApellido));
 }
 public Persona()
 {
 miequals = equalsNombreApell;
 }

 public Persona(String Apellido, String Nombre, DateTime FechaNac, String dni)
 {
 miequals = equalsNombreApell;
 this.Apellido = Apellido;
 this.Nombre = Nombre;
 this.FechaNac= FechaNac;
 this.DNI = dni;
 laEdad = this.Edad;
 }

 public String Apellido
 {

```

```

 get { return elApellido; }
 set { elApellido = value.ToUpper(); }
 }
 public String Nombre
 {
 get { return elNombre; }
 set { elNombre = value.ToLower(); }
 }
 public DateTime FechaNac
 {
 get { return laFechaNac; }
 set
 {
 if (value.Year >= 1900)
 {
 laFechaNac = value;
 }
 }
 }

 public int Edad
 {
 get { return DateTime.Now.Year - FechaNac.Year; }
 }

 public string DNI
 {
 get { return this.dni; }
 set { this.dni = value; }
 }

 public MiEquals MiEqual
 {
 get { return this.miequals; }
 set { this.miequals = value; }
 }

 public override string ToString()
 {
 return String.Format("Apellidos: {0}, Nombre: {1}, FechaNac: {2:d}, Edad: {3} ,
DNI {4}", this.Apellido, this.Nombre, this.FechaNac, this.Edad, this.DNI);
 }

 public override bool Equals(object obj)
 {
 return miequals(this, (Persona)obj);
 }

 public bool miComparacion(MiEquals metodo, Persona p)
 {
 return (metodo(this, p));
 }

```

```

 }
}

```

## Programa principal

```

using System;

namespace DelegadoComoParametro
{
 class Program
 {
 private static bool equalsDNI(Persona p1, Persona p2) {
 return (p1.DNI.Equals(p2.DNI));
 }
 static void Main(string[] args)
 {
 Persona persona = new Persona("Perez Lopez", "Rodolfo", new DateTime(1990, 6,
12), "44888555G");

 Persona persona2 = new Persona("Perez Lopez", "Julian", new DateTime(1980, 6, 12),
"44888555G");

 Console.WriteLine("Persona {0} y Persona {1} son iguales por nombre y apellidos:
{2}", persona, persona2, persona.Equals(persona2));

 persona.MiEqual = equalsDNI;

 Console.WriteLine("Persona {0} y Persona {1} son iguales por dni:{2} ", persona,
persona2, persona.Equals(persona2));

 Console.WriteLine("Persona {0} y Persona {1} son iguales por dni con mi metodo de
comparacion:{2} ",
 persona, persona2, persona.miComparacion(equalsDNI, persona2));

 Console.WriteLine("Persona {0} y Persona {1} son iguales por nombre y apellidos con mi
método de comparacion: {2}"
 , persona, persona2, persona.miComparacion(persona.equalsNombreApell,
persona2));
 }
 }
}

```

### 5.2.1 Ejercicio

Crear un método nuevo igualesCompleto de comparación en el que dos personas son iguales sólo si su nombre apellidos y dni son iguales.

Modificar la propiedad miequals de persona para realizar la comparación con igualesCompleto.

Usar el método miComparación para probar igualesCompleto.

### 5.2.2 Ejercicio

Implementar IComparable usando delegados para que sea posible cambiar el comportamiento de la comparación de un objeto Persona.

1. Crear un delegado Comparable(Persona1, Persona2) que compare dos objetos Persona
2. Crear una propiedad Comparable miComparable y su getter y setter
3. Usarla en el método CompareTo.

### 5.2.3 Ejercicio

Partimos de la premisa de que podemos crear un Delegado genérico, por ejemplo:

```
public delegate void Del<T>(T item);
```

Para el ejemplo de interfaces genéricos, crear una propiedad delegado que nos permita implementar el ejercicio anterior con un Persona<T>, comparando por ID

```
namespace ClasesEInterfacesGenericos
{
 public class Persona<T> where T : IComparable
 {
```

## 5.3 Delegados predefinidos en C#

Ya hemos visto que podemos definir nuestros propios delegados pero C# tiene delegados predefinidos ya en sus librerías. De esta manera no hace falta andar definiendo delegados todo el tiempo, usamos los de c#. podemos distinguir tres tipos:

Func: que puede recibir parámetros y devolver un resultado.

Action: puede recibir parámetros y no devuelve resultados

Predicate: puede recibir parámetros y devuelve un booleano.

Tenemos diferentes versiones de Func, empecemos por la definición del que no recibe parámetros. Delegado Func<TResult>.

Declaración en C#:

```
public delegate TResult Func<out TResult>();
```

Para usarlo en nuestros programas solo

```
Func<int> delegadoFuncionInt=devEntero;
```

Donde devEnteroDouble es el método sin parámetros que devuelve un entero:

```
private static int devEntero()
{
 return 1;
}
```

Con parámetros, se indica antes los tipos de los parámetros de entrada.

Para un parámetro la declaración de C#

```
public delegate TResult Func<in T,out TResult>(T arg);
```

Para usarlo

```
Func<double, int> delegadoUnParam = devEnteroDouble;
```

, podéis ver la declaración del método devEnteroDouble en el ejemplo.

Para dos parámetros la declaracion en C#:

```
public delegate TResult Func<in T1,in T2,out TResult>(T1 arg1, T2 arg2);
```

Para usarlo:

```
Func<double, double, double> delegadoDosParam = suma;
```

Tenemos delegados predefinidos Func de hasta 16 parámetros. En el ejemplo completo podeis ver los métodos declarados asignados a los delegados como hemos hecho en ejemplos anteriores.

```
using System;
```

```
namespace DelegadosPredefinidos
```

```

{
 class Program
 {
 private static int devEntero()
 {
 return 1;
 }

 private static int devEnteroDouble(double d)
 {
 return (int) d;
 }

 private static double suma(double s1, double s2)
 {
 return s1 + s2;
 }

 static void Main(string[] args)
 {
 Func<int> delegadoFuncionInt=devEntero;

 Func<double, int> delegadoUnParam = devEnteroDouble;

 Func<double, double, double> delegadoDosParam = suma;
 }
 }
}

```

En resumen, estamos haciendo lo mismo pero usando los delegados predefinidos de C#.

Igualmente cuando no queráis devolver parámetros en vuestros delegados podéis usar los delegados predefinidos Action y Predicate, hasta 16 parámetros.

```

public delegate void Action();
public delegate void Action<in T>(T obj);
public delegate void Action<in T1,in T2>(T1 arg1, T2 arg2);

```

```

Action accion;
Action<int> accionUnParametro;
Action<int, double> accionDosParametros;

```

```

public delegate bool Predicate();

public delegate bool Predicate<in T>(T obj);

```



```
Predicate<string> predicado;
```

## 6 Expresiones lambda en C#. Funciones o métodos anónimos

### 6.1 Introducción

Recordamos de nuestro conocimiento en teoría de lenguajes **tres paradigmas de lenguajes de programación: imperativa, funcional y declarativa**. C# ha sido tradicionalmente un lenguaje de programación imperativo o declarativo, basado en la mutabilidad, asignar valores a variables y transportar o transformar datos de una variable o estructura de datos a otra.

El **paradigma funcional** está basado en el **concepto matemático de función**. Los programas escritos en **lenguajes funcionales** estarán constituidos por un **conjunto de definiciones de funciones** (entendiendo estas no como subprogramas clásicos de un lenguaje imperativo) **junto con los argumentos** sobre los que se aplican.

A partir de **C versión 6** un nuevo paradigma de programación esta disponible en Java. Es la **programación funcional**. Realizar código usando única o básicamente funciones y listas como si estuviéramos usando un lenguaje funcional como Lisp. Esta nueva manera de programar se basa en:

**Delegados:** definición de funciones como variables o parámetros a las que permiten asignar o asociar métodos reales.

**Expresiones lambda:** están basadas en el **calculo lambda**. El concepto es la declaración de **funciones anónimas**. En calculo lambda una función se puede declarar anónimamente. **Por ejemplo**, **Cuadrado(x) =  $x^2$**  puede ser definida como  **$\lambda x.x^2$  o  $(x) \rightarrow x^2$** . De esta manera **vamos a definir nuestras funciones en java**. Y resolveremos nuestros algoritmos usando básicamente funciones.

El objetivo es **asignar a los delegados de C# expresiones Lambda, que son como métodos, funciones, pero anónimas**, lambda. Ahorramos instrucciones y líneas de código haciéndolo de esta manera. Puede decirse expresión lambda representa una función anónima que se le aporta al delegado que define la signature de la función anónima. Añaden la acción real a la definición

### 6.2 Clases anónimas en C#

El objetivo de las clases anónimas en C# es crear objetos sin ser previamente definidos. El compilador creará la clase.

```
var producto1 = new { nombre = "galleta", precio = 1.99 };
```

La clase en este caso contendrá dos propiedades: *nombre* y *precio*. Además, tendremos una serie de métodos heredados de la clase **Object**.

Si el compilador detecta que estamos creando dos objetos a partir de clases anónimas y tienen el mismo número, nombre, tipo y orden de las propiedades entonces reutilizará la clase para crear la instancia.

Ejecución. Podéis ver que el tipo es Anonymous.

La clase de producto1 es <>f\_\_AnonymousType0`2

La clase de producto2 es <>f\_\_AnonymousType0`2

```
using System;

namespace ClasesAnonimas
{
 class Program
 {
 static void Main(string[] args)
 {
 var producto1 = new { nombre = "galleta", precio = 1.99 };
 var producto2 = new { nombre = "mantequilla", precio = 2.56 };
 Console.WriteLine("La clase de producto1 es {0}", producto1.GetType().Name);
 Console.WriteLine("La clase de producto2 es {0}", producto2.GetType().Name);
 }
 }
}
```

## 6.3 Expresiones lambda en c#

En el ejemplo anterior hemos visto que no hace falta declarar delegados pues podemos usar los delegados ya definidos en C# Func y Action. Igualmente, si os fijáis en los ejemplos anteriores es muy aparatoso definir un método que luego hay que asignar a un delegado. Una manera más sencilla de resolver esto es el uso de expresiones lambda en C#, funciones anónimas.

Se usan las *expresión lambda* para crear una función anónima. Use el operador de declaración lambda => para separar la lista de parámetros de la lambda de su cuerpo. Una expresión lambda puede tener cualquiera de las dos formas siguientes:

- Una lambda de expresión que tiene una expresión como cuerpo:  
(input-parameters) => expression
- Una lambda de instrucción que tiene un bloque de instrucciones como cuerpo:  
(input-parameters) => { <sequence-of-statements> }

Un ejemplo sencillo sería asignar a un delegado una expression lambda:

```
Func<double, int> delLambda = (num) => (int) Math.Round(num);
```

El delegado define los tipos de la expresión lambda en este caso, num será double el resultado devuelto por la expresión lambda es int. En estas expresiones lambda de una sola línea no hace falta escribir return, es implícito. La función devuelve el resultado de la expresión de la parte derecha de la lambda.

Toda expresión lambda se puede convertir en un tipo delegado. El tipo delegado al que se puede convertir una expresión lambda se define según los tipos de sus parámetros y el valor devuelto. Si una expresión lambda no devuelve un valor, se puede convertir en uno de los tipos delegados Action. La siguiente lambda escribe un número por pantalla.

```
Action<int> accionLambda = (num) => Console.WriteLine("Escribimos el número {0}", num);
```

La parte derecha de la expresión Console.WriteLine en principio no devuelve nada.

En las expresiones lambda de bloque de instrucciones si que debemos indicar un return en caso de que la expresión lambda deba devolver un valor. Como veis es como una función pero anónima. En este caso calcula el factorial

```
Func<int, int> delLambdaBloqueFact = (num) =>
{
 int facto = 1;

 for (int i = 1; i <= num; i++)
 {
 facto = facto * i;
 }
 return facto;
};
```

Para llamar a la lambda hacemos con el delegado lo mismo que hacíamos en los casos anteriores, asignar parámetros.

```
accionLambda(delLambdaBloqueFact(5));
```

Podemos hacer lambdas también con el delegado Predicate. La siguiente lambda es una función que devolverá verdadero si el numero pasado es mayor que 5 asignada a un Predicate.

```
Predicate<int> predicado = (num) => num > 5;
```

```
Console.WriteLine("Escribe un numero por pantalla");
```

```

int numero = int.Parse(Console.ReadLine());

if (predicado(numero))
 Console.WriteLine("el número {0} es mayor que ", numero);

else
 Console.WriteLine("el número {0} es menor o igual que ", numero);

```

## Ejecución del ejemplo completo

Escribimos el número 120

## Ejemplo completo

```

using System;

namespace ExpresionesLambda
{
 class Program
 {
 static void Main(string[] args)
 {
 Func<double, int> delLambda = (num) => (int) Math.Round(num);
 Action<int> accionLambda = (num) => Console.WriteLine("Escribimos el número {0}",
num);

 Func<int, int> delLambdaBloqueFact = (num) =>
 {
 int facto = 1;

 for (int i = 1; i <= num; i++)
 {
 facto = facto * i;
 }
 return facto;
 };

 accionLambda(delLambdaBloqueFact(5));

 Predicate<int> predicado = (num) => num > 5;

 Console.WriteLine("Escribe un numero por pantalla");

 int numero = int.Parse(Console.ReadLine());

 if (predicado(numero))
 Console.WriteLine("el número {0} es mayor que ", numero);

 else
 Console.WriteLine("el número {0} es menor o igual que ", numero);

 }
 }
}

```

```
}
}
```

### 6.3.1 Ejercicios

1. **Escribid una expresión lambda de bloque con un bucle while que escriba la lista de los n primeros números pares, calcule la suma de los n primeros números pares y la muestre por pantalla. (0,4 puntos)**
2. **Escribid una expresión lambda que escriba los números primos entre 1 y n. Usad un bucle for. (0,4 puntos)**
3. Implementar IComparable usando delegados para que sea posible cambiar el comportamiento de la comparación de un objeto Persona.
  - a) Crear un delegado Comparable(Persona1, Persona2) que compare dos objetos Persona
  - b) Crear una propiedad Comparable miComparable y su getter y setter
  - c) Usarla en el método CompareTo.
  - d) Crear una expresión lambda para realizar la comparación por nombre.

## 7 Excepciones

Las características de control de excepciones del lenguaje C# le ayudan a afrontar cualquier situación inesperada o excepcional que se produce cuando se ejecuta un programa. El control de excepciones usa las palabras clave try, catch y finally para intentar realizar acciones que pueden no completarse correctamente, para controlar errores cuando decide que es razonable hacerlo y para limpiar recursos más adelante. Las excepciones las puede generar Common Language Runtime (CLR), .NET, bibliotecas de terceros o el código de aplicación. Las excepciones se crean mediante el uso de la palabra clave throw.

En muchos casos, una excepción la puede no producir un método al que el código ha llamado directamente, sino otro método más bajo en la pila de llamadas. Cuando se genera una excepción, CLR desenreda la pila, busca un método con un bloque catch para el tipo de excepción específico y ejecuta el primer bloque catch que encuentra. Si no encuentra ningún bloque catch adecuado en cualquier parte de la pila de llamadas, finalizará el proceso y mostrará un mensaje al usuario.

En este ejemplo, un método prueba a hacer la división entre cero y detecta el error. Sin el control de excepciones, este programa finalizaría con un error DivideByZeroException no controlada.

Podemos usar Excepciones definidas en el Framework para controlar errores. Hemos definido

el método `DivisionSegura`, que lanza una la excepción `DivideByZeroException` si se intenta dividir por cero. Las excepciones puede generarlas explícitamente un programa con la palabra clave `throw`.

El bloque `try` hace que todo el código que contiene sea susceptible de capturar la excepción con su `catch` asociado.

El bloque `catch` captura la excepción `catch` (`DivideByZeroException`) y realiza algún tipo de acción relativa a tratar esa excepción.

```
using System;

namespace Excepciones
{
 class Program
 {
 static double DivisionSegura(double x, double y)
 {
 if (y == 0)
 throw new DivideByZeroException();
 return x / y;
 }

 public static void Main()
 {
 double a = 98, b = 0;
 double result;

 try
 {
 result = DivisionSegura(a, b);
 Console.WriteLine("{0} dividido por{1} = {2}", a, b, result);
 }
 catch (DivideByZeroException)
 {
 Console.WriteLine("Excepcion. Division por cero.");
 }
 finally
 {
 Console.WriteLine("Para liberar recursos si es necesario se ejecuta siempre con o sin excepcion");
 }
 }
 }
}
```

Si no hay ningún controlador de excepciones para una excepción determinada, el programa deja de ejecutarse con un mensaje de error. El código de un bloque `finally` se ejecuta incluso si se produce una excepción. Use un bloque `finally` para liberar recursos, por ejemplo, para cerrar las secuencias o los archivos que se abrieron en el bloque `try`. En .NET, una excepción es un objeto que hereda de la clase `System.Exception`. Una excepción

se inicia desde un área del código en la que ha producido un problema. La excepción se pasa hacia arriba en la pila hasta que la aplicación la controla o el programa finaliza.

## 7.1 *La clase Exception*

Representa los errores que se producen durante la ejecución de la aplicación.

```
public class Exception : System.Runtime.Serialization.ISerializable
```

Herencia Object -> Exception

### Propiedades destacadas

InnerException: obtiene la instancia Exception que produjo la excepción actual.

Message: mensaje de error

Source: Devuelve o establece el nombre de la aplicación o del objeto que generó el error.

StackTrace: pila

### Métodos

GetBaseException: excepción base

GetObjectData: Cuando se produce la excepción en una clase derivada, establece SerializationInfo con información sobre la excepción

GetType

ToString

## 7.2 *Creando nuestras propias excepciones*

Heredando de Exception podemos crear y lanzar nuestras propias excepciones en C#. Es aconsejable cuando queremos controlar errores propios de nuestra lógica de negocios. En el siguiente ejemplo podemos ver como hacerlo. Sobreescribimos los 3 posibles constructores de Exception como por ejemplo `public ExcepcionAccesoIndebido() : base()`.

Es aconsejable llamar a los constructores de la clase Exception. Además añadimos un nuevo constructor para controlar los datos propios de nuestra Excepción, para guardar la fuente de datos que produjo nuestro error.

```

 public ExcepcionAccesoIndebido(String mensajesError, Exception inner, string fuente) :
base(mensajesError, inner)
 {
 this.FuenteDeDatos = fuente;
 }
 }

using System;
using System.Collections.Generic;
using System.Text;

namespace Excepciones
{
 class ExcepcionAccesoIndebido : Exception
 {
 private string FuenteDeDatos;

 public ExcepcionAccesoIndebido() : base()
 {
 }

 public ExcepcionAccesoIndebido(String mensajesError, string fuente) : base(mensajesError)
 {
 this.FuenteDeDatos = fuente;
 }

 public ExcepcionAccesoIndebido(String mensajesError, Exception inner) :
base(mensajesError, inner)
 {
 }

 public ExcepcionAccesoIndebido(String mensajesError, Exception inner, string fuente) :
base(mensajesError, inner)
 {
 this.FuenteDeDatos = fuente;
 }

 public string Fuente
 {
 get { return FuenteDeDatos; }
 }
 }
}

```

Para controlar el error en un programa que este accediendo a datos podríamos lanzar la excepción creada de la siguiente manera:



```
throw new ExcepcionAccesoIndebido("Error en acceso a datos", "Fuente XML");
```