



1. Programación Básica C#.....	5
1.1. Fundamentos de programación en C#.....	5
1.1.1. Introducción	5
1.1.2 Arquitectura .Net Core.....	5
1.1.3. Lenguajes soportados por .Net Core	5
1.1.4. Ventajas y desventajas de Net Core	6
1.2. Configuración del entorno de desarrollo.....	7
1.2.1. Configuración en Linux.....	8
1.2.2. Configuración en Windows	10
1.3. Programación en C#	11
1.3.1. Creación de proyectos de consola en C# con Visual Studio Code	11
1.3.2. Ejecución de proyectos	12
1.3.3. Estructura de un proyecto de consola en .NET.....	13
1.3.4. Variables y constantes	13
1.3.4.1 Variables.....	13
1.3.4.2. Constantes	13

1.3.5. bool (Referencia de C#).....	13
1.3.6. Tipos numéricos enteros.....	14
1.3.6.1. Tipos Simples.....	15
1.3.6.2. Tipos enteros.....	15
1.3.6.3. Tipos numéricos de punto flotante (referencia de C#)	16
1.3.6.4. Cadenas con formato numérico estándar	19
1.4. Entrada y salida de datos	23
1.5. Conversión de tipos de datos.....	25
1.5.1. Conversiones implícitas	25
1.6. Estructuras de control.....	27
1.6.1. Estructuras condicionales	28
1.6.1.1 if	28
1.6.1.2. if ...else	28
1.6.1.3. if....else if..else	29
1.6.1.4. switch	29
1.6.2 Estructura repetitivas (Iterativas o Ciclos)	30
1.6.2.1. for	30
1.6.1.2. while	31
1.6.1.3. do...while.....	31
2. Programación Basica	32
2.1. Estructura de datos	32
2.1.1. Arreglos	32
2.1.1.1. Declaración	32
2.1.1.2. Inicialización.....	33
2.1.1.3 Acceso a elementos de un arreglo.....	33
2.1.1.4. Métodos de arreglos	33
2.1.2. ArrayList	34
2.1.2.1. Métodos	34
2.1.2.1.1. Add	34
2.1.2.1.2 Clear	37
2.1.2.1.3 Contains	38
2.1.2.1.4 IndexOf.....	39
2.1.2.1.5. Insert e InsertRange.....	41
2.1.2.1.7. Remove y RemoveAt.....	44
2.1.2.1.8. ToArray.....	45
2.1.3. Clase List.....	46

2.1.3.1. Propiedades	46
2.1.3.1.1. Count.....	46
2.1.3.1.2. Item	47
2.1.3.2. Métodos	48
2.1.3.2.1. Add	48
2.1.3.2.2. AddRange	49
2.1.3.2.3. Clear	50
2.1.3.2.4. Exists	51
2.1.3.2.5. Find.....	52
2.1.3.2.6. FindAll	53
2.1.3.2.7. FindLast	54
2.1.4. Clase Dictionary.....	56
2.1.4.1 Propiedades	57
2.1.4.1.1. Count.....	57
2.1.4.1.2. Keys	59
2.1.4.1.3. Values.....	61
2.1.4.2 Métodos	62
2.1.4.2.1. Add	62
2.1.4.2.2. Clear	62
2.1.4.2.3. ContainsKey	63
2.1.4.2.4. ContainsValue	64
2.1.4.2.5. Remove	65
3. Programación Avanzada	66
3.1. Programación Orientada a objetos	66
3.1.3. Encapsulamiento.....	67
3.1.3.1. Modificadores de acceso	67
3.1.3.1.1 Modificador de acceso Private.....	67
3.1.3.1.2 Modificador de acceso protected.....	68
3.1.3.1.3 Modificador de acceso public	68
3.1.4. Clases C#.....	68
3.1.4.1. Creación de clases.....	70
3.1.4.2 Get y Set	71
3.1.4.3. Instancia de clase	73
3.1.4.5 Herencia	76
3.1.5. Clases Abstractas	78
3.1.6. Polimorfismo	82

3.1.7. Interfaces	82
3.2. LinQ	83
3.2.1. Usando Linq.....	85
3.2.1.1 Operador Where en Linq	88
3.2.1.2 Operador All y Any en Linq	90
3.2.1.3 Operador Contains en Linq	92
3.2.1.4 Operador OrderBy y OrderByDescending en Linq.....	93
3.2.1.5 Operador skip y take en Linq	94
3.2.1.6 Operador select en Linq.....	96
3.2.1.7 Operador LongCount y Count en Linq	97
3.2.1.8 Operador Min y Max en Linq	99
3.2.1.9 Operador Min y Max en Linq	101
3.2.1.10 Operador sum y Aggregate en Linq	102
3.2.1.11 operador average en Linq.....	103
3.2.1.12 GroupBy en Linq.....	104
3.2.1.13 LookUp Linq	105
3.2.1.14 Join Linq	105
3.3. Persistencia de Datos JSON.....	106

1. Programación Básica C#

1.1. Fundamentos de programación en C#

1.1.1. Introducción

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. En el fondo, C# es un lenguaje orientado a objetos. Defina los tipos y su comportamiento. (<https://learn.microsoft.com/es-es/dotnet/csharp/tour-of-csharp/>)

1.1.2 Arquitectura .Net Core

.NET Core es un marco de trabajo (framework) de código abierto y multiplataforma desarrollado por Microsoft. Proporciona una plataforma para construir aplicaciones modernas, incluyendo aplicaciones web, servicios web, aplicaciones de consola y más. .NET Core es una versión modular y ligera de .NET Framework, diseñada para ser más rápida y eficiente, y es compatible con Windows, macOS y Linux.

Una de las características clave de .NET Core es su capacidad para crear aplicaciones que se ejecutan en múltiples sistemas operativos. Esto significa que puede desarrollar una aplicación en .NET Core y ejecutarla en diferentes plataformas sin necesidad de cambios significativos en el código fuente. Además, .NET Core ofrece una mayor flexibilidad en cuanto a la elección del entorno de desarrollo, ya que se puede utilizar con herramientas como Visual Studio, Visual Studio Code o la línea de comandos.

Otra ventaja de .NET Core es su rendimiento mejorado y su menor consumo de recursos. Está diseñado para ser más rápido y escalable que su predecesor, lo que lo hace adecuado para aplicaciones de alto rendimiento y escala. Además, .NET Core se integra con tecnologías modernas como Docker y la computación en la nube, lo que facilita la implementación y el despliegue de aplicaciones en entornos distribuidos.

En resumen, .NET Core es un marco de trabajo de desarrollo de software multiplataforma y de alto rendimiento, que permite la creación de aplicaciones modernas y escalables para diferentes sistemas operativos.

1.1.3. Lenguajes soportados por .Net Core

.NET Core admite varios lenguajes de programación, aunque algunos de ellos pueden tener un nivel de soporte y compatibilidad diferente. Los lenguajes principales que se pueden utilizar con .NET Core son:

C#: Es el lenguaje principal utilizado en el ecosistema de .NET Core. Es un lenguaje de programación multiparadigma y orientado a objetos que se utiliza ampliamente para desarrollar aplicaciones en .NET.

F#: Es un lenguaje funcional que se ejecuta en la plataforma .NET. F# es compatible con .NET Core y ofrece ventajas en el desarrollo de aplicaciones funcionales y científicas.

Visual Basic (VB.NET): Aunque no es tan utilizado como C#, Visual Basic es un lenguaje compatible con .NET Core. Es un lenguaje orientado a objetos y de propósito general.

Además de estos lenguajes principales, .NET Core también admite otros lenguajes, aunque con un nivel de soporte y compatibilidad variado. Algunos ejemplos incluyen:

C++/CLI: Permite utilizar el lenguaje C++ en combinación con .NET Core.

IronPython: Implementación de Python que se ejecuta en la plataforma .NET.

IronRuby: Implementación de Ruby que se ejecuta en la plataforma .NET.

TypeScript: Aunque TypeScript es un lenguaje de programación desarrollado por Microsoft, no se ejecuta directamente en .NET Core. Sin embargo, se puede integrar fácilmente en proyectos de .NET Core para desarrollar aplicaciones web utilizando Angular u otras bibliotecas de C#Script.

Es importante tener en cuenta que el nivel de soporte y compatibilidad puede variar entre los diferentes lenguajes en función de las herramientas y bibliotecas disponibles. C# es el lenguaje más ampliamente utilizado y mejor soportado en el ecosistema de .NET Core.

1.1.4. Ventajas y desventajas de Net Core

Utilizar .NET Core tiene varias ventajas y desventajas. A continuación, se presentan algunas de ellas:

Ventajas de utilizar .NET Core:

- **Multiplataforma:** .NET Core es compatible con Windows, macOS y Linux. Esto permite desarrollar aplicaciones que se ejecutan en diferentes sistemas operativos sin necesidad de realizar cambios significativos en el código fuente.
- **Rendimiento y escalabilidad:** .NET Core ha sido diseñado para ofrecer un rendimiento mejorado y un menor consumo de recursos en comparación con versiones anteriores de .NET Framework. Esto lo hace adecuado para aplicaciones de alto rendimiento y escala.
- **Modularidad:** .NET Core adopta un enfoque modular, lo que significa que solo se incluyen los componentes necesarios para una aplicación específica. Esto resulta en aplicaciones más ligeras y eficientes, y facilita la administración de dependencias.
- **Open source:** .NET Core es un proyecto de código abierto, lo que significa que su desarrollo es transparente y existe una comunidad activa que contribuye al proyecto. Esto permite una mayor transparencia, participación y mejora continua.
- **Integración con tecnologías modernas:** .NET Core se integra bien con tecnologías modernas como Docker, Kubernetes y la computación en la nube. Esto facilita la implementación y el despliegue de aplicaciones en entornos distribuidos.

Desventajas de utilizar .NET Core:

- **Menor compatibilidad con algunas bibliotecas y herramientas:** Debido a que .NET Core es una versión más reciente y modular de .NET, puede haber algunas bibliotecas y herramientas que no son completamente compatibles con él. Esto puede requerir adaptaciones o buscar alternativas.

- **Curva de aprendizaje:** Si estás familiarizado con versiones anteriores de .NET Framework, puede requerir un tiempo de aprendizaje adicional para adaptarse a los cambios y características de .NET Core.
- **Ecosistema menos maduro:** Aunque .NET Core ha ganado popularidad y ha crecido su ecosistema, aún puede haber una menor disponibilidad de ciertas bibliotecas o herramientas en comparación con .NET Framework. Sin embargo, este problema se ha ido mitigando con el tiempo y muchas bibliotecas populares ahora tienen soporte para .NET Core.
- **Menor soporte para algunas características específicas de Windows:** Aunque .NET Core es multiplataforma, algunas características específicas de Windows pueden tener un soporte limitado o requerir un enfoque diferente en comparación con .NET Framework.

1.2. Configuración del entorno de desarrollo

Para desarrollar en .NET Core, necesitarás cumplir con los siguientes requisitos:

Sistema operativo compatible: .NET Core es compatible con Windows, macOS y Linux. Asegúrate de tener un sistema operativo compatible instalado en tu máquina.

SDK de .NET Core: Debes descargar e instalar el SDK (Software Development Kit) de .NET Core correspondiente a tu sistema operativo desde el sitio web oficial de .NET Core. El SDK incluye las herramientas necesarias para desarrollar aplicaciones con .NET Core.

Entorno de desarrollo integrado (IDE): Aunque no es estrictamente necesario, se recomienda utilizar un IDE para facilitar el desarrollo en .NET Core. Microsoft Visual Studio es el IDE principal para .NET Core y ofrece características avanzadas para la programación en C# y otros lenguajes de .NET. También puedes utilizar Visual Studio Code, un editor de código ligero y altamente personalizable, que también es compatible con .NET Core.

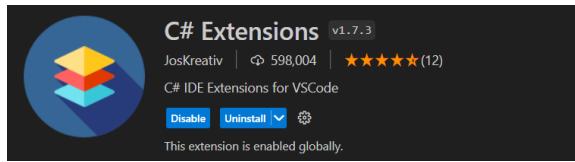
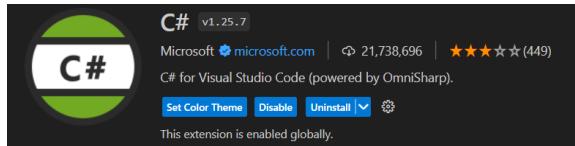
Conocimientos de programación: Para desarrollar en .NET Core, es necesario tener conocimientos de programación en C# u otros lenguajes compatibles con .NET Core, como F# o Visual Basic. Familiarizarte con los conceptos de programación orientada a objetos y los principios básicos de .NET Framework también es útil.

Control de versiones: Es recomendable utilizar un sistema de control de versiones, como Git, para mantener un registro de los cambios realizados en tu proyecto y facilitar la colaboración con otros desarrolladores.

Estos son los requisitos básicos para comenzar a desarrollar en .NET Core. A medida que te familiarices con el entorno y el desarrollo en .NET Core, también podrías necesitar aprender sobre las bibliotecas y frameworks adicionales que se utilizan comúnmente en el desarrollo de aplicaciones, como ASP.NET Core para el desarrollo web o Entity Framework Core para el acceso a bases de datos.

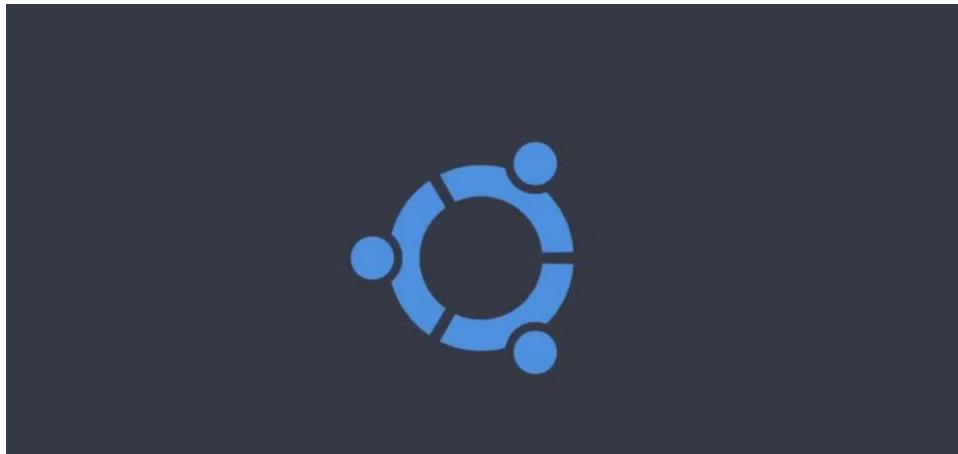
Software necesario

- **Visual Studio Code**
- **Extensiones de visual Studio Code :**

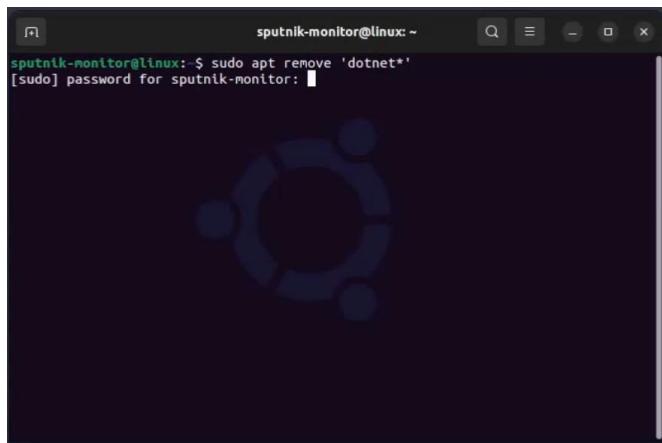


- SDK Net Core V 6.0 o 7.0
- En sistemas operativos se puede usar visual studio community edition.

1.2.1. Configuración en Linux



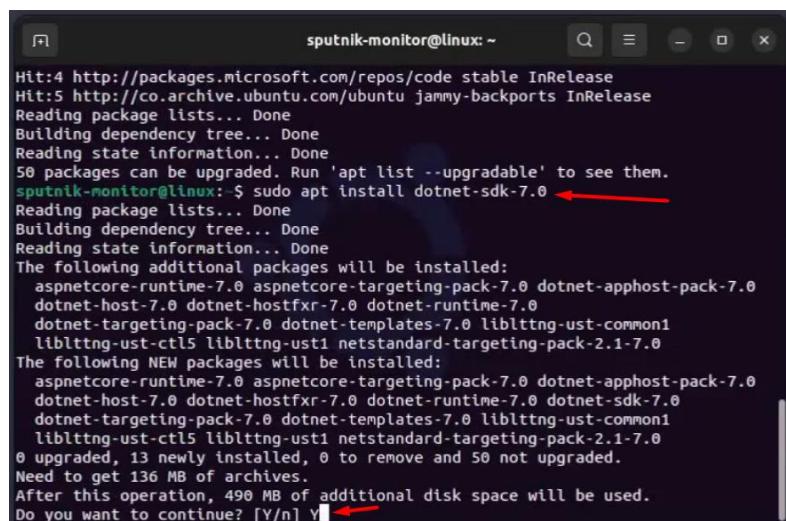
- Abrir la terminal de linux **Ctrl+Alt+T**
- Ingresar el comando **sudo apt remove 'dotnet*'** para eliminar los paquetes de net core que se encuentren instalados.



En caso de no contar con paquetes instalados se podrá visualizar el siguiente resultado:

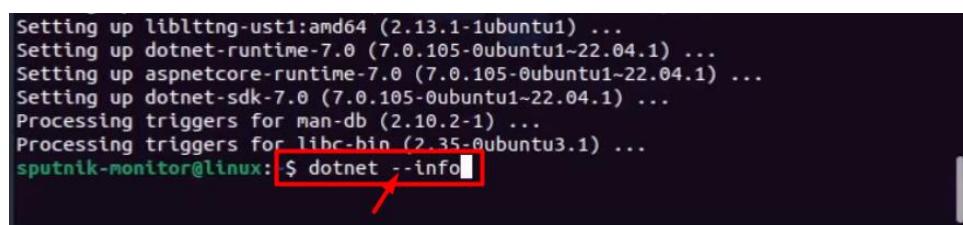
```
Note, selecting 'dotnet-apphost-pack-7.0' for glob 'dotnet*'
Note, selecting 'dotnet-sdk-7.0-source-built-artifacts' for glob 'dotnet*'
Note, selecting 'dotnet-runtime-6.0' for glob 'dotnet*'
Note, selecting 'dotnet-runtime-7.0' for glob 'dotnet*'
Package 'dotnet-apphost-pack-6.0' is not installed, so not removed
Package 'dotnet-apphost-pack-7.0' is not installed, so not removed
Package 'dotnet-host' is not installed, so not removed
Package 'dotnet-host-7.0' is not installed, so not removed
Package 'dotnet-hostfxr-6.0' is not installed, so not removed
Package 'dotnet-runtime-6.0' is not installed, so not removed
Package 'dotnet-runtime-7.0' is not installed, so not removed
Package 'dotnet-sdk-6.0' is not installed, so not removed
Package 'dotnet-sdk-6.0-source-built-artifacts' is not installed, so not removed
Package 'dotnet-sdk-7.0' is not installed, so not removed
Package 'dotnet-sdk-7.0-source-built-artifacts' is not installed, so not removed
Package 'dotnet-targeting-pack-6.0' is not installed, so not removed
Package 'dotnet-targeting-pack-7.0' is not installed, so not removed
Package 'dotnet-templates-6.0' is not installed, so not removed
Package 'dotnet-templates-7.0' is not installed, so not removed
Package 'dotnet7' is not installed, so not removed
Package 'dotnet7' is not installed, so not removed
0 upgraded, 0 newly installed, 0 to remove and 50 not upgraded.
sputnik-monitor@linux: $
```

- Ingresar el comando **sudo apt remove 'aspnetcore*'** para remover paquetes y servicios runtime de aspnetcore.
- Ejecutar el comando **sudo rm /etc/apt/sources.list.d/microsoft-prod.list** para eliminar repositorios en caso de haber instalado otras versiones de Net Core.
- Ejecutar el comando **sudo apt update** para realizar actualización de paquetes en linux.
- Ejecutamos el comando **sudo apt install dotnet-sdk-6.0** si deseamos instalar la versión 6 de .Net Core o el comando **sudo apt install dotnet-sdk-7.0** si se desea instalar la versión 7.0



```
sputnik-monitor@linux: ~
Hit:4 http://packages.microsoft.com/repos/code stable InRelease
Hit:5 http://co.archive.ubuntu.com/ubuntu jammy-backports InRelease
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
50 packages can be upgraded. Run 'apt list --upgradable' to see them.
sputnik-monitor@linux: ~$ sudo apt install dotnet-sdk-7.0 ↗
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
  aspnetcore-runtime-7.0 aspnetcore-targeting-pack-7.0 dotnet-apphost-pack-7.0
  dotnet-host-7.0 dotnet-hostfxr-7.0 dotnet-runtime-7.0
  dotnet-targeting-pack-7.0 dotnet-templates-7.0 libliltng-ust-common1
  libliltng-ust-ctl5 libliltng-ust1 netstandard-targeting-pack-2.1-7.0
The following NEW packages will be installed:
  aspnetcore-runtime-7.0 aspnetcore-targeting-pack-7.0 dotnet-apphost-pack-7.0
  dotnet-host-7.0 dotnet-hostfxr-7.0 dotnet-runtime-7.0 dotnet-sdk-7.0
  dotnet-targeting-pack-7.0 dotnet-templates-7.0 libliltng-ust-common1
  libliltng-ust-ctl5 libliltng-ust1 netstandard-targeting-pack-2.1-7.0
0 upgraded, 13 newly installed, 0 to remove and 50 not upgraded.
Need to get 136 MB of archives.
After this operation, 490 MB of additional disk space will be used.
Do you want to continue? [Y/n] Y ↗
```

- Como último paso tendremos que verificar la versión del SDK instalada, para esto se ingresa el comando **dotnet --info**



```
Setting up libliltng-ust1:amd64 (2.13.1-1ubuntu1) ...
Setting up dotnet-runtime-7.0 (7.0.105-0ubuntu1~22.04.1) ...
Setting up aspnetcore-runtime-7.0 (7.0.105-0ubuntu1~22.04.1) ...
Setting up dotnet-sdk-7.0 (7.0.105-0ubuntu1~22.04.1) ...
Processing triggers for man-db (2.10.2-1) ...
Processing triggers for libc-bin (2.35-0ubuntu3.1) ...
sputnik-monitor@linux: ~$ dotnet --info ↗
```

Como resultado final obtenemos la siguiente información:

```
sputnik-monitor@linux: ~
.NET SDKs installed:
7.0.105 [/usr/lib/dotnet/sdk]

.NET runtimes installed:
Microsoft.AspNetCore.App 7.0.5 [/usr/lib/dotnet/shared/Microsoft.AspNetCore.Ap
p]
Microsoft.NETCore.App 7.0.5 [/usr/lib/dotnet/shared/Microsoft.NETCore.App]

Other architectures found:
None

Environment variables:
Not set

global.json file:
Not found

Learn more:
https://aka.ms/dotnet/info

Download .NET:
https://aka.ms/dotnet/download
sputnik-monitor@linux:~$
```

1.2.2. Configuración en Windows

- Ingresar a la Url <https://dotnet.microsoft.com/en-us/download>



Download .NET 7.0

.NET 8 Preview Want to try out the latest preview? .NET 8.0.0-preview.4 is available.

7.0.5 Security patch (0)

Release notes Latest release date April 11, 2023

Build apps - SDK

SDK 7.0.302

OS	Installers	Binaries
Linux	Package manager instructions	Arm32 Arm32 Alpine Arm64 Arm64 Alpine x64 x64 Alpine
macOS	Arm64 x64	Arm64 x64
Windows	Arm64 x64 x86 winget instructions	Arm64 x64 x86
All	dotnet-install scripts	

Visual Studio support
Visual Studio 2022 (v17.6)
Visual Studio 2022 for Mac (v17.6)

Included in

Run apps - Runtime

ASP.NET Core Runtime 7.0.5

The ASP.NET Core Runtime enables you to run existing web/server applications. On Windows, we recommend installing the Hosting Bundle, which includes the .NET Runtime and IIS support.

IIS runtime support (ASP.NET Core Module v2)
17.0.23084.5

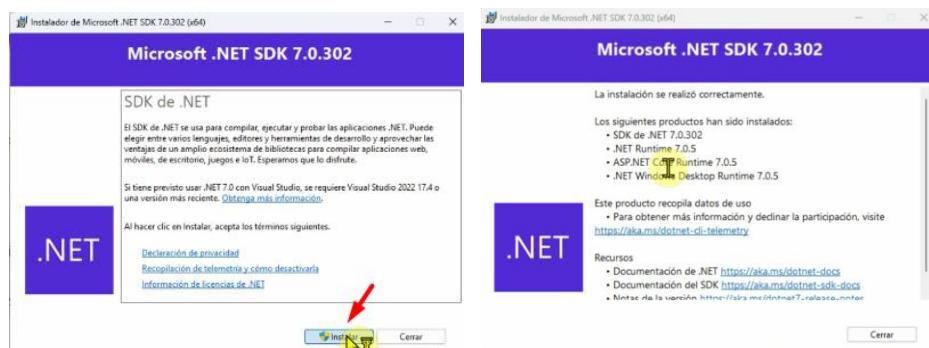
OS	Installers	Binaries
Linux	Package manager instructions	Arm32 Arm32 Alpine Arm64 Arm64 Alpine x64 x64 Alpine
macOS		Arm64 x64
Windows	Hosting Bundle x64 x86 winget instructions	Arm64 x64 x86

- Descargar el SDK 7.0 teniendo en cuenta la arquitectura del sistema operativo ya sea 32(x86) o 64(x64) bits.

SDK 7.0.302

OS	Installers	Binaries
Linux	Package manager instructions	Arm32 Arm32 Alpine Arm64 Arm64 Alpine x64 x64 Alpine
macOS	Arm64 x64	Arm64 x64
Windows	Arm64 x64 x86 winget instructions	Arm64 x64 x86
All	dotnet-install scripts	

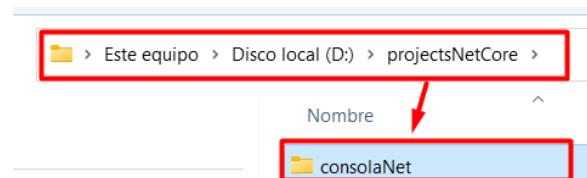
- Ejecutar el asistente de instalación y seguir los pasos indicados por el asistente.



1.3. Programación en C#

1.3.1. Creación de proyectos de consola en C# con Visual Studio Code

- Crear una carpeta principal donde van a crear los proyectos.



- Abrir la consola de windows(PowerShell o CMD) y en linux la terminal (Ctrl+Alt+T)
- Ingresar a la carpeta donde se desea crear el (los) proyectos.

```

desarrollo@DESKTOP-L4V647N MINGW64 ~
$ cd /d ←

desarrollo@DESKTOP-L4V647N MINGW64 /d
$ cd proyecto
ProyectoCursoPhpp/ proyectosJs/      proyectosPHP/

desarrollo@DESKTOP-L4V647N MINGW64 /d
$ cd project
projectosJs/      projectsNetCore/

desarrollo@DESKTOP-L4V647N MINGW64 /d
$ cd projectsNetCore/
$ ls
consolaNet/ →

desarrollo@DESKTOP-L4V647N MINGW64 /d/projectsNetCore
$ cd consolaNet/
$ →

desarrollo@DESKTOP-L4V647N MINGW64 /d/projectsNetCore/consolaNet
$ →

```

- Ingresar el comando **dotnet new console -n [Nombre del proyecto]**

```

desarrollo@DESKTOP-L4V647N MINGW64 /d/projectsNetCore/consolaNet
$ dotnet new console -n Ejercicio1_01

Esto es .NET 7.0.
-----
Versión del SDK: 7.0.302

```

1.3.2. Ejecución de proyectos

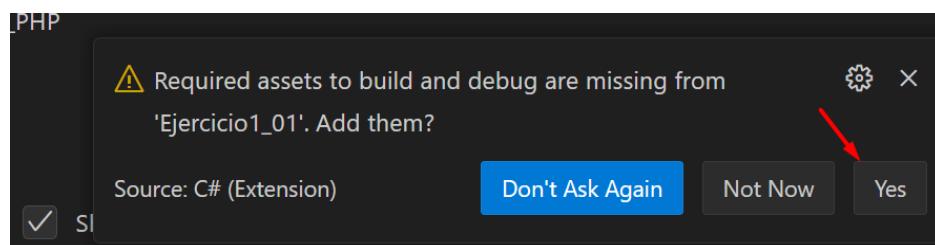
- Abra el proyecto en visual studio code

```

desarrollo@DESKTOP-L4V647N MINGW64 /d/projectsNetCore/consolaNet/Ejercicio1_01
$ code .

```

- Cuando se abre el proyecto en VS code se muestra el siguiente mensaje.



- Abrir la terminal en vs Code y digitar el comando **dotnet run**

```

desarrollo@DESKTOP-L4V647N MINGW64 /d/projectsNetCore/consolaNet/Ejercicio1_01
$ dotnet run ←
Hello, World! ←

desarrollo@DESKTOP-L4V647N MINGW64 /d/projectsNetCore/consolaNet/Ejercicio1_01
$ []

```

1.3.3. Estructura de un proyecto de consola en .NET

```
Program.cs
1 // See https://aka.ms/new-console-template for more information
2 Console.WriteLine("Hello, World!");
3
```

Bin : Contiene archivos dll, exe que son generados automáticamente cuando se ejecuta el proyecto con el comando dotnet run.

Obj : Contiene paquetes o extensiones incorporadas en el proyecto.

Program.cs : Programa principal creado al momento de generar el proyecto.

1.3.4. Variables y constantes

1.3.4.1 Variables

En programación, una variable es un espacio de memoria reservado para almacenar un valor específico. Las variables tienen un nombre único y pueden contener diferentes tipos de datos, como números, texto, booleanos, objetos, entre otros.

Al utilizar variables, los programadores pueden almacenar y manipular datos de manera dinámica en un programa. Las variables permiten almacenar valores temporales o resultados intermedios de cálculos, y también facilitan la comunicación y transferencia de datos entre diferentes partes de un programa.

```
string nombre = string.Empty;
int edad = 0;
double sueldo = 0.0;
bool estado = false;
```

1.3.4.2. Constantes

En C#, una constante es un valor inmutable que no puede cambiar una vez que se le ha asignado un valor inicial. Las constantes se declaran utilizando la palabra clave "const" y deben recibir un valor en el momento de la declaración.

```
const double iva = 16.8;
```

1.3.5. bool (Referencia de C#)

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 **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.

Ejemplo

```
C# Program.cs X  
C# Program.cs  
1  bool check = true;  
2  Console.WriteLine(check ? "Activo" : "Inactivo");  
3  Console.WriteLine(false ? "Activo" : "Inactivo");
```

- **bool check = true;** En esta línea, se declara y se inicializa una variable booleana llamada **check** con el valor **true**, lo que indica que la condición se cumple.
- **Console.WriteLine(check ? "Activo" : "Inactivo");** En esta línea, se utiliza el operador ternario **? :** para evaluar la expresión **check**. Si **check** es **true**, se devuelve el valor "**Activo**", de lo contrario, se devuelve "**Inactivo**". En este caso, dado que **check** es **true**, el resultado será "**Activo**". Luego, se imprime el resultado en la consola utilizando **Console.WriteLine()**.
- **Console.WriteLine(false ? "Activo" : "Inactivo");** En esta línea, se realiza una evaluación similar a la anterior, pero en este caso, la expresión evaluada es **false**. Dado que **false** es falso, se devuelve el valor "**Inactivo**". Luego, se imprime el resultado en la consola utilizando **Console.WriteLine()**.

En resumen, el código muestra cómo utilizar el operador ternario para tomar decisiones basadas en el valor booleano de una variable y devolver diferentes resultados en función de esa evaluación.

1.3.6. Tipos numéricos enteros

Los tipos numéricos integrales representan números enteros. Todos los tipos numéricos integrales son tipos de valor. También son tipos simples y se pueden inicializar con literales. Todos los tipos numéricos enteros admiten operadores aritméticos, lógicos bit a bit, de comparación y de igualdad.

Palabra clave/tipo de C#	Intervalo	Tamaño	Tipo de .NET
<code>sbyte</code>	De -128 a 127	Entero de 8 bits con signo	<code>System.SByte</code>
<code>byte</code>	De 0 a 255	Entero de 8 bits sin signo	<code>System.Byte</code>
<code>short</code>	De -32 768 a 32 767	Entero de 16 bits con signo	<code>System.Int16</code>
<code>ushort</code>	De 0 a 65.535	Entero de 16 bits sin signo	<code>System.UInt16</code>
<code>int</code>	De -2.147.483.648 a 2.147.483.647	Entero de 32 bits con signo	<code>System.Int32</code>
<code>uint</code>	De 0 a 4.294.967.295	Entero de 32 bits sin signo	<code>System.UInt32</code>
<code>long</code>	De -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807	Entero de 64 bits con signo	<code>System.Int64</code>
<code>ulong</code>	De 0 a 18.446.744.073.709.551.615	Entero de 64 bits sin signo	<code>System.UInt64</code>
<code>nint</code>	Depende de la plataforma (calculada en tiempo de ejecución)	Entero de 64 bits o 32 bits con signo	<code>System.IntPtr</code>
<code>nuint</code>	Depende de la plataforma (calculada en tiempo de ejecución)	Entero de 64 bits o 32 bits sin signo	<code>System.UIntPtr</code>

Fuente : <https://learn.microsoft.com/es-es/dotnet/csharp/language-reference/builtin-types/integral-numeric-types>

1.3.6.1. Tipos Simples

Palabra reservada	Tipo con alias
<code>sbyte</code>	<code>System.SByte</code>
<code>byte</code>	<code>System.Byte</code>
<code>short</code>	<code>System.Int16</code>
<code>ushort</code>	<code>System.UInt16</code>
<code>int</code>	<code>System.Int32</code>
<code>uint</code>	<code>System.UInt32</code>
<code>long</code>	<code>System.Int64</code>
<code>ulong</code>	<code>System.UInt64</code>
<code>char</code>	<code>System.Char</code>
<code>float</code>	<code>System.Single</code>
<code>double</code>	<code>System.Double</code>
<code>bool</code>	<code>System.Boolean</code>
<code>decimal</code>	<code>System.Decimal</code>

1.3.6.2. Tipos enteros

C# admite nueve tipos enteros: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong` y `char`.

Los tipos enteros tienen los siguientes tamaños y rangos de valores:

- El sbyte tipo representa enteros de 8 bits con signo con valores comprendidos entre -128 y 127.
- El byte tipo representa enteros de 8 bits sin signo con valores comprendidos entre 0 y 255.
- El short tipo representa enteros de 16 bits con signo con valores comprendidos entre -32768 y 32767.
- El ushort tipo representa enteros de 16 bits sin signo con valores comprendidos entre 0 y 65535.
- El int tipo representa enteros de 32 bits con signo con valores comprendidos entre -2147483648 y 2147483647.
- El uint tipo representa enteros de 32 bits sin signo con valores comprendidos entre 0 y 4294967295.
- El long tipo representa enteros de 64 bits con signo con valores comprendidos entre -9223372036854775808 y 9223372036854775807.
- El ulong tipo representa enteros de 64 bits sin signo con valores comprendidos entre 0 y 18446744073709551615.
- El char tipo representa enteros de 16 bits sin signo con valores comprendidos entre 0 y 65535. El conjunto de valores posibles para el tipo char corresponde al juego de caracteres Unicode. Aunque char tiene la misma representación que ushort , no todas las operaciones permitidas en un tipo se permiten en el otro.

1.3.6.3. Tipos numéricos de punto flotante (referencia de C#)

Los tipos numéricos de punto flotante representan números reales. Todos los tipos numéricos de punto flotante son tipos de valor. También son tipos simples y se pueden inicializar con literales. Todos los tipos de punto flotante numéricos admiten operadores aritméticos, de comparación y de igualdad.

C# admite los siguientes tipos de punto flotante predefinidos:

Palabra clave/tipo de C#	Intervalo aproximado	Precisión	Tamaño	Tipo de .NET
<code>float</code>	De $\pm 1,5 \times 10^{-45}$ a $\pm 3,4 \times 10^{38}$	De 6 a 9 dígitos aproximadamente	4 bytes	System.Single
<code>double</code>	De $\pm 5,0 \times 10^{-324}$ a $\pm 1,7 \times 10^{308}$	De 15 a 17 dígitos aproximadamente	8 bytes	System.Double
<code>decimal</code>	De $\pm 1,0 \times 10^{-28}$ to $\pm 7,9228 \times 10^{28}$	28-29 dígitos	16 bytes	System.Decimal

```
1  using System.Numerics;
2
3      0 references
4  internal class Program
5  {
6      0 references
7      private static void Main(string[] args)
8      {
9          var a = 12.3;
10         double b = 12.3;
11         Console.WriteLine($"El valor de a es = {a} ");
12         Console.WriteLine($"El valor de a es = {b} ");
13     }
14 }
```

- **var a = 12.3;** En esta línea, se declara e inicializa una variable llamada **a** utilizando la inferencia de tipos (**var**). El valor asignado a **a** es **12.3**, que es un número de tipo **double**.
- **double b = 12.3;** En esta línea, se declara y se inicializa explícitamente una variable llamada **b** de tipo **double**. El valor asignado a **b** es también **12.3**.
- **Console.WriteLine(\$"El valor de a es = {a} ");** En esta línea, se utiliza la función **Console.WriteLine()** para imprimir un mensaje en la consola. La cadena se define utilizando interpolación de cadenas, que se indica con el símbolo **\$**. Dentro de la cadena, se utiliza la sintaxis **{a}** para incluir el valor de la variable **a** en la cadena. El resultado impreso será "El valor de a es = 12.3".
- **Console.WriteLine(\$"El valor de a es = {b} ");** En esta línea, se realiza una operación similar a la anterior, pero en este caso, se imprime el valor de la variable **b**. El resultado impreso será "El valor de b es = 12.3".

En resumen, el código muestra cómo asignar valores a variables de tipo **double** y cómo imprimir esos valores en la consola utilizando interpolación de cadenas para construir mensajes con los valores de las variables.

Literales reales

El tipo de un literal real viene determinado por su sufijo, como se indica a continuación:

- El literal sin sufijo o con el sufijo **d** o **D** es del tipo **double**
- El literal con el sufijo **f** o **F** es del tipo **float**
- El literal con el sufijo **m** o **M** es del tipo **decimal**

```

using System.Numerics;

internal class NewBaseType
{
    private static void Main(string[] args)
    {
        double d = 3D;
        d = 4d;
        d = 3.934_001;
        Console.WriteLine($"{d}");
        float f = 3_000.5F;
        f = 5.4f;
        Console.WriteLine($"{f}");
        decimal myMoney = 3_000.5m;
        myMoney = 400.75M;
        Console.WriteLine($"{myMoney}");
    }
}

internal class Program : NewBaseType
{
}

```

- **double d = 3D;** Se declara una variable **d** de tipo **double** y se le asigna el valor **3D**. El sufijo **D** indica que el número literal es de tipo **double**.
- **d = 4d;** Aquí se actualiza el valor de la variable **d** a **4d**, utilizando el sufijo **d** para indicar que es un **double**.
- **d = 3.934_001;** Esta línea asigna a **d** el valor **3.934_001**, que es un número decimal representado en notación de punto flotante.
- **Console.WriteLine(\$"{d}");** Se utiliza la interpolación de cadenas para imprimir el valor de **d** en la consola.
- **float f = 3_000.5F;** Se declara una variable **f** de tipo **float** y se le asigna el valor **3_000.5F**. El sufijo **F** indica que el número literal es de tipo **float**.
- **f = 5.4f;** Esta línea actualiza el valor de **f** a **5.4f**, utilizando el sufijo **f** para indicar que es un **float**.
- **Console.WriteLine(\$"{f}");** Se utiliza la interpolación de cadenas para imprimir el valor de **f** en la consola.
- **decimal myMoney = 3_000.5m;** Se declara una variable **myMoney** de tipo **decimal** y se le asigna el valor **3_000.5m**. El sufijo **m** indica que el número literal es de tipo **decimal**.
- **myMoney = 400.75M;** Aquí se actualiza el valor de **myMoney** a **400.75M**, utilizando el sufijo **M** para indicar que es un **decimal**.
- **Console.WriteLine(\$"{myMoney}");** Se utiliza la interpolación de cadenas para imprimir el valor de **myMoney** en la consola.
- En resumen, el código declara variables de tipo **double**, **float** y **decimal**, les asigna valores y luego los imprime en la consola utilizando la interpolación de cadenas.

desarrollo@DESKTOP-L4V647N MINGW64 /d/projectsNetCore/consolaNet/Ejercicio1_01

\$ dotnet run

El valor de a es = 12,3
El valor de a es = 12,3

3,934001
5,4
400,75

1.3.6.4. Cadenas con formato numérico estándar

Las cadenas de formato numérico estándar se utilizan para dar formato a tipos numéricos comunes. La forma de una cadena de formato numérico estándar es [format specifier][precision specifier], donde:

El especificador de formato es un carácter alfabético único que especifica el tipo de formato numérico, como, por ejemplo, moneda o porcentaje. Cualquier cadena de formato numérico que contenga más de un carácter alfabético, incluido el espacio en blanco, se interpreta como una cadena de formato numérico personalizado. Para obtener más información, consulte Cadenas con formato numérico personalizado.

El especificador de precisión es un entero opcional que afecta al número de dígitos de la cadena resultante. En .NET 7 y versiones posteriores, el valor de precisión máximo es 999,999,999. En .NET 6, el valor de precisión máximo es Int32.MaxValue. En versiones anteriores de .NET, la precisión puede oscilar entre 0 y 99. El especificador de precisión controla el número de dígitos en la representación de cadena de un número. No redondea el número en sí. Para realizar una operación de redondeo, use el método Math.Ceiling, Math.Floor o Math.Round.

Especificadores de formato estándar

Especificador de formato	NOMBRE	Descripción	Ejemplos
"C" o "c"	Moneda	Resultado: un valor de divisa. Compatible con: todos los tipos numéricos. Especificador de precisión: número de dígitos decimales. Especificador de precisión predeterminado: Definido por NumberFormatInfo.CurrencyDecimalDigits . Más información: Especificador de formato de divisa ("C") .	123.456 ("C", en-US) -> \$123.46 123.456 ("C", fr-FR) -> 123,46 € 123.456 ("C", ja-JP) -> ¥123 -123.456 ("C3", en-US) -> (\$123.456) -123.456 ("C3", fr-FR) -> -123,456 € -123.456 ("C3", ja-JP) -> -¥123.456

"D" o "d"	Decimal	<p>Resultado: dígitos enteros con signo negativo opcional.</p> <p>Compatible con: solo tipos enteros.</p> <p>Especificador de precisión: número mínimo de dígitos.</p> <p>Especificador de precisión predeterminado: número mínimo de dígitos necesario.</p> <p>Más información: Especificador de formato decimal ("D").</p>	1234 ("D") -> 1234 -1234 ("D6") -> -001234 1052.0329112756 ("E", en-US) -> 1.052033E+003 1052.0329112756 ("e", fr-FR) -> 1.052033e+003 -1052.0329112756 ("e2", en-US) -> -1.05e+003 -1052.0329112756 ("E2", fr-FR) -> -1.05E+003
"E" o "e"	Exponencial (científico)	<p>Resultado: notación exponencial.</p> <p>Compatible con: todos los tipos numéricos.</p> <p>Especificador de precisión: número de dígitos decimales.</p> <p>Especificador de precisión predeterminado: 6.</p> <p>Más información: Especificador de formato exponencial ("E").</p>	1052.0329112756 ("E", en-US) -> 1.052033E+003 1052.0329112756 ("e", fr-FR) -> 1.052033e+003 -1052.0329112756 ("e2", en-US) -> -1.05e+003 -1052.0329112756 ("E2", fr-FR) -> -1.05E+003
"F" o "f"	Punto fijo	<p>Resultado: dígitos integrales y decimales con signo negativo opcional.</p> <p>Compatible con: todos los tipos numéricos.</p> <p>Especificador de precisión: número de dígitos decimales.</p> <p>Especificador de precisión predeterminado: Definido por NumberFormatInfo.NumberDecimalDigits.</p> <p>Más información: Especificador de formato de punto fijo ("F").</p>	1234.567 ("F", en-US) -> 1234.57 1234.567 ("F", de-DE) -> 1234.57 1234 ("F1", en-US) -> 1234.0 1234 ("F1", de-DE) -> 1234.0 -1234.56 ("F4", en-US) -> -1234.5600 -1234.56 ("F4", de-DE) -> -1234.5600
"G" o "g"	General	<p>Resultado: notación de punto fijo o científica, la que sea más compacta.</p> <p>Compatible con: todos los tipos numéricos.</p> <p>Especificador de precisión: número de dígitos significativos.</p> <p>Especificador de precisión predeterminado: depende del tipo numérico.</p> <p>Más información: Especificador de formato general ("G").</p>	-123.456 ("G", en-US) -> -123.456 -123.456 ("G", sv-SE) -> -123.456 123.4546 ("G4", en-US) -> 123.5 123.4546 ("G4", sv-SE) -> 123.5 -1.234567890e-25 ("G", en-US) -> -1.23456789E-25 -1.234567890e-25 ("G", sv-SE) -> -1.23456789E-25

"N" o "n"	número	Resultado: dígitos integrales y decimales, separadores de grupos y un separador decimal con signo negativo opcional. Compatible con: todos los tipos numéricos. Especificador de precisión: número deseado de decimales. Especificador de precisión predeterminado: Definido por NumberFormatInfo.NumberDecimalDigits . Más información: Especificador de formato numérico ("N") .	1234.567 ("N", en-US) -> 1,234.57 1234.567 ("N", ru-RU) -> 1 234.57 1234 ("N1", en-US) -> 1,234.0 1234 ("N1", ru-RU) -> 1 234.0 -1234.56 ("N3", en-US) -> -1,234.560 -1234.56 ("N3", ru-RU) -> -1 234.560
"P" o "p"	Porcentaje	Resultado: número multiplicado por 100 y mostrado con un símbolo de porcentaje. Compatible con: todos los tipos numéricos. Especificador de precisión: número deseado de decimales. Especificador de precisión predeterminado: Definido por NumberFormatInfo.PercentDecimalDigits . Más información: Especificador de formato de porcentaje ("P") .	1 ("P", en-US) -> 100.00 % 1 ("P", fr-FR) -> 100,00 % -0.39678 ("P1", en-US) -> -39.7 % -0.39678 ("P1", fr-FR) -> -39.7 %
"R" o "r"	Acción de ida y vuelta	Resultado: cadena que puede aplicar acciones de ida y vuelta (round-trip) a un número idéntico. Compatible con: Single , Double y BigInteger . Nota: Se recomienda solo para el tipo BigInteger . Para los tipos Double , use "G17"; para los tipos Single , use "G9". Especificador de precisión: ignorado. Más información: Especificador de formato de operación de ida y vuelta ("R") .	123456789.12345678 ("R") -> 123456789.12345678 -1234567890.12345678 ("R") -> -1234567890.12345678
"X" o "x"	Hexadecimal	Resultado: cadena hexadecimal. Compatible con: solo tipos enteros. Especificador de precisión: número de dígitos en la cadena de resultado. Más información: Especificador de formato hexadecimal (X) .	255 ("X") -> FF -1 ("X") -> ff 255 ("x4") -> 00ff -1 ("X4") -> 0OFF
Cualquier otro carácter único	Especificador desconocido	Resultado: produce FormatException en tiempo de ejecución.	

Ejemplos

```
using System.Numerics;

internal class NewBaseType
{
    private static void Main(string[] args)
    {
        decimal value = 123.456m;
        /*
         * Se puede pasar al método TryFormat o a una sobrecarga de
         el método ToString que tiene un parámetro format. En el ejemplo s
         iguiente se da formato a un valor numérico como una cadena de div
         isa en la referencia cultural actual (en este caso, en-US).
        */
        Console.WriteLine(value.ToString("C2"));
        // Displays $123.46
    }
}

internal class Program : NewBaseType
{
}
```

```
using System.Numerics;

internal class NewBaseType
{
    private static void Main(string[] args)
    {
        /*
         * Se puede proporcionar como el argumento formatString de un elemento de format
         o usado con métodos como String.Format, Console.WriteLine y String.Format.AppendFormat.
         Para obtener más información, consulte Formatos compuestos. En el ejemplo siguiente se
         usa un elemento de formato para insertar un valor de divisa en una cadena.
        */
        decimal value = 123.456m;
        Console.WriteLine("Your account balance is {0:C2}.", value);
        // Displays "Your account balance is $123.46."
    }
}
```

```
using System.Numerics;

internal class NewBaseType
{
    private static void Main(string[] args)
    {
        decimal[] amounts = { 16305.32m, 18794.16m };
        Console.WriteLine("    Balance Inicial      Balance final");
        Console.WriteLine("    {0,-28:C2}{{1,14:C2}", amounts[0], amounts[1]);
    }
}
```

- **using System;** Esta línea permite el acceso a los miembros del espacio de nombres **System**, que contiene tipos y funciones fundamentales de .NET.
- **internal class NewBaseType** Se declara una clase llamada **NewBaseType**. La palabra clave **internal** indica que la clase solo es accesible dentro del ensamblado actual.
- **private static void Main(string[] args)** Se define el método **Main**, que es el punto de entrada del programa. Recibe un arreglo de cadenas llamado **args** como parámetro.
- **decimal[] amounts = { 16305.32m, 18794.16m };** Se declara y se inicializa un arreglo de tipo **decimal** llamado **amounts**. Contiene dos valores decimales.
- **Console.WriteLine("{0,-28} {1,14}", "Balance Inicial", "Balance Final");** Esta línea imprime una línea de encabezado en la consola, con los textos "Balance Inicial" y "Balance Final". Los números entre corchetes {} indican las posiciones donde se insertarán los valores proporcionados.
- **Console.WriteLine("{0,-28:C2} {1,14:C2}", amounts[0], amounts[1]);** Se utiliza la interpolación de cadenas para imprimir los valores del arreglo **amounts**. El formato **C2** especifica que los valores se deben mostrar como moneda con dos decimales. Los números entre corchetes {} indican las posiciones donde se insertarán los valores proporcionados. La opción **-28** alinea el texto a la izquierda ocupando 28 caracteres, y la opción **,14** alinea el texto a la derecha ocupando 14 caracteres.

En resumen, el código declara un arreglo de tipo **decimal** con valores de saldo inicial y saldo final. Luego, imprime una línea de encabezado y los valores del arreglo formateados como moneda en la consola.

1.4. Entrada y salida de datos

En c# al igual que cualquier lenguaje de programación existen métodos que permiten ingresar datos a través de dispositivos de entrada y salida a continuación se estudiaran estos dos métodos.

Console -> Console es una clase que permite enviar mensajes en pantalla. En c# la clase console posee los siguientes métodos.

WriteLine -> Permite imprimir en consola un texto e inserta un salto de línea al final.

```

1  using System.Numerics;
2
3      0 references
4  class Program
5  {
6      0 references
7      private static void Main(string[] args)
8      {
9          Console.WriteLine("Este es un mensaje de texto");
10         Console.WriteLine("Este es otro mensaje de texto");
11     }
12 }
```

- **Console.WriteLine("Este es un mensaje de texto");** Esta línea imprime en la consola el texto "Este es un mensaje de texto". El método **WriteLine** muestra el texto en una nueva línea.

- **Console.WriteLine("Este es otro mensaje de texto");** Esta línea imprime en la consola el texto "Este es otro mensaje de texto". Al igual que la línea anterior, el método **WriteLine** muestra el texto en una nueva línea.

En resumen, el código imprime dos mensajes de texto en la consola, cada uno en una línea separada.

```
desarrollo@DESKTOP-L4V647N MINGW64 /d/projectsNetCore/consolaNet/Ejercicio1_01
$ dotnet run
Este es un mensaje de texto
Este es otro mensaje de texto
```

Write -> Permite imprimir en consola un texto sin insertar salto de línea al final.

```
1  using System.Numerics;
2
3  0 references
4  class Program
5  {
6      0 references
7      private static void Main(string[] args)
8      {
9          Console.Write("El dia esta soleado y");
10         Console.WriteLine(" hace calor");
11     }
12 }
```

```
desarrollo@DESKTOP-L4V647N MINGW64 /d/projectsNetCore/consolaNet/Ejercicio1_01
$ dotnet run
El dia esta soleado y hace calor
```

Entrada de datos -> C# permite ingresar información desde el teclado a través de la función **ReadLine**

```
1  using System.Numerics;
2
3  0 references
4  class Program
5  {
6      0 references
7      private static void Main(string[] args)
8      {
9          Console.WriteLine("Hola este es mi primer programa. Cual es tu nombre?");
10         string ? name = Console.ReadLine();
11         Console.WriteLine($"hola {name}, Esta es una buena forma de conocerte.");
12     }
13 }
```

```
desarrollo@DESKTOP-L4V647N MINGW64 /d/projectsNetCore/consolaNet/Ejercicio1_01
$ dotnet run
Hola este es mi primer programa. Cual es tu nombre?
Johlver
hola Johlver, Esta es una buena forma de conocerte.
```

1.5. Conversión de tipos de datos

En C#, se pueden realizar conversiones entre diferentes tipos de datos utilizando diferentes métodos y operadores proporcionados por el lenguaje. En C#, se pueden realizar las siguientes conversiones de tipos:

- **Conversiones implícitas:** no se requiere ninguna sintaxis especial porque la conversión siempre es correcta y no se perderá ningún dato. Los ejemplos incluyen conversiones de tipos enteros más pequeños a más grandes, y conversiones de clases derivadas a clases base.
- **Conversiones explícitas:** las conversiones explícitas requieren una [expresión Cast](#). La conversión es necesaria si es posible que se pierda información en la conversión, o cuando es posible que la conversión no sea correcta por otros motivos. Entre los ejemplos típicos están la conversión numérica a un tipo que tiene menos precisión o un intervalo más pequeño, y la conversión de una instancia de clase base a una clase derivada.
- **Conversiones definidas por el usuario:** las conversiones definidas por el usuario se realizan por medio de métodos especiales que se pueden definir para habilitar las conversiones explícitas e implícitas entre tipos personalizados que no tienen una relación de clase base-clase derivada. Para obtener más información, vea [Operadores de conversión definidos por el usuario](#).
- **Conversiones con clases del asistente:** para realizar conversiones entre tipos no compatibles, como enteros y objetos [System.DateTime](#), o cadenas hexadecimales y matrices de bytes puede usar la clase [System.BitConverter](#), la clase [System.Convert](#) y los métodos Parse de los tipos numéricos integrados, como [Int32.Parse](#). Para obtener más información, consulte Procedimiento Convertir una matriz de bytes en un valor int, Procedimiento Convertir una cadena en un número y Procedimiento Convertir cadenas hexadecimales en tipos numéricos.

1.5.1. Conversiones implícitas

Para los tipos numéricos integrados, se puede realizar una conversión implícita cuando el valor que se va a almacenar se puede encajar en la variable sin truncarse ni redondearse. Para los tipos enteros, esto significa que el intervalo del tipo de origen es un subconjunto apropiado del intervalo para el tipo de destino. Por ejemplo, una variable de tipo long (entero de 64 bits) puede almacenar cualquier valor que un tipo int (entero de 32 bits) pueda almacenar. En el ejemplo siguiente, el compilador convierte de forma implícita el valor de num en la parte derecha a un tipo long antes de asignarlo a bigNum.

```

1  using System.Numerics;
2
3      0 references
4  class Program
5  {
6      0 references
7      private static void Main(string[] args)
8      {
9          int num = 2147483647;
10         long bigNum = num;
11     }
12 }
```

Conversiones numéricas implícitas

En la tabla siguiente se muestran las conversiones implícitas predefinidas entre los tipos numéricos integrados:

De	En
sbyte	short, int, long, float, double, decimal, nint
byte	short, ushort, int, uint, long, ulong, float, double, decimal, nint, uint
short	int, long, float, double, decimal, nint
ushort	int, uint, long, ulong, float, double, decimal, nint, uint
int	long, float, double, decimal, nint
uint	long, ulong, float, double, decimal, nint
long	float, double, decimal
ulong	float, double, decimal
float	double
nint	long, float, double, decimal
uint	ulong, float, double, decimal

Conversiones numéricas explícitas

La siguiente tabla muestra las conversiones explícitas predefinidas entre tipos numéricos integrados para los que no hay ninguna conversión implícita:

De	En
sbyte	<code>byte, ushort, uint, ulong o uint</code>
byte	<code>sbyte</code>
short	<code>sbyte, byte, ushort, uint, ulong o uint</code>
ushort	<code>sbyte, byte o short</code>
int	<code>sbyte, byte, short, ushort, uint, ulong o uint</code>
uint	<code>sbyte, byte, short, ushort, int o int</code>
long	<code>sbyte, byte, short, ushort, int, uint, ulong, nint o uint</code>
ulong	<code>sbyte, byte, short, ushort, int, uint, long, nint o uint</code>
float	<code>sbyte, byte, short, ushort, int, uint, long, ulong, decimal, nint o uint</code>
double	<code>sbyte, byte, short, ushort, int, uint, long, ulong, float, decimal, nint o uint</code>
decimal	<code>sbyte, byte, short, ushort, int, uint, long, ulong, float, double, nint o uint</code>
nint	<code>sbyte, byte, short, ushort, int, uint, ulong o uint</code>
nuint	<code>sbyte, byte, short, ushort, int, uint, long o uint</code>

1.6. Estructuras de control

Las estructuras de control en programación son mecanismos o bloques de código que permiten controlar el flujo de ejecución de un programa. Estas estructuras se utilizan para tomar decisiones y repetir bloques de código según ciertas condiciones.

Hay tres tipos principales de estructuras de control:

- Estructuras de control condicional: Estas estructuras permiten tomar decisiones basadas en una condición. Los bloques de código se ejecutan solo si se cumple la condición especificada. Los ejemplos más comunes de estructuras de control condicional son:
 - La estructura "if" (si): Permite ejecutar un bloque de código solo si una condición es verdadera.
 - La estructura "if-else" (si-sino): Permite ejecutar un bloque de código si una condición es verdadera y otro bloque de código si la condición es falsa.
 - La estructura "switch" (interruptor): Permite seleccionar uno de varios bloques de código para ejecutar, según el valor de una expresión.
- Estructuras de control de bucle: Estas estructuras permiten repetir un bloque de código múltiples veces mientras se cumpla una condición o un número determinado de veces. Los bucles son útiles cuando se necesita realizar una tarea repetitiva. Los ejemplos más comunes de estructuras de control de bucle son:
 - El bucle "while" (mientras): Ejecuta un bloque de código siempre que se cumpla una condición.
 - El bucle "do-while" (hacer-mientras): Ejecuta un bloque de código al menos una vez y luego repite mientras se cumpla una condición.
 - El bucle "for": Repite un bloque de código un número específico de veces, controlado por una variable contador.

- Estructuras de control de salto: Estas estructuras permiten alterar el flujo normal de ejecución del programa, saltando a una parte específica del código. Los ejemplos más comunes de estructuras de control de salto son:
 - La instrucción "break": Termina la ejecución de un bucle o una estructura de control de manera anticipada.
 - La instrucción "continue": Salta a la siguiente iteración de un bucle, ignorando el resto del código en esa iteración.
 - La instrucción "return": Finaliza la ejecución de una función y devuelve un valor opcional.

Estas estructuras de control proporcionan a los programadores la capacidad de tomar decisiones y repetir bloques de código según sea necesario, lo que les permite crear programas más flexibles y eficientes.

1.6.1. Estructuras condicionales

1.6.1.1 if

La sentencia if en C# se utiliza para evaluar una expresión lógica y ejecutar un bloque de código si la expresión se evalúa como verdadera.

```
if (expresion) {
    // bloque de código a ejecutar si la
    // expresión es verdadera
}
```

Sintaxis:

Ejemplo:

```
int totalJugador = 0;
int totalDealer = 15;
string message = "";

if (totalJugador > totalDealer)
{
    message = "Haz ganado";
}
```

1.6.1.2. if ...else

```
if(expresion){

}else{

}
```

Sintaxis:

Ejemplo

```
if (totalJugador > totalDealer)
{
    message = "Haz ganado";
}else{
    message = "Haz perdido";
}
```

1.6.1.3. if....else if..else

```
if(expresion){  
    }else if(expresion){  
    }else(  
    )
```

Sintaxis:)

Ejemplo

```
int totalJugador = 0;  
int totalDealer = 15;  
string message = "";  
  
if (totalJugador > totalDealer)  
{  
    message = "Haz ganado";  
}else if (totalDealer == totalJugador){  
    message = "Haz empatado con el dealer";  
}else{  
    message = "Haz perdido con el dealer";  
}
```

1.6.1.4. switch

En C#, la estructura de control "switch" (interruptor) se utiliza para seleccionar uno de varios bloques de código para ejecutar, según el valor de una expresión. Proporciona una forma más concisa y estructurada de manejar múltiples casos.

```
switch (expresión)  
{  
    case valor1:  
        // Bloque de código para el caso valor1  
        break;  
    case valor2:  
        // Bloque de código para el caso valor2  
        break;  
    // Más casos...  
    default:  
        // Bloque de código para el caso por defecto  
        break;  
}
```

Sintaxis: }

Ejemplo:

```

int diaSemana = 3;
string nombreDia;

switch (diaSemana)
{
    case 1:
        nombreDia = "Lunes";
        break;
    case 2:
        nombreDia = "Martes";
        break;
    case 3:
        nombreDia = "Miércoles";
        break;
    case 4:
        nombreDia = "Jueves";
        break;
    case 5:
        nombreDia = "Viernes";
        break;
    case 6:
        nombreDia = "Sábado";
        break;
    case 7:
        nombreDia = "Domingo";
        break;
    default:
        nombreDia = "Día inválido";
        break;
}

Console.WriteLine("El día seleccionado es: " + nombreDia);

```

1.6.2 Estructura repetitivas (Iterativas o Ciclos)

1.6.2.1. for

El ciclo "for" en C# es una estructura de control de bucle que permite ejecutar un bloque de código un número específico de veces. Es especialmente útil cuando se conoce la cantidad exacta de iteraciones que se deben realizar.

```

for (inicialización; condición; iteración)
{
    // Bloque de código a repetir
}

```

Sintaxis:

- "inicialización": Es una expresión que se ejecuta antes de comenzar el bucle y se utiliza para inicializar una variable de control. Por ejemplo, se puede establecer una variable contador en 0.
- "condición": Es una expresión booleana que se evalúa antes de cada iteración. Si la condición es verdadera, se ejecuta el bloque de código; de lo contrario, se sale del bucle. Por ejemplo, se puede establecer una condición para que el bucle se ejecute mientras el contador sea menor que cierto valor.

- "iteración": Es una expresión que se ejecuta después de cada iteración y se utiliza para actualizar la variable de control. Por ejemplo, se puede incrementar el contador en 1 en cada iteración.

Ejemplo:

```
for (int i = 1; i <= 5; i++)
{
    Console.WriteLine(i);
}
```

1.6.1.2. while

El ciclo "while" en C# es una estructura de control de bucle que permite repetir un bloque de código mientras se cumpla una condición. A diferencia del ciclo "for", el ciclo "while" se repite indefinidamente hasta que la condición se evalúe como falsa.

```
while (condición)
{
    // Bloque de código a repetir
}
```

Sintaxis:

"condición": Es una expresión booleana que se evalúa antes de cada iteración. Si la condición es verdadera, se ejecuta el bloque de código; de lo contrario, se sale del bucle.

Ejemplo

```
int i = 1;
while (i <= 5)
{
    Console.WriteLine(i);
    i++;
}
```

1.6.1.3. do...while

El ciclo "do-while" en C# es una estructura de control de bucle similar al ciclo "while", pero con una diferencia clave: el bloque de código se ejecuta al menos una vez antes de evaluar la condición.

```
do
{
    // Bloque de código a repetir
} while (condición);
```

Sintaxis:

Ejemplo:

```

int numero;
do
{
    Console.WriteLine("Ingrese un número mayor que cero: ");
    string input = Console.ReadLine();
    numero = Convert.ToInt32(input);
} while (numero <= 0);

Console.WriteLine("Número válido ingresado: " + numero);

```

- **int numero;** Se declara una variable llamada **numero** de tipo entero para almacenar el número ingresado por el usuario.
- **do** Inicio del bucle **do-while**. El código dentro de este bloque se ejecutará al menos una vez y luego se repetirá mientras se cumpla la condición especificada.
- **{** Apertura del bloque de código del bucle **do-while**.
- **Console.WriteLine("Ingrese un número mayor que cero: ");** Muestra un mensaje en la consola para solicitar al usuario que ingrese un número mayor que cero.
- **string input = Console.ReadLine();** Lee la entrada del usuario desde la consola y la almacena en la variable **input** como una cadena de texto.
- **numero = Convert.ToInt32(input);** Convierte la cadena **input** en un número entero utilizando el método **Convert.ToInt32** y asigna el valor a la variable **numero**.
- **} while (numero <= 0);** Verifica si el número ingresado es menor o igual a cero. Si es así, se repite el bucle; de lo contrario, se sale del bucle.
- **Console.WriteLine("Número válido ingresado: " + numero);** Muestra un mensaje en la consola para indicar que se ha ingresado un número válido, junto con el valor de la variable **numero**.

En resumen, el código solicita al usuario que ingrese un número mayor que cero y se repite hasta que se ingrese un número válido. Una vez que se ingresa un número válido, se muestra en la consola.

2. Programación Basica

2.1. Estructura de datos

2.1.1. Arreglos

En C#, los arreglos son estructuras de datos que permiten almacenar y acceder a múltiples elementos del mismo tipo. Los arreglos son una forma conveniente de organizar y manipular conjuntos de datos relacionados.

2.1.1.1. Declaración

```

int[] numeros;
string[] nombres;
double[] precios;

```

2.1.1.2. Inicialización

```
int[] numeros = new int[5];
string[] nombres = new string[] {"Ana", "Juan", "Maria"};
double[] precios = { 2.99, 4.5, 9.99 };
```

2.1.1.3 Acceso a elementos de un arreglo

```
int[] numeros = { 10, 20, 30, 40, 50 };
int primerNumero = numeros[0]; // Accede al primer elemento del arreglo
int tercerNumero = numeros[2]; // Accede al tercer elemento del arreglo
```

The screenshot shows a code editor window with a dark theme. It displays a file named 'Program.cs' with the following content:

```
# Program.cs > Program
1 internal class Program
2 {
3     0 references
4     private static void Main(string[] args)
5     {
6         int[] numeros = { 10, 20, 30, 40, 50 };
7         foreach(int n in numeros){
8             Console.WriteLine(n);
9         }
10    }
11 }
```

A yellow rectangular box highlights the following code block:

```
foreach(int n in numeros){
    Console.WriteLine(n);
```

2.1.1.4. Métodos de arreglos

Método Length:

- Descripción: Devuelve la longitud (número de elementos) del arreglo.
- Uso:

```
int[] numeros = { 10, 20, 30, 40, 50 };
int longitud = numeros.Length; // Devuelve 5
```

Método Sort():

- Descripción: Ordena los elementos del arreglo en orden ascendente.
- Uso:

```
int[] numeros = { 40, 10, 30, 20, 50 };
Array.Sort(numeros);
```

Método Reverse():

- Descripción: Devuelve el índice de la primera aparición de un elemento en el arreglo.
- Uso:

```
string[] nombres = { "Juan", "María", "Ana", "Juan", "Pedro" };
int indice = Array.IndexOf(nombres, "Juan"); // Devuelve 0
```

Método Copy():

- Descripción: Copia los elementos de un arreglo a otro arreglo.
- Uso:

```
int[] arreglo1 = { 1, 2, 3, 4, 5 };
int[] arreglo2 = new int[5];
Array.Copy(arreglo1, arreglo2, 5);
foreach(int n in arreglo2){
    Console.WriteLine(n);
}
```

- **int[] arreglo1 = { 1, 2, 3, 4, 5 };** Se declara e inicializa un arreglo llamado **arreglo1** con los valores 1, 2, 3, 4 y 5.
- **int[] arreglo2 = new int[5];** Se declara un nuevo arreglo llamado **arreglo2** con una longitud de 5 elementos. Los elementos se inicializan con el valor predeterminado de **int**, que es 0.
- **Array.Copy(arreglo1, arreglo2, 5);** Se utiliza el método **Copy** de la clase **Array** para copiar los elementos del arreglo **arreglo1** al arreglo **arreglo2**. La cantidad de elementos a copiar es 5.
- **foreach (int n in arreglo2)** Se utiliza un bucle **foreach** para recorrer cada elemento del arreglo **arreglo2**. En cada iteración, el valor del elemento se asigna a la variable **n**.
- **Console.WriteLine(n);** Se muestra en la consola el valor actual de la variable **n**.

En resumen, el código copia los elementos del arreglo **arreglo1** al arreglo **arreglo2** y luego muestra en la consola cada elemento del arreglo **arreglo2**. Esto permite verificar que los elementos se han copiado correctamente.

2.1.2. ArrayList

En C#, **ArrayList** es una clase que proporciona una colección dinámica de objetos. A diferencia de los arreglos convencionales, **ArrayList** puede crecer o disminuir automáticamente según sea necesario para acomodar elementos adicionales.

2.1.2.1. Métodos

2.1.2.1.1. Add

Agrega un objeto al final de **ArrayList**.

Clase

```
public virtual int Add (object? value);
```

Parámetros

value **Object**

Objeto **Object** que se va a agregar al final de la colección **ArrayList**. El valor puede ser **null**.

Fuente:

[https://learn.microsoft.com/es-](https://learn.microsoft.com/es-es/dotnet/api/system.collections.arraylist.add?view=net-7.0)

[es/dotnet/api/system.collections.arraylist.add?view=net-7.0](https://learn.microsoft.com/es-es/dotnet/api/system.collections.arraylist.add?view=net-7.0)

Ejemplo

```
1  using System.Collections;
2  
3  internal class Program
4  {
5      0 references
6      private static void Main(string[] args)
7      {
8          ArrayList myAL = new ArrayList();
9          myAL.Add( "The" );
10         myAL.Add( "quick" );
11         myAL.Add( "brown" );
12         myAL.Add( "fox" );
13         foreach(Object n in myAL){
14             Console.WriteLine(n.ToString());
15         }
16     }
17 }
```

- **ArrayList myAL = new ArrayList();** Se crea una nueva instancia de la clase **ArrayList** llamada **myAL**. **ArrayList** es una clase en .NET que permite almacenar y manipular una colección dinámica de objetos.
- **myAL.Add("The");** Se agrega la cadena "**The**" al final de la colección **myAL** utilizando el método **Add** de **ArrayList**. Este método permite añadir elementos a la colección.
- **myAL.Add("quick");** Se agrega la cadena "**quick**" al final de la colección **myAL**.
- **myAL.Add("brown");** Se agrega la cadena "**brown**" al final de la colección **myAL**.
- **myAL.Add("fox");** Se agrega la cadena "**fox**" al final de la colección **myAL**.
- **foreach (object n in myAL)** Se utiliza un bucle **foreach** para recorrer cada elemento de la colección **myAL**. En cada iteración, el elemento actual se asigna a la variable **n** de tipo **object**.
- **Console.WriteLine(n.ToString());** Se muestra en la consola el valor actual de la variable **n** convertido a una representación de cadena utilizando el método **ToString()**.

En resumen, el código crea un **ArrayList** llamado **myAL** y agrega algunas cadenas a la colección. Luego, utiliza un bucle **foreach** para recorrer cada elemento de la colección y mostrarlo en la consola. Esto permite imprimir cada cadena agregada al **ArrayList** en líneas separadas.

```

C# Program.cs > 🏃 Program > ⚑ Main(string[] args)
1   using System.Collections;
2
3     0 references
3   internal class Program
4   {
5     0 references
5   private static void Main(string[] args)
6   {
7     ArrayList Carnivoros = new ArrayList();
8     ArrayList Acuaticos = new ArrayList();
9     ArrayList Animales = new ArrayList();
10
11    Carnivoros.Add("Tigre");
12    Carnivoros.Add("León");
13    Carnivoros.Add("Pantera");
14    Carnivoros.Add("Leopardo");
15    Acuaticos.Add("Tiburón");
16    Acuaticos.Add("Delfín");
17    Acuaticos.Add("Ballena");
18    Acuaticos.Add("Pulpo");
19
20    Animales.AddRange(Carnivoros);
21    foreach (Object n in Animales)
22    {
23      Console.WriteLine(n.ToString());
24    }
25    Console.WriteLine("=====");
26
27    Animales.AddRange(Acuaticos);
28    foreach (Object n in Animales)
29    {
30      Console.WriteLine(n.ToString());
31    }
32  }
33
34 }

```

Este código muestra un ejemplo de cómo utilizar la clase **ArrayList** en C# para almacenar y combinar listas de animales carnívoros y acuáticos.

- Se importa el espacio de nombres **System.Collections** que contiene la clase **ArrayList**.
- Se declara la clase **Program** como **internal** (accesible solo dentro del ensamblado) y se define el método **Main** como **private static void**.
- Se crean tres objetos **ArrayList** llamados **Carnivoros**, **Acuáticos** y **Animales**.
- Se agregan elementos a los **ArrayList Carnívoros** y **Acuáticos** utilizando el método **Add()**:
- **Carnivoros** se llena con nombres de animales carnívoros, como "Tigre", "León", "Pantera" y "Leopardo".
- **Acuáticos** se llena con nombres de animales acuáticos, como "Tiburón", "Delfín", "Ballena" y "Pulpo".
- Se utiliza el método **AddRange()** para combinar los elementos de **Carnivoros** y **Acuáticos** en el **ArrayList Animales**. Esto permite unir los dos **ArrayList** en uno solo.

- Se utiliza un bucle **foreach** para recorrer el **ArrayList Animales** e imprimir cada elemento en la consola. **n.ToString()** se utiliza para convertir cada elemento en una cadena antes de imprimirla.
- Se imprime una línea de separación para distinguir los dos grupos de animales.
- Se utiliza nuevamente un bucle **foreach** para recorrer el **ArrayList Animales** después de agregar los elementos de **Acuáticos**. Esto muestra cómo se combinaron los elementos de **Carnívoros** y **Acuáticos** en un solo **ArrayList**.

En resumen, este código demuestra cómo utilizar **ArrayList** para almacenar y combinar listas de elementos, en este caso, nombres de animales carnívoros y acuáticos. Luego, muestra cómo recorrer e imprimir los elementos de **Animales** en la consola.

2.1.2.1.2 Clear

Quita todos los elementos de **ArrayList**.

Ejemplo

```

1  using System.Collections;
2  0 references
3  internal class Program
4  {
5      0 references
6      private static void Main(string[] args)
7      {
8          ArrayList Carnivoros = new ArrayList();
9          ArrayList Acuaticos = new ArrayList();
10         ArrayList Animales = new ArrayList();
11
12         Carnivoros.Add("Tigre");
13         Carnivoros.Add("Leon");
14         Carnivoros.Add("Pantera");
15         Carnivoros.Add("Leopardo");
16         Acuaticos.Add("Tiburon");
17         Acuaticos.Add("Delfin");
18         Acuaticos.Add("Ballena");
19         Acuaticos.Add("Pulpo");
20
21         Animales.AddRange(Carnivoros);
22         foreach (Object n in Animales)
23         {
24             Console.WriteLine(n.ToString());
25         }
26         Console.WriteLine("Presiona Enter para continuar...");
27         Console.ReadLine();
28
29         Animales.Clear();
30         Console.WriteLine("=====");
31
32         Animales.AddRange(Acuaticos);
33         foreach (Object n in Animales)
34         {
35             Console.WriteLine(n.ToString());
36         }
37     }
38 }
```

Este código muestra un ejemplo de cómo utilizar la clase **ArrayList** en C# para almacenar y manipular diferentes listas de animales carnívoros y acuáticos. A continuación, se detalla cada parte del código:

- Se importa el espacio de nombres **System.Collections**, que contiene la clase **ArrayList**.
- Se declara la clase **Program** como **internal**, lo que significa que solo es accesible dentro del ensamblado.
- Se define el método **Main** como **private static void**, que es el punto de entrada del programa.
- Se crean tres objetos **ArrayList** llamados **Carnivoros**, **Acuaticos** y **Animales**.
- Se utilizan los métodos **Add()** de cada **ArrayList** para agregar elementos:
 - En el **ArrayList Carnivoros**, se agregan los nombres de animales carnívoros como "Tigre", "León", "Pantera" y "Leopardo".
 - En el **ArrayList Acuaticos**, se agregan los nombres de animales acuáticos como "Tiburón", "Delfín", "Ballena" y "Pulpo".
- Se utiliza el método **AddRange()** para combinar los elementos de **Carnivoros** en el **ArrayList Animales**. Esto agrega los elementos de **Carnivoros** al final de **Animales**. Luego, se utiliza un bucle **foreach** para recorrer **Animales** e imprimir cada elemento en la consola.
- Se muestra el mensaje "Presiona Enter para continuar..." y se utiliza **Console.ReadLine()** para pausar la ejecución del programa hasta que el usuario presione la tecla Enter.
- Se llama al método **Clear()** de **Animales** para eliminar todos los elementos de este **ArrayList**.
- Se muestra una línea de separación en la consola para distinguir los dos grupos de animales.
- Se utiliza nuevamente el método **AddRange()** para combinar los elementos de **Acuaticos** en el **ArrayList Animales**. Se utiliza otro bucle **foreach** para recorrer **Animales** después de agregar los elementos de **Acuaticos** e imprimir cada elemento en la consola.

2.1.2.1.3 Contains

Determina si un elemento se encuentra en **ArrayList**.

Devoluciones

Boolean

`true` si `item` se encuentra en la matriz `ArrayList`; en caso contrario, `false`.

Ejemplo:

```

1  using System.Collections;
2  internal class Program
3  {
4      private static void Main(string[] args)
5      {
6          string ? palabra;
7          ArrayList Materias = new ArrayList();
8
9          Materias.Add("Calculo");
10         Materias.Add("Español");
11         Materias.Add("Dibujo tecnico");
12         Materias.Add("Ingles");
13
14         Console.WriteLine("Escriba una materia buscar: ");
15         palabra = Console.ReadLine();
16         Console.WriteLine(Materias.Contains(palabra) ? "La materia esta disponible" : "La materia no se encontro");
17     }
18 }

```



```
PS D:\projectsNetCore\Practica01> dotnet run
Escriba una materia buscar: Ingles
La materia esta disponible
○ PS D:\projectsNetCore\Practica01> █
```

Este código muestra un ejemplo de cómo utilizar la clase **ArrayList** en C# para almacenar una lista de materias y buscar una materia específica dentro de ella. A continuación, se explica cada parte del código:

- Se importa el espacio de nombres **System.Collections**, que contiene la clase **ArrayList**.
- Se declara la clase **Program** como **internal**, lo que significa que solo es accesible dentro del ensamblado.
- Se define el método **Main** como **private static void**, que es el punto de entrada del programa.
- Se declara una variable **string** llamada **palabra** con el operador de anulabilidad (?). Esto permite que la variable **palabra** acepte un valor nulo.
- Se crea un objeto **ArrayList** llamado **Materias**.
- Se utilizan los métodos **Add()** de **Materias** para agregar nombres de materias a la lista.
- Se muestra el mensaje "Escriba una materia buscar: " en la consola y se utiliza **Console.ReadLine()** para leer la entrada del usuario y asignarla a la variable **palabra**.
- Se utiliza el método **Contains()** de **Materias** para verificar si la materia ingresada por el usuario se encuentra en la lista. Se utiliza el operador ternario (? :) para imprimir un mensaje correspondiente según el resultado de la búsqueda. Si la materia está en la lista, se imprime "La materia está disponible"; de lo contrario, se imprime "La materia no se encontró".

Resumiendo, este código demuestra cómo utilizar **ArrayList** en C# para almacenar una lista de materias y buscar una materia específica ingresada por el usuario. El método **Contains()** se utiliza para realizar la búsqueda y se imprime un mensaje indicando si la materia se encontró en la lista o no.

2.1.2.1.4 IndexOf

Busca el objeto Object especificado y devuelve el índice de base cero de la primera aparición en toda la colección ArrayList.

Devoluciones

Int32

Índice de base cero de la primera aparición de `value` en la totalidad de `ArrayList`, si se encuentra; en caso contrario, -1.

```

1  using System.Collections;
2  0 references
3  internal class Program
4  {
5      0 references
6      private static void Main(string[] args)
7      {
8          string ? palabra;
9          ArrayList Materias = new ArrayList();
10
11         Materias.Add("Calculo");
12         Materias.Add("Español");
13         Materias.Add("Dibujo tecnico");
14         Materias.Add("Ingles");
15
16         Console.WriteLine("Escriba una materia buscar: ");
17         palabra = Console.ReadLine();
18         Console.WriteLine(Materias.IndexOf(palabra));
19     }

```



- **string palabra;** Se declara una variable **palabra** de tipo **string**, que se utilizará para almacenar la materia que se desea buscar.
- **ArrayList Materias = new ArrayList();** Se crea una instancia de **ArrayList** llamada **Materias**. Esta lista será utilizada para almacenar una lista de materias.
- **Materias.Add("Calculo");** Se agrega la cadena "**Calculo**" a la lista **Materias** utilizando el método **Add** de **ArrayList**. Esto añade la materia "Calculo" a la lista.
- **Materias.Add("Español");** Se agrega la cadena "**Español**" a la lista **Materias**. Esto añade la materia "Español" a la lista.
- **Materias.Add("Dibujo tecnico");** Se agrega la cadena "**Dibujo tecnico**" a la lista **Materias**. Esto añade la materia "Dibujo técnico" a la lista.
- **Materias.Add("Ingles");** Se agrega la cadena "**Ingles**" a la lista **Materias**. Esto añade la materia "Inglés" a la lista.
- **Console.WriteLine("Escriba una materia buscar: ");** Se muestra un mensaje en la consola solicitando al usuario que escriba la materia que desea buscar.
- **palabra = Console.ReadLine();** Se lee la entrada del usuario desde la consola y se asigna a la variable **palabra**.
- **Console.WriteLine(Materias.IndexOf(palabra));** Se utiliza el método **IndexOf** de **ArrayList** para buscar la posición de la primera aparición de la materia especificada en **palabra** dentro de la lista **Materias**. El resultado se muestra en la consola.

En resumen, el código crea una lista de materias utilizando la clase **ArrayList** llamada **Materias**, y luego permite al usuario ingresar una materia para buscar en esa lista. La posición de la materia encontrada se muestra en la consola utilizando el método **IndexOf**.

```
PS D:\projectsNetCore\Practica01> dotnet run
Escriba una materia buscar: Ingles
3
○ PS D:\projectsNetCore\Practica01> [
```

2.1.2.1.5. Insert e InsertRange

Insert Inserta un elemento en ArrayList en el índice especificado e InsertRange Inserta los elementos de una colección en ArrayList en el índice especificado.

Ejemplo:

```
1  using System.Collections;
2  0 references
3  internal class Program
4  [
5      0 references
6      private static void Main(string[] args)
7      {
8          string ? palabra;
9          int idx;
10         ArrayList Materias = new ArrayList();
11
12         Materias.Add("Calculo");
13         Materias.Add("Español");
14         Materias.Add("Dibujo tecnico");
15         Materias.Add("Ingles");
16
17         do{
18             Console.WriteLine("Ingrese la posicion donde desea insertar la Asignatura:");
19             idx = Convert.ToInt32(Console.ReadLine());
20             }while(idx > Materias.Count);
21
22             Console.WriteLine("Escriba una materia a insertar: ");
23             palabra = Console.ReadLine();
24
25             Materias.Insert(idx, palabra);
26
27             foreach(Object materia in Materias){
28                 Console.WriteLine(materia.ToString());
29             }
30 }
```

```
PS D:\projectsNetCore\Practica01> dotnet run
Ingrese la posicion donde desea insertar la Asignatura:3
Escriba una materia a insertar: Quimica
Calculo
Español
Dibujo tecnico
Quimica ←
Ingles
○ PS D:\projectsNetCore\Practica01> [
```

```

1  using System.Collections;
2  0 references
3  internal class Program
4  {
5      0 references
6      private static void Main(string[] args)
7      {
8          int idx;
9          ArrayList Materias = new ArrayList();
10         ArrayList Cursos = new ArrayList();
11
12         Materias.Add("Calculo");
13         Materias.Add("Español");
14         Materias.Add("Dibujo tecnico");
15         Materias.Add("Ingles");
16
17         Cursos.Add("Octavo");
18         Cursos.Add("Noveno");
19
20         do{
21             Console.WriteLine("Ingrese la posicion donde desea insertar la Asignatura:");
22             idx = Convert.ToInt32(Console.ReadLine());
23             }while(idx > Materias.Count);
24
25
26         Materias.InsertRange(idx, Cursos);
27
28         foreach(Object materia in Materias){
29             Console.WriteLine(materia.ToString());
30         }
31     }

```

```

● PS D:\projectsNetCore\Practica01> dotnet run
    Ingrese la posicion donde desea insertar la Asignatura:2
    Calculo
    Español
    Octavo
    Noveno
    Dibujo tecnico
    Ingles
○ PS D:\projectsNetCore\Practica01>

```

Este código muestra un ejemplo de cómo utilizar la clase **ArrayList** en C# para insertar elementos de un **ArrayList** en una posición específica de otro **ArrayList**. A continuación, se explica cada parte del código:

- Se importa el espacio de nombres **System.Collections**, que contiene la clase **ArrayList**.
- Se declara la clase **Program** como **internal**, lo que significa que solo es accesible dentro del ensamblado.
- Se define el método **Main** como **private static void**, que es el punto de entrada del programa.
- Se declara una variable **int** llamada **idx** para almacenar la posición donde se desea insertar la asignatura en el **ArrayList Materias**.
- Se crean dos objetos **ArrayList**: **Materias** y **Cursos**.
- Se utilizan los métodos **Add()** de **Materias** y **Cursos** para agregar nombres de asignaturas y cursos, respectivamente, a los **ArrayList**.
- Se utiliza un bucle **do-while** para solicitar al usuario que ingrese la posición donde desea insertar la asignatura. Se utiliza **Console.ReadLine()** para leer la entrada del usuario y

`Convert.ToInt32()` para convertirlo a un valor entero. El bucle se repetirá hasta que la posición ingresada (`idx`) sea menor o igual a la cantidad de elementos en `Materias` (verificado con `Materias.Count`).

- Se utiliza el método `InsertRange()` de `Materias` para insertar los elementos de `Cursos` en la posición especificada por `idx` dentro de `Materias`. Esto agrega los elementos de `Cursos` en esa posición, desplazando los elementos existentes hacia la derecha.
- Se utiliza un bucle `foreach` para recorrer `Materias` y se imprime cada elemento en la consola utilizando `Console.WriteLine()`.

En resumen, este código muestra cómo utilizar `ArrayList` en C# para insertar elementos de un `ArrayList` en una posición específica de otro `ArrayList`. El programa solicita al usuario la posición deseada y realiza la inserción utilizando el método `InsertRange()`. Luego, muestra todos los elementos de `Materias`, incluyendo los elementos recién insertados.

2.1.2.1.6. LastIndexOf

El método `LastIndexOf` en C# se utiliza para buscar la última aparición de un elemento en un `ArrayList`, un arreglo u otra colección de elementos. Devuelve el índice de la última ocurrencia del elemento buscado o -1 si no se encuentra.

Ejemplo:

```
1  using System.Collections;
2  0 references
3  internal class Program
4  {
5      0 references
6      private static void Main(string[] args)
7      {
8          string ? palabra;
9          int idx;
10         ArrayList Materias = new ArrayList();
11
12         Materias.Add("Calculo");
13         Materias.Add("Español");
14         Materias.Add("Dibujo tecnico");
15         Materias.Add("Ingles");
16
17         Console.WriteLine("Ingrese la materia a buscar");
18         palabra = Console.ReadLine();
19
20         idx = Materias.LastIndexOf(palabra);
21
22         Console.WriteLine("La asignatura {0} se encontro en la ultima pos de idx {1} ",palabra,idx);
23     }
24 }
```

```
● PS D:\projectsNetCore\Practica01> dotnet run
    Ingrese la materia a buscar
    Ingles
    La asignatura Ingles se encontro en la ultima pos de idx 3
○ PS D:\projectsNetCore\Practica01>
```

Este código muestra un ejemplo de cómo utilizar el método `LastIndexOf` en un `ArrayList` en C# para buscar la última aparición de una materia específica. A continuación, se explica cada parte del código:

- Se importa el espacio de nombres `System.Collections`, que contiene la clase `ArrayList`.
- Se declara la clase `Program` como `internal`, lo que significa que solo es accesible dentro del ensamblado.

- Se define el método **Main** como **private static void**, que es el punto de entrada del programa.
- Se declara una variable **string** llamada **palabra** con el operador de anulabilidad (?). Esto permite que la variable **palabra** acepte un valor nulo.
- Se declara una variable **int** llamada **idx** para almacenar el índice de la última aparición de la materia en el **ArrayList**.
- Se crea un **ArrayList** llamado **Materias**.
- Se utilizan los métodos **Add()** de **Materias** para agregar nombres de asignaturas al **ArrayList**.
- Se muestra el mensaje "Ingrese la materia a buscar" en la consola y se utiliza **Console.ReadLine()** para leer la entrada del usuario y asignarla a la variable **palabra**.
- Se utiliza el método **LastIndexOf()** de **Materias** para buscar la última aparición de la materia ingresada por el usuario. El resultado se almacena en la variable **idx**.
- Se utiliza **Console.WriteLine()** para mostrar un mensaje que indica la materia buscada y el índice de su última aparición en el **ArrayList**.

En resumen, este código muestra cómo utilizar el método **LastIndexOf** en un **ArrayList** en C# para buscar la última aparición de una materia específica ingresada por el usuario. El programa solicita al usuario que ingrese la materia a buscar y luego muestra un mensaje que indica la materia buscada y el índice de su última aparición en el **ArrayList**.

2.1.2.1.7. Remove y RemoveAt

En C#, puedes eliminar elementos de un **ArrayList** utilizando el método **Remove()** o el método **RemoveAt()**. A continuación, se explica cómo utilizar cada uno de estos métodos:

Método Remove():

- Descripción: Elimina la primera aparición de un elemento específico en el **ArrayList**.
- Sintaxis:

```
miArrayList.Remove(elemento);
```

Ejemplo:

```
ArrayList miArrayList = new ArrayList();
miArrayList.Add("Manzana");
miArrayList.Add("Banana");
miArrayList.Add("Naranja");

miArrayList.Remove("Banana"); // Elimina la primera aparición de "Banana"
```

Método RemoveAt():

- Descripción: Elimina el elemento en el índice especificado del **ArrayList**.
- Sintaxis:

```
miArrayList.RemoveAt(indice);
```

```

ArrayList miArrayList = new ArrayList();
miArrayList.Add("Manzana");
miArrayList.Add("Banana");
miArrayList.Add("Naranja");

miArrayList.RemoveAt(1); // Elimina el elemento en la posición

```

2.1.2.1.8. ToArray

En C#, el método `ToArray()` se utiliza para convertir una colección o secuencia en un arreglo. Aquí tienes un ejemplo de cómo usar el método `ToArray()` en C#:

Ejemplo:

```

1  using System.Collections;
2  0 references
3  internal class Program
4  {
5      0 references
6      private static void Main(string[] args)
7      {
8
9          ArrayList Materias = new ArrayList();
10
11         Materias.Add("Calculo");
12         Materias.Add("Español");
13         Materias.Add("Dibujo tecnico");
14         Materias.Add("Ingles");
15
16         String[] myArray = (String[]) Materias.ToArray(typeof(string));
17
18         foreach(String myMateria in myArray){
19             Console.WriteLine(myMateria);
20         }
21     }

```

- PS D:\projectsNetCore\Practica01> dotnet run ←

```

Calculo
Español
Dibujo tecnico
Ingles

```

- PS D:\projectsNetCore\Practica01> []

El código que proporcionaste es un ejemplo en C# que utiliza la clase `ArrayList` y el método `ToArray()` para convertir los elementos en un arreglo de tipo `string`. A continuación, se muestra una explicación paso a paso del código:

- Se importa el espacio de nombres `System.Collections`, que contiene la clase `ArrayList` utilizada en el código.
- Se define una clase `Program` como `internal`, lo que significa que solo es accesible dentro del mismo ensamblado.

- El método principal **Main** se declara como **private** y se pasa un arreglo de cadenas **args** como parámetro. Es el punto de entrada para la ejecución del programa.
- Se crea una instancia de la clase **ArrayList** llamada **Materias**. **ArrayList** es una colección dinámica en C# que puede contener elementos de diferentes tipos.
- Se agregan elementos a la colección **Materias** utilizando el método **Add()**. En este caso, se agregan cuatro cadenas que representan diferentes materias.
- Se utiliza el método **ToArray()** en **Materias** para convertir los elementos en un arreglo. Se especifica el tipo de elemento que se espera en el arreglo utilizando **typeof(string)**.
- El resultado de **ToArray()** se almacena en una variable **myArray** de tipo **string[]**, que es un arreglo de cadenas.
- Se utiliza un bucle **foreach** para iterar sobre cada elemento del arreglo **myArray**.
- Dentro del bucle, se imprime cada materia en la consola utilizando **Console.WriteLine()**.

En resumen, el código crea una colección de materias utilizando **ArrayList**, la convierte en un arreglo de cadenas utilizando el método **ToArray()**, y luego imprime cada materia en la consola utilizando un bucle **foreach**.

2.1.3. Clase List

En C#, la clase **List<T>** se utiliza para crear y manipular listas genéricas. Una lista genérica es una colección que puede contener elementos de un tipo específico (**T**).

Declaración:

```
List<T> nombreLista = new List<T>();
```

- **List<T>**: Especifica el tipo de elementos que se almacenarán en la lista. **T** representa el tipo de elemento deseado, como **int**, **string**, **Person**, etc.
- **nombreLista**: Es el nombre que le das a tu lista. Puedes elegir cualquier nombre válido de acuerdo con las convenciones de nomenclatura de C#.
- **new List<T>()**: Crea una nueva instancia de la clase **List<T>** utilizando el operador **new**. Asegúrate de incluir los paréntesis **()** después del **new List<T>()** para invocar el constructor de la lista.

2.1.3.1. Propiedades

2.1.3.1.1. Count

Obtiene el número de elementos incluidos en **List<T>**.

```
C#
public int Count { get; }
```

```
1  using System.Collections;
2  0 references
3  internal class Program
4  {
5      0 references
6      private static void Main(string[] args)
7      {
8
9          List<String> Materias = new List<String>();
10
11         Materias.Add("Calculo");
12         Materias.Add("Español");
13         Materias.Add("Dibujo tecnico");
14         Materias.Add("Ingles");
15
16         Console.WriteLine("Total de elementos de la lista {0}", Materias.Count());
17     }
18 }
```

```
PS D:\projectsNetCore\Practica01> dotnet run
Total de elementos de la lista 4
PS D:\projectsNetCore\Practica01> 
```

- La línea **using System.Collections;** indica que se está utilizando el espacio de nombres **System.Collections**, que contiene la clase **List<T>** y otras colecciones.
- La clase **Program** se declara como **internal**, lo que significa que solo es accesible dentro del mismo ensamblado.
- El método principal **Main** se declara como **private** y se pasa un arreglo de cadenas **args** como parámetro. Este método es el punto de entrada para la ejecución del programa.
- Se crea una instancia de la clase **List<string>** llamada **Materias**. Esta lista se utiliza para almacenar objetos de tipo **string**.
- Se utilizan los métodos **Add()** para agregar elementos a la lista **Materias**. En este caso, se agregan cuatro cadenas que representan diferentes materias.
- Se utiliza el método **Count()** para obtener el número total de elementos en la lista **Materias**. La función **Console.WriteLine()** se utiliza para imprimir un mensaje que muestra el resultado del conteo.
- En la cadena de formato del mensaje, **{0}** se reemplaza con el valor devuelto por **Materias.Count()**.

En resumen, este código crea una lista llamada **Materias** utilizando **List<string>**. Luego, se agregan elementos a la lista y se muestra el número total de elementos utilizando el método **Count()**. El resultado se imprime en la consola mediante **Console.WriteLine()**.

2.1.3.1.2. Item

Obtiene o establece el elemento en el índice especificado.

```

1  using System.Collections;
2  0 references
3  internal class Program
4  {
5      0 references
6      private static void Main(string[] args)
7      {
8
9          List<String> Materias = new List<String>();
10
11         Materias.Add("Calculo");
12         Materias.Add("Español");
13         Materias.Add("Dibujo tecnico");
14         Materias.Add("Ingles");
15
16         Console.WriteLine("Total de elementos de la lista {0}", Materias.Count());
17
18         Console.WriteLine("Capacidad total: {0}", Materias.Capacity);
19
20     }
21 }

```

```

● PS D:\projectsNetCore\Practica01> dotnet run
    Total de elementos de la lista 4
    Capacidad total: 4
    Materia[3]: Ingles ←
○ PS D:\projectsNetCore\Practica01> █

```

- **Console.WriteLine("Total de elementos de la lista {0}", Materias.Count());** Esta línea imprime el número total de elementos en la lista utilizando el método **Count()** de la clase **List<T>**. **{0}** en la cadena de formato se reemplaza con el valor devuelto por **Materias.Count()**. El resultado se imprime en la consola.
- **Console.WriteLine("Capacidad total: {0}", Materias.Capacity);** Esta línea imprime la capacidad total de la lista utilizando la propiedad **Capacity** de la clase **List<T>**. La capacidad es el número de elementos que la lista puede contener antes de tener que redimensionarse automáticamente. El valor se imprime en la consola.
- **Console.WriteLine("Materia[3]: {0}", Materias[3]);** Esta línea accede al cuarto elemento de la lista utilizando el operador de índice **[3]**. El índice comienza desde cero, por lo que **Materias[3]** obtiene el cuarto elemento de la lista. El valor se imprime en la consola.

2.1.3.2. Métodos

2.1.3.2.1. Add

Agrega un objeto al final de **List<T>**.

Ejemplo

```

1  using System.Collections;
2  internal class Program
3  {
4      private static void Main(string[] args)
5      {
6
7          List<String> Materias = new List<String>();
8
9          Materias.Add("Calculo");
10         Materias.Add("Español");
11         Materias.Add("Dibujo tecnico");
12         Materias.Add("Ingles");
13
14         foreach (String str in Materias){
15             Console.WriteLine(str);
16         }
17
18     }
19 }
20

```

```

● PS D:\projectsNetCore\Practica01> dotnet run
    Calculo
    Español
    Dibujo tecnico
    Ingles
○ PS D:\projectsNetCore\Practica01> []

```

- **List<String> Materias = new List<String>();** Esta línea crea una instancia de la clase **List<string>** llamada **Materias**, que se utilizará para almacenar objetos de tipo **string**.
- **Materias.Add("Calculo");** Aquí se utiliza el método **Add()** de la clase **List<T>** para agregar el elemento "Calculo" a la lista **Materias**. Esto se repite para los otros elementos, como "Español", "Dibujo técnico" e "Inglés".
- **foreach (String str in Materias)** Este es un bucle **foreach** que se utiliza para recorrer cada elemento en la lista **Materias**. La variable **str** toma el valor de cada elemento en la lista en cada iteración del bucle.
- **Console.WriteLine(str);** Dentro del bucle **foreach**, se imprime el valor de la variable **str** en la consola utilizando **Console.WriteLine()**. Esto mostrará cada elemento de la lista en líneas separadas.

2.1.3.2.2. AddRange

Agrega los elementos de la colección especificada al final de List<T>.

```

C#
public void AddRange (System.Collections.Generic.IEnumerable<T>
collection);

```

Ejemplo:

```
1  using System.Collections;
2  0 references
3  internal class Program
4  {
5      0 references
6
7      private static void Main(string[] args)
8      {
9
10         List<String> Dinosaurios = new List<String>();
11         string[] Dinos = { "Brachiosaurus",
12                            "Amargasaurus",
13                            "Mamenchisaurus" };
14         Dinosaurios.AddRange(Dinos); ←
15
16
17     }
18 }
```

```
● PS D:\projectsNetCore\Practica01> dotnet run
    Brachiosaurus
    Amargasaurus
    Mamenchisaurus
○ PS D:\projectsNetCore\Practica01> □
```

2.1.3.2.3. Clear

Quita todos los elementos de List<T>.

```
C#
public void Clear();
```

Ejemplo:

```

1  using System.Collections;
2  internal class Program
3  {
4      0 references
5      private static void Main(string[] args)
6      {
7          List<String> Dinosaurios = new List<String>();
8          string[] Dinos = { "Brachiosaurus",
9                             "Amargasaurus",
10                            "Mamenchisaurus" };
11          Dinosaurios.AddRange(Dinos);
12
13          foreach (String str in Dinosaurios){
14              Console.WriteLine(str);
15          }
16          Dinosaurios.Clear();
17          Console.WriteLine("Total de elementos {0} :",Dinosaurios.Count);
18
19      }
20  }

```

```

PS D:\projectsNetCore\Practica01> dotnet run
Brachiosaurus
Amargasaurus
Mamenchisaurus
Total de elementos 0 :
PS D:\projectsNetCore\Practica01> █

```

2.1.3.2.4. Exists

Determina si List<T> contiene elementos que cumplen las condiciones definidas por el predicado especificado.

Ejemplo:

```

1  using System.Collections;
2  internal class Program
3  {
4      0 references
5      private static void Main(string[] args)
6      {
7          String ? palabra;
8          List<String> Dinosaurios = new List<String>();
9          string[] Dinos = { "Brachiosaurus",
10                         "Amargasaurus",
11                         "Mamenchisaurus" };
12          Dinosaurios.AddRange(Dinos);
13          Console.WriteLine("Ingrese el nombre del Dinosaurio a buscar: ");
14          palabra = Console.ReadLine();
15          Console.WriteLine(Dinosaurios.Exists(item => item.Equals(palabra)) ? "Se encontro el
16                      dinosaurio" : "No se encontro el dinosaurio");
}

```

- PS D:\projectsNetCore\Practica01> dotnet run
Ingrese el nombre del Dinosaurio a buscar:
Amargasaurus
Se encontro el dinosaurio
- PS D:\projectsNetCore\Practica01> █

- Se define una clase **Program** que es accesible solo dentro del mismo ensamblado.

- El método principal **Main** se declara como **private** y se pasa un arreglo de cadenas **args** como parámetro. Este método es el punto de entrada para la ejecución del programa.
- Se declara una variable **palabra** de tipo **string?**. El tipo **string?** indica que la variable **palabra** puede ser nula (**null**) o contener una cadena.
- Se crea una instancia de la clase **List<string>** llamada **Dinosaurios** para almacenar los nombres de los dinosaurios.
- Se declara un arreglo de cadenas llamado **Dinos** que contiene algunos nombres de dinosaurios.
- Se utiliza el método **AddRange()** para agregar todos los elementos del arreglo **Dinos** a la lista **Dinosaurios**.
- Se muestra un mensaje en la consola solicitando al usuario que ingrese el nombre de un dinosaurio a buscar.
- Se utiliza **Console.ReadLine()** para leer la entrada del usuario y asignarla a la variable **palabra**.
- Se utiliza el método **Exists()** en la lista **Dinosaurios** con una expresión lambda **item => item.Equals(palabra)** para verificar si algún elemento de la lista es igual a **palabra**.
- Finalmente, se muestra un mensaje en la consola indicando si el dinosaurio se encontró en la lista o no utilizando un operador ternario **? :**.

2.1.3.2.5. Find

Busca un elemento que coincida con las condiciones definidas por el predicado especificado y devuelve la primera aparición en toda la matriz **List<T>**.

C#

```
public T? Find (Predicate<T> match);
```

Ejemplo:

```
1 internal class Program
2 {
3     0 references
4     private static void Main(string[] args)
5     {
6         String ?palabra=null;
7         List<String> Dinosaurios = new List<String>();
8         string[] Dinos = { "Brachiosaurus",
9                            "Amargasaurus",
10                           "Mamenchisaurus" };
11         Dinosaurios.AddRange(Dinos);
12         Console.WriteLine("Ingrese el nombre del Dinosaurio a buscar: ");
13         palabra = Console.ReadLine();
14         string ? nombre = Dinosaurios.Find(n => n.StartsWith(palabra ?? String.Empty));
15         Console.WriteLine(nombre != null ? "Se encontro el dinosaurio" : "No se encontro el
16                           dinosaurio");
17     }
}
```

- PS D:\projectsNetCore\Practica01> **dotnet run**
Ingrese el nombre del Dinosaurio a buscar:

A

Se encontro el dinosaurio

- PS D:\projectsNetCore\Practica01>

- Declara una variable llamada **nombre** de tipo **string?**. El tipo **string?** indica que la variable **nombre** puede ser nula (**null**) o contener una cadena.
- Utiliza el método **Find()** en la lista **Dinosaurios** para buscar un elemento que cumpla con la condición especificada.
- La condición de búsqueda se define mediante una expresión lambda **n => n.StartsWith(palabra)**. En esta expresión, se verifica si cada elemento **n** de la lista comienza con la cadena **palabra**.
- El resultado de **Find()** se asigna a la variable **nombre**. Si se encuentra un elemento que cumple la condición, **nombre** contendrá dicho elemento; de lo contrario, **nombre** será **null**.

En resumen, este código busca en la lista **Dinosaurios** un elemento que comience con la cadena **palabra** y asigna el resultado a la variable **nombre**, que puede ser nula. La variable **nombre** contendrá el primer elemento encontrado que cumpla con la condición o será **null** si no se encuentra ningún elemento.

Es importante tener en cuenta que al utilizar el tipo **string?**, debes considerar manejar los casos donde **nombre** sea **null** en tu código, ya que ahora puede contener un valor nulo.

2.1.3.2.6. FindAll

Recupera todos los elementos que coinciden con las condiciones definidas por el predicado especificado.

```
public System.Collections.Generic.List<T> FindAll (Predicate<T> match);
```

Ejemplo:

```
1 internal class Program
2 {
3     0 references
4     private static void Main(string[] args)
5     {
6         String ?palabra=null;
7         List<String> Dinosaurios = new List<String>();
8         List<String> Encontrados = new List<String>();
9         string[] Dinos = { "Brachiosaurus",
10                         "Amargasaurus",
11                         "Mamenchisaurus" };
12         Dinosaurios.AddRange(Dinos);
13         Console.WriteLine("Escriba la letra que desea buscar en los dinosaurios: ");
14         palabra = Console.ReadLine();
15         Encontrados = Dinosaurios.FindAll(n => n.Contains(palabra ?? String.Empty));
16         foreach (String n in Encontrados){
17             Console.WriteLine(n);
18         }
19     }
```

```

PS D:\projectsNetCore\Practica01> dotnet run
Escriba la letra que desea buscar en los dinosaurios:
B ←
Brachiosaurus
PS D:\projectsNetCore\Practica01> dotnet run
Escriba la letra que desea buscar en los dinosaurios:
i ←
Brachiosaurus
Mamenchisaurus
PS D:\projectsNetCore\Practica01>

```

- Define una clase **Program** que es accesible solo dentro del mismo ensamblado.
- El método principal **Main** se declara como **private** y se pasa un arreglo de cadenas **args** como parámetro. Este método es el punto de entrada para la ejecución del programa.
- Declara una variable **palabra** de tipo **string?** (nullable). Esta variable se utilizará para almacenar la letra que el usuario desea buscar en los dinosaurios.
- Crea dos listas de cadenas: **Dinosaurios** y **Encontrados**. **Dinosaurios** contendrá los nombres de los dinosaurios y **Encontrados** almacenará los dinosaurios encontrados que cumplan con la condición de búsqueda.
- Se declara un arreglo de cadenas llamado **Dinos** que contiene algunos nombres de dinosaurios.
- Utiliza el método **AddRange()** para agregar todos los elementos del arreglo **Dinos** a la lista **Dinosaurios**.
- Muestra un mensaje en la consola solicitando al usuario que escriba la letra que desea buscar en los dinosaurios.
- Utiliza **Console.ReadLine()** para leer la entrada del usuario y asignarla a la variable **palabra**.
- Utiliza el método **FindAll()** en la lista **Dinosaurios** con una expresión lambda **n => n.Contains(palabra ?? String.Empty)**. Esta expresión lambda verifica si cada elemento **n** de la lista contiene la letra ingresada por el usuario. El operador **??** se utiliza para proporcionar una cadena vacía como valor predeterminado en caso de que **palabra** sea **null**.
- El resultado de **FindAll()** se asigna a la lista **Encontrados**, que contendrá todos los nombres de dinosaurios que cumplan con la condición de búsqueda.
- Utiliza un bucle **foreach** para recorrer la lista **Encontrados** y muestra cada nombre de dinosaurio encontrado en la consola.

En resumen, este código permite al usuario ingresar una letra y luego busca y muestra los nombres de los dinosaurios que contienen esa letra en la lista **Dinosaurios**. Los nombres encontrados se almacenan en la lista **Encontrados** y se imprimen en la consola utilizando un bucle **foreach**.

2.1.3.2.7. **FindLast**

Busca un elemento que coincida con las condiciones definidas por el predicado especificado y devuelve la última aparición en toda la matriz **List<T>**.

C#

```

public T? FindLast (Predicate<T> match);

```

Ejemplo:

```
1 internal class Program
2 {
3     0 references
4     private static void Main(string[] args)
5     {
6         String ?palabra=null;
7         List<String> Dinosaurios = new List<String>();
8         List<String> Encontrados = new List<String>();
9         string[] Dinos = { "Brachiosaurus",
10                           "Amargasaurus",
11                           "Mamenchisaurus" };
12         Dinosaurios.AddRange(Dinos);
13         Console.WriteLine("Escriba la letra que desea buscar en los dinosaurios: ");
14         palabra = Console.ReadLine();
15         string ? ultimoEncontrado = Dinosaurios.FindLast(n => n.Contains(palabra ?? String.Empty));
16         Console.WriteLine("El ultimo dinosaurio que contiene la esta es : {0}", ultimoEncontrado);
17     }
}
```

```
Escriba la letra que desea buscar en los dinosaurios:
u
El ultimo dinosaurio que contiene la esta es : Mamenchisaurus
PS D:\projectsNetCore\Practica01> dotnet run
Escriba la letra que desea buscar en los dinosaurios:
i
El ultimo dinosaurio que contiene la esta es : Mamenchisaurus
PS D:\projectsNetCore\Practica01> []
```

- Define una clase **Program** que es accesible solo dentro del mismo ensamblado.
- En el método principal **Main**, se crea una variable **palabra** de tipo **string?** (nullable) y se inicializa con el valor **null**. Esta variable se utilizará para almacenar la letra que el usuario desea buscar en los nombres de dinosaurios.
- Se crea una lista de cadenas llamada **Dinosaurios** para almacenar los nombres de dinosaurios.
- Se declara una lista de cadenas llamada **Encontrados** que se utilizará para almacenar los nombres de dinosaurios que contienen la letra buscada.
- Se crea un arreglo de cadenas llamado **Dinos** que contiene algunos nombres de dinosaurios.
- Se utiliza el método **AddRange()** para agregar todos los elementos del arreglo **Dinos** a la lista **Dinosaurios**.
- Se muestra un mensaje en la consola utilizando **Console.WriteLine()** para solicitar al usuario que escriba la letra que desea buscar en los nombres de dinosaurios.
- Se utiliza **Console.ReadLine()** para leer la entrada del usuario y asignarla a la variable **palabra**.
- Se utiliza el método **FindLast()** en la lista **Dinosaurios** con una expresión lambda **n => n.Contains(palabra ?? String.Empty)**. Esta expresión lambda verifica si cada nombre de dinosaurio **n** de la lista contiene la letra ingresada por el usuario. El operador **??** se utiliza para proporcionar una cadena vacía como valor predeterminado en caso de que **palabra** sea **null**.
- El resultado de **FindLast()** se asigna a la variable **ultimoEncontrado**, que contendrá el último nombre de dinosaurio que contiene la letra buscada.
- Se imprime un mensaje en la consola utilizando **Console.WriteLine()**. El mensaje muestra el valor de **ultimoEncontrado** utilizando un marcador de posición **{0}**.

En resumen, este código permite al usuario ingresar una letra y luego busca en la lista de nombres de dinosaurios aquellos que contengan esa letra. Luego, muestra el último nombre de dinosaurio encontrado que cumpla con la condición de búsqueda.

2.1.4. Clase Dictionary

En C#, un diccionario es una estructura de datos que almacena pares de clave-valor. Cada elemento en un diccionario consiste en una clave única y su correspondiente valor. La clave se utiliza para acceder y recuperar el valor asociado.

Los diccionarios en C# están implementados por la clase **Dictionary<TKey, TValue>**, que se encuentra en el espacio de nombres **System.Collections.Generic**. Aquí, **TKey** representa el tipo de datos de la clave y **TValue** representa el tipo de datos del valor. Ambos pueden ser cualquier tipo de datos válido en C#.

Algunas características importantes de los diccionarios en C# son:

- Claves únicas: Cada clave en un diccionario debe ser única. Si se intenta agregar una clave que ya existe, se producirá una excepción.
- Acceso eficiente: Los diccionarios proporcionan un acceso rápido a los valores asociados con una clave. Utilizan una estructura de datos interna llamada tabla hash para lograr una búsqueda eficiente.
- Operaciones comunes: Los diccionarios en C# admiten operaciones comunes como agregar un elemento, eliminar un elemento, buscar un valor por clave y verificar si una clave existe en el diccionario.

```
public class Dictionary<TKey,TValue> :  
    System.Collections.Generic.ICollection<System.Collections.Generic.KeyValuePair<TKey,TValue>>,  
    System.Collections.Generic.IDictionary<TKey,TValue>,  
    System.Collections.Generic.IEnumerable<System.Collections.Generic.KeyValuePair<TKey,TValue>>,  
    System.Collections.Generic.IReadOnlyCollection<System.Collections.Generic.KeyValuePair<TKey,TValue>>,  
    System.Collections.Generic.IReadOnlyDictionary<TKey,TValue>, System.Collections.IDictionary,  
    System.Runtime.Serialization.IDeserializationCallback, System.Runtime.Serialization.ISerializable
```

Parámetros de tipo

TKey

Tipo de las claves del diccionario.

TValue

Tipo de los valores del diccionario.

<https://learn.microsoft.com/es-es/dotnet/api/system.collections.generic.dictionary-2?view=net-7.0>

```
1 internal class Program
2 {
3     0 references
4     private static void Main(string[] args)
5     {
6         Dictionary<int, string> diccionario = new Dictionary<int, string>();
7     }
8 }
```

2.1.4.1 Propiedades

2.1.4.1.1. Count

Obtiene el número de pares clave-valor incluidos en Dictionary<TKey,TValue>.

C#

```
public int Count { get; }
```

```
1 internal class Program
2 {
3     0 references
4     private static void Main(string[] args)
5     {
6         Dictionary<int, string> diccionario = new Dictionary<int, string>();
7         Console.WriteLine("El total de elementos del diccionario es : {0}", diccionario.Count);
8     }
9 }
10 }
```

```
● PS D:\projectsNetCore\Practica01> dotnet run
    El total de elementos del diccionario es : 0
○ PS D:\projectsNetCore\Practica01> █
```

Ejemplo:

```

1 internal class Program
2 {
3     0 references
4     private static void Main(string[] args)
5     {
6         Dictionary<string, string> openWith = new Dictionary<string, string>();
7
8         // Add some elements to the dictionary. There are no
9         // duplicate keys, but some of the values are duplicates.
10        openWith.Add("txt", "notepad.exe");
11        openWith.Add("bmp", "paint.exe");
12        openWith.Add("dib", "paint.exe");
13        openWith.Add("rtf", "wordpad.exe");
14
15        // The Add method throws an exception if the new key is
16        // already in the dictionary.
17        try
18        {
19            openWith.Add("txt", "winword.exe");
20        }
21        catch (ArgumentException)
22        {
23            Console.WriteLine("Un elemento con la llave = \"txt\" ya existe.");
24        }
25    }
26 }

```

```

PS D:\projectsNetCore\Practica01> dotnet run
Un elemento con la llave = "txt" ya existe.
PS D:\projectsNetCore\Practica01>

```

- Se define una clase interna llamada **Program**, que es la clase principal del programa.
- El método principal **Main** se declara como **private** y **static**. Es el punto de entrada del programa y se ejecuta cuando se inicia la aplicación.
- Se crea un nuevo diccionario llamado **openWith** utilizando la clase **Dictionary<string, string>**. El diccionario almacena pares de tipo **string**, donde la clave es de tipo **string** y el valor también es de tipo **string**.
- Se agregan elementos al diccionario utilizando el método **Add()**. Se agregan cuatro elementos con diferentes claves y valores. Algunos de los valores son duplicados, pero las claves son únicas.
- A continuación, se intenta agregar un nuevo elemento al diccionario utilizando **Add("txt", "winword.exe")**. Sin embargo, como la clave "txt" ya existe en el diccionario debido al primer **Add("txt", "notepad.exe")**, se produce una excepción de tipo **ArgumentException**.
- Para manejar la excepción, se utiliza un bloque **try-catch**. El código dentro del bloque **try** intenta agregar el elemento, pero al producirse la excepción, el flujo de ejecución salta al bloque **catch**.
- El bloque **catch** captura la excepción de tipo **ArgumentException** y ejecuta el código dentro de él. En este caso, muestra un mensaje en la consola que indica que ya existe un elemento con la clave "txt".

En resumen, el código muestra cómo crear y utilizar un diccionario en C#, cómo agregar elementos al diccionario y cómo manejar una excepción que se produce al intentar agregar un elemento con una clave que ya existe.

2.1.4.1.2. Keys

Obtiene una colección que contiene las claves de `Dictionary<TKey,TValue>`.

```
public System.Collections.Generic.Dictionary<TKey,TValue>.KeyCollection Keys { get; }
```

```
0 references
1 internal class Program
2 {
3     0 references
4     private static void Main(string[] args)
5     {
6         Dictionary<string, string> openWith = new Dictionary<string, string>();
7
8         // Add some elements to the dictionary. There are no
9         // duplicate keys, but some of the values are duplicates.
10        openWith.Add("txt", "notepad.exe");
11        openWith.Add("bmp", "paint.exe");
12        openWith.Add("dib", "paint.exe");
13        openWith.Add("rtf", "wordpad.exe");
14
15        Dictionary<string, string>.KeyCollection keyColl = openWith.Keys;
16
17        foreach (string key in keyColl){
18            Console.WriteLine(key);
19        }
20    }
21 }
```

```
PS D:\projectsNetCore\Practica01> dotnet run
txt
bmp
dib
rtf
PS D:\projectsNetCore\Practica01> █
```

Explicación:

- Se define una clase interna llamada **Program**, que es la clase principal del programa.
- El método principal **Main** se declara como **private** y **static**. Es el punto de entrada del programa y se ejecuta cuando se inicia la aplicación.
- Se crea un nuevo diccionario llamado **openWith** utilizando la clase `Dictionary<string, string>`. El diccionario almacena pares de tipo **string**, donde la clave es de tipo **string** y el valor también es de tipo **string**.
- Se agregan elementos al diccionario utilizando el método **Add()**. Se agregan cuatro elementos con diferentes claves y valores. Algunos de los valores son duplicados, pero las claves son únicas.
- A continuación, se intenta agregar un nuevo elemento al diccionario utilizando `Add("txt", "winword.exe")`. Sin embargo, como la clave "txt" ya existe en el diccionario

debido al primer `Add("txt", "notepad.exe")`, se produce una excepción de tipo **ArgumentException**.

- Para manejar la excepción, se utiliza un bloque **try-catch**. El código dentro del bloque **try** intenta agregar el elemento, pero al producirse la excepción, el flujo de ejecución salta al bloque **catch**.
- El bloque **catch** captura la excepción de tipo **ArgumentException** y ejecuta el código dentro de él. En este caso, muestra un mensaje en la consola que indica que ya existe un elemento con la clave "txt".

En resumen, el código muestra cómo crear y utilizar un diccionario en C#, cómo agregar elementos al diccionario y cómo manejar una excepción que se produce al intentar agregar un elemento con una clave que ya existe.

Ejemplo:

```
internal class Program
{
    private static void Main(string[] args)
    {
        Dictionary<string, string> openWith = new Dictionary<string, string>();

        // Add some elements to the dictionary. There are no
        // duplicate keys, but some of the values are duplicates.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");

        Dictionary<string, string>.KeyCollection keyColl = openWith.Keys;

        foreach (string key in keyColl){
            Console.WriteLine(key);
        }

        foreach(KeyValuePair<string, string> pair in openWith){
            Console.WriteLine("Llave {0} - Valor {1}",pair.Key,pair.Value);
        }
    }
}
```

```
PS D:\projectsNetCore\Practica01> dotnet run
txt
bmp
dib
rtf
Llave txt - Valor notepad.exe
Llave bmp - Valor paint.exe
● Llave dib - Valor paint.exe
● Llave rtf - Valor wordpad.exe
○ PS D:\projectsNetCore\Practica01> [red arrow]
```

Explicación:

- Se define una clase interna llamada **Program**, que es la clase principal del programa.
- El método principal **Main** se declara como **private** y **static**. Es el punto de entrada del programa y se ejecuta cuando se inicia la aplicación.

- Se crea un nuevo diccionario llamado **openWith** utilizando la clase **Dictionary<string, string>**. El diccionario almacena pares de tipo **string**, donde la clave es de tipo **string** y el valor también es de tipo **string**.
- Se agregan elementos al diccionario utilizando el método **Add()**. Se agregan cuatro elementos con diferentes claves y valores. No hay claves duplicadas en este caso, pero algunos de los valores son duplicados.
- Se declara una variable **keyColl** de tipo **Dictionary<string, string>.KeyCollection**. Esta variable almacena la colección de claves del diccionario **openWith**.
- Se utiliza un bucle **foreach** para recorrer cada elemento en la colección de claves **keyColl**. En cada iteración del bucle, se asigna el valor actual de la clave a la variable **key**, y se muestra esa clave en la consola utilizando **Console.WriteLine(key)**.
- A continuación, se utiliza otro bucle **foreach** para recorrer todos los pares clave-valor del diccionario **openWith**. En cada iteración del bucle, se asigna el par clave-valor actual a la variable **pair**, y se muestra la clave y el valor en la consola utilizando **Console.WriteLine("Llave {0} - Valor {1}", pair.Key, pair.Value)**.

En resumen, el código muestra cómo obtener y recorrer tanto la colección de claves de un diccionario como todos los pares clave-valor del diccionario utilizando bucles **foreach**. En el primer bucle, se muestra cada clave en la consola, mientras que en el segundo bucle se muestra tanto la clave como el valor de cada par en la consola.

2.1.4.1.3. Values

Obtiene una colección que contiene los valores de **Dictionary<TKey,TValue>**.

```
public System.Collections.Generic.Dictionary<TKey,TValue>.ValueCollection Values { get; }
```

```

1 internal class Program
2 {
3     0 references
4     private static void Main(string[] args)
5     {
6         Dictionary<string, string> openWith = new Dictionary<string, string>();
7
8         // Add some elements to the dictionary. There are no
9         // duplicate keys, but some of the values are duplicates.
10        openWith.Add("txt", "notepad.exe");
11        openWith.Add("bmp", "paint.exe");
12        openWith.Add("dib", "paint.exe");
13        openWith.Add("rtf", "wordpad.exe");
14
15        Dictionary<string, string>.ValueCollection valueColl = openWith.Values;
16
17        foreach (string valor in valueColl){
18            Console.WriteLine(valor);
19        }
20    }
21 }
```



```
● PS D:\projectsNetCore\Practica01> dotnet run  
notepad.exe  
paint.exe  
paint.exe  
wordpad.exe  
○ PS D:\projectsNetCore\Practica01> █
```

2.1.4.2 Métodos

2.1.4.2.1. Add

Agrega la clave y el valor especificados al diccionario.

```
C#  
  
public void Add (TKey key, TValue value);
```

Ejemplo:

```
1 internal class Program  
2 {  
3     0 references  
4     private static void Main(string[] args)  
5     {  
6         Dictionary<string, string> openWith = new Dictionary<string, string>();  
7  
8         // Add some elements to the dictionary. There are no  
9         // duplicate keys, but some of the values are duplicates.  
10        openWith.Add("txt", "notepad.exe");  
11        openWith.Add("bmp", "paint.exe");  
12        openWith.Add("dib", "paint.exe");  
13        openWith.Add("rtf", "wordpad.exe");  
14  
15        Dictionary<string, string>.ValueCollection valueColl = openWith.Values;  
16  
17        foreach (string valor in valueColl){  
18            Console.WriteLine(valor);  
19        }  
20    }  
21 }
```

2.1.4.2.2. Clear

Quita todas las claves y valores de Dictionary<TKey,TValue>.

```
C#  
  
public void Clear ();
```

```

1 internal class Program
2 {
3     0 references
4     private static void Main(string[] args)
5     {
6         Dictionary<string, string> openWith = new Dictionary<string, string>();
7
8         // Add some elements to the dictionary. There are no
9         // duplicate keys, but some of the values are duplicates.
10        openWith.Add("txt", "notepad.exe");
11        openWith.Add("bmp", "paint.exe");
12        openWith.Add("dib", "paint.exe");
13        openWith.Add("rtf", "wordpad.exe");
14
15        Dictionary<string, string>.ValueCollection valueColl = openWith.Values;
16
17        foreach (string valor in valueColl){
18            Console.WriteLine(valor);
19        }
20
21        openWith.Clear();
22
23        Console.WriteLine("Total de datos en el diccionario: " + openWith.Count);
24    }
25 }

```

```

● PS D:\projectsNetCore\Practica01> dotnet run
notepad.exe
paint.exe
paint.exe
wordpad.exe
● Total de datos en el diccionario: 0
○ PS D:\projectsNetCore\Practica01> █

```

2.1.4.2.3. ContainsKey

Determina si Dictionary<TKey,TValue> contiene la clave especificada.

```

C#
public bool ContainsKey (TKey key);

```

Ejemplo:

```

1  internal class Program
2  {
3      0 references
4      private static void Main(string[] args)
5      {
6          Dictionary<string, string> openWith = new Dictionary<string, string>();
7
8          // Add some elements to the dictionary. There are no
9          // duplicate keys, but some of the values are duplicates.
10         openWith.Add("txt", "notepad.exe");
11         openWith.Add("bmp", "paint.exe");
12         openWith.Add("dib", "paint.exe");
13         openWith.Add("rtf", "wordpad.exe");
14
15         if(openWith.ContainsKey("bmp")){
16             Console.WriteLine("El tipo de formato esta soportado.");
17         }
18         Dictionary<string, string>.ValueCollection valueColl = openWith.Values;
19
20         foreach (string valor in valueColl){
21             Console.WriteLine(valor);
22         }
23     }
24 }
```

```

PS D:\projectsNetCore\Practica01> dotnet run
El tipo de formato esta soportado.
notepad.exe
paint.exe
paint.exe
wordpad.exe
○ PS D:\projectsNetCore\Practica01> []
```

2.1.4.2.4. ContainsValue

Determina si [Dictionary<TKey,TValue>](#) contiene un valor específico.

```
C#
public bool ContainsValue (TValue value);
```

Ejemplo:

```
 0 REFERENCES
1  internal class Program
2  {
3      0 references
4      private static void Main(string[] args)
5      {
6          Dictionary<string, string> openWith = new Dictionary<string, string>();
7
8          // Add some elements to the dictionary. There are no
9          // duplicate keys, but some of the values are duplicates.
10         openWith.Add("txt", "notepad.exe");
11         openWith.Add("bmp", "paint.exe");
12         openWith.Add("dib", "paint.exe");
13         openWith.Add("rtf", "wordpad.exe");
14
15         ↓
16         if(openWith.ContainsKey("notepad")){
17             Console.WriteLine("El valor se encuentra registrado.");
18         }else{
19             Console.WriteLine("El valor no se encuentra registrado.");
20         }
21         Dictionary<string, string>.ValueCollection valueColl = openWith.Values;
22
23         foreach (string valor in valueColl){
24             Console.WriteLine(valor);
25         }
26     }
27 }
```

```
PS D:\projectsNetCore\Practica01> dotnet run
El valor no se encuentra registrado.
notepad.exe
paint.exe
paint.exe
wordpad.exe
○ PS D:\projectsNetCore\Practica01>
```

2.1.4.2.5. Remove

Quita el valor con la clave especificada de Dictionary<TKey,TValue>.

C#

```
public bool Remove (TKey key);
```

Ejemplo:

```

1 internal class Program
2 {
3     0 references
4     private static void Main(string[] args)
5     {
6         Dictionary<string, string> openWith = new Dictionary<string, string>();
7
8         // Add some elements to the dictionary. There are no
9         // duplicate keys, but some of the values are duplicates.
10        openWith.Add("txt", "notepad.exe");
11        openWith.Add("bmp", "paint.exe");
12        openWith.Add("dib", "paint.exe");
13        openWith.Add("rtf", "wordpad.exe");
14
15        if(openWith.ContainsKey("notepad")){
16            Console.WriteLine("El valor se encuentra registrado.");
17        }else{
18            Console.WriteLine("El valor no se encuentra registrado.");
19        }
20        Dictionary<string, string>.ValueCollection valueColl = openWith.Values;
21
22        openWith.Remove("dib");
23
24        foreach (string valor in valueColl){
25            Console.WriteLine(valor);
26        }
27    }
28 }

```

```

PS D:\projectsNetCore\Practica01> dotnet run
El valor no se encuentra registrado.
notepad.exe
paint.exe
wordpad.exe
PS D:\projectsNetCore\Practica01>

```

3. Programación Avanzada

3.1. Programación Orientada a objetos

La programación orientada a objetos (POO) es un paradigma de programación que organiza el código en unidades llamadas "objetos" que representan entidades del mundo real. Estos objetos se definen mediante características (atributos) y comportamientos (métodos) relacionados entre sí.

El concepto fundamental de la POO es la abstracción, que permite modelar entidades complejas del mundo real en el código. Los objetos son instancias de clases, que actúan como plantillas o moldes para crear objetos con características y comportamientos similares.

Los principales conceptos de la programación orientada a objetos incluyen:

- Clases: Son las estructuras que definen las propiedades y comportamientos de los objetos. Una clase es como un plano o molde a partir del cual se crean múltiples instancias (objetos). Define los atributos (variables) y métodos (funciones) que los objetos de esa clase pueden tener.

- Objetos: Son instancias de una clase específica. Cada objeto tiene sus propios valores de atributos y puede realizar acciones (métodos) asociadas a él. Los objetos interactúan entre sí a través de mensajes para realizar tareas.
- Atributos: Son las características o propiedades de un objeto. Pueden ser variables que almacenan datos o referencias a otros objetos. Los atributos definen el estado de un objeto y representan información sobre el objeto.
- Métodos: Son las acciones o comportamientos que un objeto puede realizar. Los métodos son funciones asociadas a un objeto específico y pueden acceder y modificar los atributos del objeto. Permiten que los objetos realicen operaciones específicas y se comuniquen entre sí.
- Encapsulación: Es el concepto de ocultar los detalles internos de un objeto y exponer solo una interfaz pública para interactuar con él. Los objetos encapsulan su estado y comportamiento, y solo permiten el acceso a través de métodos específicos.
- Herencia: Permite que una clase herede características y comportamientos de otra clase existente, creando una relación de jerarquía entre ellas. La herencia permite reutilizar código y extender las funcionalidades de las clases base.
- Polimorfismo: Es la capacidad de objetos de diferentes clases de responder al mismo mensaje de diferentes maneras. Permite que diferentes objetos respondan a un mismo método de acuerdo a su propia implementación, lo que facilita la flexibilidad y la generalización del código.

3.1.3. Encapsulamiento

En C#, el concepto de encapsulamiento implica el ocultamiento de los atributos de un objeto para que solo puedan ser modificados a través de operaciones específicas definidas dentro del objeto. Esta práctica está íntimamente ligada con la noción de visibilidad y los modificadores de acceso.

3.1.3.1. Modificadores de acceso

Los modificadores de acceso nos introducen al concepto de encapsulamiento en C#. El objetivo del encapsulamiento es controlar el acceso a los datos que componen un objeto o instancia, lo que implica que una clase y sus objetos que utilizan modificadores de acceso (especialmente el modificador `private`) son considerados objetos encapsulados.

Los modificadores de acceso brindan un nivel de seguridad más alto a nuestras aplicaciones al restringir el acceso a diferentes atributos, métodos y constructores. Esto asegura que los usuarios deban seguir una "ruta" específica definida por nosotros para acceder a la información.

Es probable que nuestras aplicaciones sean utilizadas por otros programadores o usuarios con diferentes niveles de experiencia. Mediante el uso de modificadores de acceso, podemos asegurarnos de que un valor no sea modificado incorrectamente por otro programador o usuario. Por lo general, se logra acceder a los atributos a través de los métodos `get` y `set`, ya que es necesario que los atributos de una clase sean declarados como privados. A continuación, veremos detalladamente cada uno de los modificadores de acceso.

3.1.3.1.1 Modificador de acceso Private

El modificador private en C# es el más restrictivo de todos. Por lo tanto, cualquier elemento de una clase que sea declarado como private solo puede ser accedido por esa misma clase, y no por ninguna otra clase, sin importar la relación que tengan entre sí. Esto significa que, por ejemplo, si un atributo es declarado como private, solo los métodos y constructores de esa misma clase podrán acceder a él, y ninguna otra clase podrá tener acceso a dicho atributo.

3.1.3.1.2 Modificador de acceso protected

El modificador de acceso "protected" permite acceder a los componentes con ese modificador desde la misma clase, clases en el mismo paquete y clases que hereden de ella, incluso si se encuentran en diferentes paquetes.

3.1.3.1.3 Modificador de acceso public

El modificador de acceso "public" es el más amplio y permite acceder a los componentes de una clase desde cualquier otra clase o instancia, sin importar el paquete en el que se encuentren. En contraste con el modificador "private", el modificador "public" permite un acceso más abierto y no impone restricciones en términos de paquetes o procedencia.

Modificador	La misma clase	Mismo paquete	Subclase	Otro paquete
private	✓	✗	✗	✗
default	✓	✓	✗	✗
protected	✓	✓	✓	✗
public	✓	✓	✓	✓

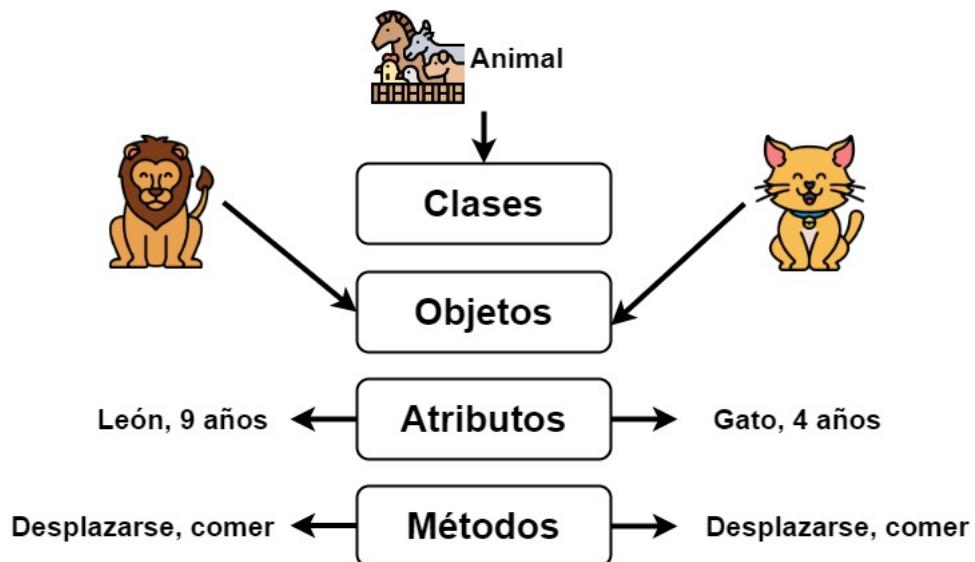
Fuente: Documento MD_C#. Autor. Ing Juan Mariño

3.1.4. Clases C#

El proceso de creación de programas orientados a objetos implica, de manera resumida, la creación de clases y la posterior creación de objetos basados en estas clases. Las clases sirven como el modelo a partir del cual se estructuran los datos y comportamientos del programa.

Una clase actúa como una **plantilla** genérica que define cómo serán los objetos de un tipo específico. Por ejemplo, una clase que representa a los animales podría llamarse "Animal" y tendría atributos como "tipo" y "edad" (que generalmente son propiedades). Además, incluiría una serie de comportamientos que los animales pueden tener, como desplazarse o comer, los cuales se implementan como métodos de la clase (funciones).

Un ejemplo básico de un objeto, como mencionamos anteriormente, podría ser un animal. Para representar este objeto, creamos un atributo llamado "edad" que almacena la edad del animal. Además, el animal puede envejecer, por lo que podríamos definir un método que se encarga de actualizar la edad. En resumen, una clase se encarga de definir tanto los datos (como el atributo "edad") como la lógica (como el método de envejecimiento) relacionados con muchos objetos en un programa. La clase proporciona una definición global y genérica que se utiliza para crear múltiples instancias de objetos. La siguiente imagen nos ayuda a entender mejor la diferencia entre clases y objetos.



Escuela ← Objeto

Atributos →

- Nombre
- Ciudad
- Tipo:
 - Preescolar
 - Primaria
 - Secundaria

← Métodos

- Iniciar Jornada Académica
- Terminar Jornada Académica
- Timbrar
- Iniciar Emergencia

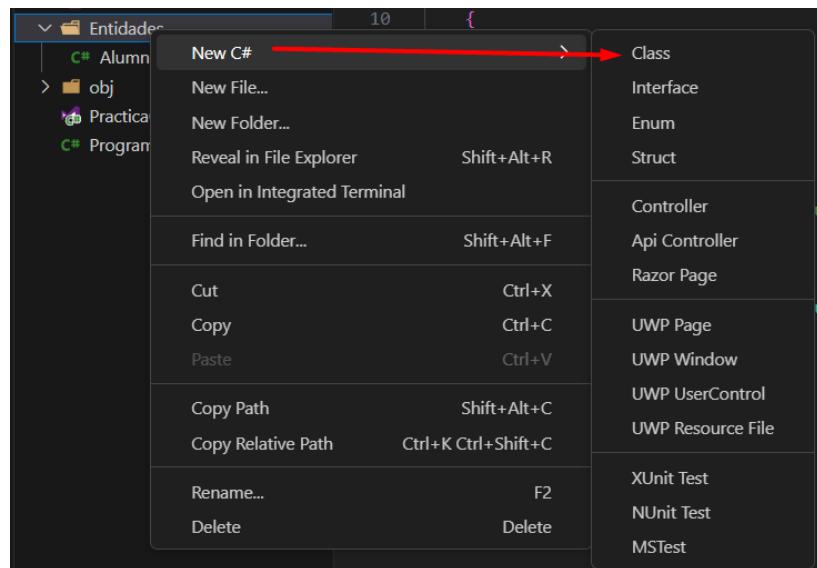


Objeto

- Es descrito por sus atributos
- Sus acciones se representan con métodos (funciones)

3.1.4.1. Creación de clases

- Cree una carpeta llamada Entidades
- Cree una nueva clase llamada Alumno en la carpeta Entidades. Haga clic derecho sobre la carpeta Entidades y en el menú emergente seleccione la opción New C#>Class



- Defina los siguientes atributos

```
1 reference
private string ? idAlumno;
0 references
private string ? nombre;
0 references
private string ? email;
0 references
private int edad;
```

- Genere el constructor vacío.

Forma 1

```
1 reference
public Alumno(){
    this.idAlumno = Guid.NewGuid().ToString();
}
```

Forma 2

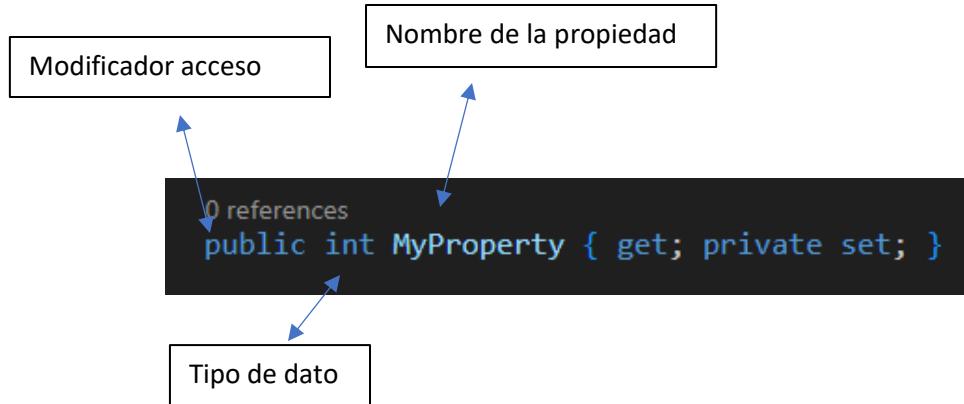
```
1 reference
public Alumno()=> idAlumno = Guid.NewGuid().ToString();
```

- Las primeras líneas de código incluyen directivas **using** para importar los espacios de nombres necesarios, como **System** y **System.Collections.Generic**. Estas directivas permiten utilizar las clases y funciones definidas en esos espacios de nombres sin tener que escribir el nombre completo cada vez.
- Luego, se declara el espacio de nombres **PracticaOop.Entidades** utilizando la palabra clave **namespace**. Un espacio de nombres proporciona un ámbito lógico para agrupar clases y otros elementos relacionados.
- Dentro del espacio de nombres, se define la clase **Alumno** utilizando la palabra clave **class**. Esta clase representa a un estudiante y tiene propiedades para almacenar su identificador (**idAlumno**), nombre, email y edad.
- Las propiedades **idAlumno**, **nombre** y **email** se declaran como **string?**, lo que indica que son tipos de referencia y pueden aceptar valores nulos. La propiedad **edad** se declara como un tipo **int**, que es un tipo de valor y no puede ser nulo.
- El constructor de la clase **Alumno** se define utilizando la sintaxis **public Alumno() => idAlumno = Guid.NewGuid().ToString();**. Este constructor no tiene parámetros y se declara como público, lo que significa que puede ser accedido desde cualquier parte del código.
- En el cuerpo del constructor, se utiliza **Guid.NewGuid().ToString()** para generar un nuevo identificador único global (GUID) y se asigna a la propiedad **idAlumno** del objeto **Alumno**. El método **Guid.NewGuid()** genera un nuevo GUID, y **ToString()** lo convierte en una cadena de texto.

En resumen, el código define la clase **Alumno** con propiedades para almacenar información sobre un estudiante y un constructor que genera un identificador único para cada instancia de la clase **Alumno**.

3.1.4.2 Get y Set

En programación orientada a objetos (POO), los métodos **get** y **set**, también conocidos como "getters" y "setters", son métodos utilizados para acceder y modificar los atributos (propiedades) de una clase de manera controlada. Estos métodos permiten mantener el principio de encapsulamiento al controlar el acceso a los datos de la clase, evitando así el acceso directo a los atributos desde fuera de la clase.



Método `get` (Getter):

- Un método `get` es utilizado para obtener el valor de un atributo privado de una clase.
- Método `set` (Setter):

- Un método `set` es utilizado para asignar un valor a un atributo privado de una clase.

```

1  namespace Clases01;
2
3  public class Persona
4  {
5      private string idAlumno;
6      private string nombreAlumno;
7
8      public Persona(string _nombreAlumno){
9          this.idAlumno = Guid.NewGuid().ToString();
10         this.idAlumno = _nombreAlumno;
11     }
12
13     public string IdAlumno { get; }
14     public string NombreAlumno { get; set; }
  
```

3.1.4.3. Instancia de clase

Para crear una instancia de clase debe incluir referencia a el nameSpace y la carpeta donde se encuentra creada la clase.

```
1  using Clases01;
2
3      0 references
4  internal class Program
5  {
6      0 references
7      private static void Main(string[] args)
8      {
9          Persona estudiante = new Persona("Pepito");
10     }
11 }
```

```
1  using Clases01;
2
3      0 references
4  internal class Program
5  {
6      0 references
7      private static void Main(string[] args)
8      {
9          Persona estudiante = new Persona("Pepito");
10         Console.WriteLine("Id estudiante {0}", estudiante.IdAlumno);
11         Console.WriteLine("Id Nombre {0}", estudiante.NombreAlumno);
12     }
13 }
```

```
desarrollo@DESKTOP-L4V647N MINGW64 /d/projectsNetCore/Clases01
$ dotnet run
Id estudiante 7805caf4-ee52-48f2-9bb1-0790b8dbc960
Id Nombre Pepito
```


3.1.4.4. Constructor

En la programación orientada a objetos, incluyendo C#, un constructor es una función especial que se emplea para iniciar un objeto recién creado y asignar valores iniciales a sus variables de instancia. Su propósito principal es establecer el estado inicial del objeto antes de que se utilice en el programa. En resumen, un constructor es un método que se invoca automáticamente al crear un objeto de una clase determinada.

Un constructor tiene la finalidad de inicializar un objeto y establecer sus propiedades y valores predeterminados. Se caracteriza por tener el mismo nombre que la clase y no retornar ningún valor, ya que su objetivo principal es la inicialización del objeto.

Adicionalmente, un constructor puede recibir argumentos, lo que permite configurar el objeto con valores específicos durante su creación. Esto facilita la personalización del objeto al proporcionar valores iniciales personalizados. En C# hay dos diferentes tipos de constructores que pueden utilizarse dependiendo de las necesidades de la clase y de los objetos que se van a crear.

```
1  namespace Clases01;
2
3      0 references
4  public class Persona
5  {
6      2 references
7      |    private string idAlumno;
8      |    0 references
9      |    private string nombreAlumno;
10
11     0 references
12     |    public Persona(string _nombreAlumno){
13     |        this.idAlumno = Guid.NewGuid().ToString();
14     |        this.idAlumno = _nombreAlumno;
15     }
16 }
```

Namespace: namespace Clases01;

- El **namespace** es un contenedor para organizar clases y otros tipos en C#. Ayuda a evitar conflictos de nombres y proporciona un ámbito para las clases.

Constructor:

- El constructor es un método especial que se llama cuando se crea una nueva instancia de la clase **Persona**.
- Recibe un parámetro **_nombreAlumno**, que se utiliza para inicializar el campo **nombreAlumno**.
- La línea **this.idAlumno = Guid.NewGuid().ToString();** genera un nuevo identificador único (GUID) y lo asigna al campo **idAlumno**.
- Luego, la línea **this.idAlumno = _nombreAlumno;** sobrescribe el valor de **idAlumno** con el valor del parámetro **_nombreAlumno**.

3.1.4.5 Herencia

La herencia en programación orientada a objetos (POO) es un mecanismo fundamental que permite a una clase heredar las características (atributos y métodos) de otra clase. No se refiere a la herencia de bienes materiales, sino a la capacidad de una clase de aprovechar y extender el código existente.

La herencia en POO permite la creación de nuevas clases basadas en clases existentes para reutilizar el código y establecer una jerarquía de clases en una aplicación. Una clase hija hereda los atributos y métodos de su clase padre, y además puede agregar nuevos atributos, métodos o redefinir los heredados.

La reutilización del código es una de las ventajas clave de la herencia, ya que permite utilizar el mismo código una y otra vez, lo que agiliza el desarrollo y lo hace más eficiente. Además, la herencia ayuda a mantener un código más limpio y estructurado, al reducir la cantidad de líneas de código necesarias y mejorar su legibilidad.

En C#, se utiliza la terminología de superclase y subclase. La superclase es la clase de la que se heredan las características, también conocida como clase base o principal. La subclase es la clase que hereda de la superclase, y puede agregar sus propios elementos únicos, además de heredar los campos y métodos de la superclase.

En C#, la herencia se logra mediante la extensión de una superclase, también conocida como clase padre, con una subclase o clase secundaria. Esto se hace utilizando la palabra clave `extends`. Si no se declara una superclase específica, la clase se extiende implícitamente de la clase `Object`. La clase `Object` es la raíz de todas las jerarquías de herencia en C# y es la única clase que no se extiende de otra clase.

Diagrama de clases

Un diagrama de clases es una representación visual de las clases en un sistema o programa orientado a objetos. Es una herramienta de modelado estática que muestra la estructura y las relaciones entre las clases, sus atributos y métodos.

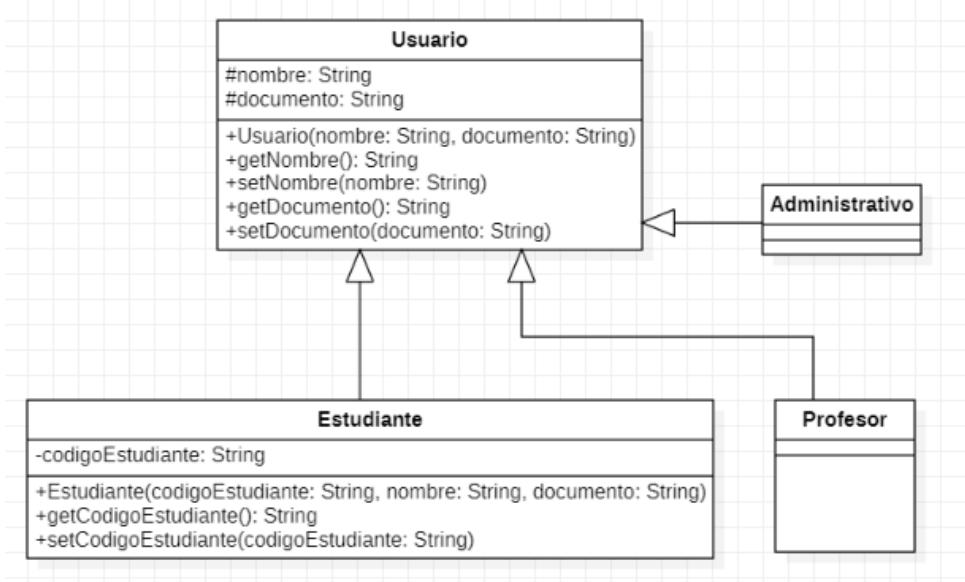
En un diagrama de clases, se utilizan símbolos gráficos para representar las clases, las relaciones entre ellas y los diferentes componentes de cada clase, como los atributos y métodos. Los símbolos más comunes utilizados en un diagrama de clases incluyen cajas rectangulares para representar las clases, líneas con flechas para mostrar las relaciones entre las clases y etiquetas para indicar los atributos y métodos de cada clase.

El diagrama de clases proporciona una vista de alto nivel de la estructura del sistema, lo que facilita la comprensión de las relaciones y la organización de las clases. También puede incluir información adicional, como la visibilidad de los atributos y métodos (público +, privado -, protegido #), los tipos de datos, las asociaciones entre clases y las herencias.

Animal
-tipo: String -edad: int
+getEdad(): int +setEdad(edad: int) +getTipo(): String +setTipo(tipo: String) +caminar() +comer()

Para la herencia un diagrama de clase se representa con una flecha de la siguiente forma:

Volviendo a la explicación práctica de herencia, imaginemos que tenemos que desarrollar un aplicativo para registro de usuarios en una institución educativa. Sabemos que cada usuario tendrá muchos atributos, pero, todos los usuarios no serán del mismo tipo. En situaciones como estas es necesario separar el tipo de usuarios; sin embargo, habrá ciertas características que todos comparten por igual y deberíamos tenerla en una clase que las agrupe (super clase) y así, evitar repetir código.



Ejercicio:

La federación colombiana de fútbol desea generar crear un programa que permita llevar el registro de cada uno de los equipos que integran el torneo Liga BetPlay. Cada uno de los equipos que integran la liga están conformados por jugadores, entrenadores y masajistas.

La información que se tiene por cada uno de los equipos e integrantes es la siguiente.

- Equipo: Nombre, año de fundación, propietario y ciudad de origen.
- Jugador: Id jugador, nombre, edad, dorsal, posición de juego, ciudad de origen
- Entrenador: Id entrenador, nombre, edad, código de federación, especialidad, ciudad de origen.
- Masajista: Id, nombre, edad, titulación, años de experiencia y ciudad de origen

3.1.5. Clases Abstractas

Las clases abstractas son clases que no se pueden instanciar directamente, es decir, no se pueden crear objetos a partir de ellas. En cambio, se utilizan como plantillas o bases para derivar subclases que sí se pueden instanciar.

A continuación, se enumeran algunas características y conceptos clave relacionados con las clases abstractas:

- No se pueden crear instancias directamente: Como se mencionó anteriormente, no se pueden crear objetos directamente a partir de una clase abstracta. Debe derivarse una subclase de la clase abstracta y crear objetos de la subclase.
- Pueden contener implementaciones parciales: Las clases abstractas pueden tener tanto métodos abstractos como métodos con implementaciones concretas. Los métodos abstractos son métodos que solo se declaran en la clase abstracta, sin proporcionar una implementación. Estos métodos deben ser implementados por las subclases derivadas.
- Proporcionan una interfaz común: Una clase abstracta puede definir una interfaz común y comportamientos generales que se comparten entre sus subclases. Esto facilita la reutilización del código y garantiza que las subclases tengan una estructura y comportamiento coherentes.
- Permiten herencia múltiple limitada: Al heredar de una clase abstracta, una subclase puede heredar tanto la interfaz y el comportamiento de la clase abstracta, como también heredar de otra clase base. Esto permite una forma limitada de herencia múltiple, que no está permitida en algunos lenguajes de programación.
- Sirven como punto de extensión: Las clases abstractas brindan un punto de extensión para futuras subclases. Pueden definir métodos y comportamientos que se espera que sean implementados y extendidos por las subclases. Esto permite una mayor flexibilidad y capacidad de crecimiento en la estructura de la jerarquía de clases.

Para declarar una clase abstracta en C#, se utiliza la palabra clave **abstract** antes de la declaración de la clase:

```
public abstract class MiClaseAbstracta
{
    // Definición de métodos abstractos y/o métodos concretos
}
```

Ejemplo:

```
// Clase abstracta
public abstract class Vehiculo
{
    // Propiedad regular
    public string Marca { get; set; }

    // Método abstracto que debe ser implementado por las clases hijas
    public abstract void Conducir();

    // Método no abstracto (con implementación) que puede ser heredado por las clases hijas
    public void MostrarMarca()
    {
        Console.WriteLine("Marca del vehículo: " + Marca);
    }
}

// Clase que hereda de la clase abstracta
public class Coche : Vehiculo
{
    // Implementación del método abstracto Conducir
    public override void Conducir()
    {
        Console.WriteLine("Conduciendo el coche...");
    }
}

// Clase que hereda de la clase abstracta
public class Moto : Vehiculo
{
    // Implementación del método abstracto Conducir
    public override void Conducir()
    {
        Console.WriteLine("Conduciendo la moto...");
    }
}

class Program
{
    static void Main()
    {
        // No se puede instanciar un objeto directamente de la clase abstracta
        // Vehiculo miVehiculo = new Vehiculo(); // Esto dará error

        // Sin embargo, se pueden crear objetos de las clases hijas
        Vehiculo miCoche = new Coche();
        miCoche.Marca = "Toyota";
        miCoche.MostrarMarca();
        miCoche.Conducir();

        Vehiculo miMoto = new Moto();
        miMoto.Marca = "Honda";
        miMoto.MostrarMarca();
        miMoto.Conducir();
    }
}
```

```
// Clase abstracta
public abstract class Vehiculo
{
    // Propiedad regular
    public string Marca { get; set; }

    // Método abstracto que debe ser implementado por las clases hijas
    public abstract void Conducir();

    // Método no abstracto (con implementación) que puede ser heredado por las clases hijas
    public void MostrarMarca()
    {
        Console.WriteLine("Marca del vehículo: " + Marca);
    }
}

// Clase que hereda de la clase abstracta
public class Coche : Vehiculo
{
    // Implementación del método abstracto Conducir
    public override void Conducir()
    {
        Console.WriteLine("Conduciendo el coche...");
    }
}

// Clase que hereda de la clase abstracta
public class Moto : Vehiculo
{
    // Implementación del método abstracto Conducir
    public override void Conducir()
    {
        Console.WriteLine("Conduciendo la moto...");
    }
}

class Program
{
    static void Main()
    {
        // No se puede instanciar un objeto directamente de la clase abstracta
        // Vehiculo miVehiculo = new Vehiculo(); // Esto dará error

        // Sin embargo, se pueden crear objetos de las clases hijas
        Vehiculo miCoche = new Coche();
        miCoche.Marca = "Toyota";
        miCoche.MostrarMarca();
        miCoche.Conducir();

        Vehiculo miMoto = new Moto();
        miMoto.Marca = "Honda";
        miMoto.MostrarMarca();
        miMoto.Conducir();
    }
}
```

3.1.6. Polimorfismo

El polimorfismo es uno de los conceptos clave en la programación orientada a objetos (POO) que permite a un objeto ser tratado como si fuera de un tipo diferente en tiempo de ejecución. En C#, el polimorfismo se logra a través de la herencia y la implementación de interfaces.

Hay dos tipos principales de polimorfismo en C#:

Polimorfismo de tiempo de compilación (Compile-time Polymorphism): También conocido como enlace estático o polimorfismo estático, ocurre en tiempo de compilación y se logra mediante la sobrecarga de métodos y operadores. En la sobrecarga, un mismo método o operador tiene diferentes implementaciones según los tipos de parámetros que reciba.

```
1
2  namespace Clases01;
3    2 references
4  public class Calculadora
5  {
6    1 reference
7    public int Sumar(int a, int b)
8    {
9      return a + b;
10   }
11  1 reference
12  public double Sumar(double a, double b)
13  {
14    return a + b;
15 }
```

Polimorfismo de tiempo de ejecución (Runtime Polymorphism): También conocido como enlace dinámico o polimorfismo dinámico, ocurre en tiempo de ejecución y se logra mediante la herencia y la implementación de interfaces.

3.1.7. Interfaces

Una interfaz en C# es un tipo de referencia que define un contrato que debe ser implementado por las clases que la utilicen. Es un mecanismo para lograr la abstracción y el polimorfismo en la programación orientada a objetos. Una interfaz define un conjunto de miembros (métodos, propiedades, eventos e indexadores) sin proporcionar una implementación concreta de esos miembros.

```

using Clases01;

internal class Program
{
    private static void Main(string[] args)
    {
        // Uso de la clase Calculadora
        // Calculadora calculadora = new Calculadora();

        // int resultadoEntero = calculadora.Sumar(5, 10);      // Utiliza el método int Sumar(int, int)

        // double resultadoDouble = calculadora.Sumar(3.14, 2.71); // Utiliza el método double Sumar(double, do
        // uble)
        // Uso del polimorfismo de tiempo de ejecución
        Animal animal1 = new Perro();
        Animal animal2 = new Gato();

        animal1.HacerSonido(); // Salida: El perro ladra: ¡Guau, guau!
        animal2.HacerSonido(); // Salida: El gato maulla: ¡Miau, miau!
    }
    public class Animal
    {
        public virtual void HacerSonido()
        {
            Console.WriteLine("Hace un sonido indefinido.");
        }
    }

    public class Perro : Animal
    {
        public override void HacerSonido()
        {
            Console.WriteLine("El perro ladra: ¡Guau, guau!");
        }
    }

    public class Gato : Animal
    {
        public override void HacerSonido()
        {
            Console.WriteLine("El gato maulla: ¡Miau, miau!");
        }
    }
}

```

3.2. LinQ

LINQ (Language Integrated Query) es una característica de Microsoft .NET Framework que proporciona una forma consistente de consultar y manipular diferentes fuentes de datos utilizando consultas similares a SQL en lenguajes de programación como C# y Visual Basic. LINQ permite realizar consultas sobre colecciones de objetos, bases de datos, servicios web y otros orígenes de datos, todo ello utilizando una sintaxis común y un conjunto de operadores estándar.

Con LINQ, puedes escribir consultas expresivas y legibles para filtrar, ordenar y proyectar datos de manera sencilla. Puedes utilizar consultas LINQ con objetos en memoria, como listas o matrices, o con fuentes de datos externas, como bases de datos SQL. LINQ también admite la composición de consultas, lo que significa que puedes combinar varias consultas en una sola y aplicar operaciones adicionales.

La principal ventaja de LINQ es que permite escribir consultas de manera declarativa, centrándote en lo que quieres obtener en lugar de como obtenerlo. Esto hace que el código sea

más legible y fácil de mantener. Además, LINQ aprovecha el sistema de tipos fuertes de .NET y realiza comprobaciones de tipos en tiempo de compilación, lo que ayuda a reducir errores y mejorar el rendimiento.

¿Qué no es LINQ?

- No es un lenguaje de programación.
- No es un componente de SQL.
- No es un componente de base de datos.
- No es una librería de terceros.

Programación declarativa vs. imperativa

La programación imperativa se basa en instrucciones detalladas que indican explícitamente cómo realizar una tarea. En este enfoque, se describe el algoritmo paso a paso, especificando el flujo de control y las acciones a realizar en cada paso. El código imperativo se enfoca en cómo se deben realizar las operaciones y cómo se deben manipular los datos. Ejemplos de lenguajes de programación imperativos son C, C++, C# y Python.

Por otro lado, la programación declarativa se centra en describir qué se desea obtener, sin entrar en los detalles de cómo lograrlo. En lugar de especificar los pasos y acciones específicas, se define el resultado deseado. El código declarativo se enfoca en la lógica y la relación entre los datos, dejando que el lenguaje o el entorno de programación se encargue de la implementación subyacente. Ejemplos de lenguajes de programación declarativos son SQL (Structured Query Language) utilizado para consultas de bases de datos, HTML (HyperText Markup Language) utilizado para describir la estructura de una página web, y LINQ (Language Integrated Query) que mencioné anteriormente.

En resumen, la programación imperativa se basa en instrucciones detalladas sobre cómo realizar una tarea, mientras que la programación declarativa se enfoca en describir el resultado deseado

sin especificar los pasos específicos para lograrlo. Ambos enfoques tienen sus ventajas y se utilizan en diferentes contextos y paradigmas de programación.

Comparativa

Programación declarativa

- Paradigma de la programación.
- Instrucciones donde específico lo que quiero y no como lo quiero.
- Contraposición a la programación imperativa.
- Fiable y simple.

Programación imperativa

- Paradigma de la programación.
- Secuencia paso a paso de instrucciones.
- Contraposición a la programación declarativa.
- código más extenso pero fácil de interpretar.

3.2.1. Usando Linq

Cuando se utiliza Linq en c# es indispensable utilizar el espacio de nombres system.linq

```
1  using System.Linq; ←
2  internal class Program
3  {
4      private static void Main(string[] args)
5      {
6          var frutas = new string[] { "Sandia", "Fresas", "Mango", "Ciruelas", "Mango Azucar" };
7          var lstMangos = frutas.Where(p => p.StartsWith("Mango")).ToList();
8          lstMangos.ForEach(item => Console.WriteLine(item));
9
10     }
11 }
```

```
$ dotnet run
Mango
Mango Azucar
```

- ② Primero, se importa el espacio de nombres **System.Linq**. Esto es necesario para utilizar las funcionalidades de LINQ.
- ② A continuación, se define una clase llamada **Program**. Dado que la clase tiene el modificador **internal**, solo es accesible desde el ensamblado actual.
- ② Dentro de la clase **Program**, se declara un método estático llamado **Main**, que es el punto de entrada del programa. Este método recibe un argumento de tipo arreglo de cadenas llamado **args**.
- ② En el cuerpo del método **Main**, se crea un arreglo de cadenas llamado **frutas** y se inicializa con algunos valores.
- ② A continuación, se utiliza LINQ para filtrar las frutas que comienzan con la palabra "Mango". La línea de código **frutas.Where(p => p.StartsWith("Mango")).ToList()** realiza el filtrado utilizando el método **Where** de LINQ y la expresión lambda **p =>**

`p.StartsWith("Mango")` como criterio de filtrado. Esto devuelve una lista (`List<string>`) que contiene todas las frutas que cumplen con el filtro.

- ② La lista de mangos filtrada se almacena en la variable `IstMangos`.
- ② Luego, se utiliza el método `ForEach` de la lista `IstMangos` para iterar sobre cada elemento y realizar una acción en cada uno de ellos. En este caso, se utiliza una expresión lambda `item => Console.WriteLine(item)` para imprimir cada mango en la consola.

En resumen, el código muestra cómo utilizar LINQ para filtrar elementos de una lista de frutas y luego imprimir los mangos filtrados en la consola.

En el siguiente ejercicio vamos a utilizar un archivo de extensión json como fuente de datos para realizar las diferentes pruebas y estudiar las funciones utilizadas en linq.

Lo primero que tenemos que hacer es crear un proyecto y configurarlo debidamente para que soporte el archivo de recurso de datos y poderlo utilizar de forma global. Para configurar el proyecto que soporta el archivo json siga los siguientes pasos:

1. Arrastre el archivo books.json

2. Abra el archivo de configuración del proyecto  `linq-csharp.csproj` y agregue la directiva `<ItemGroup>`

```
1 <Project Sdk="Microsoft.NET.Sdk">
2
3   <PropertyGroup>
4     <OutputType>Exe</OutputType>
5     <TargetFramework>net7.0</TargetFramework>
6     <RootNamespace>linq_csharp</RootNamespace>
7     <ImplicitUsings>enable</ImplicitUsings>
8     <Nullable>enable</Nullable>
9   </PropertyGroup>
10  <ItemGroup>
11    <Content Include="books.json">
12  </ItemGroup>
13 </Project>
```

3. Genere la clase Book

```

1  public class Book{
2      2 references
3      string ? title;
4      2 references
5      int pageCount;
6      2 references
7      DateTime publishedDate;
8      2 references
9      string ? status;
10     2 references
11     string[] ? authors;
12     0 references
13     string[] ? categories;
14
15     0 references
16     public string? Title { get => title; set => title = value; }
17     0 references
18     public int PageCount { get => pageCount; set => pageCount = value; }
19     0 references
20     public DateTime PublishedDate { get => publishedDate; set => publishedDate = value; }
21     0 references
22     public string? Status { get => status; set => status = value; }
23     0 references
24     public string[]? Authors { get => authors; set => authors = value; }
25     2 references
26     public string[]? Categories { get => Categories; set => Categories = value; }
27
28 }

```

4. Cree una clase llamada LinqQueries.cs

```

1  public class LinqQueries{
2
3      0 references
4      public LinqQueries(){
5          }
6

```

5. Modifique la clase LinqQueries como se muestra en la imagen inferior:

```

1  2 references
2  public class LinqQueries{
3      2 references
4      List<Book> lstBooks = new List<Book>();
5      1 reference
6      public LinqQueries(){
7          using(StreamReader reader = new StreamReader("books.json")){
8              string json = reader.ReadToEnd();
9              this.lstBooks = System.Text.Json.JsonSerializer.Deserialize<List<Book>>(json,new System.Text.Json.JsonSerializerOptions(){PropertyNameCaseInsensitive = true}) ?? new List<Book>();
10         }
11     }
12 }

```

- En la línea 2, se declara una lista de objetos del tipo "Book" llamada "lstBooks" y se inicializa como una nueva instancia de la clase "List<Book>".

- En el constructor de la clase (líneas 4 a 9), se utiliza un bloque "using" para leer el contenido del archivo "books.json" utilizando un objeto "StreamReader". El archivo debe estar ubicado en el directorio de ejecución del programa. Se lee todo el contenido del archivo y se almacena en la variable "json".
- En la línea 7, se utiliza el método "Deserialize" de la clase "JsonSerializer" para convertir el contenido JSON en una lista de objetos del tipo "Book". Se especifica el tipo de objeto esperado (List<Book>) como argumento genérico y se pasa la variable "json" como la representación JSON a ser deserializada.
- Se utiliza la clase "JsonSerializerOptions" en la línea 7 para configurar las opciones de serialización/deserialización. En este caso, se establece la opción "PropertyNameCaseInsensitive" como verdadera, lo que permite que las propiedades de los objetos JSON sean insensibles a mayúsculas y minúsculas.
- Si la deserialización es exitosa y se obtiene la lista de libros, se asigna a la variable "IstBooks". Si la deserialización falla o el archivo no existe, se asigna una nueva instancia de la lista de libros utilizando el operador de coalescencia nula ("??").
- En la línea 12, se define un método llamado "AllCollection" que devuelve un objeto "IEnumerable<Book>". Este método simplemente retorna la lista de libros almacenada en "IstBooks".

En resumen, la clase "LinqQueries" carga una lista de libros desde un archivo JSON en su constructor y proporciona un método para acceder a todos los libros en esa lista. Esto puede ser útil para realizar consultas y operaciones en la colección de libros utilizando LINQ u otras técnicas de consulta.

3.2.1.1 Operador Where en Linq

El operador **where** se utiliza en consultas LINQ para filtrar elementos de una secuencia basándose en una condición específica. Su objetivo es seleccionar solo aquellos elementos que cumplan con la condición proporcionada.

La sintaxis general del operador **where** es la siguiente:

```
var resultado = from elemento in secuencia
                where condicion
                select elemento;
```

O también puede ser utilizada la sintaxis de métodos de extensión:

```
var resultado = secuencia.Where(elemento => condicion);
```

Donde:

- **secuencia** representa la colección de elementos sobre la cual se va a realizar el filtro.
- **elemento** es la variable que representa cada elemento de la secuencia en la que se está iterando.
- **condición** es una expresión booleana que define el criterio de filtrado. Solo los elementos que cumplan con esta condición serán seleccionados en el resultado.

```
var numeros = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// Utilizando la sintaxis de consultas
var numerosPares = from num in numeros
                    where num % 2 == 0
                    select num;

// Utilizando la sintaxis de métodos de extensión
var numerosPares = numeros.Where(num => num % 2 == 0);

foreach (var num in numerosPares)
{
    Console.WriteLine(num); // Imprime los números pares: 2, 4, 6, 8, 10
}
```

Reto operador Where 1

Utilizando el operador Where retorna los libros que fueron publicados después del año 2000.

Solución

```
public IEnumerable<Book> LibrosDespues2000(){
    //Extension method
    //return lstBooks.Where(book => book.PublishedDate.Year > 2000);
    return from book in lstBooks where book.PublishedDate.Year > 2000 select book;
}
```

El método utiliza dos formas diferentes de sintaxis de consulta de LINQ para lograr el mismo resultado. Ambas formas son equivalentes y producirán el mismo resultado.

La primera forma utiliza el método de extensión **Where** en la lista **lstBooks** para filtrar los libros basados en una condición. La condición en este caso es **book.PublishedDate.Year > 2000**, lo que significa que solo se seleccionarán los libros cuya fecha de publicación sea posterior al año 2000.

La segunda forma utiliza la sintaxis de consulta de LINQ utilizando la cláusula **from** y **where**. En este caso, la consulta selecciona los libros (**book**) de la lista **lstBooks** donde **book.PublishedDate.Year > 2000**.

Ambas formas de consulta producirán la misma secuencia de libros que cumplen con la condición de haber sido publicados después del año 2000.

Es importante destacar que el código asume que la clase **Book** tiene una propiedad **PublishedDate** de tipo **DateTime** que representa la fecha de publicación de un libro.

Reto operador where 2

Utilizando el operador Where retorna los libros que tengan más de 250 páginas y el título contenga las palabras in Action.

Solución

```
public IEnumerable<Book> LibrosMas250Pag(){
    return from book in lstBooks
           where book.PageCount > 250
             && (book.Title ?? String.Empty).Contains("in Action")
           select book;
}
```

- **from book in lstBooks** establece el rango o dominio de la consulta como la lista **lstBooks**, permitiendo iterar sobre cada objeto **book** en la lista.
- **where book.PageCount > 250** filtra los libros para incluir solo aquellos con una cantidad de páginas mayor a 250.
- **(book.Title ?? String.Empty).Contains("in Action")** establece una condición adicional utilizando el operador de coalescencia nula (**??**) para manejar posibles valores nulos en la propiedad **Title**. Si el título del libro es nulo, se considerará como una cadena vacía antes de verificar si contiene la subcadena "in Action" utilizando el método **Contains**.
- En resumen, el método **LibrosMas250Pag** devuelve una secuencia de libros que tienen más de 250 páginas y cuyo título (manejado como una cadena vacía si es nulo) contiene la subcadena "in Action".

3.2.1.2 Operador All y Any en Linq

Los operadores **All** y **Any** son operadores de consulta en LINQ que se utilizan para verificar condiciones en una secuencia de elementos. Aquí tienes una explicación de cada uno de ellos:

- Operador **All**: El operador **All** se utiliza para verificar si todos los elementos de una secuencia cumplen con una condición específica. Devuelve **true** si todos los elementos satisfacen la condición y **false** en caso contrario.

```
bool resultado = secuencia.All(elemento => condicion);
```

```
var numeros = new List<int> { 2, 4, 6, 8, 10 };

bool todosSonPares = numeros.All(num => num % 2 == 0);
// Devuelve true, ya que todos los números de la lista son pares
```

En este ejemplo, el operador **All** verifica si todos los elementos de la lista **numeros** son números pares. La condición **num % 2 == 0** se aplica a cada elemento de la lista y debe ser verdadera para todos los elementos para que el resultado sea **true**.

- Operador **Any**: El operador **Any** se utiliza para verificar si al menos un elemento de una secuencia cumple con una condición específica. Devuelve **true** si al menos un elemento satisface la condición y **false** en caso contrario.

```
bool resultado = secuencia.Any(elemento => condicion);
```

```
var numeros = new List<int> { 1, 2, 3, 4, 5 };

bool hayNumerosPares = numeros.Any(num => num % 2 == 0);
// Devuelve true, ya que al menos uno de los números de la lista es par
```

Reto operador All

Utilizando el operador All verifica que todos los elementos de la colección tenga un valor en el campo Status.

Solución

```
public bool ValidarStatus(){
    return lstBooks.All(book => book.Status != String.Empty);
}
```

Reto operador Any

Utilizando el operador Any verifica si alguno de los libros fue publicado en 2005.

Solución

```
0 references
public bool ValidarFechaPub(){
    return lstBooks.Any(book => book.PublishedDate.Year == 2005);
}
```

3.2.1.3 Operador Contains en Linq

El operador **Contains** en LINQ se utiliza para verificar si una secuencia contiene un elemento específico. Puedes usarlo para verificar si un valor o una subcadena está presente en una colección.

El operador **Contains** devuelve un valor booleano (**true** o **false**) dependiendo de si el elemento buscado se encuentra en la secuencia o no.

Sintaxis:

```
bool resultado = secuencia.Contains(elemento);
```

Ejemplo

```
var numeros = new List<int> { 1, 2, 3, 4, 5 };

bool contieneTres = numeros.Contains(3);
// Devuelve true, ya que el número 3 está presente en la lista
```

Ejemplo

```
var palabras = new List<string> { "hola", "adiós", "buenos días" };

bool contieneHola = palabras.Contains("hola");
```

Reto operador Contains

Utilizando el operador Contains retorna los elementos que pertenezcan a la categoría de Python.

Solución

```
0 references
public IEnumerable<Book> GetBooksPython(){
    return lstBooks.Where(
        book => (book.Categories ?? Array.Empty<string>()).Contains("Python"));
}
```

3.2.1.4 Operador OrderBy y OrderByDescending en Linq

Los operadores **OrderBy** y **OrderByDescending** se utilizan en LINQ para ordenar una secuencia de elementos según un criterio específico. Aquí tienes una explicación de cada uno de ellos:

- Operador **OrderBy**: El operador **OrderBy** se utiliza para ordenar una secuencia de elementos en orden ascendente (de menor a mayor) según un criterio específico. Puedes especificar la clave de ordenamiento utilizando una expresión lambda que represente el valor por el cual deseas ordenar.

```
var resultado = secuencia.OrderBy(elemento => clave);
```

- Operador **OrderByDescending**: El operador **OrderByDescending** se utiliza para ordenar una secuencia de elementos en orden descendente (de mayor a menor) según un criterio específico. Al igual que con **OrderBy**, puedes especificar la clave de ordenamiento utilizando una expresión lambda.

```
var resultado = secuencia.OrderByDescending(elemento => clave);
```

Reto operador OrderBy

Utilizando el operador OrderBy retorna todos los elementos que sean de la categoría de Java ordenados por nombre.

Solución

```
1 reference
public IEnumerable<Book> GetBooksSortAsc(){
    return lstBooks.Where(
        book => (book.Categories ?? Array.Empty<string>()).Contains("Java"))
        .OrderBy(book => book.Title);
}
```

Reto operador OrderByDescending

Utilizando el operador OrderByDescending retorna los libros que tengan más de 450 páginas, ordenados por número de páginas en forma descendente.

Solución

```
public IEnumerable<Book> GetBooksSortDsc(){
    return lstBooks.Where(
        book => book.PageCount > 450)
        .OrderByDescending(book => book.PageCount);
}
```

3.2.1.5 Operador skip y take en LINQ

En LINQ en C#, los operadores **Take** y **Skip** son utilizados para realizar operaciones de filtrado y particionado en las secuencias de datos. Aquí está la descripción de cada uno de estos operadores:

- Operador **Take**:

- **Take** se utiliza para obtener los primeros **n** elementos de una secuencia.

- Toma una secuencia de datos y devuelve una nueva secuencia que contiene únicamente los primeros **n** elementos de la secuencia original.
- La sintaxis básica es: **secuencia.Take(n)**
- Donde **secuencia** es la secuencia de datos original y **n** es el número de elementos que se desean tomar.
- Por ejemplo, **frutas.Take(5)** devuelve una nueva secuencia que contiene los primeros 5 elementos de la secuencia **frutas**.
- Operador **Skip**:
 - **Skip** se utiliza para omitir los primeros **n** elementos de una secuencia.
 - Toma una secuencia de datos y devuelve una nueva secuencia que excluye los primeros **n** elementos de la secuencia original.
 - La sintaxis básica es: **secuencia.Skip(n)**
 - Donde **secuencia** es la secuencia de datos original y **n** es el número de elementos que se desean omitir.
 - Por ejemplo, **frutas.Skip(3)** devuelve una nueva secuencia que excluye los primeros 3 elementos de la secuencia **frutas**.

Ambos operadores (**Take** y **Skip**) son útiles cuando necesitas particionar o dividir una secuencia en partes más pequeñas. Puedes combinar estos operadores con otros operadores de LINQ para realizar consultas y operaciones más complejas en tus datos.

Es importante tener en cuenta que los operadores **Take** y **Skip** preservan el orden de los elementos en la secuencia original. Esto significa que los primeros **n** elementos se tomarán o se omitirán en función de su posición original en la secuencia.

Reto operador Take

Utilizando el operador Take selecciona los primeros 3 libros con fecha de publicación más reciente que estén categorizados en Java.

Solución

```
1 reference
public I Enumerable<Book> GetBooksTake(){
    return lstBooks
        .Where(book => (book.Categories ?? Array.Empty<string>()).Contains("Java"))
        .OrderByDescending(book => book.PublishedDate).Take(3);
}
```

```
//Muestra los ultimos tres libro
0 references
public I Enumerable<Book> GetBooksTakeLast(){
    return lstBooks
        .Where(book => (book.Categories ?? Array.Empty<string>()).Contains("Java"))
        .OrderBy(book => book.PublishedDate).TakeLast(3);

}
```

Reto operador Skip

Utilizando el operador Skip
selecciona el tercer y cuarto libro de
los que tengan más de 400 páginas.

Solución

```
1 reference
public I Enumerable<Book> GetBooksSkipTercerYCuarto(){
    return lstBooks
        .Where(book => book.PageCount > 400)
        .Take(4)
        .Skip(2);
}
```

3.2.1.6 Operador select en Linq

El operador **Select** en LINQ se utiliza para transformar o proyectar los elementos de una secuencia en una nueva secuencia. Permite seleccionar y devolver solo las propiedades o transformaciones específicas de los elementos originales.

Aquí hay una descripción del operador **Select** en LINQ en C#:

- El operador **Select** se utiliza para aplicar una transformación a cada elemento de una secuencia y proyectar los resultados en una nueva secuencia.
- La sintaxis básica es: **secuencia.Select(transformacion)**.
- Donde **secuencia** es la secuencia de datos original y **transformacion** es una función o expresión que especifica cómo se debe transformar cada elemento.
- La función **transformacion** toma un elemento de la secuencia original y devuelve el resultado transformado.

- Por ejemplo, puedes utilizar **Select** para extraer una propiedad específica de cada elemento de una lista, aplicar una operación matemática o incluso crear nuevos objetos basados en los elementos originales.

Reto operador selección dinámica

Utilizando el operador Select selecciona el título y el número de páginas de los primeros 3 libros de la colección.

Solución

```
0 references
public IEnumerable<Book> GetBooksSelect(){
    return lstBooks.Take(3)
        .Select(book => new Book{ Title = book.Title, PageCount = book.PageCount });
}
```

3.2.1.7 Operador LongCount y Count en Linq

En LINQ, existen dos operadores relacionados con el conteo de elementos en una secuencia: **Count** y **LongCount**. Ambos operadores se utilizan para obtener el número de elementos en una secuencia, pero hay una diferencia clave entre ellos en cuanto al tipo de retorno:

- Operador **Count**:
 - El operador **Count** se utiliza para contar el número de elementos en una secuencia y devuelve un valor de tipo **int**.
 - La sintaxis básica es: **secuencia.Count()**.
 - Donde **secuencia** es la secuencia de datos para la cual deseas obtener el conteo.
 - Por ejemplo, **frutas.Count()** devuelve el número de elementos en la secuencia **frutas** como un valor entero.
- Operador **LongCount**:
 - El operador **LongCount** se utiliza para contar el número de elementos en una secuencia y devuelve un valor de tipo **long**.
 - La sintaxis básica es: **secuencia.LongCount()**.
 - Al igual que con **Count**, **secuencia** representa la secuencia de datos para la cual deseas obtener el conteo.
 - **LongCount** es útil cuando la secuencia puede contener un número muy grande de elementos que no se pueden representar con un valor entero (**int**).

- Por ejemplo, `frutas.LongCount()` devuelve el número de elementos en la secuencia `frutas` como un valor `long`.

```
var frutas = new List<string> { "Manzana", "Plátano", "Naranja" };

int count = frutas.Count(); // 3
long longCount = frutas.LongCount(); // 3
```

Reto operador Count

Utilizando el operador Count,
retorna el número de libros que
tengan entre 200 y 500 páginas.

Solución

```
1 reference
public int GetBooksCount(){
    return lstBooks
        .Where(book => book.PageCount >= 200 && book.PageCount <= 500)
        .Count();
}

0 references
public long GetBooksLongCount(){
    return lstBooks
        .Where(book => book.PageCount >= 200 && book.PageCount <= 500)
        .LongCount();
}
```

Solución Ideal

```

1 reference
public int GetBooksCount(){
    return lstBooks
    .Count(book => book.PageCount >= 200 && book.PageCount <= 500);
}
0 references
public long GetBooksLongCount(){
    return lstBooks
    .LongCount(book => book.PageCount >= 200 && book.PageCount <= 500);
}

```

3.2.1.8 Operador Min y Max en Linq

Los operadores **Min** y **Max** en LINQ se utilizan para obtener el valor mínimo y máximo de una secuencia de datos, respectivamente. Estos operadores se aplican a secuencias numéricas o secuencias de elementos que pueden ser comparados entre sí.

Aquí tienes una descripción de cada uno de estos operadores:

- Operador **Min**:
 - El operador **Min** se utiliza para encontrar el valor mínimo en una secuencia de datos.
 - Toma una secuencia de elementos y devuelve el valor mínimo de acuerdo con el criterio de comparación.
 - La sintaxis básica es: **secuencia.Min()**
 - Donde **secuencia** es la secuencia de datos en la que deseas encontrar el valor mínimo.
 - Por ejemplo, **numeros.Min()** devuelve el valor mínimo de la secuencia **numeros**.
- Operador **Max**:
 - El operador **Max** se utiliza para encontrar el valor máximo en una secuencia de datos.
 - Toma una secuencia de elementos y devuelve el valor máximo de acuerdo con el criterio de comparación.
 - La sintaxis básica es: **secuencia.Max()**
 - Donde **secuencia** es la secuencia de datos en la que deseas encontrar el valor máximo.
 - Por ejemplo, **numeros.Max()** devuelve el valor máximo de la secuencia **numeros**.

Es importante destacar que estos operadores (**Min** y **Max**) pueden utilizarse en secuencias de datos numéricos, como enteros o decimales. También se pueden aplicar a secuencias de elementos que implementen la interfaz **IComparable**, lo que permite la comparación entre ellos.

Aquí tienes un ejemplo de uso de los operadores **Min** y **Max**:

```
var numeros = new List<int> { 5, 2, 9, 1, 7 };

int min = numeros.Min(); // 1
int max = numeros.Max(); // 9
```

```
var personas = new List<Persona>
{
    new Persona { Nombre = "Juan", Edad = 25 },
    new Persona { Nombre = "Maria", Edad = 30 },
    new Persona { Nombre = "Pedro", Edad = 28 }
};

int minEdad = personas.Min(p => p.Edad); // 25 (Obtiene la edad mínima)
int maxEdad = personas.Max(p => p.Edad); // 30 (Obtiene la edad máxima)
```

Reto operador Min

Utilizando el operador Min, retorna la menor fecha de publicación de la lista de libros.

```
1 reference
public DateTime GetBooksDatePublishMinor(){
    return lstBooks.Min(book => book.PublishedDate);
}
```

```
1 reference
public IEnumerable<Book> GetBooksListDatePublishMinor(){
    var fechaPublicacionMinima = lstBooks.Min(libro => libro.PublishedDate);
    return lstBooks.Where(libro => libro.PublishedDate == fechaPublicacionMinima);
}
```

Reto operador Max

Utilizando el operador Max, retorna la cantidad de páginas del libro con mayor número de páginas en la colección.

Solución

```
1 reference
public DateTime GetBooksDatePublishMax(){
    return lstBooks.Max(book => book.PublishedDate);
}
```

2.3.1.9 Operador Min y Max en Linq

Reto operador MinBy

Retorna el libro que tenga la menor cantidad de páginas mayor a 0.

Solucion

```
1 reference
public Book GetBooksMayorCero(){
    return lstBooks.Where(book => book.PageCount>0)
        .MinBy(myBook => myBook.PageCount) ?? new Book();
}
```

Reto operador MaxBy

Retorna el libro con la fecha de publicación más reciente.

```
0 references
public Book GetBooksFechaReciente(){
    return lstBooks
        .MaxBy(myBook => myBook.PublishedDate) ?? new Book();
}
```

3.2.1.10 Operador sum y Aggregate en Linq

En C#, tanto **Sum** como **Aggregate** son funciones utilizadas en el contexto de LINQ para procesar colecciones de datos.

- **Sum()**: La función **Sum()** se utiliza para calcular la suma de los valores numéricos en una colección. Puede ser aplicada a colecciones de tipos numéricos como **int**, **double**, **decimal**, etc.
- **Aggregate()**: La función **Aggregate()** se utiliza para combinar los elementos de una colección aplicando una función acumuladora personalizada. A diferencia de **Sum()**, **Aggregate()** te permite especificar una lógica de agregación más compleja.

Reto operador Sum

Retorna la suma de la cantidad de páginas, de todos los libros que tengan entre 0 y 500.

```
1 reference
public int GetBooksOperadorSum(){
    return lstBooks.Where(book => book.PageCount>0 && book.PageCount<=500)
        .Sum(myBook => myBook.PageCount);
}
```

Reto operador Aggregate

Retorna el título de los libros que tienen fecha de publicación posterior a 2015.

```
public string TitulosDeLibroDespues2015(){
    return lstBooks.Where(book => book.PublishedDate.Year >2015)
        .Aggregate("",(tituloLibros,next) =>{
            if (tituloLibros != string.Empty){
                tituloLibros += " - " + next.Title;
            }else{
                tituloLibros += next.Title;
            }
            return tituloLibros;
        });
}
```

3.2.1.11 operador average en Linq

En LINQ (Language Integrated Query) en C#, el método **Average()** se utiliza para calcular el valor promedio de una secuencia numérica. Este método está disponible en la clase **Enumerable** y puede ser aplicado a colecciones que contengan elementos numéricos, como **int**, **double**, **decimal**, etc.

Es importante tener en cuenta que, si la colección está vacía, **Average()** lanzará una excepción **InvalidOperationException**. Por lo tanto, es recomendable verificar si la colección contiene elementos antes de utilizar **Average()**, o utilizar la sobrecarga de **Average()** que permite proporcionar un valor predeterminado en caso de que la colección esté vacía.

Reto operador Average

Utilizando el operador Average, retorna el promedio de caracteres que tienen los títulos de la colección.

```
0 references
public double GetBooksAverage(){
    return lstBooks.Average(book => (book.Title ?? string.Empty).Length);
}
```

3.2.1.12 GroupBy en Linq

La cláusula **group by** en LINQ se utiliza para agrupar elementos de una colección según un criterio específico. Permite crear grupos basados en una o más propiedades de los elementos y realizar operaciones en cada grupo. La cláusula **group by** se combina comúnmente con otras cláusulas de consulta LINQ, como **select**, **where**, etc.

Reto operador GroupBy

Retorna todos los libros que fueron publicados a partir del 2000, agrupados por año.

```
1 reference
public IEnumerable<IGrouping<int, Book>> LibrosAgrupados(){
    return lstBooks.Where(book => book.PublishedDate.Year >= 2000)
        .GroupBy(myBook => myBook.PublishedDate.Year);
}
```

3.2.1.13 LookUp Linq

En LINQ (Language Integrated Query) en C#, la clase **Lookup** se utiliza para crear una estructura de datos que permite buscar valores basados en una clave. La clase **Lookup** proporciona una funcionalidad similar a un diccionario, pero permite múltiples valores asociados a una misma clave.

Reto operador Lookup

Retorna un diccionario usando Lookup que permita consultar los libros de acuerdo a la letra con la que inicia el título del libro

```
1 reference
public ILookup<char, Book> DiccionarioDeLibros(){
    return lstBooks.ToLookup(book => book.Title?[0] ?? default(char), book => book);
}
```

3.2.1.14 Join Linq

La cláusula **join** en LINQ se utiliza para combinar dos fuentes de datos diferentes en función de una clave común. Se pueden realizar operaciones de combinación similares a las operaciones de **JOIN** en SQL. La cláusula **join** se combina comúnmente con las cláusulas **on** y **equals** para especificar la condición de combinación.

Reto operador Join

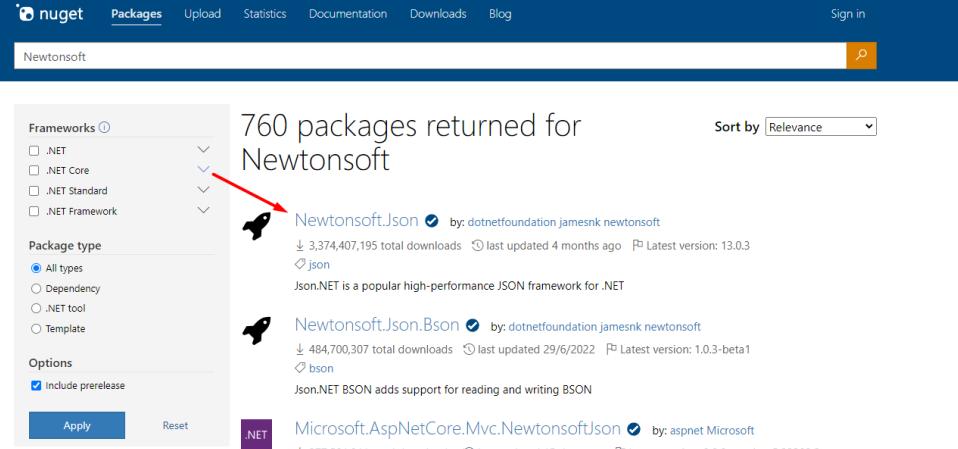
Obtén una colección que tenga todos los libros con más de 500 páginas y otra que contenga los libros publicados después del 2005. Utilizando la cláusula Join, retorna los libros que estén en ambas colecciones.

```
1 reference
public IEnumerable<Book> LibrosDespues2005ConMas500Pag(){
    var LibrosDespues2005 = lstBooks.Where(b => b.PublishedDate.Year > 2005);
    var LibrosConMasde500Pag = lstBooks.Where(b => b.PageCount > 500);
    return LibrosDespues2005.Join(LibrosConMasde500Pag, p => p.Title, x => x.Title,(p,x) => p);
}
```

3.3. Persistencia de Datos JSON

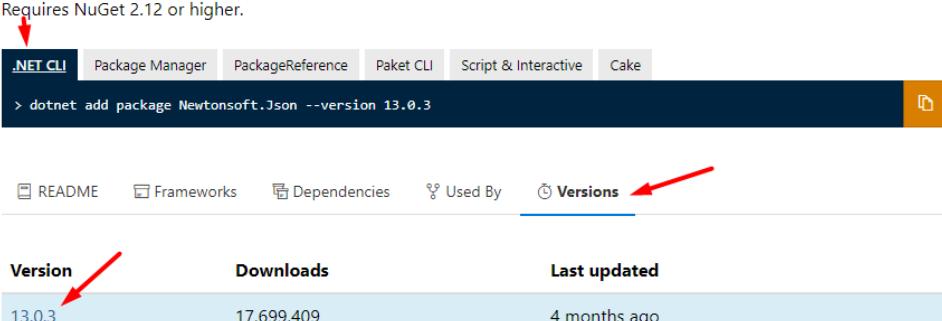
1. Ingresar a la URL <https://www.nuget.org/>
2. En el buscador de la página web buscar el Nuget Newtonsoft. Hacer clic sobre el vínculo

Newtonsoft.Json  by: dotnetfoundation jamesnk newtonsoft



The screenshot shows the NuGet search interface. The search bar at the top contains 'Newtonsoft'. Below it, a summary says '760 packages returned for Newtonsoft'. On the left, there are filters for 'Frameworks' (including .NET, .NET Core, .NET Standard, and .NET Framework), 'Package type' (All types selected), and 'Options' (Include prerelease checked). The main list shows three packages: 'Newtonsoft.Json' (version 13.0.3), 'Newtonsoft.Json.Bson' (version 1.0.3-beta1), and 'Microsoft.AspNetCore.Mvc.NewtonsoftJson' (version 8.0.0-preview5.23302.2). A red arrow points from the text 'Hacer clic sobre el vínculo' to the 'Newtonsoft.Json' link.

3. Seleccionar la versión a instalar y el modo de instalación



The screenshot shows the NuGet package details page for 'Newtonsoft.Json'. At the top, it says 'Requires NuGet 2.12 or higher.' Below that is a navigation bar with tabs: '.NET CLI' (selected), 'Package Manager', 'PackageReference', 'Paket CLI', 'Script & Interactive', and 'Cake'. Underneath is a command line input field containing '> dotnet add package Newtonsoft.Json --version 13.0.3'. The main content area shows the package's version history. A red arrow points from the text 'Hacer clic en el botón de' to the 'Copy' icon next to the 'Versions' tab. Another red arrow points from the text 'selección' to the '13.0.3' version row.

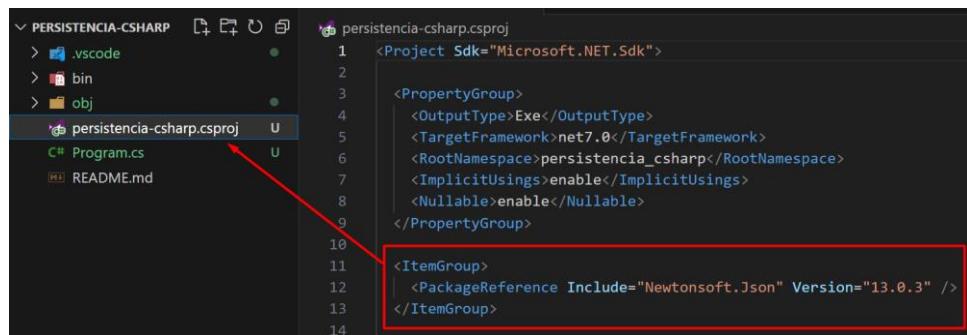
Version	Downloads	Last updated
13.0.3	17,699,409	4 months ago

4. Hacer clic en el botón de  para copiar el comando de instalación y pegar el comando en el terminal. Nota. Se debe estar ubicado en la carpeta del proyecto donde se desea instalar el componente.



The screenshot shows the VS Code terminal window. It displays the command '\$ dotnet add package Newtonsoft.Json --version 13.0.3' which was copied from the NuGet package details page. The terminal is located in a dark-themed workspace with tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (selected), and COMMENTS.

Cuando finaliza la instalación se puede verificar en el archivo de configuración del Proyecto el paquete instalado. Ver imagen.



```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net7.0</TargetFramework>
    <RootNamespace>persistencia_csharp</RootNamespace>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Newtonsoft.Json" Version="13.0.3" />
  </ItemGroup>

```