C#

- 🔑 **Introducción**
- 🔑 **Objetivos**
- 🎖️ **Fundamentos C#**
  - 🟫 Conociendo Lenguaje C#
  - 🟫 Arquitectura .NET Core
  - 🟫 Lenguajes soportados por .Net Core
  - 🟫 Ventajas y Desventajas de .NET Core
    - Ventajas de utilizar .NET Core
      - 📖 Desventajas de utilizar .NET Core
  - 🟫 Configuración del entorno de desarrollo
    - 📖 Instalación Windows
    - 📖 Instalación Linux
      - 🏷️ Instalación del SDK
      - 🏷️ Instalación de la instancia en tiempo de ejecución
      - 🏷️ Validación de la Instalación dotnet
  - 🟫 Creacion de proyectos en C#
    - 📖 Windows

  Abriendo el proyecto (Windows y Linux)
- 🎖️ **Programación Básica C#**
  - 🟫 Variables y constantes
    - ⌨️ Variables
    - ⌨️ Constantes
  - 🟫 Tipos de Datos y Conversiones

    Tipos numéricos enteros
    - 📖 Tipos de punto flotante
      - 🏷️ Literales reales
      - 🏷️ Propiedad NumberFormatInfo.CurrencySymbol
      - 🏷️ Propiedad NumberFormatInfo.CurrencyDecimalDigits
      - 🏷️ Propiedad NumberFormatInfo.CurrencyDecimalSeparator
      - 🏷️ Especificador de formato decimal (D)
      - 🏷️ Especificador de formato de punto fijo (F)
  - 🟫 Palabras claves
  - 🟫 Palabras clave contextuales
  - 🟫 Entrada y salida de datos
    - 📖 Salida
      - `Console.WriteLine()`
        - 🏷️ Ejemplo:
      - `Console.WriteLine` con Interpolación de Cadenas (`$""`)
      - `Console.Write()`
        - 🏷️ Ejemplo:
      - Diferencia Entre `Console.WriteLine` y `Console.Write`
        - 🏷️ Ejemplo Comparativo:
        - 🏷️ **Salida en consola:**
    - 📖 Entrada
      - **Entrada desde Consola (** `Console.ReadLine` , `Console.ReadKey` , `Console.Read` **)**
      - `Console.ReadLine()` → Lee una línea completa de entrada
      - `Console.ReadKey()` → Captura una tecla sin necesidad de presionar Enter
      - `Console.Read()` → Lee un solo carácter como entero (`int`)
  - 🟫 Caracteres de escape
    - 📖 Lista de Caracteres de Escape en C#
      - 2️⃣ `\t` - Tabulación (Espacios extra)
      - 3️⃣ `\\` - Barra invertida
      - 4️⃣ `\'` - Comilla simple y `\"` - Comilla doble

- 5️⃣ `\b` - Retroceso (Elimina el carácter anterior)
- 6️⃣ `\r` - Retorno de carro (Reemplaza lo anterior en la misma línea)
- 7️⃣ `\0` - Carácter Nulo (Null)
- 8️⃣ `\uXXXX` - Unicode
  - 🏷️ **Tabla de Caracteres Unicode Comunes**
  - 🏷️ **Símbolos Especiales y Matemáticos**
  - 🏷️ **Letras Griegas y Científicas**
  - 🏷️ **Caras y Emoticones Unicode**
  - 🏷️ **Ejemplo de Uso en C#**
- 🏷️ Alternativa: **Cadenas Verbatim (`@""`)**

📓 Conversión de tipos de datos

📖 Conversiones implícitas

- 🏷️ **Ejemplo de Conversión Implícita**
- 🏷️ **Otras Conversión Implícitas Comunes**

📖 Conversiones Explícitas

- 🏷️ **Ejemplo de Conversión Explícita**
- 🏷️ **Otras Conversiones Explícitas Comunes**
- 🔹 **3. Métodos de Conversión en C# (`Convert` y `Parse`)**
  - 🏷️ `Convert.ToTipo()` → **Conversión Segura**
  - 🏷️ **Diferencia con Casting:**
  - 🏷️ `Parse()` → **Para convertir cadenas en números**
  - 🏷️ `TryParse()` → **Evita Errores**
  - 🏷️ **Diferencia con `Parse()`:**
- 🔹 **Resumen de Conversión en C#**

📓 Operadores Matemáticos

- 📌 **Orden de Precedencia en C#**
- 📖 **Operadores de Asignación**
- 📖 **Operadores de Incremento y Decremento**
- 📖 **Operadores Matemáticos en `Math`**

📓 Operadores de Comparación

- 📖 **Lista de Operadores de Comparación**

📓 Operadores Lógicos

- 📖 **Lista de Operadores Lógicos**

📓 Estructuras de Control: Condicionales

📖 If

1. `if` **Simple**
2. `if` **-** `else`
3. `if` **-** `else if` **-** `else` **(Múltiples Condiciones)**
4. `if` **Anidado**
5. `if` **con Operadores Lógicos**
6. `if` en una sola línea (Operador Ternario `? :`)

📖 Switch

- 🏷️ **Sintaxis del `switch`**
- 🏷️ **Uso de `switch` con `when` (Filtros en C# 7.0+)**

📓 Estructuras de Iterativas: Ciclos

- 📖 **Tipos de Bucles en C#**
- 📖 Bucle `for` (Controlado por un Contador)
  - 🖥️ **Sintaxis**
  - 📝 **Explicación de sus partes**
- 📖 Bucle `while` (Ejecución Basada en una Condición)
  - 🖥️ **Sintaxis**
  - 📝 **Explicación**
- 📖 Bucle `do..while` (Ejecución Basada en una Condición)

  Características principales:

- Ejemplo
- **Diferencia entre `while` y `do..while`**
- 📖 Bucle `foreach` (Ejecución Basada en una Condición)
- 📖 **Diferencias entre `for` y `foreach`**
- 📖 **Cuándo usar `foreach`**

Arreglos (Arrays) en C#

- 📖 **Declaración y Creación de un Array**
  - **Declaración e Inicialización Simultánea**
- 📖 **Declaración con Tamaño Fijo e Inicialización Posterior**
- 📖 **Declaración con `new` y Elementos**
- 📖 **Acceder a los Elementos del Array**
- 📖 **Recorrer un Array**
  - 📝 **Usando `for`**
  - 📝 **Usando `foreach` (Recomendado para solo lectura)**
- 📖 **Modificar un Array**
- 📖 **Arrays Multidimensionales**
  - 📝 **Array Bidimensional (Matriz)**
  - 📝 **Arrays Irregulares (Jagged Arrays)**
- 📖 **Métodos Útiles para Arrays**
  - **Ejemplo: Ordenar y Buscar en un Array**

Ejemplo

- 📖 **Conclusiones**

🥊 **Programación Avanzada C#**

Programación orientada a Objetos

- 📖 **Introducción a la Programación Orientada a Objetos (POO) en C#** 🚀
- 📖 **Conceptos Claves de POO**
- 📖 **Clases y Objetos**

Ejemplo: Definir una clase en C#

- 📖 **Encapsulamiento** 🔒

Ejemplo usando `private` y `public` con métodos `get` y `set`

- 📖 **Herencia** 👪
  - **Ejemplo: Herencia en C#**
- 📖 **Polimorfismo** 🎭
  - **Ejemplo: Sobreescritura de Métodos (`override`)**
- 📖 **Abstracción** 📦
  - **Ejemplo con Clases Abstractas**
- 📖 **Conclusión** 🏆

Listas

- 📖 Definición
- 📖 Constructores
- 📖 Propiedades
- 📖 Métodos

ArrayList

- 📖 **Características de `ArrayList`**
- 📖 Constructores
- 📖 Propiedades
- 📖 **Principales Métodos de `ArrayList`**
- 📖 **Diferencia entre `ArrayList` y `List<T>`**

HashSet en C#

- 📖 Propiedades
- 📖 **Principales Métodos de `HashSet<T>`**
- 📖 **Diferencias entre `HashSet<T>` y `List<T>`**
- 📖 **Características principales de `HashSet<T>`**

Expresiones Lambda

Qué es una expresión lambda (Lambda expression)

- 📖 Ejemplo de un Método Anónimo (Sin Lambda)
- 📖 **Expresión Lambda Equivalente**
- 📖 **Ejemplo con Múltiples Parámetros**
- 📖 Expresiones Lambda con Cuerpo de Bloque

Linq

- 📖 El problema
- 📖 **Qué no es LINQ?**
- 📖 **Beneficios de LINQ:**
- 📖 **Tipos de LINQ**
- 📖 **Sintaxis de LINQ**
- 📖 Consultas
  - 🗝️ **Clase `Persona`**
  - 🗝️ **Clase `Estudiante`**
  - 🗝️ **Clase `Producto`**
  - 🗝️ **Clase `Empleado`**
  - 📑 **¿Qué es una consulta y cuál es su propósito?**
  - 📑 **¿Qué puede hacer una consulta?**
    - 🗝️ **Filtrar y organizar datos sin modificar los elementos originales**
    - 🗝️ **Transformar los datos en otro tipo**
    - 🗝️ **Obtener un único valor de la fuente de datos**
    - 🗝️ **Filtrar Datos en una Lista de Personas**
    - 🗝️ **Seleccionar Solo los Nombres de los Estudiantes**
    - 🗝️ **3. Ordenar Productos por Precio Descendente**
    - 🗝️ **Contar Cuántos Números Son Pares en un Array**
    - 🗝️ **Agrupar Empleados por Departamento**
  - 📖 **¿Expresión de consulta en C#?**
    - 🗝️ **Características de una expresión de consulta**
  - 📖 **Uso de diferentes cláusulas en expresiones de consulta**
    - 🗝️ **Filtrar elementos con `where`**
    - 🗝️ **Ordenar elementos con `orderby`**
    - 🗝️ **Agrupar elementos con `group`**
    - 🗝️ **Declarar variables dentro de la consulta con `let`**
    - 🗝️ **Usar `join` para unir dos colecciones**
  - 📖 **¿Qué es una variable de consulta en LINQ?**
  - 📖 Variables que No Son de Consulta en LINQ
  - 📖 Iniciar una expresión de consulta

    Cláusula `from`
  - 📖 **Finalizar una Expresión de Consulta en LINQ**
    1. Usando `group` para Agrupar por la Primera Letra del Nombre de un País
    2. **Usando `select` para Devolver una Lista de Datos**
  - 📖 **Transformación de Datos con `select` en LINQ (Proyección)**
    1. **Proyección a Tipos Anónimos**
    2. **Proyección con Cálculo en `select`**
    3. **Proyección desde una Lista de Productos**
    4. **Proyección desde una Lista de Ciudades**
  - 📖 **Uso de `into` en LINQ para Continuaciones de Consulta**
  - 📖 **Subconsultas en LINQ**
    1. **Subconsulta para Filtrar Países Basado en su Ciudad más Poblada**
    2. **Subconsulta para Obtener la Ciudad más Poblada de Cada País**
    3. **Subconsulta para Obtener Categorías con Productos Caros**
  - 📖 Operaciones set [C#]
    1. **`Distinct` y `DistinctBy` (Eliminar Duplicados)**
    2. **`Except` y `ExceptBy` (Diferencia de Conjuntos)**
    3. **`Intersect` y `IntersectBy` (Intersección de Conjuntos)**

4️⃣ `Union` y `UnionBy` (Unión de Conjuntos)

C#

🔑 Introducción

Este Skill está orientado en la creación de diversos tipos de aplicaciones seguras y sólidas de escritorio, web y móviles, mediante la utilización del lenguaje de programación C#, para que se ejecutan en una plataforma de código abierto (.NET).

🔑 Objetivos

🔓 Construir proyectos de consola en C#, mediante la práctica, con el fin de adquirir los fundamentos del lenguaje de programación y comenzar a utilizar el Framework .NET Core

🔓 Aprender los diferentes conceptos del framework (.NET Core), que es una versión modular y ligera de (.NET Framework) a través de la implementación de distintos ejercicios.

🔓 Construir programas usando los conceptos de la programación orientada a objetos y su implementación en C#.

📮 Fundamentos C#

📓 Conociendo Lenguaje C#

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

Recurso oficial : https://learn.microsoft.com/es-es/dotnet/csharp/tour-of-csharp/

📓 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.

🟪 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.

🟪 Ventajas y Desventajas de .NET Core

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.

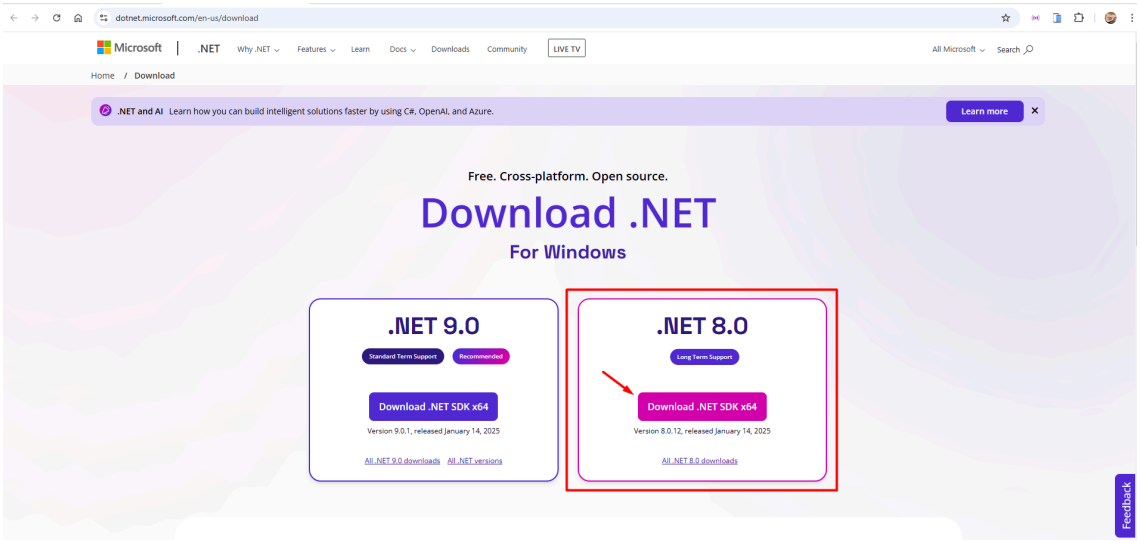
👾 Configuración del entorno de desarrollo

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

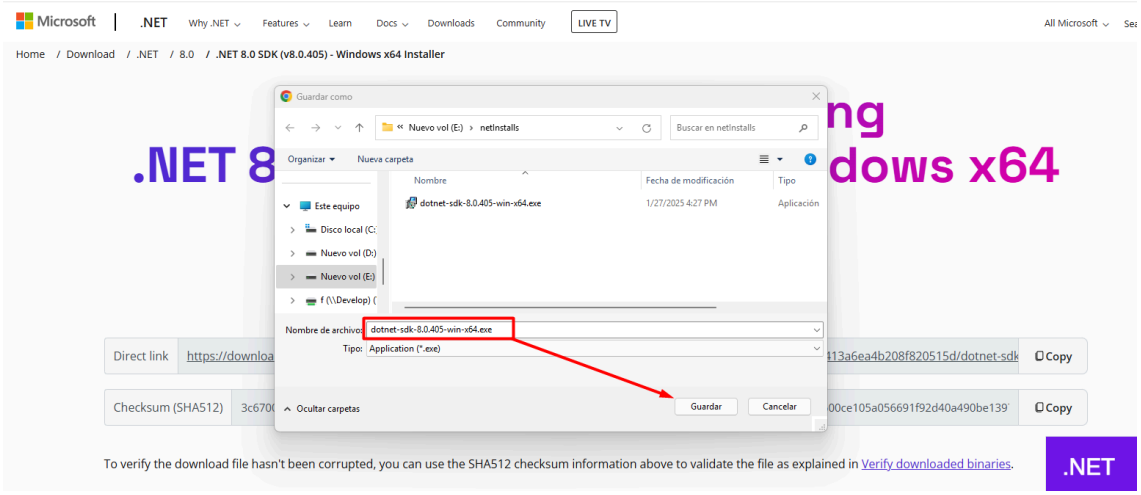
- Sistema operativo compatible: .NET Core es compatible con Windows, macOS y Linux. Asegúrate de tener un sistema operativo compatible instalado en tu máquina.
- SDK de .NET Core: Debes descargar e instalar el SDK (Software Development Kit) de .NET Core correspondiente a tu sistema operativo desde el 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.

📖 Instalación Windows

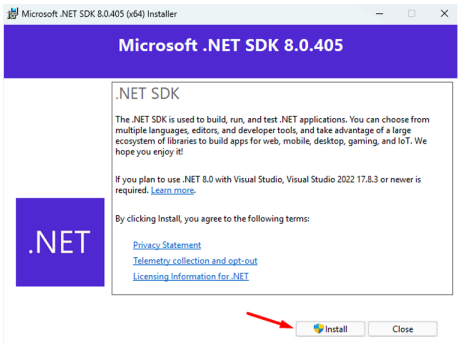
1. Ingrese a la Url oficial https://dotnet.microsoft.com/en-us/download y descargue larsión LTS 8.0

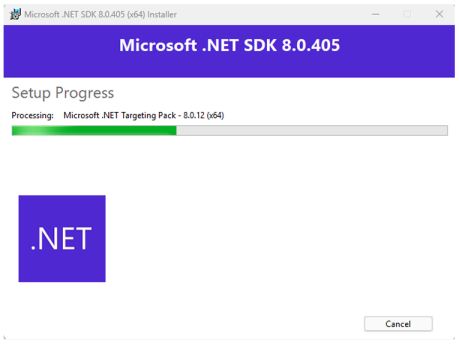


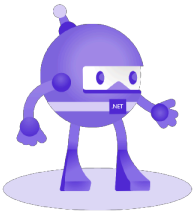
2. En la ventana de descarga seleccione la carpeta de descarga y haga clic en guardar

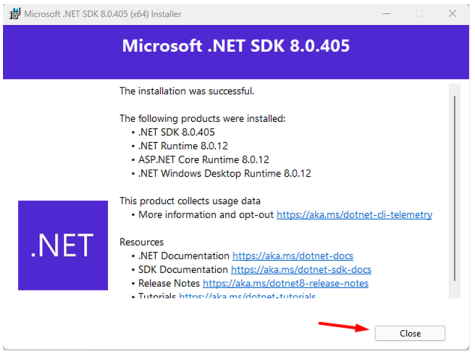


3. Hacer doble clic sobre el instalador y siga los pasos del asistente









4. Abra la terminal del sistema operativo windows (cmd o powershell)
5. Ingrese el comando **dotnet --info** A continuación se observa el resultado de la ejecucion donde se valida la instalacion del SDK y las herramientas necesarias para la creación de proyecto usando c#.

```
PS C:\Users\developer> dotnet --info
SDK DE .NET:
 Version:           8.0.405
 Commit:            fb1830d421
 Workload version:  8.0.400-manifests.87fdb0b5
 MSBuild version:   17.11.9+a69bbaaf5

Entorno de tiempo de ejecución:
 OS Name:     Windows
 OS Version:  10.0.26100
 OS Platform: Windows
 RID:         win-x64
 Base Path:   C:\Program Files\dotnet\sdk\8.0.405\

Cargas de trabajo de .NET instaladas:
Configurado para usar loose manifests al instalar nuevos manifiestos.
No hay cargas de trabajo instaladas para mostrar.

Host:
  Version:      8.0.12
  Architecture: x64
  Commit:       89ef51c5d8

.NET SDKs installed:
  8.0.403 [C:\Program Files\dotnet\sdk]
  8.0.405 [C:\Program Files\dotnet\sdk]
```

```
.NET runtimes installed:
  Microsoft.AspNetCore.App 8.0.10 [C:\Program
Files\dotnet\shared\Microsoft.AspNetCore.App]
  Microsoft.AspNetCore.App 8.0.12 [C:\Program
Files\dotnet\shared\Microsoft.AspNetCore.App]
  Microsoft.NETCore.App 6.0.16 [C:\Program
Files\dotnet\shared\Microsoft.NETCore.App]
  Microsoft.NETCore.App 8.0.10 [C:\Program
Files\dotnet\shared\Microsoft.NETCore.App]
  Microsoft.NETCore.App 8.0.12 [C:\Program
Files\dotnet\shared\Microsoft.NETCore.App]
  Microsoft.WindowsDesktop.App 8.0.10 [C:\Program
Files\dotnet\shared\Microsoft.WindowsDesktop.App]
  Microsoft.WindowsDesktop.App 8.0.12 [C:\Program
Files\dotnet\shared\Microsoft.WindowsDesktop.App]

Other architectures found:
  x86   [C:\Program Files (x86)\dotnet]
    registered at
[HKLM\SOFTWARE\dotnet\Setup\InstalledVersions\x86\InstallLocation]

Environment variables:
  Not set

global.json file:
  Not found

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

Download .NET:
  https://aka.ms/dotnet/download
```

📖 Instalación Linux

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

```
jjpardo@johlver-virtual-machine:~$ sudo apt remove 'dotnet*'
[sudo] password for jjpardo:
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
Note, selecting 'dotnet-runtime-dbg-8.0' for glob 'dotnet*'
Note, selecting 'dotnet-sdk-8.0-source-built-artifacts' for glob 'dotnet*'
Note, selecting 'dotnet-hostfxr-3.1' for glob 'dotnet*'
Note, selecting 'dotnet-hostfxr-6.0' for glob 'dotnet*'
Note, selecting 'dotnet-hostfxr-7.0' for glob 'dotnet*'
Note, selecting 'dotnet-hostfxr-8.0' for glob 'dotnet*'
Note, selecting 'dotnet-hostfxr-9.0' for glob 'dotnet*'
Note, selecting 'dotnet-sdk-3.1' for glob 'dotnet*'
```

```
Note, selecting 'dotnet-sdk-6.0' for glob 'dotnet*'
Note, selecting 'dotnet-sdk-7.0' for glob 'dotnet*'
Note, selecting 'dotnet-sdk-8.0' for glob 'dotnet*'
Note, selecting 'dotnet-sdk-9.0' for glob 'dotnet*'
Note, selecting 'dotnet-targeting-pack-3.1' for glob 'dotnet*'
Note, selecting 'dotnet6' for glob 'dotnet*'
Note, selecting 'dotnet7' for glob 'dotnet*'
Note, selecting 'dotnet8' for glob 'dotnet*'
Note, selecting 'dotnet-targeting-pack-6.0' for glob 'dotnet*'
Note, selecting 'dotnet-targeting-pack-7.0' for glob 'dotnet*'
Note, selecting 'dotnet-targeting-pack-8.0' for glob 'dotnet*'
Note, selecting 'dotnet-targeting-pack-9.0' for glob 'dotnet*'
Note, selecting 'dotnet-nightly' for glob 'dotnet*'
Note, selecting 'dotnet-sdk-dbg-8.0' for glob 'dotnet*'
Note, selecting 'dotnet-templates-6.0' for glob 'dotnet*'
Note, selecting 'dotnet-templates-7.0' for glob 'dotnet*'
Note, selecting 'dotnet-templates-8.0' for glob 'dotnet*'
Note, selecting 'dotnet-host-7.0' for glob 'dotnet*'
Note, selecting 'dotnet-host-8.0' for glob 'dotnet*'
Note, selecting 'dotnet-host' for glob 'dotnet*'
Note, selecting 'dotnet-runtime-deps-3.1' for glob 'dotnet*'
Note, selecting 'dotnet-runtime-deps-6.0' for glob 'dotnet*'
Note, selecting 'dotnet-runtime-deps-7.0' for glob 'dotnet*'
Note, selecting 'dotnet-runtime-deps-8.0' for glob 'dotnet*'
Note, selecting 'dotnet-runtime-deps-9.0' for glob 'dotnet*'
Note, selecting 'dotnet' for glob 'dotnet*'
Note, selecting 'dotnet-sdk-6.0-source-built-artifacts' for glob 'dotnet*'
Note, selecting 'dotnet-apphost-pack-3.1' for glob 'dotnet*'
Note, selecting 'dotnet-apphost-pack-6.0' for glob 'dotnet*'
Note, selecting 'dotnet-apphost-pack-7.0' for glob 'dotnet*'
Note, selecting 'dotnet-apphost-pack-8.0' for glob 'dotnet*'
Note, selecting 'dotnet-apphost-pack-9.0' for glob 'dotnet*'
Note, selecting 'dotnet-sdk-7.0-source-built-artifacts' for glob 'dotnet*'
Note, selecting 'dotnet-runtime-3.1' for glob 'dotnet*'
Note, selecting 'dotnet-runtime-6.0' for glob 'dotnet*'
Note, selecting 'dotnet-runtime-7.0' for glob 'dotnet*'
Note, selecting 'dotnet-runtime-8.0' for glob 'dotnet*'
Note, selecting 'dotnet-runtime-9.0' for glob 'dotnet*'
Package 'dotnet' is not installed, so not removed
Package 'dotnet-nightly' is not installed, so not removed
Package 'dotnet-targeting-pack-3.1' is not installed, so not removed
Package 'dotnet-apphost-pack-6.0' is not installed, so not removed
Package 'dotnet-host-8.0' 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-sdk-6.0-source-built-artifacts' is not installed, so not
removed
Package 'dotnet-sdk-8.0-source-built-artifacts' is not installed, so not
removed
Package 'dotnet-targeting-pack-6.0' is not installed, so not removed
Package 'dotnet-templates-6.0' is not installed, so not removed
Package 'dotnet-templates-8.0' is not installed, so not removed
Package 'dotnet6' is not installed, so not removed
Package 'dotnet8' is not installed, so not removed
Package 'dotnet-apphost-pack-7.0' is not installed, so not removed
Package 'dotnet-host-7.0' is not installed, so not removed
```

```
Package 'dotnet-hostfxr-7.0' is not installed, so not removed
Package 'dotnet-runtime-7.0' is not installed, so not removed
Package 'dotnet-runtime-dbg-8.0' is not installed, so not removed
Package 'dotnet-sdk-7.0-source-built-artifacts' is not installed, so not
removed
Package 'dotnet-sdk-dbg-8.0' is not installed, so not removed
Package 'dotnet-targeting-pack-7.0' 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 'dotnet-runtime-deps-3.1' is not installed, so not removed
Package 'dotnet-hostfxr-3.1' is not installed, so not removed
Package 'dotnet-apphost-pack-3.1' is not installed, so not removed
Package 'dotnet-runtime-3.1' is not installed, so not removed
Package 'dotnet-sdk-3.1' is not installed, so not removed
Package 'dotnet-sdk-7.0' is not installed, so not removed
Package 'dotnet-runtime-deps-7.0' is not installed, so not removed
Package 'dotnet-runtime-deps-6.0' is not installed, so not removed
Package 'dotnet-sdk-6.0' is not installed, so not removed
Package 'dotnet-apphost-pack-9.0' is not installed, so not removed
Package 'dotnet-hostfxr-9.0' is not installed, so not removed
Package 'dotnet-runtime-9.0' is not installed, so not removed
Package 'dotnet-runtime-deps-9.0' is not installed, so not removed
Package 'dotnet-targeting-pack-9.0' is not installed, so not removed
Package 'dotnet-sdk-9.0' is not installed, so not removed
Package 'dotnet-runtime-deps-8.0' is not installed, so not removed
The following packages were automatically installed and are no longer
required:
  aspnetcore-targeting-pack-8.0 liblttng-ust-common1 liblttng-ust-ctl5
  liblttng-ust1 netstandard-targeting-pack-2.1
Use 'sudo apt autoremove' to remove them.
The following packages will be REMOVED:
  aspnetcore-runtime-8.0 dotnet-apphost-pack-8.0 dotnet-host
  dotnet-hostfxr-8.0 dotnet-runtime-8.0 dotnet-sdk-8.0
  dotnet-targeting-pack-8.0
0 upgraded, 0 newly installed, 7 to remove and 121 not upgraded.
After this operation, 542 MB disk space will be freed.
Do you want to continue? [Y/n] Y
(Reading database ... 279953 files and directories currently installed.)
Removing dotnet-sdk-8.0 (8.0.404-1) ...
Removing aspnetcore-runtime-8.0 (8.0.12-0ubuntu1~22.04.1) ...
Removing dotnet-apphost-pack-8.0 (8.0.12-0ubuntu1~22.04.1) ...
Removing dotnet-runtime-8.0 (8.0.12-0ubuntu1~22.04.1) ...
Removing dotnet-hostfxr-8.0 (8.0.11-1) ...
Removing dotnet-host (8.0.11-1) ...
Removing dotnet-targeting-pack-8.0 (8.0.12-0ubuntu1~22.04.1) ...
jjpardo@johlver-virtual-machine:~$
```

3. Ingresar el comando **sudo apt remove 'aspnetcore*'** para remover paquetes y servicios runtime de aspnetcore.

```
jjpardo@johlver-virtual-machine:~$ sudo apt remove 'aspnetcore*'
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
Note, selecting 'aspnetcore-runtime-3.1' for glob 'aspnetcore*'
```

```
Note, selecting 'aspnetcore-runtime-6.0' for glob 'aspnetcore*'
Note, selecting 'aspnetcore-runtime-7.0' for glob 'aspnetcore*'
Note, selecting 'aspnetcore-runtime-8.0' for glob 'aspnetcore*'
Note, selecting 'aspnetcore-runtime-9.0' for glob 'aspnetcore*'
Note, selecting 'aspnetcore-runtime-dbg-8.0' for glob 'aspnetcore*'
Note, selecting 'aspnetcore-targeting-pack-3.1' for glob 'aspnetcore*'
Note, selecting 'aspnetcore-targeting-pack-6.0' for glob 'aspnetcore*'
Note, selecting 'aspnetcore-targeting-pack-7.0' for glob 'aspnetcore*'
Note, selecting 'aspnetcore-targeting-pack-8.0' for glob 'aspnetcore*'
Note, selecting 'aspnetcore-targeting-pack-9.0' for glob 'aspnetcore*'
Package 'aspnetcore-targeting-pack-3.1' is not installed, so not removed
Package 'aspnetcore-runtime-6.0' is not installed, so not removed
Package 'aspnetcore-targeting-pack-6.0' is not installed, so not removed
Package 'aspnetcore-runtime-7.0' is not installed, so not removed
Package 'aspnetcore-runtime-dbg-8.0' is not installed, so not removed
Package 'aspnetcore-targeting-pack-7.0' is not installed, so not removed
Package 'aspnetcore-runtime-3.1' is not installed, so not removed
Package 'aspnetcore-runtime-9.0' is not installed, so not removed
Package 'aspnetcore-targeting-pack-9.0' is not installed, so not removed
Package 'aspnetcore-runtime-8.0' is not installed, so not removed
The following packages were automatically installed and are no longer
required:
  liblttng-ust-common1 liblttng-ust-ctl5 liblttng-ust1
  netstandard-targeting-pack-2.1
Use 'sudo apt autoremove' to remove them.
The following packages will be REMOVED:
  aspnetcore-targeting-pack-8.0
0 upgraded, 0 newly installed, 1 to remove and 120 not upgraded.
After this operation, 15,6 MB disk space will be freed.
Do you want to continue? [Y/n] Y
(Reading database ... 274953 files and directories currently installed.)
Removing aspnetcore-targeting-pack-8.0 (8.0.12-0ubuntu1~22.04.1) ...
jjpardo@johlver-virtual-machine:~$
```

4. 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.
5. Ejecutar el comando **sudo apt update** para realizar actualización de paquetes en linux.

🏷️ Instalación del SDK

El SDK de .NET permite desarrollar aplicaciones con .NET. Si instala el SDK de .NET, no necesita instalar el entorno de ejecución correspondiente. Para instalar el SDK de .NET, ejecute los comandos siguientes:

```
sudo apt-get install -y dotnet-sdk-8.0
```

🏷️ Instalación de la instancia en tiempo de ejecución

El entorno de ejecución de ASP.NET Core le permite ejecutar aplicaciones creadas con .NET en las que no se ha proporcionado el entorno de ejecución. Los comandos siguientes instalan el entorno de ejecución de ASP.NET Core, el más compatible con .NET. En el terminal, ejecute los comandos siguientes:

```
sudo apt-get install -y aspnetcore-runtime-8.0
```

🏷️ Validación de la Instalación dotnet

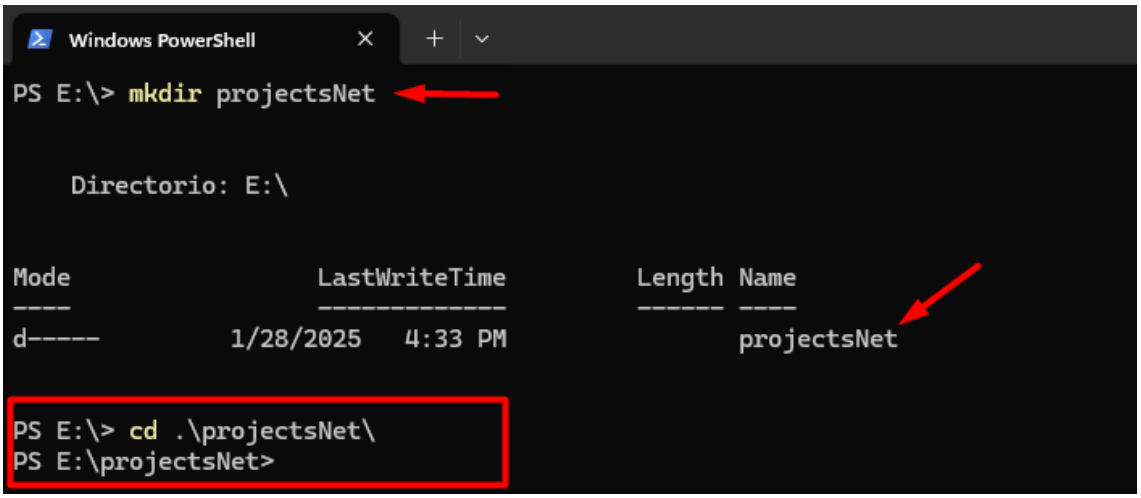
Para validar la instalación ejecute el comando **dotnet --info** debera tener el siguiente resultado:

```
trainer@johlver-virtual-machine:~$ dotnet --info
.NET SDK:
 Version:           8.0.405
 Commit:            fb1830d421
 Workload version:  8.0.400-manifests.c7afa696
 MSBuild version:   17.11.9+a69bbaaf5

Runtime Environment:
 OS Name:     ubuntu
 OS Version:  22.04
 OS Platform: Linux
 RID:         linux-x64
 Base Path:   /usr/share/dotnet/sdk/8.0.405/

.NET workloads installed:
Configured to use loose manifests when installing new manifests.
There are no installed workloads to display.

Host:
  Version:      8.0.12
  Architecture: x64
  Commit:       89ef51c5d8

.NET SDKs installed:
  8.0.405 [/usr/share/dotnet/sdk]

.NET runtimes installed:
  Microsoft.AspNetCore.App 8.0.12
[/usr/share/dotnet/shared/Microsoft.AspNetCore.App]
  Microsoft.NETCore.App 8.0.12 [/usr/share/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
```

🧱 Creacion de proyectos en C#

📖 Windows

1. Abrir la consola del sistema operativo (cmd) o PowerShell(Windows 11)
2. Cree una carpeta donde se guarden los proyectos.



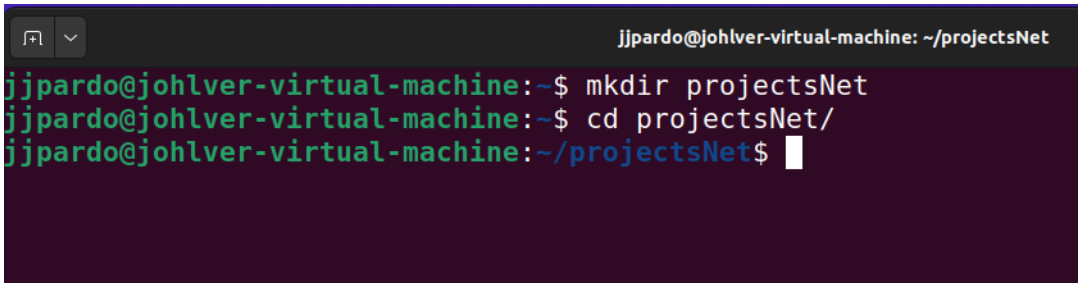
3. Ejecute el comando **dotnet new console -n NombreProyecto**

```
PS E:\projectsNet> dotnet new console -n baseApp

Esto es .NET 8.0.
----------------------
Versión del SDK: 8.0.405

Telemetría
---------
Las herramientas de .NET recopilan datos de uso para ayudarnos a mejorar su
experiencia. Microsoft los recopila y los comparte con la comunidad. Puede
optar por no participar en la telemetría si establece la variable de entorno
DOTNET_CLI_TELEMETRY_OPTOUT en "1" o "true" mediante su shell favorito.

Lea más sobre la telemetría de las herramientas de la CLI de .NET:
https://aka.ms/dotnet-cli-telemetry

----------------
Instalar un certificado de desarrollo HTTPS de ASP.NET Core.
Para confiar en el certificado, ejecute "dotnet dev-certs https --trust"
Obtenga información sobre HTTPS: https://aka.ms/dotnet-https

----------------
Escribir su primera aplicación: https://aka.ms/dotnet-hello-world
Descubra las novedades: https://aka.ms/dotnet-whats-new
Explore la documentación: https://aka.ms/dotnet-docs
Notificar problemas y encontrar el código fuente en GitHub:
https://github.com/dotnet/core
Use "dotnet --help" para ver los comandos disponibles o visite:
https://aka.ms/dotnet-cli
--------------------------------------------------------------------------------
---------
La plantilla "Aplicación de consola" se creó correctamente.
```

```
Procesando acciones posteriores a la creación...
Restaurando E:\projectsNet\baseApp\baseApp.csproj:
  Determinando los proyectos que se van a restaurar...
  Se ha restaurado E:\projectsNet\baseApp\baseApp.csproj (en 56 ms).
Restauración realizada correctamente.
```

Linux

1. Abra la terminal de linux Ctrl+Alt+T
2. Cree una nueva carpeta donde se guarden los proyectos



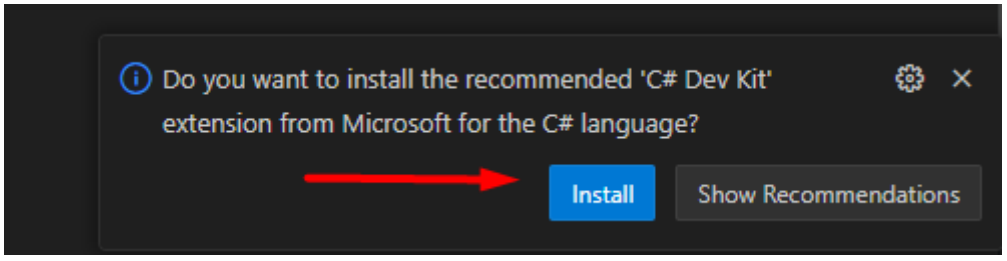
3. Ejecute el comando **dotnet new console -n NombreProyecto**

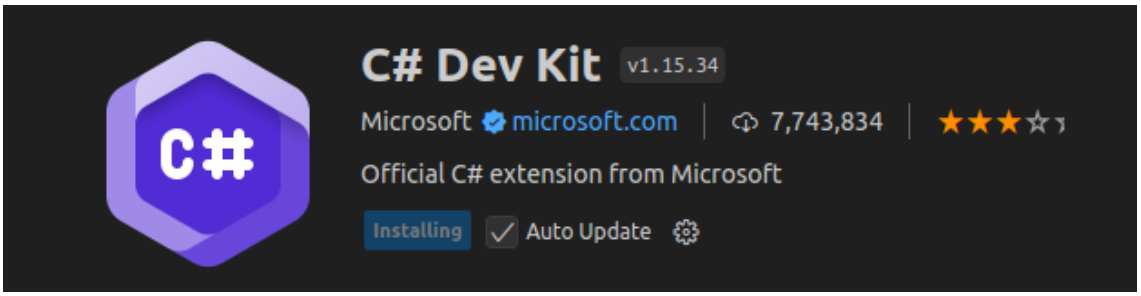
```
jjpardo@johlver-virtual-machine:~/projectsNet$ dotnet new console -n
baseApp
The template "Console App" was created successfully.

Processing post-creation actions...
Restoring /home/jjpardo/projectsNet/baseApp/baseApp.csproj:
  Determining projects to restore...
  Restored /home/jjpardo/projectsNet/baseApp/baseApp.csproj (in 107 ms).
Restore succeeded.

jjpardo@johlver-virtual-machine:~/projectsNet$
```

Abriendo el proyecto (Windows y Linux)

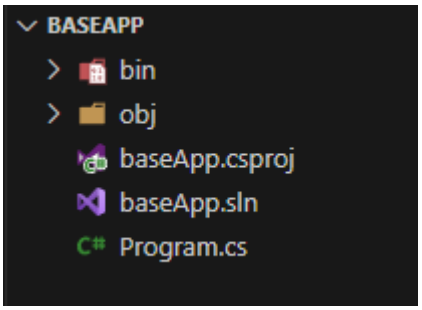
1. Ingrese a la carpeta del proyecto creado y ejecute el comando **code .**
2. Abra el archivo program.cs

   Si se esta abriendo por primera vez un proyecto de net se solicitara instalar la extensión de soporte de .Net para visual Studio Code.





3. A continuación se abre la siguiente estructura de proyecto.



📂 bin/

Esta carpeta contiene los archivos compilados (ejecutables y bibliotecas). En .NET, cuando se compila un proyecto, los archivos generados se almacenan aquí en diferentes configuraciones:

- **Debug**: Contiene los archivos compilados en modo depuración.
- **Release**: Contiene los archivos optimizados para producción.

📂 obj/

Esta carpeta almacena archivos temporales y metadatos generados por el compilador. Contiene:

- Archivos intermedios antes de la compilación final.
- Metadatos sobre la compilación del proyecto.

📄 baseApp.csproj

Es el **archivo de configuración del proyecto**. Contiene información sobre:

- El **SDK** de .NET que usa el proyecto.
- Las **dependencias y paquetes NuGet**.
- La **versión de C#** utilizada.
- Configuraciones como el **tipo de salida** (ejecutable o biblioteca).

📄 baseApp.sln

Es el **archivo de solución de Visual Studio**. Una solución puede contener múltiples proyectos relacionados. Se usa para administrar varios proyectos en un solo entorno de desarrollo.

📄 Program.cs

Este es el **archivo principal del programa**. En una aplicación .NET moderna (como una aplicación de consola o una API en .NET Core), este archivo contiene el **punto de entrada** (`Main`).

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

📮 Programación Básica C#

🧱 Variables y constantes

⌨️ 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 name = string.Empty;
```

⌨️ 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 PI = 3.14159;
```

🧱 Tipos de Datos y Conversiones

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

📖 Tipos de punto flotante

Palabra clave/tipo de C#	Intervalo aproximado	Precisión	Tamaño	Tipo de .NET
`float`	De ±1,5 x 10-45 a ±3,4 x 1038	De 6 a 9 dígitos aproximadamente	4 bytes	System.Single
`double`	De ±5,0 × 10−324 a ±1,7 × 10308	De 15 a 17 dígitos aproximadamente	8 bytes	System.Double
`decimal`	De ±1,0 x 10-28 to ±7,9228 x 1028	28-29 dígitos	16 bytes	System.Decimal

```
internal class Program
{
    private static void Main(string[] args)
    {
        var a = 12.3;
        double b = 12.3;

        Console.WriteLine($"El valor de a es = {a} ");
        Console.WriteLine($"El valor de a es = {b} ");

    }
}
```

> Ejecucion del programa
>
> **dotnet run**

🏷️ 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 doublé
- El literal con el sufijo f o F es del tipo float
- El literal con el sufijo m o M es del tipo decimal

```
internal class Program
{
    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}");
    }
}
```

- **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 **934_001**, que es un número decimal representado en notación de punto flotante.
- **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 **5F**. El sufijo **F** indica que el número literal es de tipo **float**.
- **f = 5.4f;** Esta línea actualiza el valor de **f** a **4f**, utilizando el sufijo **f** para indicar que es un **float**.
- **WriteLine($"{f}");** Se utiliza la interpolación de cadenas para imprimir el valor de **f** en la consola.
- **decimal myMoney = 3_000.5m;** Se declara una variable **myMoney** de tipo **decimal** y se le asigna el valor **5m**. El sufijo **m** indica que el número literal es de tipo **decimal**.
- **myMoney = 400.75M;** Aquí se actualiza el valor de **myMoney** a **75M**, utilizando el sufijo **M** para indicar que es un **decimal**.
- **WriteLine($"{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.

Especificador de formato	NOMBRE	Descripción
"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").

El especificador de formato "C" (divisa) convierte un número en una cadena que representa una cantidad de divisa. El especificador de precisión indica el número deseado de posiciones decimales en la cadena de resultado. Si se omite el especificador de precisión, el número predeterminado de posiciones decimales que se van a usar en los valores de moneda es 2.

🏷️ Propiedad NumberFormatInfo.CurrencySymbol

Obtiene o establece la cadena que se va a utilizar como símbolo de divisa.

En el ejemplo siguiente se muestra el símbolo de moneda de la referencia cultural actual y se usa la cadena de formato numérico estándar "C" para dar formato a un valor de moneda.

```
using System.Globalization;
internal class Program
{
    private static void Main(string[] args)
    {
      Decimal value = 106.25m;
      Console.WriteLine("Current Culture: {0}",
                        CultureInfo.CurrentCulture.Name);
      Console.WriteLine("Currency Symbol: {0}",
                        NumberFormatInfo.CurrentInfo.CurrencySymbol);
      Console.WriteLine("Currency Value:  {0:C2}", value);
    }
}
```

🏷️ Propiedad NumberFormatInfo.CurrencyDecimalDigits

En el ejemplo siguiente se muestra el efecto de cambiar la propiedad CurrencyDecimalDigits.

```
using System.Globalization;
internal class Program
{
    private static void Main(string[] args)
    {
      // Gets a NumberFormatInfo associated with the en-US culture.
      NumberFormatInfo nfi = new CultureInfo( "en-US", false ).NumberFormat;

      // Displays a negative value with the default number of decimal digits (2).
      Int64 myInt = -1234;
      Console.WriteLine( myInt.ToString( "C", nfi ) );

      // Displays the same value with four decimal digits.
      nfi.CurrencyDecimalDigits = 4;
      Console.WriteLine( myInt.ToString( "C", nfi ) );
    }
}
```

Desglose del Código:

```
using System.Globalization;
```

🔹 Se importa el espacio de nombres `System.Globalization`, que permite trabajar con formatos específicos de cultura (`CultureInfo`).

```
internal class Program
{
    private static void Main(string[] args)
```

🔹 Se define la clase `Program` con un método `Main`, que es el punto de entrada de la aplicación.

1️⃣ Creación del Formato de Moneda con Cultura `en-US`

```
// Gets a NumberFormatInfo associated with the en-US culture.
NumberFormatInfo nfi = new CultureInfo("en-US", false).NumberFormat;
```

🔹 Se obtiene una instancia de `NumberFormatInfo` basada en la cultura **"en-US"** (**Estados Unidos**).
🔹 `NumberFormatInfo` permite modificar configuraciones de **formato de moneda, separadores de decimales, miles, etc.**
🔹 El segundo parámetro `false` indica que **no** se debe usar `UseUserOverride` (evita configuraciones personalizadas del usuario).

2️⃣ Formateo de un Número Negativo en Formato Moneda

```
// Displays a negative value with the default number of decimal digits (2).
Int64 myInt = -1234;
Console.WriteLine(myInt.ToString("C", nfi));
```

🔹 Se declara un número entero de 64 bits (`Int64`) con valor **-1234**.
🔹 Se convierte el número a **formato de moneda (`"C"`)** usando la configuración de `nfi` (cultura `en-US`).
🔹 En `en-US`, la moneda por defecto es **dólares estadounidenses (`$`)**, y el formato de moneda estándar usa **2 decimales**.

🖥️ **Salida esperada (formato por defecto, 2 decimales):**

```
($1,234.00)
```

3️⃣ Modificación de los Decimales en el Formato de Moneda

```
nfi.CurrencyDecimalDigits = 4;
Console.WriteLine(myInt.ToString("C", nfi));
```

🔹 **Se cambia la cantidad de decimales de la moneda a 4.**
🔹 Ahora, al volver a convertir el número `myInt` a moneda (`"C"`), se mostrarán **4 decimales en lugar de 2**.

🖥️ **Salida esperada (modificado a 4 decimales):**

```
($1,234.0000)
```

🔹 Resumen de la Ejecución

Código	Salida esperada (`en-US`)
`Console.WriteLine(myInt.ToString("C", nfi));`	**($1,234.00)**
`nfi.CurrencyDecimalDigits = 4;` `Console.WriteLine(myInt.ToString("C", nfi));`	**($1,234.0000)**

🎯 Conclusión

✅ `NumberFormatInfo` permite personalizar el formato de moneda según la cultura.
✅ `CurrencyDecimalDigits` define el número de decimales en el formato monetario.
✅ Se puede usar **diferentes culturas** (`fr-FR`, `ja-JP`, `es-ES`, etc.) para cambiar el símbolo de moneda y el separador decimal.

🏷️ Propiedad NumberFormatInfo.CurrencyDecimalSeparator

En el ejemplo siguiente se muestra el efecto de cambiar la propiedad CurrencyDecimalSeparator .

```
using System.Globalization;
internal class Program
{
    private static void Main(string[] args)
    {
      NumberFormatInfo nfi = new CultureInfo( "en-US", false ).NumberFormat;

      // Displays a value with the default separator (".").
      Int64 myInt = 123456789;
      Console.WriteLine( myInt.ToString( "C", nfi ) );

      // Displays the same value with a blank as the separator.
      nfi.CurrencyDecimalSeparator = " ";
      Console.WriteLine( myInt.ToString( "C", nfi ) );
    }
}
```

Desglose del Código

```
using System.Globalization;
```

🔹 Se importa `System.Globalization`, que permite trabajar con configuraciones regionales (`CultureInfo`).

```
internal class Program
{
    private static void Main(string[] args)
```

🔹 Se define la clase `Program` con el método `Main`, que es el punto de entrada de la aplicación.

1️⃣ Configuración de `NumberFormatInfo` con `en-US`

```
NumberFormatInfo nfi = new CultureInfo("en-US", false).NumberFormat;
```

🔹 Se obtiene una instancia de `NumberFormatInfo` basada en la cultura **"en-US"** (**Estados Unidos**).
🔹 `NumberFormatInfo` permite modificar configuraciones de **formato de moneda, separadores de decimales, separadores de miles, etc.**
🔹 El parámetro `false` evita usar configuraciones personalizadas del usuario.

2️⃣ Mostrar un Número con el Formato de Moneda por Defecto

```
// Displays a value with the default separator (".").
Int64 myInt = 123456789;
Console.WriteLine(myInt.ToString("C", nfi));
```

🔹 Se define un número `myInt = 123456789` (**sin decimales**).
🔹 Se imprime usando `ToString("C", nfi)`, que convierte el número a **formato de moneda** en `en-US`.
🔹 En `en-US`, el separador decimal predeterminado es **"." (punto)**.

🖥️ **Salida esperada en consola (por defecto, `en-US` con `.` como separador decimal):**

```
$123,456,789.00
```

🏷️ **Explicación del formato:**

- **$** → Símbolo de dólar (`en-US`).
- **,** → Separador de miles (`123,456,789`).
- **.** → Separador decimal (`.00` porque `en-US` usa 2 decimales por defecto en moneda).

3️⃣ Cambiar el Separador Decimal a un Espacio en Blanco

```
// Displays the same value with a blank as the separator.
nfi.CurrencyDecimalSeparator = " ";
Console.WriteLine(myInt.ToString("C", nfi));
```

🔹 **Se modifica el separador decimal (`CurrencyDecimalSeparator`) para que sea un espacio en blanco `" "` en lugar de `"."`.**
🔹 Cuando se vuelve a imprimir el número en formato moneda, en lugar de **"123,456,789.00"**, ahora mostrará **"123,456,789 00"**.

🖥️ **Salida esperada en consola (con espacio en blanco como separador decimal):**

```
$123,456,789 00
```

🏷️ **Explicación del nuevo formato:**

- **$** → Se mantiene el símbolo de dólar.
- **,** → Se mantiene el separador de miles.
- **(espacio en blanco)** → Ahora reemplaza al punto decimal.

🔹 Resumen de la Ejecución

Código	**Salida esperada (`en-US`)**
`Console.WriteLine(myInt.ToString("C", nfi));` (por defecto)	**$123,456,789.00**

Código	Salida esperada (`en-US`)
`nfi.CurrencyDecimalSeparator = " ";` `Console.WriteLine(myInt.ToString("C", nfi));`	**$123,456,789 00**

🎯 Conclusión

✅ `NumberFormatInfo` permite personalizar el separador decimal en formatos de moneda.
✅ `CurrencyDecimalSeparator` se puede modificar para usar **puntos, comas, espacios u otros caracteres**.
✅ **Ejemplo práctico:** Se puede usar `" "` en lugar de `"."` si se necesita un formato especial en ciertas regiones.

```
using System.Globalization;

internal class Program
{
    private static void Main(string[] args)
    {
        decimal value = 123.456m;
        Console.OutputEncoding = System.Text.Encoding.UTF8; // Asegurar
compatibilidad con caracteres especiales

        Console.WriteLine($"{value.ToString("C3", new CultureInfo("en-US"))}");
 // $123.46
        Console.WriteLine($"{value.ToString("C", new CultureInfo("fr-FR"))}");
 // 123,46 €
        Console.WriteLine($"{value.ToString("C", new CultureInfo("ja-JP"))}");
    // ¥123

        Console.WriteLine($"${value.ToString("C3", new CultureInfo("en-US"))}");
 // $123.456
        Console.WriteLine($"{value.ToString("C3", new CultureInfo("fr-FR"))}");
// 123,456 €
        Console.WriteLine($"{value.ToString("C3", new CultureInfo("ja-JP"))}");
   // ¥123.456

        Console.WriteLine($"-${value.ToString("C3", new CultureInfo("en-US"))}");
// -$123.46
        Console.WriteLine($"-{value.ToString("C3", new CultureInfo("fr-FR"))}");
// -123,46 €
        Console.WriteLine($"-{value.ToString("C3", new CultureInfo("ja-JP"))}");
    // -¥123
    }
}
```

🏷️ Especificador de formato decimal (D)

El especificador de formato "D" (o decimal) convierte un número en una cadena de dígitos decimales (0-9), precedida por un signo menos si el número es negativo. Este formato sólo es compatible con los tipos enteros.

```
internal class Program
{
    private static void Main(string[] args)
    {
        int num1 = 1234;
        int num2 = -1234;

        // Formato decimal sin ceros de relleno
        Console.WriteLine(num1.ToString("D"));   // 1234

        // Formato decimal con 6 dígitos, rellenando con ceros a la izquierda
        Console.WriteLine(num2.ToString("D6"));  // -001234
    }
}
```

🏷️ Especificador de formato de punto fijo (F)

El especificador de formato de punto fijo ("F") convierte un número en una cadena con el formato "-ddd.ddd…", donde cada "d" indica un dígito (0-9). La cadena comienza con un signo menos si el número es negativo.

El especificador de precisión indica el número deseado de cifras decimales. Si se omite el especificador de precisión, el número predeterminado de posiciones decimales que se van a usar en valores numéricos es 2.

La cadena de resultado se ve afectada por la información de formato de la referencia cultural actual.

Especificador de formato	Nombre	Descripción
"F" o "f"	Punto fijo	Resultado: dígitos integrales y decimales con signo negativo opcional.
		Compatible con: todos los tipos numéricos.
		Especificador de precisión: número de dígitos decimales.
		Especificador de precisión predeterminado: definido por la referencia cultural.
		Más información: Especificador de formato de punto fijo ("F").

```
using System.Globalization;
```

```
internal class Program
{
    private static void Main(string[] args)
    {
        double num1 = 1234.567;
        int num2 = 1234;
        double num3 = -1234.56;

        Console.OutputEncoding = System.Text.Encoding.UTF8; // Asegurar
compatibilidad con caracteres especiales
        // Formato "F" (Número de punto flotante fijo)
        Console.WriteLine(num1.ToString("F", new CultureInfo("en-US"))); //
1234.57
        Console.WriteLine(num1.ToString("F", new CultureInfo("de-DE"))); //
1234,57

        // Formato "F1" (1 decimal)
        Console.WriteLine(num2.ToString("F1", new CultureInfo("en-US"))); //
1234.0
        Console.WriteLine(num2.ToString("F1", new CultureInfo("de-DE"))); //
1234,0

        // Formato "F4" (4 decimales)
        Console.WriteLine(num3.ToString("F4", new CultureInfo("en-US"))); //
-1234.5600
        Console.WriteLine(num3.ToString("F4", new CultureInfo("de-DE"))); //
-1234,5600
    }
}
```

`ToString("F")` → Formatea el número con 2 decimales por defecto.

`ToString("F1")` → Formatea con **1 decimal**.

`ToString("F4")` → Formatea con **4 decimales**.

Diferencia entre culturas:

- `"en-US"` usa **punto (`.`) como separador decimal**.
- `"de-DE"` usa **coma (`,`) como separador decimal**.

📓 Palabras claves

Las palabras clave son identificadores reservados predefinidos que tienen un significado especial para el compilador. No podrá utilizarlos como identificadores en el programa a no ser que incluyan `@` como prefijo. Por ejemplo, `@if` es un identificador válido, pero `if` no lo es, porque `if` es una palabra clave.

abstract	event	namespace	static
as	explicit	new	string

base	extern	null	struct
bool	false	object	switch
break	finally	operator	this
byte	fixed	out	throw
case	float	override	true
catch	for	params	try
char	foreach	private	typeof
checked	goto	protected	uint
class	if	public	ulong
const	implicit	readonly	unchecked
continue	in	ref	unsafe
decimal	int	return	ushort
default	interface	sbyte	using
delegate	internal	sealed	virtual
do	is	short	void
double	lock	sizeof	volatile
else	long	stackalloc	while
enum			

🎹 Palabras clave contextuales

add	**get**	**notnull**	**set**
and	global	nuint	unmanaged (convención de llamada de puntero de función)
alias	group	on	unmanaged (restricción de tipo genérico)
ascending	init	or	value
args	into	orderby	var
async	join	partial (tipo)	when (condición de filtro)
await	let	partial (método)	where (restricción de tipo genérico)

add	get	notnull	set
by	managed (convención de llamada de puntero de función)	record	where (cláusula de consulta)
descending	nameof	remove	con
dynamic	nint	select	yield
equals	not		
from			

📓 Entrada y salida de datos

📖 Salida

Los métodos `Console.WriteLine` y `Console.Write` son usados para imprimir texto en la consola en C#.

🔹 `Console.WriteLine()`

El método `Console.WriteLine()` imprime texto en la consola y agrega un salto de línea (`\n`) automáticamente al final.

🏷️ Ejemplo:

```
Console.WriteLine("Hola, mundo!");
Console.WriteLine("Esto es una nueva línea.");
```

🏷️ **Nota:** Cada `Console.WriteLine` mueve el cursor a la siguiente línea.

🔹 `Console.WriteLine` con Interpolación de Cadenas (`$""`)

```
Console.WriteLine($"Hola, {nombre}. Tienes {edad} años.");
```

```
internal class Program
{
    private static void Main(string[] args)
    {
      string nombre = "Camila";
      int edad = 25;
      Console.Clear();
      Console.WriteLine($"Hola, {nombre}. Tienes {edad} años.");

    }
}
```

🔹 `Console.Write()`

El método `Console.Write()` imprime texto en la consola **sin agregar un salto de línea**.

🏷️ Ejemplo:

```
Console.Write("Hola");
Console.Write(" ");
Console.Write("mundo!");
```

🔹 Diferencia Entre `Console.WriteLine` y `Console.Write`

Método	Comportamiento
`Console.WriteLine("Hola");`	Imprime `"Hola"` y pasa a la siguiente línea.
`Console.Write("Hola");`	Imprime `"Hola"` y mantiene el cursor en la misma línea.

🏷️ Ejemplo Comparativo:

```
Console.Write("Esto es ");
Console.Write("un mensaje ");
Console.Write("en la misma línea.");

Console.WriteLine("\n"); // Salto de línea manual

Console.WriteLine("Esto es un mensaje con salto de línea.");
Console.WriteLine("Cada WriteLine está en una nueva línea.");
```

🏷️ Salida en consola:

```
Esto es un mensaje en la misma línea.

Esto es un mensaje con salto de línea.
Cada WriteLine está en una nueva línea.
```

📖 Entrada

Las funciones de entrada en C# permiten al usuario proporcionar datos a través del teclado, archivos, bases de datos, formularios web y otros medios.

🔹 Entrada desde Consola (`Console.ReadLine`, `Console.ReadKey`, `Console.Read`)

En aplicaciones de consola, estos métodos permiten capturar información del usuario.

🔹 `Console.ReadLine()` → Lee una línea completa de entrada

Este método captura una línea de texto ingresada por el usuario y la devuelve como una cadena (`string`).

```
Console.Write("Ingrese su nombre: ");
string nombre = Console.ReadLine();
Console.WriteLine($"Hola, {nombre}!");
```

🔹 `Console.ReadKey()` → Captura una tecla sin necesidad de presionar Enter

Este método permite leer una sola tecla ingresada sin necesidad de presionar **Enter**.

```
Console.WriteLine("Presiona una tecla para continuar...");
ConsoleKeyInfo tecla = Console.ReadKey();
Console.WriteLine($"\nHas presionado: {tecla.Key}");
```

🏷️ **Salida :**

```
Presiona una tecla para continuar...
Has presionado: A
```

> 📌 **Nota:** Si agregas `true` como parámetro (`Console.ReadKey(true);`), la tecla no se mostrará en la consola.

🔹 `Console.Read()` → Lee un solo carácter como entero (`int`)

Este método captura un solo carácter ingresado y devuelve su código ASCII.

```
Console.Write("Ingrese un carácter: ");
int codigoAscii = Console.Read();
Console.WriteLine($"Código ASCII ingresado: {codigoAscii}");
```

📌 **Entrada:** `A`
📌 **Salida:**

```
Código ASCII ingresado: 65
```

> 🏷️ **Nota:** Se debe presionar `Enter` después de ingresar el carácter.

```
internal class Program
{
    private static void Main(string[] args)
    {
        // 🔹 1. Entrada de datos: Pedir nombre y edad
        Console.Write("Ingrese su nombre: ");
        string nombre = Console.ReadLine(); // Captura el nombre

        Console.Write("Ingrese su edad: ");
        int edad;

        // Verifica que el usuario ingrese un número válido
        while (!int.TryParse(Console.ReadLine(), out edad))
        {
            Console.Write("Edad no válida. Ingrese un número: ");
        }
```

```
        // 🔹 2. Salida de datos con interpolación de cadenas
        Console.WriteLine($"\nHola {nombre}, tienes {edad} años.");

        // 🔹 3. Leer un solo carácter
        Console.Write("Presiona cualquier tecla para continuar...");
        Console.ReadKey(); // Captura la tecla presionada

        // 🔹 4. Limpiar la consola y mostrar mensaje final
        Console.Clear();
        Console.WriteLine("¡Gracias por usar este programa! 🚀");
    }
}
```

🎆 Caracteres de escape

Los **caracteres de escape** son combinaciones especiales de caracteres precedidos por una barra invertida (`\`), que permiten insertar caracteres no imprimibles o especiales en una cadena de texto.

Se utilizan para representar elementos como saltos de línea, tabulaciones, comillas y otros símbolos dentro de una cadena.

📖 Lista de Caracteres de Escape en C#

Caracter	Descripción
`\n`	Nueva línea (salto de línea)
`\t`	Tabulación horizontal
`\r`	Retorno de carro (usado en Windows junto con `\n`)
`\\`	Barra invertida (`\`)
`\'`	Comilla simple (`'`)
`\"`	Comilla doble (`"`)
`\b`	Retroceso (Backspace)
`\f`	Salto de página
`\v`	Tabulación vertical
`\0`	Carácter nulo (null)
`\uXXXX`	Unicode (carácter basado en un código hexadecimal)
`\xXX`	Código ASCII en hexadecimal

1️⃣ `\n` - Nueva línea

```
Console.WriteLine("Hola, mundo!\nBienvenido a C#.");
```

```
Hola, mundo!
Bienvenido a C#.
```

2 `\t` - Tabulación (Espacios extra)

```
Console.WriteLine("Nombre:\tJose Manuel");
Console.WriteLine("Edad:\t25");
```

3 `\\` - Barra invertida

```
Console.WriteLine("Ruta en Windows: C:\\Archivos de Programa\\MiApp");
```

> Ruta en Windows: C:\Archivos de Programa\MiApp

4 `\'` - Comilla simple y `\"` - Comilla doble

```
Console.WriteLine("Ella dijo: \"C# es genial!\"");
Console.WriteLine("Caracter: \'A\'");
```

> Ella dijo: "C# es genial!"
> Caracter: 'A'

5 `\b` - Retroceso (Elimina el carácter anterior)

```
Console.WriteLine("Hola\b Mundo!");  // Borra la "a"
```

> Hol Mundo!

6 `\r` - Retorno de carro (Reemplaza lo anterior en la misma línea)

```
Console.Write("Primera línea\rSegunda línea");
```

> Segunda línea

7 `\0` - Carácter Nulo (Null)

```
string texto = "Hola\0Mundo";
Console.WriteLine(texto);
Console.WriteLine("Longitud real: " + texto.Length);
```

> HolaMundo
> Longitud real: 9 // La longitud real no incluye el carácter nulo

8️⃣ `\uXXXX` - Unicode

```
Console.WriteLine("\u2665 Esto es un corazón!");
```

> ♥ Esto es un corazón!

🏷️ Tabla de Caracteres Unicode Comunes

Código	Carácter	Descripción
`\u0020`	``	Espacio
`\u0021`	`!`	Signo de exclamación
`\u0022`	`"`	Comillas dobles
`\u0023`	`#`	Símbolo numeral
`\u0024`	`$`	Signo de dólar
`\u0025`	`%`	Porcentaje
`\u0026`	`&`	Ampersand
`\u0027`	`'`	Comilla simple
`\u0028`	`(`	Paréntesis izquierdo
`\u0029`	`)`	Paréntesis derecho
`\u002A`	`*`	Asterisco
`\u002B`	`+`	Signo más
`\u002C`	`,`	Coma
`\u002D`	`-`	Guion
`\u002E`	`.`	Punto
`\u002F`	`/`	Barra inclinada
`\u003A`	`:`	Dos puntos
`\u003B`	`;`	Punto y coma
`\u003C`	`<`	Menor que
`\u003D`	`=`	Igual
`\u003E`	`>`	Mayor que
`\u003F`	`?`	Signo de interrogación
`\u0040`	`@`	Arroba
`\u005B`	`[`	Corchete izquierdo

Código	Carácter	Descripción
`\u005C`	\	Barra invertida
`\u005D`	]	Corchete derecho
`\u005E`	^	Circunflejo
`\u005F`	_	Guion bajo
`\u0060`	```	Acento grave
`\u007B`	{	Llave izquierda
`\u007C`	|	
`\u007D`	}	Llave derecha
`\u007E`	~	Tilde (~)

🏷️ Símbolos Especiales y Matemáticos

Código	Carácter	Descripción
`\u00A9`	©	Símbolo de copyright
`\u00AE`	®	Marca registrada
`\u2122`	™	Marca comercial
`\u2660`	♠	Pica
`\u2663`	♣	Trébol
`\u2665`	♥	Corazón
`\u2666`	♦	Diamante
`\u00B1`	±	Más-menos
`\u00D7`	×	Multiplicación
`\u00F7`	÷	División
`\u221E`	∞	Infinito
`\u03C0`	π	Pi
`\u221A`	√	Raíz cuadrada
`\u222B`	∫	Integral
`\u2248`	≈	Aproximado

🏷️ Letras Griegas y Científicas

Código	Carácter	Descripción
`\u03B1`	α	Alfa
`\u03B2`	β	Beta
`\u03B3`	γ	Gama
`\u03B4`	δ	Delta
`\u03B5`	ε	Épsilon
`\u03B7`	η	Eta
`\u03B8`	θ	Theta
`\u03BB`	λ	Lambda
`\u03BC`	μ	Mi
`\u03C1`	ρ	Rho
`\u03C3`	σ	Sigma
`\u03C4`	τ	Tau
`\u03C9`	ω	Omega

🏷️ Caras y Emoticones Unicode

Código	Carácter	Descripción
`\u263A`	☺	Carita sonriente
`\u2639`	☹	Carita triste
`\u263C`	☼	Sol
`\u2602`	☂	Paraguas
`\u2709`	✉	Sobre de carta

🏷️ Ejemplo de Uso en C#

Puedes usar caracteres Unicode en C# con `\uXXXX` dentro de una cadena:

```
using System;

class Program
{
    static void Main()
    {
        Console.WriteLine("Símbolos Unicode en C#:");
```

```
        Console.WriteLine("\u2665 Corazón");
        Console.WriteLine("\u221E Infinito");
        Console.WriteLine("\u03C0 Pi");
        Console.WriteLine("\u263A Carita feliz");
        Console.WriteLine("\u00A9 Copyright 2024");
    }
}
```

🏷️ **Salida en consola:**

```
Símbolos Unicode en C#:
♥ Corazón
∞ Infinito
π Pi
☺ Carita feliz
© Copyright 2024
```

🏷️ Alternativa: Cadenas Verbatim (`@""`)

Si no quieres usar caracteres de escape, puedes usar cadenas verbatim con `@`, que ignoran `\` como escape:

```
Console.WriteLine(@"Ruta en Windows: C:\Archivos de Programa\MiApp");
```

🏷️ **Salida en consola:**

```
Ruta en Windows: C:\Archivos de Programa\MiApp
```

🏷️ **Nota:** No se necesitan dobles barras invertidas (`\\`).

Los caracteres de escape son fundamentales para manejar texto en C#. Permiten: ✅ Formatear texto correctamente (`\n`, `\t`)
✅ Incluir caracteres especiales (`\\`, `\'`, `\"`)
✅ Manipular visualmente la salida (`\b`, `\r`)
✅ Representar caracteres Unicode (`\uxxxx`)

🎆 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 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.

📖 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.

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` o `nint`
byte	`short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, `decimal`, `nint` o `nuint`
short	`int`, `long`, `float`, `double`, o `decimal`, o `nint`
ushort	`int`, `uint`, `long`, `ulong`, `float`, `double`, o `decimal`, `nint`, o `nuint`
int	`long`, `float`, `double`, o `decimal`, `nint`
uint	`long`, `ulong`, `float`, `double`, o `decimal`, o `nuint`
long	`float`, `double` o `decimal`
ulong	`float`, `double` o `decimal`
float	`double`
nint	`long`, `float`, `double` o `decimal`
nuint	`ulong`, `float`, `double` o `decimal`

🏷️ Ejemplo de Conversión Implícita

```
int numeroEntero = 100;
double numeroDecimal = numeroEntero; // Conversión implícita de int a double

Console.WriteLine(numeroDecimal); // Salida: 100
```

📌 **Explicación:** `int` → `double` es seguro porque `double` puede almacenar valores enteros sin pérdida.

🏷️ Otras Conversión Implícitas Comunes

```
byte numByte = 50;
int numInt = numByte;  // ✅ byte → int (Seguro)
float numFloat = numInt;  // ✅ int → float (Seguro)
long numLong = numInt; // ✅ int → long (Seguro)
```

✅ **C# realiza estas conversiones automáticamente porque no hay riesgo de pérdida de datos.**

📖 Conversiones Explícitas

🏷️ Ejemplo de Conversión Explícita

```
double numeroDecimal = 9.7;
int numeroEntero = (int) numeroDecimal; // Conversión explícita (Casting)
Console.WriteLine(numeroEntero); // Salida: 9 (se pierde la parte decimal)
```

📌 **Explicación:**

- `double` a `int` **NO es seguro** porque `int` no tiene decimales.
- Se usa `(int)` **para forzar** la conversión.
- **Se pierde la parte decimal** (`9.7` → `9`).

🏷️ Otras Conversiones Explícitas Comunes

```
long numLong = 100000;
int numInt = (int)numLong;  // ⚠️ Puede perder datos si el número es muy grande
float numFloat = 12.34f;
int numInt2 = (int)numFloat;  // ⚠️ Pierde decimales, resultado: 12
```

🔹 3. Métodos de Conversión en C# (`Convert` y `Parse`)

Si necesitas convertir datos sin usar **casting**, C# ofrece métodos como `Convert.ToInt32()`, `int.Parse()` y `int.TryParse()`.

🏷️ `Convert.ToTipo()` → **Conversión Segura**

```
double numDouble = 15.75;
int numInt = Convert.ToInt32(numDouble); // Redondea a 16

Console.WriteLine(numInt); // Salida: 16
```

🏷️ **Diferencia con Casting:**

- `Convert.ToInt32(15.75)` **redondea** a `16`.
- `(int) 15.75` simplemente **trunca** a `15`.

🏷️ `Parse()` → **Para convertir cadenas en números**

```
string textoNumero = "123";
int numero = int.Parse(textoNumero);

Console.WriteLine(numero); // Salida: 123
```

⚠️ **Nota:** Si `textoNumero` contiene caracteres no numéricos (`"abc"`), lanzará una **excepción**.

🏷️ `TryParse()` → **Evita Errores**

Si el usuario ingresa datos incorrectos, `TryParse()` evita que el programa falle.

```
Usuario = "456";
int numero;

if (int.TryParse(entradaUsuario, out numero))
{
    Console.WriteLine($"Número válido: {numero}");
}
else
{
    Console.WriteLine("Entrada inválida");
}
```

🏷️ **Diferencia con `Parse()`:**

- `TryParse()` **NO lanza excepciones** si la conversión falla.
- Retorna `true` si la conversión es exitosa, `false` si falla.

🔹 Resumen de Conversión en C#

Tipo de Conversión	Método	Ejemplo	Notas
Implícita ✅	Automática	`int → double`	Sin riesgo de pérdida

Tipo de Conversión	Método	Ejemplo	Notas
Explícita ⚠️	Casting `(tipo)`	`(int) 9.7` → `9`	Puede perder datos
Convert.ToTipo()	`Convert.ToInt32()`	`Convert.ToInt32(15.7)` → `16`	Redondea valores
Parse()	`int.Parse("123")`	`"123"` → `123`	⚠️ Lanza excepción si es inválido
TryParse()	`int.TryParse("123", out num)`	`"abc"` → `false`	✅ No lanza excepción

Ejemplo

```
internal class Program
{
    private static void Main(string[] args)
    {
        // 🔹 1. Entrada de un número decimal desde la consola
        Console.Write("Ingrese un número decimal: ");
        double numeroDecimal;

        while (!double.TryParse(Console.ReadLine(), out numeroDecimal))
        {
            Console.Write("Entrada inválida. Ingrese un número decimal válido:
");
        }

        // 🔹 2. Conversión Explícita (Casting)
        int numeroEntero = (int)numeroDecimal;  // Convierte double a int (pierde
decimales)

        // 🔹 3. Otras conversiones explícitas con casting
        long numeroGrande = 1000000000;   // Un número muy grande
        short numeroPequeño = (short)numeroGrande;  // Puede perder información

        float numeroFlotante = 123.99f;
        byte numeroByte = (byte)numeroFlotante;  // Trunca el número flotante

        // 🔹 4. Mostrar resultados
        Console.WriteLine("\n📌 Resultados de la conversión explícita:");
        Console.WriteLine($"Número original (double): {numeroDecimal}");
        Console.WriteLine($"Convertido a int (pierde decimales):
{numeroEntero}");
        Console.WriteLine($"Número largo (long): {numeroGrande} → Convertido a
short: {numeroPequeño} (puede haber pérdida)");
        Console.WriteLine($"Número flotante (float): {numeroFlotante} →
Convertido a byte: {numeroByte} (trunca el valor)");

        // 🔹 5. Leer una tecla antes de salir
        Console.Write("\nPresiona cualquier tecla para salir...");
```

```
            Console.ReadKey();
        }
    }
```

🧮 Operadores Matemáticos

📖 Operadores Aritméticos

Operador	Descripción	Ejemplo	Resultado
`+`	Suma	`int suma = 5 + 3;`	`8`
`-`	Resta	`int resta = 10 - 4;`	`6`
`*`	Multiplicación	`int multi = 7 * 2;`	`14`
`/`	División	`int div = 9 / 2;`	`4` (división entera)
`%`	Módulo (Residuo)	`int mod = 10 % 3;`	`1`

💡 **Nota:** La división entre enteros devuelve un resultado entero, truncando la parte decimal. Para obtener un resultado con decimales, al menos uno de los operandos debe ser `double` o `float`.

📌 Orden de Precedencia en C#

De mayor a menor prioridad:

Nivel	Operadores	Descripción	Asociatividad
1️⃣	`()`	**Paréntesis** (se evalúan primero)	Izquierda a Derecha
2️⃣	`++` `--`	**Incremento/Decremento** (postfijo)	Izquierda a Derecha
3️⃣	`+` `-` (unario) `!`	**Signo positivo/negativo**, **Negación lógica**	Derecha a Izquierda
4️⃣	`*` `/` `%`	**Multiplicación, División y Módulo**	Izquierda a Derecha
5️⃣	`+` `-`	**Suma y Resta**	Izquierda a Derecha
6️⃣	`<` `<=` `>` `>=`	**Comparaciones**	Izquierda a Derecha
7️⃣	`==` `!=`	**Igualdad y Diferencia**	Izquierda a Derecha
8️⃣	`&&`	**AND lógico**	Izquierda a Derecha
9️⃣	`||`		

Nivel	Operadores	Descripción	Asociatividad
🔟	`=` `+=` `-=` `*=` `/=` `%=`	**Asignación**	Derecha a Izquierda

```
internal class Program
{
    private static void Main(string[] args)
    {
         int resultado = 10 + 5 * 2 - 8 / 4;
        Console.WriteLine(resultado); // 18
    }
}
```

📌 **Explicación:**

1. Multiplicación (`5 * 2 = 10`) y división (`8 / 4 = 2`) se resuelven primero.
2. Luego se evalúa la suma y resta: `10 + 10 - 2 = 18`.

```
internal class Program
{
    private static void Main(string[] args)
    {
        int a = 5, b = 10, c = 15;

        int resultado = a + b * c / 5 - 3;
        Console.WriteLine($"Resultado sin paréntesis: {resultado}");

        int resultadoConParentesis = (a + b) * (c / 5) - 3;
        Console.WriteLine($"Resultado con paréntesis: {resultadoConParentesis}");
    }
}
```

```
internal class Program
{
    private static void Main(string[] args)
    {
        // 📌 1️⃣ Operaciones básicas y jerarquía de operadores matemáticos
        int a = 10, b = 5, c = 2;
        int resultado1 = a + b * c - 8 / 4;  // Multiplicación y división primero
        Console.WriteLine($"Resultado1 (10 + 5 * 2 - 8 / 4): {resultado1}"); //
18

        int resultado2 = (a + b) * c - 8 / 4; // Paréntesis alteran el orden
        Console.WriteLine($"Resultado2 ((10 + 5) * 2 - 8 / 4): {resultado2}"); //
26

        // 📌 2️⃣ Operaciones de incremento y decremento
        int x = 5;
        int y = ++x + x--;  // ++x incrementa primero, x-- se evalúa antes de
restar
        Console.WriteLine($"y = ++x + x--: {y} (x final: {x})"); // 11 (x = 5)
```

```
        // 📌 3️⃣ Operaciones lógicas y relacionales combinadas
        int edad = 20;
        bool tieneLicencia = true;
        bool puedeConducir = (edad >= 18) && tieneLicencia;
        Console.WriteLine($"¿Puede conducir? {puedeConducir}"); // true

        // 📌 4️⃣ Uso de operadores lógicos con relacionales
        int temperatura = 30;
        bool haceFrio = !(temperatura > 15); // Negación lógica
        Console.WriteLine($"¿Hace frío? {haceFrio}"); // false

        // 📌 5️⃣ Uso de operadores de asignación combinados
        int numero = 10;
        numero += 5;  // Equivalente a numero = numero + 5
        numero *= 2;  // Equivalente a numero = numero * 2
        Console.WriteLine($"Número tras operaciones: {numero}"); // 30

        // 📌 6️⃣ Expresión Compleja combinando todo
        int resultadoFinal = ((a * b) / c + x) - (numero % 4) * (y - 2);
        Console.WriteLine($"Resultado final: {resultadoFinal}");
    }
}
```

📖 Operadores de Asignación

Operador	Descripción	Ejemplo
`=`	Asignación	`int x = 10;`
`+=`	Suma y asigna	`x += 5; // x = x + 5`
`-=`	Resta y asigna	`x -= 3; // x = x - 3`
`*=`	Multiplica y asigna	`x *= 2; // x = x * 2`
`/=`	Divide y asigna	`x /= 4; // x = x / 4`
`%=`	Módulo y asigna	`x %= 2; // x = x % 2`

📖 Operadores de Incremento y Decremento

Operador	Descripción	Ejemplo	Resultado
`++`	Incremento en 1 (prefijo)	`++x;`	Aumenta primero y luego usa el valor
`++`	Incremento en 1 (sufijo)	`x++;`	Usa el valor y luego lo incrementa
`--`	Decremento en 1 (prefijo)	`--y;`	Disminuye primero y luego usa el valor
`--`	Decremento en 1 (sufijo)	`y--;`	Usa el valor y luego lo decrementa

📖 Operadores Matemáticos en `Math`

C# incluye la clase `Math` que proporciona métodos matemáticos avanzados:

Método	Descripción	Ejemplo	Resultado
`Math.Abs(x)`	Valor absoluto	`Math.Abs(-10)`	`10`
`Math.Pow(x, y)`	Potencia	`Math.Pow(2, 3)`	`8.0`
`Math.Sqrt(x)`	Raíz cuadrada	`Math.Sqrt(25)`	`5.0`
`Math.Round(x)`	Redondeo estándar	`Math.Round(4.7)`	`5`
`Math.Floor(x)`	Redondeo hacia abajo	`Math.Floor(4.9)`	`4`
`Math.Ceiling(x)`	Redondeo hacia arriba	`Math.Ceiling(4.1)`	`5`
`Math.Max(x, y)`	Máximo entre dos números	`Math.Max(10, 20)`	`20`
`Math.Min(x, y)`	Mínimo entre dos números	`Math.Min(10, 20)`	`10`
`Math.Sin(x)`, `Math.Cos(x)`, `Math.Tan(x)`	Funciones trigonométricas	`Math.Sin(Math.PI / 2)`	`1.0`

🧱 Operadores de Comparación

📖 Lista de Operadores de Comparación

Operador	Descripción	Ejemplo	Resultado
`==`	Igual a	`5 == 5`	`true`
`!=`	Diferente de	`5 != 3`	`true`
`>`	Mayor que	`10 > 7`	`true`
`<`	Menor que	`4 < 9`	`true`
`>=`	Mayor o igual que	`6 >= 6`	`true`
`<=`	Menor o igual que	`3 <= 5`	`true`

🧱 Operadores Lógicos

📖 Lista de Operadores Lógicos

Operador	Descripción	Ejemplo	Resultado
`&&`	**AND (y lógico)** - Devuelve `true` si ambas expresiones son `true`	`(5 > 3) && (8 < 10)`	`true`
`||`	**OR (o lógico)** - Devuelve `true` si al menos una expresión es `true`		
`!`	**NOT (negación lógica)** - Invierte el valor booleano	`!(5 > 3)`	`false`

🧱 Estructuras de Control: Condicionales

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 2 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. 📖

📖 If

En C#, la estructura de control `if` se utiliza para ejecutar un bloque de código únicamente cuando se cumple una condición específica.

1️⃣ `if` Simple

```
if (condición)
{
    // Código a ejecutar si la condición es verdadera
}
```

2 `if` - `else`

```
if (condición)
{
    // Código si la condición es verdadera
}
else
{
    // Código si la condición es falsa
}
```

---

3 `if` - `else if` - `else` (Múltiples Condiciones)

```
if (condición1)
{
    // Código si condición1 es verdadera
}
else if (condición2)
{
    // Código si condición2 es verdadera
}
else
{
    // Código si ninguna de las condiciones anteriores se cumple
}
```

4 `if` Anidado

Se pueden anidar múltiples `if` dentro de otros `if`.

```
int edad = 25;
bool tieneLicencia = true;

if (edad >= 18)
{
    if (tieneLicencia)
    {
        Console.WriteLine("Puedes conducir.");
    }
    else
    {
        Console.WriteLine("Necesitas una licencia para conducir.");
    }
}
else
{
    Console.WriteLine("Eres menor de edad, no puedes conducir.");
}
```

5️⃣ `if` con Operadores Lógicos

Se pueden combinar múltiples condiciones con `&&` (AND) y `||` (OR).

```
int edad = 22;
bool estudiante = true;

if (edad >= 18 && estudiante)
{
    Console.WriteLine("Tienes descuento por ser estudiante mayor de edad.");
}
```

6️⃣ `if` en una sola línea (Operador Ternario `? :`)

El operador condicional `?:`, conocido como operador ternario, evalúa una expresión booleana y devuelve uno de dos posibles resultados según si la condición es `true` o `false`.

```
string mensaje = (edad >= 18) ? "Mayor de edad" : "Menor de edad";
Console.WriteLine(mensaje);
```

📖 Switch

El `switch` en C# es una estructura de control que permite ejecutar diferentes bloques de código según el valor de una variable. Es una alternativa más clara y eficiente que una serie de `if-else if`.

🏷️ Sintaxis del `switch`

```
switch (expresión)
{
    case valor1:
        // Código a ejecutar si expresión == valor1
        break;
    case valor2:
        // Código a ejecutar si expresión == valor2
        break;
    default:
        // Código si ningún caso coincide
        break;
}
```

📌 **Importante:**

- Cada `case` **debe terminar con** `break`, a menos que se use una estructura diferente.
- El `default` es **opcional**, pero recomendable para manejar casos no previstos.

🏷️ Uso de `switch` con `when` (Filtros en C# 7.0+)

A partir de C# 7, `switch` permite evaluar condiciones adicionales con `when`.

```
internal class Program
{
    private static void Main(string[] args)
    {
        int edad = 20;

        switch (edad)
        {
            case int e when (e < 18):
                Console.WriteLine("Eres menor de edad.");
                break;
            case int e when (e >= 18 && e < 60):
                Console.WriteLine("Eres adulto.");
                break;
            default:
                Console.WriteLine("Eres adulto mayor.");
                break;
        }

    }
}
```

El `switch` evalúa la variable `edad` y clasifica a la persona en **menor de edad**, **adulto** o **adulto mayor** usando `when` para establecer condiciones adicionales.

📌 **Explicación rápida de cada caso:**

1. Si `edad < 18` → Imprime `"Eres menor de edad."`
2. Si `edad >= 18 && edad < 60` → Imprime `"Eres adulto."`
3. Si `edad >= 60` (por defecto) → Imprime `"Eres adulto mayor."`

La variable `e` dentro de cada `case` es una **variable de patrón** que captura el valor de `edad` y lo usa dentro de la condición `when`.

📌 **Explicación paso a paso:**

```
case int e when (e < 18):
```

- `int e` → Captura el valor de `edad` en la variable `e`.
- `when (e < 18)` → Evalúa si `e` es menor de 18.

```
case int e when (e >= 18 && e < 60):
```

- `int e` captura `edad` en `e`.
- `when (e >= 18 && e < 60)` evalúa si la edad está entre 18 y 59.

🔁 Estructuras de Iterativas: Ciclos

Las **estructuras repetitivas** permiten ejecutar un bloque de código varias veces mientras se cumpla una condición.

📖 Tipos de Bucles en C#

Bucle	Uso Principal	Característica
`for`	Iteraciones controladas con contador	Se ejecuta un número determinado de veces
`while`	Repetición mientras se cumpla una condición	Evalúa la condición **antes** de ejecutar
`do-while`	Similar a `while`, pero garantiza al menos una ejecución	Evalúa la condición **después** de ejecutar
`foreach`	Recorre colecciones (`arrays`, `List<T>`, etc.)	Itera sobre elementos sin necesidad de índice

📖 Bucle `for` (Controlado por un Contador)

El **bucle** `for` se usa cuando conocemos el número exacto de iteraciones que deben ejecutarse. Se basa en un **contador** que cambia en cada iteración.

🖥️ Sintaxis

```
for (inicialización; condición; actualización)
{
    // Código que se ejecuta en cada iteración
}

Ejemplo

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

📝 Explicación de sus partes

- **Inicialización:** Se ejecuta **una sola vez** antes de que comience el bucle (ejemplo: `int i = 0`).
- **Condición:** Se evalúa **antes de cada iteración**. Si es `true`, el bloque de código se ejecuta; si es `false`, el bucle termina.
- **Actualización:** Se ejecuta **al final de cada iteración** (ejemplo: `i++`).

📖 Bucle `while` (Ejecución Basada en una Condición)

El **bucle** `while` se usa cuando **no sabemos cuántas veces** debe repetirse el código, pero queremos que se ejecute **mientras se cumpla una condición**.

🖥️ Sintaxis

```
while (condición)
{
    // Código que se ejecuta mientras la condición sea verdadera
}
```

📝 Explicación

- La **condición** se evalúa **antes** de cada iteración.
- Si la condición es `true`, el código dentro del `while` se ejecuta.
- Si la condición es `false`, el bucle termina.

📖 Bucle `do..while` (Ejecución Basada en una Condición)

En **C#**, el ciclo `do..while` es una estructura de control de flujo que ejecuta un bloque de código al menos una vez y luego evalúa una condición para determinar si debe repetirse.

Características principales:

1. **Ejecución garantizada al menos una vez**: A diferencia de `while`, el bloque de código dentro de `do` siempre se ejecuta al menos una vez, ya que la condición se evalúa después de la primera iteración.
2. **Verificación de condición al final**: Después de ejecutar el bloque de código, el `while` evalúa la condición. Si es `true`, el ciclo se repite; si es `false`, el bucle se detiene.

Ejemplo

```
internal class Program
{
    private static void Main(string[] args)
    {
        int numero;
        int suma = 0;

        do
        {
            Console.WriteLine("Ingrese un número (ingrese un número negativo para
terminar):");
            numero = Convert.ToInt32(Console.ReadLine());

            if (numero >= 0)
            {
                suma += numero;
                Console.WriteLine($"Suma actual: {suma}");
            }
```

```
        } while (numero >= 0);

        Console.WriteLine($"Programa terminado. La suma total fue: {suma}");
    }
}
```

> 📝 Este ejemplo demuestra un uso común del bucle do..while:
>
> El programa ejecutará el código dentro del do al menos una vez
>
> Después de cada ejecución, verifica la condición en el while
>
> Continuará ejecutándose mientras el número ingresado sea mayor o igual a 0
>
> Cuando el usuario ingrese un número negativo, el bucle terminará
>
> La diferencia principal entre while y do..while es que do..while siempre ejecuta el código al menos una vez, mientras que while verifica la condición antes de la primera ejecución.

Diferencia entre `while` y `do..while`

Característica	`while`	`do..while`
Verificación de condición	Antes de ejecutar el bloque	Después de ejecutar el bloque
Número mínimo de ejecuciones	Puede ser `0` si la condición es falsa desde el inicio	Siempre ejecuta al menos 1 vez
Uso común	Cuando se necesita verificar antes de ejecutar	Cuando se necesita ejecutar al menos una vez antes de verificar

📖 Bucle `foreach` (Ejecución Basada en una Condición)

El ciclo `foreach` en **C#** es una estructura de control que permite recorrer de manera sencilla los elementos de una colección o arreglo, sin necesidad de gestionar índices manualmente.

📖Diferencias entre `for` y `foreach`

Característica	`for`	`foreach`
Uso de índice	Sí, se necesita un índice (`i`)	No, accede directamente a los elementos
Modificación de elementos	Sí, es posible modificar los valores en la colección	No se pueden modificar elementos directamente
Simplicidad	Más complejo, pero flexible	Más simple y legible para iterar

---

📖 Cuándo usar `foreach`

✅ Cuando se quiere recorrer una colección sin preocuparse por los índices.
✅ Cuando se necesita escribir código más limpio y legible.
✅ Cuando no se requiere modificar los elementos de la colección.

```
internal class Program
{
    private static void Main(string[] args)
    {
        // Creamos un array de frutas
        string[] frutas = { "Manzana", "Banana", "Naranja", "Pera", "Uva" };

        Console.WriteLine("Lista de frutas disponibles:");

        foreach (string fruta in frutas)
        {
            Console.WriteLine($"- {fruta}");
        }

        // También podemos usar foreach con una lista de números
        int[] numeros = { 1, 2, 3, 4, 5 };
        int suma = 0;

        foreach (int numero in numeros)
        {
            suma += numero;
            Console.WriteLine($"Sumando {numero}, la suma actual es: {suma}");
        }

        Console.WriteLine($"\nLa suma total es: {suma}");
    }
}
```

> 📝Este ejemplo muestra dos usos comunes de foreach:
>
> Primero, recorre un array de strings (frutas) e imprime cada elemento, Luego, recorre un array de números, sumándolos e imprimiendo el progreso
>
> Las ventajas de usar foreach son:
>
> - Es más simple y legible que un for tradicional
> - No necesitas manejar índices
> - Es menos propenso a errores
> - Es ideal para recorrer colecciones cuando no necesitas saber la posición del elemento
>
> El foreach se puede usar con cualquier colección que implemente IEnumerable, como:
>
> - Arrays
> - List
> - Dictionary<K,V>
> - HashSet

> - Y otras colecciones de C#

🧮 Arreglos (Arrays) en C#

En **C#**, un **array** (o arreglo) es una estructura de datos que permite almacenar múltiples valores del mismo tipo en una sola variable. Se accede a los elementos mediante índices numéricos, comenzando desde `0`.

📖 Declaración y Creación de un Array

Declaración e Inicialización Simultánea

```
int[] numeros = { 10, 20, 30, 40, 50 };
```

📝 **Explicación:**

- Se declara un array `numeros` de tipo `int` y se inicializa con valores `{10, 20, 30, 40, 50}`.

📖 Declaración con Tamaño Fijo e Inicialización Posterior

```
int[] edades = new int[3]; // Se define un array de 3 elementos
edades[0] = 25;
edades[1] = 30;
edades[2] = 35;
```

📝 **Explicación:**

- Se crea un array `edades` con **3 posiciones**.
- Se asignan valores usando **índices** (`0`, `1`, `2`).

📖 Declaración con `new` y Elementos

```
string[] nombres = new string[] { "Ana", "Luis", "Pedro" };
```

📝 **Explicación:**

- Similar a la **Forma 1**, pero se usa `new string[]`.

📖 Acceder a los Elementos del Array

Para acceder a un elemento, se usa su índice:

```
Console.WriteLine(numeros[0]); // Imprime 10
Console.WriteLine(nombres[1]); // Imprime "Luis"
```

📖 Recorrer un Array

Para recorrer los elementos, se usan estructuras de control como `for` o `foreach`.

📝Usando `for`

```
for (int i = 0; i < numeros.Length; i++)
{
    Console.WriteLine(numeros[i]);
}
```

📌 Explicación:

- `numeros.Length` devuelve la cantidad de elementos en el array.

📝Usando `foreach` (Recomendado para solo lectura)

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

📌 Explicación:

- `foreach` simplifica la iteración al acceder directamente a los valores.

📖 Modificar un Array

```
numeros[2] = 100; // Cambia el valor en la posición 2
Console.WriteLine(numeros[2]); // Imprime 100
```

📖 Arrays Multidimensionales

En C#, puedes tener **arrays bidimensionales (matrices)** y de más dimensiones.

📝Array Bidimensional (Matriz)

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

Console.WriteLine(matriz[1, 2]); // Imprime 6
```

📌 Explicación:

- `matriz[1,2]` accede a la **segunda fila, tercera columna**.

📝 Arrays Irregulares (Jagged Arrays)

Son arrays donde cada fila tiene diferente cantidad de columnas.

```
int[][] jaggedArray = new int[3][];
jaggedArray[0] = new int[] { 1, 2, 3 };
jaggedArray[1] = new int[] { 4, 5 };
jaggedArray[2] = new int[] { 6, 7, 8, 9 };

Console.WriteLine(jaggedArray[2][3]); // Imprime 9
```

📌 **Explicación:**

- `jaggedArray[2][3]` accede a la **tercera fila, cuarta columna**.

📖 Métodos Útiles para Arrays

C# proporciona métodos útiles en la clase `Array`:

Método	Descripción
`Array.Sort(array)`	Ordena el array en orden ascendente
`Array.Reverse(array)`	Invierte los elementos del array
`Array.IndexOf(array, valor)`	Retorna la posición del valor
`Array.Resize(ref array, nuevoTamaño)`	Cambia el tamaño del array

Ejemplo: Ordenar y Buscar en un Array

```
int[] numeros = { 30, 10, 50, 20, 40 };
Array.Sort(numeros); // Ordena el array
Console.WriteLine(string.Join(", ", numeros)); // Imprime: 10, 20, 30, 40, 50

int posicion = Array.IndexOf(numeros, 30);
Console.WriteLine($"El 30 está en la posición {posicion}"); // Imprime 2
```

Ejemplo

```
internal class Program
{
    private static void Main(string[] args)
    {
        // Declaración e inicialización de un array
        int[] numeros = { 5, 2, 8, 1, 9, 3, 7, 4, 6 };

        Console.WriteLine("Array original:");
        MostrarArray(numeros);

        // Ordenar el array
        Array.Sort(numeros);
```

```
        Console.WriteLine("\nArray ordenado:");
        MostrarArray(numeros);

        // Invertir el array
        Array.Reverse(numeros);
        Console.WriteLine("\nArray invertido:");
        MostrarArray(numeros);

        // Buscar elementos
        int numeroBuscado = 7;
        int posicion = Array.IndexOf(numeros, numeroBuscado);
        Console.WriteLine($"\nEl número {numeroBuscado} está en la posición:
{posicion}");

        // Redimensionar el array
        Array.Resize(ref numeros, numeros.Length + 2);
        numeros[numeros.Length - 2] = 10;
        numeros[numeros.Length - 1] = 11;
        Console.WriteLine("\nArray redimensionado:");
        MostrarArray(numeros);

        // Crear una copia del array
        int[] copiaArray = new int[numeros.Length];
        Array.Copy(numeros, copiaArray, numeros.Length);
        Console.WriteLine("\nCopia del array:");
        MostrarArray(copiaArray);

        // Obtener máximo y mínimo
        int maximo = numeros.Max();
        int minimo = numeros.Min();
        Console.WriteLine($"\nValor máximo: {maximo}");
        Console.WriteLine($"Valor mínimo: {minimo}");
    }

    // Método auxiliar para mostrar el array
    private static void MostrarArray(int[] array)
    {
        foreach (int numero in array)
        {
            Console.Write($"{numero} ");
        }
        Console.WriteLine();
    }
}
```

Este ejemplo muestra varios métodos importantes para trabajar con arrays:

- Array.Sort(): Ordena los elementos del array
- Array.Reverse(): Invierte el orden de los elementos
- Array.IndexOf(): Busca un elemento y devuelve su posición
- Array.Resize(): Cambia el tamaño del array
- Array.Copy(): Crea una copia del array
- Max() y Min(): Obtienen el valor máximo y mínimo

Además, el ejemplo incluye:

- Declaración e inicialización de arrays
- Un método auxiliar para mostrar el array
- Uso de foreach para recorrer el array
- Acceso a elementos por índice

📖 Conclusiones

✅ Un array en C# es una estructura que **almacena elementos del mismo tipo**.
✅ Se accede mediante índices y **siempre inicia en** `0`.
✅ Puede ser **unidimensional, multidimensional o irregular**.
✅ La clase `Array` proporciona métodos útiles como `Sort()`, `Reverse()`, `IndexOf()`.

📮 Programación Avanzada C#

🎆 Programación orientada a Objetos

📖 Introducción a la Programación Orientada a Objetos (POO) en C# 🚀

La **Programación Orientada a Objetos (POO)** es un paradigma de programación basado en la creación y manipulación de **objetos**. Se utiliza para modelar el mundo real de una manera más estructurada, facilitando el mantenimiento, reutilización y escalabilidad del código.

📖 Conceptos Claves de POO

La POO se basa en **cuatro principios fundamentales**:

Concepto	Descripción
Encapsulamiento	Restringe el acceso a los datos y métodos para proteger la integridad del objeto.
Herencia	Permite que una clase derive propiedades y métodos de otra, evitando duplicación de código.
Polimorfismo	Habilidad de un objeto para tomar múltiples formas, permitiendo diferentes implementaciones de un mismo método.
Abstracción	Oculta detalles innecesarios y expone solo lo relevante de un objeto.

📖 Clases y Objetos

Una **clase** es una plantilla para crear objetos. Un **objeto** es una instancia de una clase.

Ejemplo: Definir una clase en C#

```
class Estudiante
{
    // Campos privados
    private string nombre;
    private int edad;
    private double[] calificaciones;

    // Constructor
    public Estudiante(string nombre, int edad)
    {
        this.nombre = nombre;
        this.Edad = edad;
        this.calificaciones = new double[0];
    }

}

internal class Program
{
    private static void Main(string[] args)
    {
        // Crear una instancia de la clase Estudiante
        Estudiante estudiante1 = new Estudiante("Ana García", 20);

    }
}
```

📌 **Explicación:**

- Se define la clase `Persona` con atributos (`Nombre`, `Edad`) y un método (`Presentarse`).
- Se crea un objeto `persona1`, se asignan valores y se llama al método `Presentarse`.

📖 Encapsulamiento 🔒

Permite ocultar los detalles internos de un objeto y controlar el acceso a sus atributos.

Ejemplo usando `private` y `public` con métodos `get` y `set`

```
class Estudiante
{
    // Campos privados
    private string nombre;
    private int edad;
    private double[] calificaciones;

    // Propiedades públicas
    public string Nombre
    {
        get { return nombre; }
        set { nombre = value; }
    }
```

```
    public int Edad
    {
        get { return edad; }
        set
        {
            if (value >= 0)
                edad = value;
            else
                throw new ArgumentException("La edad no puede ser negativa");
        }
    }

    // Constructor
    public Estudiante(string nombre, int edad)
    {
        this.nombre = nombre;
        this.Edad = edad;
        this.calificaciones = new double[0];
    }

    // Métodos públicos
    public void AgregarCalificacion(double calificacion)
    {
        Array.Resize(ref calificaciones, calificaciones.Length + 1);
        calificaciones[calificaciones.Length - 1] = calificacion;
    }

    public double CalcularPromedio()
    {
        if (calificaciones.Length == 0)
            return 0;
        return calificaciones.Average();
    }

    public string ObtenerResumen()
    {
        return $"Estudiante: {nombre}\n" +
               $"Edad: {edad} años\n" +
               $"Promedio: {CalcularPromedio():F2}";
    }
}

internal class Program
{
    private static void Main(string[] args)
    {
        // Crear una instancia de la clase Estudiante
        Estudiante estudiante1 = new Estudiante("Ana García", 20);

        // Agregar algunas calificaciones
        estudiante1.AgregarCalificacion(8.5);
        estudiante1.AgregarCalificacion(9.0);
        estudiante1.AgregarCalificacion(7.8);

        // Mostrar información del estudiante
```

```
        Console.WriteLine(estudiante1.ObtenerResumen());

        // Crear otro estudiante
        Estudiante estudiante2 = new Estudiante("Carlos López", 22);
        estudiante2.AgregarCalificacion(6.5);
        estudiante2.AgregarCalificacion(8.0);

        // Cambiar el nombre usando la propiedad
        estudiante2.Nombre = "Carlos Rodríguez";

        Console.WriteLine("\n" + estudiante2.ObtenerResumen());
    }
}
```

📌 **Encapsulamiento con `get` y `set` protege el acceso a la variable `edad`.**

📖 Herencia 👨‍👧

Permite que una clase (hija) herede atributos y métodos de otra clase (padre), reutilizando código.

Ejemplo: Herencia en C#

```
          [Clase Base: Animal]
                  |
       -------------------------------------
       |                 |                 |
[Clase Derivada]  [Clase Derivada]  [Clase Derivada]
    Perro             Gato              Pájaro
 hacerSonido()     hacerSonido()     hacerSonido()
 "Guau guau"       "Miau miau"       "Pío pío"
```

```
public abstract class Animal
{
    // Propiedades comunes para todos los animales
    public string Nombre { get; set; }
    public int Edad { get; set; }

    // Constructor de la clase base
    public Animal(string nombre, int edad)
    {
        Nombre = nombre;
        Edad = edad;
    }

    // Método abstracto que deberán implementar las clases derivadas
    public abstract string HacerSonido();

    // Método común para todos los animales
    public string ObtenerInformacion()
    {
        return $"{GetType().Name} - Nombre: {Nombre}, Edad: {Edad} años";
```

```
    }
}

// Clase derivada Perro
public class Perro : Animal
{
    public Perro(string nombre, int edad) : base(nombre, edad)
    {
    }

    public override string HacerSonido()
    {
        return "Guau guau";
    }
}

// Clase derivada Gato
public class Gato : Animal
{
    public Gato(string nombre, int edad) : base(nombre, edad)
    {
    }

    public override string HacerSonido()
    {
        return "Miau miau";
    }
}

// Clase derivada Pájaro
public class Pajaro : Animal
{
    public Pajaro(string nombre, int edad) : base(nombre, edad)
    {
    }

    public override string HacerSonido()
    {
        return "Pío pío";
    }
}

internal class Program
{
    private static void Main(string[] args)
    {
        // Crear instancias de diferentes animales
        Perro perro = new Perro("Max", 3);
        Gato gato = new Gato("Luna", 2);
        Pajaro pajaro = new Pajaro("Piolín", 1);

        // Crear un array de animales
        Animal[] animales = { perro, gato, pajaro };

        // Mostrar información y sonidos de cada animal
        Console.WriteLine("Información de los animales:\n");
```

```
        foreach (Animal animal in animales)
        {
            Console.WriteLine(animal.ObtenerInformacion());
            Console.WriteLine($"Sonido: {animal.HacerSonido()}\n");
        }
    }
}
```

📌 **Explicación:** Este ejemplo demuestra varios conceptos importantes de la programación orientada a objetos:

Clase Abstracta:

Animal es una clase base abstracta

- Define un método abstracto HacerSonido() que las clases derivadas deben implementar

Herencia:

- Las clases Perro, Gato y Pajaro heredan de Animal Cada clase derivada implementa su propia versión de HacerSonido()

Polimorfismo:

- Podemos tratar todos los objetos como Animal
- Cada objeto mantiene su comportamiento específico

Encapsulamiento:

- Las propiedades están encapsuladas usando propiedades auto-implementadas
- La lógica común está en la clase base

Constructores:

- Cada clase derivada llama al constructor de la clase base usando base

---

📖 Polimorfismo 🎭

Permite que métodos con el mismo nombre actúen de forma diferente según la clase.

Ejemplo: Sobreescritura de Métodos (`override`)

```
public abstract class Forma
{
    public string Color { get; set; }

    protected Forma(string color)
    {
        Color = color;
    }

    // Métodos abstractos que deben implementar las clases derivadas
    public abstract double CalcularArea();
    public abstract double CalcularPerimetro();
```

```
    // Método virtual que puede ser sobrescrito
    public virtual string Describir()
    {
        return $"Soy una forma de color {Color}";
    }
}

// Clase derivada Círculo
public class Circulo : Forma
{
    public double Radio { get; set; }

    public Circulo(string color, double radio) : base(color)
    {
        Radio = radio;
    }

    public override double CalcularArea()
    {
        return Math.PI * Radio * Radio;
    }

    public override double CalcularPerimetro()
    {
        return 2 * Math.PI * Radio;
    }

    public override string Describir()
    {
        return $"Soy un círculo {Color} con radio {Radio}";
    }
}

// Clase derivada Rectángulo
public class Rectangulo : Forma
{
    public double Base { get; set; }
    public double Altura { get; set; }

    public Rectangulo(string color, double baseRect, double altura) : base(color)
    {
        Base = baseRect;
        Altura = altura;
    }

    public override double CalcularArea()
    {
        return Base * Altura;
    }

    public override double CalcularPerimetro()
    {
        return 2 * (Base + Altura);
    }

    public override string Describir()
```

```
    {
        return $"Soy un rectángulo {Color} de {Base}x{Altura}";
    }
}

// Clase derivada Triángulo
public class Triangulo : Forma
{
    public double Lado1 { get; set; }
    public double Lado2 { get; set; }
    public double Lado3 { get; set; }

    public Triangulo(string color, double lado1, double lado2, double lado3) :
base(color)
    {
        Lado1 = lado1;
        Lado2 = lado2;
        Lado3 = lado3;
    }

    public override double CalcularArea()
    {
        // Fórmula de Herón
        double s = (Lado1 + Lado2 + Lado3) / 2;
        return Math.Sqrt(s * (s - Lado1) * (s - Lado2) * (s - Lado3));
    }

    public override double CalcularPerimetro()
    {
        return Lado1 + Lado2 + Lado3;
    }

    public override string Describir()
    {
        return $"Soy un triángulo {Color} con lados {Lado1}, {Lado2} y {Lado3}";
    }
}

internal class Program
{
    private static void Main(string[] args)
    {
        // Crear un array de formas
        Forma[] formas = new Forma[]
        {
            new Circulo("rojo", 5),
            new Rectangulo("azul", 4, 6),
            new Triangulo("verde", 3, 4, 5),
            new Circulo("amarillo", 3)
        };

        // Demostrar polimorfismo procesando cada forma
        Console.WriteLine("Demostración de Polimorfismo:\n");

        foreach (Forma forma in formas)
        {
```

```
            // El comportamiento específico se determina en tiempo de ejecución
            Console.WriteLine(forma.Describir());
            Console.WriteLine($"Área: {forma.CalcularArea():F2}");
            Console.WriteLine($"Perímetro: {forma.CalcularPerimetro():F2}");
            Console.WriteLine();
        }

        // Demostrar polimorfismo con método
        Console.WriteLine("Procesando formas específicas:\n");
        ProcesarForma(new Circulo("morado", 2));
        ProcesarForma(new Rectangulo("naranja", 3, 7));
    }

    // Método que acepta cualquier tipo de Forma
    private static void ProcesarForma(Forma forma)
    {
        Console.WriteLine($"Procesando: {forma.GetType().Name}");
        Console.WriteLine(forma.Describir());
        Console.WriteLine($"Área: {forma.CalcularArea():F2}");
        Console.WriteLine();
    }
}
```

📌 **Explicación:**

```
Explicación detallada del código:

Estructura de Clases:
    - Forma: Clase base abstracta que define la estructura común para todas las
formas
    - Circulo, Rectangulo, Triangulo: Clases derivadas que implementan los
métodos abstractos

Array de Formas:
    Forma[] formas = new Forma[]
    {
        new Circulo("rojo", 5),
        new Rectangulo("azul", 4, 6),
        new Triangulo("verde", 3, 4, 5),
        new Circulo("amarillo", 3)
    };
    - Crea un array que puede contener cualquier objeto que herede de Forma
    - Demuestra polimorfismo al almacenar diferentes tipos en el mismo array

Bucle foreach:
    foreach (Forma forma in formas)
    {
        Console.WriteLine(forma.Describir());
        Console.WriteLine($"Área: {forma.CalcularArea():F2}");
        Console.WriteLine($"Perímetro: {forma.CalcularPerimetro():F2}");
        Console.WriteLine();
    }
    - foreach itera sobre cada elemento del array automáticamente
    - Forma forma declara una variable que contendrá cada elemento
    - En cada iteración, forma puede ser un Circulo, Rectangulo o Triangulo.
```

```
    - Los métodos llamados (Describir(), CalcularArea(), CalcularPerimetro()) se
ejecutan según el tipo real del objeto.

Método ProcesarForma:
    private static void ProcesarForma(Forma forma)
    {
        Console.WriteLine($"Procesando: {forma.GetType().Name}");
        Console.WriteLine(forma.Describir());
        Console.WriteLine($"Área: {forma.CalcularArea():F2}");
        Console.WriteLine();
    }
    - Acepta cualquier objeto que herede de Forma
    - forma.GetType().Name obtiene el nombre real de la clase del objeto.
    - Demuestra polimorfismo al procesar cualquier tipo de forma de manera
uniforme;Las ventajas de usar foreach en este caso
    son:
        - Código más limpio y legible
        - Menos propenso a errores (no hay que manejar índices)
        - Más idiomático en C# para iterar colecciones
        - Automáticamente maneja la longitud de la colección

El polimorfismo permite que el mismo código funcione con diferentes tipos de
objetos, haciendo el código más flexible y extensible.
```

📖 Abstracción 📦

Oculta los detalles internos y expone solo lo esencial.

Ejemplo con Clases Abstractas

```
class Figura
{
    public abstract double CalcularArea(); // Método abstracto (sin
implementación)
}

class Circulo : Figura
{
    private double radio;

    public Circulo(double radio)
    {
        this.radio = radio;
    }

    public override double CalcularArea()
    {
        return Math.PI * radio * radio;
    }
}

class Program
{
    static void Main()
```

```
    {
        Figura miFigura = new Circulo(5);
        Console.WriteLine("Área del círculo: " + miFigura.CalcularArea());
    }
}
```

📌 **Explicación:**

- `abstract` obliga a que `CalcularArea()` sea implementado en las clases hijas (`Circulo`).

📖 Conclusión 🏆

✅ **POO** en **C#** permite escribir código más modular, mantenible y reutilizable.
✅ Los **objetos** son instancias de **clases** y pueden tener atributos y métodos.
✅ **Encapsulamiento** protege los datos y controla el acceso a los mismos.
✅ **Herencia** evita duplicación de código al permitir que una clase herede de otra.
✅ **Polimorfismo** permite que diferentes clases implementen métodos de formas variadas.
✅ **Abstracción** expone solo lo esencial y oculta detalles innecesarios.

🧮 Listas

En C#, una **Lista** (`List<T>`) es una colección genérica que permite almacenar elementos de un mismo tipo y proporciona métodos útiles para manipularlos. Es parte del **espacio de nombres** `System.Collections.Generic` y es más flexible que un **array**, ya que permite agregar y eliminar elementos dinámicamente.

📖 Definición

```
public class List<T> : System.Collections.Generic.ICollection<T>,
System.Collections.Generic.IEnumerable<T>, System.Collections.Generic.IList<T>,
System.Collections.Generic.IReadOnlyCollection<T>,
System.Collections.Generic.IReadOnlyList<T>, System.Collections.IList
```

📖 Constructores

List()	**Inicializa una nueva instancia de la clase List que está vacía y tiene la capacidad inicial predeterminada.**
List(IEnumerable))	Inicializa una nueva instancia de la clase List que contiene elementos copiados de la colección especificada y tiene capacidad suficiente para acomodar el número de elementos copiados.
List(Int32)	Inicializa una nueva instancia de la clase List que está vacía y tiene la capacidad inicial especificada.

Obtenido: https://learn.microsoft.com/es-es/dotnet/api/system.collections.generic.list-1?view=net-8.0#definition

📖 Propiedades

Capacity	**Obtiene o establece el número total de elementos que la estructura de datos interna puede contener sin cambiar el tamaño.**
Count	Obtiene el número de elementos contenidos en el List.
[ItemInt32]	Obtiene o establece el elemento en el índice especificado.

Obtenido: https://learn.microsoft.com/es-es/dotnet/api/system.collections.generic.list-1?view=net-8.0#definition

📖 Métodos

Add(T)	**Agrega un objeto al final del List.**
AddRange(IEnumerable)	Agrega los elementos de la colección especificada al final del List.
AsReadOnly()	Devuelve un contenedor de ReadOnlyCollection de solo lectura para la colección actual.
BinarySearch(Int32, Int32, T, IComparer)	Busca un intervalo de elementos en el List ordenado para un elemento mediante el comparador especificado y devuelve el índice de base cero del elemento.
BinarySearch(T)	Busca en toda la List ordenada un elemento mediante el comparador predeterminado y devuelve el índice de base cero del elemento.
BinarySearch(T, IComparer)	Busca en todo el List ordenado de un elemento mediante el comparador especificado y devuelve el índice de base cero del elemento.
Clear()	Quita todos los elementos de la List.
Contains(T)	Determina si un elemento está en el List.
ConvertAll(Converter)	Convierte los elementos de la List actual en otro tipo y devuelve una lista que contiene los elementos convertidos.
CopyTo(Int32, T[])	Copia un intervalo de elementos de la List a una matriz unidimensional compatible, empezando por el índice especificado de la matriz de destino.
CopyTo(T[])	Copia toda la List en una matriz unidimensional compatible, comenzando al principio de la matriz de destino.
CopyTo(T[], Int32)	Copia toda la List en una matriz unidimensional compatible, empezando por el índice especificado de la matriz de destino.
EnsureCapacity(Int32)	Garantiza que la capacidad de esta lista sea al menos la `capacity` especificada. Si la capacidad actual es menor que `capacity`, se aumenta a al menos el `capacity` especificado.

Add(T)	**Agrega un objeto al final del List.**
Equals(Object)	Determina si el objeto especificado es igual al objeto actual. (Heredado de Object)
Exists(Predicate)	Determina si el List contiene elementos que coinciden con las condiciones definidas por el predicado especificado.
Find(Predicate)	Busca un elemento que coincida con las condiciones definidas por el predicado especificado y devuelve la primera aparición dentro de toda la List.
FindAll(Predicate)	Recupera todos los elementos que coinciden con las condiciones definidas por el predicado especificado.
FindIndex(Int32, Int32, Predicate)	Busca un elemento que coincida con las condiciones definidas por el predicado especificado y devuelve el índice de base cero de la primera aparición dentro del intervalo de elementos del List que comienza en el índice especificado y contiene el número especificado de elementos.
FindIndex(Int32, Predicate)	Busca un elemento que coincida con las condiciones definidas por el predicado especificado y devuelve el índice de base cero de la primera aparición dentro del intervalo de elementos del List que se extiende del índice especificado al último elemento.
FindIndex(Predicate)	Busca un elemento que coincida con las condiciones definidas por el predicado especificado y devuelve el índice de base cero de la primera aparición dentro de toda la List.
FindLast(Predicate)	Busca un elemento que coincida con las condiciones definidas por el predicado especificado y devuelve la última aparición dentro de toda la List.
FindLastIndex(Int32, Int32, Predicate)	Busca un elemento que coincida con las condiciones definidas por el predicado especificado y devuelve el índice de base cero de la última aparición dentro del intervalo de elementos de la List que contiene el número especificado de elementos y termina en el índice especificado.
FindLastIndex(Int32, Predicate)	Busca un elemento que coincida con las condiciones definidas por el predicado especificado y devuelve el índice de base cero de la última aparición dentro del intervalo de elementos del List que se extiende desde el primer elemento al índice especificado.
FindLastIndex(Predicate)	Busca un elemento que coincida con las condiciones definidas por el predicado especificado y devuelve el índice de base cero de la última aparición en toda la List.
ForEach(Action)	Realiza la acción especificada en cada elemento del List.
GetEnumerator()	Devuelve un enumerador que recorre en iteración el List.

Add(T)	**Agrega un objeto al final del List.**
GetHashCode()	Actúa como función hash predeterminada.(Heredado de Object)
GetRange(Int32, Int32)	Crea una copia superficial de un intervalo de elementos en el Listde origen .
GetType()	Obtiene el Type de la instancia actual.(Heredado de Object)
IndexOf(T)	Busca el objeto especificado y devuelve el índice de base cero de la primera aparición dentro de toda la List.
IndexOf(T, Int32)	Busca el objeto especificado y devuelve el índice de base cero de la primera aparición dentro del intervalo de elementos del List que se extiende desde el índice especificado hasta el último elemento.
IndexOf(T, Int32, Int32)	Busca el objeto especificado y devuelve el índice de base cero de la primera aparición dentro del intervalo de elementos del List que comienza en el índice especificado y contiene el número especificado de elementos.
Insert(Int32, T)	Inserta un elemento en el List en el índice especificado.
InsertRange(Int32, IEnumerable)	Inserta los elementos de una colección en el List en el índice especificado.
LastIndexOf(T)	Busca el objeto especificado y devuelve el índice de base cero de la última aparición en toda la List.
LastIndexOf(T, Int32)	Busca el objeto especificado y devuelve el índice de base cero de la última aparición dentro del intervalo de elementos del List que se extiende desde el primer elemento al índice especificado.
LastIndexOf(T, Int32, Int32)	Busca el objeto especificado y devuelve el índice de base cero de la última aparición dentro del intervalo de elementos del List que contiene el número especificado de elementos y termina en el índice especificado.
MemberwiseClone()	Crea una copia superficial del Objectactual.(Heredado de Object)
Remove(T)	Quita la primera aparición de un objeto específico de la List.
RemoveAll(Predicate)	Quita todos los elementos que coinciden con las condiciones definidas por el predicado especificado.
RemoveAt(Int32)	Quita el elemento en el índice especificado del List.
RemoveRange(Int32, Int32)	Quita un intervalo de elementos de la List.
Reverse()	Invierte el orden de los elementos en toda la List.

Add(T)	**Agrega un objeto al final del List.**
Reverse(Int32, Int32)	Invierte el orden de los elementos del intervalo especificado.
Slice(Int32, Int32)	Crea una copia superficial de un intervalo de elementos en el Listde origen .
Sort()	Ordena los elementos de toda la List mediante el comparador predeterminado.
Sort(Comparison)	Ordena los elementos de toda la List mediante el Comparisonespecificado.
Sort(IComparer)	Ordena los elementos de toda la List mediante el comparador especificado.
Sort(Int32, Int32, IComparer)	Ordena los elementos de un intervalo de elementos de List utilizando el comparador especificado.
ToArray()	Copia los elementos del List en una nueva matriz.
ToString()	Devuelve una cadena que representa el objeto actual. (Heredado de Object)
TrimExcess()	Establece la capacidad en el número real de elementos de la List, si ese número es menor que un valor de umbral.
TrueForAll(Predicate)	Determina si cada elemento de la List coincide con las condiciones definidas por el predicado especificado.

Obtenido: https://learn.microsoft.com/es-es/dotnet/api/system.collections.generic.list-1?view=net-8.0#definition

```
public abstract class Forma
{
    public string Color { get; set; }

    protected Forma(string color)
    {
        Color = color;
    }

    // Métodos abstractos que deben implementar las clases derivadas
    public abstract double CalcularArea();
    public abstract double CalcularPerimetro();

    // Método virtual que puede ser sobrescrito
    public virtual string Describir()
    {
        return $"Soy una forma de color {Color}";
    }
}

// Clase derivada Círculo
public class Circulo : Forma
```

```
{
    public double Radio { get; set; }

    public Circulo(string color, double radio) : base(color)
    {
        Radio = radio;
    }

    public override double CalcularArea()
    {
        return Math.PI * Radio * Radio;
    }

    public override double CalcularPerimetro()
    {
        return 2 * Math.PI * Radio;
    }

    public override string Describir()
    {
        return $"Soy un círculo {Color} con radio {Radio}";
    }
}

// Clase derivada Rectángulo
public class Rectangulo : Forma
{
    public double Base { get; set; }
    public double Altura { get; set; }

    public Rectangulo(string color, double baseRect, double altura) : base(color)
    {
        Base = baseRect;
        Altura = altura;
    }

    public override double CalcularArea()
    {
        return Base * Altura;
    }

    public override double CalcularPerimetro()
    {
        return 2 * (Base + Altura);
    }

    public override string Describir()
    {
        return $"Soy un rectángulo {Color} de {Base}x{Altura}";
    }
}

// Clase derivada Triángulo
public class Triangulo : Forma
{
    public double Lado1 { get; set; }
```

```
    public double Lado2 { get; set; }
    public double Lado3 { get; set; }

    public Triangulo(string color, double lado1, double lado2, double lado3) :
base(color)
    {
        Lado1 = lado1;
        Lado2 = lado2;
        Lado3 = lado3;
    }

    public override double CalcularArea()
    {
        // Fórmula de Herón
        double s = (Lado1 + Lado2 + Lado3) / 2;
        return Math.Sqrt(s * (s - Lado1) * (s - Lado2) * (s - Lado3));
    }

    public override double CalcularPerimetro()
    {
        return Lado1 + Lado2 + Lado3;
    }

    public override string Describir()
    {
        return $"Soy un triángulo {Color} con lados {Lado1}, {Lado2} y {Lado3}";
    }
}

internal class Program
{
    private static void Main(string[] args)
    {
        // Crear una lista de formas en lugar de un array
        List<Forma> formas = new List<Forma>
        {
            new Circulo("rojo", 5),
            new Rectangulo("azul", 4, 6),
            new Triangulo("verde", 3, 4, 5),
            new Circulo("amarillo", 3)
        };

        // Demostrar métodos de List
        Console.WriteLine("Demostración de métodos de List:\n");

        // Add - Agregar un nuevo elemento
        formas.Add(new Circulo("morado", 2));
        Console.WriteLine($"Cantidad de formas después de Add: {formas.Count}");

        // Insert - Insertar en una posición específica
        formas.Insert(0, new Rectangulo("naranja", 3, 7));
        Console.WriteLine($"Primera forma después de Insert:
{formas[0].Describir()}");

        // Contains y Exists
        bool existeRojo = formas.Exists(f => f.Color == "rojo");
```

```
        Console.WriteLine($"\n¿Existe una forma roja? {existeRojo}");

        // Find - Encontrar el primer círculo
        Forma? primerCirculo = formas.Find(f => f is Circulo);
        Console.WriteLine($"Primer círculo encontrado:
{primerCirculo?.Describir() ?? "No se encontró ningún círculo"}");

        // FindAll - Encontrar todos los círculos
        List<Forma> todosLosCirculos = formas.FindAll(f => f is Circulo);
        Console.WriteLine($"\nTotal de círculos encontrados:
{todosLosCirculos.Count}");

        // ForEach - Mostrar todos los círculos
        Console.WriteLine("\nTodos los círculos:");
        todosLosCirculos.ForEach(c => Console.WriteLine(c.Describir()));

        // RemoveAll - Eliminar todos los círculos
        int eliminados = formas.RemoveAll(f => f is Circulo);
        Console.WriteLine($"\nFormas eliminadas: {eliminados}");
        Console.WriteLine($"Formas restantes: {formas.Count}");

        // Sort - Ordenar por color
        formas.Sort((f1, f2) => f1.Color.CompareTo(f2.Color));
        Console.WriteLine("\nFormas ordenadas por color:");
        formas.ForEach(f => Console.WriteLine(f.Describir()));

        // Clear - Limpiar la lista
        // formas.Clear();  // Comentar o eliminar esta línea
        Console.WriteLine($"\nFormas después de Clear: {formas.Count}");

        // Demostrar polimorfismo procesando cada forma
        Console.WriteLine("Demostración de Polimorfismo:\n");

        foreach (Forma forma in formas)
        {
            // El comportamiento específico se determina en tiempo de ejecución
            Console.WriteLine(forma.Describir());
            Console.WriteLine($"Área: {forma.CalcularArea():F2}");
            Console.WriteLine($"Perímetro: {forma.CalcularPerimetro():F2}");
            Console.WriteLine();
        }

        // Demostrar polimorfismo con método
        Console.WriteLine("Procesando formas específicas:\n");
        ProcesarForma(new Circulo("morado", 2));
        ProcesarForma(new Rectangulo("naranja", 3, 7));
    }

    // Método que acepta cualquier tipo de Forma
    private static void ProcesarForma(Forma forma)
    {
        Console.WriteLine($"Procesando: {forma.GetType().Name}");
        Console.WriteLine(forma.Describir());
        Console.WriteLine($"Área: {forma.CalcularArea():F2}");
        Console.WriteLine();
    }
```

```
}
```

🧱 ArrayList

Un `ArrayList` en C# es una estructura de datos perteneciente al espacio de nombres `System.Collections`. Se trata de una colección que puede contener elementos de diferentes tipos y ajustar su tamaño dinámicamente, sin necesidad de definir una capacidad fija.

📖 Características de `ArrayList`

- Permite almacenar elementos de **cualquier tipo de datos** (`object`).
- **No es genérico**, a diferencia de `List<T>`, lo que significa que puede almacenar diferentes tipos de datos en la misma colección.
- Puede **cambiar de tamaño dinámicamente**, agregando o eliminando elementos sin necesidad de definir un tamaño fijo.
- Se recomienda **usar `List<T>` en lugar de `ArrayList`**, ya que `List<T>` ofrece mejor rendimiento y seguridad de tipo.

📖 Constructores

ArrayList()	**Inicializa una nueva instancia de la clase ArrayList que está vacía y tiene la capacidad inicial predeterminada.**
ArrayList(ICollection)	Inicializa una nueva instancia de la clase ArrayList que contiene elementos copiados de la colección especificada y que tiene la misma capacidad inicial que el número de elementos copiados.
ArrayList(Int32)	Inicializa una nueva instancia de la clase ArrayList que está vacía y tiene la capacidad inicial especificada.

📖 Propiedades

Capacity	**Obtiene o establece el número de elementos que puede contener ArrayList.**
Count	Obtiene el número de elementos contenido realmente en ArrayList.
IsFixedSize	Obtiene un valor que indica si la interfaz ArrayList tiene un tamaño fijo.
IsReadOnly	Obtiene un valor que indica si ArrayList es de solo lectura.
IsSynchronized	Obtiene un valor que indica si el acceso a la interfaz ArrayList está sincronizado (es seguro para subprocesos).
Item[Int32]	Obtiene o establece el elemento en el índice especificado.
SyncRoot	Obtiene un objeto que se puede usar para sincronizar el acceso a ArrayList.

📖 Principales Métodos de `ArrayList`

Método	Descripción
`Add(object)`	Agrega un elemento al final.
`Insert(index, object)`	Inserta un elemento en una posición específica.
`Remove(object)`	Elimina la primera ocurrencia del elemento.
`RemoveAt(index)`	Elimina el elemento en el índice especificado.
`Clear()`	Elimina todos los elementos de la lista.
`Contains(object)`	Verifica si un elemento existe en la lista.
`IndexOf(object)`	Devuelve el índice de la primera aparición del elemento.
`Sort()`	Ordena los elementos si son comparables.
`Count`	Propiedad que devuelve la cantidad de elementos en la lista.

📖 Diferencia entre `ArrayList` y `List<T>`

Característica	`ArrayList`	`List<T>`
Seguridad de tipo	No (puede contener diferentes tipos)	Sí (almacena un solo tipo de datos)
Rendimiento	Más lento debido a la conversión de tipos	Más rápido gracias a la seguridad de tipos
Uso recomendado	Solo si necesitas almacenar tipos diferentes	Siempre que sepas qué tipo de datos usar

```
using System.Collections;

// Clase base abstracta
public abstract class Forma
{
    public string Color { get; set; }

    protected Forma(string color)
    {
        Color = color;
    }

    // Métodos abstractos que deben implementar las clases derivadas
    public abstract double CalcularArea();
    public abstract double CalcularPerimetro();

    // Método virtual que puede ser sobrescrito
    public virtual string Describir()
    {
```

```
        return $"Soy una forma de color {Color}";
    }
}

// Clase derivada Círculo
public class Circulo : Forma
{
    public double Radio { get; set; }

    public Circulo(string color, double radio) : base(color)
    {
        Radio = radio;
    }

    public override double CalcularArea()
    {
        return Math.PI * Radio * Radio;
    }

    public override double CalcularPerimetro()
    {
        return 2 * Math.PI * Radio;
    }

    public override string Describir()
    {
        return $"Soy un círculo {Color} con radio {Radio}";
    }
}

// Clase derivada Rectángulo
public class Rectangulo : Forma
{
    public double Base { get; set; }
    public double Altura { get; set; }

    public Rectangulo(string color, double baseRect, double altura) : base(color)
    {
        Base = baseRect;
        Altura = altura;
    }

    public override double CalcularArea()
    {
        return Base * Altura;
    }

    public override double CalcularPerimetro()
    {
        return 2 * (Base + Altura);
    }

    public override string Describir()
    {
        return $"Soy un rectángulo {Color} de {Base}x{Altura}";
    }
```

```
}

// Clase derivada Triángulo
public class Triangulo : Forma
{
    public double Lado1 { get; set; }
    public double Lado2 { get; set; }
    public double Lado3 { get; set; }

    public Triangulo(string color, double lado1, double lado2, double lado3) :
base(color)
    {
        Lado1 = lado1;
        Lado2 = lado2;
        Lado3 = lado3;
    }

    public override double CalcularArea()
    {
        // Fórmula de Herón
        double s = (Lado1 + Lado2 + Lado3) / 2;
        return Math.Sqrt(s * (s - Lado1) * (s - Lado2) * (s - Lado3));
    }

    public override double CalcularPerimetro()
    {
        return Lado1 + Lado2 + Lado3;
    }

    public override string Describir()
    {
        return $"Soy un triángulo {Color} con lados {Lado1}, {Lado2} y {Lado3}";
    }
}

internal class Program
{
    private static void Main(string[] args)
    {
        // Crear un ArrayList en lugar de List<Forma>
        ArrayList formas = new ArrayList();

        // Add - Agregar elementos
        formas.Add(new Circulo("rojo", 5));
        formas.Add(new Rectangulo("azul", 4, 6));
        formas.Add(new Triangulo("verde", 3, 4, 5));
        formas.Add(new Circulo("amarillo", 3));

        Console.WriteLine("Demostración de métodos de ArrayList:\n");

        // Count - Mostrar cantidad de elementos
        Console.WriteLine($"Cantidad de elementos: {formas.Count}");

        // Insert - Insertar en una posición específica
        formas.Insert(0, new Rectangulo("naranja", 3, 7));
        var primeraForma = formas[0] as Forma;
```

```
        Console.WriteLine($"Primera forma después de Insert:
{primeraForma?.Describir() ?? "Forma no válida"}");

        // Contains - Verificar si contiene un elemento específico
        var circuloRojo = new Circulo("rojo", 5);
        Console.WriteLine($"\n¿Contiene el círculo rojo?
{formas.Contains(circuloRojo)}");

        // IndexOf - Encontrar la posición de un elemento
        var indice = formas.IndexOf(circuloRojo);
        Console.WriteLine($"Posición del círculo rojo: {indice}");

        // CopyTo - Copiar elementos a un array
        Forma[] arrayFormas = new Forma[formas.Count];
        formas.CopyTo(arrayFormas);
        Console.WriteLine("\nElementos copiados al array:");
        foreach (Forma forma in arrayFormas)
        {
            Console.WriteLine(forma.Describir());
        }

        // Remove - Eliminar un elemento específico
        formas.Remove(circuloRojo);
        Console.WriteLine($"\nCantidad después de Remove: {formas.Count}");

        // RemoveAt - Eliminar elemento en posición específica
        formas.RemoveAt(0);
        Console.WriteLine($"Cantidad después de RemoveAt: {formas.Count}");

        // Recorrer el ArrayList (requiere casting)
        Console.WriteLine("\nTodas las formas restantes:");
        foreach (object obj in formas)
        {
            if (obj is Forma forma)
            {
                Console.WriteLine(forma.Describir());
                Console.WriteLine($"Área: {forma.CalcularArea():F2}");
                Console.WriteLine();
            }
        }

        // Reverse - Invertir el orden de los elementos
        formas.Reverse();
        Console.WriteLine("\nFormas después de Reverse:");
        foreach (Forma forma in formas)
        {
            Console.WriteLine(forma.Describir());
        }

        // Clear - Limpiar el ArrayList
        formas.Clear();
        Console.WriteLine($"\nCantidad después de Clear: {formas.Count}");

        // Demostrar polimorfismo procesando cada forma
        Console.WriteLine("Demostración de Polimorfismo:\n");
```

```
        foreach (Forma forma in formas)
        {
            // El comportamiento específico se determina en tiempo de ejecución
            Console.WriteLine(forma.Describir());
            Console.WriteLine($"Área: {forma.CalcularArea():F2}");
            Console.WriteLine($"Perímetro: {forma.CalcularPerimetro():F2}");
            Console.WriteLine();
        }

        // Demostrar polimorfismo con método
        Console.WriteLine("Procesando formas específicas:\n");
        ProcesarForma(new Circulo("morado", 2));
        ProcesarForma(new Rectangulo("naranja", 3, 7));
    }

    // Método que acepta cualquier tipo de Forma
    private static void ProcesarForma(Forma forma)
    {
        Console.WriteLine($"Procesando: {forma.GetType().Name}");
        Console.WriteLine(forma.Describir());
        Console.WriteLine($"Área: {forma.CalcularArea():F2}");
        Console.WriteLine();
    }
}
```

🧮 HashSet en C#

El `HashSet<T>` es una colección en C# que pertenece al espacio de nombres `System.Collections.Generic`. Se caracteriza por almacenar elementos **únicos**, sin permitir duplicados, y por ofrecer una búsqueda y manipulación de datos de manera eficiente.

📖 Propiedades

Comparer	**Obtiene el objeto IEqualityComparer que se usa para determinar la igualdad de los valores del conjunto.**
Count	Obtiene el número de elementos contenidos en un conjunto.

📖 Principales Métodos de `HashSet<T>`

Método	**Descripción**
`Add(T item)`	Agrega un elemento si no está en el conjunto.
`Remove(T item)`	Elimina un elemento del conjunto.
`Contains(T item)`	Verifica si el elemento existe en el conjunto.
`Clear()`	Elimina todos los elementos.
`Count`	Devuelve el número de elementos en el conjunto.
`UnionWith(HashSet<T>)`	Une dos conjuntos, eliminando duplicados.

Método	Descripción
`IntersectWith(HashSet<T>)`	Mantiene solo los elementos que están en ambos conjuntos.
`ExceptWith(HashSet<T>)`	Elimina del conjunto los elementos que están en otro conjunto.

📖 Diferencias entre `HashSet<T>` y `List<T>`

Característica	`HashSet<T>`	`List<T>`
Permite duplicados	❌ No	✅ Sí
Orden de los elementos	❌ No garantizado	✅ Sí, conserva el orden de inserción
Rendimiento en búsquedas	🚀 Muy rápido (`O(1)`)	🐢 Más lento (`O(n)`)
Uso recomendado	Cuando se necesita evitar duplicados y mejorar la velocidad de búsqueda	Cuando se requiere mantener el orden de los elementos

📖 Características principales de `HashSet<T>`

- **No permite elementos duplicados**.
- **No garantiza un orden específico** de los elementos.
- **Optimizado para búsquedas rápidas**, ya que internamente usa una tabla hash.
- Admite operaciones de **conjunto matemático** como unión, intersección y diferencia.
- Es una alternativa eficiente a `List<T>` cuando se necesita **evitar duplicados y mejorar el rendimiento** en búsquedas.

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

// Clase base abstracta
class Persona
{
    public string Nombre { get; set; }
    public int Edad { get; set; }
    public string Documento { get; set; } // Usado como identificador único

    public Persona(string nombre, int edad, string documento)
    {
        Nombre = nombre;
        Edad = edad;
        Documento = documento;
    }
```

```
    // Sobrescribimos Equals para comparar por el Documento
    public override bool Equals(object obj)
    {
        if (obj is Persona otraPersona)
        {
            return Documento == otraPersona.Documento;
        }
        return false;
    }

    // Sobrescribimos GetHashCode para asegurar que no haya duplicados
    public override int GetHashCode()
    {
        return Documento.GetHashCode();
    }

    public override string ToString()
    {
        return $"Nombre: {Nombre}, Edad: {Edad}, Documento: {Documento}";
    }
}

internal class Program
{
    private static void Main(string[] args)
    {
        HashSet<Persona> personas = new HashSet<Persona>();

        // Agregar personas al conjunto
        personas.Add(new Persona("Juan", 30, "12345678"));
        personas.Add(new Persona("María", 25, "87654321"));
        personas.Add(new Persona("Carlos", 35, "11223344"));

        Console.WriteLine("Lista de Personas:");
        MostrarPersonas(personas);

        // Editar información de una persona
        Console.WriteLine("\nEditando la información de Juan...");
        EditarPersona(personas, "12345678", "Juan Pérez", 31);

        Console.WriteLine("\nLista de Personas después de la edición:");
        MostrarPersonas(personas);
    }

    // Método para mostrar todas las personas en el HashSet
    static void MostrarPersonas(HashSet<Persona> personas)
    {
        foreach (var persona in personas)
        {
            Console.WriteLine(persona);
        }
    }

    // Método para editar información de una persona
    static void EditarPersona(HashSet<Persona> personas, string documento, string
nuevoNombre, int nuevaEdad)
```

```
    {
        // Buscar la persona por documento
        Persona? personaExistente = null;
        foreach (var persona in personas)
        {
            if (persona.Documento == documento)
            {
                personaExistente = persona;
                break;
            }
        }

        // Si se encontró, eliminarla y agregar la nueva versión
        if (personaExistente != null)
        {
            personas.Remove(personaExistente); // Eliminar persona antigua
            personas.Add(new Persona(nuevoNombre, nuevaEdad, documento)); //
Agregar la versión editada
            Console.WriteLine($"Persona con documento {documento} actualizada con
éxito.");
        }
        else
        {
            Console.WriteLine($"No se encontró ninguna persona con el documento
{documento}.");
        }
    }
}
```

🎆 Expresiones Lambda

Qué es una expresión lambda (Lambda expression)

Una **expresión lambda** es una forma concisa de escribir **métodos anónimos** en C#. Es una **función sin nombre** que podemos usar donde se necesite un **delegado** o una **función anónima**.

💡 **Explicación sencilla:**
Las expresiones lambda **permiten escribir funciones más cortas y legibles** eliminando la necesidad de usar `delegate` explícitamente.

📖 Ejemplo de un Método Anónimo (Sin Lambda)

Antes de conocer las expresiones lambda, en C# se usaban **delegados y métodos anónimos**:

```
Func<int, bool> mayorDeEdad = delegate(int edad) { return edad >= 18; };
```

📌 **Explicación:**

- `Func<int, bool>` → Es un **delegado genérico** que recibe un `int` y devuelve un `bool`.
- `delegate(int edad) { return edad >= 18; };` → Es un **método anónimo** que verifica si la edad es mayor o igual a 18.

📖 Expresión Lambda Equivalente

Podemos escribir el mismo código de forma más compacta con **una expresión lambda**:

```
Func<int, bool> mayorDeEdad = edad => edad >= 18;
```

📌 **Diferencias y simplificaciones:**

1. **Se elimina** `delegate`, ya que el compilador infiere que es una función anónima.
2. **No se necesita especificar el tipo de** `edad`, porque `Func<int, bool>` ya lo define.
3. **Usamos el operador** `=>` **(lambda)** para separar los parámetros del cuerpo de la función.
4. **No hay llaves** `{}` **ni** `return` porque la expresión es de una sola línea.

📖 Ejemplo con Múltiples Parámetros

Si la expresión lambda tiene **más de un parámetro**, deben ir entre paréntesis:

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


internal class Program
{
    private static void Main(string[] args)
    {
        Func<int, int, int> sumar = (a, b) => a + b;
        Console.WriteLine(sumar(5, 3)); // Salida: 8
    }
}
```

📌 **Explicación:**

- `Func<int, int, int>` → Define una función que recibe **dos enteros** y devuelve un **entero**.
- `(a, b) => a + b` → Es una **expresión lambda** que suma `a` y `b`.

📖 Expresiones Lambda con Cuerpo de Bloque

Si la función es más compleja y requiere varias líneas, **podemos usar** `{}` **y** `return`:

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


internal class Program
{
    private static void Main(string[] args)
    {
        Func<int, bool> esPar = numero =>
        {
            Console.WriteLine($"Verificando si {numero} es par...");
            return numero % 2 == 0;
        };
```

```
        Console.WriteLine(esPar(8)); // Salida: Verificando si 8 es par... True
    }
}
```

🎴 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.

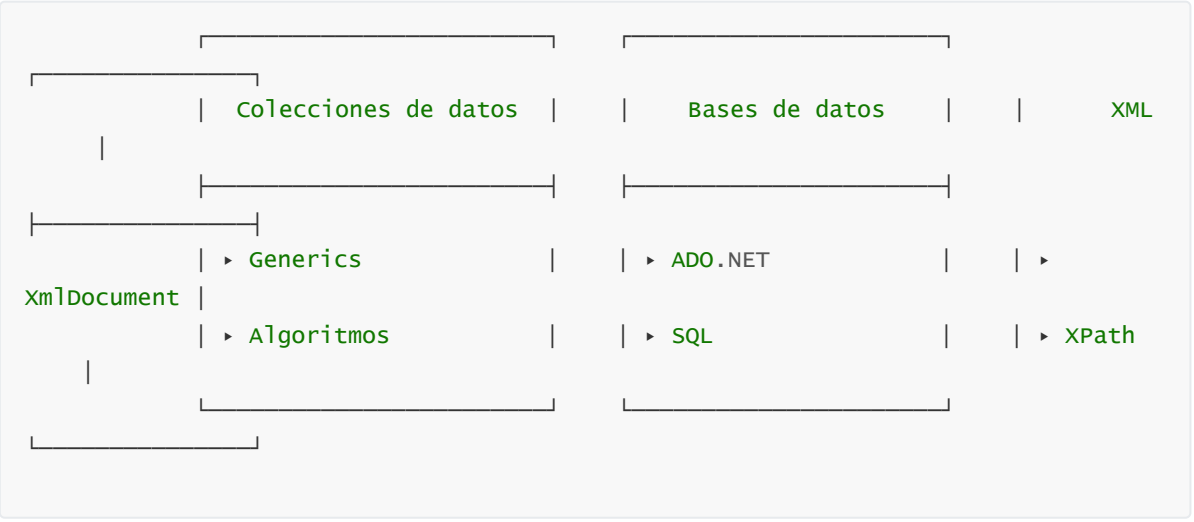
📖 El problema

Los desarrolladores de C# identificaron un problema común al trabajar con datos: necesitaban realizar consultas de manera eficiente, pero no existía una forma unificada y sencilla para hacerlo.

La dificultad principal radicaba en que los datos podían provenir de distintas fuentes, cada una con su propia manera de ser consultada:

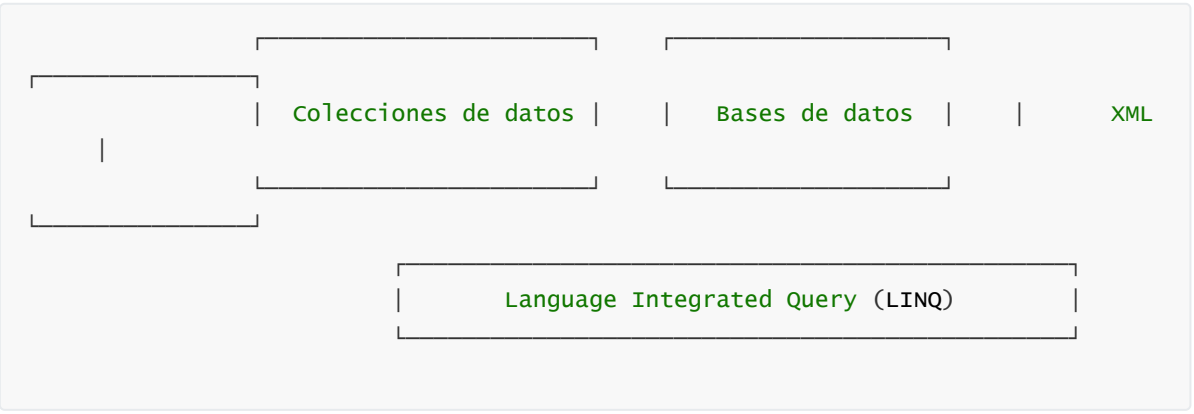
1. **Colecciones en memoria**: Requerían el uso de métodos proporcionados por `Generics` o la implementación de algoritmos específicos para buscar y manipular datos.
2. **Bases de datos**: Era necesario utilizar conectores de `ADO.NET` y escribir consultas SQL manualmente para recuperar información.
3. **Ficheros XML**: Para trabajar con este formato, se necesitaban herramientas como `XmlDocument` o `XPath` para recorrer y extraer datos.

Dado que cada fuente de datos exigía un enfoque diferente, no existía una solución unificada que permitiera consultar datos de manera homogénea y sencilla en C#. Esta necesidad llevó a la creación de **LINQ**, una herramienta que permite consultar diferentes tipos de datos utilizando una sintaxis común y más intuitiva.



Cada una de estas APIs y librerías tiene su propia sintaxis y forma de interactuar con los datos, lo que dificultaba el desarrollo de aplicaciones que trabajaran con múltiples fuentes de información.

Para solucionar este problema, **Microsoft decidió unificar la manera en que se realizan consultas** en diferentes tipos de datos, introduciendo **Language Integrated Query (LINQ)**. Esta tecnología permite escribir consultas de una forma consistente, sin importar si los datos provienen de colecciones en memoria, bases de datos o archivos XML, simplificando así el desarrollo y mantenimiento del código.



LINQ no solo nos permite escribir consultas con una sintaxis unificada, sino que también proporciona **comprobaciones de tipo en tiempo de compilación**, lo que ayuda a detectar errores antes de ejecutar el programa.

Sin embargo, aunque estas consultas se validan durante la compilación, su ejecución real ocurre en **tiempo de ejecución**, interactuando con diferentes tipos de fuentes de datos, como:

- **Colecciones en memoria** (`List<T>`, `Array`, `Dictionary<T, T>`).
- **Bases de datos relacionales**, a través de **Entity Framework** o `ADO.NET`.
- **Documentos XML**, utilizando **LINQ to XML**.

Además, una vez que comprendemos cómo funciona LINQ, podemos expandir su uso a otros tipos de datos, como:

- **Sistema de archivos**, para consultar archivos y directorios.
- **Bases de datos NoSQL**, como **MongoDB**.
- **Ficheros CSV**, para manipular datos tabulares.
- **APIs REST**, al consumir datos de servicios web.

- **Otras estructuras de datos personalizadas**.

LINQ es una herramienta poderosa que nos permite consultar distintos tipos de datos de forma consistente y eficiente. 🚀

📖 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.

📖 Beneficios de LINQ:

- Código más limpio y legible.
- Menos errores, ya que las consultas están tipadas en tiempo de compilación.
- Independencia de la fuente de datos (colecciones, bases de datos, XML, JSON, etc.).
- Soporte para expresiones lambda y funciones anónimas.

📖 Tipos de LINQ

Existen varios tipos de LINQ dependiendo de la fuente de datos que consultes:

Tipo de LINQ	Descripción
LINQ to Objects	Para consultar listas, arreglos, colecciones en memoria (`List<T>`, `Array`, etc.).
LINQ to SQL	Para interactuar con bases de datos SQL usando mapeo objeto-relacional (ORM).
LINQ to Entities	Para trabajar con Entity Framework (similar a LINQ to SQL pero más avanzado).
LINQ to XML	Para consultar y manipular documentos XML.
LINQ to JSON	Para procesar JSON usando `System.Text.Json` o `Newtonsoft.Json`.
PLINQ (Parallel LINQ)	Para ejecutar consultas LINQ en paralelo y mejorar el rendimiento.

📖 Sintaxis de LINQ

LINQ tiene dos formas de escribir consultas:

Sintaxis de consulta (Query Syntax)

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

var pares = from n in numeros
            where n % 2 == 0
            select n;

foreach (var num in pares)
{
    Console.WriteLine(num);
}
```

📖 Consultas

🦴 Clase `Persona`

Esta clase representa una persona con nombre y edad.

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

    public Persona(string nombre, int edad)
    {
        Nombre = nombre;
        Edad = edad;
    }
}
```

🦴 Clase `Estudiante`

Esta clase representa un estudiante con nombre y nota.

```
class Estudiante
{
    public string Nombre { get; set; }
    public int Nota { get; set; }

    public Estudiante(string nombre, int nota)
    {
        Nombre = nombre;
        Nota = nota;
    }
}
```

🦴 Clase `Producto`

Esta clase representa un producto con nombre y precio.

```
class Producto
{
    public string Nombre { get; set; }
    public decimal Precio { get; set; }

    public Producto(string nombre, decimal precio)
    {
        Nombre = nombre;
        Precio = precio;
    }
}
```

🩹 Clase `Empleado`

Esta clase representa un empleado con nombre y departamento.

```
class Empleado
{
    public string Nombre { get; set; }
    public string Departamento { get; set; }

    public Empleado(string nombre, string departamento)
    {
        Nombre = nombre;
        Departamento = departamento;
    }
}
```

📑 ¿Qué es una consulta y cuál es su propósito?

Una consulta es un conjunto de instrucciones diseñadas para **recuperar y organizar datos** desde una o varias fuentes. Estas consultas especifican **qué información obtener** y **cómo estructurarla**, pero es importante notar que una consulta no es lo mismo que los datos resultantes que produce.

Generalmente, los datos de origen se organizan en **secuencias de elementos del mismo tipo**:

- En una **base de datos SQL**, los datos se estructuran en **filas** dentro de una tabla.
- En un **archivo XML**, los datos se almacenan en una jerarquía de **elementos XML** organizados en forma de árbol.
- En una **colección en memoria**, los datos se presentan como una secuencia de **objetos** en una lista o array.

Desde la perspectiva de una aplicación, el formato original de los datos no es relevante, ya que estos se representan en **LINQ** como una colección de tipo `IEnumerable<T>` o `IQueryable<T>`. Por ejemplo, en **LINQ to XML**, los elementos XML se exponen como `IEnumerable<XElement>`.

📑 ¿Qué puede hacer una consulta?

Dependiendo de los requerimientos, una consulta puede realizar tres tareas principales:

🖌️ Filtrar y organizar datos sin modificar los elementos originales

En una consulta LINQ, entre la cláusula `from` (inicio) y `select` o `group` (finalización), podemos incluir **cláusulas opcionales** que nos permiten **filtrar, ordenar y combinar datos**. Estas cláusulas incluyen:

- `where` → Filtra elementos según una condición.
- `orderby` → Ordena los resultados.
- `join` → Une colecciones basadas en una clave común.
- `from` adicional → Permite recorrer colecciones anidadas.
- `let` → Crea una variable intermedia dentro de la consulta.

🔹 **Ejemplo:** Obtener y ordenar calificaciones mayores a 80 en orden descendente.

```
IEnumerable<int> calificacionesAltas =
    from calificacion in calificaciones
    where calificacion > 80
    orderby calificacion descending
    select calificacion;
```

Aquí:

- `where calificacion > 80` filtra solo las calificaciones mayores a 80.
- `orderby calificacion descending` las ordena de mayor a menor.
- `select calificacion` selecciona los valores filtrados.

1️⃣ Filtrar Datos con `where`

🔹 **Ejemplo:** Filtrar países con población mayor a 100,000 habitantes.

```
csharpCopiarEditarvar paisesFiltrados =
    from country in countries
    where country.Population > 100_000
    select country;

foreach (var country in paisesFiltrados)
{
    Console.WriteLine($"{country.Name} - {country.Population} habitantes");
}
```

📌 **Explicación:**

- `where country.Population > 100_000` → Filtra países con más de 100,000 habitantes.
- `select country` → Devuelve el país completo.

🔹 **Salida esperada:**

```
python-replCopiarEditarSaint Vincent & the Grenadines - 110000 habitantes
Grenada - 113000 habitantes
Barbados - 287000 habitantes
...
```

2️⃣ Ordenar Datos con `orderby`

🔹 **Ejemplo:** Ordenar países por población **de mayor a menor**.

```
csharpCopiarEditarvar paisesOrdenados =
    from country in countries
    orderby country.Population descending
    select country;

foreach (var country in paisesOrdenados)
{
    Console.WriteLine($"{country.Name} - {country.Population} habitantes");
}
```

📌 **Explicación:**

- `orderby country.Population descending` → Ordena en orden **descendente** (de mayor a menor).
- Si no usamos `descending`, **se ordena ascendente por defecto**.

🔹 **Salida esperada:**

```
python-replCopiarEditarBahamas - 393000 habitantes
Barbados - 287000 habitantes
Saint Vincent & the Grenadines - 110000 habitantes
...
```

3️⃣ Unir Colecciones con `join`

🔹 **Ejemplo:** Unir productos con sus categorías.

```
csharpCopiarEditarvar productosConCategoria =
    from product in products
    join category in categories on product.Category equals category.Name
    select new { Producto = product.Name, Categoria = category.Name };

foreach (var item in productosConCategoria)
{
    Console.WriteLine($"{item.Producto} pertenece a la categoría
{item.Categoria}");
}
```

📌 **Explicación:**

- `join category in categories on product.Category equals category.Name` → Une productos con su categoría.

- `select new { Producto, Categoria }` → Devuelve un objeto con el nombre del producto y la categoría.

🔹 **Salida esperada:**

```
cssCopiarEditarLaptop pertenece a la categoría Electrónica
Smartwatch pertenece a la categoría Electrónica
Camiseta pertenece a la categoría Ropa
...
```

4️⃣ Recorrer Colecciones Internas con `from` Adicional

🔹 **Ejemplo:** Obtener todas las ciudades de los países.

```
csharpCopiarEditarvar ciudades =
    from country in countries
    from city in country.Cities
    select new { Ciudad = city.Name, Pais = country.Name };

foreach (var ciudad in ciudades)
{
    Console.WriteLine($"{ciudad.Ciudad} está en {ciudad.Pais}");
}
```

📌 **Explicación:**

- **Doble** `from` → Permite recorrer ciudades dentro de cada país.
- `select` devuelve un nuevo objeto con `Ciudad` y `Pais`.

🔹 **Salida esperada:**

```
python-replCopiarEditarAndorra la Vella está en Andorra
Ngerulmud está en Palau
Kingstown está en Saint Vincent & the Grenadines
...
```

5️⃣ Usar `let` para Crear Variables Temporales

🔹 **Ejemplo:** Crear una variable que almacene la densidad de población de cada país.

```
var paisesConDensidad =
    from country in countries
    let densidad = country.Population / country.Area
    where densidad > 100
    select new { country.Name, Densidad = densidad };

foreach (var pais in paisesConDensidad)
{
    Console.WriteLine($"{pais.Name} - Densidad: {pais.Densidad:F2} hab/km²");
}
```

📌 **Explicación:**

- `let densidad = country.Population / country.Area` → Calcula la densidad poblacional.
- `where densidad > 100` → Filtra países con **densidad mayor a 100 habitantes/km²**.
- **El resultado es una lista de países con su densidad.**

🔹 **Salida esperada:**

```
Andorra - Densidad: 164.53 hab/km²
Saint Vincent & the Grenadines - Densidad: 282.26 hab/km²
...
```

🩹 Transformar los datos en otro tipo

En lugar de devolver los datos en su formato original, la consulta puede proyectarlos en un **nuevo tipo de objeto**. Por ejemplo, podemos convertir números en cadenas de texto para hacerlos más comprensibles.

🔹 **Ejemplo:** Convertir calificaciones en mensajes de texto.

```
IEnumerable<string> mensajesCalificaciones =
    from calificacion in calificaciones
    where calificacion > 80
    orderby calificacion descending
    select $"La calificación es {calificacion}";
```

Aquí:

- **Transformamos** cada calificación en un string con el formato `"La calificación es {valor}"`.
- **Se mantiene el filtrado y el orden** de la consulta anterior.

🩹 Obtener un único valor de la fuente de datos

A veces, en lugar de una lista de elementos, es necesario calcular un único valor, como:

- **La cantidad de elementos que cumplen un criterio.**
- **El elemento con el valor más alto o más bajo.**
- **El primer elemento que cumple una condición.**
- **La suma de un conjunto de valores.**

🔹 **Ejemplo:** Contar cuántas calificaciones superan 80.

```
var cantidadCalificacionesAltas = (
    from calificacion in calificaciones
    where calificacion > 80
    select calificacion
).Count();
```

Aquí:

- La consulta selecciona todas las calificaciones mayores a 80.
- `.Count()` cuenta cuántas calificaciones cumplen la condición.

🔹 **Ejemplo Alternativo:** Almacenar la consulta antes de ejecutar el conteo.

```
IEnumerable<int> calificacionesAltasQuery =
    from calificacion in calificaciones
    where calificacion > 80
    select calificacion;

var cantidad = calificacionesAltasQuery.Count();
```

📌 **Diferencia clave:**

- En el primer caso, la consulta se ejecuta inmediatamente dentro de `.Count()`.
- En el segundo caso, la consulta se almacena en `calificacionesAltasQuery` y luego se ejecuta al llamar a `.Count()`.

🩹 Filtrar Datos en una Lista de Personas

Supongamos que tenemos una lista de personas y queremos encontrar aquellas mayores de 30 años.

🔹 **Usando Sintaxis de Consulta**

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


internal class Program
{

    class Persona
    {
        public string Nombre { get; set; }
        public int Edad { get; set; }

        public Persona(string nombre, int edad)
        {
            Nombre = nombre;
            Edad = edad;
        }
    }
    private static void Main(string[] args)
    {
        List<Persona> personas =
        [
            new Persona("Ana", 25),
            new Persona("Carlos", 35),
            new Persona("María", 40),
            new Persona("Pedro", 22)
        ];

        var mayoresDe30 =
            from p in personas
            where p.Edad > 30
            select p;
```

```
        foreach (var persona in mayoresDe30)
        {
            Console.WriteLine($"{persona.Nombre} - {persona.Edad} años");
        }
    }
}
```

🔹 **Usando Expresión Lambda**

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


internal class Program
{

    class Persona
    {
        public string Nombre { get; set; }
        public int Edad { get; set; }

        public Persona(string nombre, int edad)
        {
            Nombre = nombre;
            Edad = edad;
        }
    }
    private static void Main(string[] args)
    {
        List<Persona> personas =
        [
            new Persona("Ana", 25),
            new Persona("Carlos", 35),
            new Persona("María", 40),
            new Persona("Pedro", 22)
        ];

        var mayoresDe30Lambda = personas.Where(p => p.Edad > 30);

        foreach (var persona in mayoresDe30Lambda)
        {
            Console.WriteLine($"{persona.Nombre} - {persona.Edad} años");
        }
    }
}
```

🟢 **Salida esperada:**

```
$ dotnet run (Comando para ejecutar el programa)
Carlos - 35 años
María - 40 años
```

🩹 Seleccionar Solo los Nombres de los Estudiantes

Si queremos obtener solo los nombres de una lista de estudiantes.

🔹 Usando Sintaxis de Consulta

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


internal class Program
{

    class Estudiante
    {
        public string Nombre { get; set; }
        public int Nota { get; set; }

        public Estudiante(string nombre, int nota)
        {
            Nombre = nombre;
            Nota = nota;
        }
    }
    private static void Main(string[] args)
    {
        List<Estudiante> estudiantes =
        [
            new Estudiante("Laura", 95),
            new Estudiante("Jorge", 87),
            new Estudiante("Sofía", 72)
        ];

        var nombresEstudiantes =
            from e in estudiantes
            select e.Nombre;

        foreach (var nombre in nombresEstudiantes)
        {
            Console.WriteLine(nombre);
        }
    }
}
```

🔹 Usando Expresión Lambda

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


internal class Program
{

    class Estudiante
    {
```

```
        public string Nombre { get; set; }
        public int Nota { get; set; }

        public Estudiante(string nombre, int nota)
        {
            Nombre = nombre;
            Nota = nota;
        }
    }
    private static void Main(string[] args)
    {
        List<Estudiante> estudiantes =
        [
            new Estudiante("Laura", 95),
            new Estudiante("Jorge", 87),
            new Estudiante("Sofía", 72)
        ];

        // var nombresEstudiantes =
        //     from e in estudiantes
        //     select e.Nombre;
        var nombresLambda = estudiantes.Select(e => e.Nombre);
        foreach (var nombre in nombresLambda)
        {
            Console.WriteLine(nombre);
        }
    }
}
```

🟢 **Salida esperada:**

```
$ dotnet run
Laura
Jorge
Sofía
```

🩹 3. Ordenar Productos por Precio Descendente

Si queremos ordenar una lista de productos de mayor a menor precio.

🔹 **Usando Sintaxis de Consulta**

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


internal class Program
{
    private static void Main(string[] args)
    {
        List<Producto> productos =
        [
            new Producto ("Laptop", 200),
```

```
            new Producto ("Celular",800),
            new Producto ("Monitor",300)
        ];

        var productosOrdenados =
            from p in productos
            orderby p.Precio descending
            select p;

        foreach (var producto in productosOrdenados)
        {
            Console.WriteLine($"{producto.Nombre} - ${producto.Precio}");
        }
    }
}
```

- **Usando Expresión Lambda**

```
using System;
using baseApp;


internal class Program
{
    private static void Main(string[] args)
    {
        List<Producto> productos =
        [
            new Producto ("Laptop", 200),
            new Producto ("Celular",800),
            new Producto ("Monitor",300)
        ];

        var productosOrdenadosLambda = productos.OrderByDescending(p =>
p.Precio);
        foreach (var producto in productosOrdenadosLambda)
        {
            Console.WriteLine($"{producto.Nombre} - ${producto.Precio}");
        }
    }
}
```

🩹 Contar Cuántos Números Son Pares en un Array

Si queremos contar cuántos números en un array son pares.

- **Usando Sintaxis de Consulta**

```
int[] numeros = { 3, 8, 15, 20, 42, 9 };

var cantidadPares =
    (from n in numeros
    where n % 2 == 0
    select n).Count();

Console.WriteLine($"Cantidad de números pares: {cantidadPares}");
```

🔹 **Usando Expresión Lambda**

```
var cantidadParesLambda = numeros.Count(n => n % 2 == 0);
Console.WriteLine($"Cantidad de números pares: {cantidadParesLambda}");
```

🟢 **Salida esperada:**

```
Cantidad de números pares: 3
```

🩹 Agrupar Empleados por Departamento

Si tenemos una lista de empleados y queremos agruparlos por departamento.

🔹 **Usando Sintaxis de Consulta**

```
using System;
using baseApp;


internal class Program
{
    private static void Main(string[] args)
    {
        List<Empleado> empleados = new List<Empleado>(
        [
            new Empleado ("Andrés","TI" ),
            new Empleado ("Luisa","RRHH"),
            new Empleado ("Carlos","TI"),
            new Empleado ("Mariana","RRHH")
        ]);

        var empleadosPorDepartamento =
            from e in empleados
            group e by e.Departamento into grupo
            select grupo;

        foreach (var grupo in empleadosPorDepartamento)
        {
            Console.WriteLine($"Departamento: {grupo.Key}");
            foreach (var empleado in grupo)
            {
                Console.WriteLine($" - {empleado.Nombre}");
            }
        }
```

```
    }
}
```

🔹 **Usando Expresión Lambda**

```
using System;
using baseApp;


internal class Program
{
    private static void Main(string[] args)
    {
        List<Empleado> empleados = new List<Empleado>(
        [
            new Empleado ("Andrés","TI" ),
            new Empleado ("Luisa","RRHH"),
            new Empleado ("Carlos","TI"),
            new Empleado ("Mariana","RRHH")
        ]);

        var empleadosPorDepartamentoLambda = empleados.GroupBy(e =>
e.Departamento);

        foreach (var grupo in empleadosPorDepartamentoLambda)
        {
            Console.WriteLine($"Departamento: {grupo.Key}");
            foreach (var empleado in grupo)
            {
                Console.WriteLine($" - {empleado.Nombre}");
            }
        }
    }
}
```

🟢 **Salida esperada:**

```
$ dotnet run
Departamento: TI
 - Andrés
 - Carlos
Departamento: RRHH
 - Luisa
 - Mariana
```

📖 ¿Expresión de consulta en C#?

Una **expresión de consulta** en C# es una manera de escribir consultas utilizando una sintaxis similar a SQL, lo que facilita la manipulación y filtrado de datos de una manera **declarativa**. Estas expresiones están fuertemente integradas en el lenguaje, lo que significa que pueden usarse como cualquier otra expresión en C#.

Una **expresión de consulta en LINQ** se compone de varias cláusulas que definen cómo se extraen y organizan los datos.

🦫 Características de una expresión de consulta

1. **Se escribe en sintaxis declarativa**, parecida a SQL o XQuery.
2. **Debe comenzar con** `from` para definir la fuente de datos.
3. **Debe terminar con** `select` **o** `group`, que especifica los resultados.
4. Puede contener cláusulas opcionales

   como:

   - `where`: Filtra los datos.
   - `orderby`: Ordena los resultados.
   - `join`: Une colecciones.
   - `let`: Declara variables dentro de la consulta.
   - `into`: Permite continuar con una consulta después de `join` o `group`.

📖 Uso de diferentes cláusulas en expresiones de consulta

🦫 Filtrar elementos con `where`

Queremos obtener estudiantes con notas mayores a 80.

```
var estudiantes =
    from e in listaEstudiantes
    where e.Nota > 80
    select e;
```

🦫 Ordenar elementos con `orderby`

Ordenamos los empleados por edad.

```
var empleadosOrdenados =
    from e in listaEmpleados
    orderby e.Edad descending
    select e;
```

🦫 Agrupar elementos con `group`

Agrupamos productos por categoría.

```
var productosPorCategoria =
    from p in listaProductos
    group p by p.Categoria into grupo
    select grupo;
```

🩹 Declarar variables dentro de la consulta con `let`

Creamos una variable `nombreMayusculas` que contiene el nombre en mayúsculas.

```
var nombresEnMayusculas =
    from p in listaPersonas
    let nombreMayusculas = p.Nombre.ToUpper()
    select nombreMayusculas;
```

🩹 Usar `join` para unir dos colecciones

Unimos `clientes` con `pedidos` en base al `IdCliente`.

```
var clientesConPedidos =
    from c in clientes
    join p in pedidos on c.Id equals p.IdCliente
    select new { c.Nombre, p.FechaPedido };
```

📖 ¿Qué es una variable de consulta en LINQ?

Una **variable de consulta** es aquella que almacena una consulta en lugar de los resultados de la misma.

Ejemplo:

```
var consulta = from e in listaEmpleados where e.Edad > 30 select e;
```

📌 **Nota:**

- La consulta **no se ejecuta inmediatamente**.
- Se ejecutará cuando la recorramos con `foreach` o llamemos a un método como `.ToList()`.

Data de Ejercicios prácticos

1. Defina modelos City, Country

   Modelo City

   ```
   namespace baseApp.Models
   {
       public record City(string Name, long Population);
   }
   ```

   Modelo Country

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

namespace baseApp.Models
{
    public record Country(string Name, double Area, long Population,
List<City> Cities)
    {
        public double PopulationDensity => Population / Area;
        public City? LargestCity => Cities.MaxBy(c => c.Population);
    }
}
```

2. Cree la clase GeographyData

```
using baseApp.Models;
using System.Collections.Generic;

namespace baseApp.Data
{
    public static class GeographyData
    {
        private const int MIN_POPULATION = 10_000;

        public static readonly IReadOnlyList<City> Cities =
        [
            new("Jakarta", 35_400_000),
            new("Seoul", 25_600_000),
            new("Mexico City", 21_900_000),
            new("Los Angeles", 19_500_000),
            new("London", 14_200_000),
            new("Bangkok", 10_500_000),
            new("Moscow", 12_600_000),
            new("Paris", 11_100_000),
            new("Berlin", 6_000_000),
            new("Lima", 9_900_000),
            new("Sydney", 5_500_000),
            new("Toronto", 6_900_000),
            new("Madrid", 6_700_000),
            new("Rome", 4_300_000),
            new("Johannesburg", 5_800_000),
            new("Singapore", 5_900_000),
            new("Hong Kong", 7_500_000),
            new("Buenos Aires", 15_500_000),
            new("Melbourne", 5_100_000),
            new("Dubai", 4_900_000)
        ];

        public static readonly IReadOnlyList<Country> Countries =
        [
            new("Andorra", 468, 77_000, [new("Andorra la Vella", 22_000)]),
            new("Palau", 459, 18_000, [new("Ngerulmud", 400)]),
            new("Saint Vincent & the Grenadines", 389, 110_000,
[new("Kingstown", 25_000)]),
            new("Grenada", 344, 113_000, [new("Saint George's", 38_000)]),
```

```
            new("Barbados", 431, 287_000, [new("Bridgetown", 110_000)]),
            new("Antigua & Barbuda", 443, 98_000, [new("Saint John's",
27_000)]),
            new("Malta", 316, 514_000, [new("Valletta", 6_000)]),
            new("Bahamas", 13_878, 393_000, [new("Nassau", 250_000)]),
            new("Liechtenstein", 160, 39_000, [new("Vaduz", 5_500)]),
            new("San Marino", 61, 34_000, [new("San Marino", 4_000)]),
            new("Tuvalu", 26, 11_500, [new("Funafuti", 6_500)]),
            new("Nauru", 21, 10_800, [new("Yaren", 1_000)]),
            new("Monaco", 2.02, 39_000, [new("Monte Carlo", 39_000)]),
            new("Vatican City", 0.44, 825, [new("Vatican City", 825)]),
            new("Marshall Islands", 181, 58_000, [new("Majuro", 28_000)]),
            new("Saint Kitts & Nevis", 261, 53_000, [new("Basseterre",
13_000)])

        ];
    }
}
```

Una variable de consulta puede almacenar una consulta expresada en sintaxis de consulta, en sintaxis de método o en una combinación de ambas. En los ejemplos siguientes, `queryMajorCities` y `queryMajorCities2` son variables de consulta:

```
using System;
using System.Linq;
using baseApp.Models;
using baseApp.Data;
namespace baseApp
{
    internal class Program
    {
        private static void Main(string[] args)
        {
        //Query syntax
            IEnumerable<City> queryMajorCities =
                from city in GeographyData.Cities
                where city.Population > 30_000_000
                select city;

            // Execute the query to produce the results
            foreach (City city in queryMajorCities)
            {
                Console.WriteLine(city);
            }
            // Method-based syntax
            IEnumerable<City> queryMajorCities2 =
GeographyData.Cities.Where(c => c.Population > 30_000_000);
            // Execute the query to produce the results
            foreach (City city in queryMajorCities2)
            {
                Console.WriteLine(city);
            }
            // Output:
            // City { Name = Tokyo, Population = 37833000 }
            // City { Name = Delhi, Population = 30290000 }
```

```
        }
    }
}
```

📖 Variables que No Son de Consulta en LINQ

En **LINQ**, una **variable de consulta** es aquella que almacena una consulta en sí misma, en lugar de los resultados generados por la consulta. Sin embargo, si una variable almacena los **resultados de una consulta**, ya no es una variable de consulta, sino una **colección materializada**.

```
var highestScore = (
    from score in scores
    select score
).Max();

// or split the expression
IEnumerable<int> scoreQuery =
    from score in scores
    select score;

var highScore = scoreQuery.Max();
// the following returns the same result
highScore = scores.Max();
```

```
var largeCitiesList = (
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city
).ToList();

// or split the expression
IEnumerable<City> largeCitiesQuery =
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city;
var largeCitiesList2 = largeCitiesQuery.ToList();
```

Obtenido: https://learn.microsoft.com/es-es/dotnet/csharp/linq/get-started/query-expression-basics

📖 Iniciar una expresión de consulta

Cláusula `from`

En LINQ, una **expresión de consulta** siempre comienza con la cláusula `from`, que:

- **Define el origen de datos**.
- **Declara una variable de rango** que representa cada elemento de la secuencia de origen.
- **Permite acceder a las propiedades del elemento** gracias al tipado fuerte.

📌 **Ejemplo: Consulta sobre una Lista de Países**

```
var consulta =
    from country in GeographyData.Countries
    select country.Name;

foreach (var name in consulta)
{
    Console.WriteLine(name);
}
```

📌 **Explicación:**

- `from country in GeographyData.Countries` → `country` es la **variable de rango**, que representa cada elemento de `countries`.
- `select country.Name` → Devuelve solo los nombres de los países.
- **El resultado se imprime con** `foreach`, lo que activa la ejecución de la consulta.

📖 Finalizar una Expresión de Consulta en LINQ

Una **expresión de consulta** en LINQ debe finalizar con una cláusula `group` o `select`. Ambas determinan el formato de los resultados:

- `select` → Devuelve una **secuencia de valores**.
- `group` → Devuelve una **secuencia de grupos de elementos organizados por una clave**.

1️⃣ Usando `group` para Agrupar por la Primera Letra del Nombre de un País

La cláusula `group` en LINQ permite **organizar los datos en grupos** basados en una clave específica.

- La **clave de agrupación** puede ser de cualquier tipo (string, número, objeto, etc.).
- Devuelve una **secuencia de grupos**, donde **cada grupo contiene elementos que comparten la misma clave**.

```
using System;
using System.Linq;
using baseApp.Models;
using baseApp.Data;
namespace baseApp
{
    internal class Program
    {
        private static void Main(string[] args)
        {
        //Query syntax
            var gruposPorLetra =
                from country in GeographyData.Countries
                group country by country.Name[0] into grupo
                select grupo;

            foreach (var grupo in gruposPorLetra)
            {
                Console.WriteLine($"Países que comienzan con '{grupo.Key}':");
                foreach (var country in grupo)
                {
```

```
                    Console.WriteLine($" - {country.Name}");
                }
            }

        }
    }
}
```

2️⃣ Usando `select` para Devolver una Lista de Datos

La cláusula `select` en LINQ se usa para **generar diferentes tipos de secuencias** a partir de un origen de datos.

- Una **consulta simple** con `select` devuelve **los mismos objetos** que están en la fuente de datos.
- También podemos **transformar los datos** antes de devolverlos, creando secuencias de diferentes tipos.

```
using System;
using System.Linq;
using baseApp.Models;
using baseApp.Data;
namespace baseApp
{
    internal class Program
    {
        private static void Main(string[] args)
        {
        //Query syntax
        var nombresPaises =
            from country in GeographyData.Countries
            select country.Name;

        foreach (var nombre in nombresPaises)
        {
            Console.WriteLine(nombre);
        }

        }
    }
}
```

📖Transformación de Datos con `select` en LINQ (Proyección)

En LINQ, **proyectar** significa transformar los datos de origen en una secuencia de **nuevos tipos**. Esto se logra con la cláusula `select`, que permite:

- **Extraer campos específicos** de un objeto.
- **Modificar los datos antes de devolverlos**.
- **Crear nuevos tipos de datos**, como objetos anónimos.

1️⃣ Proyección a Tipos Anónimos

🔹 **Ejemplo:** Extraer solo el nombre y la población de los países.

```
using System;
using System.Linq;
using baseApp.Models;
using baseApp.Data;
namespace baseApp
{
    internal class Program
    {
        private static void Main(string[] args)
        {
        //Query syntax
        var paisesProyectados =
            from country in GeographyData.Countries
            select new { Nombre = country.Name, Poblacion = country.Population };

        foreach (var pais in paisesProyectados)
        {
            Console.WriteLine($"{pais.Nombre} - {pais.Poblacion} habitantes");
        }

        }
    }
}
```

📌 **Explicación:**

- `select new { Nombre = country.Name, Poblacion = country.Population }`
  - **Crea un nuevo tipo anónimo** con solo `Nombre` y `Poblacion`.
- **La consulta devuelve** `IEnumerable<anonimo>`.
- **No modifica los datos originales**, solo los filtra.

2️⃣ Proyección con Cálculo en `select`

🔹 **Ejemplo:** Calcular la densidad poblacional (`Población / Área`).

```
using System;
using System.Linq;
using baseApp.Models;
using baseApp.Data;
namespace baseApp
{
    internal class Program
    {
        private static void Main(string[] args)
        {
        //Query syntax
        var paisesProyectados =
            from country in GeographyData.Countries
            select new
```

```
            {
                Nombre = country.Name,
                Densidad = country.Population / country.Area
            };

        foreach (var pais in paisesProyectados)
        {
            Console.WriteLine($"{pais.Nombre} - Densidad: {pais.Densidad:F2}
hab/km²");
        }

        }
    }
}
```

3️⃣ Proyección desde una Lista de Productos

🔹 **Ejemplo:** Extraer solo los nombres en mayúsculas y su categoría.

```
var productosProyectados =
    from product in products
    select new { Nombre = product.Name.ToUpper(), Categoria = product.Category };

foreach (var producto in productosProyectados)
{
    Console.WriteLine($"{producto.Nombre} - Categoría: {producto.Categoria}");
}
```

📌 **Explicación:**

- `product.Name.ToUpper()` convierte los nombres a **mayúsculas**.
- Se devuelve un objeto anónimo con `Nombre` y `Categoria`.

4️⃣ Proyección desde una Lista de Ciudades

🔹 **Ejemplo:** Extraer las ciudades y formatearlas con un mensaje.

```
var ciudadesProyectadas =
    from country in countries
    from city in country.Cities
    select new
    {
        Ciudad = city.Name,
        Pais = country.Name,
        Info = $"{city.Name}, {country.Name} tiene {city.Population} habitantes."
    };

foreach (var ciudad in ciudadesProyectadas)
{
    Console.WriteLine(ciudad.Info);
}
```

📌 **Explicación:**

- **Doble** `from` → Se accede a ciudades dentro de cada país.
- **Se devuelve un objeto con** `Ciudad`, `Pais` **e** `Info`.
- `Info` **contiene un mensaje formateado**.

📖 Uso de `into` en LINQ para Continuaciones de Consulta

La palabra clave `into` en LINQ permite **almacenar el resultado de una consulta en una variable temporal**, lo que permite **realizar operaciones adicionales** en la consulta. Se usa principalmente en:

- **Agrupaciones (`group`)** → Para realizar más operaciones sobre los grupos.
- **Selecciones (`select`)** → Para seguir manipulando los datos sin finalizar la consulta.

```
using System;
using System.Linq;
using baseApp.Models;
using baseApp.Data;
namespace baseApp
{
    internal class Program
    {
        private static void Main(string[] args)
        {
        //Query syntax
        // percentileQuery is an IEnumerable<IGrouping<int, Country>>
        var percentileQuery =
            from country in GeographyData.Countries
            let percentile = (int)country.Population / 1_000
            group country by percentile into countryGroup
            where countryGroup.Key >= 20
            orderby countryGroup.Key
            select countryGroup;

            // grouping is an IGrouping<int, Country>
            foreach (var grouping in percentileQuery)
            {
                Console.WriteLine(grouping.Key);
                foreach (var country in grouping)
                {
                    Console.WriteLine(country.Name + ":" + country.Population);
                }
            }

        }
    }
}
```

📖 Subconsultas en LINQ

En LINQ, una **subconsulta** es una expresión de consulta anidada dentro de otra consulta.

- Cada subconsulta **tiene su propia cláusula** `from`.
- **Puede usar diferentes fuentes de datos** en comparación con la consulta principal.
- Se utilizan para **filtrar, transformar o calcular datos** dentro de una consulta más grande.

1️⃣ Subconsulta para Filtrar Países Basado en su Ciudad más Poblada

🔹 **Ejemplo:** Obtener países cuya ciudad más grande tiene más de 100,000 habitantes.

```
var paisesConCiudadesGrandes =
    from country in countries
    where (from city in country.Cities where city.Population > 100_000 select
city).Any()
    select country;

foreach (var country in paisesConCiudadesGrandes)
{
    Console.WriteLine($"{country.Name} tiene una ciudad con más de 100,000
habitantes.");
}
```

2️⃣ Subconsulta para Obtener la Ciudad más Poblada de Cada País

🔹 **Ejemplo:** Devolver solo la ciudad más grande de cada país.

```
var ciudadMasPobladaPorPais =
    from country in countries
    select new
    {
        Pais = country.Name,
        CiudadMasPoblada = (from city in country.Cities
                            orderby city.Population descending
                            select city).First()
    };

foreach (var item in ciudadMasPobladaPorPais)
{
    Console.WriteLine($"{item.Pais}: {item.CiudadMasPoblada.Name}
({item.CiudadMasPoblada.Population} habitantes)");
}
```

3️⃣ Subconsulta para Obtener Categorías con Productos Caros

🔹 **Ejemplo:** Obtener solo las categorías donde existe **al menos un producto que cuesta más de 500**.

```
var categoriasConProductosCaros =
    from category in categories
    where (from product in products where product.Category == category.Name &&
product.Price > 500 select product).Any()
    select category;

foreach (var category in categoriasConProductosCaros)
{
    Console.WriteLine($"La categoría {category.Name} tiene al menos un producto
que cuesta más de $500.");
}
```

📖 Operaciones set [C#]

Los métodos `Distinct`, `Except`, `Intersect`, `Union` y sus variantes `By` se utilizan en **LINQ** para **realizar operaciones sobre conjuntos de datos** en C#.

1️⃣ `Distinct` y `DistinctBy` (Eliminar Duplicados)

📦 [a, b, b, c, d, c] → [a, b, c, d]

- `Distinct()` → Elimina duplicados de una colección basada en igualdad completa.
- `DistinctBy(selector)` → Elimina duplicados basándose en una propiedad específica.

```
var categoriasUnicas = products.Select(p => p.Category).Distinct();

foreach (var categoria in categoriasUnicas)
{
    Console.WriteLine(categoria);
}
```

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

class Program
{
    static void Main()
    {
        // Lista con duplicados
        List<string> letras = new List<string> { "a", "b", "b", "c", "d", "c" };

        // Usando Distinct para eliminar duplicados
        var resultado = letras.Distinct();

        // Mostrar resultado
        Console.WriteLine($"Entrada:  {string.Join(", ", letras)}");
        Console.WriteLine($"Salida:   {string.Join(", ", resultado)}");
    }
}
```

2️⃣ `Except` y `ExceptBy` (Diferencia de Conjuntos)

- `Except(otraLista)` → Devuelve elementos **únicos** de la primera colección que **no están en la segunda**.
- `ExceptBy(otraLista, selector)` → Compara basándose en una propiedad específica.

🔹 **Ejemplo: Obtener productos que no están en una lista de discontinuados**

```
var productosDisponibles = products.Except(discontinuedProducts);

foreach (var product in productosDisponibles)
{
    Console.WriteLine(product.Name);
}
```

🔹 **Ejemplo con `ExceptBy` (Comparando por categoría)**

```
var categoriasActivas = products.ExceptBy(discontinuedProducts.Select(p =>
p.Category), p => p.Category);

foreach (var product in categoriasActivas)
{
    Console.WriteLine($"{product.Name} - {product.Category}");
}
```

📌 **Explicación:**

- `ExceptBy(discontinuedProducts.Select(p => p.Category), p => p.Category)`
    - **Elimina productos de categorías que ya no están disponibles**.

3️⃣ `Intersect` y `IntersectBy` (Intersección de Conjuntos)

- `Intersect(otraLista)` → Devuelve los elementos que **existen en ambas listas**.
- `IntersectBy(otraLista, selector)` → Compara basándose en una propiedad.

🔹 **Ejemplo: Obtener productos que están en oferta y en la lista de populares**

```
var productosPopulares =
    products.Intersect(offerProducts);

foreach (var product in productosPopulares)
{
    Console.WriteLine(product.Name);
}
```

4️⃣ `Union` y `UnionBy` (Unión de Conjuntos)

- `Union(otraLista)` → Une dos listas **sin duplicados**.
- `UnionBy(otraLista, selector)` → Une dos listas basándose en una propiedad.

```
var productosTotales = products.Union(newProducts);

foreach (var product in productosTotales)
{
    Console.WriteLine(product.Name);
}
```

Método	Descripción
`Distinct()`	**Elimina duplicados** en una colección.
`DistinctBy(selector)`	**Elimina duplicados basándose en una propiedad**.
`Except(otraLista)`	**Devuelve los elementos que están en la primera lista pero no en la segunda**.
`ExceptBy(otraLista, selector)`	**Elimina elementos según una propiedad específica**.
`Intersect(otraLista)`	**Devuelve los elementos comunes en ambas listas**.
`IntersectBy(otraLista, selector)`	**Intersección basada en una propiedad**.
`Union(otraLista)`	**Une dos listas sin duplicados**.
`UnionBy(otraLista, selector)`	**Une dos listas eliminando duplicados según una propiedad**.