

LA BIBLIA DE DESARROLLO DE SOFTWARE

UNIDAD 1: REPASO.

CONTENIDO:

- Repaso: Concepto de POO. Depuración de Código.
- Control de Versiones: GitHub. Comandos Básicos. Manejo de Ramas. Resolución de Conflictos

MATERIAL DE ESTUDIO:

OBLIGATORIO:

- Pro Git, 2da edición 2021. **Cap 1-6**
- Git-scm.com

ADICIONAL:

- Pro Git, 2da edición 2021. **Cap 7+**
- Análisis y diseño orientado a objetos con aplicaciones. 2da Edición, 1996. **Cap 2-3**

REPASO DE ELEMENTOS FUNDAMENTALES DE PROGRAMACIÓN ORIENTADA A OBJETOS

Abstracción, Encapsulamiento, Modularidad y Jerarquía

GIT Y VERSIONADO

- **Control de Versiones:** Es un sistema que registra los cambios realizados en un archivo o conjunto de archivos a lo largo del tiempo, de modo que se pueda recuperar versiones específicas más adelante.
Sistemas de Control de Versiones Locales:
 - Usado por muchas personas, Propenso a errores.
- **Sistemas de Control de Versiones Centralizados (CVCS por sus siglas en inglés):**
 - Se desarrollaron para solucionar el problema de colaborar con desarrolladores en otros sistemas.
 - Tienen un único servidor central y varios clientes que descargan los archivos.
 - Fue estándar por muchos años.
 - Tiene como ventaja:
 - Conocimiento compartido del trabajo de los demás
 - Control detallado de lo que cada usuario puede hacer
 - Facilidad de Administración
 - Tiene como desventajas:
 - Muy propenso a contratiempos si este servidor se cae
 - Si no existen copias de seguridad y el disco duro se corrompe, se perderá toda la información.
- **Sistema de Control de Versiones Distribuidos (DVCS por sus siglas en inglés):** Ofrecen soluciones para los problemas que han sido mencionados.
 - Algunos ejemplos de DVCS son Git, mercurial, bazaar o darcs.
 - Los clientes no solo descargan la última copia instantánea de los archivos, sino que se refleja completamente en el repositorio
 - Cada clon es realmente una copia completa de todos los datos

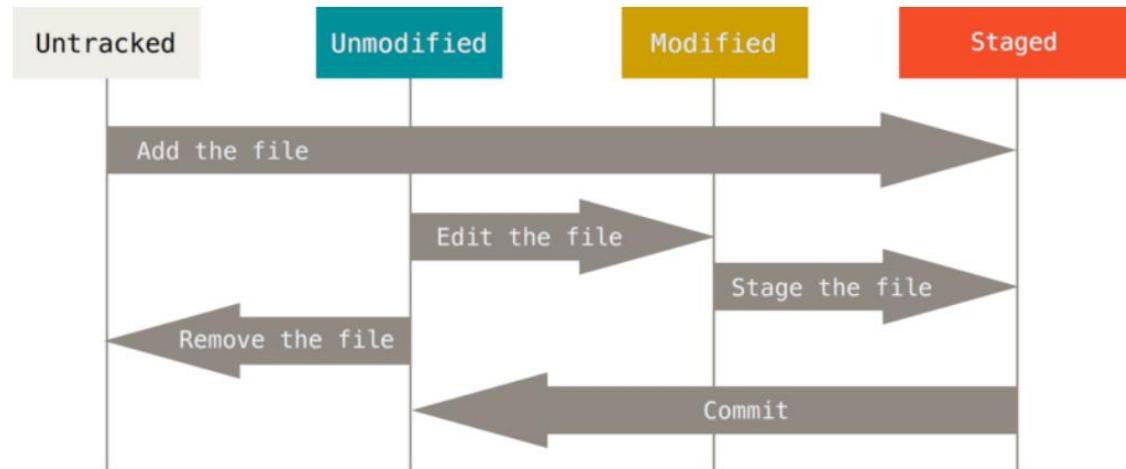
- Se encargan de manejar numerosos repositorios remotos con los cuales pueden trabajar simultáneamente con diferentes grupos de personas dentro del mismo proyecto.
- Nos permite crear flujos de trabajo

Entonces...

FUNDAMENTOS DE GIT

- **Git maneja sus datos como un conjunto de copias instantáneas de un sistema de archivos miniatura.** Básicamente toma una foto del aspecto de todos tus archivos en ese momento y guarda una referencia a esa copia instantánea
- Para ser eficiente, si los archivos no se modificaron, no almacena el archivo de nuevo, sino un enlace al archivo anterior idéntico que ya tiene almacenado.
- Casi todas sus operaciones son locales, lo cual también significa que hay muy poco que no puedas hacer si estás desconectado.
- Git usa un check-sum (suma de comprobación) para identificar los contenidos. Lo cual se almacena en el más bajo nivel y esto permite que la información se pierda en la transmisión o se corrompan archivos sin detectarlo.
- El mecanismo para generar el check-sum se conoce como “HASH SHA-1”: Una cadena de 40 caracteres hexadecimales y se calcula con base en contenidos del archivo o la estructura del directorio en Git
- Tres estados:
 - **Committed (Confirmado):** Los datos están almacenados de manera segura en tu base de datos local
 - **Modified (Modificado):** Has modificado el archivo, pero todavía no lo has confirmado a tu base de datos
 - **Staged (Preparado):** Has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación
- Tres secciones principales de un proyecto de Git:
 - **Directorio de Git (Git directory):** Es donde se almacenan los metadatos y la base de datos de objetos para tu proyecto. Es lo que se copia cuando clonas un repositorio desde otra computadora
 - **Directorio de trabajo(Working directory):** Es una copia de una versión del proyecto. Estos se sacan de la base de datos comprimida en el directorio de Git, y se colocan en disco para que los puedas usar o modificar.
 - **Área de preparación (Staging área):** Es un archivo, generalmente contenido en tu directorio de git. A veces se le denomina index.
- **El Flujo de Trabajo en Git** es algo así:
 - Modificas una serie de archivos en tu directorio de trabajo
 - Preparas los archivos, añadiéndolos a tu área de preparación
 - Confirmas los cambios, git toma los archivos como están en preparación y almacena esa copia en tu directorio.
- Si una versión concreta de un archivo está en el directorio de Git, se considera committed(confirmada). Si ha sufrido cambios, pero ha sido añadida, está staged(preparada). Y si ha sufrido cambios desde que se obtuvo el repositorio, pero no se ha preparado, está modificada (modified)
- Hay que tener en cuenta que los archivos tracked o rastreados son los que estaban en la última instantánea del proyecto, pueden ser archivos sin modificar, modificados o preparados. Los archivos sin rastrear, son los que no estaban en la última instantánea y no

están en staging área.



Configurando git por primera vez

git config: Sirve para obtener y establecer variables de configuración.

- Tiene 3 niveles:
 - **Nivel 1:** /etc/gitconfig: Contiene valores para todos los usuarios y sus repositorios. –system
 - **Nivel 2:** /gitconfig o /.config/git/config . Este archivo es de tu usuario. –global
 - **Nivel 3:** El config del repositorio que estés usando. .git/config
- Identidad: Primero se establece el nombre de usuario y la dirección de correo electrónico. Sólo para –global.
 - \$ git config –global user.name “Jaku”
 - \$ git config –global user.email jacasdavid@gmail.com
- Se pueden ver tus datos con: \$ git config—list
- Para ver una clave específica: \$ git config user.name

git help [comandogit]: Sirve para ver la página de ayuda de cualquier comando, por ejemplo, config: \$ git help config

git init : Esto crea un nuevo subdirectorio llamado .git, éste contiene todos los archivos necesarios del repositorio.

git add [archivo]: Esto añade que archivos se quieren controlar

git commit -m ‘initial Project version’ : Esto confirma los cambios

git clone [url] [nombredirectorio]: Este comando sirve para obtener una copia de un repositorio git existente.

git status: Nos dice que archivos están en qué estado.

UNIDAD 2

CONTENIDO:

- Introducción a la plataforma. Entorno de desarrollo integrado (IDE).
- Variables, operaciones y expresiones. Manejo de errores y excepciones.
- Clases abstractas, anónimas y genéricas; herencia e interfaces. Colecciones, expresiones lambda y lenguaje de consulta.
- Delegados y manejo de eventos.

- Programación asíncrona con *async* y *await*.

MATERIAL DE ESTUDIO:

OBLIGATORIO

- Documentación y videos en línea (según guía)
- Microsoft Visual C# Step by Step, Tenth Edition, John Sharp, 2022

.NET

Es una Plataforma de Desarrollo gratis, de código abierto y multi-plataforma (Web, Mobile, Cloud, AI & Desktop)

- Entorno de ejecución: Ejecuta el código de la aplicación (CLR)
- Bibliotecas: Proporciona funcionalidades como JSONs
- Compilador: Copila código fuente en C# en código ejecutable (entorno de ejecución)
- SDK y otras... : Permiten la creación y supervisión de flujos de trabajo modernos
- Pilas de aplicaciones: ASP.NET Core y Windows Forms

IMPLEMENTACIONES DE .NET

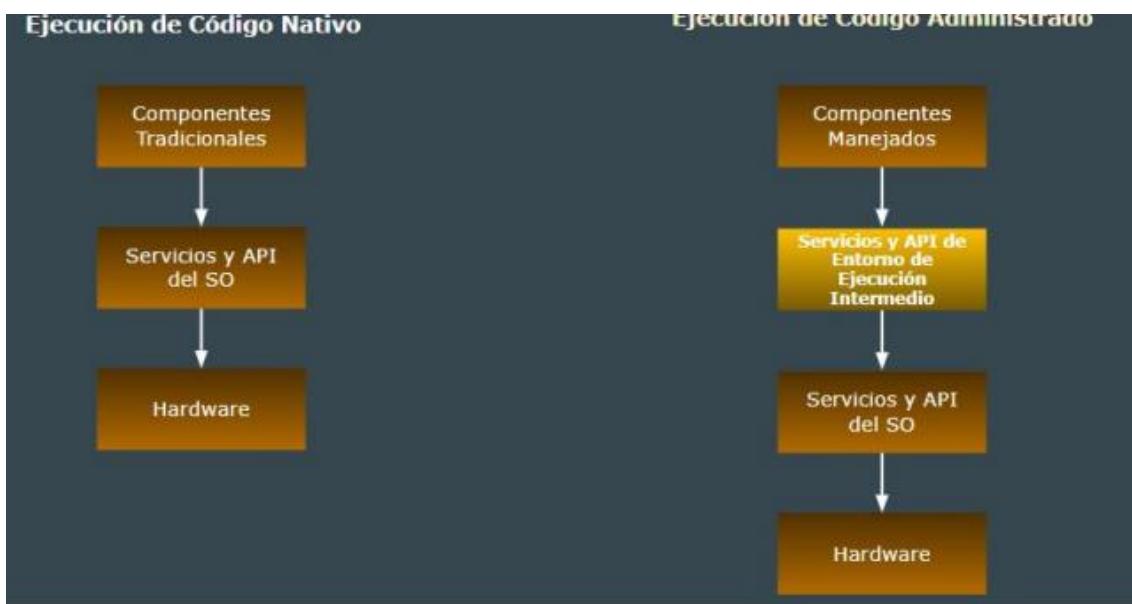
- .NET >5
- .NET Framework
- Mono
- UWP

.NET Standard: Es una especificación formal de las API de .NET. Tiene como finalidad establecer uniformidad del ecosistema.

Las versiones son a corto plazo (STS) o a largo plazo (LTS)

.NET	.NET Framework
Linux, macOS, Windows	Windows
Es de código abierto y acepta contribuciones	De código abierto, pero no acepta contribuciones
Innovaciones, más tipo de aplicaciones y mejor rendimiento	Seguridad y Fiabilidad
No tiene relación con el S.O	Se incluye en las actualizaciones de Windows
Recomendado para nuevos desarrollos	

- Namespaces: agrupación lógica de clases
- Assemblies: Archivo físico, compilado que contiene el código (con sus espacios de nombres)



GLOSARIO

CONCEPTOS FUNDAMENTALES

- **Ensamblado:** Un archivo .dll o .exe que encapsula IL, metadatos y recursos; es la unidad de despliegue y versión en .NET.
- **IL (Intermediate Language):** Lenguaje de bajo nivel, independiente del hardware, al que compilan los lenguajes .NET (como C#).
- **CLR (Common Language Runtime):** La máquina virtual fundamental para ejecutar aplicaciones .NET. Gestiona memoria (GC), compila IL a código de máquina (JIT), maneja excepciones, seguridad e interoperabilidad.
- **BCL (Biblioteca de Clases Base):** Colección central de bibliotecas (System.* , Microsoft.*) que ofrece funcionalidades esenciales y de propósito general en .NET. Es la base para frameworks de aplicación de alto nivel.
- **GC (Recolector de elementos no utilizados):** La implementación automática de gestión de memoria del CLR que libera recursos de objetos ya no utilizados.

COMPILADORES

- **JIT (Just-In-Time Compiler):** Compilador del CLR que convierte IL a código de máquina nativo “sobre la marcha” durante la ejecución de la aplicación.
- **AOT (Ahead-Of-Time Compiler):** Compilador que traduce IL a código de máquina antes de la ejecución de la aplicación. Permite optimizaciones profundas, vinculación entre módulos y análisis completo del programa, generando un único ejecutable.

IMPLEMENTACIONES Y HERRAMIENTAS DE .NET

- **.NET:** Principalmente se refiere a la implementación moderna, multiplataforma y de código abierto de .NET (antes conocida como .NET Core 5+). También puede ser un término general que abarca todas las implementaciones.
- **.NET Framework:** La implementación original de .NET, exclusiva de Windows, que incluye su propio CLR, BCL y frameworks.

- CoreCLR: La implementación multiplataforma del CLR utilizada por .NET (la versión moderna).
- CoreRT: Un runtime de .NET en desarrollo que no incluye JIT, enfocado en la compilación AOT para crear ejecutables más compactos.
- .NET Native: Cadena de herramientas que usa AOT para compilar aplicaciones UWP directamente a código nativo, permitiendo ejecutables autocontenido.
- Mono: Una implementación multiplataforma y de código abierto de .NET, optimizada para entornos pequeños y compatible con JIT y AOT.
- SDK de .NET: Conjunto de bibliotecas y herramientas para desarrollar aplicaciones .NET, incluyendo la CLI de .NET y el runtime.
- CLI de .NET: La herramienta de línea de comandos multiplataforma para construir y gestionar proyectos .NET.
- NGen (Generación Nativa de Imágenes): Una tecnología que compila código en tiempo de instalación, actuando como un JIT persistente.

MODELOS DE APLICACIÓN Y FRAMEWORKS

- Modelo de aplicación: APIs o frameworks específicos para un tipo de carga de trabajo (ej. desarrollo web, escritorio).
- Carga de trabajo: Tipo de aplicación que se construye (ej. web, móvil, escritorio, nube).
- ASP.NET: El framework web original de .NET Framework (ASP.NET 4.x).
- ASP.NET Core: La versión multiplataforma, de alto rendimiento y de código abierto de ASP.NET.
- WPF (Windows Presentation Foundation): Framework para GUIs de escritorio ricas en Windows.
- WCF (Windows Communication Foundation): Framework para servicios distribuidos en Windows.
- WF (Windows Workflow Foundation): Framework para flujos de trabajo en aplicaciones Windows.
- WinForms (Windows Forms): Framework para GUIs de escritorio nativas en Windows.
- Entity Framework (EF): Framework para interacción con bases de datos.
- POCO (Plain Old CLR Object): Objeto .NET simple, usualmente con solo propiedades públicas, empleado como DTO.

OTROS TÉRMINOS CLAVE

- Multiplataforma: Capacidad de desarrollar y ejecutar una aplicación en múltiples sistemas operativos.
- Plataforma: Un sistema operativo y el hardware en el que se ejecuta (ej., Windows, Linux).
- Plataforma de destino: Conjunto de APIs de las que depende una aplicación o biblioteca .NET, especificada por un TFM.
- TFM (Moniker de la Plataforma de Destino): Formato estandarizado (ej., net8.0) para indicar la plataforma de destino.
- Pila: Un conjunto de tecnologías de programación para construir y ejecutar aplicaciones (ej., la “pila de .NET”).
- Ecosistema: El software de runtime, herramientas y recursos de la comunidad de una tecnología.
- Biblioteca: Colección de APIs callable, compuesta por uno o más ensamblados.

- Marco compartido: Colección de bibliotecas (incluyendo la BCL) que se incluyen en el entorno de ejecución de .NET y otros frameworks como ASP.NET Core.
- Paquete: Archivo .nupkg (NuGet) que contiene ensamblados y metadatos para la distribución de bibliotecas.

VISUAL STUDIO

VARIABLES, OPERADORES Y EXPRESIONES

DECLARACIÓN O STATTEMETN

Es una acción, tal como calcular un valor y almacenar el valor o mostrar un mensaje al usuario. Se combinan declaraciones para crear métodos.

```
Console.WriteLine("Hola Mundo");
```

Esta es la sintaxis definida para un hola mundo en C#.

IDENTIFICADORES O IDENTIFIERS

Son nombres que identifican los elementos del programa, como namespaces, clases, métodos, y variables. Sensibles a mayúsculas y minúsculas

Se pueden usar solo letras en mayúscula, minúscula, dígitos y guion bajo. Debe empezar con una letra o guion bajo.

```
//Forma correcta
string resultado, equipoDeFutbol, EquipoDeFutbol, plan9, _puntos;
//Forma incorrecta
string 9plan, resultado%,equipodeFutbol$;
```

VARIABLES

Es una unidad de almacenamiento que contiene un valor.

CONVENCIONES

- No empezar variables con “_”
 - No crear variables que solo se diferencien con una mayúscula o minúscula. Por ejemplo, arbolVerde y ArbolVerde .
 - Empezar con minúscula
 - Usar camelCase. Del tipo, arbolVerde, clubDeFutbol, colorAuto, volumenParlante.
-

DECLARACIONES DE VARIABLES

Para declarar una variable, tenés que especificarle el tipo de dato que va a almacenar y al final el punto y coma.

TIPOS DE DATOS PRIMITIVOS

Tipo de Dato	Descripción	Bits	Rango	Ejemplo
int	Enteros	32	-2^{31} hasta $2^{31} - 1$	int contador; contador = 42;
long	Enteros grandes	64	-2^{63} hasta $2^{63} - 1$	long esperar; esperar = 42L;
float	Números de punto flotante	32	-3.4×10^{38} hasta 3.4×10^{38}	float resultado; resultado = 0.42F;
double	Doble precisión de punto flotante	64	$\pm 5.0 \times 10^{324}$ hasta $\pm 1.7 \times 10^{308}$	double trueque; trueque = 0.42;
decimal	Valores monetarios	128	28 cifras significativas	decimal moneda; moneda = 0.42M;
string	Secuencia de caracteres	16 c/c	No aplica	string nombre; nombre = "David";
char	Un solo carácter	16	Un solo carácter	char opción; opcion='x'
bool	Lógico o booleano	8	True o False	bool opcion; opcion = true;

VARIABLES INASIGNADAS

C# no te permite usar una variable no asignada, por lo que si no le asignas antes de usarla, no va a compilar tu programa. La regla se llama *definite assignment rule* o **regla de asignación definida**.

Ejemplo:



OPERADORES ARITMETICOS

Los símbolos (| + | - | * | / | %) son operadores que se pueden usar dependiendo del tipo de dato. Por ejemplo, podemos utilizarlo para sumar enteros, o reales. Pero no podemos usarlos para concatenar dos chars o dos strings.

Deberías tener en cuenta que el resultado de una operación aritmética depende del tipo de operandos usados. Por ejemplo, el resultado de sumar dos doubles, es un double. En el caso de dividir, por ejemplo 5/2, el resultado nos dará también un int, por lo que C# redondeará en circunstancias como estas.

La situación se complica cuando mezclamos tipos de operandos, por ejemplo, si dividimos un int y un double, el compilador detecta este problema y convierte el entero en double. Pero, recordar que es una mala práctica.

Si intentamos dividir, por ejemplo, 0/0 el resultado será NaN (Not a Number)

No escribas esto:	Escribe esto:
variable = variable * número;	variable *= número;
variable = variable / número;	variable /= número;
variable = variable % número;	variable %= número;
variable = variable + número;	variable += número;
variable = variable - número;	variable -= número;

INTERPOLACIÓN DE STRINGS

C# Considera el uso del + como obsoleto para concatenar strings, por lo que la interpolación sigue la siguiente sintaxis:

```
string username = "José";
string mensaje = ("Hola {username} , bienvenido al zoológico.");
```

INCREMENTACIONES Y DECREMENTACIONES

Se presentan las siguientes maneras:

```
int contador = 0;  
contador = contador + 1;  
contador++;  
++contador;  
contador--;  
--contador;  
contador += 2;  
contador -= 2;  
...
```

En principio, entre poner los operadores “++” o “—“, no hay diferencia. Pero, hay que tener en cuenta, que estos son operadores, por lo que se ejecutarán en orden. Por ejemplo:

```
int contador = 42;  
Console.WriteLine(contador++); // contador es 43 pero se imprime 42  
contador = 42;  
Console.WriteLine(++contador); // contador es 43 y se imprime 43
```

DECLARACIONES IMPLICITAS

El compilador de C# determina rápidamente el tipo de expresión usado para inicializar una variable y nos indica si la variable es compatible o no. Nos permite hacer cosas como las siguientes:

```
int edad = 20;  
int edadNueva = edad + 5;  
  
var color = "Rojo";  
var talle = "42";  
  
var animal; // ERROR
```

Como podemos observar, “var” es un tipo de variable que puede tomar variables de cualquier tipo, pero debemos dejar que el compilador infiera de alguna manera.

MÉTODOS Y SCOPE

Un método es una secuencia de declaraciones, similar a una función.

Un método tiene un nombre y un body o cuerpo. El nombre debería ser un identificador que represente el propósito del método.

```
int Operacion (int a, int b)
{
    int c = a + b;
    return c;
}
```

Algunos métodos son tan sencillos, como la suma de dos variables, que podemos escribirlos con expresiones con cuerpo de expresión:

```
int Suma(int a, int b) => a + b;
```

En realidad, no hay una diferencia funcional, pero se pueden clarificar y son una conveniencia sintáctica.

La sintaxis para llamar a un método es esta:

```
int resultado = Suma(10, b);
```

Los métodos también pueden retornar tuplas.

```
static void Main(string[] args)
{
    int valor;
    float valor2;
    (valor, valor2) = Devolver(5, 6);
}

static (int, float) Devolver(int a, int b)
{
    return (a+b, 3.2f);
}
```

SCOPES

El Scope o Alcance nos define la región donde podemos utilizar la variable.

Cuando se crean variables en distintos puntos de una aplicación tenemos que tener en cuenta que una variable existe desde el punto en el que se define y las sentencias posteriores pueden usarla.

Cuando el método ha terminado, la variable desaparece y no puede usarse en otro lugar.

Cuando se puede acceder a una variable en una ubicación particular de un programa, se dice que la variable está “*in scope*” en esa ubicación.

Si dos identificadores tienen el mismo nombre y se declaran en el mismo scope, se dice que están **SOBRECARGADOS** o **OVERLOADED**.

REFACTORIZACIÓN Y MÉTODOS ANIDADOS

A partir de VS22 el código se puede refactorizar haciéndolo más leíble y entendible. Se ve algo así:

```
double dailyRate = readDouble("Enter your daily rate: ");
int noOfDays = readInt("Enter the number of days: ");
double dailyRate;
int noOfDays;
NewMethod(out dailyRate, out noOfDays);
writeFee(calculateFee(dailyRate, noOfDays));
}

private void NewMethod(out double dailyRate, out int noOfDays)
{
    dailyRate = readDouble("Enter your daily rate: ");
    noOfDays = readInt("Enter the number of days: ");
}
```

A veces, nos conviene achicar métodos largo en piezas más pequeñas e implementarlas en el método.

PARÁMETROS Y ARGUMENTOS NOMBRADOS

Aprovechando el manejo de la sobrecarga de c# podemos definir métodos que acepten distintos argumentos y cantidad de parámetros.

```
void Main(string[] args)
{
    int Resultado;
    Resultado = Suma(10, 2, 5); //17
    Resultado = Suma(0, 2); //2
    Resultado = Suma(5); //5
}

int Suma(int num1, int num2, int num3) => num1 + num2 + num3;
int Suma(int num1, int num2) => num1 + num2;
int Suma(int num1) => num1;
```

Para definir un parámetro opcional, lo definimos proporcionando un valor por defecto. Primero especificamos todos los parámetros obligatorios antes de cualquier otro opcional.

```
int Suma(int num1, int num2 = 0, int num3 = 0) => num1 + num2 + num3;
```

Por defecto, C# utiliza la posición de cada argumento en una llamada al método para determinar a qué parámetro se aplica el argumento. Pero, también podemos especificar los parámetros por nombre. Esto nos permite pasar los argumentos en una secuencia diferente.

```

void Main(string[] args)
{
    float Resultado = Restar(b:10, a:20);
}

float Restar(float a, float b) => a - b;

```

SENTENCIAS DE DECISIÓN

ESTA TABLA RESUME LOS OPERADORES Y SU ASOCIATIVIDAD

Categoría	Descripción		Asociatividad
Primarios	() ++ --	Anulación de procedencia Post-Incremento Post-Decremento	IZQ
Unarios	! + - ++ --	NOT Devuelve el operando sin cambios Devuelve el operando negado Pre-Incremento Post-Decremento	IZQ
Multiplicativos	*	Multiplicación	IZQ
	/	División	
	%	Resto (MOD o Módulo)	
Aditivos	+	Adición	
	-	Sustracción	
Relacionales	< <=	Menor que Menor igual que	IZQ
	> >=	Mayor que Mayor igual que	
Igualitarios	== !=	Igual a Distinto de	IZQ
Condicional AND	&&	AND	IZQ
Condicional OR		OR	IZQ
Asignación	=	Asigna el operando derecho al izquierdo	DER

PATTERN MATCHING O COINCIDENCIA DE PATRONES

Utiliza el operador “is”. La forma más sencilla de verlo es en un ejemplo:

```
bool porcentajeValido;

//Versión Antigua (Sin Pattern Matching)
int porcentaje = 40;
porcentajeValido = (porcentaje >= 0) && (porcentaje <= 100);

//Nueva Versión con Pattern Matching
porcentajeValido = porcentaje is >= 0 and <= 100;
```

SENTENCIAS DE DECISIÓN

Se usan cuando queremos ejecutar dos distintas sentencias dependiendo de una expresión booleana.

EN C# PODEMOS PRESCINDIR DE LOS CORCHETES PARA SENTENCIAS DE UNA LÍNEA

```
bool valor = true;
if (valor)
    Restar(1, 2);
else
    Restar(b: 1, a: 2);
```

Tenemos el recurso de la sentencia else if también.

Tenemos el recurso de la sentencia switch case

También podemos usar el pattern matching:

```

int valor = 30;
string range = valor switch
{
    < 0 => "Negativo",
    0 => "Cero",
    > 0 and <= 9 => "Unidad",
    > 9 and <= 99 => "Decena",
    > 99 and <= 999 => "Centena",
    > 999 => "Grande"
};

```

ASIGNACIÓN COMPUSTA E INSTRUCCIONES DE ITERACIÓN

Para	Hacé esto:
Incrementar el valor de una variable	variable += cantidad
Sustraer el valor de una variable	variable -= cantidad
while	<pre> while (a > b) { Console.WriteLine("a es mayor que b"); a++; } </pre>
do while	<pre> do{ Console.WriteLine("a es mayor que b"); a++; }while (a > b); </pre>

for	<code>for (int i = 0; i < 10; i++) { Console.WriteLine("Valor de i: " } }</code>
foreach	<code>foreach (var item in args) { Console.WriteLine("Argumento: " + } }</code>

MANEJO DE ERRORES Y EXCEPCIONES

Los errores son inherentes al ciclo de vida de un programa. En C# se adopta un modelo basado en excepciones como mecanismo principal para la señalización y el manejo de errores.

Dado que es sumamente difícil garantizar que un segmento de código funcione siempre sin fallos, cualquier aplicación debe ser capaz de detectar y manejar satisfactoriamente las fallas, ya sea mediante acciones correctivas o informando al usuario las razones del error de la forma más clara posible.

C# Simplifica la separación de la lógica de negocio del código de manejo de errores a través de manejadores de excepciones y excepciones.

TRY-CATCH

Cuando escribimos el código dentro de la estructura try, la ejecución continúa normalmente si no hay errores. Sin embargo, si ocurre una condición e error, la ejecución se transfiere a un manejador o *handler* llamado **catch**.

El *Catch* se incluye luego del *try* y permite la gestión del error. Cuando el bloque *try* provoca un error, el *runtime* lanza una excepción. Entonces, este busca el primer *manejador catch* que coincida con el tipo de excepción lanzada y transfiere el control a este.

```
try//C# intenta ejecutar este bloque  
{  
| string valorEscrito = "10";  
| int valorDigito = int.Parse(valorEscrito);  
}  
catch (Exception ex)//Si hay algún error, se ejecuta este otro.  
{  
| Console.WriteLine($"El error es: {ex.Message}");  
}
```

EXCEPCIONES NO MANEJADAS O UNHANDLED EXCEPTIONS

Si el *try* lanza una excepción y no hay un *catch handler* correspondiente, intenta retornar al método que lo llamó. Si el *runtime* aun así no logra encontrar un *handler* adecuado, el programa terminará con una excepción no manejada.

USO DE MÚLTIPLES MANEJADORES CATCH

Como los errores generan distintos tipos de excepciones, C# permite definir múltiples *catch handlers* para un único bloque *try*. Cada *handler* está diseñado para capturar un tipo específico de excepción

```
try//C# intenta ejecutar este bloque
{
    string valorEscrito = "10";
    int valorDigito = int.Parse(valorEscrito);
}
catch(FormatException e)
{
    Console.WriteLine($"Error de Formato: {e.Message}");
}
catch(OverflowException e)
{
    Console.WriteLine($"Error de Desbordamiento_: {e.Message}");
}
```

Si una excepción se genera dentro de un bloque *catch*, esta excepción nueva no será capturada por los bloques *catch* adyacentes al mismo *try*. En su lugar se propagará al método que invocó el código siguiendo el mecanismo de excepciones no manejadas.

Cabe destacar que no hace falta, no es práctico ni necesario escribir un *catch handler* para cada excepción posible. La solución es simplemente una jerarquía de herencia de excepciones. Todas las excepciones en .NET derivan de una clase base fundamental llamada *Exception*. Esta es la raíz de todas las excepciones. Por ejemplo, *FormatException* y *OverflowException* son parte de la familia *SystemException*, que a su vez es miembro de la familia *Exception*

FILTRADO DE EXCEPCIONES

C# permite filtrar las excepciones que son capturadas por el *catch* mediante la adición de una condición bool. Esto se logra usando la palabra clave **when** seguida de una expresión.

```
bool mostrarErrores = false;
try
{
    string valorEscrito = "10";
    int valorDigito = int.Parse(valorEscrito);
}
catch(Exception e) when (mostrarErrores == true)
{
    Console.WriteLine($"Error de Formato: {e.Message}");
}
```

CHECKED Y UNCHECKED

Las palabras clave *checked* y *unchecked* también se pueden aplicar directamente a **expresiones individuales** encerradas entre paréntesis.

Los operadores compuestos ($+=$, $-=$) y los operadores de incremento/decremento ($++$, $--$) son operadores aritméticos y su comportamiento de desbordamiento puede controlarse con checked y unchecked.

Checked y unchecked **solo aplican a la aritmética entera**. No tienen efecto en operaciones de punto flotante (float, double), las cuales manejan los desbordamientos (y la división por cero) de manera diferente (ej. representaciones especiales para infinito).

Al aplicar checked a operaciones que pueden desbordarse, es fundamental agregar un manejador catch específico para **OverflowException** en el método llamador (como calculateClick en el ejemplo de la calculadora). Esto asegura que, en lugar de que la aplicación falle por una excepción no manejada, el error de desbordamiento sea capturado y se muestre un mensaje informativo al usuario (ej., “Arithmetic operation resulted in an overflow”).

LANZAR EXCEPCIONES O THROW EXCEPTIONS

La práctica recomendada si un error impide completar la tarea de un método es lanzar una excepción. En .NET, las librerías nos proveen muchas clases de excepción para diversos casos.

Las excepciones se lanzan con la palabra clave throw seguida de una instancia de la clase de excepción

```
public string nombreDelMes(int mes) => mes switch
{
    1 => "Enero",
    //...
    12 => "Diciembre"
    _ => throw new ArgumentOutOfRangeException("Mes inválido")
};
```

Esta función nos devuelve el nombre del mes que pasamos por argumento como un número. Con un switch cambiamos entre las distintas opciones. En caso de que ninguna de las opciones estén definidas, por ejemplo, si ponemos el número 13, el programa nos tira un error del tipo *ArgumentOutOfRangeException* o *Excepcion del tipo argumento fuera del rango*, con el mensaje “Mes inválido”.

Throw se puede usar en cualquier lugar donde se espere una expresión. Incluso con el operador condicional ternario.

FINALLY

El bloque finally garantiza que un conjunto de instrucciones siempre se ejecute, independientemente de si una excepción fue lanzada o no. Se incluye inmediatamente después de un bloque try o luego del último catch.

Es especialmente útil cuando se manejan recursos, como una conexión a base de datos, para cerrar la conexión.

Si una excepción ocurre y es capturada localmente, el manejador catch se ejecuta primero, seguido por el bloque finally. Si la excepción no es capturada localmente (se propaga hacia arriba en la pila de llamadas), el bloque finally se ejecuta antes de que la excepción continúe su propagación. La clave es que el bloque finally **siempre se ejecuta** una vez que el programa ha entrado en el bloque try asociado. Esto es crucial para la **gestión de recursos**, asegurando que los recursos adquiridos (como un manejador de archivo) sean liberados adecuadamente, previniendo fugas de recursos.

CLASES Y OBJETOS (CLASS & OBJECT)

Las clases son un mecanismo para modelar entidades por aplicaciones. Podemos pensar en las entidades como un objeto específico, como un cliente o algo aún más abstracto como una transacción. Parte del diseño es tomar las decisiones de que entidades son importantes para las operaciones del sistema.

Por su parte, la librería de .NET incluye cientos de clases listas para usar como *Console* o *Exception*.

En la programación orientada a objetos, el concepto de clasificación se traslada directamente al diseño de clases que agrupan atributos y comportamientos para modelar entidades del mundo real o conceptos abstractos de manera sistemática y comprensible.

ENCAPSULACIÓN

Es un principio fundamental y su objetivo es abstraer la complejidad interna de una clase permitiendo que esta interactúe con el sistema creando instancias y llamando a sus métodos sin necesidad de conocer los detalles de su implementación, respetando el principio de “*ocultamiento de información*”.

Básicamente sus dos objetivos son:

- Agrupar los métodos (comportamiento) y los datos (estado)
 - **Controlar la accesibilidad:** Definir que partes de la clase son accesibles desde el exterior y cuales permanecen internos, protegiendo la integridad y facilitando su mantenimiento y evolución
-

DEFINICIÓN DE CLASE

En C# usamos la palabra clave “*class*”. En el ejemplo creamos una clase llamada Círculo con un atributo llamado radio y un método que nos devuelve el área.

```
class Circulo
{
    int radio;
    double area() => Math.PI * radio * radio;
```

CREACIÓN E INICIALIZACIÓN DE OBJETOS

Para poder utilizar la clase, debemos instanciarla, comúnmente, en un objeto. Esto se hace mediante la palabra reservada “*new*”.

```
Circulo circuloA; //Declaración de un objeto de tipo Circulo
Circulo circuloB; //Declaración de otro objeto de tipo Circulo
circuloA = new Circulo(); //Instanciación del objeto
circuloB = new Circulo(); //Instanciación del otro objeto

circuloA = circuloB; //Asignación de referencia, ambos apuntan al mismo objeto
```

También podemos observar que es posible asignar una instancia de una clase a otra variable del mismo tipo.

He de destacar que en C# se puede inferir el tipo gracias a `new`, simplificandonos la sintaxis. Y podemos variar con la palabra reservada “`var`”.

```
Circulo circuloA = new Circulo(); //Forma vieja
Circulo circuloB = new(); //Forma nueva
var circuloC = new Circulo(); //Forma nueva con inferencia de tipo
```

ACCESIBILIDAD DE CLASES

Por defecto, los campos y métodos de una clase en C# son **private**. Esto significa que solo son accesibles dentro de la propia clase. Aunque se puede crear un objeto de una clase, no se puede acceder si sus campos son privados. Para controlar esta accesibilidad se utilizan las palabras clave **public** y **private**.

Convención:

- **Public:** Deben empezar con mayúsculas (PascalCase)
- **No Public:** Deben empezar con minúsculas (camelCase)
- **Excepción:** Los nombres de clase siempre comienzan con mayúsculas y los constructores deben coincidir exactamente con el nombre de su clase.

CONSTRUCTORES

Un constructor es un método especial que se ejecuta automáticamente cuando se crea una nueva instancia de la clase. Su propósito es realizar cualquier inicialización necesaria para el objeto.

- Tiene el mismo nombre que la clase
- Puede tomar parámetros
- No devuelve ningún valor, ni void
- Toda clase debe tener uno, si no está definido, el compilador lo generará automáticamente.

```
class Circulo
{
    private int radio;
    //Constructor por defecto
    public Circulo(int valor)
    {
        radio = valor;
    }
    double area() => Math.PI * radio * radio;
}
```

Al igual que otros métodos, los constructores pueden ser sobrecargados con diferentes listas de parámetros.

```

class Circulo
{
    private int radio;
    private string name;
    //Constructor por defecto
    public Circulo(int valor)
    {
        radio = valor;
    }
    public Circulo(int valor, string nombre)
    {
        radio = valor;
        name = nombre;
    }
    double area() => Math.PI * radio * radio;
}

```

CLASES PARCIALES (PARTIAL CLASS)

Para clases muy grandes, C# permite dividir el código fuente en una única clase en múltiples archivos, usando la palabra clave *partial*.

```

partial class Circulo
{
    public Circulo(double radio)
    {
        this.radio = radio;
    }
    public Circulo() : this(1) { } // Constructor por defecto con radio 1
}
partial class Circulo
{
    public double radio;
    public double Area => Math.PI * this.radio * this.radio;
}

```

DECONSTRUCTOR (DECONSTRUCT)

El deconstructor es el complemento de un constructor, este permite extraer los valores de los campos de un objeto existente.

- Se llama Deconstruct
- El retorno debe ser de tipo void
- out: Debe tomar uno o más parámetros.

MÉTODOS Y DATOS ESTÁTICOS (STATIC)

Un *método estático* o *static* pertenece a la clase en sí, no a una instancia individual del objeto. Se llama directamente usando el nombre de la clase, sin necesidad de instanciar.

UN MÉTODO ESTÁTICO SOLO PUEDE USAR OTROS MIEMBROS ESTÁTICOS

```
void Main(string[] args)
{
    Console.WriteLine($"Cantidad de Alumnos: {Escuela.CantidadAlumnos}");

class Escuela
{
    public static int CantidadAlumnos = 0;
    public Escuela()
    {
        CantidadAlumnos++; //Cada vez que se crea una escuela, se aumenta el contador de alumnos
    }
}
```

CONSTANTES (CONST)

La palabra clave es “*const*” y al declarar un campo de este tipo, se convierte en un campo estático cuyo valor no se puede cambiar. Solo se puede usar en tipos numéricos, cadenas o enumeraciones

```
class Escuela
{
    public const int codigo = 100;
```

CLASES STATIC (STATIC CLASS)

Sólo puede contener campos y métodos estáticos. Tiene como propósito ser un contenedor de utilidades. No se pueden crear instancias de una clase estática. Si necesita inicialización, puede tener un constructor estático.

using static: Permite importar miembros estáticos de una clase a un ámbito de modo que no sea necesario cualificar el nombre de la clase al llamarlos.

```
using static System.Math;
using static System.Console;

var root = Sqrt(99.9); // Se llama directamente Sqrt en lugar de Math.Sqrt
WriteLine($"La raíz cuadrada de 99.9 es {root}"); // Se llama directamente WriteLine en lugar de Console.WriteLine
```

CLASES ANÓNIMAS

Es un tipo de clase que no tiene un nombre explícito.

- Campos públicos
- Inicialización obligatoria
- Inferencia de tipo
- Nombre generado por el compilador
- Uso de var

```
void Main(string[] args)
{
    var objeto = new { Nombre = "David", Edad = 24 };
}
```

VALORES Y REFERENCIAS

Es clave entender estos conceptos, ya que impacta en la asignación de memoria, el paso de argumento a métodos y la copia de variables.

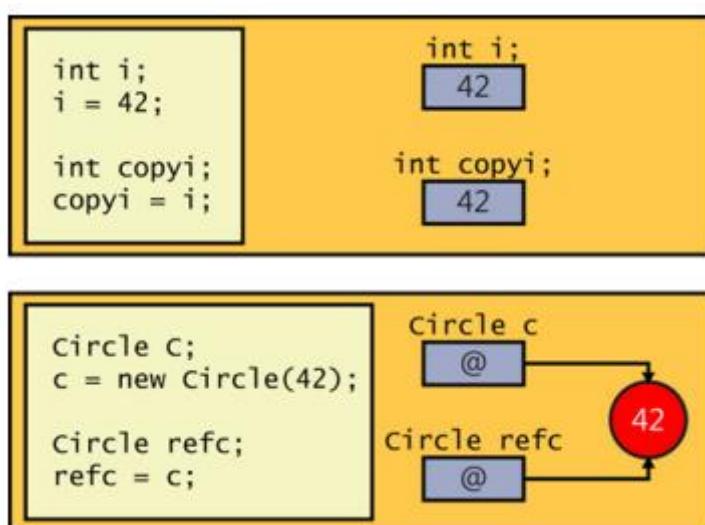
TIPOS POR VALOR VS TIPOS POR REFERENCIA

Los tipos por valor (*int, float, char, etc.*):

- Cuando declaramos una variable se asigna un bloque de memoria para el valor.
- Las asignaciones crean copias independientes de los datos.
 - Por ejemplo: `int i = 42; int copyi = i`
 - *i* y *copyi* tienen bloques de memoria distintos.

Los tipos por referencia (clases, string):

- Al declarar la variable solo se asigna memoria para una referencia al objeto.
- La memoria real se asigna solo con el operador `new`.
- Las asignaciones entre variables de referencia **copian la referencia**. Lo que significa que ambas variables apuntan al mismo objeto en memoria.



Si deseas copiar el contenido de objeto a otro, ósea un *Deep copy*, es necesario crear una nueva instancia y copiar los campos manualmente.

- *ref*: Pasa el argumento por referencia. La variable debe estar inicializada antes de la llamada y los cambios en el parámetro dentro del método afectan a la variable original.

```
static void Incrementar(ref int contador) // Declaración del parámetro ref
{
    contador++; // Modifica directamente la variable original
}

static void Main()
{
    int valor = 42;
    Incrementar(ref valor); // Llamada al método con ref
    Console.WriteLine(valor); // Muestra 43 (el valor de 'arg' fue modificado)
}
```

- *out*: Pasa el argumento por referencia. La variable no necesita estar inicializada antes de la llamada, pero el método debe asignarle un valor antes de retornar

```
static void Inicializar(out int valor)
{
    valor = 42;
}

static void Main()
{
    int contador;
    Inicializar(out contador);
    Console.WriteLine(contador); //42
}
```

VALORES NULL Y NULLEABLES

Cuando declaramos una variable en C# una buena práctica es inicializarla para evitar comportamientos indefinidos o errores en tiempo de ejecución. Cuando inicializamos tipos por valor, le damos un literal, por ejemplo, `int a = 0;`. Para los tipos por referencia, la inicialización más común es crear una nueva instancia utilizando `new` y asignársela a la variable.

Si una variable de referencia se inicializa con un objeto, pero luego se le asigna otra referencia a otro objeto, el objeto original se convierte un “*candidato a la recolección de basura*”. El CLR de C# ejecuta el garbage collection y reclama la memoria ocupada por estos objetos referenciados.

Libera memoria, pero consume tiempo, por lo que es ineficiente crear objetos que no serán utilizados.

Podemos utilizar “`null`” para inicializar variables de tipo por referencia cuando no se desea que apunten a ningún objeto en memoria. Este hace referencia a que no apunta a ningún objeto.

Podemos verificar que una variable de referencia es null utilizando “`is null`”, o “`== null`”

OPERADORES CONDICIONALES NULOS Y DE COALESCENCIA NULA

El operador condicional nulo “?.” permite invocar un método o propiedad de un objeto solo si el objeto no es null. Si el objeto es null, la expresión devuelve null en lugar de lanzar una excepción

```
void Main(string[] args)
{
    Circulo c = null;
    Console.WriteLine($"El área del círculo c es: {c?.Radio}");
}
```

OPERADORES DE COALESCENCIA NULA

Operador de Coalescencia Nula (??): Es un operador binario que devuelve el valor del operando de la izquierda si no es null, sino, el de la derecha.

```
void Main(string[] args)
{
    Circulo circulo1 = null;
    var circulo2 = circulo1 ?? new Circulo(10);
}
```

Operador de Asignación de Coalescencia Nula (??=): Asigna el valor del operando de la derecha al de la izquierda, solo si el de la izquierda es null. Si ya tiene un valor nulo, no se realiza ninguna asignación.

```
void Main(string[] args)
{
    Circulo circulo1 = new Circulo(5.0);
    Circulo circulo2 = null;

    circulo2 ??= circulo1; //Si circulo2 es null, se asigna circulo1 a circulo2
}
```

NULLEABLES

A su vez, los tipos por valor, no pueden contener null. pero c# introduce los tipos anulables para permitirlo.

```
int? distancia = null;
```

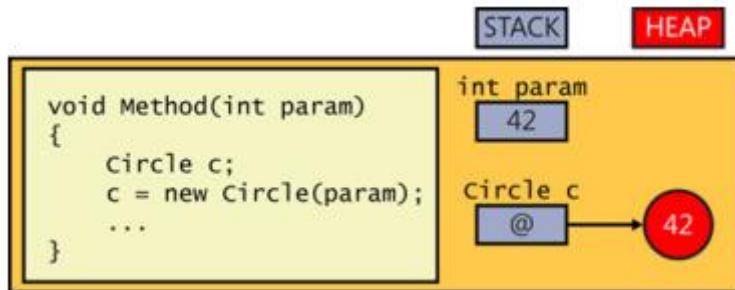
Estos poseen dos propiedades para poder gestionar su valor.

- **HasValue:** Es bool y nos indica si tiene un valor o no.
- **Value:** Si HasValue es false, lanzara una excepción.

Es fundamental no confundir los **tipos anulables** (int?) con el **operador condicional nulo** (?.). El primero modifica el *tipo de la variable*, permitiéndole ser null; el segundo es un *operador* que maneja invocaciones a miembros en referencias potencialmente null.

ORGANIZACIÓN DE MEMORIA

Para comprender la interacción entre tipos por valor y por referencia en C#, es fundamental entender cómo el sistema organiza la memoria del ordenador. C# (y los sistemas operativos modernos) divide la memoria principal en dos áreas principales: la **pila (Stack)** y el **montón (Heap)**, cada una con un propósito y método de gestión distintos.



LA PILA (STACK)

- **Propósito:** Se utiliza para almacenar los **parámetros de métodos** y las **variables locales** (tanto de tipo por valor como las referencias a objetos de tipo por referencia).
- **Gestión:** La memoria en la pila se organiza de forma estructurada, como una pila de cajas. Cuando se llama un método, se asigna espacio en la parte superior de la pila para sus parámetros y variables locales. Cuando el método termina (o un bloque de código entre llaves finaliza), la memoria asignada se libera automáticamente de forma inmediata.
- **Vida útil:** Las variables en la pila tienen una vida útil bien definida y predecible: existen solo mientras el método o bloque de código que las contiene se está ejecutando.
- **Tipos por Valor:** Por defecto, todas las instancias de **tipos por valor** se crean directamente en la pila.

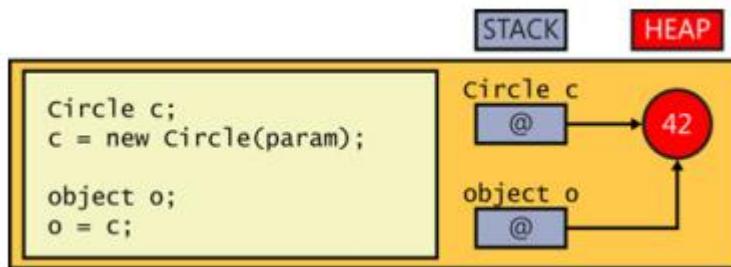
EL MONTÓN (HEAP)

- **Propósito:** Se utiliza para almacenar **objetos** (instancias de clases), que son **tipos por referencia**.
- **Gestión:** La memoria en el montón es menos estructurada. Cuando se crea un objeto usando el operador `new`, el *runtime* busca un espacio disponible en el montón y lo asigna al objeto. La **referencia** a este objeto (que es una variable de tipo por referencia) se almacena en la pila.
- **Vida útil:** La vida útil de los objetos en el montón es más indeterminada. Un objeto permanece en el montón mientras existan referencias que apunten a él. Una vez que la última referencia a un objeto desaparece, su memoria se vuelve candidata para ser recuperada por el **Recolector de Basura (Garbage Collector)**, un proceso automático que libera la memoria del montón de objetos ya no utilizados.
- **Tipos por Referencia:** Por defecto, todas las instancias de **tipos por referencia** (objetos) se crean en el montón.

LA CLASE SYSTEM.OBJECT

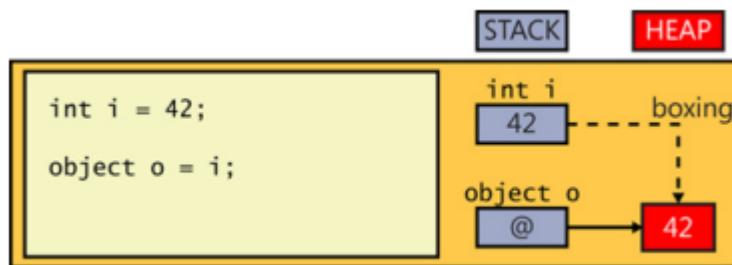
- **Base de Todo:** `System.Object` es la clase raíz de la jerarquía de tipos en .NET. Esto significa que **todas las demás clases** (y, por extensión, todos los tipos en C#) derivan de `System.Object`.
- **Alias object:** C# proporciona la palabra clave `object` como un alias directo para `System.Object`. Ambas se refieren a lo mismo.

- **Referencia Universal:** Una variable de tipo object puede referenciar a **cualquier tipo por referencia** (y, mediante boxing, también a tipos por valor).

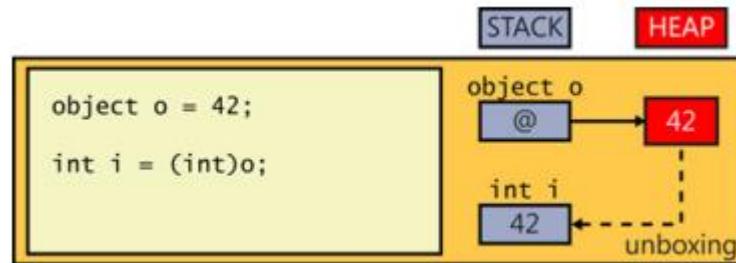


BOXING Y UNBOXING

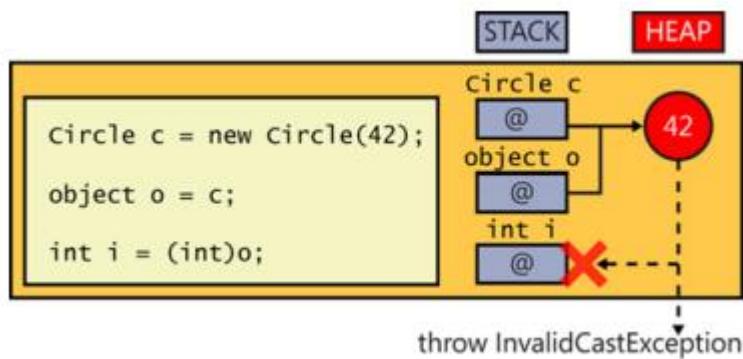
- **Boxing:** Es la conversión **implícita** de un **tipo por valor** a un **tipo por referencia** (object). Cuando ocurre, el valor de la pila se **copia** a una nueva ubicación en el montón, y la variable object apunta a esa copia. Esta es una operación costosa.



- **Unboxing:** Es la conversión **explícita** de un **tipo por referencia** (object) de vuelta a un **tipo por valor**. Requiere un cast explícito y una verificación en tiempo de ejecución.



- Si el tipo no coincide, se lanza una InvalidCastException.



CASTING SEGURO CON OPERADORES IS Y AS (RECAPITULACIÓN)

Para evitar InvalidCastException al intentar un cast de objetos, C# ofrece:

- **Operador is:** Permite verificar si un objeto es compatible con un tipo dado. Puede combinarse con una declaración de variable para un cast seguro y conciso.
- **Operador as:** Intenta un cast y devuelve null si la conversión falla, en lugar de lanzar una excepción.

SENTENCIA SWITCH CON PATRONES DE TIPOS

La sentencia switch en C# ha sido mejorada para permitir la comprobación de tipos de objetos y la extracción de valores, similar a una serie de if-else if con el operador is:

Esto mejora la legibilidad y la estructura del código al manejar múltiples tipos posibles de un objeto.

HERENCIA(INHERITANCE)

Es un concepto clave, podemos utilizarla para evitar repetir definiciones que tienen características en común.

Básicamente se trata de clasificar clases y definir sus relaciones. Es más sencillo entenderlo con un ejemplo:

Si creamos una clase *Caballo* y otra que se llame *Perro*. En principio, podemos pensar que no tienen demasiado en común, pero si nos abstraemos podemos identificar que ambos son animales mamíferos y pueden compartir características como por ejemplo, respirar o color de ojos.

Por lo que, podemos abstraernos y que *Caballo* y *Perro* hereden características propias de un animal mamífero.

```
class AnimalMamifero()
{
    string? colorOjos;
    void Respirar()
    {
        Console.WriteLine("El animal ha respirado.");
    }
}
class Caballo : AnimalMamifero
{
    string? colorMontura;
}
class Perro() : AnimalMamifero
{
    int intensidadLadrido;
}
```

Ambos tienen un color de ojos y pueden respirar (Por ser animalMamifero). Pero, individualmente un caballo tiene un color de montura y un perro por su lado, una intensidadLadrido.

OCULTAMIENTO DE MÉTODO

A veces, una clase padre y una hija tienen métodos con mismo nombre y parámetros. Esto puede causar un ocultamiento de método. El compilador nos advertirá. Si esto es intencional, se puede silenciar la advertencia utilizando la palabra clave new

```
class Caballo : AnimalMamifero
{
    string? colorMontura;
    new void Respirar()
    {
        Console.WriteLine("El caballo ha respirado.");
    }
}
```

Sin embargo, es diferente de la anulación de método. Esta busca proporcionar diferentes implementaciones de un mismo método. La anulación es un concepto útil y el ocultamiento, generalmente un error.

VIRTUAL

Un método para ser anulado se denomina método virtual. Se declara con la palabra reservada "virtual", y el "override" para sobreescribir.

```
class AnimalMamifero()
{
    string? colorOjos;
    public virtual void Respirar()
    {
        Console.WriteLine("El mamífero ha respirado.");
    }
}

class Caballo : AnimalMamifero
{
    string? colorMontura;
    public override void Respirar()
    {
        Console.WriteLine("El caballo ha respirado.");
    }
}
```

- Los métodos *virtual* y *override* no pueden ser *private*.
- Las firmas deben ser idénticas.
- Sólo se puede anular un método que haya sido declarado como *virtual*.
- Si una clase derivada no usa *override* para un método virtual, lo oculta en lugar de anularlo.

- Un método *override* es implícitamente *virtual* y puede ser anulado por clases más derivadas.
-

PROTECTED

Acceso protegido (*protected*)

Además de *public* y *private*, C# nos ofrece el modificador *protected*. Este permite que los miembros de una clase sean accesibles para todas las clases derivadas, pero sigan siendo privados para las clases que no forman parte de la jerarquía de herencia.

Si bien C# lo permite, las buenas prácticas de POO recomiendan mantener los campos estrictamente *private* siempre que sea posible para preservar la encapsulación.

MÉTODOS DE EXTENSIÓN (EXTENSION METHODS)

Ofrecen una alternativa a la herencia para añadir funcionalidades a tipos existentes sin modificar su código original.

Son especialmente útiles cuando la herencia no es viable o se desea extender un tipo sin alterar el código existente que lo utiliza.

Un método de extensión se define dentro de una clase estática y se identifica porque su primer parámetro va precedido por la palabra clave “*this*”, indicando el tipo que se está extendiendo.

IMPLEMENTANDO PROPIEDADES PARA EL ACCESO A CAMPOS (DATOS)

Este capítulo se enfatiza en acceder de manera segura y controlada los campos.

Retomamos la idea de que los campos de una clase deben ser privados y su acceso (W/R) debe hacerse a través de métodos. Esto permite validar los datos y aplicar lógica adicional.

Si los campos son *public*, se pierde control sobre los valores, por lo que puede llevar a datos inválidos. Una solución es implementar los métodos *Get()* y *Set()* para acceder a ellos. Pero, es un poco torpe o natural en comparación con el acceso directo a un campo público.

Aquí es donde entran las propiedades.

PROPIEDADES

Una propiedad es una combinación de un campo y un método, actúa como un método pero parece un campo.

Las propiedades son una característica poderosa, pero deben usarse con criterio. No sustituyen un buen diseño orientado a objetos que se centra en el **comportamiento** de los objetos, no solo en sus propiedades.

Las propiedades también pueden definirse en Interfaces. Cualquier clase o estructura que implemente esta interfaz debe proporcionar una implementación completa para las propiedades X e Y, ya sea con la sintaxis de bloque o de expresión-bodied members.

Si la implementación de la propiedad en la clase es virtual, las clases derivadas pueden anularla. Las estructuras no admiten virtual debido a que no soportan herencia de la misma manera que las clases.

También es posible usar la **implementación explícita de interfaz** para propiedades, lo que las hace no públicas y no virtuales (no se pueden anular).

- Cuando se accede a esta, el compilador lo traduce automáticamente a métodos accesores (getters y setters).
- Pueden ser estáticas
- Propiedades de solo lectura (read - only): Solo tienen get.
- Propiedades de sólo escritura (write - only): Solo tienen un set.
- Podemos especificar la accesibilidad general de una propiedad
- Podemos sobreescibir la accesibilidad
- Podemos hacer que un set sea public pero que su set sea private o viceversa. Pero, no ambos.
- C# permite usar **coincidencia de patrones** para evaluar los valores de las propiedades de un tipo, lo que simplifica la lógica condicional.

Convenciones:

- Campos privados : Con minúscula y guión bajo para distinguirlos.
- Propiedades públicas: Con mayúscola
- Dentro de get se devuelve el valor del campo privado
- Dentro del set se asigna el valor al campo.

Restricciones:

- Solo se puede asignar un valor después de que la instancia haya sido inicializada.
- No se puede usar ref u out. Ya que una propiedad no apunta directamente a una ubicación de memoria, sino a un método accesor.
- Una propiedad puede contener un get y un set. No puede contener otros métodos, campos o propiedades.
- No se puede usar const
- Get y Set no pueden tomar parámetros explícitos. Para set, se pasa a través de la variable value.

PROPIEDADES AUTOMÁTICAS (AUTOMATIC PROPERTIES)

Aunque el propósito principal de las propiedades es encapsular campos, a menudo los accesores get y set simplemente leen o asignan un valor a un campo privado. Incluso en estos casos simples, es preferible usar propiedades en lugar de campos públicos por dos razones:

1. **Compatibilidad con aplicaciones:** Si inicialmente expones campos públicos y luego los cambias a propiedades (por ejemplo, para añadir lógica de negocio), las aplicaciones existentes que usen tu clase deberán recompilarse. Las propiedades se exponen con metadatos diferentes a los campos.
2. **Compatibilidad con interfaces:** Si una interfaz define una propiedad, debes implementarla como tal, no simplemente exponiendo un campo público con el mismo nombre.

Para facilitar la escritura de propiedades simples, C# permite generar el código automáticamente. Simplemente declaras la propiedad con get; set; vacíos

INICIALIZACIÓN DE OBJETOS USANDO PROPIEDADES

Los objetos pueden inicializarse asignando valores a sus propiedades públicas que tengan un accesor set. Esto se conoce como **inicializador de objeto (object initializer)**

Esta sintaxis es más flexible que sobrecargar constructores, especialmente cuando hay muchas combinaciones posibles de campos del mismo tipo. El compilador llama primero al constructor predeterminado (o al especificado), y luego a los *setters* de las propiedades indicadas.

A partir de C# 9.0, existen las **propiedades automáticas de solo inicialización (init-only properties)** utilizando el accesor init. Esto permite inicializar la propiedad solo durante la construcción del objeto (mediante un inicializador de objeto o un constructor) y luego la hace inmutable.

El capítulo concluye con un ejercicio para definir una clase Polygon con propiedades automáticas para NumSides y SideLength, usando inicializadores de objeto para crear diferentes polígonos.

RECORDS CON PROPIEDADES PARA ESTRUCTURAS LIGERAS

Los **records**, introducidos en C# 9.0, son tipos de valor ligeros con soporte integrado para la **inmutabilidad** y una noción de **igualdad basada en valores** más eficiente que la implementación por defecto para estructuras (ValueType.Equals). Facilitan la creación de tipos inmutables al generar automáticamente la infraestructura de código necesaria con una sintaxis concisa, aliviando la verbosidad de implementar la inmutabilidad manualmente con propiedades init-only o de solo lectura en estructuras o clases.

ENUM O ENUMERACIONES Y ESTRUCTURAS

ENUM

Las enumeraciones (palabra reservada: *enum*) son tipos de valor que limitan sus posibles valores a un conjunto de nombres simbólicos, lo que mejora la legibilidad y robustez del código.

```
enum Estaciones
{
    Primavera,
    Verano,
    Otoño,
    Invierno
}
```

Una vez declarada, se usa como cualquier otro tipo, pudiendo crear variables, campos o parámetros de ese tipo.

Internamente, los literales se asocian con valores enteros desde 0 . También es posible asignar valores enteros específicos a los literales y definir el tipo entero subyacente.

```

static void Main()
{
    Estaciones estacionTucuman = Estaciones.Invierno;

    estacionTucuman = Estaciones.Primavera; //Ahora es Primavera
    estacionTucuman = (Estaciones)4; //Ahora es Otoño
}

enum Estaciones
{
    Primavera = 0,
    Verano = 2,
    Otoño = 4,
    Invierno = 6
}

```

ESTRUCTURAS

Son tipos de valor útiles para pequeñas cantidades de datos, ya que su almacenamiento en el Stack reduce la sobrecarga de gestión de memoria en comparación con los objetos en el heap. Al igual que en las clases, pueden tener campos, métodos y constructores.

En C# muchos tipos primitivos son en realidad alias de estructuras de .NET

Se utiliza la palabra clave *struct* seguida del nombre y cuerpo de la estructura. Se recomienda que los campos sean *private* para controlar los valores.

Se puede inicializar una estructura llamando a su constructor. Si no se usa, la variable se queda pero sus campos quedan sin inicializar, lo que causa un error de compilación si se intenta leer.

Para estructuras grandes que pueden afectar el rendimiento debido a la copia completa de la estructura al Stack se pueden pasar por referencia usando *ref*, aunque esos cambios serán permanentes en la estructura original.

Característica	Estructura	Clase
Tipo de valor/referencia	Tipo de valor	Tipo de referencia
Ubicación en memoria	Valores, generalmente en el stack	Objetos, generalmente en el heap
Constructor predeterminado	No se puede declarar uno propio (el compilador siempre genera uno)	Sí se puede declarar (el compilador solo lo genera si no escribes ninguno)
Inicialización de campos en constructor propio	Si declaras un constructor, todos los campos deben inicializarse	Si declaras un constructor, los campos no inicializados se

	explícitamente (no hay inicialización automática)	initializan automáticamente a 0, false o null
Inicialización de campos de instancia en la declaración	No permitido	Sí permitido
Copia de variables	Copia el valor completo (cada campo de la izquierda se inicializa con el correspondiente de la derecha). No afecta a la copia si se modifica el original.	Copia la referencia (ambas variables apuntan al mismo objeto en el heap). Modificar uno afecta al otro.

ARRAYS

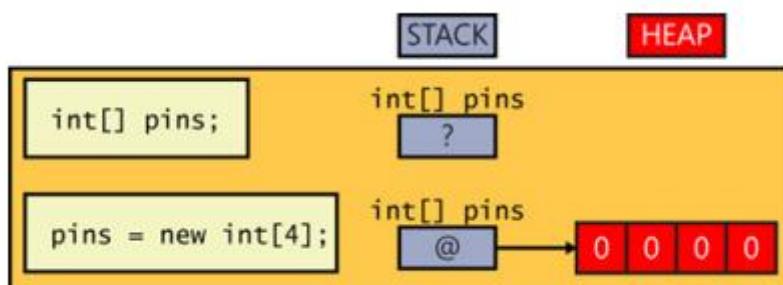
Un arreglo se declara especificando el tipo de elementos seguido de corchetes y luego el nombre del arreglo.

```
int[] numeros = { 1, 2, 3, 4};
```

Para instanciarlo se escribe la palabra clave “new” seguido del tipo de elemento y el tamaño entre corchetes.

```
string[] palabras;
palabras = new string[4];
```

Debido a que la memoria del array es dinámicamente asignada, el tamaño del array no necesita ser constante, se puede calcular en el tiempo de ejecución o runtime.



POBLANDO Y USANDO EL ARRAY

Cuando se crea la instancia de un array, todos sus elementos se inicializan automáticamente a un valor por defecto según su tipo:

- Numéricos: a 0
- Objetos: a *null*
- Datetime: a “01/01/0001 00:00:00”
- string: a *null*

También podemos poblar inicializando con valores específicos separados por comas entre corchetes.

Cuando hacemos arrays de structs u objetos podemos inicializar cada instancia en el arreglo llamando a su constructor.

```
public void Main()
{
    Tiempo[] calendario = { new Tiempo(12, 30), new Tiempo(10, 30) };
}
```

ARRAYS IMPLICITOS

Se omiten los corchetes del tipo y declaramos la variable simplemente como var. Y debe usarse el operador new[] antes de la lista de inicializadoras.

```
var nombres = new[] { "David", "Belén" };
```

Todos los inicializadores deben ser del mismo tipo, si no , habrá un error de compilación.

PUNTOS CLAVE Y COMPROBACIÓN DE LÍMITES

Los índices de los array comienzan en 0. Gracias a esta información podemos acceder a determinada posición del array. Si queremos acceder a un valor que es mayor o igual que la *Length* del array, se lanzará una excepción.

Para acceder a una serie de elementos del array, c# nos permite usar una sintaxis de rango o índices desde el final.

```
var edades = new[] { "24", "18", "30", "25", "22" };

var elementos1 = edades[1..3]; // 18, 30, 25
var elementos2 = edades[^1]; // 22, el último valor
var elementos3 = edades[^2]; // 25, el penúltimo valor
```

También con for o foreach podemos iterar el array.

```
var edades = new[] { 24, 18, 30, 25, 22 };

var suma = 0;
foreach(var edad in edades)
{
    suma = edad + suma;
}
```

ARRAYS COMO PARÁMETROS

La sintaxis es similar a la declaración de un arreglo. Dado que estos son tipos por referencia, si modificamos los elementos dentro de un método, los cambios serán visibles en el arreglo original fuera del método.

ARRAYS COMO VALORES DE RETORNO

Se especifica el tipo de arreglo como tipo de retorno del método, dentro de este se crea y se llena el arreglo y luego se devuelve.

MAIN

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

Main en C# acepta un array de cadenas como parámetro. Este array, *args*, contiene los argumentos de la línea de comandos que se pasaron al ejecutar la aplicación. Permite que las aplicaciones contengan y reciban información de configuración o archivo para procesar al inicio sin una interacción directa del usuario.

COPIA DE ARREGLOS

Recordemos que los arrays son tipos por referencia, por lo que una simple asignación no crea una copia independiente de los datos, si no que ambas referenciarían al mismo arreglo en el heap.

Para crear una copia verdadera, hay varias maneras:

- Copia manual (Elemento por Elemento)
- *CopyTo()* de *System.Array*;
- *Array.Copy()* estático de *System.Array*;
- *Clone()* de *System.Array*;

Estas son Shallow Copy o copias superficiales. En otras palabras que si los elementos a copiar, se copian referencias a esos objetos. Si necesitas una copia profunda, deberías implementar tu propia lógica con un for y copiando cada objeto individualmente.

ARRAYS MULTIDIMENSIONALES

Rectangular:

- *int[4,2]* : Se utiliza una coma dentro de los corchetes para separar las dimensiones. (En este caso creamos un arreglo rectangular 4x2 de ints).
- Se accede a los elementos usando múltiples índices, uno por cada dimensión.
- No hay un límite de dimensiones. Pero, hay que tener precaución ya que ocupan mucha memoria.

Jagged Arrays

- *int[][]* : Es un arreglo unidimensional cuyos elementos son otros arrays.
- Se declaran con los corchetes anidados.
- Es eficiente al no asignar memoria para elementos no utilizados.

COMPORTAMIENTO CON ESTRUCTURAS (TIPOS DE VALOR)

Si un array almacena instancias de una **estructura** (struct Person), los elementos del array son **copias directas de los datos** que residen en el *stack* (o directamente dentro del array si es el caso).

- Cuando recuperas un elemento de este array (ej., `mostYouthful = findYoungest()`), lo que obtienes es una **copia independiente de los datos** de esa estructura.
- Cualquier modificación realizada en esta copia (ej., `mostYouthful.Age++`) solo afectará a esa copia y **no al elemento original almacenado en el array**. La edad de Francesca no se actualizaría en el array.

Solución: Retornar Referencias (ref)

Para modificar directamente un elemento de tipo de valor dentro de un array, debes retornar una **referencia** a ese elemento, no una copia de sus datos. Esto se logra usando la palabra clave `ref`.

Esta técnica permite que las operaciones sobre `mostYouthful` afecten directamente al elemento dentro del array `family`, incluso si `Person` es una struct.

Solo puedes retornar una referencia (`ref`) a datos que **persistan después de que el método haya terminado**. No puedes retornar una referencia a una variable local creada dentro del método (en el `stack`), ya que esta dejaría de existir una vez que el método finalice, lo que resultaría en una "referencia colgante" (*dangling reference*), un error que el compilador de C# previene.

USO DE PARAMETER ARRAYS

Permiten definir métodos que aceptan una cantidad variable de argumentos, incluso de distintos tipos. La palabra clave para declarar es: `params`.

MÉTODOS VARIÁDICOS CON PARAMS

Una forma consiste en utilizar sobrecarga creando múltiples métodos con el mismo nombre pero distintos parámetros dentro del mismo ámbito.

Pero, la alternativa técnica y limpia es usar params, que permite pasar un número indeterminado de argumentos como si fueran un arreglo.

```
public void Main(string[] args)
{
    ImprimirNumeros(1, 2, 3, 4);
    ImprimirCualquierCosa("Hola", 12, true, 0.2);
}

public void ImprimirNumeros(params int[] numeros)
{
    foreach (int n in numeros)
    {
        Console.WriteLine(n);
    }
}

public void ImprimirCualquierCosa(params object[] cosas)
{
    foreach (object c in cosas)
    {
        Console.WriteLine(c);
    }
}
```

COMPARACIÓN ENTRE PARAMS Y PARÁMETROS OPCIONALES

Params:

- Representa una colección variable de argumentos
- Se usa cuando no se conoce cuantos argumentos va a proporcionar el usuario
- El método sabe cuantos elementos recibió y puede recorrerlos
- Usa Params cuando:
 - El número de argumentos es incierto o flexible.
 - El método procesará una colección de elementos.

Parámetros Opcionales:

- Cuando hay un número fijo de parámetros, pero algunos no siempre se especifican
- El compilador inserta automáticamente los valores predeterminados si no se proporcionan.
- Usa parámetros opcionales cuando:
 - Tienes una firma de método clara con valores predefinidos.
 - Quieres simplificar llamadas comunes sin sobrecargar.

Característica	params	Parámetrosopcionales
¿Cantidad de argumentos variable?	Sí, cualquier cantidad	No, la lista de parámetros es fija

¿Se pueden mezclar tipos?	Sí (si se usa params object[])	No, los tipos deben coincidir
¿Puede saber el método cuántos argumentos se pasaron?	Sí (paramList.Length)	No, no puede diferenciar entre valores por defecto y explícitos
¿Argumentos tienen valores por defecto?	No	Sí, definidos por el programador
¿Puede pasarse cero argumentos?	Sí (por ser array)	Sí, si todos tienen valores por defecto
¿Puede sobrecargarse junto a la otra técnica?	Con cuidado	Con cuidado, puede haber ambigüedad

INTERFACES

Una **interfaz** especifica *nombres, tipos de retorno y parámetros de métodos* sin definir su implementación. **Actúa como contrato:** Cualquier clase que implemente la interfaz, debe implementar **TODOS** sus métodos

```
interface IIImprimible<T>
{
    void Imprimir(T valor);
}
```

- No se permiten campos
- No se permiten constructores
- No se permiten destructores
- Acceso implícitamente público: No puedes especificar modificadores de acceso (como private o protected) para los métodos en una interfaz. Todos los métodos de una interfaz son implícitamente públicos.
- No se permiten tipos anidados: No puedes anidar otros tipos (como enumeraciones, estructuras, clases o incluso otras interfaces) dentro de una interfaz.
- Herencia limitada: Una interfaz no puede heredar de una estructura, registro o clase. Solo puede heredar de otra interfaz.
- Por convención, Inicia con una “I” mayúscula.

IMPLEMENTACIÓN

Una clase o estructura puede implementar interfaces

```
class Cerveza : Bebida, IBebidaAlcoholica
{
    public int cantidadAlcohol { get; set; }
    public Cerveza(string marca, int cantidadAlcohol)
    {
        this.marca = marca;
        this.cantidadAlcohol = cantidadAlcohol;
    }
}
class Bebida
{
    public string? marca;
}
interface IBebidaAlcoholica
{
    public int cantidadAlcohol { get; set; }
}
```

Cuando tenemos una clase, esta solo hereda de un parente, en cambio, podemos implementar múltiples interfaces.

REFERENCIANDO UNA CLASE POR SU INTERFAZ

Una clase se puede referenciar a través de una interfaz.

```
IBebidaAlcoholica quilmes = new Cerveza("Quilmes", 5);
```

Esto permite que los métodos reciban múltiples tipos con una misma interfaz

IMPLEMENTACIÓN EXPLÍCITA

Normalmente una clase implementa una interfaz de forma implícita. Sin embargo, esto puede generar un problema cuando una clase implementa múltiples interfaces que tienen métodos con el mismo nombre, pero con significados diferentes.

Para resolver esta ambigüedad se utiliza la implementación explícita de interfaces. Una característica importante de los métodos implementados explícitamente es que no se les puede aplicar modificadores de acceso como public, lo que los hace privados para la clase que los implementa. Esto significa que no se puede llamar a estos métodos directamente de una instancia de la clase, en su lugar para acceder se debe referenciar el objeto a través de la interfaz correspondiente.

En resumen, usá implementación explícita cuando sea posible para evitar confusiones y mejorar la claridad del código.

MANEJO DE VERSIONES CON INTERFACES

Las interfaces nos facilitan la colaboración en proyectos complejos, y nos ayuda a añadir extensibilidad al sistema. Permiten a los desarrolladores definir un contrato (métodos y operaciones) que otras clases pueden implementar, lo que hace que los tipos sean intercambiables.

Antiguamente había algunos problemas caóticos y difíciles para mantener la evolución en cadenas complejas de interfaces. Visual C# 8 provee una implementación por defecto de un método en una interfaz de la siguiente manera:

```
public interface IMiInterfaz
{
    public void HacerAlgo();
    public void HacerAlgoMas()
    {
        throw new NotImplementedException();
    }
}
```

Con esto, cualquier tipo que implemente la interfaz pero no proporcione su propia versión de HacerAlgoMas() usará automáticamente la implementación por defecto. Aunque se puede poner cualquier cosa, lo común es lanzar una excepción, esto asegura que las aplicaciones no se rompan y las nuevas clases puedan crear un método con la misma firma sin usar `override`.

CLASES ABSTRACTAS (ABSTRACT CLASS)

Cuando trabajamos con múltiples clases o implementaciones comunes, es muy probable que haya duplicación de código, por lo que nos conviene refactorizar en una clase abstracta.

Al declarar esta clase como abstract, el compilador impide que se creen instancias sobre ella. De esta manera, las clases abstractas sirven como plantillas o bases para otras clases, forzando la herencia y asegurando que solo las clases concretas puedan ser instanciadas

```
abstract class Animal{}
class Perro : Animal{/*Cuerpo*/}
class Caballo : Animal{/*Cuerpo*/}
```

Una clase abstracta, puede tener **métodos abstractos**. En principio, son similares a los métodos virtuales, excepto que no contienen un cuerpo de método. Una clase derivada va a sobreescribir

(*override*) este método. No pueden ser privados.

```
abstract class Animal
{
    public abstract string? HacerSonido();
}

class Perro : Animal
{
    public override string? HacerSonido()
    {
        return "Guau guau";
    }
}

class Caballo : Animal
{
    public override string? HacerSonido()
    {
        return "Hii PFFF";
    }
}
```

CLASES SELLADAS(SEALED CLASS)

Si un desarrollador decide que una clase no debe ser heredada puede usar la palabra clave *sealed* para evitarlo.

- Una clase *sealed* no puede declarar ningún método virtual.
- Una clase *abstract* no puede ser *sealed*.

También se pueden hacer métodos *sealed* o sellados.

GENÉRICOS (GENERIC)

Genéricos es muy importante. Para entenderlo, primero debemos entender el problema a resolver.

Antes, se utilizaba object para utilizar cualquier tipo de datos.

```
class Cola
{
    private object[] datos;
    // . . . Métodos
}
```

Claro, nos da flexibilidad para almacenar cualquier tipo de dato en una misma estructura. Pero, tenemos:

- Una falta de seguridad de tipos en tiempo de compilación: El compilador no sabe que dato estás almacenando realmente. Si recuperamos un object y asumimos que es de un tipo, y es de otro tendremos un error en tiempo de ejecución.
- Costos de rendimiento del boxing y unboxing.

Los genéricos permiten escribir clases, interfaces y métodos que operan con **tipos parametrizados** en lugar de usar un tipo concreto. Se utilizan marcadores de posición de tipo y el tipo real se especifica cuando se usa la clase o el método genérico. Son la solución al problema. Podemos reusar el código, al eliminar unboxing y boxing mejoraremos el rendimiento y a su vez, tendremos un problema de compilación en cualquier caso, no en tiempo de ejecución.

Para definir una clase genérica, se agregan parámetros de tipo entre corchetes angulares después del nombre de la clase

```
public class Ejemplo<T>
{
    //Ahora, se puede usar T como si fuera un tipo real dentro de la clase
    private T datos;
    public void Add(T animales) { }
    public T Get() { return datos; }
}
```

Cuando lo instanciamos y usamos la clase, especificamos el tipo concreto:

```
Ejemplo<int> ejemplo = new Ejemplo<int>();
Ejemplo<List<string>> ejemplo2 = new();
```

PARÁMETROS

A veces, necesitamos que el tipo que se sustituya por el parámetros, cumpla con ciertos requisitos, por ejemplo, que sea comparable, que sea una clase o que tenga un constructor sin parámetros. Para esto, se utilizan las cláusulas where. Algunas de las restricciones comunes son:

- *where T: class* (T debe ser un tipo de referencia)
- *where T: struct* (T debe ser un tipo de valor)
- *where T: new()* (T debe tener un constructor público sin parámetros)
- *where T: NombreDeClaseBase* (T debe heredar de NombreDeClaseBase)
- *where T: NombreDeInterfaz* (T debe implementar NombreDeInterfaz)

```

public void Main(string[] args)
{
    Ejemplo<int> ejemplo = new Ejemplo<int>();
    Ejemplo<List<string>> ejemplo2 = new(); //No es comparable
}

public class Ejemplo<T> where T : IComparable<T>
{
    //Ahora, se puede usar T como si fuera un tipo real dentro de la clase
    private T datos;
}

```

MÉTODOS GENÉRICOS

Además de esto, se pueden definir métodos genéricos. El parámetro de tipo se especifica antes de la lista de parámetros del método

```

static void Main(string[] args)
{
    Impresora.Imprimir(1);
    Impresora.Imprimir("Jamón");
    Impresora.Imprimir(Math.PI);
    Impresora.Imprimir(new DateTime(2023, 10, 1));
}

public class Impresora
{
    public Impresora() { }
    public static void Imprimir<T>(T value)
    {
        Console.WriteLine($"El valor es: {value}({value.GetType().Name})");
    }
}

```

INTERFACES GENÉRICAS

Anteriormente, vimos que se puede asignar un string a una referencia object del tipo:

```
object Objeto = "Hola Mundo";
```

Pero, esto no se aplica automáticamente a las interfaces genéricas por defecto.

Supongamos que:

Tenemos una interfaz genérica llamada ICambiador<T> y una clase Cambiador<T> que la implementa.

Si intentamos asignar una instancia de Cambiador<string> a una referencia ICambiador<object> el código no compilará. O si un cast explícito, compilará pero fallará en runtime.

Esto pasa porque ICambiador<T> es por defecto **invariante**.

```

class Cambiador<T> : ICambiador
{
    public void Cambiar(T)
    {
        Console.WriteLine("El objeto ha sido cambiado.");
    }
}

interface ICambiador <T>
{
    void Cambiar(T);
}

```

Si un método en una interfaz genérica puede retornar strings, también puede retornar objetos.
(COVARIANTE)

Si un método en una interfaz genérica puede tomar parámetros de tipo objeto, puede tomar parámetros de tipo string.

INTERFACES COVARIANTES (OUT)

La covarianza permite que una interfaz genérica sea más flexible en la asignación de tipos, de manera que se pueda usar un tipo más específico(derivado) donde se espera uno más genérico (base), pero solo bajo condiciones estrictas.

- Permite asignar `IInterfaz<TipoDerivado>` a `IInterfaz<TipoBase>`
- out solo aparece como tipo de retorno de los métodos de la interfaz, nunca como parámetro de entrada.
- Solo funciona con tipos de referencia y out T no puede usarse como tipo de ningún parámetro de método en la interfaz.
- `IEnumerable<T>` es covariante.

INTERFACES CONTRAVARIANTES(IN)

Es el principio opuesto a la covarianza, permite usar un tipo más genérico(base) donde se espera uno más específico (derivado).

- Permite asignar `IInterfaz<TipoBase>` a `IInterfaz<TipoDerivado>`
- in se usa cuando el parámetro T de la interfaz solo aparece como tipo de parámetro de entrada de los métodos (nunca como tipo de retorno)
- Solo funciona con tipos de referencia
- in T no puede usarse como tipo de retorno de ningún método en la interfaz.
- `IComparer<T>` es una contravariante.

COLECCIONES (<T>)

Son clases que agrupan elementos y permiten acceder a ellos de maneras especializadas. Se encuentran en el namespace : System.Collections.Generic.

Son genéricas, ósea que esperan proporciones de un parámetro de tipo (como T para List<T>), lo que les permite ser seguras y eficientes.

MÉTODOS DE BÚSQUEDA EN COLECCIONES

Buscar en colecciones orientadas a diccionario (Dictionary, SortedDictionary, SortedList):

- Usamos su clave
- Utilizan notación de corchetes.

Buscar en colecciones con acceso no clave (como List o LinkedList)

- Proporcionan Find para localizar un elemento
 - Devuelve un bool indicando si el elemento coincide con el criterio de búsqueda.
 - Devuelve el primer elemento que cumple con el predicado
 - FindLast: Devuelve el último objeto que coincide
 - List.FindAll: Devuelve una List con todos los objetos que coinciden. (LinkedList no es admitido)

EXPRESIONES LAMBDA

Son una forma concisa y flexible de definir funciones o métodos anónimos directamente en el lugar donde se necesitan.

Las expresiones lambda nos permite omitir las llaves y el return. Y si la expresión tiene un único parámetro, se pueden omitir los paréntesis que lo rodean. En muchos casos, el compilador puede inferir el tipo, así que se puede omitir. (NO CONFUNDIR CON EL OPERADOR =>). Sintaxis:

(parámetros) => { cuerpo }

Diferentes Formas (Sintaxis):

- x => x * x: Expresión simple que devuelve el cuadrado del parámetro x. El tipo de x se infiere del contexto.
- x => { return x * x; }: Semánticamente igual a la anterior, pero usando un bloque de sentencias de C# como cuerpo.
- (int x) => x / 2: La expresión indica explícitamente el tipo del parámetro x.
- () => folder.StopFolding(0): Expresión sin parámetros que llama a un método. Puede o no devolver un valor.
- (x, y) => { x++; return x / y; }: Múltiples parámetros; el compilador infiere sus tipos. La operación x++ es local a la expresión (si x es por valor).
- (ref int x, int y) => { x++; return x / y; }: Múltiples parámetros con tipos explícitos. x se pasa por referencia (ref), por lo que los cambios (x++) son permanentes fuera de la expresión.

Características Importantes de las Expresiones Lambda:

- **Parámetros:** Se especifican entre paréntesis a la izquierda del operador =>. Sus tipos pueden inferirse o declararse explícitamente. Se pueden pasar por referencia (ref), aunque no es lo más recomendado.

- **Valores de Retorno:** Pueden devolver valores, pero el tipo de retorno debe coincidir con el delegado correspondiente.
- **Cuerpo:** Puede ser una expresión simple (sin {} ni return) o un bloque de código con múltiples sentencias, llamadas a métodos, definiciones de variables, etc.
- **Ámbito de Variables Locales:** Las variables definidas dentro del cuerpo de una lambda tienen un ámbito local y desaparecen al finalizar la ejecución de la lambda.
- **Acceso a Variables Externas :** Una expresión lambda puede acceder y modificar variables definidas fuera de ella, siempre que esas variables estén en el ámbito cuando la lambda se define.

LIST<T>

Es la más simple. Se puede usar casi como un array. En general, nos da más flexibilidad que un array

- No necesita especificación de capacidad al crearse. Puede crecer y ajustarse según se añaden elementos. Si es necesario, se puede especificar un tamaño inicial pero si se excede, la colección simplemente crece según lo necesario
- Se puede remover un elemento utilizando *Remove*. La lista se reordena automáticamente. También se puede remover un item en una cierta posición usando *RemoveAt*.
- Se puede añadir un elemento al final utilizando *Add*
- Se puede añadir en el medio utilizando *Insert*
- Se puede ordenar utilizando *Sort*
- Se puede usar *foreach*
 - NO se puede usar la variable de iteración para modificar los contenidos de la colección.

LINKEDLIST<T>

Es una lista doblemente enlazada, ósea que cada elemento guarda su valor Y una referencia al elemento siguiente y al anterior.

- El primer elemento de la lista tiene *Previous* en *null*
- El último elemento de la lista tiene su *Next* en *null*

No permite acceder, insertar o examinar elementos usando notación de corchetes como List

- **AddFirst(item):** Inserta un elemento al principio de la lista.
- **AddLast(item):** Inserta un elemento al final de la lista.
- **AddBefore(existingItem, newItem):** Inserta un elemento *newItem* antes de un *existingItem* dado. Primero debes obtener una referencia al *existingItem* en la lista
- **AddAfter(existingItem, newItem):** Inserta un elemento *newItem* después de un *existingItem* dado. Primero debes obtener una referencia al *existingItem* en la lista
- Uso de: *First*, *Last* para elementos específicos.
- Iteración:
 - Se puede recorrer manualmente empezando por *First* o *Last*, siguiendo con *Next* o *Previous* hasta encontrar un *null*.
 - O, un *foreach* desde el principio hasta el final.
- Eliminación:
 - *Remove(item)*

- RemoveFirst(item)
- RemoveLast(item)

QUEUE<T>

Implementa FIFO (First in In First in Out, El primero en entrar es el primero en salir). Un elemento es insertado a la queue o cola en el final (operación enqueue) y es removido de la cola por el frente (operación dequeue).

PRIORITYQUEUE<T>

Es una extensión de la queue tradicional, añade el concepto de prioridad. Osea que el dequeue se basa en prioridad (mayor prioridad sale primero); si las prioridades son iguales se usa FIFO.

Usa dos parámetros : Enqueue(item, prioridad);

STACK<T>

Es una colección que simula una pila física. Los elementos se añaden y retirar desde el mismo extremo. Usa LIFO (Last in In, First in Out, último en entrar, primero en salir).

- **Push:** Añadir al extremo
- **Pop:** Quitar del extremo.

DICTIONARY<T>

Permite el mapeo de clave-valor. Almacena pares con claves únicas y las utiliza para acceder rápidamente a su valor asociado. Es crucial entender que las claves no pueden repetirse, que consume más memoria para ser más eficiente y que al iterar, se obtienen pares *KeyValuePair* de solo lectura.

- ContainsKey() : Verifica la existencia de una clave
- foreach Devuelve => KeyValuePair< TKey, TValue >.
 - .Key : muestra la clave
 - .Value : muestra el valor

SORTEDLIST<T>

Parecido a Dictionary, pero mantiene sus elementos ordenados automáticamente por la clave. La inserción puede ser un poco más lento, pero usa menor memoria.

- foreach Devuelve => KeyValuePair< TKey, TValue >.

HASHSET<T>

Está diseñada para manejar conjuntos matemáticos con operaciones de conjuntos (Unión, intersección, diferencia, subconjunto, superconjunto)

- Utiliza una tabla hash para búsquedas rápidas.
- Puede requerir bastante memoria
- No está ordenado
- No permite elementos duplicados
- Operaciones de modificación(modifican el conjunto original)

- IntersectWith
- UnionWith
- ExceptWitch
- Operaciones de Consulta (devuelven bool, no modifican)
 - IsSubsetOf
 - IsSupersetOf
 - IsProperSubsetOf
 - IsProperSupersetOf

DELEGADOS Y EVENTOS (DELEGATE & EVENT)

DELEGADOS

Un delegado es esencialmente una referencia a un método. Se les llama así porque “delegan” el procesamiento al método al que hacen referencia cuando son invocados. Es un concepto simple, pero con implicaciones muy potentes.

Normalmente, se llama a un método especificando su nombre y tu código sabe exactamente qué método se está ejecutando.

Una sentencia asignada al delegado no ejecuta el método en ese momento. Es simplemente una sentencia de asignación. Además, un delegado puede referenciar más de un método a la vez.

.NET utiliza delegados en muchos de sus tipos. Por ejemplo en los métodos *Find* y *Exists* de la clase *List<T>*. Cuando los diseñadores de esta clase, no tenían idea de que debería constituir realmente una coincidencia en el código de tu aplicación. Por lo que, te permiten definirla proporcionando tu código en forma de predicado. **Y, un predicado es en realidad un delegado que devuelve un valor booleano.**

FUNC Y ACTION

En List<T> también utilizamos operaciones como Average, Max, Min, Count o Sum, estos métodos toman un delegado Func como parámetro . Este se refiere a un método que devuelve un valor (una función). Puede tener muchos parámetros de entrada pero solo uno de salida.

Por ejemplo List<T> es del tipo Func<T, TResult>

Además de Func, también tenemos Action. Este se utiliza para referenciar a un método que realiza una acción en lugar de devolver un valor.

Supongamos el ejemplo de una fábrica. Tenemos 3 máquinas:

```
class MaquinaPlegado
{
    public void DetenerPlegado() {/* . . . */ }
}
class MaquinaSoldadura
{
    public void DetenerSoldadura() { /* . . . */ }
}
class MaquinaPintura
{
    public void DetenerPintado() { /* . . . */ }
}
```

Deseo tener una clase controlador que detenga mis máquinas todas juntas:

```
class Controlador
{
    private MaquinaPlegado plegado;
    private MaquinaSoldadura soldadura;
    private MaquinaPintura pintura;

    public void DetenerTodo()
    {
        plegado.DetenerPlegado();
        soldadura.DetenerSoldadura();
        pintura.DetenerPintado();
    }
}
```

Aunque este enfoque funciona, no es muy extensible ni flexible. Por cada máquina nueva que la fábrica obtenga, se debería modificar este código.

Aquí es donde un delegado puede ser útil. Se puede usar un delegado que coincida con la forma

EVENTOS

En .NET se proporcionan eventos, los cuales se pueden utilizar para definir y atrapar acciones significativas y hacer que se llame a un delegado para manejar la situación.

Se declara un evento en una clase destinada a actuar como fuente de eventos, este monitoriza su entorno y lanza un evento cuando ocurre algo significativo.

Supongamos un ejemplo de una fábrica automatizada. Una fuente de eventos podría ser una clase que monitoriza la temperatura de cada máquina. Si detecta que una máquina excedió su límite de radiación térmica, la clase de monitorización lanzaría un evento.

Suscriptores: Lista de métodos a llamar cuando se lanza un evento

Un evento se declara un campo, pero como están destinados a utilizar con delegados, debe ser un delegado y se debe prefijar la declaración con la palabra clave **evento**:

```
event tipoDelegado nombreEvento;
```

Al igual que los delegados, los eventos vienen listo para usarse con un operador += y se usa para **suscribirse a un evento**. Incluso se puede suscribir utilizando una expresión lambda.

Para desvincular un delegado de un evento se puede utilizar el operador -=, este elimina el método de la colección interna de delegados del evento. Esta acción se denomina **cancelar la suscripción al evento**.

Para lanzar un evento, se puede llamar como un método.. Luego, todos los delegados adjuntos se invocan en secuencia.

Es una práctica común verificar la nulidad.

En .NET para construir las GUIs, se emplean eventos de forma extensiva. Cuando se usa la vista de diseñador, se escribe la lógica de la aplicación en el método de manejo de eventos. Siempre siguen el mismo patrón.

Son un tipo de delegado cuya firma tiene void como tipo de retorno y dos argumentos.

El primer argumento siempre es el *sender* y el segundo siempre *EventArgs*

Con el *Sender* se puede reutilizar un único método para múltiples eventos.

LINQ

Está inspirado en SQL y nos permite simplificar y flexibilizar la consulta y manipulación de datos en memoria. LINQ es más versátil que SQL en general.

LINQ requiere una estructura de datos que implementen `IEnumerable` o `IEnumerable<T>`. No importa cual se use mientras sea enumerable.

SELECT

Es fundamental, nos permite extraer datos específicos de una colección. Cuando utilizamos `Select` con una `IEnumerable<T>`, pasamos una expresión lambda que define como transformar cada elemento de la colección. Esta expresión es nuestro selector que toma un elemento de entrada (Como por ejemplo, `id`) y nos devuelve los datos que nos interesan (Por ejemplo, El nombre de ese cliente).

`Select` es un método de extensión de `Enumerable`. Opera con evaluación diferida, ósea que cuando se lo llama, los datos no se procesan de inmediato, sino que se devuelve un object con la “receta” de la consulta. La recuperación y ejecución real ocurre cuando se itera sobre este objeto con, por ejemplo, un `foreach`.

Si se necesita extraer varios datos de cada elemento, ya que `Select` devuelve una colección con un único tipo, tenemos las siguientes opciones:

- Concatenar datos en una sola cadena: Uniendo los datos en una única cadena dentro de la expresión lambda del `Select`
- Crear un tipo anónimo que contenga las propiedades que queremos extraer.

Con `Select` se pueden restringir los elementos que esa colección contiene. Por ejemplo, podemos utilizar `where` para filtrar compañías que tengan ubicación==”Tucuman”.

Sintácticamente, `where` es similar a `Select`. Espera un parámetro que define un método (Usualmente una expresión lambda) que filtra los datos según criterios especificados.

ORDENAMIENTO, AGRUPACIÓN Y AGREGACIÓN DE DATOS

LINQ nos permite utilizar **OrderBy**, que espera un método (lambda) como argumento. Este método identifica la expresión que se quiere utilizar para ordenar los datos.

Si se quiere enumerar los datos en orden descendente, se puede utilizar **OrderByDescending**.

Para ordenar por más de un valor clave, **ThenBy** o **ThenByDescending** después de **OrderBy** u **OrderByDescending**. También, **Distinct** para eliminar duplicados del cálculo.

JOIN

Te permite combinar datos de una colección enumerable con otra colección enumerable. Los parámetros clave son:

- La colección con la que se quiere unir (tabla a la derecha)
- Un método que identifica el campo clave común de la colección de la izquierda
- Un método que identifica el campo clave común de la colección de la derecha
- Un método que especifica las columnas que se necesita en el conjunto de resultados enumerable que devuelve el método `Join`

Las **colecciones en memoria no son lo mismo que las tablas en una base de datos relacional**, y los datos que contienen no están sujetos a las mismas restricciones de integridad de datos. En una base de datos relacional, podrías asumir que cada cliente tiene una compañía correspondiente y que cada compañía tiene su propia dirección única. Las colecciones no imponen el mismo nivel de integridad de datos, lo que significa que es bastante fácil tener un cliente que referencia una compañía que no existe en el array `addresses`, o incluso que la misma compañía aparezca más de una vez en el array `addresses`. En estas situaciones, los resultados que obtenés pueden ser precisos pero inesperados. Las operaciones `Join` funcionan mejor cuando comprendés completamente las relaciones entre los datos que estás uniendo.

OPERADORES LINQ

Si bien los métodos anteriores, la sintaxis con lambdas y encadenamiento puede a veces ser compleja de leer y mantener. Para simplificar esto, los diseñadores de C# agregaron operadores de consulta al lenguaje, que permiten usar las características LINQ con una sintaxis más parecida a SQL:

Por ejemplo para recuperar el nombre de cada cliente, antes hacíamos así:

```
IEnumerable<string> nombres = cliente.Select(c.Nombre);
```

Ahora:

- var nombresClientes = from cliente in clientes select cliente.Nombre;
- var nombreClientes = from cliente in clientes select new{cliente.Nombre,cliente.Apellido};

Podemos utilizar:

- where: Para filtrar datos:
 - var companiasArg = from a in direccion where String.Equals(a.Pais,"Argentina")
select a.NombreCompania
- orderby: Para ordenar datos en ascendente por defecto
 - var nombreCompanias = from a in direcciones orderby a.NombreCompania select a.NombreCompania
 - Se puede usar orderby a.CompanyName descending
- group by: Para agrupar valores comunes por uno o más campos
 - var companiasAgrupadasPorPais = from compania in direcciones group compania by compania.Pais
- Count() y Distinct()
- Join: LINQ solo soporta uniones basadas en igualdad. Y a diferencia de SQL el orden de on es crucial. El elemento de from debe ir a la izquierda, y el elemento de join a la derecha:
 - var paisYClientes= from dire in direcciones
join clie in clients
on dire.NombreCompania equals clie.NombreCompania
select new { clie.Nombre, clie.Apellido, dire.Pais};
- Intersect, Union
- Any, All: Verificar si algún elemento cumple con el predicado
- Take, Skip: Particionar los valores en una colección enumerable

PROGRAMACIÓN ASINCRÓNICA (ASYNC)

Es un modelo de lenguaje que facilita la escritura que puede realizar operaciones sin bloquear el hilo principal de la ejecución. Es crucial para actividades que pueden tardar mucho, como por ejemplo, acceder a la web, operaciones de (I/O) o cálculo complejos que harían parecer la aplicación congelada.

Palabras clave: *async, await* junto con *Task, Task<T>*

Async: Marca un método como asíncrono. Indica al compilador que puede contener expresiones *await* y que su ejecución podría suspenderse temporalmente.

Await: Se usa dentro de *async* para “esperar” la finalización de la operación asíncrona. Cuando el código llega a un *await*, el control se devuelve al llamador del método. Esto significa que el hilo que estaba usando *async* queda libre para hacer otras cosas, como por ejemplo, mantener la interfaz responsive mientras la tarea se completa en segundo plano. Una vez se finaliza la tarea, *async* se reanuda desde el punto donde se suspendió

Task y Task<T>: Son objetos que modelan una operación asíncrona. Un *Task* representa una operación que no devuelve un valor, mientras que *Task<T>* devolverá un *T*. Cuando un método *async* se suspende debido a un *await*, devuelve un *Task* o *Task<T>* a su llamador, permitiendo que el llamador también espere por su finalización o continue con otras tareas.

El compilador de C# transforma el código *async/await* en una **máquina de estado**. Esta máquina de estado es la que se encarga de:

- Realizar un seguimiento del progreso del método asíncrono.
- Suspender la ejecución en los puntos *await*.
- Reanudar la ejecución cuando la tarea en segundo plano finaliza.

- Manejar la propagación de resultados y excepciones.

Beneficios y Usos:

- **Interfaces de Usuario Responsivas:** Evita que la aplicación se "congele" al realizar operaciones que consumen mucho tiempo, permitiendo que el usuario siga interactuando.
- **Escalabilidad:** Mejora la capacidad de una aplicación (especialmente las aplicaciones web) para manejar múltiples peticiones concurrentemente, al liberar hilos que de otro modo estarían bloqueados esperando operaciones externas.
- **Eficiencia de Hilos:** Utiliza los hilos de manera más eficiente al no bloquearlos innecesariamente mientras se espera una respuesta de una operación de I/O (por ejemplo, una descarga de internet o una consulta a una base de datos). Para operaciones ligadas a la CPU (cálculos intensivos), Task.Run se combina con async/await para delegar el trabajo a un grupo de hilos.
- **Código más Limpio:** Permite escribir código asincrónico de una manera que se parece mucho al código sincrónico, evitando las complejidades de las devoluciones de llamada anidadas (callbacks).

UNIDAD 3

PRIMERA PARTE

Definición

Vistas

Proceso

Arquitectura



Marco
Teórico

Requisitos
Funcionales
y No
Funcionales

Estilos

Estándares

MARCO TEORICO

CONCEPTOS

INTERFACES

Es una colección de operaciones que sirven para especificar un servicio de una clase o un componente. Una interfaz bien estructurada nos da una clara separación entre las vistas externas e internas de una abstracción, haciendo posible comprender y abordar una abstracción sin tener que sumergirse en los detalles de su implementación.

Ejemplo: Un cable USB-C a USB-A, es una interfaz. Nos permite conectar dos componentes sin importar cuales sean.

COMPONENTES

Un componente es una parte lógica y reemplazable de un sistema que conforma y proporciona la realización de interfaces.

Los buenos componentes definen abstracciones precisas, con interfaces bien definidas.

PAQUETES

Se utilizan para organizar los elementos de modelado en partes mayores que se pueden manipular como un grupo. La visibilidad puede controlarse.

Un paquete bien diseñado agrupa elementos cercanos semánticamente que suelen cambiar juntos

Representación graficologica de una arquitectura

ARTEFACTOS Y NODOS

Se utilizan para modelar los elementos físicos que pueden hallarse en un nodo. Por ejemplo, ejecutables, bibliotecas, tablas, archivos y documentos.

Un nodo es un elemento físico que existe en runtime y representa un recurso computacional. Un nodo representa un procesador o un dispositivo sobre el que se pueden desplegar los artefactos.

Los buenos nodos representan con claridad el vocabulario de hardware en el dominio de la solución

PRINCIPIOS

Son guías generales o reglas fundamentales que ayudan a diseñar, escribir y mantener código de buena calidad. Buscan mejorar la legibilidad, mantenibilidad, escalabilidad y robustez de las aplicaciones, facilitando el trabajo en equipo y la evolución del software a lo largo del tiempo.

A la hora de diseñar un sistema, ayudan a crear una arquitectura que se ajuste a las prácticas demostradas, que minimicen los costes de mantenimiento y maximicen la usabilidad y extensibilidad.

SEPARACION DE INTERESES (SEPARATION OF CONCERNS)

A un nivel bajo, se relaciona con el Principio de Responsabilidad Única, evitando que diferentes funcionalidades se mezclen en el código o diseño.

La separación de preocupaciones suele ser una consecuencia natural de seguir el principio "**No te repitas**" (**DRY**), ya que obliga a crear abstracciones que encapsulan conceptos repetidos. Si estas abstracciones están lógicamente agrupadas, se logrará la separación de preocupaciones.

Finalmente, la separación de preocupaciones es un principio rector de la **Arquitectura Limpia**, que enfatiza una fuerte división entre la lógica de negocio y las decisiones técnicas.

SOLID

Single Responsibility Principle (Responsabilidad Única):

- Una clase debe tener una y solo una razón para cambiar.

Open Close Princple (Abierto/Cerrado)

- Las entidades deben estar abiertas para extensión pero cerradas para modificación

Liskov Substitution Principile (Substitución de Liskov)

- Si un programa utiliza un tipo base, debe poder usar cualquier subtipo de ese tipo base sin que el programa se rompa. Los objetos deben ser reemplazables por instancias de sus subtipos sin alterar la corrección de ese programa.

Interface Segregation Principle (Segregación de Interfaces o ISP)

- Es mejor tener muchas interfaces pequeñas y específicas que una interfaz grande y monolítica

Dependency Inversion Principile (Inversión de Dependencias)

- Los módulos de alto nivel no deben depender de módulos de bajo nivel.

PATRONES

Son soluciones habituales a problemas comunes en el diseño de software. Cada patrón es como un plano que se puede resolver un problema de diseño particular del código.

ADAPTER O ADAPTADOR

Es un patrón que permite la colaboración entre objetos con interfaces incompatibles

FACADE

Proporciona una interfaz simplificada a una biblioteca, un frameworks o cualquier otro grupo complejo de clases

INYECCIÓN DE DEPENDENCIAS

Un patrón donde un objeto no crea directamente sus dependencias, sino que estas le son proporcionadas externamente.

MVC O MODELO VISTA CONTROLADOR

Busca separar las responsabilidades en:

- Modelo: Representa la lógica de negocio y los datos de la aplicación

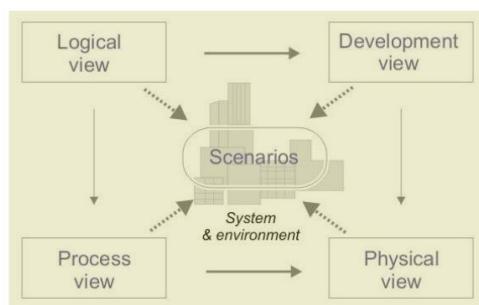
- Interactua con la base de datos
 - Contiene reglas de validación y lógica para manipular la información
 - Es independiente de la interfaz de usuario
- Vista
 - Se encarga de la presentación de datos al usuario
 - Define como se muestra la información(interfaz)
 - No tiene lógica de negocio.
- Controlador
 - Actua como intermediario entre el modelo y la vista
 - Recibe solicitudes del usuario, como un URL o un evento.
 - Procesa esas solicitudes
 - Decide que vista debe mostrarse y le pasa los datos relevantes del modelo
 - No tiene lógica de negocio.

ARQUITECTURA

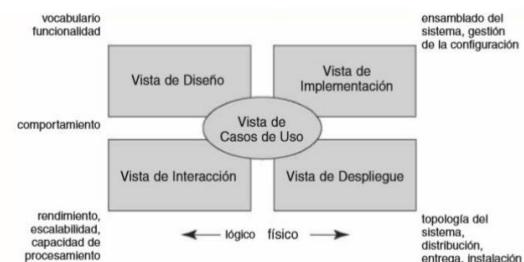
La Arquitectura de Software es el conjunto de las decisiones de diseño fundamentales y significativas sobre la organización de un sistema de software. Estas decisiones abarcan:

- Estructuras del sistema
- Estilo y patrones
- La forma en que el sistema se divide y como sus partes colaboran con requisitos funcionales y requisitos de calidad
- Abstracciones de alto nivel.

Es imposible representar toda la información relevante sobre la arquitectura en un solo modelo arquitectónico. Para el desarrollo, por lo general, se necesita presentar múltiples vistas de la arquitectura de software

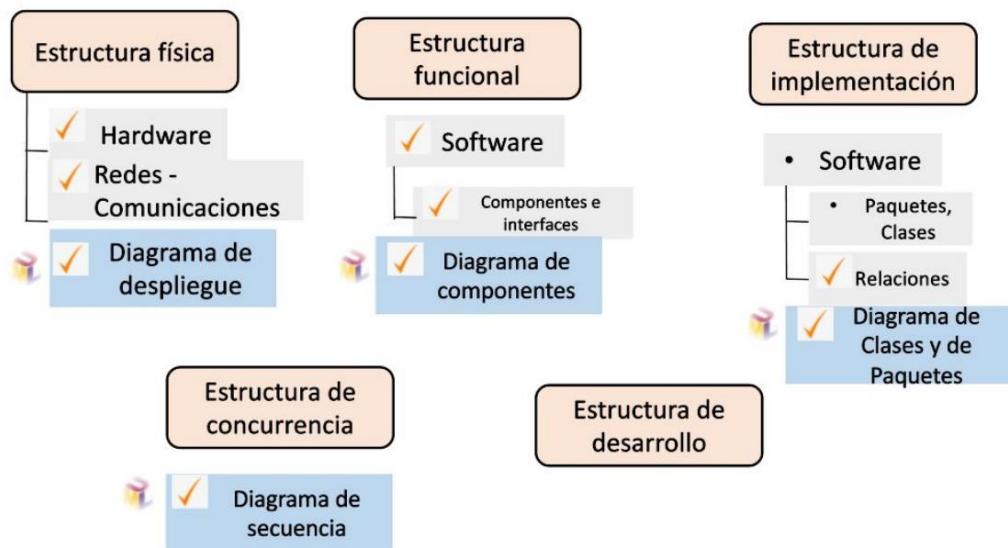


Vistas 4 + 1 (Kruchten)



5 vistas del UP

Diagramas para modelar la Arquitectura



El Proceso de Diseño de la Arquitectura



CLEAN ARCHITECTURE

Es un conjunto de principios de diseño de software que busca crear sistemas independientes, testables, mantenibles y flexibles a lo largo del tiempo. Su objetivo principal es proteger la lógica de negocio central de los detalles de implementación externos, como frameworks, bases de datos o interfaces de usuario.

Se basa en el principio de **separación de preocupaciones** y, fundamentalmente, en la **Regla de Dependencia**: las dependencias siempre deben apuntar hacia adentro, de las capas externas a las capas internas. Las capas internas no deben saber nada sobre las capas externas.

CAPAS INTERNAS Y EXTERNAS

Las capas internas (Entidades y Casos de Uso) contienen la lógica de negocio central y son independientes de todo

Las capas eternas (Adaptadores de interfaz, Frameworks y Drivers) manejan los detalles de la implementación y dependen de las capas enteras.

REST Y RESTFUL

REST es un estilo arquitectónico para WWW. El objetivo es lograr un sistema distribuido escalable, robusto y eficiente.

Se basa en la idea de que los recursos pueden ser manipulados utilizando un conjunto limitado de operaciones predefinidas(métodos HTTP estándar) sobre representaciones de esos recursos.

Las **restricciones arquitectónicas** clave de REST son:

1. **Client-Server (Cliente-Servidor):** Clara separación de responsabilidades entre la interfaz de usuario (cliente) y el almacenamiento de datos (servidor).
2. **Stateless (Sin Estado):** Cada solicitud del cliente al servidor debe contener toda la información necesaria para que el servidor la entienda y la procese. El servidor no debe almacenar ningún estado de sesión del cliente entre solicitudes.
3. **Cacheable (Cachable):** Las respuestas del servidor deben indicar si los datos son cachables o no, para que los clientes puedan almacenar en caché las respuestas y mejorar el rendimiento.
4. **Uniform Interface (Interfaz Uniforme):** La interacción entre cliente y servidor debe ser consistente y genérica, facilitando la escalabilidad y simplificando el sistema. Esto se logra mediante:
 - **Identificación de recursos:** Los recursos se identifican mediante URIs (Uniform Resource Identifiers).
 - **Manipulación de recursos a través de representaciones:** El cliente interactúa con el recurso a través de su representación (ej. JSON, XML).
 - **Mensajes autodocumentados:** Cada mensaje contiene suficiente información para describir cómo procesarlo.
 - **HATEOAS (Hypermedia as the Engine of Application State):** El servidor guía al cliente a través de la aplicación proporcionando enlaces dentro de las representaciones de los recursos.
5. **Layered System (Sistema en Capas):** Permite que un sistema esté compuesto por múltiples capas (proxies, gateways, balanceadores de carga) sin que el cliente se dé cuenta.
6. **Code on Demand (Código bajo demanda - opcional):** Permite al servidor extender la funcionalidad del cliente enviando código ejecutable (ej. JavaScript).

Una API es considerada **RESTful** si cumple con las restricciones arquitectónicas de REST. Es decir que, si REST es el estilo, RESTful es la implementación práctica de ese estilo.

Todas las APIs RESTful son, por definición, implementaciones de REST. Pero no todas las APIs que usan HTTP son realmente "RESTful" si no cumplen con todas las restricciones de REST (especialmente la de "interfaz uniforme" y "sin estado"). Una API "RESTful" es una API que sigue el estilo REST de manera apropiada.

Buenas Prácticas (Nombres, Verbos HTTP, Estado)

Para construir APIs RESTful de calidad, se siguen ciertas convenciones y buenas prácticas:

- **Nombres de Recursos (URIs):**
 - **Usar sustantivos (en plural) para representar colecciones:** Evitar verbos en las URIs. Los verbos son la función de los métodos HTTP.
 - /users
 - ✗ /getAllUsers
 - **Identificar recursos individuales con IDs:**
 - /users/123
 - ✗ /getUserById/123
 - **Reflejar relaciones entre recursos:**
 - /users/123/orders (órdenes del usuario 123)
 - /orders/456/user (usuario de la orden 456, si aplica)
 - **Nombres en plural y consistentes:** Si decides usar plurales, úsalos siempre.
 - /products, /orders
 - ✗ /product, /order (aunque algunos prefieren singular, la pluralidad es más común para colecciones)
- **Verbos HTTP (Métodos HTTP):** Utilizar los verbos HTTP de manera semántica para indicar la acción deseada sobre el recurso.
- **Códigos de Estado HTTP:** Utilizar los códigos de estado HTTP para indicar el resultado de la operación.

HTTP

HyperText Transfer Protocol, es un protocolo para la comunicación de datos en la WWW. Define como los clientes y los servidores se comunican para intercambiar información

VERBOS HTTP

Son métodos que indican la acción deseada. Comandos que un cliente envía a un servidor

GET:

- Sirve para recuperar una representación de un recurso o una colección de recursos

POST:

- Sirve para enviar datos al servidor para crear un nuevo recurso subordinado a la URI dada

PUT:

- Sirve para reemplazar completamente la representación de un recurso existente con la nueva representación proporcionada.

DELETE:

- Eliminar el recurso identificado por la URI

PATH:

- Aplica modificaciones parciales a un recurso

CÓDIGOS DE ESTADO

- 1xx : Informacional, la solicitud fue recibida y se continua el proceso
- 2xx: Éxito, la acción fue recibida, entendida y aceptada con éxito
 - 200: OK
 - 201: CREATED
 - 204: NO CONTENT
- 3xx: Redirección
 - 301 MOVED PERMANENTLY
 - 302 FOUND
- 4xx: Errores del cliente
 - 400 BAD REQUEST
 - 401 UNAUTHORIZED
 - 404 FORBIDDEN
 - 404 NOT FOUND
 - 409 CONFLICT
 - 422 UNPROCESSABLE ENTITY
- 5xx: Errores de Servidor
 - 500 INTERNAL SERVER ERROR
 - 502 BAD GATEWAY
 - 503 SERVICE UNAVAILABLE

HEADERS, BODY, MÉTODOS SEGUROS VS. IDEMPOTENTES

Los headers son campos adicionales en el inicio de la solicitud o respuesta HTTP que proporcionan metadatos sobre la comunicación

El body es la parte principal de una solicitud o respuesta HTTP que contiene los datos reales que se están transfiriendo

Los métodos seguros, son solo así si, al ser invocados, no causan ningún cambio de estado en el servidor.

Los métodos son idempotente, si al invocarlo múltiples veces con los mismos parámetros, el efecto final en el servidor es el mismo que si se hubiera invocado una sola vez

B. ASP.NET CORE

Es un Framework de código abierto y multiplataforma para construir aplicaciones modernas basadas en la nube, como aplicaciones web, APIs REST y microservicios.

Middleware: Es un componente de ASP.NET CORE que se encarga de procesar las solicitudes HTTP y las respuestas. Puede realizar una acción específica y luego pasar la solicitud al siguiente middleware en la cadena o terminar la solicitud.

Pipeline de ejecución: Cuando la solicitud HTTP llega a una aplicación, atraviesa una serie de middlewares configurados secuencialmente.

TIPOS DE API

REST API vs. gRPC:

- **REST API (Representational State Transfer API):**

- **Protocolo:** Basada en HTTP/1.1 (predominantemente).
- **Formato de Datos:** Generalmente JSON, a veces XML.
- **Comunicación:** Request/Response síncrono.
- **Filosofía:** Centrada en recursos (URIs) y métodos HTTP (GET, POST, PUT, DELETE).
- **Ventajas:** Ubicua, fácil de entender, compatible con navegadores, muchos clientes disponibles, legible por humanos.
- **Desventajas:** Más verbosa (JSON), menos eficiente para comunicaciones de alto rendimiento o streaming, no tiene tipado fuerte de los contratos de servicio por defecto (depende de documentación como OpenAPI/Swagger).
- **Casos de Uso:** APIs públicas, integración con navegadores, aplicaciones móviles, sistemas donde la legibilidad y la simplicidad son prioritarias.

- **gRPC (gRPC Remote Procedure Call):**

- **Protocolo:** Basado en HTTP/2.
- **Formato de Datos:** Protocol Buffers (protobuf) serializado binario, que es mucho más compacto y eficiente que JSON/XML.
- **Comunicación:** Soporta Request/Response, Server Streaming, Client Streaming y Bidirectional Streaming.
- **Filosofía:** Centrada en servicios y funciones/métodos (RPC - Remote Procedure Call).
- **Ventajas:** Mucho más eficiente y rápido (HTTP/2, protobuf), soporta streaming, tipado fuerte de los contratos de servicio (.proto files), generación de código para múltiples lenguajes (Polyglot).
- **Desventajas:** Menos amigable para el desarrollo web directo (no se puede llamar desde el navegador sin un proxy), curva de aprendizaje ligeramente mayor, herramientas y ecosistema menos maduros que REST en algunos aspectos.
- **Casos de Uso:** Microservicios inter-servicio de alto rendimiento, comunicación interna en sistemas distribuidos, IoT, servicios en tiempo real, entornos polyglot.

Minimal API vs. API con Controlador:

- **API con Controlador (Controller-based API):**

- **Estructura:** Utiliza clases que heredan de ControllerBase (o Controller para APIs con vistas) y organizan las acciones de la API en métodos dentro de esos controladores.
- **Convención:** Sigue un patrón MVC para APIs. Cada controlador maneja un conjunto de recursos relacionados.
- **Características:**
 - Soporte completo para atributos de enrutamiento, validación, filtros de acción, filtros de autorización, etc.
 - Mayor estructura y separación de responsabilidades.
 - Ideal para APIs grandes, complejas o cuando se necesita una estructura más organizada.
- **Configuración:** Se habilita con builder.Services.AddControllers(); y app.MapControllers();.

- **Minimal API:**

- **Estructura:** Permite definir endpoints HTTP directamente en el archivo Program.cs (o en archivos separados con WebApplication.Map* métodos) utilizando expresiones lambda. No requiere clases de controlador.
- **Filosofía:** Diseñada para ser simple y concisa, ideal para microservicios pequeños, APIs con pocos endpoints o prototipos rápidos.
- **Características:**
 - Menos boilerplate (código repetitivo).
 - Enrutamiento directo y manejo de solicitudes en línea.
 - Aún soporta inyección de dependencias, filtros y validación, pero de una manera más concisa.
- **Configuración:** Habilitada por defecto en proyectos .NET 6+ WebApplication.CreateBuilder(). Los endpoints se mapean con app.MapGet(), app.MapPost(), etc.

¿Cuándo usar cuál?

Minimal APIs: Pequeñas APIs, microservicios simples, prototipos, aprender ASP.NET Core de forma rápida.

API con Controlador: APIs complejas, grandes, cuando se requiere mucha organización y aplicación de patrones de diseño, equipos grandes, cuando se mezclan APIs con vistas (MVC).

Los Health Checks en ASP.NET Core permiten que un servicio o herramienta externa (ej., un orquestador de contenedores como Kubernetes, o un balanceador de carga) determine el **estado de salud de tu aplicación**.

Se implementan creando clases que implementan la interfaz `IHealthCheck`. Cada clase contiene la lógica para verificar el estado de un componente específico (ej., conectividad a la base de datos, disponibilidad de un servicio externo, espacio en disco).

El método `CheckHealthAsync` de la interfaz `IHealthCheck` debe devolver un `HealthCheckResult` que indica `Healthy`, `Degraded` o `Unhealthy`, junto con un mensaje opcional y datos de diagnóstico.

Configuración: Se agregan al servicio de Health Checks en `Program.cs`:

```
builder.Services.AddHealthChecks().AddCheck<MyDatabaseHealthCheck>("MyDatabase");
```

/health o /healthz:

Una vez configurados los Health Checks, se exponen como endpoints HTTP.

Se mapean en el pipeline de middleware: `app.MapHealthChecks("/healthz")` o `app.MapHealthChecks("/health")`.

Cuando una solicitud llega a `/healthz`, ASP.NET Core ejecuta todos los Health Checks registrados y devuelve una respuesta HTTP basada en su estado combinado.

Respuestas comunes:

- Si todos los checks son `Healthy`.
- `503 Service Unavailable`: Si al menos un check es `Unhealthy`.

Variantes de endpoints:

- `MapHealthChecks("/healthz")`: Endpoint simple que devuelve solo el código de estado (útil para balanceadores de carga).
- `MapHealthChecks("/health")`: Puede configurarse para devolver un JSON más detallado con el estado de cada componente, útil para monitoreo. También se puede usar `MapHealthChecks("/health/ready")` y `MapHealthChecks("/health/live")` para distinguir entre "listo para recibir tráfico" y "vivo pero quizás no listo".

ENTITY FRAMEWORK (EF CORE)

Es un ORM, un mapeador objeto-relacional que permite interactuar con bases de datos relacionales usando clases C# y LINQ, abstracto directamente el SQL. Rastrea cambios y genera SQL al guardar.

MIGRACIONES

Es una característica para evolucionar el esquema de la base de datos de forma versionada. `Add-Migration` genera archivos de cambios; `Update-Database` los aplica.

DBCONTEXT

Es la clase central en EFCORE, actuando como puerta de entrada principal a la base de datos. La aplicación tendrá una clase que hereda de `DbContext` y dentro de esta clase definirías propiedades `DbSet<T>`.

Una propiedad **DbSet< TEntity >** representa una colección de entidades de un tipo específico en tu base de datos, mapeándose a una tabla.

FLUENT API

Es una API basada en código que se utiliza en el método OnModelCreating del DbContext. Ofrece una forma potente y flexible de configurar el modelo.

Se usa en configuraciones complejas que los atributos no pueden manejar (ej., claves compuestas, índices únicos, relaciones many-to-many sin entidad de unión explícita).

También es preferible cuando quieras **separar completamente tu modelo de dominio (tus clases de entidad) de la lógica de persistencia**, lo cual es crucial en arquitecturas limpias donde las entidades no deberían depender de librerías de infraestructura como EF Core.

DATA ANNOTATIONS

Son atributos de C# que se aplican directamente a las propiedades de las clases de entidad. Son buenas para configuraciones sencillas y comunes, ya que son fáciles de leer y están junto a la propiedad que modifican.

SWAGGER / OPENAPI

Swagger (o más precisamente, la **especificación OpenAPI** y sus herramientas como Swagger UI) es un conjunto de herramientas esenciales para el desarrollo de APIs RESTful.

Documentación Automática: Herramientas como Swashbuckle.AspNetCore en .NET Core inspeccionan tu código (controladores, acciones, modelos) y **generan automáticamente un archivo JSON** que describe tu API según la especificación OpenAPI. Esta documentación detalla tus endpoints, los métodos HTTP que soportan, los parámetros de entrada, las estructuras de datos esperadas y los posibles códigos de respuesta HTTP. Lo mejor es que se mantiene sincronizada con tu código, reduciendo la carga de documentación manual.

Herramienta para Probar Endpoints: La **Swagger UI** es una interfaz de usuario web interactiva generada a partir de esa documentación. Se expone como una página en tu aplicación (comúnmente en /swagger o /swagger/index.html). Permite:

- Visualizar todos los endpoints de tu API.
- Ver sus detalles (parámetros, tipos de retorno, códigos de estado).
- Lo más importante: **probar los endpoints directamente desde el navegador**. Puedes introducir los datos de la solicitud y ver la respuesta en tiempo real. Esto es invaluable para desarrolladores (tanto de backend como frontend) y para equipos de QA, acelerando el desarrollo y la depuración.