

GENERAL

https://github.com/LuisMirandaMallqui/Programacion3_LuisMirandaPucp.git

CONCEPTOS:

Ligadura dinamica y estatica

mensaje = objeto + método

polimorfismo = mismo mensaje, diferente comportamiento

this. = referencia al objeto = puntero

Herencia: Sobreescritura

//ESTO ESTA MAL:

// this.autores = this.autores // VIOLA ENCAPSULAMIENTO, SE DEBE COPIAR LOS
DATOS EN OTRO ARREGLO

<https://www.youtube.com/watch?v=wySGHbBILvk>

🤔 ¿Qué es el patrón Builder?

El Builder es un patrón de diseño que te permite construir objetos complejos paso a paso. Imagina que estás construyendo una casa:

- Sin Builder: Tienes que dar todos los detalles de una vez
- Con Builder: Vas agregando características una por una

📝 Ejemplo SIN Builder:

```
csharp
// ¡Demasiados parámetros de una vez!
Cancion cancion = new Cancion(
    "Cuando estoy triste",           // título
    "Cajita de música",             // otro título
    new List<string> {"Mercedes Sosa"}, // intérpretes
    new List<string> {"Jose Pedroni", "Damián Sánchez"}, // compositores
    "A que florezca mi pueblo",      // álbum
    0,                               // opus (no aplica)
    0,                               // sub-opus (no aplica)
    null,                            // dedicatoria (no aplica)
    Genero.FOLKLORE                  // género
);
```

Problemas:





- ❌ Muchos parámetros
- ❌ Parámetros opcionales necesitan valores "dummy" (0, null)
- ❌ Código difícil de leer
- ❌ Fácil equivocarse en el orden

🎯 Ejemplo CON Builder:

```
csharp
// ¡Más claro y paso a paso!
Cancion cancion = new CancionBuilder()
    .ConTitulo("Cuando estoy triste")
    .TambienConocidaComo("Cajita de música")
    .InterpretadoPor("Mercedes Sosa")
```

```
.CompuestoPor("Jose Pedroni")
.CompuestoPor("Damián Sánchez")
.DelGenero(Genero.FOLKLORE)
.EnElAlbum("A que florezca mi pueblo")
.BuildCancion();
```

Ventajas:

-  Fluido y legible: Se lee como frases
-  Flexible: Solo usas lo que necesitas
-  Orden libre: No importa el orden de los métodos
-  Opcional: Omities lo que no necesitas

¿Cómo funciona?

1. El Builder - Es el "constructor inteligente"

```
csharp
public class CancionBuilder
{
    private Cancion _cancion = new Cancion();

    public CancionBuilder ConTitulo(string titulo)
    {
        _cancion.Titulo = titulo;
        return this; // ← ¡Importante! Devuelve el mismo builder
    }

    public Cancion BuildCancion()
    {
        return _cancion; // ← Finalmente devuelve la canción terminada
    }
}
```

2. El Truco Mágico: `return this`

Cada método devuelve el mismo builder, permitiendo encadenar llamadas:

```
csharp
// Esto funciona porque cada método devuelve el builder
```

```
builder.ConTitulo("...")
    .InterpretadoPor("...")
    .CompuestoPor("...");
```

🎵 Ejemplos reales de uso:

Ejemplo 1: Canción simple

```
csharp
Cancion simple = new CancionBuilder()
    .ConTitulo("Imagine")
    .InterpretadoPor("John Lennon")
    .BuildCancion();
```

Ejemplo 2: Canción clásica con detalles

```
csharp
Cancion clasica = new CancionBuilder()
    .ConTitulo("Sonata para piano No. 14")
    .TambienConocidaComo("Claro de Luna")
    .CompuestoPor("Ludwig van Beethoven")
    .DelGenero(Genero.CLASICA)
    .IdentificadoConOpus(27)
    .IdentificadoConSubOpus(2)
    .DedicadoA("Condesa Giulietta Guicciardi")
    .BuildCancion();
```

Ejemplo 3: Canción de rock

```
csharp
Cancion rock = new CancionBuilder()
    .ConTitulo("Bohemian Rhapsody")
    .InterpretadoPor("Queen")
    .CompuestoPor("Freddie Mercury")
    .DelGenero(Genero.ROCK)
    .EnElAlbum("A Night at the Opera")
    .BuildCancion();
```

🔧 ¿Cuándo usar el patrón Builder?

Úsalo cuando:

- El objeto tiene muchos parámetros
- Algunos parámetros son opcionales
- Quieres código más legible
- Necesitas diferentes variaciones del mismo objeto

No lo uses cuando:

- El objeto es muy simple (solo 2-3 parámetros)
- Todos los parámetros son siempre obligatorios



Comparación final:

Sin Builder

```
new Cancion(tit, null, interp,  
comp, null, 0, 0, null, gen)
```

Código confuso

Parámetros obligatorios

Orden fijo

Con Builder

```
.ConTitulo(tit).InterpretadoPor(int  
erp).CompuestoPor(comp).DelGenero(g  
en)
```

Código claro

Parámetros opcionales

Orden flexible

¡El Builder hace tu código más limpio, flexible y mantenible! 🚀

CLASE DAO

BACKEND

Del caso Zavala:

Roles del negocio

Servicio de Reglas del Negocio

Servicio del Negocio

JASPERSOFT

VISUAL STUDIO COM

Tiene la opción para encapsular campos: Te codea los setters y getters base !!

CLION:

<https://www.jetbrains.com/es-es/clion/>

1. Quién puede usar la licencia

Solo estudiantes (matriculados al menos 1 año en un centro educativo) e instructores (que enseñan en una institución educativa).

Debes poder demostrar tu condición de estudiante/instructor cuando JetBrains lo solicite.

Si dejas de ser estudiante/instructor, debes dejar de usar el software.

2. Tipo de uso permitido

Uso únicamente educativo y no comercial (incluye investigación académica o dar clases).

Puedes instalarlo en varias computadoras, pero solo tú puedes usarlo.

No se permite usarlo para proyectos con fines comerciales o lucrativos.

3. Restricciones principales

No compartir tu cuenta o suscripción.

No vender, alquilar, modificar, distribuir ni transferir el software.

No hacer ingeniería inversa ni intentar obtener su código fuente.

No quitar avisos de propiedad intelectual.

4. Duración y renovación

La licencia dura 1 año.

Puedes pedir renovación hasta 30 días antes de que expire.

5. Propiedad

No eres dueño del software, solo tienes un derecho limitado de uso.

JetBrains mantiene todos los derechos de propiedad intelectual.

6. Acceso y activación

Requiere conexión a Internet para verificar la licencia periódicamente.

Debes usar tu JetBrains Account para activarlo.

7. Terminación

JetBrains puede cancelar tu licencia si incumples el acuerdo, das datos falsos o usas el software de forma no autorizada.

Si se termina, debes dejar de usar el producto inmediatamente.

8. Garantías y responsabilidad


El software se proporciona “tal cual” sin garantías de que esté libre de errores o interrupciones.

JetBrains no se hace responsable por daños derivados del uso, y su responsabilidad máxima es de 10 USD.

9. Ley aplicable

El acuerdo se rige por las leyes de la República Checa.

Teoría:


 Teoria.docx

 JAVA_LIBROS

Clases de Paz:

https://youtube.com/playlist?list=PLnYlANzM__0oLI_96iE7RshZYUg70EXxv&si=-wUpclEr0KObTzYM

Clases Sanchez(Una parte):

 P3_2025_1_clasesSanchez

Extra:

 JavaScript desde Cero 2020

CICLO 25-2

COMO PROGRAMAR EN C# EN LINUX?

1 Mono

- **Qué es:**
Implementación libre y multiplataforma de **.NET Framework** (original de Windows).
- **Versiones:**
 - Compatible con .NET Framework 4.5 → 4.7 (dependiendo de la versión de Mono).
 - Consola, librerías estándar, lógica de negocio funcionan perfectamente.
- **Instalación en Linux:**

```
sudo apt install mono-complete
```

- **Ventajas:**
 - Permite ejecutar código .NET Framework en Linux.
 - Compatible con VS Code + OmniSharp para IntelliSense y depuración.
 - Ligero, fácil de instalar y usar.

- **Limitaciones:**

- WPF, WinForms y otras interfaces gráficas **no funcionan bien o no están soportadas** en Linux.
- Algunas APIs avanzadas pueden no estar implementadas al 100%.
- Depuración integrada requiere configurar **Mono Debug** en VS Code.

- **Compilación y ejecución:**

```
mcs Program.cs -out:Program.exe    # Compilar
mono Program.exe                   # Ejecutar
```

2) .NET Core / .NET 6+ (oficial multiplataforma)

- **Qué es:**

Implementación moderna de .NET, multiplataforma, oficial de Microsoft.
Incluye .NET Core (6/7/8) y reemplaza .NET Framework en proyectos nuevos.

- **Instalación en Linux Mint:**

```
sudo apt install dotnet-sdk-8.0
dotnet --version
```

- **Ventajas:**

- 100% multiplataforma (Linux, Windows, macOS).
- Compatible con proyectos modernos y herramientas como `dotnet run`, `dotnet build`.
- Excelente integración con VS Code y Visual Studio Code Dev Kit.
- Permite depuración avanzada y extensiones modernas.

- **Limitaciones:**

- No es compatible con **.NET Framework antiguo (4.5)** de forma nativa.
- Proyectos legacy que dependen de APIs de Windows (WinForms, WPF) requieren Windows o Wine.

- **Compilación y ejecución:**

```
dotnet new console -o MiApp
cd MiApp
dotnet run
```

3 Comparativa rápida

Característica	Mono	.NET Core / 6+
Plataforma	Linux, Windows, macOS	Linux, Windows, macOS
Compatible .NET 4.5	Sí (con limitaciones gráficas)	No
Consola y lógica	Sí	Sí
GUI (WinForms/WPF)	Parcial / limitado	No (solo en Windows)
Extensiones VS Code	OmniSharp + Mono Debug	C# Dev Kit + .NET Debug
Instalación en Linux	<pre>sudo apt install mono-complete</pre>	<pre>sudo apt install dotnet-sdk-8.0</pre>

✓ Conclusión práctica para Linux Mint

- **Proyectos antiguos .NET 4.5 → Mono + OmniSharp**
- **Proyectos nuevos multiplataforma → .NET Core / 6+**

Mono te permite **trabajar con C# y la mayoría de APIs de .NET Framework en Linux**, ideal para tu caso donde quieres .NET 4.5 sin Windows.

```
{
"omnisharp.useModernNet": false,
"omnisharp.monoPath": "/usr/lib/mono",
"omnisharp.useGlobalMono": "always"
}
```

CONSEJO: Instalar Mono pero para el MSBuild usar el .NET, no el del mono: ahorrar dolores de cabeza

C#

✓ 1. ¿Los atributos empiezan por mayúscula?

En **C#** la **convención** (no una regla obligatoria del compilador) es:

- **Propiedades públicas** (con `get; set;`) → **Mayúscula inicial** (PascalCase).
Ejemplo:

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

Campos privados → **minúscula inicial con guion bajo** (camelCase o `_camelCase`).
Ejemplo:

```
private string _nombre;  
private int edad;
```

-

Entonces, en el ejemplo que te di, **Nombre**, **DNI**, **Carrera** son **propiedades públicas**, por eso están en mayúscula.

✓ 2. ¿Los métodos también empiezan en mayúscula?

Sí, la convención para métodos (funciones dentro de clases) en C# es **PascalCase**, igual que las propiedades:
Ejemplo:

```
public void ImprimirReporte() { ... }  
private string GenerarTexto() { ... }
```

✓ 3. ¿Cuál es la diferencia entre atributo y método?

- **Atributo (o propiedad):**
Es un **dato** asociado a un objeto (estado).
Ejemplo:

```
public string Nombre { get; set; } // Atributo
```

Si tienes un **Alumno**, su **Nombre** y **Carrera** son atributos que describen **qué es** el alumno.

Método:

Es **una acción o comportamiento** del objeto (qué hace).

Ejemplo:

```
public void Estudiar() { ... } // Método
```

- Si el alumno tiene un método **Estudiar()**, es algo **que el alumno puede hacer**.

Analogía sencilla:

- Atributo = **Sustantivo** (característica): “nombre”, “edad”.
- Método = **Verbo** (acción): “correr”, “imprimir”, “calcular”.

✓ Ejemplo combinado:

```
public class Alumno {  
    public string Nombre { get; set; } // Atributo  
    public int Edad { get; set; }      // Atributo  
  
    public void Estudiar() {           // Método  
        Console.WriteLine($"{Nombre} está estudiando.");  
    }  
}
```

¿Quieres que te haga **un esquema visual** que muestre **cómo se ven atributos vs métodos en una clase**, tipo diagrama UML simple?

COMO ADJUNTAR UN ARCHIVO DE LECTURA PARA LEERLO FACIL EN C#?

La Solución Definitiva: El Método Profesional

Vamos a dejar de pelear con rutas absolutas y a hacerlo de la forma estándar y robusta, que siempre funciona. Vamos a incluir el archivo en tu proyecto.

Sigue estos 3 pasos:

1. Agrega el Archivo a tu Proyecto de Pruebas

- En el **Explorador de Soluciones** de Visual Studio, haz clic derecho sobre tu proyecto de prueba (**SoftInvTest** o similar).
- Selecciona **Agregar > Elemento existente...**
- Busca y selecciona tu archivo **password.txt**. Ahora debería aparecer en tu lista de archivos del proyecto.

2. Configura el Archivo para que se Copie al Compilar

- Haz clic **una vez** sobre el archivo **password.txt** en el Explorador de Soluciones para seleccionarlo.
- Ve a la ventana de **Propiedades** (si no la ves, presiona **F4**).
- Busca la propiedad **"Copiar en el directorio de salida"** (Copy to Output Directory).
- Cámbiala de "No copiar" a **"Copiar si es posterior"** (Copy if newer) o **"Copiar siempre"** (Copy always).

3. Modifica tu Código para Usar una Ruta Relativa

Ahora que el archivo **password.txt** se copiará automáticamente al lado de tu ejecutable de prueba, ya no necesitas la ruta completa.

Cambia tu código para que sea así de simple:

```
C#
[TestMethod]
public void TestProbarLectura()
{
    try
    {
        // Simplemente el nombre del archivo.
        // El programa lo buscará en la misma carpeta donde se está ejecutando.
        string contra = File.ReadAllText("password.txt");

        Console.WriteLine("¡Éxito! Contraseña leída:");
        Console.WriteLine(contra);
    }
    catch (Exception ex)
    {
        // ... tu manejo de error
    }
}
```

JAVA

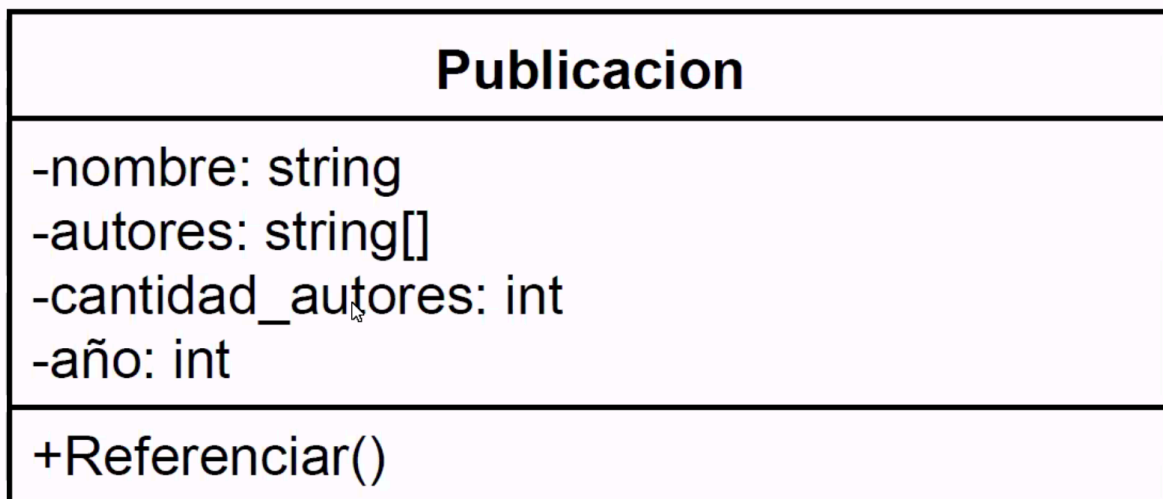
En este curso por lo que tengo entendido se trabaja con este lenguaje más orientado al backend

Servlets

AWS

UML

Diagrama de clases UML - ejemplo v1



Estándar para impresion

cuando hay triangulito entre líneas es herencia, sino solo es asociación

GIT BASH

Para usar Git Bash y trabajar con GitHub en tu PC, sigue estos pasos:

1. Abre Git Bash: Ya lo tienes abierto en la imagen, lo que indica que el terminal está listo para usarse.

2. Configura tu usuario de Git (si no lo has hecho antes):

- Ejecuta los siguientes comandos para configurar tu nombre y correo electrónico en Git (esto es importante para el historial de tus commits):

- ```
git config --global user.name "Tu Nombre" git config --global user.email "tu.email@example.com"
```

3. Clona tu repositorio de GitHub:

- Si ya tienes un repositorio en GitHub y quieres descargarlo en tu PC, usa el comando `git clone` seguido de la URL del repositorio:

- ```
git clone https://github.com/tu-usuario/nombre-del-repositorio.git
```

- Esto creará una copia local de tu repositorio en tu máquina.

4. Accede a la carpeta del repositorio:

- Una vez clonado el repositorio, navega a la carpeta del repositorio usando `cd`:

- ```
cd nombre-del-repositorio
```

5. Modifica los archivos:

- Realiza las modificaciones que necesites en los archivos dentro del directorio del repositorio usando cualquier editor de texto o IDE.

6. Agrega los cambios:

- Después de modificar los archivos, agrega los cambios al área de preparación (staging area) con el siguiente comando:

- ```
git add .
```

- El `.` significa que agregarás todos los cambios. Si solo quieres agregar archivos específicos, usa:

- ```
git add nombre-del-archivo
```

7. Haz un commit:

- Después de agregar los cambios, crea un commit con un mensaje que describa los cambios realizados:

- ```
git commit -m "Mensaje describiendo los cambios"
```

8. Haz un push para subir los cambios a GitHub:

Reglas rápidas de nombres en C#

- Clases, enums y propiedades públicas: `PascalCase` → `FichaEvaluacion`, `EstadoCandidato`, `FechaHora`.
- Campos privados: `camelCase` con prefijo `_` → `_fechaHora`, `_evaluacionExpediente`.
- Sin guiones bajos internos (solo el prefijo de campos) y mejor sin tildes/acentos en identificadores.
- Tipos adecuados: usa `DateTime` para fecha/hora (no `string`).
- Si un puntaje puede no existir, usa nullable `int?`; si siempre existe, usa `int`.

Cómo dejarlos bien

```
public enum EstadoCandidato { Admitido, NoAdmitido, SinEvaluacion }

public class FichaEvaluacion { // Campos privados private DateTime
    _fechaHora; private int? _evaluacionExpediente; // null = sin evaluar
    private int? _evaluacionEntrevista; // ojo: "Entrevista", no
    "Entrevita" private int? _evaluacionExamen; private EstadoCandidato
    _estadoCandidato; // Propiedades públicas public Postulante Candidato {
    get; set; } public DateTime FechaHora { get => _fechaHora; set =>
    _fechaHora = value; } public int? EvaluacionExpediente { get =>
    _evaluacionExpediente; set { if (value is null || (value >= 0 && value
    <= 25)) _evaluacionExpediente = value; else throw new
    ArgumentOutOfRangeException(nameof(EvaluacionExpediente), "0..25 o
    null."); } } public int? EvaluacionEntrevista { get =>
    _evaluacionEntrevista; set { if (value is null || (value >= 0 && value
    <= 50)) _evaluacionEntrevista = value; else throw new
    ArgumentOutOfRangeException(nameof(EvaluacionEntrevista), "0..50 o
    null."); } } public int? EvaluacionExamen { get => _evaluacionExamen;
    set { if (value is null || (value >= 0 && value <= 25))
    _evaluacionExamen = value; else throw new
    ArgumentOutOfRangeException(nameof(EvaluacionExamen), "0..25 o null.");
    } } public EstadoCandidato EstadoCandidato { get => _estadoCandidato;
    set => _estadoCandidato = value; } }
```

Puntos clave:

- Corrige el typo: `evaluacion_entrevita` → `EvaluacionEntrevista`.

- Evita `string fecha_hora;` usa `DateTime FechaHora`.
- Expón propiedades en PascalCase y deja los campos privados con `_`.

Con esto cumples las convenciones y te evitas problemas de estilo y mantenimiento.

Dijiste:



aca esta bien el enum? o deberia ir afuera del namespace'

ChatGPT dijo:

El `enum` está dentro del `namespace`, lo cual está bien. De hecho, no hay ningún problema en definir un `enum` dentro de un `namespace` o incluso dentro de una clase. Sin embargo, normalmente los `enum` se colocan dentro del `namespace`, pero fuera de las clases, para que tengan un ámbito global dentro del espacio de nombres, lo cual los hace accesibles desde cualquier clase dentro de ese `namespace`.

Forma recomendada (dentro del `namespace`, fuera de la clase):

```
namespace LAB2_2025_1_Miranda { // Enum dentro del namespace public
enum EstadoCandidato { Admitido, NoAdmitido, SinEvaluacion } // Clase
dentro del mismo namespace public class FichaEvaluacion { // Aquí va el
código de la clase } }
```

Razón:

1. Visibilidad y Organización: Definir el `enum` fuera de la clase pero dentro del `namespace` hace que el `enum` sea accesible en todo el espacio de nombres, mientras que si lo pones dentro de una clase, solo será accesible a través de esa clase (lo que no siempre es conveniente para este tipo de valores globales).
2. Convención: Es más común ver `enums` como entidades separadas en el `namespace` y no dentro de clases, para que puedan ser fácilmente reutilizables.

Conclusión:

El código que tienes está bien si deseas que el `enum` esté accesible solo dentro de la clase. Pero te recomendaría moverlo fuera de la clase y dentro del `namespace` para mayor claridad y accesibilidad en el proyecto, especialmente si planeas usar este `enum` en múltiples lugares.

Si prefieres mantenerlo dentro de la clase por alguna razón, también está bien, solo que limitarás su acceso fuera de esa clase.

¿Te gustaría ajustar algo más o necesitas más detalles?

Dijiste:

diferencia entre atributos y propiedades

ChatGPT dijo:

En C#, la diferencia entre atributos (también llamados campos) y propiedades tiene que ver con su propósito, visibilidad, y cómo se accede a ellos. A continuación te explico las principales diferencias:

1. Atributos (o Campos)

- **Definición:** Son variables declaradas dentro de una clase o estructura que almacenan datos. Los atributos suelen ser privados o protegidos para seguir el principio de encapsulamiento.
- **Acceso:** Generalmente, los atributos son privados (aunque pueden ser públicos o protegidos) y no tienen control sobre su acceso o modificación, ya que son simples variables.
- **Convención:** Se suele usar camelCase para los nombres de los atributos privados (en C#, la convención es usar el guion bajo al principio del nombre, por ejemplo: `_nombre`), pero esto no es obligatorio.
- **Uso:** Los atributos se acceden directamente dentro de la clase, pero no se deben exponer directamente desde fuera de la clase (por cuestiones de encapsulamiento).

Ejemplo de atributo:

```
public class Persona { private string _nombre; // Atributo privado }
```

2. Propiedades

- Definición: Son métodos que permiten el acceso controlado a los atributos de una clase. En lugar de acceder directamente a los campos, se utilizan getters y setters para obtener y modificar el valor del atributo.
- Acceso: Las propiedades se definen con métodos especiales llamados `get` y `set`, que permiten controlar cómo se obtiene o establece el valor. Pueden ser públicas para permitir su acceso desde fuera de la clase, pero el atributo subyacente generalmente sigue siendo privado.
- Convención: Se usa PascalCase para los nombres de las propiedades.
- Uso: Las propiedades permiten validación, lógica adicional y protección de los datos internos de la clase. Usar propiedades en lugar de atributos directamente es una buena práctica porque permite un mayor control.

Ejemplo de propiedad:

```
public class Persona { private string _nombre; // Atributo privado
public string Nombre // Propiedad pública { get { return _nombre; } set
{ _nombre = value; } // Se puede agregar lógica en el setter si es
necesario } }
```

Diferencias clave:

Característica	Atributos (Campos)	Propiedades
Visibilidad	Normalmente privados, aunque pueden ser públicos	Normalmente públicas, pero pueden ser privadas o protegidas.
Acceso directo	Acceso directo a los datos (no hay control)	Acceso controlado mediante <code>get</code> y <code>set</code> .
Convención de nombre	<code>camelCase</code> (usualmente con guion bajo)	<code>PascalCase</code>
Propósito	Almacenar datos dentro de la clase	Controlar el acceso a los datos (leer/escribir) y agregar lógica adicional
Capacidad de validación	No suele haber validación de datos	Se puede agregar lógica para validación o transformación de datos

Ejemplo comparativo:

Uso de atributo (directo):

```
public class Persona { // Atributo privado private int edad; public
void EstablecerEdad(int edad) { this.edad = edad; // Directamente se
establece el valor } public int ObtenerEdad() { return edad; // Se
obtiene directamente el valor } }
```

Uso de propiedad (con control):

```
public class Persona { // Atributo privado private int _edad; //
Propiedad pública con control public int Edad { get { return _edad; }
set { if (value < 0) throw new ArgumentException("La edad no puede ser
negativa"); _edad = value; // Se valida antes de establecer el valor }
} }
```

Resumen:

- Atributos son los datos de la clase, usualmente privados.
- Propiedades permiten un acceso controlado a esos datos, a menudo con lógica adicional como validaciones o transformaciones, y se siguen las convenciones de PascalCase.

Siempre es recomendable usar propiedades en lugar de acceder directamente a los atributos de la clase, ya que te da flexibilidad para agregar lógica adicional sin cambiar la interfaz pública de tu clase.

¿Te gustaría más ejemplos o tienes alguna duda adicional sobre esto?

Dijiste:

y entre propiedades y metodos? cual es la diferencia?

ChatGPT dijo:

La diferencia entre propiedades y métodos en C# tiene que ver con el propósito y el modo de uso. Aunque ambos permiten manipular y acceder a los datos de una clase, se utilizan en contextos diferentes y tienen características propias. Aquí te explico la diferencia:

2. Propiedades

- Definición: Una propiedad en C# es una combinación de un método getter y setter que proporciona un acceso controlado a un atributo de una clase. Las propiedades se usan generalmente para exponer valores de un objeto, pero con la posibilidad de agregar lógica adicional.
- Propósito: Las propiedades se usan principalmente para leer y escribir valores de atributos de la clase, pero con la posibilidad de aplicar validaciones o transformaciones en los datos.
- Sintaxis: Se definen con `get` y `set`, y se accede a ellas de manera similar a un campo, pero su acceso está controlado por la lógica que pongas dentro de los métodos `get` y `set`.

Ejemplo de propiedad:

```
public class Persona { private int _edad; // Propiedad para acceder al
atributo _edad public int Edad { get { return _edad; } // Getter:
Accede al valor de _edad set { if (value < 0) // Setter: Validación
antes de asignar el valor throw new ArgumentException("Edad no puede
ser negativa"); _edad = value; } } }
```

Características de las propiedades:

- Se acceden como si fueran campos (`persona.Edad = 30;`).
- Son una interfaz controlada para los campos de la clase.
- Se usan para exponer datos de la clase, pero controlando cómo se obtienen y asignan.

3. Métodos

- Definición: Un método es una función dentro de una clase que puede realizar acciones, cálculos, o manipulación de datos. Los métodos pueden devolver valores o realizar operaciones sin necesidad de devolver nada (`void`).
- Propósito: Los métodos se usan para realizar operaciones o ejecutar lógica dentro de una clase, no solo para acceder a atributos. Pueden recibir parámetros y devolver resultados.
- Sintaxis: Un método tiene un tipo de retorno (puede ser `void` si no devuelve nada) y puede tomar uno o más parámetros. Los métodos se invocan mediante su nombre y pueden hacer cualquier operación que necesite la clase.

Ejemplo de método:

```
public class Persona { private int _edad; // Método para cambiar la
edad con validación public void CambiarEdad(int nuevaEdad) { if
(nuevaEdad < 0) throw new ArgumentException("Edad no puede ser
negativa"); _edad = nuevaEdad; } // Método para obtener el mensaje de
bienvenida public string ObtenerBienvenida() { return $"Bienvenido, la
edad es {_edad} años."; } }
```

Características de los métodos:

- Se usan para ejecutar lógica o acciones en la clase.
- Pueden recibir parámetros y devolver valores.
- No están relacionados directamente con los atributos de la clase, pero sí pueden usarlos.

Diferencias clave entre Propiedades y Métodos:

Característica	Propiedades	Métodos
Propósito	Exponer y controlar el acceso a los datos de la clase	Realizar acciones, cálculos o manipulación de datos
Sintaxis	Se acceden como si fueran campos (objeto.Propiedad)	Se invocan con paréntesis (objeto.Metodo())
Acción	Usada para leer/escribir valores con control	Usada para realizar operaciones o lógica
Acceso	get y set controlan el acceso	Los métodos son invocados para ejecutar una acción
Retorno	Puede devolver un valor, pero está más enfocado a la lectura/escritura	Puede devolver cualquier tipo de valor o ser void
Ejemplo	public int Edad { get; set; }	public void CambiarEdad(int nuevaEdad) { }

Cuándo usar propiedades vs métodos:

- Usar propiedades cuando deseas exponer valores de forma controlada o hacer que el acceso a los datos de tu clase sea más limpio y directo, pero con la posibilidad de agregar validaciones, cálculos o lógica adicional al establecer o recuperar el valor.
- Usar métodos cuando necesitas ejecutar lógica más compleja, manipular datos de una manera específica, o realizar acciones sin necesariamente involucrar un valor específico que debe ser leído o escrito.

Ejemplo comparativo:

Usando una propiedad:

```
public class Persona { private int _edad; // Propiedad para obtener y
establecer la edad public int Edad { get { return _edad; } set { _edad
= value; } } } var persona = new Persona(); persona.Edad = 25; // Uso
de la propiedad Console.WriteLine(persona.Edad); // Uso de la propiedad
```

Usando un método:

```
public class Persona { private int _edad; // Método para cambiar la
edad public void CambiarEdad(int nuevaEdad) { _edad = nuevaEdad; } //
Método para obtener el mensaje de bienvenida public string
ObtenerBienvenida() { return $"La edad es {_edad} años."; } } var
persona = new Persona(); persona.CambiarEdad(25); // Uso del método
Console.WriteLine(persona.ObtenerBienvenida()); // Uso del método
```

Resumen:

- Propiedades: Acceso controlado a los datos (getter/setter).
- Métodos: Realizan acciones o cálculos.

Ambas son herramientas poderosas, pero se usan en situaciones diferentes según la acción que desees realizar y el tipo de acceso o control que necesites para los datos.

¿Te gustaría algún ejemplo más detallado o tienes alguna otra pregunta?

Cómo llamo a una propiedad para SET y GET

En C#, para llamar a una propiedad y acceder a su valor (mediante el getter) o establecer su valor (mediante el setter), lo haces de manera similar a un campo (sin necesidad de paréntesis), pero con la ventaja de que estás invocando la lógica definida en los métodos `get` y `set` de la propiedad.

Ejemplo de cómo llamar a una propiedad:

Supongamos que tienes la clase `Persona` con una propiedad llamada `Edad`:

```
public class Persona { private int _edad; // Propiedad con getter y setter
public int Edad { get { return _edad; } set { _edad = value; } } }
```

Acceso a la propiedad:

Para acceder a la propiedad y usar su getter (para obtener el valor), o setter (para establecer el valor), simplemente haces lo siguiente:

Establecer un valor con el setter:

```
// Crear una instancia de la clase Persona
Persona persona = new Persona(); // Usar el setter para asignar el valor
persona.Edad = 25;
// Aquí se está usando el setter para establecer el valor
```

Obtener el valor con el getter:

```
// Usar el getter para obtener el valor
int edad = persona.Edad; // Aquí se está usando el getter para acceder al valor
Console.WriteLine(edad); // Imprime: 25
```

Explicación:

- Setter (`persona.Edad = 25`): Cuando asignas un valor a la propiedad `Edad`, internamente se llama al setter. Este setter valida o realiza alguna operación antes de establecer el valor en el atributo privado `_edad`.

- **Getter** (`int edad = persona.Edad`): Cuando accedes a la propiedad `Edad`, internamente se llama al getter. Este getter devuelve el valor actual del atributo privado `_edad`.

Acceso a propiedades con lógica adicional:

A veces, los getters y setters tienen lógica adicional, como validaciones o transformaciones antes de obtener o establecer el valor. A continuación te muestro un ejemplo donde se realiza una validación en el setter.

```
public class Persona { private int _edad; // Propiedad con validación
en el setter public int Edad { get { return _edad; } set { //
Validación: la edad no puede ser negativa if (value < 0) { throw new
ArgumentException("La edad no puede ser negativa."); } _edad = value; }
} }
```

Ejemplo de uso con validación:

```
try { Persona persona = new Persona(); persona.Edad = -5; // Esto
lanzará una excepción porque la edad no puede ser negativa } catch
(ArgumentException ex) { Console.WriteLine(ex.Message); // Imprime: "La
edad no puede ser negativa." }
```

Resumen:

- Para llamar al getter: simplemente usas el nombre de la propiedad sin paréntesis, como si fuera un campo.
- Para llamar al setter: también usas el nombre de la propiedad sin paréntesis, asignando un valor.

Las propiedades en C# te permiten encapsular el acceso a los datos, lo que te da la flexibilidad de agregar lógica dentro de las propiedades para realizar validaciones o transformaciones antes de obtener o establecer el valor de un campo privado.

¿Te gustaría que te ayudara con algún otro ejemplo o aclaración?

9.

- Finalmente, sube tus cambios al repositorio remoto en GitHub con el siguiente comando:

- `git push origin main`
- Si tu rama principal tiene un nombre diferente, reemplaza `main` por el nombre de tu rama.

Con estos pasos, podrás clonar, modificar, agregar, hacer commit y push a tu repositorio de GitHub usando Git Bash. ¿Te gustaría que te ayude con algo más en este proceso?

DUDAS

Siempre que tenga un atributo, ya sea privado