Gestor de Alumnos

Trabajo Práctico Final

Alumno: De Palma, Giuliana
Fecha: 2025

Introducción.

El presente manual de usuario describe el funcionamiento de la aplicación GestorAlumnos, un sistema de escritorio desarrollado en Windows Forms con .NET 8, cuyo objetivo principal es facilitar la gestión de información de alumnos mediante el uso de archivos.

La aplicación permite crear, leer, modificar y eliminar archivos de alumnos en distintos formatos (TXT, CSV, JSON y XML), así como también convertir archivos entre dichos formatos y generar reportes organizados a partir de los datos almacenados. Todas las operaciones se realizan a través de una interfaz gráfica sencilla e intuitiva, utilizando un menú principal para acceder a cada funcionalidad.

Este manual está destinado a usuarios que deseen operar el sistema sin necesidad de conocimientos técnicos avanzados, brindando instrucciones paso a paso, ejemplos de uso y la descripción de posibles errores comunes junto con sus soluciones. Su finalidad es servir como guía práctica para el correcto uso de la aplicación.

Descripción general del sistema

El sistema **GestorAlumnos** es una aplicación de escritorio diseñada para la administración básica de datos de alumnos mediante el manejo de archivos. Su finalidad es permitir al usuario almacenar, consultar y mantener actualizada la información de alumnos de forma estructurada y persistente.

La aplicación trabaja con un modelo de datos sencillo, donde cada alumno posee los siguientes campos: **legajo, apellido, nombres, número de documento, correo electrónico y teléfono**. Estos datos pueden ser guardados en archivos con distintos formatos, lo que brinda flexibilidad para el intercambio de información y la compatibilidad con otros sistemas.

El sistema ofrece las siguientes funcionalidades principales:

- Creación de archivos de alumnos en formatos TXT, CSV, JSON y XML.
- Lectura y visualización de archivos existentes en formato tabular.
- Modificación de archivos, permitiendo agregar, editar o eliminar alumnos.
- Eliminación de archivos del sistema con confirmación previa.
- Conversión de archivos entre diferentes formatos soportados.
- Generación de reportes agrupados por apellido, con opción de guardado en archivo de texto.

La interacción con el usuario se realiza a través de una interfaz gráfica basada en menús, lo que permite acceder a cada funcionalidad de manera clara y ordenada. La salida de información se muestra en un área de texto que simula una consola, facilitando la lectura de resultados y mensajes del sistema.

El sistema fue desarrollado siguiendo una estructura modular, separando responsabilidades en distintas clases, lo que mejora la organización del código, su mantenimiento y su posible ampliación futura.

Requisitos del sistema

Para poder ejecutar correctamente el sistema GestorAlumnos, es necesario contar con los siguientes requisitos de hardware y software:

Requisitos de software

- Sistema operativo:
  Windows 10 o superior.

- Framework:
  .NET 8 SDK instalado.

- Entorno de desarrollo (para compilación):
  Visual Studio 2022 o superior, con el paquete de desarrollo para aplicaciones de escritorio (.NET).

- Componentes adicionales:
  Microsoft Visual Basic Runtime (utilizado para los cuadros de ingreso de datos mediante InputBox).

Requisitos de hardware

- Procesador compatible con arquitectura x64.

- Mínimo 4 GB de memoria RAM.

- Espacio disponible en disco suficiente para la ejecución del programa y el almacenamiento de archivos generados (TXT, CSV, JSON, XML y reportes).

Requisitos para el usuario

- Conocimientos básicos de uso de aplicaciones de escritorio en Windows.

- Permisos de lectura y escritura en la carpeta donde se ejecuta la aplicación, para poder crear, modificar y eliminar archivos.

Funcionalidades

4.1- Crear Archivo.

```
public void CrearArchivo()
{
    imprimir("=== CREAR ARCHIVO ===");

    // Solicita el nombre base del archivo
    string nombre = Input("Nombre del archivo (sin extensión):");
    if (string.IsNullOrWhiteSpace(nombre))
    {
        imprimir("Nombre inválido.");
        return;
    }
```

`string nombre = Input("Nombre del archivo (sin extensión):");`

Esta línea solicita al usuario que ingrese el nombre del archivo que se va a crear.
El nombre se ingresa sin la extensión, ya que esta se definirá más adelante según el formato elegido (TXT, CSV, JSON o XML).
El valor ingresado se almacena en la variable `nombre`.

```
if (string.IsNullOrWhiteSpace(nombre))
{
   imprimir("Nombre inválido.");
   return;
}
```

Este bloque valida que el nombre ingresado sea correcto:
string.IsNullOrWhiteSpace(nombre) verifica si el texto está vacío o contiene únicamente espacios.
Si el nombre no es válido, se muestra un mensaje de error al usuario.
La instrucción return finaliza el método, evitando que el sistema continúe con un nombre de archivo incorrecto.
Esta validación asegura que no se intente crear un archivo con un nombre inválido, lo que podría provocar errores en el sistema de archivos.

```
// Solicita el formato del archivo
string formato = Input("Formato (TXT, CSV, JSON, XML):").ToUpper().Trim();
if (formato != "TXT" && formato != "CSV" && formato != "JSON" && formato != "XML")
{
    imprimir("Formato inválido.");
    return;
}

// Solicita la cantidad de alumnos a ingresar
int cantidad;
if (!int.TryParse(Input("Cantidad de alumnos:"), out cantidad) || cantidad <= 0)
{
    imprimir("Cantidad inválida.");
    return;
}
```

`string formato = Input("Formato (TXT, CSV, JSON, XML):").ToUpper().Trim();`

Esta línea solicita al usuario que ingrese el formato en el cual se guardará el archivo.
El valor ingresado se procesa de la siguiente manera:

- `ToUpper()` convierte el texto a mayúsculas para evitar errores por diferencias entre mayúsculas y minúsculas.
- `Trim()` elimina espacios en blanco al inicio y al final del texto.

De esta forma, se estandariza la entrada del usuario antes de validarla.

```
if (formato != "TXT" && formato != "CSV" && formato != "JSON" && formato != "XML")
{
   imprimir("Formato inválido.");
   return;
}
```

Este bloque valida que el formato ingresado sea uno de los formatos permitidos por el sistema:

- TXT
- CSV
- JSON
- XML

Si el formato ingresado no coincide con ninguno de los valores válidos, se muestra un mensaje de error y se interrumpe la ejecución del método mediante `return`.
Esto evita la creación de archivos con formatos no soportados.

```
int cantidad;
```

Se declara una variable entera llamada `cantidad`, que almacenará la cantidad de alumnos que el usuario desea cargar en el archivo.

```
if (!int.TryParse(Input("Cantidad de alumnos:"), out cantidad) || cantidad <= 0)
{
    imprimir("Cantidad inválida.");
    return;
}
```

Este bloque realiza la validación de la cantidad de alumnos ingresada:

- `int.TryParse(...)` intenta convertir el valor ingresado por el usuario a un número entero.
- Si la conversión falla o si el número ingresado es menor o igual a cero, la validación no se cumple.
- En caso de error, se muestra un mensaje indicando que la cantidad es inválida y se detiene la ejecución del método.

Esta validación garantiza que el sistema solo procese una cantidad válida de alumnos, evitando errores en los bucles de carga posteriores.

```
    // Lista donde se almacenan los alumnos ingresados
    List<Alumno> alumnos = new();

    // Carga de datos de cada alumno
    for (int i = 0; i < cantidad; i++)
    {
        imprimir($"--- Alumno {i + 1} ---");

        Alumno a = new Alumno();
        a.Legajo = Input("Legajo:");
        a.Apellido = Input("Apellido:");
        a.Nombres = Input("Nombres:");
        a.NumeroDocumento = Input("Documento:");

        // Validación del email
        string email;
        do
        {
            email = Input("Email:");
            if (!EmailValido(email))
                imprimir("Email inválido. Debe contener @ y dominio.");
        }
        while (!EmailValido(email));

        a.Email = email;
        a.Telefono = Input("Teléfono:");

        alumnos.Add(a);
    }

    // Construcción de la ruta final del archivo
    string ruta = $"{nombre}.{formato.ToLower()}";

    // Guarda el archivo según el formato elegido
    GuardarArchivo(ruta, alumnos, formato);

    imprimir($"Archivo '{ruta}' creado con éxito.");
}
```

`List<Alumno> alumnos = new();`

Se crea una lista genérica de tipo `Alumno` que se utilizará para almacenar todos los alumnos ingresados por el usuario.

Esta lista funcionará como estructura intermedia antes de guardar los datos en el archivo.

`for (int i = 0; i < cantidad; i++)`
Se inicia un bucle `for` que se ejecutará tantas veces como alumnos haya indicado el usuario previamente.
Cada iteración del bucle corresponde a la carga de un alumno.

`}imprimir($"--- Alumno {i + 1} ---");`
Se muestra un encabezado indicando el número del alumno que se está cargando, mejorando la claridad durante la interacción con el usuario.

`Alumno a = new Alumno();`

Se crea una nueva instancia de la clase `Alumno`, que representará a un alumno individual con todos sus datos.

```
a.Legajo = Input("Legajo:");
```

```
a.Apellido = Input("Apellido:");
```

```
a.Nombres = Input("Nombres:");
```

```
a.NumeroDocumento = Input("Documento:");
```

Se solicitan y asignan los datos básicos del alumno:

- Legajo
- Apellido
- Nombres
- Número de documento

Cada dato se obtiene mediante una ventana de entrada y se almacena directamente en el objeto `Alumno`.

```
string email;
do
{
   email = Input("Email:");
   if (!EmailValido(email))
       imprimir("Email inválido. Debe contener @ y dominio.");
}
while (!EmailValido(email));
```

Este bloque implementa la validación del correo electrónico:

- Se solicita el email al usuario.
- Se verifica que cumpla con una estructura básica válida mediante el método EmailValido.
- Si el email no es válido, se muestra un mensaje de error y se vuelve a solicitar.
- El ciclo se repite hasta que el usuario ingrese un email válido.

Esto asegura que todos los alumnos tengan un correo electrónico correctamente formado.

a.Email = email;

Una vez validado, el email se asigna al objeto Alumno.

a.Telefono = Input("Teléfono:");

Se solicita y almacena el número de teléfono del alumno.

alumnos.Add(a);

El alumno completamente cargado se agrega a la lista de alumnos.

string ruta = $"{nombre}.{formato.ToLower()}";

Se construye el nombre final del archivo utilizando:

- El nombre ingresado por el usuario.
- La extensión correspondiente al formato elegido.
- El formato se pasa a minúsculas para respetar las convenciones de nombres de archivos.

GuardarArchivo(ruta, alumnos, formato);

Se invoca el método GuardarArchivo, que se encarga de persistir los datos de los alumnos en el formato seleccionado (TXT, CSV, JSON o XML).

imprimir($"Archivo '{ruta}' creado con éxito.");

Se muestra un mensaje final indicando que el archivo fue creado correctamente.

4.2- Guardar Archivo

```
3 referencias
public void GuardarArchivo(string ruta, List<Alumno> alumnos, string formato)
{
    switch (formato)
    {
        case "TXT":
            GuardarTXT(ruta, alumnos);
            break;

        case "CSV":
            GuardarCSV(ruta, alumnos);
            break;

        case "JSON":
            GuardarJSON(ruta, alumnos);
            break;

        case "XML":
            GuardarXML(ruta, alumnos);
            break;
    }
}
```

Este método se encarga de **guardar la información de los alumnos en un archivo**, utilizando el **formato seleccionado por el usuario**.

Recibe tres parámetros:

- **ruta**: indica el nombre y la ubicación del archivo que se va a crear o sobrescribir.
- **alumnos**: lista de alumnos que se desea guardar.
- **formato**: tipo de archivo en el que se guardarán los datos (TXT, CSV, JSON o XML).

El funcionamiento del método es el siguiente:

1. Según el valor del parámetro *formato*, el sistema determina qué tipo de archivo debe generarse.

2. Mediante una estructura de selección (*switch*), se llama al método específico encargado de guardar los datos en el formato correspondiente:

    - **TXT**: guarda los alumnos en un archivo de texto plano, una línea por alumno.
    - **CSV**: guarda los alumnos en un archivo separado por comas, incluyendo un encabezado.
    - **JSON**: guarda los alumnos serializados en formato JSON.
    - **XML**: guarda los alumnos serializados en formato XML.

3. Cada método especializado se ocupa de aplicar la estructura correcta según el formato elegido.

De esta forma, el sistema centraliza la operación de guardado en un único método, asegurando que los datos se almacenen correctamente sin que el usuario tenga que preocuparse por los detalles internos del formato.

```
1 referencia
private static void GuardarTXT(string ruta, List<Alumno> alumnos)
{
    File.WriteAllLines(ruta, alumnos.Select(a => a.ToString()));
}

1 referencia
private static void GuardarCSV(string ruta, List<Alumno> alumnos)
{
    using StreamWriter sw = new StreamWriter(ruta);
    sw.WriteLine("Legajo,Apellido,Nombres,NumeroDocumento,Email,Telefono");

    foreach (var a in alumnos)
    {
        sw.WriteLine($"{a.Legajo},{a.Apellido},{a.Nombres},{a.NumeroDocumento},{a.Email},{a.Telefono}");
    }
}
1 referencia
private static void GuardarJSON(string ruta, List<Alumno> alumnos)
{
    string json = JsonSerializer.Serialize(
        alumnos,
        new JsonSerializerOptions { WriteIndented = true }
    );

    File.WriteAllText(ruta, json);
}
1 referencia
private static void GuardarXML(string ruta, List<Alumno> alumnos)
{
    XmlSerializer ser = new XmlSerializer(typeof(List<Alumno>));
    using FileStream fs = new FileStream(ruta, FileMode.Create);
    ser.Serialize(fs, alumnos);
}
```

Estos métodos se encargan de **almacenar la información de los alumnos en distintos formatos de archivo**, según el tipo seleccionado por el usuario.
Cada método recibe:

- **ruta**: ubicación y nombre del archivo a crear.
- **alumnos**: lista de alumnos que se desea guardar.

GuardarTXT

Este método guarda los datos de los alumnos en un **archivo de texto plano (.txt)**.

- Cada alumno se escribe en una línea del archivo.
- Para generar el contenido de cada línea se utiliza el método `ToString()` de la clase Alumno.
- El archivo se crea o se sobrescribe automáticamente si ya existe.

Este formato es simple y legible, pensado para visualización rápida o edición manual.

GuardarCSV

Este método guarda los datos en un **archivo CSV (.csv)**, separado por comas.

- Primero escribe una línea de encabezado con los nombres de los campos.
- Luego recorre la lista de alumnos y escribe una línea por cada uno.
- Cada dato se separa por comas, respetando el formato estándar CSV.
- Se utiliza un `StreamWriter` para escribir el archivo de forma controlada.

Este formato es ideal para abrir los datos en planillas de cálculo como Excel.

GuardarJSON

Este método guarda los datos en **formato JSON (.json)**.

- Convierte la lista de alumnos a una estructura JSON mediante serialización.
- Se utiliza la opción `WriteIndented` para que el archivo sea fácil de leer.
- El contenido generado se guarda completo en el archivo.

Este formato es ampliamente utilizado para intercambio de datos entre sistemas.

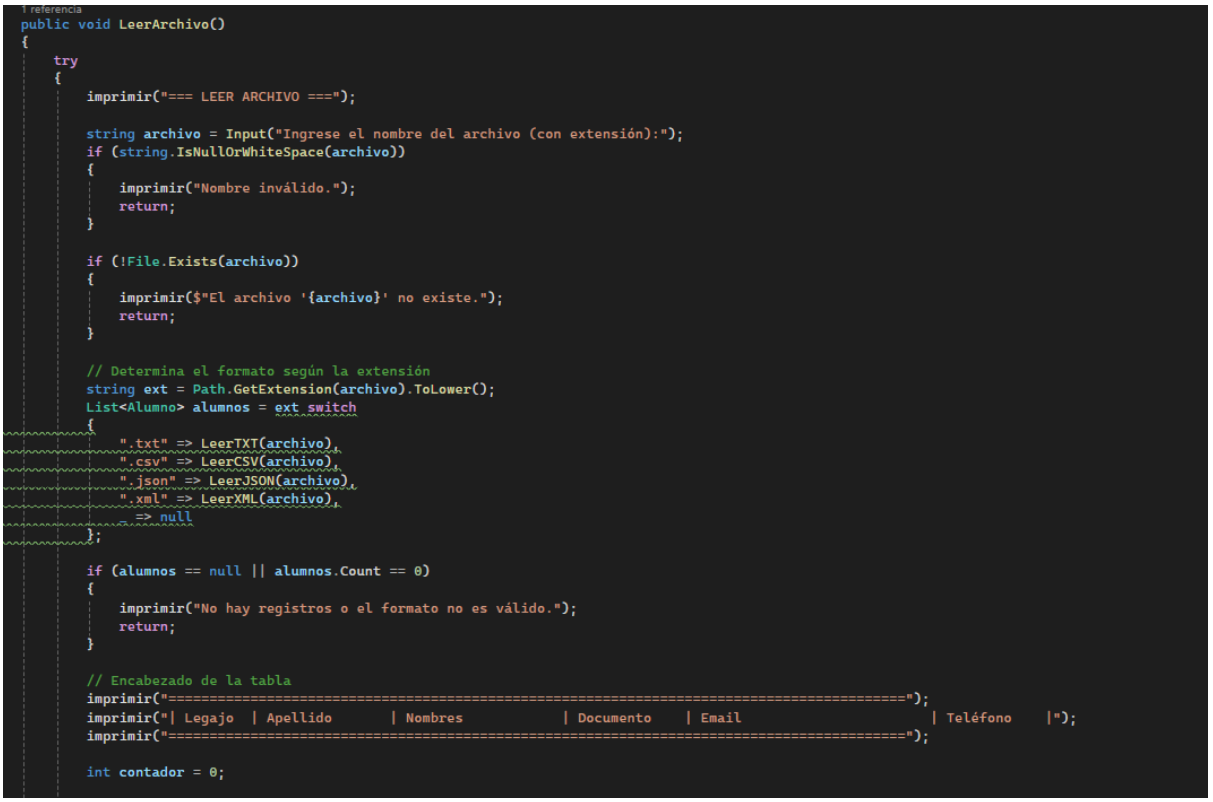
GuardarXML

Este método guarda los datos en **formato XML (.xml)**.

- Serializa la lista de alumnos utilizando `XmlSerializer`.
- Genera un archivo estructurado con etiquetas XML.
- El archivo se crea o se reemplaza si ya existe.

Este formato es útil para sistemas que requieren datos estructurados y validables.

4.3- Leer Archivo.



```
1 referencia
public void LeerArchivo()
{
    try
    {
        imprimir("=== LEER ARCHIVO ===");

        string archivo = Input("Ingrese el nombre del archivo (con extensión):");
        if (string.IsNullOrWhiteSpace(archivo))
        {
            imprimir("Nombre inválido.");
            return;
        }

        if (!File.Exists(archivo))
        {
            imprimir($"El archivo '{archivo}' no existe.");
            return;
        }

        // Determina el formato según la extensión
        string ext = Path.GetExtension(archivo).ToLower();
        List<Alumno> alumnos = ext switch
        {
            ".txt" => LeerTXT(archivo),
            ".csv" => LeerCSV(archivo),
            ".json" => LeerJSON(archivo),
            ".xml" => LeerXML(archivo),
            _ => null
        };

        if (alumnos == null || alumnos.Count == 0)
        {
            imprimir("No hay registros o el formato no es válido.");
            return;
        }

        // Encabezado de la tabla
        imprimir("=============================================================================================");
        imprimir("| Legajo  | Apellido       | Nombres            | Documento    | Email                      | Teléfono    |");
        imprimir("=============================================================================================");

        int contador = 0;
```

public void LeerArchivo()

{

try

{

imprimir("=== LEER ARCHIVO ===");

El método LeerArchivo se encarga de leer un archivo de alumnos existente y mostrar su contenido en pantalla.

Todo el código se encuentra dentro de un bloque try para capturar posibles errores durante la lectura.

string archivo = Input("Ingrese el nombre del archivo (con extensión):");

if (string.IsNullOrWhiteSpace(archivo))

{

imprimir("Nombre inválido.");

return;

```
}
```

Se solicita al usuario el nombre del archivo a leer, incluyendo su extensión.

Si el usuario no ingresa ningún valor o ingresa solo espacios en blanco, se muestra un mensaje de error y se interrumpe la ejecución del método.

```
if (!File.Exists(archivo))
{
    imprimir($"El archivo '{archivo}' no existe.");
    return;
}
```

Se verifica que el archivo indicado exista físicamente en el sistema.

Si el archivo no existe, se informa al usuario y se finaliza el método para evitar errores posteriores.

```
// Determina el formato según la extensión
string ext = Path.GetExtension(archivo).ToLower();
```

Se obtiene la extensión del archivo utilizando Path.GetExtension.

La extensión se convierte a minúsculas para facilitar su comparación y evitar problemas por diferencias de formato.

```
List<Alumno> alumnos = ext switch
{
    ".txt" => LeerTXT(archivo),
    ".csv" => LeerCSV(archivo),
    ".json" => LeerJSON(archivo),
    ".xml" => LeerXML(archivo),
    _ => null
};
```

Se utiliza una expresión switch para determinar qué método de lectura utilizar según la extensión del archivo:

- .txt → LeerTXT
- .csv → LeerCSV
- .json → LeerJSON
- .xml → LeerXML

Cada uno de estos métodos devuelve una lista de alumnos leída desde el archivo correspondiente.

Si la extensión no es válida, se devuelve null.

```
if (alumnos == null || alumnos.Count == 0)

{

   imprimir("No hay registros o el formato no es válido.");

   return;

}
```

Se valida el resultado de la lectura:

- Si la lista es null, el formato no está soportado.
- Si la lista está vacía, el archivo no contiene alumnos.
- En ambos casos, se muestra un mensaje informativo y se detiene la ejecución del método.

```
// Encabezado de la tabla

imprimir("================================================================================");

imprimir("| Legajo  | Apellido       | Nombres            | Documento    | Email                  | Teléfono    |");

imprimir("================================================================================");
```

Se imprime el encabezado de una tabla formateada que servirá para mostrar los datos de los alumnos de manera ordenada y legible en la interfaz.

```
int contador = 0;
```

Se inicializa un contador que se utilizará posteriormente para llevar control de la cantidad de registros mostrados y aplicar paginación si es necesario.

```
        // Muestra los alumnos en formato de tabla
        foreach (var a in alumnos)
        {
            string linea =
                $"| {a.Legajo.PadRight(7)}" +
                $"| {a.Apellido.PadRight(14)}" +
                $"| {a.Nombres.PadRight(18)}" +
                $"| {a.NumeroDocumento.PadRight(12)}" +
                $"| {a.Email.PadRight(27)}" +
                $"| {a.Telefono.PadRight(12)}|";

            imprimir(linea);
            contador++;

            // Paginación cada 20 registros
            if (contador % 20 == 0)
            {
                imprimir("---- Pulse ENTER para continuar ----");
                MessageBox.Show("Mostrando 20 registros.\nPulse Aceptar para continuar.");
            }
        }

        imprimir("=============================================================================================");
        imprimir($"Total de alumnos: {alumnos.Count}");
    }
    catch (Exception ex)
    {
        imprimir("Ocurrió un error al leer el archivo:");
        imprimir(ex.Message);
    }
}
```

// Muestra los alumnos en formato de tabla
foreach (var a in alumnos)
{
Se recorre la lista de alumnos obtenida desde el archivo.
Cada iteración representa un alumno que será mostrado en pantalla.

string linea =
$"| {a.Legajo.PadRight(7)}" +
$"| {a.Apellido.PadRight(14)}" +
$"| {a.Nombres.PadRight(18)}" +
$"| {a.NumeroDocumento.PadRight(12)}" +
$"| {a.Email.PadRight(27)}" +
$"| {a.Telefono.PadRight(12)}|";
Se construye una línea de texto formateada para mostrar los datos del alumno en forma de tabla:
PadRight(n) completa cada campo con espacios para que todas las columnas tengan el mismo ancho.

Esto permite que los datos queden alineados visualmente, facilitando la lectura.
Cada línea representa una fila de la tabla.

```
imprimir(linea);
contador++;
```
La línea formateada se imprime en la interfaz y se incrementa el contador de registros mostrados.

```
// Paginación cada 20 registros
if (contador % 20 == 0)
{
    imprimir("---- Pulse ENTER para continuar ----");
    MessageBox.Show("Mostrando 20 registros.\nPulse Aceptar para continuar.");
}
```
Este bloque implementa un sistema de paginación:

- Cada 20 alumnos mostrados, se detiene momentáneamente la visualización.
- Se informa al usuario que se han mostrado 20 registros.
- Se muestra una ventana de mensaje que permite continuar al presionar Aceptar.
- Esto evita que la salida sea demasiado extensa y mejora la experiencia de uso.

```
imprimir("================================================================
==================================");
imprimir($"Total de alumnos: {alumnos.Count}");
```
Se imprime una línea de cierre de la tabla y se muestra el total de alumnos leídos desde el archivo.

```
}
catch (Exception ex)
{
    imprimir("Ocurrió un error al leer el archivo:");
    imprimir(ex.Message);
}
```
Este bloque catch captura cualquier excepción que pueda ocurrir durante la lectura o visualización del archivo:

- Se muestra un mensaje indicando que ocurrió un error.
- Se imprime el mensaje de la excepción para facilitar la identificación del problema.

```
public List<Alumno> LeerTXT(string ruta)
{
    List<Alumno> lista = new();

    foreach (var linea in File.ReadAllLines(ruta))
    {
        string[] campos = linea.Split('|');

        if (campos.Length == 6)
        {
            lista.Add(new Alumno
            {
                Legajo = campos[0],
                Apellido = campos[1],
                Nombres = campos[2],
                NumeroDocumento = campos[3],
                Email = campos[4],
                Telefono = campos[5]
            });
        }
    }

    return lista;
}
```

public List<Alumno> LeerTXT(string ruta)
Este método se encarga de leer un archivo de texto plano (.txt) que contiene información de alumnos y convertir su contenido en una lista de objetos Alumno.

Recibe como parámetro:

ruta: la ubicación del archivo TXT que se desea leer.
Devuelve:
Una lista de alumnos cargados desde el archivo.

List<Alumno> lista = new();
Se crea una lista vacía de tipo Alumno donde se almacenarán los registros leídos del archivo.

foreach (var linea in File.ReadAllLines(ruta))
{
Se leen todas las líneas del archivo utilizando File.ReadAllLines.
Cada línea representa un alumno y se procesa individualmente.

string[] campos = linea.Split('|');
Cada línea se divide en partes utilizando el carácter | como separador.
Este separador coincide con el formato utilizado al guardar archivos TXT mediante el método ToString() de la clase Alumno.

if (campos.Length == 6)

{
Se valida que la línea contenga exactamente seis campos, correspondientes a:

- Legajo
- Apellido
- Nombres
- Documento
- Email
- Teléfono

Esto evita errores si alguna línea está mal formada.

```
lista.Add(new Alumno
{
   Legajo = campos[0],
   Apellido = campos[1],
   Nombres = campos[2],
   NumeroDocumento = campos[3],
   Email = campos[4],
   Telefono = campos[5]
});
```

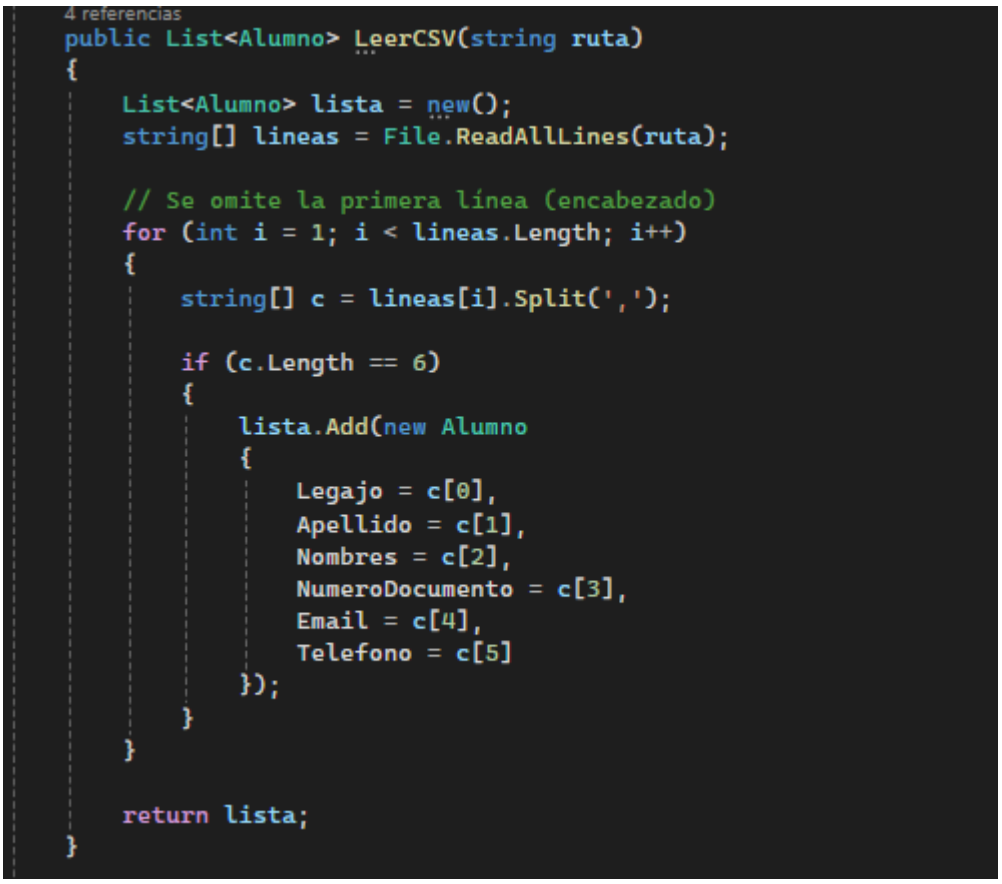
Si la línea es válida, se crea un nuevo objeto Alumno asignando cada campo leído a su propiedad correspondiente, y se agrega a la lista.

```
return lista;
```

Se devuelve la lista completa de alumnos cargados desde el archivo TXT.

Resumen
El método LeerTXT:}

- Lee archivos de texto plano.}
- Convierte cada línea en un objeto Alumno.
- Valida la estructura del archivo.
- Devuelve una lista lista para ser utilizada por otras funcionalidades del sistema.



```
4 referencias
public List<Alumno> LeerCSV(string ruta)
{
    List<Alumno> lista = new();
    string[] lineas = File.ReadAllLines(ruta);

    // Se omite la primera línea (encabezado)
    for (int i = 1; i < lineas.Length; i++)
    {
        string[] c = lineas[i].Split(',');

        if (c.Length == 6)
        {
            lista.Add(new Alumno
            {
                Legajo = c[0],
                Apellido = c[1],
                Nombres = c[2],
                NumeroDocumento = c[3],
                Email = c[4],
                Telefono = c[5]
            });
        }
    }

    return lista;
}
```

public List<Alumno> LeerCSV(string ruta)
Este método se encarga de leer un archivo CSV que contiene información de alumnos y convertir cada registro en un objeto Alumno.

Recibe como parámetro:
ruta: la ubicación del archivo CSV que se desea leer.
Devuelve:
Una lista de alumnos cargados desde el archivo.

List<Alumno> lista = new();
Se crea una lista vacía de tipo Alumno donde se almacenarán los alumnos leídos del archivo.

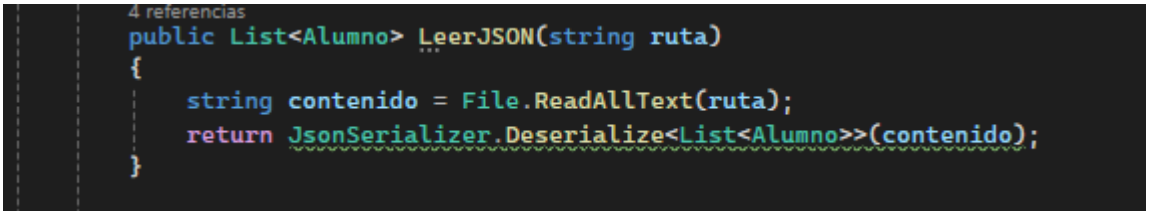
string[] lineas = File.ReadAllLines(ruta);
Se leen todas las líneas del archivo CSV y se almacenan en un arreglo de strings. Cada línea del archivo representa un registro.

for (int i = 1; i < lineas.Length; i++)
{
Se utiliza un bucle for que comienza desde la segunda línea del archivo (índice 1). Esto se debe a que la primera línea contiene el encabezado con los nombres de las columnas.

string[] c = lineas[i].Split(',');
Cada línea se divide utilizando la coma , como separador, obteniendo los distintos campos del alumno.

if (c.Length == 6)
{
Se valida que cada línea contenga exactamente seis campos, correspondientes a los datos del alumno.

lista.Add(new Alumno
{
  Legajo = c[0],
  Apellido = c[1],
  Nombres = c[2],
  NumeroDocumento = c[3],
  Email = c[4],
  Telefono = c[5]
});
Si la línea es válida, se crea un nuevo objeto Alumno, se asignan los valores correspondientes y se agrega a la lista.
return lista;
Se devuelve la lista completa de alumnos cargados desde el archivo.



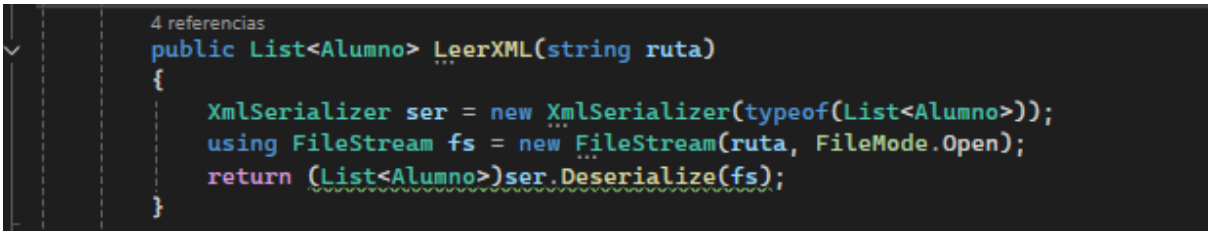
public List<Alumno> LeerJSON(string ruta)
Este método se encarga de leer un archivo en formato JSON que contiene una lista de alumnos y convertirlo en una lista de objetos Alumno.

Recibe como parámetro:
ruta: la ubicación del archivo JSON que se desea leer.
Devuelve:
Una lista de alumnos deserializados desde el archivo.

string contenido = File.ReadAllText(ruta);
Se lee todo el contenido del archivo JSON ubicado en la ruta indicada y se almacena en una variable de tipo string.

return JsonSerializer.Deserialize<List<Alumno>>(contenido);
Se utiliza el serializador JSON de .NET para:

- Convertir el texto JSON en una lista de objetos Alumno.
- Mapear automáticamente cada propiedad del JSON con las propiedades de la clase Alumno.
- El resultado de la deserialización se devuelve directamente.



```
4 referencias
public List<Alumno> LeerXML(string ruta)
{
    XmlSerializer ser = new XmlSerializer(typeof(List<Alumno>));
    using FileStream fs = new FileStream(ruta, FileMode.Open);
    return (List<Alumno>)ser.Deserialize(fs);
}
```

public List<Alumno> LeerXML(string ruta)
Este método se encarga de leer un archivo en formato XML que contiene una lista de alumnos y convertirlo en una lista de objetos Alumno.

Recibe como parámetro:
ruta: la ubicación del archivo XML que se desea leer.

Devuelve:
Una lista de objetos Alumno obtenida a partir del archivo XML.

XmlSerializer ser = new XmlSerializer(typeof(List<Alumno>));
Se crea una instancia de XmlSerializer indicando que el tipo de dato a deserializar es una lista de alumnos (List<Alumno>).
Esto le permite al sistema conocer la estructura del XML esperado.

using FileStream fs = new FileStream(ruta, FileMode.Open);
Se abre el archivo XML en modo lectura mediante un FileStream.

La palabra clave using garantiza que el archivo se cierre correctamente una vez finalizada la operación.

return (List<Alumno>)ser.Deserialize(fs);
Se deserializa el contenido del archivo XML y se lo convierte en una lista de objetos Alumno.
El resultado se devuelve para su posterior procesamiento en el sistema.

4.3- Modificar Archivo.

```
public void ModificarArchivo()
{
    imprimir("=== MODIFICAR ARCHIVO ===");

    string archivo = Input("Ingrese el nombre del archivo (con extensión):");

    if (string.IsNullOrWhiteSpace(archivo))
    {
        imprimir("Nombre inválido.");
        return;
    }

    if (!File.Exists(archivo))
    {
        imprimir($"El archivo '{archivo}' no existe.");
        return;
    }

    string ext = Path.GetExtension(archivo).ToLower();

    // Carga los alumnos según el formato
    List<Alumno> alumnos = ext switch
    {
        ".txt" => LeerTXT(archivo),
        ".csv" => LeerCSV(archivo),
        ".json" => LeerJSON(archivo),
        ".xml" => LeerXML(archivo),
        _ => null
    };

    if (alumnos == null)
    {
        imprimir("Formato no soportado.");
        return;
    }

    bool salir = false;
    bool guardar = false;
```

public void ModificarArchivo()
Este método permite modificar el contenido de un archivo existente de alumnos, independientemente de su formato (TXT, CSV, JSON o XML).

imprimir("=== MODIFICAR ARCHIVO ===");
Muestra un encabezado en pantalla indicando que se ingresó a la funcionalidad de modificación de archivos.

string archivo = Input("Ingrese el nombre del archivo (con extensión):");
Solicita al usuario el nombre del archivo que desea modificar, incluyendo su extensión.

if (string.IsNullOrWhiteSpace(archivo))
{
  imprimir("Nombre inválido.");
  return;
}

Valida que el nombre ingresado no esté vacío ni compuesto solo por espacios.
Si es inválido, se muestra un mensaje de error y se cancela la operación.

```
if (!File.Exists(archivo))
{
   imprimir($"El archivo '{archivo}' no existe.");
   return;
}
```

Verifica que el archivo realmente exista en el sistema.
Si no existe, se informa al usuario y se finaliza el método.

```
string ext = Path.GetExtension(archivo).ToLower();
```

Obtiene la extensión del archivo para determinar su formato y decidir cómo leerlo.

```
List<Alumno> alumnos = ext switch
{
   ".txt" => LeerTXT(archivo),
   ".csv" => LeerCSV(archivo),
   ".json" => LeerJSON(archivo),
   ".xml" => LeerXML(archivo),
   _ => null
};
```

Carga los datos del archivo en una lista de alumnos utilizando el método de lectura correspondiente según el formato.
Esto permite trabajar internamente con una estructura común (List<Alumno>), sin importar el tipo de archivo original.

```
if (alumnos == null)
{
   imprimir("Formato no soportado.");
   return;
}
```

Controla que el formato del archivo sea válido y compatible.
Si no lo es, se muestra un mensaje de error y se cancela la operación.

```
bool salir = false;
bool guardar = false;
```

Inicializa dos variables de control:

- salir: determina cuándo se debe abandonar el submenú de modificación.
- guardar: indica si los cambios realizados deben guardarse o descartarse.
- Estas variables se utilizan más adelante para manejar el flujo del submenú.

```
while (!salir)
{
    imprimir("");
    imprimir("=== SUB-MENÚ DE MODIFICACIÓN ===");
    imprimir("1. Agregar alumno");
    imprimir("2. Modificar alumno por legajo");
    imprimir("3. Eliminar alumno por legajo");
    imprimir("4. Guardar cambios y salir");
    imprimir("5. Cancelar (salir sin guardar)");

    string opcion = Input("Ingrese una opción:");

    switch (opcion)
    {
        case "1":
            AgregarAlumno(alumnos);
            break;
        case "2":
            EditarAlumno(alumnos);
            break;
        case "3":
            EliminarAlumno(alumnos);
            break;
        case "4":
            guardar = true;
            salir = true;
            break;
        case "5":
            salir = true;
            break;
        default:
            imprimir("Opción inválida.");
            break;
    }
}
```

while (!salir)
Este bucle mantiene activo el sub-menú de modificación hasta que el usuario decida salir.
La variable salir controla cuándo finalizar el ciclo.

imprimir("");
imprimir("=== SUB-MENÚ DE MODIFICACIÓN ===");
imprimir("1. Agregar alumno");
imprimir("2. Modificar alumno por legajo");
imprimir("3. Eliminar alumno por legajo");
imprimir("4. Guardar cambios y salir");
imprimir("5. Cancelar (salir sin guardar)");
Muestra en pantalla las opciones disponibles para modificar el archivo:

- Agregar un nuevo alumno.
- Editar los datos de un alumno existente, buscándolo por su legajo.
- Eliminar un alumno del archivo.
- Guardar los cambios realizados y salir del sub-menú.
- Salir sin guardar los cambios realizados.
-

string opcion = Input("Ingrese una opción:");
Solicita al usuario que seleccione una opción del sub-menú.

switch (opcion)
Evalúa la opción ingresada y ejecuta la acción correspondiente.

```
case "1":
    AgregarAlumno(alumnos);
    break;
```
Permite agregar un nuevo alumno a la lista cargada en memoria.
```
case "2":
    EditarAlumno(alumnos);
    break;
```
Permite modificar los datos de un alumno existente, identificado por su legajo.
```
case "3":
    EliminarAlumno(alumnos);
    break;
```
Permite eliminar un alumno de la lista utilizando su legajo como referencia.
```
case "4":
    guardar = true;
    salir = true;
    break;
```
Indica que los cambios deben guardarse y finaliza el sub-menú.
Más adelante, esta opción provoca que el archivo sea sobrescrito con los nuevos datos.
```
case "5":
    salir = true;
    break;
```
Sale del sub-menú sin guardar ningún cambio realizado.
```
default:
    imprimir("Opción inválida.");
    break;
```

```
    // Guarda cambios creando backup
    if (guardar)
    {
        string backup = archivo + ".bak";
        File.Copy(archivo, backup, true);
        imprimir($"Backup creado: {backup}");

        GuardarArchivo(archivo, alumnos, ext.Substring(1).ToUpper());
        imprimir("Cambios guardados.");
    }
    else
    {
        imprimir("Cambios descartados.");
    }
}
```
`if (guardar)`
Evalúa si el usuario eligió la opción “Guardar cambios y salir” dentro del sub-menú de modificación.
La variable guardar se establece en true solo cuando el usuario selecciona esa opción.

string backup = archivo + ".bak";
Construye el nombre del archivo de respaldo agregando la extensión .bak al archivo original.
Este archivo funcionará como copia de seguridad.

File.Copy(archivo, backup, true);
Crea el backup del archivo original antes de sobrescribirlo.
El parámetro true indica que, si el backup ya existe, será reemplazado.

imprimir($"Backup creado: {backup}");
Informa al usuario que el respaldo fue creado correctamente.

GuardarArchivo(archivo, alumnos, ext.Substring(1).ToUpper());
Guarda nuevamente el archivo original utilizando:

- El mismo nombre de archivo.
- La lista de alumnos modificada en memoria.
- El formato correspondiente, obtenido a partir de la extensión del archivo.
- Esto sobrescribe el archivo original con los cambios realizados.

imprimir("Cambios guardados.");
Confirma al usuario que los cambios fueron persistidos correctamente.

```
else
{
    imprimir("Cambios descartados.");
}
```

Si el usuario eligió salir sin guardar, no se realiza ninguna modificación sobre el archivo original.
Todos los cambios hechos en memoria se descartan.

```
private void AgregarAlumno(List<Alumno> alumnos)
{
    imprimir("=== AGREGAR ALUMNO ===");

    string legajo = Input("Legajo:");

    if (string.IsNullOrWhiteSpace(legajo))
    {
        imprimir("Legajo inválido.");
        return;
    }

    if (alumnos.Any(a => a.Legajo == legajo))
    {
        imprimir("Ese legajo ya existe.");
        return;
    }

    Alumno nuevo = new Alumno
    {
        Legajo = legajo,
        Apellido = Input("Apellido:"),
        Nombres = Input("Nombres:"),
        NumeroDocumento = Input("Documento:"),
        Email = Input("Email:"),
        Telefono = Input("Teléfono:")
    };

    alumnos.Add(nuevo);
    imprimir("Alumno agregado.");
}
```

Método `AgregarAlumno`

Este método permite **agregar un nuevo alumno** a una lista existente de alumnos que se encuentra cargada en memoria.

```
private void AgregarAlumno(List<Alumno> alumnos)
```

El método recibe como parámetro una **lista de alumnos** (`List<Alumno>`).
Las modificaciones se realizan directamente sobre esa lista.

```
imprimir("=== AGREGAR ALUMNO ===");
```

Muestra un título para indicar que se está ejecutando la opción de agregar un alumno.

```
string legajo = Input("Legajo:");
```

Solicita al usuario el **legajo** del nuevo alumno, que será utilizado como identificador único.

```
if (string.IsNullOrWhiteSpace(legajo))
{
    imprimir("Legajo inválido.");
    return;
}
```

Valida que el legajo no esté vacío ni contenga solo espacios.
Si el valor no es válido, el método finaliza sin agregar el alumno.

```
if (alumnos.Any(a => a.Legajo == legajo))
{
    imprimir("Ese legajo ya existe.");
    return;
}
```

Verifica que **no exista otro alumno con el mismo legajo** dentro de la lista.
Esto evita duplicados y asegura la integridad de los datos.

```
Alumno nuevo = new Alumno
{
    Legajo = legajo,
    Apellido = Input("Apellido:"),
    Nombres = Input("Nombres:"),
    NumeroDocumento = Input("Documento:"),
    Email = Input("Email:"),
    Telefono = Input("Teléfono:")
};
```

Crea una nueva instancia de la clase `Alumno` y solicita al usuario los datos necesarios para completar sus propiedades.

```
alumnos.Add(nuevo);
```

Agrega el nuevo alumno a la lista recibida como parámetro.

```
imprimir("Alumno agregado.");
```

Informa al usuario que el alumno fue agregado correctamente.

```
1 referencia
private void EditarAlumno(List<Alumno> alumnos)
{
    string leg = Input("Legajo del alumno a modificar:");

    var alum = alumnos.FirstOrDefault(a => a.Legajo == leg);

    if (alum == null)
    {
        imprimir("No existe ese legajo.");
        return;
    }

    imprimir("Deje vacío para mantener el valor anterior.");

    string nuevo;

    nuevo = Input($"Apellido ({alum.Apellido}):");
    if (!string.IsNullOrWhiteSpace(nuevo)) alum.Apellido = nuevo;

    nuevo = Input($"Nombres ({alum.Nombres}):");
    if (!string.IsNullOrWhiteSpace(nuevo)) alum.Nombres = nuevo;

    nuevo = Input($"Documento ({alum.NumeroDocumento}):");
    if (!string.IsNullOrWhiteSpace(nuevo)) alum.NumeroDocumento = nuevo;

    nuevo = Input($"Email ({alum.Email}):");
    if (!string.IsNullOrWhiteSpace(nuevo))
    {
        if (!EmailValido(nuevo))
        {
            imprimir("Email inválido. No se actualizó el email.");
        }
        else
        {
            alum.Email = nuevo;
        }
    }

    nuevo = Input($"Teléfono ({alum.Telefono}):");
    if (!string.IsNullOrWhiteSpace(nuevo)) alum.Telefono = nuevo;

    imprimir("Alumno actualizado.");
}
```

Este método permite **modificar los datos de un alumno existente** dentro de una lista, identificándose por su legajo.

```
private void EditarAlumno(List<Alumno> alumnos)
```

El método recibe una **lista de alumnos** y trabaja directamente sobre ella.

```
string leg = Input("Legajo del alumno a modificar:");
```

Solicita al usuario el **legajo** del alumno que desea modificar.

```
var alum = alumnos.FirstOrDefault(a => a.Legajo == leg);
```

Busca dentro de la lista el alumno cuyo legajo coincida con el ingresado.
Si no se encuentra, `alum` queda en `null`.

```
if (alum == null)
{
    imprimir("No existe ese legajo.");
    return;
}
```

Si el alumno no existe, se informa al usuario y el método finalizar.

```
imprimir("Deje vacío para mantener el valor anterior.");
```

Indica al usuario que puede **presionar Enter sin escribir nada** para conservar el valor actual de cada campo.

```
string nuevo;
```

Variable auxiliar que se utiliza para almacenar cada nuevo dato ingresado.

Modificación de apellido

```
nuevo = Input($"Apellido ({alum.Apellido}):");
if (!string.IsNullOrWhiteSpace(nuevo)) alum.Apellido = nuevo;
```

Muestra el apellido actual entre paréntesis.
Si el usuario ingresa un valor, se actualiza; si deja vacío, se mantiene el anterior.

Modificación de nombres

```
nuevo = Input($"Nombres ({alum.Nombres}):");
if (!string.IsNullOrWhiteSpace(nuevo)) alum.Nombres = nuevo;
```

Funciona de la misma manera que el apellido.

Modificación de documento

```
nuevo = Input($"Documento ({alum.NumeroDocumento}):");
if (!string.IsNullOrWhiteSpace(nuevo)) alum.NumeroDocumento =
nuevo;
```

Permite actualizar el número de documento si se ingresa un valor.

Modificación de email con validación

```
nuevo = Input($"Email ({alum.Email}):");
if (!string.IsNullOrWhiteSpace(nuevo))
{
    if (!EmailValido(nuevo))
    {
        imprimir("Email inválido. No se actualizó el email.");
    }
    else
    {
        alum.Email = nuevo;
    }
}
```

- Si el usuario deja el campo vacío, el email no se modifica.
- Si ingresa un valor:
    - Se valida el formato del email.
    - Si es inválido, se informa y no se actualiza.
    - Si es válido, se guarda el nuevo email.

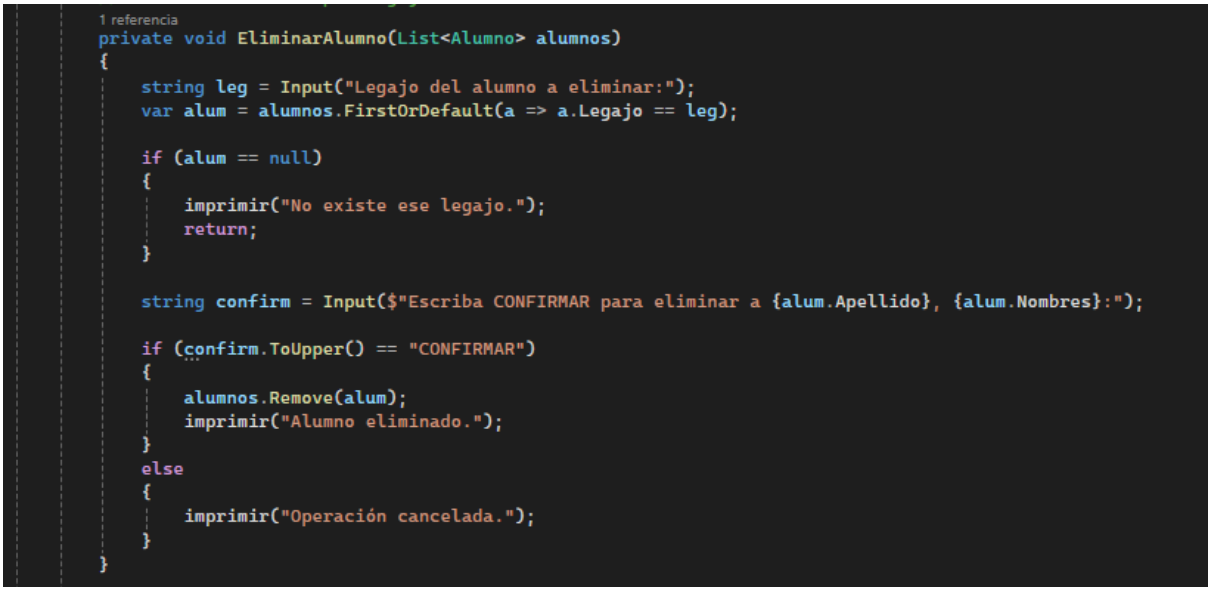
Modificación del teléfono

```
nuevo = Input($"Teléfono ({alum.Telefono}):");
if (!string.IsNullOrWhiteSpace(nuevo)) alum.Telefono = nuevo;
```

Actualiza el teléfono solo si se ingresa un valor nuevo.

```
imprimir("Alumno actualizado.");
```

Confirma al usuario que los datos del alumno fueron modificados correctamente.



```
1 referencia
private void EliminarAlumno(List<Alumno> alumnos)
{
    string leg = Input("Legajo del alumno a eliminar:");
    var alum = alumnos.FirstOrDefault(a => a.Legajo == leg);

    if (alum == null)
    {
        imprimir("No existe ese legajo.");
        return;
    }

    string confirm = Input($"Escriba CONFIRMAR para eliminar a {alum.Apellido}, {alum.Nombres}:");

    if (confirm.ToUpper() == "CONFIRMAR")
    {
        alumnos.Remove(alum);
        imprimir("Alumno eliminado.");
    }
    else
    {
        imprimir("Operación cancelada.");
    }
}
```

Este método permite **eliminar un alumno de la lista**, identificándolo por su legajo y solicitando una confirmación explícita antes de realizar la eliminación.

```
private void EliminarAlumno(List<Alumno> alumnos)
```

El método recibe una **lista de alumnos** y trabaja directamente sobre ella.

```
string leg = Input("Legajo del alumno a eliminar:");
```

Solicita al usuario el **legajo** del alumno que desea eliminar.

```
var alum = alumnos.FirstOrDefault(a => a.Legajo == leg);
```

Busca en la lista el alumno cuyo legajo coincida con el ingresado.
Si no se encuentra, `alum` será `null`.

```
if (alum == null)
{
    imprimir("No existe ese legajo.");
    return;
}
```

Si el alumno no existe, se informa al usuario y se cancela la operación.

```
string confirm = Input($"Escriba CONFIRMAR para eliminar a
{alum.Apellido}, {alum.Nombres}:");
```

Solicita una **confirmación explícita** al usuario, mostrando el nombre del alumno a eliminar para evitar errores.

```
if (confirm.ToUpper() == "CONFIRMAR")
```

Convierte el texto ingresado a mayúsculas y verifica que sea exactamente `"CONFIRMAR"`.

```
alumnos.Remove(alum);
```

Elimina el alumno de la lista si la confirmación es correcta.

```
imprimir("Alumno eliminado.");
```

Informa que el alumno fue eliminado exitosamente.

```
else
{
    imprimir("Operación cancelada.");
}
```

Si la confirmación no es correcta, la eliminación se cancela y no se realizan cambios.

4.4- Eliminar Archivo

```
1 referencia
public void EliminarArchivo()
{
    try
    {
        imprimir("=== ELIMINAR ARCHIVO ===");

        string archivo = Input("Ingrese el nombre del archivo (con extensión):");

        if (string.IsNullOrWhiteSpace(archivo))
        {
            imprimir("Nombre inválido.");
            return;
        }

        if (!File.Exists(archivo))
        {
            imprimir($"El archivo '{archivo}' no existe.");
            return;
        }

        // Muestra información del archivo
        FileInfo info = new FileInfo(archivo);

        imprimir("-----------------------------------------------");
        imprimir($"Nombre: {info.Name}");
        imprimir($"Tamaño: {info.Length / 1024.0:F2} KB");
        imprimir($"Creado: {info.CreationTime}");
        imprimir($"Última modificación: {info.LastWriteTime}");
        imprimir("-----------------------------------------------");

        string confirmacion = Input("Escriba CONFIRMAR para eliminar el archivo:");

        if (confirmacion.ToUpper() == "CONFIRMAR")
        {
            File.Delete(archivo);
            imprimir($"Archivo '{archivo}' eliminado correctamente.");
        }
        else
        {
            imprimir("Operación cancelada.");
        }
    }
    catch (Exception ex)
    {
        imprimir("Error al intentar eliminar el archivo:");
        imprimir(ex.Message);
    }
}
```

Este método permite **eliminar un archivo del sistema**, mostrando previamente su información y solicitando confirmación explícita al usuario para evitar eliminaciones accidentales.

`public void EliminarArchivo()`

Define un método público que puede ser invocado desde el menú principal del sistema.

`try`

`{`

Encierra toda la lógica en un bloque `try` para **capturar errores inesperados**, como problemas de acceso al archivo.

```
imprimir("=== ELIMINAR ARCHIVO ===");
```

Muestra un encabezado informativo indicando la operación que se va a realizar.

```
string archivo = Input("Ingrese el nombre del archivo (con
extensión):");
```

Solicita al usuario el **nombre completo del archivo**, incluyendo su extensión.

```
if (string.IsNullOrWhiteSpace(archivo))

{

    imprimir("Nombre inválido.");

    return;

}
```

Valida que el nombre ingresado no esté vacío ni contenga solo espacios.
Si es inválido, se cancela la operación.

```
if (!File.Exists(archivo))

{

    imprimir($"El archivo '{archivo}' no existe.");

    return;

}
```

Verifica si el archivo existe en el sistema.
Si no existe, se informa al usuario y se cancela la eliminación.

```
FileInfo info = new FileInfo(archivo);
```

Crea un objeto `FileInfo` para obtener **información detallada del archivo**.

```
imprimir("-----------------------------------------");

imprimir($"Nombre: {info.Name}");

imprimir($"Tamaño: {info.Length / 1024.0:F2} KB");
```

```
imprimir($"Creado: {info.CreationTime}");

imprimir($"Última modificación: {info.LastWriteTime}");

imprimir("-----------------------------------------------");
```

Muestra al usuario los **datos relevantes del archivo**:

- Nombre
- Tamaño en kilobytes
- Fecha de creación.
- Fecha de última modificación.

Esto permite verificar que se está eliminando el archivo correcto.

```
string confirmacion = Input("Escriba CONFIRMAR para eliminar
el archivo:");
```

Solicita una **confirmación explícita** antes de proceder con la eliminación.

```
if (confirmacion.ToUpper() == "CONFIRMAR")
```

Convierte el texto ingresado a mayúsculas y verifica que coincida exactamente con `"CONFIRMAR"`.

```
File.Delete(archivo);
```

Elimina el archivo del sistema si la confirmación es correcta.

```
imprimir($"Archivo '{archivo}' eliminado correctamente.");
```

Informa que el archivo fue eliminado con éxito.

```
else

{

    imprimir("Operación cancelada.");

}
```

Si la confirmación no es válida, la eliminación se cancela y no se realizan cambios.

```
catch (Exception ex)
```

```
{
    imprimir("Error al intentar eliminar el archivo:");
    imprimir(ex.Message);
}
```

Captura cualquier error ocurrido durante el proceso y muestra un mensaje descriptivo al usuario.

4.5- Convertir Formato.

```
public void ConvertirFormato()
{
    try
    {
        // Imprime un mensaje indicando que se iniciará la conversión
        imprimir("=== CONVERSIÓN DE FORMATOS ===");

        // Solicita el nombre del archivo de origen (incluyendo la extensión)
        string archivoOrigen = Input("Ingrese el archivo origen con extensión:");

        // Verifica si el nombre del archivo está vacío o nulo
        if (string.IsNullOrWhiteSpace(archivoOrigen))
        {
            imprimir("Nombre inválido.");
            return; // Finaliza el método si el nombre del archivo es inválido
        }

        // Verifica si el archivo existe en la ubicación especificada
        if (!File.Exists(archivoOrigen))
        {
            imprimir($"El archivo '{archivoOrigen}' no existe.");
            return; // Finaliza el método si el archivo no se encuentra
        }

        // Obtiene la extensión del archivo y la convierte a minúsculas para procesarlo
        string ext = Path.GetExtension(archivoOrigen).ToLower();

        // Dependiendo de la extensión del archivo, se leerán los datos de diferentes formas
        List<Alumno> alumnos = ext switch
        {
            ".txt" => gestor.LeerTXT(archivoOrigen),  // Lee archivo en formato TXT
            ".csv" => gestor.LeerCSV(archivoOrigen),  // Lee archivo en formato CSV
            ".json" => gestor.LeerJSON(archivoOrigen), // Lee archivo en formato JSON
            ".xml" => gestor.LeerXML(archivoOrigen),   // Lee archivo en formato XML
            _ => null  // Si el archivo no tiene una extensión válida, retorna null
        };

        // Verifica si los datos no fueron leídos correctamente o si el formato no es soportado
        if (alumnos == null)
        {
            imprimir("Formato no soportado.");
            return; // Finaliza el método si el formato no es soportado
        }
```

Este método permite **convertir un archivo de alumnos de un formato a otro** entre TXT, CSV, JSON y XML, validando previamente la existencia y el formato del archivo de origen.

`public void ConvertirFormato()`

Define un método público encargado de realizar la conversión de formatos de archivos.

`try`

`{`

Se utiliza un bloque `try` para capturar y manejar posibles errores durante el proceso de conversión.

`imprimir("=== CONVERSIÓN DE FORMATOS ===");`

Muestra un mensaje en pantalla indicando el inicio del proceso de conversión.

`string archivoOrigen = Input("Ingrese el archivo origen con extensión:");`

Solicita al usuario el nombre del archivo de origen, incluyendo su extensión.

```
if (string.IsNullOrWhiteSpace(archivoOrigen))
{
    imprimir("Nombre inválido.");
    return;
}
```

Verifica que el nombre del archivo no esté vacío ni contenga solo espacios.
Si la validación falla, el método finaliza.

```
if (!File.Exists(archivoOrigen))
{
    imprimir($"El archivo '{archivoOrigen}' no existe.");
    return;
}
```

Comprueba que el archivo exista en el sistema.
Si no existe, se informa al usuario y se cancela la operación.

```
string ext = Path.GetExtension(archivoOrigen).ToLower();
```

Obtiene la extensión del archivo y la convierte a minúsculas para facilitar su evaluación.

```
List<Alumno> alumnos = ext switch
{
    ".txt" => gestor.LeerTXT(archivoOrigen),
    ".csv" => gestor.LeerCSV(archivoOrigen),
    ".json" => gestor.LeerJSON(archivoOrigen),
    ".xml" => gestor.LeerXML(archivoOrigen),
    _ => null
```

```
};
```

Según la extensión del archivo, se llama al método correspondiente para leer los datos y cargarlos en una lista de alumnos.
Si el formato no es reconocido, se devuelve `null`.

```
if (alumnos == null)

{

    imprimir("Formato no soportado.");

    return;

}
```

Verifica si el formato del archivo no es válido o no pudo ser leído correctamente.
En ese caso, se muestra un mensaje de error y se finaliza el método.

```
            // Obtiene la extensión del archivo y la convierte a minúsculas para procesarlo
            string ext = Path.GetExtension(archivoOrigen).ToLower();

            // Dependiendo de la extensión del archivo, se leerán los datos de diferentes formas
            List<Alumno> alumnos = ext switch
            {
                ".txt" => gestor.LeerTXT(archivoOrigen),  // Lee archivo en formato TXT
                ".csv" => gestor.LeerCSV(archivoOrigen),  // Lee archivo en formato CSV
                ".json" => gestor.LeerJSON(archivoOrigen), // Lee archivo en formato JSON
                ".xml" => gestor.LeerXML(archivoOrigen),   // Lee archivo en formato XML
                _ => null  // Si el archivo no tiene una extensión válida, retorna null
            };

            // Verifica si los datos no fueron leídos correctamente o si el formato no es soportado
            if (alumnos == null)
            {
                imprimir("Formato no soportado.");
                return; // Finaliza el método si el formato no es soportado
            }

            // Solicita al usuario que ingrese el formato de destino para la conversión
            string destino = Input("Formato destino (TXT, CSV, JSON, XML):")
                             .Trim()
                             .ToUpper();

            // Verifica que el formato destino sea válido
            if (destino != "TXT" && destino != "CSV" && destino != "JSON" && destino != "XML")
            {
                imprimir("Formato destino inválido.");
                return; // Finaliza el método si el formato de destino es inválido
            }

            // Obtiene el nombre base del archivo (sin la extensión) para crear el archivo destino
            string nombreBase = Path.GetFileNameWithoutExtension(archivoOrigen);
            // Crea el nombre del archivo destino, agregando la nueva extensión
            string archivoDestino = nombreBase + "." + destino.ToLower();

            // Utiliza el método GuardarArchivo del GestorArchivos para guardar el archivo convertido
            gestor.GuardarArchivo(archivoDestino, alumnos, destino);

            // Informa al usuario que la conversión fue exitosa
            imprimir($"Conversión realizada con éxito. Archivo generado: {archivoDestino}");
        }
        catch (Exception ex)
        {
            // Si ocurre una excepción, imprime el mensaje de error
            imprimir("Error durante la conversión:");
            imprimir(ex.Message);
        }
```

`string ext = Path.GetExtension(archivoOrigen).ToLower();`

Obtiene la extensión del archivo de origen y la convierte a minúsculas para facilitar la comparación y evitar errores por diferencias de mayúsculas.

```
List<Alumno> alumnos = ext switch

{

   ".txt" => gestor.LeerTXT(archivoOrigen),

   ".csv" => gestor.LeerCSV(archivoOrigen),

   ".json" => gestor.LeerJSON(archivoOrigen),

   ".xml" => gestor.LeerXML(archivoOrigen),

   _ => null

};
```

Según la extensión del archivo, se invoca el método correspondiente del GestorArchivos para leer los datos y cargarlos en una lista de alumnos.

Si la extensión no coincide con ningún formato soportado, se devuelve null.

```
if (alumnos == null)

{

   imprimir("Formato no soportado.");

   return;

}
```

Verifica si la lectura del archivo fue exitosa.

Si el formato no es compatible o no se pudo leer correctamente, se informa al usuario y se finaliza el método.

```
string destino = Input("Formato destino (TXT, CSV, JSON, XML):")

                .Trim()

                .ToUpper();
```

Solicita al usuario el formato de destino para la conversión y normaliza el valor ingresado eliminando espacios y convirtiéndolo a mayúsculas.

```
if (destino != "TXT" && destino != "CSV" && destino != "JSON" && destino != "XML")
```

{

imprimir("Formato destino inválido.");

return;

}

Valida que el formato de destino ingresado sea uno de los formatos soportados.

Si no lo es, se muestra un mensaje de error y se cancela la conversión.

string nombreBase = Path.GetFileNameWithoutExtension(archivoOrigen);

Obtiene el nombre del archivo de origen sin su extensión, para reutilizarlo al crear el archivo convertido.

string archivoDestino = nombreBase + "." + destino.ToLower();

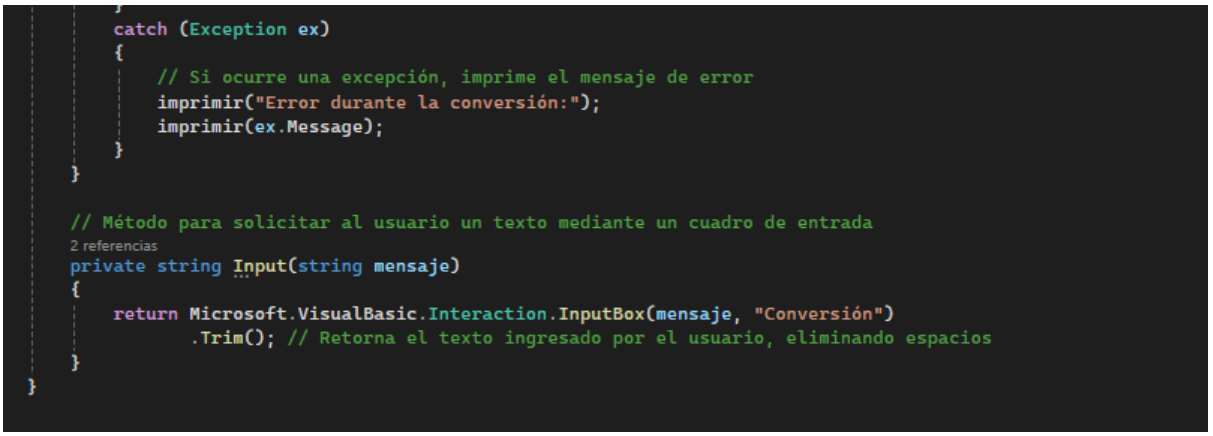
Construye el nombre del archivo de destino agregando la nueva extensión correspondiente al formato elegido.

gestor.GuardarArchivo(archivoDestino, alumnos, destino);

Llama al método GuardarArchivo del GestorArchivos para guardar los datos en el nuevo formato seleccionado.

imprimir($"Conversión realizada con éxito. Archivo generado: {archivoDestino}");

Informa al usuario que la conversión se realizó correctamente y muestra el nombre del archivo generado.



```
        catch (Exception ex)
        {
            // Si ocurre una excepción, imprime el mensaje de error
            imprimir("Error durante la conversión:");
            imprimir(ex.Message);
        }
    }

    // Método para solicitar al usuario un texto mediante un cuadro de entrada
    2 referencias
    private string Input(string mensaje)
    {
        return Microsoft.VisualBasic.Interaction.InputBox(mensaje, "Conversión")
               .Trim(); // Retorna el texto ingresado por el usuario, eliminando espacios
    }
}
```

catch (Exception ex)

{

imprimir("Error durante la conversión:");

```
    imprimir(ex.Message);
}
```

Este bloque catch captura cualquier excepción que pueda ocurrir durante el proceso de conversión del archivo, como errores de lectura, escritura o formatos incorrectos.

Cuando ocurre un error, se informa al usuario mostrando un mensaje general y luego el detalle específico del error producido por el sistema.

Esto evita que la aplicación se cierre inesperadamente y mejora la experiencia del usuario al brindar información clara sobre el problema.

Método de entrada de datos del usuario

```
private string Input(string mensaje)
{
    return Microsoft.VisualBasic.Interaction.InputBox(mensaje, "Conversión")
            .Trim();
}
```

Este método se utiliza para solicitar datos al usuario mediante un cuadro de diálogo.

- Recibe como parámetro el mensaje que se mostrará en pantalla.
- Utiliza un InputBox para permitir el ingreso de texto.
- Aplica Trim() al valor ingresado para eliminar espacios en blanco al inicio y al final.
- El método centraliza la entrada de datos, facilitando su reutilización y manteniendo el código ordenado.

4.6- Generar reporte

```
1 referencia
public void GenerarReporte()
{
    // Mensaje inicial para informar que se está generando el reporte
    imprimir("=== GENERAR REPORTE ===");

    // Solicita al usuario el nombre del archivo fuente
    string archivo = Input("Ingrese el archivo fuente con extensión:");

    // Verifica si el nombre del archivo es inválido (vacío o nulo)
    if (string.IsNullOrWhiteSpace(archivo))
    {
        imprimir("Nombre inválido.");
        return; // Finaliza el método si el nombre es inválido
    }

    // Verifica si el archivo existe en la ruta indicada
    if (!File.Exists(archivo))
    {
        imprimir($"El archivo '{archivo}' no existe.");
        return; // Finaliza el método si el archivo no existe
    }

    // Obtiene la extensión del archivo y la convierte a minúsculas
    string ext = Path.GetExtension(archivo).ToLower();

    // Dependiendo de la extensión del archivo, se leen los datos de una forma u otra
    List<Alumno> alumnos = ext switch
    {
        ".txt" => gestor.LeerTXT(archivo),   // Llama al método LeerTXT si es un archivo TXT
        ".csv" => gestor.LeerCSV(archivo),   // Llama al método LeerCSV si es un archivo CSV
        ".json" => gestor.LeerJSON(archivo), // Llama al método LeerJSON si es un archivo JSON
        ".xml" => gestor.LeerXML(archivo),   // Llama al método LeerXML si es un archivo XML
        _ => null // Si no es ninguno de esos formatos, retorna null
    };

    // Verifica si no se pudieron leer alumnos o si el archivo está vacío
    if (alumnos == null || alumnos.Count == 0)
    {
        imprimir("El archivo no contiene alumnos o el formato no es soportado.");
        return; // Finaliza el método si no se obtienen alumnos
    }

    // Agrupa los alumnos por apellido, y los ordena alfabéticamente por apellido y nombre
    var grupos = alumnos
        .OrderBy(a => a.Apellido)   // Ordena por apellido
        .ThenBy(a => a.Nombres)     // Luego ordena por nombre
        .GroupBy(a => a.Apellido);  // Agrupa por apellido
```

Este bloque solicita al usuario el archivo desde el cual se generará el reporte y realiza todas las validaciones necesarias antes de continuar:

- Se verifica que el nombre del archivo no esté vacío.
- Se comprueba que el archivo exista físicamente.
- Se obtiene la extensión del archivo para determinar el formato.
- Según la extensión, se utiliza el método correspondiente para leer los alumnos.
- Si el archivo no contiene datos o el formato no es compatible, el proceso se detiene.

Esto garantiza que el reporte solo se genere cuando los datos son válidos y accesibles.

Ordenamiento y agrupación de los datos

```
var grupos = alumnos
```

```
.OrderBy(a => a.Apellido)

.ThenBy(a => a.Nombres)

.GroupBy(a => a.Apellido);
```

En este bloque se procesan los alumnos leídos del archivo para preparar el reporte:

- Primero se ordenan los alumnos alfabéticamente por apellido.
- Luego se ordenan por nombre dentro del mismo apellido.
- Finalmente, se agrupan por apellido.

Este procesamiento permite generar un reporte organizado y fácil de leer, donde los alumnos aparecen agrupados por apellido y ordenados correctamente dentro de cada grupo.

```
// StringBuilder para ir construyendo el reporte en formato texto
StringBuilder sb = new StringBuilder();

// Títulos del reporte
sb.AppendLine("=============================================");
sb.AppendLine("      REPORTE DE ALUMNOS POR APELLIDO");
sb.AppendLine($"      Fecha: {DateTime.Now}");  // Muestra la fecha actual
sb.AppendLine("=============================================");
sb.AppendLine(); // Línea en blanco

int totalGeneral = 0; // Contador para el total de alumnos

// Itera a través de los grupos de alumnos
foreach (var grupo in grupos)
{
    // Agrega el apellido del grupo al reporte
    sb.AppendLine($"APELLIDO: {grupo.Key}");
    sb.AppendLine("---------------------------------------------");  // Línea de separación

    // Itera a través de los alumnos de cada grupo
    foreach (var a in grupo)
    {
        // Agrega los detalles del alumno al reporte
        sb.AppendLine($"Legajo: {a.Legajo}");
        sb.AppendLine($"Nombre: {a.Nombres}");
        sb.AppendLine($"Documento: {a.NumeroDocumento}");
        sb.AppendLine($"Email: {a.Email}");
        sb.AppendLine($"Teléfono: {a.Telefono}");
        sb.AppendLine(); // Línea en blanco después de los datos del alumno
    }

    // Agrega el subtotal de alumnos en ese grupo (por apellido)
    sb.AppendLine($"Subtotal {grupo.Key}: {grupo.Count()} alumno(s)");
    sb.AppendLine(); // Línea en blanco después del subtotal

    totalGeneral += grupo.Count(); // Suma la cantidad de alumnos al total general
}

// Agrega la sección final del reporte con el total de alumnos
sb.AppendLine("=============================================");
sb.AppendLine($"Total de alumnos: {totalGeneral}");
sb.AppendLine("=============================================");

// Imprime el reporte en la consola
imprimir(sb.ToString());

// Pregunta al usuario si desea guardar el reporte
string guardar = Input("¿Guardar reporte en TXT? (S/N):").ToUpper();
```

`StringBuilder sb = new StringBuilder();`

Se utiliza un StringBuilder para construir el contenido del reporte en formato texto de manera eficiente, evitando concatenaciones innecesarias de strings.

Encabezado del reporte

```
sb.AppendLine("================================================");

sb.AppendLine("      REPORTE DE ALUMNOS POR APELLIDO");

sb.AppendLine($"      Fecha: {DateTime.Now}");

sb.AppendLine("================================================");

sb.AppendLine();
```

En este bloque se arma el encabezado del reporte:

- Se imprime el título del reporte.
- Se muestra la fecha y hora actual de generación.
- Se agregan líneas separadoras para mejorar la presentación.
- Se deja una línea en blanco antes de comenzar con los datos.

Inicialización del contador general

```
int totalGeneral = 0;
```

Se declara una variable que se utilizará para contabilizar el total de alumnos incluidos en el reporte.

Recorrido de los grupos de alumnos

```
foreach (var grupo in grupos)
```

- Se recorren los grupos de alumnos, donde cada grupo corresponde a un apellido distinto.
- Información del grupo (apellido)

```
sb.AppendLine($"APELLIDO: {grupo.Key}");

sb.AppendLine("------------------------------------------");
```

Para cada grupo se agrega al reporte:

- El apellido del grupo.
- Una línea separadora para distinguir visualmente cada sección.
- Detalle de cada alumno

foreach (var a in grupo)

Dentro de cada grupo se recorren los alumnos y se agregan sus datos individuales:

- Legajo
- Nombre
- Documento
- Email
- Teléfono

Cada alumno se separa con una línea en blanco para facilitar la lectura.

Subtotal por apellido

sb.AppendLine($"Subtotal {grupo.Key}: {grupo.Count()} alumno(s)");

Se agrega un subtotal indicando cuántos alumnos pertenecen a ese apellido.

Acumulación del total general

totalGeneral += grupo.Count();

Se suma la cantidad de alumnos del grupo al total general.

Cierre del reporte

sb.AppendLine("==");

sb.AppendLine($"Total de alumnos: {totalGeneral}");

sb.AppendLine("==");

Se agrega la sección final del reporte con el total de alumnos procesados.

Visualización del reporte

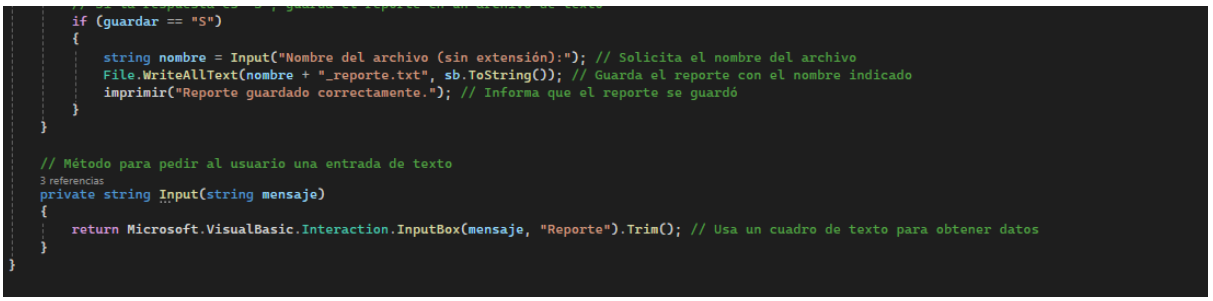
imprimir(sb.ToString());

El contenido completo del reporte se muestra en el TextBox multiline del formulario, simulando una consola.

Opción de guardado

string guardar = Input("¿Guardar reporte en TXT? (S/N):").ToUpper();

Se le pregunta al usuario si desea guardar el reporte en un archivo de texto, dejando la decisión en manos del usuario.



```
        if (guardar == "S")
        {
            string nombre = Input("Nombre del archivo (sin extensión):"); // Solicita el nombre del archivo
            File.WriteAllText(nombre + "_reporte.txt", sb.ToString()); // Guarda el reporte con el nombre indicado
            imprimir("Reporte guardado correctamente."); // Informa que el reporte se guardó
        }
    }

    // Método para pedir al usuario una entrada de texto
    3 referencias
    private string Input(string mensaje)
    {
        return Microsoft.VisualBasic.Interaction.InputBox(mensaje, "Reporte").Trim(); // Usa un cuadro de texto para obtener datos
    }
}
```

if (guardar == "S")

Se verifica si el usuario respondió "S" (sí) cuando se le preguntó si deseaba guardar el reporte.

string nombre = Input("Nombre del archivo (sin extensión):");

Se solicita al usuario el nombre base del archivo donde se guardará el reporte, sin incluir la extensión.

File.WriteAllText(nombre + "_reporte.txt", sb.ToString());

Se crea un archivo de texto con el nombre ingresado por el usuario seguido de _reporte.txt.

El contenido del archivo será el texto completo del reporte generado previamente con StringBuilder.

imprimir("Reporte guardado correctamente.");

Se informa al usuario que el reporte fue guardado con éxito.

Método Input del Generador de Reportes

private string Input(string mensaje)

{

return Microsoft.VisualBasic.Interaction.InputBox(mensaje, "Reporte").Trim();

}

Este método se utiliza para:

- Mostrar un cuadro de diálogo solicitando datos al usuario.
- Obtener el texto ingresado.
- Eliminar espacios en blanco al inicio y al final del texto.

Se usa exclusivamente dentro de la clase GeneradorReportes para solicitar información relacionada con la generación del reporte.

5. Casos de error comunes y soluciones

A continuación se detallan los errores más frecuentes que pueden presentarse durante el uso del sistema, junto con sus posibles causas y soluciones.

5.1 Nombre de archivo inválido

Descripción:
El sistema muestra un mensaje indicando que el nombre del archivo es inválido.

Causa:
El usuario no ingresó ningún nombre o ingresó solo espacios en blanco.

Solución:
Ingresar un nombre válido para el archivo, sin dejar el campo vacío.

5.2 Archivo inexistente

Descripción:
El sistema informa que el archivo no existe.

Causa:
El nombre del archivo ingresado no coincide con ningún archivo existente en el directorio del programa.

Solución:
Verificar que el archivo exista y que el nombre y la extensión sean correctos (por ejemplo: .txt, .csv, .json, .xml).

5.3 Formato de archivo no soportado

Descripción:
Se muestra un mensaje indicando que el formato no es válido o no está soportado.

Causa:
Se ingresó un archivo con una extensión distinta a las permitidas.

Solución:
Utilizar únicamente los formatos admitidos por el sistema: TXT, CSV, JSON o XML.

5.4 Cantidad de alumnos inválida

Descripción:
El sistema indica que la cantidad de alumnos ingresada es incorrecta.

Causa:
Se ingresó un valor no numérico o un número menor o igual a cero.

Solución:
Ingresar un número entero positivo mayor a cero.

5.5 Email inválido

Descripción:
El sistema muestra un mensaje de error indicando que el email es inválido.

Causa:
El correo electrónico ingresado no contiene el carácter @ o un dominio válido.

Solución:
Ingresar un email con formato correcto, por ejemplo: usuario@dominio.com.

5.6 Legajo duplicado

Descripción:
El sistema informa que el legajo ya existe.

Causa:
Se intenta agregar un alumno con un legajo que ya está registrado en el archivo.

Solución:
Ingresar un legajo diferente y único para cada alumno.

5.7 Cancelación de operaciones

Descripción: Los cambios no se guardan.

Causa:
El usuario eligió la opción de cancelar o no confirmó la acción solicitada.

Solución:
Seleccionar la opción de guardar cambios o confirmar la operación cuando el sistema lo solicite.

Conclusión

El sistema **GestorAlumnos** permite gestionar de forma clara y ordenada la información de alumnos mediante una aplicación de escritorio desarrollada en **Windows Forms con .NET 8**.

A lo largo del proyecto se implementaron funcionalidades completas para:

- Crear, leer, modificar y eliminar archivos.
- Trabajar con múltiples formatos de almacenamiento (TXT, CSV, JSON y XML).
- Convertir archivos entre distintos formatos.
- Generar reportes organizados y legibles.
- Validar datos ingresados por el usuario para evitar errores comunes.

La separación del código en distintas clases mejora la organización, el mantenimiento y la escalabilidad del sistema.
La interfaz gráfica basada en `MenuStrip` facilita el uso de la aplicación y permite acceder de manera intuitiva a cada funcionalidad.

En conclusión, el sistema cumple con los requisitos planteados en el enunciado del trabajo práctico y constituye una solución funcional, clara y robusta para la gestión de alumnos.