Estructuras Dinámicas: Listas

Para esta clase, vamos a trabajar con una `List<T>` de objetos. Olvida los tipos simples (solo strings); vamos a crear una clase `Estudiante` para que el material sea de nivel profesional.

Material de Clase: CRUD con Listas en WinForms

1. El Modelo (La Clase)

Primero, definimos qué vamos a guardar. Crea una clase llamada `Estudiante.cs`.

C#
```
public class Estudiante
{
    public int Id { get; set; }
    public string Nombre { get; set; }
    public string Carrera { get; set; }

    // Esto es para que el ListBox o ComboBox muestre el nombre
    public override string ToString()
    {
        return $"{Id} - {Nombre} ({Carrera})";
    }
}
```

---

2. Preparación del Formulario (UI)

Diseña tu `Form1` con los siguientes controles:

- **TextBoxes:** `txtId`, `txtNombre`, `txtCarrera`.
- **Buttons:** `btnAgregar`, `btnConsultar`, `btnEliminar`.
- **ListBox:** `lstEstudiantes` (donde veremos la lista).

---

3. El Código del Formulario (`Form1.cs`)

Aquí es donde ocurre la magia. Declaramos la lista a nivel de clase para que persista mientras la ventana esté abierta.

C#
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace MiAppWinForms
{
    public partial class Form1 : Form
    {
        // LA LISTA (Nuestra "Base de Datos" en memoria)
        private List<Estudiante> listaEstudiantes = new List<Estudiante>();
```

```
        public Form1()
        {
            InitializeComponent();
        }

        // --- C: CREATE (Agregar) ---
        private void btnAgregar_Click(object sender, EventArgs e)
        {
            var nuevo = new Estudiante {
                Id = int.Parse(txtId.Text),
                Nombre = txtNombre.Text,
                Carrera = txtCarrera.Text
            };

            listaEstudiantes.Add(nuevo);
            ActualizarInterfaz();
            LimpiarCampos();
        }

        // --- R: READ (Consultar) ---
        private void btnConsultar_Click(object sender, EventArgs e)
        {
            // Buscamos por ID usando LINQ
            int idBusqueda = int.Parse(txtId.Text);
            var encontrado = listaEstudiantes.FirstOrDefault(est => est.Id
== idBusqueda);

            if (encontrado != null) {
                MessageBox.Show($"Encontrado: {encontrado.Nombre} de
{encontrado.Carrera}");
            } else {
                MessageBox.Show("Estudiante no encontrado.");
            }
        }

        // --- D: DELETE (Eliminar) ---
        private void btnEliminar_Click(object sender, EventArgs e)
        {
            if (lstEstudiantes.SelectedItem != null)
            {
                // Obtenemos el objeto seleccionado en el ListBox
                Estudiante seleccionado =
(Estudiante)lstEstudiantes.SelectedItem;

                listaEstudiantes.Remove(seleccionado);
                ActualizarInterfaz();
            }
        }

        // --- MÉTODOS DE APOYO ---
        private void ActualizarInterfaz()
        {
            // Refrescar el ListBox
            lstEstudiantes.DataSource = null; // Truco para forzar el
refresco
            lstEstudiantes.DataSource = listaEstudiantes;
        }

        private void LimpiarCampos()
        {
            txtId.Clear();
```

```
            txtNombre.Clear();
            txtCarrera.Clear();
            txtId.Focus();
        }
    }
}
```

---

4. Puntos clave para explicar en la clase (Tips de Programador)

1. **Instanciación:** Explica que si la `List<Estudiante>` se declara dentro de un botón, se borrará cada vez que el botón termine su ejecución. Por eso debe ser un **campo de la clase** (private).
2. **LINQ (`FirstOrDefault`):** Es la forma moderna de buscar en C#. Explica que devuelve el primer elemento que cumpla la condición o `null` si no existe.
3. **DataSource:** El `ListBox` puede "amarrarse" a la lista directamente. El "truco" de ponerlo en `null` antes es necesario en WinForms clásico para que detecte cambios en el contenido de la lista.
4. **Casteo:** Cuando recuperamos algo del `ListBox`, viene como tipo `object`. Debemos decirle a C#: *"Tranquilo, yo sé que esto es un Estudiante"* usando `(Estudiante)lstEstudiantes.SelectedItem`.
5. ---

Resumen Visual para los alumnos

Operación	Método de List<T>	Descripción
Crear	`.Add(objeto)`	Inserta el objeto al final de la lista.
Consultar	`.FirstOrDefault(x => ...)`	Busca un elemento específico.
Eliminar	`.Remove(objeto)`	Quita la referencia del objeto de la lista.
Limpiar	`.Clear()`	Vacía la lista por completo.

El **Update** (Modificar) es el paso que separa a un principiante de un programador con lógica sólida. Para modificar un objeto en una lista en memoria, el flujo mental es: **Buscar → Reemplazar → Refrescar.**

Material para completar el CRUD en tu clase de WinForms.

---

1. El Concepto: ¿Cómo modificamos en una lista?

Hay dos formas de hacerlo en C#:

1. **Por Referencia:** Buscas el objeto y cambias sus propiedades. Como la lista guarda la "dirección" del objeto, los cambios se reflejan automáticamente.
2. **Por Índice:** Buscas en qué posición (`index`) está el objeto y lo reemplazas por uno nuevo.

Usaremos la **opción 1** por ser la más elegante y eficiente en .NET.

---

2. El Código del Botón Modificar (`btnModificar_Click`)

Añade este botón a tu formulario. La lógica asume que el usuario escribió el **ID** del estudiante que quiere cambiar y los **Nuevos Datos** en los campos de texto.

C#

```
private void btnModificar_Click(object sender, EventArgs e)
{
    // 1. Validar que el ID no esté vacío
    if (string.IsNullOrEmpty(txtId.Text)) {
        MessageBox.Show("Por favor, ingrese el ID del estudiante a
modificar.");
        return;
    }

    int idBusqueda = int.Parse(txtId.Text);

    // 2. BUSCAR el objeto original en la lista usando LINQ
    var estudianteOriginal = listaEstudiantes.FirstOrDefault(est => est.Id
== idBusqueda);

    if (estudianteOriginal != null)
    {
        // 3. ACTUALIZAR las propiedades del objeto encontrado
        estudianteOriginal.Nombre = txtNombre.Text;
        estudianteOriginal.Carrera = txtCarrera.Text;

        // 4. REFRESCAR la interfaz
        ActualizarInterfaz();
        LimpiarCampos();
        MessageBox.Show("Estudiante actualizado correctamente.");
    }
    else
    {
        MessageBox.Show("No se encontró un estudiante con ese ID.");
    }
}
```

---

3. Mejora Visual: "Cargar datos al hacer clic"

Para que la clase sea realmente profesional, enseña a tus alumnos cómo "cargar" los datos en los TextBoxes simplemente haciendo clic en la lista. Esto evita errores de escritura.

Crea el evento **`SelectedIndexChanged`** del `ListBox`:
C#
```
private void lstEstudiantes_SelectedIndexChanged(object sender, EventArgs e)
{
    if (lstEstudiantes.SelectedItem != null)
    {
        // Casteamos el objeto seleccionado
        Estudiante seleccionado = (Estudiante)lstEstudiantes.SelectedItem;

        // Pasamos los datos a los cuadros de texto
        txtId.Text = seleccionado.Id.ToString();
        txtNombre.Text = seleccionado.Nombre;
        txtCarrera.Text = seleccionado.Carrera;

        // Bloqueamos el ID para que no lo cambien durante la edición
        txtId.Enabled = false;
    }
}
```

Tip para la clase: Explica que el **ID** suele ser la "Llave Primaria". Si el usuario cambia el ID, ya no estamos modificando al mismo estudiante, sino creando un conflicto. Por eso es buena práctica deshabilitar `txtId` al editar.

---

4. El Resumen del CRUD Completo (Cheat Sheet para alumnos)

Acción	Lógica en C#
Create	`lista.Add(nuevoObjeto);`
Read	`lista.FirstOrDefault(x => x.Id == id);`
Update	`objetoEncontrado.Propiedad = nuevoValor;`
Delete	`lista.Remove(objeto);`

¿Qué sigue para tu clase?

Con esto ya tienes un sistema funcional en memoria. Si quieres subir el nivel un escalón más, podríamos añadir una **Validación** para que no se puedan agregar dos estudiantes con el mismo ID (`lista.Any(x => x.Id == nuevoId)`).

Para que esta clase sea realmente útil, lo ideal es ver **ambas**, pero en un orden lógico. Primero aseguramos que los datos sean correctos (**Validación**) y luego los hacemos permanentes (**Persistencia**).

Aquí tienes el material para estas dos funciones vitales en C#.

---

1. Validación de Duplicados (Evitar IDs repetidos)

Antes de hacer el `listaEstudiantes.Add()`, debemos preguntar si ya existe alguien con ese mismo ID. Usaremos el método **`.Any()`** de LINQ, que devuelve `true` o `false`.

Modifica tu `btnAgregar_Click` así:

C#
```
private void btnAgregar_Click(object sender, EventArgs e)
{
    int idIngresado = int.Parse(txtId.Text);

    // VALIDACIÓN: ¿Existe ya este ID?
    if (listaEstudiantes.Any(est => est.Id == idIngresado))
    {
        MessageBox.Show("Error: Ya existe un estudiante con el ID " +
idIngresado);
        return; // Salimos del método para no agregar el duplicado
    }

    // Si no es duplicado, procedemos
    var nuevo = new Estudiante {
        Id = idIngresado,
        Nombre = txtNombre.Text,
        Carrera = txtCarrera.Text
    };

    listaEstudiantes.Add(nuevo);
    ActualizarInterfaz();
    LimpiarCampos();
}
```

---

2. Persistencia: Guardar en un Archivo de Texto

En .NET, la forma más profesional y sencilla de guardar una lista de objetos en un archivo hoy en día es usando **JSON**. Esto convierte tu lista en un texto estructurado que cualquier programa puede leer.

Necesitaremos importar: `using System.IO;` y `using System.Text.Json;` (disponible en .NET Core / .NET 5+).

Función para GUARDAR (Al cerrar o al agregar)

C#
```
private void GuardarDatos()
{
    // Convertimos la lista a una cadena de texto JSON
    string jsonString = JsonSerializer.Serialize(listaEstudiantes);

    // Lo guardamos en un archivo llamado "datos.json"
    File.WriteAllText("datos.json", jsonString);
}
```

Función para CARGAR (Al abrir el programa)

Llama a esta función en el **Constructor** o en el evento `Load` del formulario.

C#

```
private void CargarDatos()
{
    if (File.Exists("datos.json"))
    {
        string jsonString = File.ReadAllText("datos.json");

        // Convertimos el texto de vuelta a nuestra lista de objetos
        listaEstudiantes =
JsonSerializer.Deserialize<List<Estudiante>>(jsonString);

        ActualizarInterfaz();
    }
}
```

---

3. El Toque Final: Automatización

Para que el usuario no tenga que presionar un botón de "Guardar", puedes llamar a `GuardarDatos()` al final de cada operación (Agregar, Editar, Eliminar).

Resumen de Conceptos para la Clase:

1. **`Any()` vs `FirstOrDefault()`:** * `Any()` se usa cuando solo quieres saber si algo existe (devuelve un Sí/No).
   - `FirstOrDefault()` se usa cuando necesitas trabajar con el objeto encontrado.
2. **Serialización:** Explica que es el proceso de convertir un objeto vivo en memoria a un formato de transporte (como texto JSON).
3. **Deserialización:** Es el proceso inverso: reconstruir el objeto a partir del texto.

Tarea sugerida para los alumnos:

"Modifiquen el programa para que, al intentar eliminar un estudiante, el sistema pida una confirmación con un `MessageBox.Show` con botones de 'Sí' y 'No' antes de borrar los datos del archivo."

En el desarrollo profesional, **nunca** se debe eliminar un dato sin confirmar, porque los errores de clic son humanos.

En **Windows Forms**, esto se logra evaluando el resultado de un `MessageBox`, que devuelve un tipo de dato llamado `DialogResult`.

---

1. El Código: Botón Eliminar con Confirmación

Sustituye tu código anterior del botón eliminar por este. Es una estructura que todo alumno de .NET debe memorizar:

C#
```
private void btnEliminar_Click(object sender, EventArgs e)
{
    // 1. Verificar si hay algo seleccionado
    if (lstEstudiantes.SelectedItem == null)
    {
        MessageBox.Show("Por favor, seleccione un estudiante de la lista.");
        return;
    }

    // 2. Obtener el objeto seleccionado
    Estudiante seleccionado = (Estudiante)lstEstudiantes.SelectedItem;

    // 3. Mostrar el cuadro de diálogo de confirmación
    // El orden es: Mensaje, Título, Botones, Ícono
    DialogResult respuesta = MessageBox.Show(
        $"¿Está seguro de que desea eliminar a {seleccionado.Nombre}?",
        "Confirmar Eliminación",
        MessageBoxButtons.YesNo,
        MessageBoxIcon.Warning
    );

    // 4. Evaluar la respuesta del usuario
    if (respuesta == DialogResult.Yes)
    {
        // Procedemos a borrar
        listaEstudiantes.Remove(seleccionado);

        // OPCIONAL: Guardar automáticamente en el archivo si implementaste
JSON
        // GuardarDatos();

        ActualizarInterfaz();
        LimpiarCampos();

        MessageBox.Show("Registro eliminado correctamente.");
    }
    // Si la respuesta es 'No', no hacemos nada y el registro se salva
}
```

---

2. Puntos Clave para la Clase (Explicación técnica)

Para que tus alumnos entiendan qué está pasando "detrás de cámaras":

- **`DialogResult`**: Explica que es un tipo de enumeración (`Enum`). No devuelve un string "Sí", devuelve un valor numérico interno que C# entiende como "Yes".
- **El Ícono (`MessageBoxIcon`)**: Es un detalle de experiencia de usuario (UX). Usar `Warning` (Triángulo amarillo) o `Question` ayuda al usuario a entender la gravedad de la acción.
- **El flujo de control**: Nota que si el usuario presiona "No", el código simplemente termina el método y no entra al bloque `if`, por lo que el `Remove` nunca ocurre.

---

3. Resumen final del Material Didáctico

Aquí tienes el "Checklist" de lo que tus alumnos habrán aprendido con este proyecto:

1. **POO (Programación Orientada a Objetos):** Creación y uso de clases (`Estudiante`).
2. **Manejo de Colecciones:** Uso de `List<T>` para gestionar datos en memoria.
3. **LINQ:** Búsqueda eficiente con `FirstOrDefault` y validación con `Any`.
4. **Interfaz de Usuario (UI):** Enlace de datos (`DataSource`) y eventos de control.
5. **Persistencia:** Guardar y cargar datos en archivos físicos (JSON).
6. **Seguridad y UX:** Confirmación de acciones críticas.

Guía de Supervivencia: Errores Comunes en C# (WinForms & Listas)

Esta guía resume los fallos más frecuentes al construir un CRUD en memoria.

Error 1: La lista que "se olvida" de todo (`NullReferenceException`)

El error: Declarar la lista pero no instanciarla.
C#
```
private List<Estudiante> listaEstudiantes; // ✕ Error: Solo declarada
```

Resultado: El programa truena al intentar usar `.Add()`.

La solución: Usar siempre el `new`.
C#
```
private List<Estudiante> listaEstudiantes = new List<Estudiante>(); // ✓ Correcto
```

Error 2: El objeto "Fantasma" (Casteo Inválido)

El error: Intentar usar el objeto seleccionado del ListBox sin decirle a C# qué es.
C#
```
var seleccionado = lstEstudiantes.SelectedItem;
string nombre = seleccionado.Nombre; // ✕ Error: 'object' no tiene la propiedad 'Nombre'
```

La solución: Hacer un **Casteo Explícito**.
C#
```
Estudiante seleccionado = (Estudiante)lstEstudiantes.SelectedItem; // ✓ Correcto
```

Error 3: La interfaz "Congelada" (No se ven los cambios)

El error: Agregar un elemento a la lista y esperar que el `ListBox` se entere solo.

La solución: Forzar el refresco del `DataSource`. En WinForms, el control no "vigila" la lista, debes avisarle:

C#
```
lstEstudiantes.DataSource = null;           // Limpiamos el vínculo
lstEstudiantes.DataSource = listaEstudiantes; // Lo volvemos a vincular
```

Error 4: Confundir `Any()` con `FirstOrDefault()`

El error: Usar el método equivocado para buscar datos.

- **`Any()`**: Devuelve un `bool` (True/False). Úsalo solo para **Validar duplicados**.
- **`FirstOrDefault()`**: Devuelve el **Objeto** encontrado. Úsalo para **Consultar o Modificar**.

Error 5: Tipos de datos en los TextBox

El error: Intentar guardar un texto en un campo numérico sin convertirlo.
C#
```
nuevoEstudiante.Id = txtId.Text; // ✕ Error: No se puede convertir string a int
```

La solución: Usar `int.Parse()` o `Convert.ToInt32()`.
C#
```
nuevoEstudiante.Id = int.Parse(txtId.Text); // ✓ Correcto
```

El "Gran Olvido": El `using` de LINQ

Si intentas usar `.Any()` o `.FirstOrDefault()` y Visual Studio te marca un error en rojo, es porque te falta la directiva en la parte superior del archivo:
C#
```
using System.Linq; // ☜ Sin esto, no hay magia de búsqueda
```

Consejo Pro: El método `ToString()`

Si tu ListBox muestra algo raro como `ProyectoEscuela.Models.Estudiante` en lugar del nombre, es porque olvidaste hacer el **Override** del método `ToString()` en tu clase. C# por defecto imprime el nombre de la clase, no el contenido.

Resumen del Flujo de Trabajo (Para el examen)

1. **Capturar:** Leer de los TextBox.
2. **Validar:** ¿Está vacío? ¿Es un ID repetido? (`Any`).
3. **Procesar:** Agregar, buscar o eliminar de la `List<T>`.
4. **Refrescar:** Actualizar el `DataSource` del ListBox.
5. **Persistir:** Guardar en el archivo JSON.

Manejo de Excepciones

Es lo que diferencia un programa "de juguete" de uno profesional. En el mundo real, los usuarios cometen errores: escriben letras donde van números, dejan campos vacíos o borran archivos necesarios.

Sección Técnica: Blindando el Código con Try-Catch

El manejo de excepciones evita que el programa se "cierre solo" (crash) cuando ocurre un error inesperado. En C#, usamos el bloque `try { ... } catch { ... }`.

1. El Error más común: `FormatException`

Ocurre cuando intentas convertir un texto como "Hola" en un número entero (`int.Parse`).

Código Vulnerable:

C#
```
int id = int.Parse(txtId.Text); // Si el usuario escribe "ABC", el programa muere aquí.
```

Código Blindado:

C#
```
try
{
    int id = int.Parse(txtId.Text);
    // ... resto de la lógica ...
}
catch (FormatException)
{
    MessageBox.Show("Error: El ID debe ser un número entero válido.", "Error de Formato",
                    MessageBoxButtons.OK, MessageBoxIcon.Error);
}
```

---

2. Manejo de Errores en Persistencia (Archivos)

Al trabajar con `File.WriteAllText` o `File.ReadAllText`, pueden ocurrir errores si el archivo está siendo usado por otro programa o si no hay permisos.

Ejemplo Profesional de Guardado:

C#
```
private void GuardarDatos()
{
    try
    {
        string jsonString = JsonSerializer.Serialize(listaEstudiantes);
        File.WriteAllText("datos.json", jsonString);
    }
    catch (IOException ex)
    {
        MessageBox.Show("Error al acceder al archivo: " + ex.Message);
    }
    catch (Exception ex)
    {
        MessageBox.Show("Ocurrió un error inesperado: " + ex.Message);
    }
}
```

Reglas de Oro para los Alumnos (Tips de Clase)

- **No captures todo con `Exception`:** Es mejor capturar excepciones específicas (como `FormatException` o `FileNotFoundException`) antes que la genérica. Es como ir al médico: prefieres que te digan exactamente qué tienes a que te digan "estás enfermo".
- **El bloque `Finally` (Opcional):** Existe un tercer bloque llamado `finally` que se ejecuta **siempre**, haya habido error o no. Es ideal para cerrar conexiones o limpiar recursos.
- **No uses Try-Catch para todo:** Si puedes evitar el error con un `if`, es mejor. Por ejemplo:
    - **Mal:** Usar `try-catch` para ver si un archivo existe.
    - **Bien:** Usar `if (File.Exists("ruta"))`. El `try-catch` debe ser para errores que tú no puedes controlar fácilmente.

Resumen Visual

Excepción Común	¿Cuándo ocurre?
`FormatException`	El usuario escribió letras en un campo numérico.
`NullReferenceException`	Olvidaste el `new` o intentas usar un objeto que no existe.
`FileNotFoundException`	Intentaste cargar el JSON pero el archivo no existe.
`IndexOutOfRangeException`	Intentaste acceder a una posición de la lista que no existe.

Evaluación

Laboratorio de Errores: "El Sistema Quebrado"

Instrucciones: El siguiente código tiene **5 errores críticos** que impiden que el programa funcione correctamente o que causan que "truene" (crash). Tu misión es encontrarlos, marcarlos y escribir la corrección.

El Código "Roto" (C# WinForms)

C#

```
public partial class FormEstudiantes : Form
{
    // Error 1: _____________________________
    private List<Estudiante> lista;

    private void btnAgregar_Click(object sender, EventArgs e)
    {
        // Error 2: _____________________________
        int id = txtId.Text;

        Estudiante nuevo = new Estudiante();
        nuevo.Id = id;
        nuevo.Nombre = txtNombre.Text;

        lista.Add(nuevo);

        // Error 3: _____________________________
        lstDatos.DataSource = lista;
        MessageBox.Show("Guardado");
    }

    private void btnEliminar_Click(object sender, EventArgs e)
    {
        // Error 4: _____________________________
        Estudiante seleccionado = lstDatos.SelectedItem;

        lista.Remove(seleccionado);
        ActualizarInterfaz();
    }

    private void btnCargar_Click(object sender, EventArgs e)
    {
        // Error 5: _____________________________
        string json = File.ReadAllText("datos.json");
        lista = JsonSerializer.Deserialize<List<Estudiante>>(json);
    }
```

```
}
```

Guía de Respuestas (Solo para el Profesor)

Aquí tienes la explicación de los errores para cuando hagas la puesta en común con la clase:

1. **Falta la Instanciación:** La lista está declarada pero no tiene el `= new List<Estudiante>();`. Esto lanzará un `NullReferenceException` al primer clic en Agregar.
2. **Error de Tipo (Casteo):** No se puede asignar un `string` (`txtId.Text`) directamente a un `int`. Falta el `int.Parse()`.
3. **Refresco de Interfaz:** En WinForms, asignar el `DataSource` no funciona si ya tenía uno asignado. Hay que ponerlo en `null` primero o llamar a un método que refresque.
4. **Error de Objeto (Casteo):** `SelectedItem` devuelve un `object`. Se requiere el casteo manual: `(Estudiante)lstDatos.SelectedItem`.
5. **Falta de Validación de Archivo:** Si el archivo "datos.json" no existe (la primera vez que se corre el programa), `File.ReadAllText` lanzará una excepción y cerrará el programa. Falta un `if (File.Exists(...))`.
6. ---

Reto Plus (Opcional):

Pide a los alumnos que rodeen el código del **Error 2** y el **Error 5** con un bloque **Try-Catch** para que, incluso si fallan, el programa muestre un mensaje amigable en lugar de cerrarse.

✅ Código Corregido: "Sistema de Estudiantes Blindado"

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;
using System.IO;            // Necesario para archivos
using System.Text.Json;     // Necesario para JSON

public partial class FormEstudiantes : Form
{
    // CORRECCIÓN 1: Se instancia la lista con 'new' para evitar
NullReferenceException
    private List<Estudiante> lista = new List<Estudiante>();

    private void btnAgregar_Click(object sender, EventArgs e)
    {
        try
        {
```

```
            // CORRECCIÓN 2: Conversión de string a int necesaria
            int id = int.Parse(txtId.Text);

            // Validación extra: No permitir IDs duplicados
            if (lista.Any(x => x.Id == id)) {
                MessageBox.Show("Este ID ya existe.");
                return;
            }

            Estudiante nuevo = new Estudiante {
                Id = id,
                Nombre = txtNombre.Text
            };

            lista.Add(nuevo);
            ActualizarInterfaz();
        }
        catch (FormatException)
        {
            MessageBox.Show("El ID debe ser un número válido.");
        }
    }

    private void btnEliminar_Click(object sender, EventArgs e)
    {
        // CORRECCIÓN 4: Casteo obligatorio de 'object' a 'Estudiante'
        if (lstDatos.SelectedItem != null)
        {
            Estudiante seleccionado = (Estudiante)lstDatos.SelectedItem;
            lista.Remove(seleccionado);
            ActualizarInterfaz();
        }
    }

    private void btnCargar_Click(object sender, EventArgs e)
    {
        try
        {
            // CORRECCIÓN 5: Validar si el archivo existe antes de leerlo
            if (File.Exists("datos.json"))
            {
                string json = File.ReadAllText("datos.json");
                lista = JsonSerializer.Deserialize<List<Estudiante>>(json);
                ActualizarInterfaz();
            }
            else {
                MessageBox.Show("No hay datos guardados previamente.");
            }
        }
        catch (Exception ex) {
            MessageBox.Show("Error al cargar: " + ex.Message);
        }
    }

    // CORRECCIÓN 3: Método centralizado para refrescar la UI correctamente
    private void ActualizarInterfaz()
    {
        lstDatos.DataSource = null;
        lstDatos.DataSource = lista;
    }
}
```

Notas Pedagógicas para la Clase

1. **El poder de `ActualizarInterfaz()`:** Explica a los alumnos que crear un método dedicado para refrescar el `ListBox` es mejor que repetir `DataSource = null` en cada botón. Es el principio **DRY** (*Don't Repeat Yourself*).
2. **El "Casteo" (Error 4):** Recuérdales que `SelectedItem` es como una caja genérica. El programa sabe que hay "algo" adentro, pero tú debes decirle específicamente que es un `Estudiante` para poder acceder a sus propiedades.
3. **Manejo de Nulos:** Nota que en el botón eliminar agregamos un `if (lstDatos.SelectedItem != null)`. Esto evita que el programa truene si el usuario hace clic en "Eliminar" sin haber seleccionado nada.

Rúbrica de Evaluación: Laboratorio de C# (Escala 0-20)

Criterio de Evaluación	Excelente (4 pts)	Regular (2 pts)	Insuficiente (0 pts)	Puntos
Instanciación y Memoria	Instancia la lista con `new` a nivel de clase. Entiende el ámbito (scope).	Instancia la lista pero dentro de un método (se borra al salir).	La lista permanece nula, causando error al ejecutar.	
Conversión y Tipos	Usa `int.Parse()` y rodea con `try-catch` para evitar cierres por letras.	Usa `int.Parse()` pero el programa "truena" si el usuario escribe texto.	Intenta asignar el `string` al `int` directamente (error de sintaxis).	
Refresco de Interfaz	Implementa un método `ActualizarInterfaz()` limpiando el `DataSource`.	Actualiza la lista pero de forma manual o repetitiva en cada botón.	No logra que el ListBox muestre los cambios realizados.	
Casteo de Objetos	Realiza el casteo `(Estudiante)` y valida que no sea nulo antes de operar.	Realiza el casteo pero el programa falla si no hay nada seleccionado.	No comprende por qué el objeto `SelectedItem` requiere conversión.	
Manejo de Archivos	Implementa `File.Exists()` y `JsonSerializer` de forma correcta y segura.	Logra guardar/leer pero el programa falla si el archivo no existe físicamente.	No logra implementar la persistencia de datos en el JSON.	

📝 Tabla de Conversión (Calificación Final)

- **18 - 20 (Sobresaliente):** Código impecable, maneja excepciones y valida datos de entrada.
- **14 - 17 (Notable):** El sistema funciona bien, pero faltan algunas validaciones de seguridad.
- **10 - 13 (Aprobado):** El CRUD funciona, pero es inestable ante errores del usuario.
- **00 - 09 (Reprobado):** El código tiene errores de sintaxis que impiden la compilación o ejecución básica.

🎓 Pregunta de "Bonus" (Punto Extra)

Para que los alumnos demuestren que realmente entendieron la lógica, puedes lanzarles este reto al final:

"Si yo cambio el nombre de la clase de `Estudiante` a `Alumno`, ¿en cuántas partes del código del formulario tendrías que hacer cambios para que el programa siga funcionando?"

(Respuesta correcta: En la declaración de la Lista, en el casteo del Botón Eliminar y en el Deserialize del archivo).

¿Te gustaría que te ayude a redactar una **Guía de Instalación Rápida** de los paquetes NuGet necesarios (`System.Text.Json`) por si algún alumno tiene problemas con su Visual Studio?