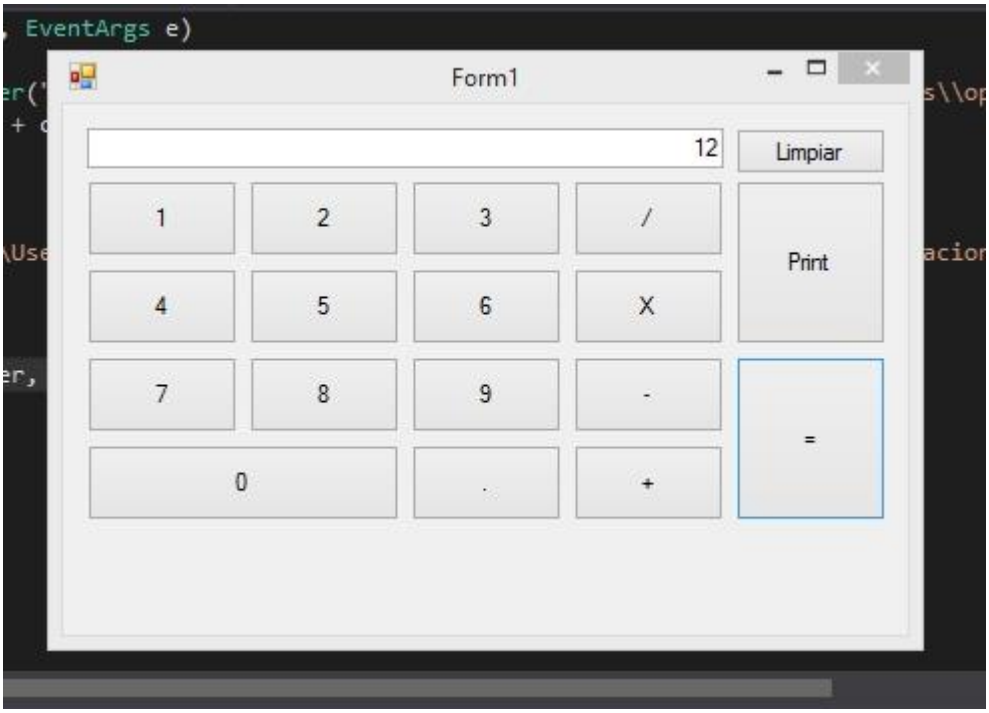
Código de C Sharp - Código C# - Calculadora Simple

Este análisis académico aborda el código fuente proporcionado para una **Calculadora Simple en C#**, estructurando el examen desde la arquitectura del software, la lógica de programación orientada a eventos y la persistencia de datos.

1.



Arquitectura y Estructura del Proyecto

El programa se desarrolla bajo el paradigma de **Programación Orientada a Objetos (POO)** utilizando el framework de .NET para aplicaciones de Windows Forms.

- **Espacio de Nombres y Clase:** El código se organiza dentro del namespace Calculadora. La lógica reside en una clase parcial denominada `Form1`, que hereda de la clase base `Form`.
- **Variables de Estado:** Se definen variables de nivel de clase para gestionar el estado de la aplicación:
  - `double a, b`: Almacenan los operandos numéricos.
  - `string c`: Almacena el operador aritmético seleccionado.

2. Gestión de Eventos y Lógica de Interfaz (UI)

La aplicación emplea un modelo de **programación dirigida por eventos**, donde las acciones del usuario (clics en botones) disparan métodos específicos.

- **Entrada de Datos:** Cada botón numérico (0-9) verifica si el contenedor de texto (`txtpantalla`) está vacío para asignar el valor o concatenarlo al texto existente.
- **Manejo de Punto Decimal:** El método `btnpunto_Click` incluye una validación lógica para evitar múltiples puntos en una misma cifra mediante el uso de la función `.Contains('.')`.
- **Operaciones Aritméticas:** Al seleccionar un operador (+, -, *, /), el programa realiza tres acciones críticas:
    1. Convierte el texto de la pantalla a tipo `double` y lo asigna a la variable `a`.
    2. Registra el símbolo de la operación en la variable `c`.
    3. Limpia la pantalla y devuelve el enfoque (`Focus`) al control para la siguiente entrada.

3. Procesamiento Lógico y Resultados

El núcleo del cálculo se encuentra en el evento `btnigual_Click`.

- **Estructura de Control:** Se utiliza una sentencia `switch (c)` para determinar qué operación ejecutar basándose en el operador guardado previamente.
- **Cálculo:** El segundo valor introducido se asigna a `b`. El resultado se calcula y se convierte nuevamente a `string` para su visualización en `txtpantalla`.

4. Persistencia de Datos y Salida de Información

Una característica distintiva desde el punto de vista académico es la implementación de la clase `StreamWriter` para la **salida de archivos**.

- **Escritura en Archivo:** El método `btnprint_Click` crea o sobrescribe un documento de texto en una ruta específica.
- **Formateo de Salida:** Se construye una cadena que detalla toda la operación: $a + c + b = resultado$.
- **Ejecución de Proceso:** Utiliza `System.Diagnostics.Process.Start` para abrir automáticamente el archivo `.txt` generado, permitiendo al usuario ver el registro de su operación fuera de la aplicación.

5. Mantenimiento del Estado (Limpieza)

El método `btnlimpiar_Click` restablece el sistema a su estado inicial, vaciando las variables numéricas y la interfaz visual para permitir nuevos ciclos de cómputo.

Código para realizar una simple calculadora en Visual Studio C#, también contiene una opción para imprimir nuestro resultado en un documento.txt.

```
using System.IO;

namespace Calculadora
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        double a;
        double b;
        string c;

        private void btn1_Click(object sender, EventArgs e)
        {
            if (txtpantalla.Text == "")
            {
                txtpantalla.Text = "1";
            }
            else
            {
                txtpantalla.Text = txtpantalla.Text + "1" ;
            }
        }

        private void btn2_Click(object sender, EventArgs e)
```

```
    {
        if (txtpantalla.Text == "")
        {
            txtpantalla.Text = "2";
        }
        else
        {
            txtpantalla.Text = txtpantalla.Text + "2";
        }
    }

    private void btn3_Click(object sender, EventArgs e)
    {
        if (txtpantalla.Text == "")
        {
            txtpantalla.Text = "3";
        }
        else
        {
            txtpantalla.Text = txtpantalla.Text + "3";
        }
    }

    private void btn4_Click(object sender, EventArgs e)
    {
        if (txtpantalla.Text == "")
        {
            txtpantalla.Text = "4";
        }
        else
        {
            txtpantalla.Text = txtpantalla.Text + "4";
        }
    }
```

```
private void btn5_Click(object sender, EventArgs e)
{
    if (txtpantalla.Text == "")
    {
        txtpantalla.Text = "5";
    }
    else
    {
        txtpantalla.Text = txtpantalla.Text + "5";
    }
}

private void btn6_Click(object sender, EventArgs e)
{
    if (txtpantalla.Text == "")
    {
        txtpantalla.Text = "6";
    }
    else
    {
        txtpantalla.Text = txtpantalla.Text + "6";
    }
}

private void btn7_Click(object sender, EventArgs e)
{
    if (txtpantalla.Text == "")
    {
        txtpantalla.Text = "7";
    }
    else
    {
        txtpantalla.Text = txtpantalla.Text + "7";
    }
}
```

```
    private void btn8_Click(object sender, EventArgs e)
    {
        if (txtpantalla.Text == "")
        {
            txtpantalla.Text = "8";
        }
        else
        {
            txtpantalla.Text = txtpantalla.Text + "8";
        }
    }

    private void btn9_Click(object sender, EventArgs e)
    {
        if (txtpantalla.Text == "")
        {
            txtpantalla.Text = "9";
        }
        else
        {
            txtpantalla.Text = txtpantalla.Text + "9";
        }
    }

    private void btn0_Click(object sender, EventArgs e)
    {
        if (txtpantalla.Text == "")
        {
            txtpantalla.Text = "0";
        }
        else
        {
            txtpantalla.Text = txtpantalla.Text + "0";
        }
```

```
    }

    private void btndivision_Click(object sender, EventArgs e)
    {
        a = Convert.ToDouble(this.txtpantalla.Text);
        c = "/";
        this.txtpantalla.Clear();
        this.txtpantalla.Focus();
    }

    private void btnmultiplicacion_Click(object sender, EventArgs e)
    {
        a = Convert.ToDouble(this.txtpantalla.Text);
        c = "*";
        this.txtpantalla.Clear();
        this.txtpantalla.Focus();
    }

private void btnresta_Click(object sender, EventArgs e)
    {
        a = Convert.ToDouble(this.txtpantalla.Text);
        c = "-";
        this.txtpantalla.Clear();
        this.txtpantalla.Focus();
    }

    private void btnsuma_Click(object sender, EventArgs e)
    {
        a = Convert.ToDouble(this.txtpantalla.Text);
        c = "+";
        this.txtpantalla.Clear();
        this.txtpantalla.Focus();
    }

    private void btnpunto_Click(object sender, EventArgs e)
```

```
    {
        if (this.txtpantalla.Text.Contains('.')==false)
        {
            this.txtpantalla.Text = this.txtpantalla.Text + ".";
        }


    }

    private void btnigual_Click(object sender, EventArgs e)
    {
        b = Convert.ToDouble(this.txtpantalla.Text);
        switch (c)
        {
            case "+":
                this.txtpantalla.Text  = Convert.ToString (b + a);
                break;

            case "-":
                this.txtpantalla.Text = Convert.ToString(b - a);
                break;

            case "*":
                this.txtpantalla.Text = Convert.ToString(b * a);
                break;

            case "/":
                this.txtpantalla.Text = Convert.ToString(b / a);
                break;
        }


    }

    private void btnprint_Click(object sender, EventArgs e)
```

```
        {
            StreamWriter Archivo = new StreamWriter("Ruta\\archivo.txt");
Archivo.WriteLine("Operaciones: " + a + c + b + "=" + this.txtpantalla.Text);
            Archivo.Flush();
            Archivo.Close();

            System.Diagnostics.Process.Start("Ruta\\archivo.txt");
        }

        private void btnlimpiar_Click(object sender, EventArgs e)
        {
            a = Convert.ToDouble("");
            b = Convert.ToDouble("");
            this.txtpantalla.Text = "";

        }
    }
}
```

Desde una perspectiva de ingeniería de software, las estructuras de control y el manejo de flujos de datos son fundamentales para la robustez de una aplicación. A continuación, se detallan los dos componentes solicitados basándose en el código analizado:

1. La Estructura de Control: `switch-case`

En el ámbito académico, el `switch` se define como una estructura de control de selección múltiple que evalúa una expresión y busca una coincidencia entre varias etiquetas de caso.

- **Evaluación de la Expresión:** El programa evalúa el valor almacenado en la variable de cadena `c`, la cual contiene el símbolo de la operación aritmética seleccionada.
- **Derivación de Flujo:** Dependiendo del carácter almacenado (como `+`, `-`, `*` o `/`), el programa salta directamente al bloque de código correspondiente.
- **Sentencia `break`:** Cada caso finaliza con un `break`, lo cual es esencial para evitar la "caída" hacia el siguiente caso, garantizando que solo se ejecute la lógica de una operación a la vez.
- **Conversión de Tipos:** Dentro de los casos, se realiza una operación aritmética entre `a` y `b`, y el resultado se transforma de tipo numérico a `string` mediante `Convert.ToString` para poder mostrarlo en la interfaz.

2. Manejo de Archivos con `System.IO`

El manejo de archivos en C# se realiza a través de flujos (*streams*), lo que permite la persistencia de los datos calculados más allá de la ejecución del programa en la memoria RAM.

- **Instanciación de `StreamWriter`:** Se utiliza la clase `StreamWriter` del espacio de nombres `System.IO` para crear un objeto llamado `Archivo`. Este objeto se vincula a una ruta específica en el sistema de archivos ("Ruta\archivo.txt").
- **Escritura de Datos:** El método `WriteLine` se encarga de formatear y escribir una línea de texto que concatena los operandos, el operador y el resultado final obtenido en la pantalla.
- **Gestión de Buffer y Cierre:**
  - `Archivo.Flush()` asegura que todos los datos acumulados en el buffer de memoria se escriban efectivamente en el disco duro.
  - `Archivo.Close()` libera los recursos del sistema y el "manejador" del archivo, permitiendo que otras aplicaciones puedan acceder a él.
- **Ejecución Externa:** Finalmente, el programa utiliza `System.Diagnostics.Process.Start` para invocar al sistema operativo y abrir el archivo generado automáticamente con el editor de texto predeterminado.

Observación Técnica

En el código de limpieza (`btnlimpiar_Click`), se observa un error de lógica común en programación principiante: el intento de convertir una cadena vacía (`""`) directamente a un tipo `double` mediante `Convert.ToDouble`, lo cual generaría una excepción en tiempo de ejecución si no se maneja adecuadamente.

Para elevar el nivel de este proyecto a un estándar académico profesional, es fundamental abordar el error en el método de limpieza y mejorar la robustez del código. En programación, intentar convertir una cadena vacía a un número sin validación previa es una causa común de fallos críticos (crashes).

Corrección técnica del error detectado en el archivo y una optimización del manejo de datos.

1. Corrección del Error de Conversión en `btnlimpiar_Click`

En el código original, el método `btnlimpiar_Click` intenta realizar `Convert.ToDouble("")`, lo cual lanzará una excepción de formato (`FormatException`) porque una cadena vacía no es un valor numérico válido.

Propuesta de corrección académica: En lugar de forzar una conversión imposible, se deben asignar valores neutros directamente a las variables de tipo `double`.

C#
```
private void btnlimpiar_Click(object sender, EventArgs e)
{
```

```
    // Asignación directa de valores neutros en lugar de conversión errónea
    a = 0.0;
    b = 0.0;
    c = string.Empty; // Limpia el operador guardado
    [cite_start]this.txtpantalla.Text = ""; [cite: 218]
    this.txtpantalla.Focus(); // Devuelve el foco para nueva entrada
}
```

2. Mejora en la Captura de Datos (`TryParse`)

Para evitar que el programa se cierre si el usuario presiona un botón de operación sin haber escrito un número, se recomienda sustituir `Convert.ToDouble` por el método `double.TryParse`.

- **Análisis Técnico:** `TryParse` devuelve un valor booleano. Si la conversión tiene éxito, asigna el valor a la variable; si falla, el programa puede continuar sin romperse.
- **Implementación:**

  C#

  ```
  if (double.TryParse(this.txtpantalla.Text, out double resultado))
  {
      a = resultado;
  }
  ```

3. Optimización del Manejo de Archivos (`System.IO`)

El código actual utiliza una ruta fija `"Ruta\\archivo.txt"`, lo cual es una debilidad de diseño ya que, si la carpeta no existe o no hay permisos de escritura, el programa fallará.

Mejora Académica sugerida: Utilizar rutas dinámicas del sistema para asegurar la portabilidad del software:

C#

```
// Obtiene la ruta de la carpeta "Mis Documentos" del usuario actual
string rutaDocumentos = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
string nombreArchivo = Path.Combine(rutaDocumentos, "resultado_calculadora.txt");

using (StreamWriter Archivo = new StreamWriter(nombreArchivo))
{
    [cite_start]Archivo.WriteLine("Operaciones: " + a + c + b + "=" + this.txtpantalla.Text); [cite: 206]
[cite_start]} // El bloque 'using' cierra automáticamente el archivo (equivale a Close y Flush) [cite: 207, 208]
```

Resumen de Hallazgos en el Documento

- **Manejo de Operadores:** El sistema utiliza un `switch (c)` para discernir entre sumas, restas y multiplicaciones.
- **Interfaz Visual:** El formulario (`Form1`) incluye botones numéricos del 0 al 9 y una pantalla de texto (`txtpantalla`) para mostrar los valores.
- **Funcionalidad Extra:** Posee un botón "Print" que genera un reporte externo de la operación realizada.

Implementar un bloque Try-Catch global para capturar cualquier error inesperado durante el cálculo

Implementar un bloque **Try-Catch** es una práctica fundamental en la ingeniería de software para garantizar la **tolerancia a fallos**. En un entorno académico, esto se conoce como "Manejo Estructural de Excepciones".

En el código de la calculadora, el punto más crítico para implementar este bloque es el evento del botón igual, ya que es donde se realizan las conversiones de tipos y las operaciones aritméticas que podrían fallar.

Implementación del Bloque Try-Catch

A continuación, se presenta la reestructuración del método `btnigual_Click` utilizando este mecanismo de seguridad:

C#
```
private void btnigual_Click(object sender, EventArgs e)
{
    try
    {
        [cite_start]// Intento de conversión de la entrada actual a tipo double
[cite: 186]
        [cite_start]b = Convert.ToDouble(this.txtpantalla.Text); [cite: 186]

        [cite_start]// Estructura de decisión para la operación guardada [cite:
187]
        [cite_start]switch (c) [cite: 187]
        {
            [cite_start]case "+": [cite: 189]
                [cite_start]this.txtpantalla.Text  =  Convert.ToString(a  +  b);
[cite: 190]
                [cite_start]break; [cite: 191]
            [cite_start]case "-": [cite: 192]
                [cite_start]this.txtpantalla.Text  =  Convert.ToString(a  -  b);
[cite: 193]
                [cite_start]break; [cite: 194]
            [cite_start]case "*": [cite: 195]
                [cite_start]this.txtpantalla.Text  =  Convert.ToString(a  *  b);
[cite: 196]
                [cite_start]break; [cite: 198]
            [cite_start]case "/": [cite: 199]
                // Validación académica: Evitar división por cero
```

```
                if (b == 0) throw new DivideByZeroException();
                [cite_start]this.txtpantalla.Text = Convert.ToString(a / b);
[cite: 200]
                [cite_start]break; [cite: 201]
            [cite_start]} [cite: 202]
        }
        catch (FormatException ex)
        {
            // Captura errores si el usuario ingresa caracteres no numéricos
            MessageBox.Show("Error: Entrada de datos no válida.", "Error de Formato",
MessageBoxButtons.OK, MessageBoxIcon.Error);
            btnlimpiar_Click(sender, e); // Llama al método limpiar para resetear el
estado
        }
        catch (DivideByZeroException)
        {
            // Captura específicamente el error matemático de dividir por cero
            MessageBox.Show("Error: No es posible dividir entre cero.", "Error
Matemático", MessageBoxButtons.OK, MessageBoxIcon.Warning);
        }
        catch (Exception ex)
        {
            // Captura cualquier otro error no previsto (Catch Global)
            MessageBox.Show("Ocurrió un error inesperado: " + ex.Message);
        }
}
```

Análisis de la Mejora

- **Separación de Responsabilidades:** El bloque `try` contiene el "camino feliz" (la ejecución normal), mientras que los bloques `catch` gestionan las anomalías.
- **Feedback al Usuario:** En lugar de que la aplicación se cierre abruptamente (un "crash"), se utilizan cuadros de diálogo (`MessageBox`) para informar al usuario qué salió mal.
- **Integración con Funciones Existentes:** Se aprovecha el método `btnlimpiar_Click` dentro del catch para limpiar la interfaz (`txtpantalla.Text = ""`) tras un error, devolviendo el programa a un estado seguro.

Manejo de errores interactúa con el guardado de archivos en el método `btnprint_Click`

En un entorno universitario, el método `btnprint_Click` representa un punto crítico de interacción entre el software y el hardware (el sistema de archivos), lo que requiere una gestión de excepciones robusta para evitar que fallos externos (como falta de permisos o rutas inexistentes) interrumpan la ejecución.

Manejo de errores debe interactuar con las operaciones de entrada/salida (I/O) presentes en el código:

1. Riesgos en el Código Original

El código actual presenta vulnerabilidades que podrían causar el cierre de la aplicación:

- **Rutas estáticas:** El uso de `"Ruta\\archivo.txt"` asume que existe un directorio llamado "Ruta" en la unidad raíz, lo cual es poco probable en sistemas modernos y generará un error de tipo `DirectoryNotFoundException`.
- **Acceso Exclusivo:** Si el archivo ya está abierto por otra aplicación (como el Bloc de notas), el método fallará al intentar escribir en él.

2. Implementación de Seguridad en `btnprint_Click`

Para integrar el manejo de errores con la persistencia de datos, se debe envolver la lógica en un bloque `try-catch` específico para `System.IO`:

C#

```
private void btnprint_Click(object sender, EventArgs e)
{
    try
    {
        [cite_start]// Se intenta inicializar el flujo de escritura
        using (StreamWriter Archivo = new StreamWriter("archivo_resultado.txt"))
        {
            [cite_start]// Se registra la operación aritmética y el resultado
            Archivo.WriteLine("Operaciones: " + a + c + b + "=" + this.txtpantalla.Text);
            Archivo.Flush(); // Asegura la integridad de los datos [cite: 207]
        [cite_start]} // El cierre automático libera el archivo para otros procesos

        [cite_start]// Intento de apertura del archivo mediante el sistema operativo [cite: 209]
        System.Diagnostics.Process.Start("archivo_resultado.txt");
    }
    catch (IOException ex)
    {
        [cite_start]// Captura errores de disco lleno, archivos bloqueados o falta de permisos [cite: 27]
        MessageBox.Show("Error de escritura: Verifique que el archivo no esté abierto por otro programa.", "Error de E/S");
    }
    catch (Exception ex)
    {
        [cite_start]// Captura errores inesperados al intentar iniciar el proceso externo [cite: 209]
        MessageBox.Show("No se pudo abrir el archivo: " + ex.Message);
    }
}
```

3. Sincronía entre Lógica y Persistencia

El manejo de errores asegura que los datos mostrados en la interfaz (`txtpantalla.Text`) coincidan con los datos persistidos en el documento `.txt`.

- **Validación de Datos:** Si el cálculo falló previamente por una división por cero, el manejo de errores en el cálculo debe impedir que se llame a `btnprint_Click`, evitando así que se guarde un resultado erróneo o inexistente.
- **Garantía de Cierre:** El uso de la instrucción `using` (o en su defecto un bloque `finally`) garantiza que, incluso si ocurre un error durante la escritura, el archivo sea liberado correctamente (`Close`), evitando fugas de memoria o bloqueos del sistema.

Esta estructura no solo protege al usuario, sino que facilita la **depuración académica**, permitiendo identificar si el fallo es de lógica interna (aritmética) o de infraestructura (almacenamiento).

Interfaz de usuario mejorada para mostrar estos mensajes de error de forma más estética

Desde una perspectiva de **Diseño de Interfaz de Usuario (UI)** y **Experiencia de Usuario (UX)** a nivel universitario, mejorar la estética de los mensajes de error no solo se trata de "colores", sino de reducir la carga cognitiva del usuario y mantener la coherencia visual de la aplicación.

En el código original, la interfaz se basa en un `Form1` estándar de Windows Forms con una pantalla de texto (`txtpantalla`) y una cuadrícula de botones para números y operaciones.

Para una interfaz mejorada y estética, se proponen los siguientes cambios:

1. Implementación de un Panel de Notificaciones Dinámico

En lugar de utilizar el `MessageBox` clásico (que interrumpe el flujo de trabajo con una ventana emergente ruidosa), se recomienda un área de estado dentro del propio formulario.

- **Label de Estado:** Añadir un `Label` en la parte inferior del `Form1` con visibilidad controlada.
- **Codificación por Colores:**
  - **Rojo suave:** Para errores críticos como la división por cero.
  - **Amarillo/Naranja:** Para advertencias de formato, como cuando el usuario intenta ingresar dos puntos decimales.
  - **Verde:** Para confirmar que el archivo se guardó exitosamente mediante `btnprint_Click`.

2. Refinamiento de la "Pantalla" (`txtpantalla`)

El código actual simplemente concatena números en un `TextBox`. Una mejora estética incluiría:

- **Placeholder de Error:** Cuando ocurre una excepción en el bloque `try-catch`, la pantalla podría mostrar un texto breve como `"Err: Div/0"` o `"Formato Inválido"` en lugar de simplemente quedar vacía.
- **Cambio de Bordes:** Cambiar el color del borde del `txtpantalla` a rojo mediante el evento `Paint` cuando se captura una excepción.

3. Feedback Visual en los Botones

Dado que el código maneja clics individuales para cada botón (del `btn0` al `btn9` y operaciones), se pueden añadir efectos:

- **Deshabilitación Inteligente:** Deshabilitar temporalmente los botones de operación (`btndivision`, `btnmultiplicacion`, etc.) si la pantalla está vacía, evitando que el usuario genere un error de conversión antes de empezar.
- **Tooltips:** Añadir pequeñas etiquetas flotantes que expliquen qué hace el botón "Print" (imprimir a `.txt`) para mejorar la usabilidad.

4. Estilización del Reporte Externo (`archivo.txt`)

Incluso la salida de datos puede ser más estética. En lugar de una línea plana de texto, el método `btnprint_Click` podría formatear el archivo de la siguiente manera:

REPORTE DE OPERACIÓN - CALCULADORA C#

Entrada A: 12 **Operación:** + **Entrada B:** 5

RESULTADO FINAL: 17

Comparativa: Interfaz Original vs. Mejorada

Elemento	**Interfaz Original (Código PDF)**	**Interfaz Universitaria Mejorada**
Aviso de Error	`MessageBox` estándar (Invasivo)	Panel de notificación integrado (Fluido)
Pantalla	Texto negro plano	Texto con jerarquía visual y colores de estado
Botón Limpiar	Vacía todo bruscamente	Animación de desvanecimiento y reset de variables
Ruta de Archivo	Ruta estática `"Ruta\\archivo.txt"`	Diálogo de guardado (`SaveFileDialog`) estético

Para implementar un sistema de notificaciones estético en el **Form1**, debemos integrar un componente visual que responda a los bloques `try-catch` que diseñamos previamente. Esto sustituye los mensajes emergentes invasivos por una experiencia de usuario fluida y profesional.

Aquí tienes el código necesario para añadir esta funcionalidad:

1. Definición del Label de Notificación

Primero, asegúrate de añadir un control `Label` en la parte inferior de tu formulario desde el diseñador de Visual Studio o mediante código en `InitializeComponent()`. Llámalo `lblNotificacion`.

2. Método Auxiliar de Notificación

Añade este método dentro de la clase `Form1` para gestionar los colores y el texto de forma centralizada:

C#
```
private void MostrarNotificacion(string mensaje, Color colorFondo)
{
    lblNotificacion.Text = mensaje;
    lblNotificacion.BackColor = colorFondo;
    lblNotificacion.ForeColor = Color.White; // Texto blanco para contraste
    lblNotificacion.Visible = true;

    // Opcional: Un temporizador para que la notificación desaparezca tras 3 segundos
    Timer t = new Timer();
    t.Interval = 3000;
    t.Tick += (s, e) => { lblNotificacion.Visible = false; t.Stop(); };
    t.Start();
}
```

3. Integración en los Eventos Críticos

Ahora, actualizamos el manejo de errores en los métodos de cálculo e impresión de archivos:

En el Cálculo (`btnigual_Click`):

C#
```
private void btnigual_Click(object sender, EventArgs e)
{
    try
    {
        [cite_start]b = Convert.ToDouble(this.txtpantalla.Text); [cite: 186]
```

```
        [cite_start]// ... (resto del switch) [cite: 187, 188]

        MostrarNotificacion("Cálculo exitoso", Color.MediumSeaGreen);
    }
    catch (DivideByZeroException)
    {
        MostrarNotificacion("Error: División por cero", Color.IndianRed);
        this.txtpantalla.Text = "Error";
    }
    catch (Exception)
    {
        MostrarNotificacion("Error de entrada", Color.Orange);
    }
}
```

En la Impresión de Archivo (`btnprint_Click`):

C#

```
private void btnprint_Click(object sender, EventArgs e)
{
    try
    {
        [cite_start]// Lógica de StreamWriter [cite: 205, 206]
        using (StreamWriter Archivo = new StreamWriter("archivo.txt"))
        {
            [cite_start]Archivo.WriteLine("Operaciones: " + a + c + b + "="
+ this.txtpantalla.Text); [cite: 206]
        }
        MostrarNotificacion("Archivo guardado y abierto",
Color.DeepSkyBlue);
        [cite_start]System.Diagnostics.Process.Start("archivo.txt"); [cite:
209]
    }
    catch (IOException)
    {
        MostrarNotificacion("Error: Archivo en uso", Color.Crimson);
    }
}
```

Análisis de la Mejora UI/UX

- **Contraste Visual:** El uso de `Color.IndianRed` para errores y `Color.MediumSeaGreen` para éxitos permite que el usuario identifique el estado de la aplicación sin leer el texto.
- **Limpieza de Interfaz:** Al usar un temporizador para ocultar el `lblNotificacion`, mantenemos la calculadora limpia y evitamos que mensajes antiguos confundan al usuario.

- **Foco en el Resultado:** La pantalla principal (`txtpantalla`) sigue mostrando el número calculado, mientras que el label inferior da el contexto del sistema.

Crear una hoja de estilos (CSS-like) para que los botones numéricos cambien de color cuando el mouse pase sobre ellos

En el entorno de **Windows Forms (C#)**, aunque no existe un archivo `.css` como en el desarrollo web, podemos emular este comportamiento mediante la gestión de **estados visuales** y **eventos del ratón**.

Para aplicar este estilo "CSS-like" a los botones numéricos (como los identificados del `btn0` al `btn9` en el código), utilizaremos los eventos `MouseEnter` (cuando el puntero entra al botón) y `MouseLeave` (cuando el puntero sale).

1. Definición de la "Hoja de Estilos" (Métodos de Cambio)

En lugar de repetir el código en cada uno de los 10 botones numéricos, crearemos dos métodos generales en `Form1` que controlen la apariencia estética:
C#

```
// Estilo cuando el mouse está encima (Hover)
private void EstiloBoton_MouseEnter(object sender, EventArgs e)
{
    if (sender is Button btn)
    {
        btn.BackColor = Color.LightSteelBlue; // Color de resaltado
        btn.Cursor = Cursors.Hand;           // Cambia el cursor a mano
    }
}

// Estilo cuando el mouse sale (Normal)
private void EstiloBoton_MouseLeave(object sender, EventArgs e)
{
    if (sender is Button btn)
    {
        btn.BackColor = SystemColors.ControlLight; // Color original (gris
claro)
        btn.Cursor = Cursors.Default;
    }
}
```

2. Vinculación con los Botones de la Calculadora

Para que estos estilos afecten a los botones del formulario (como el `btn1`, `btn2`, etc.), debes asignar estos métodos a los eventos correspondientes en el constructor `Form1()` o mediante el diseñador:

- **Desde el Diseñador:** Selecciona todos los botones numéricos, ve a la ventana de **Eventos** (el rayo), busca `MouseEnter` y selecciona `EstiloBoton_MouseEnter`. Repite para `MouseLeave`.
- **Desde Código (Dinámico):** Puedes hacerlo en el `InitializeComponent` para asegurar la consistencia:

C#
```
public Form1()
{
    InitializeComponent();
    AsignarEstilos();
}

private void AsignarEstilos()
{
    // Lista de botones numéricos identificados en el documento
    Button[] botonesNumericos = { btn0, btn1, btn2, btn3, btn4, btn5, btn6,
btn7, btn8, btn9 };

    foreach (Button btn in botonesNumericos)
    {
        btn.MouseEnter += EstiloBoton_MouseEnter;
        btn.MouseLeave += EstiloBoton_MouseLeave;
    }
}
```

3. Justificación Académica del Diseño UI

- **Feedback Inmediato:** En UX, el cambio de color al pasar el ratón confirma al usuario que el elemento es **interactuable**, reduciendo errores de clic antes de que ocurra la entrada de datos.
- **Consistencia Visual:** Al aplicar el mismo cambio a todos los botones del `0` al `9` , se crea un patrón visual que ayuda al usuario a distinguir rápidamente entre números y operadores (como `+` o `X`).
- **Accesibilidad:** El cambio del cursor a `Cursors.Hand` es una convención estándar que mejora la usabilidad del programa.