UNIVERSIDAD
Blas Pascal

“TAREA 8 - Excepciones por partes”

Asignatura: PROGRAMACION GENERICA Y EVENTOS

Fecha: 20/09/24

Autor: Battiston, Juan Ignacio
Dematias, Fabrizio
Muñoz, Francisco

Profesores: Mónica L. Nano
De Aragon, Juan Manuel
Olariaga Oliveto, Ignacio Jesús

“Excepciones por partes en c#”

Por:

Battiston, Juan Ignacio
Dematias, Fabrizio
Muñoz, Francisco Agustín

RESUMEN

En el siguiente informe, se va a adjuntar y desarrollar todos los ejemplos, explicaciones y resultados de la investigación sobre excepciones por partes, como se desarrollaron los 4 ejercicios que se solicitaron y una información breve sobre cada uno de los mismos.

Córdoba, 20 de septiembre del 2023

INDICE

Contenido

1. INTRODUCCIÓN

Este informe explora dos áreas en C#: la gestión de recursos mediante el uso del patrón “using” y el método “Dispose()”, y el manejo de excepciones con bloques try-catch.

La correcta liberación de recursos, como archivos, conexiones de base de datos y objetos gráficos, es importante para evitar fugas de memoria y problemas de rendimiento. A través del patrón “using”, C# ofrece una forma simplificada de gestionar tanto recursos administrados como no administrados, asegurando su liberación automática sin intervención manual.

Por otro lado, el manejo de excepciones específicas permite anticipar y gestionar errores comunes, como archivos no encontrados o permisos insuficientes, mejorando el desarrollo del código. El uso adecuado de excepciones generales y específicas asegura que las aplicaciones funcionen adecuadamente a los errores, manteniendo una operación estable y fluida.

2. DESARROLLO

2.1 Parte 1: Gestión de Recursos y Uso de Dispose()

Objetivos Específicos:

- Comprender la importancia de liberar recursos no administrados.
- Implementar el uso de using para manejar recursos como archivos.

- **Funcionamiento del Patrón using y Dispose()**: Se requiere investigar cómo el patrón using en C# facilita la gestión de recursos, asegurando que se liberen correctamente los recursos administrados, como archivos y conexiones, sin que el desarrollador tenga que preocuparse por liberar manualmente estos recursos.
- **Diferencias entre Recursos Administrados y No Administrados**: Es importante entender las diferencias entre estos dos tipos de recursos. Los recursos administrados son aquellos que el CLR de .NET maneja automáticamente, como memoria, mientras que los recursos no administrados son aquellos que necesitan ser liberados manualmente, como handles de archivos o conexiones a bases de datos.
- **Documentación de Ejemplos Adicionales**: Investigar y documentar situaciones en las que el método Dispose() es crucial, más allá del uso de archivos, como en el manejo de conexiones a bases de datos o en el trabajo con imágenes.

2.1.1 Cómo el patrón "using" simplifica la gestión de recursos

En C#, el patrón "using" se utiliza para simplificar la gestión de recursos administrados y no administrados. Los recursos administrados son aquellos que el runtime de .NET puede manejar automáticamente, como la memoria, mientras que los no administrados son recursos que el runtime no puede liberar automáticamente, como los manejadores de archivos, conexiones de red, etc.

Este patrón asegura que los objetos que implementan la interfaz "IDisponsable" (que incluye el método Dispose()) sean liberados correctamente una vez que ya no se necesiten. Esto es crucial para evitar fugas de memoria y otros problemas de rendimiento.

2.1.2 Diferencias entre recursos administrados y no administrados

- **Recursos Administrados:** Son aquellos que el runtime de .NET puede manejar y limpiar automáticamente. Ejemplos incluyen objetos de la clase String, List, etc., donde el recolector de basura se encarga de liberar la memoria ocupada por estos objetos cuando ya no son accesibles.
- **Recursos No Administrados:** Son recursos que no son gestionados por el runtime de .NET y requieren una liberación manual para evitar fugas de memoria. Ejemplos incluyen manejadores de archivos, conexiones de base de datos, y otros recursos de sistema operativo. Para liberar estos recursos, se debe implementar la interfaz

2.1.3 Documentar ejemplos adicionales donde Dispose() es crucial.

El método Dispose() es crucial en situaciones donde se trabaja con recursos no administrados. Algunos ejemplos incluyen:

Manejo de Conexiones a Bases de Datos:

```
using (SqlConnection connection = new SqlConnection(connectionString))
{
    connection.Open();
    // Operaciones con la base de datos
}
// Aquí, Dispose() se llama para cerrar la conexión y liberar recursos.
```

Manejo de archivos o streams:

```
using (FileStream fileStream = new FileStream("rutaArchivo", FileMode.Open))
{
    // Operaciones con el archivo
}
// Dispose() se llama automáticamente, cerrando el archivo.
```

Trabajando con objetos gráficos:

```
using (Bitmap bitmap = new Bitmap("imagen.png"))
{
    // Operaciones con la imagen
}
// Dispose() libera los recursos gráficos
```

Estos ejemplos muestran cómo el patrón Using() y el método Dispose() permiten una gestión más sencilla y segura de recursos, asegurando que se liberen adecuadamente, lo cual es esencial para la estabilidad y el rendimiento de las aplicaciones.

2.2 Parte 2: Manejo de excepciones en C#, utilizando bloques try-catch.

Objetivos Específicos:

- Implementar el manejo de excepciones utilizando bloques try-catch.
- Capturar excepciones específicas y manejar errores comunes de entrada/salida.

- **Tipos de Excepciones Específicas**: Investigar las diferencias entre varios tipos de excepciones en C#, como “FileNotFoudException” , “IQException”, “UnauthorizedAccessExceptioon”, etc. para entender cuándo es más apropiado capturar excepciones específicas en lugar de usar excepciones generales.

- **Captura de Excepciones Específicas vs Generales**: Explicar la importancia de manejar excepciones específicas en escenarios concretos para evitar que errores críticos pasen desapercibidos, comparado con la captura de excepciones generales que pueden servir para manejar errores inesperados de una manera más general.

2.2.1 Tipos de Excepciones Específicas:

- FileNotFoundException: Se lanza cuando se intenta acceder a un archivo que no existe. Es útil en operaciones de lectura de archivos.
- IOException: Representa un error general de entrada/salida y puede capturar una amplia gama de errores relacionados con la lectura/escritura en archivos.
- UnauthorizedAccessException: Ocurre cuando se intenta acceder a un recurso sin los permisos necesarios, como escribir en un directorio protegido.

2.2.2 Diferencias entre excepciones específicas:

- FileNotFoundException vs IOException:
  - FileNotFoundException es una excepción específica que indica que un archivo no se encontró. Por otro lado, IOException es más general y puede representar cualquier problema relacionado con la entrada/salida, no solo que un archivo no exista.

- UnauthorizedAccessException vs IOException:
  - UnauthorizedAccessException está más relacionada con problemas de permisos y acceso, mientras que IOException es un paraguas más amplio que puede incluir problemas de acceso, pero también otros problemas de I/O.

2.2.3 Captura de excepciones específicas vs generales:

- Excepciones Específicas:
  - Capturar excepciones específicas es útil cuando deseas manejar un tipo particular de error de manera diferente. Por ejemplo, si deseas mostrar un mensaje específico cuando un archivo no se encuentra, capturar FileNotFoundException es más apropiado.

- Excepciones Generales (Exception o IOException):
  - Usar una excepción general es útil cuando quieres atrapar cualquier error sin importar su tipo y posiblemente realizar una acción común, como registrar el error o lanzar una nueva excepción. Sin embargo, capturar una excepción muy general puede ocultar errores específicos que podrían manejarse de manera más efectiva.

2.2.4 Ejemplo en código

```
try
{
    // Código que podría lanzar excepciones
    string text = File.ReadAllText("ruta_del_archivo.txt");
}
catch (FileNotFoundException ex)
{
    Console.WriteLine("Archivo no encontrado: " + ex.Message);
}
catch (UnauthorizedAccessException ex)
{
    Console.WriteLine("Acceso no autorizado: " + ex.Message);
}
catch (IOException ex)
{
    Console.WriteLine("Error de entrada/salida: " + ex.Message);
}
catch (Exception ex)
{
    Console.WriteLine("Ocurrió un error: " + ex.Message);
}

```

En este ejemplo, se manejan excepciones específicas para casos particulares como archivos no encontrados o acceso no autorizado, pero también se incluye una captura general para manejar cualquier otro tipo de excepción no específica.

2.3 Parte 3: Uso de excepciones específicas de la librería estándar de .NET para manejar situaciones particulares.

Objetivos Específicos:

- Familiarizarse con excepciones estándar adicionales y cómo utilizarlas para validar y manejar errores específicos.
- Implementar validaciones de entrada y lanzar excepciones estándar según corresponda.

2.3.1 Exploraciones de Excepciones Estándar Adicionales

1. **InvalidOperationException**

Cuando se utiliza: Esta excepción se lanza cuando el estado actual de un objeto no permite la operación solicitada. Por ejemplo, intentar eliminar un elemento de una lista que está vacía.

Ejemplo de uso:

```
List<int> numbers = new List<int>();
numbers.Remove(1); // Esto lanzará InvalidOperationException si la lista está vacía

```

2. **NullReferenceException**

Cuando se utiliza: Esta excepción se lanza cuando se intenta acceder a un miembro de una referencia que es “Null”. Por ejemplo, al intentar invocar un método en un objeto que no ha sido instanciado.

Ejemplo de uso:

```
string text = null;
Console.WriteLine(text.Length); // Lanza NullReferenceException

```

3. **IndexOutOfRangeException**

Cuando se utiliza: Se lanza cuando se intenta acceder a un índice que está fuera del rango de una matriz o una colección. Por ejemplo, al acceder al quinto elemento de una lista que solo tiene tres elementos.

Ejemplo de uso:

```
int[] array = new int[] { 1, 2, 3 };
Console.WriteLine(array[5]); // Lanza IndexOutOfRangeException

```

2.3.2 Documentación de uso en aplicaciones desarrolladas

Estas excepciones se utilizan comúnmente para manejar errores inesperados en el flujo de ejecución de una aplicación. Es una buena práctica capturar estas excepciones y manejarlas de manera adecuada para evitar que la aplicación falle abruptamente. Aquí hay un ejemplo general de cómo manejar estas excepciones:

```
try
{
    // Código que puede lanzar una excepción
}
catch (InvalidOperationException ex)
{
    Console.WriteLine("Operación no permitida: " + ex.Message);
}
catch (NullReferenceException ex)
{
    Console.WriteLine("Referencia nula encontrada: " + ex.Message);
}
catch (IndexOutOfRangeException ex)
{
    Console.WriteLine("Índice fuera de rango: " + ex.Message);
}

```

Este manejo asegura que la aplicación pueda continuar funcionando o al menos cerrar de manera controlada, mostrando un mensaje apropiado al usuario o registrando el error para un análisis posterior.

2.4 Parte 4: Proyecto Integrador y Profundización

Objetivos Específicos:

- Integrar todas las funcionalidades desarrolladas en las partes anteriores en una sola aplicación de consola robusta.
- Realizar una investigación de profundización sobre un tema avanzado relacionado, como patrones de diseño para manejo de recursos o técnicas avanzadas de logging.

2.4.1 investigación: Implementación de un sistema de logging avanzado en C#

El logging es una práctica esencial en el desarrollo de software, especialmente en aplicaciones que requieren monitoreo, depuración y mantenimiento continuo. Un sistema de logging bien diseñado permite registrar eventos significativos que ocurren durante la ejecución de un programa, proporcionando una fuente de información invaluable para los desarrolladores y administradores del sistema.

En esta investigación, nos enfocaremos en la implementación de un sistema de logging avanzado en aplicaciones C# utilizando patrones de diseño y librerías como NLog y log4net. Exploraremos las mejores prácticas para configurar el logging en diferentes tipos de aplicaciones, así como un análisis comparativo entre el uso de estas librerías y una implementación casera.

2.4.2 Patrones de Diseño para el Logging

Patrón Singleton

El patrón Singleton es uno de los más utilizados en la implementación de sistemas de logging. Este patrón asegura que una clase tenga solo una instancia y proporciona un punto global de acceso a ella. Es crucial en el contexto de logging porque garantiza que todas las partes de la aplicación utilicen la misma instancia de logger, evitando duplicaciones y asegurando la consistencia en los logs.

- **Ejemplo de implementación en c#**

```
public sealed class Logger
{
    private static readonly Logger _instance = new Logger();

    private Logger()
    {
        // Configuración del logger
    }

    public static Logger Instance
    {
        get
        {
            return _instance;
        }
    }

    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}
```

Patrón Decorador

El patrón Decorador permite agregar responsabilidades a un objeto de manera dinámica. Es útil en logging para extender las capacidades del logger sin modificar su estructura base. Por ejemplo, puedes usarlo para añadir funcionalidad de logging en diferentes niveles o destinos.

- **Ejemplo de implementación en c#**

```
public interface ILogger
{
    void Log(string message);
}

public class SimpleLogger : ILogger
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}

public class FileLoggerDecorator : ILogger
{
    private readonly ILogger _logger;
    private readonly string _filePath;

    public FileLoggerDecorator(ILogger logger, string filePath)
    {
        _logger = logger;
        _filePath = filePath;
    }

    public void Log(string message)
    {
        _logger.Log(message);
        File.AppendAllText(_filePath, message + Environment.NewLine);
    }
}
```

Patrón Estrategia

El patrón Estrategia permite definir una familia de algoritmos, encapsularlos y hacerlos intercambiables. En el contexto de logging, este patrón se puede utilizar para cambiar dinámicamente el método de logging utilizado, como enviar logs a la consola, a un archivo, o a una base de datos.

- **Ejemplo de implementación en C#**

```
public interface ILogStrategy
{
    void Log(string message);
}

public class ConsoleLogStrategy : ILogStrategy
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}

public class FileLogStrategy : ILogStrategy
{
    private readonly string _filePath;

    public FileLogStrategy(string filePath)
    {
        _filePath = filePath;
    }

    public void Log(string message)
    {
        File.AppendAllText(_filePath, message + Environment.NewLine);
    }
}

public class LoggerContext
{
    private ILogStrategy _logStrategy;

    public LoggerContext(ILogStrategy logStrategy)
    {
        _logStrategy = logStrategy;
    }

    public void SetStrategy(ILogStrategy logStrategy)
    {
        _logStrategy = logStrategy;
    }

    public void Log(string message)
    {
        _logStrategy.Log(message);
    }
}
```

2.4.3 Uso de Librerías de Logging en C#

Nlog

NLog es una de las librerías más populares para logging en .NET. Es conocida por su facilidad de uso y su flexibilidad para configurar destinos de logging, como archivos, bases de datos, correos electrónicos, etc.

log4net

log4net es otra librería ampliamente utilizada, derivada de la famosa log4j de Java. Es extremadamente flexible y permite una configuración detallada de los logs.

2.4.4 Mejores Prácticas para el Logging

- **Consistencia en el formato de los logs**

Mantener un formato uniforme para los mensajes de log es esencial para facilitar la lectura y el análisis de estos. Utiliza patrones de layout en NLog y log4net para asegurarte de que todos los mensajes sigan un formato predefinido.

- **Uso adecuado de niveles de logging**

Define niveles de logging (Debug, Info, Warn, Error, Fatal) y utilízalos de manera coherente. Por ejemplo, los mensajes de nivel “Debug” deberían proporcionar detalles técnicos para la depuración, mientras que los mensajes error deberían destacar problemas que requieren atención inmediata.

- **Inclusión de contexto en los mensajes de log**

Siempre que sea posible, incluye información contextual como IDs de usuario, nombres de funciones, y otros detalles relevantes. Esto ayuda a identificar rápidamente la causa de un problema.

- **Consideraciones de rendimiento**

Evita loggear en exceso, especialmente en bucles o en partes críticas de rendimiento de la aplicación. Si necesitas loggear grandes volúmenes de datos, considera utilizar logging asíncrono para no bloquear el hilo principal de la aplicación.

- **Seguridad en el logging**

Nunca loguees información sensible como contraseñas, números de tarjetas de crédito, o cualquier otro dato que pueda comprometer la seguridad de los usuarios.

2.4.5 Comparación entre librerías de logging y una implementación casera

Ventajas de usar librerías de Logging

- **Facilidad de uso:** Librerías como NLog y log4net vienen con una amplia gama de funcionalidades listas para usar, como el soporte para diferentes destinos de logging y niveles de severidad.
- **Flexibilidad:** Estas librerías ofrecen configuraciones avanzadas a través de archivos de configuración, lo que permite adaptar el comportamiento del logging sin necesidad de cambiar el código fuente.

- **Mantenimiento:** Las librerías populares suelen estar bien mantenidas, con actualizaciones frecuentes que mejoran la funcionalidad y corrigen errores.

Desventajas de usar una implementación casera

- **Complejidad:** Implementar un sistema de logging desde cero puede ser una tarea compleja y propensa a errores, especialmente cuando se necesitan características avanzadas como logging asincrónico o múltiples destinos.

- **Mantenimiento:** Mantener una implementación casera requiere tiempo y esfuerzo, especialmente cuando se trata de mejorar la funcionalidad o corregir problemas de rendimiento.

2.4.6 Ejemplos de Implementación

Implementacion con NLog

1. Instalar las dependencias necesarias desde NuGet.
2. Crear el archivo “nlog.config” con las configuraciones deseadas.
3. Usar el logger en el código de la aplicación.

```
private static readonly Logger Logger = LogManager.GetCurrentClassLogger();

static void Main(string[] args)
{
    Logger.Info("Inicio de la aplicación.");
    Logger.Debug("Mensaje de depuración.");
    Logger.Warn("Advertencia: posible problema detectado.");
    Logger.Error("Error en la aplicación.");
}
```

Implementacion con log4net

1. Instalar log4net desde “NuGet”.
2. Configurar el archivo “Web.config” para definir los “appenders” y los niveles de logging.
3. Usar el logger en el código de la aplicación.

```
private static readonly ILog Logger = LogManager.GetLogger(typeof(HomeController));

public ActionResult Index()
{
    log4net.Config.XmlConfigurator.Configure();
    Logger.Info("Cargando la página de inicio.");
    return View();
}

```

3. CONCLUSIONES

Implementar un sistema de logging robusto es crucial para el éxito de cualquier aplicación. El uso de patrones de diseño como Singleton, Decorador y Estrategia puede mejorar significativamente la flexibilidad y mantenibilidad del sistema. Además, el uso de librerías como NLog y log4net permite a los desarrolladores concentrarse en la lógica de negocio en lugar de reinventar la rueda. Seguir las mejores prácticas de logging asegurará que el sistema no solo sea eficiente, sino también seguro y fácil de mantener.