

Índice

- 1. Introducción.....2
- 2. Tipos de errores2
 - 2.1. Errores de escritura2
 - 2.2. Errores de ejecución.....3
 - 2.3. Errores lógicos3
- 3. Errores y Excepciones3
 - 3.1. Manipulador de excepciones4
- 4. Formas de interceptar los errores4
 - 4.1. Control no estructurado de errores4
 - 4.2. Control estructurado de errores4
- 5. Estructura Try...Catch...Finally4
- 6. La clase Exception5
 - Ejercicio práctico8
 - 6.1. Lanzar nuevas excepciones. Throw9
 - 6.1.1. Modificando el ArgumentException9
 - 6.1.2. Creando nuestras propias excepciones10

1. Introducción

Difícil es, por no decir imposible, encontrar al programador que no tenga errores en su código. Por mucho cuidado que pongamos al codificar nuestras aplicaciones, los errores de ejecución serán ese incómodo, pero inevitable compañero de viaje que seguirá a nuestros programas allá donde estos vayan.

En primer lugar, antes de abordar el tratamiento de errores en nuestras aplicaciones, y los elementos que nos proporciona el entorno para manipularlos, podemos clasificar los tipos de errores en una serie de categorías genéricas.

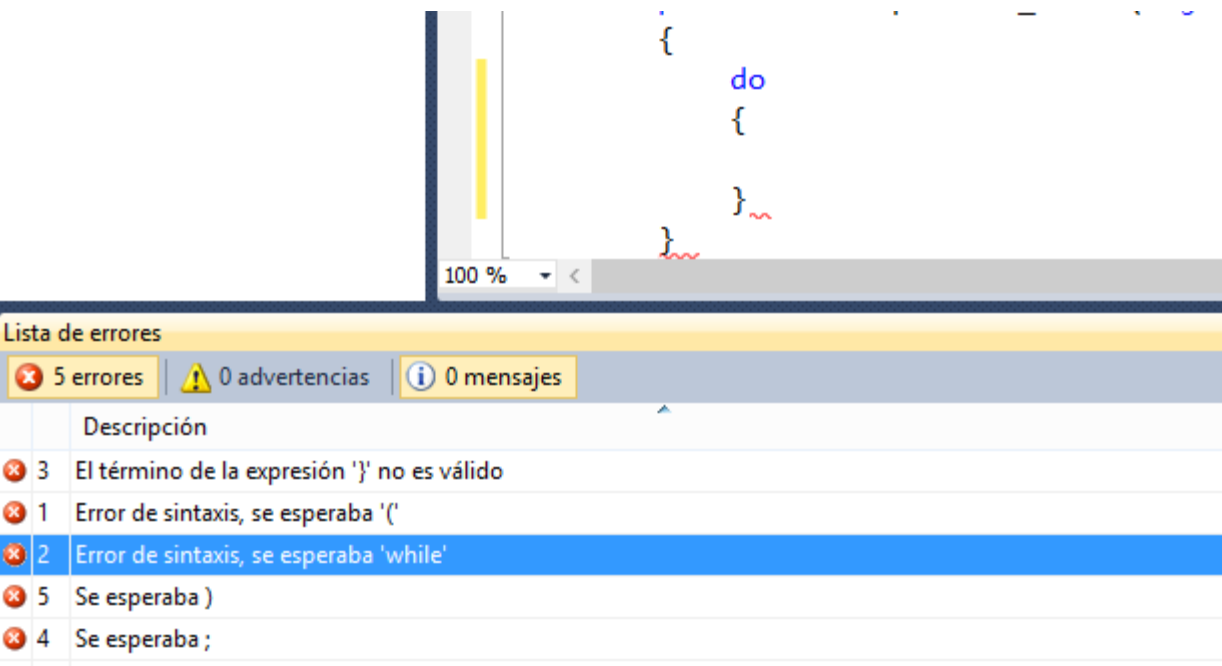
2. Tipos de errores

En general distinguiremos entre errores **en tiempo de diseño**, no podremos ejecutar la aplicación hasta corregirlos; y errores en tiempo de ejecución, es decir, sólo sabremos que hay algo mal cuando el ejecutable esté funcionando u obtengamos resultados no esperados.

2.1. Errores de escritura

Son los de localización más inmediata, ya que se producen por un error sintáctico al escribir nuestro código, y gracias al IDE de Visual Studio, podemos detectarlos rápidamente.

Cuando escribimos una sentencia incorrectamente, dejamos algún paréntesis sin cerrar, etc., el IDE subraya la parte de código errónea, y nos muestra un mensaje informativo del error al situar el cursor del ratón sobre el mismo. En el ejemplo siguiente hemos declarado una **estructura do** que no hemos cerrado con la correspondiente instrucción `} While...` ; por lo tanto, el IDE nos lo indica.



2.2. Errores de ejecución

Este tipo de errores son los que provocan un fallo en la ejecución del programa y su interrupción. No obstante, si utilizamos los gestores de error que proporciona la herramienta de desarrollo correspondiente, podremos en algunos casos, evitar la cancelación de la ejecución, recuperando su control.

El ejemplo siguiente provoca un error, ya que se intenta asignar un valor que no corresponde al tipo de dato de una variable.

```
int i;  
i = "hola";|
```

class System.String
(La caché de documentación todavía se está construyendo. Vuelva a intentarlo en unos segundos).

Error:
No se puede convertir implícitamente el tipo 'string' en 'int'

2.3. Errores lógicos



Estos errores son los de más difícil captura, ya que el código se encuentra correctamente escrito, produciéndose el problema por un fallo de planteamiento en el código, motivo por el cual, por ejemplo, el control del programa no entra en un bucle porque una variable no ha tomado determinado valor; el flujo del programa sale antes de lo previsto de un procedimiento, al evaluar una expresión que esperábamos que tuviera un resultado diferente, etc.

3. Errores y Excepciones

Dentro del esquema de gestión de errores del entorno .NET Framework, encontramos las figuras del error y la excepción. Estos elementos son utilizados indistintamente en muchas ocasiones para hacer referencia genérica a los errores producidos; sin embargo, aunque complementarios, cada uno tiene su propia funcionalidad dentro del proceso de tratamiento de un error.

- **Error.** Un error es un evento que se produce durante el funcionamiento de un programa, provocando una interrupción en su flujo de ejecución. Al producirse esta situación, el error genera un objeto excepción.
- **Excepción.** Una excepción es un objeto generado por un error, que contiene información sobre las características del error que se ha producido.

Nuestra labor consistirá en que cuando se produzca un error, seamos capaces de manejar la excepción lanzada y para que así no se aborte mi programa de una forma incontrolada.

 CIFF VIRGEN DE GRACIA	<div>TEMA 7</div> <div>Tratamiento de Errores</div>		 Página 4 de 11
--	---	--	---

3.1. Manipulador de excepciones

Un manipulador de excepciones es un bloque de código que proporciona una respuesta al error que se ha producido, y que se incluye en una estructura proporcionada por el lenguaje a tal efecto, es decir, para la captura de excepciones.

4. Formas de interceptar los errores

Tened en cuenta que no se pueden mezclar en un mismo procedimiento las dos formas de tratar los errores.

4.1. Control no estructurado de errores

Si conoces las versiones anteriores de Visual Basic, sabrás que la captura de errores se hacía con las instrucciones **On Error**. Esa forma de tratar o interceptar errores no es la recomendable y es una forma, que si no se usa correctamente, nos puede ser más perjudicial que beneficiosa.

Es un sistema que está basado en la detección y captura de errores a través de etiquetas de código, mediante saltos no estructurados en el flujo de la ejecución. (Volvemos al **GoTo**)

4.2. Control estructurado de errores

En este tipo de tratamiento, cada vez que se produce un error, se genera un objeto de la clase **Exception** o alguna de sus derivadas, conteniendo la información del error ocurrido. La manera de capturar este tipo de objetos pasa por utilizar una estructura de control del lenguaje, proporcionada para esta finalidad.

5. Estructura Try...Catch...Finally



La forma de usar esta estructura será algo así:

```

try
{
    // El código sensible a errores
}
catch // tipo de error a capturar
{
    // código a ejecutar cuando se produzca un error
}
finally
{
    // código que se ejecutará se produzca o no un error
}

```

En primer lugar nos encontramos con su declaración mediante la palabra clave **Try**. Todo el código que escribimos a partir de dicha palabra clave, y hasta la **primera** sentencia Catch, es el código que definimos como **sensible a errores**, o dicho de otro modo, el bloque de instrucciones sobre las que deseamos que se active el control de errores cuando se produzca algún fallo en su ejecución.

 CIFF VIRGEN DE GRACIA	<div style="text-align: center;"> TEMA 7 Tratamiento de Errores </div>		 Página 5 de 11
--	---	--	---

A continuación, establecemos cada uno de los manipuladores de excepción mediante la palabra clave **Catch**. Junto a esta palabra clave, situaremos de **forma opcional**, un identificador que contendrá el objeto con la excepción generada.

Podemos escribir uno o varios manipuladores Catch dentro de una estructura de control Try. Cada vez que se produzca un error, el flujo de la ejecución saltará a la sentencia Catch más acorde con el tipo de excepción generada por el error, siempre y cuando hayamos situado varios manipuladores de excepciones en el controlador de errores.

También es posible utilizar Catch de un modo genérico, es decir, sin establecer qué tipo de excepción se ha producido. Este es el tipo de control de errores más sencillo que podemos implementar, aunque también el más limitado, ya que sólo podemos tener un manipulador de excepciones.

Tanto si se produce un error como si no, la sentencia **Finally** de la estructura Try, nos permite escribir un bloque de código que será ejecutado al darse una condición de error, o bajo ejecución normal del procedimiento. Es decir, el bloque Finally siempre es ejecutado.

Concretando:

- En el bloque Try pondremos el código que puede que produzca un error.
- Los bloques Catch y Finally no son los dos obligatorios, pero al menos hay que usar uno de ellos, es decir o usamos Catch o usamos Finally o, usamos los dos, pero como mínimo uno de ellos.
- Si usamos Catch, esa parte se ejecutará si se produce un error, es la parte que "capturará" el error.
- Después de Catch podemos indicar el tipo de error que queremos capturar, incluso podemos usar más de un bloque Catch, si es que nuestra intención es detectar diferentes tipos de errores.
- En el caso de que sólo usemos Finally, tendremos que tener en cuenta de que si se produce un error, el programa se detendrá de la misma forma que si no hubiésemos usado la detección de errores.
- Es más que recomendable que siempre usemos una cláusula Catch, aunque en ese bloque no hagamos nada, pero aunque no "tratemos" correctamente el error, al menos no se detendrá porque se haya producido el error. No ejecutará esa instrucción, pero tampoco se detendrá el programa, seguirá con las instrucciones del bloque Finally, en caso de que exista, y posteriormente, con las instrucciones que siguen bloque Try.

6. La clase Exception

Cada vez que se produce un error, el entorno de ejecución genera una excepción con la información del error provocado.

Para facilitarnos la tarea de manipulación de la excepción producida, en un controlador de excepciones obtenemos un objeto de la clase Exception, o de alguna de sus derivadas, de

forma que, a través de sus miembros, podemos saber qué ha pasado. Entre las propiedades y métodos que podemos utilizar, se encuentran las siguientes.

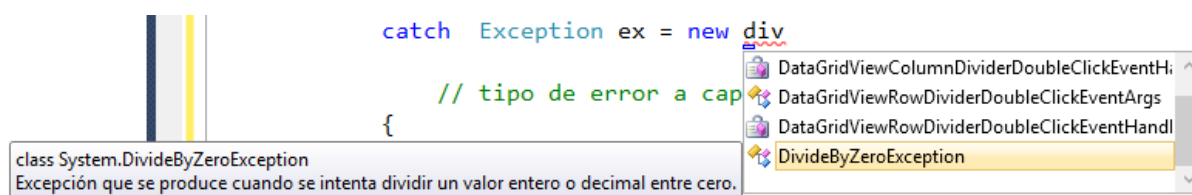
- **Message.** Descripción del error.
- **Source.** Nombre del objeto o aplicación que provocó el error.
- **StackTrace.** Ruta o traza del código en la que se produjo el error.
- **ToString().** Devuelve una cadena con información detallada del error. En esta cadena podemos encontrar también, los valores obtenidos de las propiedades anteriores; por lo que el uso de este método, en muchas ocasiones será el modo más recomendable para obtener los datos de la excepción.

Podemos obtener el objeto de excepción creado a partir de un error, utilizando la sentencia **Catch** de la estructura **Try**. Para ello, a continuación de **Catch**, escribimos el nombre de un identificador, definiéndolo como tipo **Exception** o alguno de los tipos de su jerarquía.

Exception representa la clase base en la jerarquía de tipos de excepción que se pueden producir dentro del entorno de ejecución. Partiendo de **Exception**, disponemos de un conjunto de clases derivadas, que nos permiten un tratamiento más particular del error producido, como **ApplicationException**, **IOException**, **SystemException**, etc. Cada una de ellas puede tener, además de los miembros generales de **Exception**, una serie de métodos y propiedades particulares de su tipo de excepción; por ello, lo más conveniente, será utilizar estas clases, a través de las que podremos averiguar más detalles sobre el problema producido.

No es fácil averiguar qué tipos de excepciones tenemos, aunque es fácil de identificarlas, porque todas ellas terminan en **Exception**.

- Una forma sería cuando estamos declarando la variable:



... y otra forma, sería recurrir a la ayuda, a su espacio de nombre **System**, y ver qué **Excepciones** existen. Seleccionas la palabra **Exception** y pulsa **F1** que te lleve a la ayuda de msdn.microsoft.com













En la tabla que se muestra encontrarás todos los tipos de excepciones que hay y la definición de los errores que capturan.

System (Espacio de nombres)

.NET Framework 4 | [Otras versiones](#) | Personas que lo han encontrado útil: 3 de 3 - [Valorar este tema](#)

Contiene clases que le permiten hacer coincidir con los URI con plantillas URI y grupos de plantillas URI.

▲Clases

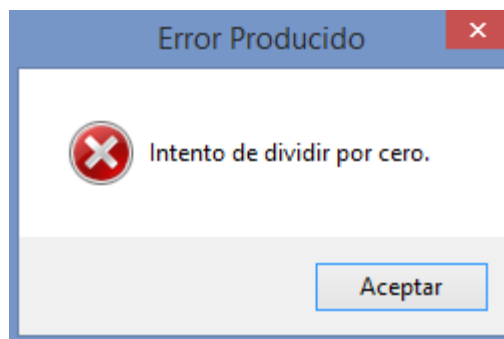
	Clase	Descripción
	AccessViolationException	Excepción que se produce cuando se intenta leer o escribir en la memoria protegida.
	ApplicationContext	Identifica el contexto de activación de la aplicación actual. Esta clase no puede heredarse.
	Activator	Contiene métodos para crear tipos de objetos de forma local o remota, o para obtener referencias a objetos remotos existentes. Esta clase no puede heredarse.
	AggregateException	Representa uno o varios errores que se producen durante la ejecución de una aplicación.
	AppDomain	Representa un dominio de aplicación, que es un entorno aislado donde se ejecutan las aplicaciones. Esta clase no puede heredarse.
	AppDomainManager	Proporciona un equivalente administrado de un host no administrado.
	AppDomainSetup	Representa la información de enlace del ensamblado que puede agregarse a una instancia de AppDomain .
	AppDomainUnloadedException	Excepción que se produce al intentar obtener acceso a un dominio de aplicaciones descargado.
	ApplicationException	Excepción que se produce cuando se produce un error de aplicación que no es grave.
	ApplicationId	Contiene información utilizada para identificar de forma única una aplicación basada en manifiesto. Esta clase no puede heredarse.
	ApplicationIdentity	Permite identificar de manera única una aplicación activada por manifiesto. Esta clase no puede heredarse.
	ArgumentException	Excepción que se produce cuando no es válido uno de los argumentos proporcionados para un método.

Cuando en el código de un controlador de errores puedan producirse errores de distintos tipos de excepción, debemos situar tantas sentencias Catch como excepciones queramos controlar.

Ej. Controlando si el error es al dividir por cero

```
try
{
    // El código sensible a errores
    i = i / j;
}
catch (DivideByZeroException ex)// tipo de error a capturar
{
    // código a ejecutar cuando se produzca un error
    MessageBox.Show(ex.Message, "Error Producido", MessageBoxButtons.OK, MessageBoxIcon.Error);
}
catch (Exception ex)    // para cualquier otro error
{
    // código a ejecutar cuando se produzca un error
    MessageBox.Show(ex.Message, "Error Producido", MessageBoxButtons.OK, MessageBoxIcon.Error);
}
-- --
```

Lo que veríamos en pantalla al producirse el error sería lo siguiente:



Veamos un ejemplo muy simple, en el que tendremos un formulario con dos cuadros de texto, una etiqueta y un botón de comando. De forma que cuando el usuario haga clic sobre el botón, divida el contenido del primer cuadro de texto por el del segundo.

```
private void btnDividir_Click(object sender, EventArgs e)
{
    int i, j, k;
    try
    {
        i = int.Parse(this.txtA.Text);
        j = int.Parse(this.txtB.Text);
        k = i / j;
        this.txtRdo.Text = k.ToString();
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.ToString(), "Error Producido", MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
}
```

Ejercicio práctico

Modifica el código anterior para que en vez del mensaje de: “Error Debes introducir datos numéricos”, me muestre el mensaje propio que me da la excepción producida.

Averigua que excepción es la que realmente se ocasiona cuando se dejan los cuadros de textos en blanco, por ejemplo.

6.1. Lanzar nuevas excepciones. Throw

Con la palabra clave **throw** lo que hacemos es lanzar una nueva excepción, o dicho de otra forma, provocar una excepción para que el código salte al catch.

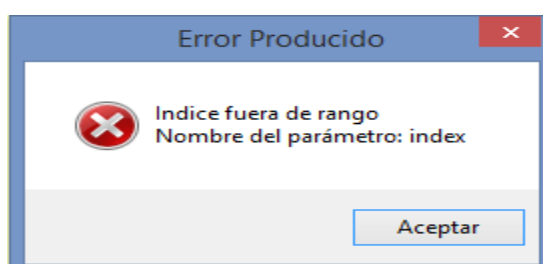
6.1.1. Modificando el ArgumentException

Ej: Tenemos el botón **btnVector** que llama al siguiente método... como observas tenemos un vector de 3 enteros y queremos que nos muestre el `nums[4]`, lo cual estamos generando una excepción por exceder los límites del vector. La llamada nos devolverá un error, cuyo mensaje de excepción lo hemos generado nosotros a través de **ArgumentException**:

```
private void btnVector_Click(object sender, EventArgs e)
{
    int[] nums = { 300, 600, 900 };
    //
    try
    {
        this.txtRdo.Text = valorVector(nums, 4).ToString();
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message, "Error Producido", MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
}

static int valorVector(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (System.IndexOutOfRangeException ex)
    {
        System.ArgumentException argEx = new System.ArgumentException("Indice fuera de rango", "index", ex);
        throw argEx;
    }
}
```

Mensaje que mostrará:



6.1.2. Creando nuestras propias excepciones

Aunque existen muchos tipos de objetos derivados de Exception, son muchos los programas que se pueden realizar y muchas las posibles causas de error que se pueden plantear. Por ello en ocasiones es necesario que creamos **nuestras propias clases de error**.

Las clases de error son similares a las ya vistas en los temas 2 y 3, solo que en este caso, éstas derivan de **ApplicationException**, con lo que nos encontraremos:

```
namespace Tema7
{
    class ErrorLoginException: ApplicationException
    {
        // Código de la clase
    }
}
```

Supongamos que estamos desarrollando una aplicación que tiene protegido el acceso mediante un formulario que solicita un “login” y un “password”. En caso de que el usuario no introduzca correctamente login y password, o alguno de ellos, queremos que se produzca un error. Para ello, el proceso a seguir será:

- 1.- Creamos una nueva clase en nuestro proyecto
- 2.- Modificamos el encabezamiento para hacerla derivar de la clase ApplicationException.
- 3.- En la clase, tendremos una variable “mensaje” donde almacenar la descripción que queremos darle al usuario sobre el error producido,
- 4.- Configuraremos al menos un método constructor que inicialice el mensaje de error
- 5.- Sobrescribimos la propiedad Message para que muestre nuestro mensaje.

```
namespace Tema7
{
    class ErrorLoginException: ApplicationException
    {
        // Código de la clase
        string mensaje;
        public ErrorLoginException(string mensaje)
        {
            this.mensaje = mensaje;
        }
        public string Message // Estamos Sobrescribiendo este método
        {
            get
            {
                return this.mensaje;
            }
        }
    }
}
```

Desde el formulario de entrada:



Modificamos la Aplicación de la División por Cero (Tema7) para que este sea el formulario inicial.

El Login será "lucas" y la Password "1234".

Al pulsar Enter se ejecutará "Aceptar" y al pulsar Escape se ejecutará "Salir".

Y si el login y la clave son correctas, se visualizará únicamente el otro formulario.

Y ya simplemente como hemos hecho con el resto de excepciones...

```
static string LOGIN = "lucas";
static string PWD = "1234";

private void btnAceptar_Click(object sender, EventArgs e)
{
    Form1 fr = new Form1();
    try
    {
        if ((!this.txtLogin.Text.Equals(LOGIN)) || (!this.txtPwd.Text.Equals(PWD)))
            throw new ErrorLoginException("Login y/o Password incorrectos...");
        else
        {
            this.Visible = false; // ojo! Al hacerlo invisible: fr debe EXIT de la aplicación
            fr.Show();
        }
    }
    catch (ErrorLoginException ex)
    {
        MessageBox.Show(ex.Message, "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
}
```