Tema 6 – Manejo de excepciones

TRATAMIENTO DE ERRORES

Los programas pueden poseer errores, algunos de los cuales son detectados en tiempo de compilación (errores de sintaxis, semánticos, etc.) pero otros podrían permanecer inadvertidos por ser errores de lógica o requerir ciertas condiciones para manifestarse.

C# ofrece un sistema para manejar este tipo de situaciones, heredado del lenguaje C++, que se denomina "manejo de excepciones" (exception handling).

Una excepción es una situación anormal que se da a lo largo de la ejecución del programa. El manejador de la excepción, es una estructura de control que permite ejecutar un código de acuerdo a esa situación anormal de forma controlada.

Las excepciones se usan principalmente para:

- Separar el código principal del programa, del tratamiento de errores
- Propagar el error en la pila de ejecución.
- Agrupar y clasificar los errores

Ejemplos de errores no controlados en tiempo de ejecución:

```
static void Main(string[] args){
      Console.WriteLine("Ingrese un numero: ");
      int num = Convert.ToInt32(Console.ReadLine());
      num += 2;
      Console.WriteLine(num);
}
```

En el ejemplo anterior al introducir un número cualquiera no ocurre ningún error pero al introducir una letra se lanza una excepción.

Ingrese un numero:
a
Unhandled Exception: System.FormatException: Input string was not in the correct format
at System.Int32.Parse (System.String s) [0x00000]
at System.Convert.ToInt32 (System.String value) [0x00000]

Lo que ocurrió fue que el método estático ToInt32 de la clase Convert arrojó una excepción. Éste es un modo de avisar que algo salió mal, por lo tanto se debería de hacer algo al respecto o la aplicación no podrá seguir su curso normal.

El siguiente ejemplo lanza un error no controlado en tiempo de ejecución llamado DivideByZeroException al intentar realizar una división por cero.

```
static int Division(int x, int y){
      return (x/y);
}
static void Main(string[] args){
      Console.WriteLine(Division(20, 0));
}
```

Controlar las excepciones

Es posible capturar las excepciones que puedan arrojar los diversos métodos que se invoquen. Para esto se encierra el código que podría arrojar la excepción en un bloque **try/cach**. Los bloques try se utilizan para separar el código al que puede afectar una excepción y los bloques catch para controlar las excepciones resultantes.

```
Console.WriteLine("Ingrese un numero: ");
try{
	int num = Convert.ToInt32(Console.ReadLine());
	num += 2;
	Console.WriteLine(num);
}catch{
	Console.WriteLine("se ha producido una excepcion");
}
```

Este es el método más básico de manejar una excepción. Ahora, si se llega a producir alguna dentro del código del bloque try, el flujo de ejecución saltará automáticamente a la primera línea del bloque catch. Un bloque try se debe utilizar con un bloque catch, y puede incluir varios bloques catch.

Bloque finally

Siempre se debe ejecutar cierta limpieza de recursos, como el cierre de un archivo, incluso si se produce una excepción. Para lograr esto, se puede usar un bloque finally. Los bloques finally se ejecutan siempre, independientemente de si se produce una excepción o no.

```
Console.WriteLine("Ingrese un numero: ");
try{
	int num = Convert.ToInt32(Console.ReadLine());
	num += 2;
	Console.WriteLine(num);
}catch{
	Console.WriteLine("se ha producido una excepcion");
}finally{
	Console.WriteLine("este bloque siempre se ejecuta");
}
```

Clases de excepciones

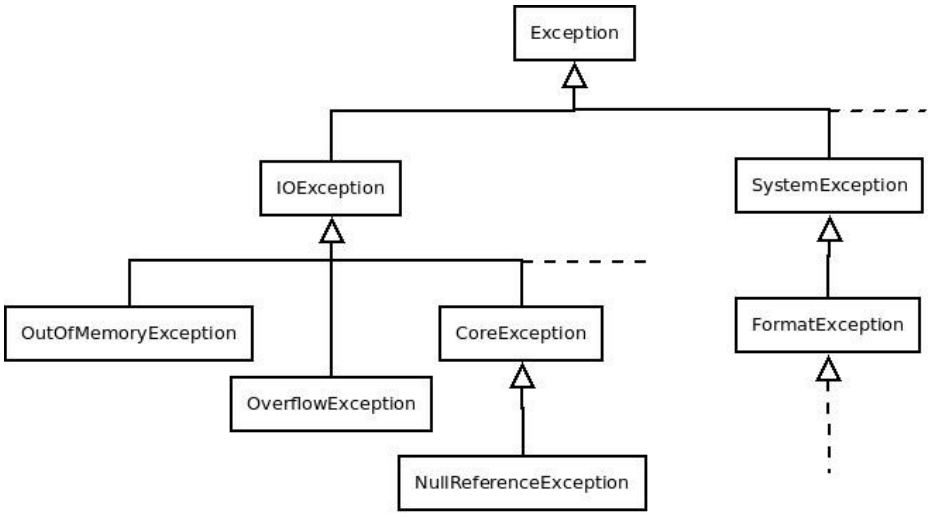
El bloque catch puede recibir un objeto de algún tipo de excepción, esto permite informar del error producido.

```
try{
	int num = Convert.ToInt32(Console.ReadLine());
	num += 2;
	Console.WriteLine(num);
}catch(FormatException ex){
	Console.WriteLine(ex.Message);
}
```

Al introducir una letra el bloque catch recibe un tipo de objeto que mediante la propiedad Message se imprime su información, en este caso sería:

Input string was not in the correct format

Las clases de excepción en .Net descienden de la clase Exception, la cual ofrece un conjunto de propiedades y métodos que ayudan en la obtención de mayor información y en un mejor tratamiento de la ocurrencia que generó la excepción. Existen muchas clases descendientes de Exception.



En caso de que se produzca una excepción de tipo FormatExeption, se podrá capturar especificando como parámetro un objeto del tipo FormatExeption o un objeto de tipo superior.

```
try{
	int num = Convert.ToInt32(Console.ReadLine());
	num += 2;
	Console.WriteLine(num);
}catch(Exception ex){
	Console.WriteLine(ex.Message);
}
```

La idea de este sistema cobra sentido cuando pueden existir múltiples bloques catch por cada bloque try, y el flujo de ejecución se desviará al que corresponda según la excepción generada.

En el siguiente ejemplo se podrían producir dos excepciones, una de tipo FormatException y otra de tipo DivideByZeroException. Es por ello que se usan dos bloques catch, cada uno controla una excepción.

```
public static void Main(){
	int num, res;
	Console.WriteLine("Ingrese un numero: ");
	try{
		num = Convert.ToInt32(Console.ReadLine());
		res = 5 / num;
		Console.WriteLine(res);
	}catch(FormatException ex){
		Console.WriteLine(ex.Message);
	}catch(DivideByZeroException ex){
		Console.WriteLine(ex.Message);
	}
}
```

Se pueden especificar tantos bloques catch como tipos de excepciones que se quieran controlar. La única restricción es que no se puede especificar un bloque catch con una clase de excepción que sea base de la clase de excepción de otro catch que se encuentre debajo de él. **Para bloques catch consecutivos el orden siempre es de más específico a más general.**

```
public static void Main(){
	int num, res;
	Console.WriteLine("Ingrese un numero: ");
	try{
		num = Convert.ToInt32(Console.ReadLine());
		res = 5 / num;
		Console.WriteLine(res);
	}catch(FormatException ex){
		Console.WriteLine(ex.Message);
	}catch(Exception ex){
		Console.WriteLine(ex.Message);
	}
}
```

La clase System.Exception

Como ya se ha dicho, todas las excepciones específicas derivan de un tipo de clase llamada System.Exception. Los principales miembros que se heredan son:

string Message {virtual get;}: Contiene un mensaje descriptivo de las causas de la excepción. Por defecto este mensaje es una cadena vacía (“”)

Exception InnerException {virtual get;}: Si una excepción fue causada como consecuencia de otra, esta propiedad contiene el objeto System.Exception que representa a la excepción que la causó.

string StackTrace {virtual get;}: Contiene la pila de llamadas a métodos que se tenía en el momento en que se produjo la excepción. Esta pila es una cadena con información sobre cuál es el método en que se produjo la excepción, cuál es el método que llamó a este, cuál es el que llamó a ese otro, etc.

string Source {virtual get; virtual set;}: Almacena información sobre cuál fue la aplicación u objeto que causó la excepción.

MethodBase TargetSite {virtual get;}: Almacena cuál fue el método donde se produjo la excepción en forma de objeto System.Reflection.MethodBase.

string HelpLink {virtual get;}: Contiene una cadena con información sobre cuál es la URI donde se puede encontrar información sobre la excepción. El valor de esta cadena puede establecerse con virtual Exception SetHelpLink (string URI), que devuelve la excepción sobre la que se aplica pero con la URI ya actualizada.

Para crear objetos de clase System.Exception se puede usar los constructores:

Exception()
Exception(string msg)
Exception(string msg, Exception causante)

El primer constructor crea una excepción cuyo valor para Message será “” y no causada por ninguna otra excepción (InnerException valdrá null) El segundo la crea con el valor indicado para Message, y el último la crea con además la excepción causante indicada.

En la práctica, cuando se crean nuevos tipos derivados de System.Exception no se suele redefinir sus miembros ni añadirles nuevos, sino que sólo se hace la derivación para distinguir una excepción de otra por el nombre del tipo al que pertenecen. Ahora bien, es conveniente respetar el convenio de darles un nombre acabado en Exception y redefinir los tres constructores antes comentados.

Excepciones predefinidas comunes

En el espacio de nombres System hay predefinidas múltiples excepciones derivadas de System.Exception que se corresponden con los errores más comunes que pueden surgir durante la ejecución de una aplicación. En la Tabla se recogen algunas:

Excepción	Descripción
ArithmeticException	Clase base de las excepciones producidas durante operaciones aritméticas, como DivideByZeroException y OverflowException.
ArrayTypeMismatchException	Asignación a array de elemento que no es de su tipo.
DivideByZeroException	dividir un valor por cero.
IndexOutOfRangeException	Índice de acceso a elemento de array fuera del rango válido (menor que cero o mayor que el tamaño de la tabla)
InvalidCastException	Conversión explícita entre tipos no válida.
NullReferenceException	Se produce al intentar hacer referencia a un objeto cuyo valor es null.
OutOfMemoryException	Falta de memoria para crear un objeto con new.
OverflowException	Se produce cuando una operación aritmética en un contexto checked produce un desbordamiento.
StackOverflowException	Se produce cuando se agota la pila de excepciones debido a la existencia de demasiadas llamadas al método pendientes.
TypeInitializationException	Se produce cuando un constructor estático produce una excepción sin que haya cláusulas catch compatibles para capturarla.
ArgumentNullException	Pasado argumento nulo
ArgumentException	Pasado argumento no válido (base de excepciones de argumentos).
ArgumentOutOfRangeException	Pasado argumento fuera de rango

En el ejemplo de código siguiente se usa un bloque try/catch para detectar una InvalidCastException. En el ejemplo se crea una clase denominada Employee con una única propiedad, el nivel de empleado (Emlevel). Un método, PromoteEmployee, toma un objeto e incrementa el nivel del empleado. Se produce una InvalidCastException cuando se pasa una instancia de DateTime al método PromoteEmployee.

```
class Employee {
	private int emlevel;
	public int Emlevel{
		get{  return(emlevel); }
		set{  emlevel = value; }
	}
}
class Ex13 {
	public static void PromoteEmployee(Object emp) {
		// Cast object to Employee.
		Employee e = (Employee) emp;
		// Increment employee level.
		e.Emlevel = e.Emlevel + 1;
	}
	public static void Main() {
		try {
			Object o = new Employee();
			DateTime newyears = new DateTime(2001, 1, 1);
			// Promote the new employee.
			PromoteEmployee(o);
			// Promote DateTime; results in InvalidCastException
			// as newyears is not an employee instance.
			PromoteEmployee(newyears);
		}  catch (InvalidCastException ex) {
			Console.WriteLine(ex.Message);
		}
	}
}
```

Lanzamiento de excepciones. Instrucción throw

Se puede lanzar una excepción de manera explícita utilizando la instrucción **throw**.

```
class Program {
	public static void Calcula(int num) {
		if (num % 2 == 0)
			throw new FormatException("El numero sebe ser impar");
		else
			Console.WriteLine(num);
	}
	public static void Main() {
		int num;
		try {
			num = Convert.ToInt32(Console.ReadLine());
			Calcula(num);
		}  catch (FormatException ex) {
			Console.WriteLine(ex.Message);
		}
	}
}
```

En el ejemplo anterior, la sentencia throw va acompañada por el objeto de tipo excepción que mejor se adapte al tipo de error que ha ocurrido. Cuando se ejecuta throw, se abandona el método y se produce la excepción.

```
class Program {
	public static double SafeDivision(double x, double y) {
		if (y == 0)
			throw new DivideByZeroException("Intentas dividir entre cero.");
		return x / y;
	}

	public static void Main() {
		double a = 98, b = 0;
		double result;
		try{
			result = SafeDivision(a, b);
			Console.WriteLine("{0} dividido entre {1} = {2}", a, b, result);
		}catch (DivideByZeroException ex){
			Console.WriteLine(ex.Message);
		}
	}
}
```

En el ejemplo de código siguiente se usa un bloque try/catch para detectar una posible FileNotFoundException. Después del bloque Try hay un bloque Catch que detecta FileNotFoundException y escribe un mensaje en la consola si no se encuentra el archivo de datos.

```
public static void Main() {
	FileStream fs = null;
	try {
		// open file
		fs = new FileStream("data.txt", FileMode.Open);
		StreamReader sr = new StreamReader(fs);
		string line;
		// Read from the file and output to the console.
		line = sr.ReadLine();
		Console.WriteLine(line);
	}  catch(FileNotFoundException ex) {
		Console.WriteLine("[Data File Missing] {0}", ex.Message);
	}  finally {
		if (fs != null){
			fs.Close();
		}
	}
}
```

Overthrowing: Número excesivo de excepciones.

En ocasiones ocurre que el código provoque que se lancen numerosas excepciones. Las excepciones son costosas en tiempo de ejecución.

Un ejemplo de esta situación, bastante habitual, ocurre cuando se necesita realizar una conversión de una cadena, leída desde un archivo, a un valor numérico.

Suponiendo que se tenga un archivo que contiene gran cantidad de cadenas, en las que unas contienen solo un número y otras no. Se desea tratar las cadenas de solo números y convertirlos en tipos enteros descartando las demás. Si la cadena, leída del archivo, esta contenida en una variable de tipo string *cadena*, parece lógico tener un código como el siguiente:

```
try{
      int i = Int32.Parse(cadena);
      Console.WriteLine(i);
}  catch (FormatException ex) {
      Console.WriteLine(ex.Message);
}
```

En principio no hay problema con este código, pero si hay un gran porcentaje de cadenas que no son números, esto provocará que se produzcan un gran número de excepciones lo que penalizará en gran medida el rendimiento de la aplicación.

La solución a este caso pasa por comprobar esta situación sin recibir una excepción. El método **TryParse** sirve para verificar si la conversión se hace correctamente o no. **Este método recibe como parámetros un tipo string y un parámetro de salida del tipo que se quiera convertir. Devuelve boolean indicando si la conversión ha sido exitosa o no.**

public static bool TryParse(string s, out int result)

El código anterior quedaría de la siguiente manera:

```
int num;
if(Int32.TryParse(cadena, out num)){
      Console.WriteLine("Converted '{0}' to {1}.", cadena, num);
}  else {
      Console.WriteLine("Attempted conversion of '{0}' failed.", cadena);
}
```

Ahora se provee de un método alternativo que permite realizar la misma operación sin recibir excepciones.

Solo las situaciones excepcionales deben comunicarse usando excepciones, no valores de retorno, esta es la norma general y solo debe implementarse un mecanismo alternativo si hay escenarios en los que se vayan a producir un número excesivo de excepciones.

Overcatching: Recoger más tipos de excepción que lo que interesa.

Para este problema se deben capturar tipos de excepción concretos que se encuentren entre los que es previsible que se produzcan y evitar enmascarar el resto. Aún así existen casos en los que capturar excepciones de manera genérica, llamadas a recursos remotos, en la carga de plugins de terceros, etc.

Log de errores

Las clases para el diagnostico y generación de un log para una aplicación están en el namespace **System.Diagnostics**, de manera que se puede registrar información sobre la excepción que se ha producido. Siempre se deben agrupar los mensajes de diagnostico para una mejor legibilidad.

```
using System.Diagnotics;
...
Trace.WriteLine("Error en acceso a fichero\nLa ruta del fichero es muy larga");
```

Control de errores en aplicación con interfaz

Un método para poder controlar excepciones de una aplicación sin abusar de try/catch es usando el evento llamado **Application.ThreadException**, que controla todas aquellas excepciones que se produzcan.

```
using System;
using System.Windows.Forms;
using System.Threading;
class MiPrograma {
	// Este atributo debe estar presente en el punto de entrada de
	// cualquier aplicacion que utilice Formularios Windows Forms;
	// si se omite, los componentes de Windows podrian no funcionar
	// correctamente.
	[STAThread]
	private static void Main(string[] args) {
		Application.ThreadException +=
			new ThreadExceptionEventHandler(MiPrograma.excepcion);
		Application.Run(new Formulario());
	}
	public static void excepcion(Object sender,
					ThreadExceptionEventArgs excepcion) {
		// enviar log al desarrollador con info del error
		// informar al usuario del error
		MessageBox.Show(excepcion.Exception.Message);
	}
}
```

La clase **Application** proporciona métodos y propiedades static para administrar una aplicación, como métodos para iniciar y detener una aplicación, para procesar los mensajes, y propiedades para obtener información sobre una aplicación. Esta clase no puede heredarse. Esta clase esta dentro del namespace System.Windows.Forms

El método excepcion enlazado al evento es lanzado al producirse la excepción en la aplicación. Dentro de este método se puede optar por varias opciones como se describe en el ejemplo.

Esto permite poder controlar tipos de errores conocidos como controlando aquellos no tan comunes.

El ejemplo completo incluyendo la parte de la interfaz de usuario en Windows.Forms

```
using System;
using System.Windows.Forms;
using System.Threading;

namespace ejemplo_06 {
	public class Formulario : Form{

		private Button button1;
		private TextBox textNum;

		public Formulario(){
			// dimensiones de ventana
			this.Width = 200;
			this.Height = 70;
```

```
                button1 = new Button();
                button1.Left = 110;
                button1.Top = 5;
                button1.Text = "operar";
                button1.Click += new EventHandler(button1_Click);

                // Caja de entrada de texto
                textNum = new TextBox();
                textNum.Top = 5;
                textNum.Left = 5;

                this.Controls.Add(button1);
                this.Controls.Add(textNum);
        }

        private void button1_Click(Object sender, EventArgs e){
                int num = Convert.ToInt32(textNum.Text);
                float result = (float)200/num;
                string res = Convert.ToString(result);
                MessageBox.Show(res);
                Application.Exit();
        }
    }

    class MiPrograma {
        // Este atributo debe estar presente en el punto de entrada de
        // cualquier aplicacion que utilice Formularios Windows Forms;
        // si se omite, los componentes de Windows podrian no funcionar
        // correctamente.
        [STAThread]
        private static void Main(string[] args) {
                Application.ThreadException +=
                        new ThreadExceptionEventHandler(MiPrograma.excepcion);
                Application.Run(new Formulario());
        }
        public static void excepcion(Object sender,
                                     ThreadExceptionEventArgs excepcion) {
                // enviar log al desarrollador con info del error
                // informar al usuario del error
                MessageBox.Show(excepcion.Exception.Message);
        }
    }
}
```

Crear excepciones definidas por el usuario

Para crear excepciones propias hay que heredar de alguna excepción del tipo Exception o descendiente de ésta.

```
class NumImparException : FormatException{
    // Se definen dos constructores en la clase, cada
    // uno con parametros diferentes.
    public NumImparException() : base("numero no impar"){}
    public NumImparException(string err) : base(err){}
}
```

Para lanzar y controlar la excepción ahora se podría hacer de la siguiente manera:

```
class MiPrograma {
	public static void Calcula(int num) {
		if (num % 2 == 0)
			throw new NumImparException();
		else
			Console.WriteLine(num);
	}
	public static void Main() {
		int num;
		try {
			num = Convert.ToInt32(Console.ReadLine());
			Calcula(num);
		}  catch (NumImparException ex) {
			Console.WriteLine(ex.Message);
		}  catch (Exception ex) {
			Console.WriteLine(ex.Message);
		}
	}
}
```

Resumen

- Una excepción es el indicador de un problema: una mala operación o situación anómala que ocurre durante la ejecución de un programa.
- Las excepciones deben ser interceptadas y manejadas para no desestabilizar el programa.
- Todas las excepciones son instancias de System.Exception o de sus derivadas.
- La librería base provee un número amplio de excepciones predefinidas.
- Se pueden crear excepciones personalizadas derivando de System.Exception.

Lanzamiento de excepciones

- Evitar excepciones para casos normales o esperados.
- Nunca crear ni lanzar objetos de clase Exception.
- Incluir una cadena de descripción en un objeto Exception.
- Lanzar objetos de la clase más específica posible.

Captura de excepciones

- Ordenar los bloques catch de lo específico a lo general.
- No permitir que salgan excepciones de Main.

Actividades

1.- ¿En qué caso es mejor usar un mecanismo alternativo de control al de try/catch?

2.- ¿Que excepción lanzaría la siguiente instrucción?

```
int num = Convert.ToInt32("87887878787");
```

a) OverflowException
b) OutOfMemoryException
c) DivideByZeroException

3.- Encierra las sentencias siguientes para capturar las posibles excepciones que se produzcan. Puede haber mas de un tipo de excepción.

```
int num = Convert.ToInt32(Console.ReadLine());
Console.WriteLine(num);
```

4.- El siguiente método devuelve un valor de un elemento de un array en el índice que se indique o lanza una excepción si el índice es mayor al array. Crea el método Main() que capture esta excepción e imprima por consola la información de la excepción capturada.

```
public static int GetValueFromArray(int[] array, int index) {
	if(array.Length-1 < index ) {
		throw new IndexOutOfRangeException("index esta fuera del rango");
	}else{
		return array[index];
	}
}
```