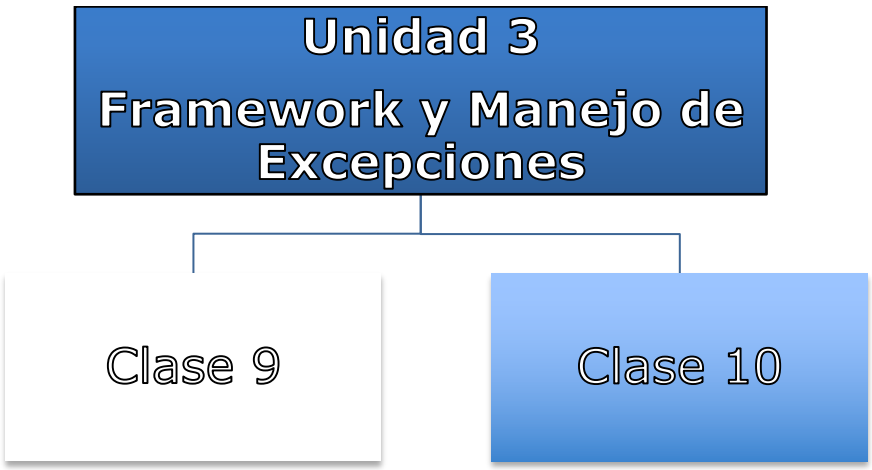
PROGRAMACIÓN ORIENTADA A OBJETOS



Docente titular y autor de contenidos: Prof. Ing. Darío Cardacci

Presentación

En esta unidad abordaremos los temas referidos a los Frameworks y en particular nos concentraremos en uno denominado .NET Framework.

En particular haremos énfasis en las partes constitutivas del Framework, los esquemas de compilación y la administración de los objetos en la memoria.

El uso de Framework en los desarrollos actuales es muy importante ya que permite obtener soluciones fiables en tiempos razonables de producción.

Esperamos que luego de analizar estos temas Ud. pueda percibir el aporte que los temas tratados le otorgan a los desarrollos de los sistemas de información.

Por todo lo expresado hasta aquí es que esperamos que usted, a través del estudio de esta unidad logre:

- Comprender la noción de framework a través del análisis de sus particularidades.
- Distinguir los distintos tipos de frameworks.
- Analizar y reconocer las particularidades del framework .NET para poder utilizarlas en el desarrollo de sistemas

Los siguientes **contenidos** conforman el marco teórico y práctico de esta unidad. A partir de ellos lograremos alcanzar el resultado de aprendizaje propuesto. En negrita encontrará lo que trabajaremos en la clase 10.

- Concepto de frameworks. Elementos de un framework. Tipos de frameworks.
- Arquitectura de .NET. Interoperatividad entre .NET y COM.
- Código administrado y no administrado.
- Common Language Runtime CLR.
- Lenguaje intermedio IL.
- El compilador Just-in-Time (JIT).
- Concepto de assembly.
- Administración de la memoria en .NET. El Garbage Collector.
- **Manejo de excepciones. Control de excepciones. El objeto Exception. La instrucción Try – Catch – Finally. La instrucción Throw.**
- **Depuración de aplicaciones. Herramientas de depuración. Análisis del comportamiento de las aplicaciones.**

A continuación, le presentamos un detalle de los contenidos y actividades que integran esta unidad. Usted deberá ir avanzando en el estudio y profundización de los diferentes temas, realizando las lecturas requeridas y elaborando las actividades propuestas, algunas de desarrollo individual y otras para resolver en colaboración con otros estudiantes y con su profesor tutor.

1. Administración de Excepciones

Lectura requerida

- Deitel Harvey M. Y Paul J. Deitel. Cómo programar en C#. Segunda edición. Pearson. México 2007. Capítulo 12

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/try-catch

Material multimedia requerido

- Material multimedia Nº 6. Administración de Memoria.pptx

Lo/a invitamos ahora a comenzar con el estudio de los contenidos que conforman esta unidad.

4 Administración de Excepciones

El manejo de excepciones del lenguaje C# y el .Net Framework proporcionan una manera de afrontar situaciones inesperadas o no deseadas, que se presentan durante la ejecución del programa.

Las palabras claves involucradas en el control de excepciones son:

try

catch

finally

when

El sistema de manejo de excepciones también permite extender lo ofrecido por el .NET framework con excepciones especializadas por el usuario. Esto permite incluir aspectos de nuestros sistemas al sistema de excepciones. Las excepciones se crean mediante la palabra clave **throw.**

En muchos casos, puede que una excepción no la produzca un método que el código ha llamado directamente, sino otro método que aparece más adelante en la pila de llamadas. Cuando esto sucede, el CLR buscará en la pila a fin de buscar un método que posea un bloque **catch** para el tipo de excepción específico. Si lo encuentra, ejecutará el primer bloque catch de este tipo de excepción que encuentre. Si no encuentra ningún bloque catch adecuado en la pila de llamadas, lo atrapará el .NET framework de manera genérica y le enviará un mensaje al usuario.

Entre las propiedades que poseen las excepciones podemos mencionar:

- Las excepciones son tipos que se derivan en última instancia de System.Exception.
- Cuando se produce una excepción dentro del bloque **try**, el flujo de control salta al primer controlador de excepciones asociado, que se encuentre en cualquier parte de la pila de llamadas. En C#, la palabra clave **catch** se utiliza para definir un controlador de excepciones.

- Si no hay un controlador de excepciones para una excepción determinada, el programa deja de ejecutarse y presenta un mensaje de error (lo realiza el framework).
- Si un bloque **catch** define una variable de excepción, puede utilizar dicho bloque para obtener más información sobre el tipo de excepción que se ha producido.
- Un programa que utiliza la palabra clave **throw** puede generar explícitamente excepciones.
- Los objetos de excepción contienen información detallada sobre el error, tal como el estado de la pila de llamadas y una descripción del error entre otros datos de interés.
- El código de un bloque **finally** se ejecuta siempre, se produzca o no una excepción. Se debe usar un bloque **finally** para liberar recursos, por ejemplo, para cerrar las secuencias o archivos que se abrieron en el bloque **try**.

Analicemos el ejemplo **Ej0003.** En él, se puede observar el uso del sistema de excepciones, considerando una excepción provista por el .Net framework.

El ejemplo **Ej0003** expone la funcionalidad para intentar dividir dos números. Esta acción se desarrolla de tres maneras distintas para forzar la generación de excepciones.

Opción 1: Se dividen dos números enteros, lo cual sucede sin problemas y solo se ejecuta el **finally.**

Opción 2: Se divide un número entero por cero, eso genera un tipo de **exception** denominada **DivideByZeroException.**

Opción 3: Se dividir un número entero por un string, generando tipo de **exception** denominada **FormatException.**

En el siguiente código se observa la implementación asociada a la opción 1.

```
1 referencia
private void button1_Click(object sender, EventArgs e)
{
    try
    {
        textBox1.Text = "1000"; textBox2.Text = "250"; textBox3.Text = "";
        textBox4.Text = "";
        textBox3.Text = (int.Parse(textBox1.Text) / int.Parse(textBox2.Text)).ToString();

    }
    catch (DivideByZeroException Ex) { MuestraError(Ex); }
    catch (Exception Ex) { MuestraError(Ex); }
    finally
    {
        textBox4.Text += NuevaLinea(1) + "--------------------------" +
        NuevaLinea(1) + "Se ejecutó finally !!!!!!!!";
    }
}
1 referencia
```

Ej0003

El código anterior muestra claramente que lo que se desea controlar se encuentra en el bloque **try.** Luego se observan dos **catch** que, en caso de generarse una excepción, se irá a aquel cuyo tipo de excepción coincida. También se observa el **finally** que es un espacio por el cual se pasará independientemente a que se produzca una excepción o no.

Los próximos dos bloques de código son similares. Se diferencian el tipo de excepción que generan. El primero fuerza la división por cero y el segundo la división por un string.

```
private void button2_Click(object sender, EventArgs e)
{
    try
    {
        textBox1.Text = "1000"; textBox2.Text = "0"; textBox3.Text = "";
        textBox4.Text = "";
        textBox3.Text = (int.Parse(textBox1.Text) / int.Parse(textBox2.Text)).ToString();

    }
    catch (DivideByZeroException Ex) { MuestraError(Ex); }
    catch (Exception Ex) { MuestraError(Ex); }
    finally
    {
        textBox4.Text += NuevaLinea(1) + "--------------------------" +
        NuevaLinea(1) + "Se ejecutó finally !!!!!!!!";
    }
}
```

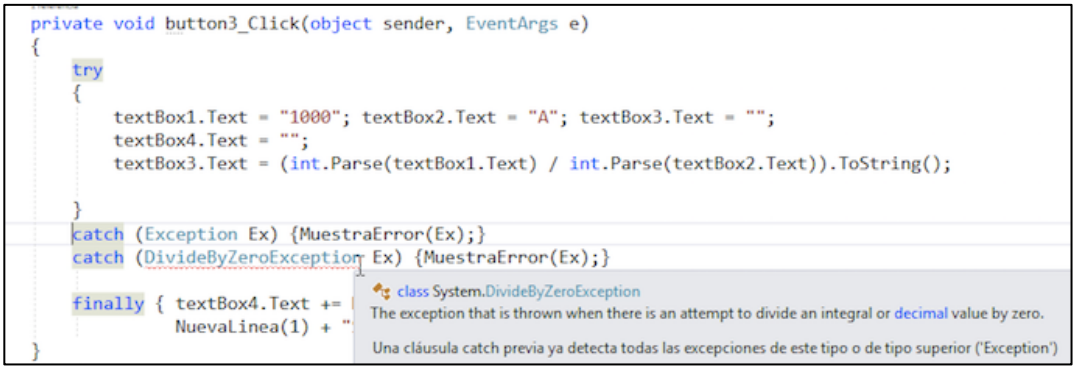
Ej0003

```
private void button3_Click(object sender, EventArgs e)
{
    try
    {
        textBox1.Text = "1000"; textBox2.Text = "A"; textBox3.Text = "";
        textBox4.Text = "";
        textBox3.Text = (int.Parse(textBox1.Text) / int.Parse(textBox2.Text)).ToString();

    }
    catch (DivideByZeroException Ex) {MuestraError(Ex);}
    catch (Exception Ex) {MuestraError(Ex);}
    finally { textBox4.Text += NuevaLinea(1) + "--------------------------" +
              NuevaLinea(1) + "Se ejecutó finally !!!!!!!!";}
}
```

Ej0003

Claramente se puede observar que la excepción **DivideByZeroException** está en un **catch** previo al que contiene la excepción **Exception.** Esto es debido a que las excepciones más especializadas se deben evaluar antes que las más genéricas. Si se coloca la excepción más genérica antes, nunca se llegaría a la más específica, pues la más genérica atraparía al error más específico. Esto se da pues las excepciones no escapan a la realidad de una relación jerárquica del tipo **"es-un".** En nuestro ejemplo **DivideByZeroException** es un **Exception.** En caso de producirse un **DivideByZeroException,** y que el **Exception** se encuentra antes, este atrapará la excepción. Además, C# generará un error como el siguiente.



Ej0003

Extendamos este Ejemplo a dos divisiones. Si además de atrapar la excepción **DivideByZeroException** se desea conocer que división la produjo se debe

proceder como se muestra a continuación. El ejemplo **Ej0004** expone esta situación.

```
1 referencia
private void button2_Click(object sender, EventArgs e)
{
    try
    {
        textBox1.Text = "1000"; textBox2.Text = "250"; textBox3.Text = "";
        textBox5.Text = "1000"; textBox6.Text = "0"; textBox7.Text = "";
        textBox4.Text = "";
        textBox3.Text = (int.Parse(textBox1.Text) / int.Parse(textBox2.Text)).ToString();
        textBox7.Text = (int.Parse(textBox5.Text) / int.Parse(textBox6.Text)).ToString();
    }
    catch (DivideByZeroException Ex) when (int.Parse(textBox2.Text)==0) { MuestraError(Ex,"Número 2"); }
    catch (DivideByZeroException Ex) when (int.Parse(textBox6.Text) == 0) { MuestraError(Ex, "Número 4"); }
    catch (Exception Ex) { MuestraError(Ex,"Genérico"); }
    finally
    {
        textBox4.Text += NuevaLinea(1) + "-------------------------" +
        NuevaLinea(1) + "Se ejecutó finally !!!!!!!!";
    }
}
```

Ej0004

En el fragmento de código anterior se observa dos **catch** del tipo **DivideByZeroException** uno que será utilizado cuando:

```
when (int.Parse(textBox2.Text)==0)
```

y el otro cuando:

```
when (int.Parse(textBox6.Text)==0)
```

Usar **when** permite tener más de un **catch** del mismo tipo de excepción, pero discriminar que tratamiento que se dará dependiendo de las condiciones que acompañan al **when.**

Excepciones personalizadas.

Es muy útil poder definir excepciones personalizadas. Esta característica permite integrar en el sistema de manejo de excepciones aquellas que no vienen provistas por el entorno, pero que para nuestro esquema funcional, representan condiciones no deseadas.

Supongamos que tenemos una cuenta y un requerimiento funcional. El requerimiento funcional estabece que si el saldo de la cuenta es inferior a cero se debe producir una excepción llamada **"Saldo Negativo".** El entorno no provee esta excepción pues no en todos los sistemas esto sería considerado una situación no deseada.

Para poder construirla debemos generar una clase que herede de **Exception.** Es una buena práctica de programación que cuando una clase representa a una excepción, el nombre de la misma termine en **Exception.** En nuestro caso el nombre quedaría como **SaldoNegativoException.** El siguiente código del ejemplo **Ej0005** muestra esto.

```
public class SaldoNegativoException : Exception
{
    Cuenta Vcuenta;
    1 referencia
    public SaldoNegativoException(Cuenta pCuenta)
    { Vcuenta = pCuenta; }
    6 referencias
    public Cuenta Cuenta { get { return Vcuenta; } set { Vcuenta = value; } }
    2 referencias
    public override string Message => "Error por saldo negativo !!!!";

}
6 referencias
```

Ej0005

La clase que representa a la excepción posee una propiedad denominada **Cuenta**. Esta permitirá tener un puntero a la cuenta que su saldo hace que se genere la excepción.

La clase cuenta posee tres propiedades. La primera para el número de cuenta, la segunda para su descripción y la tercera para el saldo. También puede observarse un constructor que permite recibir estas características al momento de instanciarla.

Como aspecto relevante a lo que estamos analizando, se observa en el **set** de la propiedad **Saldo,** luego de asignarle el valor del saldo al campo **Vsaldo** (Vsaldo = value), procede a evaluar si el valor de **Vsaldo** es menor que cero.

En caso afirmativo se ejecuta la instrucción que provoca que la excepción se produzca.

```
throw new SaldoNegativoException(this)
```

La forma correcta de hacerlo es usando **throw** acompañado de una instancia de la excepción que deseo que ocurra, en nuestro caso **SaldoNegativoException.** Por otra parte la instancia recibe en su constructor una referencia de la cuenta (this).

También se puede observar que se sobrescribe el método **Message** para que al ser consultado arroje la leyenda que se desea.

La implementación de Cuenta es la siguiente:

```
public class Cuenta
{
    decimal Vsaldo;
    1 referencia
    public Cuenta(int pNumero, string pDescripcion, decimal pSaldo)
    { Numero = pNumero; Descripcion = pDescripcion;Saldo = pSaldo; }

    1 referencia
    public int Numero { get; set; }
    1 referencia
    public string Descripcion {get;set;}
    1 referencia
    public decimal Saldo
    {
        get
            { return Vsaldo; }
        set
            { Vsaldo = value; if (Vsaldo < 0) throw new SaldoNegativoException(this); }
    }

}
```

Ej0005

La implementación del ejemplo **Ej0005** es la siguiente:

```
private void button1_Click(object sender, EventArgs e)
{
    try
    { C = new Cuenta(int.Parse(Interaction.InputBox("Número: ")),
                                Interaction.InputBox("Descripción: "),
                    decimal.Parse(Interaction.InputBox("Saldo: ")));
    }
    catch (SaldoNegativoException Ex)
    { MessageBox.Show("El error es: " + Ex.Message + Environment.NewLine +
                      "Nro. Cuenta: " + Ex.Cuenta.Numero.ToString() + Environment.NewLine +
                      "Descripción: " + Ex.Cuenta.Descripcion + Environment.NewLine + Environment.NewLine +
                      "Saldo: " + Ex.Cuenta.Saldo);
    }
    catch (Exception Ex) {MessageBox.Show(Ex.Message);}
}
private void button2_Click(object sender, EventArgs e)
{
    try
    {
        C.Saldo=decimal.Parse(Interaction.InputBox("Saldo: "));
    }
    catch (SaldoNegativoException Ex)
    {
        MessageBox.Show("El error es: " + Ex.Message + Environment.NewLine + Environment.NewLine +
                      "Nro. Cuenta: " + Ex.Cuenta.Numero.ToString() + Environment.NewLine +
                      "Descripción: " + Ex.Cuenta.Descripcion + Environment.NewLine +
                      "Saldo: " + Ex.Cuenta.Saldo);
    }
    catch (Exception Ex) {MessageBox.Show(Ex.Message);}
}
```

Ej0005

- NOTA: TODO EL CÓDIGO QUE SE UTILIZA EN LAS EXPLICACIONES LO PUEDE BAJAR DEL **MÓDULO RECURSOS Y BIBLIOGRAFÍA**.

Excepciones

Actividades asincrónicas

Guía de preguntas de repaso conceptual

1. ¿Qué es una excepción?
2. ¿Qué se coloca en el bloque "Catch"?
3. ¿Cómo construiría un objeto del tipo "Exception" personalizado?
4. ¿Qué ocurre si en el bloque de código donde se produce la excepción el error no está siendo tratado?
5. ¿Cuál es el objeto de mayor jerarquía para el manejo de excepciones?
6. ¿En qué namespace se encuentra la clase Exception?
7. ¿Cuáles son las dos clases genéricas más importantes definidas en el FrameWork además de Exception?
8. ¿Qué instrucción se utiliza para poner en práctica el control e interceptar las excepciones?
9. ¿Dónde se coloca el código protegido contra excepciones si se iniciara una excepción?
10. ¿Qué tipo de excepción se utiliza para interceptar un error de división por cero?
11. ¿Qué tipo de excepción se utiliza para interceptar una DLL que tiene problemas al ser cargada?
12. ¿Qué colocaría dentro de una clausula "Catch" para especificar una condición adicional que el bloque "Catch" deberá evaluar como verdadero para que sea seleccionada?
13. ¿Si se desea colocar código de limpieza y liberación de recursos para que se ejecute cuando una excepción se produzca, dónde lo colocaría?

14. ¿Qué instrucción se utiliza para provocar un error y que el mismo se adapte al mecanismo de control de excepciones?

15. ¿Escriba el código que permitiría provocar una excepción del tipo "ArgumentException"?

16. ¿Cómo construiría un objeto del tipo "Exception" personalizado?

17. ¿Cómo armaría un "Catch" personalizado para que se ejecute cuando se de la excepción "ClienteNoExisteException"?

Guía de ejercicios

1. Desarrollar un programa que aplique el concepto de manejo de errores. La estructura propuesta debe tener al menos 5 Catch y el finally.

2. Desarrollar un programa que aplique el concepto de manejo de errores. Generar un error personalizado por medio de una clase que herede de Exception y disparar el error con Throw.