Carrera: **Analista Programador**

PROGRAMACIÓN ORIENTADA A OBJETOS

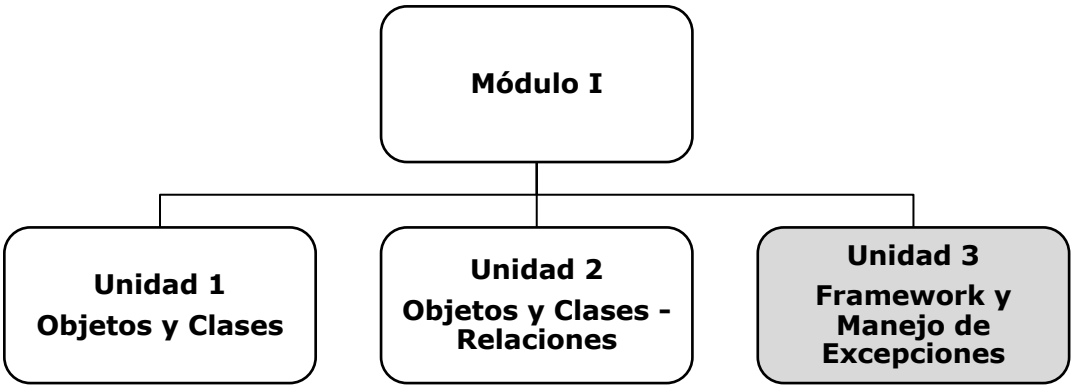
Módulo I

Conocimiento de la Teoría Orientada a Objetos, Frameworks y la definición, construcción y uso de las clases.

Unidad 2

Framework y Manejo de Excepciones

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, se oriente hacia el logro de las siguientes metas de aprendizaje:

- 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 que contribuirá a alcanzar las metas de aprendizaje propuestas:

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.

Índice de contenidos y Actividades

4. Administración de Excepciones

Lectura requerida

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

Para el estudio de estos contenidos usted deberá consultar la bibliografía que aquí se menciona:

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

1. Introducción a los Frameworks.

Un **framework** es un marco de trabajo que ofrece a quien lo utiliza, una serie de herramientas que le facilitan la realización de tareas.

Puede contener librerías de clases, documentación, ayuda, ejemplos, tutoriales, pero sobre todo *"contiene experiencia sobre un dominio de problema específico"*.

Un **framework** tiene como objetivos tres cosas: normalizar, estandarizar y reutilizar la experiencia. Sabemos lo importante que es la normalización de datos y procesos en cualquier aplicación. Los usuarios pueden manejar su información informalmente, en su propia memoria, tenerla duplicada, con incoherencias y omisiones, etc. Muchas veces parece que la única elección importante es la tecnología concreta a utilizar (lenguaje de programación, gestor de bases de datos, etc.) pero, a partir de ahí, cada

programador puede crear, aplicando su imaginación y experiencia, que puede ser mucha o poca, su propia manera de generar código fuente.

La pregunta que surge es ¿por qué permitir ese "desorden" y "falta de administración" en un desarrollo, si en realidad están demostradas las bondades de estructurar y normalizar la información y el código generado?

Los **frameworks** intentan dotarnos de herramientas que permitan hacer realidad las bondades de estructurar y normalizar la información y el código generado. Los **frameworks** no necesariamente están ligados a un lenguaje concreto. Como ejemplo, podemos mencionar el .**NET framework**, sobre el cual se puede desarrollar en distintos lenguajes (VB.NET, C#, J#, etc). También es posible que el **framework** defina una estructura para una aplicación completa, o bien sólo se centre en un aspecto de ella.

Un párrafo aparte se le debe dedicar al concepto *"reutilizar la experiencia".* Un **framework** se desarrolla a partir del conocimiento profundo de algún dominio de problema, e intenta estandarizar y normalizar aquellas cosas que son comunes a todos esos tipos de problemas. Por ejemplo, si construimos un **framework** que sirve para el desarrollo de aplicaciones financieras, este contendrá herramientas y elementos que son el resultado de haber capturado mucha experiencia en esa área en particular. Seguramente esas herramientas y elementos servirán para la mayoría de los sistemas de análisis financiero.

Los elementos básicos de un Framework son: "los puntos congelados o Frozen-Spots", "los puntos calientes o Hot Spots" y además poseen una característica denominada *"Inversión de Control".*

Los puntos congelados son aquellas partes del código provistas por el **framework**. Los puntos calientes son las partes de código que coloca el programador e interactúan con el **framework**. El código colocado en los puntos calientes es lo que el programador considera *"su programa" pero en realidad para que todo funcione correctamente al código generado por él se le suman muchos fragmentos proporcionados por el* **framework.**

Normalmente cuando se trabaja con un **framework,** este se encuentra funcionando desde antes que el programador escriba su primera línea de código. En muchos casos lo asiste en la generación de código, control de errores etc. Se podría decir que el **framework** está en funcionamiento mucho antes que el programa escrito por el programador, es decir *"lo controla".* A esta característica se la denomina **"inversión de control".**

En general se puede optar por utilizar un **framework** desarrollado por terceros o bien desarrollar uno propio. En caso de desarrollar uno propio se deberá seguir una secuencia de etapas constructivas.

Las tres etapas principales del desarrollo del **framework** son: análisis del dominio, diseño del **framework**, y la "instantiación" del **framework**.

El análisis del dominio procura descubrir los requisitos del dominio y los posibles requerimientos futuros. Para completar los requerimientos sirven las experiencias previamente publicadas, los sistemas similares de software existentes, las experiencias personales, y los estándares considerados. Durante el análisis del dominio, los puntos calientes y los puntos congelados se destapan parcialmente.

La fase del diseño del **framework** define sus abstracciones. Se modelan los puntos calientes y los puntos congelados (quizás con diagramas de UML "Lenguaje de Modelado Unificado" - *Unified Modeling Language* -) , y la extensión y la flexibilidad propuesta en el análisis del dominio se esboza en líneas generales. Según lo mencionado arriba, los modelos del diseño se utilizan en esta fase.

Un **framework** se puede también clasificar según su extensibilidad, puede ser utilizado como una **caja blanca** o **caja negra**.

En los **frameworks** de **caja blanca** se crean de nuevas clases por herencia o composición.

En los **frameworks** de **caja negra** se producen instancias usando código o scripts de configuración. Después de la configuración, una herramienta automática de instanciación crea las clases. En los **framework** de caja negra no se requiere que el usuario sepa los detalles internos del framework.

Entre las ventajas de utilizar **frameworks** podemos decir que el programador no necesita plantearse una estructura global de la aplicación, sino que el **framework** le proporciona un esqueleto que hay que "rellenar". También proporciona mejores formas para definir y estandarizar, lo que redundará en ahorro de tiempo.

2 .NET Framework

El **.NET Framework** provee las herramientas necesarias en run-time y compile-time para construir y ejecutar aplicaciones basadas en .NET.

La plataforma **.NET Framework** debe ejecutarse sobre un Sistema Operativo. En la actualidad forma parte del sistema operativo y viene provista por este.

El **.NET Framework** expone servicios de aplicaciones a través de clases de la **.NET Framework Classs Library**.

La **Common Language Runtime** simplifica el desarrollo de aplicaciones, provee un entorno de ejecución robusto y seguro, soporta varios lenguajes y simplifica el despliegue y la administración.

La **CLR** es un entorno administrado (managed), en el cual los servicios comunes, como garbage collection y seguridad, son provistos automáticamente.

La librería de clases de .NET Framework (.NET Framework Class Library) expone características en tiempo de ejecución y provee otros servicios útiles para todos los desarrolladores. Las clases simplifican el desarrollo basado en .NET. Los desarrolladores pueden extenderlas creando sus propias librerías de clases. Las librerías de clases base son implementadas por el .NET Framework. Todas las aplicaciones (web, windows, web services) acceden a las mismas clases base. Estas están almacenadas en namespaces. Los diferentes lenguajes acceden a las mismas librerías.

El framework tambien provee **ADO.NET** como soporte para el acceso a datos en sus modalidades conectado y desconectado.

Por otro lado **XML Web Services** nos pone a disposición componentes Web programables que pueden ser compartidos entre aplicaciones, sobre Internet o una intranet. El .NET Framework provee herramientas y clases para desarrollo, testeo y distribución de XML Web Services.

El .NET Framework soporta tres **tipos de Interfaces** de usuario: **Web Forms, Windows Forms, Aplicaciones de Consola**. En este curso básicamente utilizaremos Windows Forms.

En cuanto a la programación de aplicaciones cualquier lenguaje que sea acorde a la **Common Language Specification (CLS)** puede ejecutarse sobre la CLR. En .NET Framework, Microsoft provee Visual Basic, Visual C++, Visual C#, Visual J#. Terceros nos pueden proveer de nuevos lenguajes.

El .NET Framework constituye las bases sobre las que, tanto aplicaciones como servicios, son ejecutadas y construidas. La naturaleza unificada del

.NET Framework permite que cualquier tipo de aplicación sea desarrollada mediante herramientas comunes, haciendo la integración mucho más simple. Dos de los elementos más importantes en el .NET Framework son:

El CLR (Common Language Runtime)

La BCL (Base Class Library)

El **Common Language Runtime** es el corazón del .NET Framework. Los compiladores y herramientas exponen funcionalidades en tiempo de ejecución y permiten escribir código con el beneficio de un entorno de ejecución administrado. El código que se desarrolla con un compilador de lenguaje que trabaja con el runtime se llama código administrado (**managed code**). Esto permite beneficios como la integración y el manejo de excepciones entre distintos lenguajes, seguridad mejorada, versionamiento y soporte para la implementación del sistema, además de un modelo simplificado para la interacción de componentes y servicios de debugging.

Para permitir al runtime proveer servicios al código administrado, los compiladores deben emitir **metadatos** (información adicional) que describen tipos, miembros y referencias en el código. Los metadatos se almacenan con el código. Cada archivo que el CLR puede cargar contiene metadatos. El runtime los utiliza para localizar y cargar las clases, mantener las instancias en memoria, resolver el llamado de métodos, generar código nativo, mejorar la seguridad y definir las fronteras del contexto de ejecución.

CLR administra la memoria utilizada por las aplicaciones, evitando perdidas de memoria, que podrían estar originadas por errores en el código escrito. El entorno de ejecución brinda además una forma de ejecución que permitirá y administrará la conversión de tipos con los que operan las aplicaciones, la inicialización de las variables, el control de overflows, etc.

A continuación, se realiza una breve reseña de cada uno de los componentes del CLR.

Class loader:
Administra metadatos (información provista con los archivos, analizada más adelante), carga y disponibilidad de las clases.

Microsoft intermediate language (MSIL) to native compiler:
Convierte MSIL a código nativo (JIT).

Code manager:
Administra la ejecución de código.

Garbage collector (GC):

Provee la administración automática del ciclo de vida de todos los objetos.

Security engine:
Provee la seguridad al código que se ejecuta.

Debug engine:
Permite realizar el debug de la aplicación a partir de un seguimiento del código que esta siendo ejecutado.

Type checker:
Evita que se realicen cambios de tipos inseguros o se utilicen variables no inicializadas.

Exception manager:
Provee una estructura de manejo de excepciones, la cual se integra con Windows Structured Exception Handling.

Thread support:
Provee clases e interfaces para trabajar con programación multihilos.

COM marshaler:
Provee interoperabilidad entre .NET y COM.

Base Class Library (BCL) support:
Conjunto de clases que provee el Framework.

Sistema Común de Tipos

CTS (Common Type System). El sistema común de tipos define la forma en la que los tipos deben ser declarados, utilizados y administrados por el runtime. Además, provee una forma para que el runtime de soporte a la integración de varios lenguajes.
El sistema de tipos comunes realiza las siguientes funciones:

> Establecer un framework para soporte de integración de múltiples lenguajes, seguridad de tipos y alta performance en la ejecución de código.

Provee un modelo orientado a objetos que soporta la implementación de varios lenguajes de programación.

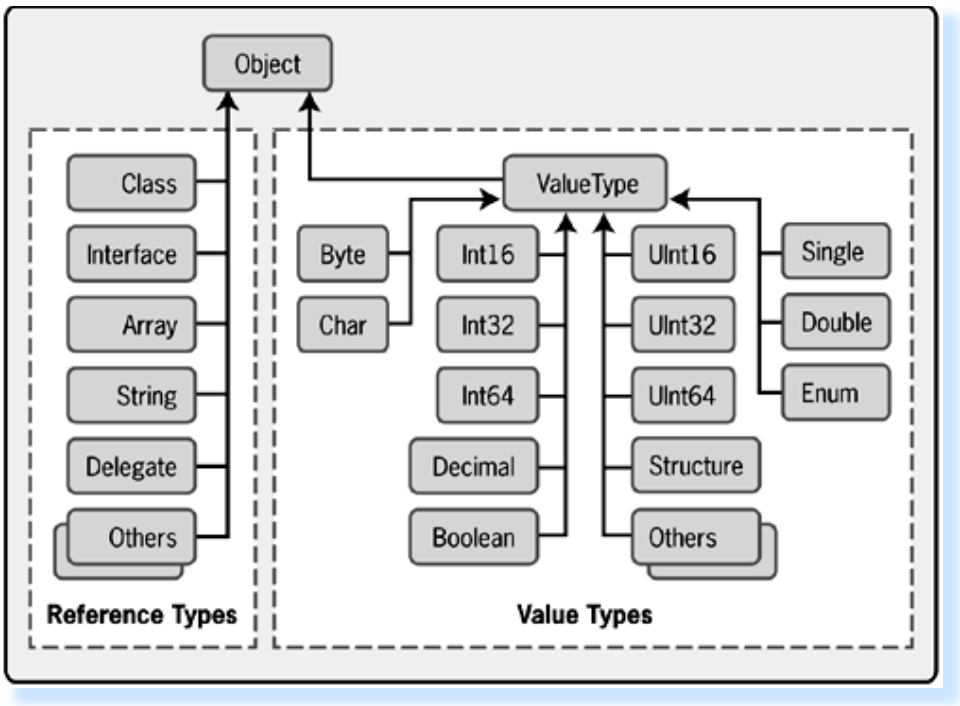
Define las reglas que debe seguir un lenguaje, lo que asegura que distintos lenguajes puedan interactuar sin problemas.

Clasificación de tipos:

CTS soporta dos categorías generales de tipos, cada una de las cuales se divide en subcategorías:

Value types: Directamente contienen sus datos. En memoria funcionan de manera similar a las variables tradicionales.

Reference types: Almacenan una referencia a una dirección de memoria con un valor, y son alojados en la "heap" o "pila adminisrada". Los **Reference types** pueden contener tipos auto-descriptivos, punteros, o interfaces.
Todos los tipos derivan de **System.Object**, que es el tipo base.



Microsoft Intermediate Language (MSIL)

Cuando se compila código soportado en .NET Framework, el compilador convierte el código fuente en Lenguaje intermedio de Microsoft **(MSIL)**, que es un conjunto de instrucciones independiente de la CPU que se pueden convertir de forma eficaz en código nativo.

MSIL incluye instrucciones para cargar, almacenar, inicializar y llamar a métodos en los objetos, así como instrucciones para operaciones lógicas y aritméticas, flujo de control, acceso directo a la memoria, control de excepciones y otras operaciones.

Antes de poder ejecutar código, se debe convertir **MSIL** al código específico de la CPU, normalmente mediante un compilador **JIT (Just in time compiler)**.

Common Language Runtime proporciona uno o varios compiladores **JIT** para cada arquitectura de equipo compatible, por lo que se puede compilar y ejecutar el mismo conjunto de **MSIL** en cualquier arquitectura compatible.

Cuando el compilador produce **MSIL**, también genera información adicional sobre el código. Esta información describe los tipos que aparecen en el código, incluidas las definiciones de los tipos, las firmas de los miembros de tipos, los miembros a los que se hace referencia en el código y otros datos que el motor de tiempo de ejecución utiliza en tiempo de ejecución.

El lenguaje intermedio de Microsoft **(MSIL)** y los datos adicionales, conocidos como **metadatos**, se incluyen en un archivo ejecutable portable **(PE)**, que se basa y extiende el **PE** de Microsoft publicado y el formato **Common Object File Format (COFF)** utilizado tradicionalmente para contenido ejecutable. Este formato de archivo, que contiene código MSIL o código nativo así como metadatos, permite al sistema operativo reconocer lo que debe hacer el **Common Language Runtime**.

La presencia de **metadatos** junto con el **Lenguaje intermedio de Microsoft (MSIL)** permite crear **códigos autodescriptivos.**

En consecuencia la utilización de un **framework** en el desarrollo de una aplicación implica un cierto costo inicial de aprendizaje, aunque a largo plazo es probable que facilite tanto el desarrollo como el mantenimiento.

Existen multitud de **frameworks** orientados a diferentes lenguajes, funcionalidades, etc. Aunque la elección de uno de ellos puede ser una tarea complicada, lo más probable es que a largo plazo sólo los mejor definidos (o más utilizados, que no siempre coinciden con los primeros) permanezcan. Y si ninguno de ellos se adapta a las necesidades de desarrollo, siempre es mejor definir uno propio que desarrollar "al por mayor".

El tema de los **frameworks** es algo que se encuentra en desarrollo. En consecuencia, aún quedan muchos problemas por resolver, tales como la documentación del **framework**. No hay un estándar oficial o de facto para la documentación de los **frameworks**. La falta de un terreno común crea un espacio vacío entre los desarrolladores de **frameworks**, los extendedores de **frameworks** y los usuarios.

Finalmente, los recién llegados al escenario de desarrollo basado en **frameworks**, deberían chequear los puntos expuestos aquí con sumo cuidado. Deberán considerar qué necesita, qué es lo que está haciendo y estar consciente que los **frameworks** tienen sus pros y sus contras, debido a que no son la solución para todos los problemas. Además, si Ud. está considerando usar un **framework** ya existente, estudie su documentación y verifique si hay una buena explicación sobre como utilizar sus facilidades.

3 Administración de Memoria

En .Net Framework el encargado de liberar memoria es el **Garbage Colector (GC).**

En general libera memoria colocando como memoria disponible aquella que posee objetos que ya no se utilizan, pues no tienen referencias que los apunten.

Funciona automáticamente, aunque se lo puede invocar manualmente. Esto último no es recomendable debido a la gran cantidad de recursos que insume su ejecución.

Los objetos pertenecen a una generación. Al funcionar el recolector de basura los objetos que no son borrados sobreviven, esto causa que se eleven a la generación siguiente, esta puede oscilar entre 0 y 2. De esta forma se puede saber cuales objetos han sobrevivido más, asumiendo que esto se debe a que son más utilizados.

Para poder comprender como funciona y cuál es la generación de un objeto usaremos el ejemplo **Ej0001** de esta unidad. Se recomienda que abra el programa y lo ejecute paso a paso a fin de observar el funcionamiento del mismo.

En el código se observa a G, que es el objeto sobre el cual trabajamos y es del tipo string. En primera instancia el mismo será de generación 0.

Ejecutar el Ejemplo Ej0001

Es oportuno recordar que, en las relaciones de agregación con contención física, el objeto que agrega suele crear y agregar al resto en lo que denominamos **constructor**, tema ya abordado. Por otro lado, al finalizar el ciclo de vida del objeto que agrega a su fin, existe un último procedimiento que se ejecuta, denominado **destructor o finalizador**. En este contexto este concepto se divide en dos procedimientos, uno denominado **FINALIZE** y otro llamado **DISPOSE**.

El **FINALIZE** es invocado por el **garbage colector** cuando se ejecuta, mientras que el **DISPOSE** lo invoca el programador.

El detalle que debemos analizar está relacionado con el momento en que se libera un recurso. Por ejemplo, si tenemos un objeto que apunta a un archivo y considerando que el **garbage colector** se ejecutará solo si se necesita liberar memoria, ese archivo referenciado quedará apuntado hasta que esto ocurra. Mientras el objeto siga existiendo en memoria ese archivo no estará disponible para algunas acciones, como borrarlo. Considerando que pueden pasar horas o días para que esto ocurra, sería recomendable colocar la liberación del archivo en el **DISPOSE** y no en el **FINALIZE** ya que el **DISPOSE** puede ser ejecutado por el programador cuando lo desee. Observemos como se pude combinar el uso del **FINALIZE** y **DISPOSE** en el ejemplo **Ej0002**.

Ejecutar el Ejemplo Ej0002

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 provistas 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.

```
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 (Exception Ex) {MuestraError(Ex);}
    catch (DivideByZeroException Ex) {MuestraError(Ex);}

    finally { textBox4.Text +=
              NuevaLinea(1) + "
}
```

class System.DivideByZeroException
The exception that is thrown when there is an attempt to divide an integral or decimal value by zero.

Una cláusula catch previa ya detecta todas las excepciones de este tipo o de tipo superior ('Exception')

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);}
}
1 referencia
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

Lectura requerida

Guía para la lectura y la revisión conceptual

1. ¿Qué es un Framework?
2. ¿Qué son los frozen-spots en un Framework?
3. ¿Qué son los hot-spots en un Framework?
4. ¿Cómo se puede clasificar un Framework según su extensibilidad?
5. ¿Qué es un Framework de Caja Blanca?
6. ¿Qué es un Framework de Caja Negra?
7. ¿Qué ventajas posee utilizar un Framework?
8. ¿Qué problemas resuelve .NET Framework?
9. ¿Qué es y qué permite hacer el CLR?
10. ¿Qué es el MSIL?

11. ¿Qué es el CTS?

12. ¿Qué es el CLS?

13. ¿Qué es una excepción?

14. ¿Qué se coloca en el bloque “Catch”?

15. ¿Cómo construiría un objeto del tipo “Exception” personalizado?

16. ¿Qué ocurre si en el bloque de código donde se produce la excepción el error no está siendo tratado?

17. ¿Cuál es el objeto de mayor jerarquía para el manejo de excepciones?

18. ¿En qué namespace se encuentra la clase Exception?

19. ¿Cuáles son las dos clases genéricas más importantes definidas en el FrameWork además de Exception?

20. ¿Qué instrucción se utiliza para poner en práctica el control e interceptar las excepciones?

21. ¿Dónde se coloca el código protegido contra excepciones si se iniciara una excepción?

22. ¿Qué tipo de excepción se utiliza para interceptar un error de división por cero?

23. ¿Qué tipo de excepción se utiliza para interceptar una DLL que tiene problemas al ser cargada?

24. ¿Qué colocaría dentro de una clausula “Catch” para especificar una condición adicional que el bloque “Catch” deberá evaluar como verdadero para que sea seleccionada?

25. ¿Si se desea colocar código de limpieza y liberación de recursos para que se ejecute cuando una excepción se produzca, dónde lo colocaría?

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

27. ¿Escriba el código que permitiría provocar una excepción del tipo “ArgumentException”?

28. ¿Cómo construiría un objeto del tipo “Exception” personalizado?

29. ¿Cómo armaría un “Catch” personalizado para que se ejecute cuando se de la excepción “ClienteNoExisteException”?

30. ¿Dónde se encuentran las instancias de los objetos administrados por el GC?

31. ¿Cuáles son los dos métodos más notorios que deben implementar las clases para trabajar correctamente con la recolección de elementos no utilizados y matar las instancias administradas y no administradas?

32. ¿De dónde heredan las clases el método Finalize?

33. ¿Cuál es la firma que implementa el método “Finalize”?

34. ¿Qué método se utiliza para que el GC recolecte los elementos no utilizados?

35. ¿Qué método se utiliza para suspender el subproceso actual hasta que el subproceso que se está procesando en la cola de finalizadores vacíe dicha cola?

36. ¿Cuándo se ejecuta el método collect del GC que método se ejecuta en los objetos afectados?

37. ¿Qué método debería exponer una clase bien diseñada teniendo en consideración que no posee destructor?

38. ¿Cómo obtengo el método “Dispose”?

39. ¿Qué se programa en el método “Dispose”?

40. ¿Se pueden combinar el uso de “Dispose” y “Finalize”?

41. ¿A qué se denomina “Resurrección de Objetos”?

42. ¿A qué se denomina “Generación” en el contexto de la recolección de elementos no utilizados?

43. ¿Qué valores puede adoptar la “Generación” de un objeto?

44. ¿Cómo se puede obtener el número de veces que se ha producido la recolección de elementos no utilizados para la generación de objetos especificada?

45. ¿Cómo se obtiene el número de generación actual de un objeto?

46. ¿Cómo se puede recuperar el número de bytes que se considera que están asignados en la actualidad?

47. ¿Qué utiliza para convertir un objeto en “no” válido para la recolección de elementos no utilizados desde el principio de la rutina actual hasta el momento en que se llamó a este método?

48. ¿Cómo se solicita que el sistema no llame al finalizador del objeto especificado?

49. ¿Cómo se solicita que el sistema llame al finalizador del objeto especificado, para el que previamente se ha llamado a “SuppressFinalize”?

50. ¿Cómo se obtiene el número máximo de generaciones que el sistema admite en la actualidad?

Cierre de la unidad

Como cierre de la unidad le proponemos…

Que finalizado con la lectura de las unidades bibliográfica realice el seguimiento de los ejemplos.

Que participe de los espacios de Chat para intercambiar experiencias.

Que investigue en la web más sobre los temas tratados.

Tenga en cuenta que los trabajos que produzca durante los procesos de estudio son insumos muy valiosos y de preparación para la Evaluaciones Parciales. Por lo tanto, guarde sus notas, apuntes y gráficos, le serán de utilidad.