Unidad

3

DIPLOMATURA EN PROGRAMACION .NET

Capítulo 6

Excepciones y Aserciones

Excepciones y Aserciones

En este Capítulo

- Introducción
- Excepciones
- La clase Exception
- Excepciones que finalizan la ejecución de un programa
- Categorías
- Excepciones que pueden ser lanzadas desde el ámbito de un método
- Capturar y Manejar Excepciones
- El bloque try
- Los bloques catch
- Manejo de sentencias try – catch
- Rescritura y Excepciones
- Creación de excepciones personalizadas
- Buenas prácticas en el uso de excepciones
- Características especiales de las excepciones dependientes del lenguaje
- Aserciones, aseveraciones o afirmaciones
- La clase Debug
- El método Debug.Assert
- Usos recomendados de las aserciones

Introducción

Las excepciones son un mecanismo utilizado por numerosos lenguajes de programación para describir lo que debe hacerse cuando ocurre algo inesperado. En general, algo inesperado suele ser algún tipo de error, por ejemplo, la llamada a un método con argumentos no válidos, el fallo de una conexión de red o la solicitud de apertura de un archivo que no existe.

Las aserciones son una forma de verificar ciertos supuestos sobre la lógica de un programa. Por ejemplo, si cree que, en un determinado punto, el valor de una variable siempre será positivo, una aserción puede comprobar si esto es verdad. Las aserciones suelen utilizarse para verificar supuestos sobre la lógica local dentro de un método y no para comprobar si se cumplen las expectativas externas.

Un aspecto importante de las aserciones es que pueden suprimirse enteramente al ejecutar el código. Esto permite habilitar las aserciones durante el desarrollo del programa, pero evitar la ejecución de las pruebas en el tiempo de ejecución, cuando el producto final se entrega al cliente. Por otro lado, las excepciones son una transferencia del control del flujo del programa desde el lugar donde ocurre la excepción al código que se escribió para manejar la misma.

Ésta es una diferencia importante entre aserciones y excepciones.

Excepciones

> *Una excepción es un evento que ocurre durante la ejecución del programa que interrumpe el flujo normal de las sentencias.*

Muchas clases de errores pueden utilizar excepciones, desde serios problemas de hardware, como la avería de un disco, a los simples errores de programación, como tratar de acceder a un elemento de un vector fuera de sus límites. Cuando dicho error ocurre dentro de un método en .Net, este crea un objeto del tipo Exception y lo maneja en un sistema de ejecución especial dedicado a este fin dentro del lenguaje. Este objeto contiene información sobre la excepción, incluyendo su tipo y el estado del programa cuando ocurrió el error. El sistema de ejecución es el responsable de buscar algún código para manejar el error. En terminología de .Net, crear un objeto del tipo Exception y manejarlo por el sistema de ejecución se llama lanzar una excepción.

Después que un método lance una excepción, el sistema de ejecución entra en acción para buscar el manejador de la misma. Para ello se recorre en sentido inverso aquellos métodos que se encuentran en ejecución desde que comenzó el programa para encontrar el manejador apropiado. Esto es posible porque desde el comienzo del programa cada método invocado reserva espacio en el stack formando la denominada “pila de llamados”, la cual sirve de “lista” para recorrerla inversamente en busca del código que maneje la excepción. Si un método no maneja la excepción que acaba de ocurrir, simplemente se recorre en forma inversa el stack para revisar si el anterior (que invocó a este) al que se acaba de revisar contiene el código que lo maneje. En caso de

encontrarlo se va al siguiente y así sucesivamente hasta llegar a Main. Si en Main, el cual es el primer método que reserva espacio en el stack, no se encuentra el código que maneja la excepción, el programa termina indicando el motivo del error y el recorrido inverso que realizo en la pila de llamados proveyendo además la información de los métodos analizados en busca del código de manejo de la excepción.

El código que maneja la excepción se considera adecuado para su manejo por la comparación de la variable de referencia que dicho código recibe como argumento. Esta variable debe ser del mismo tipo que el error que acaba de ocurrir o tiene que tener una conversión de tipo a una superclase bien definida. Esto quiere decir que la variable de referencia declarada deberá ser igual al objeto del tipo excepción ocurrida o al menos igual al de algún objeto del tipo Exception que se encuentre antecediéndolo en la misma cadena de herencia.

Así la excepción sube sobre la pila de llamadas hasta que encuentra el manejador apropiado y una de las llamadas a métodos maneja la excepción, se dice que el manejador de excepción elegido *captura* la excepción.

Si el sistema de ejecución busca exhaustivamente por todos los métodos de la pila de llamadas sin encontrar el manejador de excepción adecuado (el que "capture la excepción"), el sistema de ejecución finaliza (y consecuentemente y el programa también).

Mediante el uso de excepciones para manejar errores, los programas tienen las siguientes ventajas frente a las técnicas de manejo de errores tradicionales:

- **Ventaja 1**: Separar el manejo de errores del código "normal"
- **Ventaja 2**: Propagar los errores sobre la pila de llamadas
- **Ventaja 3**: Agrupar los tipos de errores y la diferenciación de éstos

Ventaja 1: Separar el manejo de errores del código "normal"

En la programación tradicional, la detección, el informe y el manejo de errores se convierten al código en algo muy complicado. Por ejemplo, suponiendo que existe una función que lee un archivo completo dentro de la memoria. En seudocódigo, la función se podría parecer a esto.

Ejemplo

```
1. leerArchivo {
2.         abrir el archivo;
3.         determinar su tamaño;
4.         asignar suficiente memoria;
5.         leer el archivo a la memoria;
6.         cerrar el archivo;
7. }
```

A primera vista esta función parece bastante sencilla, pero ignora todos aquellos errores potenciales, por ejemplo:

- ¿Qué sucede si no se puede abrir el archivo?
- ¿Qué sucede si no se puede determinar la longitud del archivo?
- ¿Qué sucede si no hay suficiente memoria libre?
- ¿Qué sucede si la lectura falla?
- ¿Qué sucede si no se puede cerrar el archivo?

Para responder a estas cuestiones dentro de la función, se debería añadir mucho código para la detección y el manejo de errores. El aspecto final de la función se parecería a lo siguiente:

Ejemplo

```
codigodeError LeerArchivo {
inicializar codigodeError = 0;
abrir el archivo;
if (archivoAbierto) {
            determinar la longitud del archivo;
            if (obtenerLongitudDelArchivo) {
                  asignar suficiente memoria;
                  if (obtenerSuficienteMemoria) {
                        leer el archivo a memoria;
                        if (falloDeLectura) {
                              codigodeError = -1;
                        } else {
                              codigodeError = -2;
                        }
                  } else {
                        codigodeError = -3;
                  }
                  cerrar el archivo;
                  if (archivoNoCerrado && codigodeError == 0) {
                        codigodeError = -4;
                  } else {
                        codigodeError = codigodeError and -4;
                  }
            } else {
                  codigodeError = -5;
            }
            return codigodeError;
      }
}
```


Con la detección de errores, las 7 líneas originales se han convertido en 29 líneas de código, por lo tanto ha aumentado más de un 400 %. Para peor, existe tanta detección y manejo de errores como de retorno respecto a las 7 líneas originales que el código está totalmente enrarecido. Y aún peor, el flujo lógico del código también se pierde, haciendo difícil poder decidir si el código hace lo correcto (si se cierra el archivo realmente o si falla la asignación de memoria) e incluso es difícil

asegurar que el código continúe haciendo las cosas correctas cuando se modifique el método, por ejemplo, tres meses después de haberlo escrito.

Muchos programadores "resuelven" este problema ignorándolo, motivo por el cual se informa de los errores cuando el programa no funciona. Sin embargo esto viola directamente un principio básico de las “buenas prácticas” en la programación orientada a objetos:

Toda clase debe ser capaz de manejar los errores que ocurren dentro de ella

Esto tiene varias consecuencias en la codificación, porque si un método no maneja el error dentro de él, se vuelve en reversa sobre el stack, lo cual no asegura que el próximo código a evaluar para el manejo del error se encuentre en el mismo objeto o dentro de otro que simplemente llamó al método donde ocurrió el error.

El Framework de .Net proporciona una solución elegante al problema del tratamiento de errores: las excepciones. Las excepciones le permiten escribir el flujo principal de su código y tratar los casos excepcionales en otro lugar. Si el método leerArchivo utilizara excepciones en lugar de las técnicas de manejo de errores tradicionales se podría parecer a esto:

Ejemplo

```
1. LeerArchivo {
2.        try {
3.              abrir el archivo;
4.              determinar su tamaño;
5.              asignar suficiente memoria;
6.              leer el archivo a la memoria;
7.              cerrar el archivo;
8.        } catch (falloAbrirArchivo) {
9.              hacerAlgo;
10.       } catch (falloDeterminacionTamaño) {
11.             hacerAlgo;
12.       } catch (falloAsignaciondeMemoria) {
13.             hacerAlgo;
14.       } catch (falloLectura) {
15.             hacerAlgo;
16.       } catch (falloCerrarArchivo) {
17.             hacerAlgo;
18.       }
19.}
```

Observar que las excepciones no evitan el esfuerzo de hacer el trabajo de detectar, informar y manejar errores. Lo que proporcionan las excepciones es la posibilidad de separar los detalles oscuros de qué hacer cuando ocurre algo fuera de la normal.

Además, el factor de aumento de código de este es programa es de un 250%, comparado con más del 400% del ejemplo anterior.

Ventaja 2: Propagar los errores sobre el stack (pila) de llamadas

Una segunda ventaja de las excepciones es la posibilidad del propagar el error encontrado sobre la pila de llamadas a métodos. Suponiendo que el método leerArchivo es el cuarto método en una serie de llamadas a métodos anidadas realizadas por un programa principal: metodo1 llama a metodo2, que llama a metodo3, que finalmente llama a leerArchivo.

Ejemplo

```
1. Metodo1 {
2.        llamar a Metodo2;
3. }
4. Metodo2 {
5.        llamar a Metodo3;
6. }
7. Metodo3 {
8.        llamar a LeerArchivo;
9. }
```

Suponiendo también que metodo1 es el único método interesado en el error que ocurre dentro de leerArchivo. Tradicionalmente las técnicas de notificación del error forzarían a metodo2 y metodo3 a propagar el código de error devuelto por leerArchivo sobre la pila de llamadas hasta que el código de error llegue finalmente a metodo1, el único método que está interesado en él.

```
1. Metodo1 {
2.        numCodigoTipoDeError error;
3.        error = llamar a Metodo2;
4.        if (error)
5.              procesodelError;
6.        else
7.              proceder;
8. }
9.
10. numCodigoTipoDeError Metodo2 {
11.       numCodigoTipoDeError error;
12.       error = llamar a Metodo3;
13.       if (error)
14.             return error;
15.       else
16.             proceder;
17. }
18.
19. numCodigoTipoDeError Metodo3 {
20.       numCodigoTipoDeError error;
```

```
21.       error = llamar a LeerArchivo;
22.       if (error)
23.             return error;
24.       else
25.             proceder;
26. }
```

Como se mencionó anteriormente, el sistema de ejecución del CLR busca hacia atrás en la pila de llamadas para encontrar cualquier método que esté interesado en manejar una excepción particular. Un método puede "esquivar" cualquier excepción lanzada dentro de él (esto es, simplemente, no capturando la excepción), por lo tanto permite a los métodos que están por encima de él en la pila de llamadas poder capturarlo. Sólo los métodos interesados en el error deben preocuparse de detectarlo.

```
1.  Metodo1 {
2.        try {
3.              llamar a Metodo2;
4.        } catch (excepcion) {
5.              procesodelError;
6.        }
7.  }
8.
9.  Metodo2 {
10.       llamar a Metodo3;
11. }
12.
13. Metodo3 {
14.       llamar a LeerArchivo;
15. }
```

Observar de nuevo la diferencia del factor de aumento de código como así también la diferencia de claridad entre las dos técnicas de manejo de errores. El código que utiliza excepciones es más compacto y más fácil de entender.

Ventaja 3: Agrupar los tipos de errores y la diferenciación de éstos

Frecuentemente las excepciones se dividen en categorías o grupos. Por ejemplo, se puede suponer un grupo de excepciones, cada una de las cuales representara un tipo de error específico que pudiera ocurrir durante una operación aritmética:

- Se intenta dividir por cero
- El resultado de la operación está fuera del rango del tipo de datos definido
- Una operación con punto flotante no retorna un número finito o el resultado no es un número

Además, se puede suponer que algunos métodos querrán manejar todas las excepciones de esa categoría (todas las excepciones de las operaciones aritméticas), y otros métodos podrían manejar sólo algunas excepciones específicas (como la excepción de división por cero).

Como todas las excepciones lanzadas dentro de los programas son objetos, agrupar o categorizar las excepciones es una solución natural al modelar clases y cadenas de herencia. Las excepciones deben ser subclases de la clase Exception, o de cualquier subclase de ésta. Al igual que con otras clases, se pueden crear subclases de Exception y subclases de estas subclases.

Existen clases de excepción que no tienen subclase y otras que si las tienen. La herencia es un tipo de relación entre clases y por consiguiente, los objetos creados a partir de ellas. Esta relación permite agrupar conceptualmente a las excepciones. Cada clase sin subclases representa un tipo específico de excepción y cada clase que tiene una o más subclases representa un grupo de excepciones relacionadas.

Por ejemplo, un grupo de clase que no tienen subclases predefinidas en .Net son:

- DivideByZeroException
- OverflowException
- NotFiniteNumberException

Cada una representa un tipo específico de error que puede ocurrir cuando se manipula una operación aritmética. Un método puede capturar una excepción basada en su tipo específico (la clase que la define, una superclase de la cadena o sub cadena de herencia a la que pertenece o una interfaz que alguna de ellas implemente).

Por ejemplo, un manejador de excepción que sólo controle la excepción de dividir por cero, tiene una sentencia `catch` o `Catch` (C# o VB), a través de la cual captura una excepción, como la siguiente:

C#

```
catch (DivideByZeroException e)
{
  …
}
```

VB

```
Catch e As DivideByZeroException
  …
End Try
```

De esta manera, si suponemos que ArithmeticException podría ser una superclase de la sub cadena de herencia que representa (agrupa a través de la herencia) cualquier error que pueda ocurrir durante las operaciones aritméticas, incluyendo aquellos errores representados específicamente por una de sus subclases, un método puede capturar una excepción basado en este grupo o tipo general especificando cualquiera de las superclases de la excepción en la

sentencia `catch` o `Catch` (C# o VB). Por ejemplo, para capturar todas las excepciones de operaciones aritméticas, sin importar sus tipos específicos, un manejador de excepción especificaría un argumento de esta clase ArithmeticException.

C#

```
catch (ArithmeticException e)
{
  …
}
```

VB

```
Catch e As ArithmeticException
  …
End Try
```

Este manejador podría capturar todas las excepciones de operaciones aritméticas, incluyendo DivideByZeroException, OverflowException y NotFiniteNumberException. Se puede descubrir el tipo de excepción preciso que ha ocurrido comprobando el argumento que recibe el manejador. Incluso se puede seleccionar un manejador de excepciones que controlara cualquier excepción con el siguiente manejador.

C#

```
catch (Exception e)
{
  …
}
```

VB

```
Catch e As Exception
  …
End Try
```

Los manejadores de excepciones que son demasiado genéricos, como el mostrado anteriormente, pueden hacer que el código sea propenso a errores mediante la captura y manejo de excepciones que no se hubieran anticipado y por lo tanto no son gestionadas correctamente dentro de manejador. Como regla no se recomienda escribir manejadores de excepciones genéricos.

Como se ha visto, se pueden crear grupos de excepciones y manejarlas de una forma general, o se puede especificar un tipo de excepción específico para diferenciar excepciones y manejarlas de un modo exacto.

Como se mencionó anteriormente, la clase Exception maneja condiciones de errores que no sean críticas que un programa pueda encontrar en el transcurso de su ejecución. En lugar de terminar abruptamente, a través de ella se puede escribir código que maneje las condiciones de error y continuar con el procesamiento normal del programa.

Cualquier condición anormal que influya sobre la ejecución de un programa mientras este se encuentre corriendo es un error o una excepción. Por ejemplo, las excepciones pueden ocurrir porque:

- Se trata de abrir un archivo que no existe
- La conexión de red falla
- Se excede el límite de un vector
- Cuando se utilizan operandos que se van del rango posible de trabajo
- No se encuentra una clase que se quiere cargar con el CLR

Las excepciones de .Net tiene el estilo de las de C++ para ayudar al programador a construir código más flexible. Cuando ocurre un error en un programa, el método que lo encuentra puede lanzar una excepción en dirección al código del método que lo invocó para señalar el problema que acaba de ocurrir. El método que realizó la invocación puede de esta manera "atrapar o capturar" la excepción lanzada y, en caso de poder hacerlo, recuperarse de ella manejando la condición de error. Esta forma de trabajo permite al programador generar código específico para el manejo de dichas excepciones.

La clase Exception

Esta clase es la clase base de todas las excepciones. Cuando se produce un error, el sistema o la aplicación que se está ejecutando en ese momento puede informar del mismo lanzando una excepción que contenga información acerca del error. Una vez que se produce una excepción, la aplicación o el controlador de excepciones predeterminado la controlan.

Características del tipo Exception

Los tipos de excepción admiten las siguientes características:

- **Texto legible para el usuario donde se describe el error**. Cuando se produce una excepción, el CLR pone a disposición del usuario un mensaje de texto donde se informa de la naturaleza del error y una recomendación para llevar a cabo una determinada acción que solucione el problema. Este mensaje de texto se guarda en la propiedad Message del objeto del tipo Exception. Durante la creación del objeto de excepción, puede pasarse una cadena de texto al constructor para que se describan los detalles concretos que motivaron el lanzamiento de dicha excepción. Si no se le proporciona al constructor ningún argumento que sirva de mensaje de error, se utiliza el predeterminado.
- **Estado de la pila de llamadas en el momento en que se produjo la excepción**. La propiedad StackTrace incluye un seguimiento de pila que puede utilizarse para determinar la parte del código en la que se ha producido un error.El seguimiento de pila muestra todos los métodos a los que se ha llamado y los números de línea en el archivo de código fuente donde se realizan dichas llamadas.

Propiedades de la clase Exception

Exception incluye determinadas propiedades que ayudan a identificar la ubicación del código, el tipo, el archivo de ayuda y el motivo de la excepción: StackTrace, InnerException, Message, HelpLink, HResult, Source, TargetSite y Data. La siguiente tabla es un resumen de dichas propiedades.

Nombre	Descripción
Data	Obtiene una colección de pares clave-valor que proporcionan información adicional definida por el usuario acerca de la excepción.
HelpLink	Obtiene o establece un vínculo al archivo de ayuda asociado a esta excepción.
HResult	Obtiene o establece HRESULT, un valor numérico codificado que se asigna a una excepción específica.
InnerException	Obtiene la instancia Exception que produjo la excepción actual.
Message	Obtiene un mensaje que describe la excepción actual.
Source	Devuelve o establece el nombre de la aplicación o del objeto que generó el error.
StackTrace	Obtiene una representación de cadena de los marcos inmediatos en la pila de llamadas.
TargetSite	Obtiene el método que produjo la excepción actual.

Cuando existe una relación entre dos o más excepciones, la propiedad InnerException puede utilizarse para almacenar la referencia a la excepción relacionada con la actual. Por lo general, la excepción externa se produce en respuesta a la excepción interna cuya referencia se almacena en esta propiedad. El código que controla la excepción externa puede utilizar la información de la excepción interna para controlar el error de forma más adecuada. Se puede, además, almacenar información adicional acerca de la excepción en la propiedad Data.

La cadena de mensaje de error que se pasa al constructor durante la creación del objeto del tipo Exception se puede encontrar y suministrar desde un archivo de recursos mediante un objeto del tipo ResourceManager.

Para proporcionar al usuario abundante información sobre el motivo de la excepción, la propiedad HelpLink puede incluir una dirección URL (o URN) de un archivo de ayuda.

Exception utiliza HRESULT COR_E_EXCEPTION, que tiene el valor 0x80131500.

Además, Exception permite la construcción de objetos de su tipo según los constructores que se muestran a continuación:

Nombre	Descripción
Exception()	Inicializa una nueva instancia de la clase Exception.

Exception(String)	Inicializa una nueva instancia de la clase Exception con el mensaje de error especificado.
Exception(SerializationInfo, StreamingContext)	Inicializa una nueva instancia de la clase Exception con datos serializados.
Exception(String, Exception)	Inicializa una nueva instancia de la clase Exception con un mensaje de error especificado y una referencia a la excepción interna que representa la causa de esta excepción

Consideraciones sobre el rendimiento

Cuando se produce o se controla una excepción, se utiliza una cantidad significativa de recursos del sistema en tiempo de ejecución. Hay que producir excepciones sólo para controlar condiciones verdaderamente extraordinarias, no para controlar eventos previsibles o el control del flujo del programa. Por ejemplo, la aplicación puede producir razonablemente una excepción si un argumento de método no es válido porque se espera que se llame al método con parámetros válidos. Un argumento de método no válido significa que se ha producido algo extraordinario. A la inversa, no hay que producir una excepción si la entrada de usuario no es válida porque se puede esperar que los usuarios a veces introduzcan datos no válidos. En dicho caso, hay que proporcionar un mecanismo de reintento de modo que los usuarios puedan escribir una entrada válida.

Excepciones que finalizan la ejecución de un programa

Los errores se pueden detectar cuando un programa termina abruptamente. Por ejemplo, el siguiente código genera una terminación anormal del programa.

Ejemplo

C#

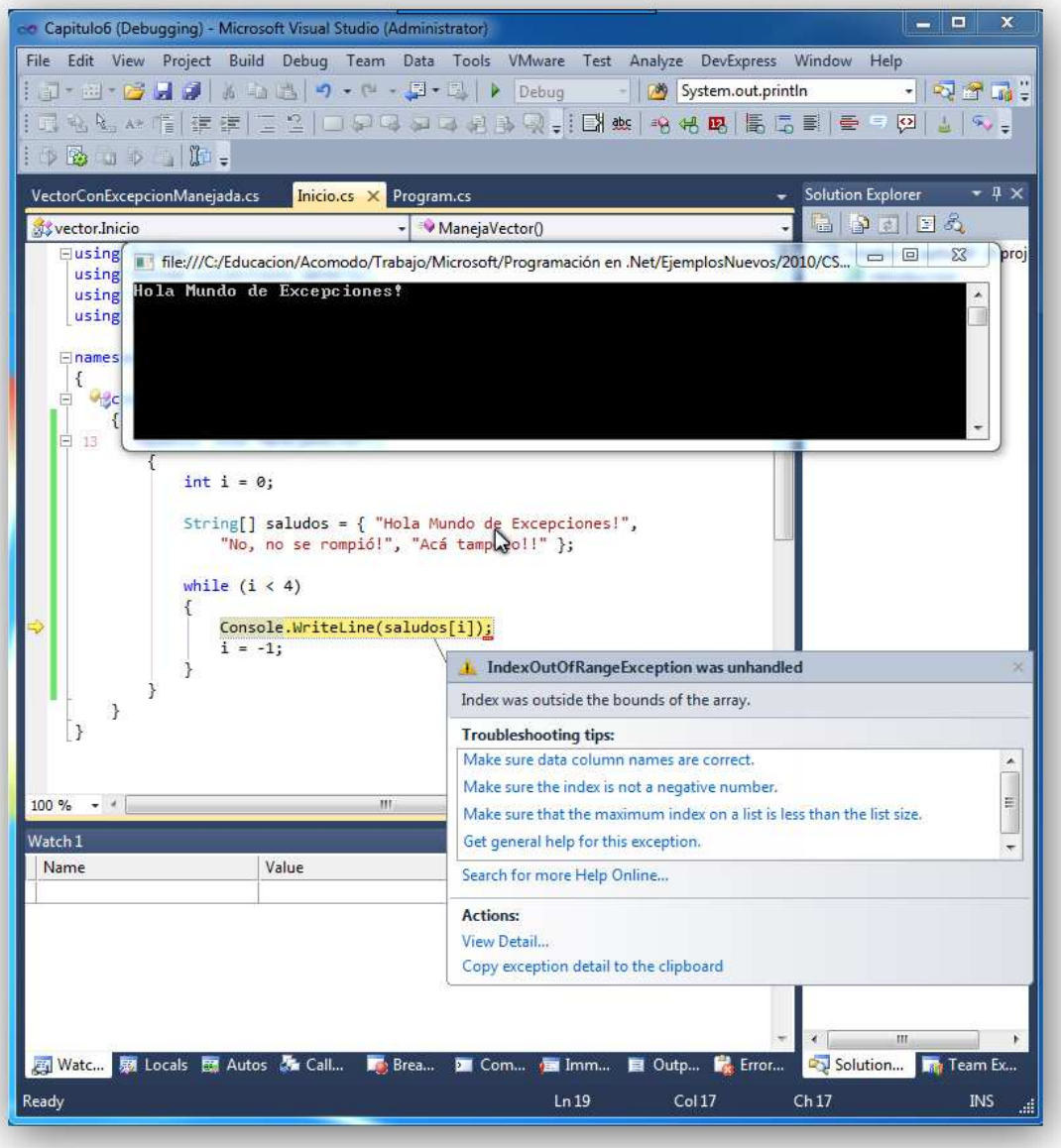
```
namespace introduccion
{
    class Program
    {
        static void Main(string[] args)
        {
            int i = 0;

            String[] mensajes = { "Hola Mundo de Excepciones!",
                                  "No, no se rompió!",
                                  "Acá tampoco!!" };

            while (i < 4)
            {
                Console.WriteLine(mensajes[i]);
                i++;
            }
```

```
        }
    }
}
```

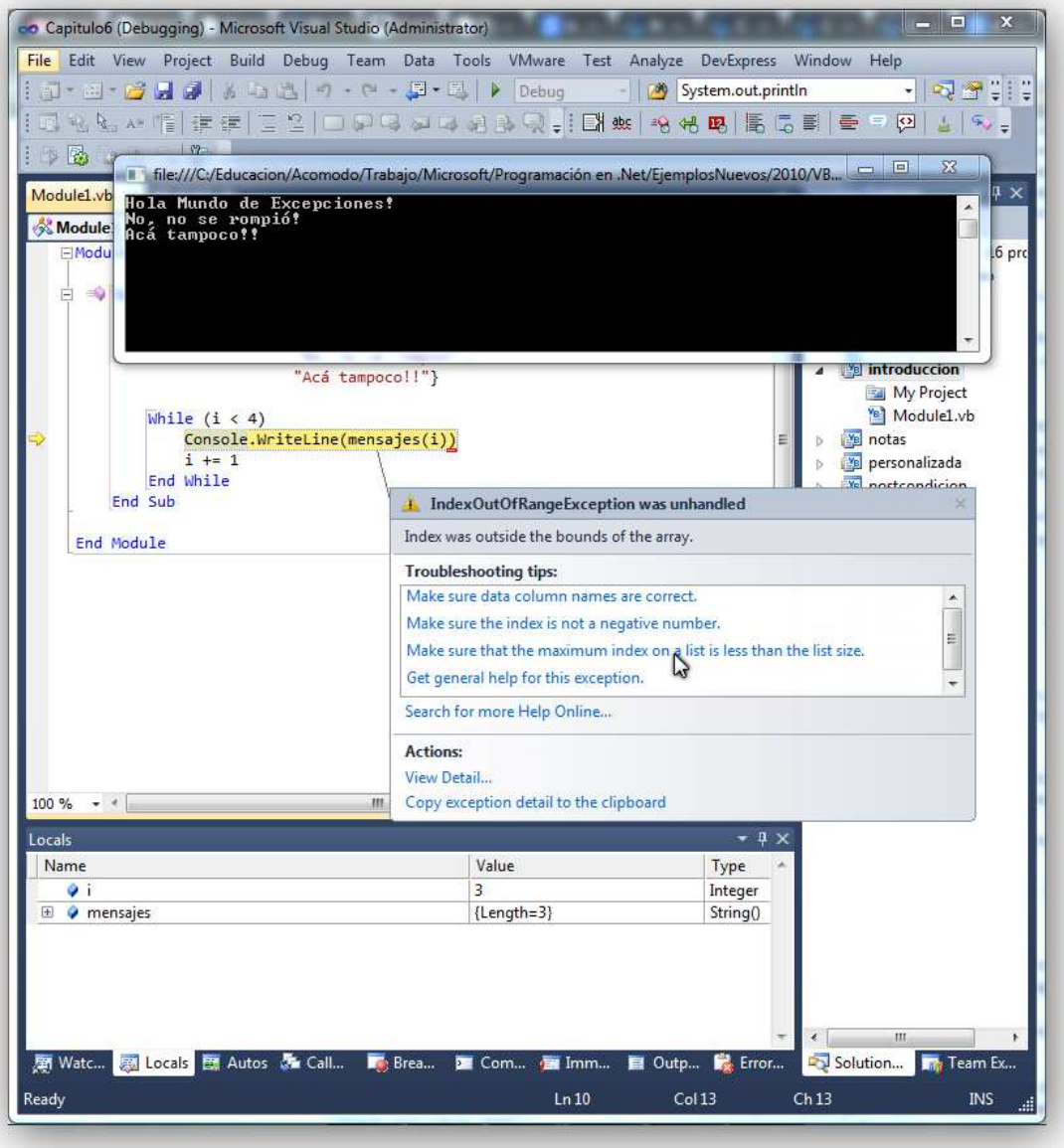
VB

```
Module Module1

    Sub Main()
        Dim i As Integer
        Dim mensajes() As String = {"Hola Mundo de Excepciones!",
                        "No, no se rompió!",
                        "Acá tampoco!!"}

        While (i < 4)
            Console.WriteLine(mensajes(i))
            i += 1
        End While
    End Sub
End Module
```

Cuando se ejecuta este programa la salida que se obtiene es la siguiente:


Capitulo6 (Debugging) - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Team Data Tools VMware Test Analyze DevExpress Window Help
Debug
System.out.println
VectorConExcepcionManejada.cs
Inicio.cs
Program.cs
Solution Explorer
vector.Inicio
ManejaVector()
file:///C:/Educacion/Acomodo/Trabajo/Microsoft/Programación en .Net/EjemplosNuevos/2010/CS...
Hola Mundo de Excepciones!
int i = 0;
String[] saludos = { "Hola Mundo de Excepciones!",
"No, no se rompió!", "Acá tampoco!!" };
while (i < 4)
Console.WriteLine(saludos[i]);
i = -1;
IndexOutOfRangeException was unhandled
Index was outside the bounds of the array.
Troubleshooting tips:
Make sure data column names are correct.
Make sure the index is not a negative number.
Make sure that the maximum index on a list is less than the list size.
Get general help for this exception.
Search for more Help Online...
Actions:
View Detail...
Copy exception detail to the clipboard
Watch 1
Name
Value
Ready
Ln 19
Col 17
Ch 17
INS

VB



La ejecución del programa se interrumpe con un mensaje de error cuando se lanza una excepción y despliega una ventana conocida como “ventana de depuración” o “Debug window”. Esto sucede luego de ejecutar cuatro veces el ciclo.

El manejo de excepciones le permite al programa capturarlas, manejarlas y luego continuar con la ejecución normal del mismo. Es una estructura de manejo que provee el lenguaje para aquellos casos en los que no se sigue el flujo normal planificado para un programa. El ejemplo anterior muestra en un programa simple donde se puede dar este tipo de situación.

Estos casos especiales deben ser manejados en el momento que ocurren, en bloques de código separados, de manera que dichos bloques se puedan "asociar" a la ejecución normal del programa. Además, que sólo entren en acción cuando la situación anormal ocurre, generando en consecuencia código más simple y fácil de mantener.

Categorías

Existen dos categorías de excepciones en la clase base Exception:

- Clases de excepción predefinidas de Common Language Runtime que se derivan de SystemException.
- Clases de excepción de aplicaciones definidas por el usuario que se derivan de ApplicationException o Exception.

Nota: En las primeras versiones del Framework .Net, las excepciones creadas por el usuario derivaban de ApplicationException. En la actualidad esto está desaconsejado y se promueve la creación de excepciones derivadas de Exception.

Para comprender y manejar una determinada condición de error que pueda ocurrir cuando el Framework .Net lanza excepciones por errores, hay que entenderlos en base a las categorías definidas para ellos a través de sus cadenas de herencia, las cuales permiten realizar agrupaciones según su funcionalidad.

Por ejemplo, toda clase que sea capaz de "lanzar" un objeto que sea manejado por la estructura de gestión de excepciones que provee .Net debe ser subclase de Exception, ya que esta actúa como superclase de todos los objetos de este tipo. Las subclases a partir de ella pueden crear instancias de objetos que se podrán "lanzar y capturar" utilizando el mecanismo de manejo de excepciones.

El siguiente ejemplo muestra como el Framework de .Net lanza una excepción cuando se intenta realizar una operación inválida como una conversión de tipo no definida.

Ejemplo

C#

```
namespace framework
{
    class Empleado
    {
        // Crear una propiedad para el nivel del empleado
        private int nivel;

        public int Nivel
        {
            get { return nivel; }
            set { nivel = value; }
        }
```

```
    }
}

namespace framework
{
    class EjemploExcepciones
    {
        public static void PromoverEmpleado(Object emp)
        {
            //Convertir el tipo del objeto a Empleado.
            Empleado e = (Empleado)emp;
            // Incrementar el nivel del empleado
            e.Nivel  = e.Nivel + 1;
        }

        public static void Main()
        {
            try
            {
                Object o = new Empleado();
                DateTime fecha = new DateTime(2001, 1, 1);
                //Promover al nuevo empleado.
                PromoverEmpleado(o);
                // Fecha de promoción: resulta en una InvalidCastException
                // porque fecha no es una instancia de Empleado
                PromoverEmpleado(fecha);
            }
            catch (InvalidCastException e)
            {
                Console.WriteLine(
              "Error al pasar datos al método PromoverEmpleado. \n" + e.Message);
                Console.WriteLine("");
                Console.WriteLine("Presione un tecla para salir.");
                Console.ReadKey();
            }
        }
    }
}
```

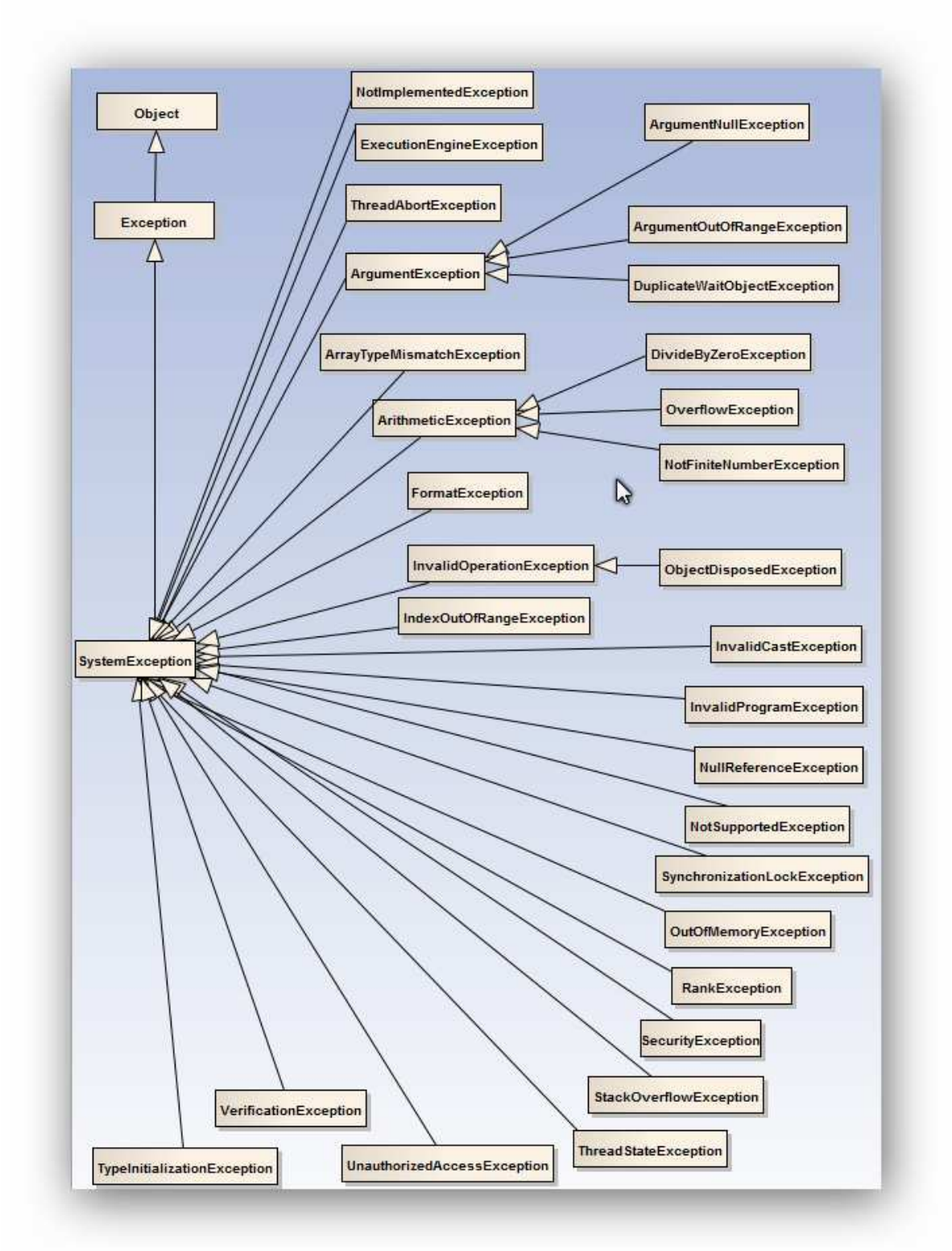
VB

```
Public Class Empleado
    ' Crear una propiedad para el nivel del empleado
    Private _nivel As Integer
    Public Property Nivel() As Integer
        Get
            Return _nivel
        End Get
        Set(ByVal value As Integer)
            _nivel = value
        End Set
    End Property
End Class

Public Class EjemploExcepciones
    Public Shared Sub PromoverEmpleado(ByVal emp As [Object])
```

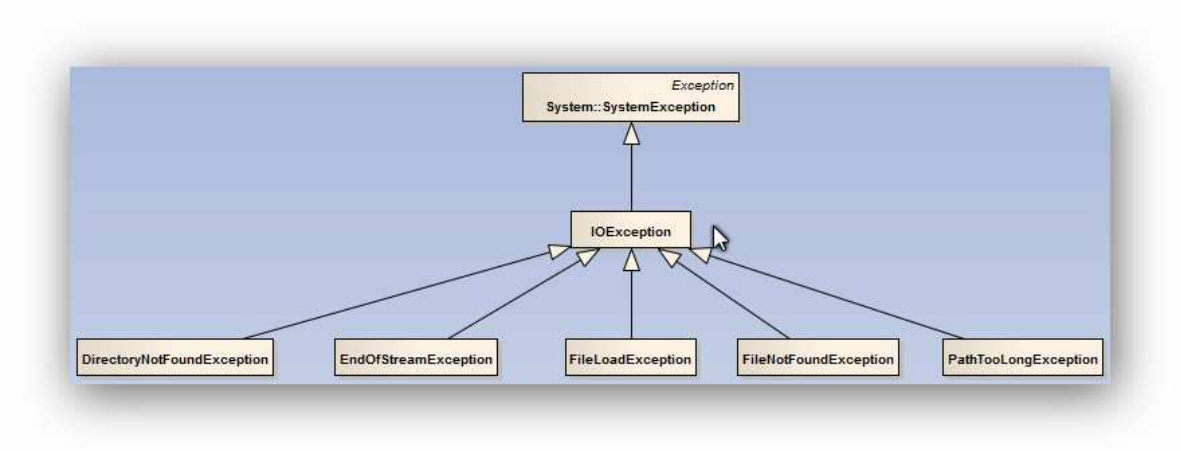
```
        'Convertir el tipo del objeto a Empleado.
        Dim e As Empleado = CType(emp, Empleado)
        ' Incrementar el nivel del empleado.
        e.Nivel = e.Nivel + 1
    End Sub

    Public Shared Sub Main()
        Try
            Dim o = New Empleado()
            Dim fecha As New DateTime(2001, 1, 1)
            'Promover al nuevo empleado.
            PromoverEmpleado(o)
            'Fecha de promoción: resulta en una InvalidCastException
            ' porque fecha no es una instancia de Empleado.
            PromoverEmpleado(fecha)
        Catch e As InvalidCastException
            Console.WriteLine(("Error al pasar datos al método PromoverEmpleado. " _
                     + e.Message))
            Console.WriteLine("")
            Console.WriteLine("Presione un tecla para salir.")
            Console.ReadKey()
        End Try
    End Sub 'Main
End Class
```

Los métodos definidos en la clase Exception recuperan el mensaje de error asociado con la excepción y permite acceder a la información acerca del recorrido que esta realice en el stack buscando un manejador, además del lugar donde dicha excepción ocurrió.

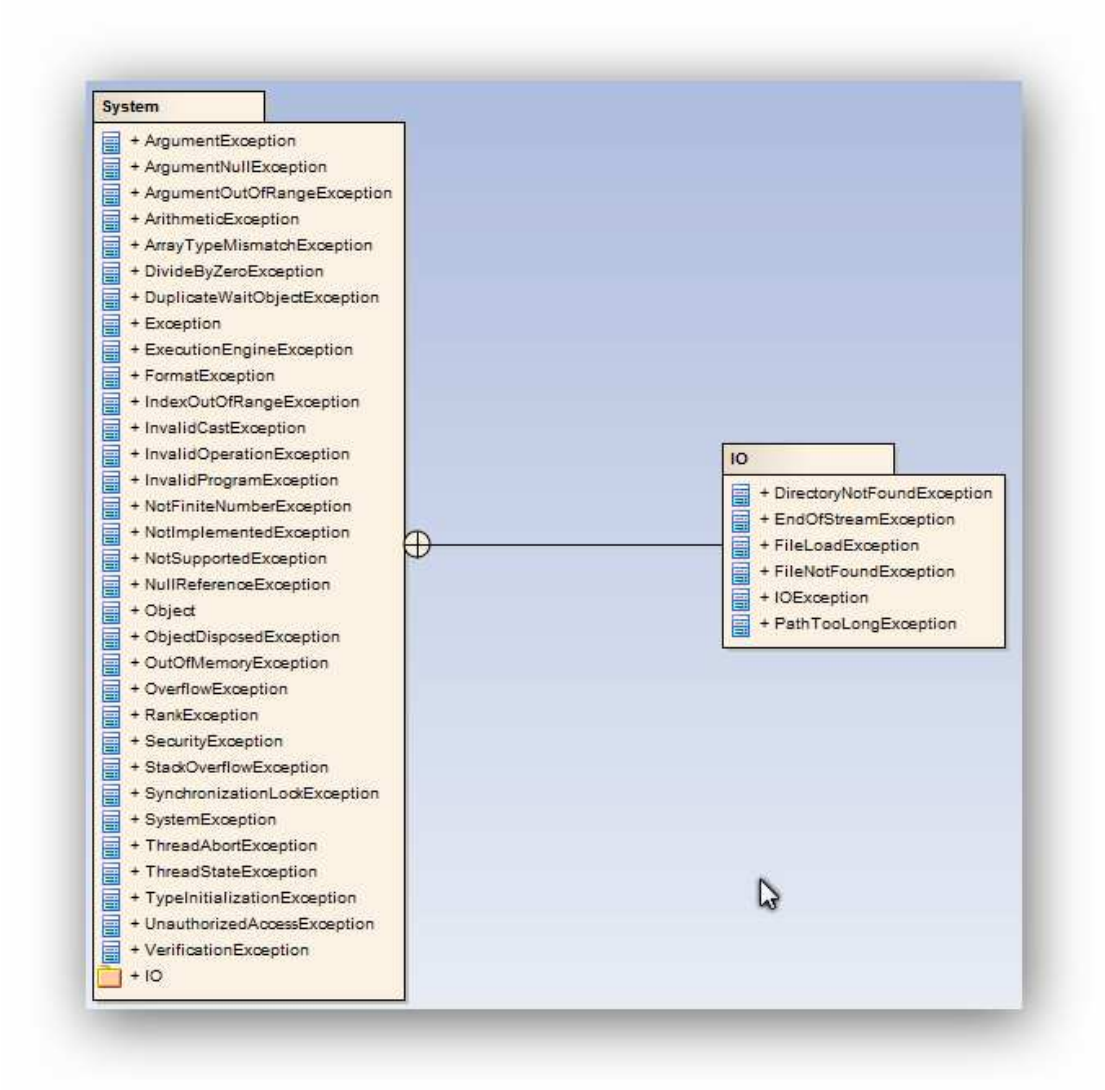
Algunas de las más utilizadas cadenas de herencias que .Net provee para la gestión de las diferentes posibles excepciones del Framework de .Net se muestran en la siguiente figura:



Se puede ver por separado, como ejemplo, las cadenas de herencia de IO.



El siguiente diagrama de paquetes muestra cómo se relacionan las excepciones de ejemplo mostradas



Como se pudo apreciar en los diagramas anteriores, los lenguajes proveen una cantidad de excepciones predefinidas. Algunas de las más comunes son las siguientes:

- `ArithmeticException`: La excepción se produce por errores en una operación aritmética, una conversión de tipo (cast) o una conversión numérica no admitida

  Ejemplo

  ```
  int i = 12 / 0 ó Dim i as Integer = 12 / 0
  ```

- `NullReferenceException`: Cualquier intento de acceder a un atributo o método de un objeto cuando no se creó una instancia del mismo, por ejemplo, se declara una variable de referencia del tipo de la clase que define el objeto pero no se creó una instancia de la clase con el operador `new` o `New` (C# o VB).

Ejemplo

```
Fecha f;
Console.WriteLine(f.toString());
```

- `InvalidCastException`: La excepción que se produce por la conversión de tipo inválida explícita o implícita
- `IndexOutOfRangeException`: Cuando se intenta acceder a un elemento de un vector más allá del límite que este tiene, se produce esta excepción. Esta clase no se puede heredar.
- `FormatException`: La excepción se produce cuando el formato de un argumento no satisface las especificaciones de los parámetros del método invocado.
- `FileNotFoundException`: La excepción se produce cuando se intenta acceder a un archivo que no existe en el disco.
- `ArgumentNullException`: La excepción se produce cuando una referencia de objeto `null` (`Nothing` en Visual Basic) se pasa a un método que no la acepta como argumento válido.

Excepciones que pueden ser lanzadas desde el ámbito de un método

.Net permite que un método lance una excepción desde una sentencia dentro de dicho método. Para ello debe utilizar la palabra reservada `throw` o `Throw` (C# o VB) que lleva a cabo el mencionado lanzamiento.

La palabra reservada `throw` o `Throw` (C# o VB) para lanzar excepciones es utilizada ampliamente en el Framework de .Net. Esto quiere decir que las clases provistas para programar lanzan excepciones y se debe conocer qué métodos hacen uso de cada una de ellas. Sin embargo, el hecho de conocer todos los métodos con todas las posibles excepciones lanzadas es un trabajo arduo y poco práctico. En lugar de eso, lo que los programadores hacen habitualmente es determinar qué áreas de código crítico poseen y en base a ello ver cuáles son las posibles excepciones a manejar.

Otra técnica muy común es provocar errores en el código para luego analizar las excepciones a utilizar y como gestionarlas.

Capturar y Manejar Excepciones

El manejo de excepciones se divide en un grupo de tres bloques que trabajan en conjunto:

- El bloque `try`
- Los bloques `catch`
- El bloque `finally`

Definición del bloque `try` o `Try`

El primer paso en la escritura de una manejador de excepciones es poner la sentencia en la cual se puede producir una determinada excepción dentro de un bloque `try` o `Try` (C# o VB). Se dice que el bloque `try` o `Try` (C# o VB) gobierna las sentencias encerradas dentro de él y define el ámbito (visibilidad) de cualquier manejador de excepciones (establecido por el o los bloques `catch` o `Catch` - C# o VB - subsiguientes) asociado con él.

Definición de los bloques `catch` o `Catch`

Se debe asociar un manejador de excepciones con un bloque `try` o `Try` (C# o VB) proporcionándole uno o más bloques `catch` o `Catch` (C# o VB) directamente después del bloque `try` o `Try` (C# o VB). La declaración debe realizarse a continuación de la marca que indique el fin del bloque `try` o `Try` (C# o VB).

Definición del bloque `finally` o `Finally`

El bloque `finally` o `Finally` (C# o VB) proporciona un mecanismo que permite a los métodos limpiarse a sí mismos sin importar lo que sucede dentro del bloque `try` o `Try` (C# o VB). Se utiliza el bloque `finally` o `Finally` (C# o VB) para cerrar archivos o liberar otros recursos del sistema.

El bloque `try` o `Try`

El primer paso en la construcción de un manejador de excepciones es encerrar las sentencias que podrían lanzar una excepción dentro de un bloque `try` o `Try` (C# o VB). En general, este bloque tiene el siguiente formato.

Ejemplo

C#

```
try {
        sentencias
}
```

VB

```
Try
        sentencias
End Try
```

El segmento de código etiquetado **sentencias** está compuesto por una o más sentencias legales de los lenguajes de .Net que podrían lanzar una excepción.

Para construir un manejador de excepción para un método de una clase, se necesita encerrar la sentencia que lanza la excepción en el método que realiza la invocación, dentro de un bloque `try` o `Try` (C# o VB).

Existe más de una forma de realizar esta tarea. Se puede poner cada una de las sentencias que potencialmente pudieran lanzar una excepción dentro un bloque `try` o `Try` (C# o VB) propio, y proporcionar manejadores de excepciones separados para cada uno de los bloques `try` o `Try` (C# o VB), o se puede poner todas las sentencias del método que realiza las invocaciones dentro de un sólo bloque `try` o `Try` (C# o VB) y asociar varios manejadores con él.

Es por esto que el bloque `try` o `Try` (C# o VB) gobierna las sentencias encerradas dentro del él y define el ámbito de cualquier manejador de excepción (establecido por el o los bloques `catch` o `Catch` - C# o VB) asociados al mismo. En otras palabras, si ocurre una excepción dentro del bloque `try` o `Try` (C# o VB), esta será manejada por el manejador de excepción asociado con el bloque de sentencias que encierra `try` o `Try` (C# o VB).

Los bloques `catch` o `Catch`

Como se mencionó anteriormente, la sentencia `try` o `Try` (C# o VB) define el ámbito de sus manejadores de excepción asociados. Se pueden asociar manejadores de excepción a una sentencia `try` o `Try` (C# o VB) proporcionando uno o más bloques `catch` o `Catch` (C# o VB) directamente después del bloque `try` o `Try` (C# o VB).

Ejemplo

C#

```
try {
…
} catch ( … ) {
…
} catch ( … ) {
…
} …
```

VB

```
Try
        …
Catch …
        …
Catch …
        …
End Try
```

No puede haber ningún código entre el final de la sentencia `try` o `Try` (C# o VB) y el principio de la primera sentencia `catch` o `Catch` (C# o VB).

La forma general de una sentencia `catch` o `Catch` (C# o VB) es la siguiente:

C#

```
catch (AlgunObjetoException nombreVariable) {
        sentencias
}
```

VB

```
Catch nombreVariable AlgunObjetoException
        sentencias
End Try
```

Como se puede observar, la sentencia `catch` o `Catch` (C# o VB) requiere *un sólo argumento formal*. El formato de esta sentencia es similar al de una declaración de método. El tipo del argumento AlgunObjetoException declara el tipo de excepción que el manejador puede gestionar y debe ser el nombre de una clase heredada de la clase Exception. Cuando los programas lanzan una excepción realmente están lanzado un objeto, y estos sólo pueden ser los derivados de la clase Exception. Por otro lado, nombreVariable es el nombre a través del cual el manejador puede referirse al objeto del tipo de la excepción capturada.

Manejo de sentencias `try – catch` o `Try – Catch`

Hasta el momento se mencionó las posibilidades que tiene el mecanismo de manejo de excepciones pero no se integró sus partes bajo las reglas que gobiernan su uso. A partir de este punto se tratará el tema.

Como se mencionó anteriormente, el código que puede lanzar una excepción en particular deberá colocarse dentro de un bloque `try` o `Try` (C# o VB) y a este se le debe adjuntar las lista de los correspondientes bloques `catch` o `Catch` (C# o VB) con las posibles excepciones que se pudieran producir. Cada bloque de sentencias en un `catch` o `Catch` (C# o VB) maneja la excepción en particular que este puede recibir como argumento. La forma en la cual se asocia un bloque cuando se produce la excepción es por el argumento definido para esta.

C#

```
try {
// código que puede lanzar una excepción
} catch (MiTipoException miExcep) {
// código que se ejecuta si la excepción
// MiTipoException es lanzada
} catch (Exception otraExcep) {
// código que se ejecuta si se lanza cualquier
// otra excepción
}
```

VB

```
Try
// código que puede lanzar una excepción
Catch miExcep As MiTipoException
// código que se ejecuta si la excepción
// MiTipoException es lanzada
Catch otraExcep As Exception
// código que se ejecuta si se lanza cualquier
// otra excepción
End Try
```

Los manejadores de excepciones pueden recibir como argumento una referencia a un objeto del tipo de una excepción y llamar a la propiedad Message de la excepción utilizando el nombre de la referencia declarado para ella, por ejemplo, e.

```
e.Message
```

Se puede acceder a las propiedades y métodos de las excepciones en la misma forma que accede a los de cualquier otro objeto, en particular, Message es una propiedad proporcionada por la clase Exception que almacena información adicional sobre el error ocurrido. La clase Exception también implementa una propiedad de sólo lectura con la que se puede imprimir el contenido de la pila de ejecución cuando ocurre la excepción. Las subclases de Exception pueden añadir otros métodos, propiedades o variables de instancia si así lo requirieran.

Para buscar qué métodos implementar en una excepción, se puede comprobar la definición de la clase y las definiciones de las clases que la preceden en la cadena de herencia, en caso de estar utilizando una subclase.

El bloque `catch` o `Catch` (C# o VB) contiene una serie de sentencias. Estas se ejecutan cuando se llama al manejador de cada excepción en particular. El CLR llama al manejador de excepción cuando este es el primero en la pila de llamadas cuyo tipo coincide con el de la excepción lanzada.

Se debe tener en cuenta, como se mencionó anteriormente, que si una excepción no es manejada por un bloque `try` – `catch` o `Try` – `Catch` (C# o VB), esta se lanza nuevamente al método que llamó al que recibió la excepción, continuando este proceso a lo largo de todos los métodos registrados en el stack.

Si una excepción es lanzada nuevamente hacia atrás a lo largo de todos los métodos que hicieron los respectivos llamados desde el que generó la excepción, llega a la función Main() y el programa termina en forma anormal si no se declara al menos un manejador para la excepción. En otras palabras, revisa hacia atrás la secuencia de llamados registrados a través del stack y si no encuentra un manejador apropiado para la excepción termina el programa indicando cual es la excepción que se produce y que no está manejada en toda la secuencia de llamados.

El bloque `finally` o `Finally`

El paso final en la creación de un manejador de excepción es proporcionar un mecanismo que limpie el estado del método antes (posiblemente) de permitir que el control pase a otra parte diferente del programa. Se puede hacer esto encerrando el código de limpieza dentro de un bloque `finally` o `Finally` (C# o VB).

Esta sentencia define un bloque que ***siempre*** se ejecuta, más allá que se haya capturado o no una excepción.

Ejemplo

C#

```
try {
        metodo1();
        metodo2();
} catch (Exception e) {
        CreaLogDelProblema(e);
} finally {
        realizarLiberacionRecursos();
}
```

VB

```
Try
        metodo1()
        metodo2()
Catch e As Exception
        CreaLogDelProblema(e))
```

```
            Finally
                  realizarLiberacionRecursos()
            End Try
```

El CLR siempre ejecuta las sentencias que hay dentro del bloque `finally` o `Finally` (C# o VB) sin importar lo que suceda dentro del bloque `try` o `Try` (C# o VB). Esto es, si se lanza la excepción y se deriva a un `catch` o `Catch` (C# o VB) o si no se lanza y se ejecuta normalmente el código. El principal beneficio de utilizarla es liberar recursos o la gestión apropiada de los mismos.

En el ejemplo anterior, realizarLiberacionRecursos() se ejecuta más allá que ocurra una excepción o no cuando se ejecute tanto metodo1() como metodo2() (al código dentro del bloque `try` o `Try` (C# o VB) se lo denomina también "código protegido").

La única posibilidad en la cual una sentencia `finally` o `Finally` (C# o VB) y su bloque asociado no se ejecutan es cuando se ejecuta un Environment.Exit() o Application.Exit() (el primero en aplicaciones por consola y el segundo en entornos gráficos) en algún lugar antes de su procesamiento en el código protegido, porque esto termina inmediatamente el programa, lo cual implica que el control del flujo se desvía de su secuencia normal. Si, por ejemplo, dentro del bloque `try` o `Try` (C# o VB) se encontrase una sentencia `return` o `Return` (C# o VB), el código dentro del bloque `finally` o `Finally` (C# o VB) se ejecuta igualmente antes que el método termine y retorne.

Una primera aproximación simple al mecanismo para el manejo de excepciones es analizar un caso simple en el cuál la excepción se lanza y se atrapa en un mismo método. Si bien este es el caso de uso más sencillo, es el más utilizado frecuentemente para manejar un error en el lugar donde se produce el mismo.

Ejemplo

C#

```
namespace sencillo
{
    class EjemploExcepciones
    {
        static void Main(string[] args)
        {
            try
            {
                Console.WriteLine("Ejecutando la sentencia try.");
                throw new NullReferenceException();
            }
           catch (NullReferenceException e)
            {
                Console.WriteLine("Atrapando la excepción #1.\n{0}", e.Message);
                Console.ReadKey();
            }
            catch
            {
                Console.WriteLine("Atrapando la excepción #2.");
            }
           finally
```

```
            {
                Console.WriteLine("ejecutando el bloque finally.");
                Console.WriteLine("");
                Console.WriteLine("Presione un tecla para salir.");
                Console.ReadKey();
            }
            Console.WriteLine("Siguiendo la ejecución");
            Console.WriteLine("Presione una tecla para salir ...");
        }
    }
}
```

VB

```
Public Class EjemploExcepciones
    Public Sub Inicio()
        Try
            Console.WriteLine("Ejecutando la sentencia try.")
            Throw New NullReferenceException()
        Catch e As NullReferenceException
            Console.WriteLine("Atrapando la excepción #1.\n{0}", e.Message)
            Console.ReadKey()
        Catch
            Console.WriteLine("Atrapando la excepción #2.")
        Finally
            Console.WriteLine("ejecutando el bloque finally.")
            Console.WriteLine("")
            Console.WriteLine("Presione un tecla para salir.")
            Console.ReadKey()
        End Try
        Console.WriteLine("Siguiendo la ejecución")
        Console.WriteLine("Presione una tecla para salir ...")
        Console.ReadKey()
    End Sub
End Class
```

El cual produce la siguiente salida

```
Ejecutando la sentencia try.
Atrapando la excepción #1.
Object reference not set to an instance of an object.
ejecutando el bloque finally.

Presione una tecla para salir.
```

Para completar lo enunciado hasta el momento, se puede retomar el ejemplo del recorrido del vector que termina anormalmente e incluir en el código el manejo de la excepción que se lanza. El código resultante sería el siguiente.

Ejemplo

C#

```
namespace vector
{
```

```
    class VectorConExcepcionManejada
    {
        public void ManejaVector()
        {
            int i = 0;
            int j = 0;

            String[] mensajes = { "Hola Mundo de Excepciones!",
                                  "No, no se rompió!", "Acá tampoco!!" };

            while (i < 4)
            {
                try
                {
                    Console.WriteLine(mensajes[i]);
                }
                catch (IndexOutOfRangeException e)
                {
                    Console.WriteLine("Error: " + e.Message);
                    Console.WriteLine("Reiniciando el valor del índice");
                    i = -1;
                }
                finally
                {
                    Console.WriteLine("Esto se imprime siempre");
                }
                i++;
                j++;
                if (j == 20)
                {
                    Console.WriteLine("Forzando la finalización del ciclo");
                    i = 4;
                }
            }
        }
    }
}
```

VB

```
Public Class VectorConExcepcionManejada
    Public Sub ManejaVector()
        Dim i As Integer = 0
        Dim j As Integer = 0
        Dim saludos() As String = {"Hola Mundo de Excepciones!",
                          "No, no se rompió!", "Acá tampoco!!"}
        While (i < 4)
            Try
                Console.WriteLine(saludos(i))
            Catch e As IndexOutOfRangeException
                Console.WriteLine("Error: " + e.Message)
                Console.WriteLine("Reiniciando el valor del índice")
                i = -1
            Finally

                Console.WriteLine("Esto se imprime siempre")
```

```
            End Try
            i += 1

            j += 1
            If j = 20 Then
                Console.WriteLine("Forzando la finalización del ciclo")
                i = 4
            End If
        End While
    End Sub
End Class
```

Notar el uso de una segunda variable para detener la ejecución del programa, ya que al manejar la excepción cuando se produce y cambiar el valor de la variable de control del ciclo, este se convierte en un ciclo infinito.

La salida producida es la siguiente:

```
Hola Mundo de Excepciones!
Esto se imprime siempre
No, no se rompió!
Esto se imprime siempre
Acá tampoco!!
Esto se imprime siempre
Error: Index was outside the bounds of the array.
Reiniciando el valor del índice
Esto se imprime siempre
Hola Mundo de Excepciones!
Esto se imprime siempre
No, no se rompió!
Esto se imprime siempre
Acá tampoco!!
Esto se imprime siempre
Error: Index was outside the bounds of the array.
Reiniciando el valor del índice
Esto se imprime siempre
Hola Mundo de Excepciones!
Esto se imprime siempre
No, no se rompió!
Esto se imprime siempre
Acá tampoco!!
Esto se imprime siempre
Error: Index was outside the bounds of the array.
Reiniciando el valor del índice
Esto se imprime siempre
Hola Mundo de Excepciones!
Esto se imprime siempre
No, no se rompió!
Esto se imprime siempre
Acá tampoco!!
Esto se imprime siempre
Error: Index was outside the bounds of the array.
```

```
Reiniciando el valor del índice
Esto se imprime siempre
Hola Mundo de Excepciones!
Esto se imprime siempre
No, no se rompió!
Esto se imprime siempre
Acá tampoco!!
Esto se imprime siempre
Error: Index was outside the bounds of the array.
Reiniciando el valor del índice
Esto se imprime siempre
Forzando la finalización del ciclo
```

Como se pudo apreciar en el ejemplo anterior, las excepciones se manejan dentro de bloques `try` – `catch` – `finally` o `Try` – `Catch` – `Finally` (C# o VB). Lo que el lenguaje requiere para manejar estos bloques es que el código que los utilice debe manejar el lanzamiento de la excepción mientras el método se encuentra en el stack (recordar que el mecanismo de excepciones está fuertemente asociado al recorrido a efectuar cuando se lanza una excepción a través de la pila de ejecución)

Rescritura y Excepciones

En .Net el manejo de excepciones es independiente de la rescritura de clases. Si un método en una superclase lanza una excepción es independiente de las posibles excepciones que lancen sus subclases. Esto es así aún si se utiliza rescritura del método.

.Net considera a las excepciones lanzadas en cada método de cada clase como propios de ellas y que no deben relacionarse entre sí con otras excepciones dentro de la cadena de herencia en la que se encuentren utilizadas.

El siguiente ejemplo muestra el caso puntual de la rescritura de un método y como las excepciones que se lanzan no están relacionadas con la cadena de herencia en la que se encuentran. Más aún, el método de la superclase lanza una excepción que es superclase de la excepción que se lanza en el método rescrito de su subclase.

Ejemplo

C#

```
namespace rescritura
{
    class BaseA
    {
        public virtual void MetodoA()
        {
            throw new FormatException("Error de formato");
        }
    }
}

amespace rescritura
{
```

```
    class DerivadaA1 : BaseA
    {
       public override void MetodoA(){
              throw new DivideByZeroException("Se intentó dividir por cero");
       }
    }
}

namespace rescritura
{
    class DerivadaA2 : DerivadaA1
    {
        public override void MetodoA()
        {
            throw new ArithmeticException("Acaba de ocurrir un error Aritmético");

        }
    }
}

namespace rescritura
{
    class Program
    {
        static void Main(string[] args)
        {
            BaseA ba = new BaseA();
            DerivadaA1 da1 = new DerivadaA1();
            DerivadaA2 da2 = new DerivadaA2();

            try
            {
                ba.MetodoA();
            }
            catch (FormatException fe)
            {
                Console.WriteLine("Se lanzó la excepción: " + fe.Message);
            }

            try
            {
                da1.MetodoA();
            }
            catch (DivideByZeroException de)
            {
                Console.WriteLine("Se lanzó la excepción: " + de.Message);
            }

            try
            {
                da2.MetodoA();
            }
            catch (ArithmeticException ae)
            {
                Console.WriteLine("Se lanzó la excepción: " + ae.Message);
```

```
            }

            try
            {
                da1.MetodoA();
            }
            catch (ArithmeticException ae)
            {
                Console.WriteLine("Se lanzó la excepción: " + ae.Message);
            }

            Console.ReadKey();
        }
    }
}
```

VB

```
Public Class BaseA
    Public Overridable Sub MetodoA()
        Throw New FormatException("Error de formato")
    End Sub
End Class

Public Class DerivadaA1
    Inherits BaseA

    Public Overrides Sub MetodoA()
        Throw New DivideByZeroException("Se intentó dividir por cero")
    End Sub
End Class

Public Class DerivadaA2
    Inherits DerivadaA1

    Public Overrides Sub MetodoA()
        Throw New ArithmeticException("Acaba de ocurrir un error Aritmético")
    End Sub
End Class

Module Module1

    Sub Main()
        Dim ba As New BaseA()
        Dim da1 As New DerivadaA1()
        Dim da2 As New DerivadaA2()

        Try
            ba.MetodoA()
        Catch fe As FormatException
            Console.WriteLine("Se lanzó la excepción: " + fe.Message)
        End Try

        Try
            da1.MetodoA()
        Catch de As DivideByZeroException
```

```
            Console.WriteLine("Se lanzó la excepción: " + de.Message)
        End Try

        Try
            da2.MetodoA()
        Catch ae As ArithmeticException
            Console.WriteLine("Se lanzó la excepción: " + ae.Message)
        End Try

        Try
            da1.MetodoA()
        Catch ae As ArithmeticException
            Console.WriteLine("Se lanzó la excepción: " + ae.Message)
        End Try

        Console.ReadKey()
    End Sub
End Module
```

Notar que la instrucción catch no sólo funciona normalmente sino que también cuando es posible asignarle un objeto del tipo Exception por conversión de tipo, lo hace y gestiona la excepción, como lo demuestra la siguiente salida:

```
Se lanzó la excepción: Error de formato
Se lanzó la excepción: Se intentó dividir por cero
Se lanzó la excepción: Acaba de ocurrir un error Aritmético
Se lanzó la excepción: La división dio cero
```

Creación de excepciones personalizadas

Una de las principales ventajas de manejar excepciones para el control de errores en el flujo de un programa es que éstas son clases en sí mismas y tiene habilitado todas las características de lo que se puede realizar con una clase. De esta manera, se puede sacar ventaja de este hecho creando excepciones propias tan sólo con generar una subclase de la clase Exception. Esto permite personalizar el tipo de excepción a manejar haciéndola adecuada al tipo de tratamiento que se quiera otorgar al error producido. Como ejemplo de creación de clases de excepción, se muestra el siguiente código.

Ejemplo

C#

```
namespace personalizada
{
    class ExcepcionSinSaldo : Exception
    {
        private double cantidad;

        public ExcepcionSinSaldo(string mensaje, double cantidad)
            : base(mensaje)
        {
            this.cantidad = cantidad;
```

```
        }

        public double Cantidad
        {
            get
            {
                return cantidad;
            }
        }
    }
}

namespace personalizada
{
    class ExcepcionFueraDeRango : Exception
    {
        private double cantidad;

        public ExcepcionFueraDeRango(string mensaje, double cantidad):base(mensaje)
        {
            this.cantidad = cantidad;
        }
        public double Cantidad
        {
            get
            {
                return cantidad;
            }
        }
    }
}
```

VB

```
Public Class ExcepcionSinSaldo
    Inherits Exception

    Private _cantidad As Double

    Public Sub New(mensaje As String, cantidad As Double)
        MyBase.New(mensaje)
        Me._cantidad = cantidad
    End Sub
    Public ReadOnly Property Cantidad() As Double
        Get
            Return _cantidad
        End Get
    End Property
End Class

Public Class ExcepcionFueraDeRango
    Inherits Exception

    Private _cantidad As Double

    Public Sub New(mensaje As String, cantidad As Double)
```

```
        MyBase.New(mensaje)
        Me._cantidad = cantidad
    End Sub
    Public ReadOnly Property Cantidad() As Double
        Get
            Return _cantidad
        End Get
    End Property
End Class
```

Uno de los detalles a tener en cuenta cuando se crea la excepción es construir adecuadamente la superclase. Por lo general, se le pasa una cadena con un mensaje que sea descriptivo del error que representa el objeto del tipo excepción, el cual puede recuperarse posteriormente cuando se lance este objeto con la propiedad Message.

Si se quiere utilizar la excepción que se creó anteriormente, un ejemplo del código que se encontraría en una clase que defina los métodos que utilicen la excepción es el siguiente:

Ejemplo

C#

```
namespace personalizada
{
    class Cuenta
    {
        private double balance;

        public Cuenta(double balance)
        {
            this.balance = balance;
        }

        public void Deposita(double cantidad)
        {
            if (cantidad < 0)
                throw new ExcepcionFueraDeRango(
                    "No se puede depositar valores negativos", cantidad);
            else
                if(cantidad == 0)
                    throw new ExcepcionFueraDeRango("No se puede depositar 0",
                                        cantidad);
                else
                balance += cantidad;
        }

        public void Retira(double cantidad)
        {
            if (balance < cantidad)
                throw new ExcepcionSinSaldo("La cuenta no tiene suficiente saldo",
                                        cantidad);
            else
                balance -= cantidad;
```

```
        }
    }
}
```

VB

```
Public Class Cuenta
    Private _balance As Double

    Public Sub New(_balance As Double)
        Me._balance = _balance
    End Sub

    Public Sub Deposita(cantidad As Double)
        If cantidad < 0 Then
            Throw New ExcepcionFueraDeRango(
                    "No se puede depositar valores negativos", cantidad)
        Else
            If cantidad = 0 Then
                Throw New ExcepcionFueraDeRango("No se puede depositar 0", cantidad)
            Else
                _balance += cantidad
            End If
        End If
    End Sub

    Public Sub Retira(cantidad As Double)
        If (_balance < cantidad) Then
            Throw New ExcepcionSinSaldo(
                    "La cuenta no tiene suficiente saldo", cantidad)
        Else
            _balance -= cantidad
        End If
    End Sub
End Class
```

El siguiente ejemplo muestra cómo utilizar las propiedades de una excepción personalizada. Notar que todas las propiedades de Exception se encuentran disponibles por medio de la herencia.

Ejemplo

C#

Primero: se declara la excepción

```
namespace propiedades
{
    class DesbordamientoDeTablaDeLogException : Exception
    {
        const string mensajeDeDesbordamiento = "La tabla de log se desbordó.";

        public DesbordamientoDeTablaDeLogException(
            string mensajeAuxiliar, Exception interna) :
            base(String.Format("{0} - {1}",
                    mensajeDeDesbordamiento, mensajeAuxiliar), interna)
```

```
        {
            this.HelpLink = "http://msdn.microsoft.com";
            this.Source = "DesbordamientoDeTablaDeLogException";
        }

    }
}
```

VB

```
Public Class DesbordamientoDeTablaDeLogException
    Inherits Exception

    Private Const _mensajeDeDesbordamiento As String = "La tabla de log se
desbordó."

    Public Sub New(mensajeAuxiliar As String, interna As Exception)
        MyBase.New(String.Format("{0} - {1}", _
                   _mensajeDeDesbordamiento, mensajeAuxiliar), interna)
        Me.HelpLink = "http://msdn.microsoft.com"
        Me.Source = "DesbordamientoDeTablaDeLogException"
    End Sub
End Class
```

Segundo: se utiliza la excepción en una clase

C#

```
namespace propiedades
{
    class TablaDeLog
    {
        protected string[] areaDeLog;
        protected int elementosEnUso;

        public TablaDeLog(int numElementos)
        {
            areaDeLog = new string[numElementos];
            elementosEnUso = 0;
        }

        // El método AgregarRegistro lanza una excepción derivada de Exception
        // si el límite del vector se sobrepasá y se lanza la excepción
        public int AgregarRegistro(string nuevoRegistro)
        {
            try
            {
                areaDeLog[elementosEnUso] = nuevoRegistro;
                return elementosEnUso++;
            }
            catch (Exception e)
            {
                throw new DesbordamientoDeTablaDeLogException(
                    String.Format("El registro \"{0}\" no se guardó en el log.",
                        nuevoRegistro), e);
            }
```

```
        }
    }
}
```

VB

```
Public Class TablaDeLog
    Protected areaDeLog() As String
    Protected elementosEnUso As Integer

    Public Sub New(numElementos As Integer)
        Dim areaDeLog(numElementos) As String
        Me.areaDeLog = areaDeLog
        elementosEnUso = 0
    End Sub
    ' El método AgregarRegistro lanza una excepción derivada de Exception
    ' si el límite del vector se sobrepasá y se lanza la excepción
    Public Function AgregarRegistro(nuevoRegistro As String) As Integer
        Try
            areaDeLog(elementosEnUso) = nuevoRegistro
            elementosEnUso += 1
            Return elementosEnUso
        Catch e As Exception
            Throw New DesbordamientoDeTablaDeLogException( _
                String.Format("El registro '{0}' no se guardó en el log.", _
                    nuevoRegistro), e)
        End Try
    End Function
End Class
```

Tercero: se atrapa la excepción y se muestra la información relevante

C#

```
namespace propiedades
{
    class EjemploExcepcionDeUsuario
    {
        // Crear una tabla de log y forzar un desbordamiento.
        public static void Main()
        {
            TablaDeLog log = new TablaDeLog(4);

            Console.WriteLine(
                "Este ejemplo de \n   Exception.Message, \n" +
                "   Exception.HelpLink, \n   Exception.Source, \n" +
                "   Exception.StackTrace, y \n   Exception." +
                "TargetSite \ngenera la siguiente salida.");

            try
            {
                for (int cantidad = 1; ; cantidad++)
                {
                    log.AgregarRegistro(
                        String.Format(
                            "Número de registro de log {0}", cantidad));
```

```
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine("\nMessage (mensaje) ---\n{0}", ex.Message);
                Console.WriteLine(
                    "\nHelpLink (enlace para la ayuda) ---\n{0}", ex.HelpLink);
                Console.WriteLine("\nSource (fuente) ---\n{0}", ex.Source);
                Console.WriteLine(
                    "\nStackTrace (volcado de pila) ---\n{0}", ex.StackTrace);
                Console.WriteLine(
                    "\nTargetSite (sitio al que apunta) ---\n{0}", ex.TargetSite);
                Console.WriteLine("");
                Console.WriteLine("Presione un tecla para salir.");
                Console.ReadKey();
            }
        }
    }
}
```

VB

```
Public Class EjemploExcepcionDeUsuario
    Public Sub Inicio()
        Dim log As New TablaDeLog(4)

        Console.WriteLine(
            "Este ejemplo de " + Environment.NewLine + _
             "   Exception.Message, " + Environment.NewLine + _
            "   Exception.HelpLink, " + Environment.NewLine + _
            "   Exception.Source, " + Environment.NewLine + _
            "   Exception.StackTrace, y " + Environment.NewLine + _
            "   Exception.TargetSite " + Environment.NewLine + _
            "genera la siguiente salida.")

        Try
            While (True)
                Dim cantidad As Integer
                cantidad += 1
                log.AgregarRegistro(
                    String.Format("Número de registro de log {0}", cantidad))
            End While
        Catch e As Exception
            Console.WriteLine(
                Environment.NewLine + _
                "Message (mensaje) ---" _
                + Environment.NewLine + "{0}", e.Message)
            Console.WriteLine(
               vbNewLine + "HelpLink (enlace para la ayuda) ---" + _
               vbNewLine + "{0}", e.HelpLink)
            Console.WriteLine(
                vbNewLine + "Source (fuente) ---" + _
                vbNewLine + "{0}", e.Source)
            Console.WriteLine(
               vbNewLine + "StackTrace (volcado de pila) ---" + _
               vbNewLine + "{0}", e.StackTrace)
```

```
            Console.WriteLine(
                vbNewLine + "TargetSite (sitio al que apunta) ---" + _
                vbNewLine + "{0}", e.TargetSite)
            Console.WriteLine("")
            Console.WriteLine("Presione un tecla para salir.")
            Console.ReadKey()
        End Try
    End Sub
End Class
```

Relanzamiento de una excepción

Existen situaciones en las cuales se atrapa una excepción y se relanza la misma para que el error sea gestionado por el método que realizó la invocación. Este tipo de diseño no es el más apropiado para la mayoría de las situaciones. Sin embargo hay un principio de buen diseño de clases que establece:

Cada clase debe manejar las situaciones anómalas que pertenecen a los servicios que brinde.

Este principio plantea la disyuntiva de cómo se debe manejar correctamente una situación anómala (o error) cuando parte del mismo pertenece a un servicio de una clase y otra parte pertenece a otro. La solución parece simple en un primer momento pero el diseño no lo es tanto. Uno de los problemas principales es realizar el seguimiento del error y donde se originó.

La principal herramienta para el seguimiento de un error es el volcado de pila. El seguimiento de la pila es una de las informaciones más útiles que lleva una excepción. Para no perder información útil del seguimiento del error y no perder información del volcado de la pila, existen dos formas de lanzar una excepción: una correcta y otra errada.

La errada es:

C#

```
            try
            {
                // Código que lanza una excepción
            }
            catch (Exception ex)
            {
                // El código que maneja la excepción lanzada
                throw ex;
            }
```

VB

```
             Try
             // Código que lanza una excepción
             Catch ex As Exception
             // El código que maneja la excepción lanzada
               Throw ex
             End Try
```

¿Por qué está mal? Porque, cuando se examina el seguimiento de la pila, el punto de origen de la excepción será la línea "`throw ex;`" en C# o "`Throw ex`" en VB, ocultando la ubicación del error real.

La forma correcta es:

C#

```
try
{
    // Código que lanza una excepción
}
catch (Exception ex)
{
    // El código que maneja la excepción lanzada
    throw;
}
```

VB

```
Try
// Código que lanza una excepción
Catch ex As Exception
// El código que maneja la excepción lanzada
  Throw
End Try
```

¿Qué ha cambiado? En lugar de "`throw ex;`" en C# o "`Throw ex`" en VB, que lanzará una nueva excepción y borrar el seguimiento que da el volcado de la pila, simplemente se utiliza "`throw;`" en C# o "`Throw`" en VB. Si no se especifica la excepción, la sentencia `throw` o `Throw` (C# o VB) simplemente vuelve a lanzar la misma excepción que atrapó la sentencia `catch` o `Catch` (C# o VB). Esto mantendrá el seguimiento de la pila intacto y además permite colocar código en bloques de captura. Si se modifica la clase Uno mostrada en un ejemplo anterior, se puede apreciar su utilización.

Ejemplo

C#

```
namespace relanzar
{
    class Uno
    {
        public void PrimerMetodo()
        {
            Dos d = new Dos();
            try
            {
                d.SegundoMetodo();
            }
            catch (ArithmeticException e)
            {
                Console.WriteLine("Se atrapó la excepción de la clase Dos");
```

```
                Console.WriteLine("-------------------------------------");
                Console.WriteLine("Volcado de pila: " + e.StackTrace);
                Console.WriteLine("-------------------------------------");
                throw;
            }
            finally
            {
                Console.WriteLine("Esto se ejecuta siempre. Estoy en la clase Uno");
            }
        }
    }
}
```

VB

```
Public Class Uno
    Public Sub PrimerMetodo()
        Dim d As New Dos()

        Try
            d.SegundoMetodo()
        Catch e As ArithmeticException
            Console.WriteLine("Se atrapó la excepción de la clase Dos")
            Console.WriteLine("-------------------------------------")
            Console.WriteLine("Volcado de pila: " + e.StackTrace)
            Console.WriteLine("-------------------------------------")
            Throw
        Finally
            Console.WriteLine("Esto se ejecuta siempre. Estoy en la clase Uno")
        End Try
    End Sub
End Class
```


Buenas prácticas en el uso de excepciones

Implementar la gestión de excepciones en las aplicaciones es fácil de usar, pero como con cualquier construcción de programación, es importante seguir un buen diseño. La siguiente lista explica algunas de las mejores prácticas para el manejo de excepciones:

- Una excepción debe ser apropiada para la condición de error que se detecta.
- La lógica de la aplicación no debe confiar en bloques `try` – `catch` o `Try` – `Catch` (C# o VB) para funcionar en condiciones normales. Se debe diseñar los métodos, de modo que, en circunstancias normales, no se produzcan excepciones. Sólo atrapar y lanzar excepciones para condiciones que están fuera del flujo esperado de la lógica de una aplicación.
- Los errores que se produzcan por anomalías en las reglas de negocio que gobiernan el diseño de una aplicación, deben tratarse como situaciones anómalas y diseñarse las excepciones acordes para manejarlas. Un ejemplo de esto puede ser crear una excepción que maneja un débito en cuenta corriente para un cliente que no la posee.
- Al definir varios bloques catch, ordenarlos desde el más al menos específico. Si se atrapa el tipo Exception, debe ser el último manejador en un conjunto de bloques de captura de excepción `catch` o `Catch` (C# o VB)

- Capturar y registrar los mensajes detallados de las excepciones con fines de diagnóstico, y luego mostrar mensajes fáciles al usuario. Recordar que cualquier texto que se muestra al usuario debe ser localizable, y dicho texto puede ser recuperada de archivos de recursos.
- No mostrar mensajes detallados de excepción a los usuarios, ya que uno malintencionado podría utilizar la información detallada para hacer que la aplicación no funcione correctamente, o incluso tener acceso a información protegida. Un error común que se produce en las capas de acceso a datos es proporcionar información detallada de error resultante de una consulta incorrecta de una base de datos. Esto puede permitir a un usuario malicioso entender la lógica subyacente en la aplicación y usar este conocimiento para atacar el sistema.
- Un manejo de excepciones eficaz debe permitir a la aplicación recuperarse de excepciones, y que el usuario pueda seguir utilizando la aplicación. En el caso de una excepción, el usuario no debe perder los datos y la aplicación no debe fallar.

Un buen uso de estas buenas prácticas es el de utilizar el registro de eventos del sistema operativo. El siguiente ejemplo modifica levemente la clase Uno utilizada anteriormente para conseguir este objetivo.

Ejemplo

C#

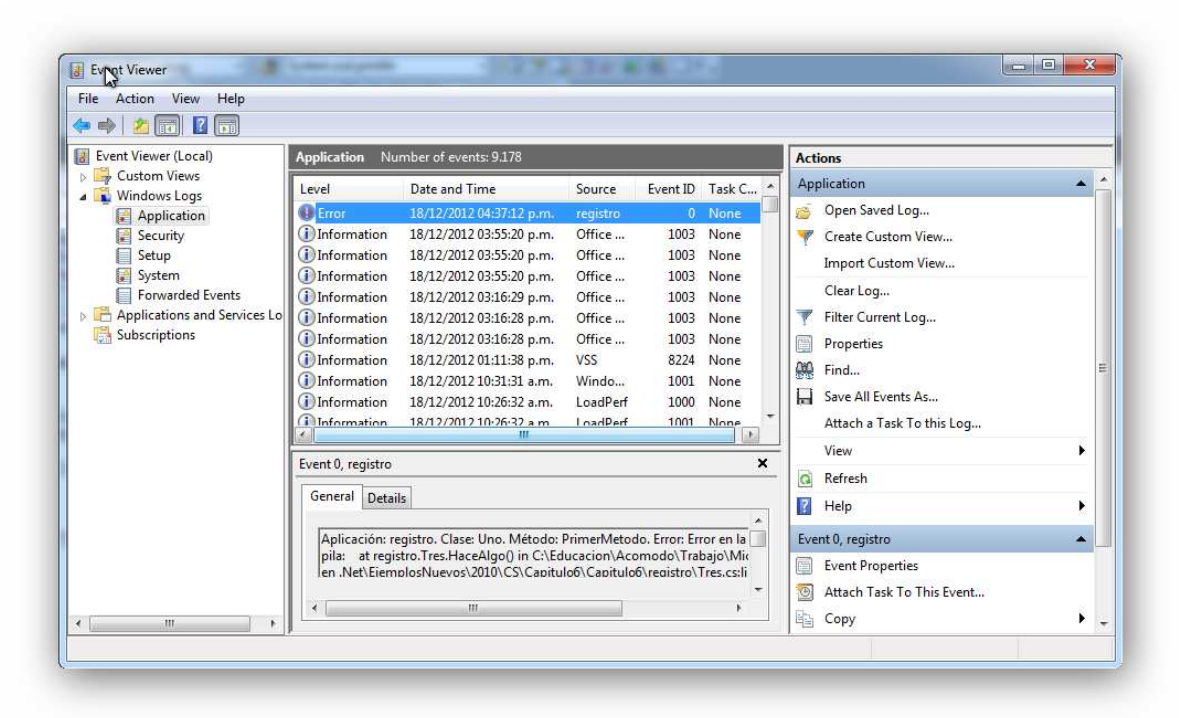
```
namespace registro
{
    class Uno
    {
        public void PrimerMetodo()
        {
            Dos d = new Dos();
            try
            {
                d.SegundoMetodo();
            }
            catch (ArithmeticException e)
            {
                Console.WriteLine("Se atrapó la excepción de la clase Dos");
                string fuente = "registro";
                string log = "Application";
                string mensaje =
             "Aplicación: registro. Clase: Uno. Método: PrimerMetodo. Error: "
                    + e.Message + ". Volcado de pila: " + e.StackTrace;
                // Verificar si el origen del evento existe, y si no, crearlo.
                if (!EventLog.SourceExists(fuente))
                {
                    EventLog.CreateEventSource(fuente, log);
                }

                // Escribir el mensaje en el log de eventos de Windows.
                EventLog.WriteEntry(fuente, mensaje, EventLogEntryType.Error);
```

```
                throw new ArithmeticException("Error en la clase Uno");
            }
            finally
            {
                Console.WriteLine("Esto se ejecuta siempre. Estoy en la clase Uno");
            }
        }
    }
}
```

VB

```
Public Class Uno
    Public Sub PrimerMetodo()
        Dim d As New Dos()

        Try
            d.SegundoMetodo()
        Catch e As ArithmeticException
            Console.WriteLine("Se atrapó la excepción de la clase Dos")
            Dim fuente As String = "registro"
            Dim log As String = "Application"
            Dim mensaje As String = _
                "Aplicación: registro. Clase: Uno. Método: PrimerMetodo. Error: " _
                    + e.Message + ". Volcado de pila: " + e.StackTrace
            ' Verificar si el origen del evento existe, y si no, crearlo.
            If Not EventLog.SourceExists(fuente) Then
                EventLog.CreateEventSource(fuente, log)
            End If
            ' Escribir el mensaje en el log de eventos de Windows.
            EventLog.WriteEntry(fuente, mensaje, EventLogEntryType.Error)
            Throw New ArithmeticException("Error en la clase Uno")
        Finally
            Console.WriteLine("Esto se ejecuta siempre. Estoy en la clase Uno")
        End Try
    End Sub
End Class
```


El resultado obtenido cuando se produce el error se puede verificar en la aplicación del “visor de eventos” en las herramientas administrativas. Cuando se ejecuta la aplicación se debe revisar la sección Windows Logs →Application para visualizar lo ocurrido, como muestra la siguiente imagen.



Características especiales de las excepciones dependientes del lenguaje

C#

Usando las palabras claves checked y unchecked

Todas las variables numéricas tienen un valor máximo. Si se incrementa un número, por ejemplo un entero, que alcanzó dicho valor máximo, el resultado es un desbordamiento numérico. Sin embargo, la aritmética de enteros es tan común que la comprobación de desbordamiento numérico después de cada operación podría afectar seriamente el rendimiento de las aplicaciones.

Por lo tanto, las aplicaciones C# se ejecutan por defecto con la comprobación de desbordamiento de valores enteros numéricos desactivada. En estas aplicaciones, existe el riesgo de desbordamiento numérico que puede conducir a resultados incorrectos. Si se incrementa una variable numérica entera que almacena el valor entero más grande posible, el resultado es un valor negativo.

Si se tiene una sección de código que puede provocar un desbordamiento numérico, se puede restablecer la comprobación de desbordamiento mediante la palabra clave `checked`.

También puede habilitar la comprobación de desbordamiento para una aplicación completa y desactivarla de forma local mediante el uso de la palabra clave `unchecked`.

Nota: se puede activar y desactivar la comprobación de desbordamiento en Microsoft Visual Studio 2010 mediante las propiedades del proyecto en la pestaña Build y presionando el botón Advanced en la esquina inferior derecha de la página. En el cuadro de diálogo Configuración de generación avanzada, activar o desactivar la casilla la comprobación de desbordamiento. aritmético.

Uso de un bloque ***checked***

Se utiliza la palabra clave `checked` para definir un bloque de código entre llaves que incluya la comprobación de desbordamiento numérico para dicho bloque. Si se produce un desbordamiento numérico, la instrucción que lo provocó lanzará una excepción OverflowException. En el ejemplo de código siguiente se muestra cómo habilitar la comprobación de desbordamiento y detectar la excepción OverflowException.

Ejemplo

C#

```
checked
{
       int x = ...;
       int y = ...;
       int z = ...;
       ...

       try
       {
              z = x * y; // Esto puede causar un desbordamiento numérico
       }

       catch (OverflowException ex)
       {
              ... // Manejar excepciones de desbordamiento
       }
       ...
}
```

También se puede utilizar la palabra clave para definir una expresión individual. El único cambio es que dicha expresión se coloca entre paréntesis, como muestra el siguiente ejemplo.

Ejemplo

C#

```
namespace verificaciones
{
    class OverFlowTest
    {
        // Inicializar la variable con máximo entero posible.
        static int maxIntValue = 2147483647;

        // Utilizar la palabra clave checked.
        static int MetodoConVerificacion()
```

```
        {
            int z = 0;
            try
            {
                // La siguiente instrucción lanza una excepción
                // porque verifica el desbordamiento.
                z = checked(maxIntValue + 10);
            }
            catch (System.OverflowException e)
            {
                Console.WriteLine("Atapado a causa del checked:  " + e.ToString());
            }
            // El valor de z se conserva en 0.
            return z;
        }

        // Uso del código sin verificar
        static int MetodoSinVerificacion()
        {
            int z = 0;
            try
            {
                // Realizar el mismo cálculo sin verificar
                // No lanza excepción alguna
                z = maxIntValue + 10;
            }
            catch (System.OverflowException e)
            {
                // Esto nunca se ejecuta.
                Console.WriteLine("Atapado a causa del checked:  " + e.ToString());
            }
            // Por no detectar el desbordamiento, la suma de 2147483647 + 10 es
            // -2147483639 lo cual es el valor retornado.
            return z;
        }

        static void Main()
        {
            Console.WriteLine("\nEl valor de salida CON checked: {0}",
                              MetodoConVerificacion());
            Console.WriteLine("El valor de salida SIN checked: {0}",
                              MetodoSinVerificacion());
            Console.ReadKey();
        }
    }
}
```


El uso de un bloque sin comprobar

Si se ha habilitado la comprobación de desbordamiento para una aplicación mediante las propiedades del proyecto, se puede utilizar la palabra clave `unchecked` para suprimir la comprobación de desbordamiento en un bloque o una expresión individual. La sintaxis para la declaración es la misma que para un bloque comprobado.

VB

Manejo estructurado y no estructurado de errores

El control estructurado de excepciones simplemente es aquel que con ayuda de una estructura de control que contiene excepciones, bloques aislados de código y filtros, poder crear un mecanismo para el manejo de dichas excepciones. Esto permite que el código pueda diferenciar entre los diferentes tipos de errores y reaccionar de acuerdo con las circunstancias. En el manejo de excepciones NO estructurado, una instrucción `On Error` en el inicio del código controla todas las excepciones.

El control estructurado de excepciones es significativamente más versátil, robusto y flexible que el no estructurado. Si es posible, se debe utilizar el control de excepciones estructurado. Sin embargo, es posible utilizar el control no estructurado de excepciones en estas circunstancias:

- Se actualiza una aplicación escrita en una versión anterior de Visual Basic.
- Es necesario utilizar la instrucción `Resume Next`, que no se admite en el manejo de excepciones estructurado.

No es posible combinar el control de excepciones estructurado y no estructurado en el mismo método. Si se utiliza una instrucción `On Error`, no se puede utilizar una instrucción `Try ... Catch` en la misma función o subproceso.

Nota: si bien se menciona `On Error ... Resume Next` **se recomienda ampliamente no utilizarlo**

Extensiones a la cláusula `Catch`

Una cláusula `Catch` puede tomar tres formas posibles: `Catch`, `Catch...As` y `Catch...When`.

Una cláusula `Catch` sin la palabra clave `When` permite al bloque de instrucciones asociado manejar cualquier excepción. Las cláusulas `Catch ... As` y `Catch ... When` capturan una excepción específica y permiten al bloque de instrucciones asociado indicarle a la aplicación qué hacer. Las cláusulas Catch .. . As y Catch ... When también se pueden combinar en una sola sentencia, como `Catch ex As Exception When intResult <> 0`.

Notar en el siguiente ejemplo como se condicionan las capturas en base al resultado de una expresión aritmética.

Ejemplo

VB

```
Module Module1

    Sub Main()
        Dim contador As Integer = 0
        Dim total As Integer = 10
        While contador < 11
            contador += 1
            Try
                If contador - total = 0 Then
```

```
                    Throw New Exception
                Else
                    If contador - total = -1 Then
                        Throw New Exception
                    End If
                End If
            Catch ex As Exception When contador - total = 0
                Console.WriteLine("Se acabaron los números para restar del total")
            Catch ex As Exception When contador - total < 0
                Console.WriteLine("Se intentó restar y no había más en total.")
            Finally
                Console.WriteLine("Se continúa el ciclo.")
            End Try
        End While
        Console.ReadKey()
    End Sub

End Module
```

La salida que produce el código anterior es la siguiente:

```
Se continúa el ciclo.
Se continúa el ciclo.
Se continúa el ciclo.
Se continúa el ciclo.
Se continúa el ciclo.
Se continúa el ciclo.
Se continúa el ciclo.
Se continúa el ciclo.
Se intentó restar y no había más en total.
Se continúa el ciclo.
Se acabaron los números para restar del total
Se continúa el ciclo.
Se continúa el ciclo.
```

Aserciones, aseveraciones o afirmaciones

Una aserción es una instrucción que contiene una expresión booleana la cual el programador sabe que en un momento dado de la ejecución del programa se debe evaluar a verdadero (de ahí la aserción o afirmación).

Este es un método habitual para la verificación formal de algoritmos. Básicamente se trata de afirmaciones respecto de precondiciones, post condiciones e invariantes a lo largo del flujo de un programa en puntos determina dos del mismo. Esto implica que verificando que es cierta la expresión booleana propuesta en la afirmación, se comprueba que el programa se ejecuta dentro de los límites que el programador demarca y además reduce la posibilidad de errores.

La clase Debug

Una descripción completa de esta clase en este punto excede los objetivos de este capítulo. Sin embargo, no se puede hablar de aseveraciones sin mencionarla porque las afirmaciones se realizan con un método estático que le pertenece.

El espacio de nombre en el cual se encuentra la clase es System.Diagnostics y deberá ser incluido para utilizarla.

Al usar los métodos de la clase Debug para imprimir información de depuración y comprobar la lógica con aserciones, se puede hacer que el código sea más sólido sin afectar el rendimiento ni el tamaño del código cuando se crea la versión final.

En Visual Studio. NET, la creación de proyectos se realiza siempre en modo de depuración mientras se trabaja en los mismos. Esto determina que se habilite automáticamente la inclusión de código para este fin que el entorno agrega de manera transparente para el programador. Sin embargo, **dicho código no se incluye en la creación de versiones finales** (Release). **Se puede deshabilitar** este funcionamiento por defecto en el entorno con una bandera en las opciones de compilación o con marcas especiales en los archivos fuentes para que sólo se deshabilite en el que tiene la declaración. El objetivo de la explicación actual es comprender este funcionamiento por defecto para generar código robusto.

De los diferentes métodos que posee la clase Debug, se centrará la atención en el método Assert().

El método Debug.Assert

Las aserciones admiten tres tipos de sintaxis como se muestran en las siguientes tablas:

C#	VB
`public static void Assert(` `bool condición)`	`Overloads Public Shared Sub Assert( _` `ByVal condición As Boolean)`

`public static void Assert(`
`bool condición,`
`string mensaje);`
	`Overloads Public Shared Sub Assert( _`
	`ByVal condición As Boolean, _`
	`ByVal mensaje As String)`

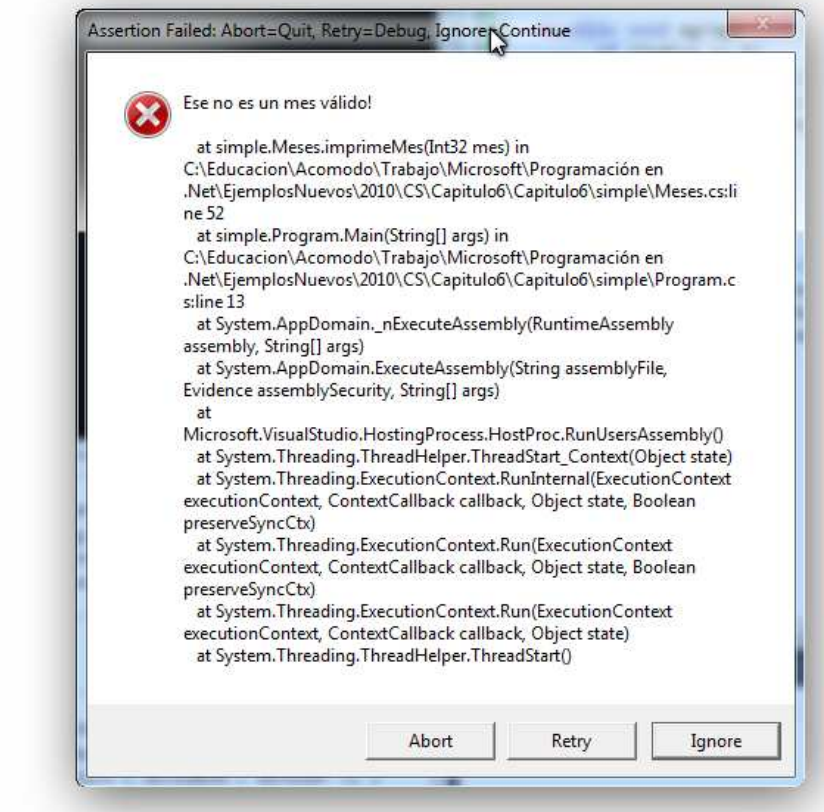
`public static void Assert(`
`bool condición,`
`string mensaje,`
`string mensajeDetallado);`
	`Overloads Public Shared Sub Assert( _`
	`ByVal condición As Boolean, _`
	`ByVal mensaje As String, _`
	`ByVal mensajeDetallado As String)`

En donde el significado de los parámetros es el siguiente:

Parámetro	Descripción

condición	`true` o `True` (C# o VB) para evitar que se muestre el mensaje. En otro caso, `false` o `False` (C# o VB)
mensaje	El mensaje a mostrar
mensajeDetallado	Un mensaje detallado a mostrar

Se debe definir una expresión que dará como resultado una expresión booleana (verdadero o falso), dicha expresión es precisamente la que será aseverada por .Net. Si esta expresión resulta verdadera, el flujo de ejecución del programa continuará normalmente. Sin embargo, si resulta falsa la expresión, la ejecución del programa será interrumpida indicando el error de aseveración ("assertion") y mostrando una ventana similar a la de la siguiente figura.



La única diferencia que existe entre las declaraciones antes mencionadas, es la cantidad de información adicional que se presenta cunado ocurre el error de afirmación. Por ejemplo, la segunda sobrecarga del método define una expresión adicional que permite agregar información extra acerca de la aseveración levantada por el programa, dicha declaración puede ser desde una

variable hasta un método que dé como resultado una cadena. Este resultado es un String que sirve como argumento del método que dispara la aseveración para ser desplegado como información adicional al momento de ejecución. La primera declaración de las sobrecargas del método para la aseveración simplemente verifica la validez de la primera expresión sin proporcionar detalles específicos del error.

En ambos casos, si la expresión booleana se evalúa como falso, se genera un error de aserción y el programa se interrumpe brindando en una ventana emergente la posibilidad de tomar tres acciones por parte del usuario:

- **Abort**: cancela la ejecución del programa
- **Retry**: retorna el punto de ejecución al programa en modo seguimiento (Trace)
- **Ignore**: Continua la ejecución del programa como si no hubiese habido un error de afirmación.

Si se utiliza el segundo o tercer formato, las expresiones que determinan el mensaje o la información adicional respecto del error, deben ser del tipo cadena y se utiliza para complementar el o los mensajes que aparece en la pantalla cuando se notifica la aserción.

Ejemplo

C#

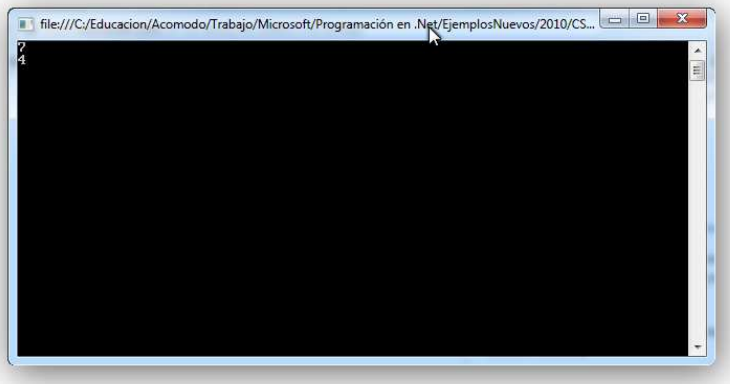
```
namespace notas
{
    class Alumnos
    {
        private String[] alumnos = { "Pedro", "Juan", "Helena", "Mónica",
                    "Sebastián" };
        private int[] notas = { 7, 4, 6, 7, 8 };

        public String ControlDeNotas() {
              for (int i = 0; i < notas.Length; i++) {
                    Console.WriteLine(notas[i]);
                    Debug.Assert (notas[i] > 5 , determinarNota(i));
              }
              return "Control de Notas correcto";
       }

        public String DeterminarNota(int indice)
        {
            int noAprobado = notas[indice];
            String alumno = alumnos[indice];
            String resultado = "El alumno "
                    + alumno + " no aprueba con " + noAprobado + " de nota";
            return resultado;
        }
    }
}

namespace notas
{
```

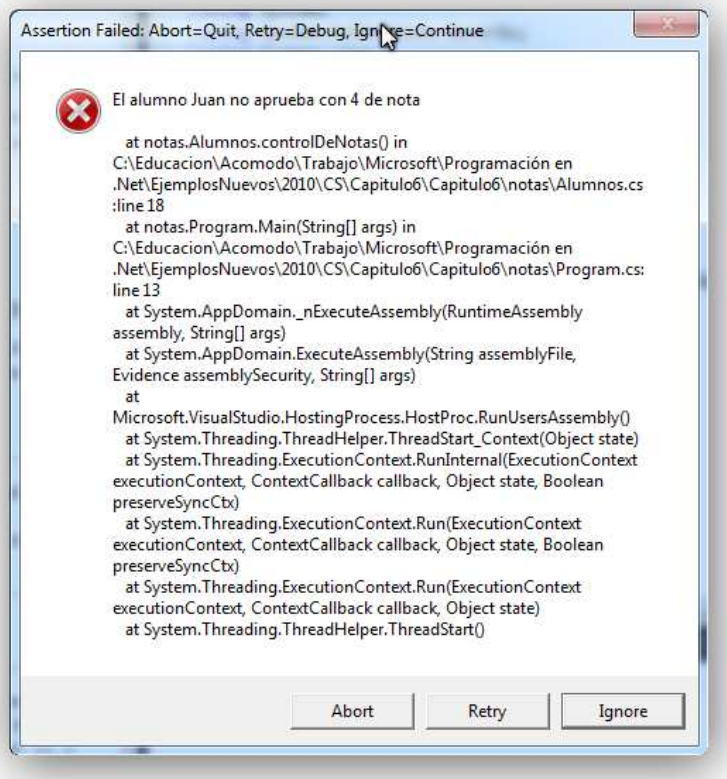
```
    class Program
    {
        static void Main(string[] args)
        {
            Alumnos examenes = new Alumnos();
            Console.WriteLine(examenes.controlDeNotas());
            Console.ReadKey();
        }
    }
}
```

VB

```
Public Class Alumnos
    Private alumnos() As String = {"Pedro", "Juan", "Helena", "Mónica", "Sebastián"}
    Private notas() As Integer = {7, 4, 6, 7, 8}

    Public Function ControlDeNotas() As String
        For i As Integer = 0 To notas.Length - 1
            Console.WriteLine(notas(i))
            Debug.Assert(notas(i) > 5, DeterminarNota(i))
        Next
        Return "Control de Notas correcto"
    End Function

    Public Function DeterminarNota(indice As Integer) As String
        Dim noAprobado As Integer = notas(indice)
        Dim alumno As String = alumnos(indice)
        Dim resultado As String = "El alumno " _
                  + alumno + " no aprueba con " + noAprobado.ToString + " de nota"
        Return resultado
    End Function
End Class

Module Module1

    Sub Main()
        Dim examenes As New Alumnos()
        Console.WriteLine(examenes.ControlDeNotas())
        Console.ReadKey()
    End Sub
End Module
```

El resultado obtenido es:





Notar que la ventana de salida por consola está mostrando la impresión normal realizada por el programa hasta el momento en el cual se produce el error de aserción.

Usos recomendados de las aserciones

Las aserciones pueden aportar datos valiosos sobre las suposiciones y expectativas que maneja el programador. Por este motivo, resultan especialmente útiles cuando otros programadores trabajan con el código para operaciones de mantenimiento.

En general, las aserciones deberían utilizarse para verificar la lógica interna de un solo método o un pequeño grupo de métodos fuertemente vinculados entre sí. No deberían utilizarse para comprobar si el código se utiliza correctamente, sino para verificar que se cumplan sus propias expectativas internas.

Invariantes internas

Las invariantes internas se dan cuando se cree que una situación se va a producir en todos los casos o en ningún caso y se escribe el código de acuerdo con esta convicción.

Ejemplo

C#

```
if (x > 0) {
        // hacer esto
} else {
        // hacer aquello
}
```

VB

```
If x > 0 Then
        'hacer esto
Else
        'hacer aquello
End If
```

Si parte de la premisa es que x puede tener el valor 0, y que nunca puede ser un valor negativo, y resulta que sí adopta un valor negativo, el código hará algo imprevisto. Casi con toda seguridad se comportará de forma incorrecta. Y lo que es peor, dado que el código no se detiene ni comunica el problema, no lo detectará hasta mucho después, cuando el daño causado sea aún mayor. En tales casos, resulta muy difícil determinar el origen del problema.

Para estas situaciones, la adición de una aserción al abrir el bloque `else` o `Else` (C# o VB) ayuda a garantizar la veracidad de la premisa y permite ver rápidamente el error cuando ésta no se cumple o cuando alguien cambia el código y la premisa deja de ser cierta. Una vez introducida la aserción, el código debería ser parecido al siguiente.

Ejemplo

C#

```
if (x > 0) {
        // hacer esto
```

```
} else {
        // hacer aquello a menos que x sea negativo
        Debug.Assert(x == 0);
}
```

VB

```
If x > 0 Then
        'hacer esto
Else
        'hacer aquello a menos que x sea negativo
        Debug.Assert(x = 0)
End If
```

Invariantes del control de flujo

Las invariantes del flujo de control reflejan suposiciones similares a las de las invariantes internas, pero tienen que ver con la dirección que sigue el flujo de ejecución, más que con el valor o las relaciones de las variables.

Por ejemplo, es posible que piense que ha enumerado cada valor posible de la variable de control en una sentencia `switch` o `Select Case` (C# o VB) y que, por tanto, nunca se ejecutará una sentencia `default` o `Case Else` (C# o VB). Esto debería verificarse agregando una sentencia de aserción similar a ésta:

Ejemplo

C#

```
switch (mes)
{
    case 1:
        Console.WriteLine("Enero");
        break;
    case 2:
        Console.WriteLine("Febrero");
        break;
    case 3:
        Console.WriteLine("Marzo");
        break;
    case 4:
        Console.WriteLine("Abril");
        break;
    case 5:
        Console.WriteLine("Mayo");
        break;
    case 6:
        Console.WriteLine("Junio");
        break;
    case 7:
        Console.WriteLine("Julio");
        break;
    case 8:
        Console.WriteLine("Agosto");
        break;
```

```
                case 9:
                    Console.WriteLine("Septiembre");
                    break;
                case 10:
                    Console.WriteLine("Octubre");
                    break;
                case 11:
                    Console.WriteLine("Noviembre");
                    break;
                case 12:
                    Console.WriteLine("Diciembre");
                    break;
                default:
                    Debug.Assert(false, "Ese no es un mes válido!");
                    break;
            }
```

VB

```
        Select Case mes
            Case 1
                Console.WriteLine("Enero")
            Case 2
                Console.WriteLine("Febrero")
            Case 3
                Console.WriteLine("Marzo")
            Case 4
                Console.WriteLine("Abril")
            Case 5
                Console.WriteLine("Mayo")
            Case 6
                Console.WriteLine("Junio")
            Case 7
                Console.WriteLine("Julio")
            Case 8
                Console.WriteLine("Agosto")
            Case 9
                Console.WriteLine("Septiembre")
            Case 10
                Console.WriteLine("Octubre")
            Case 11
                Console.WriteLine("Noviembre")
            Case 12
                Console.WriteLine("Diciembre")
            Case Else
                Debug.Assert(False, "Ese no es un mes válido!")
        End Select
```


Cuando se ejecuta este código, si el valor de la variable mes está fuera del rango 1 a 12 inclusive, el programa se interrumpe anormalmente mostrando un error en una ventana como la descrita anteriormente.

Post condiciones e invariantes de clase

Las post condiciones son suposiciones sobre el valor o las relaciones que presentan las variables al finalizar un método. Un ejemplo sencillo de prueba de post condición sería verificar que después de ejecutar un método para quitar un elemento de una pila (sacar), ésta contiene uno menos de los que tenía antes de llamar al método, a menos que la pila ya estuviese vacía.

Ejemplo

C#

```
namespace postcondicion
{
    class Pila
    {
        private int[] vec = new int[10];
        private int indice = 0;
        private int elementos = 0;

        public void Agregar(int valor)
        {
            if (indice == 9)
                throw new ExcepcionPila("Pila llena");
            vec[indice] = valor;
            elementos = ++indice;
        }

        public int Sacar()
        {
            int resultado = 0;
            if (indice == 0)
                throw new ExcepcionPila("Pila vacía");
            elementos = --indice;
            resultado = vec[indice];
            Debug.Assert(elementos == indice);
            return resultado;
        }
    }
}

namespace postcondicion
{
    class ExcepcionPila : Exception
    {
        public ExcepcionPila(String mensaje)
            : base(mensaje)
        {
        }
    }
}

namespace postcondicion
{
    class Program
    {
```

```
        static void Main(string[] args)
        {
              Pila p = new Pila();
              try {
                    p.Agregar(8);
                    p.Agregar(4);
                    p.Agregar(3);
                    p.Agregar(4);
                    p.Sacar();
                    p.Sacar();
                    p.Sacar();
                    p.Sacar();
                    p.Sacar();
              } catch (ExcepcionPila e) {
             Console.WriteLine(e.StackTrace);
              }
              Console.WriteLine();
        Console.ReadKey();
       }
    }
}
```

VB

```
Public Class Pila
    Private vec(10) As Integer
    Private indice As Integer = 0
    Private elementos As Integer = 0

    Public Sub Agregar(valor As Integer)
        If indice = 9 Then
            Throw New ExcepcionPila("Pila llena")
        End If
        vec(indice) = valor
        elementos = ++indice
    End Sub

    Public Function Sacar() As Integer
        Dim resultado As Integer = 0
        If indice = 0 Then
            Throw New ExcepcionPila("Pila vacía")
        End If
        elementos = --indice
        resultado = vec(indice)
        Debug.Assert(elementos = indice)
        Return resultado
    End Function
End Class

Public Class ExcepcionPila
    Inherits Exception

    Public Sub New(mensaje As String)
        MyBase.New(mensaje)
    End Sub
End Class
```

```
Module Module1

    Sub Main()
        Dim p As New Pila()
        Try
            p.Agregar(8)
            p.Agregar(4)
            p.Agregar(3)
            p.Agregar(4)
            p.Sacar()
            p.Sacar()
            p.Sacar()
            p.Sacar()
            p.Sacar()
        Catch e As ExcepcionPila
            Console.WriteLine(e.StackTrace)
        End Try
        Console.WriteLine()
        Console.ReadKey()
    End Sub
End Module
```

Observar que, si el método “sacar” no generase una excepción en caso de recibir una llamada para actuar sobre una pila vacía, sería más difícil expresar esta aserción, porque el tamaño original “cero” seguiría siendo cero en el resultado final. Observar también que, en la prueba de precondición, es decir, aquella que determina si se ha llamado al método para actuar sobre una pila vacía, no se utiliza ninguna aserción. Esto es porque, si se produce tal situación, no se debe a un error de la lógica local del método, sino a la forma en que se está utilizando el objeto. En general, estas pruebas sobre el comportamiento externo no deberían utilizar aserciones, sino simples excepciones. Esto garantiza que la prueba se realizará siempre y que no se inhabilitará, como puede ocurrir con las aserciones.

Una invariante de clase es aquella que puede comprobarse al final de cada llamada a un método de una clase. En el caso de la clase Pila, una condición invariante es aquella en que el número de elementos nunca es negativo.

Usos inapropiados de las aserciones

No utilizar las aserciones para comprobar los parámetros de un método `public`. Es el método el que debería comprobar cada parámetro y generar la excepción apropiada. La razón por la que no debería usarse aserciones para comprobar parámetros es porque es posible desactivarlas a pesar que el programador quiera seguir realizando la comprobación.

En las pruebas de las aserciones, no utilizar métodos que puedan provocar efectos secundarios, ya que dichas pruebas pueden desactivarse en el momento de la ejecución. Si la aplicación depende de estos efectos secundarios, se comportará de forma diferente cuando las pruebas de aserción se desactiven.