	UTN Facultad Regional Rosario - Ingeniería en Sistemas de Información Asignatura: Tecnologías de desarrollo de software IDE Contenidos Programa Analítico Plan 2008 (ver. 2014)	Unidad 2 Sintaxis y Orientación a Objetos de Lenguajes .Net

Contenidos

Tecnologías de Desarrollo de Software IDE

Unidad: 2

"Sintaxis y Orientación a Objetos de Lenguajes .Net"

(Última Actualización: 25 Agosto 2016)

Índice

Unidad 2: Sintaxis y Orientación a Objetos de Lenguajes .Net

IMPORTANTE: los **contenidos** aquí desarrollados **no son necesariamente completos** en su total profundidad, por lo que **se sugiere consultar otras fuentes**, tales como la **Bibliografía** sugerida por la cátedra en el Programa Analítico y docentes.

Específicamente en este documento **NO se encuentran aún desarrollados todos los temas** planteados para esta Unidad según Programa Analítico vigente.

Capítulo 1: Sintaxis de Lenguajes .Net

Objetivo(s)

- Conocer fundamentos básicos y sintaxis del lenguaje de programación C# .Net y Visual Basic .Net.
- Aplicar conceptos de Programación Orientada a Objetos (POO), aprendidos en otras asignaturas, en .Net.

Temas

1. Tipos, variables y constantes.
2. Declaración, Asignación e Inicialización.
3. Alcance y Visibilidad.

4. Operadores y palabras reservadas.
5. Conversión de Tipos
6. Estructuras de Control: Decisión y Iteración
7. Manejo de Errores y Excepciones.
8. Enumeraciones
9. Uso y creación de Clases y Objetos en .Net
10. Abstracción y Encapsulamiento.
11. Herencia.
12. Polimorfismo.
13. Conceptos de POO aplicados a C# y diferencias con VB.Net.
14. Constructores y destructores.
15. Propiedades y Enumeradores (Enums)
16. Métodos y atributos.
17. Herencia e interfaces.
18. Ocultamiento.
19. Clases abstractas y métodos virtuales.
20. Clases parciales.
21. Modificadores de acceso/alcance, visibilidad y herencia.
22. Diseñador de clases.
23. Espacio de Nombres (Namespaces)

Introducción

La plataforma .Net soporta[1] una gran variedad de lenguajes, dentro de los que se encuentran los desarrollados por el propio Microsoft, tales como C# [2], Visual Basic .NET, C++, J# y recientemente F# [3] (lenguaje funcional) o los desarrollados por otros fabricantes tales como, Cobol, Python, Delphi (Object Pascal), Perl, Fortran, Prolog y PowerBuilder entre otros.

Los lenguajes de programación de la plataforma .NET, utilizan los servicios y características de .NET Framework a través de un conjunto común de clases unificadas, la BCL.

En la mayoría de las situaciones, puede utilizar de manera eficiente todos los lenguajes de programación. Sin embargo, cada lenguaje tiene sus puntos fuertes, y es recomendable comprender las características únicas para cada uno de ellos.

La elección de un lenguaje de programación depende de sus conocimientos del lenguaje y del ámbito de la aplicación que está generando. Las aplicaciones de pequeño tamaño se suelen crear utilizando un único lenguaje, y en otros casos pueden utilizarse más de uno de acuerdo a los requerimientos particulares de la solución.

IMPORTANTE: Se sugiere la búsqueda de bibliografía disponible en Biblioteca UTN Rosario pudiendo buscar títulos en el siguiente buscador con algunas palabras claves sugeridas.

Buscador **Biblioteca UTN**:
www.frro.utn.edu.ar/biblioteca.php
Posibles **palabras claves** para realizar la búsqueda: **.Net**, **C#**, **Visual Basic .Net**, **Objetos**, **Object**.

Sintaxis de Lenguajes .Net

Se plantean diferentes aspectos de la sintaxis de diferentes lenguajes soportados por la plataforma .Net, y en especial se profundizará sobre el

lenguaje C# presentando comparativas con otros lenguajes soportados tales como Visual Basic. Net, tanto a nivel conceptual como en el planteo de ejemplos de código, mostrando diferencias, similitudes y equivalencias entre ellos.

Lenguaje C#

El lenguaje de programación **C#**, pronunciado C Sharp, es en la actualidad junto a Java, uno de los más populares, que permite tanto el desarrollo de aplicaciones para Internet como de aquellas de propósito general.

MUY IMPORTANTE: dado que los contenidos incluidos en este material no abarcan los planteados para la unidad por el momento, se solicita remitirse al resto de material que la cátedra dejo a disposición sumado a otro material al que tengan acceso (para el cual se sugiere consultar con los docentes para recibir sugerencias sobre el mismo).

Excepciones y Errores

Al escribir código es posible que se introduzcan errores, algunos de los cuales pueden ser detectados en tiempo de compilación, tales como los de sintaxis o semántica; mientras que en otros casos pueden pasar inadvertidos por ser errores de lógica o requerir ciertas condiciones para manifestarse.

Es posible que la ejecución de un programa no se realice de la forma esperada y se generen errores al presentarse alguna situación imprevista que impida que la aplicación siga corriendo.

Podríamos considerar situaciones como por ejemplo, el ingreso de una entrada incorrecta de datos, una división por cero, un formato de salida no válido, una falla en la conexión a una fuente de datos o alguna otra situación inesperada.

Los programas deben poder controlar los errores que se producen durante la ejecución de manera uniforme sin que la aplicación deje de funcionar[4]. Para ello el Framework .Net contiene un sistema de **manejo de excepciones**[5] muy potente, que permiten lidiar con situaciones no previstas de modo muy sencillo y efectivo. Este sistema cuenta con un conjunto de clases (incluidas en la BCL), con métodos y propiedades que permiten el manejo de situaciones no esperadas.

Una **excepción** es una alteración del flujo normal de ejecución de una aplicación.

Al generarse una situación no esperada, el **.NET Framework lanza una excepción**, para informar del error (o anomalía), la cual será necesario capturar y poder gestionar.

Anteriormente, el modelo de control de errores de un lenguaje dependía de la forma exclusiva que tenía el lenguaje de detectar los errores y buscarles controladores o del mecanismo de control de errores proporcionado por el sistema operativo.

Ahora el motor en tiempo de ejecución (runtime) [6] implementa el control de excepciones con las características siguientes:

- Controla las excepciones independientemente del lenguaje que las genera o controla.
- No requiere una sintaxis concreta del lenguaje para controlar las excepciones, aunque permite a cada lenguaje definir la propia.
- Permite que las excepciones se inicien traspasando los límites de los procesos e, incluso, los equipos.

Ventajas

Las excepciones ofrecen varias ventajas respecto a otros métodos de notificación de error, tal como aquellos que devuelven códigos de error, entre los que podemos citar:

- Ningún error pasa desapercibido.
- Los valores no válidos no se siguen propagando por el sistema.
- No es necesario comprobar los códigos devueltos.
- Es muy sencillo agregar código de control de excepciones para aumentar la confiabilidad del programa.

Las excepciones predefinidas en la plataforma .Net, son subclases derivadas de la clase base **System.Exception**[7] que abarcan diferentes tipo de excepciones tales como, las aritméticas, de formato, acceso al índice de una posición de un arreglo fuera del limite, división por cero, entre otras. Además se pueden extender el abanico ofrecido por el sistema, creando nuevas derivadas de alguna de ellas por medio de herencia.

En esta sección veremos:
Cómo: Utilizar el bloque Try/Catch para detectar excepciones
Describe cómo usar el bloque try/catch para controlar excepciones.
Cómo: Utilizar excepciones específicas en un bloque Catch
Describe cómo detectar excepciones específicas.
Cómo: Iniciar una excepción explícitamente
Describe cómo producir excepciones y cómo detectarlas y, a continuación, volver a producirlas otra vez.
Cómo: Crear excepciones definidas por el usuario
Describe cómo crear clases de excepciones personalizadas.
Utilizar controladores filtrados por el usuario
Describe cómo configurar excepciones filtradas.
Cómo: Utilizar un bloque Finally
Explica cómo utilizar la instrucción Finally en un bloque de excepciones.
Cómo: Crear clases personalizadas de Excepciones
Describe cómo crear excepciones personalizadas.

Secciones relacionadas
Información general sobre excepciones
Proporciona una descripción general de las excepciones de Common Language Runtime.
Clase Exception y propiedades
Describe los elementos de un objeto de excepción.

Cómo: Detectar y manejar excepciones utilizando el bloque Try/Catch

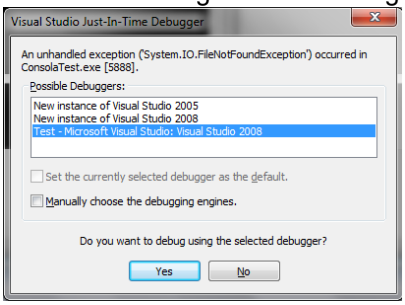
Para capturar una excepción se deben realizar minimamente dos cuestiones:

1. Colocar el código que pueda lanzar una excepción dentro de un **bloque Try**. En C# usando la palabra reservada **try**.
2. Colocar el código que maneja (controla) la excepción, cuando se lance, en un **bloque Catch**. En C# usando la palabra reservada **catch** seguida por un tipo de excepción y la acción que se debe tomar.

A modo de ejemplo supongamos que tenemos una aplicación de consola que tiene que recuperar datos de un archivo de texto, en el código aparece un método Main contiene un una instrucción StreamReader que abre un archivo de datos denominado data.txt y escribe una cadena del archivo.

```
// (C#) Manejo de excepción básico
using System;
using System.IO;
using System.Security.Permissions;
// Security permission request.
[assembly:FileIOPermissionAttribute(SecurityAction.RequestMinimum,
All = @"c:\data.txt")]
public class ProcessFile {
  public static void Main() {
        StreamReader sr = File.OpenText("data.txt");
        Console.WriteLine("The first line of this file is {0}",
sr.ReadLine());
  }
}
```

Al ejecutarse el archivo y en caso que el archivo no este disponible, vamos a obtener un error en la aplicación que estando en desarrollo aparecerá una ventana de diálogo similar a la siguiente.



Ahora vamos a agregar un bloque try/catch para detectar una posible excepción. Colaremos las instrucciones para abrir el archivo y escribir una cadena del archivo dentro de un bloque **try**. Seguido agregaremos un bloque **catch** que detecta cualquier excepción que resulte del bloque Try quedando el código como se muestra a continuación.

```
// (C#) Manejo de excepción básico
using System;
using System.IO;
using System.Security.Permissions;
// Security permission request.
[assembly:FileIOPermissionAttribute(SecurityAction.RequestMinimum,
All = @"c:\data.txt")]
public class ProcessFile {
  public static void Main() {
      try {
        StreamReader sr = File.OpenText("data.txt");
```

```
        Console.WriteLine("The first line of this file is {0}",
sr.ReadLine());
     }
     catch(Exception e) {
        Console.WriteLine("An error occurred: '{0}'", e);
     }
  }
}
```

De esta forma obtendremos la siguiente salida donde nos muestra el mensaje que agregamos más el detalle de la excepción contenida en el objeto **e**.

```
An error occurred:
'System.IO.FileNotFoundException: Could not find
file 'c:\data.txt'.
File name: 'c:\data.txt'
          at   System.IO.__Error.WinIOError(Int32
errorCode, String maybeFullPath)
      at  System.IO.FileStream.Init(String  path,
FileMode mode, FileAccess access, I
nt32 rights, Boolean useRights, FileShare share,
Int32 bufferSize, FileOptions o
ptions,  SECURITY_ATTRIBUTES  secAttrs,  String
msgPath, Boolean bFromProxy)
     at  System.IO.FileStream..ctor(String  path,
FileMode mode, FileAccess access,
FileShare  share,  Int32  bufferSize,  FileOptions
options, String msgPath, Boolean
bFromProxy)
     at  System.IO.FileStream..ctor(String  path,
FileMode mode, FileAccess access,
FileShare  share,  Int32  bufferSize,  FileOptions
options)
    at System.IO.StreamReader..ctor(String path,
Encoding encoding, Boolean detec
tEncodingFromByteOrderMarks, Int32 bufferSize)
    at System.IO.StreamReader..ctor(String path,
Boolean detectEncodingFromByteOr
derMarks)
   at System.IO.File.OpenText(String path)
               at     ProcessFile.Main()     in
C:\Temp\Net\Test\Test\ConsolaTest\Program.cs:line
36'
```

Este ejemplo ilustra una instrucción Catch básica que detectará cualquier excepción. Por lo general, se recomienda, cuando se esté programando, detectar un tipo de excepción específico en lugar de utilizar la instrucción Catch básica. Para obtener información sobre la detección de excepciones específicas, vea Utilizar excepciones específicas en un bloque Catch.

Cómo: Utilizar excepciones específicas en un bloque Catch

Cuando se produce una excepción, asciende por la pila y se da a cada uno de los bloques Catch la oportunidad de controlarla. El orden de las instrucciones Catch es importante. Ponga los bloques Catch dirigidos a excepciones específicas antes de un bloque Catch de excepción general, pues de lo contrario el compilador puede emitir un error. El bloque Catch adecuado se determina haciendo coincidir el tipo de excepción con el nombre de la excepción especificada en el bloque Catch. Si no hay un bloque Catch específico, un bloque Catch general, si lo hay, detecta la excepción.

En el ejemplo de código siguiente se usa un bloque try/catch para detectar una InvalidCastException. En el ejemplo se crea una clase denominada Employee con una única propiedad, el nivel de

empleado (Emlevel). Un método, PromoteEmployee, toma un objeto e incrementa el nivel del empleado. Se produce una InvalidCastException cuando se pasa una instancia de DateTime al método PromoteEmployee.

```
// (C#) Manejo de Excepciones Especificas
using System;
public class Employee
{
  // Create employee level property.
  int emlevel;
   public int Emlevel
  {
     get { return(emlevel); }
     set { emlevel = value; }
  }
}

public class Ex13
{
  public static void
PromoteEmployee(Object emp)
  {
     // Cast object to Employee.
     Employee = (Employee) emp;
     // Increment employee level.
     e.Emlevel = e.Emlevel + 1;
  }

  public static void Main()
  {
     try
     {
        Object o = new Employee();
        DateTime newyears = new
DateTime(2001, 1, 1);
        //Promote the new employee.
        PromoteEmployee(o);
        // Promote DateTime;
         // results in InvalidCastException
as newyears is not an employee instance.
        PromoteEmployee(newyears);
     }
     catch (InvalidCastException e)
     {
        Console.WriteLine("Error passing
data to PromoteEmployee method. " + e);
     }
  }
}
```

El Common Language Runtime detecta las excepciones que no detecta un bloque Catch. Según cómo esté configurado el motor en tiempo de ejecución, aparece un cuadro de diálogo de depuración o el programa se deja de ejecutar y aparece un cuadro de diálogo con información de la excepción. Para obtener información sobre la depuración, vea Depurar y generar perfiles de aplicaciones.

Cómo: Iniciar excepciones explícitamente

Se puede producir una excepción de manera explícita utilizando la instrucción **Throw**. También se puede volver a producir una excepción detectada utilizando la instrucción Throw. Es muy recomendable, cuando se está escribiendo código, agregar información a una excepción que se vuelve

a producir para proporcionar más información cuando se vaya a depurar.

En el ejemplo de código siguiente se usa un bloque try/catch para detectar una posible FileNotFoundException. Después del bloque Try hay un bloque Catch que detecta FileNotFoundException y escribe un mensaje en la consola si no se encuentra el archivo de datos. La siguiente instrucción es una instrucción Throw que produce una nueva FileNotFoundException y agrega información sobre el texto a la excepción.

```
// (C#) Manejo de excepción básico
using System;
using System.IO;
public class ProcessFile
{
  public static void Main()
   {
      FileStream fs = null;
      try
      {
         // Opens a text tile.
         fs = new FileStream ("data.txt",
FileMode.Open);
        StreamReader sr = new
StreamReader(fs);
        string line;
        // A value is read from the file and
output to the console.
        line = sr.ReadLine();
         Console.WriteLine(line);
     }
     catch(FileNotFoundException e)
     {
        Console.WriteLine("[Data File
Missing] {0}", e);
        throw new FileNotFoundException("
[data.txt not in c:\\dev directory]",e);
     }
      finally
     {
        fs.Close();
     }
  }
}
```

Notar que se incluye el bloque **Finally** el cual va a ser ejecutado siempre independientemente de lo que pase en los bloques Try y Catch anteriores y en ese caso particular se agrega una sentencia para cerrar el archivo abierto dentro del bloque Try que permite liberar el recurso, una buena práctica a tomar en cuenta para cuando se utilizan fuentes externas de datos.

Cómo: Utilizar controladores de excepciones filtrados por el usuario

Actualmente, Visual Basic admite las excepciones filtradas por el usuario. Los controladores de excepciones filtrados por el usuario detectan y controlan las excepciones basándose en requisitos que se definen para la excepción. Estos controladores utilizan la instrucción Catch con la palabra clave **When**.

Esta técnica resulta útil cuando un objeto de excepción concreto corresponde a varios errores. En este caso, normalmente, el objeto tiene una

propiedad que contiene el código de error específico asociado al error. La propiedad del código de error se puede usar en la expresión para seleccionar sólo el error concreto que se desea controlar en esa cláusula Catch.

Ver un ejemplo de Visual Basic que ilustra la instrucción Catch/When en el sitio MSDN en Utilizar controladores de excepciones filtrados por el usuario.

Cómo: Utilizar bloques Finally

Cuando se produce una excepción, se detiene la ejecución y da el control al controlador de excepciones más cercano. A menudo, esto significa que no se ejecutan líneas de código que se espera que se llamen siempre. Siempre se debe ejecutar cierta limpieza de recursos, como el cierre de un archivo, incluso si se produce una excepción. Para lograr esto, se puede usar un bloque Finally. Los bloques Finally se ejecutan siempre, independientemente de si se produce una excepción o no.

En el ejemplo de código siguiente se usa un bloque try/catch para detectar una ArgumentOutOfRangeException. El método Main crea dos matrices e intenta copiar una en la otra. La acción genera una ArgumentOutOfRangeException y el error se escribe en la consola. El bloque Finally se ejecuta, sea cual sea el resultado de la acción de copia.

```
// (C#) Ejemplo: crear propia
especialización de clase Excepción
using System;
class ArgumentOutOfRangeExample
{
  static public void Main()
     {
     int[] array1={0,0};
     int[] array2={0,0};
        try
        {
        Array.Copy(array1,array2,-1);
        }
        catch (ArgumentOutOfRangeException
e)
        {
        Console.WriteLine("Error: {0}",e);
        }
        finally
        {
        Console.WriteLine("This statement is
always executed.");
        }
     }
}
```

Cómo: Crear clases personalizadas de Excepciones

Si desea que los usuarios puedan distinguir programáticamente, ciertas condiciones de error de otras, puede crear sus propias excepciones definidas por el usuario (user-defined).

El .NET Framework proporciona una jerarquía de clases de excepción que, en última instancia, derivan de la clase base **Exception**. Cada una de estas clases define una excepción específica, por lo que en muchos casos sólo hay que detectar la excepción que se ajuste al error que deseamos manejar, aunque también se pueden crear clases de excepción personalizadas derivadas de la clase Exception o de alguna de sus derivadas.

Cuando se creen excepciones personalizadas, es recomendable finalizar el nombre de la clase de la excepción definida por el usuario con la palabra "*Excepción*". También se recomienda implementar los tres constructores comunes recomendados, como se muestra en el ejemplo siguiente, donde se deriva una nueva clase de excepción, *EmployeeListNotFoundException*, como especialización de la clase Exception. Se definen tres constructores en la clase, cada uno con parámetros diferentes.

```
// (C#) Ejemplo: crear propia especialización
de clase Excepción
using System;
public class EmployeeListNotFoundException:
Exception
{
   public EmployeeListNotFoundException()
  {   }
  public
EmployeeListNotFoundException(string message)
  : base(message)
  {   }
  public
EmployeeListNotFoundException(string message,
Exception inner)
  : base(message, inner)
  {   }
}
```

NOTA: se puede ampliar con mayor este tema desde los recursos disponibles o indicados por la cátedra.

***Fuentes donde se trata el tema**:*
CEBALLOS SIERRA, Francisco. "Microsoft C#: Lenguaje y Aplicaciones." Capítulo 6 “Clases de uso común”, pag. 106 “Excepciones”. [Nivel: básico/inicial]

Modificadores de Acceso

Los modificadores de acceso son palabras clave que especifican la accesibilidad declarada de un miembro o un tipo. Esta sección presenta los cuatro modificadores de acceso:

- public
- protected
- internal
- private

Mediante los modificadores de acceso se pueden especificar los siguientes cinco niveles de accesibilidad:

public: acceso no restringido.
protected: acceso limitado a la clase contenedora o a los tipos derivados de esta clase.
Internal: acceso limitado al ensamblado actual.

protected internal: acceso limitado al ensamblado actual o los tipos derivados de la clase contenedora.
private: acceso limitado al tipo contenedor.

Ampliar en MSDN "Modificadores de Acceso" y otras fuentes disponibles.

Namespaces (Espacios de Nombres)

La palabra clave ***namespace*** se utiliza para declarar un ámbito. Este ámbito permite organizar el código y proporciona una forma de crear tipos globalmente únicos.

Dentro de un espacio de nombres, se pueden declarar uno o varios de los siguientes tipos:

- otro espacio de nombres
- clase
- interfaz
- struct
- enum
- delegado

Los espacios de nombres disponen implícitamente de un acceso público que no puede modificarse. Para obtener una descripción de los modificadores de acceso que se pueden asignar a los elementos de un espacio de nombres, vea Modificadores de acceso (Referencia de C#).

Un espacio de nombres se puede definir en dos o más declaraciones. Por ejemplo, en el siguiente ejemplo se definen dos clases como parte del espacio de nombres MyCompany:

```
// (C#) Definición de Espacios de Nombres en
más de una declaración

namespace MyCompany.ProjectName
{
   class MyClass
    {    }
}

namespace MyCompany.ProjectName
{
   class MyOtherClass
    {    }
}
```

Ampliar en MSDN "Espacios de Nombres" y otras fuentes disponibles.

Palabra reservada using

La palabra clave using tiene dos usos principales:

- Como **directiva**, cuando se utiliza para crear un alias para un espacio de nombres o para importar[8] tipos definidos en otros espacios de nombres. Vea Directiva using.
- Como **instrucción**, cuando define un ámbito al final del cual el objeto se destruye. Vea Instrucción using.

Ampliar en MSDN "using" y otras fuentes disponibles.

Enumeración (enum)

Nota IMPORTANTE: este tema no está desarrollado de forma completa por favor consultar otras fuentes.

Una *enumeración* es un tipo por valor que permite definir un conjunto de constantes con nombre, (denominado lista de enumeradores) que pueden asignarse a una variable. Para declararla se utiliza la palabra clave ***enum***.

Ejemplo: suponga que tiene que definir una variable cuyo valor representará un día de la semana. Sólo tiene siete valores significativos que almacenará dicha variable.

```
enum Days {Sat, Sun, Mon, Tue, Wed, Thu,
Fri};
enum Dias {Sab, Dom, Lun, Mar, Mie, Jue,
Vie};
```

Nota: relacionado al ejemplo planteado considere que el Framework ya contiene una enumeración similar predefinida llamada *DaysOfWeeks*

Normalmente suele ser recomendable definir una enumeración directamente dentro de un espacio de nombres para que todas las clases de dicho espacio puedan tener acceso a ésta con la misma facilidad. Sin embargo, una enumeración también puede anidarse dentro de una clase (class) o estructura (struct).

Nota Importante: Un enumerador no puede contener espacio en blanco en su nombre.

De forma predeterminada, el primer enumerador asume el valor 0 y el valor de cada enumerador sucesivo se incrementa en 1. En el ejemplo anterior Sat asume el valor 0, Sun es 1, Mon es 2 y así sucesivamente.

Los enumeradores pueden usar inicializadores para invalidar los valores predeterminados, como se muestra en el ejemplo siguiente.

```
enum Days {Sat = 1, Sun, Mon, Tue, Wed, Thu,
Fri};
enum Dias {Sab = 1, Dom, Lun, Mar, Mie, Jue,
Vie};
```

Tipos Subyacentes de los enum

Cada tipo de enumeración tiene un tipo subyacente, que puede ser cualquier tipo entero excepto char. El **tipo predeterminado subyacente** de los elementos de la enumeración es **int**. Para declarar una enumeración de otro tipo entero, como byte, use un carácter de dos puntos después del identificador y escriba a continuación el tipo, como se muestra en el ejemplo siguiente.

```
enum Months: byte {Jan = 1, Feb, Mar, Apr,
May, Jun, Jul, Aug, Sep, Oct, Nov, Dec};
```

Los **tipos admitidos** para una enumeración son byte, sbyte, short, ushort, int, uint, long o ulong.

El tipo subyacente especifíca el almacenamiento asignado para cada enumerador. No obstante, se

necesita una conversión explícita para convertir un tipo enum a un tipo entero.

Por ejemplo, la siguiente instrucción asigna el enumerador Sun a una variable de tipo int utilizando una conversión de tipos para convertir de enum a int.

```
int x = (int)Days.Sun;
```

Puede comprobar los valores numéricos subyacentes y convertir al tipo subyacente, como se muestra en el ejemplo siguiente.

```
Days today = Days.Monday;
int dayNumber =(int)today;
Console.WriteLine("{0} is day number #{1}.",
today, dayNumber);

Months thisMonth = Months.Dec;
byte monthNumber = (byte)thisMonth;
Console.WriteLine("{0} is month number #
{1}.", thisMonth, monthNumber);

// Output:
// Monday is day number #1.
// Dec is month number #11.
```

Cuando se aplica *System.FlagsAttribute* a una enumeración que contiene algunos elementos que se pueden combinar con una operación OR bit a bit, se observará que el atributo afecta al comportamiento de enum cuando se utiliza con algunas herramientas. Se pueden observar estos cambios al utilizar herramientas tales como los métodos de la clase Console y el Evaluador de expresiones.

```
// Add the attribute Flags or
FlagsAttribute.
[Flags]
public enum CarOptions
{
   // The flag for SunRoof is 0001.
    SunRoof = 0x01,
   // The flag for Spoiler is 0010.
    Spoiler = 0x02,
   // The flag for FogLights is 0100.
    FogLights = 0x04,
   // The flag for TintedWindows is 1000.
    TintedWindows = 0x08,
}

class FlagTest
{
    static void Main()
   {
         // The bitwise OR of 0001 and 0100
is 0101.
       CarOptions options =
CarOptions.SunRoof | CarOptions.FogLights;
       // Because the Flags attribute is
specified, Console.WriteLine displays
       // the name of each enum element that
corresponds to a flag that has
       // the value 1 in variable options.
       Console.WriteLine(options);
       // The integer value of 0101 is 5.
```

```
        Console.WriteLine((int)options);
    }
}
// Output:
// SunRoof, FogLights
// 5
```

Otro Ejemplo que utiliza Flags mostrando valores combinados de un enum

```
[Flags]
public enum DaysOfWeek
{
   Sun = 1, Mon = 2, Tue = 4,
   Wed = 8, Thu = 16, Fri = 32,
   Sat = 64
}
var alarmGoesOffOn = DaysOfWeek.Mon | DaysOfWeek.Wed | DaysOfWeek.Fri;
Console.WriteLine("Alarm goes off on: " + alarmGoesOffOn);
// Imprime: Alarm goes off on: Mon, Wed, Fri
```

Ver más sobre Enums en http://thatcsharpguy.com/post/c-sharp-enums/
Ver más sobre FlagsAttribute (Clase)

La Clase Enum

Todas las enumeraciones son instancias del tipo System.Enum. No puede derivar clases nuevas de System.Enum, pero puede utilizar sus métodos para detectar información relacionada y manipular los valores de una instancia de enumeración.

La clase Enum proporciona métodos para comparar instancias de esta, convertir el valor de una instancia a su representación de cadena de caracteres, convertir la representación de cadena a un valor numérico, a una instancia de esta clase, y a crear una instancia de una determinada enumeración y valor especificado.

Ver más sobre la Clase Enum (Clase)

Buenas Prácticas en el uso de Enumeraciones
Se recomienda utilizar las siguientes buenas prácticas al definir tipos de enumeración:

- Si no ha definido un miembro de la enumeración cuyo valor es 0, considere crear una constante Ninguna. De forma predeterminada, se inicializa a cero por el Common Language Runtime. En consecuencia, si no se define una constante cuyo valor es cero, la enumeración contiene un valor no válido cuando se crea.
- Si hay un caso por defecto obvio que su aplicación tiene para representar, considere el uso de una constante enumerada cuyo valor sea cero para representarla. Si no hay ningún caso por defecto, considerar el uso de una constante enumerado cuyo valor es cero para especificar el caso de que no está representado por cualquiera de las otras constantes enumeradas.
- No especifique constantes enumeradas que están reservados para uso futuro.
- Cuando se define un método o propiedad que tiene una constante enumerado como un valor, considere validar el valor. La razón

es que se puede convertir un valor numérico para el tipo de enumeración, incluso si ese valor numérico no está definido en la enumeración.

Algunas Ventajas

Las ventajas de utilizar una enumeración en lugar de un tipo numérico:

- Se especifica claramente en el código qué valores son válidos para la variable.
- En Visual Studio, IntelliSense muestra los valores definidos.

Para más información puede visitar los siguientes enlaces:

- La clase Enum
- enum (Referencia de C#)
- Enum (Clase)

Capítulo 2: Orientación a Objetos de Lenguajes .Net

Objetivo(s)

- Introducir los elementos básicos de Programación Orientada a Objetos empleando como herramienta de desarrollo la plataforma .NET y el lenguaje C#.
- Analizar los beneficios de la herencia, del polimorfismo y el uso básico de colecciones de objetos.
- Poder modelar un problema básico utilizando Programación Orientada a Objetos, detectando clases, responsabilidades y jerarquías, y luego implementarlo utilizando C# en la plataforma .NET.

NOTA IMPORTANTE: **este tema NO ha sido desarrollado aún en el presente documento**. Remitirse momentáneamente a materiales brindados por la cátedra sumados a los recursos digitales que se detallan debajo y aquellos adicionales a los que pudieran tener acceso.

Fuente MSDN
- **Programación orientada a objetos (C# y Visual Basic)**
Ingles: http://msdn.microsoft.com/en-us/library/dd460654.aspx
Traduccion automatica al espanol: http://msdn.microsoft.com/es-ar/library/dd460654.aspx

- **Objetos, Clases y Estructuras**
http://msdn.microsoft.com/es-es/library/ms173109(v=vs.80)

- **Polimorfismo**
http://msdn.microsoft.com/es-es/library/ms173152(v=vs.80).aspx

Fuente Blog El Guille
- **Programación Orientada a Objetos en .NET**
http://www.elguille.info/NET/dotnet/POO_VB_NET_tp6.htm

- **Programación Orientada a Objetos en .NET (2)**
http://www.elguille.info/NET/dotnet/POO_NET_tp7.htm

- **Conceptos y principios orientado a objetos**
http://www.elguille.info/colabora/NET2005/Percynet_Conceptosyprincipiosorientadoaobjetosht
- **Cuando usar override y new**
https://msdn.microsoft.com/es-ar/library/ms173153(v=VS.80).aspx

Ejemplos de Código

Puede ver algunos ejemplos de código de MSDN aquí: Ejemplos de Visual C#

Miembros de clase

Los miembros de una clase se componen de los miembros declarados en la clase y los miembros que ésta hereda de su clase base directa.

Categorías
Los miembros de una clase se dividen en las siguientes categorías:

- **Constantes**: representan valores constantes asociados a la clase (Sección 10.3).
- **Campos**: son las variables de la clase (Sección 10.4).
- **Métodos**: que implementan las acciones (comportamiento) que puede realizar la clase (Sección 10.5).
- **Propiedades**: definen elementos con nombre asociados a la lectura y escritura de dichos elementos (Sección 10.6).
- **Eventos**: definen las notificaciones que puede generar la clase (Sección 10.7).
- **Indizadores**: permiten indizar las instancias de la clase de la misma manera (sintácticamente) que las matrices (Sección 10.8).
- **Operadores**: definen los operadores de expresión que se pueden aplicar a las instancias de la clase (Sección 10.9).
- **Constructores de instancia**: implementan las acciones necesarias para inicializar las instancias de la clase (Sección 10.10)
- **Destructores**: implementan las acciones a realizar antes de que las instancias de la clase sean descartadas de forma permanente (Sección 10.12).
- **Constructores estáticos**: implementan las acciones necesarias para inicializar la propia clase (Sección 10.11).
- **Tipos**: representan los tipos locales de la clase (Sección 9.5).

Resumen de la Unidad / Capítulo

IMPORTANTE: los **contenidos** desarrollados en esta sección **no son completos**, sino que son un mini resumen de conceptos generales de los temas de mayor importancia del capítulo a modo de un repaso general de los mismos una vez completada la lectura del mismo.

Los programas deben poder controlar los **errores** que se producen durante la ejecución de manera uniforme. EL Common Language Runtime (CLR) ofrece una gran ayuda para diseñar software con tolerancia a errores mediante un modelo que informa a los programas de los errores de manera uniforme iniciando (lanzando) **excepciones**.

Una **excepción** es una alteración del flujo normal de ejecución de una aplicación.

.Net incorpora mecanismos para el tratamiento de las situaciones anómalas, denominadas **excepciones**, que pueden producirse durante la ejecución de un programa. Estas excepciones se controlan mediante código situado fuera del flujo normal de control mediante bloques de control que en el lenguaje C# utiliza las palabras reservadas **try-catch-finally**.

El **sistema de manejo de excepciones** ofrecido por la plataforma tiene como **finalidad** principal permitir el **tratamiento de errores** y otras situaciones no esperadas en la ejecución de las aplicaciones.

La **BCL** contiene una jerarquía de clases de excepciones que derivan de la clase base *System.Exception* y admite ser extendida creando las propias como especialización de la clase base o algunas de sus clases más específicas derivadas de ella. Además se encuentran las clases *System.SystemException* y *System.ApplicationException*.

El orden de las instrucciones Catch es importante. Ubique los bloques Catch dirigidos a excepciones específicas antes de un bloque Catch de una excepción general, ya que serán evaluadas en forma secuencial desde la primera en adelante.

Enumeraciones: tipo por valor que permite definir un conjunto de constantes relacionadas bajo un nombre utilizando la palabra reservada enum.

De forma predeterminada, el primer enumerador asume el valor 0 y el valor de cada enumerador sucesivo se incrementa automáticamente en 1. Si no se especifican valores para los elementos en la lista de enumeradores, los valores se incrementan automáticamente en 1.

Bibliografía[9]

IMPORTANTE: Se sugiere buscar bibliografía adicional a la citada aquí, en el buscador disponible en Biblioteca UTN Rosario.

BALENA, Francesco “Programación avanzada con Microsoft Visual Basic.Net.”, Madrid, McGraw-Hill, 2003.
ISBN: 8448137159
Capítulo 1 “Primeros pasos con visual basic.Net”, 2 “Modulos y variables”, 3 “Control de flujo y manejo de errores” y 9 “Matrices, listas y colecciones”
Ejemplares Disponibles: en biblioteca

BOOCH, Grady; RUMBAUGH, James y JACOBSON, Ivar. “El Lenguaje Unificado de Modelado: Manual de referencia.” , Pearson, 2007
ISBN: 9788478290765
Ejemplares Disponibles: en Biblioteca

CEBALLOS SIERRA, Francisco. "El lenguaje de programación Visual Basic.Net.", México, Alfaomega, 2002.
ISBN: 9701508297
Capítulos:
2. Fundamentos de Visual Basic.NET .
3. Programación orientada a objetos.
4. Elementos del lenguaje.
5. Estructura de un programa.
6. Clases de uso común.
7. Sentencias de control.
8. Matrices.
9. Más sobre procedimientos.
Ejemplares Disponibles: en Biblioteca

CEBALLOS SIERRA, Francisco. "Microsoft C#: Curso de programación.", México, Alfaomega, 2007.
ISBN: 9701512227
Capítulos:
2. Fundamentos de C#.
3. Programación orientada a objetos.
4. Elementos del lenguaje.
5. Estructura de un programa.
6. Clases de uso común.
7. Sentencias de control.
8. Matrices.
Ejemplares Disponibles: en Biblioteca

CEBALLOS SIERRA, Francisco. "Microsoft C#: Lenguaje y Aplicaciones.", México, Alfaomega, 2008.
ISBN: 9789701513712
Capítulos:
2. Fundamentos e C#.
3. Programación orientada a objetos.
4. Elementos del lenguaje.
5. Estructura de un programa.
6. Clases de uso común.
7. Sentencias de control.
8. Matrices.
9. Más sobre métodos.
Ejemplares Disponibles: en Biblioteca

DEITEL, Harvey y DEITEL, Paul. "Como programar en C#.", Mexico, Pearson Educacion, 2007.
ISBN: 9789702610564
Capítulo 3 "Introducción a las aplicaciones de C#", 4 "Introducción a las clases y los objetos", 5 "Instrucciones de control: parte 1", 6 "Instrucciones de control: parte 2", 7 "Métodos: un análisis más detallado", 8 "Arreglos", 9 "Clases y objetos: un análisis más detallado", 10 "Programación orientada a objetos: herencia", 11 "Polimorfismo, interfaces y sobrecarga de operadores", 12 "Manejo

de excepciones", 16 "Cadenas, caracteres y expresiones regulares", 24 "Estructuras de datos.", 25 "Genéricos", 26 "Colecciones" y Apéndice A "Tabla de precedencia de los operadores"
Ejemplares Disponibles: en Biblioteca

FOXALL, James "Visual C# 2005.", Madrid, Anaya, 2006.
ISBN: 9788441521216
Capítulos
2. Explorar visual C# 2005.
3. Objetos y colecciones.
10. Métodos: Creación y llamadas.
11. Utilizar constantes, tipos de datos, variables y matrices.
12. Cálculos aritméticos, manipulación de cadenas y ajustes de fecha y hora.
13. Tomas decisiones en visual C#.
14. Bucles para obtener eficiencia.
15. Depuración del código.
16. Diseñar objetos con clases.
Disponibilidad: en biblioteca

FIRESMITH, Donald. "Object-oriented requirements analysis and design: A software approach.", New York, Wiley & Sons, 1992.
ISBN: 047157807x
Ejemplares Disponibles: en Biblioteca

HILYARD, Jayy TEILHET, Stephen "C# 3.0: Guía de referencia.", Madrid, Anaya Multimedia, 2009.
ISBN: 9788441524910
Capítulo 2 "Cadenas y caracteres.", 3 "Clases y estructuras.", 4 "Genéricos.", 5 "Colecciones.", 7 "Gestión de excepciones.", 10 "Expresiones regulares.", 11 "Estructuras de datos y algoritmos." y 20 "Números y enumeraciones."
Ejemplares Disponibles: en Biblioteca

RAMIREZ, Felipe. "Programación: algoritmos y su implementación en Vb.Net, C#, Java y C++.", México, Alfaomega, 2007.
ISBN: 9789701512807
Capítulos:
10. Implementación de algoritmos en Visual Basic.Net.
11. Implementación de algoritmos en C#.
14. Fundamentos de la programación orientada a objetos.
Ejemplares Disponibles: en Biblioteca

SHARP, John. "Visual C# 2008: paso a paso." Madrid, Anaya, 2008.
ISBN: 9788441524491
Capítulos:
1 "Bienvenido a C#"
Disponibilidad: en biblioteca

SPENCER, Ken ; EBARDHARD, Tom y ALEXANDER, John. "OOP : Buildind reusable componentes with Microsoft Visual Basic.Net", Washington, Microsoft Press, 2003.
ISBN: 0735613796
Ejemplares Disponibles: en Biblioteca

Fuentes Digitales

msdn
+ Microsoft Developer Network (MSDN) On Line: http://msdn.microsoft.com
Referencia C#: http://msdn.microsoft.com/en-us/library/618ayhy6(v=vs.80).aspx
Manejo y Lanzamiento de Excepciones:

http://msdn.microsoft.com/en-us/library/5b2yeyab(v=vs.80).aspx
Clase Exception:
http://msdn.microsoft.com/en-us/library/system.exception.aspx
Programación Orientada a Objetos (C# y VB.Net)
https://msdn.microsoft.com/es-es/library/dd460654.aspx

+ Blog El Guille: http://www.elguille.info

+ Wikipedia (conceptos genéricos y específicos)
Programación Orientada a Objetos:
http://en.wikipedia.org/wiki/Object-oriented_programming
http://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos
Lenguajes Orientados a Objetos:
http://es.wikipedia.org/wiki/Lenguaje_orientado_a_objetos

Anexo A: Sentencias, expresiones y operadores

Este anexo intentó poner de manifiesto el significado entre los términos "sentencia", "expresión" y operadores.

Sentencias (Statements)

Todas las "líneas de código" con acciones que un programa realiza son sentencias, desde la declaración de una variable hasta el control de flujo del programa con un if o un case. Las sentencias de este tipo se conocen como sentencias de una línea.

Algunos ejemplos de sentencias de una sola línea son:

Tipo	Descripción
Declaraciones	Crea una nueva variable o constante en el programa como, por ejemplo string t; o const ix = 0; usualmente se encadena con una sentencia de asignación.
Expresiones	Sentencias que calculan un valor.
Salto	Son otro elemento para controlar el flujo del programa, sirven para mover el punto de ejecución del programa, por ejemplo goto, break y return
Vacío	Sí, existe una sentencia vacía y es ;

También existen sentencias más complejas, que no son de *una sola línea* en las cuales se suelen agrupar más sentencias usando { y } como delimitadores, a esta acción se le conoce como *anidamiento*.

Expresiones

Las expresiones son un tipo especial de sentencias que se componen de uno o más operandos en conjunto con cero o más operadores, la diferencia con una sentencia común es que una expresión siempre puede ser evaluada a un solo valor. En la mayoría de los casos una expresión es una sentencia que siempre devuelve un valor. Ya sea que la expresión devuelva una referencia a un objeto o un valor.

Algunos ejemplos de expresión son los siguientes:

Operador new

```
new Object();
```

Cuando usamos el operador new para instanciar un objeto, el constructor de la clase (en este caso Object) es el operando y el resultado de esta expresión es una referencia a una instancia de la clase (en este caso, de Object).

Incremento y decremento

```
i++;
--i;
```

Los operadores ++ y -- son denominados de incremento y decremento, el resultado de la expresión varía en función de la posición en la que estén colocados respecto al operando (que en nuestro caso es i).

Antes del operando: es el valor del operando antes de ser incrementado/decrementado
Después del operando: es el valor del operando después de ser incrementado.
Llamada (call)

```
ConvertInt(i);

i.ToString();
```

Las llamadas a métodos también son ejemplos de expresiones, estas expresiones no requieren de un operador como las dos anteriores, basta con escribir el nombre del método y agregarle sus argumentos en caso de ser necesarios.

Suponiendo que tenemos la siguiente firma de método decimal ConvertInt(int i), la primera expresión en el código anterior tendrá como resultado un decimal, mientras que la segunda tendrá como resultado una cadena.

En el ejemplo anterior de código, las expresiones estaban escritas solas y funcionan como están. Sin embargo, en C#, no todas las expresiones pueden existir de esa manera, muchas tienen que existir en acompañadas por una sentencia para que el programa compile, de otra manera obtendremos el error de compilación CS0201: Sólo se pueden

utilizar las expresiones de objeto assignment, call, increment, decrement y new como instrucción. Como ejemplo de dichas expresiones tenemos:

Literales

```
int n = 10;
var hola = "Hola";
```

En este caso estamos frente a las literales, las expresiones más simples que se pueden usar en C#, la primera expresión es 10, que se evalúa a un tipo por valor tipo int; la segunda expresión es "Hola" que se evalúa a un tipo por referencia tipo string. Si se hubiera usado únicamente 10; u "Hola"; sin sentencia, un error de compilación habría aparecido.

Literales y operadores

```
int i = 3 + 2;
```

En este caso 3 + 2 es una expresión en la que el operador es + cuyo resultado es un tipo por valor del tipo int, la sentencia que lo acompaña es declaración int i. De haber usado solo 3 + 2; habríamos obtenido un error de compilación.

Nombres simples

```
string hola2 = hola;
return hola; // Recordando de 'Hola' es un cadena
```

Dentro del código anterior, hola es un nombre simple, es el nombre de una variable de tipo string. Pero además de eso, es una expresión que se evalúa al valor que contenga en ese momento. En el primer caso, se usa una sentencia de asignación y en el segundo está acompañada de la sentencia return, ya que de otro modo si hubiéramos usado hola; de nuevo tendríamos un error de compilación.

Operadores

Los operadores son elementos dentro de C# que cuando se aplican a los operadores para devolver un valor a partir de ellos. Dentro del lenguaje existen tres tipos de operadores:

Unarios

Son los que toman un solo operando, entre ellos están ++, new, typeof, ~ y (T).

Binarios

Son los que toman dos operandos, entre ellos están *, +, >, ==.

Ternarios

Son los que toman más de dos operandos, el único caso dentro del lenguaje es el operador condicional ?: que toma tres operandos.

Pueden ver lista completa de operadores en este enlace.

Es importante destacar que algunos de los operadores pueden ser sobrecargados para modificarlos de acuerdo a nuestras necesidades, pero ese es tema de otro post. Recuerda que si

tienes dudas o algo no quedó tan claro no dudes en mandarme un tuit o un correo electrónico.

Anexo B: Equivalencias y diferencias entre lenguajes .Net

Se detallan una serie de contenidos y recursos donde se plantean las diferencias principales existentes entre los lenguajes más difundidos de la plataforma .Net.

Language Equivalentes
(Aplicado a VS 2008 y otras versiones anteriores y posteriores disponibles)
http://msdn.microsoft.com/en-us/library/czz35az4(vs.71).aspx

Equivalencias entre Visual Basic .NET y C#
Presenta una guía general de referencia sobre equivalencias entre los dos lenguajes más utilizados, C# y VB.Net
(Última Actualización: Agosto 2006)
http://www.elguille.info/NET/dotnet/equivalenciavbcs1.htm

Differences Between Visual Basic .NET and Visual C# .NET (white paper)
(Última Actualización: Abril 2007)
http://support.microsoft.com/kb/308470

Comparison of C Sharp and Visual Basic .NET
(Última Actualización: Abril 2012)
http://en.wikipedia.org/wiki/Comparison_of_C_Sharp_and_Visual_Basic_.NET

Comparison of C Sharp and Java
http://en.wikipedia.org/wiki/Comparison_of_C_Sharp_and_Java

Historial de Versiones

Versión	Fecha	Autor	Detalle
1.0	15/03/2012	Ezequiel Porta	Versión Inicial de contenidos de la Unidad 3 “Sintaxis y Orientación a Objetos de Lenguajes .Net” sobre tema Manejo de Errores y Excepciones (Try/Catch/Finally).
1.1	10/04/2012	Ezequiel Porta	Anexo de contenidos y referencias sobre equivalencias y diferencias entre lenguajes .Net Lista de enlaces y referencia a documentación de cátedra sobre el tema Orientación a Objetos.
1.2	01/04/2015	Ezequiel Porta	Adaptación a organización Programa Analitico 2014 donde Unidad 1 y 2 se unifican convirtiéndose cada en capítulos de la misma Unidad 1.

			Correcciones y agregados menores en particular lo relativo a la evolución del .Net Framework

---

[1] Puede ver un detalle de los "Lenguajes de Programación" .Net en el sitio de MSDN

[2] Lenguaje totalmente nuevo con la aparición de .Net cuya sintaxis tiene muchas similitudes a C con plena implementación del Paradigma Orientado a Objetos (POO).

[3] Lenguaje funcional creado por Don Syme del equipo de investigación de Microsoft.

[4] Ofrecer un diseño de software con tolerancia a errores.

[5] Componente del CLR conocido por su término en inglés como *Exception Manager* o *Exception Handling*.

[6] Vea en MSDN "Cómo administra las excepciones el motor de ejecución"

[7] Puede ampliar su conocimiento en MSDN sobre la clase System.Exception. Además puede consultar estas otras clases del namespace System: SystemException y ApplicationException.

[8] En Visual Basic .Net la palabra reservada equivalente a ***using*** es justamente ***Imports***.

[9] La Bibliografía está organizada en una primer sección de libros impresos, ordenados alfabéticamente por apellido del autor, muchos de ellos disponibles en Biblioteca de UTN Rosario en cuyo caso aparecerá un enlace a los datos de la publicación que figuran en línea en el buscador del sitio web de la biblioteca. Seguido se detallan aquellos recursos digitales ya sean otros libros disponibles en dicho formato que podrá encontrar en el aula virtual o de alguna otra fuente de información como pueden ser los libros en línea del MSDN del sitio oficial de Microsoft o sitios de empresas o profesionales que poseen contenidos de valor.