

# .NET FRAMEWORK

## y C#

### **Clase 3**

# Formatos compuestos en .NET

```
Console.Write("El doble de {0} es {1}", n, 2*n);
```

- La función de formato compuesto de .NET Framework le permite proporcionar una **lista de valores** y una **cadena de destino** que consiste en alternar texto fijo y marcadores de posición indizados, para obtener fácilmente una cadena de resultado que consta del texto fijo original entremezclado con los valores con formato.

# Formatos compuestos en .NET

- **Sintaxis de elemento de formato**

Cada elemento de formato adopta la siguiente forma.

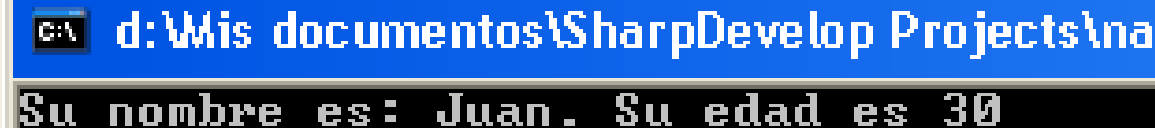
*{index[,alignment][:formatString]}*

# Formatos compuestos en .NET

- Index (Componente obligatorio)

Es un número que empieza por 0, que identifica un elemento de la lista de valores.

```
private static void Main(string[] vector)
{
    string nombre="Juan";
    int edad=30;
    Console.WriteLine("Su nombre es: {0}. Su edad es {1} ", nombre, edad);
    Console.ReadLine();
}
```

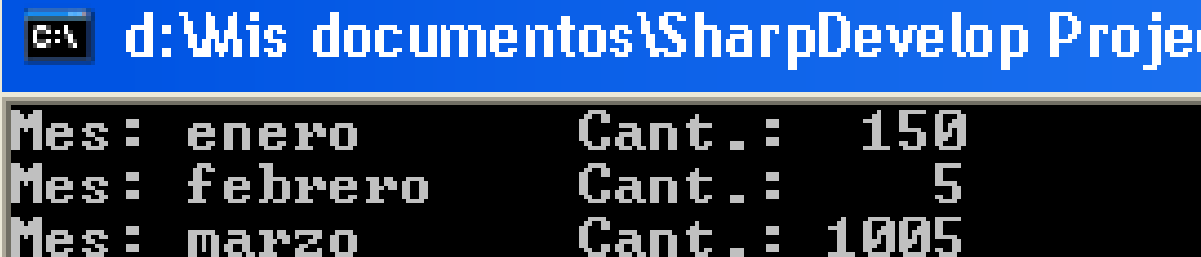


The screenshot shows a Windows command prompt window with a blue title bar. The title bar text is "d:\Mis documentos\SharpDevelop Projects\na". The command prompt itself has a black background and displays the output of the program: "Su nombre es: Juan. Su edad es 30".

# Formatos compuestos en .NET

- Alignment (Componente opcional)  
Es un entero con signo que indica el ancho de campo.  
Positivo → alineación derecha, negativo → alineación izquierda.

```
Console.WriteLine("Mes: {0,-10} Cant.: {1,4}", "enero", 150);  
Console.WriteLine("Mes: {0,-10} Cant.: {1,4}", "febrero", 5);  
Console.WriteLine("Mes: {0,-10} Cant.: {1,4}", "marzo", 1005);
```

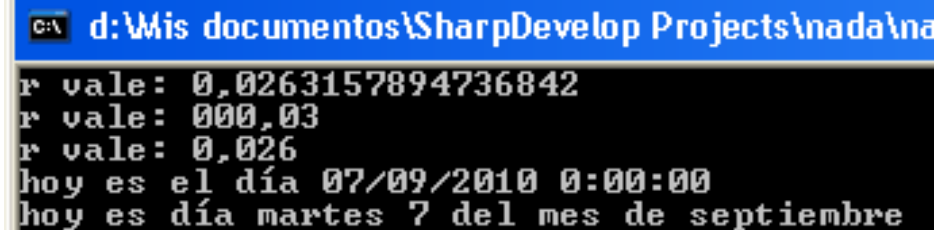


```
C:\ d:\Mis documentos\SharpDevelop Proje  
Mes: enero      Cant.: 150  
Mes: febrero    Cant.: 5  
Mes: marzo      Cant.: 1005
```

# Formatos compuestos en .NET

- Format String (Componente opcional)  
Consta de especificadores de formato personalizados o estándar.

```
private static void Main(string[] vector)
{
    double r=1/38.0 ;
    Console.WriteLine("r vale: {0} ", r);
    Console.WriteLine("r vale: {0:000.00} ", r);
    Console.WriteLine("r vale: {0:0.000} ", r);
    Console.WriteLine("hoy es el día {0}",DateTime.Today);
    Console.WriteLine("hoy es día {0:dddd d 'del mes de' MMMM} ",DateTime.Today);
    Console.ReadLine();
}
```



```
C:\ d:\Mis documentos\SharpDevelop Projects\nada\nada
r vale: 0,0263157894736842
r vale: 000,03
r vale: 0,026
hoy es el día 07/09/2010 0:00:00
hoy es día martes 7 del mes de septiembre
```

# Arreglos en varias dimensiones

- Matrices

```
int[,] matriz = new int[,]  
    {{1,2,3,4},  
     {5,6,7,8},  
     {9,10,11,12}};
```

- Acceso

```
matriz[2,2]+=matriz[1,1];
```

# Arreglos en varias dimensiones

```
int[,] mat = new int[3,4];  
for(int i=0;i<12;i++) {  
    mat[i/4,i%4]=i+1;  
}
```

1	2	3	4
5	6	7	8
9	10	11	12



# Arreglos de Arreglos

- Elemento del arreglo es un arreglo

```
int[][] tabla = {new int[2], new int[3]};
```

- Es equivalente a:

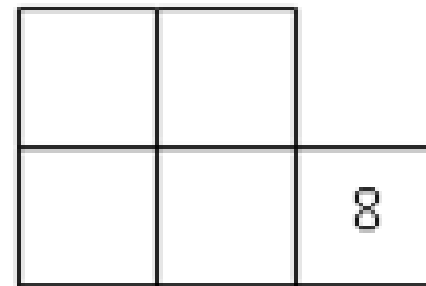
```
int[][] tabla2=new int[2][];
```

```
tabla2[0]=new int[2];
```

```
tabla2[1]=new int[3];
```

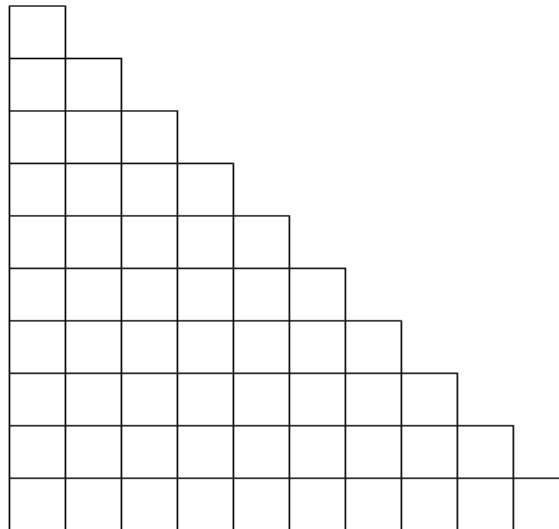
- Accesos

```
tabla[1][2] = 8;
```



# Arreglos de Arreglos

```
int[][] tablaEscalonada = new int[10][];  
for (int i=0;i<10;i++){  
    tablaEscalonada[i]=new int[i+1];  
}
```



# Colecciones

- Sería bueno manejar un **array que creciera dinámicamente**, o también, disponer de un array a cuyos valores pudiéramos **acceder a través de claves**, y no por el número de índice.
- Estas funcionalidades y algunas más, se encuentran disponibles en un tipo especial de array denominado **colección** (collection).
- Una colección es un objeto que internamente gestiona un array, pero que está preparado para manejarlo de una manera especial; podríamos definirla como **un array especializado** en ciertas tareas.

# El espacio de nombres System.Collections

Entre las clases más significativas encontramos:

- **ArrayList**. Una colección cuyo array se redimensiona dinámicamente.
- **BitArray**. Tabla de bits especialmente diseñada para evitar el desaprovechamiento de memoria que supone un arreglo de bool (bool[ ])
- **Stack**. Representa una lista de valores, en el que el último valor que entra, es el primero que sale.
- **Queue**. Representa una lista de valores, en el que el primer valor que entra, es el primero que sale.
- **Hashtable**. Las colecciones de este tipo, contienen un array cuyos elementos se basan en una combinación de clave y valor, de manera que el acceso a los valores se facilita, al realizarse mediante la clave.

# La clase ArrayList

```
public static void Main(){  
  
    //crear un arrayList sin elementos  
    ArrayList al1= new ArrayList();  
    //crear un arrayList indicando el número de  
    //elementos (Capacity) pero sin darles valor  
    ArrayList al2=new ArrayList(5);  
    //crear un arrayList a partir de un vector  
    string[] vector = new string[] { "a", "b", "c" };  
    ArrayList al3=new ArrayList(vector);  
}
```

# Agregar valores a un ArrayList

Una vez creado un ArrayList, podemos utilizar algunos de los métodos indicados a continuación para añadir valores a la colección.

- **Add(Valor)**. Añade el valor representado por Valor.
- **AddRange(Colección)**. Añade un conjunto de valores
- **Insert(Posición, Valor)**. Inserta el valor Valor en una posición determinada desplazando el resto de valores
- **InsertRange(Posición, Colección)**. Inserta un conjunto de valores a partir de una posición determinada.
- **SetRange(Posición, Colección)**. Sobrescribe elementos en un array con los valores de la colección Colección, comenzando en la posición Posición.

# Recorrer un ArrayList

```
public static void Main(){
    ArrayList al1= new ArrayList();
    al1.AddRange(new char[] { 'a', 'e', 'i', 'o', 'u' });

    //recorrer el arrayList con bucle for
    for(int i=0;i<al1.Count;i++)
        Console.WriteLine(al1[i]);

    //recorrer el arrayList con bucle foreach
    foreach(char c in al1)
        Console.WriteLine(c);

    Console.ReadKey();
}
```

# Capacidad y valores en una colección ArrayList

- La **capacidad** de un ArrayList hace referencia al número de **elementos reservados** en el array subyacente
- Los **valores asignados** son aquellos elementos del array a los que se ha asignado valor mediante métodos como **Add( )** o **AddRange( )**.
- Pregunta: ¿Por qué cree que la capacidad y la cantidad de valores asignados suelen ser distintas?



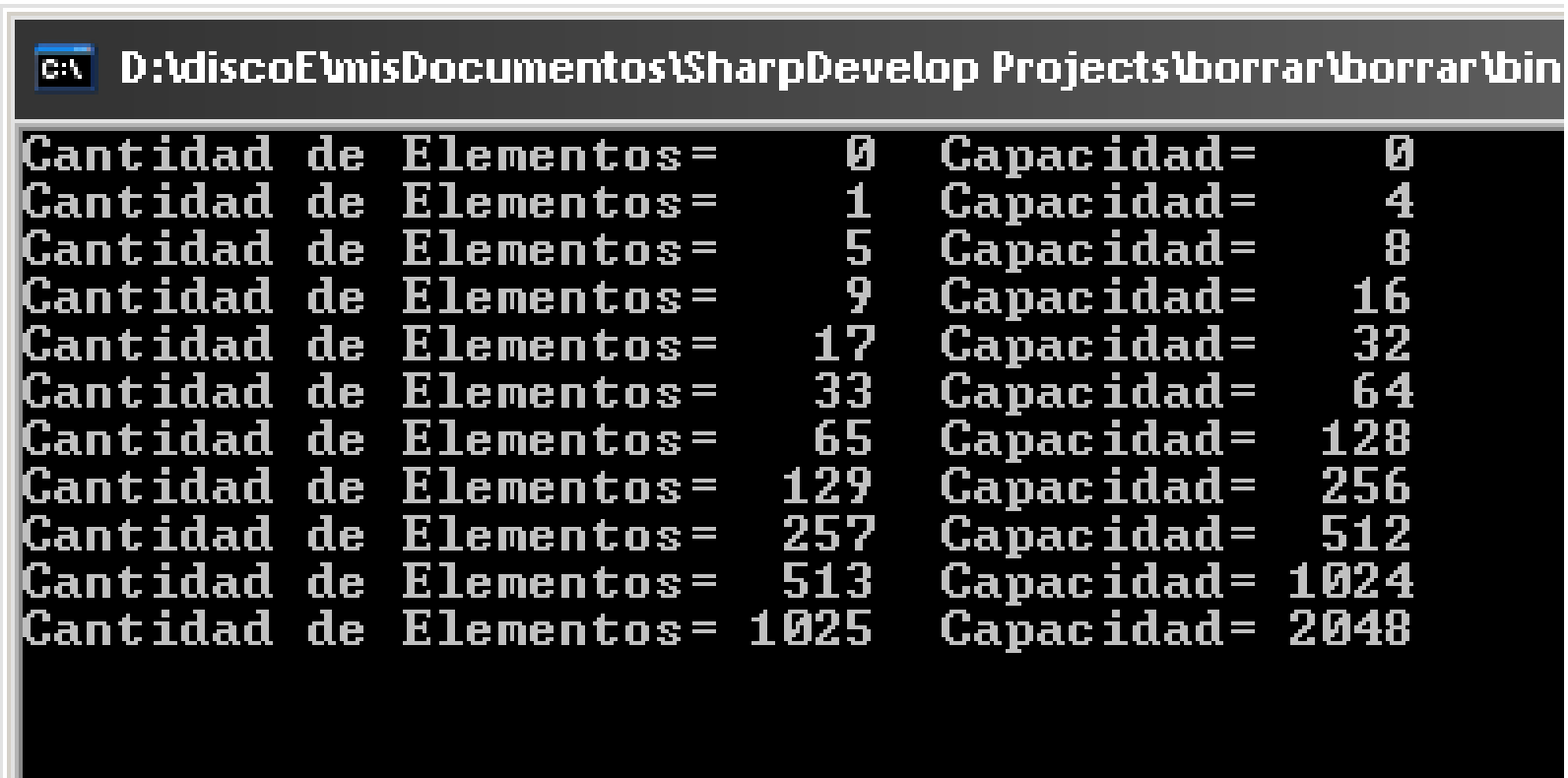
# Capacidad y valores en una colección ArrayList

¿Qué hace el siguiente programa?

```
public static void Main() {
    ArrayList al1= new ArrayList();
    int capacidad=-1;
    for (int i=1;i<=2000;i++){
        if (capacidad != al1.Capacity){
            Console.WriteLine("Cantidad de Elementos={0,5}  " +
                              "Capacidad={1,5}",al1.Count,al1.Capacity);
            capacidad=al1.Capacity;
        }
        al1.Add(i);
    }
    Console.ReadKey();
}
```

# Capacidad y valores en una colección ArrayList

La salida del programa es:



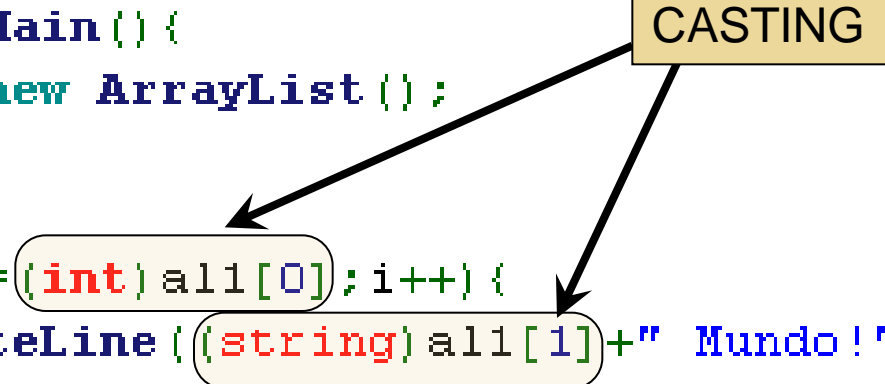
```
D:\discoE\misDocumentos\SharpDevelop Projects\borrar\borrar\bin
Cantidad de Elementos= 0    Capacidad= 0
Cantidad de Elementos= 1    Capacidad= 4
Cantidad de Elementos= 5    Capacidad= 8
Cantidad de Elementos= 9    Capacidad= 16
Cantidad de Elementos= 17   Capacidad= 32
Cantidad de Elementos= 33   Capacidad= 64
Cantidad de Elementos= 65   Capacidad= 128
Cantidad de Elementos= 129  Capacidad= 256
Cantidad de Elementos= 257  Capacidad= 512
Cantidad de Elementos= 513  Capacidad= 1024
Cantidad de Elementos= 1025 Capacidad= 2048
```

# Elementos del ArrayList

- Los objetos en un **ArrayList**, son siempre **object**, por lo tanto muchas veces habrá que realizar operaciones de conversión explícita de tipos (**Casting**)

# Elementos del ArrayList

```
public static void Main(){
    ArrayList al1= new ArrayList();
    al1.Add(5);
    al1.Add("Hola");
    for (int i=1;i<= (int) al1[0];i++){
        Console.WriteLine((string) al1[1]+ " Mundo!");
    }
    Console.ReadKey();
}
```



# BitArray

- Aunque lógicamente los elementos de un **BitArray** se manipulan como si fuese **bool[ ]**, en realidad cada valor lógico incluido en un BitArray ocupa sólo un bit. (El tipo básico **bool** ocupa en memoria 1 byte)
- El truco consiste en representar internamente un **BitArray** como **int[ ]** en las que cada bit de sus elementos es tratado como un valor lógico.

# Creación de BitArray

```
// tabla = {false, false, false}
BitArray tabla = new BitArray(3);

// tabla2 = {true, true, true}
BitArray tabla2 = new BitArray(3, true);

// tabla3 = {false, true, false }
BitArray tabla3 = new BitArray(new bool[] {false, true, false});

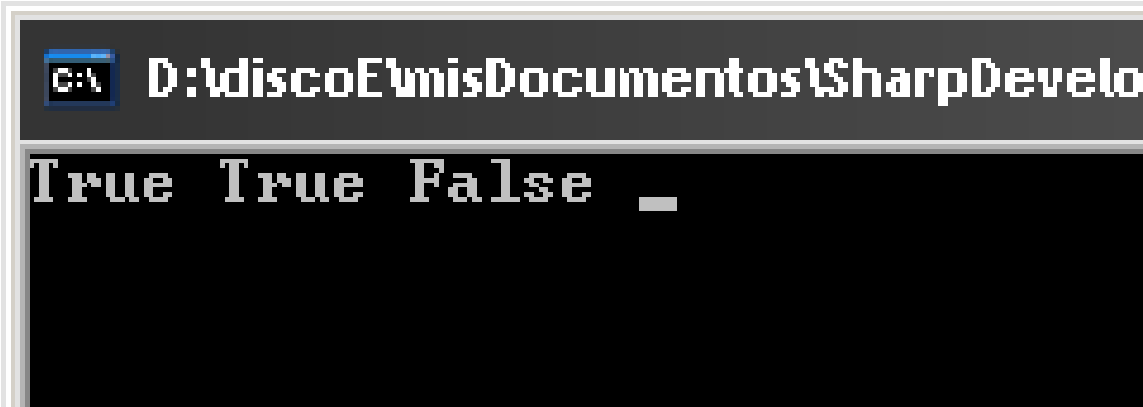
// tabla4 = {false, true, false, true, false, false, false, false}
BitArray tabla4 = new BitArray(new byte[] {10});

// tabla5 = {false, true, false, true, false, false, false, false}
BitArray tabla5 = new BitArray(tabla4);
```

# Operaciones lógicas

- En **BitArray** se han definido un conjunto de métodos que permiten hacer con comodidad las operaciones lógicas más comunes

```
public static void Main(){  
    BitArray tabla1 = new BitArray(new bool[] {false, true, false});  
    BitArray tabla2 = new BitArray(new bool[] {true, false, false});  
    tabla1.Or(tabla2);  
    foreach(bool b in tabla1) Console.Write(b + " ");  
    Console.ReadKey();  
}
```



C:\ D:\discoE\misDocumentos\SharpDevelo  
True True False \_

# La Clase Stack

- Las pilas están implementadas como objetos **System.Collections.Stack**, y aparte de los métodos comunes a todas las colecciones cuentan con:
  - **Push(object o)**: Coloca el objeto indicado en la cima de la pila.
  - **object Pop()**: Devuelve el elemento que hubiese en la cima de la pila y lo saca.
  - **object Peek()**: Devuelve el elemento de la cima de la pila pero sin sacarlo de ella.



# La Clase Stack

```
public static void Main() {  
    Stack pila=new Stack();  
    pila.Push("CASA");  
    pila.Push(10);  
    pila.Push('a');  
    while (pila.Count>0)  
        Console.WriteLine(pila.Pop());  
    Console.ReadKey();  
}
```



```
c:\> D:\discoE\misDocumentos\SharpDe  
a  
10  
CASA
```

# La clase Queue

- Las colas están implementadas como objetos **System.Collections.Queue**, y aparte de los métodos comunes a todas las colecciones también cuentan con:
- **Enqueue(object o)**: Coloca el objeto indicado al final de la cola.
- **object Dequeue()**: Devuelve el primer objeto de la cola y lo saca de ella.
- **object Peek()**: Devuelve el primer objeto de la cola pero no lo saca de ella.

# La clase Queue

```
public static void Main() {  
    Queue cola=new Queue();  
    cola.Enqueue("CASA");  
    cola.Enqueue(10);  
    cola.Enqueue('a');  
    while (cola.Count>0)  
        Console.WriteLine(cola.Dequeue());  
    Console.ReadKey();  
}
```



The screenshot shows a Windows command prompt window with a dark background. The title bar at the top indicates the file path 'D:\discoE\misDocumentos'. The command prompt displays the output of the program, which consists of three lines: 'CASA', '10', and 'a'. Each line is preceded by a small white square, indicating that the program is waiting for a key press before displaying the next output.

```
C:\> D:\discoE\misDocumentos  
CASA  
10  
a
```

# La clase Hashtable

- El acceso a los valores del array que gestiona internamente se realiza a través de una **clave** asociada a cada elemento.

```
public static void Main(){
    Hashtable ht=new Hashtable();
    ht["nombre"]="juan";
    ht["edad"]=40;
    DateTime undia=DateTime.Parse("5/10/2009");
    ht[undia]="descansó";
    ht[DayOfWeek.Monday]="Trabaja";
    Console.WriteLine("Su nombre es: {0}",ht["nombre"]);
    Console.WriteLine("Tiene {0} años",ht["edad"]);
    Console.WriteLine("El día {0} {1}",
        undia.ToString("d 'de' MMMM 'de' yyyy"),
        ht[undia]);
    Console.WriteLine("Los lunes {0}",ht[DayOfWeek.Monday]);
    Console.ReadKey();
}
```

# La clase Hashtable

```
public static void Main(){
    Hashtable ht=new Hashtable();
    ht["nombre"]="juan";
    ht["edad"]=40;
    DateTime undia=DateTime.Parse("5/10/2009");
    ht[undia]="descansó";
    ht[DayOfWeek.Monday]="Trabaja";
    Console.WriteLine("Su nombre es: {0}",ht["nombre"]);
    Console.WriteLine("Tiene {0} años",ht["edad"]);
    Console.WriteLine("El día {0} {1}",
        undia.ToString("d 'de' MMMM 'de' yyyy"),
        ht[undia]);
    Console.WriteLine("Los lunes {0}",ht[DayOfWeek.Monday]);
    Console.ReadKey();
}
```

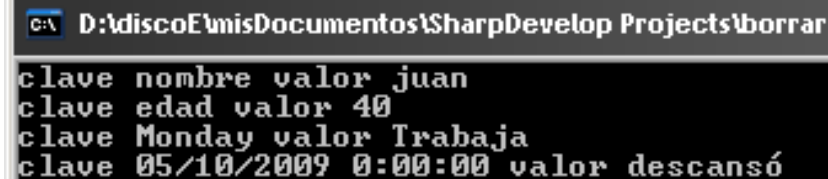
ca D:\discoElmisDocumentos\SharpDevelop Projects\bor

```
Su nombre es: juan
Tiene 40 años
El día 5 de octubre de 2009 descansó
Los lunes Trabaja
```

# La clase Hashtable

- Observe que no se respeta el orden en que se agregaron los elementos

```
public static void Main() {  
    Hashtable ht=new Hashtable();  
    ht[DayOfWeek.Monday]="Trabaja";  
    ht["edad"]=40;  
    ht["nombre"]="juan";  
    DateTime undia=DateTime.Parse("5/10/2009");  
    ht[undia]="descansó";  
    foreach(DictionaryEntry elemento in ht)  
        Console.WriteLine("clave {0} valor {1}",  
                            elemento.Key,  
                            elemento.Value);  
  
    Console.ReadKey();  
}
```



```
C:\D:\discoE\misDocumentos\SharpDevelop Projects\borrar  
clave nombre valor juan  
clave edad valor 40  
clave Monday valor Trabaja  
clave 05/10/2009 0:00:00 valor descansó
```

# Excepciones

- Las excepciones son **errores en tiempo de ejecución**.
- Ejemplos de excepciones: Intentar dividir por cero, escribir un archivo de sólo lectura, referencias a null, acceder a un arreglo con un índice fuera del rango válido, etc.

# Excepciones

- Ejemplo

```
static void Main()
```

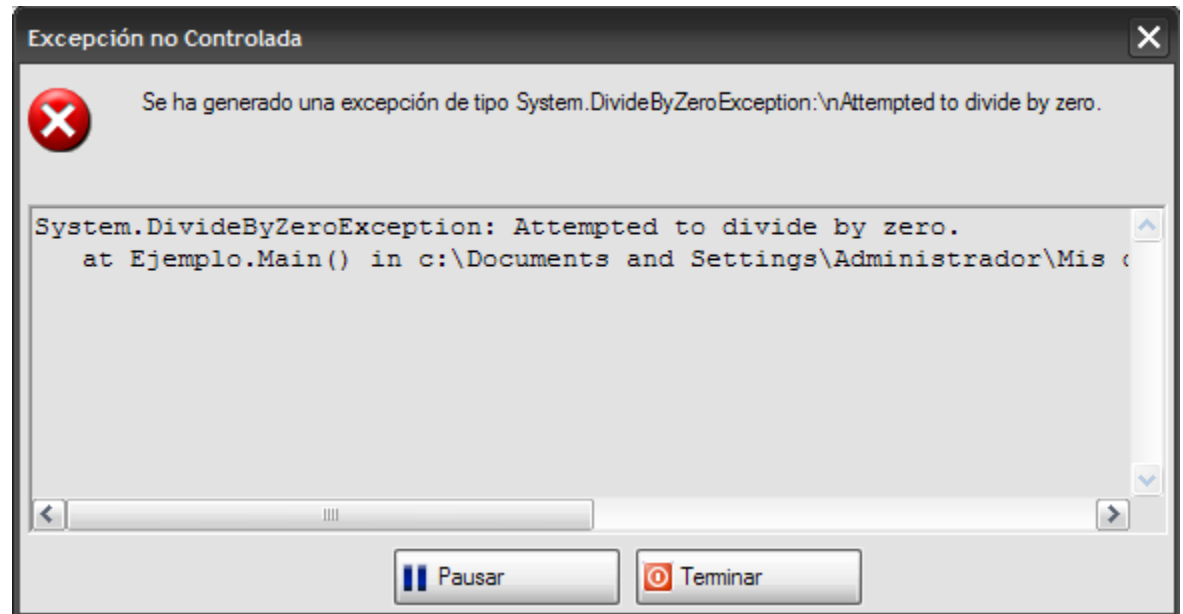
```
{
```

```
    int x = 10, y = 0;
```

→ 

```
x /= y;    // intento de dividir por cero
```

```
}
```





# Excepciones

- **Predefinidas**

- DivideByZeroException
- OverflowException
- NullReferenceException
- IndexOutOfRangeException
- IO.IOException
- ... y muchas más

# Excepciones. La sentencia try

Boque try: Aquí dentro se controla la ocurrencia de excepciones

Cláusulas catch: Esta sección contiene manejadores para el caso de producirse excepciones en el bloque try

Bloque finally: Contiene código que se ejecuta siempre, se haya producido o no alguna excepción

```
try
{
    statements
}
```

```
catch( ... )
{
    statements
}
catch( ... )
{
    statements
}
catch ...
```

```
finally
{
    statements
}
```

} Esta sección es requerida

} Al menos una de estas secciones debe estar presente

# Excepciones. La cláusula catch

```
catch
```

```
{
```

```
    Statements
```

```
}
```

## Cláusula catch general

- No lleva parámetro
- "Hace Matching" con cualquier tipo de excepción

```
catch( ExceptionType )
```

```
{
```

```
    Statements
```

```
}
```

## Cláusula catch específica

- Toma como parámetro el nombre de una excepción
- "Hace Matching" con cualquier excepción de ese tipo

```
catch( ExceptionType ExceptionVariable )
```

```
{
```

```
    Statements
```

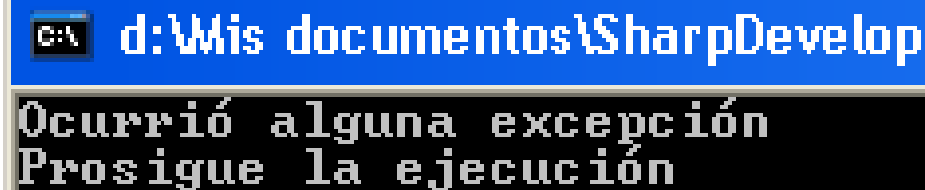
```
}
```

## Cláusula catch específica con objeto

- Incluye un identificador luego del nombre de la excepción
- El identificador actúa como una variable local dentro del bloque catch

# Excepciones. Catch general

```
byte b=255;  
try{  
    b++;  
}catch{  
    Console.WriteLine("Ocurrió alguna excepción");  
}  
Console.WriteLine("Prosigue la ejecución");
```

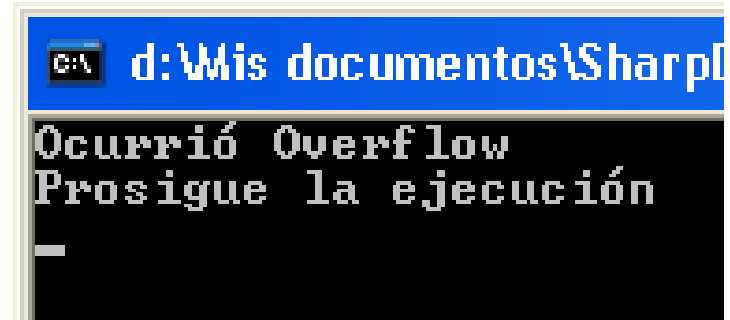


A screenshot of a Windows command prompt window. The title bar is blue and contains the text "d:\Mis documentos\SharpDevelop". The command prompt itself has a black background with white text. It shows two lines of output: "Ocurrió alguna excepción" and "Prosigue la ejecución".

```
d:\Mis documentos\SharpDevelop  
Ocurrió alguna excepción  
Prosigue la ejecución
```

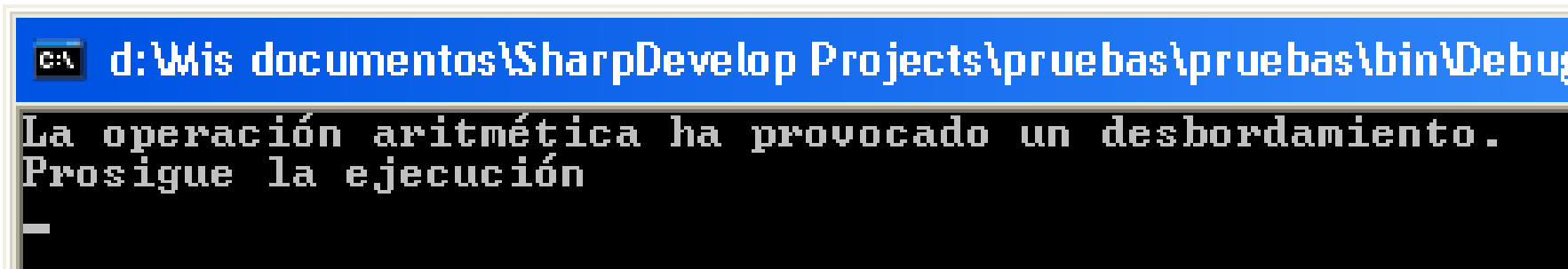
# Excepciones. Catch específica

```
byte b=255;
try{
    b++;
}catch (InvalidCastException){
    Console.WriteLine("Ocurrió InvalidCast");
}catch (OverflowException){
    Console.WriteLine("Ocurrió Overflow");
}catch{
    Console.WriteLine("Ocurrió alguna excepción");
}
Console.WriteLine("Prosigue la ejecución");
```



# Excepciones. Catch con variable

```
byte b=255;  
try{  
    b++;  
} catch (Exception e) {  
    Console.WriteLine(e.Message);  
}  
Console.WriteLine("Prosigue la ejecución");
```



The screenshot shows a Windows command prompt window with a blue title bar. The title bar text is "C:\ d:\Mis documentos\SharpDevelop Projects\pruebas\pruebas\bin\Debug". The command prompt window has a black background with white text. The text displayed is "La operación aritmética ha provocado un desbordamiento." followed by "Prosigue la ejecución" on the next line. A white cursor is visible at the end of the second line.

# Excepciones. Bloque finally

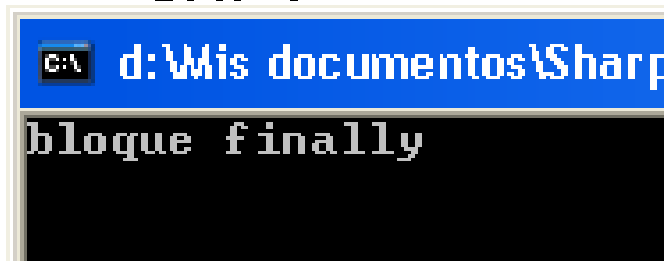
- El bloque `finally` se ejecuta **SIEMPRE** antes de finalizar el `try` independientemente de la ejecución o no de alguna cláusula `catch`
- El bloque `finally` se ejecuta aún si el bloque `try` posee una sentencia `return`

# Excepciones. Bloque finally

Siendo **x** e **y** variables enteras:

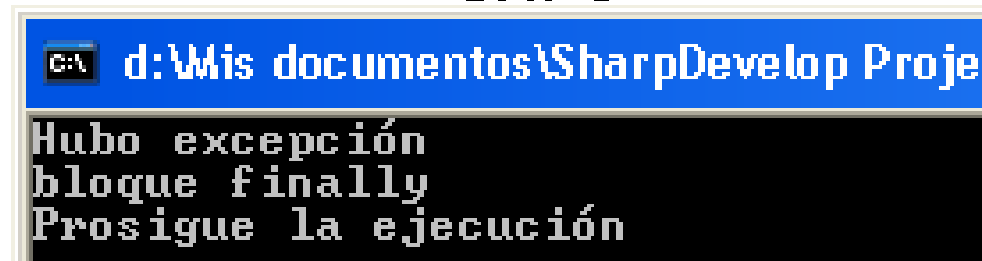
```
try{  
    y=1/x;  
    return;  
}catch{  
    Console.WriteLine("Hubo excepción");  
}finally{  
    Console.WriteLine("bloque finally");  
}  
Console.WriteLine("Prosigue la ejecución");
```

Si x=1



```
d:\Mis documentos\Sharp  
bloque finally
```

Si x=0



```
d:\Mis documentos\SharpDevelop Proje  
Hubo excepción  
bloque finally  
Prosigue la ejecución
```



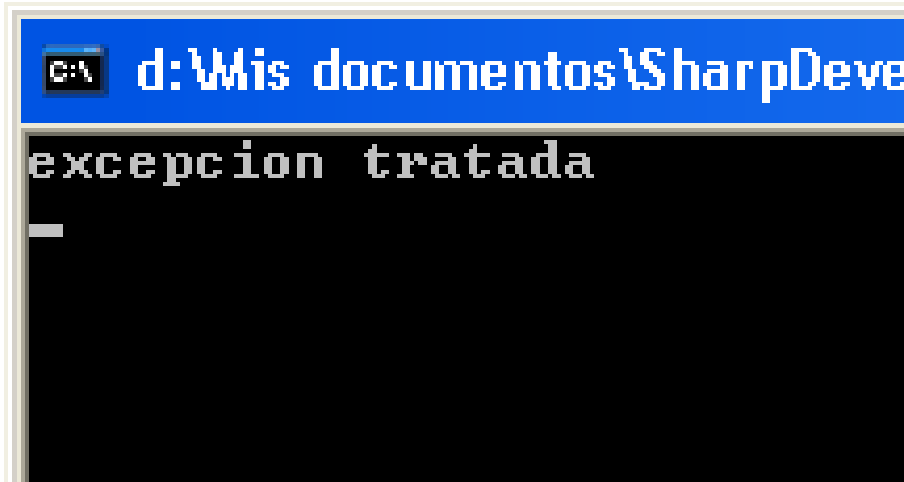
# Propagación de una excepción

- Si **metodo1** invoca a **metodo2** y dentro de este último se produce una excepción para la cual **no existe una cláusula catch** adecuada, la excepción se propaga a **metodo1**

# Propagación de una excepción

```
static void metodo1() {  
    try{  
        metodo2();  
    }catch{  
        Console.WriteLine("excepcion tratada");  
    }  
}
```

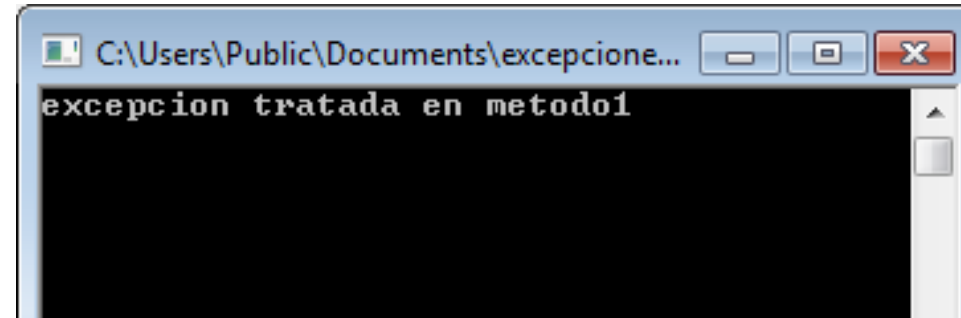
```
static void metodo2() {  
    byte b=255;  
    b++;  
}
```



The screenshot shows a Windows command prompt window with a blue title bar. The title bar text is "d:\Mis documentos\SharpDeve". The command prompt window has a black background and displays the text "excepcion tratada" in white, which is the output of the program. A white cursor is visible on the line following the output text.

# Propagación de una excepción

```
static void metodo1(){  
    try{  
        metodo2();  
    }catch{  
        Console.WriteLine("excepcion tratada en metodo1");  
    }  
}  
static void metodo2(){  
    try{  
        byte b=255;  
        b++;  
    } catch (DivideByZeroException){  
        Console.WriteLine("excepcion tratada en metodo2");  
    }  
}
```



# Trabajo con archivos. Introducción

- La **BCL** incluye todo un espacio de nombres llamado **System.IO** especialmente orientado al trabajo con archivos.
- La clase **Path** definida en **System.IO** incluye un conjunto de miembros estáticos diseñados para realizar cómodamente las operaciones más frecuentes relacionadas con rutas.

# Clase Path

Debe incluirse

`using System.IO;`

```
string archivo="c:\\documentos\\notas.txt";  
Console.WriteLine(Path.GetFullPath(archivo));  
Console.WriteLine(Path.GetDirectoryName(archivo));  
Console.WriteLine(Path.GetFileName(archivo));  
Console.WriteLine(Path.GetFileNameWithoutExtension(archivo));  
Console.WriteLine(Path.GetExtension(archivo));  
Console.WriteLine(Path.ChangeExtension(archivo,"doc"));  
Console.WriteLine(Path.GetTempPath());  
Console.WriteLine(Path.GetRandomFileName());
```

 d:\Mis documentos\SharpDevelop Projects\Ejercicio1\Ejercicio1\bin\Debug\

```
c:\documentos\notas.txt  
c:\documentos  
notas.txt  
notas  
.txt  
c:\documentos\notas.doc  
C:\Documents and Settings\usuario\Configuración local\Temp\  
moalbdya.ams
```

# DirectoryInfo - FileInfo

- Para trabajar con archivos se utilizan objetos de la clase **FileInfo** y para trabajar con directorios objetos de la clase **DirectoryInfo**.
- También existen las clases **File** y **Directory** que sólo tienen métodos estáticos (al igual que **Path**) útiles para realizar tareas sencillas. Aunque pueden ser más directas que las anteriores, pues no requieren la creación de ningún objeto son **menos poderosas** y **menos eficientes**.

# Ejemplo (DirectoryInfo – FileInfo)

```
public static void Main(string[] args) {  
    DirectoryInfo carpeta;  
    carpeta = new DirectoryInfo(Path.GetTempPath());  
    FileInfo[] archivos = carpeta.GetFiles();  
    foreach(FileInfo archivo in archivos){  
        if (archivo.Extension.ToLower() != ".tmp"){  
            Console.WriteLine("{0} {1} bytes",  
                               archivo.Name,  
                               archivo.Length);  
        }  
    }  
}
```

Imprime en la consola los nombres y tamaño en bytes de todos los archivos de extensión distinta a “.tmp” que se encuentren en la carpeta temporal del sistema

# Archivos de texto

- El trabajo con archivos en .NET está ligado al concepto de **stream o flujo de datos**, que consiste en tratar su contenido como una secuencia ordenada de datos.
- El concepto de **stream** es aplicable también a otros tipos de almacenes de información tales como **conexiones de red** o **buffers en memoria**.
- La **BCL** proporciona las clases **StreamReader** y **StreamWriter**. Los objetos de estas clases facilitan la lectura y escritura de archivos de textos.



# StreamReader

- Para facilitar la lectura de flujos de texto **StreamReader** ofrece una familia de métodos que permiten leer sus caracteres de diferentes formas:
- **De uno en uno:** El método **int Read()** devuelve el próximo carácter del flujo. Tras cada lectura la posición actual en el flujo se mueve un carácter hacia delante.
- **Por líneas:** El método **string ReadLine()** devuelve la siguiente línea del flujo (y avanza la posición en el flujo). Una línea de texto es cualquier secuencia de caracteres terminada en **'\n'**, **'\r'** ó **"\r\n"**, aunque la cadena que devuelve no incluye dichos caracteres.
- **Por completo:** **string ReadToEnd()**, que nos devuelve una cadena con todo el texto que hubiese desde la posición actual del flujo sobre el que se aplica hasta el final del mismo (y avanza hasta el final del flujo).

# StreamWriter

- **StreamWriter** ofrece métodos que permiten:
- **Escribir cadenas de texto:** El método **Write()** escribe cualquier cadena de texto en el destino que tenga asociado. Pueden utilizarse formatos compuestos.
- **Escribir líneas de texto:** El método **WriteLine()** funciona igual que **Write()** pero añade un indicador de fin de línea. Pueden utilizarse formatos compuestos
- Dado que el indicador de fin de línea depende de cada sistema operativo, **StreamWriter** dispone de una propiedad **string NewLine** mediante puede configurarse este indicador. Su valor por defecto es el “**\r\n**” correspondiente al indicador de fin de línea en Windows, pero también puede dársele el valor “**\n**” correspondiente a Linux.

# Ejemplo 1

```
using System;
using System.IO;
class Program{
    public static void Main(string[] args) {
        StreamReader sr = new StreamReader("fuente.txt");
        StreamWriter sw = new StreamWriter("destino.txt");
        string linea;
        while (!sr.EndOfStream) {
            linea=sr.ReadLine();
            sw.WriteLine(linea);
        }
        sr.Close();sw.Close();
    }
}
```

La propiedad `EndOfStream` indica si está al final de la secuencia

El método `Close()` libera los recursos ocupados

# Ejemplo 2

```
using System;
using System.IO;
class Program{
    public static void Main(string[] args) {
        StreamReader sr = new StreamReader("fuente.txt");
        StreamWriter sw = new StreamWriter("destino.txt");
        try {
            sw.Write(sr.ReadToEnd());
        } catch (Exception e){
            Console.WriteLine(e.Message);
        } finally {
            sr.Close(); sw.Close();
        }
    }
}
```

Esta línea hace todo el trabajo

Es altamente recomendable prever excepciones y liberar los recursos en el bloque finally

# Repaso práctica 2

- Qué líneas del siguiente código provocan conversiones **boxing** y **unboxing**.

```
char c1='A';
```

```
string st1="A";
```

```
object o1=c1;
```

boxing

```
object o2=st1;
```

```
char c2=(char) o1;
```

unboxing

```
string st2=(string) o2;
```

No hay ni boxing ni unboxing porque el tipo string no es un tipo valor

# Repaso práctica 2

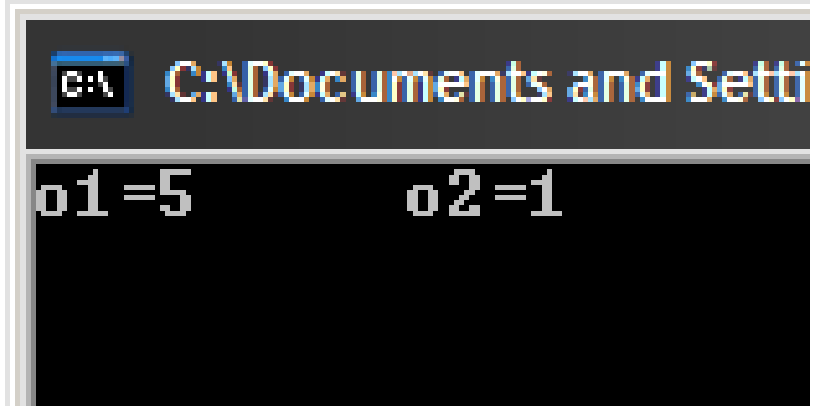
- Por qué si **object** es un tipo referencia la salida por consola no es **o1=5 o2=5** ?

```
object o1 = 1;
```

```
object o2 = o1;
```

```
o1 = 5;
```

```
Console.Write("o1={0}\t o2={1}", o1, o2);
```



A screenshot of a Windows command prompt window. The title bar shows the path "C:\Documents and Settings". The command prompt displays the output of the code: "o1=5" followed by a tab character and "o2=1".

```
o1=5      o2=1
```

# Repaso práctica 2

- ¿Qué se imprime por consola? (no es igual al ejercicio de la práctica)

```
int sum=0,i=1;
```

```
while (i++<=10)(;)
```

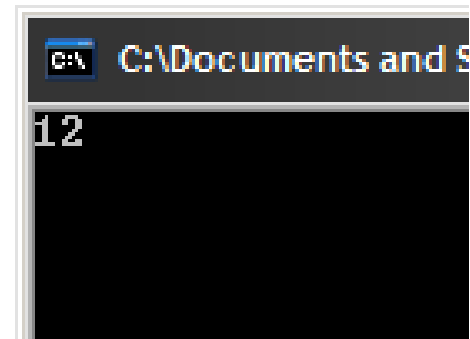
Cuidado!

```
{
```

```
    sum += i;
```

```
}
```

```
Console.WriteLine(sum);
```



# Repaso práctica 2

```
string st="";  
for (int i=1; i<=100000; i++) {  
    st+="a";  
}
```

- ¿No sería más eficiente utilizar un **StringBuilder**?



# Repaso práctica 2

- ¿Qué hacen las siguientes instrucciones?

```
DateTime f1=DateTime.Now;
```

```
//. . .Se procesa algo
```

```
DateTime f2=DateTime.Now;
```

```
double lapso=f2.Subtract(f1).TotalMilliseconds;
```