Programación con C# .NET

Tema 2:

El lenguaje C#

Introducción

- C# es el último en una línea de evolución de los lenguajes derivados de C, que incluye C++ y Java.
- Usado por Microsoft para desarrollar la mayoría del código de .NET.
- Por tanto, es el lenguaje ideal para el desarrollo en .NET
- C# introduce varias mejoras sobre C++ en las áreas de seguridad de datos, versionamiento, eventos y recolección de basura.
- C# provee acceso al SO, COM y APIs y soporta el modo *unsafe* que permite el uso de punteros como en C.
- Más simple que C++ pero tan poderoso y flexible como él.

Microsoft C#.net

Estructura de los programas

- Un programa en C# contiene:
  - Uno o más **ficheros** que contienen:
    - Uno o más **espacios de nombres** que contienen:
      - **Tipos de datos**: clases, estructuras, interfaces, enumeraciones y delegados
- Si no se declara un ***namespace*** se asume el global por defecto
- Un ejecutable ha de contener obligatoriamente una función Main (punto de entrada al programa)

```
static void Main()
static int Main()
static void Main(string[] args)
static int Main(string[] args)
```

- Para acceder a un tipo podemos usar un camino absoluto:

```
System.Console.WriteLine(...);
```

  o relativo:

```
using System;...;
Console.WriteLine(...);
```

Ejemplo:

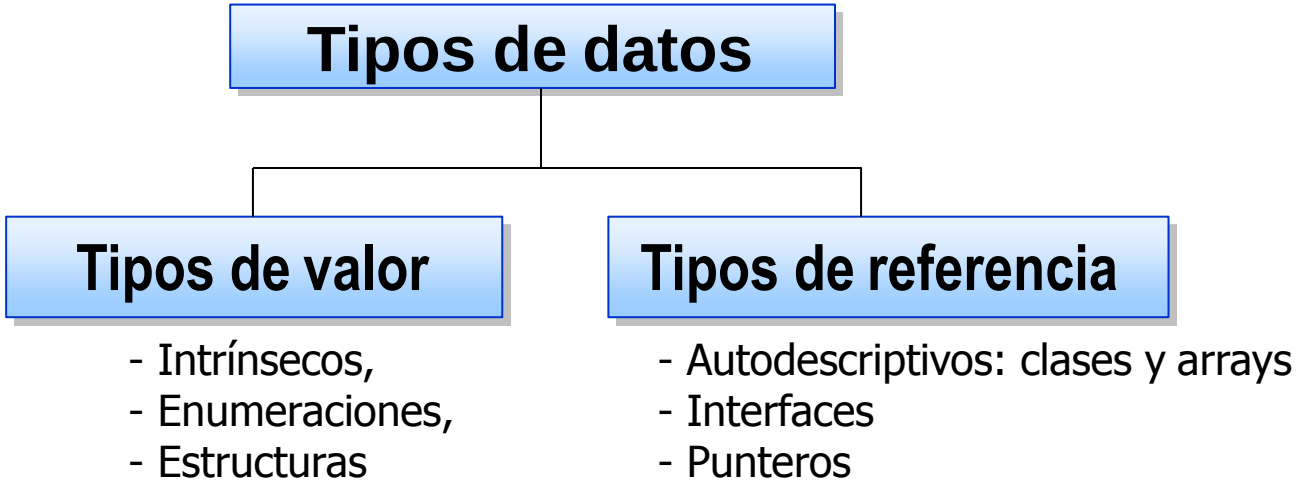
```
namespace N1 {
  class C1 {
    // ...
  }
  struct S1 {
    // ...
  }
  interface I1 {
    // ...
  }
  delegate int D1();
  enum E1 {
    // ...
  }
}
```

- Comentarios de una línea: //
- Comentarios de varias líneas: /* .... */

Microsoft C#.net

Tipos de datos

- **Sistema común de tipos**
  - **CTS** € sistema común de tipos compartido por todos los lenguajes .NET
  - El CTS admite tanto tipos de valor como tipos de referencia



  - **Todos** los tipos de datos en C# derivan de **System.Object** € sistema de tipos unificado.
  - Cualquier tipo puede ser tratado como un objeto.

Microsoft C#.net

Tipos de datos básicos

Tipo	Descripción	Bits	Rango de valores	Alias
SByte	*Bytes* con signo	8	**[-128, 127]**	**sbyte**
Byte	*Bytes* sin signo	8	**[0 , 255]**	**byte**
Int16	Enteros cortos con signo	16	**[-32.768, 32.767]**	**short**
UInt16	Enteros cortos sin signo	16	**[0, 65.535]**	**ushort**
Int32	Enteros normales	32	**[-2.147.483.648, 2.147.483.647]**	**int**
UInt32	Enteros normales sin signo	32	**[0, 4.294.967.295]**	**uint**
Int64	Enteros largos	64	**[-9.223.372.036.854.775.808, 9.223.372.036.854.775.807]**	**long**
UInt64	Enteros largos sin signo	64	**[0-18.446.744.073.709.551.615]**	**ulong**
Single	Reales con 7 dígitos de precisión	32	**[1,5×10^{-45} - 3,4×10^{38}]**	**float**
Double	Reales de 15-16 dígitos de precisión	64	**[5,0×10^{-324} - 1,7×10^{308}]**	**double**
Decimal	Reales de 28-29 dígitos de precisión	128	**[1,0×10^{-28} - 7,9×10^{28}]**	**decimal**
Boolean	Valores lógicos	32	**true, false**	**bool**
Char	Caracteres Unicode	16	**['\u0000', '\uFFFF']**	**char**
String	Cadenas de caracteres	Variable	**El permitido por la memoria**	**string**
Object	Cualquier objeto	Variable	**Cualquier objeto**	**object**

Microsoft C#.net

Tipos de datos (cont)

- **Identificadores**
  - Se usan para dar nombres a los elementos de un programa como variables, constantes y métodos.
  - Consta de caracteres alfanuméricos y _
  - **Sensible** a mayúsculas y minúsculas. Comenzar con letra o _
  - Palabras reservadas:

abstract, as, base, bool, break, byte, case, catch, char, checked, class, const, continue, decimal, default, delegate, do, double, else, enum, event, explicit, extern, false, finally, fixed, float, for, foreach, goto, if, implicit, in, int, interface, internal, lock, is, long, namespace, new, null, object, operator, out, override, params, private, protected, public, readonly, ref, return, sbyte, sealed, short, sizeof, stackalloc, static, string, struct, switch, this, throw, true, try, typeof, uint, ulong, unchecked, unsafe, ushort, using, virtual, void, while.

  - Si se quiere usar un identificador que es una palabra reservada hay que usar como prefijo el carácter '**@**':

```
Object @this; // @ previene el conflicto con "this"
```

Microsoft C#.net

Tipos de datos (cont)

- **Variables**
  - Una variable en C# representa la localización en memoria donde una instancia de un tipo es guardada
  - Es simplemente una capa encima del sistema de tipos independiente del lenguaje de .NET (CTS)
  - Recordar la distinción entre tipos de valor y tipos de referencia
    - Tipos de valor son tipos simples como '`int`', '`long`' y '`char`'
    - Los objetos, strings y arrays son ejemplos de tipos de referencia
  - Los tipos de valor se derivan de System.ValueType

Tipos de datos (cont)

- Comparación entre variables tipo valor y tipo referencia:

Variables de tipo valor:

- Contienen sus datos directamente
- Cada una tiene su propia copia de datos
- Las operaciones sobre una no afectan a otra

Variables de tipo referencia:

- Almacenan referencias a sus datos (conocidos como objetos)
- Dos variables de referencia pueden apuntar al mismo objeto
- Las operaciones sobre una pueden afectar a otra

Tipos de datos (cont)

- Las variables de valor pueden iniciarse al declararse:

```
bool bln = true;
byte byt1 = 22;
char ch1='x', ch2='\u0061';   // Unicode para 'a'
decimal dec1 = 1.23M;
double dbl1=1.23, dbl2=1.23D;
short sh = 22;
int i = 22;
long lng1 =22, lng2 =22L;     // 'L' long
sbyte sb = 22;
float f=1.23F;
ushort us1=22;
uint ui1=22, ui2=22U;         // 'U' unsigned
ulong ul1 =22, ul2=22U, ul3=22L, ul4=2UL;
```

Tipos de datos (cont)

- Los valores de referencia son creados con la palabra clave new:

```
object o = new System.Object();
```

- Una variable String se puede inicializar directamente:

```
string s = "Hola"; // usan caracteres Unicode de 2 cars
```

- C# soporta secuencias de escape como en C:

```
string s1 = "Hola\n"; // salto de línea
string s2 = "Hola\tque\ttal"; // tabulador
```

- Como las sentencias de escape comienzan con '\', para escribir este carácter hay que doblarlo, o usar '@':

```
string s3 = "c:\\WINNT";
string s4 = @"C:\WINNT";
```

Tipos de datos (cont)

- **Ámbito**: conjunto de código donde una variable es visible.

- La **visibilidad** de una variable fuera de su ámbito se puede modificar anteponiendo ***public*** o ***private*** en la declaración.

```
using System;
namespace ConsoleAppVisibilitat
{
    class Simple
    {
        public int Var1=10;
        private int Var2=20; //local
        public int Var3=30;
    }
    class Class1
    {
        [STAThread]
        static void Main(string[] args)
        {
            Simple s = new Simple();

            Console.WriteLine("Simple:");
            Console.WriteLine("Var1={0},
                Var3={1}",s.Var1, s.Var3);

            Console.ReadLine();
        }
    }
}
```

Microsoft C#.net

Tipos de datos (cont)

- **static:** Variable estática, existe durante toda la ejecución del programa.
  - Sólo existe una sola copia de la misma sin necesidad de crear objeto alguno.
  - Hay que referirse a ella usando el nombre completo del tipo al que pertenece.
  - No afecta a su visibilidad

```
using System;
namespace ConsoleAppVisibilitatSTAT  {
    class Simple   {
          public static int Var3=30; //pública i
                                      estàtica
          static int Var4=40; //privada i estàtica
          public void mostraVar4()
          {
             Console.WriteLine("Var4={0}", ++Var4);
          }
    }
    class Class1   {
      [STAThread]
      static void Main(string[] args)  {
          Simple s = new Simple();
          Simple k = new Simple();

          Console.WriteLine("Simple:");
          Console.WriteLine("Var3={0}",Simple.Var3++);
          Console.WriteLine("Var3={0}",Simple.Var3);
          s.mostraVar4();
          k.mostraVar4();
          Console.ReadLine();
      }
    }
}
```

Microsoft C#.net

Tipos de datos (cont)

- **Constantes**
  El modificador *const* permite crear constantes de programas:

```
Class Class1 {
 private const int min = 1;
 private const int max = 100;
 public const int rango = max – min;
 …
 static void Main()
 {
   Console.WriteLine(Class1.max,Class1.min,Class1.rango);
   // también Console.WriteLine(max, min, rango);
 }
}
```

Tipos de datos

- **Tipos definidos por el usuario: Enumeraciones**
  - Definición de una enumeración:

```
enum Color { Rojo, Verde, Azul }
```

  - Uso de una enumeración:

```
Color colorPaleta = Color.Rojo;
```

o

```
colorPaleta = (Color)0; // Tipo int a Color
```

  - Visualización de una variable de enumeración:

```
Console.WriteLine("{0}", colorPaletta);
// Muestra Rojo
```

Tipos de datos (cont)

- **Tipos definidos por el usuario: Estructuras**
  - Definición:

```
struct Empleado {
    public string nombre;
    public int edad;
}
```

  - Uso:

```
Empleado empresaEmpleado, otroEmpleado;
empresaEmpleado.Nombre = "Juan";
empresaEmpleado.edad = 23;
otroEmpleado = empresaEmpleado; //asignación directa
```

  - Comparación: método **Equals()**, no usar operador ==

Tipos de datos (cont)

- **Conversión entre tipos de datos**

Conversión implícita
- Ocurre de forma automática
- Siempre tiene éxito
- Nunca se pierde información en la conversión

Conversión explícita
- Requiere la realización del **cast**
- Puede no tener éxito
- Puede perderse información en la conversión

```
int intValor = 123;
long longValor = intValor;   // implícita de int a long
intvalor = (int) longValor;  // explícita de long a int con cast

int x = 123456;
long y = x;                  // implícita
short z = (short)x;          // explícita
double d = 1.2345678901234;
float f = (float)d;          // explícita
long l = (long)d;            // explícita
```

Microsoft C#.net

Operadores y expresiones

- Operadores aritméticos:
  - +, Suma unaria , +a
  - -, Resta unaria, -a
  - ++, Incremento, ++a o a++
  - --, Decremento, --a o a--
  - +, Suma, a+b
  - -, Resta, a-b
  - *, Multiplicación, a*b
  - /, División, a/b
  - %, Resto, a%b
- Operadores de manipulación de bits:
  - `int i1=32;`
  - `int i2=i1<<2; // i2==128`
  - `int i3=i1>>3; // i3==4`
- Operadores relacionales:
  - ==, Igualdad , a==b
  - !=, Inigualdad, a!=b
  - <, Menor que, a<b
  - <=, Menor o igual, a<=b
  - >, Mayor que, a>b
  - >=, Mayor que o Igual a, a>=b
  - !, Negación , !a
  - &, And binario, a&b
  - |, Or binario, a|b
  - ^, Or exclusivo, a^b
  - ~, Negación binaria, ~a
  - &&, And lógico, a&&b
  - ||, Or lógico, a||b

Microsoft C#.net

Operadores y expresiones (cont)

- Otros operadores
  - min=a<b **?** a**:**b; // equivale a: if a<b min=a else min=b;
  - **.** € para acceso a miembros, e.j. args.Length
  - **()** € para conversión de tipos
  - **[]** € como índice de arrays, punteros, propiedades y atributos
  - **new** € para crear nuevos objetos
  - **typeof** € para obtener el tipo de un objeto
  - **is** € para comparar el tipo de un objeto en runtime
  - **sizeof** € para obtener el tamaño de un tipo en bytes
  - * € para obtener la variable a la que apunta un puntero
  - **->**, p->m es lo mismo que (*).m
  - **&** € devuelve la dirección de un operando
- Las expresiones en C# son similares a C y C++

Operadores y expresiones (cont)

- Precedencia

+	Operadores	Asociatividad	Tipo
	`()`	Izquierda a derecha	Paréntesis
	`*  /  %`	Izquierda a derecha	Multiplicativos
	`+  -`	Izquierda a derecha	Adición
	`<  <=  >  >=`	Izquierda a derecha	Relacionales
	`==  !=`	Izquierda a derecha	Igualdad
-	`=`	Derecha a izquierda	Asignación

- Asociatividad
  - Todos los operadores binarios, salvo los de asignación, son asociativos por la izquierda
  - Los operadores de asignación y el operador condicional(?) son asociativos por la derecha

Microsoft C#.net

Tipos de datos (cont)

- **Arrays**
  - Los arrays son tipos por referencia
  - Sus índices comienzan en 0
  - Derivados de System.Array
    ejemplo: `string[] a;`
  - El tipo de datos viene dado por string[], el nombre del array es una referencia al array
  - Para crear espacio para los elementos usar:
    ```
    string[] a = new string[100];
    ```
  - Los arrays se pueden inicializar directamente:
    ```
    string[] animales = {"gato", "perro", "caballo"};
    int[] a2 = {1, 2, 3};
    ```
  - Puede haber arrays multidimensionales :
    ```
    string[,] ar = {{"perro","conejo"}, {"gato","caballo"}};
    ```

Tipos de datos (cont)

- El rango de los elementos del array es dinámico:
  - Si se crea un nuevo array sobre el mismo se libera la memoria que ocupaba y se pierde toda la información que contenia.
- Información sobre un array:
  - Dimensiones: `Rank`
  - Número de elementos: `GetLength()`
  - Índice superior e inferior: `GetLowerBound(d); GetUpperBound(d)`

    (d=dimensión, desde 0 hasta Rank-1)
  - Saber si es un array: `if (a is Array) ....`
- Recorrido de los elementos de un array sin conocer sus índices

```
foreach (string a in animales)   Console.WriteLine(a);
```

- Otras operaciones: `Clone(); Copy(); Sort();`

Microsoft C#.net

Tipos de datos (cont)

- **Caracteres y cadenas**
  - Dos tipos para manipular caracteres: **char** y **string**
  - **char** puede contener cualquier carácter Unicode (16 bits)
  - Manipulacion de caracteres: `IsDigit(); IsLetter(); IsPunctuation(); ToUpper(); ToLower(), ToString();`...
  - Una variable tipo **string** es una referencia al lugar donde se guarda la cadena.
  - Cada vez que se modifica el valor de la cadena se asigna un nuevo bloque de memoria y se libera el anterior.
  - Concatenación: operador **+** (no existe en C ,C++) o usar Concat():

```
string a, b;
a="Programación ";
b="con C#";
Console.WriteLine("Usando +: {0}", a+b);
Console.WriteLine(" Usando concat: {0}", string.Concat(a, b));
```

Microsoft C#.net

Estructuras de control

- **Instrucciones**
  - Pueden ocupar más de una línea y deben terminarse con un **;**

```
int i, j;
i=1;
```

  - Grupos de instrucciones se pueden agrupar en bloques con **{** y **}**

```
{
   j=2;
   i=i+j;
}
```

  - Un bloque y su bloque padre o pueden tener una variable con el mismo nombre

```
{
      int i;
      ...
      {
            int i;
            ...
      }
}
```

  - Bloques hermanos pueden tener variables con el mismo nombre

```
{
      int i;
      ...
}
      ...
{
      int i;
      ...
}
```

Microsoft C#.net

Estructuras de control (cont)

- **Condicionales**

```
if (<condición>)
   <sentenciasCondTrue>

if (<condición>)
{
   <sentenciasCondTrue>
[else
{
   <sentenciasCondFalse>
} ]
```

Ejemplo:

```
if (a>b) Mayor=a;

if (a>b)
{
   Mayor=a;
   Menor=b;
}
else
{
   Mayor=b;
   Menor=a;
}
```

Microsoft C#.net

Estructuras de control (cont)

- **Ejemplo if en cascada**

```
enum Palo { Treboles, Corazones, Diamantes, Picas}
Palo cartas = Palo.Corazones;
string color;

if (cartas == Palo.Treboles)
    color = “Negro”;
else if (cartas == Palo.Corazones)
    color = “Rojo";
else if (palo == Palo.Diamantes)
    color = "Rojo";
else
    color = “Negro";
```

Estructuras de control (cont)

- **Condicionales múltiples**

```
switch (<expressión>)
{
   case Opc1 :
     [<sentencias-1>]
     break;
   [case Opc2 :
     [<sentencias-2>]
     break;
     …
   default:
     <sentencias-def>
     break;]
}
```

Ejemplo:

```
switch(B)
{
      case 5:
            Console.WriteLine("B es óptimo);
            A=10;
            break;
      case 1:
      case 2:
      case 3:
      case 4:
            Console.WriteLine("B está por
            debajo del valor óptimo);
            A=3;
            break;
      default:
             Console.WriteLine("B no es válido);
            break;
}
```

Microsoft C#.net

Estructuras de control (cont)

- **De repetición**
  - **for**

```
for (int i=0; i < 5; i++) {    // i declarada dentro del bucle
   Console.WriteLine(i);
}

for (;;) {
   ... // bucle infinito
}

for (int i=1, j=2; i<=5; i++, j+=2) {  //múltiples expresiones
      System.Console.WriteLine("i=" + i + ", j=" + j);
}
```

Microsoft C#.net

Estructuras de control (cont)

- **while**
  - Ejecuta instrucciones en función de un valor booleano
  - Evalúa la expresión booleana al principio del bucle

```
while (true) {
    ...
}
```

```
int i = 10;
while (i > 5) {
    ...
    i--;
}
```

```
int i = 0;
while (i < 10) {
    Console.WriteLine(i);
    i++;
}
```

```
0 1 2 3 4 5 6 7 8 9
```

Microsoft C#.net

Estructuras de control (cont)

- **do**
  - Ejecuta instrucciones en función de un valor booleano
  - Evalúa la expresión booleana al final del bucle

```
do {
    ...
} while (true);
```

```
int i = 10;
do {
    ...
    i--;
}
while (i > 5);
```

```
int i = 0;
do {
    Console.WriteLine(i);
    i++;
} while (i <= 10);
```

0 1 2 3 4 5 6 7 8 9 10

Microsoft C#.net

Estructuras de control (cont)

- **Instrucciones de salto**
  - **continue**: Salta el resto de la iteración y comienza la siguiente

```
for (int i=1; i<=5; i++) {
    if (i==3)
        continue;
    Console.WriteLine(i);
}
```

  - **break** : Permite salir de un bucle:

```
for (int i=1; i<=5; i++) {
    if (i==3)
        break;
    Console.WriteLine(i);
}
```

  - **foreach**: ejecuta instrucciones para cada elemento de una colección

```
public static void Main(string[] args) {
     foreach (string s in args)
         Console.WriteLine(s);  //muestra las cadenas de args una a una
}
```

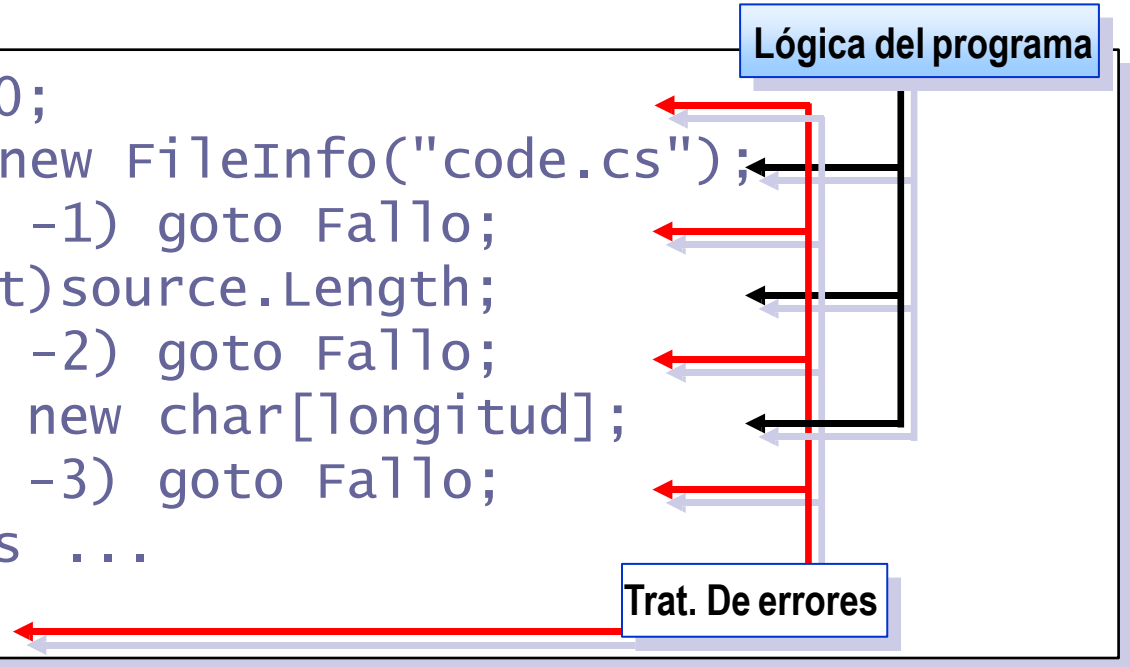
  - **return [<expresión>]:** Sale del método actual
  - **throw :** lanza una excepción

Microsoft C#.net

Estructuras de control (cont)

- **Excepciones**
  - Las excepciones son el mecanismo de C# para controlar las situaciones de error.
  - **¿por qué usar excepciones?**
  - El tradicional tratamiento procedural de errores es demasiado complicado:

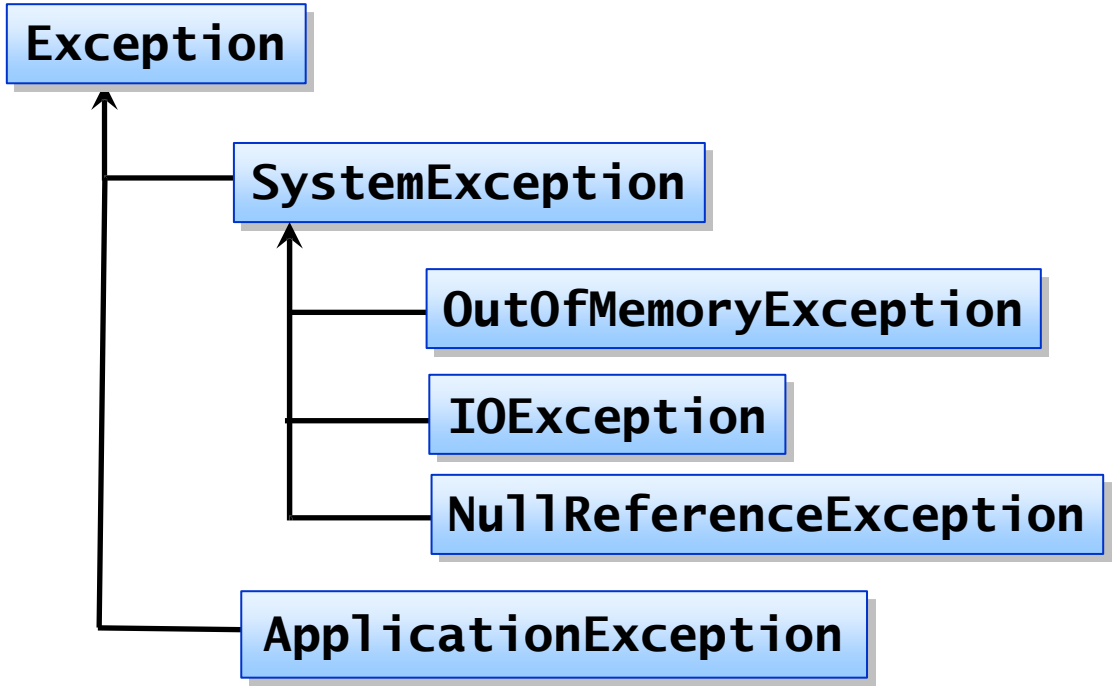
```
int errorCodigo = 0;
FileInfo source = new FileInfo("code.cs");
if (errorCodigo == -1) goto Fallo;
int longitud = (int)source.Length;
if (errorCodigo == -2) goto Fallo;
char[] contenido = new char[longitud];
if (errorCodigo == -3) goto Fallo;
// No hay problemas ...
Fallo: ...
```



Microsoft C#.net

Estructuras de control (cont)

- Todas las excepciones derivan de System.Exception



Microsoft C#.net

Estructuras de control (cont)

- Tratamiento de excepciones orientado a objetos :

```
try {
        ...     // bloque normal de código
}
catch {
        ...     // bloque que controla la excepción
}[ finally {
        ...     // bloque de finalización que siempre se ejecuta
}]
```

- Ejemplo:

```
try {
  Console.WriteLine("Escriba un número");
  int i = int.Parse(Console.ReadLine());
}
catch (OverflowException capturada)
{
  Console.WriteLine(capturada);
}
```

Microsoft C#.net

Estructuras de control (cont)

- Cada bloque *catch* captura una clase de excepción
- Un bloque *try* puede tener un bloque *catch* general que capture excepciones no tratadas (uno solo y el último de los bloques *catch*)
- Un bloque *try* no puede capturar una excepción derivada de una clase capturada en un bloque catch anterior

```
…
try
{
  Console.WriteLine(“Escriba el primer número");
  int i = int.Parse(Console.ReadLine());
  Console.WriteLine("Escriba el segundo número");
  int j = int.Parse(Console.ReadLine());
  int k = i / j;
}
catch (OverflowException capturada) {
  Console.WriteLine(capturada); }
catch (DivideByZeroException capturada)
  {Console.WriteLine(capturada); }
catch {…}  // también: catch (Exception x) { … }
...
```

Microsoft C#.net

Estructuras de control (cont)

- **Funciones y métodos**
  - En C# todo son funciones, no existen procedimientos.
  - Todas las funciones siempre pertenecen a una clase, luego son todas las funciones son métodos.
  - Los métodos, por defecto, son privados (**private**)
  - Main es un método y para definir métodos propios se usa la misma sintaxis:

```
using System;
class EjemploClase
{
    static void EjemploMetodo( )
    {
        Console.WriteLine("Este es un ejemplo de método");
    }
    static void Main( )
    {
        // ...
    }
}
```

Microsoft C#.net

Estructuras de control (cont)

- Una vez definido un método, se puede:
  - Llamar a un método desde dentro de la misma clase
    - Se usa el nombre del método seguido de una lista de parámetros entre paréntesis
  - Llamar a un método que está en una clase diferente
    - Hay que indicar al compilador cuál es la clase que contiene el método que se desea llamar
    - El método llamado se debe declarar con la palabra clave **public**
  - Usar llamadas anidadas
    - Unos métodos pueden hacer llamadas a otros, que a su vez pueden llamar a otros métodos, y así sucesivamente.

Estructuras de control (cont)

- Variables locales
  - Se crean cuando comienza el método
  - Son privadas para el método
  - Se destruyen a la salida
- Variables compartidas
  - Para compartir se utilizan variables de clase
- Conflictos de ámbito
  - El compilador no avisa si hay conflictos entre nombres locales y de clase.

Estructuras de control (cont)

- Devolución de valores
  - El método se debe declarar con un tipo que no sea void
  - Se añade una instrucción return con una expresión
    - Fija el valor de retorno
    - Se devuelve al llamador
  - Los métodos que no son void deben devolver un valor

```
static int DosMasDos( ) {
    int a,b;
    a = 2;
    b = 2;
    return a + b;
}
```

```
int x;
x = DosMasDos( );
Console.WriteLine(x);
```

Microsoft C#.net

Estructuras de control (cont)

- Parámetros
  - Declaración de parámetros
    - Se ponen entre paréntesis después del nombre del método
    - Se definen el tipo y el nombre de cada parámetro
  - Llamadas a métodos con parámetros
    - Un valor para cada parámetro
  - Paso por valor, por referencia y parámetros de salida

```
static void  MetodoConParametros(int n, string y)
{ ... }

MetodoConParametros(2, "Hola, mundo");
```

Microsoft C#.net

Estructuras de control (cont)

- Paso por valor
  - Se copia el valor del parámetro
  - Se puede cambiar el nombre de la variable dentro del método
  - No afecta al valor fuera del método
  - El parámetro debe ser de un tipo igual o compatible

```
static void SumaUno(int x)
{
        x++; // Incrementar x
}
static void Main( )
{
        int k = 6;
        SumaUno(k);
        Console.WriteLine(k); // Muestra el valor 6, no 7
```

Estructuras de control (cont)

- Paso por referencia
  - Se pasa una referencia a una posición de memoria
  - Se usa la palabra clave **ref** en la declaración y las llamadas al método
  - Los tipos y valores de variables deben coincidir
  - Los cambios hechos en el método afectan al llamador
  - Hay que asignar un valor al parámetro antes de la llamada al método

```
static int RaizCuarta(ref int x)
{
        x = (int)Math.Sqrt(x);
        return (int)Math.Sqrt(x);
}
static void Main( ) {
        int num = 625; //siempre inicializar!!
        Console.WriteLine("Raiz cuarta: {0}, el número
        original es: {1}", RaizCuarta(ref num), num);
}
```

Microsoft C#.net

Estructuras de control (cont)

- Parámetros de salida
  - Pasan valores hacia fuera, pero no hacia dentro
  - No se pasan valores al método
  - Se usa la palabra clave **out** en la declaración y las llamadas al método

```
static void OutDemo(out int p)
{
      // ...
}
static void Main( ) {
      int n;
      OutDemo(out n);
}
```

Microsoft C#.net

Estructuras de control (cont)

- Lista de parámetros de longitud variable
  - Se usa la palabra clave **params**
  - Se declara como array al final de la lista de parámetros
  - Siempre paso por valor

```
static long SumaLista(params long[ ] v)
{
    long total, i;
    for (i = 0, total = 0; i < v.Length; i++)
        total += v[i];
    return total;
}
static void Main( )
{
    long x = SumaLista(63,21,84);
}
```

Microsoft C#.net

Estructuras de control (cont)

- Normas para el paso de parámetros:
  - El paso por valor es el más habitual y suele ser el más eficaz
  - El valor de retorno del método es útil para un solo valor
  - **ref** y/o **out** son útiles para más de un valor de retorno
  - **ref** sólo se usa si los datos se pasan en ambos sentidos
- Devolución de arrays desde métodos

```
class EjemploArr1 {
    static void Main( ) {
        int[ ] array = CreateArray(42);
        ...
    }
    static int[ ] CreateArray(int tamano) {
        int[ ] creada = new int[tamano];
        return creada;
    }
}
```

Microsoft C#.net

Estructuras de control (cont)

- Paso de arrays como parámetros
  - Un parámetro de array es una copia de la variable de array
  - No es una copia del array

```
class EjemploArr2 {
    static void Main( )
   {
        int[ ] arg = {10, 9, 8, 7};
        Method(arg);
        System.Console.WriteLine(arg[0]);
    }

    static void Metodo(int[ ] parametro) {
        parametro[0]++;
    }
}
```

Este método modificará el array original creado en Main

Microsoft C#.net

Estructuras de control (cont)

- Métodos recursivos
  - Hacen llamadas a sí mismos
  - Útil para resolver ciertos problemas

```
static ulong Fibonacci(ulong n)
{
   if (n <= 2)
      return 1;
   else
      return Fibonacci(n-1) + Fibonacci(n-2);
}
```

Estructuras de control (cont)

- Métodos sobrecargados (*overloading*)
  - Comparten un nombre en una clase.
  - Se distinguen examinando la lista de parámetros.
  - Usarlos cuando hay métodos similares que requieren parámetros diferentes o si se quiere añadir funcionalidad al código existente.
  - No abusar pues son difíciles de mantener y de depurar.

```
class OverloadingExample
{
    static int Suma(int a, int b)
    {
        return a + b;
    }
    static int Suma(int a, int b, int c)
    {
        return a + b + c;
    }
    static void Main( )
    {
        Console.WriteLine(Suma(1,2) + Suma(1,2,3));
    }
}
```

Microsoft C#.net

Bibliografía

- **C#. Curso de Programación**.
  - Autor: Fco. Javier Ceballos Sierra.
  - Editorial:
    - **RA-MA** en España.
    - **Alfaomega Grupo Editor** en América.
- **Enciclopedia de Microsoft Visual C#.**
  - Autor: Fco. Javier Ceballos Sierra.
  - Editorial:
    - **RA-MA** en España.
    - **Alfaomega Grupo Editor** en América.

Microsoft .net