Tema 4.
Introducción a C#

Formación específica, cursos verano 2008
ETS de Informática Aplicada
Universidad Politécnica de Valencia

© Juan Carlos Ruiz, ITACA-UPV, 2008

Objetivos

- Describir la estructura básica de un programa C#
- Describir los aspectos básicos del lenguaje de programación C#
- Utilizar el Visual Studio .NET para el desarrollo, depuración y ejecución de aplicaciones C#

© Juan Carlos Ruiz, ITACA-UPV, 2008

Índice

- Aspectos básicos
- Variables, tipos y operadores
- Instrucciones
- Excepciones
- Arrays
- Métodos

© Juan Carlos Ruiz, ITACA-UPV, 2008

Índice

- Aspectos básicos
- Variables, tipos y operadores
- Instrucciones
- Excepciones
- Arrays
- Métodos

© Juan Carlos Ruiz, ITACA-UPV, 2008

Características de C#

- Sencillez
- Modernidad
- Orientación a objetos
- Orientación a componentes
- Gestión automática de memoria
- Seguridad de tipos
- Instrucciones seguras
- Sistema de tipos unificado
- Extensibilidad de tipos básicos
- Extensibilidad de operadores
- Extensibilidad de modificadores
- Versionable
- Eficiente
- Compatible

© Juan Carlos Ruiz, ITACA-UPV, 2008

Mi primer programa C#

- ¡Hola Mundo!

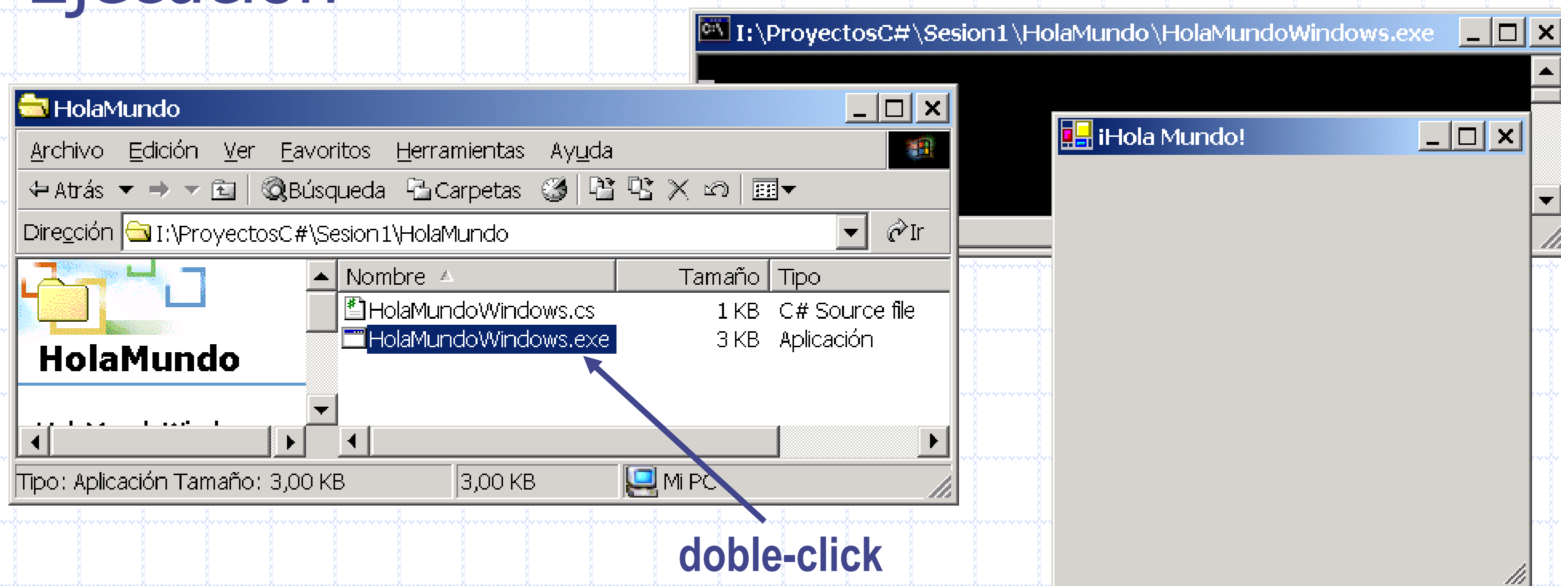
```
using System;
using System.Windows.Forms;

class HolaMundoWindows {
    public static void Main() {
            Form holaForm = new Form();
            holaForm.Text = "¡Hola Mundo!";
            Application.Run(holaForm);
    }
}
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

¡Hola Mundo!

- Compilación
  - Símbolo de sistema de Visual Studio .NET

    `csc HolaMundoWindows.cs`
- Ejecución



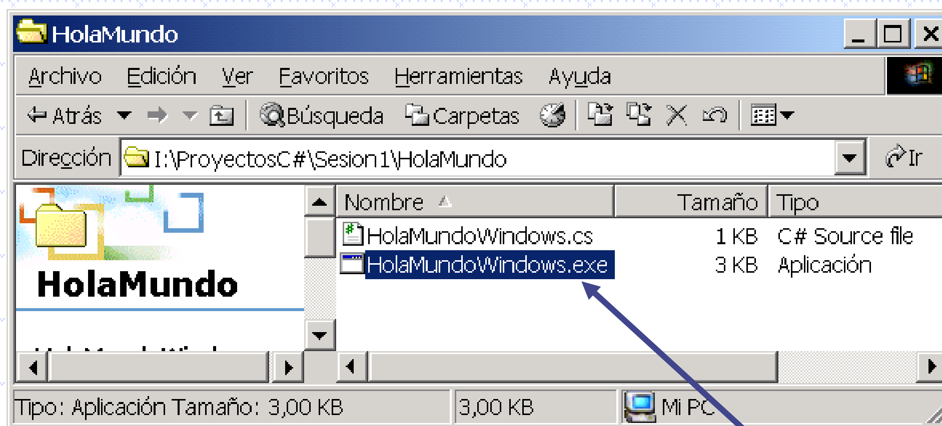
© Juan Carlos Ruiz, ITACA-UPV, 2008

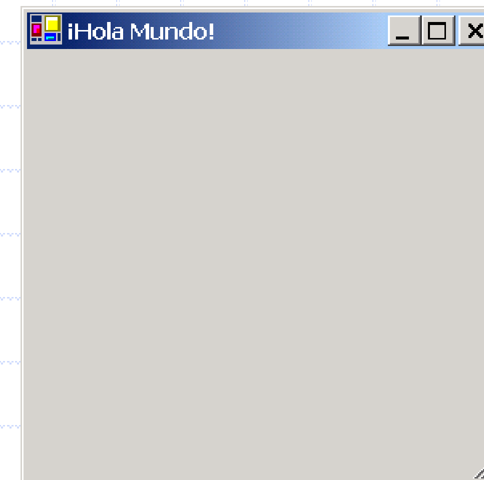
¡Hola Mundo!

- Compilación
  - Símbolo de sistema de Visual Studio .NET

```
csc /target:winexe HolaMundoWindows.cs
```

- Ejecución





doble-click

© Juan Carlos Ruiz, ITACA-UPV, 2008

¡Hola Mundo!

- ILDASM
  - Abrir el fichero HolaMundoWindows.exe

```
I:\ProyectosC#\Sesion1\HolaMundo\HolaMundoWindows.exe - IL DASM
Archivo  Ver  Ayuda
I:\ProyectosC#\Sesion1\HolaMundo\HolaMundoWindows.exe
  MANIFEST
  HolaMundoWindows
    .class private auto ansi beforefieldinit
    .ctor : void()
    Main : void()
.ver 0:0:0:0
```

```
HolaMundoWindows::Main : void()
.method public hidebysig static void  Main() cil managed
{
  .entrypoint
  // Code size       25 (0x19)
  .maxstack  2
  .locals init (class [System.Windows.Forms]System.Windows.Forms.Form V_0)
  IL_0000:  newobj     instance void [System.Windows.Forms]System.Windows.Forms.Form::.ctor()
  IL_0005:  stloc.0
  IL_0006:  ldloc.0
  IL_0007:  ldstr      bytearray (A1 00 48 00 6F 00 6C 00 61 00 20 00 4D 00 75 00   // ..H.o.l.a. .M.u.
                                  6E 00 64 00 6F 00 21 00 )                         // n.d.o.!.
  IL_000c:  callvirt   instance void [System.Windows.Forms]System.Windows.Forms.Control::set_Text(string)
  IL_0011:  ldloc.0
  IL_0012:  callvirt   instance valuetype [System.Windows.Forms]System.Windows.Forms.DialogResult [System.
  IL_0017:  pop
  IL_0018:  ret
} // end of method HolaMundoWindows::Main
```

```
MANIFEST
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 1:0:5000:0
}
.assembly extern System.Windows.Forms
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 1:0:5000:0
}
.assembly HolaMundoWindows
{
  // --- The following custom attribute is added
  //  .custom instance void [mscorlib]System.Diag
  //
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module HolaMundoWindows.exe
// MVID: {CB0A05C2-9BFA-42CC-A4E2-8C3E94FFFCB7}
.imagebase 0x00400000
.subsystem 0x00000003
.file alignment 512
.corflags 0x00000001
// Image base: 0x03c20000
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

La clase

- Toda aplicación C# es un conjunto de clases, estructuras y tipos de datos
- Una clase es un conjunto de propiedades y métodos
- Sintaxis

```
class nombre
{
    ...
}
```

- Una clase está contenida en un único fichero
- Un fichero puede contener múltiples clases

© Juan Carlos Ruiz, ITACA-UPV, 2008

El método Main

- Es el punto de entrada de la aplicación

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

© Juan Carlos Ruiz, ITACA-UPV, 2008

Espacios de nombres

- Existe un gran número de clases organizadas por espacios de nombres
- Hacer referencia a una clase por su espacio de nombres

```
System.Windows.Forms.Form holaForm =
    new System.Windows.Forms.Form();
```

- Utilizando la directiva *using*

```
using System.Windows.Forms;
...
Form holaForm = new Form();
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Espacios de nombres

HolaMundoConsola - Microsoft Visual C# .NET [diseñar] - Examinador de objetos

Archivo Edición Ver Proyecto Generar Depurar Herramientas Ventana Ayuda

Examinador de objetos | Class1.cs*

Examinar: Componentes seleccionados | Personalizar...

Objetos:
- BitConverter
- Boolean
- Buffer
- Byte
- CannotUnloadAppDomainException
- Char
- CharEnumerator
- CLSCompliantAttribute
- Console
- ContextBoundObject
- ContextMarshalException
- ContextStaticAttribute
- Convert
- CrossAppDomainDelegate
- DateTime
- DayOfWeek
- DBNull
- Decimal
- Delegate
- DivideByZeroException
- DllNotFoundException
- Double
- DuplicateWaitObjectException

Miembros de 'Console':
- Write(char[],int,int)
- Write(char[])
- Write(char)
- Write(bool)
- Write(string,params object[])
- Write(string,object,object,object,object)
- Write(string,object,object,object)
- Write(string,object,object)
- Write(string,object)
- WriteLine(string,params object[])
- WriteLine(string,object,object,object,object)
- WriteLine(string,object,object,object)
- WriteLine(string,object,object)
- WriteLine(string,object)
- WriteLine(string)
- WriteLine(object)
- WriteLine(ulong)
- WriteLine(long)
- WriteLine(uint)
- WriteLine(int)
- WriteLine(float)
- WriteLine(double)
- WriteLine(decimal)

public static void **WriteLine** (System.String *value*)
Miembro de System.Console

Resumen:
Escribe el valor de cadena especificado, seguido del terminador de línea actual, en la secuencia de salida estándar.

Parámetros:
value: Valor que se va a escribir.

Excepciones:
System.IO.IOException: Error de E/S.

Explorador de soluciones - H...
- Solución 'HolaMundoConsola' (1 proy
- HolaMundoConsola
  - References
    - System
    - System.Data
    - System.Windows.Forms
    - System.XML
  - App.ico
  - AssemblyInfo.cs
  - Class1.cs

Propiedades: WriteLine CodeFunction

(Name)	WriteLine
Access	public
CanOverride	False
FullName	System.Console.Writ
IsOverloaded	True
IsShared	True
MustImplement	False

(Name)
Sets/returns the name of the object.

Resultados — Depurar

Lista de tareas | Resultados | Resultados de la búsqueda de biblioteca: 500 temas encontrados

Listo

© Juan Carlos Ruiz, ITACA-UPV, 2008

Comentarios

- Proporcionan documentación adecuada para determinadas secciones de código
- Comentarios de una línea

```
// Esto es un comentario de una sola línea
```

- Comentarios de varias líneas

```
/* Esto es un comentario
     que se extiende
     a lo largo
     de varias líneas */
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Índice

- Aspectos básicos
- Variables, tipos y operadores
- Instrucciones
- Excepciones
- Arrays
- Métodos

© Juan Carlos Ruiz, ITACA-UPV, 2008

Variables

- Disponen de un determinado tipo de datos que indica qué valores puede contener

- Variables tipo valor
  - Almacenan los datos directamente
  - Cada una dispone de su propia copia de los datos
  - Las operaciones realizadas sobre una **no** afectan a las demás

- Variables tipo referencia
  - Almacenan referencias a los datos (objetos)
  - Diferentes variables pueden referenciar al mismo objeto
  - Las operaciones realizadas sobre una **pueden** afectan a las demás

© Juan Carlos Ruiz, ITACA-UPV, 2008

Tipos valor

- Todos derivan de *System.ValueType*
  - Tipos predefinidos o tipos básicos o tipos simples
  - Tipos definidos por el usuario
    - *struct*
    - *enum*
- Todos almacenan directamente sus datos y no pueden ser *null* (excepto *string* y *object*)
- Los tipos predefinidos pueden contener un valor literal

© Juan Carlos Ruiz, ITACA-UPV, 2008

Tipos predefinidos/básicos/simples

- Se identifican por palabras reservadas que son alias de tipos *struct* predefinidos

Tipo	Descripción	Bits	Rango de valores	Alias
System.SByte	*Bytes* con signo	8	[-128, 127]	sbyte
System.Byte	*Bytes* sin signo	8	[0, 255]	byte
System.Int16	Enteros cortos con signo	16	[-32768, 32767]	short
System.UInt16	Enteros cortos sin signo	16	[0, 65535]	ushort
System.Int32	Enteros normales	32	[-2147483648, 2147483647]	int
System.UInt32	Enteros normales sin signo	32	[0, 4294967295]	unit
System.Int64	Enteros largos	64	[-9223372036854775808, 9223372036854775807]	long
System.UInt64	Enteros largos sin signo	64	[0, 18446744073709551615]	ulong
System.Single	Reales con 7 dígitos de precisión	32	[-1.5×10^{-45}, 3.4×10^{38}]	float
System.Double	Reales con 15-16 dígitos de precisión	64	[-5.0×10^{-324}, 1.7×10^{308}]	double
System.Decimal	Reales con 28-29 dígitos de precisión	128	[-1.0×10^{-28}, 7.9×10^{28}]	Decimal
System.Boolean	Valores lógicos	32	true, false	Bool
System.Char	Caracteres Unicode	16	['\u0000', '\uFFFF']	char
System.String	Cadenas de caracteres	Variable	El permitido por la memoria	string
System.Object	Cualquier objeto	Variable	Cualquier objeto	object

© Juan Carlos Ruiz, ITACA-UPV, 2008

Identificadores

- Nombre con el que se identifica un elemento del código
- Reglas
  - Deben comenzar por carácter alfanumérico (excepto dígitos) o *underscore*
  - El resto de caracteres pueden ser caracteres alfanuméricos o *underscores*
  - No utilizar palabras reservadas

`C3PO` ✓ `2horas` ✗ `a!b` ✗ `A°B` ✓

`Älíêñ` ✓ `C#` ✗ `_hola_` ✓ `__OK` ✓

© Juan Carlos Ruiz, ITACA-UPV, 2008

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 is lock 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
volatile while

© Juan Carlos Ruiz, ITACA-UPV, 2008

Declaración de variables

- Declaración

```
tipoVariable nombreVariable;

int a;
int x, y, z;
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Asignación de valores a variables

- No es posible utilizar variables sin inicializar
- Asignación de valores a variables ya declaradas

```
nombreVariable = valor;

int x;
x = 10;
int a = 20;
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Literales

- Los literales son la representación explícita de los valores que puede adoptar los tipos básicos de datos
- Literales enteros
  - Decimal (0-9)
  - Hexadecimal (0-9, a-f, A-F) con prefijo (0x)
  - Operadores positivo (+) y negativo (-)

```
0    -12    +34    0x5A6        -0xB78
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Literales

- Literales reales
  - Como los enteros decimales
  - Parte entera y real separada por punto decimal (.)
  - Notación científica, exponente (e, E)

```
0.0  -12.5  +34.8  56.3e-2   7.88E5
```

- Literales lógicos

```
true      false
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Literales

- Literales de carácter
  - Cualquier carácter entre comillas simples (‘x’), excepto

Carácter	Código Unicode	Código de escape
Comilla simple	\u0027	\’
Comilla doble	\u0022	\”
Carácter nulo	\u0000	\0
Alarma	\u0007	\a
Retroceso	\u0008	\b
Salto de página	\u000C	\f
Nueva línea	\u000A	\n
Retorno de carro	\u000D	\r
Tabulación horizontal	\u0009	\t
Tabulación vertical	\u000B	\v
Barra invertida	\u005C	\\

© Juan Carlos Ruiz, ITACA-UPV, 2008

Literales

- Literales de carácter
  - Pueden utilizarse también los códigos Unicode
    - Formato comprimido (\x)
      - No es necesario escribir los ceros a la izquierda
      - Sólo válido para los literales

```
'a' 'G' 'x' '?' '\'' '\f'
'\U00000008' '\u0008' '\x8'
```

- Literal nulo
  - Se utilizan para variables de objeto no inicializadas

```
null
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Literales

- Literales de cadena
  - Secuencia de caracteres entre comillas dobles ("x")
  - Cadenas planas o **verbatim**
    - Preceder la cadena por (**@**)
    - Los códigos de escape no se interpretan

Literal de cadena	Interpretado como …
"Hola\tMundo"	**Hola Mundo**
@"Hola\tMundo"	**Hola\tMundo**
@"Hola Mundo"	**Hola Mundo**
@"""Hola Mundo"""	**"Hola Mundo"**

© Juan Carlos Ruiz, ITACA-UPV, 2008

Operadores

- Un operador es un símbolo que permite realizar una operación entre operandos y devuelve un resultado
- Operadores aritméticos
  - Detección de desbordamiento
    - checked
      - lanza *System.OverflowException*
    - unchecked
      - devuelve el resultado truncado

Operación	Operador
Suma	+
Resta	-
Multiplicación	*
División	/
Módulo	%
Menos unario	-
Más unario	+

```
checked(expresiónAritmética)
unchecked(expresiónAritmética)
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Operadores

- Operaciones lógicas
  - Evaluación perezosa
- Operaciones relacionales
- Operaciones de manipulación de bits

<table>
<tr><th>Operación</th><th>Operador</th></tr>
<tr><td>and</td><td>&& y &</td></tr>
<tr><td>or</td><td>| | y |</td></tr>
<tr><td>not</td><td>!</td></tr>
<tr><td>xor</td><td>^</td></tr>
</table>

<table>
<tr><th>Operación</th><th>Operador</th></tr>
<tr><td>and</td><td>&</td></tr>
<tr><td>or</td><td>|</td></tr>
<tr><td>xor</td><td>^</td></tr>
<tr><td>Desplazamiento a izquierda</td><td><<</td></tr>
<tr><td>Desplazamiento a derecha</td><td>>></td></tr>
</table>

Operación	Operador
Igualdad	==
Desigualdad	!=
Mayor que	>
Menor que	<
Mayor o igual que	>=
Menor o igual que	<=

© Juan Carlos Ruiz, ITACA-UPV, 2008

Operadores

- Operaciones de asignación (=)
  - Además de realizar la asignación, devuelve el valor asignado

```
int distancia = 0;
distancia = distancia + 10;
```

  - Asignación compuesta

    (+=, -=, *=, /=, %=, &=, |=, ^=, >>=, <<=)

```
distancia += 10;
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Operadores

- Operaciones de asignación
  - Incremento (++) y decremento(--)
    - Incrementar en 1 es muy común

```
cuenta = cuenta + 1;
cuenta += 1;
cuenta++;
```

    - Existe en dos formas

```
int cuenta = 10;
int a = cuenta++; // a = 10; cuenta = 11;
int b = ++cuenta; // b = 12; cuenta = 12;
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Operadores

- Operaciones con cadenas
  - Concatenación de cadenas (+)

```
string holaMundo1 = “Hola” + “ Mundo”;
string holaMundo2 = “Hola Mund” + ‘o’;
```

- Operaciones de acceso a tablas

tabla[*posiciónElemento*]

© Juan Carlos Ruiz, ITACA-UPV, 2008

Operadores

- Operación condicional

```
condición?expresión1:expresión2
```

  - Si *condición* es cierta devuelve el resultado de evaluar *expresión1*, sino devuelve el resultado de evaluar *expresión2*

```
int cociente=(divisor>0)?(dividendo/divisor):0;
```

  - Operaciones de acceso a objetos

```
objeto.miembro
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Operadores

- Operaciones de obtención de información sobre tipos
  - Objeto *System.Type* con información del tipo

typeof(*nombreTipo***)**

  - Determinar si una expresión es de un tipo determinado

expresión **is** *nombreTipo*

  - Número de bytes que ocupa un objeto de ese tipo en memoria (código inseguro)

sizeof(*nombreTipo***)**

© Juan Carlos Ruiz, ITACA-UPV, 2008

Operadores

- Operaciones de conversión entre tipos

```
(nombreTipo) expresión
```

  - Devuelve *System.InvalidCastException* en caso de conversión inválida

```
expresión as tipoDestino
```

  - Aplicable sólo a tipos referencia con conversiones predefinidas en el lenguaje
  - Devuelve null en caso de conversión inválida

© Juan Carlos Ruiz, ITACA-UPV, 2008

Tipos definidos por el usuario: Enumeración

- Estructura que define los valores literales que pueden tomar sus objetos
  - Soluciona los problemas de los **números mágicos**

```
enum nombreEnumeración:nombreTipo
{
    literales
}
```

  - El tipo por defecto de los literales es *int*

© Juan Carlos Ruiz, ITACA-UPV, 2008

Tipos definidos por el usuario: Enumeración

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

enum TipoCuenta
{
    Corriente,
    Ahorros,
    Vivienda,
    PlanJubiliacion
}
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Tipos definidos por el usuario: Estructura

- Tipo especial de clase que representa objetos ligeros
- Todos los tipos básicos son estructuras
  - Excepto **string** y **object**

```
struct Alumno
{
    string Nombre;
    int numeroExpediente;
}
Alumno nuevoAlumno;
nuevoAlumno.Nombre = “John Doe”;
nuevoAlumno.numeroExpediente = 123456;
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Índice

- Aspectos básicos
- Variables, tipos y operadores
- Instrucciones
- Excepciones
- Arrays
- Métodos

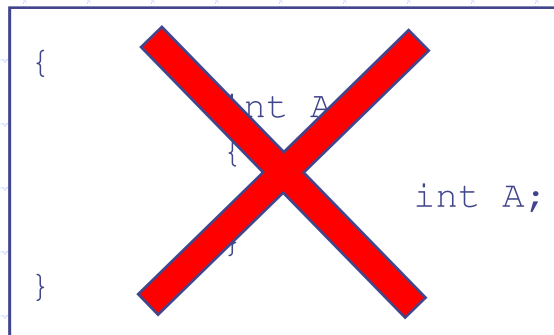
© Juan Carlos Ruiz, ITACA-UPV, 2008

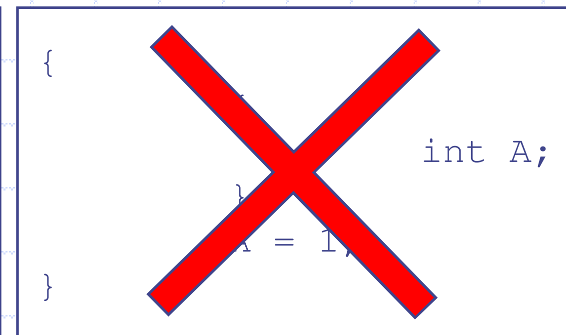
Instrucciones

- Una instrucción es cualquier acción que se pueda realizar en el cuerpo de un método
- Las instrucciones se agrupan en bloques

```
{
    listaInstrucciones
}
```

```
{
        int A;
}

{
        int A;
}
```





© Juan Carlos Ruiz, ITACA-UPV, 2008

Tipos de instrucciones

- Condicionales o de selección (**if** y **switch**)
  - Permiten ejecutar bloques de instrucciones sólo si se cumple una determinada condición
- Iterativas (**while**, **do**, **for** y **foreach**)
  - Permiten ejecutar repetidamente un bloque de instrucciones si se cumple una determinada condición
- De salto (**goto**, **break** y **continue**)
  - Permiten variar el orden normal en el que se ejecutan las instrucciones

© Juan Carlos Ruiz, ITACA-UPV, 2008

Instrucciones condicionales

- Instrucción *if*
  - Permite ejecutar ciertas instrucciones si se da determinada condición

```
if condición
     instruccionesIf
else
     instruccionesElse

if ((carta = Tipo.Treboles) || (carta = Tipo.Picas))
     Console.WriteLine(“Color negro”);
else
     Console.WriteLine(“Color rojo”);
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Instrucciones condicionales

- Instrucción *switch*
  - Permite ejecutar unos bloques de instrucción u otros atendiendo a determinada condición

```
switch(expresión)
{
   case valor1: bloque1
                siguienteAcción
   case valor2: bloque2
                siguienteAcción
   ...
   default:     bloqueDefault
                siguienteAcción
}
```

siguienteAcción puede ser:

```
goto case valorI;
goto default;
break;
```

```
switch(carta) {
   case Tipo.Corazones:
   case Tipo.Diamantes:
         Console.WriteLine("Color rojo");
         break;
   case Tipo.Treboles:
   case Tipo.Picas:
         Console.WriteLine("Color negro");
         break;
   default:
         Console.WriteLine("ERROR");
         break;
}
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Instrucciones iterativas

- Instrucción *while*
  - Permite ejecutar un bloque de instrucciones mientras se de una determinada condición

```
while(condición)
    instrucciones

int i = 0;
while(i < 10) {
    Console.WriteLine(i);
    i++;
}   // 0 1 2 3 4 5 6 7 8 9
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Instrucciones iterativas

- Instrucción *do...while*
  - Variante de *while* que evalúa la condición al final del bucle
  - Las instrucciones se ejecutan por lo menos una vez

```
do
    instrucciones
while (condición);

int i = 0;
do
{
    Console.WriteLine(i);
    i++;
} while(i < 10);  // 0 1 2 3 4 5 6 7 8 9
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Instrucciones iterativas

- Instrucción *for*
  - Variante de *while* que permite reducir el código necesario para los bucles más utilizados

```
for(inicialización;condición;modificación)
     instrucciones

for(int i = 0; i < 10; i++) {
     Console.WriteLine(i);
}    // 0 1 2 3 4 5 6 7 8 9
```

  - *inicialización* y *modificación* pueden incluir diversos valores separados por comas (,)

```
for(int i = 0, j = 10; i < 10; i++, j--) {
     Console.WriteLine("({0} {1})", i, j);
}    //(0 10)(1 9)(2 8)(3 7)(4 6)(5 5)(6 4)(7 3)(8 2)(9 1)
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

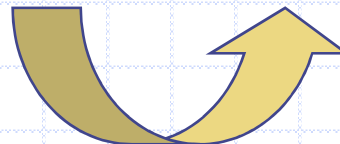
Instrucciones iterativas

- Instrucción *foreach*
  - Variante de *for* para el tratamiento de todos los elementos de una colección

foreach(*tipoElemento elemento* **in** *colección***)**
instrucciones

```
public static void Main(string[] args) {
        for(int i = 0; i < args.Length;
   i++) {
        Console.WriteLine(args[i]);
        }
   }
```

```
public static void Main(string[] args) {
   foreach(string dato in args) {
        Console.WriteLine(dato);
   }
}
```



© Juan Carlos Ruiz, ITACA-UPV, 2008

Instrucciones de salto

- Instrucción *break*
  - Se utiliza en bloques de instrucciones asociados a instrucciones iterativas o a la instrucción *switch*
  - Termina la ejecución del bloque y continúa ejecutando la instrucción siguiente al mismo

```
break;
```

- Instrucción *continue*
  - Se utiliza en bloques de instrucciones asociados a instrucciones iterativas
  - Reevalúa la condición del bucle sin ejecutar el resto de instrucciones del mismo

```
continue;
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Instrucciones de salto

- Instrucción *goto*
  - Pasa a ejecutar la instrucción etiquetada en el *goto*

```
goto etiqueta;
```

  - Sólo se pueden etiquetar instrucciones
  - No se pueden etiquetar instrucciones en bloques anidados ni en métodos diferentes al del *goto*
  - Evitar su uso si es posible

© Juan Carlos Ruiz, ITACA-UPV, 2008

Índice

- Aspectos básicos
- Variables, tipos y operadores
- Instrucciones
- Excepciones
- Arrays
- Métodos

© Juan Carlos Ruiz, ITACA-UPV, 2008

Excepciones

- Preparan los programas para lo inesperado
- Tradicionalmente, los métodos devuelven códigos informativos
  - El código de la lógica del programa y del tratamiento del error están mezclados
  - Todos los códigos de error son parecidos
  - Códigos de error no significativos por sí mismos
  - Los códigos de error se definen a nivel de método
  - Los códigos de error son muy fáciles de ignorar

© Juan Carlos Ruiz, ITACA-UPV, 2008

Excepciones

- Objetos que se generan en tiempo de ejecución cuando se produce algún error
- Derivan de la clase **System.Exception**
- Proporcionan
  - Claridad en el código generado
  - Información acerca del error ocurrido
  - Tratamiento del error asegurado

© Juan Carlos Ruiz, ITACA-UPV, 2008

Excepciones

- Ejemplo de excepciones predefinidas

Excepción	Causa
ArgumentException	Argumento no válido
DivideByZeroException	División por cero
IndexOutOfRangeException	Índice de acceso a un elemento fuera del rango permitido
OverflowException	Desbordamiento
OutOfMemoryException	Falta de memoria al crear un nuevo objeto
StackOverflowException	Desbordamiento de la pila

© Juan Carlos Ruiz, ITACA-UPV, 2008

Captura de excepciones

- Instrucción *try...catch*
  - Separa la lógica del programa del tratamiento del error

```
try
    instrucciones
catch(excepción1)
    tratamiento1
catch(excepción2)
    tratamiento2
...
finally
    instruccionesFinally
```

```
try {
    Console.Write("Dividendo: ");
    int i = Int32.Parse(Console.ReadLine());
    Console.Write("Divisor: ");
    int j = Int32.Parse(Console.ReadLine());
    int k = i/j;
}
catch(OverflowException oe) {
    Console.WriteLine(oe);
}
catch(DivideByZeroException dbze) {
    Console.WriteLine(dbze);
}
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Captura de excepciones

- Bloque *catch* general
  - Puede capturar cualquier excepción independientemente de su clase
  - No proporciona información acerca del error

```
try {
      Console.Write(“Introduce el dividendo: ”);
      int i = Int32.Parse(Console.ReadLine());
      Console.Write(“Introduce el divisor:”);
      int j = Int32.Parse(Console.ReadLine());
      int k = i/j;
}
catch {
      Console.WriteLine(“ERROR EN LOS DATOS INTRODUCIDOS”);
}
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Lanzamiento de excepciones

- Los programadores pueden lanzar las excepciones del sistema o las suyas propias

throw *objetoExcepciónALanzar*;

- Ejemplo

```
Console.Write("Introduce un número del 1 al 10: ");
int i = Int32.Parse(Console.ReadLine());
if ((i < 0) || (i > 10)) {
    throw new NumeroNoValidoException();
}
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Lanzamiento de excepciones

- Desde un bloque *catch* es posible relanzar la excepción que ha sido capturada

```
catch {                          catch {
    ...                               ...
    throw caught;                throw;
}                                }
```

- También es posible lanzar una excepción de un tipo distinto

```
catch (IOException ioe) {
    ...
    throw new FileNotFoundException();
}
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Índice

- Aspectos básicos
- Variables, tipos y operadores
- Instrucciones
- Excepciones
- Arrays
- Métodos

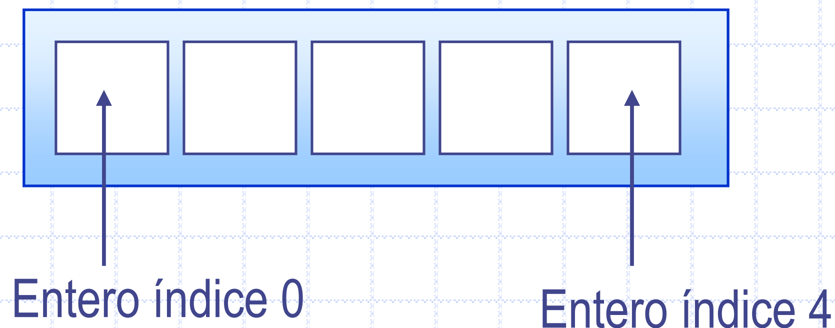
© Juan Carlos Ruiz, ITACA-UPV, 2008

Arrays

- Los *arrays* son secuencias de datos del mismo tipo

```
tipoDatos[] nombreArray;
tipoDatos[] nombreArray = new tipoDatos[númeroDatos];
tipoDatos[] nombreArray = new tipoDatos[númeroDatos]{valores};
```

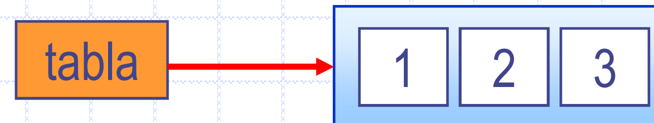
- Los elementos se acceden por medio de índices enteros



© Juan Carlos Ruiz, ITACA-UPV, 2008

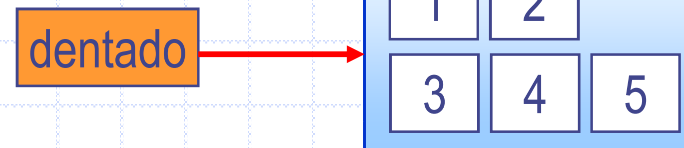
Creación de arrays

- Arrays unidimensionales




```
int[] tabla = new int[]{1,2,3};
int[] tabla = {1,2,3};
```

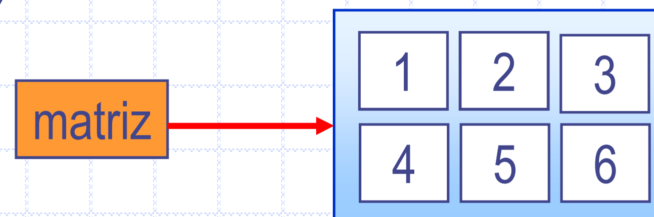
- Arrays dentados




```
int[][] dentado =
     new int[][]{new int[]{1,2},new int[]{3,4,5}};
int[][] dentado = {{1,2},{3,4,5}};
```

- Arrays multidimensionales

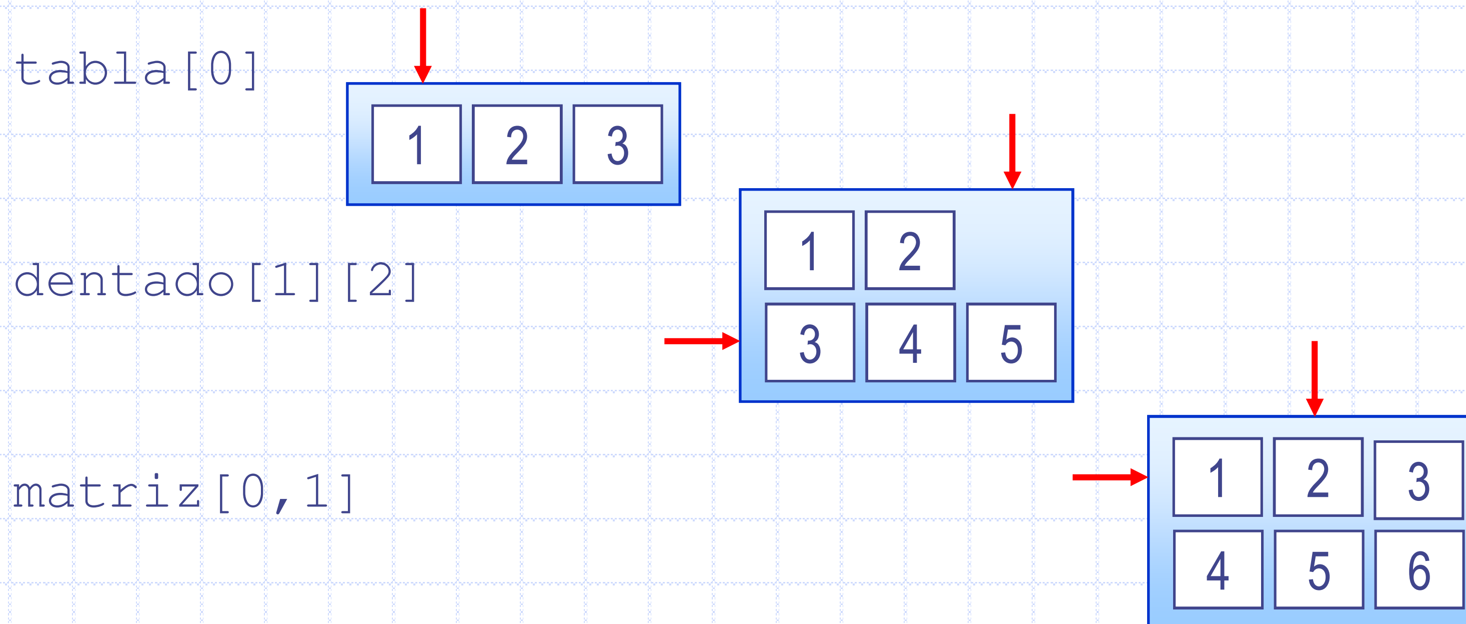
```
int[,] matriz =
     new int[,]{new int[]{1,2,3},new int[]{4,5,6}};
int[,] matriz = {{1,2,3},{4,5,6}};
```



© Juan Carlos Ruiz, ITACA-UPV, 2008

Acceder a elementos del array

- Se proporciona un entero por cada dimensión del array
  - El primer elemento se numera desde cero

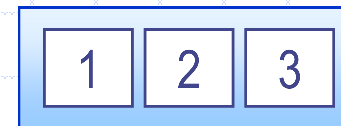


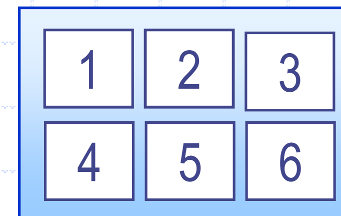
© Juan Carlos Ruiz, ITACA-UPV, 2008

Límites del array

- Se comprueban siempre los accesos fuera de los límites del array
  - Se lanza una excepción *IndexOutOfBounds*
  - Uso de la propiedad *Length* y del método *GetLength*

```
tabla.Rank;               // 1
tabla.GetLength(0);       // 3
tabla.Length;             // 3

matriz.Rank;              // 2
matriz.GetLength(0);      // 2
matriz.GetLength(1);      // 3
matriz.Length;            // 6
```

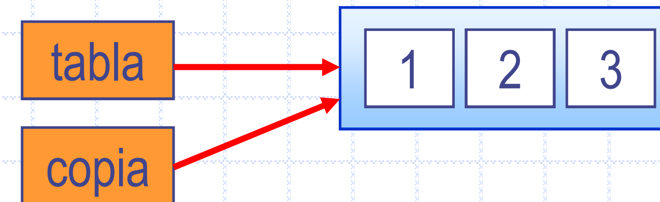




© Juan Carlos Ruiz, ITACA-UPV, 2008

Copiar variables de tipo array

- Al copiar una variable de tipo array se copia únicamente la variable, no la instancia




```
int[] tabla = {1,2,3};
int[] copia = tabla;

copia[0]++;

Console.WriteLine("{0}", tabla[0]);        // 2
Console.WriteLine("{0}", copia[0]);        // 2
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Métodos de arrays

- Todos los arrays soportan la clase *System.Array*
  - Sort – ordena los elementos del array

```
int[] datos = {4,6,3,8,9,3};    // Desordenado
Array.Sort(datos);              // Ordenado
```

  - Clear – pone un conjunto de elementos a cero o *null*

```
int[] datos = {4,6,3,8,9,3};
Array.Clear(datos, 0, datos.Length);  //{0,0,0,0,0,0}
```

  - Clone – crea una copia de la instancia del array

```
int[] datos = {4,6,3,8,9,3};
int[] clon = (int[]) datos.Clone();   //{4,6,3,8,9,3}
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Métodos de arrays

- Todos los arrays soportan la clase *System.Array*
  - GetLength – devuelve la longitud de una dimensión

```
int[,] datos = {{4,6,3},{8,9,3}};
int dim0 = datos.GetLength(0); // 2
int dim1 = datos.GetLength(1); // 3
```

  - IndexOf – devuelve el índice de la primera aparición de un valor

```
int[] datos = {4,6,3,8,9,3};
int indice = Array.IndexOf(datos, 9); // 4
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Paso de arrays como parámetros

- Se pasa una copia de la variable del array, no de su instancia
  - Los cambios efectuados afectarán a la instancia del array original

```
public void MasUno(int[] vector) {
    foreach(int i in vector) { vector[i]++;  }
}

public static void Main() {
    int[] tabla = {0,1,2,3};
    MasUno(tabla);
    foreach(int i in vector) {
          Console.WriteLine(tabla[i]); // {1,2,3,4}
    }
}
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Instrucción *foreach*

- La instrucción *foreach* puede ser muy útil para recorrer los elementos de un array

```
public static void Main(string[] args) {
    foreach(string s in args)
            Console.WriteLine(s);
}
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Índice

- Aspectos básicos
- Variables, tipos y operadores
- Instrucciones
- Excepciones
- Arrays
- Métodos

© Juan Carlos Ruiz, ITACA-UPV, 2008

Métodos

- Aplicaciones divididas en pequeños pedazos de código son más fáciles de entender, diseñar, desarrollar, depurar y mantener
- Un método es un conjunto de instrucciones agrupadas bajo un nombre determinado

```
tipoRetorno nombreMétodo(parámetros)
{
    cuerpo
}
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Llamada a métodos

- Puede llamarse a un método dentro de la misma clase

```
nombreMétodo(parámetros);
```

- Puede llamarse a un método de otra clase si ha sido declarado *public*

```
nombreClase.nombreMétodo(parámetros);
```

- Pueden utilizarse llamadas anidadas
  - Métodos que llaman a métodos que llaman a métodos…

© Juan Carlos Ruiz, ITACA-UPV, 2008

Llamada a métodos

```
using System;

class Hola
{
    public static void SaludarEstatico() {
            Console.WriteLine(“Hola Estatico”);
    }
    public void SaludarDinamico() {
            Console.WriteLine(“Hola Dinamico”);
    }
    public static void Main()
    {
            SaludarEstatico();
            Hola hola = new Hola();
            hola.SaludarDinamico();
    }
}
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Instrucción *return*

- La instrucción *return* vuelve inmediatamente al llamador del método
- Si el tipo de dato que devuelve es distinto de *void*, puede utilizarse para devolver un valor

```
public void Saludar() {
        bool saludoCompleto = false;
        Console.WriteLine("Hola");
        if (!saludoCompleto)
                return;
        Console.WriteLine(" Mundo");
}
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Valores de retorno

- El método debe estar declarado con un tipo no *void*
  - En este caso DEBE devolver un valor

```
public int DosMasDos() {
    int a = 2, b = 2;
    return (a + b);
}
public static void Main() {
    int x = DosMasDos();
    Console.WriteLine("2 + 2 = {0}", x);
}
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Declaración y llamada con parámetros

- Declaración
  - Se debe definir el tipo y nombre para cada uno de los parámetros
- Llamada
  - Se debe proporcionar un valor del tipo adecuado para cada uno de los parámetros
- Hay 3 mecanismos para el paso de parámetros
  - Por valor (o parámetros de entrada)
  - Por referencia (o parámetros de entrada y salida)
  - Por salida (o parámetros de salida)

© Juan Carlos Ruiz, ITACA-UPV, 2008

Paso por valor

- Se pasa una copia del valor del parámetro
- La variable se puede modificar en el interior del método
- No tiene ningún efecto en su valor fuera del método
- El parámetro debe ser del mismo tipo o tipo compatible

```
public int Suma(int x, int y) {
    return (x + y);
}
public static void Main() {
    int i = 3, j = 5;
    int k = Suma(i, j);
    Console.WriteLine("{0} + {1} = {2}", i, j, k);
}
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Paso por referencia

- Una referencia es una posición de memoria
- Se utiliza la palabra reservada **ref** en la definición del método y en la llamada
- Deben coincidir el tipo y el valor de la variable
- Los cambios que se efectúen en el método afectan al llamador

```
public void SumaUno(ref int x) {
      x++;
}
public static void Main() {
     int i = 3;
     SumaUno(ref i);
     Console.WriteLine(“i++ = {0}”, i);
}
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Parámetros de salida

- Son análogos a los parámetros por referencia
  - Transfieren datos fuera del método en lugar de dentro
  - Se utiliza la palabra clave **out** en la declaración del método y en la llamada

```
public void Saludo(out string s) {
      s = “Hola Mundo”;
}
public static void Main() {
     string i;
     Saludo(out i);
     Console.WriteLine(“Saludo = {0}”, i);
}
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Listas de parámetros de longitud variable

- Se utiliza la palabra clave *params*
- Se declara como un *array* al final de la lista de parámetros
- Siempre se pasa por valor

```
public int SumaLista(params int[] lista) {
      int i = 0;
      foreach(int x in lista)
            i += x;
      return(i);
}
public static void Main() {
      int i = SumaLista(1, 5, 22, 12);
      Console.WriteLine("Suma = {0}", i);
}
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Métodos recursivos

- Un método puede llamarse a sí mismo

```
public int Factorial(int x) {
    if (x <= 1)
        return(1);
    else
        return(x * Factorial(x - 1));
}

public static void Main() {
    int i = Factorial(5);
    Console.WriteLine("Factorial = {0}",i);
}
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Métodos sobrecargados

- Los métodos sobrecargados son aquellos que, dentro de una misma clase, comparten el mismo nombre
- Se distinguen por su lista de parámetros

```
public int Suma(int x, int y) {
     return(x + y);
}
public int Suma(int x, int y, int z) {
     return(x + y + z);
}
public static void Main() {
    Console.WriteLine(Suma(Suma(3,5),Suma(1,2,3)));
}
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Signatura de los métodos

- La signatura de los métodos debe ser diferente dentro de una misma clase
- La signatura la forman
  - El nombre del método
  - El tipo de los parámetros
  - El modificador de los parámetros
- No tienen efecto en la signatura
  - El nombre de los parámetros
  - El tipo de retorno del método

© Juan Carlos Ruiz, ITACA-UPV, 2008

C# en 60 minutos

- Practiquemos lo aprendido mediante un sencillo ejercicio…

© Juan Carlos Ruiz, ITACA-UPV, 2008

Ejercicio

- Obtener el mes y día a partir del número del día
  - **El día 40 es el 9 de Febrero**

```
using System;
class QueDiaEs{
   public static void Main() {
        Console.Write("Introduce un número de día entre 1 y 365: ");
        int dato = Convert.ToInt32(Console.ReadLine());
        int numeroDia = dato;
        int numeroMes = 0;
        string nombreMes;

        //
        // TODO: Añadir el código aquí
        //

        Console.WriteLine("El día {0} es el {1} de {2}", dato, numeroDia, nombreMes);
   }
}
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Ejercicio

- Obtener el mes y día a partir del número del día

```
if (numeroDia > 31) { //No es Enero

   numeroDia-= 31;
   numeroMes++;

   if (numeroDia > 28) { //No es Febrero

      numeroDia-=28;
      numeroMes++;

      ...
   }
}
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Ejercicio

- Obtener el mes y día a partir del número del día

```
switch(numeroMes) {
   case 0:
      nombreMes = "Enero";
      break;
   case 1:
      nombreMes = "Febrero";
      break;
   ...
   default:
      nombreMes = "NO EXISTE";
      break;
}
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Ejercicio

- Obtener el mes y día a partir del número del día

Número	Día y mes
32	1 de Febrero
60	1 de Marzo
91	1 de Abril
186	5 de Julio
304	31 de Octubre
309	5 de Noviembre
327	23 de Noviembre
359	25 de Diciembre

© Juan Carlos Ruiz, ITACA-UPV, 2008

Ejercicio

- Calcular el nombre del mes con una enumeración
  - Eliminar la instrucción switch y crear la enumeración

```
enumMes {
   Enero,
   Febrero,
   ...,
   Diciembre};

Mes mes = (Mes) numeroMes;
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Ejercicio

- Cambiar los *if* por un *foreach*
  - Eliminar las instrucciones *if* y crear la enumeración

```
int[] DiasMes = new int[]{31, 28, 31, 30, ..., 31};

foreach(int dias in DiasMes) {
   if (numeroDia> dias) {
      numeroDia -= dias;
      numeroMes++;
   }
   else {
      break;
   }
}
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Ejercicio

- Validar el número del día
  - Comprobar si el número del día está entre 1 y 365
  - Si no, lanzar una excepción del tipo ArgumentOutOfRangeException

```
try {
   ...
   if ((numeroDia < 1) || (numeroDia > 365)) {
      throw new ArgumentOutOfRangeException("Día inválido");
   }
}
catch (Exception e) {
   Console.WriteLine(e);
}
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Ejercicio

- Gestionar años bisiestos
  - Solicitar el número de año
  - Determinar si es bisiesto
    - Divisible por 4
    - No es divisible por 100 o sí es divisible por 400

Año bisiesto	Año no bisiesto
1996	1999
2000	1900
2004	2001

© Juan Carlos Ruiz, ITACA-UPV, 2008

Ejercicio

- Gestionar años bisiestos
  - Validar el día del año dependiendo de si es bisiesto (366)
  - Calcular correctamente el día y mes para años bisiestos

```
int[] DiasMesBisiesto = new int[]{31, 29, 31, 30, ..., 31};
if (añoBisiesto) {
   foreach(int dias in DiasMesBisiesto) {
      if (numeroDia > dias) {
         numeroDia-= dias;
         numeroMes++;
      }
      else {
         break;
      }
   }
}
```

© Juan Carlos Ruiz, ITACA-UPV, 2008

Ejercicio

- Gestionar años bisiestos

Año	Número	Día y mes
1999	32	1 de Febrero
2000	32	1 de Febrero
1999	60	1 de Marzo
2000	60	20 de Febrero
1999	91	1 de Abril
2000	91	31 de Mazo
1999	186	5 de Julio
2000	186	4 de Julio

Año	Número	Día y mes
1999	304	31 de Octubre
2000	304	30 de Octubre
1999	309	5 de Noviembre
2000	309	4 de Noviembre
1999	327	23 de Noviembre
2000	327	22 de Noviembre
1999	359	25 de Diciembre
2000	359	24 de Diciembre

© Juan Carlos Ruiz, ITACA-UPV, 2008